NiBabel Documentation

Release 3.2.1+99.g9853e6d6.dirty

NiBabel Authors

May 14, 2021, 15:58 PDT

CONTENTS

Read / write access to some common neuroimaging file formats

This package provides read +/- write access to some common medical and neuroimaging file formats, including: ANALYZE (plain, SPM99, SPM2 and later), GIFTI, NIfTI1, NIfTI2, CIFTI-2, MINC1, MINC2, AFNI BRIK/HEAD, MGH and ECAT as well as Philips PAR/REC. We can read and write FreeSurfer geometry, annotation and morphometry files. There is some very limited support for DICOM. NiBabel is the successor of PyNIfTI.

The various image format classes give full or selective access to header (meta) information and access to the image data is made available via NumPy arrays.

CHAPTER

ONE

WEBSITE

Current documentation on nibabel can always be found at the NIPY nibabel website.

CHAPTER

TWO

MAILING LISTS

Please send any questions or suggestions to the neuroimaging mailing list.

CHAPTER

THREE

CODE

Install nibabel with:

```
pip install nibabel
```

You may also be interested in:

- the nibabel code repository on Github;
- documentation for all releases and current development tree;
- download the current release from pypi;
- download current development version as a zip file;
- downloads of all available releases.

CHAPTER

FOUR

LICENSE

Nibabel is licensed under the terms of the MIT license. Some code included with nibabel is licensed under the BSD license. Please see the COPYING file in the nibabel distribution.

CHAPTER

FIVE

CITING NIBABEL

Please see the available releases for the release of nibabel that you are using. Recent releases have a Zenodo Digital Object Identifier badge at the top of the release notes. Click on the badge for more information.

CHAPTER

SIX

DOCUMENTATION

- *User Documentation* (manual)
- *Tutorials* (relevant tutorials on imaging)
- *API Documentation* (comprehensive reference)
- *Developer Guidelines* (for those who want to contribute)
- *Development Changelog* (see what has changed)
- *DICOM concepts* (details about implementing DICOM reading)
- genindex (access by keywords)
- search (online and offline full-text search)

See also the *Developer documentation page* for development discussions, release procedure and more.

CHAPTER

SEVEN

AUTHORS AND CONTRIBUTORS

Most work on NiBabel so far has been by Matthew Brett, Chris Markiewicz, Michael Hanke, Marc-Alexandre Côté, Ben Cipollini, Paul McCarthy and Chris Cheng. The authors are grateful to the following people who have contributed code and discussion (in rough order of appearance):

- Yaroslav O. Halchenko
- Chris Burns
- Gaël Varoquaux
- Ian Nimmo-Smith
- Jarrod Millman
- Bertrand Thirion
- Thomas Ballinger
- Cindee Madison
- Valentin Haenel
- Alexandre Gramfort
- Christian Haselgrove
- Krish Subramaniam
- Yannick Schwartz
- Bago Amirbekian
- Brendan Moloney
- Félix C. Morency
- JB Poline
- Basile Pinsard
- Satrajit Ghosh
- Nolan Nichols
- Ly Nguyen
- Philippe Gervais
- Demian Wassermann
- Justin Lecher
- Oliver P. Hinds

- Nikolaas N. Oosterhof
- Kevin S. Hahn
- Michiel Cottaar
- Erik Kastman
- Github user `freec84`
- Peter Fischer
- Clemens C. C. Bauer
- Samuel St-Jean
- Gregory R. Lee
- Eric M. Baker
- Ariel Rokem
- Eleftherios Garyfallidis
- Jaakko Leppäkangas
- Syam Gadde
- Robert D. Vincent
- Ivan Gonzalez
- Demian Wassermann
- Paul McCarthy
- Fernando Pérez García
- Venky Reddy
- Mark Hymers
- Jasper J.F. van den Bosch
- Bennet Fauber
- Kesshi Jordan
- Jon Stutters
- Serge Koudoro
- Christopher P. Cheng
- Mathias Goncalves
- Jakub Kaczmarzyk
- Dimitri Papadopoulos Orfanos
- Ross Markello
- Miguel Estevan Moreno
- Thomas Roos
- Igor Solovey
- Jon Haitz Legarreta Gorroño
- Katrin Leinweber

- Soichi Hayashi
- Samir Reddigari
- Konstantinos Raktivan
- Matt Cieslak
- Egor Panfilov
- Jath Palasubramaniam
- Henry Braun
- Oscar Esteban
- Cameron Riddell
- Hao-Ting Wang
- Dorota Jarecka
- Chris Gorgolewski
- Benjamin C Darwin
- Zvi Baratz
- Roberto Guidotti
- Or Duek
- Anibal Sólon
- Jonathan Daniel
- Markéta Calábková
- Carl Gauthier
- Julian Klug

CHAPTER

EIGHT

LICENSE REPRISE

NiBabel is free-software (beer and speech) and covered by the MIT License. This applies to all source code, documentation, examples and snippets inside the source distribution (including this website). Please see the *appendix of the manual* for the copyright statement and the full text of the license.

CHAPTER

NINE

DOWNLOAD AND INSTALLATION

Please find detailed *download and installation instructions* in the manual.

CHAPTER

TEN

SUPPORT

If you have problems installing the software or questions about usage, documentation or anything else related to NiBabel, you can post to the NiPy mailing list.

Mailing list neuroimaging@python.org [subscription, archive]

We recommend that anyone using NiBabel subscribes to the mailing list. The mailing list is the preferred way to announce changes and additions to the project. You can also search the mailing list archive using the *mailing list archive search* located in the sidebar of the NiBabel home page.

10.1 NiBabel Manual

10.1.1 Installation

NiBabel is a pure Python package at the moment, and it should be easy to get NiBabel running on any system. For the most popular platforms and operating systems there should be packages in the respective native packaging format (DEB, RPM or installers). On other systems you can install NiBabel using pip or by downloading the source package and running the usual `python setup.py install`.

Installer and packages

pip and the Python package index

If you are not using a Linux package manager, then best way to install NiBabel is via pip. If you don't have pip already, follow the pip install instructions.

Then open a terminal (`Terminal.app` on OSX, `cmd` or `Powershell` on Windows), and type:

```
pip install nibabel
```

This will download and install NiBabel.

If you really like doing stuff manually, you can install NiBabel by downoading the source from NiBabel pypi . Go to the pypi page and select the source distribution you want. Download the distribution, unpack it, and then, from the unpacked directory, run:

```
pip install .
```

If you get permission errors, this may be because `pip` is trying to install to the system directories. You can solve this error by using `sudo`, but we strongly suggest you either do an install into your "user" directories, like this:

```
pip install --user .
```

or you work inside a virtualenv.

Debian/Ubuntu

Our friends at NeuroDebian have packaged NiBabel at NiBabel NeuroDebian. Please follow the instructions on the NeuroDebian website on how to access their repositories. Once this is done, installing NiBabel is:

```
apt-get update
apt-get install python-nibabel
```

Install a development version

If you want to test the latest development version of nibabel, or you'd like to help by contributing bug-fixes or new features (excellent!), then this section is for you.

Requirements

- Python 3.6 or greater
- NumPy 1.14 or greater
- Packaging 14.3 or greater
- SciPy (optional, for full SPM-ANALYZE support)
- h5py (optional, for MINC2 support)
- PyDICOM 0.9.9 or greater (optional, for DICOM support)
- Python Imaging Library (optional, for PNG conversion in DICOMFS)
- pytest (optional, to run the tests)
- sphinx (optional, to build the documentation)

Get the development sources

You can download a tarball of the latest development snapshot (i.e. the current state of the *master* branch of the NiBabel source code repository) from the NiBabel github page.

If you want to have access to the full NiBabel history and the latest development code, do a full clone (AKA checkout) of the NiBabel repository:

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

Installation

Just install the modules by invoking:

```
pip install .
```

See *pip and the Python package index* for advice on what to do for permission errors.

Validating your install

For a basic test of your installation, fire up Python and try importing the module to see if everything is fine. It should look something like this:

```
Python 2.7.8 (v2.7.8:ee879c0ffa11, Jun 29 2014, 21:07:35)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import nibabel
>>>
```

To run the nibabel test suite, from the terminal run `pytest nibabel` or `python -c "import nibabel; nibabel.test()`.

To run an extended test suite that validates `nibabel` for long-running and resource-intensive cases, please see *Advanced Testing*.

10.1.2 Getting Started

NiBabel supports an ever growing collection of neuroimaging file formats. Every file format format has its own features and pecularities that need to be taken care of to get the most out of it. To this end, NiBabel offers both high-level format-independent access to neuroimages, as well as an API with various levels of format-specific access to all available information in a particular file format. The following examples show some of NiBabel's capabilities and give you an idea of the API.

For more detail on the API, see *Nibabel images*.

When loading an image, NiBabel tries to figure out the image format from the filename. An image in a known format can easily be loaded by simply passing its filename to the `load` function.

To start the code examples, we load some useful libraries:

```
>>> import os
>>> import numpy as np
```

Then we fine the nibabel directory containing the example data:

```
>>> from nibabel.testing import data_path
```

There is a NIfTI file in this directory called `example4d.nii.gz`:

```
>>> example_filename = os.path.join(data_path, 'example4d.nii.gz')
```

Now we can import nibabel and load the image:

```
>>> import nibabel as nib
>>> img = nib.load(example_filename)
```

A NiBabel image knows about its shape:

```
>>> img.shape
(128, 96, 24, 2)
```

It also records the data type of the data as stored on disk. In this case the data on disk are 16 bit signed integers:

```
>>> img.get_data_dtype() == np.dtype(np.int16)
True
```

The image has an affine transformation that determines the world-coordinates of the image elements (see *Coordinate systems and affines*):

```
>>> img.affine.shape
(4, 4)
```

This information is available without the need to load anything of the main image data into the memory. Of course there is also access to the image data as a NumPy array

```
>>> data = img.get_fdata()
>>> data.shape
(128, 96, 24, 2)
>>> type(data)
<... 'numpy.ndarray'>
```

The complete information embedded in an image header is available via a format-specific header object.

```
>>> hdr = img.header
```

In case of this NIfTI file it allows accessing all NIfTI-specific information, e.g.

```
>>> hdr.get_xyzt_units()
('mm', 'sec')
```

Corresponding "setter" methods allow modifying a header, while ensuring its compliance with the file format specifications.

In some situations we need even more flexibility and, for those with great courage, NiBabel also offers access to the raw header information

```
>>> raw = hdr.structarr
>>> raw['xyzt_units']
array(10, dtype=uint8)
```

This lowest level of the API is designed for people who know the file format well enough to work with its internal data, and comes without any safety-net.

Creating a new image in some file format is also easy. At a minimum it only needs some image data and an image coordinate transformation (affine):

```
>>> import numpy as np
>>> data = np.ones((32, 32, 15, 100), dtype=np.int16)
>>> img = nib.Nifti1Image(data, np.eye(4))
>>> img.get_data_dtype() == np.dtype(np.int16)
True
>>> img.header.get_xyzt_units()
('unknown', 'unknown')
```

In this case, we used the identity matrix as the affine transformation. The image header is initialized from the provided data array (i.e. shape, dtype) and all other values are set to resonable defaults.

Saving this new image to a file is trivial:

```
>>> nib.save(img, os.path.join('build', 'test4d.nii.gz'))
```

This short introduction only gave a quick overview of NiBabel's capabilities. Please have a look at the *API Documentation* for more details about supported file formats and their features.

10.1.3 Nibabel images

A nibabel image object is the association of three things:

- an N-D array containing the image *data*;
- a (4, 4) *affine* matrix mapping array coordinates to coordinates in some RAS+ world coordinate space (*Coordinate systems and affines*);
- image metadata in the form of a *header*.

The image object

First we load some libraries we are going to need for the examples:

```
>>> import os
>>> import numpy as np
```

There is an example image in the nibabel distribution.

```
>>> from nibabel.testing import data_path
>>> example_file = os.path.join(data_path, 'example4d.nii.gz')
```

We load the file to create a nibabel *image object*:

```
>>> import nibabel as nib
>>> img = nib.load(example_file)
```

The object `img` is an instance of a nibabel image. In fact it is an instance of a nibabel *`nibabel.nifti1.Nifti1Image`*:

```
>>> img
<nibabel.nifti1.Nifti1Image object at ...>
```

As with any Python object, you can inspect `img` to see what attributes it has. We recommend using IPython tab completion for this, but here are some examples of interesting attributes:

`dataobj` is the object pointing to the image array data:

```
>>> img.dataobj
<nibabel.arrayproxy.ArrayProxy object at ...>
```

See *Array proxies and proxy images* for more on why this is an array *proxy*.

`affine` is the affine array relating array coordinates from the image data array to coordinates in some RAS+ world coordinate system (*Coordinate systems and affines*):

```
>>> # Set numpy to print only 2 decimal digits for neatness
>>> np.set_printoptions(precision=2, suppress=True)
```

```
>>> img.affine
array([[ -2.  ,   0.  ,   0.  , 117.86],
       [ -0.  ,   1.97,  -0.36, -35.72],
       [  0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]])
```

`header` contains the metadata for this image. In this case it is specifically NIfTI metadata:

```
>>> img.header
<nibabel.nifti1.Nifti1Header object at ...>
```

The image header

The header of an image contains the image metadata. The information in the header will differ between different image formats. For example, the header information for a NIfTI1 format file differs from the header information for a MINC format file.

Our image is a NIfTI1 format image, and it therefore has a NIfTI1 format header:

```
>>> header = img.header
>>> print(header)
<class 'nibabel.nifti1.Nifti1Header'> object, endian='<'
sizeof_hdr      : 348
data_type       : b''
db_name         : b''
extents         : 0
session_error   : 0
regular         : b'r'
dim_info        : 57
dim             : [  4 128  96  24   2   1   1   1]
intent_p1       : 0.0
intent_p2       : 0.0
intent_p3       : 0.0
intent_code     : none
datatype        : int16
bitpix          : 16
slice_start     : 0
pixdim          : [   -1.      2.      2.      2.2  2000.      1.      1.      1. ]
vox_offset      : 0.0
scl_slope       : nan
scl_inter       : nan
slice_end       : 23
slice_code      : unknown
xyzt_units      : 10
cal_max         : 1162.0
cal_min         : 0.0
slice_duration  : 0.0
toffset         : 0.0
glmax           : 0
glmin           : 0
descrip         : b'FSL3.3\x00 v2.25 NIfTI-1 Single file format'
aux_file        : b''
qform_code      : scanner
```

(continues on next page)

(continued from previous page)

```
sform_code      : scanner
quatern_b       : -1.94510681403e-26
quatern_c       : -0.996708512306
quatern_d       : -0.081068739295
qoffset_x       : 117.855102539
qoffset_y       : -35.7229423523
qoffset_z       : -7.24879837036
srow_x          : [  -2.      0.      0.    117.86]
srow_y          : [ -0.     1.97  -0.36 -35.72]
srow_z          : [ 0.    0.32  2.17 -7.25]
intent_name     : b''
magic           : b'n+1'
```

The header of any image will normally have the following methods:

- `get_data_shape()` to get the output shape of the image data array:

```
>>> print(header.get_data_shape())
(128, 96, 24, 2)
```

- `get_data_dtype()` to get the numpy data type in which the image data is stored (or will be stored if you save the image):

```
>>> print(header.get_data_dtype())
int16
```

- `get_zooms()` to get the voxel sizes in millimeters:

```
>>> print(header.get_zooms())
(2.0, 2.0, 2.19999..., 2000.0)
```

  The last value of `header.get_zooms()` is the time between scans in milliseconds; this is the equivalent of voxel size on the time axis.

The image data array

The image data array is a little more complicated, because the image array can be stored in the image object as a numpy array or stored on disk for you to access later via an *array proxy*.

Array proxies and proxy images

When you load an image from disk, as we did here, the data is likely to be accessible via an array proxy. An array proxy is not the array itself but something that represents the array, and can provide the array when we ask for it.

Our image does have an array proxy, as we have already seen:

```
>>> img.dataobj
<nibabel.arrayproxy.ArrayProxy object at ...>
```

The array proxy allows us to create the image object without immediately loading all the array data from disk.

Images with an array proxy object like this one are called *proxy images* because the image data is not yet an array, but the array proxy points to (proxies) the array data on disk.

You can test if the image has a array proxy like this:

```
>>> nib.is_proxy(img.dataobj)
True
```

Array images

We can also create images from numpy arrays. For example:

```
>>> array_data = np.arange(24, dtype=np.int16).reshape((2, 3, 4))
>>> affine = np.diag([1, 2, 3, 1])
>>> array_img = nib.Nifti1Image(array_data, affine)
```

In this case the image array data is already a numpy array, and there is no version of the array on disk. The `dataobj` property of the image is the array itself rather than a proxy for the array:

```
>>> array_img.dataobj
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]], dtype=int16)
>>> array_img.dataobj is array_data
True
```

`dataobj` is an array, not an array proxy, so:

```
>>> nib.is_proxy(array_img.dataobj)
False
```

Getting the image data the easy way

For either type of image (array or proxy) you can always get the data with the `get_fdata()` method.

For the array image, `get_fdata()` just returns the data array, if it's already the required floating point type (default 64-bit float). If it isn't that type, `get_fdata()` casts it to one:

```
>>> image_data = array_img.get_fdata()
>>> image_data.shape
(2, 3, 4)
>>> image_data.dtype == np.dtype(np.float64)
True
```

The cast to floating point means the array is not the one attached to the image:

```
>>> image_data is array_img.dataobj
False
```

Here's an image backed by a floating point array:

```
>>> farray_img = nib.Nifti1Image(image_data.astype(np.float64), affine)
>>> farray_data = farray_img.get_fdata()
>>> farray_data.dtype == np.dtype(np.float64)
True
```

There was no cast, so the array returned is exactly the array attached to the image:

```
>>> farray_data is farray_img.dataobj
True
```

For the proxy image, the `get_fdata()` method fetches the array data from disk using the proxy, and returns the array.

```
>>> image_data = img.get_fdata()
>>> image_data.shape
(128, 96, 24, 2)
```

The image `dataobj` property is still a proxy object:

```
>>> img.dataobj
<nibabel.arrayproxy.ArrayProxy object at ...>
```

Proxies and caching

You may not want to keep loading the image data off disk every time you call `get_fdata()` on a proxy image. By default, when you call `get_fdata()` the first time on a proxy image, the image object keeps a cached copy of the loaded array. The next time you call `img.get_fdata()`, the image returns the array from cache rather than loading it from disk again.

```
>>> data_again = img.get_fdata()
```

The returned data is the same (cached) copy we returned before:

```
>>> data_again is image_data
True
```

See *Images and memory* for more details on managing image memory and controlling the image cache.

Image slicing

At times it is useful to manipulate an image's shape while keeping it in the same coordinate system. The `slicer` attribute provides an array-slicing interface to produce new images with an appropriately adjusted header, such that the data at a given RAS+ location is unchanged.

```
>>> cropped_img = img.slicer[32:-32, ...]
>>> cropped_img.shape
(64, 96, 24, 2)
```

The data is identical to cropping the data block directly:

```
>>> np.array_equal(cropped_img.get_fdata(), img.get_fdata()[32:-32, ...])
True
```

However, unused data did not need to be loaded into memory or scaled. Additionally, the image affine was adjusted so that the X-translation is 32 voxels (64mm) less:

```
>>> cropped_img.affine
array([[ -2.  ,   0.  ,   0.  ,  53.86],
       [ -0.  ,   1.97,  -0.36, -35.72],
```

(continues on next page)

(continued from previous page)

```
       [  0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]])
```

```
>>> img.affine - cropped_img.affine
array([[ 0.,  0.,  0., 64.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Another use for the slicer object is to choose specific volumes from a time series:

```
>>> vol0 = img.slicer[..., 0]
>>> vol0.shape
(128, 96, 24)
```

Or a selection of volumes:

```
>>> img.slicer[..., :1].shape
(128, 96, 24, 1)
>>> img.slicer[..., :2].shape
(128, 96, 24, 2)
```

It is also possible to use an integer step when slicing, downsampling the image without filtering. Note that this *will induce artifacts* in the frequency spectrum (aliasing) along any axis that is down-sampled.

```
>>> downsampled = vol0.slicer[::2, ::2, ::2]
>>> downsampled.header.get_zooms()
(4.0, 4.0, 4.399998)
```

Finally, an image can be flipped along an axis, maintaining an appropriate affine matrix:

```
>>> nib.orientations.aff2axcodes(img.affine)
('L', 'A', 'S')
>>> ras = img.slicer[::-1]
>>> nib.orientations.aff2axcodes(ras.affine)
('R', 'A', 'S')
>>> ras.affine
array([[  2.  ,   0.  ,   0.  , 117.86],
       [  0.  ,   1.97,  -0.36, -35.72],
       [ -0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]])
```

Loading and saving

The `save` and `load` functions in nibabel should do all the work for you:

```
>>> nib.save(array_img, 'my_image.nii')
>>> img_again = nib.load('my_image.nii')
>>> img_again.shape
(2, 3, 4)
```

You can also use the `to_filename` method:

```
>>> array_img.to_filename('my_image_again.nii')
>>> img_again = nib.load('my_image_again.nii')
>>> img_again.shape
(2, 3, 4)
```

You can get and set the filename with `get_filename()` and `set_filename()`:

```
>>> img_again.set_filename('another_image.nii')
>>> img_again.get_filename()
'another_image.nii'
```

Details of files and images

If an image can be loaded or saved on disk, the image will have an attribute called `file_map`. `img.file_map` is a dictionary where the keys are the names of the files that the image uses to load / save on disk, and the values are `FileHolder` objects, that usually contain the filenames that the image has been loaded from or saved to. In the case of a NiFTI1 single file, this is just a single image file with a `.nii` or `.nii.gz` extension:

```
>>> list(img_again.file_map)
['image']
>>> img_again.file_map['image'].filename
'another_image.nii'
```

Other file types need more than one file to make up the image. The NiFTI1 pair type is one example. NIfTI pair images have one file containing the header information and another containing the image array data:

```
>>> pair_img = nib.Nifti1Pair(array_data, np.eye(4))
>>> nib.save(pair_img, 'my_pair_image.img')
>>> sorted(pair_img.file_map)
['header', 'image']
>>> pair_img.file_map['header'].filename
'my_pair_image.hdr'
>>> pair_img.file_map['image'].filename
'my_pair_image.img'
```

The older Analyze format also has a separate header and image file:

```
>>> ana_img = nib.AnalyzeImage(array_data, np.eye(4))
>>> sorted(ana_img.file_map)
['header', 'image']
```

It is the contents of the `file_map` that gets changed when you use `set_filename` or `to_filename`:

```
>>> ana_img.set_filename('analyze_image.img')
>>> ana_img.file_map['image'].filename
'analyze_image.img'
>>> ana_img.file_map['header'].filename
'analyze_image.hdr'
```

10.1.4 Images and memory

We saw in *Nibabel images* that images loaded from disk are usually *proxy images*. Proxy images are images that have a `dataobj` property that is not a numpy array, but an *array proxy* that can fetch the array data from disk.

```
>>> import os
>>> import numpy as np
>>> from nibabel.testing import data_path
>>> example_file = os.path.join(data_path, 'example4d.nii.gz')
```

```
>>> import nibabel as nib
>>> img = nib.load(example_file)
>>> img.dataobj
<nibabel.arrayproxy.ArrayProxy object at ...>
```

Nibabel does not load the image array from the proxy when you `load` the image. It waits until you ask for the array data. The standard way to ask for the array data is to call the `get_fdata()` method:

```
>>> data = img.get_fdata()
>>> data.shape
(128, 96, 24, 2)
```

We also saw in *Proxies and caching* that this call to `get_fdata()` will (by default) load the array data into an internal image cache. The image returns the cached copy on the next call to `get_fdata()`:

```
>>> data_again = img.get_fdata()
>>> data is data_again
True
```

This behavior is convenient if you want quick and repeated access to the image array data. The down-side is that the image keeps a reference to the image data array, so the array can't be cleared from memory until the image object gets deleted. You might prefer to keep loading the array from disk instead of keeping the cached copy in the image.

This page describes ways of using the image array proxies to save memory and time.

Using `in_memory` to check the state of the cache

You can use the `in_memory` property to check if the image has cached the array.

The `in_memory` property is always True for array images, because the image data is always an array in memory:

```
>>> array_data = np.arange(24, dtype=np.int16).reshape((2, 3, 4))
>>> affine = np.diag([1, 2, 3, 1])
>>> array_img = nib.Nifti1Image(array_data, affine)
>>> array_img.in_memory
True
```

For a proxy image, the `in_memory` property is False when the array is not in cache, and True when it is in cache:

```
>>> img = nib.load(example_file)
>>> img.in_memory
False
>>> data = img.get_fdata()
>>> img.in_memory
True
```

Using `uncache`

As y'all know, the proxy image has the array in cache, `get_fdata()` returns the cached array:

```
>>> data_again = img.get_fdata()
>>> data_again is data  # same array returned from cache
True
```

You can uncache a proxy image with the `uncache()` method:

```
>>> img.uncache()
>>> img.in_memory
False
>>> data_once_more = img.get_fdata()
>>> data_once_more is data  # a new copy read from disk
False
```

`uncache()` has no effect if the image is an array image, or if the cache is already empty.

You need to be careful when you modify arrays returned by `get_fdata()` on proxy images, because `uncache` will then change the result you get back from `get_fdata()`:

```
>>> proxy_img = nib.load(example_file)
>>> data = proxy_img.get_fdata()  # array cached and returned
>>> data[0, 0, 0, 0]
0.0
>>> data[0, 0, 0, 0] = 99  # modify returned array
>>> data_again = proxy_img.get_fdata()  # return cached array
>>> data_again[0, 0, 0, 0]  # cached array modified
99.0
```

So far the proxy image behaves the same as an array image. `uncache()` has no effect on an array image, but it does have an effect on the returned array of a proxy image:

```
>>> proxy_img.uncache()  # cached array discarded from proxy image
>>> data_once_more = proxy_img.get_fdata()  # new copy of array loaded
>>> data_once_more[0, 0, 0, 0]  # array modifications discarded
0.0
```

Saving memory

Uncache the array

If you do not want the image to keep the array in its internal cache, you can use the `uncache()` method:

```
>>> img.uncache()
```

Use the array proxy instead of `get_fdata()`

The `dataobj` property of a proxy image is an array proxy. We can ask the proxy to return the array directly by passing `dataobj` to the numpy `asarray` function:

```
>>> proxy_img = nib.load(example_file)
>>> data_array = np.asarray(proxy_img.dataobj)
>>> type(data_array)
<... 'numpy.ndarray'>
```

This also works for array images, because `np.asarray` returns the array:

```
>>> array_img = nib.Nifti1Image(array_data, affine)
>>> data_array = np.asarray(array_img.dataobj)
>>> type(data_array)
<... 'numpy.ndarray'>
```

If you want to avoid caching you can avoid `get_fdata()` and always use `np.asarray(img.dataobj)`.

Use the `caching` keyword to `get_fdata()`

The default behavior of the `get_fdata()` function is to always fill the cache, if it is empty. This corresponds to the default `'fill'` value to the `caching` keyword. So, this:

```
>>> proxy_img = nib.load(example_file)
>>> data = proxy_img.get_fdata()  # default caching='fill'
>>> proxy_img.in_memory
True
```

is the same as this:

```
>>> proxy_img = nib.load(example_file)
>>> data = proxy_img.get_fdata(caching='fill')
>>> proxy_img.in_memory
True
```

Sometimes you may want to avoid filling the cache, if it is empty. In this case, you can use `caching='unchanged'`:

```
>>> proxy_img = nib.load(example_file)
>>> data = proxy_img.get_fdata(caching='unchanged')
>>> proxy_img.in_memory
False
```

`caching='unchanged'` will leave the cache full if it is already full.

```
>>> data = proxy_img.get_fdata(caching='fill')
>>> proxy_img.in_memory
True
>>> data = proxy_img.get_fdata(caching='unchanged')
>>> proxy_img.in_memory
True
```

See the `get_fdata()` `docstring` for more detail.

Saving time and memory

You can use the array proxy to get slices of data from disk in an efficient way.

The array proxy API allows you to do slicing on the proxy. In most cases this will mean that you only load the data from disk that you actually need, often saving both time and memory.

For example, let us say you only wanted the second volume from the example dataset. You could do this:

```
>>> proxy_img = nib.load(example_file)
>>> data = proxy_img.get_fdata()
>>> data.shape
(128, 96, 24, 2)
>>> vol1 = data[..., 1]
>>> vol1.shape
(128, 96, 24)
```

The problem is that you had to load the whole data array into memory before throwing away the first volume and keeping the second.

You can use array proxy slicing to do this more efficiently:

```
>>> proxy_img = nib.load(example_file)
>>> vol1 = proxy_img.dataobj[..., 1]
>>> vol1.shape
(128, 96, 24)
```

The slicing call in `proxy_img.dataobj[..., 1]` will only load the data from disk that you need to fill the memory of `vol1`.

10.1.5 Working with NIfTI images

This page describes some features of the nibabel implementation of the NIfTI format. Generally all these features apply equally to the NIfTI 1 and the NIfTI 2 format, but we will note the differences when they come up. NIfTI 1 is much more common than NIfTI 2.

Preliminaries

We first set some display parameters to print out numpy arrays in a compact form:

```
>>> import numpy as np
>>> # Set numpy to print only 2 decimal digits for neatness
>>> np.set_printoptions(precision=2, suppress=True)
```

Example NIfTI images

```
>>> import os
>>> import nibabel as nib
>>> from nibabel.testing import data_path
```

This is the example NIfTI 1 image:

```
>>> example_ni1 = os.path.join(data_path, 'example4d.nii.gz')
>>> n1_img = nib.load(example_ni1)
>>> n1_img
<nibabel.nifti1.Nifti1Image object at ...>
```

Here is the NIfTI 2 example image:

```
>>> example_ni2 = os.path.join(data_path, 'example_nifti2.nii.gz')
>>> n2_img = nib.load(example_ni2)
>>> n2_img
<nibabel.nifti2.Nifti2Image object at ...>
```

The NIfTI header

The NIfTI 1 header is a small C structure of size 352 bytes. It contains the following fields:

```
>>> n1_header = n1_img.header
>>> print(n1_header)
<class 'nibabel.nifti1.Nifti1Header'> object, endian='<'
sizeof_hdr      : 348
data_type       : b''
db_name         : b''
extents         : 0
session_error   : 0
regular         : b'r'
dim_info        : 57
dim             : [  4 128  96  24   2   1   1   1]
intent_p1       : 0.0
intent_p2       : 0.0
intent_p3       : 0.0
intent_code     : none
datatype        : int16
bitpix          : 16
slice_start     : 0
pixdim          : [   -1.      2.      2.      2.2  2000.      1.      1.      1. ]
vox_offset      : 0.0
scl_slope       : nan
scl_inter       : nan
slice_end       : 23
slice_code      : unknown
xyzt_units      : 10
cal_max         : 1162.0
cal_min         : 0.0
slice_duration  : 0.0
toffset         : 0.0
glmax           : 0
glmin           : 0
descrip         : b'FSL3.3\x00 v2.25 NIfTI-1 Single file format'
aux_file        : b''
qform_code      : scanner
sform_code      : scanner
quatern_b       : -1.94510681403e-26
quatern_c       : -0.996708512306
quatern_d       : -0.081068739295
qoffset_x       : 117.855102539
qoffset_y       : -35.7229423523
```

(continues on next page)

(continued from previous page)

```
qoffset_z       : -7.24879837036
srow_x          : [  -2.      0.      0.    117.86]
srow_y          : [ -0.     1.97  -0.36 -35.72]
srow_z          : [ 0.    0.32  2.17 -7.25]
intent_name     : b''
magic           : b'n+1'
```

The NIfTI 2 header is similar, but of length 540 bytes, with fewer fields:

```
>>> n2_header = n2_img.header
>>> print(n2_header)
    <class 'nibabel.nifti2.Nifti2Header'> object, endian='<'
    sizeof_hdr      : 540
    magic           : b'n+2'
    eol_check       : [13 10 26 10]
    datatype        : int16
    bitpix          : 16
    dim             : [ 4 32 20 12  2  1  1  1]
    intent_p1       : 0.0
    intent_p2       : 0.0
    intent_p3       : 0.0
    pixdim          : [   -1.      2.      2.      2.2  2000.      1.      1.      1.␣
↪]
    vox_offset      : 0
    scl_slope       : nan
    scl_inter       : nan
    cal_max         : 1162.0
    cal_min         : 0.0
    slice_duration  : 0.0
    toffset         : 0.0
    slice_start     : 0
    slice_end       : 23
    descrip         : b'FSL3.3\x00 v2.25 NIfTI-1 Single file format'
    aux_file        : b''
    qform_code      : scanner
    sform_code      : scanner
    quatern_b       : -1.94510681403e-26
    quatern_c       : -0.996708512306
    quatern_d       : -0.081068739295
    qoffset_x       : 117.855102539
    qoffset_y       : -35.7229423523
    qoffset_z       : -7.24879837036
    srow_x          : [  -2.      0.      0.    117.86]
    srow_y          : [ -0.     1.97  -0.36 -35.72]
    srow_z          : [ 0.    0.32  2.17 -7.25]
    slice_code      : unknown
    xyzt_units      : 10
    intent_code     : none
    intent_name     : b''
    dim_info        : 57
    unused_str      : b''
```

You can get and set individual fields in the header using dict (mapping-type) item access. For example:

```
>>> n1_header['cal_max']
array(1162., dtype=float32)
>>> n1_header['cal_max'] = 1200
```

(continues on next page)

(continued from previous page)

```
>>> n1_header['cal_max']
array(1200., dtype=float32)
```

Check the attributes of the header for `get_` / `set_` methods to get and set various combinations of NIfTI header fields.

The `get_` / `set_` methods should check and apply valid combinations of values from the header, whereas you can do anything you like with the dict / mapping item access. It is safer to use the `get_` / `set_` methods and use the mapping item access only if the `get_` / `set_` methods will not do what you want.

The NIfTI affines

Like other nibabel image types, NIfTI images have an affine relating the voxel coordinates to world coordinates in RAS+ space:

```
>>> n1_img.affine
array([[ -2.  ,   0.  ,   0.  , 117.86],
       [ -0.  ,   1.97,  -0.36, -35.72],
       [  0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]])
```

Unlike other formats, the NIfTI header format can specify this affine in one of three ways — the *sform* affine, the *qform* affine and the *fall-back header* affine.

Nibabel uses an *algorithm* to chose which of these three it will use for the overall image `affine`.

The sform affine

The header stores the three first rows of the 4 by 4 affine in the header fields `srow_x`, `srow_y`, `srow_z`. The header does not store the fourth row because it is always `[0, 0, 0, 1]` (see *Coordinate systems and affines*).

You can get the sform affine specifically with the `get_sform()` method of the image or the header.

For example:

```
>>> print(n1_header['srow_x'])
[ -2.     0.     0.   117.86]
>>> print(n1_header['srow_y'])
[ -0.     1.97  -0.36 -35.72]
>>> print(n1_header['srow_z'])
[ 0.    0.32  2.17 -7.25]
>>> print(n1_header.get_sform())
[[ -2.     0.     0.   117.86]
 [ -0.     1.97  -0.36 -35.72]
 [  0.     0.32   2.17  -7.25]
 [  0.     0.     0.     1.  ]]
```

This affine is valid only if the `sform_code` is not zero.

```
>>> print(n1_header['sform_code'])
1
```

The different sform code values specify which RAS+ space the sform affine refers to, with these interpretations:

Code	Label	Meaning
0	unknown	sform not defined
1	scanner	RAS+ in scanner coordinates
2	aligned	RAS+ aligned to some other scan
3	talairach	RAS+ in Talairach atlas space
4	mni	RAS+ in MNI atlas space

In our case the code is 1, meaning "scanner" alignment.

You can get the affine and the code using the `coded=True` argument to `get_sform()`:

```
>>> print(n1_header.get_sform(coded=True))
(array([[ -2.  ,   0.  ,   0.  , 117.86],
       [ -0.  ,   1.97,  -0.36, -35.72],
       [  0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]]), 1)
```

You can set the sform with the `set_sform()` method of the header and the image.

```
>>> n1_header.set_sform(np.diag([2, 3, 4, 1]))
>>> n1_header.get_sform()
array([[2., 0., 0., 0.],
       [0., 3., 0., 0.],
       [0., 0., 4., 0.],
       [0., 0., 0., 1.]])
```

Set the affine and code using the `code` parameter to `set_sform()`:

```
>>> n1_header.set_sform(np.diag([3, 4, 5, 1]), code='mni')
>>> n1_header.get_sform(coded=True)
(array([[3., 0., 0., 0.],
       [0., 4., 0., 0.],
       [0., 0., 5., 0.],
       [0., 0., 0., 1.]]), 4)
```

The qform affine

This affine can be calculated from a combination of the voxel sizes (entries 1 through 4 of the `pixdim` field), a sign flip called `qfac` stored in entry 0 of `pixdim`, and a quaternion that can be reconstructed from fields `quatern_b`, `quatern_c`, `quatern_d`.

See the code for the *`get_qform() method`* for details.

You can get and set the qform affine using the equivalent methods to those for the sform: `get_qform()`, `set_qform()`.

```
>>> n1_header.get_qform(coded=True)
(array([[ -2.  ,   0.  ,   0.  , 117.86],
       [ -0.  ,   1.97,  -0.36, -35.72],
       [  0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]]), 1)
```

The qform also has a corresponding `qform_code` with the same interpretation as the *sform_code*.

The fall-back header affine

This is the affine of last resort, constructed only from the `pixdim` voxel sizes. The NIfTI specification says that this should set the first voxel in the image as [0, 0, 0] in world coordinates, but we nibabblers follow SPM in preferring to set the central voxel to have [0, 0, 0] world coordinate. The NIfTI spec also implies that the image should be assumed to be in RAS+ *voxel* orientation for this affine (see *Coordinate systems and affines*). Again like SPM, we prefer to assume LAS+ voxel orientation by default.

You can always get the fall-back affine with `get_base_affine()`:

```
>>> n1_header.get_base_affine()
array([[ -2. ,   0. ,   0. , 127. ],
       [  0. ,   2. ,   0. , -95. ],
       [  0. ,   0. ,   2.2, -25.3],
       [  0. ,   0. ,   0. ,   1. ]])
```

Choosing the image affine

Given there are three possible affines defined in the NIfTI header, nibabel has to chose which of these to use for the image `affine`.

The algorithm is defined in the `get_best_affine()` method. It is:

1. If `sform_code` != 0 ('unknown') use the sform affine; else
2. If `qform_code` != 0 ('unknown') use the qform affine; else
3. Use the fall-back affine.

Default sform and qform codes

If you create a new image, e.g.:

```
>>> data = np.random.random((20, 20, 20))
>>> xform = np.eye(4) * 2
>>> img = nib.nifti1.Nifti1Image(data, xform)
```

The sform and qform codes will be initialised to 2 (aligned) and 0 (unknown) respectively:

```
>>> img.get_sform(coded=True)
(array([[2., 0., 0., 0.],
       [0., 2., 0., 0.],
       [0., 0., 2., 0.],
       [0., 0., 0., 1.]]), 2)
>>> img.get_qform(coded=True)
(None, 0)
```

This is based on the assumption that the affine you specify for a newly created image will align the image to some known coordinate system. According to the NIfTI specification, the qform is intended to encode a transformation into scanner coordinates - for a programmatically created image, we have no way of knowing what the scanner coordinate system is; furthermore, the qform cannot be used to store an arbitrary affine transform, as it is unable to encode shears. So the provided affine will be stored in the sform, and the qform will be left uninitialised.

If you create a new image and specify an existing header, e.g.:

```
>>> example_ni1 = os.path.join(data_path, 'example4d.nii.gz')
>>> n1_img = nib.load(example_ni1)
>>> new_header = header=n1_img.header.copy()
>>> new_data = np.random.random(n1_img.shape[:3])
>>> new_img = nib.nifti1.Nifti1Image(data, None, header=new_header)
```

then the newly created image will inherit the same sform and qform codes that are in the provided header. However, if you create a new image with both an affine and a header specified, e.g.:

```
>>> xform = np.eye(4)
>>> new_img = nib.nifti1.Nifti1Image(data, xform, header=new_header)
```

then the sform and qform codes will *only* be preserved if the provided affine is the same as the affine in the provided header. If the affines do not match, the sform and qform codes will be set to their default values of 2 and 0 respectively. This is done on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. So the original sform and qform codes can no longer be assumed to be valid.

If you wish to set the sform and qform affines and/or codes to some other value, you can always set them after creation using the `set_sform` and `set_qform` methods, as described above.

Data scaling

NIfTI uses a simple scheme for data scaling.

By default, nibabel will take care of this scaling for you, but there may be times that you want to control the data scaling yourself. If so, the next section describes how the scaling works and the nibabel implementation of same.

There are two scaling fields in the header called `scl_slope` and `scl_inter`.

The output data from a NIfTI image comes from:

1. Loading the binary data from the image file;
2. Casting the numbers to the binary format given in the header and returned by `get_data_dtype()`;
3. Reshaping to the output image shape;
4. Multiplying the result by the header `scl_slope` value, if both of `scl_slope` and `scl_inter` are defined;
5. Adding the value header `scl_inter` value to the result, if both of `scl_slope` and `scl_inter` are defined;

'Defined' means, the value is not NaN (not a number).

All this gets built into the array proxy when you load a NIfTI image.

When you load an image, the header scaling values automatically get set to NaN (undefined) to mark the fact that the scaling values have been consumed by the read. The scaling values read from the header on load only appear in the array proxy object.

To see how this works, let's make a new image with some scaling:

```
>>> array_data = np.arange(24, dtype=np.int16).reshape((2, 3, 4))
>>> affine = np.diag([1, 2, 3, 1])
>>> array_img = nib.Nifti1Image(array_data, affine)
>>> array_header = array_img.header
```

The default scaling values are NaN (undefined):

```
>>> array_header['scl_slope']
array(nan, dtype=float32)
>>> array_header['scl_inter']
array(nan, dtype=float32)
```

You can get the scaling values with the `get_slope_inter()` method:

```
>>> array_header.get_slope_inter()
(None, None)
```

None corresponds to the NaN scaling value (undefined).

We can set them in the image header, so they get saved to the header when the image is written. We can do this by setting the fields directly, or with `set_slope_inter()`:

```
>>> array_header.set_slope_inter(2, 10)
>>> array_header.get_slope_inter()
(2.0, 10.0)
>>> array_header['scl_slope']
array(2., dtype=float32)
>>> array_header['scl_inter']
array(10., dtype=float32)
```

Setting the scale factors in the header has no effect on the image data before we save and load again:

```
>>> array_img.get_fdata()
array([[[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]],

       [[12., 13., 14., 15.],
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

Now we save the image and load it again:

```
>>> nib.save(array_img, 'scaled_image.nii')
>>> scaled_img = nib.load('scaled_image.nii')
```

The data array has the scaling applied:

```
>>> scaled_img.get_fdata()
array([[[10., 12., 14., 16.],
        [18., 20., 22., 24.],
        [26., 28., 30., 32.]],

       [[34., 36., 38., 40.],
        [42., 44., 46., 48.],
        [50., 52., 54., 56.]]])
```

The header for the loaded image has had the scaling reset to undefined, to mark the fact that the scaling has been "consumed" by the load:

```
>>> scaled_img.header.get_slope_inter()
(None, None)
```

The original slope and intercept are still accessible in the array proxy object:

```
>>> scaled_img.dataobj.slope
2.0
>>> scaled_img.dataobj.inter
10.0
```

If the header scaling is undefined when we save the image, nibabel will try to find an optimum slope and intercept to best preserve the precision of the data in the output data type. Because nibabel will set the scaling to undefined when loading the image, or creating a new image, this is the default behavior.

10.1.6 Image voxel orientation

It is sometimes useful to know the approximate world-space orientations of the image voxel axes.

See *Coordinate systems and affines* for background on voxel and world axes.

For example, let's say we had an image with an identity affine:

```
>>> import numpy as np
>>> import nibabel as nib
>>> affine = np.eye(4)  # identity affine
>>> voxel_data = np.random.normal(size=(10, 11, 12))
>>> img = nib.Nifti1Image(voxel_data, affine)
```

Because the affine is an identity affine, the voxel axes align with the world axes. By convention, nibabel world axes are always in RAS+ orientation (left to Right, posterior to Anterior, inferior to Superior).

Let's say we took a single line of voxels along the first voxel axis:

```
>>> single_line_axis_0 = voxel_data[:, 0, 0]
```

The first voxel axis is aligned to the left to Right world axes. This means that the first voxel is towards the left of the world, and the last voxel is towards the right of the world.

Here is a single line in the second axis:

```
>>> single_line_axis_1 = voxel_data[0, :, 0]
```

The first voxel in this line is towards the posterior of the world, and the last towards the anterior.

```
>>> single_line_axis_2 = voxel_data[0, 0, :]
```

The first voxel in this line is towards the inferior of the world, and the last towards the superior.

This image therefore has RAS+ *voxel* axes.

In other cases, it is not so obvious what the orientations of the axes are. For example, here is our example NIfTI 1 file again:

```
>>> import os
>>> from nibabel.testing import data_path
>>> example_file = os.path.join(data_path, 'example4d.nii.gz')
>>> img = nib.load(example_file)
```

Here is the affine (to two digits decimal precision):

```
>>> np.set_printoptions(precision=2, suppress=True)
>>> img.affine
```

(continues on next page)

(continued from previous page)

```
array([[ -2.  ,   0.  ,   0.  , 117.86],
       [ -0.  ,   1.97,  -0.36, -35.72],
       [  0.  ,   0.32,   2.17,  -7.25],
       [  0.  ,   0.  ,   0.  ,   1.  ]])
```

What are the orientations of the voxel axes here?

Nibabel has a routine to tell you, called `aff2axcodes`.

```
>>> nib.aff2axcodes(img.affine)
('L', 'A', 'S')
```

The voxel orientations are nearest to:

1. First voxel axis goes from right to Left;
2. Second voxel axis goes from posterior to Anterior;
3. Third voxel axis goes from inferior to Superior.

Sometimes you may want to rearrange the image voxel axes to make them as close as possible to RAS+ orientation. We refer to this voxel orientation as *canonical* voxel orientation, because RAS+ is our canonical world orientation. Rearranging the voxel axes means reversing and / or reordering the voxel axes.

You can do the arrangement with `as_closest_canonical`:

```
>>> canonical_img = nib.as_closest_canonical(img)
>>> canonical_img.affine
array([[   2.  ,    0.  ,    0.  , -136.14],
       [   0.  ,    1.97,   -0.36,  -35.72],
       [  -0.  ,    0.32,    2.17,   -7.25],
       [   0.  ,    0.  ,    0.  ,    1.  ]])
>>> nib.aff2axcodes(canonical_img.affine)
('R', 'A', 'S')
```

10.1.7 Copyright and Licenses

NiBabel

The nibabel package, including all examples, code snippets and attached documentation is covered by the MIT license.

```
The MIT License

Copyright (c) 2009-2019 Matthew Brett <matthew.brett@gmail.com>
Copyright (c) 2010-2013 Stephan Gerhard <git@unidesign.ch>
Copyright (c) 2006-2014 Michael Hanke <michael.hanke@gmail.com>
Copyright (c) 2011 Christian Haselgrove <christian.haselgrove@umassmed.edu>
Copyright (c) 2010-2011 Jarrod Millman <jarrod.millman@gmail.com>
Copyright (c) 2011-2019 Yaroslav Halchenko <debian@onerussian.com>
Copyright (c) 2015-2019 Chris Markiewicz <effigies@gmail.com>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
```

(continues on next page)

(continued from previous page)

```
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

3rd party code and data

Some code distributed within the nibabel sources was developed by other projects. This code is distributed under its respective licenses that are listed below.

NetCDF

The netcdf IO module has been taken from SciPy.

```
Copyright (c) 1999-2010 SciPy Developers <scipy-dev@scipy.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

  a. Redistributions of source code must retain the above copyright notice,
     this list of conditions and the following disclaimer.
  b. Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
  c. Neither the name of the Enthought nor the names of its contributors
     may be used to endorse or promote products derived from this software
     without specific prior written permission.


THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

Sphinx autosummary extension

This extension has been copied from NumPy (Jul 16, 2010) as the one shipped with Sphinx 0.6 doesn't work properly.

```
Copyright (c) 2007-2009 Stefan van der Walt and Sphinx team

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

  a. Redistributions of source code must retain the above copyright notice,
     this list of conditions and the following disclaimer.
  b. Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
  c. Neither the name of the Enthought nor the names of its contributors
     may be used to endorse or promote products derived from this software
     without specific prior written permission.


THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

OrderedSet

In `nibabel/externals/oset.py`

Copied from: https://files.pythonhosted.org/packages/d6/b1/a49498c699a3fda5d635cc1fa222ffc686ea3b5d04b84a3166c4cab0c57b/oset-0.1.3.tar.gz

```
Copyright (c) 2009, Raymond Hettinger, and others All rights reserved.

Package structured based on the one developed to odict Copyright (c) 2010,␣
↪BlueDynamics Alliance, Austria

- Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this
  list of conditions and the following disclaimer in the documentation and/or
  other materials provided with the distribution.

- Neither the name of the BlueDynamics Alliance nor the names of its
  contributors may be used to endorse or promote products derived from this
  software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY BlueDynamics Alliance AS IS AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
```

(continues on next page)

(continued from previous page)

```
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
SHALL BlueDynamics Alliance BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF SUCH DAMAGE.
```

mni_icbm152_t1_tal_nlin_asym_09a

The file `doc/source/someone.nii.gz` is a subsampled version of the file `mni_icbm152_t1_tal_nlin_asym_09a.nii` from the MNI non-linear templates archive `mni_icbm152_t1_tal_nlin_asym_09a`. The original image has the following license (where 'softwware' refers to the image):

```
Copyright (C) 1993-2004 Louis Collins, McConnell Brain Imaging Centre,
Montreal Neurological Institute, McGill University.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided
that the above copyright notice appear in all copies.  The authors and
McGill University make no representations about the suitability of this
software for any purpose.  It is provided "as is" without express or
implied warranty.  The authors are not responsible for any data loss,
equipment damage, property loss, or injury to subjects or patients
resulting from the use or misuse of this software package.
```

Philips PAR/REC data

The files:

```
nibabel/tests/data/phantom_EPI_asc_CLEAR_2_1.PAR
nibabel/tests/data/phantom_EPI_asc_CLEAR_2_1.REC
nibabel/tests/data/Phantom_EPI_3mm_cor_20APtrans_15RLrot_SENSE_15_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_cor_SENSE_8_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_sag_15AP_SENSE_13_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_sag_15FH_SENSE_12_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_sag_15RL_SENSE_11_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_sag_SENSE_7_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_tra_-30AP_10RL_20FH_SENSE_14_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_tra_15FH_SENSE_9_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_tra_15RL_SENSE_10_1.PAR
nibabel/tests/data/Phantom_EPI_3mm_tra_SENSE_6_1.PAR
```

are from http://psydata.ovgu.de/philips_achieva_testfiles, and released under the PDDL version 1.0 available at http://opendatacommons.org/licenses/pddl/1.0/

The files:

```
nibabel/nibabel/tests/data/DTI.PAR
nibabel/nibabel/tests/data/NA.PAR
nibabel/nibabel/tests/data/T1.PAR
```

(continues on next page)

(continued from previous page)

```
nibabel/nibabel/tests/data/T2-interleaved.PAR
nibabel/nibabel/tests/data/T2.PAR
nibabel/nibabel/tests/data/T2_-interleaved.PAR
nibabel/nibabel/tests/data/T2_.PAR
nibabel/nibabel/tests/data/fieldmap.PAR
```

are from https://github.com/yarikoptic/nitest-balls1, also released under the the PDDL version 1.0 available at http://opendatacommons.org/licenses/pddl/1.0/

nibabel/nibabel/tests/data/umass_anonymized.PAR

is courtesy of the University of Massachusetts Medical School, also released under the PDDL.

10.1.8 NiBabel Development Changelog

NiBabel is the successor to the much-loved PyNifti package. Here we list the releases for both packages.

The full VCS changelog is available here:

http://github.com/nipy/nibabel/commits/master

Nibabel releases

Most work on NiBabel so far has been by Matthew Brett (MB), Chris Markiewicz (CM), Michael Hanke (MH), Marc-Alexandre Côté (MC), Ben Cipollini (BC), Paul McCarthy (PM), Chris Cheng (CC), Yaroslav Halchenko (YOH), Satra Ghosh (SG), Eric Larson (EL), Demian Wassermann, Stephan Gerhard and Ross Markello (RM).

References like "pr/298" refer to github pull request numbers.

3.2.1 (Saturday 28 November 2020)

Bug fix release in the 3.2.x series.

Maintenance

- Drop references to builtin types in Numpy namespace like `np.float` (pr/964) (EL, reviewed by CM)
- Ensure compatibility with Python 3.9 (pr/963) (CM)

3.2.0 (Tuesday 20 October 2020)

New feature release in the 3.2.x series.

New features

- `nib-stats` CLI tool to expose new `nibabel.imagestats` API. Initial implementation of volume calculations, a la `fslstats -V`. (Julian Klug, reviewed by CM and GitHub user 0rC0)
- `nib-roi` CLI tool to crop images and/or flip axes (pr/947) (CM, reviewed by Chris Cheng and Mathias Goncalves)
- Parser for Siemens "ASCCONV" text format (pr/896) (Brendan Moloney and MB, reviewed by CM)

Enhancements

- Drop confusing mention of `img.to_filename()` in getting started guide (pr/946) (Fernando Pérez-Garcia, reviewed by MB, CM)
- Implement `to_bytes()`/`from_bytes()` methods for `Cifti2Image` (pr/938) (CM, reviewed by Mathias Goncalves)
- Clean up of DICOM documentation (pr/910) (Jonathan Daniel, reviewed by MB)

Bug fixes

- Use canvas manager API to set title in `OrthoSlicer3D` (pr/958) (EL, reviewed by CM)
- Record units as seconds parrec2nii; previously set TR to seconds but retained msec units (pr/931) (CM, reviewed by MB)
- Reflect on-disk dimensions in NIfTI-2 view of CIFTI-2 images (pr/930) (Mathias Goncalves and CM)
- Fix outdated Python 2 and Sympy code in DICOM derivations (pr/911) (MB, reviewed by CM)
- Change string with invalid escape to raw string (pr/909) (EL, reviewed by MB)

Maintenance

- Fix typo in docs (pr/955) (Carl Gauthier, reviewed by CM)
- Purge nose from nisext tests (pr/934) (Markéta Calábková, reviewed by CM)
- Suppress expected warnings in tests (pr/949) (CM, reviewed by Dorota Jarecka)
- Various cleanups and modernizations (pr/916, pr/917, pr/918, pr/919) (Jonathan Daniel, reviewed by CM)
- SVG logo for improved appearance in with zooming (pr/914) (Jonathan Daniel, reviewed by CM)

API changes and deprecations

- Drop support for Numpy < 1.13 (pr/922) (CM)
- Warn on use of `onetime.setattr_on_read`, which has been a deprecated alias of `auto_attr` (pr/948) (CM, reviewed by Ariel Rokem)

3.1.1 (Friday 26 June 2020)

Bug-fix release in the 3.1.x series.

These are small compatibility fixes that support ARM64 architecture and `indexed_gzip>=1.3.0`.

Bug fixes

- Detect `IndexedGzipFile` as compressed file type (pr/925) (PM, reviewed by CM)
- Correctly cast `nan` when testing `array_to_file`, fixing ARM64 builds (pr/862) (CM, reviewed by MB)

3.1.0 (Monday 20 April 2020)

New feature release in the 3.1.x series.

New features

- Conformation function (`processing.conform`) and CLI tool (`nib-conform`) to apply shape, orientation and zooms (pr/853) (Jakub Kaczmarzyk, reviewed by CM, YOH)
- Affine rescaling function (`affines.rescale_affine`) to update dimensions and voxel sizes (pr/853) (CM, reviewed by Jakub Kaczmarzyk)

Bug fixes

- Delay import of h5py until neded (pr/889) (YOH, reviewed by CM)

Maintenance

- Fix typo in documentation (pr/893) (Zvi Baratz, reviewed by CM)
- Tests converted from nose to pytest (pr/865 + many sub-PRs) (Dorota Jarecka, Krzyzstof Gorgolewski, Roberto Guidotti, Anibal Solon, and Or Duek)

API changes and deprecations

- `kw_only_meth`/`kw_only_func` decorators are deprecated (pr/848) (RM, reviewed by CM)

2.5.2 (Wednesday 8 April 2020)

Bug-fix release in the 2.5.x series. This is an extended-support series, providing bug fixes for Python 2.7 and 3.4.

This and all future releases in the 2.5.x series will be incompatible with Python 3.9. The last compatible series of numpy and scipy are 1.16.x and 1.2.x, respectively.

If you are able to upgrade to Python 3, it is recommended to upgrade to NiBabel 3.

Bug fixes

- Change strings with invalid escapes to raw strings (pr/827) (EL, reviewed by CM)
- Re-import externals/netcdf.py from scipy to resolve numpy deprecation (pr/821) (CM)

Maintenance

- Set maximum numpy to 1.16.x, maximum scipy to 1.2.x (pr/901) (CM)

3.0.2 (Monday 9 March 2020)

Bug fixes

- Attempt to find versioneer version when building docs (pr/894) (CM)
- Delay import of h5py until neded (backport of pr/889) (YOH, reviewed by CM)

Maintenance

- Fix typo in documentation (backport of pr/893) (Zvi Baratz, reviewed by CM)
- Set minimum matplotlib to 1.5.3 to ensure wheels are available on all supported Python versions. (backport of pr/887) (CM)
- Remove `pyproject.toml` for now. (issue/859) (CM)

3.0.1 (Monday 27 January 2020)

Bug fixes

- Test failed by using array method on tuple. (pr/860) (Ben Darwin, reviewed by CM)
- Validate `ExpiredDeprecationErrors`, promoted by 3.0 release from `DeprecationWarnings`. (pr/857) (CM)

Maintenance

- Remove logic accommodating numpy without float16 types. (pr/866) (CM)
- Accommodate new numpy dtype strings. (pr/858) (CM)

3.0.0 (Wednesday 18 December 2019)

New features

- ArrayProxy `__array__()` now accepts a `dtype` parameter, allowing `numpy.array(dataobj, dtype=...)` calls, as well as casting directly with a dtype (for example, `numpy.float32(dataobj)`) to control the output type. Scale factors (slope, intercept) are applied, but may be cast to narrower types, to control memory usage. This is now the basis of `img.get_fdata()`, which will scale data in single precision if the output type is `float32`. (pr/844) (CM, reviewed by Alejandro de la Vega, Ross Markello)
- GiftiImage method `agg_data()` to return usable data arrays (pr/793) (Hao-Ting Wang, reviewed by CM)
- Accept `os.PathLike` objects in place of filenames (pr/610) (Cameron Riddell, reviewed by MB, CM)
- Function to calculate obliquity of affines (pr/815) (Oscar Esteban, reviewed by MB)

Enhancements

- Improve testing of data scaling in ArrayProxy API (pr/847) (CM, reviewed by Alejandro de la Vega)
- Document `SpatialImage.slicer` interface (pr/846) (CM)
- `get_fdata(dtype=np.float32)` will attempt to avoid casting data to `np.float64` when scaling parameters would otherwise promote the data type unnecessarily. (pr/833) (CM, reviewed by Ross Markello)
- `ArraySequence` now supports a large set of Python operators to combine or update in-place. (pr/811) (MC, reviewed by Serge Koudoro, Philippe Poulin, CM, MB)
- Warn, rather than fail, on DICOMs with unreadable Siemens CSA tags (pr/818) (Henry Braun, reviewed by CM)
- Improve clarity of coordinate system tutorial (pr/823) (Egor Panfilov, reviewed by MB)

Bug fixes

- Sliced `Tractograms` no longer `apply_affine` to the original `Tractogram`'s streamlines. (pr/811) (MC, reviewed by Serge Koudoro, Philippe Poulin, CM, MB)
- Change strings with invalid escapes to raw strings (pr/827) (EL, reviewed by CM)
- Re-import externals/netcdf.py from scipy to resolve numpy deprecation (pr/821) (CM)

Maintenance

- Remove replicated metadata for packaged data from MANIFEST.in (pr/845) (CM)
- Support Python >=3.5.1, including Python 3.8.0 (pr/787) (CM)
- Manage versioning with slightly customized Versioneer (pr/786) (CM)
- Reference Nipy Community Code and Nibabel Developer Guidelines in GitHub community documents (pr/778) (CM, reviewed by MB)

API changes and deprecations

- Fully remove deprecated `checkwarns` and `minc` modules. (pr/852) (CM)
- The `keep_file_open` argument to file load operations and `ArrayProxy`s no longer acccepts the value `"auto"`, raising a `ValueError`. (pr/852) (CM)
- Deprecate `ArraySequence.data` in favor of `ArraySequence.get_data()`, which will return a copy. `ArraySequence.data` now returns a read-only view. (pr/811) (MC, reviewed by Serge Koudoro, Philippe Poulin, CM, MB)
- Deprecate `DataobjImage.get_data()` API, to be removed in nibabel 5.0 (pr/794, pr/809) (CM, reviewed by MB)

2.5.1 (Monday 23 September 2019)

Enhancements

- Ignore endianness in `nib-diff` if values match (pr/799) (YOH, reviewed by CM)

Bug fixes

- Correctly handle Philips DICOMs w/ derived volume (pr/795) (Mathias Goncalves, reviewed by CM)
- Raise CSA tag limit to 1000, parametrize for future relaxing (pr/798, backported to 2.5.x in pr/800) (Henry Braun, reviewed by CM, MB)
- Coerce data types to match NIfTI intent codes when writing GIFTI data arrays (pr/806) (CM, reported by Tom Holroyd)

Maintenance

- Require h5py 2.10 for Windows + Python < 3.6 to resolve unexpected dtypes in Minc2 data (pr/804) (CM, reviewed by YOH)

API changes and deprecations

- Deprecate `nicom.dicomwrappers.Wrapper.get_affine()` in favor of `affine` property; final removal in nibabel 4.0 (pr/796) (YOH, reviewed by CM)

2.5.0 (Sunday 4 August 2019)

The 2.5.x series is the last with support for either Python 2 or Python 3.4. Extended support for this series 2.5 will last through December 2020.

Thanks for the test ECAT file and fix provided by Andrew Crabb.

Enhancements

- Add SerializableImage class with to/from_bytes methods (pr/644) (CM, reviewed by MB)
- Check CIFTI-2 data shape matches shape described by header (pr/774) (Michiel Cottaar, reviewed by CM)

Bug fixes

- Handle stricter numpy casting rules in tests (pr/768) (CM) reviewed by PM)
- TRK header fields flipped in files written on big-endian systems (pr/782) (CM, reviewed by YOH, MB)
- Load multiframe ECAT images with Python 3 (CM and Andrew Crabb)

Maintenance

- Fix CodeCov paths on Appveyor for more accurate coverage (pr/769) (CM)
- Move to setuptools and reduce use `nisext` functions (pr/764) (CM, reviewed by YOH)
- Better handle test setup/teardown (pr/785) (CM, reviewed by YOH)

API changes and deprecations

- Effect threatened warnings and set some deprecation timelines (pr/755) (CM) * Trackvis methods now default to v2 formats * `nibabel.trackvis` scheduled for removal in nibabel 4.0 * `nibabel.minc` and `nibabel.MincImage` will be removed in nibabel 3.0

2.4.1 (Monday 27 May 2019)

Contributions from Egor Pafilov, Jath Palasubramaniam, Richard Nemec, and Dave Allured.

Enhancements

- Enable `mmap`, `keep_file_open` options when loading any `DataobjImage` (pr/759) (CM, reviewed by PM)

Bug fixes

- Ensure loaded GIFTI files expose writable data arrays (pr/750) (CM, reviewed by PM)
- Safer warning registry manipulation when checking for overflows (pr/753) (CM, reviewed by MB)
- Correctly write .annot files with duplicate lables (pr/763) (Richard Nemec with CM)

Maintenance

- Fix typo in coordinate systems doc (pr/751) (Egor Panfilov, reviewed by CM)
- Replace invalid MINC1 test file with fixed file (pr/754) (Dave Allured with CM)
- Update Sphinx config to support recent Sphinx/numpydoc (pr/749) (CM, reviewed by PM)
- Pacify `FutureWarning` and `DeprecationWarning` from h5py, numpy (pr/760) (CM)
- Accommodate Python 3.8 deprecation of collections.MutableMapping (pr/762) (Jath Palasubramaniam, reviewed by CM)

API changes and deprecations

- Deprecate `keep_file_open == 'auto'` (pr/761) (CM, reviewed by PM)

2.4.0 (Monday 1 April 2019)

New features

- Alternative `Axis`-based interface for manipulating CIFTI-2 headers (pr/641) (Michiel Cottaar, reviewed by Demian Wassermann, CM, SG)

Enhancements

- Accept TCK files produced by tools with other delimiter/EOF defaults (pr/720) (Soichi Hayashi, reviewed by CM, MB, MC)
- Allow BrainModels or Parcels to contain a single vertex in CIFTI (pr/739) (Michiel Cottaar, reviewed by CM)
- Support for `NIFTI_XFORM_TEMPLATE_OTHER` xform code (pr/743) (CM)

Bug fixes

- Skip refcheck in ArraySequence construction/extension (pr/719) (Ariel Rokem, reviewed by CM, MC)
- Use safe resizing for ArraySequence extension (pr/724) (CM, reviewed by MC)
- Fix typo in error message (pr/726) (Jon Haitz Legarreta Gorroño, reviewed by CM)
- Support DICOM slice sorting in Python 3 (pr/728) (Samir Reddigari, reviewed by CM)
- Correctly reorient dim_info when reorienting NIfTI images (Konstantinos Raktivan, CM, reviewed by CM)

Maintenance

- Import updates to reduce upstream deprecation warnings (pr/711, pr/705, pr/738) (EL, YOH, reviewed by CM)
- Delay import of `nibabel.testing`, `nose` and `mock` to speed up import (pr/699) (CM)
- Increase coverage testing, drop coveralls (pr/722, pr/732) (CM)
- Add Zenodo metadata, sorted by commits (pr/732) (CM + others)
- Update author listing and copyrights (pr/742) (MB, reviewed by CM)

2.3.3 (Wednesday 16 January 2019)

Maintenance

- Restore `six` dependency (pr/714) (CM, reviewed by Gael Varoquaux, MB)

2.3.2 (Wednesday 2 January 2019)

Enhancements

- Enable toggling crosshair with `Ctrl-x` in `OrthoSlicer3D` viewer (pr/701) (Miguel Estevan Moreno, reviewed by CM)

Bug fixes

- Read .PAR files corresponding to ADC maps (pr/685) (Gregory R. Lee, reviewed by CM)
- Increase maximum number of items read from Siemens CSA format (Igor Solovey, reviewed by CM, MB)
- Check boolean dtypes with `numpy.issubdtype(..., np.bool_)` (pr/707) (Jon Haitz Legarreta Gorroño, reviewed by CM)

Maintenance

- Fix small typos in parrec2nii help text (pr/682) (Thomas Roos, reviewed by MB)
- Remove deprecated calls to `numpy.asscalar` (pr/686) (CM, reviewed by Gregory R. Lee)
- Update QA directives to accommodate Flake8 3.6 (pr/695) (CM)
- Update DOI links to use `https://doi.org` (pr/703) (Katrin Leinweber, reviewed by CM)
- Remove deprecated calls to `numpy.fromstring` (pr/700) (Ariel Rokem, reviewed by CM, MB)
- Drop `distutils` support, require `bz2file` for Python 2.7 (pr/700) (CM, reviewed by MB)
- Replace mutable `bytes` hack, disabled in numpy pre-release, with `bytearray`/`readinto` strategy (pr/700) (Ariel Rokem, CM, reviewed by CM, MB)

API changes and deprecations

- Add `Opener.readinto` method to read file contents into pre-allocated buffers (pr/700) (Ariel Rokem, reviewed by CM, MB)

2.3.1 (Tuesday 16 October 2018)

New features

- `nib-diff` command line tool for comparing image files (pr/617, pr/672, pr/678) (CC, reviewed by YOH, Pradeep Raamana and CM)

Enhancements

- Speed up reading of numeric arrays in CIFTI2 (pr/655) (Michiel Cottaar, reviewed by CM)
- Add `ndim` property to `ArrayProxy` and `DataobjImage` (pr/674) (CM, reviewed by MB)

Bug fixes

- Deterministic deduction of slice ordering in degenerate cases (pr/647) (YOH, reviewed by CM)
- Allow 0ms TR in MGH files (pr/653) (EL, reviewed by CM)
- Allow for PPC64 little-endian long doubles (pr/658) (MB, reviewed by CM)
- Correct construction of FreeSurfer annotation labels (pr/666) (CM, reviewed by EL, Paul D. McCarthy)
- Fix logic for persisting filehandles with indexed-gzip (pr/679) (Paul D. McCarthy, reviewed by CM)

Maintenance

- Fix semantic error in coordinate systems documentation (pr/646) (Ariel Rokem, reviewed by CM, MB)
- Test on Python 3.7, minor associated fixes (pr/651) (CM, reviewed by Gregory R. Lee, MB)

2.3 (Tuesday 12 June 2018)

New features

- TRK <=> TCK streamlines conversion CLI tools (pr/606) (MC, reviewed by CM)
- Image slicing for SpatialImages (pr/550) (CM)

Enhancements

- Simplfiy MGHImage and add footer fields (pr/569) (CM, reviewed by MB)
- Force sform/qform codes to be ints, rather than numpy types (pr/575) (Paul McCarthy, reviewed by MB, CM)
- Auto-fill color table in FreeSurfer annotation file (pr/592) (PM, reviewed by CM, MB)
- Set default intent code for CIFTI2 images (pr/604) (Mathias Goncalves, reviewed by CM, SG, MB, Tim Coalson)
- Raise informative error on empty files (pr/611) (Pradeep Raamana, reviewed by CM, MB)
- Accept degenerate filenames such as `.nii` (pr/621) (Dimitri Papadopoulos-Orfanos, reviewed by Yaroslav Halchenko)
- Take advantage of `IndexedGzipFile` `drop_handles` flag to release filehandles by default (pr/614) (PM, reviewed by CM, MB)

Bug fixes

- Preserve first point of *LazyTractogram* (pr/588) (MC, reviewed by Nil Goyette, CM, MB)
- Stop adding extraneous metadata padding (pr/593) (Jon Stutters, reviewed by CM, MB)
- Accept lower-case orientation codes in TRK files (pr/600) (Kesshi Jordan, MB, reviewed by MB, MC, CM)
- Annotation file reading (pr/592) (PM, reviewed by CM, MB)
- Fix buffer size calculation in ArraySequence (pr/597) (Serge Koudoro, reviewed by MC, MB, Eleftherios Garyfallidis, CM)
- Resolve `UnboundLocalError` in Python 3 (pr/607) (Jakub Kaczmarzyk, reviewed by MB, CM)
- Do not crash on non-`ImportError` failures in optional imports (pr/618) (Yaroslav Halchenko, reviewed by CM)
- Return original array from `get_fdata` for array image, if no cast required (pr/638, MB, reviewed by CM)

Maintenance

- Use SSH address to use key-based auth (pr/587) (CM, reviewed by MB)
- Fix doctests for numpy 1.14 array printing (pr/591) (MB, reviewed by CM)
- Refactor for pydicom 1.0 API changes (pr/599) (MB, reviewed by CM)
- Increase test coverage, remove unreachable code (pr/602) (CM, reviewed by Yaroslav Halchenko, MB)
- Move `nib-ls` and other programs to a new cmdline module (pr/601, pr/615) (Chris Cheng, reviewed by MB, Yaroslav Halchenko)
- Remove deprecated numpy indexing (EL, reviewed by CM)
- Update documentation to encourage `get_fdata` over `get_data` (pr/637, MB, reviewed by CM)

API changes and deprecations

- Support for `keep_file_open = 'auto'` as a parameter to `Opener()` will be deprecated in 2.4, for removal in 3.0. Accordingly, support for `openers.KEEP_FILE_OPEN_DEFAULT = 'auto'` will be dropped on the same schedule.
- Drop-in support for `indexed_gzip < 0.7` has been removed.

2.2.1 (Wednesday 22 November 2017)

Bug fixes

- Set L/R labels in orthoview correctly (pr/564) (CM)
- Defer use of ufunc / memmap test - allows "freezing" (pr/572) (MB, reviewed by SG)
- Fix doctest failures with pre-release numpy (pr/582) (MB, reviewed by CM)

Maintenance

- Update documentation around NIfTI qform/sform codes (pr/576) (PM, reviewed by MB, CM) + (pr/580) (Bennet Fauber, reviewed by PM)
- Skip precision test on macOS, newer numpy (pr/583) (MB, reviewed by CM)
- Simplify AppVeyor script, removing conda (pr/584) (MB, reviewed by CM)

2.2 (Friday 13 October 2017)

New features

- CIFTI support (pr/249) (SG, Michiel Cottaar, BC, CM, Demian Wassermann, MB)
- Support for MRtrix TCK streamlines file format (pr/486) (MC, reviewed by MB, Arnaud Bore, J-Donald Tournier, Jean-Christophe Houde)
- Added `get_fdata()` as default method to retrieve scaled floating point data from `DataobjImages` (pr/551) (MB, reviewed by CM, SG)

Enhancements

- Support for alternative header field name variants in .PAR files (pr/507) (Gregory R. Lee)
- Various enhancements to streamlines API by MC: support for reading TRK version 1 (pr/512); concatenation of tractograms using +/+= operators (pr/495); function to concatenate multiple ArraySequence objects (pr/494)
- Support for numpy 1.12 (pr/500, pr/502) (MC, MB)
- Allow dtype specifiers as fileslice input (pr/485) (MB)
- Support "headerless" ArrayProxy specification, enabling memory-efficient ArrayProxy reshaping (pr/521) (CM)
- Allow unknown NIfTI intent codes, add FSL codes (pr/528) (PM)
- Improve error handling for `img.__getitem__` (pr/533) (Ariel Rokem)

- Delegate reorientation to SpatialImage classes (pr/544) (Mark Hymers, CM, reviewed by MB)
- Enable using `indexed_gzip` to reduce memory usage when reading from gzipped NIfTI and MGH files (pr/552) (PM, reviewed by MB, CM)

Bug fixes

- Miscellaneous MINC reader fixes (pr/493) (Robert D. Vincent, reviewed by CM, MB)
- Fix corner case in `wrapstruct.get` (pr/516) (PM, reviewed by CM, MB)

Maintenance

- Fix documentation errors (pr/517, pr/536) (Fernando Perez, Venky Reddy)
- Documentation update (pr/514) (Ivan Gonzalez)
- Update testing to use pre-release builds of dependencies (pr/509) (MB)
- Better warnings when nibabel not on path (pr/503) (MB)

API changes and deprecations

- `header` argument to `ArrayProxy.__init__` is renamed to `spec`
- Deprecation of `header` property of `ArrayProxy` object, for removal in 3.0
- `wrapstruct.get` now returns entries evaluating `False`, instead of `None`
- `DataobjImage.get_data` to be deprecated April 2018, scheduled for removal April 2020

2.1 (Monday 22 August 2016)

New features

- New API for managing streamlines and their different file formats. This adds a new module `nibabel.streamlines` that will eventually deprecate the current trackvis reader found in `nibabel.trackvis` (pr/391) (MC, reviewed by Jean-Christophe Houde, Bago Amirbekian, Eleftherios Garyfallidis, Samuel St-Jean, MB);
- A prototype image viewer using matplotlib (pr/404) (EL, based on a proto-prototype by Paul Ivanov) (Reviewed by Gregory R. Lee, MB);
- Functions for image resampling and smoothing using scipy ndimage (pr/255) (MB, reviewed by EL, BC);
- Add ability to write FreeSurfer morphology data (pr/414) (CM, BC, reviewed by BC);
- Read and write support for DICOM tags in NIfTI Extended Header using pydicom (pr/296) (Eric Kastman).

Enhancements

- Extensions to FreeSurfer module to fix reading and writing of FreeSurfer geometry data (pr/460) (Alexandre Gramfort, Jaakko Leppäkangas, reviewed by EL, CM, MB);
- Various improvements to PAR / REC handling by Gregory R. Lee: supporting multiple TR values (pr/429); output of volume labels (pr/427); fix for some diffusion files (pr/426); option for more sophisticated sorting of volumes (pr/409);
- Original trackvis reader will now allow final streamline to have fewer points than the number declared in the header, with `strict=False` argument to `read` function;
- Helper function to return voxel sizes from an affine matrix (pr/413);
- Fixes to DICOM multiframe reading to avoid assumptions on the position of the multiframe index (pr/439) (Eric M. Baker);
- More robust handling of "CSA" private information in DICOM files (pr/393) (Brendan Moloney);
- More explicit error when trying to read image from non-existent file (pr/455) (Ariel Rokem);
- Extension to *nib-ls* command to show image statistics (pr/437) and other header files (pr/348) (Yarik Halchenko).

Bug fixes

- Fixes to rotation order to generate affine matrices of PAR / REC files (MB, Gregory R Lee).

Maintenance

- Dropped support for Pythons 2.6 and 3.2;
- Comprehensive refactor and generalization of surface / GIFTI file support with improved API and extended tests (pr/352-355, pr/360, pr/365, pr/403) (BC, reviewed by CM, MB);
- Refactor of image classes (pr/328, pr/329) (BC, reviewed by CM);
- Better Appveyor testing on new Python versions (pr/446) (Ariel Rokem);
- Fix shebang lines in scripts for correct install into virtualenvs via pip (pr/434);
- Various fixes for numpy, matplotlib, and PIL / Pillow compatibility (CM, Ariel Rokem, MB);
- Improved test framework for warnings (pr/345) (BC, reviewed by CM, MB);
- New decorator to specify start and end versions for deprecation warnings (MB, reviewed by CM);
- Write qform affine matrix to NIfTI images output by `parrec2nii` (pr/478) (Jasper J.F. van den Bosch, reviewed by Gregory R. Lee, MB).

API changes and deprecations

- Minor API breakage in original (rather than new) trackvis reader. We are now raising a `DataError` if there are too few streamlines in the file, instead of a `HeaderError`. We are raising a `DataError` if the track is truncated when `strict=True` (the default), rather than a `TypeError` when trying to create the points array.
- Change sform code that `parrec2nii` script writes to NIfTI images; change from 2 ("aligned") to 1 ("scanner");
- Deprecation of `get_header`, `get_affine` method of image objects for removal in version 4.0;
- Removed broken `from_filespec` method from image objects, and deprecated `from_filespec` method of ECAT image objects for removal in 4.0;
- Deprecation of `class_map` instance in `imageclasses` module in favor of new image class attributes, for removal in 4.0;
- Deprecation of `ext_map` instance in `imageclasses` module in favor of new image loading API, for removal in 4.0;
- Deprecation of `Header` class in favor of `SpatialHeader`, for removal in 4.0;
- Deprecation of `BinOpener` class in favor of more generic `Opener` class, for removal in 4.0;
- Deprecation of `GiftiMetadata` methods `get_metadata` and `get_rgba`; `GiftiDataArray` methods `get_metadata`, `get_labeltable`, `set_labeltable`; `GiftiImage` methods `get_meta`, `set_meta`. All these deprecated in favor of corresponding properties, for removal in 4.0;
- Deprecation of `giftiio read` and `write` functions in favor of nibabel `load` and `save` functions, for removal in 4.0;
- Deprecation of `gifti.data_tag` function, for removal in 4.0;
- Deprecation of write-access to `GiftiDataArray.num_dim`, and new error when trying to set invalid values for `num_dim`. We will remove write-access in 4.0;
- Deprecation of `GiftiDataArray.from_array` in favor of `GiftiDataArray` constructor, for removal in 4.0;
- Deprecation of `GiftiDataArray to_xml_open, to_xml_close` methods in favor of `to_xml` method, for removal in 4.0;
- Deprecation of `parse_gifti_fast.Outputter` class in favor of `GiftiImageParser`, for removal in 4.0;
- Deprecation of `parse_gifti_fast.parse_gifti_file` function in favor of `GiftiImageParser.parse` method, for removal in 4.0;
- Deprecation of `loadsave` functions `guessed_image_type` and `which_analyze_type`, in favor of new API where each image class tests the file for compatibility during load, for removal in 4.0.

2.0.2 (Monday 23 November 2015)

- Fix for integer overflow on large images (pr/325) (MB);
- Fix for Freesurfer nifti files with unusual dimensions (pr/332) (Chris Markiewicz);
- Fix typos on benchmarks and tests (pr/336, pr/340, pr/347) (Chris Markiewicz);
- Fix Windows install script (pr/339) (MB);
- Support for Python 3.5 (pr/363) (MB) and numpy 1.10 (pr/358) (Chris Markiewicz);
- Update pydicom imports to permit version 1.0 (pr/379) (Chris Markiewicz);
- Workaround for Python 3.5.0 gzip regression (pr/383) (Ben Cipollini).
- tripwire.TripWire object now raises subclass of AttributeError when trying to get an attribute, rather than a direct subclass of Exception. This prevents Python 3.5 triggering the tripwire when doing inspection prior to running doctests.
- Minor API change for tripwire.TripWire object; code that checked for AttributeError will now also catch TripWireError.

2.0.1 (Saturday 27 June 2015)

Contributions from Ben Cipollini, Chris Markiewicz, Alexandre Gramfort, Clemens Bauer, github user freec84.

- Bugfix release with minor new features;
- Added `axis` parameter to `concat_images` (pr/298) (Ben Cipollini);
- Fix for unsigned integer data types in ECAT images (pr/302) (MB, test data and issue report from Github user freec84);
- Added new ECAT and Freesurfer data files to automated testing;
- Fix for Freesurfer labels error on early numpies (pr/307) (Alexandre Gramfort);
- Fixes for PAR / REC header parsing (pr/312) (MB, issue reporting and test data by Clemens C. C. Bauer);
- Workaround for reading Freesurfer ico7 surface files (pr/315) (Chris Markiewicz);
- Changed to github pages for doc hosting;
- Changed docs to point to neuroimaging@python.org mailing list.

2.0.0 (Tuesday 9 December 2014)

This release had large contributions from Eric Larson, Brendan Moloney, Nolan Nichols, Basile Pinsard, Chris Johnson and Nikolaas N. Oosterhof.

- New feature, bugfix release with minor API breakage;
- Minor API breakage: default write of NIfTI / Analyze image data offset value. The data offset is the number of bytes from the beginning of file to skip before reading the image data. Nibabel behavior changed from keeping the value as read from file, to setting the offset to zero on read, and setting the offset when writing the header. The value of the offset will now be the minimum value necessary to make room for the header and any extensions when writing the file. You can override the default offset by setting value explicitly to some value other than zero. To read the original data offset as read from the header, use the `offset` property of the image `dataobj` attribute;

- Minor API breakage: data scaling in NIfTI / Analyze now set to NaN when reading images. Data scaling refers to the data intercept and slope values in the NIfTI / Analyze header. To read the original data scaling you need to look at the `slope` and `inter` properties of the image `dataobj` attribute. You can set scaling explicitly by setting the slope and intercept values in the header to values other than NaN;
- New API for managing image caching; images have an `in_memory` property that is true if the image data has been loaded into cache, or is already an array in memory; `get_data` has new keyword argument `caching` to specify whether the cache should be filled by `get_data`;
- Images now have properties `dataobj`, `affine`, `header`. We will slowly phase out the `get_affine` and `get_header` image methods;
- The image `dataobj` can be sliced using an efficient algorithm to avoid reading unnecessary data from disk. This makes it possible to do very efficient reads of single volumes from a time series;
- NIfTI2 read / write support;
- Read support for MINC2;
- Much extended read support for PAR / REC, largely due to work from Eric Larson and Gregory R. Lee on new code, advice and code review. Thanks also to Jeff Stevenson and Bennett Landman for helpful discussion;
- `parrec2nii` script outputs images in LAS voxel orientation, which appears to be necessary for compatibility with FSL `dtifit` / `fslview` diffusion analysis pipeline;
- Preliminary support for Philips multiframe DICOM images (thanks to Nolan Nichols, Ly Nguyen and Brendan Moloney);
- New function to save Freesurfer annotation files (by Github user ohinds);
- Method to return MGH format `vox2ras_tkr` affine (Eric Larson);
- A new API for reading unscaled data from NIfTI and other images, using `img.dataobj.get_unscaled()`. Deprecate previous way of doing this, which was to read data with the `read_img_data` function;
- Fix for bug when replacing NaN values with zero when writing floating point data as integers. If the input floating point data range did not include zero, then NaN would not get written to a value corresponding to zero in the output;
- Improvements and bug fixes to image orientation calculation and DICOM wrappers by Brendan Moloney;
- Bug fixes writing GIfTI files. We were using a base64 encoding that didn't match the spec, and the wrong field name for the endian code. Thanks to Basile Pinsard and Russ Poldrack for diagnosis and fixes;
- Bug fix in `freesurfer.read_annot` with `orig_ids=False` when annot contains vertices with no label (Alexandre Gramfort);
- More tutorials in the documentation, including introductory tutorial on DICOM, and on coordinate systems;
- Lots of code refactoring, including moving to common code-base for Python 2 and Python 3;
- New mechanism to add images for tests via git submodules.

1.3.0 (Tuesday 11 September 2012)

Special thanks to Chris Johnson, Brendan Moloney and JB Poline.

- New feature and bugfix release
- Add ability to write Freesurfer triangle files (Chris Johnson)
- Relax threshold for detecting rank deficient affines in orientation detection (JB Poline)
- Fix for DICOM slice normal numerical error (issue #137) (Brendan Moloney)
- Fix for Python 3 error when writing zero bytes for offset padding

1.2.2 (Wednesday 27 June 2012)

- Bugfix release
- Fix longdouble tests for Debian PPC (thanks to Yaroslav Halchecko for finding and diagnosing these errors)
- Generalize longdouble tests in the hope of making them more robust
- Disable saving of float128 nifti type unless platform has real IEEE binary128 longdouble type.

1.2.1 (Wednesday 13 June 2012)

Particular thanks to Yaroslav Halchecko for fixes and cleanups in this release.

- Bugfix release
- Make compatible with pydicom 0.9.7
- Refactor, rename nifti diagnostic script to `nib-nifti-dx`
- Fix a bug causing an error when analyzing affines for orientation, when the affine contained all 0 columns
- Add missing `dicomfs` script to installation list and rename to `nib-dicomfs`

1.2.0 (Sunday 6 May 2012)

This release had large contributions from Krish Subramaniam, Alexandre Gramfort, Cindee Madison, Félix C. Morency and Christian Haselgrove.

- New feature and bugfix release
- Freesurfer format support by Krish Subramaniam and Alexandre Gramfort.
- ECAT read write support by Cindee Madison and Félix C. Morency.
- A DICOM fuse filesystem by Christian Haselgrove.
- Much work on making data scaling on read and write more robust to rounding error and overflow (MB).
- Import of nipy functions for working with affine transformation matrices.
- Added methods for working with nifti sform and qform fields by Bago Amirbekian and MB, with useful discussion by Brendan Moloney.
- Fixes to read / write of RGB analyze images by Bago Amirbekian.
- Extensions to `concat_images` by Yannick Schwartz.

- A new `nib-ls` script to display information about neuroimaging files, and various other useful fixes by Yaroslav Halchenko.

1.1.0 (Thursday 28 April 2011)

Special thanks to Chris Burns, Jarrod Millman and Yaroslav Halchenko.

- New feature release
- Python 3.2 support
- Substantially enhanced gifti reading support (Stephan Gerhard)
- Refactoring of trackvis read / write to allow reading and writing of voxel points and mm points in tracks. Deprecate use of negative voxel sizes; set voxel_order field in trackvis header. Thanks to Chris Filo Gorgolewski for pointing out the problem and Ruopeng Wang in the trackvis forum for clarifying the coordinate system of trackvis files.
- Added routine to give approximate array orientation in form such as 'RAS' or 'LPS'
- Fix numpy dtype hash errors for numpy 1.2.1
- Other bug fixes as for 1.0.2

1.0.2 (Thursday 14 April 2011)

- Bugfix release
- Make inference of data type more robust to changes in numpy dtype hashing
- Fix incorrect thresholds in quaternion calculation (thanks to Yarik H for pointing this one out)
- Make parrec2nii pass over errors more gracefully
- More explicit checks for missing or None field in trackvis and other classes - thanks to Marc-Alexandre Cote
- Make logging and error level work as expected - thanks to Yarik H
- Loading an image does not change qform or sform - thanks to Yarik H
- Allow 0 for nifti scaling as for spec - thanks to Yarik H
- nifti1.save now correctly saves single or pair images

1.0.1 (Wednesday 23 Feb 2011)

- Bugfix release
- Fix bugs in tests for data package paths
- Fix leaks of open filehandles when loading images (thanks to Gael Varoquaux for the report)
- Skip rw tests for SPM images when scipy not installed
- Fix various windows-specific file issues for tests
- Fix incorrect reading of byte-swapped trackvis files
- Workaround for odd numpy dtype comparisons leading to header errors for some loaded images (thanks to Cindee Madison for the report)

1.0.0 (Thursday, 13, Oct 2010)

- This is the first public release of the NiBabel package.
- NiBabel is a complete rewrite of the PyNifti package in pure python. It was designed to make the code simpler and easier to work with. Like PyNifti, NiBabel has fairly comprehensive NIfTI read and write support.
- Extended support for SPM Analyze images, including orientation affines from matlab `.mat` files.
- Basic support for simple MINC 1.0 files (MB). Please let us know if you have MINC files that we don't support well.
- Support for reading and writing PAR/REC images (MH)
- `parrec2nii` script to convert PAR/REC images to NIfTI format (MH)
- Very preliminary, limited and highly experimental DICOM reading support (MB, Ian Nimmo Smith).
- Some functions (*nibabel.funcs*) for basic image shape changes, including the ability to transform to the image with data closest to the cononical image orientation (first axis left-to-right, second back-to-front, third down-to-up) (MB, Jonathan Taylor)
- Gifti format read and write support (preliminary) (Stephen Gerhard)
- Added utilities to use nipy-style data packages, by rip then edit of nipy data package code (MB)
- Some improvements to release support (Jarrod Millman, MB, Fernando Perez)
- Huge downward step in the quality and coverage by the docs, caused by MB, mostly fixed by a lot of good work by MH.
- NiBabel will not work with Python < 2.5, and we haven't even tested it with Python 3. We will get to it soon...

PyNifti releases

Modifications are done by Michael Hanke, if not indicated otherwise. 'Closes' statement IDs refer to the Debian bug tracking system and can be queried by visiting the URL:

```
http://bugs.debian.org/<bug id>
```

0.20100706.1 (Tue, 6 Jul 2010)

- Bugfix: NiftiFormat.vx2s() used the qform not the sform. Thanks to Tom Holroyd for reporting.

0.20100412.1 (Mon, 12 Apr 2010)

- Bugfix: Unfortunate interaction between Python garbage collection and C library caused memory problems. Thanks to Yaroslav Halchenko for the diagnose and fix.

0.20090303.1 (Tue, 3 Mar 2009)

- Bugfix: Updating the NIfTI header from a dictionary was broken.
- Bugfix: Removed left-over print statement in extension code.
- Bugfix: Prevent saving of bogus 'None.nii' images when the filename was previously assign, before calling NiftiImage.save() (Closes: #517920).
- Bugfix: Extension length was to short for all *edata* whos length matches n*16-8, for all integer n.

0.20090205.1 (Thu, 5 Feb 2009)

- This release is the first in a series that aims stabilize the API and finally result in PyNIfTI 1.0 with full support of the NIfTI1 standard.
- The whole package was restructured. The included renaming *nifti.nifti(image,format,clibs)* to *nifti.(image,format,clibs)*. Redirect modules make sure that existing user code will not break, but they will issue a DeprecationWarning and will be removed with the release of PyNIfTI 1.0.
- Added a special extension that can embed any serializable Python object into the NIfTI file header. The contents of this extension is automatically expanded upon request into the *.meta* attribute of each NiftiImage. When saving files to disk the content of the dictionary is also automatically dumped into this extension. Embedded meta data is not loaded automatically, since this has security implications, because code from the file header is actually executed. The documentation explicitely mentions this risk.
- Added `NiftiExtensions`. This is a container-like handler to access and manipulate NIfTI1 header extensions.
- Exposed `MemMappedNiftiImage` in the root module.
- Moved `cropImage()` into the `utils` module.
- From now on Sphinx is used to generate the documentation. This includes a module reference that replaces that old API reference.
- Added methods `vx2q()` and `vx2s()` to convert voxel indices into coordinates defined by qform or sform respectively.
- Updating the *cal_min* and *cal_max* values in the NIfTI header when saving a file is now conditional, but remains enabled by default.
- Full set of methods to query and modify axis units. This includes expanding the previous *xyzt_units* field in the header dictionary into editable *xyz_unit* and *time_unit* fields. The former *xyzt_units* field is no longer available. See: `getXYZUnit()`, `setXYZUnit()`, `getTimeUnit()`, `setTimeUnit()`, `xyz_unit`, `time_unit`
- Full set of methods to query and manuipulate qform and sform codes. See: `getQFormCode()`, `setQFormCode()`, `getSFormCode()`, `setSFormCode()`, `qform_code`, `sform_code`
- Each image instance is now able to generate a human-readable dump of its most important header information via *__str__()*.
- `NiftiImage` objects can now be pickled.
- Switched to NumPy's distutils for building the package. Cleaned and simplified the build procedure. Added optimization flags to SWIG call.
- `nifti.image.NiftiImage.filename` can now also be used to assign a filename.
- Introduced `nifti.__version__` as canonical version string.

- Removed *updateQFormFromQuarternion()* from the list of public methods of `NiftiFormat`. This is an internal method that should not be used in user code. However, a redirect to the new method will remain in-place until PyNIfTI 1.0.
- Bugfix: `getScaledData()` returns a unmodified data array if *slope* is set to zero (as required by the NIfTI standard). Thanks to Thomas Ross for reporting.
- Bugfix: Unicode filenames are now handled properly, as long as they do not contain pure-unicode characters (since the NIfTI library does not support them). Thanks to Gaël Varoquaux for reporting this issue.

0.20081017.1 (Fri, 17 Oct 2008)

- Updated included minimal copy of the nifticlibs to version 1.1.0.
- Few changes to the Makefiles to enhance Posix compatibility. Thanks to Chris Burns.
- When building on non-Debian systems, only add include and library paths pointing to the local nifticlibs copy, when it is actually built. On Debian system the local copy is still not used at all, as a proper nifticlibs package is guaranteed to be available.
- Added minimal setup_egg.py for setuptools users. Thanks to Gaël Varoquaux.
- PyNIfTI now does a proper wrapping of the image data with NumPy arrays, which no longer leads to accidental memory leaks, when accessing array data that has not been copied before (e.g. via the *data* property of NiftiImage). Thanks to Gaël Varoquaux for mentioning this possibility.

0.20080710.1 (Thu, 7 Jul 2008)

- Bugfix: Pointer bug introduced by switch to new NumPy API in 0.20080624 Thanks to Christopher Burns for fixing it.
- Bugfix: Honored DeprecationWarning: sync() -> flush() for memory mapped arrays. Again thanks to Christopher Burns.
- More unit tests and other improvements (e.g. fixed circular imports) done by Christopher Burns.

0.20080630.1 (Tue, 30 Jun 2008)

- Bugfix: NiftiImage caused a memory leak by not calling the NiftiFormat destructor.
- Bugfix: Merged bashism-removal patch from Debian packaging.

0.20080624.1 (Tue, 24 Jun 2008)

- Converted all documentation (including docstrings) into the restructured text format.
- Improved Makefile.
- Included configuration and Makefile support for profiling, API doc generation (via epydoc) and code quality checks (with PyLint).
- Consistently import NumPy as N.
- Bugfix: Proper handling of [qs]form codes, which previously have not been handled at all. Thanks to Christopher Burns for pointing it out.
- Bugfix: Make NiftiFormat work without setFilename(). Thanks to Benjamin Thyreau for reporting.

- Bugfix: setPixDims() stored meaningless values.
- Use new NumPy API and replace deprecated function calls (*PyArray_FromDimsAndData*).
- Initial support for memory mapped access to uncompressed NIfTI files (*MemMappedNiftiImage*).
- Add a proper Makefile and setup.cfg for compiling PyNIfTI under Windows with MinGW.
- Include a minimal copy of the most recent nifticlibs (just libniftiio and znzlib; version 1.0), to lower the threshold to build PyNIfTI on systems that do not provide a developer package for those libraries.

0.20070930.1 (Sun, 30 Sep 2007)

- Relicense under the MIT license, to be compatible with SciPy license. http://www.opensource.org/licenses/mit-license.php
- Updated documentation.

0.20070917.1 (Mon, 17 Sep 2007)

- Bugfix: Can now update NIfTI header data when no filename is set (Closes: #442175).
- Unloading of image data without a filename set is no checked and prevented as it would damage data integrity and the image data could not be recovered.
- Added 'pixdim' property (Yaroslav Halchenko).

0.20070905.1 (Wed, 5 Sep 2007)

- Fixed a bug in the qform/quaternion handling that caused changes to the qform to vanish when saving to file (Yaroslav Halchenko).
- Added more unit tests.
- 'dim' vector in the NIfTI header is now guaranteed to only contain non-zero elements. This caused problems with some applications.

0.20070803.1 (Fri, 3 Aug 2007)

- Does not depend on SciPy anymore.
- Initial steps towards a unittest suite.
- pynifti_pst can now print the peristimulus signal matrix for a single voxel (onsets x time) for easier processing of this information in external applications.
- utils.getPeristimulusTimeseries() can now be used to compute mean and variance of the signal (among others).
- pynifti_pst is able to compute more than just the mean peristimulus timeseries (e.g. variance and standard deviation).
- Set default image description when saving a file if none is present.
- Improved documentation.

0.20070425.1 (Wed, 25 Apr 2007)

- Improved documentation. Added note about the special usage of the header property. Also added notes about the relevant properties in the docstring of the corresponding accessor methods.
- Added property and accessor methods to access/modify the repetition time of timeseries (dt).
- Added functions to manipulate the pixdim values.
- Added utils.py with some utility functions.
- Added functions/property to determine the bounding box of an image.
- Fixed a bug that caused a corrupted sform matrix when converting a NumPy array and a header dictionary into a NIfTI image.
- Added script to compute peristimulus timeseries (pynifti_pst).
- Package now depends on python-scipy.

0.20070315.1 (Thu, 15 Mar 2007)

- Removed functionality for "NiftiImage.save() raises an IOError exception when writing the image file fails." (Yaroslav Halchenko)
- Added ability to force a filetype when setting the filename or saving a file.
- Reverse the order of the 'header' and 'load' argument in the NiftiImage constructor. 'header' is now first as it seems to be used more often.
- Improved the source code documentation.
- Added getScaledData() method to NiftiImage that returns a copy of the data array that is scaled with the slope and intercept stored in the NIfTI header.

0.20070301.2 (Thu, 1 Mar 2007)

- Fixed wrong link to the source tarball in README.html.

0.20070301.1 (Thu, 1 Mar 2007)

- Initial upload to the Debian archive. (Closes: #413049)
- NiftiImage.save() raises an IOError exception when writing the image file fails.
- Added extent, volextent, and timepoints properties to NiftiImage class (Yaroslav Halchenko).

0.20070220.1 (Tue, 20 Feb 2007)

- NiftiFile class is renamed to NiftiImage.
- SWIG-wrapped libniftiio functions are no available in the nifticlib module.
- Fixed broken NiftiImage from Numpy array constructor.
- Added initial documentation in README.html.
- Fulfilled a number of Yarik's wishes ;)

0.20070214.1 (Wed, 14 Feb 2007)

- Does not depend on libfslio anymore.
- Up to seven-dimensional dataset are supported (as much as NIfTI can do).
- The complete NIfTI header dataset is modifiable.
- Most image properties are accessable via class attributes and accessor methods.
- Improved documentation (but still a long way to go).

0.20061114 (Tue, 14 Nov 2006)

- Initial release.

10.2 General tutorials

10.2.1 Coordinate systems and affines

A nibabel (and nipy) image is the association of three things:

- The *image data array*: a 3D or 4D *array* of image data
- An *affine array* that tells you the position of the image array data in a *reference space*.
- *image metadata* (data about the data) describing the image, usually in the form of an image *header*.

This document describes how the *affine array* describes the position of the image data in a reference space. On the way we will define what we mean by reference space, and the reference spaces that Nibabel uses.

Introducing Someone

We have scanned someone called "Someone", and we have a two MRI images of their brain, a single EPI volume, and a structural scan. In general we never use the person's name in the image filenames, but we make an exception in this case:

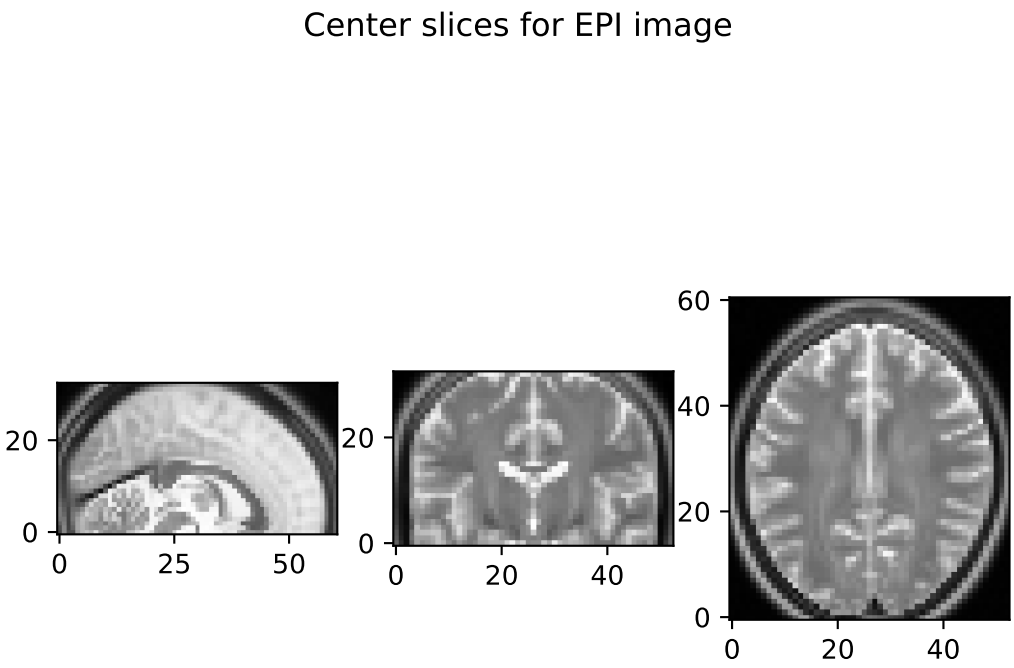
- `someones_epi.nii.gz`.
- `someones_anatomy.nii.gz`.

We can load up the EPI image to get the image data array:

```
>>> import nibabel as nib
>>> epi_img = nib.load('downloads/someones_epi.nii.gz')
>>> epi_img_data = epi_img.get_fdata()
>>> epi_img_data.shape
(53, 61, 33)
```

Then we have a look at slices over the first, second and third dimensions of the array.

