NiBabel Documentation

Release 2.2.0dev

NiBabel Authors

March 31, 2017, 18:43 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, MINC1, MINC2, 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

The main authors of NiBabel are Matthew Brett, Michael Hanke, Ben Cipollini, Marc-Alexandre Côté, Chris Markiewicz, Stephan Gerhard and Eric Larson. 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
- Nguyen, Ly
- 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

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 2.7, or >= 3.4
- NumPy 1.6 or greater
- Six_ 1.3 or greater
- SciPy (optional, for full SPM-ANALYZE support)
- PyDICOM 0.9.7 or greater (optional, for DICOM support)
- Python Imaging Library (optional, for PNG conversion in DICOMFS)
- nose 0.11 or greater (optional, to run the tests)
- mock (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 `nosetests 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_data()
>>> 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 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. We won't do it here, but it looks like:

```
img.to_filename(os.path.join('build','test4d.nii.gz'))
```

or:

```
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 inage. 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
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, 2.1999991, 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_data()` method.

For the array image, `get_data()` just returns the data array:

```
>>> image_data = array_img.get_data()
>>> image_data.shape
(2, 3, 4)
>>> image_data is array_data
True
```

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

```
>>> image_data = img.get_data()
>>> 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_data()` on a proxy image. By default, when you call `get_data()` 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_data()`, the image returns the array from cache rather than loading it from disk again.

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

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.

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_data()` method:

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

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

```
>>> data_again = img.get_data()
>>> 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_data()
>>> img.in_memory
True
```

Using `uncache`

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

```
>>> data_again = img.get_data()
>>> 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_data()
>>> 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_data()` on proxy images, because `uncache` will then change the result you get back from `get_data()`:

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

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_data()  # new copy of array loaded
>>> data_once_more[0, 0, 0, 0]  # array modifications discarded
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_data()`

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_data()` and always use `np.asarray(img.dataobj)`.

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

The default behavior of the `get_data()` 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_data()  # default caching='fill'
>>> proxy_img.in_memory
True
```

is the same as this:

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

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

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

See the `get_data() 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_data()
>>> 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
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(1162.0, dtype=float32)
>>> n1_header['cal_max'] = 1200
>>> n1_header['cal_max']
array(1200.0, 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.  ]]), array(1, dtype=int16))
```

You can set the sform with with the `get_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.]]), array(4, dtype=int16))
```

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.  ]]), array(1, dtype=int16))
```

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.

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(2.0, dtype=float32)
>>> array_header['scl_inter']
array(10.0, 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_data()
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)
```

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_data()
...([[[ 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
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) 2009-2014 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) 2011-2014 Yaroslav Halchenko <debian@onerussian.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
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.

Ordereddict

In `nibabel/externals/ordereddict.py`

Copied from: https://pypi.python.org/packages/source/o/ordereddict/ordereddict-1.1.tar.gz#md5=a0ed854ee442051b249bfad0f638bbec

```
    FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    OTHER DEALINGS IN THE SOFTWARE.
```

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
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.

Six

In `nibabel/externals/six.py`

Copied from: https://pypi.python.org/packages/source/s/six/six-1.3.0.tar.gz#md5=ec47fe6070a8a64c802363d2c2b1e2ee

```
Copyright (c) 2010-2013 Benjamin Peterson

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 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.
```

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), Michael Hanke (MH) Ben Cipollini (BC), Marc-Alexandre Côté (MC), Chris Markiewicz (CM), Stephan Gerhard (SG) and Eric Larson (EL).

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

Upcoming Release

New features

- CIFTI support (pr/249) (Satra Ghosh, Michiel Cottaar, BC, CM, Demian Wassermann, MB)

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)

Bug fixes

- Miscellaneous MINC reader fixes (pr/493) (Robert D. Vincent, reviewed by CM, MB)

Maintenance

- 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

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 (SG)
- 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:

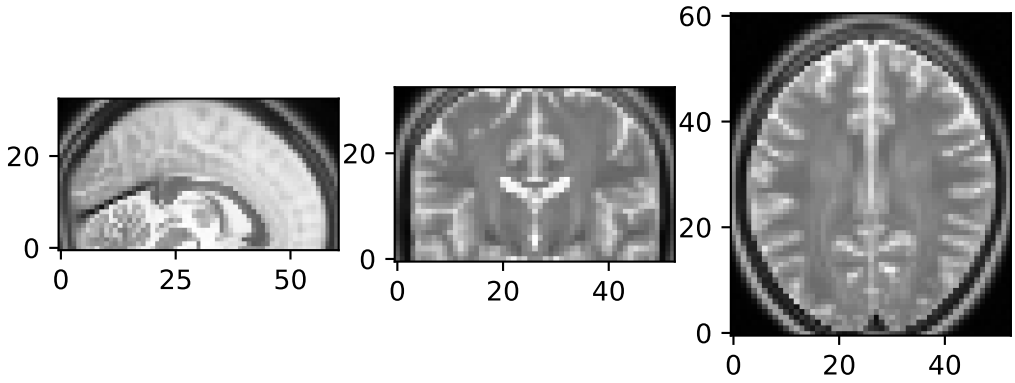
- `somones_epi.nii.gz`.
- `somones_anatomy.nii.gz`.

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_data()
>>> epi_img_data.shape
(53, 61, 33)
```

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

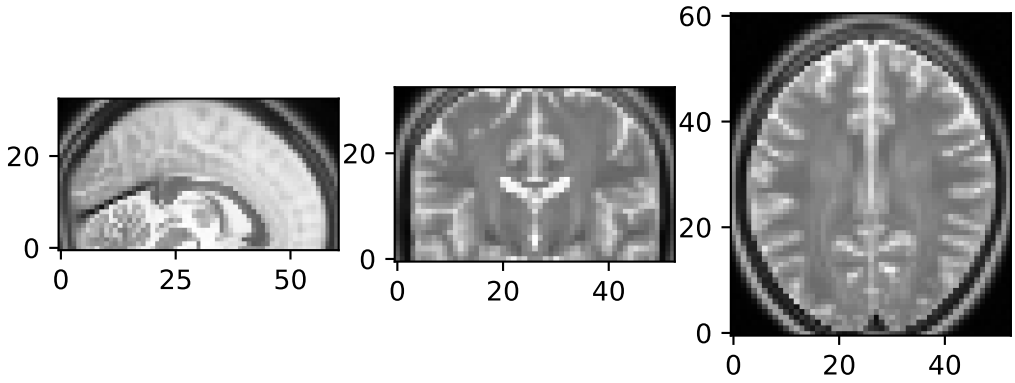
```
>>> import matplotlib.pyplot as plt
>>> def show_slices(slices):
...     """ Function to display row of image slices """
...     fig, axes = plt.subplots(1, len(slices))
...     for i, slice in enumerate(slices):
...         axes[i].imshow(slice.T, cmap="gray", origin="lower")
>>>
>>> slice_0 = epi_img_data[26,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

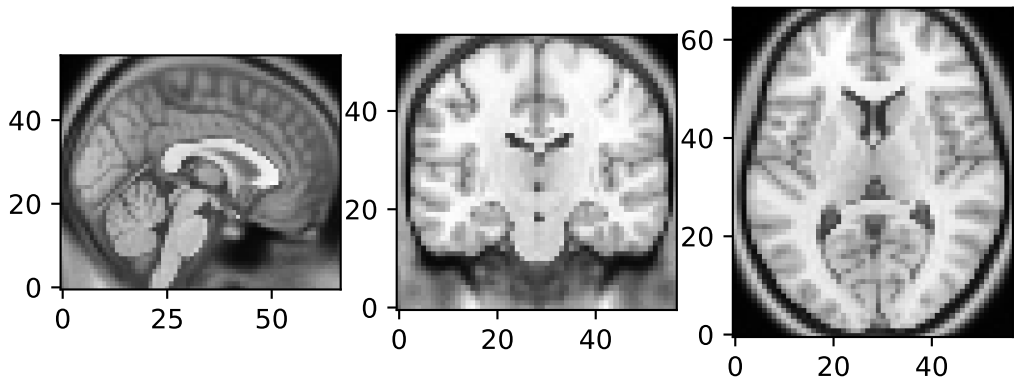


We collected an anatomical image in the same session. We can load that image and look at slices in the three axes:

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

Center slices for EPI 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:

```
>>> 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
81.549287796020508
```

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.