```
>>> import matplotlib.pyplot as plt
>>> def show_slices(slices):
...     """ Function to display row of image slices """
...     fig, axes = plt.subplots(1, len(slices))
...     for i, slice in enumerate(slices):
...         axes[i].imshow(slice.T, cmap="gray", origin="lower")
>>>
>>> slice_0 = epi_img_data[26, :, :]
>>> slice_1 = epi_img_data[:, 30, :]
>>> slice_2 = epi_img_data[:, :, 16]
>>> show_slices([slice_0, slice_1, slice_2])
>>> plt.suptitle("Center slices for EPI image")
```



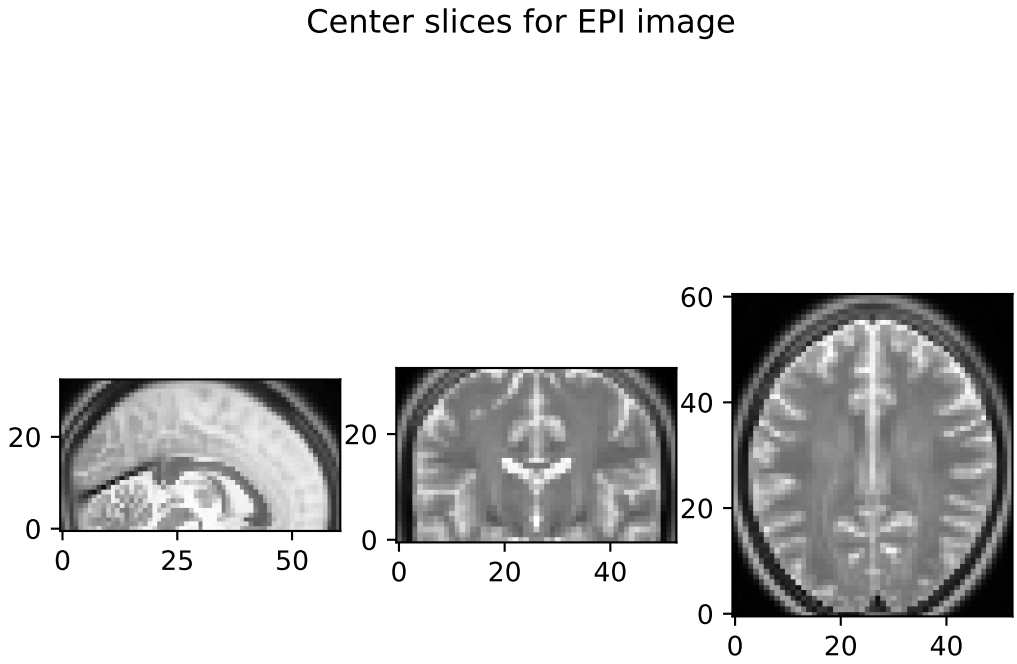
We collected an anatomical image in the same session. We can load that image and look at slices in the three axes:

```
>>> anat_img = nib.load('downloads/someones_anatomy.nii.gz')
```

(continues on next page)

(continued from previous page)

```
>>> anat_img_data = anat_img.get_fdata()
>>> anat_img_data.shape
(57, 67, 56)
>>> show_slices([anat_img_data[28, :, :],
...              anat_img_data[:, 33, :],
...              anat_img_data[:, :, 28]])
>>> plt.suptitle("Center slices for anatomical image")
```



As is usually the case, we had a different field of view for the anatomical scan, and so the anatomical image has a different shape, size, and orientation in the magnet.

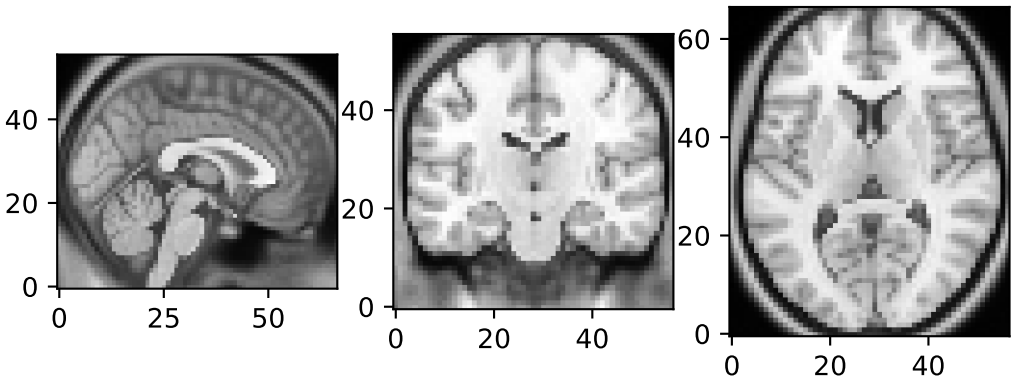
Voxel coordinates are coordinates in the image data array

As y'all know, a voxel is a pixel with volume.

In the code above, `slice_0` from the EPI data is a 2D slice from a 3D image. The plot of the EPI slices displays the slices in grayscale (graded between black for the minimum value, white for the maximum). Each pixel in the slice grayscale image also represents a voxel, because this 2D image represents a slice from the 3D image with a certain thickness.

The 3D array is therefore also a voxel array. As for any array, we can select particular values by indexing. For example, we can get the value for the middle voxel in the EPI data array like this:

Center slices for anatomical image



```
>>> n_i, n_j, n_k = epi_img_data.shape
>>> center_i = (n_i - 1) // 2  # // for integer division
>>> center_j = (n_j - 1) // 2
>>> center_k = (n_k - 1) // 2
>>> center_i, center_j, center_k
(26, 30, 16)
>>> center_vox_value = epi_img_data[center_i, center_j, center_k]
>>> center_vox_value
81.5492877960205...
```

The values (26, 30, 16) are indices into the data array `epi_img_data`. (26, 30, 16) is therefore a 'voxel coordinate' - a coordinate into the voxel array.

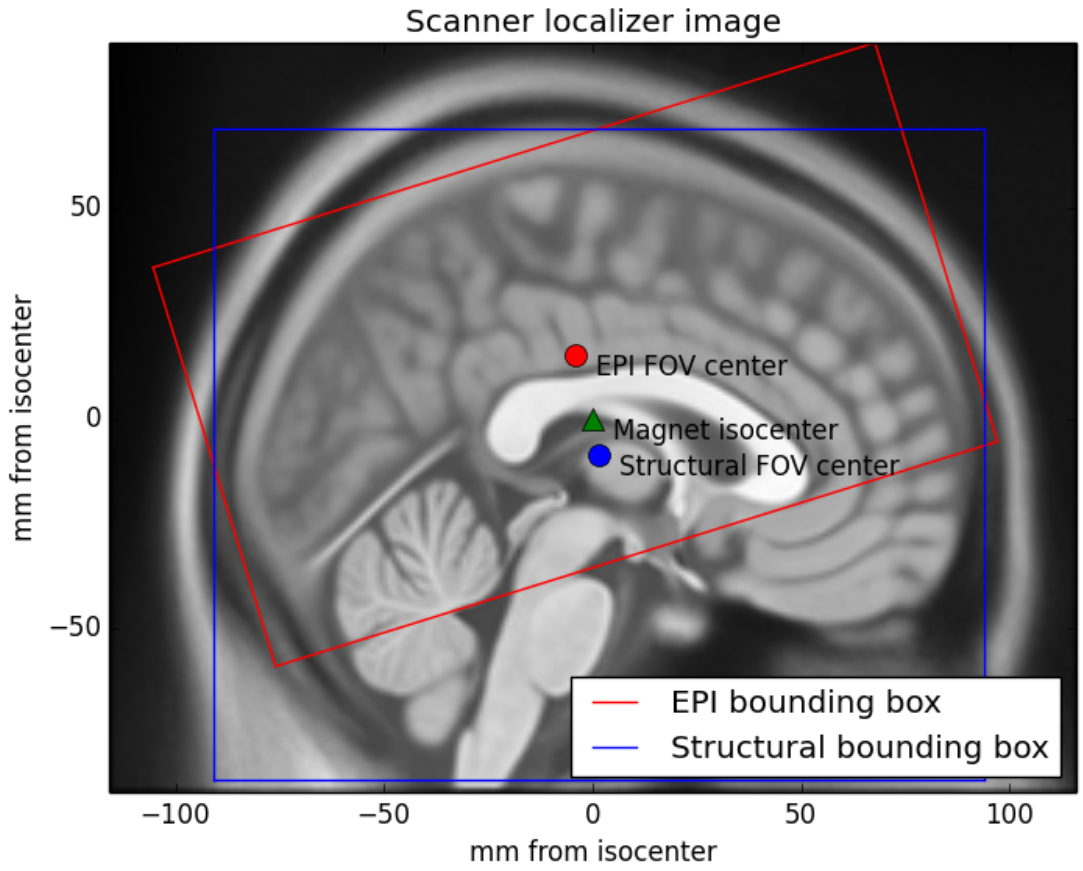
A coordinate is a set of numbers giving positions relative to a set of *axes*. In this case 26 is a position on the first array axis, where the axis is of length `epi_img_data.shape[0]`, and therefore goes from 0 to 52 (`epi_img_data.shape == (53, 61, 33)`). Similarly 30 gives a position on the second axis (0 to 60) and 16 is the position on the third axis (0 to 32).

Voxel coordinates and points in space

The voxel coordinate tells us almost nothing about where the data came from in terms of position in the scanner. For example, let's say we have the voxel coordinate (26, 30, 16). Without more information we have no idea whether this voxel position is on the left or right of the brain, or came from the left or right of the scanner.

This is because the scanner allows us to collect voxel data in almost any arbitrary position and orientation within the magnet.

In the case of Someone's EPI, we took transverse slices at a moderate angle to the floor to ceiling direction. This localizer image from the scanner console has a red box that shows the position of the slice block for `someones_epi.nii.gz` and a blue box for the slice block of `someones_anatomy.nii.gz`:



The localizer is oriented to the magnet, so that the left and right borders of the image are parallel to the floor of the scanner room, with the left border being towards the floor and the right border towards the ceiling.

You will see from the labels on the localizer that the center of the EPI voxel data block (at 26, 30, 16 in `epi_img_data`) is not quite at the center of magnet bore (the magnet *isocenter*).

We have an anatomical and an EPI scan, and later on we will surely want to be able to relate the data from `someones_epi.nii.gz` to `someones_anatomy.nii.gz`. We can't easily do this at the moment, because we collected the anatomical image with a different field of view and orientation to the EPI image, so the voxel coordinates in the EPI image refer to different locations in the magnet to the voxel coordinates in the anatomical image.

We solve this problem by keeping track of the relationship of voxel coordinates to some *reference space*. In particular, the *affine array* stores the relationship between voxel coordinates in the image data array and coordinates in the reference space. We store the relationship of voxel coordinates from `someones_epi.nii.gz` and the reference space, and also the (different) relationship of voxel coordinates in `someones_anatomy.nii.gz` to the *same* reference space. Because we know the relationship of (voxel coordinates to the reference space) for both images, we can use this information to relate voxel coordinates in `someones_epi.nii.gz` to spatially equivalent voxel coordinates in `someones_anatomy.nii.gz`.

The scanner-subject reference space

What does "space" mean in the phrase "reference space"? The space is defined by an ordered set of axes. For our 3D spatial world, it is a set of 3 independent axes.

We can decide what space we want to use, by choosing these axes. We need to choose the origin of the axes, their direction and their units.

To start with, we define a set of three orthogonal *scanner axes*.

The scanner axes

- The origin of the axes is at the magnet isocenter. This is coordinate (0, 0, 0) in our reference space. All three axes pass through the isocenter.
- The units for all three axes are millimeters.
- Imagine an observer standing behind the scanner looking through the magnet bore towards the end of the scanner bed. Imagine a line traveling towards the observer through the center of the magnet bore, parallel to the bed, with the zero point at the magnet isocenter, and positive values closer to the observer. Call this line the *scanner-bore* axis.
- Draw a line traveling from the scanner room floor up through the magnet isocenter towards the ceiling, at right angles to the scanner bore axis. 0 is at isocenter and positive values are towards the ceiling. Call this the *scanner-floor/ceiling* axis.
- Draw a line at right angles to the other two lines, traveling from the observer's left, parallel to the floor, and through the magnet isocenter to the observer's right. 0 is at isocenter and positive values are to the right. Call this the *scanner-left/right*.

If we make the axes have order (scanner left-right; scanner floor-ceiling; scanner bore) then we have an ordered set of 3 axes and therefore the definition of a 3D *space*. Call the first axis the "X" axis, the second "Y" and the third "Z". A coordinate of $(x, y, z) = (10, -5, -3)$ in this space refers to the point in space 10mm to the (fictional observer's) right of isocenter, 5mm towards the floor from the isocenter, and 3mm towards the foot of the scanner bed. This reference space is sometimes known as "scanner XYZ". It was the standard reference space for the predecessor to DICOM, called ACR / NEMA 2.0.

From scanner to subject

If the subject is lying in the usual position for a brain scan, face up and head first in the scanner, then scanner-left/right is also the left-right axis of the subject's head, scanner-floor/ceiling is the posterior-anterior axis of the head and scanner-bore is the inferior-superior axis of the head.

Sometimes the subject is not lying in the standard position. For example, the subject may be lying with their face pointing to the right (in terms of the scanner-left/right axis). In that case "scanner XYZ" will not tell us about the subject's left and right, but only the scanner left and right. We might prefer to know where we are in terms of the subject's left and right.

To deal with this problem, most reference spaces use subject- or patient- centered scanner coordinate systems. In these systems, the axes are still the scanner axes above, but the ordering and direction of the axes comes from the position of the subject. The most common subject-centered scanner coordinate system in neuroimaging is called "scanner RAS" (right, anterior, superior). Here the scanner axes are reordered and flipped so that the first axis is the scanner axis that is closest to the left to right axis of the subject, the second is the closest scanner axis to the posterior-anterior axis of the subject, and the third is the closest scanner axis to the inferior-superior axis of the subject. For example, if the subject was lying face to the right in the scanner, then the first (X) axis of the reference system would be scanner-floor/ceiling, but reversed so that positive values are towards the floor. This axis goes from left to right in the subject, with positive

values to the right. The second (Y) axis would be scanner-left/right (posterior-anterior in the subject), and the Z axis would be scanner-bore (inferior-superior).

Naming reference spaces

Reading names of reference spaces can be confusing because of different meanings that authors use for the same terms, such as 'left' and 'right'.

We are using the term "RAS" to mean that the axes are (in terms of the subject): left to Right; posterior to Anterior; and inferior to Superior, respectively. Although it is common to call this convention "RAS", it is not quite universal, because some use "R", "A" and "S" in "RAS" to mean that the axes *starts* on the right, anterior, superior of the subject, rather than *ending* on the right, anterior, superior. In other words, they would use "RAS" to refer to a coordinate system we would call "LPI". To be safe, we'll call our interpretation of the RAS convention "RAS+", meaning that Right, Anterior, Superior are all positive values on these axes.

Some people also use "right" to mean the right hand side when an observer looks at the front of the scanner, from the foot the scanner bed. Unfortunately, this means that you have to read coordinate system definitions carefully if you are not familiar with a particular convention. We nibabel / nipy folks agree with most of our brain imaging friends and many of our enemies in that we always use "right" to mean the subject's right.

Voxel coordinates are in voxel space

We have not yet made this explicit, but voxel coordinates are also in a space. In this case the space is defined by the three voxel axes (first axis, second axis, third axis), where 0, 0, 0 is the center of the first voxel in the array and the units on the axes are voxels. Voxel coordinates are therefore defined in a reference space called *voxel space*.

The affine matrix as a transformation between spaces

We have voxel coordinates (in voxel space). We want to get scanner RAS+ coordinates corresponding to the voxel coordinates. We need a *coordinate transform* to take us from voxel coordinates to scanner RAS+ coordinates.

In general, we have some voxel space coordinate (i, j, k), and we want to generate the reference space coordinate (x, y, z).

Imagine we had solved this, and we had a coordinate transform function f that accepts a voxel coordinate and returns a coordinate in the reference space:

$$(x, y, z) = f(i, j, k)$$

f accepts a coordinate in the *input* space and returns a coordinate in the *output* space. In our case the input space is voxel space and the output space is scanner RAS+.

In theory f could be a complicated non-linear function, but in practice, we know that the scanner collects data on a regular grid. This means that the relationship between (i, j, k) and (x, y, z) is linear (actually *affine*), and can be encoded with linear (actually affine) transformations comprising translations, rotations and zooms (wikipedia linear transform, wikipedia affine transform).

Scaling (zooming) in three dimensions can be represented by a diagonal 3 by 3 matrix. Here's how to zoom the first dimension by p, the second by q and the third by r units:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} pi \\ qj \\ rk \end{bmatrix} = \begin{bmatrix} p & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & r \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

A rotation in three dimensions can be represented as a 3 by 3 *rotation matrix* (wikipedia rotation matrix). For example, here is a rotation by θ radians around the third array axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

This is a rotation by ϕ radians around the second array axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

A rotation of γ radians around the first array axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

Zoom and rotation matrices can be combined by matrix multiplication.

Here's a scaling of p, q, r units followed by a rotation of θ radians around the third axis followed by a rotation of ϕ radians around the second axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & r \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

This can also be written:

$$M = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & r \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

This might be obvious because the matrix multiplication is the result of applying each transformation in turn on the coordinates output from the previous transformation. Combining the transformations into a single matrix M works because matrix multiplication is associative – $ABCD = (ABC)D$.

A translation in three dimensions can be represented as a length 3 vector to be added to the length 3 coordinate. For example, a translation of a units on the first axis, b on the second and c on the third might be written as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

We can write our function f as a combination of matrix multiplication by some 3 by 3 rotation / zoom matrix M followed by addition of a 3 by 1 translation vector (a, b, c)

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

We could record the parameters necessary for f as the 3 by 3 matrix, M and the 3 by 1 vector (a, b, c).

In fact, the 4 by 4 image *affine array* does include exactly this information. If $m_{i,j}$ is the value in row i column j of matrix M, then the image affine matrix A is:

$$A = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & a \\ m_{2,1} & m_{2,2} & m_{2,3} & b \\ m_{3,1} & m_{3,2} & m_{3,3} & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Why the extra row of $[0, 0, 0, 1]$? We need this row because we have rephrased the combination of rotations / zooms and translations as a transformation in *homogenous coordinates* (see wikipedia homogenous coordinates). This is a trick that allows us to put the translation part into the same matrix as the rotations / zooms, so that both translations and rotations / zooms can be applied by matrix multiplication. In order to make this work, we have to add an extra 1 to our input and output coordinate vectors:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & a \\ m_{2,1} & m_{2,2} & m_{2,3} & b \\ m_{3,1} & m_{3,2} & m_{3,3} & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix}$$

This results in the same transformation as applying M and (a, b, c) separately. One advantage of encoding transformations this way is that we can combine two sets of [rotations, zooms, translations] by matrix multiplication of the two corresponding affine matrices.

In practice, although it is common to combine 3D transformations using 4 by 4 affine matrices, we usually *apply* the transformations by breaking up the affine matrix into its component M matrix and (a, b, c) vector and doing:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

As long as the last row of the 4 by 4 is $[0, 0, 0, 1]$, applying the transformations in this way is mathematically the same as using the full 4 by 4 form, without the inconvenience of adding the extra 1 to our input and output vectors.

The inverse of the affine gives the mapping from scanner to voxel

The affine arrays we have described so far have another pleasant property — they are usually invertible. As y'all know, the inverse of a matrix A is the matrix A^{-1} such that $I = A^{-1}A$, where I is the identity matrix. Put another way:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = A \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix}$$

$$A^{-1} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = A^{-1}A \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix} = A^{-1} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

That means that the inverse of the affine matrix gives the transformation from scanner RAS+ coordinates to voxel coordinates in the image data.

Now imagine we have affine array A for `someones_epi.nii.gz`, and affine array B for `someones_anatomy.nii.gz`. A gives the mapping from voxels in the image data array of `someones_epi.nii.gz` to millimeters in scanner RAS+. B gives the mapping from voxels in image data array of `someones_anatomy.nii.gz` to *the same* scanner RAS+. Now let's say we have a particular voxel coordinate (i, j, k) in the data array of `someones_epi.nii.gz`, and we want to find the voxel in `someones_anatomy.nii.gz` that is in the same spatial position. Call this matching voxel coordinate (i', j', k') . We first apply the transform from `someones_epi.nii.gz` voxels to scanner RAS+ (A) and then apply the transform from scanner RAS+ to voxels in `someones_anatomy.nii.gz`

(B^{-1}):

$$\begin{bmatrix} i' \\ j' \\ k' \\ 1 \end{bmatrix} = B^{-1}A \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix}$$

The affine by example

We can get the affine from the nibabel image object. Here is the affine for the EPI scan:

```
>>> # Set numpy to print 3 decimal points and suppress small values
>>> import numpy as np
>>> np.set_printoptions(precision=3, suppress=True)
>>> # Print the affine
>>> epi_img.affine
array([[  3.   ,   0.   ,   0.   , -78.   ],
       [  0.   ,   2.866,  -0.887, -76.   ],
       [  0.   ,   0.887,   2.866, -64.   ],
       [  0.   ,   0.   ,   0.   ,   1.   ]])
```

As you see, the last row is $[0, 0, 0, 1]$

Applying the affine

To make the affine simpler to apply, we split it into M and (a, b, c):

```
>>> M = epi_img.affine[:3, :3]
>>> abc = epi_img.affine[:3, 3]
```

Then we can define our function f:

```
>>> def f(i, j, k):
...     """ Return X, Y, Z coordinates for i, j, k """
...     return M.dot([i, j, k]) + abc
```

The labels on the *localizer image* give the impression that the center voxel of `someones_epi.nii.gz` was a little above the magnet isocenter. Now we can check:

```
>>> epi_vox_center = (np.array(epi_img_data.shape) - 1) / 2.
>>> f(epi_vox_center[0], epi_vox_center[1], epi_vox_center[2])
array([ 0.   , -4.205,  8.453])
```

That means the center of the image field of view is at the isocenter of the magnet on the left to right axis, and is around 4.2mm posterior to the isocenter and ~8.5 mm above the isocenter.

The parameters in the affine array can therefore give the position of any voxel coordinate, relative to the scanner RAS+ reference space.

We get the same result from applying the affine directly instead of using M and (a, b, c) in our function. As above, we need to add a 1 to the end of the vector to apply the 4 by 4 affine matrix.

```
>>> epi_img.affine.dot(list(epi_vox_center) + [1])
array([ 0.   , -4.205,  8.453,  1.   ])
```

In fact nibabel has a function `apply_affine` that applies an affine to an (i, j, k) point by splitting the affine into M and abc then multiplying and adding as above:

```
>>> from nibabel.affines import apply_affine
>>> apply_affine(epi_img.affine, epi_vox_center)
array([ 0.   , -4.205,  8.453])
```

Now we can apply the affine, we can use matrix inversion on the anatomical affine to map between voxels in the EPI image and voxels in the anatomical image.

```
>>> import numpy.linalg as npl
>>> epi_vox2anat_vox = npl.inv(anat_img.affine).dot(epi_img.affine)
```

What is the voxel coordinate in the anatomical corresponding to the voxel center of the EPI image?

```
>>> apply_affine(epi_vox2anat_vox, epi_vox_center)
array([28.364, 31.562, 36.165])
```

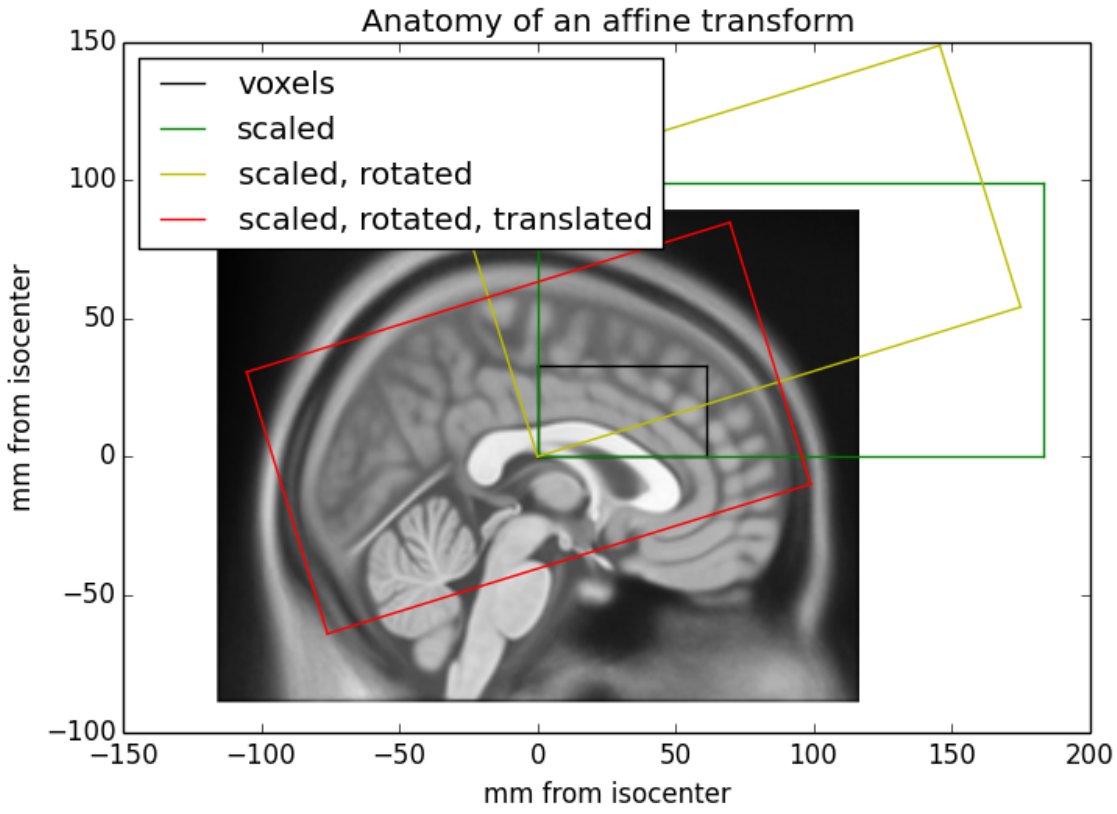
The voxel coordinate of the center voxel of the anatomical image is:

```
>>> anat_vox_center = (np.array(anat_img_data.shape) - 1) / 2.
>>> anat_vox_center
array([28. , 33. , 27.5])
```

The voxel location in the anatomical image that matches the center voxel of the EPI image is nearly exactly half way across the first axis, a voxel or two back from the anatomical voxel center on the second axis, and about 9 voxels above the anatomical voxel center. We can check the *localizer image* by eye to see whether this makes sense, by seeing how the red EPI field of view center relates to the blue anatomical field of view center and the blue anatomical image field of view.

The affine as a series of transformations

You can think of the image affine as a combination of a series of transformations to go from voxel coordinates to mm coordinates in terms of the magnet isocenter. Here is the EPI affine broken down into a series of transformations, with the results shown on the localizer image:



We start by putting the voxel grid onto the isocenter coordinate system, so a translation of one voxel equates to a translation of one millimeter in the isocenter coordinate system. Our EPI image would then have the black bounding box in the image above. Next we scale the voxels to millimeters by scaling by the voxel size (green bounding box). We could do this with an affine:

```
>>> scaling_affine = np.array([[3, 0, 0, 0],
...                            [0, 3, 0, 0],
...                            [0, 0, 3, 0],
...                            [0, 0, 0, 1]])
```

After applying this affine, when we move one voxel in any direction, we are moving 3 millimeters in that direction:

```
>>> one_vox_axis_0 = [1, 0, 0]
>>> apply_affine(scaling_affine, one_vox_axis_0)
array([3, 0, 0])
```

Next we rotate the scaled voxels around the first axis by 0.3 radians (see *rotate around first axis*):

```
>>> cos_gamma = np.cos(0.3)
>>> sin_gamma = np.sin(0.3)
>>> rotation_affine = np.array([[1, 0, 0, 0],
...                            [0, cos_gamma, -sin_gamma, 0],
...                            [0, sin_gamma, cos_gamma, 0],
...                            [0, 0, 0, 1]])
>>> affine_so_far = rotation_affine.dot(scaling_affine)
```

(continues on next page)

(continued from previous page)

```
>>> affine_so_far
array([[ 3.   ,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  2.866, -0.887,  0.   ],
       [ 0.   ,  0.887,  2.866,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  1.   ]])
```

The EPI voxel block coordinates transformed by `affine_so_far` are at the position of the yellow box on the figure.

Finally we translate the 0, 0, 0 coordinate at the bottom, posterior, left corner of our array to be at its final position relative to the isocenter, which is -78, -76, -64:

```
>>> translation_affine = np.array([[1, 0, 0, -78],
...                                [0, 1, 0, -76],
...                                [0, 0, 1, -64],
...                                [0, 0, 0, 1]])
>>> whole_affine = translation_affine.dot(affine_so_far)
>>> whole_affine
array([[  3.   ,   0.   ,   0.   , -78.   ],
       [  0.   ,   2.866,  -0.887, -76.   ],
       [  0.   ,   0.887,   2.866, -64.   ],
       [  0.   ,   0.   ,   0.   ,   1.   ]])
```

This brings the affine-transformed voxel coordinates to the red box on the figure, matching the position on the *localizer*.

Other reference spaces

The scanner RAS+ reference space is a "real-world" space, in the sense that a coordinate in this space refers to a position in the real world, in a particular scanner in a particular room.

Imagine that we used some fancy software to register `someones_epi.nii.gz` to a template image, such as the Montreal Neurological Institute (MNI) template brain. The registration has moved the voxels around in complicated ways — the image has changed shape to match the template brain. We probably do not want to know how the voxel locations relate to the original scanner, but how they relate to the template brain. So, what reference space should we use?

In this case we use a space defined in terms of the template brain — the MNI reference space.

- The origin (0, 0, 0) point is defined to be the point that the anterior commissure of the MNI template brain crosses the midline (the AC point).
- Axis units are millimeters.
- The Y axis follows the midline of the MNI brain between the left and right hemispheres, going from posterior (negative) to anterior (positive), passing through the AC point. The template defines this line.
- The Z axis is at right angles to the Y axis, going from inferior (negative) to superior (positive), with the superior part of the line passing between the two hemispheres.
- The X axis is a line going from the left side of the brain (negative) to right side of the brain (positive), passing through the AC point, and at right angles to the Y and Z axes.

These axes are defined with reference to the template. The exact position of the Y axis, for example, is somewhat arbitrary, as is the definition of the origin. Left and right are left and right as defined by the template. These are the axes and the space that MNI defines for its template.

A coordinate in this reference system gives a position relative to the particular brain template. It is not a real-world space because it does not refer to any particular place but to a position relative to a template.

The axes are still left to right, posterior to anterior and inferior to superior in terms of the template subject. This is still an RAS+ space — the MNI RAS+ space.

An image aligned to this template will therefore have an affine giving the relationship between voxels in the aligned image and the MNI RAS+ space.

There are other reference spaces. For example, we might align an image to the Talairach atlas brain. This brain has a different shape and size than the MNI brain. The origin is the AC point, but the Y axis passes through the point that the posterior commissure crosses the midline (the PC point), giving a slightly different trajectory from the MNI Y axis. Like the MNI RAS+ space, the Talairach axes also run left to right, posterior to anterior and inferior superior, so this is the Talairach RAS+ space.

There are conventions other than RAS+ for the reference space. For example, DICOM files map input voxel coordinates to coordinates in scanner LPS+ space. Scanner LPS+ space uses the same scanner axes and isocenter as scanner RAS+, but the X axis goes from right to the subject's Left, the Y axis goes from anterior to Posterior, and the Z axis goes from inferior to Superior. A positive X coordinate in this space would mean the point was to the subject's *left* compared to the magnet isocenter.

Nibabel always uses an RAS+ output space

Nibabel images always use RAS+ output coordinates, regardless of the preferred output coordinates of the underlying format. For example, we convert affines for DICOM images to output RAS+ coordinates instead of LPS+ coordinates. We chose this convention because it is the most popular in neuroimaging; for example, it is the standard used by NIfTI and MINC formats.

Nibabel does not enforce a particular RAS+ space. For example, NIfTI images contain codes that specify whether the affine maps to scanner or MNI or Talairach RAS+ space. For the moment, you have to consult the specifics of each format to find which RAS+ space the affine maps to.

See also *Radiological vs neurological conventions*

10.2.2 Radiological vs neurological conventions

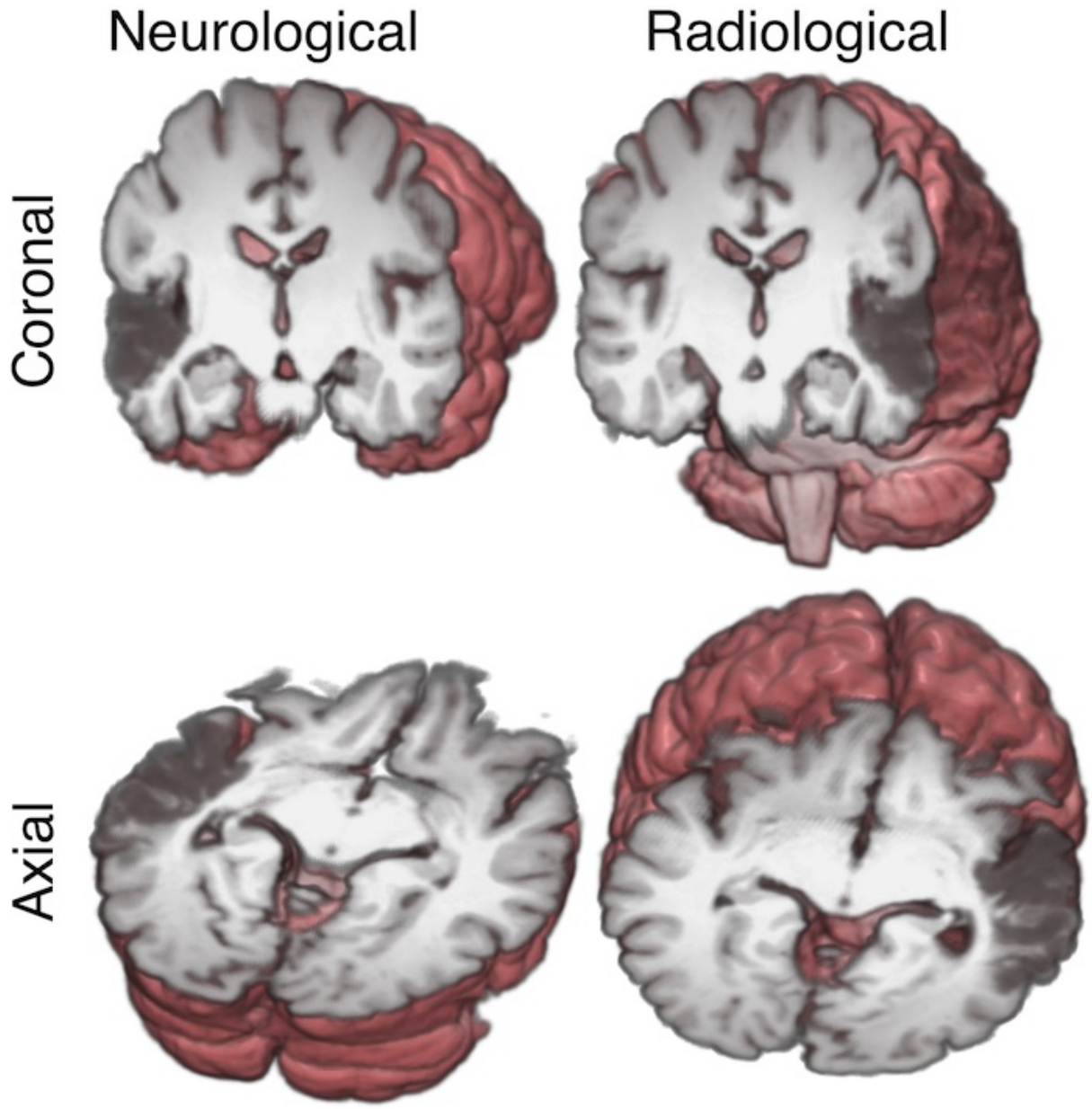
It is relatively common to talk about images being in "radiological" compared to "neurological" convention, but the terms can be used in different and confusing ways.

See *Coordinate systems and affines* for background on voxel space, reference space and affines.

Neurological and radiological display convention

Radiologists like looking at their images with the patient's left on the right of the image. If they are looking at a brain image, it is as if they were looking at the brain slice from the point of view of the patient's feet. Neurologists like looking at brain images with the patient's right on the right of the image. This perspective is as if the neurologist is looking at the slice from the top of the patient's head. The convention is one of image display. The image can have any voxel arrangement on disk or memory, and any output reference space; it is only necessary for the software displaying the image to know the reference space and the (probably affine) mapping between voxel space and reference space; then the software can work out which voxels are on the left or right of the subject and flip the images to the taste of the viewer. We could unpack these uses as *neurological display convention* and *radiological display convention*.

Here is a very nice graphic by Chris Rorden showing these display conventions, where the 3D rendering behind the sections shows the directions that the neurologist and radiologist are thinking of:



In the image above, the subject has a stroke in left temporal lobe, causing a dark area on the MRI.

Alignment of world and voxel axes

As we will see in the next section, radiological and neurological are sometimes used to refer to particular alignments of the voxel input axes to scanner RAS+ output axes. If we look at the affine mapping between voxel space and scanner RAS+, we may find that moving along the first voxel axis by one unit results in a equivalent scanner RAS+ movement that is mainly left to right. This can happen with a diagonal 3x3 part of the affine mapping to scanner RAS+ (see *Coordinate systems and affines*):

```
>>> import numpy as np
>>> from nibabel.affines import apply_affine
```

(continues on next page)

(continued from previous page)

```
>>> diag_affine = np.array([[3., 0,  0,  0],
...                         [0,  3., 0,  0],
...                         [0,  0, 4.5, 0],
...                         [0,  0,  0,  1]])
>>> ijk = [1, 0, 0] # moving one unit on the first voxel axis
>>> apply_affine(diag_affine, ijk)
array([3., 0., 0.])
```

In this case the voxel axes are aligned to the output axes, in the sense that moving in a positive direction on the first voxel axis results in increasing values on the "R+" output axis, and similarly for the second voxel axis with output "A+" and the third voxel axis with output "S+".

Some people therefore refer to this alignment of voxel and RAS+ axes as *RAS voxel axes*.

Neurological / radiological voxel layout

Very confusingly, some people refer to images with RAS voxel axes as having "neurological" voxel layout. This is because the simplest way to display slices from this voxel array will result in the left of the subject appearing towards the left hand side of the screen and therefore neurological display convention. If we take a slice k over the third axis of the image data array (`img_data[:, :, k]`), the resulting slice will have a first array axis going from left to right in terms of spatial position and the second array axis going from posterior to anterior. If we display this image with the first axis going from left to right on screen and the second from bottom to top, it will have the subject's right towards the right of the screen, and anterior towards the top of the screen, as neurologists like it. Here we are showing the middle slice of `an image` with RAS voxel axes:

```
>>> import nibabel as nib
>>> import matplotlib.pyplot as plt
>>> img = nib.load('downloads/someones_anatomy.nii.gz')
>>> # The 3x3 part of the affine is diagonal with all +ve values
>>> img.affine
array([[  2.75,   0.  ,   0.  , -78.  ],
       [  0.  ,   2.75,   0.  , -91.  ],
       [  0.  ,   0.  ,   2.75, -91.  ],
       [  0.  ,   0.  ,   0.  ,   1.  ]])
>>> img_data = img.get_fdata()
>>> a_slice = img_data[:, :, 28]
>>> # Need transpose to put first axis left-right, second bottom-top
>>> plt.imshow(a_slice.T, cmap="gray", origin="lower")
```

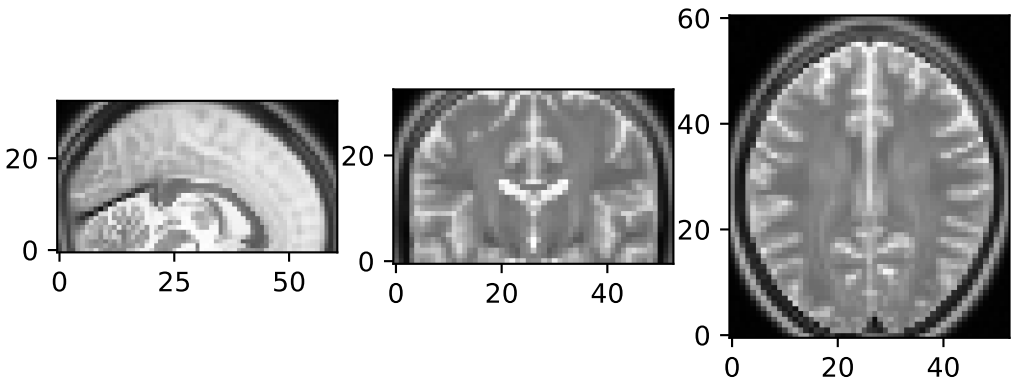
This slice does have the voxels from the right of isocenter towards the right of the screen, neurology style.

Similarly, an "LAS" alignment of voxel axes to RAS+ axes would result in an image with the left of the subject towards the right of the screen, as radiologists like it. "LAS" voxel axes can also be called "radiological" voxel layout for this reason[1].

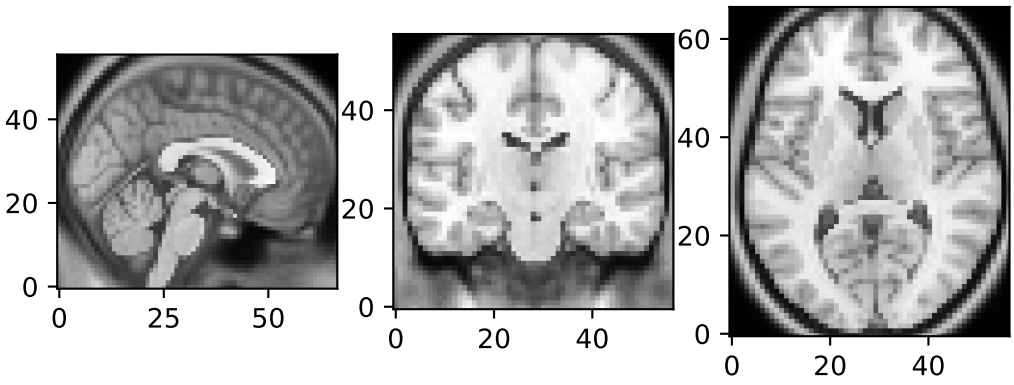
Over time it has become more common for the scanner to generate images with almost any orientation of the voxel axes relative to the reference axes. Maybe for this reason, the terms "radiological" and "neurological" are less commonly used as applied to voxel layout. We nipyers try to avoid the terms neurological or radiological for voxel layout because

[1] We have deliberately not fully defined what we mean by "voxel layout" in the text. Conceptually, an image array could be stored in any layout on disk; the definition of the image format specifies how the image reader should interpret the data on disk to return the right array value for a given voxel coordinate. The relationship of the values on disk to the coordinate values in the array has no bearing on the fact that the voxel axes align to the output axes. In practice the terms RAS / neurological and LAS / radiogical as applied to voxel layout appear to refer exclusively to the situation where image arrays are stored in "Fortran array layout" on disk. Imagine an image array of shape (I, J, K) with values of length v. For an image of 64-bit floating point values, $v = 8$. An image array is stored in Fortran array layout only if the value for voxel coordinate (1, 0, 0) is v bytes on disk from the value for (0, 0, 0); (0, 1, 0) is $I * v$ bytes from (0, 0, 0); and (0, 0, 1) is $I * J * v$ bytes from (0, 0, 0). Analyze and NIfTI images use Fortran array layout.

Center slices for EPI image



Center slices for anatomical image



they can make it harder to separate the idea of voxel and reference space axes and the affine as a mapping between them.

10.2.3 Introduction to DICOM

DICOM defines standards for storing data in memory and on disk, and for communicating this data between machines over a network.

We are interested here in DICOM data. Specifically we are interested in DICOM files.

DICOM files are binary dumps of the objects in memory that DICOM sends across the network.

We need to understand the format that DICOM uses to send messages across the network to understand the terms the DICOM uses when storing data in files.

For example, I hope, by the time you reach the end of this document, you will understand the following complicated and confusing statement from section 7 of the DICOM standards document PS 3.10:

> 7 DICOM File Format
>
> The DICOM File Format provides a means to encapsulate in a file the Data Set representing a SOP Instance related to a DICOM IOD. As shown in Figure 7-1, the byte stream of the Data Set is placed into the file after the DICOM File Meta Information. Each file contains a single SOP Instance.

DICOM is messages

The fundamental task of DICOM is to allow different computers to send messages to one another. These messages can contain data, and the data is very often medical images.

The messages are in the form of requests for an operation, or responses to those requests.

Let's call the requests and the responses - services.

Every DICOM message starts with a stream of bytes containing information about the service. This part of the message is called the DICOM Message Service Element or DIMSE. Depending on what the DIMSE was, there may follow some data related to the request.

For example, there is a DICOM service called "C-ECHO". This asks for a response from another computer to confirm it has seen the echo request. There is no associated data following the "C-ECHO" DIMSE part. So, the full message is the DIMSE "C-ECHO".

There is another DICOM service called "C-STORE". This is a request for the other computer to store some data, such as an image. The data to be stored follows the "C-STORE" DIMSE part.

We go into more detail on this later in the page.

Both the DIMSE and the subsequent data have a particular binary format - consisting of DICOM elements (see below).

Here we will cover:

- what DICOM elements are;
- how DICOM elements are arranged to form complicated data structures such as images;
- how the service part and the data part go together to form whole messages
- how these parts relate to DICOM files.

The DICOM standard

The documents defining the standard are:

Number	Name
PS 3.1	Introduction and Overview
PS 3.2	Conformance
PS 3.3	Information Object Definitions
PS 3.4	Service Class Specifications
PS 3.5	Data Structure and Encoding
PS 3.6	Data Dictionary
PS 3.7	Message Exchange
PS 3.8	Network Communication Support for Message Exchange
PS 3.9	Retired
PS 3.10	Media Storage / File Format for Media Interchange
PS 3.11	Media Storage Application Profiles
PS 3.12	Media Formats / Physical Media for Media Interchange
PS 3.13	Retired
PS 3.14	Grayscale Standard Display Function
PS 3.15	Security and System Management Profiles
PS 3.16	Content Mapping Resource
PS 3.17	Explanatory Information
PS 3.18	Web Access to DICOM Persistent Objects (WADO)
PS 3.19	Application Hosting
PS 3.20	Transformation of DICOM to and from HL7 Standards

DICOM data format

DICOM data is stored in memory and on disk as a sequence of *DICOM elements* (section 7 of PS 3.5).

DICOM elements

A DICOM element is made up of three or four fields. These are (Attribute Tag, [Value Representation,], Value Length, Value Field), where *Value Representation* may be present or absent, depending on the type of "Value Representation Encoding" (see below)

Attribute Tag

The attribute tag is a pair of 16-bit unsigned integers of form (Group number, Element number). The tag uniquely identifies the element.

The *Element number* is badly named, because the element number does not give a unique number for the element, but only for the element within the group (given by the *Group number*).

The (Group number, Element number) are nearly always written as hexadecimal numbers in the following format: `(0010, 0010)`. The decimal representation of hexadecimal 0010 is 16, so this tag refers to group number 16, element number 16. If you look this tag up in the DICOM data dictionary (PS 3.6) you'll see this must be the element called "PatientName".

These tag groups have special meanings:

Tag group	Meaning
0000	Command elements
0002	File meta elements
0004	Directory structuring elements
0006	(not used)

See Annex E (command dictionary) of PS 3.7 for details on group 0000. See sections 7 and 8 of PS 3.6 for details of groups 2 and 4 respectively.

Tags in groups 0000, 0002, 0004 are therefore not *data* elements, but Command elements; File meta elements; directory structuring elements.

Tags with groups from 0008 are *data* element tags.

Standard attribute tags

Standard tags are tags with an even group number (see below). There is a full list of all *standard* data element tags in the DICOM data dictionary in section 6 of DICOM standard PS 3.6.

Even numbered groups are defined in the DICOM standard data dictionary. Odd numbered groups are "private", are *not* defined in the standard data dictionary and can be used by manufacturers as they wish (see below).

Quoting from section 7.1 of PS 3.5:

> Two types of Data Elements are defined:
>
> —Standard Data Elements have an even Group Number that is not (0000,eeee), (0002,eeee), (0004,eeee), or (0006,eeee).
>
> Note: Usage of these groups is reserved for DIMSE Commands (see PS 3.7) and DICOM File Formats.
>
> —Private Data Elements have an odd Group Number that is not (0001,eeee), (0003,eeee), (0005,eeee), (0007,eeee), or (FFFF,eeee). Private Data Elements are discussed further in Section 7.8.

Private attribute tags

Private attribute tags are tags with an odd group number. A private element is an element with a private tag.

Private elements still use the (Tag, [Value Representation,] Value Length, Value Field) DICOM data format.

The same odd group may be used by different manufacturers in different ways.

To try and avoid collisions of private tags from different manufacturers, there is a mechanism by which a manufacturer can tell other users of a DICOM dataset that it has reserved a block in the (Group number, Element number) space for their own use. To do this they write a "Private Creator" element where the tag is of the form `(gggg, 00xx)`, the Value Representation (see below) is "LO" (Long String) and the Value Field is a string identifying what the space is reserved for. Here `gggg` is the odd group we are reserving a portion of and the `xx` is the block of elements we are reserving. A tag of `(gggg, 00xx)` reserves the 256 elements in the range `(gggg, xx00)` to `(gggg, xxFF)`.

For example, here is a real data element from a Siemens DICOM dataset:

```
(0019, 0010) Private Creator                    LO: 'SIEMENS MR HEADER'
```

This reserves the tags from `(0019, 1000)` to `(0019, 10FF)` for information on the "SIEMENS MR HEADER"

The odd group `gggg` must be greater than `0008` and the block reservation `xx` must be greater than or equal to `0010` and less than `0100`.

Here is the start of the relevant section from PS 3.5:

> 7.8.1 PRIVATE DATA ELEMENT TAGS
>
> It is possible that multiple implementors may define Private Elements with the same (odd) group number. To avoid conflicts, Private Elements shall be assigned Private Data Element Tags according to the following rules.
>
> a) Private Creator Data Elements numbered (gggg,0010-00FF) (gggg is odd) shall be used to reserve a block of Elements with Group Number gggg for use by an individual implementor. The implementor shall insert an identification code in the first unused (unassigned) Element in this series to reserve a block of Private Elements. The VR of the private identification code shall be LO (Long String) and the VM shall be equal to 1.
>
> b) Private Creator Data Element (gggg,0010), is a Type 1 Data Element that identifies the implementor reserving element (gggg,1000-10FF), Private Creator Data Element (gggg,0011) identifies the implementor reserving elements (gggg,1100-11FF), and so on, until Private Creator Data Element (gggg,00FF) identifies the implementor reserving elements (gggg,FF00- FFFF).
>
> c) Encoders of Private Data Elements shall be able to dynamically assign private data to any available (unreserved) block(s) within the Private group, and specify this assignment through the blocks corresponding Private Creator Data Element(s). Decoders of Private Data shall be able to accept reserved blocks with a given Private Creator identification code at any position within the Private group specified by the blocks corresponding Private Creator Data Element.

Value Representation

Value Representation is often abbreviated to VR.

The VR is a two byte character string giving the code for the encoding of the subsequent data in the Value Field (see below).

The VR appears in DICOM data that has "Explicit Value Representation", and is absent for data with "Implicit Value Representation". "Implicit Value Representation" uses the fact that the DICOM data dictionary gives VR values for each tag in the standard DICOM data dictionary, so the VR value is implied by the tag value, given the data dictionary.

Most DICOM data uses "Explicit Value Representation" because the DICOM data dictionary only gives VRs for standard (even group number, not private) data elements. Each manufacturer writes their own private data elements, and the VR of these elements is not defined in the standard, and therefore may not be known to software not from that manufacturer.

The VR codes have to be one of the values from this table (section 6.2 of DICOM standard PS 3.5):

Value Representation	Description
AE	Application Entity
AS	Age String
AT	Attribute Tag
CS	Code String
DA	Date
DS	Decimal String
DT	Date/Time
FL	Floating Point Single (4 bytes)
FD	Floating Point Double (8 bytes)
IS	Integer String
LO	Long String
LT	Long Text
OB	Other Byte
OF	Other Float
OW	Other Word
PN	Person Name
SH	Short String
SL	Signed Long
SQ	Sequence of Items
SS	Signed Short
ST	Short Text
TM	Time
UI	Unique Identifier
UL	Unsigned Long
UN	Unknown
US	Unsigned Short
UT	Unlimited Text

Value length

Value length gives the length of the data contained in the Value Field tag, or is a flag specifying the Value Field is of undefined length, and thus must be terminated later in the data stream with a special Item or Sequence Delimitation tag.

Quoting from section 7.1.1 of PS 3.5:

> Value Length: Either:
>
> a 16 or 32-bit (dependent on VR and whether VR is explicit or implicit) unsigned integer containing the Explicit Length of the Value Field as the number of bytes (even) that make up the Value. It does not include the length of the Data Element Tag, Value Representation, and Value Length Fields.
>
> a 32-bit Length Field set to Undefined Length (FFFFFFFFH). Undefined Lengths may be used for Data Elements having the Value Representation (VR) Sequence of Items (SQ) and Unknown (UN). For Data Elements with Value Representation OW or OB Undefined Length may be used depending on the negotiated Transfer Syntax (see Section 10 and Annex A).

Value field

An even number of bytes storing the value(s) of the data element. The exact format of this data depends on the Value Representation (see above) and the Value Multiplicity (see next section).

Data element tags and data dictionaries

We can look up data element tags in a *data dictionary*.

As we've seen, data element tags with even group numbers are *standard* data element tags. We can look these up in the standard data dictionary in section 6 of PS 3.6.

Data element tags with odd group numbers are *private* data element tags. These can be used by manufacturers for information that may be specific to the manufacturer. To look up these tags, we need the private data dictionary of the manufacturer.

A data dictionary lists (Attribute tag, Attribute name, Attribute Keyword, Value Representation, Value Multiplicity) for all tags.

For example, here is an excerpt from the table in PS 3.6 section 6:

Tag	Name	Keyword	VR	VM
(0010,0010)	Patient's Name	PatientName	PN	1
(0010,0020)	Patient ID	PatientID	LO	1
(0010,0021)	Issuer of Patient ID	IssuerOfPatientID	LO	1
(0010,0022)	Type of Patient ID	TypeOfPatientID	CS	1
(0010,0024)	Issuer of Patient ID Qualifiers Sequence	IssuerOfPatientIDQualifiersSequence	SQ	1
(0010,0030)	Patient's Birth Date	PatientBirthDate	DA	1
(0010,0032)	Patient's Birth Time	PatientBirthTime	TM	1

The "Name" column gives a standard name for the tag. "Keyword" gives a shorter equivalent to the name without spaces that can be used as a variable or attribute name in code.

Value Representation in the data dictionary

The "VR" column in the data dictionary gives the Value Representation. There is usually only one possible VR for each tag[1].

If a particular stream of data elements is using "Implicit Value Representation Encoding" then the data elements consist of (tag, Value Length, Value Field) and the Value Representation is implicit. In this case we have to get the Value Representation from the data dictionary. If a stream is using "Explicit Value Representation Encoding", the elements consist of (tag, Value Representation, Value Length, Value Field) and the Value Representation is therefore already specified along with the data.

[1] Actually, it is not quite true that there can be only one VR associated with a particular tag. A small number of tags have VRs which can be either Unsigned Short (US) or Signed Short (SS). An even smaller number of tags can be either Other Byte (OB) or Other Word (OW). For all the relevant tags the VM is a set number (1, 3, or 4). So, in the OB / OW cases you can tell which of OB or OW you have by the Value Length. The US / SS cases seem to refer to pixel values; presumably they are US if the Pixel Representation (tag 0028, 0103) is 0 (for unsigned) and SS if the Pixel Representation is 1 (for signed)

Value Multiplicity in the data dictionary

The "VM" column in the dictionary gives the Value Multiplicity for this tag. Quoting from PS 3.5 section 6.4:

> The Value Multiplicity of a Data Element specifies the number of Values that can be encoded in the Value Field of that Data Element. The VM of each Data Element is specified explicitly in PS 3.6. If the number of Values that may be encoded in an element is variable, it shall be represented by two numbers separated by a dash; e.g., "1-10" means that there may be 1 to 10 Values in the element.

The most common values for Value Multiplicity in the standard data dictionary are (in decreasing frequency) '1', '1-n', '3', '2', '1-2', '4' with other values being less common.

The data dictionary is the only way to know the Value Multiplicity of a particular tag. This means that we need the manufacturer's private data dictionary to know the Value Multiplicity of private attribute tags.

DICOM data structures

A data set

A DICOM *data set* is a ordered list of data elements. The order of the list is the order of the tags of the data elements. Here is the definition from section 3.10 of PS 3.5:

> DATA SET: Exchanged information consisting of a structured set of Attribute values directly or indirectly related to Information Objects. The value of each Attribute in a Data Set is expressed as a Data Element. A collection of Data Elements ordered by increasing Data Element Tag number that is an encoding of the values of Attributes of a real world object.

Background - the DICOM world

DICOM has abstract definitions of a set of entities (objects) in the "Real World". These real world objects have relationships between them. Section 7 of PS 3.3 has the title "DICOM model of the real world". Examples of Real World entities are Patient, Study, Series.

Here is a selected list of real world entities compiled from section 7 of PS 3.3:

- Patient
- Visit
- Study
- Modality Performed Procedure Steps
- Frame of Reference
- Equipment
- Series
- Registration
- Fiducials
- Image
- Presentation State
- SR Document
- Waveform

- MR Spectroscopy
- Raw Data
- Encapsulated Document
- Real World Value Mapping
- Stereometric Relationship
- Surface
- Measurements

DICOM refers to its model of the entities and their relationships in the real world as the DICOM Application Model. PS 3.3:

> 3.8.5 DICOM application model: an Entity-Relationship diagram used to model the relationships between Real-World Objects which are within the area of interest of the DICOM Standard.

DICOM Entities and Information Object Definitions

This is rather confusing.

PS 3.3 gives definitions of fundamental DICOM objects called *Information Object Definitions* (IODs). Here is the definition of an IOD from section 3.8.7 of PS 3.3:

> 3.8.7 Information object definition (IOD): a data abstraction of a class of similar Real-World Objects which defines the nature and Attributes relevant to the class of Real-World Objects represented.

IODs give lists of attributes (data elements) that refer to one or more objects in the DICOM Real World.

A single IOD is the usual atom of data sent in a single DICOM message.

An IOD that contains attributes (data elements) for only one object in the DICOM Real World is a *Normalized IOD*. From PS 3.3:

> 3.8.10 Normalized IOD: an Information Object Definition which represents a single entity in the DICOM Application Model. Such an IOD includes Attributes which are only inherent in the Real-World Object that the IOD represents.

Annex B of PS 3.3 defines the normalized IODs.

Many DICOM Real World objects do not have corresponding normalized IODs, presumably because there is no common need to send data only corresponding to - say - a patient - without also sending related information like - say - an image. If you do want to send information relating to a patient with information relating to an image, you need a *composite IOD*.

An IOD that contains attributes from more than one object in the DICOM Real World is a *Composite IOD*. PS 3.3 again:

> 3.8.2 Composite IOD: an Information Object Definition which represents parts of several entities in the DICOM Application Model. Such an IOD includes Attributes which are not inherent in the Real-World Object that the IOD represents but rather are inherent in related Real-World Objects

Annex A of PS 3.3 defines the composite IODs.

DICOM MR or CT image IODs are classic examples of composite IODs, because they contain information not just about the image itself, but also information about the patient, the study, the series, the frame of reference and the equipment.

The term *Information Entity* (IE) refers to a part of a composite IOD that relates to a single DICOM Real World object. PS 3.3:

> 3.8.6 Information entity: that portion of information defined by a Composite IOD which is related to one specific class of Real-World Object. There is a one-to-one correspondence between Information Entities and entities in the DICOM Application Model.

IEs are names of DICOM Real World objects that label parts of a composite IOD. IEs have no intrinsic content, but serve as meaningful labels for a group of *modules* (see below) that refer to the same Real World object.

Annex A 1.2, PS 3.3 lists all the IEs used in composite IODs.

For example, section A.4 in PD 3.3 defines the composite IOD for an MR Image - the Magnetic Resonance Image Object Definition. The definition looks like this (table A.4-1 of PS 3.3)

IE	Module	Reference	Usage
Patient	Patient	C.7.1.1	M
	Clinical Trial Subject	C.7.1.3	U
Study	General Study	C.7.2.1	M
	Patient Study	C.7.2.2	U
	Clinical Trial Study	C.7.2.3	U
Series	General Series	C.7.3.1	M
	Clinical Trial Series	C.7.3.2	U
Frame of Reference	Frame of Reference	C.7.4.1	M
Equipment	General Equipment	C.7.5.1	M
Image	General Image	C.7.6.1	M
	Image Plane	C.7.6.2	M
	Image Pixel	C.7.6.3	M
	Contrast/bolus	C.7.6.4	C - Required if contrast media was used in this image
	Device	C.7.6.12	U
	Specimen	C.7.6.22	U
	MR Image	C.8.3.1	M
	Overlay Plane	C.9.2	U
	VOI LUT	C.11.2	U
	SOP Common	C.12.1	M

As you can see, the MR Image IOD is composite and composed of Patient, Study, Series, Frame of Reference, Equipment and Image IEs.

The *module* heading defines which modules make up the information relevant to the IE.

A module is a named and defined grouping of attributes (data elements) with related meaning. PS 3.3:

> 3.8.8 Module: A set of Attributes within an Information Entity or Normalized IOD which are logically related to each other.

Grouping attributes into modules simplifies the definition of multiple composite IODs. For example, the composite IODs for a CT image and an MR Image both have modules for Patient, Clinical Trial Subject, etc.

Annex C of PS 3.3 defines all the modules used for the IOD definitions. For example, from the table above, we see that the "Patient" module is at section C.7.1.1 of PS 3.3. This section gives a table of all the attributes (data elements) in this module.

The last column in the table above records whether the particular module is Mandatory, Conditional or User Option (defined in section A 1.3 of PS 3.3)

Lastly module definitions may make use of *Attribute macros*. Attribute macros are very much like modules, in that they are a named group of attributes that often occur together in module definitions, or definitions of other macros. From PS 3.3:

> 3.11.1 Attribute Macro: a set of Attributes that are described in a single table that is referenced by multiple Modules or other tables.

For example, here is the Patient Orientation Macro definition table from section 10.12 in PS 3.3:

Attribute Name	Tag	Type	Attribute Description
Patient Orientation Code Sequence	(0054,0410)	2	Sequence that describes the orientation of the patient with respect to gravity. See C.8.11.5.1.2 for further explanation. Only a single Item shall be included in this Sequence.
>Include 'Code Sequence Macro' Table 8.8-1.			Baseline Context ID 19
>Patient Orientation Modifier Code Sequence	(0054,0412)	2C	Patient orientation modifier. Required if needed to fully specify the orientation of the patient with respect to gravity. Only a single Item shall be included in this Sequence.
>>Include 'Code Sequence Macro' Table 8.8-1.			Baseline Context ID 20
Patient Gantry Relationship Code Sequence	(0054,0414)	2	Sequence that describes the orientation of the patient with respect to the head of the table. See Section C.8.4.6.1.3 for further explanation. Only a single Item is permitted in this Sequence.
>Include 'Code Sequence Macro' Table 8.8-1.			Baseline Context ID 21

As you can see, this macro specifies some tags that should appear when this macro is "Included" - and also includes other macros.

DICOM services (DIMSE)

We now go back to messages.

The DICOM application sending the message is called the Service Class User (SCU). We might also call this the client.

The DICOM application receiving the message is called the Service Class Provider (SCP). We might also call this the server - for this particular message.

Quoting from PS 3.7 section 6.3:

> A Message is composed of a Command Set followed by a conditional Data Set (see PS 3.5 for the definition of a Data Set). The Command Set is used to indicate the operations/notifications to be performed on or with the Data Set.

The command set consists of command elements (elements with group number 0000).

Valid sequences of command elements in the command set form valid DICOM Message Service Elements (DIMSEs). Sections 9 and 10 of PS 3.7 define the valid DIMSEs.

For example, there is a DIMSE service called "C-ECHO" that requests confirmation from the responding application that the echo message arrived.

The definition of the DIMSE services specifies, for a particular DIMSE service, whether the DIMSE commend set should be followed by a data set.

In particular, the data set will be a full Information Object Definition's worth of data.

Of most interest to us, the "C-STORE" service command set should always be followed by a data set conforming to an image data IOD.

DICOM service object pairs (SOPs)

As we've seen, some DIMSE services should be followed by particular types of data.

For example, the "C-STORE" DIMSE command set should be followed by an IOD of data to store, but the "C-ECHO" has no data object following.

The association of a particular type of DIMSE (command set) with the associated IOD's-worth of data is a Service Object Pair. The DIMSE is the "Service" and the data IOD is the "Object". Thus the combination of a "C-STORE" DIMSE and an "MR Image" IOD would be a SOP. Services that do not have data following are a particular type of SOP where the Object is null. For example, the "C-ECHO" service is the entire contents of a Verification SOP (PS 3.4, section A.4).

DICOM defines which pairings are possible, by listing them all as Service Object Pair classes (SOP classes).

Usually a SOP class describes the pairing of exactly one DIMSE service with one defined IOD. For example, the "MR Image storage" SOP class pairs the "C-STORE" DIMSE with the "MR Image" IOD.

Sometimes a SOP class describes the pairings of one of several possible DIMSEs with a particular IOP. For example, the "Modality Performed Procedure Step" SOP class describes the pairing of *either* ("N-CREATE", Modality Performed Procedure Step IOD) *or* ("N-SET", Modality Performed Procedure Step IOD) (see PS 3.4 F.7.1). For this reason a SOP class is best described as the pairing of a *DIMSE service group* with an IOD, where the DIMSE service group usually contains just one DIMSE service, but sometimes has more. For example, the "MR Image Storage" SOP class has a DIMSE service group of one element ["C-STORE"]. The "Modality Performed Procedure Step" SOP class has a DIMSE service group with two elements: ["N-CREATE", "N-SET"].

From PS 3.4:

> 6.4 DIMSE SERVICE GROUP
>
> DIMSE Service Group specifies one or more operations/notifications defined in PS 3.7 which are applicable to an IOD.
>
> DIMSE Service Groups are defined in this Part of the DICOM Standard, in the specification of a Service - Object Pair Class.
>
> 6.5 SERVICE-OBJECT PAIR (SOP) CLASS
>
> A Service-Object Pair (SOP) Class is defined by the union of an IOD and a DIMSE Service Group. The SOP Class definition contains the rules and semantics which may restrict the use of the services in the DIMSE Service Group and/or the Attributes of the IOD.

The Annexes of PS 3.4 define the SOP classes.

A pairing of actual data of form (DIMSE group, IOD) that conforms to the SOP class definition, is a SOP class instance. That is, the instance comprises the actual values of the service and data elements being transmitted.

For example, there is a SOP class called "MR Image Storage". This is the association of the "C-STORE" DIMSE command with the "MR Image" IOD. A particular "C-STORE" request command set along with the particular "MR Image" IOD data set would be an *instance* of the MR Image SOP class.

DICOM files

Now let us return to the confusing definition of the DICOM file format from section 7 of PS 3.10:

> 7 DICOM File Format
>
> The DICOM File Format provides a means to encapsulate in a file the Data Set representing a SOP Instance related to a DICOM IOD. As shown in Figure 7-1, the byte stream of the Data Set is placed into the file after the DICOM File Meta Information. Each file contains a single SOP Instance.

The DICOM file Meta Information is:

- File preamble - 128 bytes, content unspecified
- DICOM prefix - 4 bytes "DICM" character string
- 5 meta information elements (group 0002) as defined in table 7.1 of PS 3.10

There follows the IOD dataset part of the SOP instance. In the case of a file storing an MR Image, this dataset will be of IOD type "MR Image"

10.3 Developer documentation page

10.3.1 NiBabel Developer Guidelines

Also see *Developer documentation page*

NiBabel source code

Working with *nibabel* source code

Contents:

Introduction

These pages describe a git and github workflow for the nibabel project.

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

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*.

Install git

Overview

Debian / Ubuntu	`sudo apt-get install git-core`
Fedora	`sudo yum install git-core`
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

Following the latest source

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

The steps are:

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

Get the local copy of the code

From the command line:

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

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

Updating the code

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

```
cd nibabel
git pull
```

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

Making a patch

You've discovered a bug or something else you want to change in nibabel .. — 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/nibabel.git
# make a branch for your patching
cd nibabel
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 master
```

Then, send the generated patch files to the nibabel 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 nibabel repository:

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

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 `master` branch:

```
git format-patch -M -C master
```

   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 nibabel mailing list.

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

```
git checkout master
```

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 nibabel repository on github — *Making your own copy (fork) of nibabel*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# 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/nibabel.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*.

Git for development

Contents:

Making your own copy (fork) of nibabel

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

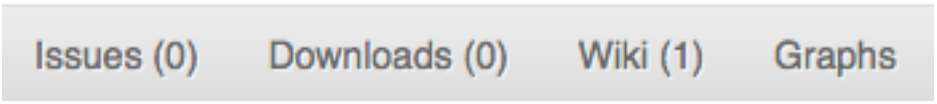
Set up and configure a github account

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 nibabel

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



Now, after a short pause and some 'Hardcore forking action', you should find yourself at the home page for your own forked copy of nibabel.

Set up your fork

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

Overview

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

In detail

Clone your fork

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

```
* master
remotes/origin/master
```

This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. 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 nibabel github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

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

`upstream` here is just the arbitrary name we're using to refer to the main nibabel repository at nibabel 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/nibabel.git (fetch)
upstream     git://github.com/nipy/nibabel.git (push)
origin       git@github.com:your-user-name/nibabel.git (fetch)
origin       git@github.com:your-user-name/nibabel.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"
```

(continues on next page)

(continued from previous page)

```
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 implimentation 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]
```

(continues on next page)

(continued from previous page)

```
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the nibabel repository, by following *Making your own copy (fork) of nibabel*. 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 nibabel `master` branch, as "trunk".

- Don't use your `master` 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 nibabel 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 master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master 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/master` - and if there have been commits since you last checked, `upstream/master` 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/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of nibabel. 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:
#   (use "git add <file>..." to include in what will be committed)
#
```

(continues on next page)

(continued from previous page)

```
#  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/nibabel`.
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 master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

(Note the colon `:` before `test-branch`. See also: https://github.com/guides/remove-a-remote-branch

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

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

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/nibabel.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 master # 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/master upstream/master 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/master
```

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 embarassing 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/nibabel.git
git fetch upstream-rw
```

Integrating changes

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

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/nibabel.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/master
```

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/master
```

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/master..
```

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/master`), 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:master
```

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

git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials
- 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 casts — video snippets giving git how-tos.
- 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

Documentation

Code Documentation

Please write documentation using Numpy documentation conventions:

> https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard

Git Repository

Layout

The main release branch is called `master`. This is a merge-only branch. Features finished or updated by some developer are merged from the corresponding branch into `master`. At a certain point the current state of `master` is tagged – a release is done.

Only usable feature should end-up in `master`. Ideally `master` should be releasable at all times.

Additionally, there are distribution branches. They are prefixed `dist/` and labeled after the packaging target (e.g. *debian* for a Debian package). If necessary, there can be multiple branches for each distribution target.

dist/debian/proper Official Debian packaging

dist/debian/dev Debian packaging of unofficial development snapshots. They do not go into the main Debian archive, but might be distributed through other channels (e.g. NeuroDebian).

Releases are merged into the packaging branches, packaging is updated if necessary and the branch gets tagged when a package version is released. Maintenance (as well as backport) releases or branches off from the respective packaging tag.

There might be additonal branches for each developer, prefixed with intials. Alternatively, several GitHub (or elsewhere) clones might be used.

Commits

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
- *NF* : new feature
- *BW* : addresses backward-compatibility
- *OPT* : optimization
- *BK* : breaks something and/or tests fail
- *PL* : making pylint happier
- *DOC*: for all kinds of documentation related commits
- *TEST*: for adding or changing tests

Merges

For easy tracking of what changes were absorbed during merge, we advise that you enable merge summaries within git:

```
git-config merge.summary true
```

See *Configure git* for more detail.

Changelog

The changelog is located in the toplevel directory of the source tree in the *Changelog* file. The content of this file should be formated as restructured text to make it easy to put it into manual appendix and on the website.

This changelog should neither replicate the VCS commit log nor the distribution packaging changelogs (e.g. debian/changelog). It should be focused on the user perspective and is intended to list rather macroscopic and/or important changes to the module, like feature additions or bugfixes in the algorithms with implications to the performance or validity of results.

It may list references to 3rd party bugtrackers, in case the reported bugs match the criteria listed above.

Community guidelines

Please see our community guidelines. Other projects call these guidelines the "code of conduct".

10.3.2 Core Developer Guide

As a core developer, you should continue making pull requests in accordance with the *NiBabel Developer Guidelines*. You are responsible for shepherding other contributors through the review process. You also have the ability to merge or approve other contributors' pull requests.

Reviewing

How to Conduct A Good Review

Always be kind to contributors. Nearly all of Nibabel is volunteer work, for which we are tremendously grateful. Provide constructive criticism on ideas and implementations, and remind yourself of how it felt when your own work was being evaluated as a novice.

Nibabel strongly values mentorship in code review. New users often need more handholding, having little to no git experience. Repeat yourself liberally, and, if you don't recognize a contributor, point them to our development guide, or other GitHub workflow tutorials around the web. Do not assume that they know how GitHub works (e.g., many don't realize that adding a commit automatically updates a pull request). Gentle, polite, kind encouragement can make the difference between a new core developer and an abandoned pull request.

When reviewing, focus on the following:

1. **API:** The API is what users see when they first use Nibabel. APIs are difficult to change once released, so should be simple, consistent with other parts of the library, and should avoid side-effects such as changing global state or modifying input variables.
2. **Documentation:** Any new feature should have a tutorial example that not only illustrates but explains it.
3. **The algorithm:** You should understand the code being modified or added before approving it. (See *Merge Only Changes You Understand* below.) Implementations should do what they claim, and be simple, readable, and efficient.
4. **Tests:** All contributions to the library *must* be tested, and each added line of code should be covered by at least one test. Good tests not only execute the code, but explore corner cases. It is tempting not to review tests, but please do so.

Other changes may be *nitpicky*: spelling mistakes, formatting, etc. Do not ask contributors to make these changes, and instead make the changes by pushing to their branch, or using GitHub's suggestion feature. (The latter is preferred because it gives the contributor a choice in whether to accept the changes.)

Please add a note to a pull request after you push new changes; GitHub may not send out notifications for these.

Merge Only Changes You Understand

Long-term maintainability is an important concern. Code doesn't merely have to *work*, but should be *understood* by multiple core developers. Changes will have to be made in the future, and the original contributor may have moved on.

Therefore, *do not merge a code change unless you understand it*. Ask for help freely: we have a long history of consulting community members, or even external developers, for added insight where needed, and see this as a great learning opportunity.

While we collectively "own" any patches (and bugs!) that become part of the code base, you are vouching for changes you merge. Please take that responsibility seriously.

Closing issues and pull requests

Sometimes, an issue must be closed that was not fully resolved. This can be for a number of reasons:

- the person behind the original post has not responded to calls for clarification, and none of the core developers have been able to reproduce their issue;
- fixing the issue is difficult, and it is deemed too niche a use case to devote sustained effort or prioritize over other issues; or
- the use case or feature request is something that core developers feel does not belong in Nibabel,

among others. Similarly, pull requests sometimes need to be closed without merging, because:

- the pull request implements a niche feature that we consider not worth the added maintenance burden;
- the pull request implements a useful feature, but requires significant effort to bring up to Nibabel's standards, and the original contributor has moved on, and no other developer can be found to make the necessary changes; or
- the pull request makes changes that make maintenance harder, such as increasing the code complexity of a function significantly to implement a marginal speedup,

among others.

All these may be valid reasons for closing, but we must be wary not to alienate contributors by closing an issue or pull request without an explanation. When closing, your message should:

- explain clearly how the decision was made to close. This is particularly important when the decision was made in a community meeting, which does not have as visible a record as the comments thread on the issue itself;
- thank the contributor(s) for their work; and
- provide a clear path for the contributor or anyone else to appeal the decision.

These points help ensure that all contributors feel welcome and empowered to keep contributing, regardless of the outcome of past contributions.

Further resources

As a core member, you should be familiar with community and developer resources such as:

- Our *NiBabel Developer Guidelines*
- Our *Community guidelines*
- PEP8 for Python style
- PEP257 and the NumPy documentation guide for docstrings. (NumPy docstrings are a superset of PEP257. You should read both.)
- The Nibabel tag on NeuroStars
- Our mailing list

Please do monitor any of the resources above that you find helpful.

10.3.3 Acknowledgments

This document is based on the NetworkX Core Developer guide.

10.3.4 Governance and Decision Making

Abstract

Nibabel is a consensus-based community project. Anyone with an interest in the project can join the community, contribute to the project design, and participate in the decision making process. This document describes how that participation takes place, how to find consensus, and how deadlocks are resolved.

Roles And Responsibilities

The Community

The Nibabel community consists of anyone using or working with the project in any way.

Contributors

Any community member can become a contributor by interacting directly with the project in concrete ways, such as:

- proposing a change to the code or documentation via a GitHub pull request;
- reporting issues on our GitHub issues page;
- discussing the design of the library, website, or tutorials on the mailing list, or in existing issues and pull requests; or
- reviewing open pull requests,

among other possibilities. By contributing to the project, community members can directly help to shape its future.

Contributors should read the *NiBabel Developer Guidelines* and our *Community guidelines*.

Core Developers

Core developers are community members that have demonstrated continued commitment to the project through ongoing contributions. They have shown they can be trusted to maintain Nibabel with care. Becoming a core developer allows contributors to merge approved pull requests, cast votes for and against merging a pull request, and be involved in deciding major changes to the API, and thereby more easily carry on with their project related activities. Core developers appear as team members on the Nibabel Core Team page and can be messaged `@nipy/nibabel-core-developers`. We expect core developers to review code contributions while adhering to the *Core Developer Guide*.

New core developers can be nominated by any existing core developer. Discussion about new core developer nominations is one of the few activities that takes place on the project's private management list. The decision to invite a new core developer must be made by "lazy consensus", meaning unanimous agreement by all responding existing core developers. Invitation must take place at least one week after initial nomination, to allow existing members time to voice any objections.

Steering Council

The Steering Council (SC) members are current or former core developers who have additional responsibilities to ensure the smooth running of the project. SC members are expected to participate in strategic planning, approve changes to the governance model, and make decisions about funding granted to the project itself. (Funding to community members is theirs to pursue and manage.) The purpose of the SC is to ensure smooth progress from the big-picture perspective. Changes that impact the full project require analysis informed by long experience with both the project and the larger ecosystem. When the core developer community (including the SC members) fails to reach such a consensus in a reasonable timeframe, the SC is the entity that resolves the issue.

Steering Council members appear as team members on the Nibabel Steering Council Team page and can be messaged `@nipy/nibabel-steering-council`.

Decision Making Process

Decisions about the future of the project are made through discussion with all members of the community. All non-sensitive project management discussion takes place on the project mailing list and the issue tracker. Occasionally, sensitive discussion may occur on a private list.

Decisions should be made in accordance with our *Community guidelines*.

Nibabel uses a *consensus seeking* process for making decisions. The group tries to find a resolution that has no open objections among core developers. Core developers are expected to distinguish between fundamental objections to a proposal and minor perceived flaws that they can live with, and not hold up the decision making process for the latter. If no option can be found without an objection, the decision is escalated to the SC, which will itself use consensus seeking to come to a resolution. In the unlikely event that there is still a deadlock, the proposal will move forward if it has the support of a simple majority of the SC. Any proposal must be described by a Nibabel *Enhancement Proposals (BIAPs)*.

Decisions (in addition to adding core developers and SC membership as above) are made according to the following rules:

- **Minor documentation changes**, such as typo fixes, or addition / correction of a sentence (but no change of the Nibabel landing page or the "about" page), require approval by a core developer *and* no disagreement or requested changes by a core developer on the issue or pull request page (lazy consensus). We expect core developers to give "reasonable time" to others to give their opinion on the pull request if they're not confident others would agree.

- **Code changes and major documentation changes** require agreement by *one* core developer *and* no disagreement or requested changes by a core developer on the issue or pull-request page (lazy consensus).
- **Changes to the API principles** require a *Enhancement Proposals (BIAPs)* and follow the decision-making process outlined above.
- **Changes to this governance model or our mission and values** require a *Enhancement Proposals (BIAPs)* and follow the decision-making process outlined above, *unless* there is unanimous agreement from core developers on the change.

If an objection is raised on a lazy consensus, the proposer can appeal to the community and core developers and the change can be approved or rejected by escalating to the SC, and if necessary, a BIAP (see below).

Enhancement Proposals (BIAPs)

Any proposals for enhancements of Nibabel should be written as a formal BIAP following the template *BIAP X — Template and Instructions*. The BIAP must be made public and discussed before any vote is taken. The discussion must be summarized by a key advocate of the proposal in the appropriate section of the BIAP. Once this summary is made public and after sufficient time to allow the core team to understand it, they vote.

The workflow of a BIAP is detailed in *BIAP 0 - Purpose and process*.

A list of all existing BIAPs is available *here*.

Acknowledgments

Many thanks to Jarrod Millman, Dan Schult and the Scikit-Image team for the draft on which we based this document.

10.3.5 Roadmap

The roadmap is intended for larger, fundamental changes to the project that are likely to take months or years of developer time. Smaller-scoped items will continue to be tracked on our issue tracker.

The scope of these improvements means that these changes may be controversial, are likely to involve significant discussion among the core development team, and may require the creation of one or more BIAPs (niBabel Increased Awesomeness Proposals).

Background

Nibabel is a workbench that provides a Python API for working with images in many formats. It is also a base library for tools implementing higher level processing.

Nibabel's success depends on:

- How easy it is to express common imaging tasks in the API.
- The range of tasks it can perform.

An expressive, broad API will increase adoption and make it easier to teach.

Expressive API

Axis and tick labels

Brain images typically have three or four axes, whose meanings depend on the way the image was acquired. Axes have natural labels, expressing meaning, such as "time" or "slice", and they may have tick labels such as acquisition time. The scanner captures this information, but typical image formats cannot store it, so it is easy to lose metadata and make analysis errors; see *BIAP6 - Identifying image axes*.

We plan to expand Nibabel's API to encode axis and tick labels by integrating the Xarray package. Xarray simplifies HDF5 serialization, and visualization.

An API for labels is not useful if we cannot read labels from the scanner data, or save them with the image. We plan to:

- Develop HDF5 equivalents of standard image formats, for serialization of data with labels.
- Expand the current standard image format, NIfTI, to store labels in a JSON addition to image metadata: *BIAP3 - A JSON nifti header extension*.
- Read image metadata from DICOM, the standard scanner format.

Reading and attaching DICOM data will start with code integrated from Dcmstack, by Brendan Moloney; see: *BIAP4 - Merging nibabel and dcmstack*.

DICOM metadata is often hidden inside "private" DICOM elements that need specialized parsers. We want to expand these parsers to preserve full metadata and build a normalization layer to abstract vendor-specific storage locations for metadata elements that describe the same thing.

API for surface data

Neuroimaging data often refers to locations on the brain surface. There are three common formats for such data: GIFTI, CIFTI and Freesurfer. Nibabel can read these formats, but lacks a standard API for reading and storing surface data with metadata; see nipy/nibabel#936, nilearn/nilearn#2171. We plan to develop a standard API, apply it to the standard formats, and design an efficient general HDF5 storage container for serializing surface data and metadata.

Range

Spatial transforms

Neuroimaging toolboxes include spatial registration methods to align the objects and features present in two or more images. Registration methods estimate and store spatial transforms. There is no standard or compatible format to store and reuse these transforms, across packages.

Because Nibabel is a workbench, we want to extend its support to read transforms calculated with AFNI, FreeSurfer, FSL, ITK/ANTs, NiftyReg, and SPM.

We have developed the NiTransforms project for this task; we plan to complete and integrate NiTransforms into Nibabel. This will make transforms more accessible to researchers, and therefore easier to work with, and reason about.

10.3.6 BIAPs

niBabel Increased Awesomeness Proposals (BIAPs) document major changes or proposals.

BIAP 0 - Purpose and process

Author Jarrod Millman <millman@berkeley.edu>

Status Draft

Type Process

Created 2020-06-25

What is a BIAP?

BIAP stands for Nibabel Enhancement Proposal. BIAPs are the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Nibabel. A BIAP should provide a concise technical specification of the feature and a rationale for the feature. The BIAP author is responsible for building consensus within the community and documenting dissenting opinions.

Because the BIAPs are maintained as text files in a versioned repository, their revision history is the historical record of the feature proposal[1].

Types

There are three kinds of BIAPs:

1. A **Standards Track** BIAP describes a new feature or implementation for Nibabel.
2. An **Informational** BIAP describes a Nibabel design issue, or provides general guidelines or information to the Python community, but does not propose a new feature. Informational BIAPs do not necessarily represent a Nibabel community consensus or recommendation, so users and implementers are free to ignore Informational BIAPs or follow their advice.
3. A **Process** BIAP describes a process surrounding Nibabel, or proposes a change to (or an event in) a process. Process BIAPs are like Standards Track BIAPs but apply to areas other than the Nibabel language itself. They may propose an implementation, but not to Nibabel's codebase; they require community consensus. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in Nibabel development. Any meta-BIAP is also considered a Process BIAP.

BIAP Workflow

The BIAP process begins with a new idea for Nibabel. It is highly recommended that a single BIAP contain a single key proposal or new idea. Small enhancements or patches often don't need a BIAP and can be injected into the Nibabel development workflow with a pull request to the Nibabel repo. The more focused the BIAP, the more successful it tends to be. If in doubt, split your BIAP into several well-focused ones.

Each BIAP must have a champion—someone who writes the BIAP using the style and format described below, shepherds the discussions in the appropriate forums, and attempts to build community consensus around the idea. The BIAP champion (a.k.a. Author) should first attempt to ascertain whether the idea is suitable for a BIAP. Posting to the Nibabel discussion mailing list is the best way to go about doing this.

[1] This historical record is available by the normal git commands for retrieving older revisions, and can also be browsed on GitHub.

The proposal should be submitted as a draft BIAP via a GitHub pull request to the `doc/source/devel/biaps` directory with the name `biap_<n>.rst` where `<n>` is an appropriately assigned four-digit number (e.g., `biap_0000.rst`). The draft must use the *BIAP X — Template and Instructions* file.

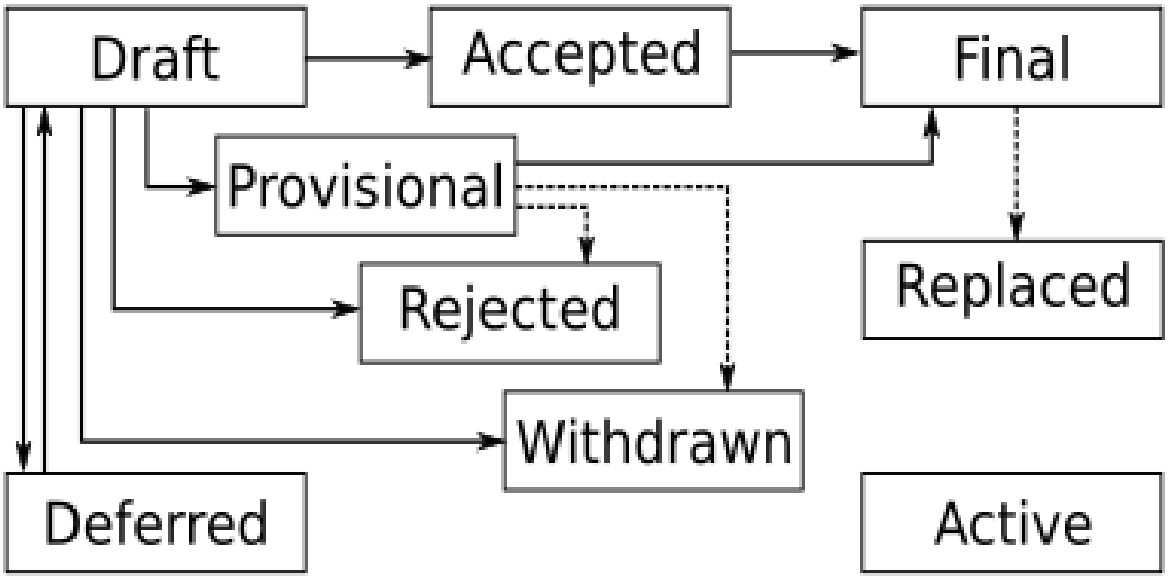
Once the PR for the BIAP is in place, a post should be made to the mailing list containing the sections up to "Backward compatibility", with the purpose of limiting discussion there to usage and impact. Discussion on the pull request will have a broader scope, also including details of implementation.

At the earliest convenience, the PR should be merged (regardless of whether it is accepted during discussion). Additional PRs may be made by the Author to update or expand the BIAP, or by maintainers to set its status, discussion URL, etc.

Standards Track BIAPs consist of two parts, a design document and a reference implementation. It is generally recommended that at least a prototype implementation be co-developed with the BIAP, as ideas that sound good in principle sometimes turn out to be impractical when subjected to the test of implementation. Often it makes sense for the prototype implementation to be made available as PR to the Nibabel repo (making sure to appropriately mark the PR as a WIP).

Review and Resolution

BIAPs are discussed on the mailing list. The possible paths of the status of BIAPs are as follows:



All BIAPs should be created with the `Draft` status.

Eventually, after discussion, there may be a consensus that the BIAP should be accepted – see the next section for details. At this point the status becomes `Accepted`.

Once a BIAP has been `Accepted`, the reference implementation must be completed. When the reference implementation is complete and incorporated into the main source code repository, the status will be changed to `Final`.

To allow gathering of additional design and interface feedback before committing to long term stability for a language feature or standard library API, a BIAP may also be marked as "Provisional". This is short for "Provisionally Accepted", and indicates that the proposal has been accepted for inclusion in the reference implementation, but additional user feedback is needed before the full design can be considered "Final". Unlike regular accepted BIAPs, provisionally accepted BIAPs may still be Rejected or Withdrawn even after the related changes have been included in a Python release.

Wherever possible, it is considered preferable to reduce the scope of a proposal to avoid the need to rely on the "Provisional" status (e.g. by deferring some features to later BIAPs), as this status can lead to version compatibility challenges in the wider Nibabel ecosystem.

A BIAP can also be assigned status `Deferred`. The BIAP author or a core developer can assign the BIAP this status when no progress is being made on the BIAP.

A BIAP can also be `Rejected`. Perhaps after all is said and done it was not a good idea. It is still important to have a record of this fact. The `Withdrawn` status is similar—it means that the BIAP author themselves has decided that the BIAP is actually a bad idea, or has accepted that a competing proposal is a better alternative.

When a BIAP is `Accepted`, `Rejected`, or `Withdrawn`, the BIAP should be updated accordingly. In addition to updating the status field, at the very least the `Resolution` header should be added with a link to the relevant thread in the mailing list archives.

BIAPs can also be `Superseded` by a different BIAP, rendering the original obsolete. The `Replaced-By` and `Replaces` headers should be added to the original and new BIAPs respectively.

Process BIAPs may also have a status of `Active` if they are never meant to be completed, e.g. BIAP 0 (this BIAP).

How a BIAP becomes Accepted

A BIAP is `Accepted` by consensus of all interested contributors. We need a concrete way to tell whether consensus has been reached. When you think a BIAP is ready to accept, send an email to the Nibabel discussion mailing list with a subject like:

> Proposal to accept BIAP #<number>: <title>

In the body of your email, you should:

- link to the latest version of the BIAP,
- briefly describe any major points of contention and how they were resolved,
- include a sentence like: "If there are no substantive objections within 7 days from this email, then the BIAP will be accepted; see BIAP 0 for more details."

After you send the email, you should make sure to link to the email thread from the `Discussion` section of the BIAP, so that people can find it later.

Generally the BIAP author will be the one to send this email, but anyone can do it – the important thing is to make sure that everyone knows when a BIAP is on the verge of acceptance, and give them a final chance to respond. If there's some special reason to extend this final comment period beyond 7 days, then that's fine, just say so in the email. You shouldn't do less than 7 days, because sometimes people are travelling or similar and need some time to respond.

In general, the goal is to make sure that the community has consensus, not provide a rigid policy for people to try to game. When in doubt, err on the side of asking for more feedback and looking for opportunities to compromise.

If the final comment period passes without any substantive objections, then the BIAP can officially be marked `Accepted`. You should send a followup email notifying the list (celebratory emoji optional but encouraged), and then update the BIAP by setting its `:Status:` to `Accepted`, and its `:Resolution:` header to a link to your followup email.

If there *are* substantive objections, then the BIAP remains in `Draft` state, discussion continues as normal, and it can be proposed for acceptance again later once the objections are resolved.

In unusual cases, disagreements about the direction or approach may require escalation to the Nibabel *Steering Council* who then decide whether a controversial BIAP is `Accepted`.

Maintenance

In general, Standards track BIAPs are no longer modified after they have reached the Final state as the code and project documentation are considered the ultimate reference for the implemented feature. However, finalized Standards track BIAPs may be updated as needed.

Process BIAPs may be updated over time to reflect changes to development practices and other details. The precise process followed in these cases will depend on the nature and purpose of the BIAP being updated.

Format and Template

BIAPs are UTF-8 encoded text files using the reStructuredText format. Please see the *BIAP X — Template and Instructions* file and the reStructuredTextPrimer for more information. We use Sphinx to convert BIAPs to HTML for viewing on the web[2].

Header Preamble

Each BIAP must begin with a header preamble. The headers must appear in the following order. Headers marked with * are optional. All other headers are required.

```
  :Author: <list of authors' real names and optionally, email addresses>
  :Status: <Draft | Active | Accepted | Deferred | Rejected |
           Withdrawn | Final | Superseded>
  :Type: <Standards Track | Process>
  :Created: <date created on, in dd-mmm-yyyy format>
* :Requires: <BIAP numbers>
* :Nibabel-Version: <version number>
* :Replaces: <BIAP number>
* :Replaced-By: <BIAP number>
* :Resolution: <url>
```

The Author header lists the names, and optionally the email addresses of all the authors of the BIAP. The format of the Author header value must be

> Random J. User <address@dom.ain>

if the email address is included, and just

> Random J. User

if the address is not given. If there are multiple authors, each should be on a separate line.

References and Footnotes

BIAP1 - Towards immutable images

> **Author** Matthew Brett
>
> **Status** Rejected
>
> **Type** Standards
>
> **Created** 2011-03-23

[2] The URL for viewing BIAPs on the web is https://nipy.org/nibabel/devel/biaps/index.html

Resolution

Retired as of nibabel 2.0 in favor of exposed *dataobj* property. See:

- http://nipy.org/nibabel/nibabel_images.html#the-image-data-array
- http://nipy.org/nibabel/images_and_memory.html

See image *in_memory* attribute and *uncache* method.

We haven't implemented an *is_as_loaded* attribute yet.

Background

Nibabel implicitly has two types of images

- array images
- proxy images

Array images

Array images are the images you get from a typical constructor call:

```
import numpy as np
import nibabel as nib
arr = np.arange(24).reshape((2,3,4))
img = nib.Nifti1Image(arr, np.eye(4))
```

`img` here is an array image, that is to say that, internally, the private `img._data` attribute is reference to `arr` above. `img.get_data()` just returns `img._data`. If you modify `arr`, you will modify the result of `img.get_data()`.

Proxy images

Proxy images are what you get from a call to `load`:

```
px_img = nib.load('test.nii')
```

It's a proxy image in the sense that, internally, `px_arr._data` is a proxy object that does not yet contain an array, but can get an array by the application of:

```
actual_arr = np.asarray(px_img._data)
```

This is in fact what `px_img.get_data()` does. Actually, `px_img.get_data()` also stores the read array in `px_img._data`, so that:

```
px_img = nib.load('test.nii')
assert not isinstance(px_img._data, np.ndarray) # it's a proxy
actual_arr = px_img.get_data()
assert isinstance(px_img._data, np.ndarray) # it's an array now
```

So, at this point, if you change `actual_arr` you'll also be changing `px_img._data` and therefore the result of `px_img.get_data()`.

In other words, `actual_arr = px_img.get_data()` turns the proxy image into an array image.

Issues for design

The code at the moment is a little bit confusing because:

- there isn't an explicit API to check if you have an array image or a proxy image and
- there isn't anywhere in the docs that you can go and see this distinction.

Use cases

Loading images, minimizing memory

I want to load lots of images, or several large images. I'm going to do something with the image data. I want to minimize memory use. This tempts me to do something like this:

```
large_img1 = nib.load('large1.nii')
large_img2 = nib.load('large2.nii')
li1_mean = large_img1.get_data().mean()
li2_mean = large_img2.get_data().mean()
```

The problem with the current design is that, after the `li1_mean =` line, `large_img1` got unproxied, and there's a huge array inside it.

Loading images, maximizing speed

On the other hand, I might want to do the same thing, but each call to unproxy the data (loading off disk, applying scalefactors) will be expensive. So, when I do `li1_mean = large_img1.get_data().mean()` I want any subsequent call to to `large_img1.get_data()` to be much faster. This is the case at the moment, at the expense of the memory hit above.

Loading images, assert not modified

In pipelines in particular, we frequently want to load images, maybe have a look at some parameters, and then pass that image *filename* to some other program such as SPM or FSL. At the moment we've got a problem:

```
img = nib.load('test.nii')
# do stuff
run_spm_thing_on(img) # is 'img' the same as test.nii?
```

The problem is that when the routine `run_spn_thing` receives `img`, it can know that `img` has a filename, `test.nii`, but it can't currently know if `img` is the same object that it was when it was loaded. That is, it can't know whether `test.img` still corresponds to `img` or not. In practice that means that `run_spm_thing` will need to save every `img` to another file before passing that filename to the SPM routine, just in case `img` has been modified. So, we would like a *dirty bit* for the image, something like:

```
# Not implemented yet
if not img.is_as_loaded():
    save(img, 'some_filename.nii')
```

The last line, like it or not, modifies `img` in-place.

Array images, proxy images, copy, view

With thanks to Roberto Viviani for some clarifying thoughts on the nipy mailing list.

At the moment, `img.get_data()` always returns a reference to an array. That is, whenever you call:

```
data = img.get_data()
```

Then, if you modify `data` you will modify the next result of `img.get_data()`.

In particular, the interface currently intends that there should be no functional difference between proxied images and non-proxied images. The proposal below exposes a functional difference between them.

When do you want a copy and when do you want a view?

This is a discussion of this proposal:

```
img.get_data(copy=True|False)
```

compared to:

```
img.get_data(unproxy=True|False)
```

Summary:

- array images - you nearly always want a view
- proxy images - you may want a copy, but you want a copy only because you want to leave the image as a proxy. You might want to leave the image as a proxy because you want to be sure the image corresponds to the file, or save memory.

For array images, it doesn't make sense to return a copy from `img.get_data()`, because it buys you nothing that you would not get from `data = img.get_data().copy()`. This is because you can't save memory (the image already contains the whole array), and it won't help you be sure that the image has not been modified compared to the original array, because there may be references to the array that existed before the image was made, that can be used to modify the data. So, for array images, you always want a reference, or you want to do a manual copy, as above.

For proxied images, it does make sense to get a copy, because a) you want to preserve memory by not unproxying the image, and / or b) you want to be able to be sure that the file associated with the image still corresponds to the data.

For the `img.get_data(copy=False)` proposal, on a proxied image, the `copy=False` call, in order to return a view, must *implicitly* unproxy the image.

Similarly, `img.get_data(unproxy=False)` must *implicitly* copy the image.

It seems to me (MB) that an implicit copy is familiar to a numpy user, but the implicit unproxying may be less obvious.

My (MBs) reasons then for preferring 'unproxy' to 'copy=True' or 'copy=False' or get_data_copy() is that 'unproxy' is closer to how I think the user would think about deciding what they wanted to do.

The `unproxy=False` case covers the situation where you want to preserve memory. It doesn't fully cover the cases where we want to keep track of when the image data has been modified.

Here there are three cases:

- array image, instantiated with an array; the image data can be modified using the array reference passed into the image - we can't know whether the data has been modified without doing hashing or similar.
- proxy image; the array data is still in the file, so we know it corresponds to the file.

- proxy images that have been converted to array images, but have not passed out a reference to the data. Let's call these *shy unproxied* images. For example, with an API like this:

```
img = load('test.nii')
data = img.get_data(copy=True)
```

the `img` is now an array image, but there's no public reference to the internal array object. Someone could get one by cheating with `ref = img._data`, but, we don't need to worry about that - following Python's "mess around if you like but take the consequences" philosophy.

Proposal

An `is_proxy` property:

```
img.is_proxy
```

This is just for clarity.

Allow the user to specify what unproxying they want with a kwarg to `get_data()`:

```
arr = large_img1.get_data(unproxy=False)
```

- for proxied images, `unproxy=False` would leave the underlying array data as a pointer to the file. The returned `arr` would be therefore a copy of the data as loaded from file, and `arr[0] = 99` would have no effect on the data in the image. `unproxy=True` would convert the proxy image into an array image (load the data into memory, return reference). Here `arr[0] = 99` would affect the data in the image
- for array images, `unproxy` would always be ignored.

Thus `unproxy=True` in fact means, `unproxy_if_this_is_a_proxy_do_nothing_otherwise`.

The default would continue to be `unproxy=True` so that the proxied image would continue, by default, to behave the same way as an unproxied image (`get_data` returns a view).

If `img.is_proxy` is True, then we know that the array data has not changed. We then need to be sure that the `header` and `affine` data haven't changed. We might be able to do this with default `copy` kwargs to the `get_header` and `get_affine` methods:

```
hdr = img.get_header(copy=True) # will be default
aff = img.get_affine(copy=True) # will be default
```

We could also do that by caching the original header and affine, but the header in particular can be rather large.

For the next version of nibabel, for backwards compatibility, we'll set `copy=False` to be the default, but warn about the upcoming change. After that we'll set `copy=True` as the default.

Now we can know whether the image has been modified, because if `get_header` and `get_affine` have only been called with `copy=True` and `img.is_proxy == True` - then it must be the same as when loaded.

This leads to an `is_as_loaded` property:

```
if img.is_as_loaded:
    fname = img.get_filename()
else:
    fname = 'tempname.nii'
    save(img, 'tempname.nii')
```

Questions

Should there also be a `set_header` and `set_affine` method?

The header may conflict with the affine. So, would we need something like:

```
img.set_header(hdr, hdr_affine_from='affine')
```

or some other nasty syntax. Or can we avoid this and just do:

```
img2 = nib.Nifti1Image(img.get_data(), new_affine, new_header)
```

?

How about the names in the proposal? `is_proxy`; `unproxy=True`?

BIAP2 - Slicecopy

Author Matthew Brett

Status Rejected

Type Standards

Created 2011-03-26

Status

Alternative implementation as of Nibabel 2.0 with image proxy slicing : see http://nipy.org/nibabel/images_and_memory.html#saving-time-and-memory

Background

Please see https://github.com/nipy/nibabel/issues#issue/9 for motivation.

Sometimes we have a biiig images and we don't want to load the whole array into memory. In this case it is useful to be able to load as a proxy:

```
img = load('my_huge_image.nii')
```

and then take out individual slices, as in something very approximately like:

```
slice0 = img.get_slice(0)
```

Questions

Should `slice0` be a copy or a view?

As from the previous discussion - *BIAP1 - Towards immutable images* - an image may be a proxy or an array.

If the image is an array, the most natural thing to return is a view. That is, modifying `slice0` will modify the underlying array in `img`.

If the image is a proxy, it would be self-defeating to return a view, because that would involve reading the whole image into memory, exactly what we are trying to avoid. So, for a proxied image, we'd nearly always want to return a copy.

What slices should the slicing allow?

The `img.get_slice(0)` syntax needs us to know what slice 0 is. In a nifti image of 3 dimensions, the first is fastest changing on disk. To be useful `0` will probably refer to the slowest changing on disk. Otherwise we'll have to load nearly the whole image anyway. So, for a nifti, 0 should be the first slice in the last dimension.

For Minc on the other hand, you can and I (MB) think always do get C ordered arrays back, so that the slowest changing dimension in the image array is the first. Actually, I don't know how to read a Minc file slice by slice, but the general point is that, to know which slice is worth reading, you need to know the relationship of the image array dimensions to fastest / slowest on disk.

We could always solve this by assuming that we always want to do this for Analyze / Nifti1 files (Fortran ordered). It's a little ugly of course.

Note that taking the slowest changing slice in a nifti image would be the equivalent of taking a slice from the last dimension:

```
arr = img.get_data()
slice0 = arr[...,0]
```

In general, we can get contiguous data off disk for the same data as contiguous data in memory (perhaps obviously). So, all of these are contiguous in the Fortran ordering case:

```
arr[...,0:5]
arr[:,:,0]
arr[:,0:,0]
arr[0:,:,0]
arr[:,1,0]
arr[1,1,1]
```

That is, in general, `:` up until the first specified dimension, then contiguous slices, followed by integer slices. So, all of these can be read directly off disk as slices. Obviously the rules are the reverse for c-ordered arrays.

Option 1: fancy slice object

It's option 1 because it's the first one I thought of:

```
slice0 = img.slicecopy[...,0]
```

Here we solve the copy or view problem with 'always copy'. We solve the 'what slicing to allow' by letting the object decide how to do the slicing. We could obviously just do the full load (deproxy the image) and return a copy of the sliced array, as in:

```
class SomeImage(object):
    class Slicer(object):
        def __init__(self, parent):
            self.parent = parent
        def __getitem__(self, slicedef):
            data = parent._data
            if is_proxy(data) and iscontinguous(slicedef, order='F'):
                return read_off_disk_somehow(slicedef, data)
            data = parent.get_data(unproxy=True)
            return data.__getitem__(slicedef)
    def __init__(self, stuff):
        self.slicecopy = Slicer(self)
```

The problem with this is that:

```
slice0 = img.slicecopy[...,1]
```

might unproxy the image. At the moment, it's rather hidden whether the image is proxied or not on the basis that it's an optimization that should be transparent.

Option 2: not-fancy method call

```
slice0 = img.get_slice(0, copy=True)
```

'slice or view' solved with explicit keyword. 'which slice' solved by assuming you always mean one slice in the last dimension. Or we could also allow:

```
slices = img.get_slices(slice(0,3), copy=True)
```

This is ugly, but fairly clear. This simple 'I mean the last dimension' might be annoying because it assumes the last dimension is the slowest changing, and it does not get to optimize the more complex contiguous cases above. So we could even allow full slicing with stuff like:

```
slice = img.get_slices((slice(None), slice(None), slice(3)), copy=True)
```

Again - this looks a lot more ugly than the `slicecopy` syntax above.

Now, when would you choose `copy=True`? I think, when the image is a proxy. Otherwise you'd want a view. Probably. So what you mean, probably, is something like this:

```
slices = img.get_slices(slicedef, copy_if='is_proxy')
```

But, we've established that for some slices, you're going to have to load the whole image anyway. So in fact probably what you want is to:

1. Take a view if this image is not a proxy
2. Take a copy if we can read this directly off disk
3. Unproxy the image if we have to read the whole thing off disk anyway to get the slices we want, on the basis that we have to read the whole thing into memory anyway, we might as well do that and save ourselves lots of disk thrashing getting the individual slices.

Of course that's what option 1 boils down to. So I think I prefer version 1.

BIAP3 - A JSON nifti header extension

Author Matthew Brett, Bob Dougherty

Status Draft

Type Standards

Created 2011-03-26

The following Wiki documents should be merged with this one:

- NIfTI metadata extension

Abstract

A draft specification of the JSON header for Nibabel.

Background

DICOM files in particular have a lot of information in them that we might want to carry with the image. There are other image file types like Minc or Nrrd that have information we'd like to support but can't with standard nifti.

One obvious place to store this information is in a nifti header extension.

Nifti extension types

From adding nifti extensions:

- 0 = NIFTI_ECODE_IGNORE = unknown private format (not recommended!)
- 2 = NIFTI_ECODE_DICOM = DICOM format (i.e., attribute tags and values): http://medical.nema.org/
- 4 = NIFTI_ECODE_AFNI = AFNI header attributes: The format of the AFNI extension in the NIfTI-1.1 format is described at http://nifti.nimh.nih.gov/nifti-1/AFNIextension1/
- 6 = NIFTI_ECODE_COMMENT = comment: arbitrary non-NUL ASCII text, with no additional structure implied
- 8 = NIFTI_ECODE_XCEDE = XCEDE metadata: http://www.nbirn.net/Resources/Users/Applications/xcede/index.htm
- 10 = NIFTI_ECODE_JIMDIMINFO = Dimensional information for the JIM software (XML format); contact info is Dr Mark A Horsfield: mah5*AT*leicester.ac.uk.
- 12 = NIFTI_ECODE_WORKFLOW_FWDS = Fiswidget XML pipeline descriptions; documented at http://kraepelin.wpic.pitt.edu/~fissell/NIFTI_ECODE_WORKFLOW_FWDS/NIFTI_ECODE_WORKFLOW_FWDS.html ; contact info is Kate Fissell: fissell+*AT*pitt.edu.

Alternatives

Summary: we need probably need our own extension format

There is a DICOM type extension - code 2. This might be OK for DICOM but:

1. We probably don't want to have to dump the entire DICOM header for every DICOM image. If we don't that means we have to edit the DICOM header, and
2. The DICOM format is awful to work with, so it is not a pleasant prospect making a new DICOM header for images (like Minc) that aren't DICOM to start with.
3. I (MB) can't find any evidence that it's being used in the wild.
4. It's not completely clear what format the data should be in. See this nifti thread.

The AFNI extension format looks as if it is specific to AFNI.

The XCEDE format looks rather heavy. I'm (MB) trying to work out where the most current schema is. Candidates are bxh-xcede-tools and the xcede website. We'd need to validate the XML with the schema. It appears the python standard library doesn't support that so we'd need extra XML tools as a dependency.

JIM is closed source.

fiswidgets seems to have been quiet recently. The link for code 12 is dead, I had to go back to the http://www.archive.org to get an old copy and that didn't have the DTD or example links that we need to understand the format.

Learning from NRRDs

Gordon Kindlmann's NRRD format has gone through a few versions and has considerable use particularly by the 3D slicer team. I've tried to summarize the NRRD innovations not properly covered by nifti in [[nifti-nrrd]].

Proposal

JSON, as y'all know, encodes strings, numbers, objects and arrays, An object is like a Python dict, with strings as keys, and an array is like a Python list.

In what follows, I will build dicts and lists corresponding to the objects and arrays of the JSON header. In each case, the `json.dumps` of the given Python object gives the corresponding JSON string.

I'll use the term *field* to refer to a (key, value) pair from a Python dict / JSON object.

General principles

We specify image axes by name in the header, and give the correspondence of the names to the image array axes by the order of the names. This is the `axis_names` field at the top level of the header.

If the user transposes or otherwise reorders the axes of the data array, the header should change only in the ordering of the axis names in `axis_names`. Call this the "axis transpose" principle.

The JSON header should make sense as a key, value pair store for DICOM fields using a standard way of selecting DICOM fields – the *simple DICOM* principle.

The NIfTI image also contains the standard image metadata in the NIfTI header C-struct (the standard NIfTI header). Nibabel and Nipy will write JSON headers correctly, and so the information in the NIfTI C-struct should always match the information in the JSON header. Other software may write the JSON incorrectly, or copy the JSON header into another image to which it may not apply, but other software should always set the C-struct correctly. For that reason the C-struct always overrides the JSON header, unless the C-struct has values implying "not-set" or "don't know". This is the *C-struct primacy* principle.

See also

- JSON-LD - provides a way of using json that can be mapped into the Resource Description Framework (RDF). It is highly recommended to take a look at the RDF Primer to get a sense of why we might want to use JSON-LD/RDF, but essentially it boils down to a couple points:
  - JSON keys are turned into URIs
  - URIs can dereference to a Web URL with additional documentation, such as a definition, a pretty label (e.g., `nipy_header_version` has_label `"NIPY Header Version"`), etc.
  - The URI link to documentation makes the meaning of your JSON keys explicit, in a machine readable way (i.e., the json key becomes a "resource" on the Web that avoids name clashes)
  - JSON-LD/RDF has a full query language called SPARQL and a python library called RDFLib that acts as a parser, serializer, database, and query engine.

– In the example below, the `@context` section provides the namespace prefix `dcm` as a placeholder for the URL `http://neurolex.org/wiki/Category:`, thus `dcm:Echo_Time` dereferences to http://neurolex.org/wiki/Category:Echo_Time where additional documentation is provided:

```
{
  "@context": {
    "dcm": "http://neurolex.org/wiki/Category:#"
  },
  "dcm:Echo_Time": 45,
  "dcm:Repetition_Time": 2,
}
```

The header must contain the header version

```
>>> hdr = dict(nipy_header_version='1.0')
```

We chose the name "nipy_header_version" in the hope that this would not often occur in an unrelated JSON file.

- First version will be "1.0".
- Versioning will use Semantic Versioning of form `major.minor[.patch[-extra]]` where `major`, `minor`, `patch` are all integers, `extra` may be a string, and both `patch` and `extra` are optional. Header versions with the same `major` value are forwards compatible – that is, a reader that can read a header with a particular major version should be able to read any header with that major version. Specifically, any changes to the header format within major version number should allow older readers of that major version to read the header correctly, but can expand on the information in the header, so that older readers can safely ignore new information in the header.
- All fields other than `nipy_header_version` are optional. The dict in `hdr` above is therefore the minimal valid header.

The header will usually contain image metadata fields

The base level header will usually also have image metadata fields giving information about the whole image. A field is an "image metadata field" if it is defined at the top level of the header. For example:

```
>>> hdr = dict(nipy_header_version='1.0',
```

... Manufacturer="SIEMENS")

All image metadata fields are optional.

As for all keys in this standard, IM (Image Metadata) keys are case sensitive. IM keys that begin with a capital letter must be from the DICOM data dictionary standard short names (DICOM keyword). Call these "DICOM IM keys". This is to conform to the *simple DICOM* principle.

Keys beginning with "extended" will be read and written, but not further processed by a header reader / writer. If you want to put extra fields into the header that are outside this standard you could use a dict / object of form:

```
>>> hdr = dict(nipy_header_version='1.0',
...            extended=dict(my_field1=0.1, my_field2='a string'))
```

or:

```
>>> hdr = dict(nipy_header_version='1.0',
...            extended_mysoft=dict(mysoft_one='expensive', mysoft_two=1000))
```

Values for DICOM IM keys are constrained by the DICOM standard. This standard constrains values for ("nipy_header_version", "axis_names", "axis_metadata"). Other values have no constraint.

Questions

- Should all DICOM values be allowed?
- Should DICOM values be allowed at this level that in fact refer to a particular axis, and therefore might go in the `axis_metadata` elements?
- How should we relate the DICOM standard values to JSON? For example, how should we store dates and times? One option would be to use the new DICOM JSON encoding for DICOM values, but omitting the tag and value representation (VR). For example, the DICOM JSON spec has:

```
"00080070": {
    "vr": "LO",
    "Value": [ "SIEMENS" ]
},
```

but we might prefer:

```
"Manufacturer": "SIEMENS"
```

Using the DICOM data dictionary we can reconstruct the necessary tag and VR, so our version is lossless if the DICOM keyword exists in the DICOM data dictionary. Of course this may well not be true for private tags, or if the keyword comes from a DICOM dictionary that is later than the one we are using to look up the keyword. For the latter, we could make sure we're always using the latest dictionary. For the private tags, we might want to recode these in any case, maybe using our own dictionary. Maybe it is unlikely we will want to reconstruct the private tags of a DICOM file from the JSON. Comments welcome.

The header will usually contain axis names

`axis_names` is a list of strings corresponding to the axes of the image data to which the header refers.

```
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names=["frequency", "phase", "slice", "time"])
```

The names must be valid Python identifiers (should not begin with a digit, nor contain spaces etc).

There must be the same number of names as axes in the image to which the header refers. For example, the header above is valid for a 4D image but invalid for a 3D or 5D image.

The names appear in fastest-slowest order in which the image data is stored on disk. The first name in `axis_names` corresponds to the axis over which the data on disk varies fastest, and the last corresponds to the axis over which the data varies slowest.

For a NIfTI image, nibabel (and nipy) will create an image where the axes have this same fastest to slowest ordering in memory. For example, let's say the read image is called `img`. `img` has shape (4, 5, 6, 10), and a 2-byte datatype such as int16. In the case of the NIfTI default fastest-slowest ordered array, the distance in memory between `img[0, 0, 0, 0]` and `img[1, 0, 0, 0]` is 2 bytes, and the distance between `img[0, 0, 0, 0]` and `img[0, 0, 0, 1]` is 4 * 5 * 6 * 2 = 240 bytes. The names in `axis_names` will then refer to the first, second, third and fourth axes respectively. In the example above, "frequency" is the first axis and "time" is the last.

`axis_names` is optional only if `axis_metadata` is empty or absent. Otherwise, the `set()` of `axis_names` must be a superset of the union of all axis names specified in the `applies_to` fields of `axis_metadata` elements.

The header will often contain axis metadata

`axis_metadata` is a list of *axis metadata elements*.

Each *axis metadata element* in the `axis_metadata` list gives data that applies to a particular axis, or combination of axes. `axis_metadata` can be empty:

```
>>> hdr['axis_metadata'] = []
```

We prefer you delete this section if it is empty, to avoid clutter, but hey, mi casa, su casa.

The axis metadata element

An axis metadata element must contain a field `applies_to`, with a value that is a list that contains one or more values from `axis_names`. From the above example, the following would be valid axis metadata elements:

```
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["frequency", "phase", "slice", "time"],
...            axis_metadata = [
...                dict(applies_to = ['time']),
...                dict(applies_to = ['slice']),
...                dict(applies_to = ['slice', 'time']),
...            ])
```

Note: The `applies_to` field plays the role of a dictionary key for each axis metadata element, where the rest of the fields in the element are a dict giving the value. For example, in Python (but not in JSON, we could represent the above as:

```
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["frequency", "phase", "slice", "time"],
...            axis_metadata = {
...                'time': {},
...                'slice': {},
...                ('slice', 'time'): {},
...            ])
```

We can't do this in JSON because all object fields must be strings, so we cannot represent the key `('slice', 'time')` directly. The `applies_to` field allows us to do that in JSON. See below for why we might want to specify more than one axis.

As for image metadata keys, keys that begin with a capital letter are DICOM standard keywords.

A single axis name for `applies_to` specifies that any axis metadata values in the element apply to the named axis.

In this case, axis metadata values may be:

- a scalar. The value applies to every point along the corresponding image axis OR
- a vector of length N (where N is the length of the corresponding image axis). Value v_i in the vector v corresponds to the image slice at point i on the corresponding axis OR
- an array of shape (1, ...) where "..." can be any further shape, expressing a vector or array that applies to all points on the given axis, OR
- an array of shape (N, ...) where "..." can be any further shape. The (N, ...) array N vectors or arrays with one (vector or array) corresponding to each point in the image axis.

More than one axis name for `applies_to` specifies that any values in the element apply to the combination of the given axes.

In the case of more than one axis for `applies_to`, the axis metadata values apply to the Cartesian product of the image axis values. For example, if the values of `applies_to` == `['slice', 'time']`, and the slice and time axes in the array are lengths (6, 10) respectively, then the values apply to all combinations of the 6 possible values for slice indices and the 10 possible values for the time indices (ie apply to all 6x10=60 values). The axis metadata values in this case can be:

- a scalar. The value applies to every combination of (slice, time)
- an array of shape (S, T) (where S is the length of the slice axis and T is the length of the time axis). Value $a_{i,j}$ in the array a corresponds to the image slice at point i on the slice axis and j on the time axis.
- an array of shape (S, T, ...) where "..." can be any further shape. The (S, T, ...) case gives N vectors or arrays with one vector / array corresponding to each combination of slice, time points in the image,

In contrast to the single axis case, we do not allow length 1 axes, to indicate a value constant across an axis. For example, we do not allow shape (1, T) arrays to indicate a value constant across slice but varying across time, as this should be specified with the single time axis metadata element.

In general, for a given value `applies_to`, we can take the corresponding axis lengths:

```
>>> shape_of_image = [4, 5, 6, 10]
>>> image_names = ['frequency', 'phase', 'slice', 'time']
>>> applies_to = ['slice', 'time']
>>> axis_indices = [image_names.index(name) for name in applies_to]
>>> axis_lengths = [shape_of_image[i] for i in axis_indices]
>>> axis_lengths
[6, 10]
```

The axis metadata value can therefore be of shape:

- () (a scalar) (a scalar value for every combination of points);
- `axis_lengths` (a scalar value for each combination of points);
- [1] + `any_other_list` if `len(axis_lengths) == 1`;
- `axis_lengths + any_other_list` (an array or vector corresponding to each combination of points, where the shape of the array or vector is given by `any_other_list`)

For any unique ordered combination of axis names, there can only be on axis metadata element. For example, this is valid:

```
>>> # VALID
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["frequency", "phase", "slice", "time"],
...            axis_metadata = [
...                dict(applies_to = ['time']),
...                dict(applies_to = ['slice', 'time']),
...                dict(applies_to = ['slice']),
...            ])
```

This is not, because of the repeated combination of axis names:

```
>>> # NOT VALID because of repeated axis combination
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["frequency", "phase", "slice", "time"],
...            axis_metadata = [
...                dict(applies_to = ['time']),
```

(continues on next page)

(continued from previous page)

```
...                 dict(applies_to = ['slice', 'time']),
...                 dict(applies_to = ['slice']),
...                 dict(applies_to = ['slice', 'time']),
...             ])
```

The `q_vector` axis metadata field

We define an axis metadata field `q_vector` which gives the q vector corresponding to the diffusion gradients applied.

The `q_vector` should apply to (`applies_to`) one axis, where that axis is the image volume axis. The `q_vector` is a dict / object with two fields, `spatial_axes` and `array`.

If there are T volumes then the array will be of shape (T, 3). One row from this array corresponds to the direction of the diffusion gradient with axes oriented to the three spatial axes of the data. To preserve the *axis transpose* principle, the `spatial_axes` field value is a list of the spatial image axes to which the first, second and third column of the `array` refer.

For example:

```
>>> import numpy as np
>>> element = dict(applies_to=['time'],
...                q_vector = dict(
...                   spatial_axes = ['frequency', 'phase', 'slice'],
...                   array = [[0, 0, 0],
...                            [1000, 0, 0],
...                            [0, 1000, 0],
...                            [0, 0, 1000],
...                            [0, 0, 0],
...                            [1000, 0, 0],
...                            [0, 1000, 0],
...                            [0, 0, 1000],
...                            [0, 0, 0],
...                            [1000, 0, 0]
...                           ]))
>>> np.array(element['q_vector']['array']).shape
(10, 3)
```

An individual (3,) vector is the unit vector expressing the direction of the gradient, multiplied by the scalar b value of the gradient. In the example, there are three b == 0 scans (corresponding to volumes 0, 4, 8), with the rest having b value of 1000.

The first value corresponds to the direction along the first named image axis ('frequency'), the second value to direction along the second named axis ('phase'), and the third to direction along the 'slice' axis.

Note that the `q_vector` is always specified in the axes of the image. This is the same convention as FSL uses for its `bvals` and `bvecs` files.

`acquisition_times` field

This gives a list of times of acquisition of each spatial unit of data.

`acquisition_times` can apply to (`applies_to`) slices or to volumes or to both.

Units are milliseconds and can be expressed as integers or as floating point. Milliseconds is a reasonable choice for units because a Python integer can decode / encode any integer number in the JSON correctly, a signed 32-bit int can encode to around 6000 hours, and a 32-bit float can encode to 23 hours without loss of precision.

`acquisition_times` applying to slices

If `acquisition_times` applies to an image axis representing slices, then the array should be of shape (S,) where S is the number of slices. Each value a_i represents the time of acquisition of slice i, relative to the start of the volume, in milliseconds. For example, to specify an ascending sequential slice acquisition scheme:

```
>>> element = dict(applies_to=['slice'],
...                acquisition_times=[0, 20, 40, 60, 80, 100])
```

We use "slice" as the axis name here, but any name is valid.

NIfTI 1 and 2 can encode some slice acquisition times using a somewhat complicated scheme, but they cannot - for example - encode multi-slice acquisitions, and NIfTI slice time encoding is rarely set. According to the *C-struct primacy* principle, if the slice timing is set, it overrides this `acquisition_times` field. Slice timing is set in the C-struct if the slice_code in the C-struct is other than 0 (=unknown). The specific slice times from the C-struct also depend on C-struct fields `slice_start` and `slice_end`.

`acquisition_times` applying to volumes

When *acquisition_times* ` applies to a volume axis, it is a list of times of acquisition of each volume in milliseconds relative to the beginning of the acquisition of the run.

These values can be useful for recording runs with missing or otherwise not-continuous time data.

We use "time" as the axis name, but any name is valid.

```
>>> element = dict(applies_to=['time'],
...                acquisition_times=[0, 120, 240, 480, 600])
```

The NIfTI C-struct can encode a non-zero start point for volumes, using the toffset field. If this is not-zero, and not equal to the first value in `acquisition_times`, JSON acquisition times applying to volumes are ignored. The C-struct `slice_code` field (see above) is not relevant to volume times, and can have any value.

`acquisition_times` applying to slices and volumes

When *acquisition_times* ` applies to both a slice and a volume axis, it is a list of times of acquisition of each slice in each volume in milliseconds relative to the beginning of the acquisition of the run.

```
>>> element = dict(applies_to=['slice', 'time'],
...                acquisition_times = [[0, 100, 200],
...                                     [10, 110, 210],
...                                     [20, 120, 220],
...                                     [30, 130, 230],
```

(continues on next page)

(continued from previous page)

```
...                                         [40, 140, 240]]
...             )
```

This meaning becomes invalid with non-zero and conflicting values for `slice_code` or `toffset` in the C-struct. Conflicting values are values different from those implied from a strict per-volume repetition of the acquisition times from `slice_code, slice_start, slice_end`, starting at `toffset`.

`axis_meanings` field

So far we are allowing any axis to be a slice or volume axis, but it might be nice to check. One way of doing this is:

```
>>> element = dict(applies_to=['mytime'],
...                axis_meanings=["volume", "time"],
...                acquisition_times=[0, 120, 240, 480, 600])
>>> element = dict(applies_to=['myslice'],
...                axis_meanings=["slice"],
...                acquisition_times=[0, 20, 40, 60, 80, 100])
```

In this case we can assert that `acquisition_times` applies to an axis with meanings that include "slice" or that it applies to an axis with meaning "volume". For example:

```
>>> # Should raise an error on reading full JSON
>>> element = dict(applies_to=['myslice'],
...                axis_meanings=["frequency"],
...                acquisition_times=[0, 20, 40, 60, 80, 100])
```

Being able to specify meanings that apply to more than one axis might also help for the situation where there is more than one frequency axis:

```
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["frequency1", "frequency2", "slice", "time"],
...            axis_metadata = [
...                dict(applies_to = ["frequency1"],
...                     axis_meanings = ["frequency"]),
...                dict(applies_to = ["frequency2"],
...                     axis_meanings = ["frequency"]),
...                dict(applies_to = ['slice'],
...                     axis_meanings = ["slice"]),
...                dict(applies_to = ['time'],
...                     axis_meanings = ["time", "volume"]),
...            ])
```

We can also check that space axes really are space axes:

```
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["frequency", "phase", "slice", "time"],
...            axis_metadata = [
...                dict(applies_to = ["frequency"],
...                     axis_meanings = ["frequency", "space"]),
...                dict(applies_to = ["phase"],
...                     axis_meanings = ["phase", "space"]),
...                dict(applies_to = ["slice"],
...                     axis_meanings = ["slice", "space"]),
...                dict(applies_to = ["time"],
...                     axis_meanings = ["time", "volume"]),
```

(continues on next page)

(continued from previous page)

```
...                 dict(applies_to=["time"],
...                      q_vector = dict(
...                         spatial_axes = ["frequency", "phase", "slice"],
...                         array = [[0, 0, 0],
...                                  [1000, 0, 0]]))
...                 ])
```

For the `q_vector` field, we can check that all of the `spatial_axes` axes ("frequency", "phase", "slice") do in fact have meaning "space".

For this check to pass, either of these must be true:

- no axes are labeled with the meaning "space" OR
- the only three axes with label "space" are those named in `spatial_axes`.

`multi_affine` field

Use case

When doing motion correction on a 4D image, we calculate the required affine transformation from |--| say |--| the second image to the first image; the third image to the first image; etc. If there are N volumes in the 4D image, we would need to store N-1 affine transformations. If we have registered to the mean volume of the volume series instead of one of the volumes in the volume series, then we need to store all N transforms.

We often want to store this set of required transformations with the image, but NIfTI does not allow us to do that. SPM therefore stores these transforms in a separate MATLAB-format `.mat` file. We currently don't read these transformations because we have no API in nibabel to present or store multiple affines.

Implementation

Assume the 4D volume has T time points (volumes).

There are two ways we could implement the multi-affines. The first would be to have (T x 3 x 4) `array` of affines, with one for each volume / time point, and a `spatial_axes` field specifying the input axes for the affine. This is the same general idea as the *q_vector* field:

```
>>> element = dict(applies_to=['time'],
...                 multi_affine = dict(
...                     spatial_axes = ['frequency', 'phase', 'slice'],
...                     array = [[[   2.86,   -0.7 ,    0.83,  -80.01],
...                               [   0.71,    2.91,    0.01, -114.59],
...                               [  -0.54,    0.13,    4.42,  -54.34]],
...                              [[   2.87,   -0.38,    1.19,  -92.77],
...                               [   0.31,    2.97,    0.45, -110.87],
...                               [  -0.82,   -0.2 ,    4.32,  -33.89]],
...                              [[   2.97,   -0.39,    0.31,  -78.95],
...                               [   0.33,    2.9 ,    1.06, -116.99],
...                               [  -0.29,   -0.68,    4.36,  -36.41]],
...                              [[   2.93,   -0.5 ,    0.61,  -78.02],
...                               [   0.4 ,    2.9 ,    0.99, -118.9 ],
...                               [  -0.5 ,   -0.59,    4.35,  -33.61]],
...                              [[   2.95,   -0.44,    0.49,  -77.86],
...                               [   0.3 ,    2.78,    1.62, -125.83],
```

(continues on next page)

(continued from previous page)

```
...                                 [  -0.46,   -1.03,    4.17,  -21.66]]]))
>>> np.array(element['multi_affine']['array']).shape
(5, 3, 4)
```

This obeys the axis transpose principle, because the spatial axes are specified. If the user transposes the image, the order of axis names in `axis_names` changes, but the correspondence between axis names and affine columns is still correctly encoded in the `spatial_axes`.

Another option would be to partially follow the NRRD format in giving the column vectors from the affine to the axis to which they apply, and split the translation into a separate offset vector:

```
>>> hdr = dict(nipy_header_version='1.0',
...            axis_names = ["time"],
...            axis_metadata = [
...                dict(applies_to=['time'],
...                     output_vector=dict(
...                        spatial_axis = ['frequency'],
...                        array = [
...                                 [ 2.86, 0.71, -0.54],
...                                 [ 2.87, 0.31, -0.82],
...                                 [ 2.97, 0.33, -0.29],
...                                 [ 2.93, 0.4 , -0.5 ],
...                                 [ 2.95, 0.3 , -0.46],
...                                 ])),
...                dict(applies_to=['time'],
...                     output_vector=dict(
...                        spatial_axis = ['phase'],
...                        array = [
...                                 [ -0.7 , 2.91,  0.13],
...                                 [ -0.38, 2.97, -0.2 ],
...                                 [ -0.39, 2.9 , -0.68],
...                                 [ -0.5 , 2.9 , -0.59],
...                                 [ -0.44, 2.78, -1.03],
...                                 ])),
...                dict(applies_to=['time'],
...                     output_vector = dict(
...                        spatial_axis = ['slice'],
...                        array = [
...                                 [ 0.83, 0.01, 4.42],
...                                 [ 1.19, 0.45, 4.32],
...                                 [ 0.31, 1.06, 4.36],
...                                 [ 0.61, 0.99, 4.35],
...                                 [ 0.49, 1.62, 4.17],
...                                 ])),
...                dict(applies_to=['time'],
...                     output_offset = [
...                               [ -80.01, -114.59, -54.34],
...                               [ -92.77, -110.87, -33.89],
...                               [ -78.95, -116.99, -36.41],
...                               [ -78.02, -118.9,  -33.61],
...                               [ -77.86, -125.83, -21.66],
...                               ])],
...            )
>>> np.array(hdr['axis_metadata'][0]['output_vector']['array']).shape
(5, 3)
>>> np.array(hdr['axis_metadata'][1]['output_vector']['array']).shape
(5, 3)
```

(continues on next page)

(continued from previous page)

```
>>> np.array(hdr['axis_metadata'][2]['output_vector']['array']).shape
(5, 3)
>>> np.array(hdr['axis_metadata'][3]['output_offset']).shape
(5, 3)
```

BIAP4 - Merging nibabel and dcmstack

Author Brendan Moloney, Matthew Brett

Status Draft

Type Standards

Created 2012-11-21

In which we set out what dcmstack does and how it might integrate with the nibabel objects and functions.

Motivation

It is very common to convert source DICOM images to another format, typically Nifti, before doing any image processing. The Nifti format is significantly easier to work with and has wide spread compatibility. However, the vast amount of meta data stored in the source DICOM files will be lost.

After implementing this proposal, users will be able to preserve all of the meta data from the DICOM files during conversion, including meta data from private elements. The meta data will then be easily accessible through the *SpatialImage* API:

```
>>> nii = nb.load('input.nii')
>>> data = nii.get_data()
>>> print data.shape
(256, 256, 24, 8)
>>> print nii.get_meta('RepetitionTime')
3500.0
>>> echo_times = [nii.get_meta('EchoTime', (0, 0, 0, idx))
                  for idx in xrange(data.shape[-1])]
>>> print echo_times
[16.4, 32.8, 49.2, 65.6, 82.0, 98.4, 114.8, 131.2]
>>> print nii.get_meta('AcquisitionTime', (0, 0, 1, 0))
110455.370000
>>> print nii.get_meta('AcquisitionTime', (0, 0, 2, 0))
110457.272500
>>> print nii.get_meta('AcquisitionTime', (0, 0, 1, 1))
110455.387500
```

Overview

dcmstack reads a series of DICOM images, works out their relationship in terms of slices and volumes, and compiles them into multidimensional volumes. It can produce the corresponding data volume and affine, or a Nifti image (with any additional header information set appropriately).

In the course of the read, dcmstack creates a *DcmMeta* object for each input file. This object is an ordered mapping that can contain a copy of all the meta data in the DICOM header. By default some filtering is applied to reduce the chance of including PHI. The set of DcmMeta objects are then merged together in the same order as the image data to create a single DcmMeta object that summarizes all of the meta data for the series.

To summarize the meta data, each element is classified based on how the values repeat (e.g. const, per_slice, per_volume, etc.). Each element has a name (the keyword from the DICOM standard) and one or more values (the number of values depends on the classification and the shape of the image). Each classification's meta data is stored stored in a separate nested dictionary.

While creating the Nifti image output, the *DcmMeta* is stored in a *DcmMetaExtension* which can be added as a header extension. This extension simply does a JSON encoding directly on the *DcmMeta* object.

When working with these images, it's possible to keep track of the meta-information in the *DcmMetaExtension*. For example, when taking slice out of a 3D volume, we keep track of the information specific to the chosen slice, and remove information for other slices. Or when merging 3D volumes to a 4D time series, we want to merge together the meta data too.

At the moment, dcmstack only creates Nifti images. There's no reason that this should be so, and the relationship of dcmstack to other spatial images should be more flexible.

Issues

DcmMetaExtension tied to *NiftiExtension*

At the moment, *DcmMetaExtension* inherits from the *NiftiExtension*, allowing the data to be dumped out to JSON when writing into the extension part of a Nifti header.

There's no reason that the *DcmMetaExtension* should be tied to the Nifti format.

Plan

Refactor *DcmMetaExtension* to inherit from *object*. Maybe rename *DcmMeta* or something. Make a *NiftiExtension* object when needed with a new object wrapping the *DcmMeta* in the Extension API?

Status

Resolved. We now have a seperate *DcmMeta* object which inherits from *OrderedDict* and contains all of the functionality previously in *DcmMetaExtension* except those related to acting as a Nifti1Extension. The *DcmMetaExtension* now provides just the functionality for being a Nifti1Extension.

Keeping track of metadata when manipulating images

When slicing images, it is good to be able to keep track of the relevant DICOM metadata for the particular slice. Or when merging images, it is good to be able to compile the metadata across slices into the (e.g) volume metadata. Or, say, when coregistering an image, it is good to be able to know that the metadata that is per-slice no longer directly corresponds to a slice of the data array.

At the moment, dcmstack deals with this by wrapping the image with DICOM meta information in *NiftiWrapper* object : see https://github.com/moloney/dcmstack/blob/master/src/dcmstack/dcmmeta.py#L1185 . This object accepts a Nifti image as input, that usually contains a *DcmMetaExtension*, and has methods *get_meta* (to get metadata from extension), *split* (for taking slice specific metadata into the split parts), *meta_valid* to check the metadata against the Nifti information, and methods to remove / replace the extension, save to a filename, and create the object with various alternative classmethod constructors.

In particular, the *meta_valid* method needs to know about both the enclosed image, and the enclosed meta data.

Can we put this stuff into the *SpatialImage* image object of nibabel, so we don't need this wrapper object?

Plan

Put the *DcmMeta* data into the *extra* object that is input to the *SpatialImage* and all other nibabel image types.

Add a *get_meta* method to *SpatialImage* that uses the to-be-defined API of the *extra* object. Maybe, by default, this would just get keys out of the mapping.

Define an API for the *extra* object to give back metadata that is potentially varying (per slice or volume). We also need a way to populate the *extra* object when loading an image that has an associated *DcmMeta* object.

Use this API to get metadata. Try and make this work with functions outside the *SpatialImage* such as *four_to_three* and *three_to_four* in *nibabel.funcs*. These functions could use the *extra* API to get varying meta-information.

** TODO : specific proposal for *SpatialImage* and *extra* API changes **

Detecting slice or volume-specific data difficult for 3D and 4D DICOMS

The *DcmMeta* object needs to be able to identify slice and volume specific information when reading the DICOM, so that it can correctly split the resulting metadata, or merge it.

This is easy for slice-by-slice DICOM files because anything that differs between the slices is by definition slice-specific. For 3D and 4D data, such as Siemens Mosaic, some of the fields in the private headers contains slice-by-slice information for the volume contained. There's not automatic way of detecting slice-by-slice information in this case, so we have to specify which fields are slice-by-slice when reading. That is, we need to specialize the DICOM read for each type of volume-containing DICOM - such as Mosaic or the Philips multi-frame format.

Plan

Add *create_dcmmeta* method to the nibabel DICOM wrapper objects, that can be specialized for each known DICOM format variation. Put the rules for slice information etc into each class.

For the Siemens files, we will need to make a list of elements from the private CSA headers that are known to be slice specific. For the multiframe DICOM files we should be able to do this in a programmatic manner, since the varying data should live in the PerFrameFunctionalSequence DICOM element. Each element that is reclassified should be simplified with the *DcmMeta.simplify* method so that it can be classified appropriately.

Meta data in nested DICOM sequences can not be independently classified

The code for summarizing meta data only works on the top level of key/value pairs. Any value that is a nested dataset is treated as a single entity, which prevents us from classifying its individual elements differently.

In a DICOM data set, any element that is a sequence contains one or more nested DICOM data sets. For most MRI images this is not an issue since they rarely contain many sequences, and the ones they do are usually small and relatively unimportant. However in multiframe DICOM files make heavy use of nested sequences to store data.

Plan

This same issue was solved for the translated Siemens CSA sub headers by unpacking each nested dataset by joining the keys from each level with a dotted notation. For example, in the *CsaSeries* subheader there is a nested *MrPhoenixProtocol* dataset which has an element *ulVersion* so the key we use after unpacking is *CsaSeries.MrPhoenixProtocol.ulVersion*.

We can take the same approach for DICOM sequence elements. One additional consideration is that each of these element is actually a list of data sets, so we would need to add an index number to the key somehow.

The alternative is to handle nested data sets recursively in the meta data summarizing code. This would be fairly complex and you would no longer be able to refer to each element with a single string, at least not without some mini-language for traversing the nested datasets.

Improving access to varying meta data through the Nifti

Currently, when accessing varying meta data through the *get_meta* method you can only get one value at a time:

```
>>> echo_times = [nii.get_meta('EchoTime', (0, 0, 0, idx))
                  for idx in xrange(data.shape[-1])]
```

You can easily get multiple values from the *DcmMeta* object itself, but then you lose the capability to automatically check if the meta data is valid in relation to the current image.

BIAP5 - A streamlines converter

Author Marc-Alexandre Côté

Status Draft

Type Standards

Created 2013-09-03

The first objective of this proposal is to add support to other streamlines format. The second objective is to be able to easily convert from one file format to another.

Motivation

There are a couple of different formats for saving streamlines to a file. Currently, NiBabel only support one of them: TRK from Trackvis. NiBabel could greatly benefit from supporting other formats: TCK (MRtrix), VTK (Camino, MITK) and more. Moreover, being able to move from one format to another would be convenient. To ease the conversion process, a generic format from which to inherit and some common header fields would be necessary. This is similar to what NiBabel already has for neuroimages.

After implementing this proposal, users could load and use streamlines file like this:

```
>>> import nibabel as nib
>>> f = nib.streamlines.load('my_trk.trk', lazy_load=False)
>>> type(f)
nibabel.streamlines.base_format.Streamlines
>>> f.points
[array([ [1, 1, 1],
         [2, 2, 2],
         [3, 3, 3] ]),
```

(continues on next page)

(continued from previous page)

```
 array([ [4, 4, 4],
         [5, 5, 5] ])]
>>> nib.streamlines.convert('my_trk.trk', 'my_tck.tck')
>>> f2 = nib.streamlines.load('my_trk.tck', lazy_load=False)
>>> type(f2)
nibabel.streamlines.base_format.Streamlines
>>> f2.points
[array([ [1, 1, 1],
         [2, 2, 2],
         [3, 3, 3] ]),
 array([ [4, 4, 4],
         [5, 5, 5] ])]
```

Of course, similar functions will be available for 'scalars' (per point) and 'properties' (per streamline) as defined in the TrackVis format. A simple example to save three streamlines with no scalars nor properties would look like this:

```
>>> import nibabel as nib
>>> points = [np.arange(1*3).reshape((1,3)),
              np.arange(2*3).reshape((2,3)),
              np.arange(5*3).reshape((5,3))]
>>> streamlines = nib.streamlines.Streamlines(points)
>>> nib.streamlines.save(streamlines, 'data1.trk')  # Default TRK header is used but␣
↪updated with streamlines information.

>>> FA = nib.load('FA.nii')
>>> streamlines.header = nib.streamlines.header.from_nifti(FA)  # Uses information of␣
↪the FA to create an header.
>>> nib.streamlines.save(streamlines, 'data2.trk')  # Streamlines' header is used but␣
↪also updated with streamlines information.

>>> from nib.streamlines.header import VOXEL_ORDER, VOXEL_SIZES
>>> hdr = nib.streamlines.TrkFile.get_empty_header()  # Default TRK header
>>> hdr[VOXEL_ORDER] = "LAS"
>>> hdr[VOXEL_SIZES] = (2, 2, 2)
>>> streamlines.header = hdr
>>> nib.streamlines.save(streamlines, 'data3.trk')  # Uses hdr to create a TRK header.
```

Overview

All code related to managing streamlines should be kept in a separate folder: `nibabel.streamlines`. A first file, `base_format.py`, would contain base classes acting as general interfaces from which new streamlines file format will inherit.

Streamlines would be represented by its own class `Streamlines` which will have three main properties: `points`, `scalars` and `properties`. Streamlines objects can be iterate over producing tuple of `points`, `scalars` and `properties` for each streamline.

The generic class `StreamlinesFile` would look like this:

```
class StreamlinesFile:
    @classmethod
    def get_magic_number(cls):
        raise NotImplementedError()

    @classmethod
```

(continues on next page)

(continued from previous page)

```
    def is_correct_format(cls, fileobj):
        raise NotImplementedError()

    @classmethod
    def get_empty_header(cls):
        raise NotImplementedError()

    @classmethod
    def load(cls, fileobj, lazy_load=True):
        raise NotImplementedError()

    @classmethod
    def save(cls, streamlines, fileobj):
        raise NotImplementedError()

    @staticmethod
    def pretty_print(streamlines):
        raise NotImplementedError()
```

When inheriting from a base class, a specific streamline format class should know how to do its i/o, in particular how to iterate through the streamlines without loading the whole file into memory.

Once, the right interface is in place, the conversion part should be quite easy. Moreover, the conversion could be done without loading the input file entirely into memory thanks to generators. Actually, the convert function should looks like this:

```
def convert(in_fileobj, out_filename):
    # Loading part
    streamlines_file = detect_format(in_fileobj)
    streamlines = streamlines_file.load(in_fileobj, lazy_load=True)

    # Saving part
    streamlines_file = detect_format(out_filename)
    streamlines_file.save(streamlines, out_filename)
```

Of course, this implies some sort of general header compatibility between every format.

Issues

Header

Like it is done in NiBabel, headers should be defined using the `numpy.dtype`. This consists of a list of tuples, each one containing information (name, datatype and shape) about one field of the header. Once loaded, the header will acted as a dictionary using the name of each field as the key. Ideally, header of different formats would be the same, but it is not. To avoid manually writing each possible conversion between header formats, a general architecture should be put in place.

One solution is to define some sort of `CommonHeader` containing an enum of the most common field (i.e. NB_FIBERS, VOXEL_SIZES, DIMENSIONS, etc). Like that, instead of specifying a field's name in the header definition, the suited enum constant should be used if there is one, otherwise the name is hard coded to a string representing the field. It should be make clear, in the documentation of `CommonHeader`, what is the expected value of a common field.

Future Work

A first interesting subclass would be the `DynamicStreamlineFile` offering a way to append streamlines to an existing file when format permits it.

BIAP6 - Identifying image axes

Author Matthew Brett

Status Draft

Type Standards

Created 2015-07-11

Background

Image axes can have meaningful labels.

For example in a typical 4D NIfTI file, as we move along the 4th dimension in the image array, we are also moving in time. For example, this would be the first volume (in time):

```
img = nibabel.load('my_4d.nii')
data = img.get_data()
vol0 = data[..., 0]
```

and this would be second volume in time:

```
vol1 = data[..., 1]
```

It would therefore be reasonable to label the 4th axis of this image as 'time' or 't'.

We need to know which axis is the "time" axis for many reasons, including being able to select whole image volumes to align during motion correction, and doing spatial smoothing, where we want to avoid smoothing along the time dimension.

It is common to acquire MRI images one slice at a time. In a 3D or 4D NIfTI, the 3rd axis often contains these slices. So this these would be the first and second slices of data collected by the scanner:

```
slice0 = vol0[:, :, 0]
slice1 = vol0[:, :, 1]
```

In this case we might refer to the 3rd axis as the "slice" axis. We might care about knowing the "slice" axis, because we do processing specific to the slice axis, such as slice-timing correction.

For an individual 2D slice, MRI physicists distinguish between the image axis encoded during a single continual readout of the signal (frequency encoding direction) and the image axis encoded in a series of stepwise changes in the phase encode gradient (phase encoding direction). We care about the phase encoding direction because we usually correct for image distortion only along this direction.

Let us say that the first axis is the frequency encoding axis, and the second is the phase encoding axis. Now we can label all four of our axes:

- "frequency";
- "phase";
- "slice";

- "time".

In fact the NIfTI format can store this information. NIfTI specifies that the fourth image dimension should have units in terms of time (seconds), frequency (Hertz, radians per second) or concentration (parts per million), where the value difference between elements on the fourth axis is in `img.header['pixdim'][4]`, and the units of this difference are available in `img.header['xyzt_units']`. The field `img.header['dim_info']` can identify the frequency, phase and slice-encoding axes.

Time axis as the fourth axis

In the NIfTI standard, time must be the fourth dimension.

In fact, the NIfTI standard specifies that the fourth axis *must* be time. If we want to store more than one volume that do not differ across time, then we have to set the 4th dimension to be length 1, and have 5th dimension have length > 1. Quoting from the standard:

```
In NIFTI-1 files, dimensions 1,2,3 are for space, dimension 4 is for time,
and dimension 5 is for storing multiple values at each spatiotemporal
voxel.
```

This arrangement happens in practice. For example, SPM deformation fields have three values for each voxel (x, y, z displacement), and have shape (I, J, K, 1, 3):

```
In [7]: img = nib.load('y_highres001.nii')
In [8]: img.shape
Out[8]: (121, 145, 121, 1, 3)
```

So, for correctly written NIfTI images, we can identify time by the fact that it is the fourth axis.

MGH format also appears to use the fourth dimension for time. The dimensions are listed in order `width, height, depth, nframes` and "frames" is always the slowest changing dimension in the image data buffer. Of course, in numpy, this does not tell us which axis this must be in the returned array, but at least the `load_mgh.m` MATLAB function (see MGH format) returns the frame axis as the last axis, as does nibabel.

The ECAT and PAR / REC formats seem to be primarily based on and stored as slices (2D arrays) which can then be concatenated to form volumes, implying a slowest-changing axis of volume. Nibabel currently arranges PAR images with volume as the 4th and last axis.

On the other hand, the MINC format:

1. gives specific names to the image data axes so we can directly find the time axis
2. expects (given the common ordering of these names in MINC files) that the time axis will be first:

```
In [31]: mnc2 = h5py.File('nibabel/tests/data/minc2_4d.mnc', 'r')['minc-2.0']
In [32]: mnc2['dimensions'].values()
Out[32]:
[<HDF5 dataset "time": shape (2,), type "<f8">,
<HDF5 dataset "xspace": shape (), type "<i4">,
<HDF5 dataset "yspace": shape (), type "<i4">,
<HDF5 dataset "zspace": shape (), type "<i4">]
```

This reflects MINC's lineage as C-library, where the C convention is for the first axis in an array is the slowest changing. `arr[0]` in a C-convention 4D array would be the first volume, where time (volume) is the slowest changing axis.

MINC2 uses HDF5 storage, and HDF5 uses C storage order for standard contiguous arrays on disk - see "7.3.2.5. C versus Fortran Dataspaces" in chapter 7 of the HDF5 user guide.

BrainVoyager STC files store data in (fastest to slowest changing) order: columns (of slice); rows (of slice); time; slice. The VTC stores the data in the (fast to slow) order: time; Anterior->Posterior; Superior->Inferior; Left->Right.

Images can have more than four axes

We've already seen the example of NIfTI images where the 4th axis is length 1 and the 5th axis is length 3, encoding a deformation field.

This is a trick NIfTI uses to allow us to identify the "time" axis.

We can also have (rarely) images of 5D, where the time axis has length > 1. For example, some MR acquisitions take two echoes per time point, so we might have an image of shape (64, 64, 32, 200, 2), where the fourth axis is time and the fifth axis is echo number.

The current nibabel convention

The nibabel rule of thumb has been that, when we return an image array, it should be in the order described in the format's user documentation.

So, for NIfTI format images, the image dimension sizes are listed in fastest to slowest changing order, implying that the expected array to be returned will have that same axis order. Time is always the fourth (rather than the first) dimension of a 4D NIfTI. Nibabel NIfTI images return the array in that order, and the time / volume axis is the last in a 4D nibabel NIfTI image array.

On the other hand MINC clearly expects that the axes will be returned in the order the axes are listed in the MINC file. This is also (usually) the slowest-to-fastest changing order in the underlying file, and by convention, the first axis is the time axis. Nibabel MINC images return the array in this same order with the time / volume axis first, but in general it returns the array with the axes in the order listed in the MINC file.

We don't currently have BrainVoyager support, so this will be a decision we have to make before finalizing the API.

Distinguishing time and volume

A *volume* is a complete set of slices making up one brain image.

In NIfTI:

- 3D image: volume == image array i.e. `arr[:, :, :]`;
- > 3D image: volume == a single slice over the final dim > 3 dimensions e.g.: `arr[:, :, :, 2]` (4D); `arr[:, :, :, 0, 3]` (5D).

We saw above that the MGH format refers to a volume (in our sense) as a *frame*. ECAT has the same usage - a frame is a 3D volume. The fmristat software uses frame in the same sense |--| e.g. line 32 of example.m.

Unfortunately DICOM appears to use "frame" to mean a 2D slice. For example, here is the definition of a "multi-frame image":

```
3.8.9 Multi-frame image:
    Image that contains multiple two-dimensional pixel planes.
```

From PS 3.3 of the 2011 DICOM standrd.

Possible solutions to finding axes

A general solution for finding axes would be to attach axis labels to the returned image data array, or to the image object.

A less general solution would be to identify the time axis by convention - say - by being the fourth axis in a 4D array.

Finding the time axis is an urgent problem, because we are currently considering utility routines for (spatial) smoothing, and viewing images, that need to know which axis is time.

General solution: associating axes and labels

Possible options:

- Add a property *time_axis_index* to the image class. This always returns 3 (4th axis) for images other than MINC. For MINC, it returns the index of the image dimension labeled `time`;
- Add a property *axis_labels* to the image class. By default, most image types return *'i', 'j', 'k', 'time'*. MINC returns the image dimension labels;
- Copy or depend on datarray (no other dependencies) or *xray* (depends on Pandas). Use these to attach labels directly to the image data array axes. These labels could then be preserved through operations like slicing.

Using convention : enforcing time as 4th axis

This solution could be implemented as well as the solution using labels.

At the moment, we can always identify the time axis in the NIfTI file, because it is the 4th axis in the returned image.

This is probably so for:

- PAR/REC
- ECAT
- MGH

but not so for MINC1 or MINC2, where time is typically (?always) the first axis.

One option would be to make a new MINC1, MINC2 image class that reorders the MINC axes to have time last. Call these new classes *NiMINC1, NiMINC2*.

In order to avoid surprise, we continue to return MINC1, MINC2 class images from *nibabel.load*, but give a `DeprecationWarning` when doing this, saying that the default load will change in future versions of nibabel, and suggesting the *as_niminc=True* keyword-only argument to load, defaulting to *as_niminc=False* (giving the current nibabel behavior).

In Nibabel 3.0, we require the *as_niminc* keyword argument.

In Nibabel 4.0, we default to *as_niminc=True*.

We would still have to deal with MINC1, MINC2 images in memory - and therefore cannot in general assume that the fourth dimension of any image data array is time. In order to deal with this, routines that need to know the time dimension would have to check whether they were dealing with MINC1, MINC2, which ends up being similar to the *time_axis_index* option above.

BIAP7 - Loading multiple images

Author Matthew Brett

Status Draft

Type Standards

Created 2015-07-18

Background

Some formats store images with different shapes in the same file

The ECAT file format can contain more than one type of image in a single image file.

ECAT can store many *frames* in a single image file. Each frame has its own *subheader*. The subheader specifies the 3D image size; each frame can therefore have a different image size.

We currently raise an error if you try and load an ECAT file where the frames do not have the same 3D dimensions.

It would be better if we could allow loading multiple images with different image dimensions, from a single ECAT file.

Vista data format and Lipsia format are other formats that allow saving multiple images with different image dimensions in the same file. We don't currently support Lipsia or Vista formats and it is not clear how we would do that with the current `load` API.

We have had some discussion about saving multiple images into a single HDF5 file - see https://github.com/nipy/nibabel/pull/215#issuecomment-122357444

It can be useful to load 4D images as multiple 3D images

We sometimes want to load a 4D image as multiple 3D images.

When we are doing motion correction, we often want to split up a 4D image into separate 3D images.

Motion estimation results in different affines for each volume in the 4D time series. At the moment we have no API for returning these affines with a 4D image. One way of doing that is to load the 4D image and affines as a sequence of 3D images, each with their own affine.

We currently have a proposal open for a JSON header extension that can store these 4D affines for a 4D NIfTI file.

SPM saves the affines in an associated `.mat` file, with one affine per volume in the 4D image.

Options

Return an image sequence from `load` for some file formats

We don't currently load ECAT files from the top-level `nibabel.load` function.

We do have `nibabel.ecat.load`, which raises an error for an ECAT file having frames with different image dimensions.

We could therefore choose to return a sequence of images from `nibabel.load` on an ECAT file, with one element per frame in the ECAT file.

Most ECAT images are 4D images, in the sense that the frames in the file do all have the same image dimensions and data type, so this might be cumbersome as a default.

We would have to work out how to deal with `nibabel.ecat.load`.

The same principles apply to the Lipsia / Vista formats, except we have no backward-compatibility problems, and it seems to be more common for these formats to mix image types in a single file.

Add a `load_multi` top-level function

`nibabel.load_multi` always returns an image sequence.

`nibabel.load` always returns a single image.

`nibabel.load` on ECAT (etc) files could first do `load_multi`, then check the resulting image dimensions, raising an error if incompatible, concatenating otherwise.

`load_multi` on current formats like NIfTI could return one image per volume, where each volume might have its own affine, as loaded from the JSON header extension or the SPM `.mat` file.

Next steps:

- Make sure there are use-cases where you could wish to call *load* vs. *load_multi* on the same image (perhaps a Nifti image with different affines for each volume)
- Investigate AFNI file formats as a use-case for this.
- Check the *nilearn* codebase, see if *iter_img* and *slice_img* functions might offer a post-*load* alternative. Also check if those functions could be deprecated in favor of slicing / iterating on *dataobj*
- Create a new issue to implement getting an iterator on *dataobj*?

BIAP8 - Always load image data as floating point

Author Matthew Brett

Status Accepted

Type Standards

Created 2018-04-18

`get_fdata` shipped as of nibabel 2.2.0.

See this mailing list thread for discussion on an earlier version of this proposal.

Background

Summary

The problem with our current `get_data` method is that the returned data type is difficult to predict, and can switch between integer and floating point types depending on values in the image header.

The underlying problem is that the author and the user of a given NIfTI image would be unlikely to expect that the scalefactors of the NIfTI header (which the user will probably not be aware of) will affect the calculations done on the image data after loading into memory.

In detail

At the moment, if you do this:

```
img = nib.load('my_image.nii')
data = img.get_data()
```

then the data type (dtype) of the returned data array depends on the values in the header of `my_image.nii`. Specifically, if the raw on-disk data type is `np.int16` (it often is) and the header scalefactor values are default (1 for slope, 0 for intercept) then you will get back an array of the on-disk data type - here `np.int16`.

This is very efficient in terms of memory, but it can be a real trap unless you are careful.

For example, let's say you had a pipeline where you did this:

```
sum = img.get_data().sum()
```

That would work fine most of the time, when the data on disk is floating point, or the scalefactors are not default (1, 0). Then one day, you get an image with `int16` data type on disk and (1, 0) scalefactors, and your *sum* calculation is now being done in int16, and silently overflows. I (MB) ran into this when teaching - I had to cast some image arrays to floating point to get sensible answers.

Current implementation

`get_data` has the following implementation, at time of writing:

```
def get_data(self):
    """ Return image data from image with any necessary scalng applied

    If the image data is a array proxy (data not yet read from disk) then
    read the data, and store in an internal cache.  Future calls to
    ``get_data`` will return the cached copy.

    Returns
    -------
    data : array
        array of image data
    """
    if self._data_cache is None:
        self._data_cache = np.asanyarray(self._dataobj)
    return self._data_cache
```

Note that:

- `self._dataobj` may well be an array proxy object;
- `np.asanyarray` forces the read of an array proxy object into a numpy array;
- the read also fills an internal cache.

Proposal - add, prefer `get_fdata` method

The future default behavior of nibabel should be to do the thing least likely to trip you up by accident. But - we do not want the result of `get_data` to change silently between nibabel versions.

- step 1: now - add `get_fdata` method:

```
def get_fdata(self, dtype=np.float64):
    """ Return floating point image data with necessary scalng applied.

    If the image data is an array proxy (data not yet read from disk) then
    read the data from file, and retain the result in an internal cache.
    Future calls to ``get_fdata`` on the same image instance will return
    the cached copy.

    Parameters
    ----------
    dtype : numpy dtype specifier
        A numpy dtype specifier specifying a floating point type.  Data is
        returned as this floating point type.  Default is ``np.float64``.

    Returns
    -------
    fdata : array
        Array of image data of data type `dtype`.
    """
    dtype = np.dtype(dtype)
    if not issubclass(dtype, np.inexact):
        raise ValueError('{} should be floating point type'.format(dtype))
    if self._fdata_cache is None:
        self._fdata_cache = np.asanyarray(self._dataobj).astype(dtype)
    return self._fdata_cache
```

  Change all instances of `get_data` in documentation to `get_fdata`.

  Add warning about pending deprecation in `get_data` method, with suggestion to use `get_fdata` or `np.asanyarray(img.dataobj)` if you want the previous behavior, on the lines of:

```
We recommend you use the ``get_fdata`` method instead of the ``get_data``
method, because it is easier to predict the return data type.  We will
deprecate the ``get_data`` method around April 2018, and remove it around
April 2020.

If you don't care about the predictability of the return data type, and
you want the minimum possible data size in memory, you can replicate the
array that would be returned by ``img.get_data()`` by using
``np.asanyarray(img.dataobj)``.
```

  Add floating point cache `self._fdata_cache` to cache cleared by `uncache` method.

- step 2: around one year from now - deprecate `get_data` method;
- step 3: around three years from now - make `get_data` method raise an error such as `NotImplementedError` with a helpful message, and remove associated `self._data_cache` attribute. Leave this error in place for a long time, to help people porting older code.

BIAP X — Template and Instructions

Author <list of authors' real names and optionally, email addresses>

Status <Draft | Active | Accepted | Deferred | Rejected | Withdrawn | Final | Superseded>

Type <Standards Track | Process>

Created <date created on, in yyyy-mm-dd format>

Resolution <url> (required for Accepted | Rejected | Withdrawn)

Abstract

The abstract should be a short description of what the BIAP will achieve.

Note that the — in the title is an elongated dash, not -.

Motivation and Scope

This section describes the need for the proposed change. It should describe the existing problem, who it affects, what it is trying to solve, and why. This section should explicitly address the scope of and key requirements for the proposed change.

Usage and Impact

This section describes how users of Nibabel will use features described in this BIAP. It should be comprised mainly of code examples that wouldn't be possible without acceptance and implementation of this BIAP, as well as the impact the proposed changes would have on the ecosystem. This section should be written from the perspective of the users of Nibabel, and the benefits it will provide them; and as such, it should include implementation details only if necessary to explain the functionality.

Backward compatibility

This section describes the ways in which the BIAP breaks backward compatibility.

The mailing list post will contain the BIAP up to and including this section. Its purpose is to provide a high-level summary to users who are not interested in detailed technical discussion, but may have opinions around, e.g., usage and impact.

Detailed description

This section should provide a detailed description of the proposed change. It should include examples of how the new functionality would be used, intended use-cases and pseudo-code illustrating its use.

Related Work

This section should list relevant and/or similar technologies, possibly in other libraries. It does not need to be comprehensive, just list the major examples of prior and relevant art.

Implementation

This section lists the major steps required to implement the BIAP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link to related pull requests as the implementation progresses.

Any pull requests or development branches containing work on this BIAP should be linked to from here. (A BIAP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

Discussion

This section may just be a bullet list including links to any discussions regarding the BIAP:

- This includes links to mailing list threads or relevant GitHub issues.

10.3.7 Adding test data

1. We really, really like test images, but
2. We are rather conservative about the size of our code repository.

So, we have two different ways of adding test data.

1. Small, open licensed files can go in the `nibabel/tests/data` directory (see below);
2. Larger files or files with extra licensing terms can go in their own git repositories and be added as submodules to the `nibabel-data` directory.

Small files

Small files are around 50K or less when compressed. By "compressed", we mean, compressed with zlib, which is what git uses when storing the file in the repository. You can check the exact length directly with Python and a script like:

```
import sys
import zlib

for fname in sys.argv[1:]:
    with open(fname, 'rb') as fobj:
        contents = fobj.read()
    compressed = zlib.compress(contents)
    print(fname, len(compressed) / 1024.)
```

One way of making files smaller when compressed is to set uninteresting values to zero or some other number so that the compression algorithm can be more effective.

Please don't compress the file yourself before committing to a git repo unless there's a really good reason; git will do this for you when adding to the repository, and it's a shame to make git compress a compressed file.

Files with open licenses

We very much prefer files with completely open licenses such as the PDDL 1.0 or the CC0 license.

The files in the `nibabel/tests/data` will get distributed with the nibabel source code, and this can easily get installed without the user having an opportunity to review the full license. We don't think this is compatible with extra license terms like agreeing to cite the people who provided the data or agreeing not to try and work out the identity of the person who has been scanned, because it would be too easy to miss these requirements when using nibabel. It is fine to use files with these kind of licenses, but they should go in their own repository to be used as a submodule, so they do not need to be distributed with nibabel.

Adding the file to `nibabel/tests/data`

If the file is less then about 50K compressed, and the license is open, then you might want to commit the file under `nibabel/tests/data`.

Put the license for any new files in the COPYING file at the top level of the nibabel repo. You'll see some examples in that file already.

Adding as a submodule to `nibabel-data`

Make a new git repository with the data.

There are example repos at

- https://github.com/yarikoptic/nitest-balls1
- https://github.com/matthew-brett/nitest-minc2

Despite the fact that both the examples are on github, Bitbucket is good for repos like this because they don't enforce repository size limits.

Don't forget to include a LICENSE and README file in the repo.

When all is done, and the repository is safely on the internet and accessible, add the repo as a submodule to the `nitests-data` directory, with something like this:

```
git submodule add https://bitbucket.org/nipy/rosetta-samples.git nitests-data/rosetta-
↪samples
```

You should now have a checked out copy of the `rosetta-samples` repository in the `nibabel-data/rosetta-samples` directory. Commit the submodule that is now in your git staging area.

If you are writing tests using files from this repository, you should use the `needs_nibabel_data` decorator to skip the tests if the data has not been checked out into the submodules. See `nibabel/tests/test_parrec_data.py` for an example. For our example repository above it might look something like:

```
from .nibabel_data import get_nibabel_data, needs_nibabel_data

ROSETTA_DATA = pjoin(get_nibabel_data(), 'rosetta-samples')
```

(continues on next page)

(continued from previous page)

```
@needs_nibabel_data('rosetta-samples')
def test_something():
    # Some test using the data
```

Using submodules for tests

Tests run via nibabel on travis start with an automatic checkout of all submodules in the project, so all test data submodules get checked out by default.

If you are running the tests locally, you may well want to do:

```
git submodule update --init
```

from the root nibabel directory. This will checkout all the test data repositories.

How much data should go in a single submodule?

The limiting factor is how long it takes travis-ci to checkout the data for the tests. Up to a hundred megabytes in one repository should be OK. The joy of submodules is we can always drop a submodule, split the repository into two and add only one back, so you aren't committing us to anything awful if you accidentally put some very large files into your own data repository.

If in doubt

If you are not sure, try us with a pull request to nibabel github, or on the nipy mailing list, we will try to help.

10.3.8 How to add a new image format to nibabel

These are some work-in-progress notes in the hope that they will help adding a new image format to NiBabel.

Philosophy

As usual, the general idea is to make your image as explicit and transparent as possible.

From the Zen of Python (`import this`), these guys spring to mind:

- Explicit is better than implicit.
- Errors should never pass silently.
- In the face of ambiguity, refuse the temptation to guess.
- Now is better than never.
- If the implementation is hard to explain, it's a bad idea.

So far we have tried to make the nibabel version of the image as close as possible to the way the user of the particular format is expecting to see it.

For example, the NIfTI format documents describe the image with the first dimension of the image data array being the fastest varying in memory (and on disk). Numpy defaults to having the last dimension of the array being the fastest varying in memory. We chose to have the first dimension vary fastest in memory to match the conventions in the NIfTI specification.

Helping us to review your code

You are likely to know the image format much much better than the rest of us do, but to help you with the code, we will need to learn. The following will really help us get up to speed:

1. Links in the code or in the docs to the information on the file format. For example, you'll see the canonical links for the NIfTI 2 format at the top of the *nifti2* file, in the module docstring;
2. Example files in the format; see *Adding test data*;
3. Good test coverage. The tests help us see how you are expecting the code and the format to be used. We recommend writing the tests first; the tests do an excellent job in helping us and you see how the API is going to work.

The format can be read-only

Read-only access to a format is better than no access to a format, and often much better. For example, we can read but not write PAR / REC and MINC files. Having the code to read the files makes it easier to work with these files in Python, and easier for someone else to add the ability to write the format later.

The image API

An image should conform to the image API. See the module docstring for *spatialimages* for a description of the API.

You should test whether your image does conform to the API by adding a test class for your image in `nibabel.tests.test_image_api`. For example, the API test for the PAR / REC image format looks like:

```
class TestPARRECAPI(LoadImageAPI):
    def loader(self, fname):
        return parrec.load(fname)

    example_images = PARREC_EXAMPLE_IMAGES
```

where your work is to define the `EXAMPLE_IMAGES` list — see the `nibabel.tests.test_parrec` file for the PAR / REC example images definition.

Where to start with the code

There is no API requirement that a new image format inherit from the general *SpatialImage* class, but in fact all our image formats do inherit from this class. We strongly suggest you do the same, to get many simple methods implemented for free. You can always override the ones you don't want.

There is also a generic header class you might consider building on to contain your image metadata — *Header*. See that class for the header API.

The API does not require it, but if it is possible, it may be good to implement the image data as loaded from disk as an array proxy. See the docstring of *arrayproxy* for a description of the API, and see the module code for an implementation of the API. You may be able to use the unmodified *ArrayProxy* class for your image type.

If you write a new array proxy class, add tests for the API of the class in `nibabel.tests.test_proxy_api`. See `TestPARRECAPI` for an example.

A nibabel image is the association of:

1. The image array data (as implemented by an array proxy or a numpy array);

2. An affine relating the image array coordinates to an RAS+ world (see *Coordinate systems and affines*);
3. Image metadata in the form of a header.

Your new image constructor may well be the default from *SpatialImage*, which looks like this:

```
def __init__(self, dataobj, affine, header=None,
             extra=None, file_map=None):
```

Your job when loading a file is to create:

1. `dataobj` - an array or array proxy;
2. `affine` - 4 by 4 array relating array coordinates to world coordinates;
3. `header` - a metadata container implementing at least `get_data_dtype`, `get_data_shape`.

You will likely implement this logic in the `from_file_map` method of the image class. See *PARRECImage* for an example.

A recipe for writing a new image format

1. Find one or more examples images;
2. Put them in `nibabel/tests/data` or a data submodule (see *Adding test data*);
3. Create a file `nibabel/tests/test_my_format_name_here.py`;
4. Use some program that can read the format correctly to fill out the needed fields for an `EXAMPLE_IMAGES` list (see `nibabel.tests.test_parrec.py` for example);
5. Add a test class using your `EXAMPLE_IMAGES` to `nibabel.tests.test_image_api`, using the PARREC image test class as an example. Now you have some failing tests — good job!;
6. If you can, extract the metadata information from the test file, so it is small enough to fit as a small test file into `nibabel/tests/data` (don't forget the license);
7. Write small maybe private functions to extract the header metadata from your new test file, testing these functions in `test_my_format_name_here.py`. See *parrec* for examples;
8. When that is working, try sub-classing *Header*, and working out how to make the `__init__` and `from_fileboj` methods for that class. Test in `test_my_format_name_here.py`;
9. When that is working, try sub-classing *SpatialImage* and working out how to load the file with the `from_file_map` class;
10. Now try seeing if you can get your `test_image_api.py` tests to pass;
11. Consider adding more test data files, maybe to a test data repository submodule (*Adding test data*). Check you can read these files correctly (see `nibabel.tests.test_parrec_data` for an example).
12. Ask for advice as early and as often as you can, either with a work-in-progress pull request (the easiest way for us to review) or on the mailing list or via github issues.

10.3.9 Developer discussions

Some miscellaneous documents on background, future development and work in progress.

Image use-cases in SPM

SPM uses a *vol struct* as a structure characterizing an object. This is a Matlab `struct`. A `struct` is like a Python dictionary, where field names (strings) are associated with values. There are various functions operating on vol structs, so the vol struct is rather like an object, where the methods are implemented as functions. Actually, the distinction between methods and functions in Matlab is fairly subtle - their call syntax is the same for example.

```
>> fname = 'some_image.nii';
>> vol = spm_vol(fname) % the vol struct

vol =

      fname: 'some_image.nii'
        mat: [4x4 double]
        dim: [91 109 91]
         dt: [2 0]
      pinfo: [3x1 double]
          n: [1 1]
    descrip: 'NIFTI-1 Image'
    private: [1x1 nifti]

>> vol.mat % the 'affine'

ans =

    -2     0     0    92
     0     2     0  -128
     0     0     2   -74
     0     0     0     1

>> help spm_vol
  Get header information etc for images.
  FORMAT V = spm_vol(P)
  P - a matrix of filenames.
  V - a vector of structures containing image volume information.
  The elements of the structures are:
        V.fname - the filename of the image.
        V.dim   - the x, y and z dimensions of the volume
        V.dt    - A 1x2 array.  First element is datatype (see spm_type).
                  The second is 1 or 0 depending on the endian-ness.
        V.mat   - a 4x4 affine transformation matrix mapping from
                  voxel coordinates to real world coordinates.
        V.pinfo - plane info for each plane of the volume.
                V.pinfo(1,:) - scale for each plane
                V.pinfo(2,:) - offset for each plane
                   The true voxel intensities of the jth image are given
                   by: val*V.pinfo(1,j) + V.pinfo(2,j)
                V.pinfo(3,:) - offset into image (in bytes).
                   If the size of pinfo is 3x1, then the volume is assumed
                   to be contiguous and each plane has the same scalefactor
                   and offset.
```

(continues on next page)

(continued from previous page)

```
  The fields listed above are essential for the mex routines, but other
  fields can also be incorporated into the structure.

  The images are not memory mapped at this step, but are mapped when
  the mex routines using the volume information are called.

  Note that spm_vol can also be applied to the filename(s) of 4-dim
  volumes. In that case, the elements of V will point to a series of 3-dim
  images.

  This is a replacement for the spm_map_vol and spm_unmap_vol stuff of
  MatLab4 SPMs (SPM94-97), which is now obsolete.
 _____________________________________________________________________
  Copyright (C) 2005 Wellcome Department of Imaging Neuroscience


>> spm_type(vol.dt(1))

ans =

uint8

>> vol.private

ans =

NIFTI object: 1-by-1
            dat: [91x109x91 file_array]
            mat: [4x4 double]
     mat_intent: 'MNI152'
           mat0: [4x4 double]
    mat0_intent: 'MNI152'
        descrip: 'NIFTI-1 Image'
```

So, in our (provisional) terms:

- `vol.mat` == `img.affine`
- `vol.dim` == `img.shape`
- `vol.dt(1)` (`vol.dt[0]` in Python) is equivalent to `img.get_data_dtype()`
- `vol.fname` == `img.get_filename()`

SPM abstracts the implementation of the image to the `vol.private` member, that is not in fact required by the image interface.

Images in SPM are always 3D. Note this behavior:

```
>> fname = 'functional_01.nii';
>> vol = spm_vol(fname)

vol =

191x1 struct array with fields:
    fname
    mat
    dim
```

(continues on next page)

(continued from previous page)

```
    dt
    pinfo
    n
    descrip
    private
```

That is, one vol struct per 3D volume in a 4D dataset.

SPM image methods / functions

Some simple ones:

```
>> fname = 'some_image.nii';
>> vol = spm_vol(fname);
>> img_arr = spm_read_vols(vol);
>> size(img_arr) % just loads in scaled data array

ans =

    91   109    91

>> spm_type(vol.dt(1)) % the disk-level (IO) type is uint8

ans =

uint8

>> class(img_arr) % always double regardless of IO type

ans =

double

>> new_fname = 'another_image.nii';
>> new_vol = vol;  % matlab always copies
>> new_vol.fname = new_fname;
>> spm_write_vol(new_vol, img_arr)

ans =

      fname: 'another_image.nii'
        mat: [4x4 double]
        dim: [91 109 91]
         dt: [2 0]
      pinfo: [3x1 double]
          n: [1 1]
    descrip: 'NIFTI-1 Image'
    private: [1x1 nifti]
```

Creating an image from scratch, and writing plane by plane (slice by slice):

```
>> new_vol = struct();
>> new_vol.fname = 'yet_another_image.nii';
>> new_vol.dim = [91 109 91];
>> new_vol.dt = [spm_type('float32') 0]; % little endian (0)
```

(continues on next page)

(continued from previous page)

```
>> new_vol.mat = vol.mat;
>> new_vol.pinfo = [1 0 0]';
>> new_vol = spm_create_vol(new_vol);
>> for vox_z = 1:new_vol.dim(3)
new_vol = spm_write_plane(new_vol, img_arr(:,:,vox_z), vox_z);
end
```

I think it's true that writing the plane does not change the image scalefactors, so it's only practical to use `spm_write_plane` for data for which you already know the dynamic range across the volume.

Simple resampling from an image:

```
>> fname = 'some_image.nii';
>> vol = spm_vol(fname);
>> % for voxel coordinate 10,15,20 (1-based)
>> hold_val = 3; % third order spline resampling
>> val = spm_sample_vol(vol, 10, 15, 20, hold_val)

val =

    0.0510

>> img_arr = spm_read_vols(vol);
>> img_arr(10, 15, 20)  % same as simple indexing for integer coordinates

ans =

    0.0510

>> % more than one point
>> x = [10, 10.5]; y = [15, 15.5]; z = [20, 20.5];
>> vals = spm_sample_vol(vol, x, y, z, hold_val)

vals =

    0.0510    0.0531

>> % you can also get the derivatives, by asking for more output args
>> [vals, dx, dy, dz] = spm_sample_vol(vol, x, y, z, hold_val)

vals =

    0.0510    0.0531


dx =

    0.0033    0.0012


dy =

    0.0033    0.0012


dz =
```

(continues on next page)

(continued from previous page)

```
    0.0020   -0.0017
```

This is to speed up optimization in registration - where the optimizer needs the derivatives.

`spm_sample_vol` always works in voxel coordinates. If you want some other coordinates, you would transform them yourself. For example, world coordinates according to the affine looks like:

```
>> wc = [-5, -12, 32];
>> vc = inv(vol.mat) * [wc 1]'

vc =

   48.5000
   58.0000
   53.0000
    1.0000

>> vals = spm_sample_vol(vol, vc(1), vc(2), vc(3), hold_val)

vals =

    0.6792
```

Odder sampling, often used, can be difficult to understand:

```
>> slice_mat = eye(4);
>> out_size = vol.dim(1:2);
>> slice_no = 4; % slice we want to fetch
>> slice_mat(3,4) = slice_no;
>> arr_slice = spm_slice_vol(vol, slice_mat, out_size, hold_val);
>> img_slice_4 = img_arr(:,:,slice_no);
>> all(arr_slice(:) == img_slice_4(:))

ans =

     1
```

This is the simplest use - but in general any affine transform can go in `slice_mat` above, giving optimized (for speed) sampling of slices from volumes, as long as the transform is an affine.

Miscellaneous functions operating on vol structs:

- `spm_conv_vol` - convolves volume with seperable functions in x, y, z
- `spm_render_vol` - does a projection of a volume onto a surface
- `spm_vol_check` - takes array of vol structs and checks for sameness of image dimensions and `mat` (affines) across the list.

And then, many SPM functions accept vol structs as arguments.

Keeping track of whether images have been modified since load

Summary

This is a discussion of a missing feature in nibabel: the ability to keep track of whether an image object in memory still corresponds to an image file (or files) on disk.

Motivation

We may need to know whether the image in memory corresponds to the image file on disk.

For example, we often need to get filenames for images when passing images to external programs. Imagine a realignment, in this case, in nipy (the package):

```
import nipy
img1 = nibabel.load('meanfunctional.nii')
img2 = nibabel.load('anatomical.nii')
realigner = nipy.interfaces.fsl.flirt()
params = realigner.run(source=img1, target=img2)
```

In `nipy.interfaces.fsl.flirt.run` there may at some point be calls like:

```
source_filename = nipy.as_filename(source_img)
target_filename = nipy.as_filename(target_img)
```

As the authors of the `flirt.run` method, we need to make sure that the `source_filename` corresponds to the `source_img`.

Of course, in the general case, if `source_img` has no corresponding filename (from `source_img.get_filename()`, then we will have to save a copy to disk, maybe with a temporary filename, and return that temporary name as `source_filename`.

In our particular case, `source_img` does have a filename (`meanfunctional.nii`). We would like to return that as `source_filename`. The question is, how can we be sure that the user has done nothing to `source_img` to make it diverge from its original state? Could `source_img` have diverged, in memory, from the state recorded in `meantunctional.nii`?

If the image and file have not diverged, we return `meanfunctional.nii` as the `source_filename`, otherwise we will have to do something like:

```
import tempfile
fname = tempfile.mkstemp('.nii')
img = source_img.to_filename(fname)
```

and return `fname` as `source_filename`.

Another situation where we might like to pass around image objects that are known to correspond to images on disk is when working in parallel. A set of nodes may have fast common access to a filesystem on which the images are stored. If a master is farming out images to nodes, a master node distribution jobs to workers might want to check if the image was identical to something on file and pass around a lightweight (proxied) image (with the data not loaded into memory), relying on the node pulling the image from disk when it uses it.

Possible implementation

One implementation is to have `dirty` flag, which, if set, would tell you that the image might not correspond to the disk file. We set this flag when anyone asks for the data, on the basis that the user may then do something to the data and you can't know if they have:

```
img = nibabel.load('some_image.nii')
data = img.get_fdata()
data[:] = 0
img2 = nibabel.load('some_image.nii')
assert not np.all(img2.get_fdata() == img.get_fdata())
```

The image consists of the data, the affine and a header. In order to keep track of the header and affine, we could cache them when loading the image:

```
img = nibabel.load('some_image.nii')
hdr = img.header
assert img._cache['header'] == img.header
hdr.set_data_dtype(np.complex64)
assert img._cache['header'] != img.header
```

When we need to know whether the image object and image file correspond, we could check the current header and current affine (the header may be separate from the affine for an SPM Analyze image) against their cached copies, if they are the same and the 'dirty' flag has not been set by a previous call to `get_fdata()`, we know that the image file does correspond to the image object.

This may be OK for small bits of memory like the affine and the header, but would quickly become prohibitive for larger image metadata such as large nifti header extensions. We could just always assume that images with large header extensions are *not* the same as for on disk.

The user might be able to override the result of these checks directly:

```
img = nibabel.load('some_image.nii')
assert img.is_dirty == False
hdr = img.header
hdr.set_data_dtype(np.complex64)
assert img.is_dirty == True
img.is_dirty == False
```

The checks are magic behind the scenes stuff that do some safe optimization (in the sense that we are not re-saving the data if that is not necessary), but drops back to the default (re-saving the data) if there is any uncertainty, or the cost is too high to be able to check.

Design of data packages for the nibabel and the nipy suite

See data-package-discuss for a more general discussion of design issues.

When developing or using nipy, many data files can be useful. We divide the data files nipy uses into at least 3 categories

1. *test data* - data files required for routine code testing
2. *template data* - data files required for algorithms to function, such as templates or atlases
3. *example data* - data files for running examples, or optional tests

Files used for routine testing are typically very small data files. They are shipped with the software, and live in the code repository. For example, in the case of `nipy` itself, there are some test files that live in the module path `nipy.testing.data`. Nibabel ships data files in `nibabel.tests.data`. See *Adding test data* for discussion.

template data and *example data* are example of *data packages*. What follows is a discussion of the design and use of data packages.

Use cases for data packages

Using the data package

The programmer can use the data like this:

```
from nibabel.data import make_datasource

templates = make_datasource(dict(relpath='nipy/templates'))
fname = templates.get_filename('ICBM152', '2mm', 'T1.nii.gz')
```

where `fname` will be the absolute path to the template image `ICBM152/2mm/T1.nii.gz`.

The programmer can insist on a particular version of a `datasource`:

```
>>> if templates.version < '0.4':
...     raise ValueError('Need datasource version at least 0.4')
Traceback (most recent call last):
...
ValueError: Need datasource version at least 0.4
```

If the repository cannot find the data, then:

```
>>> make_datasource(dict(relpath='nipy/implausible'))
Traceback (most recent call last):
  ...
nibabel.data.DataError: ...
```

where `DataError` gives a helpful warning about why the data was not found, and how it should be installed.

Warnings during installation

The example data and template data may be important, and so we want to warn the user if NIPY cannot find either of the two sets of data when installing the package. Thus:

```
python setup.py install
```

will import nipy after installation to check whether these raise an error:

```
>>> from nibabel.data import make_datasource
>>> templates = make_datasource(dict(relpath='nipy/templates'))
>>> example_data = make_datasource(dict(relpath='nipy/data'))
```

and warn the user accordingly, with some basic instructions for how to install the data.

Finding the data

The routine `make_datasource` will look for data packages that have been installed. For the following call:

```
>>> templates = make_datasource(dict(relpath='nipy/templates'))
```

the code will:

1. Get a list of paths where data is known to be stored with `nibabel.data.get_data_path()`
2. For each of these paths, search for directory `nipy/templates`. If found, and of the correct format (see below), return a datasource, otherwise raise an Exception

The paths collected by `nibabel.data.get_data_paths()` are constructed from ':' (Unix) or ';' separated strings. The source of the strings (in the order in which they will be used in the search above) are:

1. The value of the `NIPY_DATA_PATH` environment variable, if set
2. A section = `DATA`, parameter = `path` entry in a `config.ini` file in `nipy_dir` where `nipy_dir` is `$HOME/.nipy` or equivalent.
3. Section = `DATA`, parameter = `path` entries in configuration `.ini` files, where the `.ini` files are found by `glob.glob(os.path.join(etc_dir, '*.ini')` and `etc_dir` is `/etc/nipy` on Unix, and some suitable equivalent on Windows.
4. The result of `os.path.join(sys.prefix, 'share', 'nipy')`
5. If `sys.prefix` is `/usr`, we add `/usr/local/share/nipy`. We need this because Python >= 2.6 in Debian / Ubuntu does default installs to `/usr/local`.
6. The result of `get_nipy_user_dir()`

Requirements for a data package

To be a valid NIPY project data package, you need to satisfy:

1. The installer installs the data in some place that can be found using the method defined in *Finding the data*.

We recommend that:

1. By default, you install data in a standard location such as `<prefix>/share/nipy` where `<prefix>` is the standard Python prefix obtained by `>>> import sys; print sys.prefix`

Remember that there is a distinction between the NIPY project - the umbrella of neuroimaging in python - and the NIPY package - the main code package in the NIPY project. Thus, if you want to install data under the NIPY *package* umbrella, your data might go to `/usr/share/nipy/nipy/packagename` (on Unix). Note `nipy` twice - once for the project, once for the package. If you want to install data under - say - the `pbrain` package umbrella, that would go in `/usr/share/nipy/pbrain/packagename`.

Data package format

The following tree is an example of the kind of pattern we would expect in a data directory, where the `nipy-data` and `nipy-templates` packages have been installed:

```
<ROOT>
`-- nipy
    |-- data
    |   |-- config.ini
    |   `-- placeholder.txt
    `-- templates
        |-- ICBM152
        |   `-- 2mm
        |       `-- T1.nii.gz
        |-- colin27
        |   `-- 2mm
        |       `-- T1.nii.gz
        `-- config.ini
```

The `<ROOT>` directory is the directory that will appear somewhere in the list from `nibabel.data.get_data_path()`. The `nipy` subdirectory signifies data for the `nipy` package (as opposed to other NIPY-related packages such as `pbrain`). The `data` subdirectory of `nipy` contains files from the `nipy-data` package. In the `nipy/data` or `nipy/templates` directories, there is a `config.ini` file, that has at least an entry like this:

```
[DEFAULT]
version = 0.2
```

giving the version of the data package.

Installing the data

We use python distutils to install data packages, and the `data_files` mechanism to install the data. On Unix, with the following command:

```
python setup.py install --prefix=/my/prefix
```

data will go to:

```
/my/prefix/share/nipy
```

For the example above this will result in these subdirectories:

```
/my/prefix/share/nipy/nipy/data
/my/prefix/share/nipy/nipy/templates
```

because `nipy` is both the project, and the package to which the data relates.

If you install to a particular location, you will need to add that location to the output of `nibabel.data.get_data_path()` using one of the mechanisms above, for example, in your system configuration:

```
export NIPY_DATA_PATH=/my/prefix/share/nipy
```

Packaging for distributions

For a particular data package - say `nipy-templates` - distributions will want to:

1. Install the data in set location. The default from `python setup.py install` for the data packages will be `/usr/share/nipy` on Unix.
2. Point a system installation of NIPY to these data.

For the latter, the most obvious route is to copy an `.ini` file named for the data package into the NIPY `etc_dir`. In this case, on Unix, we will want a file called `/etc/nipy/nipy_templates.ini` with contents:

```
[DATA]
path = /usr/share/nipy
```

Current implementation

This section describes how we (the nipy community) implement data packages at the moment.

The data in the data packages will not usually be under source control. This is because images don't compress very well, and any change in the data will result in a large extra storage cost in the repository. If you're pretty clear that the data files aren't going to change, then a repository could work OK.

The data packages will be available at a central release location. For now this will be: http://nipy.org/data-packages/ .

A package, such as `nipy-templates-0.2.tar.gz` will have the following sort of structure:

```
<ROOT>
  |-- setup.py
  |-- README.txt
  |-- MANIFEST.in
  `-- templates
      |-- ICBM152
      |   |-- 1mm
      |   |   `-- T1_brain.nii.gz
      |   `-- 2mm
      |       `-- T1.nii.gz
      |-- colin27
      |   `-- 2mm
      |       `-- T1.nii.gz
      `-- config.ini
```

There should be only one `nipy/packagename` directory delivered by a particular package. For example, this package installs `nipy/templates`, but does not contain `nipy/data`.

Making a new package tarball is simply:

1. Downloading and unpacking e.g. `nipy-templates-0.1.tar.gz` to form the directory structure above;
2. Making any changes to the directory;
3. Running `setup.py sdist` to recreate the package.

The process of making a release should be:

1. Increment the major or minor version number in the `config.ini` file;
2. Make a package tarball as above;
3. Upload to distribution site.

There is an example nipy data package `nipy-examplepkg` in the `examples` directory of the NIPY repository.

The machinery for creating and maintaining data packages is available at https://github.com/nipy/data-packaging.

See the `README.txt` file there for more information.

10.3.10 A guide to making a nibabel release

This is a guide for developers who are doing a nibabel release.

The general idea of these instructions is to go through the following steps:

- Make sure that the code is in the right state for release;
- update release-related docs such as the Changelog;
- update various documents giving dependencies, dates and so on;
- check all standard and release-specific tests pass;
- make the *release commit* and release tag;
- check Windows binary builds and slow / big memory tests;
- push source and windows builds to pypi;
- push docs;
- push release commit and tag to github;
- announce.

We leave pushing the tag to the last possible moment, because it's very bad practice to change a git tag once it has reached the public servers (in our case, github). So we want to make sure of the contents of the release before pushing the tag.

Release checklist

- Review the open list of nibabel 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:

```
git log 2.0.0.. | grep '^Author' | cut -d' ' -f 2- | sort | uniq
```

  where `2.0.0` was the last release tag name.

  Then manually go over `git shortlog 2.0.0..` to make sure the release notes are as complete as possible and that every contributor was recognized.

- Look at `doc/source/index.rst` and add any authors not yet acknowledged. You might want to use the following to list authors by the date of their contributions:

```
git log --format="%aN <%aE>" --reverse | perl -e 'my %dedupe; while (<STDIN>) {␣
↪print unless $dedupe{$_}++}'
```

  (From: http://stackoverflow.com/questions/6482436/list-of-authors-in-git-since-a-given-commit#6482473)

  Consider any updates to the `AUTHOR` file.

- Use the opportunity to update the `.mailmap` file if there are any duplicate authors listed from `git shortlog -nse`.

- Check the copyright year in `doc/source/conf.py`
- Refresh the `README.rst` text from the `LONG_DESCRIPTION` in `info.py` by running `make refresh-readme`.

  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 `nibabel/info.py` (e.g. `NUMPY_MIN_VERSION`) and in `doc/source/installation.rst` and in `requirements.txt` and `.travis.yml`. They should at least match. Do they still hold? Make sure nibabel on travis is testing the minimum dependencies specifically.
- Do a final check on the nipy buildbot. Use the `try_branch.py` scheduler available in nibotmi to test particular schedulers.
- Make sure all tests pass (from the nibabel root directory):

```
pytest --doctest-modules nibabel
```

- Make sure you are set up to use the `try_branch.py` - see https://github.com/nipy/nibotmi/blob/master/install.rst#trying-a-set-of-changes-on-the-buildbots
- Make sure all your changes are committed or removed, because `try_branch.py` pushes up the changes in the working tree;
- The following checks get run with the `nibabel-release-checks`, as in:

```
try_branch.py nibabel-release-checks
```

  Beware: this build does not usually error, even if the steps do not give the expected output. You need to check the output manually by going to https://nipy.bic.berkeley.edu/builders/nibabel-release-checks after the build has finished.

  - Make sure all tests pass from sdist:

```
make sdist-tests
```

    and the three ways of installing (from tarball, repo, local in repo):

```
make check-version-info
```

    The last may not raise any errors, but you should detect in the output lines of this form:

```
{'sys_version': '2.6.6 (r266:84374, Aug 31 2010, 11:00:51) \n[GCC 4.0.1␣
↪(Apple Inc. build 5493)]', 'commit_source': 'archive substitution', 'np_
↪version': '1.5.0', 'commit_hash': '25b4125', 'pkg_path': '/var/folders/jg/
↪jgfZ12ZXHwGSFKD85xLpLk+++TI/-Tmp-/tmpGPiD3E/pylib/nibabel', 'sys_executable
↪': '/Library/Frameworks/Python.framework/Versions/2.6/Resources/Python.app/
↪Contents/MacOS/Python', 'sys_platform': 'darwin'}
/var/folders/jg/jgfZ12ZXHwGSFKD85xLpLk+++TI/-Tmp-/tmpGPiD3E/pylib/nibabel/__
↪init__.pyc
{'sys_version': '2.6.6 (r266:84374, Aug 31 2010, 11:00:51) \n[GCC 4.0.1␣
↪(Apple Inc. build 5493)]', 'commit_source': 'installation', 'np_version':
↪'1.5.0', 'commit_hash': '25b4125', 'pkg_path': '/var/folders/jg/
↪jgfZ12ZXHwGSFKD85xLpLk+++TI/-Tmp-/tmpGPiD3E/pylib/nibabel', 'sys_executable
↪': '/Library/Frameworks/Python.framework/Versions/2.6/Resources/Python.app/
↪Contents/MacOS/Python', 'sys_platform': 'darwin'}
```

(continues on next page)

(continued from previous page)

```
/Users/mb312/dev_trees/nibabel/nibabel/__init__.pyc
{'sys_version': '2.6.6 (r266:84374, Aug 31 2010, 11:00:51) \n[GCC 4.0.1␣
↪(Apple Inc. build 5493)]', 'commit_source': 'repository', 'np_version': '1.
↪5.0', 'commit_hash': '25b4125', 'pkg_path': '/Users/mb312/dev_trees/nibabel/
↪nibabel', 'sys_executable': '/Library/Frameworks/Python.framework/Versions/
↪2.6/Resources/Python.app/Contents/MacOS/Python', 'sys_platform': 'darwin'}
```

  - Check the `setup.py` file is picking up all the library code and scripts, with:

```
make check-files
```

    Look for output at the end about missed files, such as:

```
Missed script files:  /Users/mb312/dev_trees/nibabel/bin/nib-dicomfs, /Users/
↪mb312/dev_trees/nibabel/bin/nifti1_diagnose.py
```

    Fix `setup.py` to carry across any files that should be in the distribution.

  - Check the documentation doctests:

```
make -C doc doctest
```

    This should also be tested by nibabel on travis.

  - Check everything compiles without syntax errors:

```
python -m compileall .
```

  - Check that nibabel correctly generates a source distribution:

```
make source-release
```

- Edit `nibabel/info.py` to set `_version_extra` to `''`; commit;
- You may have virtualenvs for different Python versions. Check the tests pass for different configurations. The long-hand way looks like this:

```
workon python26
make distclean
make sdist-tests
deactivate
```

  etc for the different virtualenvs;

- Check on different platforms, particularly windows and PPC. Look at the nipy buildbot automated test runs for this;
- Force build of your release candidate branch with the slow and big-memory tests on the `zibi` buildslave:

```
try_branch.py nibabel-py2.7-osx-10.10
```

  Check the build web-page for errors:

  - https://nipy.bic.berkeley.edu/builders/nibabel-py2.7-osx-10.10

- Force builds of your local branch on the win32 and amd64 binaries on buildbot:

```
try_branch.py nibabel-bdist32-27
try_branch.py nibabel-bdist32-33
try_branch.py nibabel-bdist32-34
try_branch.py nibabel-bdist32-35
try_branch.py nibabel-bdist64-27
```

Check the builds completed without error on their respective web-pages:

- https://nipy.bic.berkeley.edu/builders/nibabel-bdist32-27
- https://nipy.bic.berkeley.edu/builders/nibabel-bdist32-33
- https://nipy.bic.berkeley.edu/builders/nibabel-bdist32-34
- https://nipy.bic.berkeley.edu/builders/nibabel-bdist32-35
- https://nipy.bic.berkeley.edu/builders/nibabel-bdist64-27

- Make sure you have travis-ci building set up for your own repo. Make a new `release-check` (or similar) branch, and push the code in its current state to a branch that will build, e.g:

```
git branch -D release-check # in case branch already exists
git co -b release-check
# You might need the --force flag here
git push your-github-user release-check -u
```

- 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:

```
[distutils]
index-servers =
    pypi
    warehouse

[pypi]
username:your.pypi.username
password:your-password

[warehouse]
repository: https://upload.pypi.io/legacy/
username:your.pypi.username
password:your-password
```

- Clean:

```
make distclean
# Check no files outside version control that you want to keep
git status
# Nuke
git clean -fxd
```

- When ready:

```
python setup.py register
python setup.py sdist --formats=gztar,zip
# -s flag to sign the release
twine upload -r warehouse -s dist/nibabel*
```

- Tag the release with signed tag of form `2.0.0`:

```
git tag -s 2.0.0
```

- Push the tag and any other changes to trunk with:

```
git push origin 2.0.0
git push
```

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

```
make upload-html
```

- Finally (for the release uploads) upload the Windows binaries you built with `try_branch.py` above;
- Set up maintenance / development branches

  If this is this is a full release you need to set up two branches, one for further substantial development (often called 'trunk') and another for maintenance releases.

  - Branch to maintenance:

    ```
    git co -b maint/2.0.x
    ```

    Set `_version_extra` back to `.dev` and bump `_version_micro` by 1. Thus the maintenance series will have version numbers like - say - '2.0.1.dev' until the next maintenance release - say '2.0.1'. Commit. Don't forget to push upstream with something like:

    ```
    git push upstream-remote maint/2.0.x --set-upstream
    ```

  - Start next development series:

    ```
    git co main-master
    ```

    then restore `.dev` to `_version_extra`, and bump `_version_minor` by 1. Thus the development series ('trunk') will have a version number here of '2.1.0.dev' and the next full release will be '2.1.0'.

    Next merge the maintenance branch with the "ours" strategy. This just labels the maintenance *info.py* edits as seen but discarded, so we can merge from maintenance in future without getting spurious merge conflicts:

    ```
    git merge -s ours maint/2.0.x
    ```

  If this is just a maintenance release from `maint/2.0.x` or similar, just tag and set the version number to - say - `2.0.2.dev`.

- Push the main branch:

```
git push upstream-remote main-master
```

- Make next development release tag

  After each release the master branch should be tagged with an annotated (or/and signed) tag, naming the intended next version, plus an 'upstream/' prefix and 'dev' suffix. For example 'upstream/1.0.0.dev' means "development start for upcoming version 1.0.0.

  This tag is used in the Makefile rules to create development snapshot releases to create proper versions for those. The version derives its name from the last available annotated tag, the number of commits since that, and an abbreviated SHA1. See the docs of `git describe` for more info.

  Please take a look at the Makefile rules `devel-src`, `devel-dsc` and `orig-src`.

- Go to: https://github.com/nipy/nibabel/tags and select the new tag, to fill in the release notes. Copy the relevant part of the Changelog into the release notes. Click on "Publish release". This will cause Zenodo to generate a new release "upload", including a DOI. After a few minutes, go to https://zenodo.org/deposit and click on the new release upload. Click on the "View" button and click on the DOI badge at the right to display the text for adding a DOI badge in various formats. Copy the DOI Markdown text. The markdown will look something like this:

```
[![DOI](https://zenodo.org/badge/doi/10.5281/zenodo.60847.svg)](https://doi.org/
↪10.5281/zenodo.60847)
```

  Go back to the Github release page for this release, click "Edit release". and copy the DOI into the release notes. Click "Update release".

  See: https://guides.github.com/activities/citable-code

- Announce to the mailing lists.

10.3.11 Advanced Testing

Setup

Before running advanced tests, please update all submodules of nibabel, by running `git submodule update --init`

Long-running tests

Long-running tests are not enabled by default, and can be resource-intensive. To run these tests:

- Set environment variable `NIPY_EXTRA_TESTS=slow`;
- Run `pytest nibabel`.

Note that some tests may require a machine with >4GB of RAM.

10.4 DICOM concepts and implementations

Contents:

10.4.1 DICOM information

DICOM is a large and sometimes confusing imaging data format.

In the other pages in this series we try and document our understanding of various aspects of DICOM relevant to converting to formats such as NIfTI .

There are a large number of DICOM image conversion programs already, partly because it is a complicated format with features that vary from manufacturer to manufacturer.

We use the excellent PyDICOM as our back-end for reading DICOM.

Here is a selected list of other tools and relevant resources:

- Grassroots DICOM : GDCM . It is C++ code wrapped with swig and so callable from Python. ITK apparently uses it for DICOM conversion. BSD license.

- dcm2nii - a BSD licensed converter by Chris Rorden. As usual, Chris has done an excellent job of documentation, and it is well battle-tested. There's a nice set of example data to test against and a list of other DICOM software. The MRIcron install page points to the source code. Chris has also put effort into extracting diffusion parameters from the DICOM images.
- SPM8 - SPM has a stable and robust general DICOM conversion tool implemented in the `spm_dicom_convert.m` and `spm_dicom_headers.m` scripts. The conversions don't try to get the diffusion parameters. The code is particularly useful because it has been well-tested and is written in Matlab - and so is relatively easy to read. GPL license. We've described some of the algorithms that SPM uses for DICOM conversion in *SPM DICOM conversion*.
- DICOM2Nrrd: a command line converter to convert DICOM images to Nrrd format. You can call the command from within the Slicer GUI. It does have algorithms for getting diffusion information from the DICOM headers, and has been tested with Philips, GE and Siemens data. It's not clear whether it yet supports the *Siemens mosaic format*. BSD style license.
- The famous Philips cookbook: https://www.archive.org/details/DicomCookbook
- http://dicom.online.fr/fr/dicomlinks.htm

10.4.2 Sample images

- http://www.barre.nom.fr/medical/samples/
- http://pubimage.hcuge.ch:8080/
- Via links from the dcm2nii page.

10.4.3 Defining the DICOM orientation

DICOM patient coordinate system

First we define the standard DICOM patient-based coordinate system. This is what DICOM means by x, y and z axes in its orientation specification. From section C.7.6.2.1.1 of the DICOM object definitions (2009):

> If Anatomical Orientation Type (0010,2210) is absent or has a value of BIPED, the x-axis is increasing to the left hand side of the patient. The y-axis is increasing to the posterior side of the patient. The z-axis is increasing toward the head of the patient.

(we'll ignore the quadupeds for now).

In a way it's funny to call this the 'patient-based' coordinate system. 'Doctor-based coordinate system' is a better name. Think of a doctor looking at the patient from the foot of the scanner bed. Imagine the doctor's right hand held in front of her like Spiderman about to shoot a web, with her palm towards the patient, defining a right-handed coordinate system. Her thumb points to her right (the patient's left), her index finger points down, and the middle finger points at the patient.

DICOM pixel data

C.7.6.3.1.4 - Pixel Data Pixel Data (7FE0,0010) for this image. The order of pixels sent for each image plane is left to right, top to bottom, i.e., the upper left pixel (labeled 1,1) is sent first followed by the remainder of row 1, followed by the first pixel of row 2 (labeled 2,1) then the remainder of row 2 and so on.

The resulting pixel array then has size ('Rows', 'Columns'), with row-major storage (rows first, then columns). We'll call this the DICOM *pixel array*.

Pixel spacing

Section 10.7.1.3: Pixel Spacing The first value is the row spacing in mm, that is the spacing between the centers of adjacent rows, or vertical spacing. The second value is the column spacing in mm, that is the spacing between the centers of adjacent columns, or horizontal spacing.

DICOM voxel to patient coordinate system mapping

See:

- http://www.dclunie.com/medical-image-faq/html/part2.html;
- this dicom mailing list post;

See wikipedia direction cosine for a definition of direction cosines.

From section C.7.6.2.1.1 of the DICOM object definitions (2009):

> The Image Position (0020,0032) specifies the x, y, and z coordinates of the upper left hand corner of the image; it is the center of the first voxel transmitted. Image Orientation (0020,0037) specifies the direction cosines of the first row and the first column with respect to the patient. These Attributes shall be provide as a pair. Row value for the x, y, and z axes respectively followed by the Column value for the x, y, and z axes respectively.

From Section C.7.6.1.1.1 we see that the 'positive row axis' is left to right, and is the direction of the rows, given by the direction of last pixel in the first row from the first pixel in that row. Similarly the 'positive column axis' is top to bottom and is the direction of the columns, given by the direction of the last pixel in the first column from the first pixel in that column.

Let's rephrase: the first three values of 'Image Orientation Patient' are the direction cosine for the 'positive row axis'. That is, they express the direction change in (x, y, z), in the DICOM patient coordinate system (DPCS), as you move along the row. That is, as you move from one column to the next. That is, as the *column* array index changes. Similarly, the second triplet of values of 'Image Orientation Patient' (`img_ornt_pat[3:]` in Python), are the direction cosine for the 'positive column axis', and express the direction you move, in the DPCS, as you move from row to row, and therefore as the *row* index changes.

Further down section C.7.6.2.1.1 (RCS below is the *reference coordinate system* - see DICOM object definitions section 3.17.1):

> The Image Plane Attributes, in conjunction with the Pixel Spacing Attribute, describe the position and orientation of the image slices relative to the patient-based coordinate system. In each image frame the Image Position (Patient) (0020,0032) specifies the origin of the image with respect to the patient-based coordinate system. RCS and the Image Orientation (Patient) (0020,0037) attribute values specify the orientation of the image frame rows and columns. The mapping of pixel location (i, j) to the RCS is

calculated as follows:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} X_x\Delta i & Y_x\Delta j & 0 & S_x \\ X_y\Delta i & Y_y\Delta j & 0 & S_y \\ X_z\Delta i & Y_z\Delta j & 0 & S_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 0 \\ 1 \end{bmatrix} = M \begin{bmatrix} i \\ j \\ 0 \\ 1 \end{bmatrix}$$

Where:

1. P_{xyz} : The coordinates of the voxel (i,j) in the frame's image plane in units of mm.
2. S_{xyz} : The three values of the Image Position (Patient) (0020,0032) attributes. It is the location in mm from the origin of the RCS.
3. X_{xyz} : The values from the row (X) direction cosine of the Image Orientation (Patient) (0020,0037) attribute.
4. Y_{xyz} : The values from the column (Y) direction cosine of the Image Orientation (Patient) (0020,0037) attribute.
5. i : Column index to the image plane. The first column is index zero.
6. Δi: Column pixel resolution of the Pixel Spacing (0028,0030) attribute in units of mm.
7. j : Row index to the image plane. The first row index is zero.
8. Δj - Row pixel resolution of the Pixel Spacing (0028,0030) attribute in units of mm.

(i, j), columns, rows in DICOM

We stop to ask ourselves, what does DICOM mean by voxel (i, j)?

Isn't that obvious? Oh dear, no it isn't. See the *DICOM voxel to patient coordinate system mapping* formula above. In particular, you'll see:

- i : Column index to the image plane. The first column is index zero.
- j : Row index to the image plane. The first row index is zero.

That is, if we have the *DICOM pixel data* as defined above, and we call that `pixel_array`, then voxel (i, j) in the notation above is given by `pixel_array[j, i]`.