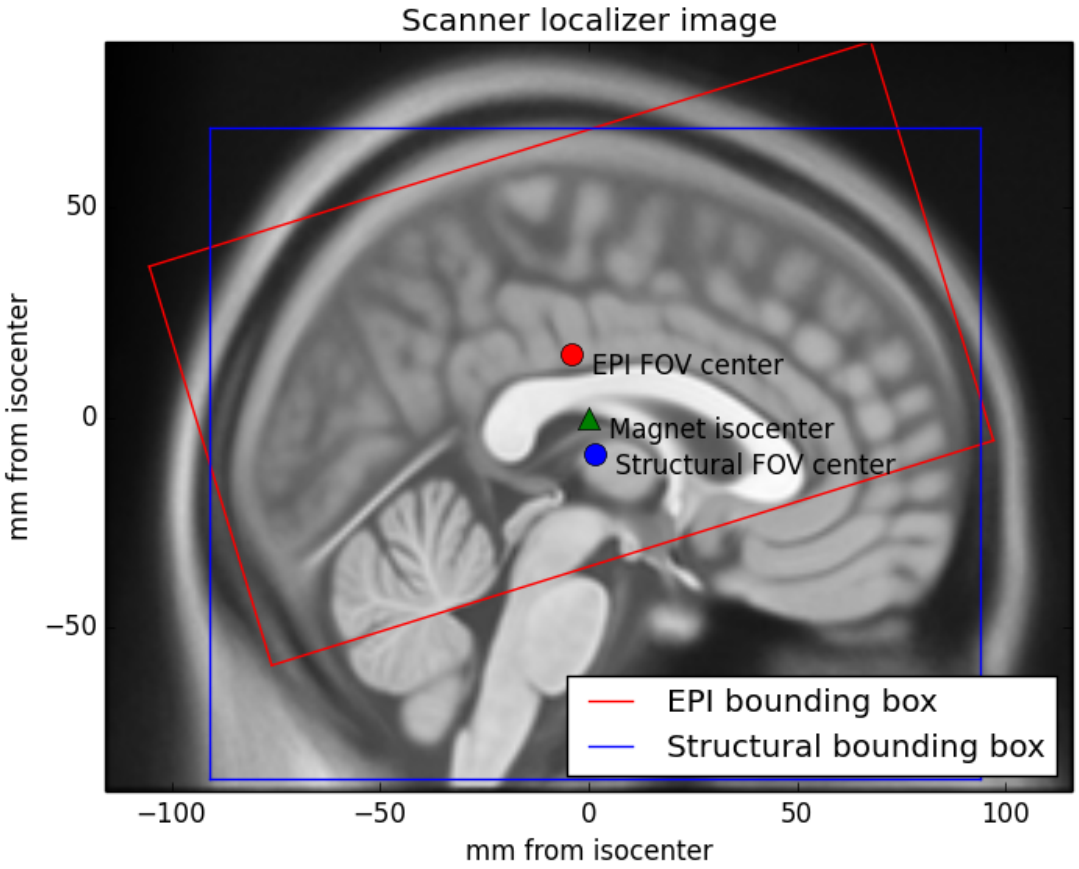
A coordinate is a set of numbers giving positions relative to a set of *axes*. In this case 26 is a position on the first array axis, where the axis is of length `epi_img_data.shape[0]`, and therefore goes from 0 to 52 (`epi_img_data.shape == (53, 61, 33)`). Similarly 30 gives a position on the second 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 anterior-posterior axis of the head and scanner-bore is the inferior-posterior 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 anterior-posterior 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 (anterior-posterior in the subject), and the Z axis would be scanner-bore (inferior-posterior).