What does this mean? It means that, if we want to apply the formula above to array indices in `pixel_array`, we first have to apply a column / row flip to the indices. Say M_{pixar} (sorry) is the affine to go from array indices in `pixel_array` to mm in the DPCS. Then, given M above:

$$M_{pixar} = M \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

DICOM affines again

The *(i, j), columns, rows in DICOM* is rather confusing, so we're going to rephrase the affine mapping; we'll use r for the row index (instead of j above), and c for the column index (instead of i).

Next we define a flipped version of 'ImageOrientationPatient', F, that has flipped columns. Thus if the vector of 6 values in 'ImageOrientationPatient' are $(i_1..i_6)$, then:

$$F = \begin{bmatrix} i_4 & i_1 \\ i_5 & i_2 \\ i_6 & i_3 \end{bmatrix}$$

Now the first column of F contains what the DICOM docs call the 'column (Y) direction cosine', and second column contains the 'row (X) direction cosine'. We prefer to think of these as (respectively) the row index direction cosine and the column index direction cosine.

Now we can rephrase the DICOM affine mapping with:

DICOM affine formula

$$\begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} F_{11}\Delta r & F_{12}\Delta c & 0 & S_x \\ F_{21}\Delta r & F_{22}\Delta c & 0 & S_y \\ F_{31}\Delta r & F_{32}\Delta c & 0 & S_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r \\ c \\ 0 \\ 1 \end{bmatrix} = A \begin{bmatrix} r \\ c \\ 0 \\ 1 \end{bmatrix}$$

Where:

- P_{xyz} : The coordinates of the voxel (c, r) in the frame's image plane in units of mm.
- S_{xyz} : The three values of the Image Position (Patient) (0020,0032) attributes. It is the location in mm from the origin of the RCS.
- $F_{:,1}$: The values from the column (Y) direction cosine of the Image Orientation (Patient) (0020,0037) attribute - see above.
- $F_{:,2}$: The values from the row (X) direction cosine of the Image Orientation (Patient) (0020,0037) attribute - see above.
- r : Row index to the image plane. The first row index is zero.
- Δr - Row pixel resolution of the Pixel Spacing (0028,0030) attribute in units of mm.
- c : Column index to the image plane. The first column is index zero.
- Δc: Column pixel resolution of the Pixel Spacing (0028,0030) attribute in units of mm.

For later convenience we also define values useful for 3D volumes:

- s : Slice index to the slice plane. The first slice index is zero.
- Δs - Spacing in mm between slices.

Getting a 3D affine from a DICOM slice or list of slices

Let us say, we have a single DICOM file, or a list of DICOM files that we believe to be a set of slices from the same volume. We'll call the first the *single slice* case, and the second, *multi slice*.

In the *multi slice* case, we can assume that the 'ImageOrientationPatient' field is the same for all the slices.

We want to get the affine transformation matrix A that maps from voxel coordinates in the DICOM file(s), to mm in the *DICOM patient coordinate system*.

By voxel coordinates, we mean coordinates of form (r, c, s) - the row, column and slice indices - as for the *DICOM affine formula*.

In the single slice case, the voxel coordinates are just the indices into the pixel array, with the third (slice) coordinate always being 0.

In the multi-slice case, we have arranged the slices in ascending or descending order, where slice numbers range from 0 to $N - 1$ - where N is the number of slices - and the slice coordinate is a number on this scale.

We know, from *DICOM affine formula*, that the first, second and fourth columns in A are given directly by the (flipped) 'ImageOrientationPatient', 'PixelSpacing' and 'ImagePositionPatient' field of the first (or only) slice.

Our job then is to fill the first three rows of the third column of A. Let's call this the vector $\mathbf{k}$ with values k_1, k_2, k_3.

DICOM affine Definitions

See also the definitions in *DICOM affine formula*. In addition

- T^1 is the 3 element vector of the 'ImagePositionPatient' field of the first header in the list of headers for this volume.
- T^N is the 'ImagePositionPatient' vector for the last header in the list for this volume, if there is more than one header in the volume.
- vector $\mathbf{n} = (n_1, n_2, n_3)$ is the result of taking the cross product of the two columns of F from *DICOM affine formula*.

Derivations

For the single slice case we just fill $\mathbf{k}$ with $\mathbf{n} \cdot \Delta s$ - on the basis that the Z dimension should be right-handed orthogonal to the X and Y directions.

For the multi-slice case, we can fill in $\mathbf{k}$ by using the information from T^N, because T^N is the translation needed to take the first voxel in the last (slice index = $N - 1$) slice to mm space. So:

$$\begin{pmatrix} T^N \\ 1 \end{pmatrix} = A \begin{pmatrix} 0 \\ 0 \\ N-1 \\ 1 \end{pmatrix}$$

From this it follows that:

$$\left\{ k_1 : \frac{T_1^N - T_1^1}{N-1}, \quad k_2 : \frac{T_2^N - T_2^1}{N-1}, \quad k_3 : \frac{T_3^N - T_3^1}{N-1} \right\}$$

and therefore:

3D affine formulae

$$A_{multi} = \begin{pmatrix} F_{11}\Delta r & F_{12}\Delta c & \frac{T_1^N - T_1^1}{N-1} & T_1^1 \\ F_{21}\Delta r & F_{22}\Delta c & \frac{T_2^N - T_2^1}{N-1} & T_2^1 \\ F_{31}\Delta r & F_{32}\Delta c & \frac{T_3^N - T_3^1}{N-1} & T_3^1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_{single} = \begin{pmatrix} F_{11}\Delta r & F_{12}\Delta c & \Delta s n_1 & T_1^1 \\ F_{21}\Delta r & F_{22}\Delta c & \Delta s n_2 & T_2^1 \\ F_{31}\Delta r & F_{32}\Delta c & \Delta s n_3 & T_3^1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

See `derivations/spm_dicom_orient.py` for the derivations and some explanations.

For a single slice $N = 1$ the affine matrix is A_{single}. In this case, the slice spacing Δs may be obtained by the Spacing Between Slices (0018,0088) attribute in units of mm, if it exists.

Working out the Z coordinates for a set of slices

We may have the problem (see e.g. *Sorting files into volumes*) of trying to sort a set of slices into anatomical order. For this we want to use the orientation information to tell us where the slices are in space, and therefore, what order they should have.

To do this sorting, we need something that is proportional, plus a constant, to the voxel coordinate for the slice (the value for the slice index).

Our DICOM might have the 'SliceLocation' field (0020,1041). 'SliceLocation' seems to be proportianal to slice location, at least for some GE and Philips DICOMs I was looking at. But, there is a more reliable way (that doesn't depend on this field), and uses only the very standard 'ImageOrientationPatient' and 'ImagePositionPatient' fields.

Consider the case where we have a set of slices, of unknown order, from the same volume.

Now let us say we have one of these slices - slice i. We have the affine for this slice from the calculations above, for a single slice (A_{single}).

Now let's say we have another slice j from the same volume. It will have the same affine, except that the 'ImagePositionPatient' field will change to reflect the different position of this slice in space. Let us say that there a translation of d slices between i and j. If A_i (A for slice i) is A_{single} then A_j for j is given by:

$$A_j = A_{single} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and 'ImagePositionPatient' for j is:

$$T^j = \begin{pmatrix} T^1_1 + \Delta s d n_1 \\ T^1_2 + \Delta s d n_2 \\ T^1_3 + \Delta s d n_3 \end{pmatrix}$$

Remember that the third column of A gives the vector resulting from a unit change in the slice voxel coordinate. So, the 'ImagePositionPatient' of slice - say slice j - can be thought of the addition of two vectors $T^j = \mathbf{a} + \mathbf{b}$, where $\mathbf{a}$ is the position of the first voxel in some slice (here slice 1, therefore $\mathbf{a} = T^1$) and $\mathbf{b}$ is d times the third colum of A. Obviously d can be negative or positive. This leads to various ways of recovering something that is proportional to d plus a constant. The algorithm suggested in this ITK post on ordering slices - and the one used by SPM - is to take the inner product of T^j with the unit vector component of third column of A_j - in the descriptions here, this is the vector $\mathbf{n}$:

$$T^j \cdot \mathbf{c} = \left(T^1_1 n_1 + T^1_2 n_2 + T^1_3 n_3 + \Delta s d n_1^2 + \Delta s d n_2^2 + \Delta s d n_3^2 \right)$$

This is the distance of 'ImagePositionPatient' along the slice direction cosine.

The unknown T^1 terms pool into a constant, and the operation has the neat feature that, because the n^2_{123} terms, by definition, sum to 1, the whole can be expressed as $\lambda + \Delta s d$ - i.e. it is equal to the slice voxel size (Δs) multiplied by d, plus a constant.

Again, see `derivations/spm_dicom_orient.py` for the derivations.

10.4.4 DICOM fields

In which we pick out some interesting fields in the DICOM header.

We're getting the information mainly from the standard DICOM object definitions

We won't talk about the orientation, patient position-type fields here because we've covered those somewhat in *DICOM voxel to patient coordinate system mapping*.

Fields for ordering DICOM files into images

You'll see some discussion of this in *SPM DICOM conversion*.

Section 7.3.1: general series module

- Modality (0008,0060) - Type of equipment that originally acquired the data used to create the images in this Series. See C.7.3.1.1.1 for Defined Terms.
- Series Instance UID (0020,000E) - Unique identifier of the Series.
- Series Number (0020,0011) - A number that identifies this Series.
- Series Time (0008,0031) - Time the Series started.

Section C.7.6.1:

- Instance Number (0020,0013) - A number that identifies this image.
- Acquisition Number (0020,0012) - A number identifying the single continuous gathering of data over a period of time that resulted in this image.
- Acquisition Time (0008,0032) - The time the acquisition of data that resulted in this image started

Section C.7.6.2.1.2:

Slice Location (0020,1041) is defined as the relative position of the image plane expressed in mm. This information is relative to an unspecified implementation specific reference point.

Section C.8.3.1 MR Image Module

- Slice Thickness (0018,0050) - Nominal reconstructed slice thickness, in mm.

Section C.8.3.1 MR Image Module

- Spacing Between Slices (0018,0088) - Spacing between slices, in mm. The spacing is measured from the center-tocenter of each slice.
- Temporal Position Identifier (0020,0100) - Temporal order of a dynamic or functional set of Images.
- Number of Temporal Positions (0020,0105) - Total number of temporal positions prescribed.
- Temporal Resolution (0020,0110) - Time delta between Images in a dynamic or functional set of images

Multi-frame images

An image for which the pixel data is a continuous stream of sequential frames.

Section C.7.6.6: Multi-Frame Module

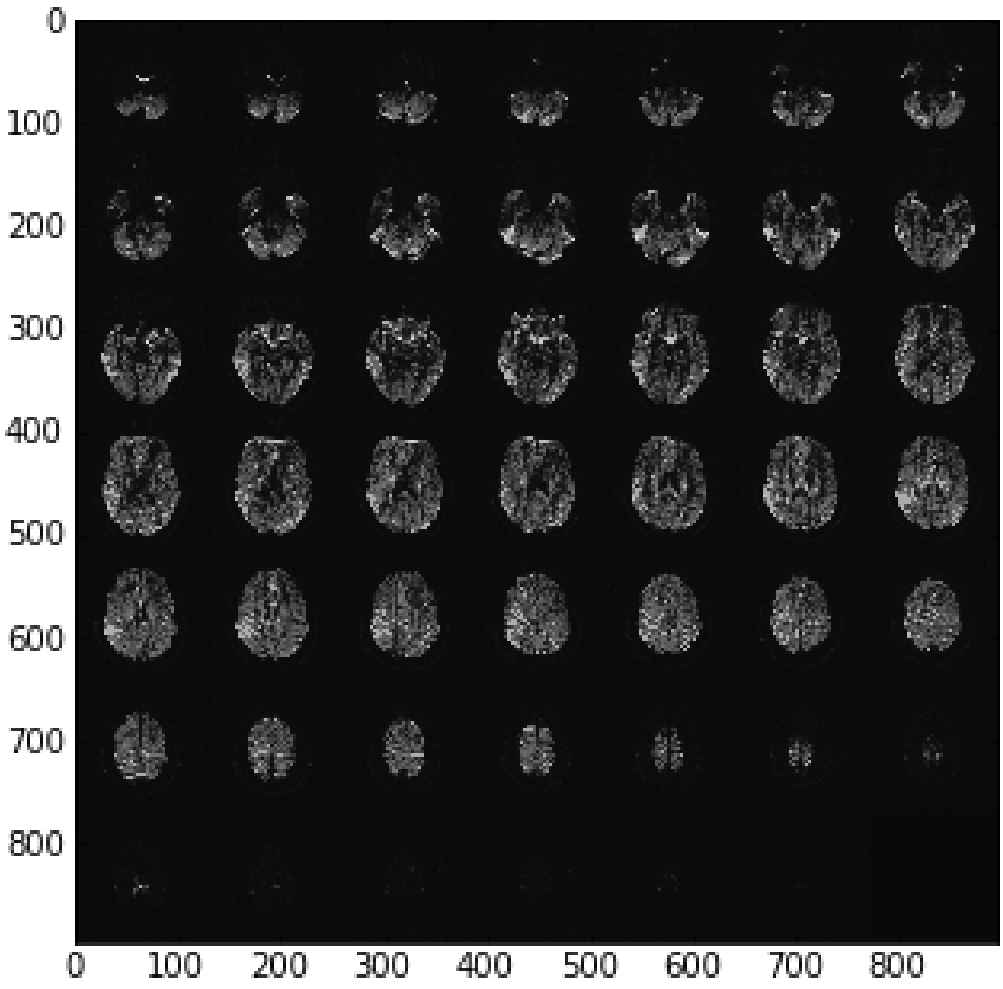
- Number of Frames (0028,0008) - Number of frames in a Multi-frame Image.
- Frame Increment Pointer (0028,0009) - Contains the Data Element Tag of the attribute that is used as the frame increment in Multi-frame pixel data.

10.4.5 Siemens mosaic format

Siemens mosaic format is a way of storing a 3D image in a DICOM image file. The simplest DICOM images only knows how to store 2D files. For example, a 3D image in DICOM is usually stored as a series of 2D slices, each slices as a separate DICOM image. . Mosaic format stores the 3D image slices as a 2D grid - or mosaic.

For example here are the pixel data as loaded directly from a DICOM image with something like:

```
import matplotlib.pylab as plt
import dicom
dcm_data = dicom.read_file('my_file.dcm')
plt.imshow(dcm_data.pixel_array)
```



Getting the slices from the mosaic

The apparent image in the DICOM file is a 2D array that consists of blocks, that are the output 2D slices. Let's call the original array the *slab*, and the contained slices *slices*. The slices are of pixel dimension `n_slice_rows` x `n_slice_cols`. The slab is of pixel dimension `n_slab_rows` x `n_slab_cols`. Because the arrangement of blocks in the slab is defined as being square, the number of blocks per slab row and slab column is the same. Let `n_blocks` be the number of blocks contained in the slab. There is also `n_slices` - the number of slices actually collected, some number <= `n_blocks`. We have the value `n_slices` from the 'NumberOfImagesInMosaic' field of the Siemens private (CSA) header. `n_row_blocks` and `n_col_blocks` are therefore given by `ceil(sqrt(n_slices))`, and `n_blocks` is `n_row_blocks ** 2`. Also `n_slice_rows == n_slab_rows / n_row_blocks`, etc. Using these numbers we can therefore reconstruct the slices from the 2D DICOM pixel array.

DICOM orientation for mosaic

See *DICOM patient coordinate system* and *DICOM voxel to patient coordinate system mapping*. We want a 4 x 4 affine A that will take us from (transposed) voxel coordinates in the DICOM image to mm in the *DICOM patient coordinate system*. See *(i, j), columns, rows in DICOM* for what we mean by transposed voxel coordinates.

We can think of the affine A as the (3,3) component, RS, and a (3,1) translation vector $\mathbf{t}$. RS can in turn be thought of as the dot product of a (3,3) rotation matrix R and a scaling matrix S, where `S = diag(s)` and $\mathbf{s}$ is a (3,) vector of voxel sizes. $\mathbf{t}$ is a (3,1) translation vector, defining the coordinate in millimeters of the first voxel in the voxel volume (the voxel given by `voxel_array[0,0,0]`).

In the case of the mosaic, we have the first two columns of R from the F - the left/right flipped version of the `ImageOrientationPatient` DICOM field described in *DICOM affines again*. To make a full rotation matrix, we can generate the last column from the cross product of the first two. However, Siemens defines, in its private *CSA header*, a `SliceNormalVector` which gives the third column, but possibly with a z flip, so that R is orthogonal, but not a rotation matrix (it has a determinant of < 0).

The first two values of $\mathbf{s}$ (s_1, s_2) are given by the `PixelSpacing` field. We get s_3 (the slice scaling value) from `SpacingBetweenSlices`.

The *SPM DICOM conversion* code has a comment saying that mosaic DICOM images have an incorrect `ImagePositionPatient` field. The `ImagePositionPatient` field usually gives the $\mathbf{t}$ vector. The comments imply that Siemens has derived `ImagePositionPatient` from the (correct) position of the center of the first slice (once the mosaic has been unpacked), but has then adjusted the vector to point to the top left voxel, where the slice size used for this adjustment is the size of the mosaic, before it has been unpacked. Let's call the correct position in millimeters of the center of the first slice $\mathbf{c} = [c_x, c_y, c_z]$. We have the derived RS matrix from the calculations above. The unpacked (eventual, real) slice dimensions are (rd_{rows}, rd_{cols}) and the mosaic dimensions are (md_{rows}, md_{cols}). The `ImagePositionPatient` vector $\mathbf{i}$ resulted from:

$$\mathbf{i} = \mathbf{c} + RS \begin{bmatrix} -(md_{rows} - 1)/2 \\ -(md_{cols} - 1)/2 \\ 0 \end{bmatrix}$$

To correct the faulty translation, we reverse it, and add the correct translation for the unpacked slice size (rd_{rows}, rd_{cols}), giving the true image position $\mathbf{t}$:

$$\mathbf{t} = \mathbf{i} - (RS \begin{bmatrix} -(md_{rows} - 1)/2 \\ -(md_{cols} - 1)/2 \\ 0 \end{bmatrix}) + (RS \begin{bmatrix} -(rd_{rows} - 1)/2 \\ -(rd_{cols} - 1)/2 \\ 0 \end{bmatrix})$$

Because of the final zero in the voxel translations, this simplifies to:

$$\mathbf{t} = \mathbf{i} + Q \begin{bmatrix} (md_{rows} - rd_{rowss})/2 \\ (md_{cols} - rd_{cols})/2 \end{bmatrix}$$

where:

$$Q = \begin{bmatrix} rs_{11} & rs_{12} \\ rs_{21} & rs_{22} \\ rs_{31} & rs_{32} \end{bmatrix}$$

Data scaling

SPM gets the DICOM scaling, offset for the image ('RescaleSlope', 'RescaleIntercept'). It writes these scalings into the nifti header. Then it writes the raw image data (unscaled) to disk. Obviously these will have the corrent scalings applied when the nifti image is read again.

A comment in the code here says that the data are not scaled by the maximum amount. I assume by this they mean that the DICOM scaling may not be the maximum scaling, whereas the standard SPM image write is, hence the difference, because they are using the DICOM scaling rather then their own. The comment continues by saying that the scaling as applied (the DICOM - not maximum - scaling) can lead to rounding errors but that it will get around some unspecified problems.

10.4.6 Siemens format DICOM with CSA header

Recent Siemens DICOM images have useful information stored in a private header. We'll call this the *CSA header*.

CSA header

See this Siemens Syngo DICOM conformance statement, and a GDCM Siemens header dump.

The CSA header is stored in DICOM private tags. In the images we are looking at, there are several relevant tags:

```
(0029, 1008) [CSA Image Header Type]                OB: 'IMAGE NUM 4 '
(0029, 1009) [CSA Image Header Version]             OB: '20100114'
(0029, 1010) [CSA Image Header Info]                OB: Array of 11560 bytes
(0029, 1018) [CSA Series Header Type]               OB: 'MR'
(0029, 1019) [CSA Series Header Version]            OB: '20100114'
(0029, 1020) [CSA Series Header Info]               OB: Array of 80248 bytes
```

In our case we want to read the 'CSAImageHeaderInfo'.

From the SPM (SPM8) code `spm_dicom_headers.m`

The CSAImageHeaderInfo and the CSA Series Header Info fields are of the same format. The fields can be of two types, CSA1 and CSA2.

Both are always little-endian, whatever the machine endian is.

The CSA2 format begins with the string 'SV10', the CSA1 format does not.

The code below keeps track of the position *within the CSA header stream*. We'll call this `csa_position`. At this point (after reading the 8 bytes of the header), `csa_position == 8`. There's a variable that sets the last byte position in the file that is sensibly still CSA header, and we'll call that `csa_max_pos`.

CSA1

Start header

1. n_tags, uint32, number of tags. Number of tags should apparently be between 1 and 128. If this is not true we just abort and move to `csa_max_pos`.
2. unused, uint32, apparently has value 77

Each tag

1. name : S64, null terminated string 64 bytes
2. vm : int32
3. vr : S4, first 3 characters only
4. syngodt : int32
5. nitems : int32
6. xx : int32 - apparently either 77 or 205

`nitems` gives the number of items in the tag. The items follow directly after the tag.

Each item

1. xx : int32 * 4 . The first of these seems to be the length of the item in bytes, modified as below.

At this point SPM does a check, by calculating the length of this item `item_len` with `xx[0]` - the `nitems` of the *first* read tag. If `item_len` is less than 0 or greater than `csa_max_pos-csa_position` (the remaining number of bytes to read in the whole header) then we break from the item reading loop, setting the value below to ‘’.

Then we calculate `item_len` rounded up to the nearest 4 byte boundary tp get `next_item_pos`.

2. value : uint8, `item_len`.

We set the stream position to `next_item_pos`.

CSA2

Start header

1. hdr_id : S4 == ‘SV10’
2. unused1 : uint8, 4
3. n_tags, uint32, number of tags. Number of tags should apparently be between 1 and 128. If this is not true we just abort and move to `csa_max_pos`.
4. unused2, uint32, apparently has value 77

Each tag

1. name : S64, null terminated string 64 bytes
2. vm : int32
3. vr : S4, first 3 characters only
4. syngodt : int32
5. nitems : int32
6. xx : int32 - apparently either 77 or 205

`nitems` gives the number of items in the tag. The items follow directly after the tag.

Each item

1. xx : int32 * 4 . The first of these seems to be the length of the item in bytes, modified as below.

Now there's a different length check from CSA1. `item_len` is given just by `xx[1]`. If `item_len > csa_max_pos - csa_position` (the remaining bytes in the header), then we just read the remaning bytes in the header (as above) into `value` below, as uint8, move the filepointer to the next 4 byte boundary, and give up reading.

2. value : uint8, `item_len`.

We set the stream position to the next 4 byte boundary.

10.4.7 SPM DICOM conversion

These are some notes on the algorithms that SPM uses to convert from DICOM to nifti. There are other notes in *Siemens mosaic format*.

The relevant SPM files are `spm_dicom_headers.m`, `spm_dicom_dict.mat` and `spm_dicom_convert.m`. These notes refer the version in SPM8, as of around January 2010.

`spm_dicom_dict.mat`

This is obviously a Matlab `.mat` file. It contains variables `group` and `element`, and `values`, where `values` is a struct array, one element per (group, element) pair, with fields `name` and `vr` (the last a cell array).

`spm_dicom_headers.m`

Reads the given DICOM files into a struct. It looks like this was written by John Ahsburner (JA). Relevant fixes are:

File opening

When opening the DICOM file, SPM (subfunction `readdicomfile`)

1. opens as little endian
2. reads 4 characters starting at pos 128
3. checks if these are `DICM`; if so then continues file read; otherwise, tests to see if this is what SPM calls *truncated DICOM file format* - lacking 128 byte lead in and `DICM` string:
    1. Seeks to beginning of file
    2. Reads two unsigned short values into `group` and `tag`
    3. If the (`group`, `element`) pair exist in `spm_dicom_dict.mat`, then set file pointer to 0 and continue read with `read_dicom` subfunction..
    4. If `group` == 8 and `element` == 0, this is apparently the signature for a 'GE Twin+excite' for which JA notes there is no documentation; set file pointer to 0 and continue read with `read_dicom` subfunction.
    5. Otherwise - crash out with error saying that this is not DICOM file.

tag read for Philips Integra

The `read_dicom` subfunction reads a tag, then has a loop during which the tag is processed (by setting values into the return structure). At the end of the loop, it reads the next tag. The loop breaks when the current tag is empty, or is the item delimitation tag (group=FFFE, element=E00D).

After it has broken out of the loop, if the last tag was (FFFE, E00D) (item delimitation tag), and the tag length was not 0, then SPM sets the file pointer back by 4 bytes from the current position. JA comments that he didn't find that in the standard, but that it seemed to be needed for the Philips Integra.

Tag length

Tag lengths as read in `read_tag` subfunction. If current format is explicit (as in 'explicit little endian'):

1. For VR of x00x00, then group, element must be (FFFE, E00D) (item delimitation tag). JA comments that GE 'ImageDelimitationItem' has no VR, just 4 0 bytes. In this case the tag length is zero, and we read another two bytes ahead.

There's a check for not-even tag length. If not even:

1. 4294967295 appears to be OK - and decoded as Inf for tag length.
2. 13 appears to mean 10 and is reset to be 10
3. Any other odd number is not valid and gives a tag length of 0

SQ VR type (Sequnce of items type)

tag length of 13 set to tag length 10.

spm_dicom_convert.m

Written by John Ashburner and Jesper Andersson.

File categorization

SPM makes a special case of Siemens 'spectroscopy images'. These are images that have 'SOPClassUID' == '1.3.12.2.1107.5.9.1' and the private tag of (29, 1210); for these it pulls out the affine, and writes a volume of ones corresponding to the acquisition planes.

For images that are not spectroscopy:

- Discards images that do not have any of ('MR', 'PT', 'CT') in 'Modality' field.
- Discards images lacking any of 'StartOfPixelData', 'SamplesperPixel', 'Rows', 'Columns', 'BitsAllocated', 'BitsStored', 'HighBit', 'PixelRespresentation'
- Discards images lacking any of 'PixelSpacing', 'ImagePositionPatient', 'ImageOrientationPatient' - presumably on the basis that SPM cannot reconstruct the affine.
- Fields 'SeriesNumber', 'AcquisitionNumber' and 'InstanceNumber' are set to 1 if absent.

Next SPM distinguishes between *Siemens mosaic format* and standard DICOM.

Mosaic images are those with the Siemens private tag:

```
(0029, 1009) [CSA Image Header Version]          OB: '20100114'
```

and a readable CSA header (see *Siemens mosaic format*), and with non-empty fields from that header of 'AcquisitionMatrixText', 'NumberOfImagesInMosaic', and with non-zero 'NumberOfImagesInMosaic'. The rest are standard DICOM.

For converting mosaic format, see *Siemens mosaic format*. The rest of this page refers to standard (slice by slice) DICOMs.

Sorting files into volumes

First pass

Take first header, put as start of first volume. For each subsequent header:

1. Get `ICE_Dims` if present. Look for Siemens 'CSAImageHeaderInfo', check it has a 'name' field, then pull dimensions out of 'ICE_Dims' field in form of 9 integers separated by '_', where 'X' in this string replaced by '-1' - giving 'ICE1'

Then, for each currently identified volume:

1. If we have ICE1 above, and we do have 'CSAIMageHeaderInfo', with a 'name', in the first header in this volume, then extract ICE dims in the same way as above, for the first header in this volume, and check whether all but ICE1[6:8] are the same as ICE2. Set flag that all ICE dims are identical for this volume. Set this flag to True if we did not have ICE1 or CSA information.
2. Match the current header to the current volume iff the following match:

1. SeriesNumber
2. Rows
3. Columns
4. ImageOrientationPatient (to tolerance of sum squared difference 1e-4)
5. PixelSpacing (to tolerance of sum squared difference 1e-4)
6. ICE dims as defined above
7. ImageType (iff imagetype exists in both)
8. SequenceName (iff sequencename exists in both)
9. SeriesInstanceUID (iff exists in both)
10. EchoNumbers (iff exists in both)

3. If the current header matches the current volume, insert it there, otherwise make a new volume for this header

Second pass

We now have a list of volumes, where each volume is a list of headers that may match.

For each volume:

1. Estimate the z direction cosine by (effectively) finding the cross product of the x and y direction cosines contained in 'ImageOrientationPatient' - call this `z_dir_cos`
2. For each header in this volume, get the z coordinate by taking the dot product of the 'ImagePositionPatient' vector and `z_dir_cos` (see *Working out the Z coordinates for a set of slices*).
3. Sort the headers according to this estimated z coordinate.
4. If this volume is more than one slice, and there are any slices with the same z coordinate (as defined above), run the *Possible volume resort* on this volume - on the basis that it may have caught more than one volume-worth of slices. Return one or more volume's worth of lists.

Final check

For each volume, recalculate z coordinate as above. Calculate the z gaps. Subtract the mean of the z gaps from all z gaps. If the average of the (gap-mean(gap)) is greater than 1e-4, then print a warning that there are missing DICOM files.

Possible volume resort

This step happens if there were volumes with slices having the same z coordinate in the *Second pass* step above. The resort is on the set of DICOM headers that were in the volume, for which there were slices with identical z coordinates. We'll call the list of headers that the routine is still working on - `work_list`.

1. If there is no 'InstanceNumber' field for the first header in `work_list`, bail out.
2. Print a message about the 'AcquisitionNumber' not changing from volume to volume. This may be a relic from previous code, because this version of SPM does not use the 'AcquisitionNumber' field except for making filenames.
3. Calculate the z coordinate as for *Second pass*, for each DICOM header.

4. Sort the headers by 'InstanceNumber'
5. If any headers have the same 'InstanceNumber', then discard all but the first header with the same number. At this point the remaining headers in `work_list` will have different 'InstanceNumber's, but may have the same z coordinate.
6. Now sort by z coordinate
7. If there are `N` headers, make a `N` length vector of flags `is_processed`, for which all values == False
8. Make an output list of header lists, call it `hdr_vol_out`, set to empty.
9. While there are still any False elements in `is_processed`:
    1. Find first header for which corresponding `is_processed` is False - call this `hdr_to_check`
    2. Collect indices (in `work_list`) of headers which have the same z coordinate as `hdr_to_check`, call this list `z_same_indices`.
    3. Sort `work_list[z_same_indices]` by 'InstanceNumber'
    4. For each index in `z_same_indices` such that `i` indexes the indices, and `zsind` is `z_same_indices[i]`: append header corresponding to `zsind` to `hdr_vol_out[i]`. This assumes that the original `work_list` contained two or more volumes, each with an identical set of z coordinates.
    5. Set corresponding `is_processed` flag to True for all `z_same_indices`.
10. Finally, if the headers in `work_list` have 'InstanceNumber's that cannot be sorted to a sequence ascending in units of 1, or if any of the lists in `hdr_vol_out` have different lengths, emit a warning about missing DICOM files.

Writing DICOM volumes

This means - writing DICOM volumes from standard (slice by slice) DICOM datasets rather than *Siemens mosaic format*.

Making the affine

We need the (4,4) affine A going from voxel (array) coordinates in the DICOM pixel data, to mm coordinates in the *DICOM patient coordinate system*.

This section tries to explain how SPM achieves this, but I don't completely understand their method. See *Getting a 3D affine from a DICOM slice or list of slices* for what I believe to be a simpler explanation.

First define the constants, matrices and vectors as in *DICOM affine Definitions*.

N is the number of slices in the volume.

Then define the following matrices:

$$R = \begin{pmatrix} 1 & a & 1 & 0 \\ 1 & b & 0 & 1 \\ 1 & c & 0 & 0 \\ 1 & d & 0 & 0 \end{pmatrix}$$

$$L = \begin{pmatrix} T_1^1 & e & F_{11}\Delta r & F_{12}\Delta c \\ T_2^1 & f & F_{21}\Delta r & F_{22}\Delta c \\ T_3^1 & g & F_{31}\Delta r & F_{32}\Delta c \\ 1 & h & 0 & 0 \end{pmatrix}$$

For a volume with more than one slice (header), then $a = 1; b = 1, c = N, d = 1$. e, f, g are the values from T^N, and $h == 1$.

For a volume with only one slice (header) $a = 0, b = 0, c = 1, d = 0$ and e, f, g, h are $n_1\Delta s, n_2\Delta s, n_3\Delta s, 0$.

The full transform appears to be $A_{spm} = RL^{-1}$.

Now, SPM, don't forget, is working in terms of Matlab array indexing, which starts at (1,1,1) for a three dimensional array, whereas DICOM expects a (0,0,0) start (see *DICOM affine formula*). In this particular part of the SPM DICOM code, somewhat confusingly, the (0,0,0) to (1,1,1) indexing is dealt with in the A transform, rather than the `analyze_to_dicom` transformation used by SPM in other places. So, the transform A_{spm} goes from (1,1,1) based voxel indices to mm. To get the (0, 0, 0)-based transform we want, we need to pre-apply the transform to take 0-based voxel indices to 1-based voxel indices:

$$A = RL^{-1} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This formula with the definitions above result in the single and multi slice formulae in *3D affine formulae*.

See `derivations/spm_dicom_orient.py` for the derivations and some explanations.

Writing the voxel data

Just apply scaling and offset from 'RescaleSlope' and 'RescaleIntercept' for each slice and write volume.

10.4.8 DICOM Tags in the NIfTI Header

NIfTI images include an extended header (see the NIfTI Extensions Standard) to store, amongst others, DICOM tags and attributes. When NiBabel loads a NIfTI file containing DICOM information (a NIfTI extension with `ecode == 2`), it parses it and returns a pydicom dataset as the content of the NIfTI extension. This can be read and written to in order to facilitate communication with software that uses specific DICOM codes found in the NIfTI header.

For example, the commercial PMOD software stores the Frame Start and Duration times of images using the DICOM tags (0055, 1001) and (0055, 1004). Here's an example of an image created in PMOD with those stored times accessed through nibabel.

```
>> import nibabel as nib
>> nim = nib.load('pmod_pet.nii')
>> dcmext = nim.header.extensions[0]
>> dcmext
Nifti1Extension('dicom', '(0054, 1001) Units                                CS: 'Bq/ml'
(0055, 0010) Private Creator                    LO: 'PMOD_1'
(0055, 1001) [Frame Start Times Vector]         FD: [0.0, 30.0, 60.0, ..., 13720.0,␣
↪14320.0]
(0055, 1004) [Frame Durations (ms) Vector]      FD: [30000.0, 30000.0, 30000.0,
↪600000.0, 600000.0]'))
```

Tag	Name	Value
(0054, 1001)	Units	CS: 'Bq/ml'
(0055, 0010)	Private Creator	LO: 'PMOD_1'
(0055, 1001)	[Frame Start Times Vector]	FD: [0.0, 30.0, 60.0, . . . , 13720.0, 14320.0
(0055, 1004)	[Frame Durations (ms) Vector]	FD: [30000.0, 30000.0, 30000.0, . . . , 600000.0, 600000.0

Access each value as you would with pydicom:

```
>> ds = dcmext.get_content()
>> start_times = ds[0x0055, 0x1001].value
>> durations   = ds[0x0055, 0x1004].value
```

Creating a PMOD-compatible header is just as easy:

```
>> nim = nib.load('pet.nii')
>> nim.header.extensions
[]
>> from dicom.dataset import Dataset
>> ds = Dataset()
>> ds.add_new((0x0054,0x1001),'CS','Bq/ml')
>> ds.add_new((0x0055,0x0010),'LO','PMOD_1')
>> ds.add_new((0x0055,0x1001),'FD',[0.,30.,60.,13720.,14320.])
>> ds.add_new((0x0055,0x1004),'FD',[30000.,30000.,30000.,600000.,600000.])
>> dcmext = nib.nifti1.Nifti1DicomExtension(2,ds)  # Use DICOM ecode 2
>> nim.header.extensions.append(dcmext)
>> nib.save(nim,'pet_withdcm.nii')
```

Be careful! Many imaging tools don't maintain information in the extended header, so it's possible [likely] that this information may be lost during routine use. You'll have to keep track, and re-write the information if required.

Optional Dependency Note: If pydicom is not installed, nibabel uses a generic *nibabel.nifti1.Nifti1Extension* header instead of parsing DICOM data.

10.4.9 dcm2nii algorithms

dcm2nii is an open source DICOM to nifti conversion program, written by Chris Rorden, in Delphi (object orientated pascal). It's part of Chris' popular mricron collection of programs. The source appears to be best found on the mricron NITRC site. It's BSD licensed.

These are working notes looking at Chris' algorithms for working with DICOM.

Compiling dcm2nii

Follow the download / install instructions at the http://www.lazarus.freepascal.org/ site. I was on a Mac, and folowed the instructions here: http://wiki.lazarus.freepascal.org/Installing_Lazarus_on_MacOS_X . Default build with version 0.9.28.2 gave an error linking against Carbon, so I needed to download a snapshot of fixed Lazarus 0.9.28.3 from http://www.hu.freepascal.org/lazarus . Open `<mricron>/dcm2nii/dcm2nii.lpi` using the Lazarus GUI. Follow instructions for compiler setup in the mricron `Readme.txt`; in particular I set other compiler options to:

```
-k-macosx_version_min -k10.5
-XR/Developer/SDKs/MacOSX10.5.sdk/
```

Further inspiration for building also came from the `debian/rules` file in Michael Hanke's mricron debian package: http://neuro.debian.net/debian/pool/main/m/mricron/

Some tag modifications

Note - Chris tells me that `dicomfastread.pas` was an attempt to do a fast dicom read that is not yet fully compatible, and that the algorithm used is in fact `dicomcompat.pas`.

Looking in the source file `<mricron>/dcm2nii/dicomfastread.pas`.

Named fields here are as from *DICOM fields*

- If 'MOSAIC' is the last string in 'ImageType', this is a mosaic
- 'DateTime' field is combination of 'StudyDate' and 'StudyTime'; fixes in file `dicomtypes.pas` for different scanner date / time formats.
- AcquisitionNumber read as normal, but then set to 1, if this a mosaic image, as set above.

- If 'EchoNumbers' > 0 and < 16, add 'EchoNumber' * 100 to the 'AcquisitionNumber' - presumably to identify different echos from the same series as being different series.
- If 'ScanningSequence' sequence contains 'RM', add 100 to the 'SeriesNumber' - maybe to differentiate research and not-research scans with the same acquisition number.
- is_4D flag labeling DICOM file as a 4D file:
  - There's a Philips private tag (2001, 1018) - labeled 'Number of Slices MR' by pydicom call this `NS`
  - If `NS>0` and 'NumberofTemporalPositions' > 0, and 'NumberOfFrames' is > 1

Sorting slices into volumes

Looking in the source file `<mricron>/dcm2nii/sortdicom.pas`.

In function `ShellSortDCM`:

Sort compares two dicom images, call them `dcm1` and `dcm2`. Tests are:

1. Are the two images 'repeats' - defined by same 'InstanceNumber' (0020, 0013), and 'AcquisitionNumber' (0020, 0012) and 'SeriesNumber' (0020, 0011) and a combination of 'StudyDate' and 'StudyTime')? Then report an error about files having the same index, flag repeated values.
2. Is `dcm1` less than `dcm2`, defined with comparisons in the following order:
   1. StudyDate/Time
   2. SeriesNumber
   3. AcquisitionNumber
   4. InstanceNumber

   This should obviously only ever be > or <, not ==, because of the first check.

Next remove repeated values as found in the first step above.

10.5 API Documentation

nibabel	Read / write access to some common neuroimaging file formats

10.5.1 nibabel

Read / write access to some common neuroimaging file formats

This package provides read +/- write access to some common medical and neuroimaging file formats, including: ANALYZE (plain, SPM99, SPM2 and later), GIFTI, NIfTI1, NIfTI2, CIFTI-2, MINC1, MINC2, AFNI BRIK/HEAD, MGH and ECAT as well as Philips PAR/REC. We can read and write FreeSurfer geometry, annotation and morphometry files. There is some very limited support for DICOM. NiBabel is the successor of PyNIfTI.

The various image format classes give full or selective access to header (meta) information and access to the image data is made available via NumPy arrays.

Website

Current documentation on nibabel can always be found at the NIPY nibabel website.

Mailing Lists

Please send any questions or suggestions to the neuroimaging mailing list.

Code

Install nibabel with:

```
pip install nibabel
```

You may also be interested in:

- the nibabel code repository on Github;
- documentation for all releases and current development tree;
- download the current release from pypi;
- download current development version as a zip file;
- downloads of all available releases.

License

Nibabel is licensed under the terms of the MIT license. Some code included with nibabel is licensed under the BSD license. Please see the COPYING file in the nibabel distribution.

Citing nibabel

Please see the available releases for the release of nibabel that you are using. Recent releases have a Zenodo Digital Object Identifier badge at the top of the release notes. Click on the badge for more information.

Quickstart

```
import nibabel as nib

img1 = nib.load('my_file.nii')
img2 = nib.load('other_file.nii.gz')
img3 = nib.load('spm_file.img')

data = img1.get_fdata()
affine = img1.affine

print(img1)

nib.save(img1, 'my_file_copy.nii.gz')

new_image = nib.Nifti1Image(data, affine)
nib.save(new_image, 'new_image.nii.gz')
```

For more detailed information see the *NiBabel Manual*.

bench([label, verbose, extra_argv])	Run benchmarks for nibabel using pytest
get_info()	
test([label, verbose, extra_argv, doctests, ...])	Run tests for nibabel using pytest

bench

nibabel.**bench**(*label=None*, *verbose=1*, *extra_argv=None*)

Run benchmarks for nibabel using pytest

The protocol mimics the `numpy.testing.NoseTester.bench()`. Not all features are currently implemented.

Parameters

label [None] Unused.

verbose: int, optional Verbosity value for test outputs. Positive values increase verbosity, and negative values decrease it. Default is 1.

extra_argv [list, optional] List with any extra arguments to pass to pytest.

Returns

code [ExitCode] Returns the result of running the tests as a `pytest.ExitCode` enum

get_info

nibabel.**get_info**()

test

nibabel.**test**(*label=None*, *verbose=1*, *extra_argv=None*, *doctests=False*, *coverage=False*, *raise_warnings=None*, *timer=False*)

Run tests for nibabel using pytest

The protocol mimics the `numpy.testing.NoseTester.test()`. Not all features are currently implemented.

Parameters

label [None] Unused.

verbose: int, optional Verbosity value for test outputs. Positive values increase verbosity, and negative values decrease it. Default is 1.

extra_argv [list, optional] List with any extra arguments to pass to pytest.

doctests: bool, optional If True, run doctests in module. Default is False.

coverage: bool, optional If True, report coverage of NumPy code. Default is False. (This requires the coverage module).

raise_warnings [None] Unused.

timer [False] Unused.

Returns

code [ExitCode] Returns the result of running the tests as a `pytest.ExitCode` enum

10.5.2 File Formats

analyze	Read / write access to the basic Mayo Analyze format
spm2analyze	Read / write access to SPM2 version of analyze image format
spm99analyze	Read / write access to SPM99 version of analyze image format
cifti2	CIFTI-2 format IO
gifti	GIfTI format IO
freesurfer	Reading functions for freesurfer files
minc1	Read MINC1 format images
minc2	Preliminary MINC2 support
nicom	DICOM reader
nifti1	Read / write access to NIfTI1 image format
nifti2	Read / write access to NIfTI2 image format
ecat	Read ECAT format images
parrec	Read images in PAR/REC format.
streamlines	Multiformat-capable streamline format read / write interface

analyze

Read / write access to the basic Mayo Analyze format

The Analyze header format

This is a binary header format and inherits from `WrapStruct`

Apart from the attributes and methods of WrapStruct:

Class attributes are:

```
.default_x_flip
```

with methods:

```
.get/set_data_shape
.get/set_data_dtype
.get/set_zooms
.get/set_data_offset
.get_base_affine()
.get_best_affine()
.data_to_fileobj
.data_from_fileobj
```

and class methods:

```
.from_header(hdr)
```

More sophisticated headers can add more methods and attributes.

Notes

This - basic - analyze header cannot encode full affines (only diagonal affines), and cannot do integer scaling.

The inability to store affines means that we have to guess what orientation the image has. Most Analyze images are stored on disk in (fastest-changing to slowest-changing) R->L, P->A and I->S order. That is, the first voxel is the rightmost, most posterior and most inferior voxel location in the image, and the next voxel is one voxel towards the left of the image.

Most people refer to this disk storage format as 'radiological', on the basis that, if you load up the data as an array `img_arr` where the first axis is the fastest changing, then take a slice in the I->S axis - `img_arr[:,:,10]` - then the right part of the brain will be on the left of your displayed slice. Radiologists like looking at images where the left of the brain is on the right side of the image.

Conversely, if the image has the voxels stored with the left voxels first - L->R, P->A, I->S, then this would be 'neurological' format. Neurologists like looking at images where the left side of the brain is on the left of the image.

When we are guessing at an affine for Analyze, this translates to the problem of whether the affine should consider proceeding within the data down an X line as being from left to right, or right to left.

By default we assume that the image is stored in R->L format. We encode this choice in the `default_x_flip` flag that can be True or False. True means assume radiological.

If the image is 3D, and the X, Y and Z zooms are x, y, and z, then:

```
if default_x_flip is True::
    affine = np.diag((-x,y,z,1))
else:
    affine = np.diag((x,y,z,1))
```

In our implementation, there is no way of saving this assumed flip into the header. One way of doing this, that we have not used, is to allow negative zooms, in particular, negative X zooms. We did not do this because the image can be loaded with and without a default flip, so the saved zoom will not constrain the affine.

AnalyzeHeader([binaryblock, endianness, check])	Class for basic analyze header
AnalyzeImage(dataobj, affine[, header, ...])	Class for basic Analyze format image

AnalyzeHeader

class nibabel.analyze.**AnalyzeHeader**(*binaryblock=None*, *endianness=None*, *check=True*)

Bases: *nibabel.wrapstruct.LabeledWrapStruct*

Class for basic analyze header

Implements zoom-only setting of affine transform, and no image scaling

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

__init__(*binaryblock=None*, *endianness=None*, *check=True*)

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binary-block. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

as_analyze_map()

Return header as mapping for conversion to Analyze types

Collect data from custom header type to fill in fields for Analyze and derived header types (such as Nifti1 and Nifti2).

When Analyze types convert another header type to their own type, they call this this method to check if there are other Analyze / Nifti fields that the source header would like to set.

Returns

analyze_map [mapping] Object that can be used as a mapping thus:

```
for key in analyze_map:
    value = analyze_map[key]
```

where `key` is the name of a field that can be set in an Analyze header type, such as Nifti1, and `value` is a value for the field. For example, *analyze_map* might be a something like `dict(regular='y', slice_duration=0.3)` where `regular` is a field present in both Analyze and Nifti1, and `slice_duration` is a field restricted to Nifti1 and

Nifti2. If a particular Analyze header type does not recognize the field name, it will throw away the value without error. See `Analyze.from_header()`.

Notes

You can also return a Nifti header with the relevant fields set.

Your header still needs methods `get_data_dtype`, `get_data_shape` and `get_zooms`, for the conversion, and these get called *after* using the analyze map, so the methods will override values set in the map.

data_from_fileobj(*fileobj*)
Read scaled data array from *fileobj*

Use this routine to get the scaled image data from an image file *fileobj*, given a header *self*. "Scaled" means, with any header scaling factors applied to the raw data in the file. Use *raw_data_from_fileobj* to get the raw data.

Parameters

fileobj [file-like] Must be open, and implement `read` and `seek` methods

Returns

arr [ndarray] scaled data array

Notes

We use the header to get any scale or intercept values to apply to the data. Raw Analyze files don't have scale factors or intercepts, but this routine also works with formats based on Analyze, that do have scaling, such as SPM analyze formats and NIfTI.

data_to_fileobj(*data*, *fileobj*, *rescale=True*)
Write *data* to *fileobj*, maybe rescaling data, modifying *self*

In writing the data, we match the header to the written data, by setting the header scaling factors, iff *rescale* is True. Thus we modify *self* in the process of writing the data.

Parameters

data [array-like] data to write; should match header defined shape

fileobj [file-like object] Object with file interface, implementing `write` and `seek`

rescale [{True, False}, optional] Whether to try and rescale data to match output dtype specified by header. If True and scaling needed and header cannot scale, then raise `HeaderTypeError`.

Examples

```
>>> from nibabel.analyze import AnalyzeHeader
>>> hdr = AnalyzeHeader()
>>> hdr.set_data_shape((1, 2, 3))
>>> hdr.set_data_dtype(np.float64)
>>> from io import BytesIO
>>> str_io = BytesIO()
>>> data = np.arange(6).reshape(1,2,3)
>>> hdr.data_to_fileobj(data, str_io)
```

(continues on next page)

(continued from previous page)

```
>>> data.astype(np.float64).tobytes('F') == str_io.getvalue()
True
```

classmethod default_structarr(*endianness=None*)
Return header data for empty header with given endianness

default_x_flip = True

classmethod from_header(*header=None*, *check=True*)
Class method to create header from another header

Parameters

header [Header instance or mapping] a header of this class, or another class of header for conversion to this type

check [{True, False}] whether to check header for integrity

Returns

hdr [header instance] fresh header instance of our own class

get_base_affine()
Get affine from basic (shared) header fields

Note that we get the translations from the center of the image.

Examples

```
>>> hdr = AnalyzeHeader()
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.set_zooms((3, 2, 1))
>>> hdr.default_x_flip
True
>>> hdr.get_base_affine() # from center of image
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
```

get_best_affine()
Get affine from basic (shared) header fields

Note that we get the translations from the center of the image.

Examples

```
>>> hdr = AnalyzeHeader()
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.set_zooms((3, 2, 1))
>>> hdr.default_x_flip
True
>>> hdr.get_base_affine() # from center of image
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
```

get_data_dtype()
Get numpy dtype for data

For examples see `set_data_dtype`

get_data_offset()
Return offset into data file to read data

Examples

```
>>> hdr = AnalyzeHeader()
>>> hdr.get_data_offset()
0
>>> hdr['vox_offset'] = 12
>>> hdr.get_data_offset()
12
```

get_data_shape()
Get shape of data

Examples

```
>>> hdr = AnalyzeHeader()
>>> hdr.get_data_shape()
(0,)
>>> hdr.set_data_shape((1,2,3))
>>> hdr.get_data_shape()
(1, 2, 3)
```

Expanding number of dimensions gets default zooms

```
>>> hdr.get_zooms()
(1.0, 1.0, 1.0)
```

get_slope_inter()
Get scalefactor and intercept

These are not implemented for basic Analyze

get_zooms()
Get zooms from header

Returns

z [tuple] tuple of header zoom values

Examples

```
>>> hdr = AnalyzeHeader()
>>> hdr.get_zooms()
(1.0,)
>>> hdr.set_data_shape((1,2))
>>> hdr.get_zooms()
(1.0, 1.0)
>>> hdr.set_zooms((3, 4))
>>> hdr.get_zooms()
(3.0, 4.0)
```

classmethod guessed_endian(*hdr*)

Guess intended endianness from mapping-like `hdr`

Parameters

hdr [mapping-like] hdr for which to guess endianness

Returns

endianness [{'<', '>'}] Guessed endianness of header

Examples

Zeros header, no information, guess native

```
>>> hdr = AnalyzeHeader()
>>> hdr_data = np.zeros((), dtype=header_dtype)
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
```

A valid native header is guessed native

```
>>> hdr_data = hdr.structarr.copy()
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
```

And, when swapped, is guessed as swapped

```
>>> sw_hdr_data = hdr_data.byteswap(swapped_code)
>>> AnalyzeHeader.guessed_endian(sw_hdr_data) == swapped_code
True
```

The algorithm is as follows:

First, look at the first value in the `dim` field; this should be between 0 and 7. If it is between 1 and 7, then this must be a native endian header.

```
>>> hdr_data = np.zeros((), dtype=header_dtype) # blank binary data
>>> hdr_data['dim'][0] = 1
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
>>> hdr_data['dim'][0] = 6
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
>>> hdr_data['dim'][0] = -1
```

(continues on next page)

(continued from previous page)

```
>>> AnalyzeHeader.guessed_endian(hdr_data) == swapped_code
True
```

If the first `dim` value is zeros, we need a tie breaker. In that case we check the `sizeof_hdr` field. This should be 348. If it looks like the byteswapped value of 348, assumed swapped. Otherwise assume native.

```
>>> hdr_data = np.zeros((), dtype=header_dtype) # blank binary data
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
>>> hdr_data['sizeof_hdr'] = 1543569408
>>> AnalyzeHeader.guessed_endian(hdr_data) == swapped_code
True
>>> hdr_data['sizeof_hdr'] = -1
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
```

This is overridden by the `dim[0]` value though:

```
>>> hdr_data['sizeof_hdr'] = 1543569408
>>> hdr_data['dim'][0] = 1
>>> AnalyzeHeader.guessed_endian(hdr_data) == native_code
True
```

has_data_intercept = False

has_data_slope = False

classmethod may_contain_header(*binaryblock*)

raw_data_from_fileobj(*fileobj*)

Read unscaled data array from *fileobj*

Parameters

fileobj [file-like] Must be open, and implement `read` and `seek` methods

Returns

arr [ndarray] unscaled data array

set_data_dtype(*datatype*)

Set numpy dtype for data from code or dtype or type

Examples

```
>>> hdr = AnalyzeHeader()
>>> hdr.set_data_dtype(np.uint8)
>>> hdr.get_data_dtype()
dtype('uint8')
>>> hdr.set_data_dtype(np.dtype(np.uint8))
>>> hdr.get_data_dtype()
dtype('uint8')
>>> hdr.set_data_dtype('implausible')
Traceback (most recent call last):
   ...
HeaderDataError: data dtype "implausible" not recognized
>>> hdr.set_data_dtype('none')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
   ...
HeaderDataError: data dtype "none" known but not supported
>>> hdr.set_data_dtype(np.void)
Traceback (most recent call last):
   ...
HeaderDataError: data dtype "<type 'numpy.void'>" known but not supported
```

set_data_offset(*offset*)
Set offset into data file to read data

set_data_shape(*shape*)
Set shape of data

If `ndims == len(shape)` then we set zooms for dimensions higher than `ndims` to 1.0

Parameters

shape [sequence] sequence of integers specifying data array shape

set_slope_inter(*slope*, *inter=None*)
Set slope and / or intercept into header

Set slope and intercept for image data, such that, if the image data is `arr`, then the scaled image data will be `(arr * slope) + inter`

In this case, for Analyze images, we can't store the slope or the intercept, so this method only checks that *slope* is None or NaN or 1.0, and that *inter* is None or NaN or 0.

Parameters

slope [None or float] If float, value must be NaN or 1.0 or we raise a `HeaderTypeError`

inter [None or float, optional] If float, value must be 0.0 or we raise a `HeaderTypeError`

set_zooms(*zooms*)
Set zooms into header fields

See docstring for `get_zooms` for examples

sizeof_hdr = 348

template_dtype = dtype([('sizeof_hdr', '<i4'), ('data_type', 'S10'), ('db_name', 'S18'

AnalyzeImage

class nibabel.analyze.**AnalyzeImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.spatialimages.SpatialImage*

Class for basic Analyze format image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

ImageArrayProxy
alias of *nibabel.arrayproxy.ArrayProxy*

files_types = (('image', '.img'), ('header', '.hdr'))

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)
Class method to create image from mapping in `file_map`

Parameters

file_map [dict] Mapping with (kay, value) pairs of (`file_type`, FileHolder instance giving file-likes for each file needed for this image type.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_map` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [AnalyzeImage instance]

get_data_dtype()

header_class
alias of *nibabel.analyze.AnalyzeHeader*

makeable = True

rw = True

set_data_dtype(*dtype*)

to_file_map(*file_map=None*)
Write image to *file_map* or contained `self.file_map`

Parameters

file_map [None or mapping, optional] files mapping. If None (default) use object's `file_map` attribute instead

valid_exts = ('.img', '.hdr')

spm2analyze

Read / write access to SPM2 version of analyze image format

Spm2AnalyzeHeader([binaryblock, endianness, ...])	Class for SPM2 variant of basic Analyze header
Spm2AnalyzeImage(dataobj, affine[, header, ...])	Class for SPM2 variant of basic Analyze image

Spm2AnalyzeHeader

class nibabel.spm2analyze.**Spm2AnalyzeHeader**(*binaryblock=None*, *endianness=None*, *check=True*)

Bases: *nibabel.spm99analyze.Spm99AnalyzeHeader*

Class for SPM2 variant of basic Analyze header

SPM2 variant adds the following to basic Analyze format:

- voxel origin;
- slope scaling of data;
- reading - but not writing - intercept of data.

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

__init__(*binaryblock=None*, *endianness=None*, *check=True*)

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binary block. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

get_slope_inter()

Get data scaling (slope) and intercept from header data

Uses the algorithm from SPM2 spm_vol_ana.m by John Ashburner

Parameters

self [header] Mapping with fields: * scl_slope - slope * scl_inter - possible intercept (SPM2 use - shared by nifti) * glmax - the (recorded) maximum value in the data (unscaled) * glmin - recorded minimum unscaled value * cal_max - the calibrated (scaled) maximum value in the dataset * cal_min - ditto minimum value

Returns

scl_slope [None or float] slope. None if there is no valid scaling from these fields

scl_inter [None or float] intercept. Also None if there is no valid slope, intercept

Examples

```
>>> fields = {'scl_slope': 1, 'scl_inter': 0, 'glmax': 0, 'glmin': 0,
...           'cal_max': 0, 'cal_min': 0}
>>> hdr = Spm2AnalyzeHeader()
>>> for key, value in fields.items():
...     hdr[key] = value
>>> hdr.get_slope_inter()
(1.0, 0.0)
>>> hdr['scl_inter'] = 0.5
>>> hdr.get_slope_inter()
(1.0, 0.5)
>>> hdr['scl_inter'] = np.nan
>>> hdr.get_slope_inter()
(1.0, 0.0)
```

If 'scl_slope' is 0, nan or inf, cannot use 'scl_slope'. Without valid information in the gl / cal fields, we cannot get scaling, and return None

```
>>> hdr['scl_slope'] = 0
>>> hdr.get_slope_inter()
(None, None)
>>> hdr['scl_slope'] = np.nan
>>> hdr.get_slope_inter()
(None, None)
```

Valid information in the gl AND cal fields are needed

```
>>> hdr['cal_max'] = 0.8
>>> hdr['cal_min'] = 0.2
>>> hdr.get_slope_inter()
(None, None)
>>> hdr['glmax'] = 110
>>> hdr['glmin'] = 10
>>> np.allclose(hdr.get_slope_inter(), [0.6/100, 0.2-0.6/100*10])
True
```

classmethod may_contain_header(*binaryblock*)

template_dtype = dtype([('sizeof_hdr', '<i4'), ('data_type', 'S10'), ('db_name', 'S18'

Spm2AnalyzeImage

class nibabel.spm2analyze.**Spm2AnalyzeImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.spm99analyze.Spm99AnalyzeImage*

Class for SPM2 variant of basic Analyze image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

header_class
alias of *nibabel.spm2analyze.Spm2AnalyzeHeader*

spm99analyze

Read / write access to SPM99 version of analyze image format

Spm99AnalyzeHeader([binaryblock, ...])	Class for SPM99 variant of basic Analyze header
Spm99AnalyzeImage(dataobj, affine[, header, ...])	Class for SPM99 variant of basic Analyze image
SpmAnalyzeHeader([binaryblock, endianness, ...])	Basic scaling Spm Analyze header

Spm99AnalyzeHeader

class nibabel.spm99analyze.**Spm99AnalyzeHeader**(*binaryblock=None*, *endianness=None*, *check=True*)
Bases: *nibabel.spm99analyze.SpmAnalyzeHeader*

Class for SPM99 variant of basic Analyze header

SPM99 variant adds the following to basic Analyze format:

- voxel origin;
- slope scaling of data.

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

__init__(*binaryblock=None*, *endianness=None*, *check=True*)
Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

get_best_affine()

Get affine from header, using SPM origin field if sensible

The default translations are got from the `origin` field, if set, or from the center of the image otherwise.

Examples

```
>>> hdr = Spm99AnalyzeHeader()
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.set_zooms((3, 2, 1))
>>> hdr.default_x_flip
True
>>> hdr.get_origin_affine() # from center of image
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
>>> hdr['origin'][:3] = [3,4,5]
```

(continues on next page)

(continued from previous page)

```
>>> hdr.get_origin_affine() # using origin
array([[-3.,  0.,  0.,  6.],
       [ 0.,  2.,  0., -6.],
       [ 0.,  0.,  1., -4.],
       [ 0.,  0.,  0.,  1.]])
>>> hdr['origin'] = 0 # unset origin
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.get_origin_affine() # from center of image
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
```

get_origin_affine()

Get affine from header, using SPM origin field if sensible

The default translations are got from the `origin` field, if set, or from the center of the image otherwise.

Examples

```
>>> hdr = Spm99AnalyzeHeader()
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.set_zooms((3, 2, 1))
>>> hdr.default_x_flip
True
>>> hdr.get_origin_affine() # from center of image
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
>>> hdr['origin'][:3] = [3,4,5]
>>> hdr.get_origin_affine() # using origin
array([[-3.,  0.,  0.,  6.],
       [ 0.,  2.,  0., -6.],
       [ 0.,  0.,  1., -4.],
       [ 0.,  0.,  0.,  1.]])
>>> hdr['origin'] = 0 # unset origin
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.get_origin_affine() # from center of image
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
```

set_origin_from_affine(*affine*)

Set SPM origin to header from affine matrix.

The `origin` field was read but not written by SPM99 and 2. It was used for storing a central voxel coordinate, that could be used in aligning the image to some standard position - a proxy for a full translation vector that was usually stored in a separate matlab .mat file.

Nifti uses the space occupied by the SPM `origin` field for important other information (the transform codes), so writing the origin will make the header a confusing Nifti file. If you work with both Analyze and Nifti, you should probably avoid doing this.

Parameters

affine [array-like, shape (4,4)] Affine matrix to set

Returns

None

Examples

```
>>> hdr = Spm99AnalyzeHeader()
>>> hdr.set_data_shape((3, 5, 7))
>>> hdr.set_zooms((3,2,1))
>>> hdr.get_origin_affine()
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
>>> affine = np.diag([3,2,1,1])
>>> affine[:3,3] = [-6, -6, -4]
>>> hdr.set_origin_from_affine(affine)
>>> np.all(hdr['origin'][:3] == [3,4,5])
True
>>> hdr.get_origin_affine()
array([[-3.,  0.,  0.,  6.],
       [ 0.,  2.,  0., -6.],
       [ 0.,  0.,  1., -4.],
       [ 0.,  0.,  0.,  1.]])
```

Spm99AnalyzeImage

class nibabel.spm99analyze.**Spm99AnalyzeImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.analyze.AnalyzeImage*

Class for SPM99 variant of basic Analyze image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

files_types = (('image', '.img'), ('header', '.hdr'), ('mat', '.mat'))

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)

Class method to create image from mapping in `file_map`

Parameters

file_map [dict] Mapping with (kay, value) pairs of (`file_type`, FileHolder instance giving file-likes for each file needed for this image type.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_map` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [Spm99AnalyzeImage instance]

has_affine = True

header_class

alias of *nibabel.spm99analyze.Spm99AnalyzeHeader*

makeable = True

rw = True

to_file_map(*file_map=None*)

Write image to *file_map* or contained `self.file_map`

Extends Analyze `to_file_map` method by writing `mat` file

Parameters

file_map [None or mapping, optional] files mapping. If None (default) use object's `file_map` attribute instead

SpmAnalyzeHeader

class nibabel.spm99analyze.**SpmAnalyzeHeader**(*binaryblock=None*, *endianness=None*, *check=True*)

Bases: *nibabel.analyze.AnalyzeHeader*

Basic scaling Spm Analyze header

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

__init__(*binaryblock=None*, *endianness=None*, *check=True*)
Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binary-block. If None, guess endianness from the data.

check [bool, optional] Whether to check content of header in initialization. Default is True.

Examples

```
>>> hdr1 = AnalyzeHeader() # an empty header
>>> hdr1.endianness == native_code
True
>>> hdr1.get_data_shape()
(0,)
>>> hdr1.set_data_shape((1,2,3)) # now with some content
>>> hdr1.get_data_shape()
(1, 2, 3)
```

We can set the binary block directly via this initialization. Here we get it from the header we have just made

```
>>> binblock2 = hdr1.binaryblock
>>> hdr2 = AnalyzeHeader(binblock2)
>>> hdr2.get_data_shape()
(1, 2, 3)
```

Empty headers are native endian by default

```
>>> hdr2.endianness == native_code
True
```

You can pass valid opposite endian headers with the `endianness` parameter. Even empty headers can have endianness

```
>>> hdr3 = AnalyzeHeader(endianness=swapped_code)
>>> hdr3.endianness == swapped_code
True
```

If you do not pass an endianness, and you pass some data, we will try to guess from the passed data.

```
>>> binblock3 = hdr3.binaryblock
>>> hdr4 = AnalyzeHeader(binblock3)
>>> hdr4.endianness == swapped_code
True
```

classmethod default_structarr(*endianness=None*)
Create empty header binary block with given endianness

get_slope_inter()
Get scalefactor and intercept

If scalefactor is 0.0 return None to indicate no scalefactor. Intercept is always None because SPM99 analyze cannot store intercepts.

has_data_intercept = False

has_data_slope = True

set_slope_inter(*slope*, *inter=None*)
Set slope and / or intercept into header

Set slope and intercept for image data, such that, if the image data is `arr`, then the scaled image data will be `(arr * slope) + inter`

The SPM Analyze header can't save an intercept value, and we raise an error unless *inter* is None, NaN or 0

Parameters

slope [None or float] If None, implies *slope* of NaN. NaN is a signal to the image writing routines to rescale on save. 0, Inf, -Inf are invalid and cause a HeaderDataError

inter [None or float, optional] intercept. Must be None, NaN or 0, because SPM99 cannot store intercepts.

template_dtype = dtype([('sizeof_hdr', '<i4'), ('data_type', 'S10'), ('db_name', 'S18'

cifti2

CIFTI-2 format IO

cifti2	Read / write access to CIFTI-2 image format
cifti2_axes	Defines `Axis` objects to create, read, and manipulate CIFTI-2 files

—

Module: cifti2.cifti2

Read / write access to CIFTI-2 image format

Format of the NIFTI2 container format described here:

http://www.nitrc.org/forum/message.php?msg_id=3738

Definition of the CIFTI-2 header format and file extensions can be found at:

http://www.nitrc.org/projects/cifti

Cifti2BrainModel([index_offset, ...])	Element representing a mapping of the dimension to vertex or voxels.
Cifti2Header([matrix, version])	Class for CIFTI-2 header extension
Cifti2HeaderError	Error in CIFTI-2 header
Cifti2Image([dataobj, header, nifti_header, ...])	Class for single file CIFTI-2 format image
Cifti2Label([key, label, red, green, blue, ...])	CIFTI-2 label: association of integer key with a name and RGBA values
Cifti2LabelTable()	CIFTI-2 label table: a sequence of `Cifti2Labels`
Cifti2Matrix()	CIFTI-2 Matrix object
Cifti2MatrixIndicesMap(...[, ...])	Class for Matrix Indices Map
Cifti2MetaData([metadata])	A list of name-value pairs
Cifti2NamedMap([map_name, metadata, label_table])	CIFTI-2 named map: association of name and optional data with a map index

continues on next page

Table 9 – continued from previous page

Cifti2Parcel([name, voxel_indices_ijk, vertices])	CIFTI-2 parcel: association of a name with vertices and/or voxels
Cifti2Surface([brain_structure, ...])	Cifti surface: association of brain structure and number of vertices
Cifti2TransformationMatrixVoxelIndicesIJKXYZ([...])	Matrix that translates voxel indices to spatial coordinates
Cifti2VertexIndices([indices])	CIFTI-2 vertex indices: vertex indices for an associated brain model
Cifti2Vertices([brain_structure, vertices])	CIFTI-2 vertices - association of brain structure and a list of vertices
Cifti2Volume([volume_dimensions, ...])	CIFTI-2 volume: information about a volume for mappings that use voxels
Cifti2VoxelIndicesIJK([indices])	CIFTI-2 VoxelIndicesIJK: Set of voxel indices contained in a structure

Module: `cifti2.cifti2_axes`

Defines *Axis* objects to create, read, and manipulate CIFTI-2 files

These axes provide an alternative interface to the information in the CIFTI-2 header. Each type of CIFTI-2 axes describing the rows/columns in a CIFTI-2 matrix is given a unique class:

- *BrainModelAxis*: each row/column is a voxel or vertex
- *ParcelsAxis*: each row/column is a group of voxels and/or vertices
- *ScalarAxis*: each row/column has a unique name (with optional meta-data)
- *LabelAxis*: each row/column has a unique name and label table (with optional meta-data)
- *SeriesAxis*: each row/column is a timepoint, which increases monotonically

All of these classes are derived from the Axis class.

After loading a CIFTI-2 file a tuple of axes describing the rows and columns can be obtained from the *cifti2.Cifti2Header.get_axis()* method on the header object (e.g. `nibabel.load(<filename>).header.get_axis()`). Inversely, a new *cifti2.Cifti2Header* object can be created from existing Axis objects using the *cifti2.Cifti2Header.from_axes()* factory method.

CIFTI-2 Axis objects of the same type can be concatenated using the '+'-operator. Numpy indexing also works on axes (except for SeriesAxis objects, which have to remain monotonically increasing or decreasing).

Creating new CIFTI-2 axes

New Axis objects can be constructed by providing a description for what is contained in each row/column of the described tensor. For each Axis sub-class this descriptor is:

- *BrainModelAxis*: a CIFTI-2 structure name and a voxel or vertex index
- *ParcelsAxis*: a name and a sequence of voxel and vertex indices
- *ScalarAxis*: a name and optionally a dict of meta-data
- *LabelAxis*: a name, dict of label index to name and colour, and optionally a dict of meta-data
- *SeriesAxis*: the time-point of each row/column is set by setting the start, stop, size, and unit of the time-series

Several helper functions exist to create new *BrainModelAxis* axes:

- *BrainModelAxis.from_mask()* creates a new BrainModelAxis volume covering the non-zero values of a mask
- *BrainModelAxis.from_surface()* creates a new BrainModelAxis surface covering the provided indices of a surface

A *ParcelsAxis* axis can be created from a sequence of *BrainModelAxis* axes using *ParcelsAxis.from_brain_models()*.

Examples

We can create brain models covering the left cortex and left thalamus using:

```
>>> from nibabel import cifti2
>>> import numpy as np
>>> bm_cortex = cifti2.BrainModelAxis.from_mask([True, False, True, True],
...                                             name='cortex_left')
>>> bm_thal = cifti2.BrainModelAxis.from_mask(np.ones((2, 2, 2)), affine=np.eye(4),
...                                           name='thalamus_left')
```

In this very simple case `bm_cortex` describes a left cortical surface skipping the second out of four vertices. `bm_thal` contains all voxels in a 2x2x2 volume.

Brain structure names automatically get converted to valid CIFTI-2 indentifiers using *BrainModelAxis.to_cifti_brain_structure_name()*. A 1-dimensional mask will be automatically interpreted as a surface element and a 3-dimensional mask as a volume element.

These can be concatenated in a single brain model covering the left cortex and thalamus by simply adding them together

```
>>> bm_full = bm_cortex + bm_thal
```

Brain models covering the full HCP grayordinate space can be constructed by adding all the volumetric and surface brain models together like this (or by reading one from an already existing HCP file).

Getting a specific brain region from the full brain model is as simple as:

```
>>> assert bm_full[bm_full.name == 'CIFTI_STRUCTURE_CORTEX_LEFT'] == bm_cortex
>>> assert bm_full[bm_full.name == 'CIFTI_STRUCTURE_THALAMUS_LEFT'] == bm_thal
```

You can also iterate over all brain structures in a brain model:

```
>>> for idx, (name, slc, bm) in enumerate(bm_full.iter_structures()):
...     print((str(name), slc))
...     assert bm == bm_full[slc]
...     assert bm == bm_cortex if idx == 0 else bm_thal
('CIFTI_STRUCTURE_CORTEX_LEFT', slice(0, 3, None))
('CIFTI_STRUCTURE_THALAMUS_LEFT', slice(3, None, None))
```

In this case there will be two iterations, namely: ('CIFTI_STRUCTURE_CORTEX_LEFT', slice(0, <size of cortex mask>), bm_cortex) and ('CIFTI_STRUCTURE_THALAMUS_LEFT', slice(<size of cortex mask>, None), bm_thal)

ParcelsAxis can be constructed from selections of these brain models:

```
>>> parcel = cifti2.ParcelsAxis.from_brain_models([
...         ('surface_parcel', bm_cortex[:2]),  # contains first 2 cortical vertices
...         ('volume_parcel', bm_thal),  # contains thalamus
...         ('combined_parcel', bm_full[[1, 8, 10]]),  # contains selected voxels/
↪vertices
...     ])
```

Time series are represented by their starting time (typically 0), step size (i.e. sampling time or TR), and number of elements:

```
>>> series = cifti2.SeriesAxis(start=0, step=100, size=5000)
```

So a header for fMRI data with a TR of 100 ms covering the left cortex and thalamus with 5000 timepoints could be created with

```
>>> type(cifti2.Cifti2Header.from_axes((series, bm_cortex + bm_thal)))
<class 'nibabel.cifti2.cifti2.Cifti2Header'>
```

Similarly the curvature and cortical thickness on the left cortex could be stored using a header like:

```
>>> type(cifti2.Cifti2Header.from_axes((cifti2.ScalarAxis(['curvature', 'thickness']),
...                                     bm_cortex)))
<class 'nibabel.cifti2.cifti2.Cifti2Header'>
```

Axis()	Abstract class for any object describing the rows or columns of a CIFTI-2 vector/matrix
BrainModelAxis(name[, voxel, vertex, ...])	Each row/column in the CIFTI-2 vector/matrix represents a single vertex or voxel
LabelAxis(name, label[, meta])	Defines CIFTI-2 axis for label array.
ParcelsAxis(name, voxels, vertices[, ...])	Each row/column in the CIFTI-2 vector/matrix represents a parcel of voxels/vertices
ScalarAxis(name[, meta])	Along this axis of the CIFTI-2 vector/matrix each row/column has been given a unique name and optionally metadata
SeriesAxis(start, step, size[, unit])	Along this axis of the CIFTI-2 vector/matrix the rows/columns increase monotonously in time
from_index_mapping(mim)	Parses the MatrixIndicesMap to find the appropriate CIFTI-2 axis describing the rows or columns
to_header(axes)	Converts the axes describing the rows/columns of a CIFTI-2 vector/matrix to a Cifti2Header

Module: `cifti2.parse_cifti2`

Cifti2Extension([code, content])	**Parameters**
Cifti2Parser([encoding, buffer_size, verbose])	Class to parse an XML string into a CIFTI-2 header object

Cifti2BrainModel

class nibabel.cifti2.cifti2.**Cifti2BrainModel**(*index_offset=None*, *index_count=None*, *model_type=None*, *brain_structure=None*, *n_surface_vertices=None*, *voxel_indices_ijk=None*, *vertex_indices=None*)

Bases: *nibabel.xmlutils.XmlSerializable*

Element representing a mapping of the dimension to vertex or voxels.

Mapping to vertices of voxels must be specified.

- Description - Maps a range of indices to surface vertices or voxels when IndicesMapToDataType is "CIFTI_INDEX_TYPE_BRAIN_MODELS."
- Attributes
  - IndexOffset - The matrix index of the first brainordinate of this BrainModel. Note that matrix indices are zero-based.
  - IndexCount - Number of surface vertices or voxels in this brain model, must be positive.
  - ModelType - Type of model representing the brain structure (surface or voxels). Valid values are listed in the table below.
  - BrainStructure - Identifies the brain structure. Valid values for BrainStructure are listed in the table below. However, if the needed structure is not listed in the table, a message should be posted to the CIFTI Forum so that a standardized name can be created for the structure and added to the table.
  - SurfaceNumberOfVertices - When ModelType is CIFTI_MODEL_TYPE_SURFACE this attribute contains the actual (or true) number of vertices in the surface that is associated with this BrainModel. When this BrainModel represents all vertices in the surface, this value is the same as IndexCount. When this BrainModel represents only a subset of the surface's vertices, IndexCount will be less than this value.
- Child Elements
  - VertexIndices (0...1)
  - VoxelIndicesIJK (0...1)
- Text Content: [NA]
- Parent Element - MatrixIndicesMap

For ModelType values, see CIFTI_MODEL_TYPES module attribute.

For BrainStructure values, see CIFTI_BRAIN_STRUCTURES model attribute.

Attributes

index_offset [int] Start of the mapping

index_count [int] Number of elements in the array to be mapped

model_type [str] One of CIFTI_MODEL_TYPES

brain_structure [str] One of CIFTI_BRAIN_STRUCTURES

surface_number_of_vertices [int] Number of vertices in the surface. Use only for surface-type structure

voxel_indices_ijk [Cifti2VoxelIndicesIJK, optional] Indices on the image towards where the array indices are mapped

vertex_indices [Cifti2VertexIndices, optional] Indices of the vertices towards where the array indices are mapped

__init__(*index_offset=None*, *index_count=None*, *model_type=None*, *brain_structure=None*, *n_surface_vertices=None*, *voxel_indices_ijk=None*, *vertex_indices=None*)
Initialize self. See help(type(self)) for accurate signature.

property vertex_indices

property voxel_indices_ijk

Cifti2Header

class nibabel.cifti2.cifti2.**Cifti2Header**(*matrix=None*, *version='2.0'*)
Bases: *nibabel.filebasedimages.FileBasedHeader*, *nibabel.xmlutils.XmlSerializable*

Class for CIFTI-2 header extension

__init__(*matrix=None*, *version='2.0'*)
Initialize self. See help(type(self)) for accurate signature.

classmethod from_axes(*axes*)
Creates a new Cifti2 header based on the Cifti2 axes

Parameters

axes [tuple of :class`.cifti2_axes.Axis`] sequence of Cifti2 axes describing each row/column of the matrix to be stored

Returns

header [Cifti2Header] new header describing the rows/columns in a format consistent with Cifti2

get_axis(*index*)
Generates the Cifti2 axis for a given dimension

Parameters

index [int] Dimension for which we want to obtain the mapping.

Returns

axis [*cifti2_axes.Axis*]

get_index_map(*index*)
Cifti2 Mapping class for a given index

Parameters

index [int] Index for which we want to obtain the mapping. Must be in the mapped_indices sequence.

Returns

cifti2_map [Cifti2MatrixIndicesMap] Returns the Cifti2MatrixIndicesMap corresponding to the given index.

property mapped_indices
List of matrix indices that are mapped

classmethod may_contain_header(*binaryblock*)

property number_of_mapped_indices
Number of mapped indices

Cifti2HeaderError

class nibabel.cifti2.cifti2.**Cifti2HeaderError**
Bases: Exception

Error in CIFTI-2 header

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Cifti2Image

class nibabel.cifti2.cifti2.**Cifti2Image**(*dataobj=None*, *header=None*, *nifti_header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.dataobj_images.DataobjImage*, *nibabel.filebasedimages.SerializableImage*

Class for single file CIFTI-2 format image

Initialize image

The image is a combination of (dataobj, header), with optional metadata in *nifti_header* (a NIfTI2 header). There may be more metadata in the mapping *extra*. Filename / file-like objects can also go in the *file_map* mapping.

Parameters

dataobj [object] Object containing image data. It should be some object that returns an array from `np.asanyarray`. It should have a `shape` attribute or property.

header [Cifti2Header instance or sequence of `cifti2_axes.Axis`] Header with data for / from XML part of CIFTI-2 format. Alternatively a sequence of cifti2_axes.Axis objects can be provided describing each dimension of the array.

nifti_header [None or mapping or NIfTI2 header instance, optional] Metadata for NIfTI2 component of this format.

extra [None or mapping] Extra metadata not captured by *header* or *nifti_header*.

file_map [mapping, optional] Mapping giving file information for this image format.

__init__(*dataobj=None*, *header=None*, *nifti_header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (dataobj, header), with optional metadata in *nifti_header* (a NIfTI2 header). There may be more metadata in the mapping *extra*. Filename / file-like objects can also go in the *file_map* mapping.

Parameters

dataobj [object] Object containing image data. It should be some object that returns an array from `np.asanyarray`. It should have a `shape` attribute or property.

header [Cifti2Header instance or sequence of `cifti2_axes.Axis`] Header with data for / from XML part of CIFTI-2 format. Alternatively a sequence of cifti2_axes.Axis objects can be provided describing each dimension of the array.

nifti_header [None or mapping or NIfTI2 header instance, optional] Metadata for NIfTI2 component of this format.

extra [None or mapping] Extra metadata not captured by *header* or *nifti_header*.

file_map [mapping, optional] Mapping giving file information for this image format.

files_types = (('image', '.nii'),)

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)
Load a CIFTI-2 image from a file_map

Parameters

file_map [file_map]

Returns

img [Cifti2Image] Returns a Cifti2Image

classmethod from_image(*img*)
Class method to create new instance of own class from *img*

Parameters

img [instance] In fact, an object with the API of `DataobjImage`.

Returns

cimg [instance] Image, of our own class

get_data_dtype()

header_class
alias of *nibabel.cifti2.cifti2.Cifti2Header*

makeable = False

property nifti_header

rw = True

set_data_dtype(*dtype*)

to_file_map(*file_map=None*)
Write image to *file_map* or contained `self.file_map`

Parameters

file_map [None or mapping, optional] files mapping. If None (default) use object's `file_map` attribute instead.

Returns

None

update_headers()
Harmonize NIfTI headers with image data

Ensures that the NIfTI-2 header records the data shape in the last three `dim` fields. Per the spec:

> Because the first four dimensions in NIfTI are reserved for space and time, the CIFTI dimensions are stored in the NIfTI header in dim[5] and up, where dim[5] is the length of the first CIFTI dimension (number of values in a row), dim[6] is the length of the second CIFTI dimension, and dim[7] is the length of the third CIFTI dimension, if applicable. The fields dim[1] through dim[4] will be 1; dim[0] will be 6 or 7, depending on whether a third matrix dimension exists.

```
>>> import numpy as np
>>> data = np.zeros((2,3,4))
>>> img = Cifti2Image(data)
>>> img.shape == (2, 3, 4)
True
>>> img.update_headers()
>>> img.nifti_header.get_data_shape() == (1, 1, 1, 1, 2, 3, 4)
True
>>> img.shape == (2, 3, 4)
True
```

valid_exts = ('.nii',)

Cifti2Label

class nibabel.cifti2.cifti2.**Cifti2Label**(*key=0*, *label=''*, *red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*)

Bases: *nibabel.xmlutils.XmlSerializable*

CIFTI-2 label: association of integer key with a name and RGBA values

For all color components, value is floating point with range 0.0 to 1.0.

- Description - Associates a label key value with a name and a display color.
- Attributes
  - Key - Integer, data value which is assigned this name and color.
  - Red - Red color component for label. Value is floating point with range 0.0 to 1.0.
  - Green - Green color component for label. Value is floating point with range 0.0 to 1.0.
  - Blue - Blue color component for label. Value is floating point with range 0.0 to 1.0.
  - Alpha - Alpha color component for label. Value is floating point with range 0.0 to 1.0.
- Child Elements: [NA]
- Text Content - Name of the label.
- Parent Element - LabelTable

Attributes

key [int, optional] Integer, data value which is assigned this name and color.

label [str, optional] Name of the label.

red [float, optional] Red color component for label (between 0 and 1).

green [float, optional] Green color component for label (between 0 and 1).

blue [float, optional] Blue color component for label (between 0 and 1).

alpha [float, optional] Alpha color component for label (between 0 and 1).

__init__(*key=0*, *label=''*, *red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*)
Initialize self. See help(type(self)) for accurate signature.

property rgba
Returns RGBA as tuple

Cifti2LabelTable

class nibabel.cifti2.cifti2.**Cifti2LabelTable**
Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableMapping

CIFTI-2 label table: a sequence of Cifti2Labels

- Description - Used by NamedMap when IndicesMapToDataType is "CIFTI_INDEX_TYPE_LABELS" in order to associate names and display colors with label keys. Note that LABELS is the only mapping type that uses a LabelTable. Display coloring of continuous-valued data is not specified by CIFTI-2.
- Attributes: [NA]
- Child Elements
  - Label (0...N)
- Text Content: [NA]
- Parent Element - NamedMap

__init__()
Initialize self. See help(type(self)) for accurate signature.

append(*label*)

Cifti2Matrix

class nibabel.cifti2.cifti2.**Cifti2Matrix**
Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableSequence

CIFTI-2 Matrix object

This is a list-like container where the elements are instances of *Cifti2MatrixIndicesMap*.

- Description: contains child elements that describe the meaning of the values in the matrix.
- Attributes: [NA]
- Child Elements
  - MetaData (0 .. 1)
  - MatrixIndicesMap (1 .. N)
- Text Content: [NA]
- Parent Element: CIFTI

For each matrix (data) dimension, exactly one MatrixIndicesMap element must list it in the AppliesToMatrixDimension attribute.

__init__()
Initialize self. See help(type(self)) for accurate signature.

get_axis(*index*)
Generates the Cifti2 axis for a given dimension

Parameters

index [int] Dimension for which we want to obtain the mapping.

Returns

axis [*cifti2_axes.Axis*]

get_data_shape()
Returns data shape expected based on the CIFTI-2 header

Any dimensions omitted in the CIFTI-2 header will be given a default size of None.

get_index_map(*index*)
Cifti2 Mapping class for a given index

Parameters

index [int] Index for which we want to obtain the mapping. Must be in the mapped_indices sequence.

Returns

cifti2_map [Cifti2MatrixIndicesMap] Returns the Cifti2MatrixIndicesMap corresponding to the given index.

insert(*index*, *value*)
S.insert(index, value) – insert value before index

property mapped_indices
List of matrix indices that are mapped

property metadata

Cifti2MatrixIndicesMap

class nibabel.cifti2.cifti2.**Cifti2MatrixIndicesMap**(*applies_to_matrix_dimension*, *indices_map_to_data_type*, *number_of_series_points=None*, *series_exponent=None*, *series_start=None*, *series_step=None*, *series_unit=None*, *maps=[]*)
Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableSequence

Class for Matrix Indices Map

- Description - Provides a mapping between matrix indices and their interpretation.
- Attributes
  - AppliesToMatrixDimension - Lists the dimension(s) of the matrix to which this MatrixIndicesMap applies. The dimensions of the matrix start at zero (dimension 0 describes the indices along the first dimension, dimension 1 describes the indices along the second dimension, etc.). If this MatrixIndicesMap applies to more than one matrix dimension, the values are separated by a comma.
  - IndicesMapToDataType - Type of data to which the MatrixIndicesMap applies.
  - NumberOfSeriesPoints - Indicates how many samples there are in a series mapping type. For example, this could be the number of timepoints in a timeseries.
  - SeriesExponent - Integer, SeriesStart and SeriesStep must be multiplied by 10 raised to the power of the value of this attribute to give the actual values assigned to indices (e.g., if SeriesStart is "5" and SeriesExponent is "-3", the value of the first series point is 0.005).
  - SeriesStart - Indicates what quantity should be assigned to the first series point.
  - SeriesStep - Indicates amount of change between each series point.
  - SeriesUnit - Indicates the unit of the result of multiplying SeriesStart and SeriesStep by 10 to the power of SeriesExponent.

- Child Elements
  - BrainModel (0…N)
  - NamedMap (0…N)
  - Parcel (0…N)
  - Surface (0…N)
  - Volume (0…1)
- Text Content: [NA]
- Parent Element - Matrix

Attributes

applies_to_matrix_dimension [list of ints] Dimensions of this matrix that follow this mapping

indices_map_to_data_type [str one of CIFTI_MAP_TYPES] Type of mapping to the matrix indices

number_of_series_points [int, optional] If it is a series, number of points in the series

series_exponent [int, optional] If it is a series the exponent of the increment

series_start [float, optional] If it is a series, starting time

series_step [float, optional] If it is a series, step per element

series_unit [str, optional] If it is a series, units

__init__(*applies_to_matrix_dimension*, *indices_map_to_data_type*, *number_of_series_points=None*, *series_exponent=None*, *series_start=None*, *series_step=None*, *series_unit=None*, *maps=[]*)
Initialize self. See help(type(self)) for accurate signature.

property brain_models

insert(*index*, *value*)
S.insert(index, value) – insert value before index

property named_maps

property parcels

property surfaces

property volume

Cifti2MetaData

class nibabel.cifti2.cifti2.**Cifti2MetaData**(*metadata=None*)
Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableMapping

A list of name-value pairs

- Description - Provides a simple method for user-supplied metadata that associates names with values.
- Attributes: [NA]
- Child Elements
  - MD (0…N)

- Text Content: [NA]
- Parent Elements - Matrix, NamedMap

MD elements are a single metadata entry consisting of a name and a value.

Attributes

data [list of (name, value) tuples]

__init__(*metadata=None*)
Initialize self. See help(type(self)) for accurate signature.

difference_update(*metadata*)
Remove metadata key-value pairs

Parameters

metadata [dict-like datatype]

Returns

None

Cifti2NamedMap

class nibabel.cifti2.cifti2.**Cifti2NamedMap**(*map_name=None*, *metadata=None*, *label_table=None*)
Bases: *nibabel.xmlutils.XmlSerializable*

CIFTI-2 named map: association of name and optional data with a map index

Associates a name, optional metadata, and possibly a LabelTable with an index in a map.

- Description - Associates a name, optional metadata, and possibly a LabelTable with an index in a map.
- Attributes: [NA]
- Child Elements
  - MapName (1)
  - LabelTable (0...1)
  - MetaData (0...1)
- Text Content: [NA]
- Parent Element - MatrixIndicesMap

Attributes

map_name [str] Name of map

metadata [None or Cifti2MetaData] Metadata associated with named map

label_table [None or Cifti2LabelTable] Label table associated with named map

__init__(*map_name=None*, *metadata=None*, *label_table=None*)
Initialize self. See help(type(self)) for accurate signature.

property label_table

property metadata

Cifti2Parcel

class nibabel.cifti2.cifti2.**Cifti2Parcel**(*name=None*, *voxel_indices_ijk=None*, *vertices=None*)

Bases: *nibabel.xmlutils.XmlSerializable*

CIFTI-2 parcel: association of a name with vertices and/or voxels

- Description - Associates a name, plus vertices and/or voxels, with an index.
- Attributes
  - Name - The name of the parcel
- Child Elements
  - Vertices (0...N)
  - VoxelIndicesIJK (0...1)
- Text Content: [NA]
- Parent Element - MatrixIndicesMap

Attributes

name [str] Name of parcel

voxel_indices_ijk [None or Cifti2VoxelIndicesIJK] Voxel indices associated with parcel

vertices [list of Cifti2Vertices] Vertices associated with parcel

__init__(*name=None*, *voxel_indices_ijk=None*, *vertices=None*)
Initialize self. See help(type(self)) for accurate signature.

append_cifti_vertices(*vertices*)
Appends a Cifti2Vertices element to the Cifti2Parcel

Parameters

vertices [Cifti2Vertices]

pop_cifti2_vertices(*ith*)
Pops the ith vertices element from the Cifti2Parcel

property voxel_indices_ijk

Cifti2Surface

class nibabel.cifti2.cifti2.**Cifti2Surface**(*brain_structure=None*, *surface_number_of_vertices=None*)

Bases: *nibabel.xmlutils.XmlSerializable*

Cifti surface: association of brain structure and number of vertices

- Description - Specifies the number of vertices for a surface, when IndicesMapToDataType is "CIFTI_INDEX_TYPE_PARCELS." This is separate from the Parcel element because there can be multiple parcels on one surface, and one parcel may involve multiple surfaces.
- Attributes
  - BrainStructure - A string from the BrainStructure list to identify what surface structure this element refers to (usually left cortex, right cortex, or cerebellum).

  - SurfaceNumberOfVertices - The number of vertices that this structure's surface contains.

- Child Elements: [NA]
- Text Content: [NA]
- Parent Element - MatrixIndicesMap

Attributes

brain_structure [str] Name of brain structure

surface_number_of_vertices [int] Number of vertices on surface

__init__(*brain_structure=None*, *surface_number_of_vertices=None*)
Initialize self. See help(type(self)) for accurate signature.

Cifti2TransformationMatrixVoxelIndicesIJKtoXYZ

class nibabel.cifti2.cifti2.**Cifti2TransformationMatrixVoxelIndicesIJKtoXYZ**(*meter_exponent=None*, *matrix=None*)

Bases: *nibabel.xmlutils.XmlSerializable*

Matrix that translates voxel indices to spatial coordinates

- Description - Contains a matrix that translates Voxel IJK Indices to spatial XYZ coordinates (+X=>right, +Y=>anterior, +Z=> superior). The resulting coordinate is the center of the voxel.
- Attributes
  - MeterExponent - Integer, specifies that the coordinate result from the transformation matrix should be multiplied by 10 to this power to get the spatial coordinates in meters (e.g., if this is "-3", then the transformation matrix is in millimeters).
- Child Elements: [NA]
- Text Content - Sixteen floating-point values, in row-major order, that form a 4x4 homogeneous transformation matrix.
- Parent Element - Volume

Attributes

meter_exponent [int] See attribute description above.

matrix [array-like shape (4, 4)] Affine transformation matrix from voxel indices to RAS space.

__init__(*meter_exponent=None*, *matrix=None*)
Initialize self. See help(type(self)) for accurate signature.

Cifti2VertexIndices

class nibabel.cifti2.cifti2.**Cifti2VertexIndices**(*indices=None*)
Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableSequence

CIFTI-2 vertex indices: vertex indices for an associated brain model

The vertex indices (which are independent for each surface, and zero-based) that are used in this brain model[.] The parent BrainModel's `index_count` indicates the number of indices.

- Description - Contains a list of vertex indices for a BrainModel with ModelType equal to CIFTI_MODEL_TYPE_SURFACE.
- Attributes: [NA]
- Child Elements: [NA]
- Text Content - The vertex indices (which are independent for each surface, and zero-based) that are used in this brain model, with each index separated by a whitespace character. The parent BrainModel's IndexCount attribute indicates the number of indices in this element's content.
- Parent Element - BrainModel

__init__(*indices=None*)
Initialize self. See help(type(self)) for accurate signature.

insert(*index*, *value*)
S.insert(index, value) – insert value before index

Cifti2Vertices

class nibabel.cifti2.cifti2.**Cifti2Vertices**(*brain_structure=None*, *vertices=None*)
Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableSequence

CIFTI-2 vertices - association of brain structure and a list of vertices

- Description - Contains a BrainStructure type and a list of vertex indices within a Parcel.
- Attributes
  - BrainStructure - A string from the BrainStructure list to identify what surface this vertex list is from (usually left cortex, right cortex, or cerebellum).
- Child Elements: [NA]
- Text Content - Vertex indices (which are independent for each surface, and zero-based) separated by whitespace characters.
- Parent Element - Parcel

The class behaves like a list of Vertex indices (which are independent for each surface, and zero-based)

Attributes

brain_structure [str] A string from the BrainStructure list to identify what surface this vertex list is from (usually left cortex, right cortex, or cerebellum).

__init__(*brain_structure=None*, *vertices=None*)
Initialize self. See help(type(self)) for accurate signature.

insert(*index*, *value*)
S.insert(index, value) – insert value before index

Cifti2Volume

class nibabel.cifti2.cifti2.**Cifti2Volume**(*volume_dimensions=None*, *transform_matrix=None*)

Bases: *nibabel.xmlutils.XmlSerializable*

CIFTI-2 volume: information about a volume for mappings that use voxels

- Description - Provides information about the volume for any mappings that use voxels.
- Attributes
  - VolumeDimensions - Three integer values separated by commas, the lengths of the three volume file dimensions that are related to spatial coordinates, in number of voxels. Voxel indices (which are zero-based) that are used in the mapping that this element applies to must be within these dimensions.
- Child Elements
  - TransformationMatrixVoxelIndicesIJKtoXYZ (1)
- Text Content: [NA]
- Parent Element - MatrixIndicesMap

Attributes

volume_dimensions [array-like shape (3,)] See attribute description above.

transformation_matrix_voxel_indices_ijk_to_xyz [Cifti2TransformationMatrixVoxelIndicesIJKtoXYZ] Matrix that translates voxel indices to spatial coordinates

__init__(*volume_dimensions=None*, *transform_matrix=None*)
Initialize self. See help(type(self)) for accurate signature.

Cifti2VoxelIndicesIJK

class nibabel.cifti2.cifti2.**Cifti2VoxelIndicesIJK**(*indices=None*)

Bases: *nibabel.xmlutils.XmlSerializable*, collections.abc.MutableSequence

CIFTI-2 VoxelIndicesIJK: Set of voxel indices contained in a structure

- Description - Identifies the voxels that model a brain structure, or participate in a parcel. Note that when this is a child of BrainModel, the IndexCount attribute of the BrainModel indicates the number of voxels contained in this element.
- Attributes: [NA]
- Child Elements: [NA]
- Text Content - IJK indices (which are zero-based) of each voxel in this brain model or parcel, with each index separated by a whitespace character. There are three indices per voxel. If the parent element is BrainModel, then the BrainModel element's IndexCount attribute indicates the number of triplets (IJK indices) in this element's content.
- Parent Elements - BrainModel, Parcel

Each element of this sequence is a triple of integers.

__init__(*indices=None*)
Initialize self. See help(type(self)) for accurate signature.

insert (*index*, *value*)
S.insert(index, value) – insert value before index

Axis

class nibabel.cifti2.cifti2_axes.**Axis**
Bases: abc.ABC

Abstract class for any object describing the rows or columns of a CIFTI-2 vector/matrix

Mainly used for type checking.

Base class for the following concrete CIFTI-2 axes:

- *BrainModelAxis*: each row/column is a voxel or vertex
- *ParcelsAxis*: each row/column is a group of voxels and/or vertices
- *ScalarAxis*: each row/column has a unique name with optional meta-data
- *LabelAxis*: each row/column has a unique name and label table with optional meta-data
- *SeriesAxis*: each row/column is a timepoint, which increases monotonically

__init__ (**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property size

BrainModelAxis

class nibabel.cifti2.cifti2_axes.**BrainModelAxis**(*name*, *voxel=None*, *vertex=None*, *affine=None*, *volume_shape=None*, *nvertices=None*)
Bases: *nibabel.cifti2.cifti2_axes.Axis*

Each row/column in the CIFTI-2 vector/matrix represents a single vertex or voxel

This Axis describes which vertex/voxel is represented by each row/column.

New BrainModelAxis axes can be constructed by passing on the greyordinate brain-structure names and voxel/vertex indices to the constructor or by one of the factory methods:

- *from_mask()*: creates surface or volumetric BrainModelAxis axis from respectively 1D or 3D masks
- *from_surface()*: creates a surface BrainModelAxis axis

The resulting BrainModelAxis axes can be concatenated by adding them together.

Parameters

name [array_like] brain structure name or (N,) string array with the brain structure names

voxel [array_like, optional] (N, 3) array with the voxel indices (can be omitted for CIFTI-2 files only covering the surface)

vertex [array_like, optional] (N,) array with the vertex indices (can be omitted for volumetric CIFTI-2 files)

affine [array_like, optional] (4, 4) array mapping voxel indices to mm space (not needed for CIFTI-2 files only covering the surface)

volume_shape [tuple of three integers, optional] shape of the volume in which the voxels were defined (not needed for CIFTI-2 files only covering the surface)

nvertices [dict from string to integer, optional] maps names of surface elements to integers (not needed for volumetric CIFTI-2 files)

__init__(*name*, *voxel=None*, *vertex=None*, *affine=None*, *volume_shape=None*, *nvertices=None*)

New BrainModelAxis axes can be constructed by passing on the greyordinate brain-structure names and voxel/vertex indices to the constructor or by one of the factory methods:

- *from_mask()*: creates surface or volumetric BrainModelAxis axis from respectively 1D or 3D masks
- *from_surface()*: creates a surface BrainModelAxis axis

The resulting BrainModelAxis axes can be concatenated by adding them together.

Parameters

name [array_like] brain structure name or (N,) string array with the brain structure names

voxel [array_like, optional] (N, 3) array with the voxel indices (can be omitted for CIFTI-2 files only covering the surface)

vertex [array_like, optional] (N,) array with the vertex indices (can be omitted for volumetric CIFTI-2 files)

affine [array_like, optional] (4, 4) array mapping voxel indices to mm space (not needed for CIFTI-2 files only covering the surface)

volume_shape [tuple of three integers, optional] shape of the volume in which the voxels were defined (not needed for CIFTI-2 files only covering the surface)

nvertices [dict from string to integer, optional] maps names of surface elements to integers (not needed for volumetric CIFTI-2 files)

property affine

Affine of the volumetric image in which the greyordinate voxels were defined

classmethod from_index_mapping(*mim*)

Creates a new BrainModel axis based on a CIFTI-2 dataset

Parameters

mim [*cifti2.Cifti2MatrixIndicesMap*]

Returns

BrainModelAxis

classmethod from_mask(*mask*, *name='other'*, *affine=None*)

Creates a new BrainModelAxis axis describing the provided mask

Parameters

mask [array_like] all non-zero voxels will be included in the BrainModelAxis axis should be (Nx, Ny, Nz) array for volume mask or (Nvertex,) array for surface mask

name [str, optional] Name of the brain structure (e.g. 'CortexRight', 'thalamus_left' or 'brain_stem')

affine [array_like, optional] (4, 4) array with the voxel to mm transformation (defaults to identity matrix) Argument will be ignored for surface masks

Returns

BrainModelAxis which covers the provided mask

classmethod from_surface(*vertices*, *nvertex*, *name='Other'*)
Creates a new BrainModelAxis axis describing the vertices on a surface

Parameters

vertices [array_like] indices of the vertices on the surface

nvertex [int] total number of vertices on the surface

name [str] Name of the brain structure (e.g. 'CortexLeft' or 'CortexRight')

Returns

BrainModelAxis which covers (part of) the surface

get_element(*index*)
Describes a single element from the axis

Parameters

index [int] Indexes the row/column of interest

Returns

tuple with 3 elements

- **str, 'CIFTI_MODEL_TYPE_SURFACE' for vertex or 'CIFTI_MODEL_TYPE_VOXELS' for voxel**
- **vertex index if it is a surface element, otherwise array with 3 voxel indices**
- **structure.BrainStructure object describing the brain structure the element was taken from**

iter_structures()
Iterates over all brain structures in the order that they appear along the axis

Yields

tuple with 3 elements:

- **CIFTI-2 brain structure name**
- **slice to select the data associated with the brain structure from the tensor**
- **brain model covering that specific brain structure**

property name
The brain structure to which the voxel/vertices of belong

property surface_mask
(N,) boolean array which is true for any element on the surface

static to_cifti_brain_structure_name(*name*)
Attempts to convert the name of an anatomical region in a format recognized by CIFTI-2

This function returns:

- the name if it is in the CIFTI-2 format already
- if the name is a tuple the first element is assumed to be the structure name while the second is assumed to be the hemisphere (left, right or both). The latter will default to both.

- names like left_cortex, cortex_left, LeftCortex, or CortexLeft will be converted to CIFTI_STRUCTURE_CORTEX_LEFT

see `nibabel.cifti2.tests.test_name()` for examples of which conversions are possible

Parameters

name: iterable of 2-element tuples of integer and string input name of an anatomical region

Returns

CIFTI-2 compatible name

Raises

ValueError: raised if the input name does not match a known anatomical structure in CIFTI-2

to_mapping(*dim*)

Converts the brain model axis to a MatrixIndicesMap for storage in CIFTI-2 format

Parameters

dim [int] which dimension of the CIFTI-2 vector/matrix is described by this dataset (zero-based)

Returns

cifti2.Cifti2MatrixIndicesMap

property volume_mask

(N,) boolean array which is true for any element on the surface

property volume_shape

Shape of the volumetric image in which the greyordinate voxels were defined

LabelAxis

class nibabel.cifti2.cifti2_axes.**LabelAxis**(*name*, *label*, *meta=None*)

Bases: *nibabel.cifti2.cifti2_axes.Axis*

Defines CIFTI-2 axis for label array.

Along this axis of the CIFTI-2 vector/matrix each row/column has been given a unique name, label table, and optionally metadata

Parameters

name [array_like] (N,) string array with the parcel names

label [array_like] single dictionary or (N,) object array with dictionaries mapping from integers to (name, (R, G, B, A)), where name is a string and R, G, B, and A are floats between 0 and 1 giving the colour and alpha (i.e., transparency)

meta [array_like, optional] (N,) object array with a dictionary of metadata for each row/column

__init__(*name*, *label*, *meta=None*)

Parameters

name [array_like] (N,) string array with the parcel names

label [array_like] single dictionary or (N,) object array with dictionaries mapping from integers to (name, (R, G, B, A)), where name is a string and R, G, B, and A are floats between 0 and 1 giving the colour and alpha (i.e., transparency)

meta [array_like, optional] (N,) object array with a dictionary of metadata for each row/column

classmethod from_index_mapping(*mim*)

Creates a new Label axis based on a CIFTI-2 dataset

Parameters

mim [*cifti2.Cifti2MatrixIndicesMap*]

Returns

LabelAxis

get_element(*index*)

Describes a single element from the axis

Parameters

index [int] Indexes the row/column of interest

Returns

tuple with 2 elements

- **unicode name of the row/column**
- **dictionary with the label table**
- **dictionary with the element metadata**

to_mapping(*dim*)

Converts the hcp_labels to a MatrixIndicesMap for storage in CIFTI-2 format

Parameters

dim [int] which dimension of the CIFTI-2 vector/matrix is described by this dataset (zero-based)

Returns

cifti2.Cifti2MatrixIndicesMap

ParcelsAxis

class nibabel.cifti2.cifti2_axes.**ParcelsAxis**(*name*, *voxels*, *vertices*, *affine=None*, *volume_shape=None*, *nvertices=None*)

Bases: *nibabel.cifti2.cifti2_axes.Axis*

Each row/column in the CIFTI-2 vector/matrix represents a parcel of voxels/vertices

This Axis describes which parcel is represented by each row/column.

Individual parcels can be accessed based on their name, using `parcel = parcel_axis[name]`

Use of this constructor is not recommended. New ParcelsAxis axes can be constructed more easily from a sequence of BrainModelAxis axes using *from_brain_models()*

Parameters

name [array_like] (N,) string array with the parcel names

voxels [array_like] (N,) object array each containing a sequence of voxels. For each parcel the voxels are represented by a (M, 3) index array

vertices [array_like] (N,) object array each containing a sequence of vertices. For each parcel the vertices are represented by a mapping from brain structure name to (M,) index array

affine [array_like, optional] (4, 4) array mapping voxel indices to mm space (not needed for CIFTI-2 files only covering the surface)

volume_shape [tuple of three integers, optional] shape of the volume in which the voxels were defined (not needed for CIFTI-2 files only covering the surface)

nvertices [dict from string to integer, optional] maps names of surface elements to integers (not needed for volumetric CIFTI-2 files)

__init__(*name*, *voxels*, *vertices*, *affine=None*, *volume_shape=None*, *nvertices=None*)

Use of this constructor is not recommended. New ParcelsAxis axes can be constructed more easily from a sequence of BrainModelAxis axes using *from_brain_models()*

Parameters

name [array_like] (N,) string array with the parcel names

voxels [array_like] (N,) object array each containing a sequence of voxels. For each parcel the voxels are represented by a (M, 3) index array

vertices [array_like] (N,) object array each containing a sequence of vertices. For each parcel the vertices are represented by a mapping from brain structure name to (M,) index array

affine [array_like, optional] (4, 4) array mapping voxel indices to mm space (not needed for CIFTI-2 files only covering the surface)

volume_shape [tuple of three integers, optional] shape of the volume in which the voxels were defined (not needed for CIFTI-2 files only covering the surface)

nvertices [dict from string to integer, optional] maps names of surface elements to integers (not needed for volumetric CIFTI-2 files)

property affine

Affine of the volumetric image in which the greyordinate voxels were defined

classmethod from_brain_models(*named_brain_models*)

Creates a Parcel axis from a list of BrainModelAxis axes with names

Parameters

named_brain_models [iterable of 2-element tuples of string and BrainModelAxis] list of (parcel name, brain model representation) pairs defining each parcel

Returns

ParcelsAxis

classmethod from_index_mapping(*mim*)

Creates a new Parcels axis based on a CIFTI-2 dataset

Parameters

mim [cifti2.Cifti2MatrixIndicesMap]

Returns

ParcelsAxis

get_element(*index*)
Describes a single element from the axis

Parameters

index [int] Indexes the row/column of interest

Returns

tuple with 3 elements

- **unicode name of the parcel**
- **(M, 3) int array with voxel indices**
- **dict from string to (K,) int array with vertex indices** for a specific surface brain structure

to_mapping(*dim*)
Converts the Parcel to a MatrixIndicesMap for storage in CIFTI-2 format

Parameters

dim [int] which dimension of the CIFTI-2 vector/matrix is described by this dataset (zero-based)

Returns

cifti2.Cifti2MatrixIndicesMap

property volume_shape
Shape of the volumetric image in which the greyordinate voxels were defined

ScalarAxis

class nibabel.cifti2.cifti2_axes.**ScalarAxis**(*name*, *meta=None*)
Bases: *nibabel.cifti2.cifti2_axes.Axis*

Along this axis of the CIFTI-2 vector/matrix each row/column has been given a unique name and optionally metadata

Parameters

name [array_like] (N,) string array with the parcel names

meta [array_like] (N,) object array with a dictionary of metadata for each row/column. Defaults to empty dictionary

__init__(*name*, *meta=None*)

Parameters

name [array_like] (N,) string array with the parcel names

meta [array_like] (N,) object array with a dictionary of metadata for each row/column. Defaults to empty dictionary

classmethod from_index_mapping(*mim*)
Creates a new Scalar axis based on a CIFTI-2 dataset

Parameters

mim [*cifti2.Cifti2MatrixIndicesMap*]

Returns

ScalarAxis

get_element(*index*)
Describes a single element from the axis

Parameters

index [int] Indexes the row/column of interest

Returns

tuple with 2 elements

- **unicode name of the row/column**
- **dictionary with the element metadata**

to_mapping(*dim*)
Converts the hcp_labels to a MatrixIndicesMap for storage in CIFTI-2 format

Parameters

dim [int] which dimension of the CIFTI-2 vector/matrix is described by this dataset (zero-based)

Returns

cifti2.Cifti2MatrixIndicesMap

SeriesAxis

class nibabel.cifti2.cifti2_axes.**SeriesAxis**(*start*, *step*, *size*, *unit='SECOND'*)
Bases: *nibabel.cifti2.cifti2_axes.Axis*

Along this axis of the CIFTI-2 vector/matrix the rows/columns increase monotonously in time

This Axis describes the time point of each row/column.

Creates a new SeriesAxis axis

Parameters

start [float] starting time point

step [float] sampling time (TR)

size [int] number of time points

unit [str] Unit of the step size (one of 'second', 'hertz', 'meter', or 'radian')

__init__(*start*, *step*, *size*, *unit='SECOND'*)
Creates a new SeriesAxis axis

Parameters

start [float] starting time point

step [float] sampling time (TR)

size [int] number of time points

unit [str] Unit of the step size (one of 'second', 'hertz', 'meter', or 'radian')

classmethod from_index_mapping(*mim*)
Creates a new SeriesAxis axis based on a CIFTI-2 dataset

Parameters

mim [*cifti2.Cifti2MatrixIndicesMap*]

Returns

SeriesAxis

get_element(*index*)
Gives the time point of a specific row/column

Parameters

index [int] Indexes the row/column of interest

Returns

float

size = None

property time

to_mapping(*dim*)
Converts the SeriesAxis to a MatrixIndicesMap for storage in CIFTI-2 format

Parameters

dim [int] which dimension of the CIFTI-2 vector/matrix is described by this dataset (zero-based)

Returns

cifti2.Cifti2MatrixIndicesMap

property unit

from_index_mapping

nibabel.cifti2.cifti2_axes.**from_index_mapping**(*mim*)
Parses the MatrixIndicesMap to find the appropriate CIFTI-2 axis describing the rows or columns

Parameters

mim [*cifti2.Cifti2MatrixIndicesMap*]

Returns

axis [subclass of *Axis*]

to_header

nibabel.cifti2.cifti2_axes.**to_header**(*axes*)

Converts the axes describing the rows/columns of a CIFTI-2 vector/matrix to a Cifti2Header

Parameters

axes [iterable of *Axis* objects] one or more axes describing each dimension in turn

Returns

header [*cifti2.Cifti2Header*]

Cifti2Extension

class nibabel.cifti2.parse_cifti2.**Cifti2Extension**(*code=None*, *content=None*)

Bases: *nibabel.nifti1.Nifti1Extension*

Parameters

code [int or str] Canonical extension code as defined in the NIfTI standard, given either as integer or corresponding label (see `extension_codes`)

content [str] Extension content as read from the NIfTI file header. This content is converted into a runtime representation.

__init__(*code=None*, *content=None*)

Parameters

code [int or str] Canonical extension code as defined in the NIfTI standard, given either as integer or corresponding label (see `extension_codes`)

content [str] Extension content as read from the NIfTI file header. This content is converted into a runtime representation.

code = 32

Cifti2Parser

class nibabel.cifti2.parse_cifti2.**Cifti2Parser**(*encoding=None*, *buffer_size=3500000*, *verbose=0*)

Bases: *nibabel.xmlutils.XmlParser*

Class to parse an XML string into a CIFTI-2 header object

Parameters

encoding [str] string containing xml document

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

verbose [int, optional] amount of output during parsing (0=silent, by default).

__init__(*encoding=None*, *buffer_size=3500000*, *verbose=0*)

Parameters

encoding [str] string containing xml document

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

verbose [int, optional] amount of output during parsing (0=silent, by default).

CharacterDataHandler(*data*)
Collect character data chunks pending collation

The parser breaks the data up into chunks of size depending on the buffer_size of the parser. A large bit of character data, with standard parser buffer_size (such as 8K) can easily span many calls to this function. We thus collect the chunks and process them when we hit start or end tags.

EndElementHandler(*name*)

StartElementHandler(*name*, *attrs*)

flush_chardata()
Collate and process collected character data

property pending_data
True if there is character data pending for processing

gifti

GIfTI format IO

giftiio	

gifti	Classes defining Gifti objects

Module: gifti.gifti

Classes defining Gifti objects

The Gifti specification was (at time of writing) available as a PDF download from http://www.nitrc.org/projects/gifti/

GiftiCoordSystem([dataspace, xformspace, xform])	Gifti coordinate system transform matrix
GiftiDataArray([data, intent, datatype, ...])	Container for Gifti numerical data array and associated metadata
GiftiImage([header, extra, file_map, meta, ...])	GIFTI image object
GiftiLabel([key, red, green, blue, alpha])	Gifti label: association of integer key with optional RGBA values
GiftiLabelTable()	Gifti label table: a sequence of key, label pairs
GiftiMetaData([nvpair])	A sequence of GiftiNVPairs containing metadata for a gifti data array
GiftiNVPairs([name, value])	Gifti name / value pairs
data_tag(dataarray, encoding, datatype, ordering)	data_tag is an internal API that will be discontinued.

Module: gifti.giftiio

read(filename)	Load a Gifti image from a file
write(image, filename)	Save the current image to a new file

Module: gifti.parse_gifti_fast

GiftiImageParser([encoding, buffer_size, ...])	

Parameters

GiftiParseError	Gifti-specific parsing error
Outputter()	Outputter class deprecated.
parse_gifti_file([fname, fptr, buffer_size])	parse_gifti_file deprecated.
read_data_block(darray, fname, data, mmap)	Parses data from a <Data> element, or loads from an external file.

Module: gifti.util

GiftiCoordSystem

class nibabel.gifti.gifti.**GiftiCoordSystem**(*dataspace=0*, *xformspace=0*, *xform=None*)

Bases: *nibabel.xmlutils.XmlSerializable*

Gifti coordinate system transform matrix

Quotes are from the gifti spec dated 2011-01-14.

> "For a DataArray with an Intent NIFTI_INTENT_POINTSET, this element describes the stereotaxic space of the data before and after the application of a transformation matrix. The most common stereotaxic space is the Talairach Space that places the origin at the anterior commissure and the negative X, Y, and Z axes correspond to left, posterior, and inferior respectively. At least one CoordinateSystemTransformMatrix is required in a DataArray with an intent of NIFTI_INTENT_POINTSET. Multiple CoordinateSystemTransformMatrix elements may be used to describe the transformation to multiple spaces."

Attributes

dataspace [int] From the spec: Contains the stereotaxic space of a DataArray's data prior to application of the transformation matrix. The stereotaxic space should be one of:

- NIFTI_XFORM_UNKNOWN
- NIFTI_XFORM_SCANNER_ANAT
- NIFTI_XFORM_ALIGNED_ANAT
- NIFTI_XFORM_TALAIRACH
- NIFTI_XFORM_MNI_152

xformspace [int] Spec: "Contains the stereotaxic space of a DataArray's data after application of the transformation matrix. See the DataSpace element for a list of stereotaxic spaces."

xform [array-like shape (4, 4)] Affine transformation matrix

__init__(*dataspace=0*, *xformspace=0*, *xform=None*)
Initialize self. See help(type(self)) for accurate signature.

print_summary()

GiftiDataArray

class nibabel.gifti.gifti.**GiftiDataArray**(*data=None*, *intent='NIFTI_INTENT_NONE'*, *datatype=None*, *encoding='GIFTI_ENCODING_B64GZ'*, *endian='little'*, *coordsys=None*, *ordering='C'*, *meta=None*, *ext_fname=''*, *ext_offset=0*)

Bases: *nibabel.xmlutils.XmlSerializable*

Container for Gifti numerical data array and associated metadata

Quotes are from the gifti spec dated 2011-01-14.

Description of DataArray in spec: "This element contains the numeric data and its related metadata. The CoordinateSystemTransformMatrix child is only used when the DataArray's Intent is NIFTI_INTENT_POINTSET. FileName and FileOffset are required if the data is stored in an external file."

Attributes

darray [None or ndarray] Data array

intent [int] NIFTI intent code, see nifti1.intent_codes

datatype [int] NIFTI data type codes, see nifti1.data_type_codes. From the spec: "This required attribute describes the numeric type of the data contained in a Data Array and are limited to the types displayed in the table:

NIFTI_TYPE_UINT8 : Unsigned, 8-bit bytes. NIFTI_TYPE_INT32 : Signed, 32-bit integers. NIFTI_TYPE_FLOAT32 : 32-bit single precision floating point."

At the moment, we do not enforce that the datatype is one of these three.

encoding [string] Encoding of the data, see util.gifti_encoding_codes; default is GIFTI_ENCODING_B64GZ.

endian [string] The Endianness to store the data array. Should correspond to the machine endianness. Default is system byteorder.

coordsys [*GiftiCoordSystem* instance] Input and output coordinate system with tranformation matrix between the two.

ind_ord [int] The ordering of the array. see util.array_index_order_codes. Default is RowMajorOrder - C ordering

meta [*GiftiMetaData* instance] An instance equivalent to a dictionary for metadata information.

ext_fname [str] Filename in which data is stored, or empty string if no corresponding filename.

ext_offset [int] Position in bytes within *ext_fname* at which to start reading data.

Returns a shell object that cannot be saved.

__init__(*data=None*, *intent='NIFTI_INTENT_NONE'*, *datatype=None*, *encoding='GIFTI_ENCODING_B64GZ'*, *endian='little'*, *coordsys=None*, *ordering='C'*, *meta=None*, *ext_fname=''*, *ext_offset=0*)

Returns a shell object that cannot be saved.

classmethod from_array(*darray*, *intent='NIFTI_INTENT_NONE'*, *datatype=None*, *encoding='GIFTI_ENCODING_B64GZ'*, *endian='little'*, *coordsys=None*, *ordering='C'*, *meta=None*)

Creates a new Gifti data array

from_array method is deprecated. Please use GiftiDataArray constructor instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

Parameters

darray [ndarray] NumPy data array

intent [string] NIFTI intent code, see nifti1.intent_codes

datatype [None or string, optional] NIFTI data type codes, see nifti1.data_type_codes If None, the datatype of the NumPy array is taken.

encoding [string, optionaal] Encoding of the data, see util.gifti_encoding_codes; default: GIFTI_ENCODING_B64GZ

endian [string, optional] The Endianness to store the data array. Should correspond to the machine endianness. default: system byteorder

coordsys [GiftiCoordSystem, optional] If None, a identity transformation is taken.

ordering [string, optional] The ordering of the array. see util.array_index_order_codes; default: RowMajorOrder - C ordering

meta [None or dict, optional] A dictionary for metadata information. If None, gives empty dict.

Returns

da [instance of our own class]

get_metadata()

get_metadata method deprecated. Use the metadata property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

property metadata

Returns metadata as dictionary

property num_dim

print_summary()

to_xml_close()

to_xml_close method deprecated. Use the to_xml() function instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

to_xml_open()
to_xml_open method deprecated. Use the to_xml() function instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

GiftiImage

class nibabel.gifti.gifti.**GiftiImage**(*header=None*, *extra=None*, *file_map=None*, *meta=None*, *labeltable=None*, *darrays=None*, *version='1.0'*)

Bases: *nibabel.xmlutils.XmlSerializable*, *nibabel.filebasedimages.SerializableImage*

GIFTI image object

The Gifti spec suggests using the following suffixes to your filename when saving each specific type of data:

.gii Generic GIFTI File

.coord.gii Coordinates

.func.gii Functional

.label.gii Labels

.rgba.gii RGB or RGBA

.shape.gii Shape

.surf.gii Surface

.tensor.gii Tensors

.time.gii Time Series

.topo.gii Topology

The Gifti file is stored in endian convention of the current machine.

__init__(*header=None*, *extra=None*, *file_map=None*, *meta=None*, *labeltable=None*, *darrays=None*, *version='1.0'*)
Initialize self. See help(type(self)) for accurate signature.

add_gifti_data_array(*dataarr*)
Adds a data array to the GiftiImage

Parameters

dataarr [*GiftiDataArray* instance]

agg_data(*intent_code=None*)
Aggregate GIFTI data arrays into an ndarray or tuple of ndarray

In the general case, the numpy data array is extracted from each `GiftiDataArray` object and returned in a `tuple`, in the order they are found in the GIFTI image.

If all `GiftiDataArray` s have `intent` of 2001 (`NIFTI_INTENT_TIME_SERIES`), then the data arrays are concatenated as columns, producing a vertex-by-time array. If an `intent_code` is passed, data arrays are filtered by the selected intents, before being aggregated. This may be useful for images containing several intents, or ensuring an expected data type in an image of uncertain provenance. If `intent_code` is a `tuple`, then a `tuple` will be returned with the result of `agg_data` for each element, in order. This may be useful for ensuring that expected data arrives in a consistent order.

Parameters

intent_code [None, string, integer or tuple of strings or integers, optional] code(s) specifying nifti intent

Returns

tuple of ndarrays or ndarray If the input is a tuple, the returned tuple will match the order.

Examples

Consider a surface GIFTI file:

```
>>> import nibabel as nib
>>> from nibabel.testing import test_data
>>> surf_img = nib.load(test_data('gifti', 'ascii.gii'))
```

The coordinate data, which is indicated by the `NIFTI_INTENT_POINTSET` intent code, may be retrieved using any of the following equivalent calls:

```
>>> coords = surf_img.agg_data('NIFTI_INTENT_POINTSET')
>>> coords_2 = surf_img.agg_data('pointset')
>>> coords_3 = surf_img.agg_data(1008)  # Numeric code for pointset
>>> print(np.array2string(coords, precision=3))
[[-16.072 -66.188  21.267]
 [-16.706 -66.054  21.233]
 [-17.614 -65.402  21.071]]
>>> np.array_equal(coords, coords_2)
True
>>> np.array_equal(coords, coords_3)
True
```

Similarly, the triangle mesh can be retrieved using various intent specifiers:

```
>>> triangles = surf_img.agg_data('NIFTI_INTENT_TRIANGLE')
>>> triangles_2 = surf_img.agg_data('triangle')
>>> triangles_3 = surf_img.agg_data(1009)  # Numeric code for pointset
>>> print(np.array2string(triangles))
[0 1 2]
>>> np.array_equal(triangles, triangles_2)
True
>>> np.array_equal(triangles, triangles_3)
True
```

All arrays can be retrieved as a `tuple` by omitting the intent code:

```
>>> coords_4, triangles_4 = surf_img.agg_data()
>>> np.array_equal(coords, coords_4)
True
>>> np.array_equal(triangles, triangles_4)
True
```

Finally, a tuple of intent codes may be passed in order to select the arrays in a specific order:

```
>>> triangles_5, coords_5 = surf_img.agg_data(('triangle', 'pointset'))
>>> np.array_equal(triangles, triangles_5)
True
```

(continues on next page)

(continued from previous page)

```
>>> np.array_equal(coords, coords_5)
True
```

The following image is a GIFTI file with ten (10) data arrays of the same size, and with intent code 2001 (`NIFTI_INTENT_TIME_SERIES`):

```
>>> func_img = nib.load(test_data('gifti', 'task.func.gii'))
```

When aggregating time series data, these arrays are concatenated into a single, vertex-by-timestep array:

```
>>> series = func_img.agg_data()
>>> series.shape
(642, 10)
```

In the case of a GIFTI file with unknown data arrays, it may be preferable to specify the intent code, so that a time series array is always returned:

```
>>> series_2 = func_img.agg_data('NIFTI_INTENT_TIME_SERIES')
>>> series_3 = func_img.agg_data('time series')
>>> series_4 = func_img.agg_data(2001)
>>> np.array_equal(series, series_2)
True
>>> np.array_equal(series, series_3)
True
>>> np.array_equal(series, series_4)
True
```

Requesting a data array from a GIFTI file with no matching intent codes will result in an empty tuple:

```
>>> surf_img.agg_data('time series')
()
>>> func_img.agg_data('triangle')
()
```

files_types = (('image', '.gii'),)

classmethod from_file_map(*file_map*, *buffer_size=35000000*, *mmap=True*)

Load a Gifti image from a file_map

Parameters

file_map [dict] Dictionary with single key `image` with associated value which is a `FileHolder` instance pointing to the image file.

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

mmap [{True, False, 'c', 'r', 'r+'}] Controls the use of numpy memory mapping for reading data. Only has an effect when loading GIFTI images with data stored in external files (`DataArray` elements with an `Encoding` equal to `ExternalFileBinary`). If `False`, do not try numpy `memmap` for data array. If one of `{'c', 'r', 'r+'}`, try numpy `memmap` with `mode=mmap`. A *mmap* value of `True` gives the same behavior as `mmap='c'`. If the file cannot be memory-mapped, ignore *mmap* value and read array from file.

Returns

img [GiftiImage]

classmethod from_filename(*filename*, *buffer_size=35000000*, *mmap=True*)

getArraysFromIntent(*intent*)
getArraysFromIntent method deprecated. Use get_arrays_from_intent instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

get_arrays_from_intent(*intent*)
Return list of GiftiDataArray elements matching given intent

get_labeltable()
get_labeltable method deprecated. Use the gifti_img.labeltable property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

get_meta()
get_meta method deprecated. Use the gifti_img.meta property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

property labeltable

property meta

property numDA

parser
alias of *nibabel.gifti.parse_gifti_fast.GiftiImageParser*

print_summary()

remove_gifti_data_array(*ith*)
Removes the ith data array element from the GiftiImage

remove_gifti_data_array_by_intent(*intent*)
Removes all the data arrays with the given intent type

set_labeltable(*labeltable*)
set_labeltable method deprecated. Use the gifti_img.labeltable property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

set_metadata(*meta*)
set_meta method deprecated. Use the gifti_img.meta property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

to_bytes(*enc='utf-8'*)
Return XML corresponding to image content

to_file_map(*file_map=None*)
Save the current image to the specified file_map

Parameters

file_map [dict] Dictionary with single key `image` with associated value which is a `FileHolder` instance pointing to the image file.

Returns

None

to_xml(*enc='utf-8'*)
Return XML corresponding to image content

valid_exts = ('.gii',)

GiftiLabel

class nibabel.gifti.gifti.**GiftiLabel**(*key=0*, *red=None*, *green=None*, *blue=None*, *alpha=None*)
Bases: *nibabel.xmlutils.XmlSerializable*

Gifti label: association of integer key with optional RGBA values

Quotes are from the gifti spec dated 2011-01-14.

Notes

freesurfer examples seem not to conform to datatype "NIFTI_TYPE_RGBA32" because they are floats, not 4 8-bit integers.

Attributes

key [int] (From the spec): "This required attribute contains a non-negative integer value. If a DataArray's Intent is NIFTI_INTENT_LABEL and a value in the DataArray is 'X', its corresponding label is the label with the Key attribute containing the value 'X'. In early versions of the GIFTI file format, the attribute Index was used instead of Key. If an Index attribute is encountered, it should be processed like the Key attribute."

red [None or float] Optional value for red.

green [None or float] Optional value for green.

blue [None or float] Optional value for blue.

alpha [None or float] Optional value for alpha.

__init__(*key=0*, *red=None*, *green=None*, *blue=None*, *alpha=None*)
Initialize self. See help(type(self)) for accurate signature.

get_rgba()
get_rgba method deprecated. Use the rgba property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

property rgba
Returns RGBA as tuple

GiftiLabelTable

class nibabel.gifti.gifti.**GiftiLabelTable**
Bases: *nibabel.xmlutils.XmlSerializable*

Gifti label table: a sequence of key, label pairs

From the gifti spec dated 2011-01-14: The label table is used by DataArrays whose values are an key into the LabelTable's labels. A file should contain at most one LabelTable and it must be located in the file prior to any DataArray elements.

__init__()
Initialize self. See help(type(self)) for accurate signature.

get_labels_as_dict()

print_summary()

GiftiMetaData

class nibabel.gifti.gifti.**GiftiMetaData**(*nvpair=None*)
Bases: *nibabel.xmlutils.XmlSerializable*

A sequence of GiftiNVPairs containing metadata for a gifti data array

__init__(*nvpair=None*)
Initialize self. See help(type(self)) for accurate signature.

classmethod from_dict(*data_dict*)

get_metadata()
get_metadata method deprecated. Use the metadata property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

property metadata
Returns metadata as dictionary

print_summary()

GiftiNVPairs

class nibabel.gifti.gifti.**GiftiNVPairs**(*name=''*, *value=''*)
Bases: object

Gifti name / value pairs

Attributes

name [str]

value [str]

__init__(*name=''*, *value=''*)
Initialize self. See help(type(self)) for accurate signature.

data_tag

nibabel.gifti.gifti.**data_tag**(*dataarray*, *encoding*, *datatype*, *ordering*)

data_tag is an internal API that will be discontinued.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

read

nibabel.gifti.giftiio.**read**(*filename*)

Load a Gifti image from a file

giftiio.read function deprecated. Use nibabel.load() instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

Parameters

filename [string] The Gifti file to open, it has usually ending .gii

Returns

img [GiftiImage] Returns a GiftiImage

write

nibabel.gifti.giftiio.**write**(*image*, *filename*)

Save the current image to a new file

giftiio.write function deprecated. Use nibabel.load() instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

Parameters

image [GiftiImage] A GiftiImage instance to store

filename [string] Filename to store the Gifti file to

Returns

None

Notes

We write all files with utf-8 encoding, and specify this at the top of the XML file with the `encoding` attribute.

The Gifti spec suggests using the following suffixes to your filename when saving each specific type of data:

.gii Generic GIFTI File

.coord.gii Coordinates

.func.gii Functional

.label.gii Labels

.rgba.gii RGB or RGBA

.shape.gii Shape

.surf.gii Surface

.tensor.gii Tensors

.time.gii Time Series

.topo.gii Topology

The Gifti file is stored in endian convention of the current machine.

GiftiImageParser

class nibabel.gifti.parse_gifti_fast.**GiftiImageParser**(*encoding=None*, *buffer_size=35000000*, *verbose=0*, *mmap=True*)

Bases: *nibabel.xmlutils.XmlParser*

Parameters

encoding [str] string containing xml document

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

verbose [int, optional] amount of output during parsing (0=silent, by default).

__init__(*encoding=None*, *buffer_size=35000000*, *verbose=0*, *mmap=True*)

Parameters

encoding [str] string containing xml document

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

verbose [int, optional] amount of output during parsing (0=silent, by default).

CharacterDataHandler(*data*)

Collect character data chunks pending collation

The parser breaks the data up into chunks of size depending on the buffer_size of the parser. A large bit of character data, with standard parser buffer_size (such as 8K) can easily span many calls to this function. We thus collect the chunks and process them when we hit start or end tags.

EndElementHandler(*name*)

StartElementHandler(*name*, *attrs*)

flush_chardata()
Collate and process collected character data

property pending_data
True if there is character data pending for processing

GiftiParseError

class nibabel.gifti.parse_gifti_fast.**GiftiParseError**
Bases: xml.parsers.expat.ExpatError

Gifti-specific parsing error

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Outputter

class nibabel.gifti.parse_gifti_fast.**Outputter**
Bases: *nibabel.gifti.parse_gifti_fast.GiftiImageParser*

Outputter class deprecated. Use GiftiImageParser instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

__init__()
Outputter class deprecated. Use GiftiImageParser instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

initialize()
Initialize outputter

parse_gifti_file

nibabel.gifti.parse_gifti_fast.**parse_gifti_file**(*fname=None*, *fptr=None*, *buffer_size=None*)
parse_gifti_file deprecated. Use GiftiImageParser.parse() instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

read_data_block

nibabel.gifti.parse_gifti_fast.**read_data_block**(*darray*, *fname*, *data*, *mmap*)

Parses data from a <Data> element, or loads from an external file.

Parameters

darray [GiftiDataArray] GiftiDataArray object representing the parent <DataArray> of this <Data> element

fname [str or None] Name of GIFTI file being loaded, or None if in-memory

data [str or None] Data to parse, or None if data is in an external file

mmap [{True, False, 'c', 'r', 'r+'}] Controls the use of numpy memory mapping for reading data. Only has an effect when loading GIFTI images with data stored in external files (`DataArray` elements with an `Encoding` equal to `ExternalFileBinary`). If `False`, do not try numpy `memmap` for data array. If one of `{'c', 'r', 'r+'}`, try numpy `memmap` with `mode=mmap`. A *mmap* value of `True` gives the same behavior as `mmap='c'`. If the file cannot be memory-mapped, ignore *mmap* value and read array from file.

Returns

numpy.ndarray or numpy.memmap containing the parsed data

freesurfer

Reading functions for freesurfer files

Module: freesurfer.io

Read / write FreeSurfer geometry, morphometry, label, annotation formats

read_annot(filepath[, orig_ids])	Read in a Freesurfer annotation from a `.annot` file.
read_geometry(filepath[, read_metadata, ...])	Read a triangular format Freesurfer surface mesh.
read_label(filepath[, read_scalars])	Load in a Freesurfer .label file.
read_morph_data(filepath)	Read a Freesurfer morphometry data file.
write_annot(filepath, labels, ctab, names[, ...])	Write out a "new-style" Freesurfer annotation file.
write_geometry(filepath, coords, faces[, ...])	Write a triangular format Freesurfer surface mesh.
write_morph_data(file_like, values[, fnum])	Write Freesurfer morphometry data *values* to file-like *file_like*

Module: `freesurfer.mghformat`

Header and image reading / writing functions for MGH image format

Author: Krish Subramaniam

MGHError	Exception for MGH format related problems.
MGHHeader([binaryblock, check])	Class for MGH format header
MGHImage(dataobj, affine[, header, extra, ...])	Class for MGH format image

read_annot

nibabel.freesurfer.io.**read_annot**(*filepath*, *orig_ids=False*)

Read in a Freesurfer annotation from a `.annot` file.

An `.annot` file contains a sequence of vertices with a label (also known as an "annotation value") associated with each vertex, and then a sequence of colors corresponding to each label.

Annotation file format versions 1 and 2 are supported, corresponding to the "old-style" and "new-style" color table layout.

Note that the output color table `ctab` is in RGBT form, where T (transparency) is 255 - alpha.

See:

- https://surfer.nmr.mgh.harvard.edu/fswiki/LabelsClutsAnnotationFiles#Annotation
- https://github.com/freesurfer/freesurfer/blob/dev/matlab/read_annotation.m
- https://github.com/freesurfer/freesurfer/blob/8b88b34/utils/colortab.c

Parameters

filepath [str] Path to annotation file.

orig_ids [bool] Whether to return the vertex ids as stored in the annotation file or the positional colortable ids. With orig_ids=False vertices with no id have an id set to -1.

Returns

labels [ndarray, shape (n_vertices,)] Annotation id at each vertex. If a vertex does not belong to any label and orig_ids=False, its id will be set to -1.

ctab [ndarray, shape (n_labels, 5)] RGBT + label id colortable array.

names [list of bytes] The names of the labels. The length of the list is n_labels.

read_geometry

nibabel.freesurfer.io.**read_geometry**(*filepath*, *read_metadata=False*, *read_stamp=False*)

Read a triangular format Freesurfer surface mesh.

Parameters

filepath [str] Path to surface file.

read_metadata [bool, optional] If True, read and return metadata as key-value pairs.

Valid keys:

- 'head' : array of int
- 'valid' : str
- 'filename' : str
- 'volume' : array of int, shape (3,)
- 'voxelsize' : array of float, shape (3,)
- 'xras' : array of float, shape (3,)
- 'yras' : array of float, shape (3,)
- 'zras' : array of float, shape (3,)
- 'cras' : array of float, shape (3,)

read_stamp [bool, optional] Return the comment from the file

Returns

coords [numpy array] nvtx x 3 array of vertex (x, y, z) coordinates.

faces [numpy array] nfaces x 3 array of defining mesh triangles.

volume_info [OrderedDict] Returned only if *read_metadata* is True. Key-value pairs found in the geometry file.

create_stamp [str] Returned only if *read_stamp* is True. The comment added by the program that saved the file.

read_label

nibabel.freesurfer.io.**read_label**(*filepath*, *read_scalars=False*)

Load in a Freesurfer .label file.

Parameters

filepath [str] Path to label file.

read_scalars [bool, optional] If True, read and return scalars associated with each vertex.

Returns

label_array [numpy array] Array with indices of vertices included in label.

scalar_array [numpy array (floats)] Only returned if *read_scalars* is True. Array of scalar data for each vertex.

read_morph_data

nibabel.freesurfer.io.**read_morph_data**(*filepath*)

Read a Freesurfer morphometry data file.

This function reads in what Freesurfer internally calls "curv" file types, (e.g. ?h. curv, ?h.thickness), but as that has the potential to cause confusion where "curv" also refers to the surface curvature values, we refer to these files as "morphometry" files with PySurfer.

Parameters

filepath [str] Path to morphometry file

Returns

curv [numpy array] Vector representation of surface morpometry values

write_annot

nibabel.freesurfer.io.**write_annot**(*filepath*, *labels*, *ctab*, *names*, *fill_ctab=True*)

Write out a "new-style" Freesurfer annotation file.

Note that the color table `ctab` is in RGBT form, where T (transparency) is 255 - alpha.

See:

- https://surfer.nmr.mgh.harvard.edu/fswiki/LabelsClutsAnnotationFiles#Annotation
- https://github.com/freesurfer/freesurfer/blob/dev/matlab/write_annotation.m
- https://github.com/freesurfer/freesurfer/blob/8b88b34/utils/colortab.c

Parameters

filepath [str] Path to annotation file to be written

labels [ndarray, shape (n_vertices,)] Annotation id at each vertex.

ctab [ndarray, shape (n_labels, 5)] RGBT + label id colortable array.

names [list of str] The names of the labels. The length of the list is n_labels.

fill_ctab [{True, False} optional] If True, the annotation values for each vertex are automatically generated. In this case, the provided *ctab* may have shape (n_labels, 4) or (n_labels, 5) - if the latter, the final column is ignored.

write_geometry

nibabel.freesurfer.io.**write_geometry**(*filepath*, *coords*, *faces*, *create_stamp=None*, *volume_info=None*)

Write a triangular format Freesurfer surface mesh.

Parameters

filepath [str] Path to surface file.

coords [numpy array] nvtx x 3 array of vertex (x, y, z) coordinates.

faces [numpy array] nfaces x 3 array of defining mesh triangles.

create_stamp [str, optional] User/time stamp (default: "created by <user> on <ctime>")

volume_info [dict-like or None, optional] Key-value pairs to encode at the end of the file.

Valid keys:

- 'head' : array of int
- 'valid' : str
- 'filename' : str
- 'volume' : array of int, shape (3,)
- 'voxelsize' : array of float, shape (3,)
- 'xras' : array of float, shape (3,)
- 'yras' : array of float, shape (3,)

- 'zras' : array of float, shape (3,)
- 'cras' : array of float, shape (3,)

write_morph_data

nibabel.freesurfer.io.**write_morph_data**(*file_like*, *values*, *fnum=0*)

Write Freesurfer morphometry data *values* to file-like *file_like*

Equivalent to FreeSurfer's write_curv.m

See also: http://www.grahamwideman.com/gw/brain/fs/surfacefileformats.htm#CurvNew

Parameters

file_like [file-like] String containing path of file to be written, or file-like object, open in binary write (*'wb'* mode, implementing the *write* method)

values [array-like] Surface morphometry values. Shape must be (N,), (N, 1), (1, N) or (N, 1, 1)

fnum [int, optional] Number of faces in the associated surface.

MGHError

class nibabel.freesurfer.mghformat.**MGHError**

Bases: Exception

Exception for MGH format related problems.

To be raised whenever MGH is not happy, or we are not happy with MGH.

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

MGHHeader

class nibabel.freesurfer.mghformat.**MGHHeader**(*binaryblock=None*, *check=True*)

Bases: *nibabel.wrapstruct.LabeledWrapStruct*

Class for MGH format header

The header also consists of the footer data which MGH places after the data chunk.

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

check [bool, optional] Whether to check content of header in initialization. Default is True.

__init__(*binaryblock=None*, *check=True*)

Initialize header from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into header. By default, None, in which case we insert the default empty header block

check [bool, optional] Whether to check content of header in initialization. Default is True.

as_byteswapped(*endianness=None*)
Return new object with given `endianness`

If big endian, returns a copy of the object. Otherwise raises ValueError.

Parameters

endianness [None or string, optional] endian code to which to swap. None means swap from current endianness, and is the default

Returns

wstr [`MGHHeader`] `MGHHeader` object

static chk_version(*hdr*, *fix=False*)

copy()
Return copy of structure

data_from_fileobj(*fileobj*)
Read data array from *fileobj*

Parameters

fileobj [file-like] Must be open, and implement `read` and `seek` methods

Returns

arr [ndarray] data array

classmethod default_structarr(*endianness=None*)
Return header data for empty header

Ignores byte order; always big endian

classmethod diagnose_binaryblock(*binaryblock*, *endianness=None*)
Run checks over binary data, return string

classmethod from_fileobj(*fileobj*, *check=True*)
classmethod for loading a MGH fileobject

classmethod from_header(*header=None*, *check=True*)
Class method to create MGH header from another MGH header

get_affine()
Get the affine transform from the header information.

MGH format doesn't store the transform directly. Instead it's gleaned from the zooms (delta), direction cosines (Mdc), RAS centers (Pxyz_c) and the dimensions.

get_best_affine()
Get the affine transform from the header information.

MGH format doesn't store the transform directly. Instead it's gleaned from the zooms (delta), direction cosines (Mdc), RAS centers (Pxyz_c) and the dimensions.

get_data_bytespervox()
Get the number of bytes per voxel of the data

get_data_dtype()
Get numpy dtype for MGH data

For examples see `set_data_dtype`

get_data_offset()
Return offset into data file to read data

get_data_shape()
Get shape of data

get_data_size()
Get the number of bytes the data chunk occupies.

get_footer_offset()
Return offset where the footer resides. Occurs immediately after the data chunk.

get_ras2vox()
return the inverse get_affine()

get_slope_inter()
MGH format does not do scaling?

get_vox2ras()
return the get_affine()

get_vox2ras_tkr()
Get the vox2ras-tkr transform. See "Torig" here: https://surfer.nmr.mgh.harvard.edu/fswiki/CoordinateSystems

get_zooms()
Get zooms from header

Returns the spacing of voxels in the x, y, and z dimensions. For four-dimensional files, a fourth zoom is included, equal to the repetition time (TR) in ms (see The MGH/MGZ Volume Format).

To access only the spatial zooms, use *hdr['delta']*.

Returns

z [tuple] tuple of header zoom values

classmethod guessed_endian(*mapping*)
MGHHeader data must be big-endian

set_data_dtype(*datatype*)
Set numpy dtype for data from code or dtype or type

set_data_shape(*shape*)
Set shape of data

Parameters

shape [sequence] sequence of integers specifying data array shape

set_zooms(*zooms*)
Set zooms into header fields

Sets the spacing of voxels in the x, y, and z dimensions. For four-dimensional files, a temporal zoom (repetition time, or TR, in ms) may be provided as a fourth sequence element.

Parameters

zooms [sequence] sequence of floats specifying spatial and (optionally) temporal zooms

template_dtype = dtype([('version', '>i4'), ('dims', '>i4', (4,)), ('type', '>i4'), ('

writeftr_to(*fileobj*)
Write footer to fileobj

Footer data is located after the data chunk. So move there and write.

Parameters

fileobj [file-like object] Should implement `write` and `seek` method

Returns

None

writehdr_to(*fileobj*)
Write header to fileobj

Write starts at the beginning.

Parameters

fileobj [file-like object] Should implement `write` and `seek` method

Returns

None

MGHImage

class nibabel.freesurfer.mghformat.**MGHImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.spatialimages.SpatialImage*, *nibabel.filebasedimages.SerializableImage*

Class for MGH format image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

ImageArrayProxy
alias of *nibabel.arrayproxy.ArrayProxy*

files_types = (('image', '.mgh'),)

classmethod filespec_to_file_map(*filespec*)
Make *file_map* for this class from filename *filespec*

Class method

Parameters

filespec [str or os.PathLike] Filename that might be for this image file type.

Returns

file_map [dict] *file_map* dict with (key, value) pairs of (`file_type`, FileHolder instance), where `file_type` is a string giving the type of the contained file.

Raises

ImageFileError if *filespec* is not recognizable as being a filename for this image type.

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)
Class method to create image from mapping in `file_map`

Parameters

file_map [dict] Mapping with (kay, value) pairs of (`file_type`, FileHolder instance giving file-likes for each file needed for this image type.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_map` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [MGHImage instance]

header_class
alias of *nibabel.freesurfer.mghformat.MGHHeader*

makeable = True

rw = True

to_file_map(*file_map=None*)
Write image to *file_map* or contained `self.file_map`

Parameters

file_map [None or mapping, optional] files mapping. If None (default) use object's `file_map` attribute instead

valid_exts = ('.mgh', '.mgz')

minc1

Read MINC1 format images

Minc1File(mincfile)	Class to wrap MINC1 format opened netcdf object
Minc1Header([data_dtype, shape, zooms])	

Minc1Image(dataobj, affine[, header, extra, ...])	Class for MINC1 format images
MincError	Error when reading MINC files
MincHeader([data_dtype, shape, zooms])	Class to contain header for MINC formats
MincImageArrayProxy(minc_file)	MINC implementation of array proxy protocol

Minc1File

class nibabel.minc1.**Minc1File**(*mincfile*)

Bases: `object`

Class to wrap MINC1 format opened netcdf object

Although it has some of the same methods as a `Header`, we use this only when reading a MINC file, to pull out useful header information, and for the method of reading the data out

__init__(*mincfile*)

Initialize self. See help(type(self)) for accurate signature.

get_affine()

get_data_dtype()

get_data_shape()

get_scaled_data(*sliceobj=()*)

Return scaled data for slice definition *sliceobj*

Parameters

sliceobj [tuple, optional] slice definition. If not specified, return whole array

Returns

scaled_arr [array] array from minc file with scaling applied

get_zooms()

Get real-world sizes of voxels

Minc1Header

class nibabel.minc1.**Minc1Header**(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)

Bases: *nibabel.minc1.MincHeader*

__init__(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)

Initialize self. See help(type(self)) for accurate signature.

classmethod may_contain_header(*binaryblock*)

Minc1Image

class nibabel.minc1.**Minc1Image**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.spatialimages.SpatialImage*

Class for MINC1 format images

The MINC1 image class uses the default header type, rather than a specific MINC header type - and reads the relevant information from the MINC file on load.

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

ImageArrayProxy
alias of *nibabel.minc1.MincImageArrayProxy*

files_types = (('image', '.mnc'),)

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)
Class method to create image from mapping in `file_map`

Parameters

file_map [dict] Mapping with (kay, value) pairs of (`file_type`, FileHolder instance giving file-likes for each file needed for this image type.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_map` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [DataobjImage instance]

header_class
alias of *nibabel.minc1.Minc1Header*

makeable = True

rw = False

valid_exts = ('.mnc',)

MincError

class nibabel.minc1.**MincError**
Bases: `Exception`

Error when reading MINC files

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

MincHeader

class nibabel.minc1.**MincHeader**(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)

Bases: *nibabel.spatialimages.SpatialHeader*

Class to contain header for MINC formats

__init__(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)
Initialize self. See help(type(self)) for accurate signature.

data_from_fileobj(*fileobj*)
See Header class for an implementation we can't use

data_layout = 'C'

data_to_fileobj(*data*, *fileobj*, *rescale=True*)
See Header class for an implementation we can't use

MincImageArrayProxy

class nibabel.minc1.**MincImageArrayProxy**(*minc_file*)

Bases: object

MINC implementation of array proxy protocol

The array proxy allows us to freeze the passed fileobj and header such that it returns the expected data array.

__init__(*minc_file*)
Initialize self. See help(type(self)) for accurate signature.

property is_proxy

property ndim

property shape

minc2

Preliminary MINC2 support

Use with care; I haven't tested this against a wide range of MINC files.

If you have a file that isn't read correctly, please send an example.

Test reading with something like:

```
import nibabel as nib
img = nib.load('my_funny.mnc')
data = img.get_fdata()
print(data.mean())
print(data.max())
print(data.min())
```

and compare against command line output of:

```
mincstats my_funny.mnc
```

Hdf5Bunch(var)	Make object for accessing attributes of variable
Minc2File(mincfile)	Class to wrap MINC2 format file
Minc2Header([data_dtype, shape, zooms])	

Minc2Image(dataobj, affine[, header, extra, ...])	Class for MINC2 images

Hdf5Bunch

class nibabel.minc2.**Hdf5Bunch**(*var*)

Bases: object

Make object for accessing attributes of variable

__init__(*var*)

Initialize self. See help(type(self)) for accurate signature.

Minc2File

class nibabel.minc2.**Minc2File**(*mincfile*)

Bases: *nibabel.minc1.Minc1File*

Class to wrap MINC2 format file

Although it has some of the same methods as a Header, we use this only when reading a MINC2 file, to pull out useful header information, and for the method of reading the data out

__init__(*mincfile*)

Initialize self. See help(type(self)) for accurate signature.

get_data_dtype()

get_data_shape()

get_scaled_data(*sliceobj=()*)

Return scaled data for slice definition *sliceobj*

Parameters

sliceobj [tuple, optional] slice definition. If not specified, return whole array

Returns

scaled_arr [array] array from minc file with scaling applied

Minc2Header

class nibabel.minc2.**Minc2Header**(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)

Bases: *nibabel.minc1.MincHeader*

__init__(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)

Initialize self. See help(type(self)) for accurate signature.

classmethod may_contain_header(*binaryblock*)

Minc2Image

class nibabel.minc2.**Minc2Image**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.minc1.Minc1Image*

Class for MINC2 images

The MINC2 image class uses the default header type, rather than a specific MINC header type - and reads the relevant information from the MINC file on load.

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)
Class method to create image from mapping in `file_map`

Parameters

file_map [dict] Mapping with (kay, value) pairs of (`file_type`, FileHolder instance giving file-likes for each file needed for this image type.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value

of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_map` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [DataobjImage instance]

header_class
alias of *nibabel.minc2.Minc2Header*

nicom

DICOM reader

csareader	CSA header reader from SPM spec
dicomreaders	
dicomwrappers	Classes to wrap DICOM objects and files
dwiparams	Process diffusion imaging parameters
structreader	Stream-like reader for packed data

—

Module: nicom.ascconv

Parse the "ASCCONV" meta data format found in a variety of Siemens MR files.

AscconvParseError	Error parsing ascconv file
Atom(op, obj_type, obj_id)	Object to hold operation, object type and object identifier
NoValue()	Signals no value present
assign2atoms(assign_ast[, default_class])	Parse single assignment ast from ascconv line into atoms
obj_from_atoms(atoms, namespace)	Return object defined by list *atoms* in dict-like *namespace*
parse_ascconv(ascconv_str[, str_delim])	Parse the 'ASCCONV' format from *input_str*.

Module: nicom.csareader

CSA header reader from SPM spec

CSAError	
CSAReadError	
get_acq_mat_txt(csa_dict)	
get_b_matrix(csa_dict)	
get_b_value(csa_dict)	
get_csa_header(dcm_data[, csa_type])	Get CSA header information from DICOM header
get_g_vector(csa_dict)	
get_ice_dims(csa_dict)	
get_n_mosaic(csa_dict)	
get_scalar(csa_dict, tag_name)	
get_slice_normal(csa_dict)	
get_vector(csa_dict, tag_name, n)	
is_mosaic(csa_dict)	Return True if the data is of Mosaic type
nt_str(s)	Strip string to first null
read(csa_str)	Read CSA header from string *csa_str*

Module: nicom.dicomreaders

DicomReadError	
mosaic_to_nii(dcm_data)	Get Nifti file from Siemens
read_mosaic_dir(dicom_path[, globber, ...])	Read all Siemens mosaic DICOMs in directory, return arrays, params
read_mosaic_dwi_dir(dicom_path[, globber, ...])	
slices_to_series(wrappers)	Sort sequence of slice wrappers into series

Module: nicom.dicomwrappers

Classes to wrap DICOM objects and files

The wrappers encapsulate the capabilities of the different DICOM formats.

They also allow dictionary-like access to named fields.

For calculated attributes, we return None where needed data is missing. It seemed strange to raise an error during attribute processing, other than an AttributeError - breaking the 'properties manifesto'. So, any processing that needs to raise an error, should be in a method, rather than in a property, or property-like thing.

MosaicWrapper(dcm_data[, csa_header, n_mosaic])	Class for Siemens mosaic format data
MultiframeWrapper(dcm_data)	Wrapper for Enhanced MR Storage SOP Class
SiemensWrapper(dcm_data[, csa_header])	Wrapper for Siemens format DICOMs
Wrapper(dcm_data)	Class to wrap general DICOM files
WrapperError	
WrapperPrecisionError	
none_or_close(val1, val2[, rtol, atol])	Match if *val1* and *val2* are both None, or are close
wrapper_from_data(dcm_data)	Create DICOM wrapper from DICOM data object
wrapper_from_file(file_like, *args, **kwargs)	Create DICOM wrapper from *file_like* object

Module: nicom.dwiparams

Process diffusion imaging parameters

- `q` is a vector in Q space
- `b` is a b value
- `g` is the unit vector along the direction of q (the gradient direction)

Thus:

b = norm(q)

g = q / norm(q)

(`norm(q)` is the Euclidean norm of `q`)

The B matrix `B` is a symmetric positive semi-definite matrix. If `q_est` is the closest q vector equivalent to the B matrix, then:

B ~ (q_est . q_est.T) / norm(q_est)

B2q(B[, tol])	Estimate q vector from input B matrix *B*
nearest_pos_semi_def(B)	Least squares positive semi-definite tensor estimation
q2bg(q_vector[, tol])	Return b value and q unit vector from q vector *q_vector*

Module: nicom.structreader

Stream-like reader for packed data

Unpacker(buf[, ptr, endian])	Class to unpack values from buffer object

Module: nicom.utils

Utilities for working with DICOM datasets

find_private_section(dcm_data, group_no, creator)	Return start element in group *group_no* given creator name *creator*

AscconvParseError

class nibabel.nicom.ascconv.**AscconvParseError**
Bases: Exception

Error parsing ascconv file

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Atom

class nibabel.nicom.ascconv.**Atom**(*op*, *obj_type*, *obj_id*)
Bases: object

Object to hold operation, object type and object identifier

An atom represents an element in an expression. For example:

```
a.b[0].c
```

has four elements. We call these elements "atoms".

We represent objects (like `a`) as dicts for convenience.

The last element (`.c`) is an `op = ast.Attribute` operation where the object type (*obj_type*) of `c` is not constrained (we can't tell from the operation what type it is). The *obj_id* is the name of the object – "c".

The second to last element `[0]`, is `op = ast.Subscript`, with object type dict (we know from the subsequent operation `.c` that this must be an object, we represent the object by a dict). The *obj_id* is the index 0.

Parameters

op [{'name', 'attr', 'list'}] Assignment type. Assignment to name (root namespace), attribute or list element.

obj_type [{list, dict, other}] Object type being assigned to.

obj_id [str or int] Key (`obj_type is dict`) or index (`obj_type is list`)

__init__(*op*, *obj_type*, *obj_id*)
Initialize self. See help(type(self)) for accurate signature.

NoValue

class nibabel.nicom.ascconv.**NoValue**
Bases: object

Signals no value present

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

assign2atoms

nibabel.nicom.ascconv.**assign2atoms**(*assign_ast*, *default_class=<class 'int'>*)
Parse single assignment ast from ascconv line into atoms

Parameters

assign_ast [assignment statement ast] ast derived from single line of ascconv file.

default_class [class, optional] Class that will create an object where we cannot yet know the object type in the assignment.

Returns

atoms [list] List of `atoms`. See docstring for `atoms`. Defines left to right sequence of assignment in *line_ast*.

obj_from_atoms

nibabel.nicom.ascconv.**obj_from_atoms**(*atoms*, *namespace*)
Return object defined by list *atoms* in dict-like *namespace*

Parameters

atoms [list] List of `atoms`

namespace [dict-like] Namespace in which object will be defined.

Returns

obj_root [object] Namespace such that we can set a desired value to the object defined in *atoms* with `obj_root[obj_key] = value`.

obj_key [str or int] Index into list or key into dictionary for *obj_root*.

parse_ascconv

nibabel.nicom.ascconv.**parse_ascconv**(*ascconv_str*, *str_delim=''*)
Parse the 'ASCCONV' format from *input_str*.

Parameters

ascconv_str [str] The string we are parsing

str_delim [str, optional] String delimiter. Typically "" or """"

Returns

prot_dict [OrderedDict] Meta data pulled from the ASCCONV section.

attrs [OrderedDict] Any attributes stored in the 'ASCCONV BEGIN' line

Raises

AsconvParseError A line of the ASCCONV section could not be parsed.

CSAError

class nibabel.nicom.csareader.**CSAError**

Bases: Exception

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

CSAReadError

class nibabel.nicom.csareader.**CSAReadError**

Bases: *nibabel.nicom.csareader.CSAError*

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

get_acq_mat_txt

nibabel.nicom.csareader.**get_acq_mat_txt**(*csa_dict*)

get_b_matrix

nibabel.nicom.csareader.**get_b_matrix**(*csa_dict*)

get_b_value

nibabel.nicom.csareader.**get_b_value**(*csa_dict*)

get_csa_header

nibabel.nicom.csareader.**get_csa_header**(*dcm_data*, *csa_type='image'*)

Get CSA header information from DICOM header

Return None if the header does not contain CSA information of the specified *csa_type*

Parameters

dcm_data [dicom.Dataset] DICOM dataset. Should implement `__getitem__` and, if initial check for presence of `dcm_data[(0x29, 0x10)]` passes, should satisfy interface for `find_private_section`.

csa_type [{'image', 'series'}, optional] Type of CSA field to read; default is 'image'

Returns

csa_info [None or dict] Parsed CSA field of *csa_type* or None, if we cannot find the CSA information.

get_g_vector

nibabel.nicom.csareader.**get_g_vector**(*csa_dict*)

get_ice_dims

nibabel.nicom.csareader.**get_ice_dims**(*csa_dict*)

get_n_mosaic

nibabel.nicom.csareader.**get_n_mosaic**(*csa_dict*)

get_scalar

nibabel.nicom.csareader.**get_scalar**(*csa_dict*, *tag_name*)

get_slice_normal

nibabel.nicom.csareader.**get_slice_normal**(*csa_dict*)

get_vector

nibabel.nicom.csareader.**get_vector**(*csa_dict*, *tag_name*, *n*)

is_mosaic

nibabel.nicom.csareader.**is_mosaic**(*csa_dict*)

Return True if the data is of Mosaic type

Parameters

csa_dict [dict] dict containing read CSA data

Returns

tf [bool] True if the *dcm_data* appears to be of Siemens mosaic type, False otherwise

nt_str

nibabel.nicom.csareader.**nt_str**(*s*)

Strip string to first null

Parameters

s [bytes]

Returns

sdash [str] s stripped to first occurence of null (0)

read

nibabel.nicom.csareader.**read**(*csa_str*)
Read CSA header from string *csa_str*

Parameters

csa_str [str] byte string containing CSA header information

Returns

header [dict] header information as dict, where *header* has fields (at least) `type, n_tags, tags`. `header['tags']` is also a dictionary with one key, value pair for each tag in the header.

DicomReadError

class nibabel.nicom.dicomreaders.**DicomReadError**
Bases: `Exception`

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

mosaic_to_nii

nibabel.nicom.dicomreaders.**mosaic_to_nii**(*dcm_data*)
Get Nifti file from Siemens

Parameters

dcm_data [`dicom.DataSet`] DICOM header / image as read by `dicom` package

Returns

img [`Nifti1Image`] Nifti image object

read_mosaic_dir

nibabel.nicom.dicomreaders.**read_mosaic_dir**(*dicom_path*, *globber='*.dcm'*, *check_is_dwi=False*, *dicom_kwargs=None*)
Read all Siemens mosaic DICOMs in directory, return arrays, params

Parameters

dicom_path [str] path containing mosaic DICOM images

globber [str, optional] glob to apply within *dicom_path* to select DICOM files. Default is `*.dcm`

check_is_dwi [bool, optional] If True, raises an error if we don't find DWI information in the DICOM headers.

dicom_kwargs [None or dict] Extra keyword arguments to pass to the pydicom `read_file` function.

Returns

data [4D array] data array with last dimension being acquisition. If there were N acquisitions, each of shape (X, Y, Z), *data* will be shape (X, Y, Z, N)

affine [(4,4) array] affine relating 3D voxel space in data to RAS world space

b_values [(N,) array] b values for each acquisition. nan if we did not find diffusion information for these images.

unit_gradients [(N, 3) array] gradient directions of unit length for each acquisition. (nan, nan, nan) if we did not find diffusion information.

read_mosaic_dwi_dir

nibabel.nicom.dicomreaders.**read_mosaic_dwi_dir**(*dicom_path*, *globber='*.dcm'*, *dicom_kwargs=None*)

slices_to_series

nibabel.nicom.dicomreaders.**slices_to_series**(*wrappers*)

Sort sequence of slice wrappers into series

This follows the SPM model fairly closely

Parameters

wrappers [sequence] sequence of `Wrapper` objects for sorting into volumes

Returns

series [sequence] sequence of sequences of wrapper objects, where each sequence is wrapper objects comprising a series, sorted into slice order

MosaicWrapper

class nibabel.nicom.dicomwrappers.**MosaicWrapper**(*dcm_data*, *csa_header=None*, *n_mosaic=None*)

Bases: *nibabel.nicom.dicomwrappers.SiemensWrapper*

Class for Siemens mosaic format data

Mosaic format is a way of storing a 3D image in a 2D slice - and it's as simple as you'd imagine it would be - just storing the slices in a mosaic similar to a light-box print.

We need to allow for this when getting the data and (because of an idiosyncrasy in the way Siemens stores the images) calculating the position of the first voxel.

Adds attributes:

- n_mosaic : int
- mosaic_size : int

Initialize Siemens Mosaic wrapper

The Siemens-specific information is in the *csa_header*, either passed in here, or read from the input *dcm_data*.

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. If *csa_header* is None, it should also be possible for to extract a CSA header from *dcm_data*. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file. A dict should also work.

csa_header [None or mapping, optional] mapping giving values for Siemens CSA image sub-header.

n_mosaic [None or int, optional] number of images in mosaic. If None, try to get this number from *csa_header*. If this fails, raise an error

__init__(*dcm_data*, *csa_header=None*, *n_mosaic=None*)
Initialize Siemens Mosaic wrapper

The Siemens-specific information is in the *csa_header*, either passed in here, or read from the input *dcm_data*.

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. If *csa_header* is None, it should also be possible for to extract a CSA header from *dcm_data*. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file. A dict should also work.

csa_header [None or mapping, optional] mapping giving values for Siemens CSA image sub-header.

n_mosaic [None or int, optional] number of images in mosaic. If None, try to get this number from *csa_header*. If this fails, raise an error

get_data()
Get scaled image data from DICOMs

Resorts data block from mosaic to 3D

Returns

data [array] array with data as scaled from any scaling in the DICOM fields.

Notes

The apparent image in the DICOM file is a 2D array that consists of blocks, that are the output 2D slices. Let's call the original array the *slab*, and the contained slices *slices*. The slices are of pixel dimension `n_slice_rows` x `n_slice_cols`. The slab is of pixel dimension `n_slab_rows` x `n_slab_cols`. Because the arrangement of blocks in the slab is defined as being square, the number of blocks per slab row and slab column is the same. Let `n_blocks` be the number of blocks contained in the slab. There is also `n_slices` - the number of slices actually collected, some number <= `n_blocks`. We have the value `n_slices` from the 'NumberOfImagesInMosaic' field of the Siemens private (CSA) header. `n_row_blocks` and `n_col_blocks` are therefore given by `ceil(sqrt(n_slices))`, and `n_blocks` is `n_row_blocks ** 2`. Also `n_slice_rows == n_slab_rows / n_row_blocks`, etc. Using these numbers we can therefore reconstruct the slices from the 2D DICOM pixel array.

image_position()
Return position of first voxel in data block

Adjusts Siemens mosaic position vector for bug in mosaic format position. See `dicom_mosaic` in doc/theory for details.

Parameters

None

Returns

img_pos [(3,) array] position in mm of voxel (0,0,0) in Mosaic array

image_shape()
Return image shape as returned by `get_data()`

is_mosaic = True

MultiframeWrapper

class nibabel.nicom.dicomwrappers.**MultiframeWrapper**(*dcm_data*)
Bases: *nibabel.nicom.dicomwrappers.Wrapper*

Wrapper for Enhanced MR Storage SOP Class

Tested with Philips' Enhanced DICOM implementation.

The specification for the Enhanced MR image IOP / SOP began life as DICOM supplement 49, but as of 2016 it is part of the standard. In particular see:

- A.36 Enhanced MR Information Object Definitions;
- C.7.6.16 Multi-Frame Functional Groups Module;
- C.7.6.17 Multi-Frame Dimension Module.

Attributes

is_multiframe [boolean] Identifies *dcmdata* as multi-frame

frames [sequence] A sequence of `dicom.dataset.Dataset` objects populated by the `dicom.dataset.Dataset.PerFrameFunctionalGroupsSequence` attribute

shared [object] The first (and only) `dicom.dataset.Dataset` object from a `dicom.dataset.Dataset.SharedFunctionalgroupSequence`.

Methods

image_shape(self)	
image_orient_patient(self)	
voxel_sizes(self)	
image_position(self)	
series_signature(self)	
get_data(self)	

Initializes MultiframeWrapper

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file, but a dictionary should also work.

__init__(*dcm_data*)
Initializes MultiframeWrapper

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file, but a dictionary should also work.

get_data()
Get scaled image data from DICOMs

We return the data as DICOM understands it, first dimension is rows, second dimension is columns

Returns

data [array] array with data as scaled from any scaling in the DICOM fields.

image_orient_patient()
Note that this is _not_ LR flipped

image_position()
Return position of first voxel in data block

Parameters

None

Returns

img_pos [(3,) array] position in mm of voxel (0,0) in image array

image_shape()
The array shape as it will be returned by `get_data()`

The shape is determined by the *Rows* DICOM attribute, *Columns* DICOM attribute, and the set of frame indices given by the *FrameContentSequence[0].DimensionIndexValues* DICOM attribute of each element in the *PerFrameFunctionalGroupsSequence*. The first two axes of the returned shape correspond to the rows, and columns respectively. The remaining axes correspond to those of the frame indices with order preserved.

What each axis in the frame indices refers to is given by the corresponding entry in the *DimensionIndexSequence* DICOM attribute. **WARNING**: Any axis refering to the *StackID* DICOM attribute will have been removed from the frame indices in determining the shape. This is because only a file containing a single stack is currently allowed by this wrapper.

References

- C.7.6.16 Multi-Frame Functional Groups Module: http://dicom.nema.org/medical/dicom/current/output/pdf/part03.pdf#sect_C.7.6.16
- C.7.6.17 Multi-Frame Dimension Module: http://dicom.nema.org/medical/dicom/current/output/pdf/part03.pdf#sect_C.7.6.17
- Diagram of DimensionIndexSequence and DimensionIndexValues: http://dicom.nema.org/medical/dicom/current/output/pdf/part03.pdf#figure_C.7.6.17-1

is_multiframe = True

series_signature()
Signature for matching slices into series

We use *signature* in `self.is_same_series(other)`.

Returns

signature [dict] with values of 2-element sequences, where first element is value, and second element is function to compare this value with another. This allows us to pass things like arrays, that might need to be `allclose` instead of equal

voxel_sizes()
Get i, j, k voxel sizes

SiemensWrapper

class nibabel.nicom.dicomwrappers.**SiemensWrapper**(*dcm_data*, *csa_header=None*)

Bases: *nibabel.nicom.dicomwrappers.Wrapper*

Wrapper for Siemens format DICOMs

Adds attributes:

- csa_header : mapping
- b_matrix : (3,3) array
- q_vector : (3,) array

Initialize Siemens wrapper

The Siemens-specific information is in the *csa_header*, either passed in here, or read from the input *dcm_data*.

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. If *csa_header* is None, it should also be possible to extract a CSA header from *dcm_data*. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file. A dict should also work.

csa_header [None or mapping, optional] mapping giving values for Siemens CSA image sub-header. If None, we try and read the CSA information from *dcm_data*. If this fails, we fall back to an empty dict.

__init__(*dcm_data*, *csa_header=None*)

Initialize Siemens wrapper

The Siemens-specific information is in the *csa_header*, either passed in here, or read from the input *dcm_data*.

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. If *csa_header* is None, it should also be possible to extract a CSA header from *dcm_data*. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file. A dict should also work.

csa_header [None or mapping, optional] mapping giving values for Siemens CSA image sub-header. If None, we try and read the CSA information from *dcm_data*. If this fails, we fall back to an empty dict.

b_matrix()

Get DWI B matrix referring to voxel space

Parameters

None

Returns

B [(3,3) array or None] B matrix in *voxel* orientation space. Returns None if this is not a Siemens header with the required information. We return None if this is a b0 acquisition

is_csa = True

q_vector()

Get DWI q vector referring to voxel space

Parameters

None

Returns

q: (3,) array Estimated DWI q vector in *voxel* orientation space. Returns None if this is not (detectably) a DWI

series_signature()
Add ICE dims from CSA header to signature

slice_normal()

Wrapper

class nibabel.nicom.dicomwrappers.**Wrapper**(*dcm_data*)
Bases: object

Class to wrap general DICOM files

Methods:

- get_affine() (deprecated, use affine property instead)
- get_data()
- get_pixel_array()
- is_same_series(other)
- __getitem__ : return attributes from *dcm_data*
- get(key[, default]) - as usual given __getitem__ above

Attributes and things that look like attributes:

- affine : (4, 4) array
- dcm_data : object
- image_shape : tuple
- image_orient_patient : (3,2) array
- slice_normal : (3,) array
- rotation_matrix : (3,3) array
- voxel_sizes : tuple length 3
- image_position : sequence length 3
- slice_indicator : float
- series_signature : tuple

Initialize wrapper

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file, but a dictionary should also work.

__init__(*dcm_data*)
Initialize wrapper

Parameters

dcm_data [object] object should allow 'get' and '__getitem__' access. Usually this will be a `dicom.dataset.Dataset` object resulting from reading a DICOM file, but a dictionary should also work.

property affine
Mapping between voxel and DICOM coordinate system

(4, 4) affine matrix giving transformation between voxels in data array and mm in the DICOM patient coordinate system.

b_matrix = None

b_value()
Return b value for diffusion or None if not available

b_vector()
Return b vector for diffusion or None if not available

get(*key*, *default=None*)
Get values from underlying dicom data

get_affine()
get_affine method is deprecated. Please use the `img.affine` property instead.

- deprecated from version: 2.5.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

get_data()
Get scaled image data from DICOMs

We return the data as DICOM understands it, first dimension is rows, second dimension is columns

Returns

data [array] array with data as scaled from any scaling in the DICOM fields.

get_pixel_array()
Return unscaled pixel array from DICOM

image_orient_patient()
Note that this is _not_ LR flipped

image_position()
Return position of first voxel in data block

Parameters

None

Returns

img_pos [(3,) array] position in mm of voxel (0,0) in image array

image_shape()
The array shape as it will be returned by `get_data()`

instance_number()
Just because we use this a lot for sorting

is_csa = False

is_mosaic = False

is_multiframe = False

is_same_series(*other*)
Return True if *other* appears to be in same series

Parameters

other [object] object with `series_signature` attribute that is a mapping. Usually it's a `Wrapper` or sub-class instance.

Returns

tf [bool] True if *other* might be in the same series as *self*, False otherwise.

q_vector = None

rotation_matrix()
Return rotation matrix between array indices and mm

Note that we swap the two columns of the 'ImageOrientPatient' when we create the rotation matrix. This is takes into account the slightly odd ij transpose construction of the DICOM orientation fields - see doc/theory/dicom_orientaiton.rst.

series_signature()
Signature for matching slices into series

We use *signature* in `self.is_same_series(other)`.

Returns

signature [dict] with values of 2-element sequences, where first element is value, and second element is function to compare this value with another. This allows us to pass things like arrays, that might need to be `allclose` instead of equal

slice_indicator()
A number that is higher for higher slices in Z

Comparing this number between two adjacent slices should give a difference equal to the voxel size in Z.

See doc/theory/dicom_orientation for description

slice_normal()

voxel_sizes()
voxel sizes for array as returned by `get_data()`

WrapperError

class nibabel.nicom.dicomwrappers.**WrapperError**
Bases: `Exception`

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

WrapperPrecisionError

class nibabel.nicom.dicomwrappers.**WrapperPrecisionError**
Bases: *nibabel.nicom.dicomwrappers.WrapperError*

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

none_or_close

nibabel.nicom.dicomwrappers.**none_or_close**(*val1*, *val2*, *rtol=1e-05*, *atol=1e-06*)
Match if *val1* and *val2* are both None, or are close

Parameters

val1 [None or array-like]

val2 [None or array-like]

rtol [float, optional] Relative tolerance; see `np.allclose`

atol [float, optional] Absolute tolerance; see `np.allclose`

Returns

tf [bool] True iff (both *val1* and *val2* are None) or (*val1* and *val2* are close arrays, as detected by `np.allclose` with parameters *rtol* and *atal*).

Examples

```
>>> none_or_close(None, None)
True
>>> none_or_close(1, None)
False
>>> none_or_close(None, 1)
False
>>> none_or_close([1,2], [1,2])
True
>>> none_or_close([0,1], [0,2])
False
```

wrapper_from_data

nibabel.nicom.dicomwrappers.**wrapper_from_data**(*dcm_data*)
Create DICOM wrapper from DICOM data object

Parameters

dcm_data [`dicom.dataset.Dataset` instance or similar] Object allowing attribute access, with DICOM attributes. Probably a dataset as read by `pydicom`.

Returns

dcm_w [`dicomwrappers.Wrapper` or subclass] DICOM wrapper corresponding to DICOM data type

wrapper_from_file

nibabel.nicom.dicomwrappers.**wrapper_from_file**(*file_like*, **args*, ***kwargs*)

Create DICOM wrapper from *file_like* object

Parameters

file_like [object] filename string or file-like object, pointing to a valid DICOM file readable by `pydicom`

***args** [positional] args to `dicom.read_file` command.

****kwargs** [keyword] args to `dicom.read_file` command. `force=True` might be a likely keyword argument.

Returns

dcm_w [`dicomwrappers.Wrapper` or subclass] DICOM wrapper corresponding to DICOM data type

B2q

nibabel.nicom.dwiparams.**B2q**(*B*, *tol=None*)

Estimate q vector from input B matrix *B*

We require that the input *B* is symmetric positive definite.

Because the solution is a square root, the sign of the returned vector is arbitrary. We set the vector to have a positive x component by convention.

Parameters

B [(3,3) array-like] B matrix - symmetric. We do not check the symmetry.

tol [None or float] absolute tolerance below which to consider eigenvalues of the B matrix to be small enough not to worry about them being negative, in check for positive semi-definite-ness. None (default) results in a fairly tight numerical threshold proportional to the maximum eigenvalue

Returns

q [(3,) vector] Estimated q vector from B matrix *B*

nearest_pos_semi_def

nibabel.nicom.dwiparams.**nearest_pos_semi_def**(*B*)

Least squares positive semi-definite tensor estimation

Reference: Niethammer M, San Jose Estepar R, Bouix S, Shenton M, Westin CF. On diffusion tensor estimation. Conf Proc IEEE Eng Med Biol Soc. 2006;1:2622-5. PubMed PMID: 17946125; PubMed Central PMCID: PMC2791793.

Parameters

B [(3,3) array-like] B matrix - symmetric. We do not check the symmetry.

Returns

npds [(3,3) array] Estimated nearest positive semi-definite array to matrix *B*.

Examples

```
>>> B = np.diag([1, 1, -1])
>>> nearest_pos_semi_def(B)
array([[0.75, 0.  , 0.  ],
       [0.  , 0.75, 0.  ],
       [0.  , 0.  , 0.  ]])
```

q2bg

nibabel.nicom.dwiparams.**q2bg**(*q_vector*, *tol=1e-05*)

Return b value and q unit vector from q vector *q_vector*

Parameters

q_vector [(3,) array-like] q vector

tol [float, optional] q vector L2 norm below which *q_vector* considered to be *b_value* of zero, and therefore *g_vector* also considered to zero.

Returns

b_value [float] L2 Norm of *q_vector* or 0 if L2 norm < *tol*

g_vector [shape (3,) ndarray] *q_vector* / *b_value* or 0 if L2 norma < *tol*

Examples

```
>>> q2bg([1, 0, 0])
(1.0, array([1., 0., 0.]))
>>> q2bg([0, 10, 0])
(10.0, array([0., 1., 0.]))
>>> q2bg([0, 0, 0])
(0.0, array([0., 0., 0.]))
```

Unpacker

class nibabel.nicom.structreader.**Unpacker**(*buf*, *ptr=0*, *endian=None*)

Bases: object

Class to unpack values from buffer object

The buffer object is usually a string. Caches compiled `struct` format strings so that repeated unpacking with the same format string should be faster than using `struct.unpack` directly.

Examples

```
>>> a = b'1234567890'
>>> upk = Unpacker(a)
>>> upk.unpack('2s') == (b'12',)
True
>>> upk.unpack('2s') == (b'34',)
True
>>> upk.ptr
4
>>> upk.read(3) == b'567'
True
>>> upk.ptr
7
```

Initialize unpacker

Parameters

buf [buffer] object implementing buffer protocol (e.g. str)

ptr [int, optional] offset at which to begin reads from *buf*

endian [None or str, optional] endian code to prepend to format, as for `unpack` endian codes. None (the default) corresponds to the default behavior of `struct` - assuming system endian unless you specify the byte order specifically in the format string passed to `unpack`

__init__(*buf*, *ptr=0*, *endian=None*)

Initialize unpacker

Parameters

buf [buffer] object implementing buffer protocol (e.g. str)

ptr [int, optional] offset at which to begin reads from *buf*

endian [None or str, optional] endian code to prepend to format, as for `unpack` endian codes. None (the default) corresponds to the default behavior of `struct` - assuming system endian unless you specify the byte order specifically in the format string passed to `unpack`

read(*n_bytes=- 1*)

Return byte string of length *n_bytes* at current position

Returns sub-string from `self.buf` and updates `self.ptr` to the position after the read data.

Parameters

n_bytes [int, optional] number of bytes to read. Can be -1 (the default) in which case we return all the remaining bytes in `self.buf`

Returns

s [byte string]

unpack(*fmt*)

Unpack values from contained buffer

Unpacks values from `self.buf` and updates `self.ptr` to the position after the read data.

Parameters

fmt [str] format string as for `unpack`

Returns

values [tuple] values as unpacked from `self.buf` according to *fmt*

find_private_section

nibabel.nicom.utils.**find_private_section**(*dcm_data*, *group_no*, *creator*)

Return start element in group *group_no* given creator name *creator*

Private attribute tags need to announce where they will go by putting a tag in the private group (here *group_no*) between elements 1 and 0xFF. The element number of these tags give the start of matching information, in the higher tag numbers.

Parameters

dcm_data [dicom `dataset`] Iterating over *dcm_data* produces `elements` with attributes `tag`, `VR`, `value`

group_no [int] Group number in which to search

creator [str or bytes or regex] Name of section - e.g. 'SIEMENS CSA HEADER' - or regex to search for section name. Regex used via `creator.search(element_value)` where `element_value` is the value of the data element.

Returns

element_start [int] Element number at which named section starts

nifti1

Read / write access to NIfTI1 image format

NIfTI1 format defined at http://nifti.nimh.nih.gov/nifti-1/

Nifti1DicomExtension(code, content[, parent_hdr])	NIfTI1 DICOM header extension
Nifti1Extension(code, content)	Baseclass for NIfTI1 header extensions.
Nifti1Extensions([iterable])	Simple extension collection, implemented as a list-subclass.
Nifti1Header([binaryblock, endianness, ...])	Class for NIfTI1 header
Nifti1Image(dataobj, affine[, header, ...])	Class for single file NIfTI1 format image
Nifti1Pair(dataobj, affine[, header, extra, ...])	Class for NIfTI1 format image, header pair
Nifti1PairHeader([binaryblock, endianness, ...])	Class for NIfTI1 pair header
load(filename)	Load NIfTI1 single or pair from *filename*
save(img, filename)	Save NIfTI1 single or pair to *filename*

Nifti1DicomExtension

class nibabel.nifti1.**Nifti1DicomExtension**(*code*, *content*, *parent_hdr=None*)

Bases: *nibabel.nifti1.Nifti1Extension*

NIfTI1 DICOM header extension

This class is a thin wrapper around pydicom to read a binary DICOM byte string. If pydicom is available, content is exposed as a Dicom Dataset. Otherwise, this silently falls back to the standard NiftiExtension class and content is the raw bytestring loaded directly from the nifti file header.

Parameters

code [int or str] Canonical extension code as defined in the NIfTI standard, given either as integer or corresponding label (see `extension_codes`)

content [bytes or pydicom Dataset or None] Extension content - either a bytestring as read from the NIfTI file header or an existing pydicom Dataset. If a bystestring, the content is converted into a Dataset on initialization. If None, a new empty Dataset is created.

parent_hdr [*Nifti1Header*, optional] If a dicom extension belongs to an existing *Nifti1Header*, it may be provided here to ensure that the DICOM dataset is written with correctly corresponding endianness; otherwise it is assumed the dataset is little endian.

Notes

code should always be 2 for DICOM.

__init__(*code*, *content*, *parent_hdr=None*)

Parameters

code [int or str] Canonical extension code as defined in the NIfTI standard, given either as integer or corresponding label (see `extension_codes`)

content [bytes or pydicom Dataset or None] Extension content - either a bytestring as read from the NIfTI file header or an existing pydicom Dataset. If a bystestring, the content is converted into a Dataset on initialization. If None, a new empty Dataset is created.

parent_hdr [*Nifti1Header*, optional] If a dicom extension belongs to an existing *Nifti1Header*, it may be provided here to ensure that the DICOM dataset is written with correctly corresponding endianness; otherwise it is assumed the dataset is little endian.

Notes

code should always be 2 for DICOM.

Nifti1Extension

class nibabel.nifti1.**Nifti1Extension**(*code*, *content*)
Bases: object

Baseclass for NIfTI1 header extensions.

This class is sufficient to handle very simple text-based extensions, such as *comment*. More sophisticated extensions should/will be supported by dedicated subclasses.

Parameters

code [int or str] Canonical extension code as defined in the NIfTI standard, given either as integer or corresponding label (see `extension_codes`)

content [str] Extension content as read from the NIfTI file header. This content is converted into a runtime representation.

__init__(*code*, *content*)

Parameters

code [int or str] Canonical extension code as defined in the NIfTI standard, given either as integer or corresponding label (see `extension_codes`)

content [str] Extension content as read from the NIfTI file header. This content is converted into a runtime representation.

get_code()
Return the canonical extension type code.

get_content()
Return the extension content in its runtime representation.

get_sizeondisk()
Return the size of the extension in the NIfTI file.

write_to(*fileobj*, *byteswap*)
Write header extensions to fileobj

Write starts at fileobj current file position.

Parameters

fileobj [file-like object] Should implement `write` method

byteswap [boolean] Flag if byteswapping the data is required.

Returns

None

Nifti1Extensions

class nibabel.nifti1.**Nifti1Extensions**(*iterable=()*, */*)
Bases: list

Simple extension collection, implemented as a list-subclass.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

count(*ecode*)
Returns the number of extensions matching a given *ecode*.

Parameters

code [int | str] The ecode can be specified either literal or as numerical value.

classmethod from_fileobj(*fileobj*, *size*, *byteswap*)
Read header extensions from a fileobj

Parameters

fileobj [file-like object] We begin reading the extensions at the current file position

size [int] Number of bytes to read. If negative, fileobj will be read till its end.

byteswap [boolean] Flag if byteswapping the read data is required.

Returns

An extension list. This list might be empty in case not extensions

were present in fileobj.

get_codes()
Return a list of the extension code of all available extensions

get_sizeondisk()
Return the size of the complete header extensions in the NIfTI file.

write_to(*fileobj*, *byteswap*)
Write header extensions to fileobj

Write starts at fileobj current file position.

Parameters

fileobj [file-like object] Should implement `write` method

byteswap [boolean] Flag if byteswapping the data is required.

Returns

None

Nifti1Header

class nibabel.nifti1.**Nifti1Header**(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)
Bases: *nibabel.spm99analyze.SpmAnalyzeHeader*

Class for NIfTI1 header

The NIfTI1 header has many more coded fields than the simpler Analyze variants. NIfTI1 headers also have extensions.

Nifti allows the header to be a separate file, as part of a nifti image / header pair, or to precede the data in a single file. The object needs to know which type it is, in order to manage the voxel offset pointing to the data, extension reading, and writing the correct magic string.

This class handles the header-preceding-data case.

Initialize header from binary data block and extensions

__init__(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)
Initialize header from binary data block and extensions

copy()
Return copy of header

Take reference to extensions as well as copy of header contents

classmethod default_structarr(*endianness=None*)
Create empty header binary block with given endianness

exts_klass
alias of *nibabel.nifti1.Nifti1Extensions*

classmethod from_fileobj(*fileobj*, *endianness=None*, *check=True*)
Return read structure with given or guessed endiancode

Parameters

fileobj [file-like object] Needs to implement `read` method

endianness [None or endian code, optional] Code specifying endianness of read data

Returns

wstr [WrapStruct object] WrapStruct object initialized from data in fileobj

classmethod from_header(*header=None*, *check=True*)
Class method to create header from another header

Extend Analyze header copy by copying extensions from other Nifti types.

Parameters

header [`Header` instance or mapping] a header of this class, or another class of header for conversion to this type

check [{True, False}] whether to check header for integrity

Returns

hdr [header instance] fresh header instance of our own class

get_best_affine()
Select best of available transforms

get_data_shape()
Get shape of data

Notes

Applies freesurfer hack for large vectors described in issue 100 and save_nifti.m.

Allows for freesurfer hack for 7th order icosahedron surface described in issue 309, load_nifti.m, and save_nifti.m.

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.get_data_shape()
(0,)
>>> hdr.set_data_shape((1,2,3))
>>> hdr.get_data_shape()
(1, 2, 3)
```

Expanding number of dimensions gets default zooms

```
>>> hdr.get_zooms()
(1.0, 1.0, 1.0)
```

get_dim_info()

Gets NIfTI MRI slice etc dimension information

Returns

freq [{None,0,1,2}] Which data array axis is frequency encode direction

phase [{None,0,1,2}] Which data array axis is phase encode direction

slice [{None,0,1,2}] Which data array axis is slice encode direction

where `data array` is the array returned by `get_data`

Because NIfTI1 files are natively Fortran indexed: 0 is fastest changing in file 1 is medium changing in file 2 is slowest changing in file

`None` means the axis appears not to be specified.

Examples

See set_dim_info function

get_intent(*code_repr='label'*)

Get intent code, parameters and name

Parameters

code_repr [string] string giving output form of intent code representation. Default is 'label'; use 'code' for integer representation.

Returns

code [string or integer] intent code, or string describing code

parameters [tuple] parameters for the intent

name [string] intent name

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.set_intent('t test', (10,), name='some score')
>>> hdr.get_intent()
('t test', (10.0,), 'some score')
>>> hdr.get_intent('code')
(3, (10.0,), 'some score')
```

get_n_slices()
Return the number of slices

get_qform(*coded=False*)
Return 4x4 affine matrix from qform parameters in header

Parameters

coded [bool, optional] If True, return {affine or None}, and qform code. If False, just return affine. {affine or None} means, return None if qform code == 0, and affine otherwise.

Returns

affine [None or (4,4) ndarray] If *coded* is False, always return affine reconstructed from qform quaternion. If *coded* is True, return None if qform code is 0, else return the affine.

code [int] Qform code. Only returned if *coded* is True.

get_qform_quaternion()
Compute quaternion from b, c, d of quaternion

Fills a value by assuming this is a unit quaternion

get_sform(*coded=False*)
Return 4x4 affine matrix from sform parameters in header

Parameters

coded [bool, optional] If True, return {affine or None}, and sform code. If False, just return affine. {affine or None} means, return None if sform code == 0, and affine otherwise.

Returns

affine [None or (4,4) ndarray] If *coded* is False, always return affine from sform fields. If *coded* is True, return None if sform code is 0, else return the affine.

code [int] Sform code. Only returned if *coded* is True.

get_slice_duration()
Get slice duration

Returns

slice_duration [float] time to acquire one slice

Notes

The NIfTI1 spec appears to require the slice dimension to be defined for slice_duration to have meaning.

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.set_dim_info(slice=2)
>>> hdr.set_slice_duration(0.3)
>>> print("%0.1f" % hdr.get_slice_duration())
0.3
```

get_slice_times()
Get slice times from slice timing information

Returns

slice_times [tuple] Times of acquisition of slices, where 0 is the beginning of the acquisition, ordered by position in file. nifti allows slices at the top and bottom of the volume to be excluded from the standard slice timing specification, and calls these "padding slices". We give padding slices `None` as a time of acquisition

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.set_dim_info(slice=2)
>>> hdr.set_data_shape((1, 1, 7))
>>> hdr.set_slice_duration(0.1)
>>> hdr['slice_code'] = slice_order_codes['sequential increasing']
>>> slice_times = hdr.get_slice_times()
>>> np.allclose(slice_times, [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
True
```

get_slope_inter()
Get data scaling (slope) and DC offset (intercept) from header data

Returns

slope [None or float] scaling (slope). None if there is no valid scaling from these fields

inter [None or float] offset (intercept). None if there is no valid scaling or if offset is not finite.

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.get_slope_inter()
(1.0, 0.0)
>>> hdr['scl_slope'] = 0
>>> hdr.get_slope_inter()
(None, None)
>>> hdr['scl_slope'] = np.nan
>>> hdr.get_slope_inter()
(None, None)
```

(continues on next page)

(continued from previous page)

```
>>> hdr['scl_slope'] = 1
>>> hdr['scl_inter'] = 1
>>> hdr.get_slope_inter()
(1.0, 1.0)
>>> hdr['scl_inter'] = np.inf
>>> hdr.get_slope_inter()
Traceback (most recent call last):
    ...
HeaderDataError: Valid slope but invalid intercept inf
```

get_xyzt_units()

has_data_intercept = True

has_data_slope = True

is_single = True

classmethod may_contain_header(*binaryblock*)

pair_magic = b'ni1'

pair_vox_offset = 0

quaternion_threshold = -3.5762786865234375e-07

set_data_shape(*shape*)

Set shape of data # noqa

If `ndims == len(shape)` then we set zooms for dimensions higher than `ndims` to 1.0

Nifti1 images can have up to seven dimensions. For FreeSurfer-variant Nifti surface files, the first dimension is assumed to correspond to vertices/nodes on a surface, and dimensions two and three are constrained to have depth of 1. Dimensions 4-7 are constrained only by type bounds.

Parameters

shape [sequence] sequence of integers specifying data array shape

Notes

Applies freesurfer hack for large vectors described in issue 100 and save_nifti.m.

Allows for freesurfer hack for 7th order icosahedron surface described in issue 309, load_nifti.m, and save_nifti.m.

The Nifti1 standard header allows for the following "point set" definition of a surface, not currently implemented in nibabel.

```
To signify that the vector value at each voxel is really a
spatial coordinate (e.g., the vertices or nodes of a surface mesh):
  - dataset must have a 5th dimension
  - intent_code must be NIFTI_INTENT_POINTSET
  - dim[0] = 5
  - dim[1] = number of points
  - dim[2] = dim[3] = dim[4] = 1
  - dim[5] must be the dimensionality of space (e.g., 3 => 3D space).
  - intent_name may describe the object these points come from
    (e.g., "pial", "gray/white" , "EEG", "MEG").
```

set_dim_info(*freq=None*, *phase=None*, *slice=None*)
Sets nifti MRI slice etc dimension information

Parameters

freq [{None, 0, 1, 2}] axis of data array referring to frequency encoding

phase [{None, 0, 1, 2}] axis of data array referring to phase encoding

slice [{None, 0, 1, 2}] axis of data array referring to slice encoding

``None`` means the axis is not specified.

Notes

This is stored in one byte in the header

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.set_dim_info(1, 2, 0)
>>> hdr.get_dim_info()
(1, 2, 0)
>>> hdr.set_dim_info(freq=1, phase=2, slice=0)
>>> hdr.get_dim_info()
(1, 2, 0)
>>> hdr.set_dim_info()
>>> hdr.get_dim_info()
(None, None, None)
>>> hdr.set_dim_info(freq=1, phase=None, slice=0)
>>> hdr.get_dim_info()
(1, None, 0)
```

set_intent(*code*, *params=()*, *name=''*, *allow_unknown=False*)
Set the intent code, parameters and name

If parameters are not specified, assumed to be all zero. Each intent code has a set number of parameters associated. If you specify any parameters, then it will need to be the correct number (e.g the "f test" intent requires 2). However, parameters can also be set in the file data, so we also allow not setting any parameters (empty parameter tuple).

Parameters

code [integer or string] code specifying nifti intent

params [list, tuple of scalars] parameters relating to intent (see intent_codes) defaults to (). Unspecified parameters are set to 0.0

name [string] intent name (description). Defaults to ''

allow_unknown [{False, True}, optional] Allow unknown integer intent codes. If False (the default), a KeyError is raised on attempts to set the intent to an unknown code.

Returns

None

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.set_intent(0)  # no intent
>>> hdr.set_intent('z score')
>>> hdr.get_intent()
('z score', (), '')
>>> hdr.get_intent('code')
(5, (), '')
>>> hdr.set_intent('t test', (10,), name='some score')
>>> hdr.get_intent()
('t test', (10.0,), 'some score')
>>> hdr.set_intent('f test', (2, 10), name='another score')
>>> hdr.get_intent()
('f test', (2.0, 10.0), 'another score')
>>> hdr.set_intent('f test')
>>> hdr.get_intent()
('f test', (0.0, 0.0), '')
>>> hdr.set_intent(9999, allow_unknown=True) # unknown code
>>> hdr.get_intent()
('unknown code 9999', (), '')
```

set_qform(*affine*, *code=None*, *strip_shears=True*)

Set qform header values from 4x4 affine

Parameters

affine [None or 4x4 array] affine transform to write into sform. If None, only set code.

code [None, string or integer, optional] String or integer giving meaning of transform in *affine*. The default is None. If code is None, then:

- If affine is None, *code*-> 0
- If affine not None and existing qform code in header == 0, *code*-> 2 (aligned)
- If affine not None and existing qform code in header != 0, *code*-> existing qform code in header

strip_shears [bool, optional] Whether to strip shears in *affine*. If True, shears will be silently stripped. If False, the presence of shears will raise a `HeaderDataError`

Notes

The qform transform only encodes translations, rotations and zooms. If there are shear components to the *affine* transform, and *strip_shears* is True (the default), the written qform gives the closest approximation where the rotation matrix is orthogonal. This is to allow quaternion representation. The orthogonal representation enforces orthogonal axes.

Examples

```
>>> hdr = Nifti1Header()
>>> int(hdr['qform_code'])  # gives 0 - unknown
0
>>> affine = np.diag([1,2,3,1])
>>> np.all(hdr.get_qform() == affine)
False
>>> hdr.set_qform(affine)
>>> np.all(hdr.get_qform() == affine)
True
>>> int(hdr['qform_code'])  # gives 2 - aligned
2
>>> hdr.set_qform(affine, code='talairach')
>>> int(hdr['qform_code'])
3
>>> hdr.set_qform(affine, code=None)
>>> int(hdr['qform_code'])
3
>>> hdr.set_qform(affine, code='scanner')
>>> int(hdr['qform_code'])
1
>>> hdr.set_qform(None)
>>> int(hdr['qform_code'])
0
```

set_sform(*affine*, *code=None*)

Set sform transform from 4x4 affine

Parameters

affine [None or 4x4 array] affine transform to write into sform. If None, only set *code*

code [None, string or integer, optional] String or integer giving meaning of transform in *affine*. The default is None. If code is None, then:

- If affine is None, *code*-> 0
- If affine not None and existing sform code in header == 0, *code*-> 2 (aligned)
- If affine not None and existing sform code in header != 0, *code*-> existing sform code in header

Examples

```
>>> hdr = Nifti1Header()
>>> int(hdr['sform_code'])  # gives 0 - unknown
0
>>> affine = np.diag([1,2,3,1])
>>> np.all(hdr.get_sform() == affine)
False
>>> hdr.set_sform(affine)
>>> np.all(hdr.get_sform() == affine)
True
>>> int(hdr['sform_code'])  # gives 2 - aligned
2
>>> hdr.set_sform(affine, code='talairach')
>>> int(hdr['sform_code'])
3
>>> hdr.set_sform(affine, code=None)
>>> int(hdr['sform_code'])
3
>>> hdr.set_sform(affine, code='scanner')
>>> int(hdr['sform_code'])
1
>>> hdr.set_sform(None)
>>> int(hdr['sform_code'])
0
```

set_slice_duration(*duration*)
Set slice duration

Parameters

duration [scalar] time to acquire one slice

Examples

See `get_slice_duration`

set_slice_times(*slice_times*)
Set slice times into *hdr*

Parameters

slice_times [tuple] tuple of slice times, one value per slice tuple can include None to indicate no slice time for that slice

Examples

```
>>> hdr = Nifti1Header()
>>> hdr.set_dim_info(slice=2)
>>> hdr.set_data_shape([1, 1, 7])
>>> hdr.set_slice_duration(0.1)
>>> times = [None, 0.2, 0.4, 0.1, 0.3, 0.0, None]
>>> hdr.set_slice_times(times)
>>> hdr.get_value_label('slice_code')
'alternating decreasing'
>>> int(hdr['slice_start'])
```

(continues on next page)

(continued from previous page)

```
1
>>> int(hdr['slice_end'])
5
```

set_slope_inter(*slope*, *inter=None*)

Set slope and / or intercept into header

Set slope and intercept for image data, such that, if the image data is `arr`, then the scaled image data will be `(arr * slope) + inter`

(*slope*, *inter*) of (NaN, NaN) is a signal to a containing image to set *slope*, *inter* automatically on write.

Parameters

slope [None or float] If None, implies *slope* of NaN. If *slope* is None or NaN then *inter* should be None or NaN. Values of 0, Inf or -Inf raise HeaderDataError

inter [None or float, optional] Intercept. If None, implies *inter* of NaN. If *slope* is None or NaN then *inter* should be None or NaN. Values of Inf or -Inf raise HeaderDataError

set_xyzt_units(*xyz=None*, *t=None*)

single_magic = b'n+1'

single_vox_offset = 352

template_dtype = dtype([('sizeof_hdr', '<i4'), ('data_type', 'S10'), ('db_name', 'S18'

write_to(*fileobj*)

Write structure to fileobj

Write starts at fileobj current file position.

Parameters

fileobj [file-like object] Should implement `write` method

Returns

None

Examples

```
>>> wstr = WrapStruct()
>>> from io import BytesIO
>>> str_io = BytesIO()
>>> wstr.write_to(str_io)
>>> wstr.binaryblock == str_io.getvalue()
True
```

Nifti1Image

class nibabel.nifti1.**Nifti1Image**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.nifti1.Nifti1Pair*, *nibabel.filebasedimages.SerializableImage*

Class for single file NIfTI1 format image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The `set_sform()` and `set_qform()` methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The `set_sform()` and `set_qform()` methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

files_types = (('image', '.nii'),)

header_class
alias of *nibabel.nifti1.Nifti1Header*

update_header()
Harmonize header with image data and affine

valid_exts = ('.nii',)

Nifti1Pair

class nibabel.nifti1.**Nifti1Pair**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.analyze.AnalyzeImage*

Class for NIfTI1 format image, header pair

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The *set_sform()* and *set_qform()* methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The *`set_sform()`* and *`set_qform()`* methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

as_reoriented(*ornt*)

Apply an orientation change and return a new image

If ornt is identity transform, return the original image, unchanged

Parameters

ornt [(n,2) orientation array] orientation transform. `ornt[N,1]` is flip of axis N of the array implied by `shape`, where 1 means no flip and -1 means flip. For example, if ``N==0 and `ornt[0,1] == -1`, and there's an array `arr` of shape *shape*, the flip would correspond to the effect of `np.flipud(arr)`. `ornt[:,0]` is the transpose that needs to be done to the implied array, as in `arr.transpose(ornt[:,0])`

get_qform(*coded=False*)

Return 4x4 affine matrix from qform parameters in header

Parameters

coded [bool, optional] If True, return {affine or None}, and qform code. If False, just return affine. {affine or None} means, return None if qform code == 0, and affine otherwise.

Returns

affine [None or (4,4) ndarray] If *coded* is False, always return affine reconstructed from qform quaternion. If *coded* is True, return None if qform code is 0, else return the affine.

code [int] Qform code. Only returned if *coded* is True.

See also:

set_qform

get_sform

get_sform(*coded=False*)

Return 4x4 affine matrix from sform parameters in header

Parameters

coded [bool, optional] If True, return {affine or None}, and sform code. If False, just return affine. {affine or None} means, return None if sform code == 0, and affine otherwise.

Returns

affine [None or (4,4) ndarray] If *coded* is False, always return affine from sform fields. If *coded* is True, return None if sform code is 0, else return the affine.

code [int] Sform code. Only returned if *coded* is True.

See also:

set_sform

get_qform

header_class
alias of *nibabel.nifti1.Nifti1PairHeader*

rw = True

set_qform(*affine*, *code=None*, *strip_shears=True*, ***kwargs*)
Set qform header values from 4x4 affine

Parameters

affine [None or 4x4 array] affine transform to write into sform. If None, only set code.

code [None, string or integer] String or integer giving meaning of transform in *affine*. The default is None. If code is None, then:

- If affine is None, *code*-> 0
- If affine not None and existing qform code in header == 0, *code*-> 2 (aligned)
- If affine not None and existing qform code in header != 0, *code*-> existing qform code in header

strip_shears [bool, optional] Whether to strip shears in *affine*. If True, shears will be silently stripped. If False, the presence of shears will raise a `HeaderDataError`

update_affine [bool, optional] Whether to update the image affine from the header best affine after setting the qform. Must be keyword argument (because of different position in *set_qform*). Default is True

See also:

get_qform

set_sform

Examples

```
>>> data = np.arange(24).reshape((2,3,4))
>>> aff = np.diag([2, 3, 4, 1])
>>> img = Nifti1Pair(data, aff)
>>> img.get_qform()
array([[2., 0., 0., 0.],
       [0., 3., 0., 0.],
       [0., 0., 4., 0.],
       [0., 0., 0., 1.]])
>>> img.get_qform(coded=True)
(None, 0)
>>> aff2 = np.diag([3, 4, 5, 1])
>>> img.set_qform(aff2, 'talairach')
>>> qaff, code = img.get_qform(coded=True)
>>> np.all(qaff == aff2)
True
>>> int(code)
3
```

set_sform(*affine*, *code=None*, ***kwargs*)

Set sform transform from 4x4 affine

Parameters

affine [None or 4x4 array] affine transform to write into sform. If None, only set *code*

code [None, string or integer] String or integer giving meaning of transform in *affine*. The default is None. If code is None, then:

- If affine is None, *code*-> 0
- If affine not None and existing sform code in header == 0, *code*-> 2 (aligned)
- If affine not None and existing sform code in header != 0, *code*-> existing sform code in header

update_affine [bool, optional] Whether to update the image affine from the header best affine after setting the qform. Must be keyword argument (because of different position in *set_qform*). Default is True

See also:

get_sform

set_qform

Examples

```
>>> data = np.arange(24).reshape((2,3,4))
>>> aff = np.diag([2, 3, 4, 1])
>>> img = Nifti1Pair(data, aff)
>>> img.get_sform()
array([[2., 0., 0., 0.],
       [0., 3., 0., 0.],
       [0., 0., 4., 0.],
       [0., 0., 0., 1.]])
>>> saff, code = img.get_sform(coded=True)
```

(continues on next page)

(continued from previous page)

```
>>> saff
array([[2., 0., 0., 0.],
       [0., 3., 0., 0.],
       [0., 0., 4., 0.],
       [0., 0., 0., 1.]])
>>> int(code)
2
>>> aff2 = np.diag([3, 4, 5, 1])
>>> img.set_sform(aff2, 'talairach')
>>> saff, code = img.get_sform(coded=True)
>>> np.all(saff == aff2)
True
>>> int(code)
3
```

update_header()
Harmonize header with image data and affine

See AnalyzeImage.update_header for more examples

Examples

```
>>> data = np.zeros((2,3,4))
>>> affine = np.diag([1.0,2.0,3.0,1.0])
>>> img = Nifti1Image(data, affine)
>>> hdr = img.header
>>> np.all(hdr.get_qform() == affine)
True
>>> np.all(hdr.get_sform() == affine)
True
```

Nifti1PairHeader

class nibabel.nifti1.**Nifti1PairHeader**(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)

Bases: *nibabel.nifti1.Nifti1Header*

Class for NIfTI1 pair header

Initialize header from binary data block and extensions

__init__(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)
Initialize header from binary data block and extensions

is_single = False

load

nibabel.nifti1.**load**(*filename*)
Load NIfTI1 single or pair from *filename*

Parameters

filename [str] filename of image to be loaded

Returns

img [Nifti1Image or Nifti1Pair] NIfTI1 single or pair image instance

Raises

ImageFileError if *filename* doesn't look like NIfTI1;

IOError if *filename* does not exist.

save

nibabel.nifti1.**save**(*img*, *filename*)
Save NIfTI1 single or pair to *filename*

Parameters

filename [str] filename to which to save image

nifti2

Read / write access to NIfTI2 image format

Format described here:

https://www.nitrc.org/forum/message.php?msg_id=3738

Nifti2Header([binaryblock, endianness, ...])	Class for NIfTI2 header
Nifti2Image(dataobj, affine[, header, ...])	Class for single file NIfTI2 format image
Nifti2Pair(dataobj, affine[, header, extra, ...])	Class for NIfTI2 format image, header pair
Nifti2PairHeader([binaryblock, endianness, ...])	Class for NIfTI2 pair header
load(filename)	Load NIfTI2 single or pair image from *filename*
save(img, filename)	Save NIfTI2 single or pair to *filename*

Nifti2Header

class nibabel.nifti2.**Nifti2Header**(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)
Bases: *nibabel.nifti1.Nifti1Header*

Class for NIfTI2 header

NIfTI2 is a slightly simplified variant of NIfTI1 which replaces 32-bit floats with 64-bit floats, and increases some integer widths to 32 or 64 bits.

Initialize header from binary data block and extensions

__init__(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)
Initialize header from binary data block and extensions

classmethod default_structarr(*endianness=None*)
Create empty header binary block with given endianness

get_data_shape()
Get shape of data

Notes

Does not use Nifti1 freesurfer hack for large vectors described in `Nifti1Header.set_data_shape()`

Examples

```
>>> hdr = Nifti2Header()
>>> hdr.get_data_shape()
(0,)
>>> hdr.set_data_shape((1,2,3))
>>> hdr.get_data_shape()
(1, 2, 3)
```

Expanding number of dimensions gets default zooms

```
>>> hdr.get_zooms()
(1.0, 1.0, 1.0)
```

classmethod may_contain_header(*binaryblock*)

pair_magic = b'ni2'

pair_vox_offset = 0

quaternion_threshold = -6.661338147750939e-16

set_data_shape(*shape*)
Set shape of data

If `ndims == len(shape)` then we set zooms for dimensions higher than `ndims` to 1.0

Parameters

shape [sequence] sequence of integers specifying data array shape

Notes

Does not apply nifti1 Freesurfer hack for long vectors (see `Nifti1Header.set_data_shape()`)

single_magic = b'n+2'

single_vox_offset = 544

sizeof_hdr = 540

template_dtype = dtype([('sizeof_hdr', '<i4'), ('magic', 'S4'), ('eol_check', 'i1', (4

Nifti2Image

class nibabel.nifti2.**Nifti2Image**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.nifti1.Nifti1Image*

Class for single file NIfTI2 format image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The `set_sform()` and `set_qform()` methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The `set_sform()` and `set_qform()` methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

header_class
alias of *nibabel.nifti2.Nifti2Header*

Nifti2Pair

class nibabel.nifti2.**Nifti2Pair**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.nifti1.Nifti1Pair*

Class for NIfTI2 format image, header pair

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The `set_sform()` and `set_qform()` methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

Notes

If both a *header* and an *affine* are specified, and the *affine* does not match the affine that is in the *header*, the *affine* will be used, but the `sform_code` and `qform_code` fields in the header will be re-initialised to their default values. This is performed on the basis that, if you are changing the affine, you are likely to be changing the space to which the affine is pointing. The `set_sform()` and `set_qform()` methods can be used to update the codes after an image has been created - see those methods, and the *manual* for more details.

header_class
alias of *nibabel.nifti2.Nifti2PairHeader*

Nifti2PairHeader

class nibabel.nifti2.**Nifti2PairHeader**(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)

Bases: *nibabel.nifti2.Nifti2Header*

Class for NIfTI2 pair header

Initialize header from binary data block and extensions

__init__(*binaryblock=None*, *endianness=None*, *check=True*, *extensions=()*)
Initialize header from binary data block and extensions

is_single = False

load

nibabel.nifti2.**load**(*filename*)
Load NIfTI2 single or pair image from *filename*

Parameters

filename [str] filename of image to be loaded

Returns

img [Nifti2Image or Nifti2Pair] nifti2 single or pair image instance

Raises

ImageFileError if *filename* doesn't look like nifti2;

IOError if *filename* does not exist.

save

nibabel.nifti2.**save**(*img*, *filename*)
Save NIfTI2 single or pair to *filename*

Parameters

filename [str] filename to which to save image

ecat

Read ECAT format images

An ECAT format image consists of:

- a *main header*;
- at least one *matrix list* (mlist);

ECAT thinks of memory locations in terms of *blocks*. One block is 512 bytes. Thus block 1 starts at 0 bytes, block 2 at 512 bytes, and so on.

The matrix list is an array with one row per frame in the data.

Columns in the matrix list are:

- 0: Matrix identifier (frame number)
- 1: matrix data start block number (subheader followed by image data)
- 2: Last block number of matrix (image) data
- 3: Matrix status
  - 1: hxists - rw
  - 2: exists - ro
  - 3: matrix deleted

There is one sub-header for each image frame (or matrix in the terminology above). A sub-header can also be called an *image header*. The sub-header is one block (512 bytes), and the frame (image) data follows.

There is very little documentation of the ECAT format, and many of the comments in this code come from a combination of trial and error and wild speculation.

XMedcon can read and write ECAT 6 format, and read ECAT 7 format: see http://xmedcon.sourceforge.net and the ECAT files in the source of XMedCon, currently `libs/tpc/*ecat*` and `source/m-ecat*`. Unfortunately XMedCon is GPL and some of the header files are adapted from CTI files (called CTI code below). It's not clear what the licenses are for these files.

EcatHeader([binaryblock, endianness, check])	Class for basic Ecat PET header
EcatImage(dataobj, affine, header, ... [, ...])	Class returns a list of Ecat images, with one image(hdr/data) per frame
EcatImageArrayProxy(subheader)	Ecat implemention of array proxy protocol
EcatSubHeader(hdr, mlist, fileobj)	parses the subheaders in the ecat (.v) file there is one subheader for each frame in the ecat file
get_frame_order(mlist)	Returns the order of the frames stored in the file Sometimes Frames are not stored in the file in chronological order, this can be used to extract frames in correct order

continues on next page

Table 34 – continued from previous page

get_series_framenumbers(mlist)	Returns framenumber of data as it was collected, as part of a series; not just the order of how it was stored in this or across other files
read_mlist(fileobj, endianness)	read (nframes, 4) matrix list array from *fileobj*
read_subheaders(fileobj, mlist, endianness)	Retrieve all subheaders and return list of subheader recarrays

EcatHeader

class nibabel.ecat.**EcatHeader**(*binaryblock=None*, *endianness=None*, *check=True*)

Bases: *nibabel.wrapstruct.WrapStruct*

Class for basic Ecat PET header

Sub-parts of standard Ecat File

- main header
- matrix list which lists the information for each frame collected (can have 1 to many frames)
- subheaders specific to each frame with possibly-variable sized data blocks

This just reads the main Ecat Header, it does not load the data or read the mlist or any sub headers

Initialize Ecat header from bytes object

Parameters

binaryblock [{None, bytes} optional] binary block to set into header, By default, None in which case we insert default empty header block

endianness [{None, '<', '>', other endian code}, optional] endian code of binary block, If None, guess endianness from the data

check [{True, False}, optional] Whether to check and fix header for errors. No checks currently implemented, so value has no effect.

__init__(*binaryblock=None*, *endianness=None*, *check=True*)

Initialize Ecat header from bytes object

Parameters

binaryblock [{None, bytes} optional] binary block to set into header, By default, None in which case we insert default empty header block

endianness [{None, '<', '>', other endian code}, optional] endian code of binary block, If None, guess endianness from the data

check [{True, False}, optional] Whether to check and fix header for errors. No checks currently implemented, so value has no effect.

classmethod default_structarr(*endianness=None*)

Return header data for empty header with given endianness

get_data_dtype()

Get numpy dtype for data from header

get_filetype()

Type of ECAT Matrix File from code stored in header

get_patient_orient()

gets orientation of patient based on code stored in header, not always reliable

classmethod guessed_endian(*hdr*)
Guess endian from MAGIC NUMBER value of header data

template_dtype = dtype([('magic_number', 'S14'), ('original_filename', 'S32'), ('sw_ve

EcatImage

class nibabel.ecat.**EcatImage**(*dataobj*, *affine*, *header*, *subheader*, *mlist*, *extra=None*, *file_map=None*)
Bases: *nibabel.spatialimages.SpatialImage*

Class returns a list of Ecat images, with one image(hdr/data) per frame

Initialize Image

The image is a combination of (array, affine matrix, header, subheader, mlist) with optional meta data in *extra*, and filename / file-like objects contained in the *file_map*.

Parameters

dataobj [array-like] image data

affine [None or (4,4) array-like] homogeneous affine giving relationship between voxel coords and world coords.

header [None or header instance] meta data for this image format

subheader [None or subheader instance] meta data for each sub-image for frame in the image

mlist [None or array] Matrix list array giving offset and order of data in file

extra [None or mapping, optional] metadata associated with this image that cannot be stored in header or subheader

file_map [mapping, optional] mapping giving file information for this image format

Examples

```
>>> import os
>>> import nibabel as nib
>>> nibabel_dir = os.path.dirname(nib.__file__)
>>> from nibabel import ecat
>>> ecat_file = os.path.join(nibabel_dir,'tests','data','tinypet.v')
>>> img = ecat.load(ecat_file)
>>> frame0 = img.get_frame(0)
>>> frame0.shape == (10, 10, 3)
True
>>> data4d = img.get_fdata()
>>> data4d.shape == (10, 10, 3, 1)
True
```

__init__(*dataobj*, *affine*, *header*, *subheader*, *mlist*, *extra=None*, *file_map=None*)
Initialize Image

The image is a combination of (array, affine matrix, header, subheader, mlist) with optional meta data in *extra*, and filename / file-like objects contained in the *file_map*.

Parameters

dataobj [array-like] image data

affine [None or (4,4) array-like] homogeneous affine giving relationship between voxel coords and world coords.

header [None or header instance] meta data for this image format

subheader [None or subheader instance] meta data for each sub-image for frame in the image

mlist [None or array] Matrix list array giving offset and order of data in file

extra [None or mapping, optional] metadata associated with this image that cannot be stored in header or subheader

file_map [mapping, optional] mapping giving file information for this image format

Examples

```
>>> import os
>>> import nibabel as nib
>>> nibabel_dir = os.path.dirname(nib.__file__)
>>> from nibabel import ecat
>>> ecat_file = os.path.join(nibabel_dir,'tests','data','tinypet.v')
>>> img = ecat.load(ecat_file)
>>> frame0 = img.get_frame(0)
>>> frame0.shape == (10, 10, 3)
True
>>> data4d = img.get_fdata()
>>> data4d.shape == (10, 10, 3, 1)
True
```

ImageArrayProxy
alias of *nibabel.ecat.EcatImageArrayProxy*

property affine

files_types = (('image', '.v'), ('header', '.v'))

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)
class method to create image from mapping specified in file_map

classmethod from_filespec(*filespec*)
from_filespec class method is deprecated. Please use the `from_file_map` class method instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

classmethod from_image(*img*)
Class method to create new instance of own class from *img*

Parameters

img [`spatialimage` instance] In fact, an object with the API of `spatialimage` - specifically `dataobj`, `affine`, `header` and `extra`.

Returns

cimg [`spatialimage` instance] Image, of our own class

get_data_dtype(*frame*)

get_frame(*frame*, *orientation=None*)
Get full volume for a time frame

Parameters

- **frame** – Time frame index from where to fetch data
- **orientation** – None (default), 'neurological' or 'radiological'

Return type Numpy array containing (possibly oriented) raw data

get_frame_affine(*frame*)
returns 4X4 affine

get_mlist()
get access to the mlist

get_subheaders()
get access to subheaders

header_class
alias of *nibabel.ecat.EcatHeader*

classmethod load(*filespec*)
Class method to create image from filename *filename*

Parameters

filename [str] Filename of image to load

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [DataobjImage instance]

property shape

to_file_map(*file_map=None*)
Write ECAT7 image to *file_map* or contained `self.file_map`

The format consist of:

- **A main header (512L) with dictionary entries in the form** [numAvail, nextDir, previousDir, numUsed]
- For every frame (3D volume in 4D data) - A subheader (size = frame_offset) - Frame data (3D volume)

valid_exts = ('.v',)

EcatImageArrayProxy

class nibabel.ecat.**EcatImageArrayProxy**(*subheader*)
Bases: object

Ecat implemention of array proxy protocol

The array proxy allows us to freeze the passed fileobj and header such that it returns the expected data array.

__init__(*subheader*)
Initialize self. See help(type(self)) for accurate signature.

property is_proxy

property ndim

property shape

EcatSubHeader

class nibabel.ecat.**EcatSubHeader**(*hdr*, *mlist*, *fileobj*)
Bases: object

parses the subheaders in the ecat (.v) file there is one subheader for each frame in the ecat file

Parameters

hdr [EcatHeader] ECAT main header

mlist [array shape (N, 4)] Matrix list

fileobj [ECAT file <filename>.v fileholder or file object] with read, seek methods

__init__(*hdr*, *mlist*, *fileobj*)
parses the subheaders in the ecat (.v) file there is one subheader for each frame in the ecat file

Parameters

hdr [EcatHeader] ECAT main header

mlist [array shape (N, 4)] Matrix list

fileobj [ECAT file <filename>.v fileholder or file object] with read, seek methods

data_from_fileobj(*frame=0*, *orientation=None*)
Read scaled data from file for a given frame

Parameters

- **frame** – Time frame index from where to fetch data
- **orientation** – None (default), 'neurological' or 'radiological'

Return type Numpy array containing (possibly oriented) raw data

See also:

raw_data_from_fileobj

get_frame_affine(*frame=0*)
returns best affine for given frame of data

get_nframes()
returns number of frames

get_shape (*frame=0*)
returns shape of given frame

get_zooms (*frame=0*)
returns zooms . . . pixdims

raw_data_from_fileobj (*frame=0*, *orientation=None*)
Get raw data from file object.

Parameters

- **frame** – Time frame index from where to fetch data
- **orientation** – None (default), 'neurological' or 'radiological'

Return type Numpy array containing (possibly oriented) raw data

See also:

data_from_fileobj

get_frame_order

nibabel.ecat.**get_frame_order**(*mlist*)
Returns the order of the frames stored in the file Sometimes Frames are not stored in the file in chronological order, this can be used to extract frames in correct order

Returns

id_dict: dict mapping frame number -> [mlist_row, mlist_id]

(where mlist id is value in the first column of the mlist matrix)

Examples

```
>>> import os
>>> import nibabel as nib
>>> nibabel_dir = os.path.dirname(nib.__file__)
>>> from nibabel import ecat
>>> ecat_file = os.path.join(nibabel_dir,'tests','data','tinypet.v')
>>> img = ecat.load(ecat_file)
>>> mlist = img.get_mlist()
>>> get_frame_order(mlist)
{0: [0, 16842758]}
```

get_series_framenumbers

nibabel.ecat.**get_series_framenumbers**(*mlist*)
Returns framenumber of data as it was collected, as part of a series; not just the order of how it was stored in this or across other files

For example, if the data is split between multiple files this should give you the true location of this frame as collected in the series (Frames are numbered starting at ONE (1) not Zero)

Returns

frame_dict: dict mapping order_stored -> frame in series where frame in series counts from 1; [1,2,3,4. . .]

Examples

```
>>> import os
>>> import nibabel as nib
>>> nibabel_dir = os.path.dirname(nib.__file__)
>>> from nibabel import ecat
>>> ecat_file = os.path.join(nibabel_dir,'tests','data','tinypet.v')
>>> img = ecat.load(ecat_file)
>>> mlist = img.get_mlist()
>>> get_series_framenumbers(mlist)
{0: 1}
```

read_mlist

nibabel.ecat.**read_mlist**(*fileobj*, *endianness*)

read (nframes, 4) matrix list array from *fileobj*

Parameters

fileobj [file-like] an open file-like object implementing `seek` and `read`

Returns

mlist [(nframes, 4) ndarray] matrix list is an array with `nframes` rows and columns:

- 0: Matrix identifier (frame number)
- 1: matrix data start block number (subheader followed by image data)
- 2: Last block number of matrix (image) data
- 3: Matrix status
  - 1: hxists - rw
  - 2: exists - ro
  - 3: matrix deleted

Notes

A block is 512 bytes.

`block_no` in the code below is 1-based. block 1 is the main header, and the mlist blocks start at block number 2.

The 512 bytes in an mlist block contain 32 rows of the int32 (nframes, 4) mlist matrix.

The first row of these 32 looks like a special row. The 4 values appear to be (respectively):

- not sure - maybe negative number of mlist rows (out of 31) that are blank and not used in this block. Called *nfree* but unused in CTI code;
- block_no - of next set of mlist entries or 2 if no more entries. We also allow 1 or 0 to signal no more entries;
- <no idea>. Called *prvblk* in CTI code, so maybe previous block no;
- n_rows - number of mlist rows in this block (between ?0 and 31) (called *nused* in CTI code).

read_subheaders

nibabel.ecat.**read_subheaders**(*fileobj*, *mlist*, *endianness*)

Retrieve all subheaders and return list of subheader recarrays

Parameters

fileobj [file-like] implementing `read` and `seek`

mlist [(nframes, 4) ndarray] Columns are: * 0 - Matrix identifier. * 1 - subheader block number * 2 - Last block number of matrix data block. * 3 - Matrix status

endianness [{'<', '>'}] little / big endian code

Returns

subheaders [list] List of subheader structured arrays

parrec

Read images in PAR/REC format.

This is yet another MRI image format generated by Philips scanners. It is an ASCII header (PAR) plus a binary blob (REC).

This implementation aims to read version 4.0 through 4.2 of this format. Other versions could probably be supported, but we need example images to test against. If you want us to support another version, and have an image we can add to the test suite, let us know. You would make us very happy by submitting a pull request.

PAR file format

The PAR format appears to have two sections:

General information

This is a set of lines each giving one key : value pair, examples:

```
.    EPI factor        <0,1=no EPI>     :   39
.    Dynamic scan      <0=no 1=yes> ?   :   1
.    Diffusion         <0=no 1=yes> ?   :   0
```

(from `nibabel/tests/data/phantom_EPI_asc_CLEAR_2_1.PAR`)

Image information

There is a # prefixed list of fields under the heading "IMAGE INFORMATION DEFINITION". From the same file, here is the start of this list:

```
# === IMAGE INFORMATION DEFINITION =============================================
#  The rest of this file contains ONE line per image, this line contains the␣
↪following information:
#
#  slice number                             (integer)
#  echo number                              (integer)
#  dynamic scan number                      (integer)
```

There follows a space separated table with values for these fields, each row containing all the named values. Here are the first few lines from the example file above:

```
# === IMAGE INFORMATION ==========================================================
#  sl ec  dyn ph ty    idx pix scan% rec size                (re)scale                 ␣
↪window        angulation              offcentre        thick   gap   info       ␣
↪spacing     echo     dtime   ttime    diff  avg  flip    freq   RR-int  turbo delay␣
↪b grad cont anis         diffusion       L.ty

1   1    1  1 0 2     0  16    62   64   64     0.00000   1.29035 4.28404e-003  1070 ␣
↪1860 -13.26  -0.00  -0.00    2.51   -0.81   -8.69  6.000  2.000 0 1 0 2  3.750  3.
↪750  30.00    0.00     0.00    0.00   0   90.00     0    0    0    39   0.0  1   1 ␣
↪  8    0   0.000    0.000    0.000  1
2   1    1  1 0 2     1  16    62   64   64     0.00000   1.29035 4.28404e-003  1122 ␣
↪1951 -13.26  -0.00  -0.00    2.51    6.98  -10.53  6.000  2.000 0 1 0 2  3.750  3.
↪750  30.00    0.00     0.00    0.00   0   90.00     0    0    0    39   0.0  1   1 ␣
↪  8    0   0.000    0.000    0.000  1
3   1    1  1 0 2     2  16    62   64   64     0.00000   1.29035 4.28404e-003  1137 ␣
↪1977 -13.26  -0.00  -0.00    2.51   14.77  -12.36  6.000  2.000 0 1 0 2  3.750  3.
↪750  30.00    0.00     0.00    0.00   0   90.00     0    0    0    39   0.0  1   1 ␣
↪  8    0   0.000    0.000    0.000  1
```

Orientation

PAR files refer to orientations “ap”, “fh” and “rl”.

Nibabel’s required affine output axes are RAS (left to Right, posterior to Anterior, inferior to Superior). The correspondence of the PAR file’s axes to RAS axes is:

- ap = anterior -> posterior = negative A in RAS = P
- fh = foot -> head = S in RAS = S
- rl = right -> left = negative R in RAS = L

We therefore call the PAR file’s axis system “PSL” (Posterior, Superior, Left).

The orientation of the PAR file axes corresponds to DICOM’s LPS coordinate system (right to Left, anterior to Posterior, inferior to Superior), but in a different order.

Data type

It seems that everyone agrees that Philips stores REC data in little-endian format - see https://github.com/nipy/nibabel/issues/274

Philips XML header files, and some previous experience, suggest that the REC data is always stored as 8 or 16 bit unsigned integers - see https://github.com/nipy/nibabel/issues/275

Data Sorting

PAR/REC files have a large number of potential image dimensions. To handle sorting of volumes in PAR/REC files based on these fields and not the order slices first appear in the PAR file, the `strict_sort` flag of `nibabel.load` (or `parrec.load`) should be set to `True`. The fields that are taken into account during sorting are:

- slice number
- echo number
- cardiac phase number
- gradient orientation number
- diffusion b value number
- label type (ASL tag vs. control)
- dynamic scan number
- image_type_mr (Re, Im, Mag, Phase)

Slices are sorted into the third dimension and the order of preference for sorting along the 4th dimension corresponds to the order in the list above. If the image data has more than 4 dimensions these will all be concatenated along the 4th dimension. For example, for a scan with two echos and two dynamics, the 4th dimension will have both echos of dynamic 1 prior to the two echos for dynamic 2.

The``get_volume_labels`` method of the header returns a dictionary containing the PAR field labels for this 4th dimension.

The volume sorting described above can be enabled in the parrec2nii command utility via the option "–strict-sort". The dimension info can be exported to a CSV file by adding the option "–volume-info".

PARRECArrayProxy(file_like, header, *[, ...])	Initialize PARREC array proxy
PARRECError	Exception for PAR/REC format related problems.
PARRECHeader(info, image_defs[, ...])	PAR/REC header
PARRECImage(dataobj, affine[, header, ...])	PAR/REC image
exts2pars(exts_source)	Parse, return any PAR headers from NIfTI extensions in *exts_source*
one_line(long_str)	Make maybe mutli-line *long_str* into one long line
parse_PAR_header(fobj)	Parse a PAR header and aggregate all information into useful containers.
vol_is_full(slice_nos, slice_max[, slice_min])	Vector with True for slices in complete volume, False otherwise
vol_numbers(slice_nos)	Calculate volume numbers inferred from slice numbers *slice_nos*

PARRECArrayProxy

class nibabel.parrec.**PARRECArrayProxy**(*file_like*, *header*, *, *mmap=True*, *scaling='dv'*)

Bases: `object`

Initialize PARREC array proxy

Parameters

file_like [file-like object] Filename or object implementing `read, seek, tell`

header [PARRECHeader instance] Implementing `get_data_shape,`

get_data_dtype, get_sorted_slice_indices, get_data_scaling, get_rec_shape.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy memmap for data array. If one of {'c', 'r'}, try numpy memmap with mode=mmap. A *mmap* value of True gives the same behavior as mmap='c'. If *file_like* cannot be memory-mapped, ignore *mmap* value and read array from file.

scaling [{'fp', 'dv'}, optional, keyword only] Type of scaling to use - see header get_data_scaling method.

__init__(*file_like*, *header*, *, *mmap=True*, *scaling='dv'*)

Initialize PARREC array proxy

Parameters

file_like [file-like object] Filename or object implementing read, seek, tell

header [PARRECHeader instance] Implementing get_data_shape, get_data_dtype, get_sorted_slice_indices, get_data_scaling, get_rec_shape.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy memmap for data array. If one of {'c', 'r'}, try numpy memmap with mode=mmap. A *mmap* value of True gives the same behavior as mmap='c'. If *file_like* cannot be memory-mapped, ignore *mmap* value and read array from file.

scaling [{'fp', 'dv'}, optional, keyword only] Type of scaling to use - see header get_data_scaling method.

property dtype

get_unscaled()

Read data from file

This is an optional part of the proxy API

property is_proxy

property ndim

property shape

PARRECError

class nibabel.parrec.**PARRECError**

Bases: Exception

Exception for PAR/REC format related problems.

To be raised whenever PAR/REC is not happy, or we are not happy with PAR/REC.

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

PARRECHeader

class nibabel.parrec.**PARRECHeader**(*info*, *image_defs*, *permit_truncated=False*, *strict_sort=False*)

Bases: *nibabel.spatialimages.SpatialHeader*

PAR/REC header

Parameters

info [dict] "General information" from the PAR file (as returned by *parse_PAR_header()*).

image_defs [array] Structured array with image definitions from the PAR file (as returned by *parse_PAR_header()*).

permit_truncated [bool, optional] If True, a warning is emitted instead of an error when a truncated recording is detected.

strict_sort [bool, optional, keyword-only] If True, a larger number of header fields are used while sorting the REC data array. This may produce a different sort order than *strict_sort=False*, where volumes are sorted by the order in which the slices appear in the .PAR file.

__init__(*info*, *image_defs*, *permit_truncated=False*, *strict_sort=False*)

Parameters

info [dict] "General information" from the PAR file (as returned by *parse_PAR_header()*).

image_defs [array] Structured array with image definitions from the PAR file (as returned by *parse_PAR_header()*).

permit_truncated [bool, optional] If True, a warning is emitted instead of an error when a truncated recording is detected.

strict_sort [bool, optional, keyword-only] If True, a larger number of header fields are used while sorting the REC data array. This may produce a different sort order than *strict_sort=False*, where volumes are sorted by the order in which the slices appear in the .PAR file.

as_analyze_map()
Convert PAR parameters to NIFTI1 format

copy()
Copy object to independent representation

The copy should not be affected by any changes to the original object.

classmethod from_fileobj(*fileobj*, *permit_truncated=False*, *strict_sort=False*)

classmethod from_header(*header=None*)

get_affine(*origin='scanner'*)
Compute affine transformation into scanner space.

The method only considers global rotation and offset settings in the header and ignores potentially deviating information in the image definitions.

Parameters

origin [{'scanner', 'fov'}] Transformation origin. By default the transformation is computed relative to the scanner's iso center. If 'fov' is requested the transformation origin will be the center of the field of view instead.

Returns

aff [(4, 4) array] 4x4 array, with output axis order corresponding to RAS or (x,y,z) or (lr, pa, fh).

Notes

Transformations appear to be specified in (ap, fh, rl) axes. The orientation of data is recorded in the "slice orientation" field of the PAR header "General Information".

We need to:

- translate to coordinates in terms of the center of the FOV
- apply voxel size scaling
- reorder / flip the data to Philips' PSL axes
- apply the rotations
- apply any isocenter scaling offset if *origin* == "scanner"
- reorder and flip to RAS axes

get_bvals_bvecs()
Get bvals and bvecs from data

Returns

b_vals [None or array] Array of b values, shape (n_directions,), or None if not a diffusion acquisition.

b_vectors [None or array] Array of b vectors, shape (n_directions, 3), or None if not a diffusion acquisition.

get_data_offset()
PAR header always has 0 data offset (into REC file)

get_data_scaling(*method='dv'*)
Returns scaling slope and intercept.

Parameters

method [{'fp', 'dv'}] Scaling settings to be reported – see notes below.

Returns

slope [array] scaling slope

intercept [array] scaling intercept

Notes

The PAR header contains two different scaling settings: 'dv' (value on console) and 'fp' (floating point value). Here is how they are defined:

DV = PV * RS + RI FP = DV / (RS * SS)

where:

PV: value in REC RS: rescale slope RI: rescale intercept SS: scale slope

get_def(*name*)
Return a single image definition field (or None if missing)

get_echo_train_length()
Echo train length of the recording

get_q_vectors()
Get Q vectors from the data

Returns

q_vectors [None or array] Array of q vectors (bvals * bvecs), or None if not a diffusion acquisition.

get_rec_shape()

get_slice_orientation()
Returns the slice orientation label.

Returns

orientation [{'transverse', 'sagittal', 'coronal'}]

get_sorted_slice_indices()
Return indices to sort (and maybe discard) slices in REC file.

If the recording is truncated, the returned indices take care of discarding any slice indices from incomplete volumes.

If *self.strict_sort* is True, a more complicated sorting based on multiple fields from the .PAR file is used. This may produce a different sort order than *strict_sort=False*, where volumes are sorted by the order in which the slices appear in the .PAR file.

Returns

slice_indices [list] List for indexing into the last (third) dimension of the REC data array, and (equivalently) the only dimension of `self.image_defs`.

get_volume_labels()
Dynamic labels corresponding to the final data dimension(s).

This is useful for custom data sorting. A subset of the info in `self.image_defs` is returned in an order that matches the final data dimension(s). Only labels that have more than one unique value across the dataset will be returned.

Returns

sort_info [dict] Each key corresponds to volume labels for a dynamically varying sequence dimension. The ordering of the labels matches the volume ordering determined via `self.get_sorted_slice_indices`.

get_voxel_size()
Returns the spatial extent of a voxel.

get_voxel_size deprecated. Please use "get_zooms" instead.

- deprecated from version: 2.0
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

Does not include the slice gap in the slice extent.

If you need the slice thickness not including the slice gap, use `self.image_defs['slice thickness']`.

Returns

vox_size: shape (3,) ndarray

get_water_fat_shift()
Water fat shift, in pixels

set_data_offset(*offset*)
PAR header always has 0 data offset (into REC file)

PARRECImage

class nibabel.parrec.**PARRECImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Bases: *nibabel.spatialimages.SpatialImage*

PAR/REC image

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

ImageArrayProxy
alias of *nibabel.parrec.PARRECArrayProxy*

files_types = (('image', '.rec'), ('header', '.par'))

classmethod from_file_map(*file_map*, ***, *mmap=True*, *permit_truncated=False*, *scaling='dv'*, *strict_sort=False*)
Create PARREC image from file map *file_map*

Parameters

file_map [dict] dict with keys `image, header` and values being fileholder objects for the respective REC and PAR files.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

permit_truncated [{False, True}, optional, keyword-only] If False, raise an error for an image where the header shows signs that fewer slices / volumes were recorded than were expected.

scaling [{'dv', 'fp'}, optional, keyword-only] Scaling method to apply to data (see *PARRECHeader.get_data_scaling()*).

strict_sort [bool, optional, keyword-only] If True, a larger number of header fields are used while sorting the REC data array. This may produce a different sort order than *strict_sort=False*, where volumes are sorted by the order in which the slices appear in the .PAR file.

classmethod from_filename(*filename*, *, *mmap=True*, *permit_truncated=False*, *scaling='dv'*, *strict_sort=False*)

Create PARREC image from filename *filename*

Parameters

filename [str] Filename of "PAR" or "REC" file

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

permit_truncated [{False, True}, optional, keyword-only] If False, raise an error for an image where the header shows signs that fewer slices / volumes were recorded than were expected.

scaling [{'dv', 'fp'}, optional, keyword-only] Scaling method to apply to data (see *PARRECHeader.get_data_scaling()*).

strict_sort [bool, optional, keyword-only] If True, a larger number of header fields are used while sorting the REC data array. This may produce a different sort order than *strict_sort=False*, where volumes are sorted by the order in which the slices appear in the .PAR file.

header_class

alias of *nibabel.parrec.PARRECHeader*

classmethod load(*filename*, *, *mmap=True*, *permit_truncated=False*, *scaling='dv'*, *strict_sort=False*)

Create PARREC image from filename *filename*

Parameters

filename [str] Filename of "PAR" or "REC" file

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value

of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

permit_truncated [{False, True}, optional, keyword-only] If False, raise an error for an image where the header shows signs that fewer slices / volumes were recorded than were expected.

scaling [{'dv', 'fp'}, optional, keyword-only] Scaling method to apply to data (see *PARRECHeader.get_data_scaling()*).

strict_sort [bool, optional, keyword-only] If True, a larger number of header fields are used while sorting the REC data array. This may produce a different sort order than *strict_sort=False*, where volumes are sorted by the order in which the slices appear in the .PAR file.

makeable = False

rw = False

valid_exts = ('.rec', '.par')

exts2pars

nibabel.parrec.**exts2pars**(*exts_source*)

Parse, return any PAR headers from NIfTI extensions in *exts_source*

Parameters

exts_source [sequence or *Nifti1Image*, *Nifti1Header* instance] A sequence of extensions, or header containing NIfTI extensions, or an image containing a header with NIfTI extensions.

Returns

par_headers [list] A list of PARRECHeader objects, usually empty or with one element, each element contains a PARRECHeader read from the contained extensions.

one_line

nibabel.parrec.**one_line**(*long_str*)

Make maybe mutli-line *long_str* into one long line

parse_PAR_header

nibabel.parrec.**parse_PAR_header**(*fobj*)

Parse a PAR header and aggregate all information into useful containers.

Parameters

fobj [file-object] The PAR header file object.

Returns

general_info [dict] Contains all "General Information" from the header file

image_info [ndarray] Structured array with fields giving all "Image information" in the header

vol_is_full

nibabel.parrec.**vol_is_full**(*slice_nos*, *slice_max*, *slice_min=1*)

Vector with True for slices in complete volume, False otherwise

Parameters

slice_nos [sequence] Sequence of slice numbers, e.g. `[1, 2, 3, 4, 1, 2, 3, 4]`.

slice_max [int] Highest slice number for a full slice set. Slice set will be `range(slice_min, slice_max+1)`.

slice_min [int, optional] Lowest slice number for full slice set. Default is 1.

Returns

is_full [array] Bool vector with True for slices in full volumes, False for slices in partial volumes. A full volume is a volume with all slices in the `slice set` as defined above.

Raises

ValueError if any value in *slice_nos* is outside slice set indices.

vol_numbers

nibabel.parrec.**vol_numbers**(*slice_nos*)

Calculate volume numbers inferred from slice numbers *slice_nos*

The volume number for each slice is the number of times this slice number has occurred previously in the *slice_nos* sequence

Parameters

slice_nos [sequence] Sequence of slice numbers, e.g. `[1, 2, 3, 4, 1, 2, 3, 4]`.

Returns

vol_nos [list] A list, the same length of *slice_nos* giving the volume number for each corresponding slice number.

streamlines

Multiformat-capable streamline format read / write interface

detect_format(fileobj)	Returns the StreamlinesFile object guessed from the file-like object.
is_supported(fileobj)	Checks if the file-like object if supported by NiBabel.
load(fileobj[, lazy_load])	Loads streamlines in *RAS+* and *mm* space from a file-like object.
save(tractogram, filename, **kwargs)	Saves a tractogram to a file.

Module: `streamlines.array_sequence`

ArraySequence([iterable, buffer_size])	Sequence of ndarrays having variable first dimension sizes.
concatenate(seqs, axis)	Concatenates multiple *ArraySequence* objects along an axis.
create_arraysequences_from_generator(gen, n)	Creates *ArraySequence* objects from a generator yielding tuples
is_array_sequence(obj)	Return True if *obj* is an array sequence.
is_ndarray_of_int_or_bool(obj)	

Module: `streamlines.header`

Field class defining common header fields in tractogram files

Field()	Header fields common to multiple streamline file formats.

Module: `streamlines.tck`

Read / write access to TCK streamlines format.

TCK format is defined at http://mrtrix.readthedocs.io/en/latest/getting_started/image_data.html?highlight=format#tracks-file-format-tck

TckFile(tractogram[, header])	Convenience class to encapsulate TCK file format.

Module: `streamlines.tractogram`

LazyDict(*args, **kwargs)	Dictionary of generator functions.
LazyTractogram([streamlines, ...])	Lazy container for streamlines and their data information.
PerArrayDict([n_rows])	Dictionary for which key access can do slicing on the values.
PerArraySequenceDict([n_rows])	Dictionary for which key access can do slicing on the values.
SliceableDataDict(*args, **kwargs)	Dictionary for which key access can do slicing on the values.
Tractogram([streamlines, ...])	Container for streamlines and their data information.
TractogramItem(streamline, ...)	Class containing information about one streamline.
is_data_dict(obj)	True if *obj* seems to implement the `DataDict` API
is_lazy_dict(obj)	True if *obj* seems to implement the *LazyDict* API

Module: streamlines.tractogram_file

Define abstract interface for Tractogram file classes

DataError	Raised when data is missing or inconsistent in a tractogram file.
DataWarning	Base class for warnings about tractogram file data.
ExtensionWarning	Base class for warnings about tractogram file extension.
HeaderError	Raised when a tractogram file header contains invalid information.
HeaderWarning	Base class for warnings about tractogram file header.
TractogramFile(tractogram[, header])	Convenience class to encapsulate tractogram file format.
abstractclassmethod(callable)	

Module: streamlines.trk

TrkFile(tractogram[, header])	Convenience class to encapsulate TRK file format.
decode_value_from_name(encoded_name)	Decodes a value that has been encoded in the last bytes of a string.
encode_value_in_name(value, name[, max_name_len])	Return *name* as fixed-length string, appending *value* as string.
get_affine_rasmm_to_trackvis(header)	
get_affine_trackvis_to_rasmm(header)	Get affine mapping trackvis voxelmm space to RAS+ mm space

Module: streamlines.utils

get_affine_from_reference(ref)	Returns the affine defining the reference space.
peek_next(iterable)	Peek next element of iterable.

detect_format

nibabel.streamlines.**detect_format**(*fileobj*)

Returns the StreamlinesFile object guessed from the file-like object.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to a tractogram file (and ready to read from the beginning of the header)

Returns

tractogram_file [`TractogramFile` class] The class type guessed from the content of *fileobj*.

is_supported

nibabel.streamlines.**is_supported**(*fileobj*)

Checks if the file-like object if supported by NiBabel.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to a streamlines file (and ready to read from the beginning of the header)

Returns

is_supported [boolean]

load

nibabel.streamlines.**load**(*fileobj*, *lazy_load=False*)

Loads streamlines in *RAS+* and *mm* space from a file-like object.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to a streamlines file (and ready to read from the beginning of the streamlines file's header).

lazy_load [{False, True}, optional] If True, load streamlines in a lazy manner i.e. they will not be kept in memory and only be loaded when needed. Otherwise, load all streamlines in memory.

Returns

tractogram_file [`TractogramFile` object] Returns an instance of a `TractogramFile` containing data and metadata of the tractogram loaded from *fileobj*.

Notes

The streamline coordinate (0,0,0) refers to the center of the voxel.

save

nibabel.streamlines.**save**(*tractogram*, *filename*, ***kwargs*)

Saves a tractogram to a file.

Parameters

tractogram [`Tractogram` object or `TractogramFile` object] If `Tractogram` object, the file format will be guessed from *filename* and a `TractogramFile` object will be created using provided keyword arguments. If `TractogramFile` object, the file format is known and will be used to save its content to *filename*.

filename [str] Name of the file where the tractogram will be saved.

****kwargs** [keyword arguments] Keyword arguments passed to `TractogramFile` constructor. Should not be specified if *tractogram* is already an instance of `TractogramFile`.

ArraySequence

class nibabel.streamlines.array_sequence.**ArraySequence**(*iterable=None*, *buffer_size=4*)

Bases: object

Sequence of ndarrays having variable first dimension sizes.

This is a container that can store multiple ndarrays where each ndarray might have a different first dimension size but a *common* size for the remaining dimensions.

More generally, an instance of *ArraySequence* of length N is composed of N ndarrays of shape $(d_1, d_2, ...d_D)$ where d_1 can vary in length between arrays but $(d_2, ..., d_D)$ have to be the same for every ndarray.

Initialize array sequence instance

Parameters

iterable [None or iterable or *ArraySequence*, optional] If None, create an empty *ArraySequence* object. If iterable, create a *ArraySequence* object initialized from array-like objects yielded by the iterable. If *ArraySequence*, create a view (no memory is allocated). For an actual copy use *copy()* instead.

buffer_size [float, optional] Size (in Mb) for memory allocation when *iterable* is a generator.

__init__(*iterable=None*, *buffer_size=4*)

Initialize array sequence instance

Parameters

iterable [None or iterable or *ArraySequence*, optional] If None, create an empty *ArraySequence* object. If iterable, create a *ArraySequence* object initialized from array-like objects yielded by the iterable. If *ArraySequence*, create a view (no memory is allocated). For an actual copy use *copy()* instead.

buffer_size [float, optional] Size (in Mb) for memory allocation when *iterable* is a generator.

append(*element*, *cache_build=False*)

Appends *element* to this array sequence.

Append can be a lot faster if it knows that it is appending several elements instead of a single element. In that case it can cache the parameters it uses between append operations, in a "build cache". To tell append to do this, use `cache_build=True`. If you use `cache_build=True`, you need to finalize the append operations with *finalize_append()*.

Parameters

element [ndarray] Element to append. The shape must match already inserted elements shape except for the first dimension.

cache_build [{False, True}] Whether to save the build cache from this append routine. If True, append can assume it is the only player updating *self*, and the caller must finalize *self* after all append operations, with `self.finalize_append()`.

Returns

None

Notes

If you need to add multiple elements you should consider *ArraySequence.extend*.

property common_shape
Matching shape of the elements in this array sequence.

copy()
Creates a copy of this *ArraySequence* object.

Returns

seq_copy [*ArraySequence* instance] Copy of *self*.

Notes

We do not simply deepcopy this object because we have a chance to use less memory. For example, if the array sequence being copied is the result of a slicing operation on an array sequence.

property data
Elements in this array sequence.

'ArraySequence.data' property is deprecated. Please use the 'ArraySequence.get_data()' method instead

- deprecated from version: 3.0
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

extend(*elements*)
Appends all *elements* to this array sequence.

Parameters

elements [iterable of ndarrays or *ArraySequence* object] If iterable of ndarrays, each ndarray will be concatenated along the first dimension then appended to the data of this ArraySequence. If *ArraySequence* object, its data are simply appended to the data of this ArraySequence.

Returns

None

Notes

The shape of the elements to be added must match the one of the data of this *ArraySequence* except for the first dimension.

finalize_append()
Finalize process of appending several elements to *self*

append() can be a lot faster if it knows that it is appending several elements instead of a single element. To tell the append method this is the case, use `cache_build=True`. This method finalizes the series of append operations after a call to *append()* with `cache_build=True`.

get_data()
Returns a *copy* of the elements in this array sequence.

Notes

To modify the data on this array sequence, one can use in-place mathematical operators (e.g., *seq* += ...) or the use assignment operator (i.e, *seq[...]* = *value*).

property is_array_sequence

classmethod load(*filename*)
Loads a *ArraySequence* object from a .npz file.

save(*filename*)
Saves this *ArraySequence* object to a .npz file.

shrink_data()

property total_nb_rows
Total number of rows in this array sequence.

concatenate

nibabel.streamlines.array_sequence.**concatenate**(*seqs*, *axis*)
Concatenates multiple *ArraySequence* objects along an axis.

Parameters

seqs: iterable of :class:`ArraySequence` objects Sequences to concatenate.

axis [int] Axis along which the sequences will be concatenated.

Returns

new_seq: *ArraySequence* object New *ArraySequence* object which is the result of concatenating multiple sequences along the given axis.

create_arraysequences_from_generator

nibabel.streamlines.array_sequence.**create_arraysequences_from_generator**(*gen*, *n*, *buffer_sizes=None*)
Creates *ArraySequence* objects from a generator yielding tuples

Parameters

gen [generator] Generator yielding a size *n* tuple containing the values to put in the array sequences.

n [int] Number of ArraySequences object to create.

buffer_sizes [list of float, optional] Sizes (in Mb) for each ArraySequence's buffer.

is_array_sequence

nibabel.streamlines.array_sequence.**is_array_sequence**(*obj*)
Return True if *obj* is an array sequence.

is_ndarray_of_int_or_bool

nibabel.streamlines.array_sequence.**is_ndarray_of_int_or_bool**(*obj*)

Field

class nibabel.streamlines.header.**Field**
Bases: object

Header fields common to multiple streamline file formats.

In IPython, use *nibabel.streamlines.Field??* to list them.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

DIMENSIONS = 'dimensions'

ENDIANNESS = 'endianness'

MAGIC_NUMBER = 'magic_number'

METHOD = 'method'

NB_POINTS = 'nb_points'

NB_PROPERTIES_PER_STREAMLINE = 'nb_properties_per_streamline'

NB_SCALARS_PER_POINT = 'nb_scalars_per_point'

NB_STREAMLINES = 'nb_streamlines'

ORIGIN = 'origin'

STEP_SIZE = 'step_size'

VOXEL_ORDER = 'voxel_order'

VOXEL_SIZES = 'voxel_sizes'

VOXEL_TO_RASMM = 'voxel_to_rasmm'

TckFile

class nibabel.streamlines.tck.**TckFile**(*tractogram*, *header=None*)
Bases: *nibabel.streamlines.tractogram_file.TractogramFile*

Convenience class to encapsulate TCK file format.

Notes

MRtrix (so its file format: TCK) considers streamlines coordinates to be in world space (RAS+ and mm space). MRtrix refers to that space as the "real" or "scanner" space[1].

Moreover, when streamlines are mapped back to voxel space[2], a streamline point located at an integer coordinate (i,j,k) is considered to be at the center of the corresponding voxel. This is in contrast with TRK's internal convention where it would have referred to a corner.

NiBabel's streamlines internal representation follows the same convention as MRtrix.

Parameters

tractogram [`Tractogram` object] Tractogram that will be contained in this *`TckFile`*.

header [None or dict, optional] Metadata associated to this tractogram file. If None, make default empty header.

Notes

Streamlines of the tractogram are assumed to be in *RAS+* and *mm* space. It is also assumed that when streamlines are mapped back to voxel space, a streamline point located at an integer coordinate (i,j,k) is considered to be at the center of the corresponding voxel. This is in contrast with TRK's internal convention where it would have referred to a corner.

__init__(*tractogram*, *header=None*)

Parameters

tractogram [`Tractogram` object] Tractogram that will be contained in this *`TckFile`*.

header [None or dict, optional] Metadata associated to this tractogram file. If None, make default empty header.

Notes

Streamlines of the tractogram are assumed to be in *RAS+* and *mm* space. It is also assumed that when streamlines are mapped back to voxel space, a streamline point located at an integer coordinate (i,j,k) is considered to be at the center of the corresponding voxel. This is in contrast with TRK's internal convention where it would have referred to a corner.

EOF_DELIMITER = array([[inf, inf, inf]], dtype=float32)

FIBER_DELIMITER = array([[nan, nan, nan]], dtype=float32)

MAGIC_NUMBER = 'mrtrix tracks'

SUPPORTS_DATA_PER_POINT = False

SUPPORTS_DATA_PER_STREAMLINE = False

classmethod create_empty_header()
Return an empty compliant TCK header as dict

classmethod is_correct_format(*fileobj*)
Check if the file is in TCK format.

Parameters

[1] http://www.nitrc.org/pipermail/mrtrix-discussion/2014-January/000859.html
[2] http://nipy.org/nibabel/coordinate_systems.html#voxel-coordinates-are-in-voxel-space

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object in binary mode pointing to TCK file (and ready to read from the beginning of the TCK header). Note that calling this function does not change the file position.

Returns

is_correct_format [{True, False}] Returns True if *fileobj* is compatible with TCK format, otherwise returns False.

classmethod load(*fileobj*, *lazy_load=False*)

Loads streamlines from a filename or file-like object.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object in binary mode pointing to TCK file (and ready to read from the beginning of the TCK header). Note that calling this function does not change the file position.

lazy_load [{False, True}, optional] If True, load streamlines in a lazy manner i.e. they will not be kept in memory. Otherwise, load all streamlines in memory.

Returns

tck_file [*TckFile* object] Returns an object containing tractogram data and header information.

Notes

Streamlines of the tractogram are assumed to be in *RAS+* and *mm* space. It is also assumed that when streamlines are mapped back to voxel space, a streamline point located at an integer coordinate (i,j,k) is considered to be at the center of the corresponding voxel. This is in contrast with TRK's internal convention where it would have referred to a corner.

save(*fileobj*)

Save tractogram to a filename or file-like object using TCK format.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object in binary mode pointing to TCK file (and ready to write from the beginning of the TCK header data).

LazyDict

class nibabel.streamlines.tractogram.**LazyDict**(**args*, ***kwargs*)

Bases: collections.abc.MutableMapping

Dictionary of generator functions.

This container behaves like a dictionary but it makes sure its elements are callable objects that it assumes are generator functions yielding values. When getting the element associated with a given key, the element (i.e. a generator function) is first called before being returned.

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

LazyTractogram

class nibabel.streamlines.tractogram.**LazyTractogram**(*streamlines=None*, *data_per_streamline=None*, *data_per_point=None*, *affine_to_rasmm=None*)

Bases: *nibabel.streamlines.tractogram.Tractogram*

Lazy container for streamlines and their data information.

This container behaves lazily as it uses generator functions to manage streamlines and their data information. This container is thus memory friendly since it doesn't require having all this data loaded in memory.

Streamlines of a tractogram can be in any coordinate system of your choice as long as you provide the correct *affine_to_rasmm* matrix, at construction time. When applied to streamlines coordinates, that transformation matrix should bring the streamlines back to world space (RAS+ and mm space)[3].

Moreover, when streamlines are mapped back to voxel space[4], a streamline point located at an integer coordinate (i,j,k) is considered to be at the center of the corresponding voxel. This is in contrast with other conventions where it might have referred to a corner.

Notes

LazyTractogram objects do not support indexing currently. LazyTractogram objects are suited for operations that can be linearized such as applying an affine transformation or converting streamlines from one file format to another.

References

Attributes

streamlines [generator function] Generator function yielding streamlines. Each streamline is an ndarray of shape (N_t, 3) where N_t is the number of points of streamline t.

data_per_streamline [instance of *LazyDict*] Dictionary where the items are (str, instantiated generator). Each key represents a piece of information i to be kept alongside every streamline, and its associated value is a generator function yielding that information via ndarrays of shape (P_i,) where P_i is the number of values to store for that particular piece of information i.

data_per_point [*LazyDict* object] Dictionary where the items are (str, instantiated generator). Each key represents a piece of information i to be kept alongside every point of every streamline, and its associated value is a generator function yielding that information via ndarrays of shape (N_t, M_i) where N_t is the number of points for a particular streamline t and M_i is the number of values to store for that particular piece of information i.

Parameters

streamlines [generator function, optional] Generator function yielding streamlines. Each streamline is an ndarray of shape (N_t, 3) where N_t is the number of points of streamline t.

data_per_streamline [dict of generator functions, optional] Dictionary where the items are (str, generator function). Each key represents an information i to be kept alongside every streamline, and its associated value is a generator function yielding that information via ndarrays of shape (P_i,) where P_i is the number of values to store for that particular information i.

[3] http://nipy.org/nibabel/coordinate_systems.html#naming-reference-spaces
[4] http://nipy.org/nibabel/coordinate_systems.html#voxel-coordinates-are-in-voxel-space

data_per_point [dict of generator functions, optional] Dictionary where the items are (str, generator function). Each key represents an information i to be kept alongside every point of every streamline, and its associated value is a generator function yielding that information via ndarrays of shape (N_t, M_i) where N_t is the number of points for a particular streamline t and M_i is the number of values to store for that particular information i.

affine_to_rasmm [ndarray of shape (4, 4) or None, optional] Transformation matrix that brings the streamlines contained in this tractogram to *RAS+* and *mm* space where coordinate (0,0,0) refers to the center of the voxel. By default, the streamlines are in an unknown space, i.e. affine_to_rasmm is None.

__init__(*streamlines=None*, *data_per_streamline=None*, *data_per_point=None*, *affine_to_rasmm=None*)

Parameters

streamlines [generator function, optional] Generator function yielding streamlines. Each streamline is an ndarray of shape (N_t, 3) where N_t is the number of points of streamline t.

data_per_streamline [dict of generator functions, optional] Dictionary where the items are (str, generator function). Each key represents an information i to be kept alongside every streamline, and its associated value is a generator function yielding that information via ndarrays of shape (P_i,) where P_i is the number of values to store for that particular information i.

data_per_point [dict of generator functions, optional] Dictionary where the items are (str, generator function). Each key represents an information i to be kept alongside every point of every streamline, and its associated value is a generator function yielding that information via ndarrays of shape (N_t, M_i) where N_t is the number of points for a particular streamline t and M_i is the number of values to store for that particular information i.

affine_to_rasmm [ndarray of shape (4, 4) or None, optional] Transformation matrix that brings the streamlines contained in this tractogram to *RAS+* and *mm* space where coordinate (0,0,0) refers to the center of the voxel. By default, the streamlines are in an unknown space, i.e. affine_to_rasmm is None.

apply_affine(*affine*, *lazy=True*)

Applies an affine transformation to the streamlines.

The transformation given by the *affine* matrix is applied after any other pending transformations to the streamline points.

Parameters

affine [2D array (4,4)] Transformation matrix that will be applied on each streamline.

lazy [True, optional] Should always be True for *LazyTractogram* object. Doing otherwise will raise a ValueError.

Returns

lazy_tractogram [*LazyTractogram* object] A copy of this *LazyTractogram* instance but with a transformation to be applied on the streamlines.

copy()

Returns a copy of this *LazyTractogram* object.

property data

property data_per_point

property data_per_streamline

extend(*other*)
Appends the data of another *Tractogram*.

Data that will be appended includes the streamlines and the content of both dictionaries *data_per_streamline* and *data_per_point*.

Parameters

other [*Tractogram* object] Its data will be appended to the data of this tractogram.

Returns

None

Notes

The entries in both dictionaries *self.data_per_streamline* and *self.data_per_point* must match respectively those contained in the other tractogram.

classmethod from_data_func(*data_func*)
Creates an instance from a generator function.

The generator function must yield *TractogramItem* objects.

Parameters

data_func [generator function yielding *TractogramItem* objects] Generator function that whenever is called starts yielding *TractogramItem* objects that will be used to instantiate a *LazyTractogram*.

Returns

lazy_tractogram [*LazyTractogram* object] New lazy tractogram.

classmethod from_tractogram(*tractogram*)
Creates a *LazyTractogram* object from a *Tractogram* object.

Parameters

tractogram [Tractgogram object] Tractogram from which to create a *LazyTractogram* object.

Returns

lazy_tractogram [*LazyTractogram* object] New lazy tractogram.

property streamlines

to_world(*lazy=True*)
Brings the streamlines to world space (i.e. RAS+ and mm).

The transformation is applied after any other pending transformations to the streamline points.

Parameters

lazy [True, optional] Should always be True for *LazyTractogram* object. Doing otherwise will raise a ValueError.

Returns

lazy_tractogram [*LazyTractogram* object] A copy of this *LazyTractogram* instance but with a transformation to be applied on the streamlines.

PerArrayDict

class nibabel.streamlines.tractogram.**PerArrayDict**(*n_rows=0*, **args*, ***kwargs*)
Bases: *nibabel.streamlines.tractogram.SliceableDataDict*

Dictionary for which key access can do slicing on the values.

This container behaves like a standard dictionary but extends key access to allow keys for key access to be indices slicing into the contained ndarray values. The elements must also be ndarrays.

In addition, it makes sure the amount of data contained in those ndarrays matches the number of streamlines given at the instantiation of this instance.

Parameters

n_rows [None or int, optional] Number of rows per value in each key, value pair or None for not specified.

***args :**

****kwargs :** Positional and keyword arguments, passed straight through the `dict` constructor.

__init__(*n_rows=0*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

extend(*other*)
Appends the elements of another *PerArrayDict*.

That is, for each entry in this dictionary, we append the elements coming from the other dictionary at the corresponding entry.

Parameters

other [*PerArrayDict* object] Its data will be appended to the data of this dictionary.

Returns

None

Notes

The keys in both dictionaries must be the same.

PerArraySequenceDict

class nibabel.streamlines.tractogram.**PerArraySequenceDict**(*n_rows=0*, **args*, ***kwargs*)
Bases: *nibabel.streamlines.tractogram.PerArrayDict*

Dictionary for which key access can do slicing on the values.

This container behaves like a standard dictionary but extends key access to allow keys for key access to be indices slicing into the contained ndarray values. The elements must also be `ArraySequence`.

In addition, it makes sure the amount of data contained in those array sequences matches the number of elements given at the instantiation of the instance.

__init__(*n_rows=0*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

SliceableDataDict

class nibabel.streamlines.tractogram.**SliceableDataDict**(**args*, ***kwargs*)

Bases: collections.abc.MutableMapping

Dictionary for which key access can do slicing on the values.

This container behaves like a standard dictionary but extends key access to allow keys for key access to be indices slicing into the contained ndarray values.

Parameters

***args :**

****kwargs :** Positional and keyword arguments, passed straight through the `dict` constructor.

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Tractogram

class nibabel.streamlines.tractogram.**Tractogram**(*streamlines=None*, *data_per_streamline=None*, *data_per_point=None*, *affine_to_rasmm=None*)

Bases: object

Container for streamlines and their data information.

Streamlines of a tractogram can be in any coordinate system of your choice as long as you provide the correct *affine_to_rasmm* matrix, at construction time. When applied to streamlines coordinates, that transformation matrix should bring the streamlines back to world space (RAS+ and mm space)[5].

Moreover, when streamlines are mapped back to voxel space[6], a streamline point located at an integer coordinate (i,j,k) is considered to be at the center of the corresponding voxel. This is in contrast with other conventions where it might have referred to a corner.

References

Attributes

streamlines [ArraySequence object] Sequence of T streamlines. Each streamline is an ndarray of shape (N_t, 3) where N_t is the number of points of streamline t.

data_per_streamline [*PerArrayDict* object] Dictionary where the items are (str, 2D array). Each key represents a piece of information i to be kept alongside every streamline, and its associated value is a 2D array of shape (T, P_i) where T is the number of streamlines and P_i is the number of values to store for that particular piece of information i.

data_per_point [*PerArraySequenceDict* object] Dictionary where the items are (str, ArraySequence). Each key represents a piece of information i to be kept alongside every point of every streamline, and its associated value is an iterable of ndarrays of shape (N_t, M_i) where N_t is the number of points for a particular streamline t and M_i is the number values to store for that particular piece of information i.

Parameters

[5] http://nipy.org/nibabel/coordinate_systems.html#naming-reference-spaces

[6] http://nipy.org/nibabel/coordinate_systems.html#voxel-coordinates-are-in-voxel-space

streamlines [iterable of ndarrays or `ArraySequence`, optional] Sequence of T streamlines. Each streamline is an ndarray of shape (N_t, 3) where N_t is the number of points of streamline t.

data_per_streamline [dict of iterable of ndarrays, optional] Dictionary where the items are (str, iterable). Each key represents an information i to be kept alongside every streamline, and its associated value is an iterable of ndarrays of shape (P_i,) where P_i is the number of scalar values to store for that particular information i.

data_per_point [dict of iterable of ndarrays, optional] Dictionary where the items are (str, iterable). Each key represents an information i to be kept alongside every point of every streamline, and its associated value is an iterable of ndarrays of shape (N_t, M_i) where N_t is the number of points for a particular streamline t and M_i is the number scalar values to store for that particular information i.

affine_to_rasmm [ndarray of shape (4, 4) or None, optional] Transformation matrix that brings the streamlines contained in this tractogram to *RAS+* and *mm* space where coordinate (0,0,0) refers to the center of the voxel. By default, the streamlines are in an unknown space, i.e. affine_to_rasmm is None.

__init__(*streamlines=None*, *data_per_streamline=None*, *data_per_point=None*, *affine_to_rasmm=None*)

Parameters

streamlines [iterable of ndarrays or `ArraySequence`, optional] Sequence of T streamlines. Each streamline is an ndarray of shape (N_t, 3) where N_t is the number of points of streamline t.

data_per_streamline [dict of iterable of ndarrays, optional] Dictionary where the items are (str, iterable). Each key represents an information i to be kept alongside every streamline, and its associated value is an iterable of ndarrays of shape (P_i,) where P_i is the number of scalar values to store for that particular information i.

data_per_point [dict of iterable of ndarrays, optional] Dictionary where the items are (str, iterable). Each key represents an information i to be kept alongside every point of every streamline, and its associated value is an iterable of ndarrays of shape (N_t, M_i) where N_t is the number of points for a particular streamline t and M_i is the number scalar values to store for that particular information i.

affine_to_rasmm [ndarray of shape (4, 4) or None, optional] Transformation matrix that brings the streamlines contained in this tractogram to *RAS+* and *mm* space where coordinate (0,0,0) refers to the center of the voxel. By default, the streamlines are in an unknown space, i.e. affine_to_rasmm is None.

property affine_to_rasmm

Affine bringing streamlines in this tractogram to RAS+mm.

apply_affine(*affine*, *lazy=False*)

Applies an affine transformation on the points of each streamline.

If *lazy* is not specified, this is performed *in-place*.

Parameters

affine [ndarray of shape (4, 4)] Transformation that will be applied to every streamline.

lazy [{False, True}, optional] If True, streamlines are *not* transformed in-place and a *LazyTractogram* object is returned. Otherwise, streamlines are modified in-place.

Returns

tractogram [*Tractogram* or *LazyTractogram* object] Tractogram where the streamlines have been transformed according to the given affine transformation. If the *lazy* option is true, it returns a *LazyTractogram* object, otherwise it returns a reference to this *Tractogram* object with updated streamlines.

copy()
Returns a copy of this *Tractogram* object.

property data_per_point

property data_per_streamline

extend(*other*)
Appends the data of another *Tractogram*.

Data that will be appended includes the streamlines and the content of both dictionaries *data_per_streamline* and *data_per_point*.

Parameters

other [*Tractogram* object] Its data will be appended to the data of this tractogram.

Returns

None

Notes

The entries in both dictionaries *self.data_per_streamline* and *self.data_per_point* must match respectively those contained in the other tractogram.

property streamlines

to_world(*lazy=False*)
Brings the streamlines to world space (i.e. RAS+ and mm).

If *lazy* is not specified, this is performed *in-place*.

Parameters

lazy [{False, True}, optional] If True, streamlines are *not* transformed in-place and a *LazyTractogram* object is returned. Otherwise, streamlines are modified in-place.

Returns

tractogram [*Tractogram* or *LazyTractogram* object] Tractogram where the streamlines have been sent to world space. If the *lazy* option is true, it returns a *LazyTractogram* object, otherwise it returns a reference to this *Tractogram* object with updated streamlines.

TractogramItem

class nibabel.streamlines.tractogram.**TractogramItem**(*streamline*, *data_for_streamline*, *data_for_points*)

Bases: object

Class containing information about one streamline.

TractogramItem objects have three public attributes: *streamline*, *data_for_streamline*, and *data_for_points*.

Parameters

streamline [ndarray shape (N, 3)] Points of this streamline represented as an ndarray of shape (N, 3) where N is the number of points.

data_for_streamline [dict] Dictionary containing some data associated with this particular streamline. Each key `k` is mapped to a ndarray of shape (Pt,), where `Pt` is the dimension of the data associated with key `k`.

data_for_points [dict] Dictionary containing some data associated to each point of this particular streamline. Each key `k` is mapped to a ndarray of shape (Nt, Mk), where `Nt` is the number of points of this streamline and `Mk` is the dimension of the data associated with key `k`.

__init__(*streamline*, *data_for_streamline*, *data_for_points*)
Initialize self. See help(type(self)) for accurate signature.

is_data_dict

nibabel.streamlines.tractogram.**is_data_dict**(*obj*)
True if *obj* seems to implement the `DataDict` API

is_lazy_dict

nibabel.streamlines.tractogram.**is_lazy_dict**(*obj*)
True if *obj* seems to implement the *LazyDict* API

DataError

class nibabel.streamlines.tractogram_file.**DataError**
Bases: `Exception`

Raised when data is missing or inconsistent in a tractogram file.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

DataWarning

class nibabel.streamlines.tractogram_file.**DataWarning**
Bases: `Warning`

Base class for warnings about tractogram file data.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

ExtensionWarning

class nibabel.streamlines.tractogram_file.**ExtensionWarning**
Bases: Warning

Base class for warnings about tractogram file extension.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

HeaderError

class nibabel.streamlines.tractogram_file.**HeaderError**
Bases: Exception

Raised when a tractogram file header contains invalid information.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

HeaderWarning

class nibabel.streamlines.tractogram_file.**HeaderWarning**
Bases: Warning

Base class for warnings about tractogram file header.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

TractogramFile

class nibabel.streamlines.tractogram_file.**TractogramFile**(*tractogram*, *header=None*)
Bases: abc.ABC

Convenience class to encapsulate tractogram file format.

__init__(*tractogram*, *header=None*)
Initialize self. See help(type(self)) for accurate signature.

property affine
voxmm -> rasmm affine.

classmethod create_empty_header()
Returns an empty header for this streamlines file format.

property header

abstract classmethod is_correct_format(*fileobj*)
Checks if the file has the right streamlines file format.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to a streamlines file (and ready to read from the beginning of the header).

Returns

is_correct_format [{True, False}] Returns True if *fileobj* is in the right streamlines file format, otherwise returns False.

abstract classmethod load(*fileobj*, *lazy_load=True*)
Loads streamlines from a filename or file-like object.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to a streamlines file (and ready to read from the beginning of the header).

lazy_load [{False, True}, optional] If True, load streamlines in a lazy manner i.e. they will not be kept in memory. Otherwise, load all streamlines in memory.

Returns

tractogram_file [*TractogramFile* object] Returns an object containing tractogram data and header information.

abstract save(*fileobj*)
Saves streamlines to a filename or file-like object.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object opened and ready to write.

property streamlines

property tractogram

abstractclassmethod

class nibabel.streamlines.tractogram_file.**abstractclassmethod**(*callable*)
Bases: classmethod

__init__(*callable*)
Initialize self. See help(type(self)) for accurate signature.

TrkFile

class nibabel.streamlines.trk.**TrkFile**(*tractogram*, *header=None*)
Bases: *nibabel.streamlines.tractogram_file.TractogramFile*

Convenience class to encapsulate TRK file format.

Notes

TrackVis (so its file format: TRK) considers the streamline coordinate (0,0,0) to be in the corner of the voxel whereas NiBabel's streamlines internal representation (Voxel space) assumes (0,0,0) to be in the center of the voxel.

Thus, streamlines are shifted by half a voxel on load and are shifted back on save.

Parameters

tractogram [Tractogram object] Tractogram that will be contained in this *TrkFile*.

header [dict, optional] Metadata associated to this tractogram file.

Notes

Streamlines of the tractogram are assumed to be in *RAS+* and *mm* space where coordinate (0,0,0) refers to the center of the voxel.

__init__(*tractogram*, *header=None*)

Parameters

tractogram [`Tractogram` object] Tractogram that will be contained in this *TrkFile*.

header [dict, optional] Metadata associated to this tractogram file.

Notes

Streamlines of the tractogram are assumed to be in *RAS+* and *mm* space where coordinate (0,0,0) refers to the center of the voxel.

HEADER_SIZE = 1000

MAGIC_NUMBER = b'TRACK'

SUPPORTS_DATA_PER_POINT = True

SUPPORTS_DATA_PER_STREAMLINE = True

classmethod create_empty_header(*endianness=None*)
Return an empty compliant TRK header as dict

classmethod is_correct_format(*fileobj*)
Check if the file is in TRK format.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to TRK file (and ready to read from the beginning of the TRK header data). Note that calling this function does not change the file position.

Returns

is_correct_format [{True, False}] Returns True if *fileobj* is compatible with TRK format, otherwise returns False.

classmethod load(*fileobj*, *lazy_load=False*)
Loads streamlines from a filename or file-like object.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to TRK file (and ready to read from the beginning of the TRK header). Note that calling this function does not change the file position.

lazy_load [{False, True}, optional] If True, load streamlines in a lazy manner i.e. they will not be kept in memory. Otherwise, load all streamlines in memory.

Returns

trk_file [*TrkFile* object] Returns an object containing tractogram data and header information.

Notes

Streamlines of the returned tractogram are assumed to be in *RAS* and *mm* space where coordinate (0,0,0) refers to the center of the voxel.

save(*fileobj*)
Save tractogram to a filename or file-like object using TRK format.

Parameters

fileobj [string or file-like object] If string, a filename; otherwise an open file-like object pointing to TRK file (and ready to write from the beginning of the TRK header data).

decode_value_from_name

nibabel.streamlines.trk.**decode_value_from_name**(*encoded_name*)
Decodes a value that has been encoded in the last bytes of a string.

Check *encode_value_in_name()* to see how the value has been encoded.

Parameters

encoded_name [bytes] Name in which a value has been encoded or not.

Returns

name [bytes] Name without the encoded value.

value [int] Value decoded from the name.

encode_value_in_name

nibabel.streamlines.trk.**encode_value_in_name**(*value*, *name*, *max_name_len=20*)
Return *name* as fixed-length string, appending *value* as string.

Form output from *name* if *value* <= *1* else *name* + \ + str(value).

Return output as fixed length string length *max_name_len*, padded with \.

This function also verifies that the modified length of name is less than *max_name_len*.

Parameters

value [int] Integer value to encode.

name [str] Name to which we may append an ascii / latin-1 representation of *value*.

max_name_len [int, optional] Maximum length of byte string that output can have.

Returns

encoded_name [bytes] Name maybe followed by \ and ascii / latin-1 representation of *value*, padded with \ bytes.

get_affine_rasmm_to_trackvis

nibabel.streamlines.trk.**get_affine_rasmm_to_trackvis**(*header*)

get_affine_trackvis_to_rasmm

nibabel.streamlines.trk.**get_affine_trackvis_to_rasmm**(*header*)

Get affine mapping trackvis voxelmm space to RAS+ mm space

The streamlines in a trackvis file are in 'voxelmm' space, where the coordinates refer to the corner of the voxel.

Compute the affine matrix that will bring them back to RAS+ mm space, where the coordinates refer to the center of the voxel.

Parameters

header [dict or ndarray] Dict or numpy structured array containing trackvis header.

Returns

aff_tv2ras [shape (4, 4) array] Affine array mapping coordinates in 'voxelmm' space to RAS+ mm space.

get_affine_from_reference

nibabel.streamlines.utils.**get_affine_from_reference**(*ref*)

Returns the affine defining the reference space.

Parameters

ref [str or `Nifti1Image` object or ndarray shape (4, 4)] If str then it's the filename of reference file that will be loaded using `nibabel.load()` in order to obtain the affine. If `Nifti1Image` object then the affine is obtained from it. If ndarray shape (4, 4) then it's the affine.

Returns

affine [ndarray (4, 4)] Transformation matrix mapping voxel space to RAS+mm space.

peek_next

nibabel.streamlines.utils.**peek_next**(*iterable*)

Peek next element of iterable.

Parameters

iterable Iterable to peek the next element from.

Returns

next_item Element peeked from *iterable*.

new_iterable Iterable behaving like if the original *iterable* was untouched.

10.5.3 Image Utilities

eulerangles	Module implementing Euler angle rotations and their conversions
funcs	Processor functions for images
imageclasses	Define supported image classes and names
imageglobals	Defaults for images and headers
loadsave	Utilities to load and save image objects
orientations	Utilities for calculating and applying affine orientations
quaternions	Functions to operate on, or return, quaternions.
spatialimages	A simple spatial image class
volumeutils	Utility functions for analyze-like formats

eulerangles

Module implementing Euler angle rotations and their conversions

See:

- https://en.wikipedia.org/wiki/Rotation_matrix
- https://en.wikipedia.org/wiki/Euler_angles
- http://mathworld.wolfram.com/EulerAngles.html

See also: *Representing Attitude with Euler Angles and Quaternions: A Reference* (2006) by James Diebel. A cached PDF link last found here:

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.5134

Euler's rotation theorem tells us that any rotation in 3D can be described by 3 angles. Let's call the 3 angles the *Euler angle vector* and call the angles in the vector $alpha$, $beta$ and $gamma$. The vector is [$alpha$, $beta.$ $gamma$] and, in this description, the order of the parameters specifies the order in which the rotations occur (so the rotation corresponding to $alpha$ is applied first).

In order to specify the meaning of an *Euler angle vector* we need to specify the axes around which each of the rotations corresponding to $alpha$, $beta$ and $gamma$ will occur.

There are therefore three axes for the rotations $alpha$, $beta$ and $gamma$; let's call them i j, k.

Let us express the rotation $alpha$ around axis *i* as a 3 by 3 rotation matrix *A*. Similarly $beta$ around *j* becomes 3 x 3 matrix *B* and $gamma$ around *k* becomes matrix *G*. Then the whole rotation expressed by the Euler angle vector [$alpha$, $beta.$ $gamma$], *R* is given by:

```
R = np.dot(G, np.dot(B, A))
```

See http://mathworld.wolfram.com/EulerAngles.html

The order GBA expresses the fact that the rotations are performed in the order of the vector ($alpha$ around axis *i* = *A* first).

To convert a given Euler angle vector to a meaningful rotation, and a rotation matrix, we need to define:

- the axes *i*, *j*, *k*
- whether a rotation matrix should be applied on the left of a vector to be transformed (vectors are column vectors) or on the right (vectors are row vectors).

- whether the rotations move the axes as they are applied (intrinsic rotations) - compared the situation where the axes stay fixed and the vectors move within the axis frame (extrinsic)
- the handedness of the coordinate system

See: https://en.wikipedia.org/wiki/Rotation_matrix#Ambiguities

We are using the following conventions:

- axes *i*, *j*, *k* are the *z*, *y*, and *x* axes respectively. Thus an Euler angle vector [$alpha$, $beta$. $gamma$] in our convention implies a $alpha$ radian rotation around the *z* axis, followed by a $beta$ rotation around the *y* axis, followed by a $gamma$ rotation around the *x* axis.
- the rotation matrix applies on the left, to column vectors on the right, so if *R* is the rotation matrix, and *v* is a 3 x N matrix with N column vectors, the transformed vector set *vdash* is given by `vdash = np.dot(R, v)`.
- extrinsic rotations - the axes are fixed, and do not move with the rotations.
- a right-handed coordinate system

The convention of rotation around `z`, followed by rotation around `y`, followed by rotation around `x`, is known (confusingly) as "xyz", pitch-roll-yaw, Cardan angles, or Tait-Bryan angles.

angle_axis2euler(theta, vector[, is_normalized])	Convert angle, axis pair to Euler angles
euler2angle_axis([z, y, x])	Return angle, axis corresponding to these Euler angles
euler2mat([z, y, x])	Return matrix for rotations around z, y and x axes
euler2quat([z, y, x])	Return quaternion corresponding to these Euler angles
mat2euler(M[, cy_thresh])	Discover Euler angle vector from 3x3 matrix
quat2euler(q)	Return Euler angles corresponding to quaternion *q*

angle_axis2euler

nibabel.eulerangles.**angle_axis2euler**(*theta*, *vector*, *is_normalized=False*)

Convert angle, axis pair to Euler angles

Parameters

theta [scalar] angle of rotation

vector [3 element sequence] vector specifying axis for rotation.

is_normalized [bool, optional] True if vector is already normalized (has norm of 1). Default False

Returns

z [scalar]

y [scalar]

x [scalar] Rotations in radians around z, y, x axes, respectively

Notes

It's possible to reduce the amount of calculation a little, by combining parts of the `angle_axis2mat` and `mat2euler` functions, but the reduction in computation is small, and the code repetition is large.

Examples

```
>>> z, y, x = angle_axis2euler(0, [1, 0, 0])
>>> np.allclose((z, y, x), 0)
True
```

euler2angle_axis

nibabel.eulerangles.**euler2angle_axis**(*z=0*, *y=0*, *x=0*)

Return angle, axis corresponding to these Euler angles

Uses the z, then y, then x convention above

Parameters

z [scalar] Rotation angle in radians around z-axis (performed first)

y [scalar] Rotation angle in radians around y-axis

x [scalar] Rotation angle in radians around x-axis (performed last)

Returns

theta [scalar] angle of rotation

vector [array shape (3,)] axis around which rotation occurs

Examples

```
>>> theta, vec = euler2angle_axis(0, 1.5, 0)
>>> print(theta)
1.5
>>> np.allclose(vec, [0, 1, 0])
True
```

euler2mat

nibabel.eulerangles.**euler2mat**(*z=0*, *y=0*, *x=0*)

Return matrix for rotations around z, y and x axes

Uses the z, then y, then x convention above

Parameters

z [scalar] Rotation angle in radians around z-axis (performed first)

y [scalar] Rotation angle in radians around y-axis

x [scalar] Rotation angle in radians around x-axis (performed last)

Returns

M [array shape (3,3)] Rotation matrix giving same rotation as for given angles

Notes

The direction of rotation is given by the right-hand rule (orient the thumb of the right hand along the axis around which the rotation occurs, with the end of the thumb at the positive end of the axis; curl your fingers; the direction your fingers curl is the direction of rotation). Therefore, the rotations are counterclockwise if looking along the axis of rotation from positive to negative.

Examples

```
>>> zrot = 1.3 # radians
>>> yrot = -0.1
>>> xrot = 0.2
>>> M = euler2mat(zrot, yrot, xrot)
>>> M.shape == (3, 3)
True
```

The output rotation matrix is equal to the composition of the individual rotations

```
>>> M1 = euler2mat(zrot)
>>> M2 = euler2mat(0, yrot)
>>> M3 = euler2mat(0, 0, xrot)
>>> composed_M = np.dot(M3, np.dot(M2, M1))
>>> np.allclose(M, composed_M)
True
```

You can specify rotations by named arguments

```
>>> np.all(M3 == euler2mat(x=xrot))
True
```

When applying M to a vector, the vector should column vector to the right of M. If the right hand side is a 2D array rather than a vector, then each column of the 2D array represents a vector.

```
>>> vec = np.array([1, 0, 0]).reshape((3,1))
>>> v2 = np.dot(M, vec)
>>> vecs = np.array([[1, 0, 0],[0, 1, 0]]).T # giving 3x2 array
>>> vecs2 = np.dot(M, vecs)
```

Rotations are counter-clockwise.

```
>>> zred = np.dot(euler2mat(z=np.pi/2), np.eye(3))
>>> np.allclose(zred, [[0, -1, 0],[1, 0, 0], [0, 0, 1]])
True
>>> yred = np.dot(euler2mat(y=np.pi/2), np.eye(3))
>>> np.allclose(yred, [[0, 0, 1],[0, 1, 0], [-1, 0, 0]])
True
>>> xred = np.dot(euler2mat(x=np.pi/2), np.eye(3))
>>> np.allclose(xred, [[1, 0, 0],[0, 0, -1], [0, 1, 0]])
True
```

euler2quat

nibabel.eulerangles.**euler2quat**(*z=0*, *y=0*, *x=0*)

Return quaternion corresponding to these Euler angles

Uses the z, then y, then x convention above

Parameters

z [scalar] Rotation angle in radians around z-axis (performed first)

y [scalar] Rotation angle in radians around y-axis

x [scalar] Rotation angle in radians around x-axis (performed last)

Returns

quat [array shape (4,)] Quaternion in w, x, y z (real, then vector) format

Notes

We can derive this formula in Sympy using:

1. Formula giving quaternion corresponding to rotation of theta radians about arbitrary axis: http://mathworld.wolfram.com/EulerParameters.html
2. Generated formulae from 1.) for quaternions corresponding to theta radians rotations about `x, y, z` axes
3. Apply quaternion multiplication formula - https://en.wikipedia.org/wiki/Quaternions#Hamilton_product - to formulae from 2.) to give formula for combined rotations.

mat2euler

nibabel.eulerangles.**mat2euler**(*M*, *cy_thresh=None*)

Discover Euler angle vector from 3x3 matrix

Uses the conventions above.

Parameters

M [array-like, shape (3,3)]

cy_thresh [None or scalar, optional] threshold below which to give up on straightforward arctan for estimating x rotation. If None (default), estimate from precision of input.

Returns

z [scalar]

y [scalar]

x [scalar] Rotations in radians around z, y, x axes, respectively

Notes

If there was no numerical error, the routine could be derived using Sympy expression for z then y then x rotation matrix, which is:

```
[                       cos(y)*cos(z),                       -cos(y)*sin(z),
↪   sin(y)],
[cos(x)*sin(z) + cos(z)*sin(x)*sin(y), cos(x)*cos(z) - sin(x)*sin(y)*sin(z), -
↪cos(y)*sin(x)],
[sin(x)*sin(z) - cos(x)*cos(z)*sin(y), cos(z)*sin(x) + cos(x)*sin(y)*sin(z),
↪cos(x)*cos(y)]
```

with the obvious derivations for z, y, and x

z = atan2(-r12, r11) y = asin(r13) x = atan2(-r23, r33)

Problems arise when cos(y) is close to zero, because both of:

```
z = atan2(cos(y)*sin(z), cos(y)*cos(z))
x = atan2(cos(y)*sin(x), cos(x)*cos(y))
```

will be close to atan2(0, 0), and highly unstable.

The `cy` fix for numerical instability below is from: *Graphics Gems IV*, Paul Heckbert (editor), Academic Press, 1994, ISBN: 0123361559. Specifically it comes from EulerAngles.c by Ken Shoemake, and deals with the case where cos(y) is close to zero:

See: http://www.graphicsgems.org/

The code appears to be licensed (from the website) as "can be used without restrictions".

quat2euler

`nibabel.eulerangles.`**`quat2euler`**`(`*`q`*`)`

Return Euler angles corresponding to quaternion *q*

Parameters

q [4 element sequence] w, x, y, z of quaternion

Returns

z [scalar] Rotation angle in radians around z-axis (performed first)

y [scalar] Rotation angle in radians around y-axis

x [scalar] Rotation angle in radians around x-axis (performed last)

Notes

It's possible to reduce the amount of calculation a little, by combining parts of the `quat2mat` and `mat2euler` functions, but the reduction in computation is small, and the code repetition is large.

funcs

Processor functions for images

as_closest_canonical(img[, enforce_diag])	Return *img* with data reordered to be closest to canonical
concat_images(images[, check_affines, axis])	Concatenate images in list to single image, along specified dimension
four_to_three(img)	Create 3D images from 4D image by slicing over last axis
squeeze_image(img)	Return image, remove axes length 1 at end of image shape

as_closest_canonical

nibabel.funcs.**as_closest_canonical**(*img*, *enforce_diag=False*)

Return *img* with data reordered to be closest to canonical

Canonical order is the ordering of the output axes.

Parameters

img [spatialimage]

enforce_diag [{False, True}, optional] If True, before transforming image, check if the resulting image affine will be close to diagonal, and if not, raise an error

Returns

canonical_img [spatialimage] Version of *img* where the underlying array may have been reordered and / or flipped so that axes 0,1,2 are those axes in the input data that are, respectively, closest to the output axis orientation. We modify the affine accordingly. If *img* is already has the correct data ordering, we just return *img* unmodified.

concat_images

nibabel.funcs.**concat_images**(*images*, *check_affines=True*, *axis=None*)

Concatenate images in list to single image, along specified dimension

Parameters

images [sequence] sequence of SpatialImage or filenames of the same dimensionalitys

check_affines [{True, False}, optional] If True, then check that all the affines for *images* are nearly the same, raising a ValueError otherwise. Default is True

axis [None or int, optional] If None, concatenates on a new dimension. This requires all images to be the same shape. If not None, concatenates on the specified dimension. This requires all images to be the same shape, except on the specified dimension.

Returns

concat_img [SpatialImage] New image resulting from concatenating *images* across last dimension

four_to_three

nibabel.funcs.**four_to_three**(*img*)

Create 3D images from 4D image by slicing over last axis

Parameters

img [image] 4D image instance of some class with methods `get_data`, `header` and `affine`, and a class constructor allowing klass(data, affine, header)

Returns

imgs [list] list of 3D images

squeeze_image

nibabel.funcs.**squeeze_image**(*img*)

Return image, remove axes length 1 at end of image shape

For example, an image may have shape (10,20,30,1,1). In this case squeeze will result in an image with shape (10,20,30). See doctests for further description of behavior.

Parameters

img [`SpatialImage`]

Returns

squeezed_img [`SpatialImage`] Copy of img, such that data, and data shape have been squeezed, for dimensions > 3rd, and at the end of the shape list

Examples

```
>>> import nibabel as nf
>>> shape = (10,20,30,1,1)
>>> data = np.arange(np.prod(shape)).reshape(shape)
>>> affine = np.eye(4)
>>> img = nf.Nifti1Image(data, affine)
>>> img.shape == (10, 20, 30, 1, 1)
True
>>> img2 = squeeze_image(img)
>>> img2.shape == (10, 20, 30)
True
```

If the data are 3D then last dimensions of 1 are ignored

```
>>> shape = (10,1,1)
>>> data = np.arange(np.prod(shape)).reshape(shape)
>>> img = nf.ni1.Nifti1Image(data, affine)
>>> img.shape == (10, 1, 1)
True
>>> img2 = squeeze_image(img)
>>> img2.shape == (10, 1, 1)
True
```

Only *final* dimensions of 1 are squeezed

```
>>> shape = (1, 1, 5, 1, 2, 1, 1)
>>> data = data.reshape(shape)
>>> img = nf.ni1.Nifti1Image(data, affine)
>>> img.shape == (1, 1, 5, 1, 2, 1, 1)
True
>>> img2 = squeeze_image(img)
>>> img2.shape == (1, 1, 5, 1, 2)
True
```

imageclasses

Define supported image classes and names

ClassMapDict()	

ExtMapRecoder(codes[, fields, map_maker])	Create recoder object
spatial_axes_first(img)	True if spatial image axes for *img* always preceed other axes

ClassMapDict

class nibabel.imageclasses.**ClassMapDict**(*) -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Bases: dict

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

ExtMapRecoder

class nibabel.imageclasses.**ExtMapRecoder**(*codes*, *fields=('code',)*, *map_maker=<class 'collections.OrderedDict'>*)

Bases: *nibabel.volumeutils.Recoder*

Create recoder object

`codes` give a sequence of code, alias sequences `fields` are names by which the entries in these sequences can be accessed.

By default `fields` gives the first column the name "code". The first column is the vector of first entries in each of the sequences found in `codes`. Thence you can get the equivalent first column value with ob.code[value], where value can be a first column value, or a value in any of the other columns in that sequence.

You can give other columns names too, and access them in the same way - see the examples in the class docstring.

Parameters

codes [sequence of sequences] Each sequence defines values (codes) that are equivalent

fields [{('code',) string sequence}, optional] names by which elements in sequences can be accessed

map_maker: callable, optional constructor for dict-like objects used to store key value pairs. Default is `dict`. `map_maker()` generates an empty mapping. The mapping need only implement `__getitem__`, `__setitem__`, `keys`, `values`.

__init__(*codes*, *fields=('code',)*, *map_maker=<class 'collections.OrderedDict'>*)
Create recoder object

`codes` give a sequence of code, alias sequences `fields` are names by which the entries in these sequences can be accessed.

By default `fields` gives the first column the name "code". The first column is the vector of first entries in each of the sequences found in `codes`. Thence you can get the equivalent first column value with ob.code[value], where value can be a first column value, or a value in any of the other columns in that sequence.

You can give other columns names too, and access them in the same way - see the examples in the class docstring.

Parameters

codes [sequence of sequences] Each sequence defines values (codes) that are equivalent

fields [{('code',) string sequence}, optional] names by which elements in sequences can be accessed

map_maker: callable, optional constructor for dict-like objects used to store key value pairs. Default is `dict`. `map_maker()` generates an empty mapping. The mapping need only implement `__getitem__`, `__setitem__`, `keys`, `values`.

spatial_axes_first

nibabel.imageclasses.**spatial_axes_first**(*img*)
True if spatial image axes for *img* always preceed other axes

Parameters

img [object] Image object implementing at least `shape` attribute.

Returns

spatial_axes_first [bool] True if image only has spatial axes (number of axes < 4) or image type known to have spatial axes preceeding other axes.

imageglobals

Defaults for images and headers

error_level is the problem level (see BatteryRunners) at which an error will be raised, by the batteryrunners `log_raise` method. Thus a level of 0 will result in an error for any problem at all, and a level of 50 will mean no errors will be raised (unless someone's put some strange problem_level > 50 code in).

`logger` is the default logger (python log instance)

To set the log level (log message appears for problem of level >= log level), use e.g. `logger.level = 40`.

As for most loggers, if `logger.level == 0` then a default log level is used - use `logger.getEffectiveLevel()` to see what that default is.

Use `logger.level = 1` to see all messages.

ErrorLevel(level)	Context manager to set log error level
LoggingOutputSuppressor()	Context manager to prevent global logger from printing

ErrorLevel

class nibabel.imageglobals.**ErrorLevel**(*level*)
Bases: object

Context manager to set log error level

__init__(*level*)
Initialize self. See help(type(self)) for accurate signature.

LoggingOutputSuppressor

class nibabel.imageglobals.**LoggingOutputSuppressor**
Bases: object

Context manager to prevent global logger from printing

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

loadsave

Utilities to load and save image objects

guessed_image_type(filename)	Guess image type from file *filename*
load(filename, **kwargs)	Load file given filename, guessing at file type
read_img_data(img[, prefer])	Read data from image associated with files
save(img, filename)	Save an image to file adapting format to *filename*
which_analyze_type(binaryblock)	Is *binaryblock* from NIfTI1, NIfTI2 or Analyze header?

guessed_image_type

nibabel.loadsave.**guessed_image_type**(*filename*)
Guess image type from file *filename*

guessed_image_type deprecated.

- deprecated from version: 3.2
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 5.0

Parameters

filename [str] File name containing an image

Returns

image_class [class] Class corresponding to guessed image type

load

nibabel.loadsave.**load**(*filename*, ***kwargs*)

Load file given filename, guessing at file type

Parameters

filename [str or os.PathLike] specification of file to load

****kwargs** [keyword arguments] Keyword arguments to format-specific load

Returns

img [SpatialImage] Image of guessed type

read_img_data

nibabel.loadsave.**read_img_data**(*img*, *prefer='scaled'*)

Read data from image associated with files

read_img_data deprecated. Please use `img.dataobj.get_unscaled()` instead.

- deprecated from version: 3.2
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 5.0

If you want unscaled data, please use `img.dataobj.get_unscaled()` instead. If you want scaled data, use `img.get_fdata()` (which will cache the loaded array) or `np.array(img.dataobj)` (which won't cache the array). If you want to load the data as for a modified header, save the image with the modified header, and reload.

Parameters

img [SpatialImage] Image with valid image file in `img.file_map`. Unlike the `img.get_fdata()` method, this function returns the data read from the image file, as specified by the *current* image header and *current* image files.

prefer [str, optional] Can be 'scaled' - in which case we return the data with the scaling suggested by the format, or 'unscaled', in which case we return, if we can, the raw data from the image file, without the scaling applied.

Returns

arr [ndarray] array as read from file, given parameters in header

Notes

Summary: please use the `get_data` method of *img* instead of this function unless you are sure what you are doing.

In general, you will probably prefer `prefer='scaled'`, because this gives the data as the image format expects to return it.

Use *prefer* == 'unscaled' with care; the modified Analyze-type formats such as SPM formats, and nifti1, specify that the image data array is given by the raw data on disk, multiplied by a scalefactor and maybe with the addition of a constant. This function, with `unscaled` returns the data on the disk, without these format-specific scalings applied. Please use this funciton only if you absolutely need the unscaled data, and the magnitude of the data, as given by the scalefactor, is not relevant to your application. The Analyze-type formats have a single scalefactor +/- offset per image on disk. If you do not care about the absolute values, and will be removing the mean from

the data, then the unscaled values will have preserved intensity ratios compared to the mean-centered scaled data. However, this is not necessarily true of other formats with more complicated scaling - such as MINC.

save

nibabel.loadsave.**save**(*img*, *filename*)

Save an image to file adapting format to *filename*

Parameters

img [SpatialImage] image to save

filename [str or os.PathLike] filename (often implying filenames) to which to save *img*.

Returns

None

which_analyze_type

nibabel.loadsave.**which_analyze_type**(*binaryblock*)

Is *binaryblock* from NIfTI1, NIfTI2 or Analyze header?

which_analyze_type deprecated.

- deprecated from version: 3.2
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

Parameters

binaryblock [bytes] The *binaryblock* is 348 bytes that might be NIfTI1, NIfTI2, Analyze, or None of the the above.

Returns

hdr_type [str]

- a nifti1 header (pair or single) -> return 'nifti1'
- a nifti2 header (pair or single) -> return 'nifti2'
- an Analyze header -> return 'analyze'
- None of the above -> return None

Notes

Algorithm:

- read in the first 4 bytes from the file as 32-bit int sizeof_hdr
- if sizeof_hdr is 540 or byteswapped 540 -> assume nifti2
- Check for 'ni1', 'n+1' magic -> assume nifti1
- if sizeof_hdr is 348 or byteswapped 348 assume Analyze
- Return None

orientations

Utilities for calculating and applying affine orientations

OrientationError	

aff2axcodes(aff[, labels, tol])	axis direction codes for affine *aff*
apply_orientation(arr, ornt)	Apply transformations implied by *ornt* to the first n axes of the array *arr*
axcodes2ornt(axcodes[, labels])	Convert axis codes *axcodes* to an orientation
flip_axis(arr[, axis])	Flip contents of *axis* in array *arr*
inv_ornt_aff(ornt, shape)	Affine transform reversing transforms implied in *ornt*
io_orientation(affine[, tol])	Orientation of input axes in terms of output axes for *affine*
orientation_affine(ornt, shape)	orientation_affine deprecated.
ornt2axcodes(ornt[, labels])	Convert orientation *ornt* to labels for axis directions
ornt_transform(start_ornt, end_ornt)	Return the orientation that transforms from *start_ornt* to *end_ornt*.

OrientationError

class nibabel.orientations.**OrientationError**
Bases: Exception

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

aff2axcodes

nibabel.orientations.**aff2axcodes**(*aff*, *labels=None*, *tol=None*)
axis direction codes for affine *aff*

Parameters

aff [(N,M) array-like] affine transformation matrix

labels [optional, None or sequence of (2,) sequences] Labels for negative and positive ends of output axes of *aff*. See docstring for `ornt2axcodes` for more detail

tol [None or float] Tolerance for SVD of affine - see `io_orientation` for more detail.

Returns

axcodes [(N,) tuple] labels for positive end of voxel axes. Dropped axes get a label of None.

Examples

```
>>> aff = [[0,1,0,10],[-1,0,0,20],[0,0,1,30],[0,0,0,1]]
>>> aff2axcodes(aff, (('L','R'),('B','F'),('D','U')))
('B', 'R', 'U')
```

apply_orientation

nibabel.orientations.**apply_orientation**(*arr*, *ornt*)

Apply transformations implied by *ornt* to the first n axes of the array *arr*

Parameters

arr [array-like of data with ndim >= n]

ornt [(n,2) orientation array] orientation transform. `ornt[N,1]` is flip of axis N of the array implied by `shape`, where 1 means no flip and -1 means flip. For example, if ``N==0 and ornt[0,1] == -1, and there's an array `arr` of shape *shape*, the flip would correspond to the effect of `np.flipud(arr)`. `ornt[:,0]` is the transpose that needs to be done to the implied array, as in `arr.transpose(ornt[:,0])`

Returns

t_arr [ndarray] data array *arr* transformed according to ornt

axcodes2ornt

nibabel.orientations.**axcodes2ornt**(*axcodes*, *labels=None*)

Convert axis codes *axcodes* to an orientation

Parameters

axcodes [(N,) tuple] axis codes - see ornt2axcodes docstring

labels [optional, None or sequence of (2,) sequences] (2,) sequences are labels for (beginning, end) of output axis. That is, if the first element in *axcodes* is `front`, and the second (2,) sequence in *labels* is ('back', 'front') then the first row of *ornt* will be `[1, 1]`. If None, equivalent to `(('L','R'),('P','A'),('I','S'))` - that is - RAS axes.

Returns

ornt [(N,2) array-like] orientation array - see io_orientation docstring

Examples

```
>>> axcodes2ornt(('F', 'L', 'U'), (('L','R'),('B','F'),('D','U')))
array([[ 1.,  1.],
       [ 0., -1.],
       [ 2.,  1.]])
```

flip_axis

nibabel.orientations.**flip_axis**(*arr*, *axis=0*)

Flip contents of *axis* in array *arr*

flip_axis is deprecated. Please use numpy.flip instead.

- deprecated from version: 3.2
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 5.0

Equivalent to `np.flip(arr, axis)`.

Parameters

arr [array-like]

axis [int, optional] axis to flip. Default *axis* == 0

Returns

farr [array] Array with axis *axis* flipped

inv_ornt_aff

nibabel.orientations.**inv_ornt_aff**(*ornt*, *shape*)

Affine transform reversing transforms implied in *ornt*

Imagine you have an array `arr` of shape *shape*, and you apply the transforms implied by *ornt* (more below), to get `tarr`. `tarr` may have a different shape `shape_prime`. This routine returns the affine that will take a array coordinate for `tarr` and give you the corresponding array coordinate in `arr`.

Parameters

ornt [(p, 2) ndarray] orientation transform. `ornt[P, 1]` is flip of axis N of the array implied by `shape`, where 1 means no flip and -1 means flip. For example, if ``P==0 and `ornt[0, 1] == -1`, and there's an array `arr` of shape *shape*, the flip would correspond to the effect of `np.flipud(arr)`. `ornt[:,0]` gives us the (reverse of the) transpose that has been done to `arr`. If there are any NaNs in *ornt*, we raise an `OrientationError` (see notes)

shape [length p sequence] shape of array you may transform with *ornt*

Returns

transform_affine [(p + 1, p + 1) ndarray] An array `arr` (shape *shape*) might be transformed according to *ornt*, resulting in a transformed array `tarr`. *transformed_affine* is the transform that takes you from array coordinates in `tarr` to array coordinates in `arr`.

Notes

If a row in *ornt* contains NaN, this means that the input row does not influence the output space, and is thus effectively dropped from the output space. In that case one `tarr` coordinate maps to many `arr` coordinates, we can't invert the transform, and we raise an error

io_orientation

nibabel.orientations.**io_orientation**(*affine*, *tol=None*)

Orientation of input axes in terms of output axes for *affine*

Valid for an affine transformation from `p` dimensions to `q` dimensions (`affine.shape == (q + 1, p + 1)`).

The calculated orientations can be used to transform associated arrays to best match the output orientations. If `p` > `q`, then some of the output axes should be considered dropped in this orientation.

Parameters

affine [(q+1, p+1) ndarray-like] Transformation affine from `p` inputs to `q` outputs. Usually this will be a shape (4,4) matrix, transforming 3 inputs to 3 outputs, but the code also handles the more general case

tol [{None, float}, optional] threshold below which SVD values of the affine are considered zero. If *tol* is None, and `S` is an array with singular values for *affine*, and `eps` is the epsilon value for datatype of `S`, then *tol* set to `S.max() * max((q, p)) * eps`

Returns

orientations [(p, 2) ndarray] one row per input axis, where the first value in each row is the closest corresponding output axis. The second value in each row is 1 if the input axis is in the same direction as the corresponding output axis and -1 if it is in the opposite direction. If a row is [np.nan, np.nan], which can happen when p > q, then this row should be considered dropped.

orientation_affine

nibabel.orientations.**orientation_affine**(*ornt*, *shape*)

orientation_affine deprecated. Please use inv_ornt_aff instead.

- deprecated from version: 3.0
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

ornt2axcodes

nibabel.orientations.**ornt2axcodes**(*ornt*, *labels=None*)

Convert orientation *ornt* to labels for axis directions

Parameters

ornt [(N,2) array-like] orientation array - see io_orientation docstring

labels [optional, None or sequence of (2,) sequences] (2,) sequences are labels for (beginning, end) of output axis. That is, if the first row in *ornt* is `[1, 1]`, and the second (2,) sequence in *labels* is ('back', 'front') then the first returned axis code will be `'front'`. If the first row in *ornt* had been `[1, -1]` then the first returned value would have been `'back'`. If None, equivalent to `(('L','R'),('P','A'),('I','S'))` - that is - RAS axes.

Returns

axcodes [(N,) tuple] labels for positive end of voxel axes. Dropped axes get a label of None.

Examples

```
>>> ornt2axcodes([[1, 1],[0,-1],[2,1]], (('L','R'),('B','F'),('D','U')))
('F', 'L', 'U')
```

ornt_transform

nibabel.orientations.**ornt_transform**(*start_ornt*, *end_ornt*)

Return the orientation that transforms from *start_ornt* to *end_ornt*.

Parameters

start_ornt [(n,2) orientation array] Initial orientation.

end_ornt [(n,2) orientation array] Final orientation.

Returns

orientations [(p, 2) ndarray] The orientation that will transform the *start_ornt* to the *end_ornt*.

quaternions

Functions to operate on, or return, quaternions.

The module also includes functions for the closely related angle, axis pair as a specification for rotation.

Quaternions here consist of 4 values `w, x, y, z`, where `w` is the real (scalar) part, and `x, y, z` are the complex (vector) part.

Note - rotation matrices here apply to column vectors, that is, they are applied on the left of the vector. For example:

```
>>> import numpy as np
>>> q = [0, 1, 0, 0] # 180 degree rotation around axis 0
>>> M = quat2mat(q) # from this module
>>> vec = np.array([1, 2, 3]).reshape((3,1)) # column vector
>>> tvec = np.dot(M, vec)
```

angle_axis2mat(theta, vector[, is_normalized])	Rotation matrix of angle *theta* around *vector*
angle_axis2quat(theta, vector[, is_normalized])	Quaternion for rotation of angle *theta* around *vector*
conjugate(q)	Conjugate of quaternion
eye()	Return identity quaternion
fillpositive(xyz[, w2_thresh])	Compute unit quaternion from last 3 values
inverse(q)	Return multiplicative inverse of quaternion *q*
isunit(q)	Return True is this is very nearly a unit quaternion
mat2quat(M)	Calculate quaternion corresponding to given rotation matrix
mult(q1, q2)	Multiply two quaternions
nearly_equivalent(q1, q2[, rtol, atol])	Returns True if *q1* and *q2* give near equivalent transforms
norm(q)	Return norm of quaternion
quat2angle_axis(quat[, identity_thresh])	Convert quaternion to rotation of angle around axis
quat2mat(q)	Calculate rotation matrix corresponding to quaternion
rotate_vector(v, q)	Apply transformation in quaternion *q* to vector *v*

angle_axis2mat

nibabel.quaternions.**angle_axis2mat**(*theta*, *vector*, *is_normalized=False*)

Rotation matrix of angle *theta* around *vector*

Parameters

theta [scalar] angle of rotation

vector [3 element sequence] vector specifying axis for rotation.

is_normalized [bool, optional] True if vector is already normalized (has norm of 1). Default False

Returns

mat [array shape (3,3)] rotation matrix for specified rotation

Notes

From: https://en.wikipedia.org/wiki/Rotation_matrix#Axis_and_angle

angle_axis2quat

nibabel.quaternions.**angle_axis2quat**(*theta*, *vector*, *is_normalized=False*)

Quaternion for rotation of angle *theta* around *vector*

Parameters

theta [scalar] angle of rotation

vector [3 element sequence] vector specifying axis for rotation.

is_normalized [bool, optional] True if vector is already normalized (has norm of 1). Default False

Returns

quat [4 element sequence of symbols] quaternion giving specified rotation

Notes

Formula from http://mathworld.wolfram.com/EulerParameters.html

Examples

```
>>> q = angle_axis2quat(np.pi, [1, 0, 0])
>>> np.allclose(q, [0, 1, 0,  0])
True
```

conjugate

nibabel.quaternions.**conjugate**(*q*)
Conjugate of quaternion

Parameters

q [4 element sequence] w, i, j, k of quaternion

Returns

conjq [array shape (4,)] w, i, j, k of conjugate of *q*

eye

nibabel.quaternions.**eye**()
Return identity quaternion

fillpositive

nibabel.quaternions.**fillpositive**(*xyz*, *w2_thresh=None*)
Compute unit quaternion from last 3 values

Parameters

xyz [iterable] iterable containing 3 values, corresponding to quaternion x, y, z

w2_thresh [None or float, optional] threshold to determine if w squared is really negative. If None (default) then w2_thresh set equal to `-np.finfo(xyz.dtype).eps`, if possible, otherwise `-np.finfo(np.float64).eps`

Returns

wxyz [array shape (4,)] Full 4 values of quaternion

Notes

If w, x, y, z are the values in the full quaternion, assumes w is positive.

Gives error if w*w is estimated to be negative

w = 0 corresponds to a 180 degree rotation

The unit quaternion specifies that np.dot(wxyz, wxyz) == 1.

If w is positive (assumed here), w is given by:

w = np.sqrt(1.0-(x*x+y*y+z*z))

w2 = 1.0-(x*x+y*y+z*z) can be near zero, which will lead to numerical instability in sqrt. Here we use the system maximum float type to reduce numerical instability

Examples

```
>>> import numpy as np
>>> wxyz = fillpositive([0,0,0])
>>> np.all(wxyz == [1, 0, 0, 0])
True
>>> wxyz = fillpositive([1,0,0]) # Corner case; w is 0
>>> np.all(wxyz == [0, 1, 0, 0])
True
>>> np.dot(wxyz, wxyz)
1.0
```

inverse

nibabel.quaternions.**inverse**(*q*)

Return multiplicative inverse of quaternion *q*

Parameters

q [4 element sequence] w, i, j, k of quaternion

Returns

invq [array shape (4,)] w, i, j, k of quaternion inverse

isunit

nibabel.quaternions.**isunit**(*q*)

Return True is this is very nearly a unit quaternion

mat2quat

nibabel.quaternions.**mat2quat**(*M*)

Calculate quaternion corresponding to given rotation matrix

Parameters

M [array-like] 3x3 rotation matrix

Returns

q [(4,) array] closest quaternion to input matrix, having positive q[0]

Notes

Method claimed to be robust to numerical errors in M

Constructs quaternion by calculating maximum eigenvector for matrix K (constructed from input *M*). Although this is not tested, a maximum eigenvalue of 1 corresponds to a valid rotation.

A quaternion q*-1 corresponds to the same rotation as q; thus the sign of the reconstructed quaternion is arbitrary, and we return quaternions with positive w (q[0]).

References

- https://en.wikipedia.org/wiki/Rotation_matrix#Quaternion
- Bar-Itzhack, Itzhack Y. (2000), "New method for extracting the quaternion from a rotation matrix", AIAA Journal of Guidance, Control and Dynamics 23(6):1085-1087 (Engineering Note), ISSN 0731-5090

Examples

```
>>> import numpy as np
>>> q = mat2quat(np.eye(3)) # Identity rotation
>>> np.allclose(q, [1, 0, 0, 0])
True
>>> q = mat2quat(np.diag([1, -1, -1]))
>>> np.allclose(q, [0, 1, 0, 0]) # 180 degree rotn around axis 0
True
```

mult

nibabel.quaternions.**mult**(*q1*, *q2*)

Multiply two quaternions

Parameters

q1 [4 element sequence]

q2 [4 element sequence]

Returns

q12 [shape (4,) array]

Notes

See : https://en.wikipedia.org/wiki/Quaternions#Hamilton_product

nearly_equivalent

nibabel.quaternions.**nearly_equivalent**(*q1*, *q2*, *rtol=1e-05*, *atol=1e-08*)

Returns True if *q1* and *q2* give near equivalent transforms

q1 may be nearly numerically equal to *q2*, or nearly equal to *q2* * -1 (because a quaternion multiplied by -1 gives the same transform).

Parameters

q1 [4 element sequence] w, x, y, z of first quaternion

q2 [4 element sequence] w, x, y, z of second quaternion

Returns

equiv [bool] True if *q1* and *q2* are nearly equivalent, False otherwise

Examples

```
>>> q1 = [1, 0, 0, 0]
>>> nearly_equivalent(q1, [0, 1, 0, 0])
False
>>> nearly_equivalent(q1, [1, 0, 0, 0])
True
>>> nearly_equivalent(q1, [-1, 0, 0, 0])
True
```

norm

nibabel.quaternions.**norm**(*q*)

Return norm of quaternion

Parameters

q [4 element sequence] w, i, j, k of quaternion

Returns

n [scalar] quaternion norm

quat2angle_axis

nibabel.quaternions.**quat2angle_axis**(*quat*, *identity_thresh=None*)

Convert quaternion to rotation of angle around axis

Parameters

quat [4 element sequence] w, x, y, z forming quaternion

identity_thresh [None or scalar, optional] threshold below which the norm of the vector part of the quaternion (x, y, z) is deemed to be 0, leading to the identity rotation. None (the default) leads to a threshold estimated based on the precision of the input.

Returns

theta [scalar] angle of rotation

vector [array shape (3,)] axis around which rotation occurs

Notes

A quaternion for which x, y, z are all equal to 0, is an identity rotation. In this case we return a 0 angle and an arbitrary vector, here [1, 0, 0]

Examples

```
>>> theta, vec = quat2angle_axis([0, 1, 0, 0])
>>> np.allclose(theta, np.pi)
True
>>> vec
array([1., 0., 0.])
```

If this is an identity rotation, we return a zero angle and an arbitrary vector

```
>>> quat2angle_axis([1, 0, 0, 0])
(0.0, array([1., 0., 0.]))
```

quat2mat

nibabel.quaternions.**quat2mat**(*q*)

Calculate rotation matrix corresponding to quaternion

Parameters

q [4 element array-like]

Returns

M [(3,3) array] Rotation matrix corresponding to input quaternion *q*

Notes

Rotation matrix applies to column vectors, and is applied to the left of coordinate vectors. The algorithm here allows non-unit quaternions.

References

Algorithm from https://en.wikipedia.org/wiki/Rotation_matrix#Quaternion

Examples

```
>>> import numpy as np
>>> M = quat2mat([1, 0, 0, 0]) # Identity quaternion
>>> np.allclose(M, np.eye(3))
True
>>> M = quat2mat([0, 1, 0, 0]) # 180 degree rotn around axis 0
>>> np.allclose(M, np.diag([1, -1, -1]))
True
```

rotate_vector

nibabel.quaternions.**rotate_vector**(*v*, *q*)

Apply transformation in quaternion *q* to vector *v*

Parameters

v [3 element sequence] 3 dimensional vector

q [4 element sequence] w, i, j, k of quaternion

Returns

vdash [array shape (3,)] *v* rotated by quaternion *q*

Notes

See: https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation#Describing_rotations_with_quaternions

spatialimages

A simple spatial image class

The image class maintains the association between a 3D (or greater) array, and an affine transform that maps voxel coordinates to some world space. It also has a `header` - some standard set of meta-data that is specific to the image format, and `extra` - a dictionary container for any other metadata.

It has attributes:

- extra

methods:

- .get_fdata()
- .get_data() (deprecated, use get_fdata() instead)
- .get_affine() (deprecated, use affine property instead)
- .get_header() (deprecated, use header property instead)
- .to_filename(fname) - writes data to filename(s) derived from `fname`, where the derivation may differ between formats.
- to_file_map() - save image to files with which the image is already associated.

properties:

- shape
- affine
- header
- dataobj

classmethods:

- from_filename(fname) - make instance by loading from filename
- from_file_map(fmap) - make instance from file map
- instance_to_filename(img, fname) - save `img` instance to filename `fname`.

You cannot slice an image, and trying to slice an image generates an informative TypeError.

There are several ways of writing data.

There is the usual way, which is the default:

```
img.to_filename(fname)
```

and that is, to take the data encapsulated by the image and cast it to the datatype the header expects, setting any available header scaling into the header to help the data match.

You can load the data into an image from file with:

```
img.from_filename(fname)
```

The image stores its associated files in its `file_map` attribute. In order to just save an image, for which you know there is an associated filename, or other storage, you can do:

```
img.to_file_map()
```

You can get the data out again with:

```
img.get_fdata()
```

Less commonly, for some image types that support it, you might want to fetch out the unscaled array via the object containing the data:

```
unscaled_data = img.dataoobj.get_unscaled()
```

Analyze-type images (including nifti) support this, but others may not (MINC, for example).

Sometimes you might to avoid any loss of precision by making the data type the same as the input:

```
hdr = img.header
hdr.set_data_dtype(data.dtype)
img.to_filename(fname)
```

Files interface

The image has an attribute `file_map`. This is a mapping, that has keys corresponding to the file types that an image needs for storage. For example, the Analyze data format needs an `image` and a `header` file type for storage:

```
>>> import nibabel as nib
>>> data = np.arange(24, dtype='f4').reshape((2,3,4))
>>> img = nib.AnalyzeImage(data, np.eye(4))
>>> sorted(img.file_map)
['header', 'image']
```

The values of `file_map` are not in fact files but objects with attributes `filename`, `fileobj` and `pos`.

The reason for this interface, is that the contents of files has to contain enough information so that an existing image instance can save itself back to the files pointed to in `file_map`. When a file holder holds active file-like objects, then these may be affected by the initial file read; in this case, the contains file-like objects need to carry the position at which a write (with `to_file_map`) should place the data. The `file_map` contents should therefore be such, that this will work:

```
>>> # write an image to files
>>> from io import BytesIO
>>> import nibabel as nib
>>> file_map = nib.AnalyzeImage.make_file_map()
>>> file_map['image'].fileobj = BytesIO()
>>> file_map['header'].fileobj = BytesIO()
>>> img = nib.AnalyzeImage(data, np.eye(4))
>>> img.file_map = file_map
>>> img.to_file_map()
>>> # read it back again from the written files
>>> img2 = nib.AnalyzeImage.from_file_map(file_map)
>>> np.all(img2.get_fdata(dtype=np.float32) == data)
True
>>> # write, read it again
>>> img2.to_file_map()
>>> img3 = nib.AnalyzeImage.from_file_map(file_map)
>>> np.all(img3.get_fdata(dtype=np.float32) == data)
True
```

Header(*args, **kwargs)	Alias for SpatialHeader; kept for backwards compatibility.
HeaderDataError	Class to indicate error in getting or setting header data
HeaderTypeError	Class to indicate error in parameters into header functions
ImageDataError	
SpatialFirstSlicer(img)	Slicing interface that returns a new image with an updated affine
SpatialHeader([data_dtype, shape, zooms])	Template class to implement header protocol
SpatialImage(dataobj, affine[, header, ...])	Template class for volumetric (3D/4D) images
supported_np_types(obj)	Numpy data types that instance *obj* supports

Header

class nibabel.spatialimages.**Header**(**args*, ***kwargs*)

Bases: *nibabel.spatialimages.SpatialHeader*

Alias for SpatialHeader; kept for backwards compatibility.

Header class is deprecated. Please use SpatialHeader instead.instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

__init__(**args*, ***kwargs*)

Header class is deprecated. Please use SpatialHeader instead.instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

HeaderDataError

class nibabel.spatialimages.**HeaderDataError**
Bases: Exception

Class to indicate error in getting or setting header data

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

HeaderTypeError

class nibabel.spatialimages.**HeaderTypeError**
Bases: Exception

Class to indicate error in parameters into header functions

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

ImageDataError

class nibabel.spatialimages.**ImageDataError**
Bases: Exception

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

SpatialFirstSlicer

class nibabel.spatialimages.**SpatialFirstSlicer**(*img*)
Bases: object

Slicing interface that returns a new image with an updated affine

Checks that an image's first three axes are spatial

__init__(*img*)
Initialize self. See help(type(self)) for accurate signature.

check_slicing(*slicer*, *return_spatial=False*)
Canonicalize slicers and check for scalar indices in spatial dims

Parameters

slicer [object] something that can be used to slice an array as in `arr[sliceobj]`

return_spatial [bool] return only slices along spatial dimensions (x, y, z)

Returns

slicer [object] Validated slicer object that will slice image's *dataobj* without collapsing spatial dimensions

slice_affine(*slicer*)
Retrieve affine for current image, if sliced by a given index

Applies scaling if down-sampling is applied, and adjusts the intercept to account for any cropping.

Parameters

slicer [object] something that can be used to slice an array as in `arr[sliceobj]`

Returns

affine [(4,4) ndarray] Affine with updated scale and intercept

SpatialHeader

class nibabel.spatialimages.**SpatialHeader**(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)

Bases: *nibabel.filebasedimages.FileBasedHeader*

Template class to implement header protocol

__init__(*data_dtype=<class 'numpy.float32'>*, *shape=(0,)*, *zooms=None*)
Initialize self. See help(type(self)) for accurate signature.

copy()
Copy object to independent representation

The copy should not be affected by any changes to the original object.

data_from_fileobj(*fileobj*)
Read binary image data from *fileobj*

data_layout = 'F'

data_to_fileobj(*data*, *fileobj*, *rescale=True*)
Write array data *data* as binary to *fileobj*

Parameters

data [array-like] data to write

fileobj [file-like object] file-like object implementing 'write'

rescale [{True, False}, optional] Whether to try and rescale data to match output dtype specified by header. For this minimal header, *rescale* has no effect

default_x_flip = True

classmethod from_fileobj(*fileobj*)

classmethod from_header(*header=None*)

get_base_affine()

get_best_affine()

get_data_dtype()

get_data_shape()

get_zooms()

set_data_dtype(*dtype*)

set_data_shape(*shape*)

set_zooms(*zooms*)

write_to(*fileobj*)

SpatialImage

class nibabel.spatialimages.**SpatialImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.dataobj_images.DataobjImage*

Template class for volumetric (3D/4D) images

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

ImageSlicer

alias of *nibabel.spatialimages.SpatialFirstSlicer*

property affine

as_reoriented(*ornt*)

Apply an orientation change and return a new image

If ornt is identity transform, return the original image, unchanged

Parameters

ornt [(n,2) orientation array] orientation transform. `ornt[N,1]` is flip of axis N of the array implied by `shape`, where 1 means no flip

and -1 means flip. For example, if ``N==0 and ornt[0,1] == -1, and there's an array arr of shape *shape*, the flip would correspond to the effect of np.flipud(arr). ornt[:,0] is the transpose that needs to be done to the implied array, as in arr.transpose(ornt[:,0])

Notes

Subclasses should override this if they have additional requirements when re-orienting an image.

classmethod from_image(*img*)
Class method to create new instance of own class from *img*

Parameters

img [spatialimage instance] In fact, an object with the API of spatialimage - specifically dataobj, affine, header and extra.

Returns

cimg [spatialimage instance] Image, of our own class

get_affine()
Get affine from image

get_affine method is deprecated. Please use the img.affine property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

get_data_dtype()

header_class
alias of *nibabel.spatialimages.SpatialHeader*

orthoview()
Plot the image using OrthoSlicer3D

Returns

viewer [instance of OrthoSlicer3D] The viewer.

Notes

This requires matplotlib. If a non-interactive backend is used, consider using viewer.show() (equivalently plt.show()) to show the figure.

set_data_dtype(*dtype*)

property slicer
Slicer object that returns cropped and subsampled images

The image is resliced in the current orientation; no rotation or resampling is performed, and no attempt is made to filter the image to avoid aliasing.

The affine matrix is updated with the new intercept (and scales, if down-sampling is used), so that all values are found at the same RAS locations.

Slicing may include non-spatial dimensions. However, this method does not currently adjust the repetition time in the image header.

update_header()
Harmonize header with image data and affine

```
>>> data = np.zeros((2,3,4))
>>> affine = np.diag([1.0,2.0,3.0,1.0])
>>> img = SpatialImage(data, affine)
>>> img.shape == (2, 3, 4)
True
>>> img.update_header()
>>> img.header.get_data_shape() == (2, 3, 4)
True
>>> img.header.get_zooms()
(1.0, 2.0, 3.0)
```

supported_np_types

nibabel.spatialimages.**supported_np_types**(*obj*)
Numpy data types that instance *obj* supports

Parameters

obj [object] Object implementing *get_data_dtype* and *set_data_dtype*. The object should raise `HeaderDataError` for setting unsupported dtypes. The object will likely be a header or a *SpatialImage*

Returns

np_types [set] set of numpy types that *obj* supports

volumeutils

Utility functions for analyze-like formats

BinOpener(*args, **kwargs)	Class to accept, maybe open, and context-manage file-likes / filenames
DtypeMapper()	Specialized mapper for numpy dtypes
Recoder(codes[, fields, map_maker])	class to return canonical code(s) from code or aliases
allopen(fileish, *args, **kwargs)	Compatibility wrapper for old `allopen` function
apply_read_scaling(arr[, slope, inter])	Apply scaling in *slope* and *inter* to array *arr*
array_from_file(shape, in_dtype, infile[, ...])	Get array from file with specified shape, dtype and file offset
array_to_file(data, fileobj[, out_dtype, ...])	Helper function for writing arrays to file objects
best_write_scale_ftype(arr[, slope, inter, ...])	Smallest float type to contain range of `arr` after scaling
better_float_of(first, second[, default])	Return more capable float type of *first* and *second*
finite_range(arr[, check_nan])	Get range (min, max) or range and flag (min, max, has_nan) from *arr*
fname_ext_ul_case(fname)	*fname* with ext changed to upper / lower case if file exists
int_scinter_ftype(ifmt[, slope, inter, default])	float type containing int type *ifmt* * *slope* + *inter*
make_dt_codes(codes_seqs)	Create full dt codes Recoder instance from datatype codes
pretty_mapping(mapping[, getterfunc])	Make pretty string from mapping
rec2dict(rec)	Convert recarray to dictionary
seek_tell(fileobj, offset[, write0])	Seek in *fileobj* or check we're in the right place already
shape_zoom_affine(shape, zooms[, x_flip])	Get affine implied by given shape and zooms

continues on next page

Table 53 – continued from previous page

working_type(in_type[, slope, inter])	Return array type from applying *slope*, *inter* to array of *in_type*
write_zeros(fileobj, count[, block_size])	Write *count* zero bytes to *fileobj*

BinOpener

class nibabel.volumeutils.**BinOpener**(**args*, ***kwargs*)

Bases: *nibabel.openers.Opener*

Class to accept, maybe open, and context-manage file-likes / filenames

Provides context manager to close files that the constructor opened for you.

Parameters

fileish [str or file-like] if str, then open with suitable opening method. If file-like, accept as is

***args** [positional arguments] passed to opening method when *fileish* is str. `mode`, if not specified, is *rb*. `compresslevel`, if relevant, and not specified, is set from class variable `default_compresslevel`. `keep_open`, if relevant, and not specified, is `False`.

****kwargs** [keyword arguments] passed to opening method when *fileish* is str. Change of defaults as for *args

BinOpener class deprecated. Please use Opener class instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

__init__(**args*, ***kwargs*)

BinOpener class deprecated. Please use Opener class instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

DtypeMapper

class nibabel.volumeutils.**DtypeMapper**

Bases: `object`

Specialized mapper for numpy dtypes

We pass this mapper into the Recoder class to deal with numpy dtype hashing.

The hashing problem is that dtypes that compare equal may not have the same hash. This is true for numpys up to the current at time of writing (1.6.0). For numpy 1.2.1 at least, even dtypes that look exactly the same in terms of fields don't always have the same hash. This makes dtypes difficult to use as keys in a dictionary.

This class wraps a dictionary in order to implement a __getitem__ to deal with dtype hashing. If the key doesn't appear to be in the mapping, and it is a dtype, we compare (using ==) all known dtype keys to the input key, and return any matching values for the matching key.

__init__()

Initialize self. See help(type(self)) for accurate signature.

keys()

values()

Recoder

class nibabel.volumeutils.**Recoder**(*codes*, *fields=('code',)*, *map_maker=<class 'collections.OrderedDict'>*)

Bases: object

class to return canonical code(s) from code or aliases

The concept is a lot easier to read in the implementation and tests than it is to explain, so...

```
>>> # If you have some codes, and several aliases, like this:
>>> code1 = 1; aliases1=['one', 'first']
>>> code2 = 2; aliases2=['two', 'second']
>>> # You might want to do this:
>>> codes = [[code1]+aliases1,[code2]+aliases2]
>>> recodes = Recoder(codes)
>>> recodes.code['one']
1
>>> recodes.code['second']
2
>>> recodes.code[2]
2
>>> # Or maybe you have a code, a label and some aliases
>>> codes=((1,'label1','one', 'first'),(2,'label2','two'))
>>> # you might want to get back the code or the label
>>> recodes = Recoder(codes, fields=('code','label'))
>>> recodes.code['first']
1
>>> recodes.code['label1']
1
>>> recodes.label[2]
'label2'
>>> # For convenience, you can get the first entered name by
>>> # indexing the object directly
>>> recodes[2]
2
```

Create recoder object

`codes` give a sequence of code, alias sequences `fields` are names by which the entries in these sequences can be accessed.

By default `fields` gives the first column the name "code". The first column is the vector of first entries in each of the sequences found in `codes`. Thence you can get the equivalent first column value with ob.code[value], where value can be a first column value, or a value in any of the other columns in that sequence.

You can give other columns names too, and access them in the same way - see the examples in the class docstring.

Parameters

codes [sequence of sequences] Each sequence defines values (codes) that are equivalent

fields [{('code',) string sequence}, optional] names by which elements in sequences can be accessed

map_maker: callable, optional constructor for dict-like objects used to store key value pairs. Default is `dict`. `map_maker()` generates an empty mapping. The mapping need only implement `__getitem__`, `__setitem__`, `keys`, `values`.

__init__(*codes*, *fields=('code',)*, *map_maker=<class 'collections.OrderedDict'>*)

Create recoder object

`codes` give a sequence of code, alias sequences `fields` are names by which the entries in these sequences can be accessed.

By default `fields` gives the first column the name "code". The first column is the vector of first entries in each of the sequences found in `codes`. Thence you can get the equivalent first column value with ob.code[value], where value can be a first column value, or a value in any of the other columns in that sequence.

You can give other columns names too, and access them in the same way - see the examples in the class docstring.

Parameters

codes [sequence of sequences] Each sequence defines values (codes) that are equivalent

fields [{('code',) string sequence}, optional] names by which elements in sequences can be accessed

map_maker: callable, optional constructor for dict-like objects used to store key value pairs. Default is `dict`. `map_maker()` generates an empty mapping. The mapping need only implement `__getitem__`, `__setitem__`, `keys`, `values`.

add_codes(*code_syn_seqs*)

Add codes to object

Parameters

code_syn_seqs [sequence] sequence of sequences, where each sequence `S = code_syn_seqs[n]` for n in 0..len(code_syn_seqs), is a sequence giving values in the same order as `self.fields`. Each S should be at least of the same length as `self.fields`. After this call, if `self.fields == ['field1', 'field2'], then ``self.field1[S[n]] == S[0]` for all n in 0..len(S) and `self.field2[S[n]] == S[1]` for all n in 0..len(S).

Examples

```
>>> code_syn_seqs = ((2, 'two'), (1, 'one'))
>>> rc = Recoder(code_syn_seqs)
>>> rc.value_set() == set((1,2))
True
>>> rc.add_codes(((3, 'three'), (1, 'first')))
>>> rc.value_set() == set((1,2,3))
True
>>> print(rc.value_set())  # set is actually ordered
OrderedSet([2, 1, 3])
```

keys()

Return all available code and alias values

Returns same value as `obj.field1.keys()` and, with the default initializing `fields` argument of fields=('code',), this will return the same as `obj.code.keys()`

```
>>> codes = ((1, 'one'), (2, 'two'), (1, 'repeat value'))
>>> k = Recoder(codes).keys()
>>> set(k) == set([1, 2, 'one', 'repeat value', 'two'])
True
```

value_set(*name=None*)

Return OrderedSet of possible returned values for column

By default, the column is the first column.

Returns same values as `set(obj.field1.values())` and, with the default initializing``fields`` argument of fields=('code',), this will return the same as `set(obj.code.values())`

Parameters

name [{None, string}] Where default of none gives result for first column

>>> codes = ((1, 'one'), (2, 'two'), (1, 'repeat value'))

>>> vs = Recoder(codes).value_set()

>>> vs == set([1, 2]) # Sets are not ordered, hence this test

True

>>> rc = Recoder(codes, fields=('code', 'label'))

>>> rc.value_set('label') == set(('one', 'two', 'repeat value'))

True

allopen

nibabel.volumeutils.**allopen**(*fileish*, **args*, ***kwargs*)

Compatibility wrapper for old `allopen` function

allopen is deprecated. Please use "Opener" class instead.

- deprecated from version: 2.0
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

Wraps creation of `Opener` instance, while picking up module global `default_compresslevel`.

Please see docstring of `Opener` for details.

apply_read_scaling

nibabel.volumeutils.**apply_read_scaling**(*arr*, *slope=None*, *inter=None*)

Apply scaling in *slope* and *inter* to array *arr*

This is for loading the array from a file (as opposed to the reverse scaling when saving an array to file)

Return data will be `arr * slope + inter`. The trick is that we have to find a good precision to use for applying the scaling. The heuristic is that the data is always upcast to the higher of the types from *arr, `slope*, *inter* if *slope* and / or *inter* are not default values. If the dtype of *arr* is an integer, then we assume the data more or less fills the integer range, and upcast to a type such that the min, max of `arr.dtype` * scale + inter, will be finite.

Parameters

arr [array-like]

slope [None or float, optional] slope value to apply to *arr* (`arr * slope + inter`). None corresponds to a value of 1.0

inter [None or float, optional] intercept value to apply to *arr* (`arr * slope + inter`). None corresponds to a value of 0.0

Returns

ret [array] array with scaling applied. Maybe upcast in order to give room for the scaling. If scaling is default (1, 0), then *ret* may be *arr* `ret is arr`.

array_from_file

`nibabel.volumeutils.`**`array_from_file`**(*shape*, *in_dtype*, *infile*, *offset=0*, *order='F'*, *mmap=True*)

Get array from file with specified shape, dtype and file offset

Parameters

shape [sequence] sequence specifying output array shape

in_dtype [numpy dtype] fully specified numpy dtype, including correct endianness

infile [file-like] open file-like object implementing at least read() and seek()

offset [int, optional] offset in bytes into *infile* to start reading array data. Default is 0

order [{'F', 'C'} string] order in which to write data. Default is 'F' (fortran order).

mmap [{True, False, 'c', 'r', 'r+'}] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r', 'r+'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If *infile* cannot be memory-mapped, ignore *mmap* value and read array from file.

Returns

arr [array-like] array like object that can be sliced, containing data

Examples

```
>>> from io import BytesIO
>>> bio = BytesIO()
>>> arr = np.arange(6).reshape(1,2,3)
>>> _ = bio.write(arr.tobytes('F'))  # outputs int
>>> arr2 = array_from_file((1,2,3), arr.dtype, bio)
>>> np.all(arr == arr2)
True
>>> bio = BytesIO()
>>> _ = bio.write(b' ' * 10)
>>> _ = bio.write(arr.tobytes('F'))
>>> arr2 = array_from_file((1,2,3), arr.dtype, bio, 10)
>>> np.all(arr == arr2)
True
```

array_to_file

`nibabel.volumeutils.`**`array_to_file`**(*data*, *fileobj*, *out_dtype=None*, *offset=0*, *intercept=0.0*, *divslope=1.0*, *mn=None*, *mx=None*, *order='F'*, *nan2zero=True*)

Helper function for writing arrays to file objects

Writes arrays as scaled by *intercept* and *divslope*, and clipped at (prescaling) *mn* minimum, and *mx* maximum.

- Clip *data* array at min *mn*, max *max* where there are not None -> `clipped` (this is *pre scale clipping*)
- Scale `clipped` with `clipped_scaled = (clipped - intercept) / divslope`

- Clip `clipped_scaled` to fit into range of *out_dtype* (*post scale clipping*) -> `clipped_scaled_clipped`
- If converting to integer *out_dtype* and *nan2zero* is True, set NaN values in `clipped_scaled_clipped` to 0
- Write `clipped_scaled_clipped_n2z` to fileobj *fileobj* starting at offset *offset* in memory layout *order*

Parameters

data [array-like] array or array-like to write.

fileobj [file-like] file-like object implementing `write` method.

out_dtype [None or dtype, optional] dtype to write array as. Data array will be coerced to this dtype before writing. If None (default) then use input data type.

offset [None or int, optional] offset into fileobj at which to start writing data. Default is 0. None means start at current file position

intercept [scalar, optional] scalar to subtract from data, before dividing by `divslope`. Default is 0.0

divslope [None or scalar, optional] scalefactor to *divide* data by before writing. Default is 1.0. If None, there is no valid data, we write zeros.

mn [scalar, optional] minimum threshold in (unscaled) data, such that all data below this value are set to this value. Default is None (no threshold). The typical use is to set -np.inf in the data to have this value (which might be the minimum non-finite value in the data).

mx [scalar, optional] maximum threshold in (unscaled) data, such that all data above this value are set to this value. Default is None (no threshold). The typical use is to set np.inf in the data to have this value (which might be the maximum non-finite value in the data).

order [{'F', 'C'}, optional] memory order to write array. Default is 'F'

nan2zero [{True, False}, optional] Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs will be represented as numpy does when casting; this depends on the underlying C library and is undefined. In practice *nan2zero* == False might be a good choice when you completely sure there will be no NaNs in the data. This value ignored for float outut types. NaNs are treated as zero *before* applying *intercept* and *divslope* - so an array `[np.nan]` with an *intercept* of 10 becomes `[-10]` after conversion to integer *out_dtype* with *nan2zero* set. That is because you will likely apply *divslope* and *intercept* in reverse order when reading the data back, returning the zero you probably expected from the input NaN.

Examples

```
>>> from io import BytesIO
>>> sio = BytesIO()
>>> data = np.arange(10, dtype=np.float64)
>>> array_to_file(data, sio, np.float64)
>>> sio.getvalue() == data.tobytes('F')
True
>>> _ = sio.truncate(0); _ = sio.seek(0)  # outputs 0
>>> array_to_file(data, sio, np.int16)
>>> sio.getvalue() == data.astype(np.int16).tobytes()
True
```

(continues on next page)

(continued from previous page)

```
>>> _ = sio.truncate(0); _ = sio.seek(0)
>>> array_to_file(data.byteswap(), sio, np.float64)
>>> sio.getvalue() == data.byteswap().tobytes('F')
True
>>> _ = sio.truncate(0); _ = sio.seek(0)
>>> array_to_file(data, sio, np.float64, order='C')
>>> sio.getvalue() == data.tobytes('C')
True
```

best_write_scale_ftype

nibabel.volumeutils.**best_write_scale_ftype**(*arr*, *slope=1.0*, *inter=0.0*, *default=<class 'numpy.float32'>*)

Smallest float type to contain range of `arr` after scaling

Scaling that will be applied to `arr` is `(arr - inter) / slope`.

Note that `slope` and `inter` get promoted to 1D arrays for this purpose to avoid the numpy scalar casting rules, which prevent scalars upcasting the array.

Parameters

arr [array-like] array that will be scaled

slope [array-like, optional] scalar such that output array will be `(arr - inter) / slope`.

inter [array-like, optional] scalar such that output array will be `(arr - inter) / slope`

default [numpy type, optional] minimum float type to return

Returns

ftype [numpy type] Best floating point type for scaling. If no floating point type prevents overflow, return the top floating point type. If the input array `arr` already contains inf values, return the greater of the input type and the default type.

Examples

```
>>> arr = np.array([0, 1, 2], dtype=np.int16)
>>> best_write_scale_ftype(arr, 1, 0) is np.float32
True
```

Specify higher default return value

```
>>> best_write_scale_ftype(arr, 1, 0, default=np.float64) is np.float64
True
```

Even large values that don't overflow don't change output

```
>>> arr = np.array([0, np.finfo(np.float32).max], dtype=np.float32)
>>> best_write_scale_ftype(arr, 1, 0) is np.float32
True
```

Scaling > 1 reduces output values, so no upcast needed

```
>>> best_write_scale_ftype(arr, np.float32(2), 0) is np.float32
True
```

Scaling < 1 increases values, so upcast may be needed (and is here)

```
>>> best_write_scale_ftype(arr, np.float32(0.5), 0) is np.float64
True
```

better_float_of

nibabel.volumeutils.**better_float_of**(*first*, *second*, *default=<class 'numpy.float32'>*)

Return more capable float type of *first* and *second*

Return *default* if neither of *first* or *second* is a float

Parameters

first [numpy type specifier] Any valid input to *np.dtype()* `

second [numpy type specifier] Any valid input to *np.dtype()* `

default [numpy type specifier, optional] Any valid input to *np.dtype()* `

Returns

better_type [numpy type] More capable of *first* or *second* if both are floats; if only one is a float return that, otherwise return *default*.

Examples

```
>>> better_float_of(np.float32, np.float64) is np.float64
True
>>> better_float_of(np.float32, 'i4') is np.float32
True
>>> better_float_of('i2', 'u4') is np.float32
True
>>> better_float_of('i2', 'u4', np.float64) is np.float64
True
```

finite_range

nibabel.volumeutils.**finite_range**(*arr*, *check_nan=False*)

Get range (min, max) or range and flag (min, max, has_nan) from *arr*

Parameters

arr [array-like]

check_nan [{False, True}, optional] Whether to return third output, a bool signaling whether there are NaN values in *arr*

Returns

mn [scalar] minimum of values in (flattened) array

mx [scalar] maximum of values in (flattened) array

has_nan [bool] Returned if *check_nan* is True. *has_nan* is True if there are one or more NaN values in *arr*

Examples

```
>>> a = np.array([[-1, 0, 1],[np.inf, np.nan, -np.inf]])
>>> finite_range(a)
(-1.0, 1.0)
>>> a = np.array([[-1, 0, 1],[np.inf, np.nan, -np.inf]])
>>> finite_range(a, check_nan=True)
(-1.0, 1.0, True)
>>> a = np.array([[np.nan],[np.nan]])
>>> finite_range(a) == (np.inf, -np.inf)
True
>>> a = np.array([[-3, 0, 1],[2,-1,4]], dtype=int)
>>> finite_range(a)
(-3, 4)
>>> a = np.array([[1, 0, 1],[2,3,4]], dtype=np.uint)
>>> finite_range(a)
(0, 4)
>>> a = a + 1j
>>> finite_range(a)
(1j, (4+1j))
>>> a = np.zeros((2,), dtype=[('f1', 'i2')])
>>> finite_range(a)
Traceback (most recent call last):
   ...
TypeError: Can only handle numeric types
```

fname_ext_ul_case

nibabel.volumeutils.**fname_ext_ul_case**(*fname*)

fname with ext changed to upper / lower case if file exists

Check for existence of *fname*. If it does exist, return unmodified. If it doesn't, check for existence of *fname* with case changed from lower to upper, or upper to lower. Return this modified *fname* if it exists. Otherwise return *fname* unmodified

Parameters

fname [str] filename.

Returns

mod_fname [str] filename, maybe with extension of opposite case

int_scinter_ftype

nibabel.volumeutils.**int_scinter_ftype**(*ifmt*, *slope=1.0*, *inter=0.0*, *default=<class 'numpy.float32'>*)

float type containing int type *ifmt* * *slope* + *inter*

Return float type that can represent the max and the min of the *ifmt* type after multiplication with *slope* and addition of *inter* with something like `np.array([imin, imax], dtype=ifmt) * slope + inter`.

Note that `slope` and `inter` get promoted to 1D arrays for this purpose to avoid the numpy scalar casting rules, which prevent scalars upcasting the array.

Parameters

ifmt [object] numpy integer type (e.g. np.int32)

slope [float, optional] slope, default 1.0

inter [float, optional] intercept, default 0.0

default_out [object, optional] numpy floating point type, default is `np.float32`

Returns

ftype [object] numpy floating point type

Notes

It is difficult to make floats overflow with just addition because the deltas are so large at the extremes of floating point. For example:

```
>>> arr = np.array([np.finfo(np.float32).max], dtype=np.float32)
>>> res = arr + np.iinfo(np.int16).max
>>> arr == res
array([ True])
```

Examples

```
>>> int_scinter_ftype(np.int8, 1.0, 0.0) == np.float32
True
>>> int_scinter_ftype(np.int8, 1e38, 0.0) == np.float64
True
```

make_dt_codes

`nibabel.volumeutils.`**`make_dt_codes`**(*codes_seqs*)

Create full dt codes Recoder instance from datatype codes

Include created numpy dtype (from numpy type) and opposite endian numpy dtype

Parameters

codes_seqs [sequence of sequences] contained sequences make be length 3 or 4, but must all be the same length. Elements are data type code, data type name, and numpy type (such as `np.float32`). The fourth element is the nifti string representation of the code (e.g. "NIFTI_TYPE_FLOAT32")

Returns

rec [`Recoder` instance] Recoder that, by default, returns `code` when indexed with any of the corresponding code, name, type, dtype, or swapped dtype. You can also index with `niistring` values if codes_seqs had sequences of length 4 instead of 3.

pretty_mapping

nibabel.volumeutils.**pretty_mapping**(*mapping*, *getterfunc=None*)

Make pretty string from mapping

Adjusts text column to print values on basis of longest key. Probably only sensible if keys are mainly strings.

You can pass in a callable that does clever things to get the values out of the mapping, given the names. By default, we just use `__getitem__`

Parameters

mapping [mapping] implementing iterator returning keys and .items()

getterfunc [None or callable] callable taking two arguments, `obj` and `key` where `obj` is the passed mapping. If None, just use `lambda obj, key: obj[key]`

Returns

str [string]

Examples

```
>>> d = {'a key': 'a value'}
>>> print(pretty_mapping(d))
a key  : a value
>>> class C(object): # to control ordering, show get_ method
...     def __iter__(self):
...         return iter(('short_field','longer_field'))
...     def __getitem__(self, key):
...         if key == 'short_field':
...             return 0
...         if key == 'longer_field':
...             return 'str'
...     def get_longer_field(self):
...         return 'method string'
>>> def getter(obj, key):
...     # Look for any 'get_<name>' methods
...     try:
...         return obj.__getattribute__('get_' + key)()
...     except AttributeError:
...         return obj[key]
>>> print(pretty_mapping(C(), getter))
short_field   : 0
longer_field  : method string
```

rec2dict

nibabel.volumeutils.**rec2dict**(*rec*)

Convert recarray to dictionary

Also converts scalar values to scalars

Parameters

rec [ndarray] structured ndarray

Returns

dct [dict] dict with key, value pairs as for *rec*

Examples

```
>>> r = np.zeros((), dtype = [('x', 'i4'), ('s', 'S10')])
>>> d = rec2dict(r)
>>> d == {'x': 0, 's': b''}
True
```

seek_tell

nibabel.volumeutils.**seek_tell**(*fileobj*, *offset*, *write0=False*)

Seek in *fileobj* or check we're in the right place already

Parameters

fileobj [file-like] object implementing `seek` and (if seek raises an IOError) `tell`

offset [int] position in file to which to seek

write0 [{False, True}, optional] If True, and standard seek fails, try to write zeros to the file to reach *offset*. This can be useful when writing bz2 files, that cannot do write seeks.

shape_zoom_affine

nibabel.volumeutils.**shape_zoom_affine**(*shape*, *zooms*, *x_flip=True*)

Get affine implied by given shape and zooms

We get the translations from the center of the image (implied by *shape*).

Parameters

shape [(N,) array-like] shape of image data. `N` is the number of dimensions

zooms [(N,) array-like] zooms (voxel sizes) of the image

x_flip [{True, False}] whether to flip the X row of the affine. Corresponds to radiological storage on disk.

Returns

aff [(4,4) array] affine giving correspondance of voxel coordinates to mm coordinates, taking the center of the image as origin

Examples

```
>>> shape = (3, 5, 7)
>>> zooms = (3, 2, 1)
>>> shape_zoom_affine((3, 5, 7), (3, 2, 1))
array([[-3.,  0.,  0.,  3.],
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
>>> shape_zoom_affine((3, 5, 7), (3, 2, 1), False)
array([[ 3.,  0.,  0., -3.],
```

(continues on next page)

(continued from previous page)

```
       [ 0.,  2.,  0., -4.],
       [ 0.,  0.,  1., -3.],
       [ 0.,  0.,  0.,  1.]])
```

working_type

nibabel.volumeutils.**working_type**(*in_type*, *slope=1.0*, *inter=0.0*)

Return array type from applying *slope*, *inter* to array of *in_type*

Numpy type that results from an array of type *in_type* being combined with *slope* and *inter*. It returns something like the dtype type of `((np.zeros((2,), dtype=in_type) - inter) / slope)`, but ignoring the actual values of *slope* and *inter*.

Note that you would not necessarily get the same type by applying slope and inter the other way round. Also, you'll see that the order in which slope and inter are applied is the opposite of the order in which they are passed.

Parameters

in_type [numpy type specifier] Numpy type of input array. Any valid input for `np.dtype()`

slope [scalar, optional] slope to apply to array. If 1.0 (default), ignore this value and its type.

inter [scalar, optional] intercept to apply to array. If 0.0 (default), ignore this value and its type.

Returns

wtype: numpy type Numpy type resulting from applying *inter* and *slope* to array of type *in_type*.

write_zeros

nibabel.volumeutils.**write_zeros**(*fileobj*, *count*, *block_size=8194*)

Write *count* zero bytes to *fileobj*

Parameters

fileobj [file-like object] with `write` method

count [int] number of bytes to write

block_size [int, optional] largest continuous block to write.

10.5.4 Float / integer conversion

arraywriters	Array writer objects
casting	Utilties for casting numpy values in various ways

arraywriters

Array writer objects

Array writers have init signature:

```
def __init__(self, array, out_dtype=None)
```

and methods

- scaling_needed() - returns True if array requires scaling for write
- finite_range() - returns min, max of self.array
- to_fileobj(fileobj, offset=None, order='F')

They must have attributes / properties of:

- array
- out_dtype
- has_nan

They may have attributes:

- slope
- inter

They are designed to write arrays to a fileobj with reasonable memory efficiency.

Array writers may be able to scale the array or apply an intercept, or do something else to make sense of conversions between float and int, or between larger ints and smaller.

ArrayWriter(array[, out_dtype])	Initialize array writer
ScalingError	
SlopeArrayWriter(array[, out_dtype, ...])	ArrayWriter that can use scalefactor for writing arrays
SlopeInterArrayWriter(array[, out_dtype, ...])	Array writer that can use slope and intercept to scale array
WriterError	
get_slope_inter(writer)	Return slope, intercept from array writer object
make_array_writer(data, out_type[, ...])	Make array writer instance for array *data* and output type *out_type*

ArrayWriter

class nibabel.arraywriters.**ArrayWriter**(*array*, *out_dtype=None*, ***kwargs*)

Bases: object

Initialize array writer

Parameters

array [array-like] array-like object

out_dtype [None or dtype] dtype with which *array* will be written. For this class, *out_dtype* ` needs to be the same as the dtype of the input *array* or a swapped version of the same.

****kwargs** [keyword arguments] This class processes only:

- nan2zero : bool, optional Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs get converted with numpy `astype`, and the behavior is undefined. Ignored for floating point output.
- check_scaling : bool, optional If True, check if scaling needed and raise error if so. Default is True

Examples

```
>>> arr = np.array([0, 255], np.uint8)
>>> aw = ArrayWriter(arr)
>>> aw = ArrayWriter(arr, np.int8)
Traceback (most recent call last):
    ...
WriterError: Scaling needed but cannot scale
>>> aw = ArrayWriter(arr, np.int8, check_scaling=False)
```

__init__(*array*, *out_dtype=None*, ***kwargs*)
Initialize array writer

Parameters

array [array-like] array-like object

out_dtype [None or dtype] dtype with which *array* will be written. For this class, *out_dtype* ` needs to be the same as the dtype of the input *array* or a swapped version of the same.

****kwargs** [keyword arguments] This class processes only:

- nan2zero : bool, optional Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs get converted with numpy `astype`, and the behavior is undefined. Ignored for floating point output.
- check_scaling : bool, optional If True, check if scaling needed and raise error if so. Default is True

Examples

```
>>> arr = np.array([0, 255], np.uint8)
>>> aw = ArrayWriter(arr)
>>> aw = ArrayWriter(arr, np.int8)
Traceback (most recent call last):
    ...
WriterError: Scaling needed but cannot scale
>>> aw = ArrayWriter(arr, np.int8, check_scaling=False)
```

property array
Return array from arraywriter

finite_range()
Return (maybe cached) finite range of data array

property has_nan
True if array has NaNs

property out_dtype
Return *out_dtype* from arraywriter

scaling_needed()
Checks if scaling is needed for input array

Raises WriterError if no scaling possible.

The rules are in the code, but:

- If numpy will cast, return False (no scaling needed)
- If input or output is an object or structured type, raise
- If input is complex, raise
- If the output is float, return False
- If the input array is all zero, return False
- By now we are casting to (u)int. If the input type is a float, return True (we do need scaling)
- Now input and output types are (u)ints. If the min and max in the data are within range of the output type, return False
- Otherwise return True

to_fileobj(*fileobj*, *order='F'*, *nan2zero=None*)
Write array into *fileobj*

Parameters

fileobj [file-like object]

order [{'F', 'C'}] order (Fortran or C) to which to write array

nan2zero [{None, True, False}, optional, deprecated] Deprecated version of argument to __init__ with same name

ScalingError

class nibabel.arraywriters.**ScalingError**
Bases: *nibabel.arraywriters.WriterError*

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

SlopeArrayWriter

class nibabel.arraywriters.**SlopeArrayWriter**(*array*, *out_dtype=None*, *calc_scale=True*, *scaler_dtype=<class 'numpy.float32'>*, ***kwargs*)
Bases: *nibabel.arraywriters.ArrayWriter*

ArrayWriter that can use scalefactor for writing arrays

The scalefactor allows the array writer to write floats to int output types, and rescale larger ints to smaller. It can therefore lose precision.

It extends the ArrayWriter class with attribute:

- slope

and methods:

- reset() - reset slope to default (not adapted to self.array)
- calc_scale() - calculate slope to best write self.array

Initialize array writer

Parameters

array [array-like] array-like object

out_dtype [None or dtype] dtype with which *array* will be written. For this class, *out_dtype* ` needs to be the same as the dtype of the input *array* or a swapped version of the same.

calc_scale [{True, False}, optional] Whether to calculate scaling for writing *array* on initialization. If False, then you can calculate this scaling with `obj.calc_scale()` - see examples

scaler_dtype [dtype-like, optional] specifier for numpy dtype for scaling

****kwargs** [keyword arguments] This class processes only:

- nan2zero : bool, optional Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs get converted with numpy `astype`, and the behavior is undefined. Ignored for floating point output.

Examples

```
>>> arr = np.array([0, 254], np.uint8)
>>> aw = SlopeArrayWriter(arr)
>>> aw.slope
1.0
>>> aw = SlopeArrayWriter(arr, np.int8)
>>> aw.slope
2.0
>>> aw = SlopeArrayWriter(arr, np.int8, calc_scale=False)
>>> aw.slope
1.0
>>> aw.calc_scale()
>>> aw.slope
2.0
```

__init__(*array*, *out_dtype=None*, *calc_scale=True*, *scaler_dtype=<class 'numpy.float32'>*, ***kwargs*)

Initialize array writer

Parameters

array [array-like] array-like object

out_dtype [None or dtype] dtype with which *array* will be written. For this class, *out_dtype* ` needs to be the same as the dtype of the input *array* or a swapped version of the same.

calc_scale [{True, False}, optional] Whether to calculate scaling for writing *array* on initialization. If False, then you can calculate this scaling with `obj.calc_scale()` - see examples

scaler_dtype [dtype-like, optional] specifier for numpy dtype for scaling

****kwargs** [keyword arguments] This class processes only:

- nan2zero : bool, optional Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs get converted with numpy `astype`, and the behavior is undefined. Ignored for floating point output.

Examples

```
>>> arr = np.array([0, 254], np.uint8)
>>> aw = SlopeArrayWriter(arr)
>>> aw.slope
1.0
>>> aw = SlopeArrayWriter(arr, np.int8)
>>> aw.slope
2.0
>>> aw = SlopeArrayWriter(arr, np.int8, calc_scale=False)
>>> aw.slope
1.0
>>> aw.calc_scale()
>>> aw.slope
2.0
```

calc_scale(*force=False*)
Calculate / set scaling for floats/(u)ints to (u)ints

reset()
Set object to values before any scaling calculation

scaling_needed()
Checks if scaling is needed for input array

Raises WriterError if no scaling possible.

The rules are in the code, but:

- If numpy will cast, return False (no scaling needed)
- If input or output is an object or structured type, raise
- If input is complex, raise
- If the output is float, return False
- If the input array is all zero, return False
- If there is no finite value, return False (the writer will strip the non-finite values)
- By now we are casting to (u)int. If the input type is a float, return True (we do need scaling)
- Now input and output types are (u)ints. If the min and max in the data are within range of the output type, return False
- Otherwise return True

property slope
get/set slope

to_fileobj(*fileobj*, *order='F'*, *nan2zero=None*)
Write array into *fileobj*

Parameters

fileobj [file-like object]

order [{'F', 'C'}] order (Fortran or C) to which to write array

nan2zero [{None, True, False}, optional, deprecated] Deprecated version of argument to __init__ with same name

SlopeInterArrayWriter

class nibabel.arraywriters.**SlopeInterArrayWriter**(*array*, *out_dtype=None*, *calc_scale=True*, *scaler_dtype=<class 'numpy.float32'>*, ***kwargs*)

Bases: *nibabel.arraywriters.SlopeArrayWriter*

Array writer that can use slope and intercept to scale array

The writer can subtract an intercept, and divided by a slope, in order to be able to convert floating point values into a (u)int range, or to convert larger (u)ints to smaller.

It extends the ArrayWriter class with attributes:

- inter
- slope

and methods:

- reset() - reset inter, slope to default (not adapted to self.array)
- calc_scale() - calculate inter, slope to best write self.array

Initialize array writer

Parameters

array [array-like] array-like object

out_dtype [None or dtype] dtype with which *array* will be written. For this class, *out_dtype* ` needs to be the same as the dtype of the input *array* or a swapped version of the same.

calc_scale [{True, False}, optional] Whether to calculate scaling for writing *array* on initialization. If False, then you can calculate this scaling with `obj.calc_scale()` - see examples

scaler_dtype [dtype-like, optional] specifier for numpy dtype for slope, intercept

****kwargs** [keyword arguments] This class processes only:

- nan2zero : bool, optional Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs get converted with numpy `astype`, and the behavior is undefined. Ignored for floating point output.

Examples

```
>>> arr = np.array([0, 255], np.uint8)
>>> aw = SlopeInterArrayWriter(arr)
>>> aw.slope, aw.inter
(1.0, 0.0)
>>> aw = SlopeInterArrayWriter(arr, np.int8)
>>> (aw.slope, aw.inter) == (1.0, 128)
True
>>> aw = SlopeInterArrayWriter(arr, np.int8, calc_scale=False)
>>> aw.slope, aw.inter
```

(continues on next page)

(continued from previous page)

```
(1.0, 0.0)
>>> aw.calc_scale()
>>> (aw.slope, aw.inter) == (1.0, 128)
True
```

__init__(*array*, *out_dtype=None*, *calc_scale=True*, *scaler_dtype=<class 'numpy.float32'>*, ***kwargs*)

Initialize array writer

Parameters

array [array-like] array-like object

out_dtype [None or dtype] dtype with which *array* will be written. For this class, *out_dtype* ` needs to be the same as the dtype of the input *array* or a swapped version of the same.

calc_scale [{True, False}, optional] Whether to calculate scaling for writing *array* on initialization. If False, then you can calculate this scaling with `obj.calc_scale()` - see examples

scaler_dtype [dtype-like, optional] specifier for numpy dtype for slope, intercept

****kwargs** [keyword arguments] This class processes only:

- nan2zero : bool, optional Whether to set NaN values to 0 when writing integer output. Defaults to True. If False, NaNs get converted with numpy `astype`, and the behavior is undefined. Ignored for floating point output.

Examples

```
>>> arr = np.array([0, 255], np.uint8)
>>> aw = SlopeInterArrayWriter(arr)
>>> aw.slope, aw.inter
(1.0, 0.0)
>>> aw = SlopeInterArrayWriter(arr, np.int8)
>>> (aw.slope, aw.inter) == (1.0, 128)
True
>>> aw = SlopeInterArrayWriter(arr, np.int8, calc_scale=False)
>>> aw.slope, aw.inter
(1.0, 0.0)
>>> aw.calc_scale()
>>> (aw.slope, aw.inter) == (1.0, 128)
True
```

property inter

get/set inter

reset()

Set object to values before any scaling calculation

to_fileobj(*fileobj*, *order='F'*, *nan2zero=None*)

Write array into *fileobj*

Parameters

fileobj [file-like object]

order [{'F', 'C'}] order (Fortran or C) to which to write array

nan2zero [{None, True, False}, optional, deprecated] Deprecated version of argument to __init__ with same name

WriterError

class nibabel.arraywriters.**WriterError**

Bases: Exception

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

get_slope_inter

nibabel.arraywriters.**get_slope_inter**(*writer*)

Return slope, intercept from array writer object

Parameters

writer [ArrayWriter instance]

Returns

slope [scalar] slope in *writer* or 1.0 if not present

inter [scalar] intercept in *writer* or 0.0 if not present

Examples

```
>>> arr = np.arange(10)
>>> get_slope_inter(ArrayWriter(arr))
(1.0, 0.0)
>>> get_slope_inter(SlopeArrayWriter(arr))
(1.0, 0.0)
>>> get_slope_inter(SlopeInterArrayWriter(arr))
(1.0, 0.0)
```

make_array_writer

nibabel.arraywriters.**make_array_writer**(*data*, *out_type*, *has_slope=True*, *has_intercept=True*, ***kwargs*)

Make array writer instance for array *data* and output type *out_type*

Parameters

data [array-like] array for which to create array writer

out_type [dtype-like] input to numpy dtype to specify array writer output type

has_slope [{True, False}] If True, array write can use scaling to adapt the array to *out_type*

has_intercept [{True, False}] If True, array write can use intercept to adapt the array to *out_type*

****kwargs** [other keyword arguments] to pass to the arraywriter class

Returns

writer [arraywriter instance] Instance of array writer, with class adapted to *has_intercept* and *has_slope*.

Examples

```
>>> aw = make_array_writer(np.arange(10), np.uint8, True, True)
>>> type(aw) == SlopeInterArrayWriter
True
>>> aw = make_array_writer(np.arange(10), np.uint8, True, False)
>>> type(aw) == SlopeArrayWriter
True
>>> aw = make_array_writer(np.arange(10), np.uint8, False, False)
>>> type(aw) == ArrayWriter
True
```

casting

Utilties for casting numpy values in various ways

Most routines work round some numpy oddities in floating point precision and casting. Others work round numpy casting to and from python ints

CastingError	
FloatingError	
able_int_type(values)	Find the smallest integer numpy type to contain sequence *values*
as_int(x[, check])	Return python integer representation of number
best_float()	Floating point type with best precision
ceil_exact(val, flt_type)	Return nearest exact integer >= *val* in float type *flt_type*
float_to_int(arr, int_type[, nan2zero, infmax])	Convert floating point array *arr* to type *int_type*
floor_exact(val, flt_type)	Return nearest exact integer <= *val* in float type *flt_type*
floor_log2(x)	floor of log2 of abs(*x*)
have_binary128()	True if we have a binary128 IEEE longdouble
int_abs(arr)	Absolute values of array taking care of max negative int values
int_to_float(val, flt_type)	Convert integer *val* to floating point type *flt_type*
longdouble_lte_float64()	Return True if longdouble appears to have the same precision as float64
longdouble_precision_improved()	True if longdouble precision increased since initial import
ok_floats()	Return floating point types sorted by precision
on_powerpc()	True if we are running on a Power PC platform
shared_range(flt_type, int_type)	Min and max in float type that are >=min, <=max in integer type
type_info(np_type)	Return dict with min, max, nexp, nmant, width for numpy type *np_type*
ulp([val])	Return gap between *val* and nearest representable number of same type

CastingError

class nibabel.casting.**CastingError**
Bases: Exception

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

FloatingError

class nibabel.casting.**FloatingError**
Bases: Exception

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

able_int_type

nibabel.casting.**able_int_type**(*values*)
Find the smallest integer numpy type to contain sequence *values*

Prefers uint to int if minimum is >= 0

Parameters

values [sequence] sequence of integer values

Returns

itype [None or numpy type] numpy integer type or None if no integer type holds all *values*

Examples

```
>>> able_int_type([0, 1]) == np.uint8
True
>>> able_int_type([-1, 1]) == np.int8
True
```

as_int

nibabel.casting.**as_int**(*x*, *check=True*)
Return python integer representation of number

This is useful because the numpy int(val) mechanism is broken for large values in np.longdouble.

It is also useful to work around a numpy 1.4.1 bug in conversion of uints to python ints.

This routine will still raise an OverflowError for values that are outside the range of float64.

Parameters

x [object] integer, unsigned integer or floating point value

check [{True, False}] If True, raise error for values that are not integers

Returns

i [int] Python integer

Examples

```
>>> as_int(2.0)
2
>>> as_int(-2.0)
-2
>>> as_int(2.1)
Traceback (most recent call last):
    ...
FloatingError: Not an integer: 2.1
>>> as_int(2.1, check=False)
2
```

best_float

nibabel.casting.**best_float**()

Floating point type with best precision

This is nearly always np.longdouble, except on Windows, where np.longdouble is Intel80 storage, but with float64 precision for calculations. In that case we return float64 on the basis it's the fastest and smallest at the highest precision.

SPARC float128 also proved so slow that we prefer float64.

Returns

best_type [numpy type] floating point type with highest precision

Notes

Needs to run without error for module import, because it is called in `ok_floats` below, and therefore in setting module global `OK_FLOATS`.

ceil_exact

nibabel.casting.**ceil_exact**(*val*, *flt_type*)

Return nearest exact integer >= *val* in float type *flt_type*

Parameters

val [int] We have to pass val as an int rather than the floating point type because large integers cast as floating point may be rounded by the casting process.

flt_type [numpy type] numpy float type.

Returns

ceil_val [object] value of same floating point type as *val*, that is the nearest exact integer in this type such that *floor_val* >= *val*. Thus if *val* is exact in *flt_type*, *ceil_val* == *val*.

Examples

Obviously 2 is within the range of representable integers for float32

```
>>> ceil_exact(2, np.float32)
2.0
```

As is 2**24-1 (the number of significand digits is 23 + 1 implicit)

```
>>> ceil_exact(2**24-1, np.float32) == 2**24-1
True
```

But 2**24+1 gives a number that float32 can't represent exactly

```
>>> ceil_exact(2**24+1, np.float32) == 2**24+2
True
```

As for the numpy ceil function, negatives ceil towards inf

```
>>> ceil_exact(-2**24-1, np.float32) == -2**24
True
```

float_to_int

nibabel.casting.**float_to_int**(*arr*, *int_type*, *nan2zero=True*, *infmax=False*)

Convert floating point array *arr* to type *int_type*

- Rounds numbers to nearest integer
- Clips values to prevent overflows when casting
- Converts NaN to 0 (for *nan2zero* == True)

Casting floats to integers is delicate because the result is undefined and platform specific for float values outside the range of *int_type*. Define `shared_min` to be the minimum value that can be exactly represented in both the float type of *arr* and *int_type*. Define *shared_max* to be the equivalent maximum value. To avoid undefined results we threshold *arr* at `shared_min` and `shared_max`.

Parameters

arr [array-like] Array of floating point type

int_type [object] Numpy integer type

nan2zero [{True, False, None}] Whether to convert NaN value to zero. Default is True. If False, and NaNs are present, raise CastingError. If None, do not check for NaN values and pass through directly to the `astype` casting mechanism. In this last case, the resulting value is undefined.

infmax [{False, True}] If True, set np.inf values in *arr* to be *int_type* integer maximum value, -np.inf as *int_type* integer minimum. If False, set +/- infs to be `shared_min`, `shared_max` as defined above. Therefore False gives faster conversion at the expense of infs that are further from infinity.

Returns

iarr [ndarray] of type *int_type*

Notes

Numpy relies on the C library to cast from float to int using the standard `astype` method of the array.

Quoting from section F4 of the C99 standard:

> If the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the "invalid" floating-point exception is raised and the resulting value is unspecified.

Hence we threshold at `shared_min` and `shared_max` to avoid casting to values that are undefined.

See: https://en.wikipedia.org/wiki/C99 . There are links to the C99 standard from that page.

Examples

```
>>> float_to_int([np.nan, np.inf, -np.inf, 1.1, 6.6], np.int16)
array([     0,  32767, -32768,      1,      7], dtype=int16)
```

floor_exact

nibabel.casting.**floor_exact**(*val*, *flt_type*)

Return nearest exact integer <= *val* in float type *flt_type*

Parameters

val [int] We have to pass val as an int rather than the floating point type because large integers cast as floating point may be rounded by the casting process.

flt_type [numpy type] numpy float type.

Returns

floor_val [object] value of same floating point type as *val*, that is the nearest exact integer in this type such that *floor_val* <= *val*. Thus if *val* is exact in *flt_type*, *floor_val* == *val*.

Examples

Obviously 2 is within the range of representable integers for float32

```
>>> floor_exact(2, np.float32)
2.0
```

As is 2**24-1 (the number of significand digits is 23 + 1 implicit)

```
>>> floor_exact(2**24-1, np.float32) == 2**24-1
True
```

But 2**24+1 gives a number that float32 can't represent exactly

```
>>> floor_exact(2**24+1, np.float32) == 2**24
True
```

As for the numpy floor function, negatives floor towards -inf

```
>>> floor_exact(-2**24-1, np.float32) == -2**24-2
True
```

floor_log2

nibabel.casting.**floor_log2**(*x*)
floor of log2 of abs(*x*)

Embarrassingly, from https://en.wikipedia.org/wiki/Binary_logarithm

Parameters

x [int]

Returns

L [None or int] floor of base 2 log of *x*. None if *x* == 0.

Examples

```
>>> floor_log2(2**9+1)
9
>>> floor_log2(-2**9+1)
8
>>> floor_log2(0.5)
-1
>>> floor_log2(0) is None
True
```

have_binary128

nibabel.casting.**have_binary128**()
True if we have a binary128 IEEE longdouble

int_abs

nibabel.casting.**int_abs**(*arr*)
Absolute values of array taking care of max negative int values

Parameters

arr [array-like]

Returns

abs_arr [array] array the same shape as *arr* in which all negative numbers have been changed to positive numbers with the magnitude.

Examples

This kind of thing is confusing in base numpy:

```
>>> import numpy as np
>>> np.abs(np.int8(-128))
-128
```

`int_abs` fixes that:

```
>>> int_abs(np.int8(-128))
128
>>> int_abs(np.array([-128, 127], dtype=np.int8))
array([128, 127], dtype=uint8)
>>> int_abs(np.array([-128, 127], dtype=np.float32))
array([128., 127.], dtype=float32)
```

int_to_float

nibabel.casting.**int_to_float**(*val*, *flt_type*)

Convert integer *val* to floating point type *flt_type*

Why is this so complicated?

At least in numpy <= 1.6.1, numpy longdoubles do not correctly convert to ints, and ints do not correctly convert to longdoubles. Specifically, in both cases, the values seem to go through float64 conversion on the way, so to convert better, we need to split into float64s and sum up the result.

Parameters

val [int] Integer value

flt_type [object] numpy floating point type

Returns

f [numpy scalar] of type *flt_type*

longdouble_lte_float64

nibabel.casting.**longdouble_lte_float64**()

Return True if longdouble appears to have the same precision as float64

longdouble_precision_improved

nibabel.casting.**longdouble_precision_improved**()

True if longdouble precision increased since initial import

This can happen on Windows compiled with MSVC. It may be because libraries compiled with mingw (longdouble is Intel80) get linked to numpy compiled with MSVC (longdouble is Float64)

ok_floats

nibabel.casting.**ok_floats**()
Return floating point types sorted by precision

Remove longdouble if it has no higher precision than float64

on_powerpc

nibabel.casting.**on_powerpc**()
True if we are running on a Power PC platform

Has to deal with older Macs and IBM POWER7 series among others

shared_range

nibabel.casting.**shared_range**(*flt_type*, *int_type*)
Min and max in float type that are >=min, <=max in integer type

This is not as easy as it sounds, because the float type may not be able to exactly represent the max or min integer values, so we have to find the next exactly representable floating point value to do the thresholding.

Parameters

flt_type [dtype specifier] A dtype specifier referring to a numpy floating point type. For example, `f4`, `np.dtype('f4')`, `np.float32` are equivalent.

int_type [dtype specifier] A dtype specifier referring to a numpy integer type. For example, `i4`, `np.dtype('i4')`, `np.int32` are equivalent

Returns

mn [object] Number of type *flt_type* that is the minumum value in the range of *int_type*, such that `mn.astype(int_type)` >= min of *int_type*

mx [object] Number of type *flt_type* that is the maximum value in the range of *int_type*, such that `mx.astype(int_type)` <= max of *int_type*

Examples

```
>>> shared_range(np.float32, np.int32) == (-2147483648.0, 2147483520.0)
True
>>> shared_range('f4', 'i4') == (-2147483648.0, 2147483520.0)
True
```

type_info

nibabel.casting.**type_info**(*np_type*)

Return dict with min, max, nexp, nmant, width for numpy type *np_type*

Type can be integer in which case nexp and nmant are None.

Parameters

np_type [numpy type specifier] Any specifier for a numpy dtype

Returns

info [dict] with fields `min` (minimum value), `max` (maximum value), `nexp` (exponent width), `nmant` (significand precision not including implicit first digit), `minexp` (minimum exponent), `maxexp` (maximum exponent), `width` (width in bytes). (`nexp`, `nmant`, `minexp`, `maxexp`) are None for integer types. Both `min` and `max` are of type *np_type*.

Raises

FloatingError for floating point types we don't recognize

Notes

You might be thinking that `np.finfo` does this job, and it does, except for PPC long doubles (https://github.com/numpy/numpy/issues/2669) and float96 on Windows compiled with Mingw. This routine protects against such errors in `np.finfo` by only accepting values that we know are likely to be correct.

ulp

nibabel.casting.**ulp**(*val=1.0*)

Return gap between *val* and nearest representable number of same type

This is the value of a unit in the last place (ULP), and is similar in meaning to the MATLAB eps function.

Parameters

val [scalar, optional] scalar value of any numpy type. Default is 1.0 (float64)

Returns

ulp_val [scalar] gap between *val* and nearest representable number of same type

Notes

The wikipedia article on machine epsilon points out that the term *epsilon* can be used in the sense of a unit in the last place (ULP), or as the maximum relative rounding error. The MATLAB `eps` function uses the ULP meaning, but this function is `ulp` rather than `eps` to avoid confusion between different meanings of *eps*.

10.5.5 System utilities

data	Utilities to find files from NIPY data packages
environment	Settings from the system environment relevant to NIPY

data

Utilities to find files from NIPY data packages

Bomber(name, msg)	Class to raise an informative error when used
BomberError	Error when trying to access Bomber instance
DataError	
Datasource(base_path)	Simple class to add base path to relative path
VersionedDatasource(base_path[, config_filename])	Datasource with version information in config file
datasource_or_bomber(pkg_def, **options)	Return a viable datasource or a Bomber
find_data_dir(root_dirs, *names)	Find relative path given path prefixes to search
get_data_path()	Return specified or guessed locations of NIPY data files
make_datasource(pkg_def, **kwargs)	Return datasource defined by *pkg_def* as found in *data_path*

Bomber

class nibabel.data.**Bomber**(*name*, *msg*)

Bases: object

Class to raise an informative error when used

__init__(*name*, *msg*)

Initialize self. See help(type(self)) for accurate signature.

BomberError

class nibabel.data.**BomberError**

Bases: *nibabel.data.DataError*, AttributeError

Error when trying to access Bomber instance

Should be instance of AttributeError to allow Python 3 inspect to do various `hasattr` checks without raising an error

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

DataError

class nibabel.data.**DataError**
Bases: Exception

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Datasource

class nibabel.data.**Datasource**(*base_path*)
Bases: object

Simple class to add base path to relative path

Initialize datasource

Parameters

base_path [str] path to prepend to all relative paths

Examples

```
>>> from os.path import join as pjoin
>>> repo = Datasource(pjoin('a', 'path'))
>>> fname = repo.get_filename('somedir', 'afile.txt')
>>> fname == pjoin('a', 'path', 'somedir', 'afile.txt')
True
```

__init__(*base_path*)
Initialize datasource

Parameters

base_path [str] path to prepend to all relative paths

Examples

```
>>> from os.path import join as pjoin
>>> repo = Datasource(pjoin('a', 'path'))
>>> fname = repo.get_filename('somedir', 'afile.txt')
>>> fname == pjoin('a', 'path', 'somedir', 'afile.txt')
True
```

get_filename(**path_parts*)
Prepend base path to **path_parts*

We make no check whether the returned path exists.

Parameters

***path_parts** [sequence of strings]

Returns

fname [str] result of `os.path.join(*path_parts)`, with ``self.base_path`` prepended

list_files(*relative=True*)
Recursively list the files in the data source directory.

Parameters

relative: bool, optional If True, path returned are relative to the base paht of the data source.

Returns

file_list: list of strings List of the paths of all the files in the data source.

VersionedDatasource

class nibabel.data.**VersionedDatasource**(*base_path*, *config_filename=None*)
Bases: *nibabel.data.Datasource*

Datasource with version information in config file

Initialize versioned datasource

We assume that there is a configuration file with version information in datasource directory tree.

The configuration file contains an entry like:

```
[DEFAULT]
version = 0.3
```

The version should have at least a major and a minor version number in the form above.

Parameters

base_path [str] path to prepend to all relative paths

config_filaname [None or str] relative path to configuration file containing version

__init__(*base_path*, *config_filename=None*)
Initialize versioned datasource

We assume that there is a configuration file with version information in datasource directory tree.

The configuration file contains an entry like:

```
[DEFAULT]
version = 0.3
```

The version should have at least a major and a minor version number in the form above.

Parameters

base_path [str] path to prepend to all relative paths

config_filaname [None or str] relative path to configuration file containing version

datasource_or_bomber

nibabel.data.**datasource_or_bomber**(*pkg_def*, ***options*)

Return a viable datasource or a Bomber

This is to allow module level creation of datasource objects. We create the objects, so that, if the data exist, and are the correct version, the objects are valid datasources, otherwise, they raise an error on access, warning about the lack of data or the version numbers.

The parameters are as for `make_datasource` in this module.

Parameters

pkg_def [dict] dict containing at least key 'relpath'. Can optioanlly have keys 'name' (package name), 'install hint' (for helpful error messages) and 'min version' giving the minimum necessary version string for the package.

data_path [sequence of strings or None, optional]

Returns

ds [datasource or `Bomber` instance]

find_data_dir

nibabel.data.**find_data_dir**(*root_dirs*, **names*)

Find relative path given path prefixes to search

We raise a DataError if we can't find the relative path

Parameters

root_dirs [sequence of strings] sequence of paths in which to search for data directory

***names** [sequence of strings] sequence of strings naming directory to find. The name to search for is given by `os.path.join(*names)`

Returns

data_dir [str] full path (root path added to **names* above)

get_data_path

nibabel.data.**get_data_path**()

Return specified or guessed locations of NIPY data files

The algorithm is to return paths, extracted from strings, where strings are found in the following order:

1. The contents of environment variable `NIPY_DATA_PATH`
2. Any section = `DATA`, key = `path` value in a `config.ini` file in your nipy user directory (found with `get_nipy_user_dir()`)
3. Any section = `DATA`, key = `path` value in any files found with a `sorted(glob.glob(os.path.join(sys_dir, '*.ini')))` search, where `sys_dir` is found with `get_nipy_system_dir()`
4. If `sys.prefix` is `/usr`, we add `/usr/local/share/nipy`. We need this because Python 2.6 in Debian / Ubuntu does default installs to `/usr/local`.
5. The result of `get_nipy_user_dir()`

Therefore, any paths found in `NIPY_DATA_PATH` will be searched before paths found in the user directory `config.ini`

Parameters

None

Returns

paths [sequence of paths]

Notes

We have to add `/usr/local/share/nipy` if sys.prefix is `/usr`, because Debian has patched distutils in Python 2.6 to do default distutils installs there:

- https://www.debian.org/doc/packaging-manuals/python-policy/ap-packaging_tools.html#s-distutils
- https://www.mail-archive.com/debian-python@lists.debian.org/msg05084.html

Examples

```
>>> pth = get_data_path()
```

make_datasource

nibabel.data.**make_datasource**(*pkg_def*, ***kwargs*)

Return datasource defined by *pkg_def* as found in *data_path*

data_path is the only allowed keyword argument.

pkg_def is a dictionary with at least one key - 'relpath'. 'relpath' is a relative path with unix forward slash separators.

The relative path to the data is found with:

```
names = pkg_def['name'].split('/')
rel_path = os.path.join(names)
```

We search for this relative path in the list of paths given by *data_path*. By default *data_path* is given by `get_data_path()` in this module.

If we can't find the relative path, raise a DataError

Parameters

pkg_def [dict] dict containing at least the key 'relpath'. 'relpath' is the data path of the package relative to *data_path*. It is in unix path format (using forward slashes as directory separators). *pkg_def* can also contain optional keys 'name' (the name of the package), and / or a key 'install hint' that we use in the returned error message from trying to use the resulting datasource

data_path [sequence of strings or None, optional] sequence of paths in which to search for data. If None (the default), then use `get_data_path()`

Returns

datasource [`VersionedDatasource`] An initialized `VersionedDatasource` instance

environment

Settings from the system environment relevant to NIPY

get_home_dir()	Return the closest possible equivalent to a 'home' directory.
get_nipy_system_dir()	Get systemwide NIPY configuration file directory
get_nipy_user_dir()	Get the NIPY user directory

get_home_dir

nibabel.environment.**get_home_dir**()

Return the closest possible equivalent to a 'home' directory.

The path may not exist; code using this routine should not expect the directory to exist.

Parameters

None

Returns

home_dir [string] best guess at location of home directory

get_nipy_system_dir

nibabel.environment.**get_nipy_system_dir**()

Get systemwide NIPY configuration file directory

On posix systems this will be `/etc/nipy`. On Windows, the directory is less useful, but by default it will be `C:\etc\nipy`

The path may well not exist; code using this routine should not expect the directory to exist.

Parameters

None

Returns

nipy_dir [string] path to systemwide NIPY configuration directory

Examples

```
>>> pth = get_nipy_system_dir()
```

get_nipy_user_dir

nibabel.environment.**get_nipy_user_dir**()

Get the NIPY user directory

This uses the logic in *get_home_dir* to find the home directory and the adds either .nipy or _nipy to the end of the path.

We check first in environment variable `NIPY_USER_DIR`, otherwise returning the default of `<homedir>/.nipy` (Unix) or `<homedir>/_nipy` (Windows)

The path may well not exist; code using this routine should not expect the directory to exist.

Parameters

None

Returns

nipy_dir [string] path to user's NIPY configuration directory

Examples

```
>>> pth = get_nipy_user_dir()
```

10.5.6 Miscellaneous Helpers

arrayproxy	Array proxy base class
affines	Utility routines for working with points and affine transforms
batteryrunners	Battery runner classes and Report classes
data	Utilities to find files from NIPY data packages
dft	DICOM filesystem tools
fileholders	Fileholder class
filename_parser	Create filename pairs, triplets etc, with expected extensions
fileslice	Utilities for getting array slices out of file-like objects
onetime	Descriptor support for NIPY.
openers	Context manager openers for various fileobject types
optpkg	Routines to support optional packages
rstutils	ReStructured Text utilities
tmpdirs	Contexts for *with* statement providing temporary directories
tripwire	Class to raise error for missing modules or other misfortunes
wrapstruct	Class to wrap numpy structured array

arrayproxy

Array proxy base class

The proxy API is - at minimum:

- The object has a read-only attribute `shape`
- read only `is_proxy` attribute / property set to True
- the object returns the data array from `np.asarray(prox)`
- returns array slice from `prox[<slice_spec>]` where `<slice_spec>` is any ndarray slice specification that does not use numpy 'advanced indexing'.
- modifying no object outside `obj` will affect the result of `np.asarray(obj)`. Specifically:
  - Changes in position (`obj.tell()`) of passed file-like objects will not affect the output of from `np.asarray(proxy)`.
  - if you pass a header into the __init__, then modifying the original header will not affect the result of the array return.

See `nibabel.tests.test_proxy_api` for proxy API conformance checks.

ArrayProxy(file_like, spec, *[, mmap, ...])	Class to act as proxy for the array that can be read from a file
is_proxy(obj)	Return True if *obj* is an array proxy
reshape_dataobj(obj, shape)	Use *obj* reshape method if possible, else numpy reshape function

ArrayProxy

class nibabel.arrayproxy.**ArrayProxy**(*file_like*, *spec*, *, *mmap=True*, *keep_file_open=None*)

Bases: `object`

Class to act as proxy for the array that can be read from a file

The array proxy allows us to freeze the passed fileobj and header such that it returns the expected data array.

This implementation assumes a contiguous array in the file object, with one of the numpy dtypes, starting at a given file position `offset` with single `slope` and `intercept` scaling to produce output values.

The class __init__ requires a spec which defines how the data will be read and rescaled. The spec may be a tuple of length 2 - 5, containing the shape, storage dtype, offset, slope and intercept, or a `header` object with methods:

- get_data_shape
- get_data_dtype
- get_data_offset
- get_slope_inter

A header should also have a 'copy' method. This requirement will go away when the deprecated 'header' property goes away.

This implementation allows us to deal with Analyze and its variants, including Nifti1, and with the MGH format.

Other image types might need more specific classes to implement the API. See *nibabel.minc1*, *nibabel.ecat* and *nibabel.parrec* for examples.

Initialize array proxy instance

Parameters

file_like [object] File-like object or filename. If file-like object, should implement at least `read` and `seek`.

spec [object or tuple] Tuple must have length 2-5, with the following values:

1. shape: tuple - tuple of ints describing shape of data;
2. storage_dtype: dtype specifier - dtype of array inside proxied file, or input to `numpy.dtype` to specify array dtype;
3. offset: int - offset, in bytes, of data array from start of file (default: 0);
4. slope: float - scaling factor for resulting data (default: 1.0);
5. inter: float - intercept for rescaled data (default: 0.0).

OR

Header object implementing `get_data_shape`, `get_data_dtype`, `get_data_offset`, `get_slope_inter`

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If *file_like* cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_like` is an open file handle, this setting has no effect. The default value (`None`) will result in the value of `KEEP_FILE_OPEN_DEFAULT` being used.

__init__(*file_like*, *spec*, *, *mmap=True*, *keep_file_open=None*)

Initialize array proxy instance

Parameters

file_like [object] File-like object or filename. If file-like object, should implement at least `read` and `seek`.

spec [object or tuple] Tuple must have length 2-5, with the following values:

1. shape: tuple - tuple of ints describing shape of data;
2. storage_dtype: dtype specifier - dtype of array inside proxied file, or input to `numpy.dtype` to specify array dtype;
3. offset: int - offset, in bytes, of data array from start of file (default: 0);
4. slope: float - scaling factor for resulting data (default: 1.0);
5. inter: float - intercept for rescaled data (default: 0.0).

OR

Header object implementing `get_data_shape`, `get_data_dtype`, `get_data_offset`, `get_slope_inter`

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If *file_like* cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_like` is an open file handle, this setting has no effect. The default value (`None`) will result in the value of `KEEP_FILE_OPEN_DEFAULT` being used.

property dtype

get_unscaled()
Read data from file

This is an optional part of the proxy API

property inter

property is_proxy

property ndim

property offset

order = 'F'

reshape(*shape*)
Return an ArrayProxy with a new shape, without modifying data

property shape

property slope

is_proxy

nibabel.arrayproxy.**is_proxy**(*obj*)
Return True if *obj* is an array proxy

reshape_dataobj

nibabel.arrayproxy.**reshape_dataobj**(*obj*, *shape*)
Use *obj* reshape method if possible, else numpy reshape function

affines

Utility routines for working with points and affine transforms

AffineError	Errors in calculating or using affines
append_diag(aff, steps[, starts])	Add diagonal elements *steps* and translations *starts* to affine
apply_affine(aff, pts)	Apply affine matrix *aff* to points *pts*

continues on next page

Table 62 – continued from previous page

dot_reduce(*args)	Apply numpy dot product function from right to left on arrays
from_matvec(matrix[, vector])	Combine a matrix and vector into an homogeneous affine
obliquity(affine)	Estimate the *obliquity* an affine's axes represent.
rescale_affine(affine, shape, zooms[, new_shape])	Return a new affine matrix with updated voxel sizes (zooms)
to_matvec(transform)	Split a transform into its matrix and vector components.
voxel_sizes(affine)	Return voxel size for each input axis given *affine*

AffineError

class nibabel.affines.**AffineError**

Bases: ValueError

Errors in calculating or using affines

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

append_diag

nibabel.affines.**append_diag**(*aff*, *steps*, *starts=()*)

Add diagonal elements *steps* and translations *starts* to affine

Typical use is in expanding 4x4 affines to larger dimensions. Nipy is the main consumer because it uses NxM affines, whereas we generally only use 4x4 affines; the routine is here for convenience.

Parameters

aff [2D array] N by M affine matrix

steps [scalar or sequence] diagonal elements to append.

starts [scalar or sequence] elements to append to last column of *aff*, representing translations corresponding to the *steps*. If empty, expands to a vector of zeros of the same length as *steps*

Returns

aff_plus [2D array] Now P by Q where L = `len(steps)` and P == N+L, Q=N+L

Examples

```
>>> aff = np.eye(4)
>>> aff[:3,:3] = np.arange(9).reshape((3,3))
>>> append_diag(aff, [9, 10], [99,100])
array([[  0.,   1.,   2.,   0.,   0.,   0.],
       [  3.,   4.,   5.,   0.,   0.,   0.],
       [  6.,   7.,   8.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   9.,   0.,  99.],
       [  0.,   0.,   0.,   0.,  10., 100.],
       [  0.,   0.,   0.,   0.,   0.,   1.]])
```

apply_affine

nibabel.affines.**apply_affine**(*aff*, *pts*)

Apply affine matrix *aff* to points *pts*

Returns result of application of *aff* to the *right* of *pts*. The coordinate dimension of *pts* should be the last.

For the 3D case, *aff* will be shape (4,4) and *pts* will have final axis length 3 - maybe it will just be N by 3. The return value is the transformed points, in this case:

```
res = np.dot(aff[:3,:3], pts.T) + aff[:3,3:4]
transformed_pts = res.T
```

This routine is more general than 3D, in that *aff* can have any shape (N,N), and *pts* can have any shape, as long as the last dimension is for the coordinates, and is therefore length N-1.

Parameters

aff [(N, N) array-like] Homogenous affine, for 3D points, will be 4 by 4. Contrary to first appearance, the affine will be applied on the left of *pts*.

pts [(..., N-1) array-like] Points, where the last dimension contains the coordinates of each point. For 3D, the last dimension will be length 3.

Returns

transformed_pts [(..., N-1) array] transformed points

Examples

```
>>> aff = np.array([[0,2,0,10],[3,0,0,11],[0,0,4,12],[0,0,0,1]])
>>> pts = np.array([[1,2,3],[2,3,4],[4,5,6],[6,7,8]])
>>> apply_affine(aff, pts)
array([[14, 14, 24],
       [16, 17, 28],
       [20, 23, 36],
       [24, 29, 44]]...)
```

Just to show that in the simple 3D case, it is equivalent to:

```
>>> (np.dot(aff[:3,:3], pts.T) + aff[:3,3:4]).T
array([[14, 14, 24],
       [16, 17, 28],
       [20, 23, 36],
       [24, 29, 44]]...)
```

But *pts* can be a more complicated shape:

```
>>> pts = pts.reshape((2,2,3))
>>> apply_affine(aff, pts)
array([[[14, 14, 24],
        [16, 17, 28]],

       [[20, 23, 36],
        [24, 29, 44]]]...)
```

dot_reduce

nibabel.affines.**dot_reduce**(**args*)

Apply numpy dot product function from right to left on arrays

For passed arrays $A, B, C, ...Z$ returns $A\dot{B}\dot{C}...\dot{Z}$ where "." is the numpy array dot product.

Parameters

****args** [arrays] Arrays that can be passed to numpy `dot` function

Returns

dot_product [array] If there are N arguments, result of `arg[0].dot(arg[1].dot(arg[2].dot ... arg[N-2].dot(arg[N-1])))...`

from_matvec

nibabel.affines.**from_matvec**(*matrix*, *vector=None*)

Combine a matrix and vector into an homogeneous affine

Combine a rotation / scaling / shearing matrix and translation vector into a transform in homogeneous coordinates.

Parameters

matrix [array-like] An NxM array representing the the linear part of the transform. A transform from an M-dimensional space to an N-dimensional space.

vector [None or array-like, optional] None or an (N,) array representing the translation. None corresponds to an (N,) array of zeros.

Returns

xform [array] An (N+1, M+1) homogenous transform matrix.

See also:

to_matvec

Examples

```
>>> from_matvec(np.diag([2, 3, 4]), [9, 10, 11])
array([[ 2,  0,  0,  9],
       [ 0,  3,  0, 10],
       [ 0,  0,  4, 11],
       [ 0,  0,  0,  1]])
```

The *vector* argument is optional:

```
>>> from_matvec(np.diag([2, 3, 4]))
array([[2, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 0, 4, 0],
       [0, 0, 0, 1]])
```

obliquity

nibabel.affines.**obliquity**(*affine*)

Estimate the *obliquity* an affine's axes represent.

The term *obliquity* is defined here as the rotation of those axes with respect to the cardinal axes. This implementation is inspired by AFNI's implementation. For further details about *obliquity*, check AFNI's documentation.

Parameters

affine [2D array-like] Affine transformation array. Usually shape (4, 4), but can be any 2D array.

Returns

angles [1D array-like] The *obliquity* of each axis with respect to the cardinal axes, in radians.

rescale_affine

nibabel.affines.**rescale_affine**(*affine*, *shape*, *zooms*, *new_shape=None*)

Return a new affine matrix with updated voxel sizes (zooms)

This function preserves the rotations and shears of the original affine, as well as the RAS location of the central voxel of the image.

Parameters

affine [(N, N) array-like] NxN transform matrix in homogeneous coordinates representing an affine transformation from an (N-1)-dimensional space to an (N-1)-dimensional space. An example is a 4x4 transform representing rotations and translations in 3 dimensions.

shape [(N-1,) array-like] The extent of the (N-1) dimensions of the original space

zooms [(N-1,) array-like] The size of voxels of the output affine

new_shape [(N-1,) array-like, optional] The extent of the (N-1) dimensions of the space described by the new affine. If `None`, use `shape`.

Returns

affine [(N, N) array] A new affine transform with the specified voxel sizes

to_matvec

nibabel.affines.**to_matvec**(*transform*)

Split a transform into its matrix and vector components.

The tranformation must be represented in homogeneous coordinates and is split into its rotation matrix and translation vector components.

Parameters

transform [array-like] NxM transform matrix in homogeneous coordinates representing an affine transformation from an (N-1)-dimensional space to an (M-1)-dimensional space. An example is a 4x4 transform representing rotations and translations in 3 dimensions. A 4x3 matrix can represent a 2-dimensional plane embedded in 3 dimensional space.

Returns

matrix [(N-1, M-1) array] Matrix component of *transform*

vector [(M-1,) array] Vector compoent of *transform*

See also:

from_matvec

Examples

```
>>> aff = np.diag([2, 3, 4, 1])
>>> aff[:3,3] = [9, 10, 11]
>>> to_matvec(aff)
(array([[2, 0, 0],
       [0, 3, 0],
       [0, 0, 4]]), array([ 9, 10, 11]))
```

voxel_sizes

nibabel.affines.**voxel_sizes**(*affine*)

Return voxel size for each input axis given *affine*

The *affine* is the mapping between array (voxel) coordinates and mm (world) coordinates.

The voxel size for the first voxel (array) axis is the distance moved in world coordinates when moving one unit along the first voxel (array) axis. This is the distance between the world coordinate of voxel (0, 0, 0) and the world coordinate of voxel (1, 0, 0). The world coordinate vector of voxel coordinate vector (0, 0, 0) is given by `v0 = affine.dot((0, 0, 0, 1)[:3]`. The world coordinate vector of voxel vector (1, 0, 0) is `v1_ax1 = affine.dot((1, 0, 0, 1))[:3]`. The final 1 in the voxel vectors and the `[:3]` at the end are because the affine works on homogenous coodinates. The translations part of the affine is `trans = affine[:3, 3]`, and the rotations, zooms and shearing part of the affine is `rzs = affine[:3, :3]`. Because of the final 1 in the input voxel vector, `v0 == rzs.dot((0, 0, 0)) + trans`, and `v1_ax1 == rzs.dot((1, 0, 0)) + trans`, and the difference vector is `rzs.dot((0, 0, 0)) - rzs.dot((1, 0, 0)) == rzs.dot((1, 0, 0)) == rzs[:, 0]`. The distance vectors in world coordinates between (0, 0, 0) and (1, 0, 0), (0, 1, 0), (0, 0, 1) are given by `rzs.dot(np.eye(3)) = rzs`. The voxel sizes are the Euclidean lengths of the distance vectors. So, the voxel sizes are the Euclidean lengths of the columns of the affine (excluding the last row and column of the affine).

Parameters

affine [2D array-like] Affine transformation array. Usually shape (4, 4), but can be any 2D array.

Returns

vox_sizes [1D array] Voxel sizes for each input axis of affine. Usually 1D array length 3, but in general has length (N-1) where input *affine* is shape (M, N).

batteryrunners

Battery runner classes and Report classes

These classes / objects are for generic checking / fixing batteries

The `BatteryRunner` class will run a series of checks on a single object.

A check is a callable, of signature `func(obj, fix=False)` which returns a tuple `(obj, Report)` for `func(obj, False)` or `func(obj, True)`, where the obj may be a modified object, or a different object, if `fix==True`.

To run checks only, and return problem report objects:

```
>>> def chk(obj, fix=False): # minimal check
...     return obj, Report()
>>> btrun = BatteryRunner((chk,))
>>> reports = btrun.check_only('a string')
```

To run checks and fixes, returning fixed object and problem report sequence, with possible fix messages:

```
>>> fixed_obj, report_seq = btrun.check_fix('a string')
```

Reports are iterable things, where the elements in the iterations are `Problems`, with attributes `error`, `problem_level`, `problem_msg`, and possibly empty `fix_msg`. The `problem_level` is an integer, giving the level of problem, from 0 (no problem) to 50 (very bad problem). The levels follow the log levels from the logging module (e.g 40 equivalent to "error" level, 50 to "critical"). The `error` can be one of `None` if no error to suggest, or an Exception class that the user might consider raising for this sitation. The `problem_msg` and `fix_msg` are human readable strings that should explain what happened.

More about `checks`

Checks are callables returning objects and reports, like `chk` below, such that:

```
obj, report = chk(obj, fix=False)
obj, report = chk(obj, fix=True)
```

For example, for the Analyze header, we need to check the datatype:

```
def chk_datatype(hdr, fix=True):
    rep = Report(hdr, HeaderDataError)
    code = int(hdr['datatype'])
    try:
        dtype = AnalyzeHeader._data_type_codes.dtype[code]
    except KeyError:
        rep.problem_level = 40
        rep.problem_msg = 'data code not recognized'
    else:
        if dtype.type is np.void:
            rep.problem_level = 40
            rep.problem_msg = 'data code not supported'
        else:
            return hdr, rep
    if fix:
        rep.fix_problem_msg = 'not attempting fix'
    return hdr, rep
```

or the bitpix:

```
def chk_bitpix(hdr, fix=True):
    rep = Report(HeaderDataError)
    code = int(hdr['datatype'])
    try:
        dt = AnalyzeHeader._data_type_codes.dtype[code]
    except KeyError:
        rep.problem_level = 10
        rep.problem_msg = 'no valid datatype to fix bitpix'
        return hdr, rep
    bitpix = dt.itemsize * 8
    if bitpix == hdr['bitpix']:
        return hdr, rep
    rep.problem_level = 10
    rep.problem_msg = 'bitpix does not match datatype')
    if fix:
        hdr['bitpix'] = bitpix # inplace modification
        rep.fix_msg = 'setting bitpix to match datatype'
    return hdr, ret
```

or the pixdims:

```
def chk_pixdims(hdr, fix=True):
    rep = Report(hdr, HeaderDataError)
    if not np.any(hdr['pixdim'][1:4] < 0):
        return hdr, rep
    rep.problem_level = 40
    rep.problem_msg = 'pixdim[1,2,3] should be positive'
    if fix:
        hdr['pixdim'][1:4] = np.abs(hdr['pixdim'][1:4])
        rep.fix_msg = 'setting to abs of pixdim values'
    return hdr, rep
```

BatteryRunner(checks)	Class to run set of checks
Report([error, problem_level, problem_msg, ...])	Initialize report with values

BatteryRunner

class nibabel.batteryrunners.**BatteryRunner**(*checks*)

Bases: object

Class to run set of checks

Initialize instance from sequence of *checks*

Parameters

checks [sequence] sequence of checks, where checks are callables matching signature `obj, rep = chk(obj, fix=False)`. Checks are run in the order they are passed.

Examples

```
>>> def chk(obj, fix=False): # minimal check
...     return obj, Report()
>>> btrun = BatteryRunner((chk,))
```

__init__(*checks*)

Initialize instance from sequence of *checks*

Parameters

checks [sequence] sequence of checks, where checks are callables matching signature `obj, rep = chk(obj, fix=False)`. Checks are run in the order they are passed.

Examples

```
>>> def chk(obj, fix=False): # minimal check
...     return obj, Report()
>>> btrun = BatteryRunner((chk,))
```

check_fix(*obj*)

Run checks, with fixes, on *obj* returning *obj*, reports

Parameters

obj [anything] object on which to run checks, fixes

Returns

obj [anything] possibly modified or replaced *obj*, after fixes

reports [sequence] sequence of reports on checks, fixes

check_only(*obj*)

Run checks on *obj* returning reports

Parameters

obj [anything] object on which to run checks

Returns

reports [sequence] sequence of report objects reporting on result of running checks (without fixes) on *obj*

Report

class nibabel.batteryrunners.**Report**(*error=<class 'Exception'>, problem_level=0, problem_msg='', fix_msg=''*)

Bases: `object`

Initialize report with values

Parameters

error [None or Exception] Error to raise if raising error for this check. If None, no error can be raised for this check (it was probably normal).

problem_level [int] level of problem. From 0 (no problem) to 50 (severe problem). If the report originates from a fix, then this is the level of the problem remaining after the fix. Default is 0

problem_msg [string] String describing problem detected. Default is ‘’

fix_msg [string] String describing any fix applied. Default is ‘’.

Examples

```
>>> rep = Report()
>>> rep.problem_level
0
>>> rep = Report(TypeError, 10)
>>> rep.problem_level
10
```

__init__(*error=<class 'Exception'>*, *problem_level=0*, *problem_msg=''*, *fix_msg=''*)
Initialize report with values

Parameters

error [None or Exception] Error to raise if raising error for this check. If None, no error can be raised for this check (it was probably normal).

problem_level [int] level of problem. From 0 (no problem) to 50 (severe problem). If the report originates from a fix, then this is the level of the problem remaining after the fix. Default is 0

problem_msg [string] String describing problem detected. Default is ‘’

fix_msg [string] String describing any fix applied. Default is ‘’.

Examples

```
>>> rep = Report()
>>> rep.problem_level
0
>>> rep = Report(TypeError, 10)
>>> rep.problem_level
10
```

log_raise(*logger*, *error_level=40*)
Log problem, raise error if problem >= *error_level*

Parameters

logger [log] log object, implementing `log` method

error_level [int, optional] If `self.problem_level` >= *error_level*, raise error

property message
formatted message string, including fix message if present

write_raise(*stream*, *error_level=40*, *log_level=30*)
Write report to *stream*

Parameters

stream [file-like] implementing `write` method

error_level [int, optional] level at which to raise error for problem detected in `self`

log_level [int, optional] Such that if *log_level* is >= `self.problem_level` we write the report to *stream*, otherwise we write nothing.

dft

DICOM filesystem tools

CachingError	error while caching
DFTError	base class for DFT exceptions
InstanceStackError(series, i, si)	bad series of instance numbers
VolumeError	unsupported volume parameter
clear_cache()	
get_studies([base_dir, followlinks])	
update_cache(base_dir[, followlinks])	

CachingError

class nibabel.dft.**CachingError**

Bases: *nibabel.dft.DFTError*

error while caching

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

DFTError

class nibabel.dft.**DFTError**

Bases: `Exception`

base class for DFT exceptions

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

InstanceStackError

class nibabel.dft.**InstanceStackError**(*series*, *i*, *si*)

Bases: *nibabel.dft.DFTError*

bad series of instance numbers

__init__(*series*, *i*, *si*)

Initialize self. See help(type(self)) for accurate signature.

VolumeError

class nibabel.dft.**VolumeError**
Bases: *nibabel.dft.DFTError*

unsupported volume parameter

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

clear_cache

nibabel.dft.**clear_cache**()

get_studies

nibabel.dft.**get_studies**(*base_dir=None*, *followlinks=False*)

update_cache

nibabel.dft.**update_cache**(*base_dir*, *followlinks=False*)

fileholders

Fileholder class

FileHolder([filename, fileobj, pos])	class to contain filename, fileobj and file position
FileHolderError	
copy_file_map(file_map)	Copy mapping of fileholders given by *file_map*

FileHolder

class nibabel.fileholders.**FileHolder**(*filename=None*, *fileobj=None*, *pos=0*)
Bases: object

class to contain filename, fileobj and file position

Initialize FileHolder instance

Parameters

filename [str, optional] filename. Default is None

fileobj [file-like object, optional] Should implement at least 'seek' (for the purposes for this class). Default is None

pos [int, optional] position in filename or fileobject at which to start reading or writing data; defaults to 0

__init__(*filename=None*, *fileobj=None*, *pos=0*)
Initialize FileHolder instance

Parameters

filename [str, optional] filename. Default is None

fileobj [file-like object, optional] Should implement at least 'seek' (for the purposes for this class). Default is None

pos [int, optional] position in filename or fileobject at which to start reading or writing data; defaults to 0

property file_like
Return `self.fileobj` if not None, otherwise `self.filename`

get_prepare_fileobj(**args*, ***kwargs*)
Return fileobj if present, or return fileobj from filename

Set position to that given in self.pos

Parameters

***args** [tuple] positional arguments to file open. Ignored if there is a defined `self.fileobj`. These might include the mode, such as 'rb'

****kwargs** [dict] named arguments to file open. Ignored if there is a defined `self.fileobj`

Returns

fileobj [file-like object] object has position set (via `fileobj.seek()`) to `self.pos`

same_file_as(*other*)
Test if *self* refers to same files / fileobj as *other*

Parameters

other [object] object with *filename* and *fileobj* attributes

Returns

tf [bool] True if *other* has the same filename (or both have None) and the same fileobj (or both have None

FileHolderError

class nibabel.fileholders.**FileHolderError**
Bases: `Exception`

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

copy_file_map

nibabel.fileholders.**copy_file_map**(*file_map*)
Copy mapping of fileholders given by *file_map*

Parameters

file_map [mapping] mapping of `FileHolder` instances

Returns

fm_copy [dict] Copy of *file_map*, using shallow copy of `FileHolders`

filename_parser

Create filename pairs, triplets etc, with expected extensions

TypesFilenamesError	
parse_filename(filename, types_exts, . . . [, . . .])	Split filename into fileroot, extension, trailing suffix; guess type.
splitext_addext(filename[, addexts, match_case])	Split `/pth/fname.ext.gz` into `/pth/fname, .ext, .gz`
types_filenames(template_fname, types_exts)	Return filenames with standard extensions from template name

TypesFilenamesError

class nibabel.filename_parser.**TypesFilenamesError**

Bases: Exception

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

parse_filename

nibabel.filename_parser.**parse_filename**(*filename*, *types_exts*, *trailing_suffixes*, *match_case=False*)

Split filename into fileroot, extension, trailing suffix; guess type.

Parameters

filename [str or os.PathLike] filename in which to search for type extensions

types_exts [sequence of sequences] sequence of (name, extension) str sequences defining type to extension mapping.

trailing_suffixes [sequence of strings] suffixes that should be ignored when looking for extensions

match_case [bool, optional] If True, match case of extensions and trailing suffixes when searching in *filename*, otherwise do case-insensitive match.

Returns

pth [str] path with any matching extensions or trailing suffixes removed

ext [str] If there were any matching extensions, in *types_exts* return that; otherwise return extension derived from `os.path.splitext`.

trailing [str] If there were any matching *trailing_suffixes* return that matching suffix, otherwise ‘’

guessed_type [str] If we found a matching extension in *types_exts* return the corresponding `type`

Examples

```
>>> types_exts = (('t1', 'ext1'),('t2', 'ext2'))
>>> parse_filename('/path/fname.funny', types_exts, ())
('/path/fname', '.funny', None, None)
>>> parse_filename('/path/fnameext2', types_exts, ())
('/path/fname', 'ext2', None, 't2')
>>> parse_filename('/path/fnameext2', types_exts, ('.gz',))
('/path/fname', 'ext2', None, 't2')
>>> parse_filename('/path/fnameext2.gz', types_exts, ('.gz',))
('/path/fname', 'ext2', '.gz', 't2')
```

splitext_addext

nibabel.filename_parser.**splitext_addext**(*filename*, *addexts=('.gz', '.bz2')*, *match_case=False*)

Split `/pth/fname.ext.gz` into `/pth/fname, .ext, .gz`

where `.gz` may be any of passed *addext* trailing suffixes.

Parameters

filename [str or os.PathLike] filename that may end in any or none of *addexts*

match_case [bool, optional] If True, match case of *addexts* and *filename*, otherwise do case-insensitive match.

Returns

froot [str] Root of filename - e.g. `/pth/fname` in example above

ext [str] Extension, where extension is not in *addexts* - e.g. `.ext` in example above

addext [str] Any suffixes appearing in *addext* occuring at end of filename

Examples

```
>>> splitext_addext('fname.ext.gz')
('fname', '.ext', '.gz')
>>> splitext_addext('fname.ext')
('fname', '.ext', '')
>>> splitext_addext('fname.ext.foo', ('.foo', '.bar'))
('fname', '.ext', '.foo')
```

types_filenames

nibabel.filename_parser.**types_filenames**(*template_fname*, *types_exts*, *trailing_suffixes=('.gz', '.bz2')*, *enforce_extensions=True*, *match_case=False*)

Return filenames with standard extensions from template name

The typical case is returning image and header filenames for an Analyze image, that expects an 'image' file type with extension `.img`, and a 'header' file type, with extension `.hdr`.

Parameters

template_fname [str or os.PathLike] template filename from which to construct output dict of filenames, with given *types_exts* type to extension mapping. If `self.enforce_extensions` is True, then filename must have one of the defined extensions from the types list. If `self.enforce_extensions` is False, then the other filenames are guessed at by adding extensions to the base filename. Ignored suffixes (from *trailing_suffixes*) append themselves to the end of all the filenames.

types_exts [sequence of sequences] sequence of (name, extension) str sequences defining type to extension mapping.

trailing_suffixes [sequence of strings, optional] suffixes that should be ignored when looking for extensions - default is `('.gz', '.bz2')`

enforce_extensions [{True, False}, optional] If True, raise an error when attempting to set value to type which has the wrong extension

match_case [bool, optional] If True, match case of extensions and trailing suffixes when searching in *template_fname*, otherwise do case-insensitive match.

Returns

types_fnames [dict] dict with types as keys, and generated filenames as values. The types are given by the first elements of the tuples in *types_exts*.

Examples

```
>>> types_exts = (('t1','.ext1'),('t2', '.ext2'))
>>> tfns = types_filenames('/path/test.ext1', types_exts)
>>> tfns == {'t1': '/path/test.ext1', 't2': '/path/test.ext2'}
True
```

Bare file roots without extensions get them added

```
>>> tfns = types_filenames('/path/test', types_exts)
>>> tfns == {'t1': '/path/test.ext1', 't2': '/path/test.ext2'}
True
```

With enforce_extensions == False, allow first type to have any extension.

```
>>> tfns = types_filenames('/path/test.funny', types_exts,
...                        enforce_extensions=False)
>>> tfns == {'t1': '/path/test.funny', 't2': '/path/test.ext2'}
True
```

fileslice

Utilities for getting array slices out of file-like objects

calc_slicedefs(sliceobj, in_shape, itemsize, ...)	Return parameters for slicing array with *sliceobj* given memory layout
canonical_slicers(sliceobj, shape[, check_inds])	Return canonical version of *sliceobj* for array shape *shape*
fileslice(fileobj, sliceobj, shape, dtype[, ...])	Slice array in *fileobj* using *sliceobj* slicer and array definitions
fill_slicer(slicer, in_len)	Return slice object with Nones filled out to match *in_len*

continues on next page

Table 67 – continued from previous page

is_fancy(sliceobj)	Returns True if sliceobj is attempting fancy indexing
optimize_read_slicers(sliceobj, in_shape, ...)	Calculates slices to read from disk, and apply after reading
optimize_slicer(slicer, dim_len, all_full, ...)	Return maybe modified slice and post-slice slicing for *slicer*
predict_shape(sliceobj, in_shape)	Predict shape of array from slicing array shape *shape* with *sliceobj*
read_segments(fileobj, segments, n_bytes[, lock])	Read *n_bytes* byte data implied by *segments* from *fileobj*
slice2len(slicer, in_len)	Output length after slicing original length *in_len* with *slicer* Parameters ———- slicer : slice object in_len : int
slice2outax(ndim, sliceobj)	Matching output axes for input array ndim *ndim* and slice *sliceobj*
slicers2segments(read_slicers, in_shape, ...)	Get segments from *read_slicers* given *in_shape* and memory steps
strided_scalar(shape[, scalar])	Return array shape *shape* where all entries point to value *scalar*
threshold_heuristic(slicer, dim_len, stride)	Whether to force full axis read or contiguous read of stepped slice

calc_slicedefs

nibabel.fileslice.**calc_slicedefs**(*sliceobj*, *in_shape*, *itemsize*, *offset*, *order*, *heuristic*=<*function threshold_heuristic*>)

Return parameters for slicing array with *sliceobj* given memory layout

Calculate the best combination of skips / (read + discard) to use for reading the data from disk / memory, then generate corresponding *segments*, the disk offsets and read lengths to read the memory. If we have chosen some (read + discard) optimization, then we need to discard the surplus values from the read array using *post_slicers*, a slicing tuple that takes the array as read from a file-like object, and returns the array we want.

Parameters

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`

in_shape [sequence] shape of underlying array to be sliced

itemsize [int] element size in array (in bytes)

offset [int] offset of array data in underlying file or memory buffer

order [{'C', 'F'}] memory layout of underlying array

heuristic [callable, optional] function taking slice object, dim_len, stride length as arguments, returning one of 'full', 'contiguous', None. See *optimize_slicer()* and *threshold_heuristic()*

Returns

segments [list] list of 2 element lists where lists are (offset, length), giving absolute memory offset in bytes and number of bytes to read

read_shape [tuple] shape with which to interpret memory as read from *segments*. Interpreting the memory read from *segments* with this shape, and a dtype, gives an intermediate array - call this `R`

post_slicers [tuple] Any new slicing to be applied to the array `R` after reading via *segments* and reshaping via *read_shape*. Slices are in terms of *read_shape*. If empty, no new slicing to apply

canonical_slicers

nibabel.fileslice.**canonical_slicers**(*sliceobj*, *shape*, *check_inds=True*)

Return canonical version of *sliceobj* for array shape *shape*

sliceobj is a slicer for an array `A` implied by *shape*.

- Expand *sliceobj* with `slice(None)` to add any missing (implied) axes in *sliceobj*
- Find any slicers in *sliceobj* that do a full axis slice and replace by `slice(None)`
- Replace any floating point values for slicing with integers
- Replace negative integer slice values with equivalent positive integers.

Does not handle fancy indexing (indexing with arrays or array-like indices)

Parameters

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`

shape [sequence] shape of array that will be indexed by *sliceobj*

check_inds [{True, False}, optional] Whether to check if integer indices are out of bounds

Returns

can_slicers [tuple] version of *sliceobj* for which Ellipses have been expanded, missing (implied) dimensions have been appended, and slice objects equivalent to `slice(None)` have been replaced by `slice(None)`, integer axes have been checked, and negative indices set to positive equivalent

fileslice

nibabel.fileslice.**fileslice**(*fileobj*, *sliceobj*, *shape*, *dtype*, *offset=0*, *order='C'*, *heuristic=<function threshold_heuristic>*, *lock=None*)

Slice array in *fileobj* using *sliceobj* slicer and array definitions

fileobj contains the contiguous binary data for an array `A` of shape, dtype, memory layout *shape*, *dtype*, *order*, with the binary data starting at file offset *offset*.

Our job is to return the sliced array `A[sliceobj]` in the most efficient way in terms of memory and time.

Sometimes it will be quicker to read memory that we will later throw away, to save time we might lose doing short seeks on *fileobj*. Call these alternatives: (read + discard); and skip. This routine guesses when to (read+discard) or skip using the callable *heuristic*, with a default using a hard threshold for the memory gap large enough to prefer a skip.

Parameters

fileobj [file-like object] file-like object, opened for reading in binary mode. Implements `read` and `seek`.

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`.

shape [sequence] shape of full array inside *fileobj*.

dtype [dtype specifier] dtype of array inside *fileobj*, or input to `numpy.dtype` to specify array dtype.

offset [int, optional] offset of array data within *fileobj*

order [{'C', 'F'}, optional] memory layout of array in *fileobj*.

heuristic [callable, optional] function taking slice object, axis length, stride length as arguments, returning one of 'full', 'contiguous', None. See *optimize_slicer()* and see *threshold_heuristic()* for an example.

lock [{None, threading.Lock, lock-like} optional] If provided, used to ensure that paired calls to `seek` and `read` cannot be interrupted by another thread accessing the same `fileobj`. Each thread which accesses the same file via `read_segments` must share a lock in order to ensure that the file access is thread-safe. A lock does not need to be provided for single-threaded access. The default value (`None`) results in a lock-like object (a `_NullLock`) which does not do anything.

Returns

sliced_arr [array] Array in *fileobj* as sliced with *sliceobj*

fill_slicer

`nibabel.fileslice.`**`fill_slicer`**(*slicer*, *in_len*)

Return slice object with Nones filled out to match *in_len*

Also fixes too large stop / start values according to slice() slicing rules.

The returned slicer can have a None as *slicer.stop* if *slicer.step* is negative and the input *slicer.stop* is None. This is because we can't represent the `stop` as an integer, because -1 has a different meaning.

Parameters

slicer [slice object]

in_len [int] length of axis on which *slicer* will be applied

Returns

can_slicer [slice object] slice with start, stop, step set to explicit values, with the exception of `stop` for negative step, which is None for the case of slicing down through the first element

is_fancy

`nibabel.fileslice.`**`is_fancy`**(*sliceobj*)

Returns True if sliceobj is attempting fancy indexing

Parameters

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`

Returns

tf: bool True if sliceobj represents fancy indexing, False for basic indexing

optimize_read_slicers

nibabel.fileslice.**optimize_read_slicers**(*sliceobj*, *in_shape*, *itemsize*, *heuristic*)

Calculates slices to read from disk, and apply after reading

Parameters

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`. Can be assumed to be canonical in the sense of `canonical_slicers`

in_shape [sequence] shape of underlying array to be sliced. Array for *in_shape* assumed to be already in 'F' order. Reorder shape / sliceobj for slicing a 'C' array before passing to this function.

itemsize [int] element size in array (bytes)

heuristic [callable] function taking slice object, axis length, and stride length as arguments, returning one of 'full', 'contiguous', None. See *optimize_slicer()*; see *threshold_heuristic()* for an example.

Returns

read_slicers [tuple] *sliceobj* maybe rephrased to fill out dimensions that are better read from disk and later trimmed to their original size with *post_slicers*. *read_slicers* implies a block of memory to be read from disk. The actual disk positions come from *slicers2segments* run over *read_slicers*. Includes any `newaxis` dimensions in *sliceobj*

post_slicers [tuple] Any new slicing to be applied to the read array after reading. The *post_slicers* discard any memory that we read to save time, but that we don't need for the slice. Include any `newaxis` dimension added by *sliceobj*

optimize_slicer

nibabel.fileslice.**optimize_slicer**(*slicer*, *dim_len*, *all_full*, *is_slowest*, *stride*, *heuristic=<function threshold_heuristic>*)

Return maybe modified slice and post-slice slicing for *slicer*

Parameters

slicer [slice object or int]

dim_len [int] length of axis along which to slice

all_full [bool] Whether dimensions up until now have been full (all elements)

is_slowest [bool] Whether this dimension is the slowest changing in memory / on disk

stride [int] size of one step along this axis

heuristic [callable, optional] function taking slice object, dim_len, stride length as arguments, returning one of 'full', 'contiguous', None. See *threshold_heuristic()* for an example.

Returns

to_read [slice object or int] maybe modified slice based on *slicer* expressing what data should be read from an underlying file or buffer. *to_read* must always have positive `step` (because we don't want to go backwards in the buffer / file)

post_slice [slice object] slice to be applied after array has been read. Applies any transformations in *slicer* that have not been applied in *to_read*. If axis will be dropped by *to_read* slicing, so no slicing would make sense, return string `dropped`

Notes

This is the heart of the algorithm for making segments from slice objects.

A contiguous slice is a slice with `slice.step in (1, -1)`

A full slice is a continuous slice returning all elements.

The main question we have to ask is whether we should transform *to_read*, *post_slice* to prefer a full read and partial slice. We only do this in the case of all_full==True. In this case we might benefit from reading a continuous chunk of data even if the slice is not continuous, or reading all the data even if the slice is not full. Apply a heuristic *heuristic* to decide whether to do this, and adapt *to_read* and *post_slice* slice accordingly.

Otherwise (apart from constraint to be positive) return *to_read* unaltered and *post_slice* as `slice(None)`

predict_shape

`nibabel.fileslice.`**`predict_shape`**(*sliceobj*, *in_shape*)

Predict shape of array from slicing array shape *shape* with *sliceobj*

Parameters

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`

in_shape [sequence] shape of array that could be sliced by *sliceobj*

Returns

out_shape [tuple] predicted shape arising from slicing array shape *in_shape* with *sliceobj*

read_segments

`nibabel.fileslice.`**`read_segments`**(*fileobj*, *segments*, *n_bytes*, *lock=None*)

Read *n_bytes* byte data implied by *segments* from *fileobj*

Parameters

fileobj [file-like object] Implements *seek* and *read*

segments [sequence] list of 2 sequences where sequences are (offset, length), giving absolute file offset in bytes and number of bytes to read

n_bytes [int] total number of bytes that will be read

lock [{None, threading.Lock, lock-like} optional] If provided, used to ensure that paired calls to `seek` and `read` cannot be interrupted by another thread accessing the same `fileobj`. Each thread which accesses the same file via `read_segments` must share a lock in order to ensure that the file access is thread-safe. A lock does not need to be provided for single-threaded access. The default value (`None`) results in a lock-like object (a `_NullLock`) which does not do anything.

Returns

buffer [buffer object] object implementing buffer protocol, such as byte string or ndarray or mmap or ctypes `c_char_array`

slice2len

nibabel.fileslice.**slice2len**(*slicer*, *in_len*)

Output length after slicing original length *in_len* with *slicer* Parameters ———- slicer : slice object in_len : int

Returns

out_len [int] Length after slicing

Notes

Returns same as `len(np.arange(in_len)[slicer])`

slice2outax

nibabel.fileslice.**slice2outax**(*ndim*, *sliceobj*)

Matching output axes for input array ndim *ndim* and slice *sliceobj*

Parameters

ndim [int] number of axes in input array

sliceobj [object] something that can be used to slice an array as in `arr[sliceobj]`

Returns

out_ax_inds [tuple] Say ``A` is a (pretend) input array of `ndim` dimensions. Say ``B = A[sliceobj]``. *out_ax_inds* has one value per axis in `A` giving corresponding axis in `B`.

slicers2segments

nibabel.fileslice.**slicers2segments**(*read_slicers*, *in_shape*, *offset*, *itemsize*)

Get segments from *read_slicers* given *in_shape* and memory steps

Parameters

read_slicers [object] something that can be used to slice an array as in `arr[sliceobj]` Slice objects can by be assumed canonical as in `canonical_slicers`, and positive as in `_positive_slice`

in_shape [sequence] shape of underlying array on disk before reading

offset [int] offset of array data in underlying file or memory buffer

itemsize [int] element size in array (in bytes)

Returns

segments [list] list of 2 element lists where lists are [offset, length], giving absolute memory offset in bytes and number of bytes to read

strided_scalar

nibabel.fileslice.**strided_scalar**(*shape*, *scalar=0.0*)

Return array shape *shape* where all entries point to value *scalar*

Parameters

shape [sequence] Shape of output array.

scalar [scalar] Scalar value with which to fill array.

Returns

strided_arr [array] Array of shape *shape* for which all values == *scalar*, built by setting all strides of *strided_arr* to 0, so the scalar is broadcast out to the full array *shape*. *strided_arr* is flagged as not *writeable*.

The array is set read-only to avoid a numpy error when broadcasting - see https://github.com/numpy/numpy/issues/6491

threshold_heuristic

nibabel.fileslice.**threshold_heuristic**(*slicer*, *dim_len*, *stride*, *skip_thresh=256*)

Whether to force full axis read or contiguous read of stepped slice

Allows *fileslice()* to sometimes read memory that it will throw away in order to get maximum speed. In other words, trade memory for fewer disk reads.

Parameters

slicer [slice object, or int] If slice, can be assumed to be full as in `fill_slicer`

dim_len [int] length of axis being sliced

stride [int] memory distance between elements on this axis

skip_thresh [int, optional] Memory gap threshold in bytes above which to prefer skipping memory rather than reading it and later discarding.

Returns

action [{'full', 'contiguous', None}] Gives the suggested optimization for reading the data

- 'full' - read whole axis
- 'contiguous' - read all elements between start and stop
- None - read only memory needed for output

Notes

Let's say we are in the middle of reading a file at the start of some memory length B bytes. We don't need the memory, and we are considering whether to read it anyway (then throw it away) (READ) or stop reading, skip B bytes and restart reading from there (SKIP).

After trying some more fancy algorithms, a hard threshold (*skip_thresh*) for the maximum skip distance seemed to work well, as measured by times on `nibabel.benchmarks.bench_fileslice`

onetime

Descriptor support for NIPY.

Utilities to support special Python descriptors [1,2], in particular the use of a useful pattern for properties we call 'one time properties'. These are object attributes which are declared as properties, but become regular attributes once they've been read the first time. They can thus be evaluated later in the object's life cycle, but once evaluated they become normal, static attributes with no function call overhead on access or any other constraints.

A special ResetMixin class is provided to add a .reset() method to users who may want to have their objects capable of resetting these computed properties to their 'untriggered' state.

References

[1] How-To Guide for Descriptors, Raymond Hettinger. https://docs.python.org/howto/descriptor.html

[2] Python data model, https://docs.python.org/reference/datamodel.html

OneTimeProperty(func)	A descriptor to make special properties that become normal attributes.
ResetMixin()	A Mixin class to add a .reset() method to users of OneTimeProperty.
auto_attr(func)	Decorator to create OneTimeProperty attributes.
setattr_on_read(func)	Decorator to create OneTimeProperty attributes.

OneTimeProperty

class nibabel.onetime.**OneTimeProperty**(*func*)

Bases: object

A descriptor to make special properties that become normal attributes.

This is meant to be used mostly by the auto_attr decorator in this module.

Create a OneTimeProperty instance.

Parameters

func [method] The method that will be called the first time to compute a value. Afterwards, the method's name will be a standard attribute holding the value of this computation.

__init__(*func*)

Create a OneTimeProperty instance.

Parameters

func [method] The method that will be called the first time to compute a value. Afterwards, the method's name will be a standard attribute holding the value of this computation.

ResetMixin

class nibabel.onetime.**ResetMixin**

Bases: object

A Mixin class to add a .reset() method to users of OneTimeProperty.

By default, auto attributes once computed, become static. If they happen to depend on other parts of an object and those parts change, their values may now be invalid.

This class offers a .reset() method that users can call *explicitly* when they know the state of their objects may have changed and they want to ensure that *all* their special attributes should be invalidated. Once reset() is called, all their auto attributes are reset to their OneTimeProperty descriptors, and their accessor functions will be triggered again.

Warning: If a class has a set of attributes that are OneTimeProperty, but that can be initialized from any one of them, do NOT use this mixin! For instance, UniformTimeSeries can be initialized with only sampling_rate and t0, sampling_interval and time are auto-computed. But if you were to reset() a UniformTimeSeries, it would lose all 4, and there would be then no way to break the circular dependency chains.

If this becomes a problem in practice (for our analyzer objects it isn't, as they don't have the above pattern), we can extend reset() to check for a _no_reset set of names in the instance which are meant to be kept protected. But for now this is NOT done, so caveat emptor.

Examples

```
>>> class A(ResetMixin):
...     def __init__(self,x=1.0):
...         self.x = x
...
...     @auto_attr
...     def y(self):
...         print('*** y computation executed ***')
...         return self.x / 2.0
...
```

```
>>> a = A(10)
```

About to access y twice, the second time no computation is done: >>> a.y *** y computation executed *** 5.0 >>> a.y 5.0

Changing x >>> a.x = 20

a.y doesn't change to 10, since it is a static attribute: >>> a.y 5.0

We now reset a, and this will then force all auto attributes to recompute the next time we access them: >>> a.reset()

About to access y twice again after reset(): >>> a.y *** y computation executed *** 10.0 >>> a.y 10.0

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

reset()

Reset all OneTimeProperty attributes that may have fired already.

auto_attr

nibabel.onetime.**auto_attr**(*func*)

Decorator to create OneTimeProperty attributes.

Parameters

func [method] The method that will be called the first time to compute a value. Afterwards, the method's name will be a standard attribute holding the value of this computation.

Examples

```
>>> class MagicProp(object):
...     @auto_attr
...     def a(self):
...         return 99
...
>>> x = MagicProp()
>>> 'a' in x.__dict__
False
>>> x.a
99
>>> 'a' in x.__dict__
True
```

setattr_on_read

nibabel.onetime.**setattr_on_read**(*func*)

Decorator to create OneTimeProperty attributes.

setattr_on_read has been renamed to auto_attr. Please use nibabel.onetime.auto_attr

- deprecated from version: 3.2
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 5.0

Parameters

func [method] The method that will be called the first time to compute a value. Afterwards, the method's name will be a standard attribute holding the value of this computation.

Examples

```
>>> class MagicProp(object):
...     @auto_attr
...     def a(self):
...         return 99
...
>>> x = MagicProp()
>>> 'a' in x.__dict__
False
>>> x.a
99
>>> 'a' in x.__dict__
True
```

openers

Context manager openers for various fileobject types

ImageOpener(fileish, *args, **kwargs)	Opener-type class to collect extra compressed extensions
Opener(fileish, *args, **kwargs)	Class to accept, maybe open, and context-manage file-likes / filenames

ImageOpener

class nibabel.openers.**ImageOpener**(*fileish*, **args*, ***kwargs*)

Bases: *nibabel.openers.Opener*

Opener-type class to collect extra compressed extensions

A trivial sub-class of opener to which image classes can add extra extensions with custom openers, such as compressed openers.

To add an extension, add a line to the class definition (not __init__):

ImageOpener.compress_ext_map[ext] = func_def

`ext` is a file extension beginning with '.' and should be included in the image class's `valid_exts` tuple.

`func_def` is a *(function, (args,))* tuple, where *function accepts a filename as the first parameter, and `args* defines the other arguments that *function* accepts. These arguments must be any (unordered) subset of *mode*, *compresslevel*, and *buffering*.

__init__(*fileish*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

compress_ext_map = {'.gz': (<function _gzip_open>, ('mode', 'compresslevel', 'keep_op

Opener

class nibabel.openers.**Opener**(*fileish*, **args*, ***kwargs*)

Bases: `object`

Class to accept, maybe open, and context-manage file-likes / filenames

Provides context manager to close files that the constructor opened for you.

Parameters

fileish [str or file-like] if str, then open with suitable opening method. If file-like, accept as is

***args** [positional arguments] passed to opening method when *fileish* is str. `mode`, if not specified, is *rb*. `compresslevel`, if relevant, and not specified, is set from class variable `default_compresslevel`. `keep_open`, if relevant, and not specified, is `False`.

****kwargs** [keyword arguments] passed to opening method when *fileish* is str. Change of defaults as for *args

__init__(*fileish*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

bz2_def = (<class 'bz2.BZ2File'>, ('mode', 'buffering', 'compresslevel'))

close(**args*, ***kwargs*)

close_if_mine()
Close `self.fobj` iff we opened it in the constructor

property closed

compress_ext_icase = True
whether to ignore case looking for compression extensions

compress_ext_map = {'.gz': (<function _gzip_open>, ('mode', 'compresslevel', 'keep_op

default_compresslevel = 1
default compression level when writing gz and bz2 files

fileno()

gz_def = (<function _gzip_open>, ('mode', 'compresslevel', 'keep_open'))

property mode

property name
Return `self.fobj.name` or self._name if not present

self._name will be None if object was created with a fileobj, otherwise it will be the filename.

read(**args*, ***kwargs*)

readinto(**args*, ***kwargs*)

seek(**args*, ***kwargs*)

tell(**args*, ***kwargs*)

write(**args*, ***kwargs*)

optpkg

Routines to support optional packages

optional_package(name[, trip_msg, min_version])		Return package-like thing and module setup for package *name*

optional_package

nibabel.optpkg.**optional_package**(*name*, *trip_msg=None*, *min_version=None*)
Return package-like thing and module setup for package *name*

Parameters

name [str] package name

trip_msg [None or str] message to give when someone tries to use the return package, but we could not import it at an acceptable version, and have returned a TripWire object instead. Default message if None.

min_version [None or str or LooseVersion or callable] If None, do not specify a minimum version. If str, convert to a *distutils.version.LooseVersion*. If str or LooseVersion` compare to version of package *name* with `min_version <= pkg.__version__`. If callable, accepts imported `pkg` as argument, and returns value of callable is True for acceptable package versions, False otherwise.

Returns

pkg_like [module or `TripWire` instance] If we can import the package, return it. Otherwise return an object raising an error when accessed

have_pkg [bool] True if import for package was successful, false otherwise

module_setup [function] callable usually set as `setup_module` in calling namespace, to allow skipping tests.

Examples

Typical use would be something like this at the top of a module using an optional package:

```
>>> from nibabel.optpkg import optional_package
>>> pkg, have_pkg, setup_module = optional_package('not_a_package')
```

Of course in this case the package doesn't exist, and so, in the module:

```
>>> have_pkg
False
```

and

```
>>> pkg.some_function()
Traceback (most recent call last):
    ...
TripWireError: We need package not_a_package for these functions,
    but ``import not_a_package`` raised an ImportError
```

If the module does exist - we get the module

```
>>> pkg, _, _ = optional_package('os')
>>> hasattr(pkg, 'path')
True
```

Or a submodule if that's what we asked for

```
>>> subpkg, _, _ = optional_package('os.path')
>>> hasattr(subpkg, 'dirname')
True
```

rstutils

ReStructured Text utilities

- Make ReST table given array of values

rst_table(cell_values[, row_names, ...])	Return string for ReST table with entries *cell_values*

rst_table

nibabel.rstutils.**rst_table**(*cell_values*, *row_names=None*, *col_names=None*, *title=''*, *val_fmt='{0:5.2f}'*, *format_chars=None*)

Return string for ReST table with entries *cell_values*

Parameters

cell_values [(R, C) array-like] At least 2D. Can be greater than 2D, in which case you should adapt the *val_fmt* to deal with the multiple entries that will go in each cell

row_names [None or (R,) length sequence, optional] Row names. If None, use `row[0]` etc.

col_names [None or (C,) length sequence, optional] Column names. If None, use `col[0]` etc.

title [str, optional] Title for table. Add as heading above table

val_fmt [str, optional] Format string using string `format` method mini-language. Converts the result of `cell_values[r, c]` to a string to make the cell contents. Default assumes a floating point value in a 2D *cell_values*.

format_chars [None or dict, optional] With keys 'down', 'along', 'thick_long', 'cross' and 'title_heading'. Values are characters for: lines going down; lines going along; thick lines along; two lines crossing; and the title overline / underline. All missing values filled with rst defaults.

Returns

table_str [str] Multiline string with ascii table, suitable for printing

tmpdirs

Contexts for *with* statement providing temporary directories

InGivenDirectory([path])	Change directory to given directory for duration of `with` block
InTemporaryDirectory([suffix, prefix, dir])	Create, return, and change directory to a temporary directory
TemporaryDirectory([suffix, prefix, dir])	Create and return a temporary directory.

InGivenDirectory

class nibabel.tmpdirs.**InGivenDirectory**(*path=None*)

Bases: `object`

Change directory to given directory for duration of `with` block

Useful when you want to use *InTemporaryDirectory* for the final test, but you are still debugging. For example, you may want to do this in the end:

```
>>> with InTemporaryDirectory() as tmpdir:
...     # do something complicated which might break
...     pass
```

But indeed the complicated thing does break, and meanwhile the `InTemporaryDirectory` context manager wiped out the directory with the temporary files that you wanted for debugging. So, while debugging, you replace with something like:

```
>>> with InGivenDirectory() as tmpdir: # Use working directory by default
...     # do something complicated which might break
...     pass
```

You can then look at the temporary file outputs to debug what is happening, fix, and finally replace `InGivenDirectory` with `InTemporaryDirectory` again.

Initialize directory context manager

Parameters

path [None or str, optional] path to change directory to, for duration of `with` block. Defaults to `os.getcwd()` if None

__init__(*path=None*)
Initialize directory context manager

Parameters

path [None or str, optional] path to change directory to, for duration of `with` block. Defaults to `os.getcwd()` if None

InTemporaryDirectory

class nibabel.tmpdirs.**InTemporaryDirectory**(*suffix=''*, *prefix='tmp'*, *dir=None*)
Bases: *nibabel.tmpdirs.TemporaryDirectory*

Create, return, and change directory to a temporary directory

Examples

```
>>> import os
>>> my_cwd = os.getcwd()
>>> with InTemporaryDirectory() as tmpdir:
...     _ = open('test.txt', 'wt').write('some text')
...     assert os.path.isfile('test.txt')
...     assert os.path.isfile(os.path.join(tmpdir, 'test.txt'))
>>> os.path.exists(tmpdir)
False
>>> os.getcwd() == my_cwd
True
```

__init__(*suffix=''*, *prefix='tmp'*, *dir=None*)
Initialize self. See help(type(self)) for accurate signature.

TemporaryDirectory

class nibabel.tmpdirs.**TemporaryDirectory**(*suffix=''*, *prefix='tmp'*, *dir=None*)
Bases: `object`

Create and return a temporary directory. This has the same behavior as mkdtemp but can be used as a context manager.

Upon exiting the context, the directory and everthing contained in it are removed.

Examples

```
>>> import os
>>> with TemporaryDirectory() as tmpdir:
...     fname = os.path.join(tmpdir, 'example_file.txt')
...     with open(fname, 'wt') as fobj:
...         _ = fobj.write('a string\n')
>>> os.path.exists(tmpdir)
False
```

__init__(*suffix=''*, *prefix='tmp'*, *dir=None*)
Initialize self. See help(type(self)) for accurate signature.

cleanup()

tripwire

Class to raise error for missing modules or other misfortunes

TripWire(msg)	Class raising error if used
TripWireError	Exception if trying to use TripWire object
is_tripwire(obj)	Returns True if *obj* appears to be a TripWire object

TripWire

class nibabel.tripwire.**TripWire**(*msg*)
Bases: object

Class raising error if used

Standard use is to proxy modules that we could not import

Examples

```
>>> a_module = TripWire('We do not have a_module')
>>> a_module.do_silly_thing('with silly string')
Traceback (most recent call last):
    ...
TripWireError: We do not have a_module
```

__init__(*msg*)
Initialize self. See help(type(self)) for accurate signature.

TripWireError

class nibabel.tripwire.**TripWireError**
Bases: AttributeError

Exception if trying to use TripWire object

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

is_tripwire

nibabel.tripwire.**is_tripwire**(*obj*)
Returns True if *obj* appears to be a TripWire object

Examples

```
>>> is_tripwire(object())
False
>>> is_tripwire(TripWire('some message'))
True
```

wrapstruct

Class to wrap numpy structured array

wrapstruct

The *WrapStruct* class is a wrapper around a numpy structured array type.

It implements:

- Mappingness from the underlying structured array fields
- from_fileobj, write_to methods to read and write data to fileobj
- A mechanism for setting checks and fixes to the data on object creation
- Endianness guessing, and on-the-fly swapping

The *LabeledWrapStruct* subclass adds:

- A pretty printing mechanism whereby field values can be displayed as corresponding strings (see *LabeledWrapStruct.get_value_label()* and LabeledWrapStruct.__str__())

Mappingness

You can access and set fields of the contained structarr using standard __getitem__ / __setitem__ syntax:

wrapped['field'] = 10

Wrapped structures also implement general mappingness:

wrapped.keys() wrapped.items() wrapped.values()

Properties:

```
.endianness (read only)
.binaryblock (read only)
.structarr (read only)
```

Methods:

```
.as_byteswapped(endianness)
.check_fix()
.__str__
.__eq__
.__ne__
.get_value_label(name)
```

Class methods:

```
.diagnose_binaryblock
.as_byteswapped(endianness)
.write_to(fileobj)
.from_fileobj(fileobj)
.default_structarr() - return default structured array
.guessed_endian(structarr) - return guessed endian code from this structarr
```

Class variables: template_dtype - native endian version of dtype for contained structarr

Consistency checks

We have a file, and we would like information as to whether there are any problems with the binary data in this file, and whether they are fixable. `WrapStruct` can hold checks for internal consistency of the contained data:

```
wrapped = WrapStruct.from_fileobj(open('myfile.bin'), check=False)
dx_result = WrapStruct.diagnose_binaryblock(wrapped.binaryblock)
```

This will run all known checks, with no fixes, returning a string with diagnostic output. See below for the `check=False` flag.

In creating a `WrapStruct` object, we often want to check the consistency of the contained data. The checks can test for problems of various levels of severity. If the problem is severe enough, it should raise an Error. So, with data that is consistent - no error:

```
wrapped = WrapStruct.from_fileobj(good_fileobj)
```

whereas:

```
wrapped = WrapStruct.from_fileobj(bad_fileobj)
```

would raise some error, with output to logging (see below).

If we want the created object, come what may:

```
hdr = WrapStruct.from_fileobj(bad_fileobj, check=False)
```

We set the error level (the level of problem that the `check=True` versions will accept as OK) from global defaults:

```
import nibabel as nib
nib.imageglobals.error_level = 30
```

The same for logging:

```
nib.imageglobals.logger = logger
```

LabeledWrapStruct([binaryblock, endianness, ...])	A WrapStruct with some fields having value labels for printing etc
WrapStruct([binaryblock, endianness, check])	Initialize WrapStruct from binary data block
WrapStructError	

LabeledWrapStruct

class nibabel.wrapstruct.**LabeledWrapStruct**(*binaryblock=None*, *endianness=None*, *check=True*)

Bases: *nibabel.wrapstruct.WrapStruct*

A WrapStruct with some fields having value labels for printing etc

Initialize WrapStruct from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into object. By default, None, in which case we insert the default empty block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of binary data in initialization. Default is True.

Examples

```
>>> wstr1 = WrapStruct() # a default structure
>>> wstr1.endianness == native_code
True
>>> wstr1['integer']
array(0, dtype=int16)
>>> wstr1['integer'] = 1
>>> wstr1['integer']
array(1, dtype=int16)
```

__init__(*binaryblock=None*, *endianness=None*, *check=True*)

Initialize WrapStruct from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into object. By default, None, in which case we insert the default empty block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binary-block. If None, guess endianness from the data.

check [bool, optional] Whether to check content of binary data in initialization. Default is True.

Examples

```
>>> wstr1 = WrapStruct() # a default structure
>>> wstr1.endianness == native_code
True
>>> wstr1['integer']
array(0, dtype=int16)
>>> wstr1['integer'] = 1
>>> wstr1['integer']
array(1, dtype=int16)
```

get_value_label (*fieldname*)
Returns label for coded field

A coded field is an int field containing codes that stand for discrete values that also have string labels.

Parameters

fieldname [str] name of header field to get label for

Returns

label [str] label for code value in header field *fieldname*

Raises

ValueError if field is not coded.

Examples

```
>>> from nibabel.volumeutils import Recoder
>>> recoder = Recoder(((1, 'one'), (2, 'two')), ('code', 'label'))
>>> class C(LabeledWrapStruct):
...     template_dtype = np.dtype([('datatype', 'i2')])
...     _field_recoders = dict(datatype = recoder)
>>> hdr  = C()
>>> hdr.get_value_label('datatype')
'<unknown code 0>'
>>> hdr['datatype'] = 2
>>> hdr.get_value_label('datatype')
'two'
```

WrapStruct

class nibabel.wrapstruct.**WrapStruct**(*binaryblock=None*, *endianness=None*, *check=True*)

Bases: object

Initialize WrapStruct from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into object. By default, None, in which case we insert the default empty block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of binary data in initialization. Default is True.

Examples

```
>>> wstr1 = WrapStruct() # a default structure
>>> wstr1.endianness == native_code
True
>>> wstr1['integer']
array(0, dtype=int16)
>>> wstr1['integer'] = 1
>>> wstr1['integer']
array(1, dtype=int16)
```

__init__(*binaryblock=None*, *endianness=None*, *check=True*)

Initialize WrapStruct from binary data block

Parameters

binaryblock [{None, string} optional] binary block to set into object. By default, None, in which case we insert the default empty block

endianness [{None, '<','>', other endian code} string, optional] endianness of the binaryblock. If None, guess endianness from the data.

check [bool, optional] Whether to check content of binary data in initialization. Default is True.

Examples

```
>>> wstr1 = WrapStruct() # a default structure
>>> wstr1.endianness == native_code
True
>>> wstr1['integer']
array(0, dtype=int16)
>>> wstr1['integer'] = 1
>>> wstr1['integer']
array(1, dtype=int16)
```

as_byteswapped(*endianness=None*)

return new byteswapped object with given endianness

Guaranteed to make a copy even if endianness is the same as the current endianness.

Parameters

endianness [None or string, optional] endian code to which to swap. None means swap from current endianness, and is the default

Returns

wstr [`WrapStruct`] `WrapStruct` object with given endianness

Examples

```
>>> wstr = WrapStruct()
>>> wstr.endianness == native_code
True
>>> bs_wstr = wstr.as_byteswapped()
>>> bs_wstr.endianness == swapped_code
True
>>> bs_wstr = wstr.as_byteswapped(swapped_code)
>>> bs_wstr.endianness == swapped_code
True
>>> bs_wstr is wstr
False
>>> bs_wstr == wstr
True
```

If you write to the resulting byteswapped data, it does not change the original.

```
>>> bs_wstr['integer'] = 3
>>> bs_wstr == wstr
False
```

If you swap to the same endianness, it returns a copy

```
>>> nbs_wstr = wstr.as_byteswapped(native_code)
>>> nbs_wstr.endianness == native_code
True
>>> nbs_wstr is wstr
False
```

property binaryblock

binary block of data as string

Returns

binaryblock [string] string giving binary data block

Examples

```
>>> # Make default empty structure
>>> wstr = WrapStruct()
>>> len(wstr.binaryblock)
2
```

check_fix(*logger=None*, *error_level=None*)

Check structured data with checks

Parameters

logger [None or logging.Logger]

error_level [None or int] Level of error severity at which to raise error. Any error of severity >= *error_level* will cause an exception.

copy()
Return copy of structure

```
>>> wstr = WrapStruct()
>>> wstr['integer'] = 3
>>> wstr2 = wstr.copy()
>>> wstr2 is wstr
False
>>> wstr2['integer']
array(3, dtype=int16)
```

classmethod default_structarr(*endianness=None*)
Return structured array for default structure with given endianness

classmethod diagnose_binaryblock(*binaryblock*, *endianness=None*)
Run checks over binary data, return string

property endianness
endian code of binary data

The endianness code gives the current byte order interpretation of the binary data.

Notes

Endianness gives endian interpretation of binary data. It is read only because the only common use case is to set the endianness on initialization, or occasionally byteswapping the data - but this is done via the as_byteswapped method

Examples

```
>>> wstr = WrapStruct()
>>> code = wstr.endianness
>>> code == native_code
True
```

classmethod from_fileobj(*fileobj*, *endianness=None*, *check=True*)
Return read structure with given or guessed endiancode

Parameters

fileobj [file-like object] Needs to implement `read` method

endianness [None or endian code, optional] Code specifying endianness of read data

Returns

wstr [WrapStruct object] WrapStruct object initialized from data in fileobj

get(*k*, *d=None*)
Return value for the key k if present or d otherwise

classmethod guessed_endian(*mapping*)
Guess intended endianness from mapping-like `mapping`

Parameters

wstr [mapping-like] Something implementing a mapping. We will guess the endianness from looking at the field values

Returns

endianness [{'<', '>'}] Guessed endianness of binary data in `wstr`

items()
Return items from structured data

keys()
Return keys from structured data

property structarr
Structured data, with data fields

Examples

```
>>> wstr1 = WrapStruct() # with default data
>>> an_int = wstr1.structarr['integer']
>>> wstr1.structarr = None
Traceback (most recent call last):
   ...
AttributeError: can't set attribute
```

template_dtype = dtype([('integer', '<i2')])

values()
Return values from structured data

write_to(*fileobj*)
Write structure to fileobj

Write starts at fileobj current file position.

Parameters

fileobj [file-like object] Should implement `write` method

Returns

None

Examples

```
>>> wstr = WrapStruct()
>>> from io import BytesIO
>>> str_io = BytesIO()
>>> wstr.write_to(str_io)
>>> wstr.binaryblock == str_io.getvalue()
True
```

WrapStructError

class nibabel.wrapstruct.**WrapStructError**
Bases: Exception

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

10.5.7 Alphabetical API reference

API Reference

_h5py_compat

—

_version

Git implementation of _version.py.

NotThisMethod	Exception raised if a method is not valid for the current scenario.
VersioneerConfig()	Container for Versioneer configuration parameters.
get_config()	Create, populate and return the VersioneerConfig() object.
get_keywords()	Get the keywords needed to look up the version information.
get_versions()	Get version information or return default if unable to do so.
git_get_keywords(versionfile_abs)	Extract version information from the given file.
git_pieces_from_vcs(tag_prefix, root, verbose)	Get version from 'git describe' in the root of the source tree.
git_versions_from_keywords(keywords, ...)	Get version information from git keywords.
plus_or_dot(pieces)	Return a + if we don't already have one, else return a .
register_vcs_handler(vcs, method)	Create decorator to mark a method as the handler of a VCS.
render(pieces, style)	Render the given version pieces into the requested style.
render_git_describe(pieces)	TAG[-DISTANCE-gHEX][-dirty].
render_git_describe_long(pieces)	TAG-DISTANCE-gHEX[-dirty].
render_pep440(pieces)	Build up version string, with post-release "local version identifier".
render_pep440_old(pieces)	TAG[.postDISTANCE[.dev0]] .
render_pep440_post(pieces)	TAG[.postDISTANCE[.dev0]+gHEX] .
render_pep440_pre(pieces)	TAG[.post0.devDISTANCE] – No -dirty.
run_command(commands, args[, cwd, verbose, ...])	Call the given command(s).
versions_from_parentdir(parentdir_prefix, ...)	Try to determine the version from the parent directory name.

NotThisMethod

class nibabel._version.**NotThisMethod**
Bases: Exception

Exception raised if a method is not valid for the current scenario.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

VersioneerConfig

class nibabel._version.**VersioneerConfig**
Bases: object

Container for Versioneer configuration parameters.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

get_config

nibabel._version.**get_config**()
Create, populate and return the VersioneerConfig() object.

get_keywords

nibabel._version.**get_keywords**()
Get the keywords needed to look up the version information.

get_versions

nibabel._version.**get_versions**()
Get version information or return default if unable to do so.

git_get_keywords

nibabel._version.**git_get_keywords**(*versionfile_abs*)
Extract version information from the given file.

git_pieces_from_vcs

nibabel._version.**git_pieces_from_vcs**(*tag_prefix*, *root*, *verbose*, *run_command=<function run_command>*)
Get version from 'git describe' in the root of the source tree.

This only gets called if the git-archive 'subst' keywords were *not* expanded, and _version.py hasn't already been rewritten with a short version string, meaning we're inside a checked out source tree.

git_versions_from_keywords

nibabel._version.**git_versions_from_keywords**(*keywords*, *tag_prefix*, *verbose*)
Get version information from git keywords.

plus_or_dot

nibabel._version.**plus_or_dot**(*pieces*)
Return a + if we don't already have one, else return a .

register_vcs_handler

nibabel._version.**register_vcs_handler**(*vcs*, *method*)
Create decorator to mark a method as the handler of a VCS.

render

nibabel._version.**render**(*pieces*, *style*)
Render the given version pieces into the requested style.

render_git_describe

nibabel._version.**render_git_describe**(*pieces*)
TAG[-DISTANCE-gHEX][-dirty].

Like 'git describe –tags –dirty –always'.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

render_git_describe_long

nibabel._version.**render_git_describe_long**(*pieces*)
TAG-DISTANCE-gHEX[-dirty].

Like 'git describe –tags –dirty –always -long'. The distance/hash is unconditional.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

render_pep440

nibabel._version.**render_pep440**(*pieces*)
Build up version string, with post-release "local version identifier".

Our goal: TAG[+DISTANCE.gHEX[.dirty]] . Note that if you get a tagged build and then dirty it, you'll get TAG+0.gHEX.dirty

Exceptions: 1: no tags. git_describe was just HEX. 0+untagged.DISTANCE.gHEX[.dirty]

render_pep440_old

nibabel._version.**render_pep440_old**(*pieces*)
TAG[.postDISTANCE[.dev0]] .

The ".dev0" means dirty.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

render_pep440_post

nibabel._version.**render_pep440_post**(*pieces*)
TAG[.postDISTANCE[.dev0]+gHEX] .

The ".dev0" means dirty. Note that .dev0 sorts backwards (a dirty tree will appear "older" than the corresponding clean one), but you shouldn't be releasing software with -dirty anyways.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

render_pep440_pre

nibabel._version.**render_pep440_pre**(*pieces*)
TAG[.post0.devDISTANCE] – No -dirty.

Exceptions: 1: no tags. 0.post0.devDISTANCE

run_command

nibabel._version.**run_command**(*commands*, *args*, *cwd=None*, *verbose=False*, *hide_stderr=False*, *env=None*)
Call the given command(s).

versions_from_parentdir

nibabel._version.**versions_from_parentdir**(*parentdir_prefix*, *root*, *verbose*)
Try to determine the version from the parent directory name.

Source tarballs conventionally unpack into a directory that includes both the project name and a version string. We will also support searching up two directory levels for an appropriately named parent directory

benchmarks

—

Module: benchmarks.bench_array_to_file

Benchmarks for array_to_file routine

Run benchmarks with:

```
import nibabel as nib
nib.bench()
```

Run this benchmark with:

```
pytest -c <path>/benchmarks/pytest.benchmark.ini <path>/benchmarks/bench_array_to_
↪file.py
```

---

bench_array_to_file()

Module: benchmarks.bench_arrayproxy_slicing

Benchmarks for ArrayProxy slicing of gzipped and non-gzipped files

Run benchmarks with:

```
import nibabel as nib
nib.bench()
```

Run this benchmark with:

```
pytest -c <path>/benchmarks/pytest.benchmark.ini <path>/benchmarks/bench_arrayproxy_
↪slicing.py
```

---

bench_arrayproxy_slicing()

Module: benchmarks.bench_fileslice

Benchmarks for fileslicing

import nibabel as nib nib.bench()

Run this benchmark with:

```
pytest -c <path>/benchmarks/pytest.benchmark.ini <path>/benchmarks/bench_fileslice.py
```

---

bench_fileslice([bytes, file_, gz, bz2])

---

run_slices(file_like[, repeat, offset, order])

Module: benchmarks.bench_finite_range

Benchmarks for finite_range routine

Run benchmarks with:

```
import nibabel as nib
nib.bench()
```

Run this benchmark with:

```
pytest -c <path>/benchmarks/pytest.benchmark.ini <path>/benchmarks/bench_finite_range.
↪py
```

bench_finite_range()	

Module: benchmarks.bench_load_save

Benchmarks for load and save of image arrays

Run benchmarks with:

```
import nibabel as nib
nib.bench()
```

Run this benchmark with:

```
pytest -c <path>/benchmarks/pytest.benchmark.ini <path>/benchmarks/bench_load_save.py
```

bench_load_save()	

Module: benchmarks.butils

Benchmarking utilities

print_git_title(title)	Prints title string with git hash if possible, and underline

bench_array_to_file

nibabel.benchmarks.bench_array_to_file.**bench_array_to_file**()

bench_arrayproxy_slicing

nibabel.benchmarks.bench_arrayproxy_slicing.**bench_arrayproxy_slicing**()

bench_fileslice

nibabel.benchmarks.bench_fileslice.**bench_fileslice**(*bytes=True*, *file_=True*, *gz=True*, *bz2=False*)

run_slices

nibabel.benchmarks.bench_fileslice.**run_slices**(*file_like*, *repeat=3*, *offset=0*, *order='F'*)

bench_finite_range

nibabel.benchmarks.bench_finite_range.**bench_finite_range**()

bench_load_save

nibabel.benchmarks.bench_load_save.**bench_load_save**()

print_git_title

nibabel.benchmarks.butils.**print_git_title**(*title*)
Prints title string with git hash if possible, and underline

brikhead

Class for reading AFNI BRIK/HEAD datasets

See https://afni.nimh.nih.gov/pub/dist/doc/program_help/README.attributes.html for information on what is required to have a valid BRIK/HEAD dataset.

Unless otherwise noted, descriptions AFNI attributes in the code refer to this document.

Notes

In the AFNI HEAD file, the first two values of the attribute DATASET_RANK determine the shape of the data array stored in the corresponding BRIK file. The first value, DATASET_RANK[0], must be set to 3 denoting a 3D image. The second value, DATASET_RANK[1], determines how many "sub-bricks" (in AFNI parlance) / volumes there are along the fourth (traditionally, but not exclusively) time axis. Thus, DATASET_RANK[1] will (at least as far as I (RM) am aware) always be >= 1. This permits sub-brick indexing common in AFNI programs (e.g., example4d+orig'[0]').

AFNIArrayProxy(file_like, header, *[, mmap, ...])	Proxy object for AFNI image array.
AFNIHeader(info)	Class for AFNI header
AFNIHeaderError	Error when reading AFNI HEAD file

continues on next page

Table 84 – continued from previous page

AFNIImage(dataobj, affine[, header, extra, ...])	AFNI Image file
AFNIImageError	Error when reading AFNI BRIK files
parse_AFNI_header(fobj)	Parses *fobj* to extract information from HEAD file

AFNIArrayProxy

class nibabel.brikhead.**AFNIArrayProxy**(*file_like*, *header*, *, *mmap=True*, *keep_file_open=None*)

Bases: *nibabel.arrayproxy.ArrayProxy*

Proxy object for AFNI image array.

Attributes

scaling [np.ndarray] Scaling factor (one factor per volume/sub-brick) for data. Default is None

Initialize AFNI array proxy

Parameters

file_like [file-like object] File-like object or filename. If file-like object, should implement at least `read` and `seek`.

header [`AFNIHeader` object]

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If *file_like* cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_like` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

__init__(*file_like*, *header*, *, *mmap=True*, *keep_file_open=None*)

Initialize AFNI array proxy

Parameters

file_like [file-like object] File-like object or filename. If file-like object, should implement at least `read` and `seek`.

header [`AFNIHeader` object]

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If *file_like* cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the

image is accessed. If `file_like` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

property scaling

AFNIHeader

class nibabel.brikhead.**AFNIHeader**(*info*)

Bases: *nibabel.spatialimages.SpatialHeader*

Class for AFNI header

Initialize AFNI header object

Parameters

info [dict] Information from HEAD file as obtained by *parse_AFNI_header()*

Examples

```
>>> fname = os.path.join(datadir, 'example4d+orig.HEAD')
>>> header = AFNIHeader(parse_AFNI_header(fname))
>>> header.get_data_dtype().str
'<i2'
>>> header.get_zooms()
(3.0, 3.0, 3.0, 3.0)
>>> header.get_data_shape()
(33, 41, 25, 3)
```

__init__(*info*)

Initialize AFNI header object

Parameters

info [dict] Information from HEAD file as obtained by *parse_AFNI_header()*

Examples

```
>>> fname = os.path.join(datadir, 'example4d+orig.HEAD')
>>> header = AFNIHeader(parse_AFNI_header(fname))
>>> header.get_data_dtype().str
'<i2'
>>> header.get_zooms()
(3.0, 3.0, 3.0, 3.0)
>>> header.get_data_shape()
(33, 41, 25, 3)
```

copy()

Copy object to independent representation

The copy should not be affected by any changes to the original object.

classmethod from_fileobj(*fileobj*)

classmethod from_header(*header=None*)

get_affine()
Returns affine of dataset

Examples

```
>>> fname = os.path.join(datadir, 'example4d+orig.HEAD')
>>> header = AFNIHeader(parse_AFNI_header(fname))
>>> header.get_affine()
array([[ -3.    ,  -0.    ,  -0.    ,  49.5   ],
       [ -0.    ,  -3.    ,  -0.    ,  82.312 ],
       [  0.    ,   0.    ,   3.    , -52.3511],
       [  0.    ,   0.    ,   0.    ,   1.    ]])
```

get_data_offset()
Data offset in BRIK file

Offset is always 0.

get_data_scaling()
AFNI applies volume-specific data scaling

Examples

```
>>> fname = os.path.join(datadir, 'scaled+tlrc.HEAD')
>>> header = AFNIHeader(parse_AFNI_header(fname))
>>> header.get_data_scaling()
array([3.883363e-08])
```

get_slope_inter()
Use *self.get_data_scaling()* instead

Holdover because `AFNIArrayProxy` (inheriting from `ArrayProxy`) requires this functionality so as to not error.

get_space()
Return label for anatomical space to which this dataset is aligned.

Returns

space [str] AFNI "space" designation; one of [ORIG, ANAT, TLRC, MNI]

Notes

There appears to be documentation for these spaces at https://afni.nimh.nih.gov/pub/dist/atlases/elsedemo/AFNI_atlas_spaces.niml

get_volume_labels()
Returns volume labels

Returns

labels [list of str] Labels for volumes along fourth dimension

Examples

```
>>> header = AFNIHeader(parse_AFNI_header(os.path.join(datadir,
↪'example4d+orig.HEAD')))
>>> header.get_volume_labels()
['#0', '#1', '#2']
```

AFNIHeaderError

class nibabel.brikhead.**AFNIHeaderError**

Bases: *nibabel.spatialimages.HeaderDataError*

Error when reading AFNI HEAD file

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

AFNIImage

class nibabel.brikhead.**AFNIImage**(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.spatialimages.SpatialImage*

AFNI Image file

Can be loaded from either the BRIK or HEAD file (but MUST specify one!)

Examples

```
>>> import nibabel as nib
>>> brik = nib.load(os.path.join(datadir, 'example4d+orig.BRIK.gz'))
>>> brik.shape
(33, 41, 25, 3)
>>> brik.affine
array([[ -3.    ,  -0.    ,  -0.    ,  49.5   ],
       [ -0.    ,  -3.    ,  -0.    ,  82.312 ],
       [  0.    ,   0.    ,   3.    , -52.3511],
       [  0.    ,   0.    ,   0.    ,   1.    ]])
>>> head = load(os.path.join(datadir, 'example4d+orig.HEAD'))
>>> np.array_equal(head.get_fdata(), brik.get_fdata())
True
```

Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *affine*, *header=None*, *extra=None*, *file_map=None*)
Initialize image

The image is a combination of (array-like, affine matrix, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have a `shape` attribute or property

affine [None or (4,4) array-like] homogenous affine giving relationship between voxel coordinates and world coordinates. Affine can also be None. In this case, `obj.affine` also returns None, and the affine as written to disk will depend on the file format.

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

ImageArrayProxy
alias of *nibabel.brikhead.AFNIArrayProxy*

files_types = (('image', '.brik'), ('header', '.head'))

classmethod filespec_to_file_map(*filespec*)
Make *file_map* from filename *filespec*

AFNI BRIK files can be compressed, but HEAD files cannot - see afni.nimh.nih.gov/pub/dist/doc/program_help/README.compression.html. Thus, if you have AFNI files my_image.HEAD and my_image.BRIK.gz and you want to load the AFNI BRIK / HEAD pair, you can specify:

- The HEAD filename - e.g., my_image.HEAD
- The BRIK filename w/o compressed extension - e.g., my_image.BRIK
- The full BRIK filename - e.g., my_image.BRIK.gz

Parameters

filespec [str] Filename that might be for this image file type.

Returns

file_map [dict] dict with keys `image` and `header` where values are fileholder objects for the respective BRIK and HEAD files

Raises

ImageFileError If *filespec* is not recognizable as being a filename for this image type.

classmethod from_file_map(*file_map*, ***, *mmap=True*, *keep_file_open=None*)
Creates an AFNIImage instance from *file_map*

Parameters

file_map [dict] dict with keys `image, header` and values being fileholder objects for the respective BRIK and HEAD files

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{None, True, False}, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. If `file_like` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

header_class
alias of *nibabel.brikhead.AFNIHeader*

makeable = False

rw = False

valid_exts = ('.brik', '.head')

AFNIImageError

class nibabel.brikhead.**AFNIImageError**
Bases: *nibabel.spatialimages.ImageDataError*

Error when reading AFNI BRIK files

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

parse_AFNI_header

nibabel.brikhead.**parse_AFNI_header**(*fobj*)
Parses *fobj* to extract information from HEAD file

Parameters

fobj [file-like object] AFNI HEAD file object or filename. If file object, should implement at least `read`

Returns

info [dict] Dictionary containing AFNI-style key:value pairs from HEAD file

Examples

```
>>> fname = os.path.join(datadir, 'example4d+orig.HEAD')
>>> info = parse_AFNI_header(fname)
>>> print(info['BYTEORDER_STRING'])
LSB_FIRST
>>> print(info['BRICK_TYPES'])
[1, 1, 1]
```

cmdline

Functionality to be exposed in the command line

Module: cmdline.conform

Conform neuroimaging volume to arbitrary shape and voxel size.

main([args])	Main program function.

Module: cmdline.dicomfs

DICOMFS(*args, **kwargs)	
FileHandle(fno)	
dummy_fuse()	Dummy fuse "module" so that nose does not blow during doctests
fuse	alias of *nibabel.cmdline.dicomfs.dummy_fuse*
get_opt_parser()	
main([args])	

Module: cmdline.diff

Quick summary of the differences among a set of neuroimaging files

Notes:

- difference in data types for header fields will be detected, but endianness difference will not be detected. It is done so to compare files with native endianness used in data files.

are_values_different(*values)	Generically compare values, return True if different
diff(files[, header_fields, ...])	

continues on next page

Table 88 – continued from previous page

display_diff(files, diff)	Format header differences into a nice string
get_data_diff(files[, max_abs, max_rel, dtype])	Get difference between data
get_data_hash_diff(files[, dtype])	Get difference between md5 values of data
get_headers_diff(file_headers[, names])	Get difference between headers
get_opt_parser()	
main([args, out])	Getting the show on the road

Module: cmdline.ls

Output a summary table for neuroimaging files (resolution, dimensionality, etc.)

get_opt_parser()	
main([args])	Show must go on
proc_file(f, opts)	

Module: cmdline.nifti_dx

Print nifti diagnostics for header files

main([args])	Go go team

Module: cmdline.parrec2nii

Code for PAR/REC to NIfTI converter command

error(msg, exit_code)	
get_opt_parser()	
main()	
proc_file(infile, opts)	
verbose(msg[, indent])	

Module: cmdline.roi

lossless_slice(img, slicers)	
main([args])	
parse_slice(crop[, allow_step])	
sanitize(args)	

Module: cmdline.stats

Compute image statistics

main([args])	Main program function.

Module: cmdline.tck2trk

Convert tractograms (TCK -> TRK).

main()	
parse_args()	

Module: cmdline.trk2tck

Convert tractograms (TRK -> TCK).

main()	
parse_args()	

Module: cmdline.utils

Helper utilities to be used in cmdline applications

ap(helplist, format_[, sep])	Little helper to enforce consistency
safe_get(obj, name)	A getattr which would return '-' if getattr fails
table2string(table[, out])	Given list of lists figure out their common widths and print to out
verbose(thing, msg)	Print *s* if *thing* is less than the *verbose_level*

main

nibabel.cmdline.conform.**main**(*args=None*)
Main program function.

DICOMFS

class nibabel.cmdline.dicomfs.**DICOMFS**(**args*, ***kwargs*)
Bases: object

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

get_paths()

getattr(*path*)

match_path(*path*)

open(*path*, *flags*)

read(*path*, *size*, *offset*, *fh*)

readdir(*path*, *fh*)

release(*path*, *flags*, *fh*)

FileHandle

class nibabel.cmdline.dicomfs.**FileHandle**(*fno*)
Bases: object

__init__(*fno*)
Initialize self. See help(type(self)) for accurate signature.

dummy_fuse

class nibabel.cmdline.dicomfs.**dummy_fuse**
Bases: object

Dummy fuse "module" so that nose does not blow during doctests

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Fuse
alias of object

fuse_python_api = (0, 2)

fuse

nibabel.cmdline.dicomfs.**fuse**
alias of *nibabel.cmdline.dicomfs.dummy_fuse*

get_opt_parser

nibabel.cmdline.dicomfs.**get_opt_parser**()

main

nibabel.cmdline.dicomfs.**main**(*args=None*)

are_values_different

nibabel.cmdline.diff.**are_values_different**(**values*)
Generically compare values, return True if different

Note that comparison is targetting reporting of comparison of the headers so has following specifics: - even a difference in data types is considered a difference, i.e. 1 != 1.0 - nans are considered to be the "same", although generally nan != nan

diff

nibabel.cmdline.diff.**diff**(*files*, *header_fields='all'*, *data_max_abs_diff=None*, *data_max_rel_diff=None*, *dtype=<class 'numpy.float64'>*)

display_diff

nibabel.cmdline.diff.**display_diff**(*files*, *diff*)
Format header differences into a nice string

Parameters

files: list of files that were compared so we can print their names

diff: dict of different valued header fields

Returns

str string-formatted table of differences

get_data_diff

nibabel.cmdline.diff.**get_data_diff**(*files*, *max_abs=0*, *max_rel=0*, *dtype=<class 'numpy.float64'>*)

Get difference between data

Parameters

files: list of (str or ndarray) If list of strings is provided – they must be existing file names

max_abs: float, optional Maximal absolute difference to tolerate.

max_rel: float, optional Maximal relative (*abs(diff)/mean(diff)*) difference to tolerate. If *max_abs* is specified, then those data points with lesser than that absolute difference, are not considered for relative difference testing

dtype: np, optional Datatype to be used when extracting data from files

Returns

diffs: OrderedDict An ordered dict with a record per each file which has differences with other files subsequent detected. Each record is a list of difference records, one per each file pair. Each difference record is an Ordered Dict with possible keys 'abs' or 'rel' showing maximal absolute or relative differences in the file or the record ('CMP': 'incompat') if file shapes are incompatible.

get_data_hash_diff

nibabel.cmdline.diff.**get_data_hash_diff**(*files*, *dtype=<class 'numpy.float64'>*)

Get difference between md5 values of data

Parameters

files: list of actual files

Returns

list np.array: md5 values of respective files

get_headers_diff

nibabel.cmdline.diff.**get_headers_diff**(*file_headers*, *names=None*)

Get difference between headers

Parameters

file_headers: list of actual headers (dicts) from files

names: list of header fields to test

Returns

dict str: list for each header field which differs, return list of values per each file

get_opt_parser

nibabel.cmdline.diff.**get_opt_parser**()

main

nibabel.cmdline.diff.**main**(*args=None*, *out=None*)
Getting the show on the road

get_opt_parser

nibabel.cmdline.ls.**get_opt_parser**()

main

nibabel.cmdline.ls.**main**(*args=None*)
Show must go on

proc_file

nibabel.cmdline.ls.**proc_file**(*f*, *opts*)

main

nibabel.cmdline.nifti_dx.**main**(*args=None*)
Go go team

error

nibabel.cmdline.parrec2nii.**error**(*msg*, *exit_code*)

get_opt_parser

nibabel.cmdline.parrec2nii.**get_opt_parser**()

main

nibabel.cmdline.parrec2nii.**main**()

proc_file

nibabel.cmdline.parrec2nii.**proc_file**(*infile*, *opts*)

verbose

nibabel.cmdline.parrec2nii.**verbose**(*msg*, *indent=0*)

lossless_slice

nibabel.cmdline.roi.**lossless_slice**(*img*, *slicers*)

main

nibabel.cmdline.roi.**main**(*args=None*)

parse_slice

nibabel.cmdline.roi.**parse_slice**(*crop*, *allow_step=True*)

sanitize

nibabel.cmdline.roi.**sanitize**(*args*)

main

nibabel.cmdline.stats.**main**(*args=None*)
Main program function.

main

nibabel.cmdline.tck2trk.**main**()

parse_args

nibabel.cmdline.tck2trk.**parse_args**()

main

nibabel.cmdline.trk2tck.**main**()

parse_args

nibabel.cmdline.trk2tck.**parse_args**()

ap

nibabel.cmdline.utils.**ap**(*helplist*, *format_*, *sep=', '*)
Little helper to enforce consistency

safe_get

nibabel.cmdline.utils.**safe_get**(*obj*, *name*)
A getattr which would return '-' if getattr fails

table2string

nibabel.cmdline.utils.**table2string**(*table*, *out=None*)
Given list of lists figure out their common widths and print to out

Parameters

table [list of lists of strings] What is aimed to be printed

out [None or stream] Where to print. If None – will print and return string

Returns

string if out was None

verbose

nibabel.cmdline.utils.**verbose**(*thing*, *msg*)
Print *s* if *thing* is less than the *verbose_level*

dataobj_images

File-based images that have data arrays

The class:*DataObjImage* class defines an image that extends the `FileBasedImage` by adding an array-like object, named `dataobj`. This can either be an actual numpy array, or an object that:

- returns an array from `numpy.asanyarray(obj)`;
- has an attribute or property `shape`.

DataobjImage(dataobj[, header, extra, file_map])	Template class for images that have dataobj data stores

DataobjImage

class nibabel.dataobj_images.**DataobjImage**(*dataobj*, *header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.filebasedimages.FileBasedImage*

Template class for images that have dataobj data stores

Initialize dataobj image

The datobj image is a combination of (dataobj, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have `shape` and `ndim` attributes or properties

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*dataobj*, *header=None*, *extra=None*, *file_map=None*)

Initialize dataobj image

The datobj image is a combination of (dataobj, header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

dataobj [object] Object containg image data. It should be some object that retuns an array from `np.asanyarray`. It should have `shape` and `ndim` attributes or properties

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

property dataobj

classmethod from_file_map(*file_map*, *, *mmap=True*, *keep_file_open=None*)

Class method to create image from mapping in `file_map`

Parameters

file_map [dict] Mapping with (kay, value) pairs of (`file_type`, FileHolder instance giving file-likes for each file needed for this image type.

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the

image is accessed. If `file_map` refers to an open file handle, this setting has no effect. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [DataobjImage instance]

classmethod from_filename(*filename*, *, *mmap=True*, *keep_file_open=None*)

Class method to create image from filename *filename*

Parameters

filename [str] Filename of image to load

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is accessed. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [DataobjImage instance]

get_data(*caching='fill'*)

Return image data from image with any necessary scaling applied

get_data() is deprecated in favor of get_fdata(), which has a more predictable return type. To obtain get_data() behavior going forward, use numpy.asanyarray(img.dataobj).

- deprecated from version: 3.0
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 5.0

Warning: We recommend you use the `get_fdata` method instead of the `get_data` method, because it is easier to predict the return data type. `get_data` will be deprecated around November 2019 and removed around November 2021.

If you don't care about the predictability of the return data type, and you want the minimum possible data size in memory, you can replicate the array that would be returned by `img.get_data()` by using `np.asanyarray(img.dataobj)`.

The image `dataobj` property can be an array proxy or an array. An array proxy is an object that knows how to load the image data from disk. An image with an array proxy `dataobj` is a *proxy image*; an image with an array in `dataobj` is an *array image*.

The default behavior for `get_data()` on a proxy image is to read the data from the proxy, and store in an internal cache. Future calls to `get_data` will return the cached array. This is the behavior selected with *caching* == "fill".

Once the data has been cached and returned from an array proxy, if you modify the returned array, you will also modify the cached array (because they are the same array). Regardless of the *caching* flag, this is always true of an array image.

Parameters

caching [{'fill', 'unchanged'}, optional] See the Notes section for a detailed explanation. This argument specifies whether the image object should fill in an internal cached reference to the returned image data array. "fill" specifies that the image should fill an internal cached reference if currently empty. Future calls to `get_data` will return this cached reference. You might prefer "fill" to save the image object from having to reload the array data from disk on each call to `get_data`. "unchanged" means that the image should not fill in the internal cached reference if the cache is currently empty. You might prefer "unchanged" to "fill" if you want to make sure that the call to `get_data` does not create an extra (cached) reference to the returned array. In this case it is easier for Python to free the memory from the returned array.

Returns

data [array] array of image data

See also:

uncache empty the array data cache

Notes

All images have a property `dataobj` that represents the image array data. Images that have been loaded from files usually do not load the array data from file immediately, in order to reduce image load time and memory use. For these images, `dataobj` is an *array proxy*; an object that knows how to load the image array data from file.

By default (*caching* == "fill"), when you call `get_data` on a proxy image, we load the array data from disk, store (cache) an internal reference to this array data, and return the array. The next time you call `get_data`, you will get the cached reference to the array, so we don't have to load the array data from disk again.

Array images have a `dataobj` property that already refers to an array in memory, so there is no benefit to caching, and the *caching* keywords have no effect.

For proxy images, you may not want to fill the cache after reading the data from disk because the cache will hold onto the array memory until the image object is deleted, or you use the image `uncache` method. If you don't want to fill the cache, then always use `get_data(caching='unchanged')`; in this case `get_data` will not fill the cache (store the reference to the array) if the cache is empty (no reference to the array). If the cache is full, "unchanged" leaves the cache full and returns the cached array reference.

The cache can affect the behavior of the image, because if the cache is full, or you have an array image, then modifying the returned array will modify the result of future calls to `get_data()`. For example you might do this:

```
>>> import os
>>> import nibabel as nib
>>> from nibabel.testing import data_path
>>> img_fname = os.path.join(data_path, 'example4d.nii.gz')
```

```
>>> img = nib.load(img_fname) # This is a proxy image
>>> nib.is_proxy(img.dataobj)
True
```

The array is not yet cached by a call to "get_data", so:

```
>>> img.in_memory
False
```

After we call `get_data` using the default *caching* == 'fill', the cache contains a reference to the returned array `data`:

```
>>> data = img.get_data()
>>> img.in_memory
True
```

We modify an element in the returned data array:

```
>>> data[0, 0, 0, 0]
0
>>> data[0, 0, 0, 0] = 99
>>> data[0, 0, 0, 0]
99
```

The next time we call 'get_data', the method returns the cached reference to the (modified) array:

```
>>> data_again = img.get_data()
>>> data_again is data
True
>>> data_again[0, 0, 0, 0]
99
```

If you had *initially* used *caching* == 'unchanged' then the returned `data` array would have been loaded from file, but not cached, and:

```
>>> img = nib.load(img_fname)  # a proxy image again
>>> data = img.get_data(caching='unchanged')
>>> img.in_memory
False
>>> data[0, 0, 0] = 99
>>> data_again = img.get_data(caching='unchanged')
>>> data_again is data
False
>>> data_again[0, 0, 0, 0]
0
```

get_fdata(*caching='fill'*, *dtype=<class 'numpy.float64'>*)

Return floating point image data with necessary scaling applied

The image `dataobj` property can be an array proxy or an array. An array proxy is an object that knows how to load the image data from disk. An image with an array proxy `dataobj` is a *proxy image*; an image with an array in `dataobj` is an *array image*.

The default behavior for `get_fdata()` on a proxy image is to read the data from the proxy, and store in an internal cache. Future calls to `get_fdata` will return the cached array. This is the behavior selected with *caching* == "fill".

Once the data has been cached and returned from an array proxy, if you modify the returned array, you will also modify the cached array (because they are the same array). Regardless of the *caching* flag, this is always true of an array image.

Parameters

caching [{'fill', 'unchanged'}, optional] See the Notes section for a detailed explanation. This argument specifies whether the image object should fill in an internal cached reference

to the returned image data array. "fill" specifies that the image should fill an internal cached reference if currently empty. Future calls to `get_fdata` will return this cached reference. You might prefer "fill" to save the image object from having to reload the array data from disk on each call to `get_fdata`. "unchanged" means that the image should not fill in the internal cached reference if the cache is currently empty. You might prefer "unchanged" to "fill" if you want to make sure that the call to `get_fdata` does not create an extra (cached) reference to the returned array. In this case it is easier for Python to free the memory from the returned array.

dtype [numpy dtype specifier] A numpy dtype specifier specifying a floating point type. Data is returned as this floating point type. Default is `np.float64`.

Returns

fdata [array] Array of image data of data type *dtype*.

See also:

uncache empty the array data cache

Notes

All images have a property `dataobj` that represents the image array data. Images that have been loaded from files usually do not load the array data from file immediately, in order to reduce image load time and memory use. For these images, `dataobj` is an *array proxy*; an object that knows how to load the image array data from file.

By default (*caching* == "fill"), when you call `get_fdata` on a proxy image, we load the array data from disk, store (cache) an internal reference to this array data, and return the array. The next time you call `get_fdata`, you will get the cached reference to the array, so we don't have to load the array data from disk again.

Array images have a `dataobj` property that already refers to an array in memory, so there is no benefit to caching, and the *caching* keywords have no effect.

For proxy images, you may not want to fill the cache after reading the data from disk because the cache will hold onto the array memory until the image object is deleted, or you use the image `uncache` method. If you don't want to fill the cache, then always use `get_fdata(caching='unchanged')`; in this case `get_fdata` will not fill the cache (store the reference to the array) if the cache is empty (no reference to the array). If the cache is full, "unchanged" leaves the cache full and returns the cached array reference.

The cache can effect the behavior of the image, because if the cache is full, or you have an array image, then modifying the returned array will modify the result of future calls to `get_fdata()`. For example you might do this:

```
>>> import os
>>> import nibabel as nib
>>> from nibabel.testing import data_path
>>> img_fname = os.path.join(data_path, 'example4d.nii.gz')
```

```
>>> img = nib.load(img_fname) # This is a proxy image
>>> nib.is_proxy(img.dataobj)
True
```

The array is not yet cached by a call to "get_fdata", so:

```
>>> img.in_memory
False
```

After we call `get_fdata` using the default *caching* == 'fill', the cache contains a reference to the returned array `data`:

```
>>> data = img.get_fdata()
>>> img.in_memory
True
```

We modify an element in the returned data array:

```
>>> data[0, 0, 0, 0]
0.0
>>> data[0, 0, 0, 0] = 99
>>> data[0, 0, 0, 0]
99.0
```

The next time we call 'get_fdata', the method returns the cached reference to the (modified) array:

```
>>> data_again = img.get_fdata()
>>> data_again is data
True
>>> data_again[0, 0, 0, 0]
99.0
```

If you had *initially* used *caching* == 'unchanged' then the returned `data` array would have been loaded from file, but not cached, and:

```
>>> img = nib.load(img_fname)  # a proxy image again
>>> data = img.get_fdata(caching='unchanged')
>>> img.in_memory
False
>>> data[0, 0, 0] = 99
>>> data_again = img.get_fdata(caching='unchanged')
>>> data_again is data
False
>>> data_again[0, 0, 0, 0]
0.0
```

property in_memory

True when any array data is in memory cache

There are separate caches for *get_data* reads and *get_fdata* reads. This property is True if either of those caches are set.

classmethod load(*filename*, *, *mmap=True*, *keep_file_open=None*)

Class method to create image from filename *filename*

Parameters

filename [str] Filename of image to load

mmap [{True, False, 'c', 'r'}, optional, keyword only] *mmap* controls the use of numpy memory mapping for reading image array data. If False, do not try numpy `memmap` for data array. If one of {'c', 'r'}, try numpy memmap with `mode=mmap`. A *mmap* value of True gives the same behavior as `mmap='c'`. If image data file cannot be memory-mapped, ignore *mmap* value and read array from file.

keep_file_open [{ None, True, False }, optional, keyword only] *keep_file_open* controls whether a new file handle is created every time the image is accessed, or a single file handle is created and used for the lifetime of this `ArrayProxy`. If `True`, a single file handle is created and used. If `False`, a new file handle is created every time the image is

accessed. The default value (`None`) will result in the value of `nibabel.arrayproxy.KEEP_FILE_OPEN_DEFAULT` being used.

Returns

img [DataobjImage instance]

property ndim

property shape

uncache()

Delete any cached read of data from proxied data

Remember there are two types of images:

- *array images* where the data `img.dataobj` is an array
- *proxy images* where the data `img.dataobj` is a proxy object

If you call `img.get_fdata()` on a proxy image, the result of reading from the proxy gets cached inside the image object, and this cache is what gets returned from the next call to `img.get_fdata()`. If you modify the returned data, as in:

```
data = img.get_fdata()
data[:] = 42
```

then the next call to `img.get_fdata()` returns the modified array, whether the image is an array image or a proxy image:

```
assert np.all(img.get_fdata() == 42)
```

When you uncache an array image, this has no effect on the return of `img.get_fdata()`, but when you uncache a proxy image, the result of `img.get_fdata()` returns to its original value.

deprecated

Module to help with deprecating objects and classes

FutureWarningMixin(*args, **kwargs)	Insert FutureWarning for object creation
ModuleProxy(module_name)	Proxy for module that may not yet have been imported
VisibleDeprecationWarning	Deprecation warning that will be shown by default

FutureWarningMixin

class nibabel.deprecated.**FutureWarningMixin**(**args*, ***kwargs*)

Bases: `object`

Insert FutureWarning for object creation

Examples

```
>>> class C(object): pass
>>> class D(FutureWarningMixin, C):
...     warn_message = "Please, don't use this class"
```

Record the warning

```
>>> with warnings.catch_warnings(record=True) as warns:
...     d = D()
...     warns[0].message.args[0]
"Please, don't use this class"
```

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

warn_message = 'This class will be removed in future versions'

ModuleProxy

class nibabel.deprecated.**ModuleProxy**(*module_name*)
Bases: object

Proxy for module that may not yet have been imported

Parameters

module_name [str] Full module name e.g. nibabel.minc

Examples

:: arr = np.arange(24).reshape((2, 3, 4)) nifti1 = ModuleProxy('nibabel.nifti1') nifti1_image = nifti1.Nifti1Image(arr, np.eye(4))

So, the nifti1 object is a proxy that will import the required module when you do attribute access and return the attributes of the imported module.

__init__(*module_name*)
Initialize self. See help(type(self)) for accurate signature.

VisibleDeprecationWarning

class nibabel.deprecated.**VisibleDeprecationWarning**
Bases: UserWarning

Deprecation warning that will be shown by default

Python >= 2.7 does not show standard DeprecationWarnings by default:

http://docs.python.org/dev/whatsnew/2.7.html#the-future-for-python-2-x

Use this class for cases where we do want to show deprecations by default.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

deprecator

Class for recording and reporting deprecations

Deprecator(version_comparator[, warn_class, ...])	Class to make decorator marking function or method as deprecated
ExpiredDeprecationError	Error for expired deprecation

Deprecator

class nibabel.deprecator.**Deprecator**(*version_comparator*, *warn_class=<class 'DeprecationWarning'>*, *error_class=<class 'nibabel.deprecator.ExpiredDeprecationError'>*)

Bases: `object`

Class to make decorator marking function or method as deprecated

The decorated function / method will:

- Raise the given *warning_class* warning when the function / method gets called, up to (and including) version *until* (if specified);
- Raise the given *error_class* error when the function / method gets called, when the package version is greater than version *until* (if specified).

Parameters

version_comparator [callable] Callable accepting string as argument, and return 1 if string represents a higher version than encoded in the *version_comparator*, 0 if the version is equal, and -1 if the version is lower. For example, the *version_comparator* may compare the input version string to the current package version string.

warn_class [class, optional] Class of warning to generate for deprecation.

error_class [class, optional] Class of error to generate when *version_comparator* returns 1 for a given argument of `until` in the `__call__` method (see below).

__init__(*version_comparator*, *warn_class=<class 'DeprecationWarning'>*, *error_class=<class 'nibabel.deprecator.ExpiredDeprecationError'>*)

Initialize self. See help(type(self)) for accurate signature.

is_bad_version(*version_str*)

Return True if *version_str* is too high

Tests *version_str* with `self.version_comparator`

Parameters

version_str [str] String giving version to test

Returns

is_bad [bool] True if *version_str* is for version below that expected by `self.version_comparator`, False otherwise.

ExpiredDeprecationError

class nibabel.deprecator.**ExpiredDeprecationError**
Bases: RuntimeError

Error for expired deprecation

Error raised when a called function or method has passed out of its deprecation period.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

filebasedimages

Common interface for any image format–volume or surface, binary or xml.

FileBasedHeader()	Template class to implement header protocol
FileBasedImage([header, extra, file_map])	Abstract image class with interface for loading/saving images from disk.
ImageFileError	
SerializableImage([header, extra, file_map])	Abstract image class for (de)serializing images to/from byte strings.

FileBasedHeader

class nibabel.filebasedimages.**FileBasedHeader**
Bases: object

Template class to implement header protocol

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

copy()
Copy object to independent representation

The copy should not be affected by any changes to the original object.

classmethod from_fileobj(*fileobj*)

classmethod from_header(*header=None*)

write_to(*fileobj*)

FileBasedImage

class nibabel.filebasedimages.**FileBasedImage**(*header=None*, *extra=None*, *file_map=None*)
Bases: object

Abstract image class with interface for loading/saving images from disk.

The class doesn't define any image properties.

It has:

attributes:

- extra

properties:

- shape
- header

methods:

- get_header() (deprecated, use header property instead)
- to_filename(fname) - writes data to filename(s) derived from `fname`, where the derivation may differ between formats.
- to_file_map() - save image to files with which the image is already associated.

classmethods:

- from_filename(fname) - make instance by loading from filename
- from_file_map(fmap) - make instance from file map
- instance_to_filename(img, fname) - save `img` instance to filename `fname`.

It also has a `header` - some standard set of meta-data that is specific to the image format, and `extra` - a dictionary container for any other metadata.

You cannot slice an image, and trying to slice an image generates an informative TypeError.

There are several ways of writing data

There is the usual way, which is the default:

```
img.to_filename(fname)
```

and that is, to take the data encapsulated by the image and cast it to the datatype the header expects, setting any available header scaling into the header to help the data match.

You can load the data into an image from file with:

```
img.from_filename(fname)
```

The image stores its associated files in its `file_map` attribute. In order to just save an image, for which you know there is an associated filename, or other storage, you can do:

```
img.to_file_map()
```

You can get the data out again with:

```
img.get_fdata()
```

Less commonly, for some image types that support it, you might want to fetch out the unscaled array via the object containing the data:

```
unscaled_data = img.dataobj.get_unscaled()
```

Analyze-type images (including nifti) support this, but others may not (MINC, for example).

Sometimes you might to avoid any loss of precision by making the data type the same as the input:

```
hdr = img.header
hdr.set_data_dtype(data.dtype)
img.to_filename(fname)
```

Files interface

The image has an attribute `file_map`. This is a mapping, that has keys corresponding to the file types that an image needs for storage. For example, the Analyze data format needs an `image` and a `header` file type for storage:

```
>>> import numpy as np
>>> import nibabel as nib
>>> data = np.arange(24, dtype='f4').reshape((2,3,4))
>>> img = nib.AnalyzeImage(data, np.eye(4))
>>> sorted(img.file_map)
['header', 'image']
```

The values of `file_map` are not in fact files but objects with attributes `filename`, `fileobj` and `pos`.

The reason for this interface, is that the contents of files has to contain enough information so that an existing image instance can save itself back to the files pointed to in `file_map`. When a file holder holds active file-like objects, then these may be affected by the initial file read; in this case, the contains file-like objects need to carry the position at which a write (with `to_file_map`) should place the data. The `file_map` contents should therefore be such, that this will work.

Initialize image

The image is a combination of (header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*header=None*, *extra=None*, *file_map=None*)

Initialize image

The image is a combination of (header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

files_types = (('image', None),)

classmethod filespec_to_file_map(*filespec*)

Make *file_map* for this class from filename *filespec*

Class method

Parameters

filespec [str or os.PathLike] Filename that might be for this image file type.

Returns

file_map [dict] *file_map* dict with (key, value) pairs of (`file_type`, FileHolder instance), where `file_type` is a string giving the type of the contained file.

Raises

ImageFileError if *filespec* is not recognizable as being a filename for this image type.

classmethod from_file_map(*file_map*)

classmethod from_filename(*filename*)

classmethod from_image(*img*)
Class method to create new instance of own class from *img*

Parameters

img [`spatialimage` instance] In fact, an object with the API of `FileBasedImage`.

Returns

cimg [`spatialimage` instance] Image, of our own class

get_filename()
Fetch the image filename

Parameters

None

Returns

fname [None or str] Returns None if there is no filename, or a filename string. If an image may have several filenames associated with it (e.g. Analyze `.img, .hdr` pair) then we return the more characteristic filename (the `.img` filename in the case of Analyze')

get_header()
Get header from image

get_header method is deprecated. Please use the `img.header` property instead.

- deprecated from version: 2.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 4.0

property header

header_class
alias of *nibabel.filebasedimages.FileBasedHeader*

classmethod instance_to_filename(*img*, *filename*)
Save *img* in our own format, to name implied by *filename*

This is a class method

Parameters

img [`any FileBasedImage` instance]

filename [str] Filename, implying name to which to save image.

classmethod load(*filename*)

classmethod make_file_map(*mapping=None*)
Class method to make files holder for this image type

Parameters

mapping [None or mapping, optional] mapping with keys corresponding to image file types (such as 'image', 'header' etc, depending on image class) and values that are filenames or file-like. Default is None

Returns

file_map [dict] dict with string keys given by first entry in tuples in sequence klass.files_types, and values of type FileHolder, where FileHolder objects have default values, other than those given by *mapping*

makeable = True

classmethod path_maybe_image(*filename*, *sniff=None*, *sniff_max=1024*)

Return True if *filename* may be image matching this class

Parameters

filename [str or os.PathLike] Filename for an image, or an image header (metadata) file. If *filename* points to an image data file, and the image type has a separate "header" file, we work out the name of the header file, and read from that instead of *filename*.

sniff [None or (bytes, filename), optional] Bytes content read from a previous call to this method, on another class, with metadata filename. This allows us to read metadata bytes once from the image or header, and pass this read set of bytes to other image classes, therefore saving a repeat read of the metadata. *filename* is used to validate that metadata would be read from the same file, re-reading if not. None forces this method to read the metadata.

sniff_max [int, optional] The maximum number of bytes to read from the metadata. If the metadata file is long enough, we read this many bytes from the file, otherwise we read to the end of the file. Longer values sniff more of the metadata / image file, making it more likely that the returned sniff will be useful for later calls to `path_maybe_image` for other image classes.

Returns

maybe_image [bool] True if *filename* may be valid for an image of this class.

sniff [None or (bytes, filename)] Read bytes content from found metadata. May be None if the file does not appear to have useful metadata.

rw = True

set_filename(*filename*)

Sets the files in the object from a given filename

The different image formats may check whether the filename has an extension characteristic of the format, and raise an error if not.

Parameters

filename [str or os.PathLike] If the image format only has one file associated with it, this will be the only filename set into the image `.file_map` attribute. Otherwise, the image instance will try and guess the other filenames from this given filename.

to_file_map(*file_map=None*)

to_filename(*filename*)

Write image to files implied by filename string

Parameters

filename [str or os.PathLike] filename to which to save image. We will parse *filename* with `filespec_to_file_map` to work out names for image, header etc.

Returns

None

valid_exts = ()

ImageFileError

class nibabel.filebasedimages.**ImageFileError**

Bases: Exception

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

SerializableImage

class nibabel.filebasedimages.**SerializableImage**(*header=None*, *extra=None*, *file_map=None*)

Bases: *nibabel.filebasedimages.FileBasedImage*

Abstract image class for (de)serializing images to/from byte strings.

The class doesn't define any image properties.

It has:

methods:

- to_bytes() - serialize image to byte string

classmethods:

- from_bytes(bytestring) - make instance by deserializing a byte string

Loading from byte strings should provide round-trip equivalence:

```
img_a = klass.from_bytes(bstr)
img_b = klass.from_bytes(img_a.to_bytes())

np.allclose(img_a.get_fdata(), img_b.get_fdata())
np.allclose(img_a.affine, img_b.affine)
```

Further, for images that are single files on disk, the following methods of loading the image must be equivalent:

```
img = klass.from_filename(fname)

with open(fname, 'rb') as fobj:
    img = klass.from_bytes(fobj.read())
```

And the following methods of saving a file must be equivalent:

```
img.to_filename(fname)

with open(fname, 'wb') as fobj:
    fobj.write(img.to_bytes())
```

Images that consist of separate header and data files (e.g., Analyze images) currently do not support this interface. For multi-file images, `to_bytes()` and `from_bytes()` must be overridden, and any encoding details should be documented.

Initialize image

The image is a combination of (header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

__init__(*header=None*, *extra=None*, *file_map=None*)

Initialize image

The image is a combination of (header), with optional metadata in *extra*, and filename / file-like objects contained in the *file_map* mapping.

Parameters

header [None or mapping or header instance, optional] metadata for this image format

extra [None or mapping, optional] metadata to associate with image that cannot be stored in the metadata of this image type

file_map [mapping, optional] mapping giving file information for this image format

classmethod from_bytes(*bytestring*)

Construct image from a byte string

Class method

Parameters

bstring [bytes] Byte string containing the on-disk representation of an image

to_bytes()

Return a `bytes` object with the contents of the file that would be written if the image were saved.

Parameters

None

Returns

bytes Serialized image

fileutils

Utilities for reading and writing to binary file formats

read_zt_byte_strings(fobj[, n_strings, buf-size])	Read zero-terminated byte strings from a file object *fobj*

read_zt_byte_strings

nibabel.fileutils.**read_zt_byte_strings**(*fobj*, *n_strings=1*, *bufsize=1024*)

Read zero-terminated byte strings from a file object *fobj*

Returns byte strings with terminal zero stripped.

Found strings can be of any length.

The file position of *fobj* on exit will be at the byte after the terminal 0 of the final read byte string.

Parameters

f [fileobj] File object to use. Should implement `read`, returning byte objects, and `seek(n, 1)` to seek from current file position.

n_strings [int, optional] Number of byte strings to return

bufsize: int, optional Define chunk size to load from file while searching for zero terminals. We load this many bytes at a time from the file, but the returned strings can be longer than *bufsize*.

Returns

byte_strings [list] List of byte strings, where strings do not include the terminal 0

imagestats

Functions for computing image statistics

count_nonzero_voxels(img)	Count number of non-zero voxels
mask_volume(img)	Compute volume of mask image.

count_nonzero_voxels

nibabel.imagestats.**count_nonzero_voxels**(*img*)

Count number of non-zero voxels

Parameters

img [`SpatialImage`] All voxels of the mask should be of value 1, background should have value 0.

Returns

count [int] Number of non-zero voxels

mask_volume

nibabel.imagestats.**mask_volume**(*img*)

Compute volume of mask image.

Equivalent to "fslstats /path/file.nii -V"

Parameters

img [SpatialImage] All voxels of the mask should be of value 1, background should have value 0.

Returns

volume [float] Volume of mask expressed in mm3.

Examples

```
>>> import numpy as np
>>> import nibabel as nb
>>> mask_data = np.zeros((20, 20, 20), dtype='u1')
>>> mask_data[5:15, 5:15, 5:15] = 1
>>> nb.imagestats.mask_volume(nb.Nifti1Image(mask_data, np.eye(4)))
1000.0
```

keywordonly

Decorator for labeling keyword arguments as keyword only

kw_only_func(n)	Return function decorator enforcing maximum of *n* positional arguments
kw_only_meth(n)	Return method decorator enforcing maximum of *n* positional arguments

kw_only_func

nibabel.keywordonly.**kw_only_func**(*n*)

Return function decorator enforcing maximum of *n* positional arguments

kw_only_meth

nibabel.keywordonly.**kw_only_meth**(*n*)
Return method decorator enforcing maximum of *n* positional arguments

The method has at least one positional argument `self` or `cls`; allow for that.

mriutils

Utilities for calculations related to MRI

MRIError	
calculate_dwell_time(water_fat_shift, ...)	Calculate the dwell time

MRIError

class nibabel.mriutils.**MRIError**
Bases: `ValueError`

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

calculate_dwell_time

nibabel.mriutils.**calculate_dwell_time**(*water_fat_shift*, *echo_train_length*, *field_strength*)
Calculate the dwell time

Parameters

water_fat_shift [float] The water fat shift of the recording, in pixels.

echo_train_length [int] The echo train length of the imaging sequence.

field_strength [float] Strength of the magnet in Tesla, e.g. 3.0 for a 3T magnet recording.

Returns

dwell_time [float] The dwell time in seconds.

Raises

MRIError if values are out of range

processing

Image processing functions for:

- smoothing
- resampling
- converting sd to and from FWHM

Smoothing and resampling routines need scipy

adapt_affine(affine, n_dim)	Adapt input / output dimensions of spatial *affine* for *n_dims*
conform(from_img[, out_shape, voxel_size, ...])	Resample image to `out_shape` with voxels of size `voxel_size`.
fwhm2sigma(fwhm)	Convert a FWHM value to sigma in a Gaussian kernel.
resample_from_to(from_img, to_vox_map[, ...])	Resample image *from_img* to mapped voxel space *to_vox_map*
resample_to_output(in_img[, voxel_sizes, ...])	Resample image *in_img* to output voxel axes (world space)
sigma2fwhm(sigma)	Convert a sigma in a Gaussian kernel to a FWHM value
smooth_image(img, fwhm[, mode, cval, out_class])	Smooth image *img* along voxel axes by FWHM *fwhm* millimeters

adapt_affine

nibabel.processing.**adapt_affine**(*affine*, *n_dim*)

Adapt input / output dimensions of spatial *affine* for *n_dims*

Adapts a spatial (4, 4) affine that is being applied to an image with fewer than 3 spatial dimensions, or more than 3 dimensions. If there are more than three dimensions, assume an identity transformation for these dimensions.

Parameters

affine [array-like] affine transform. Usually shape (4, 4). For what follows `N, M = affine.shape`

n_dims [int] Number of dimensions of underlying array, and therefore number of input dimensions for affine.

Returns

adapted [shape (M, n_dims+1) array] Affine array adapted to number of input dimensions. Columns of the affine corresponding to missing input dimensions have been dropped, columns corresponding to extra input dimensions have an extra identity column added

conform

nibabel.processing.**conform**(*from_img*, *out_shape=(256, 256, 256)*, *voxel_size=(1.0, 1.0, 1.0)*, *order=3*, *cval=0.0*, *orientation='RAS'*, *out_class=None*)

Resample image to `out_shape` with voxels of size `voxel_size`.

Using the default arguments, this function is meant to replicate most parts of FreeSurfer's `mri_convert --conform` command. Specifically, this function:

- Resamples data to `output_shape`
- Resamples voxel sizes to `voxel_size`
- Reorients to RAS (`mri_convert --conform` reorients to LIA)

Unlike `mri_convert --conform`, this command does not:

- Transform data to range [0, 255]
- Cast to unsigned eight-bit integer

Parameters

from_img [object] Object having attributes `dataobj`, `affine`, `header` and `shape`. If *out_class* is not None, `img.__class__` should be able to construct an image from data, affine and header.

out_shape [sequence, optional] The shape of the output volume. Default is (256, 256, 256).

voxel_size [sequence, optional] The size in millimeters of the voxels in the resampled output. Default is 1mm isotropic.

order [int, optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5 (see `scipy.ndimage.affine_transform`)

cval [scalar, optional] Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0 (see `scipy.ndimage.affine_transform`)

orientation [str, optional] Orientation of output image. Default is "RAS".

out_class [None or SpatialImage class, optional] Class of output image. If None, use `from_img.__class__`.

Returns

out_img [object] Image of instance specified by *out_class*, containing data output from resampling *from_img* into axes aligned to the output space of `from_img.affine`

fwhm2sigma

`nibabel.processing.`**`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
```

resample_from_to

`nibabel.processing.`**`resample_from_to`**(*from_img*, *to_vox_map*, *order=3*, *mode='constant'*, *cval=0.0*, *out_class=<class 'nibabel.nifti1.Nifti1Image'>*)

Resample image *from_img* to mapped voxel space *to_vox_map*

Resample using N-d spline interpolation.

Parameters

from_img [object] Object having attributes `dataobj`, `affine`, `header` and `shape`. If *out_class* is not None, `img.__class__` should be able to construct an image from data, affine and header.

to_vox_map [image object or length 2 sequence] If object, has attributes `shape` giving input voxel shape, and `affine` giving mapping of input voxels to output space. If length 2 sequence, elements are (shape, affine) with same meaning as above. The affine is a (4, 4) array-like.

order [int, optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5 (see `scipy.ndimage.affine_transform`)

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' (see `scipy.ndimage.affine_transform`)

cval [scalar, optional] Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0 (see `scipy.ndimage.affine_transform`)

out_class [None or SpatialImage class, optional] Class of output image. If None, use `from_img.__class__`.

Returns

out_img [object] Image of instance specified by *out_class*, containing data output from resampling *from_img* into axes aligned to the output space of `from_img.affine`

resample_to_output

nibabel.processing.**resample_to_output**(*in_img*, *voxel_sizes=None*, *order=3*, *mode='constant'*, *cval=0.0*, *out_class=<class 'nibabel.nifti1.Nifti1Image'>*)

Resample image *in_img* to output voxel axes (world space)

Parameters

in_img [object] Object having attributes `dataobj`, `affine`, `header`. If *out_class* is not None, `img.__class__` should be able to construct an image from data, affine and header.

voxel_sizes [None or sequence] Gives the diagonal entries of `out_img.affine` (except the trailing 1 for the homogenous coordinates) (``out_img.affine == np.diag(voxel_sizes + [1])``). If None, return identity *out_img.affine*. If scalar, interpret as vector `[voxel_sizes] * len(in_img.shape)`.

order [int, optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5 (see `scipy.ndimage.affine_transform`).

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' (see `scipy.ndimage.affine_transform`).

cval [scalar, optional] Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0 (see `scipy.ndimage.affine_transform`).

out_class [None or SpatialImage class, optional] Class of output image. If None, use `in_img.__class__`.

Returns

out_img [object] Image of instance specified by *out_class*, containing data output from resampling *in_img* into axes aligned to the output space of `in_img.affine`

sigma2fwhm

nibabel.processing.**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
```

smooth_image

nibabel.processing.**smooth_image**(*img*, *fwhm*, *mode='nearest'*, *cval=0.0*, *out_class=<class 'nibabel.nifti1.Nifti1Image'>*)

Smooth image *img* along voxel axes by FWHM *fwhm* millimeters

Parameters

img [object] Object having attributes `dataobj`, `affine`, `header` and `shape`. If *out_class* is not None, `img.__class__` should be able to construct an image from data, affine and header.

fwhm [scalar or length 3 sequence] FWHM *in mm* over which to smooth. The smoothing applies to the voxel axes, not to the output axes, but is in millimeters. The function adjusts the FWHM to voxels using the voxel sizes calculated from the affine. A scalar implies the same smoothing across the spatial dimensions of the image, but 0 smoothing over any further dimensions such as time. A vector should be the same length as the number of image dimensions.

mode [str, optional] Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect' or 'wrap'). Default is 'nearest'. This is different from the default for `scipy.ndimage.affine_transform`, which is 'constant'. 'nearest' might be a better choice when smoothing to the edge of an image where there is still strong brain signal, otherwise this signal will get blurred towards zero.

cval [scalar, optional] Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0 (see `scipy.ndimage.affine_transform`).

out_class [None or SpatialImage class, optional] Class of output image. If None, use `img.__class__`.

Returns

smoothed_img [object] Image of instance specified by *out_class*, containing data output from smoothing *img* data by given FWHM kernel.

pydicom_compat

Adapter module for working with pydicom < 1.0 and >= 1.0

In what follows, "dicom is available" means we can import either a) `dicom` (pydicom < 1.0) or or b) `pydicom` (pydicom >= 1.0).

Regardless of whether dicom is available this module should be importable without error, and always defines:

- have_dicom : True if we can import pydicom or dicom;
- pydicom : pydicom module or dicom module or None of not importable;
- read_file : `read_file` function if pydicom or dicom module is importable else None;
- tag_for_keyword : `tag_for_keyword` function if pydicom or dicom module is importable else None;

A test decorator is available in nibabel.nicom.tests:

- dicom_test : test decorator that skips test if dicom not available.

A deprecated copy is available here for backward compatibility.

dicom_test(func)	dicom_test has been moved to nibabel.nicom.tests

dicom_test

nibabel.pydicom_compat.**dicom_test** (*func*)
dicom_test has been moved to nibabel.nicom.tests

- deprecated from version: 3.1
- Will raise <class 'nibabel.deprecator.ExpiredDeprecationError'> as of version: 5.0

spaces

Routines to work with spaces

A space is defined by coordinate axes.

A voxel space can be expressed by a shape implying an array, where the axes are the axes of the array.

A mapped voxel space (mapped voxels) is either:

- an image, with attributes `shape` (the voxel space) and `affine` (the mapping), or
- a length 2 sequence with the same information (shape, affine).

slice2volume(index, axis[, shape])	Affine expressing selection of a single slice from 3D volume
vox2out_vox(mapped_voxels[, voxel_sizes])	output-aligned shape, affine for input implied by *mapped_voxels*

slice2volume

nibabel.spaces.**slice2volume**(*index*, *axis*, *shape=None*)

Affine expressing selection of a single slice from 3D volume

Imagine we have taken a slice from an image data array, `s = data[:, :, index]`. This function returns the affine to map the array coordinates of `s` to the array coordinates of `data`.

This can be useful for resampling a single slice from a volume. For example, to resample slice `k` in the space of `img1` from the matching spatial voxel values in `img2`, you might do something like:

```
slice_shape = img1.shape[:2]
slice_aff = slice2volume(k, 2)
whole_aff = np.linalg.inv(img2.affine).dot(img1.affine.dot(slice_aff))
```

and then use `whole_aff` in `scipy.ndimage.affine_transform`:

rzs, trans = to_matvec(whole_aff) data = img2.get_fdata() new_slice = scipy.ndimage.affine_transform(data, rzs, trans, slice_shape)

Parameters

index [int] index of selected slice

axis [{0, 1, 2}] axis to which *index* applies

Returns

slice_aff [shape (4, 3) affine] Affine relating input coordinates in a slice to output coordinates in the embedded volume

vox2out_vox

nibabel.spaces.**vox2out_vox**(*mapped_voxels*, *voxel_sizes=None*)

output-aligned shape, affine for input implied by *mapped_voxels*

The input (voxel) space, and the affine mapping to output space, are given in *mapped_voxels*.

The output space is implied by the affine, we don't need to know what that is, we just return something with the same (implied) output space.

Our job is to work out another voxel space where the voxel array axes and the output axes are aligned (top left 3 x 3 of affine is diagonal with all positive entries) and which contains all the voxels of the implied input image at their correct output space positions, once resampled into the output voxel space.

Parameters

mapped_voxels [object or length 2 sequence] If object, has attributes `shape` giving input voxel shape, and `affine` giving mapping of input voxels to output space. If length 2 sequence, elements are (shape, affine) with same meaning as above. The affine is a (4, 4) array-like.

voxel_sizes [None or sequence] Gives the diagonal entries of *output_affine* (except the trailing 1 for the homogenous coordinates) (`output_affine == np.diag(voxel_sizes + [1])`). If None, return identity *output_affine*.

Returns

output_shape [sequence] Shape of output image that has voxel axes aligned to original image output space axes, and encloses all the voxel data from the original image implied by input shape.

output_affine [(4, 4) array] Affine of output image that has voxel axes aligned to the output axes implied by input affine. Top-left 3 x 3 part of affine is diagonal with all positive entries. The entries come from *voxel_sizes* if specified, or are all 1. If the image is < 3D, then the missing dimensions will have a 1 in the matching diagonal.

viewers

Utilities for viewing images

Includes version of OrthoSlicer3D code originally written by our own Paul Ivanov.

OrthoSlicer3D(data[, affine, axes, title])	Orthogonal-plane slice viewer.

OrthoSlicer3D

class nibabel.viewers.**OrthoSlicer3D**(*data*, *affine=None*, *axes=None*, *title=None*)

Bases: object

Orthogonal-plane slice viewer.

OrthoSlicer3d expects 3- or 4-dimensional array data. It treats 4D data as a sequence of 3D spatial volumes, where a slice over the final array axis gives a single 3D spatial volume.

For 3D data, the default behavior is to create a figure with 3 axes, one for each slice orientation of the spatial volume.

Clicking and dragging the mouse in any one axis will select out the corresponding slices in the other two. Scrolling up and down moves the slice up and down in the current axis.

For 4D data, the fourth figure axis can be used to control which 3D volume is displayed. Alternatively, the – key can be used to decrement the displayed volume and the + or = keys can be used to increment it.

Examples

```
>>> import numpy as np
>>> a = np.sin(np.linspace(0, np.pi, 20))
>>> b = np.sin(np.linspace(0, np.pi*5, 20))
>>> data = np.outer(a, b)[..., np.newaxis] * a
>>> OrthoSlicer3D(data).show()
```

Parameters

data [array-like] The data that will be displayed by the slicer. Should have 3+ dimensions.

affine [array-like or None, optional] Affine transform for the data. This is used to determine how the data should be sliced for plotting into the sagittal, coronal, and axial view axes. If None, identity is assumed. The aspect ratio of the data are inferred from the affine transform.

axes [tuple of mpl.Axes or None, optional] 3 or 4 axes instances for the 3 slices plus volumes, or None (default).

title [str or None, optional] The title to display. Can be None (default) to display no title.

__init__(*data*, *affine=None*, *axes=None*, *title=None*)

Parameters

data [array-like] The data that will be displayed by the slicer. Should have 3+ dimensions.

affine [array-like or None, optional] Affine transform for the data. This is used to determine how the data should be sliced for plotting into the sagittal, coronal, and axial view axes. If None, identity is assumed. The aspect ratio of the data are inferred from the affine transform.

axes [tuple of mpl.Axes or None, optional] 3 or 4 axes instances for the 3 slices plus volumes, or None (default).

title [str or None, optional] The title to display. Can be None (default) to display no title.

property clim
The current color limits

close()
Close the viewer figures

property cmap
The current colormap

draw()
Redraw the current image

property figs
A tuple of the figure(s) containing the axes

link_to(*other*)
Link positional changes between two canvases

Parameters

other [instance of OrthoSlicer3D] Other viewer to use to link movements.

property n_volumes
Number of volumes in the data

property position
The current coordinates

set_position(*x=None*, *y=None*, *z=None*)
Set current displayed slice indices

Parameters

x [float | None] X coordinate to use. If None, do not change.

y [float | None] Y coordinate to use. If None, do not change.

z [float | None] Z coordinate to use. If None, do not change.

set_volume_idx(*v*)
Set current displayed volume index

Parameters

v [int] Volume index.

show()
Show the slicer in blocking mode; convenience for `plt.show()`

xmlutils

Thin layer around xml.etree.ElementTree, to abstract nibabel xml support.

XmlBasedHeader()	Basic wrapper around FileBasedHeader and XmlSerializable.
XmlParser([encoding, buffer_size, verbose])	Base class for defining how to parse xml-based image snippets.
XmlSerializable()	Basic interface for serializing an object to xml

XmlBasedHeader

class nibabel.xmlutils.**XmlBasedHeader**
Bases: *nibabel.filebasedimages.FileBasedHeader*, *nibabel.xmlutils.XmlSerializable*

Basic wrapper around FileBasedHeader and XmlSerializable.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

XmlParser

class nibabel.xmlutils.**XmlParser**(*encoding='utf-8'*, *buffer_size=35000000*, *verbose=0*)
Bases: object

Base class for defining how to parse xml-based image snippets.

Image-specific parsers should define: StartElementHandler EndElementHandler CharacterDataHandler

Parameters

encoding [str] string containing xml document

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

verbose [int, optional] amount of output during parsing (0=silent, by default).

__init__(*encoding='utf-8'*, *buffer_size=35000000*, *verbose=0*)

Parameters

encoding [str] string containing xml document

buffer_size: None or int, optional size of read buffer. None uses default buffer_size from xml.parsers.expat.

verbose [int, optional] amount of output during parsing (0=silent, by default).

CharacterDataHandler(*data*)

EndElementHandler(*name*)

HANDLER_NAMES = ['StartElementHandler', 'EndElementHandler', 'CharacterDataHandler']

StartElementHandler(*name*, *attrs*)

parse(*string=None*, *fname=None*, *fptr=None*)

Parameters

string [bytes] string (as a bytes object) containing xml document

fname [str] file name of an xml document.

fptr [file pointer] open file pointer to an xml documents

XmlSerializable

class nibabel.xmlutils.**XmlSerializable**

Bases: object

Basic interface for serializing an object to xml

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

to_xml(*enc='utf-8'*)

Output should be an xml string with the given encoding. (default: utf-8)

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X