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, Posterior 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])
```

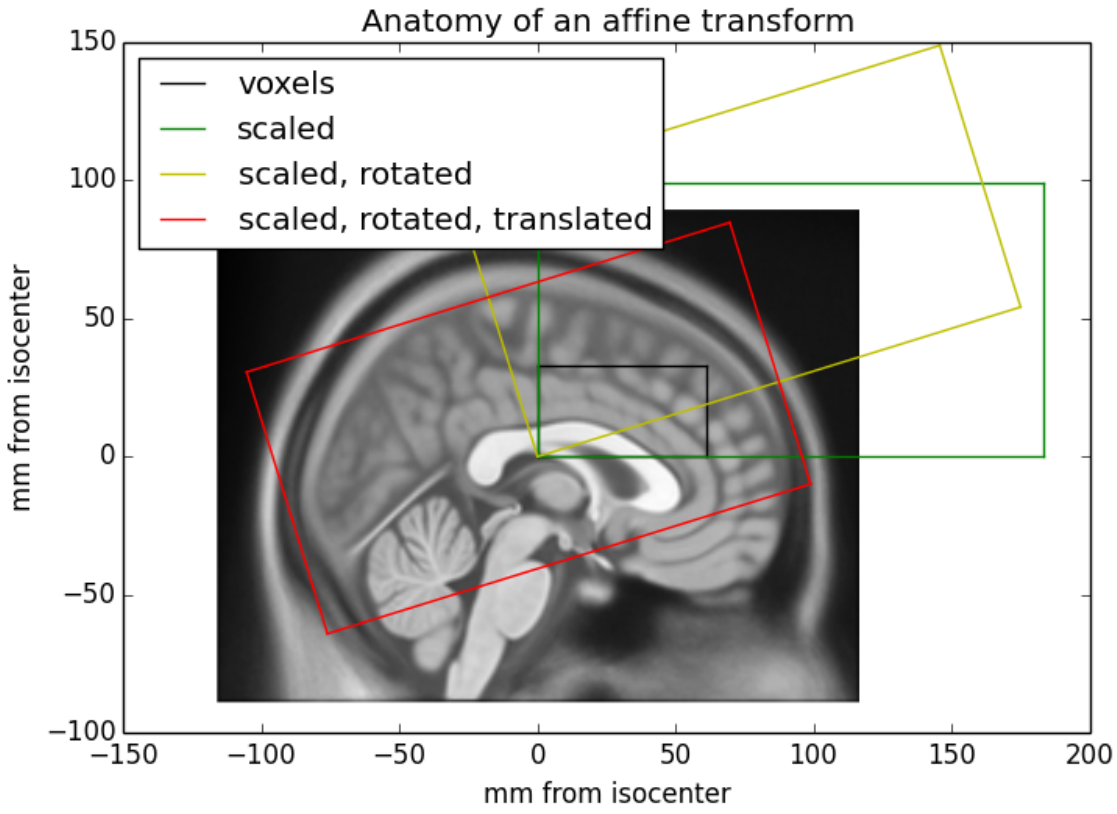
The voxel coordinate of the center voxel of the anatomical image is:

```
>>> 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)
>>> 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*.

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
>>> 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+".

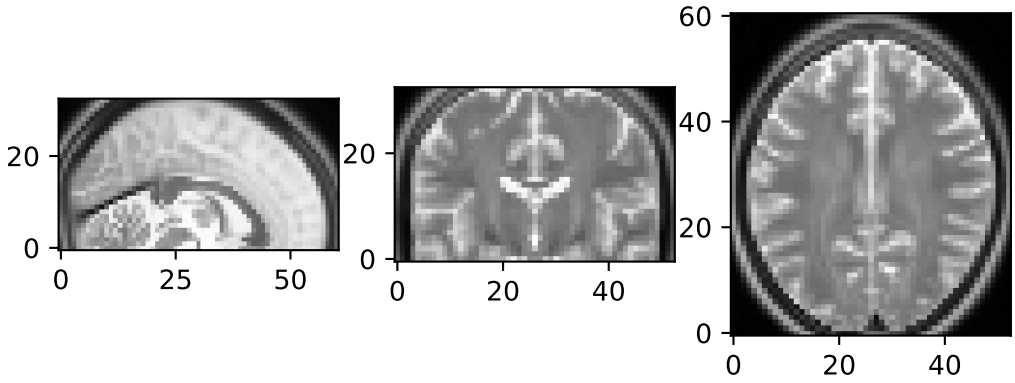
Some people therefore refer to this alignment of voxel and RAS+ axes as *RAS voxel axes*.

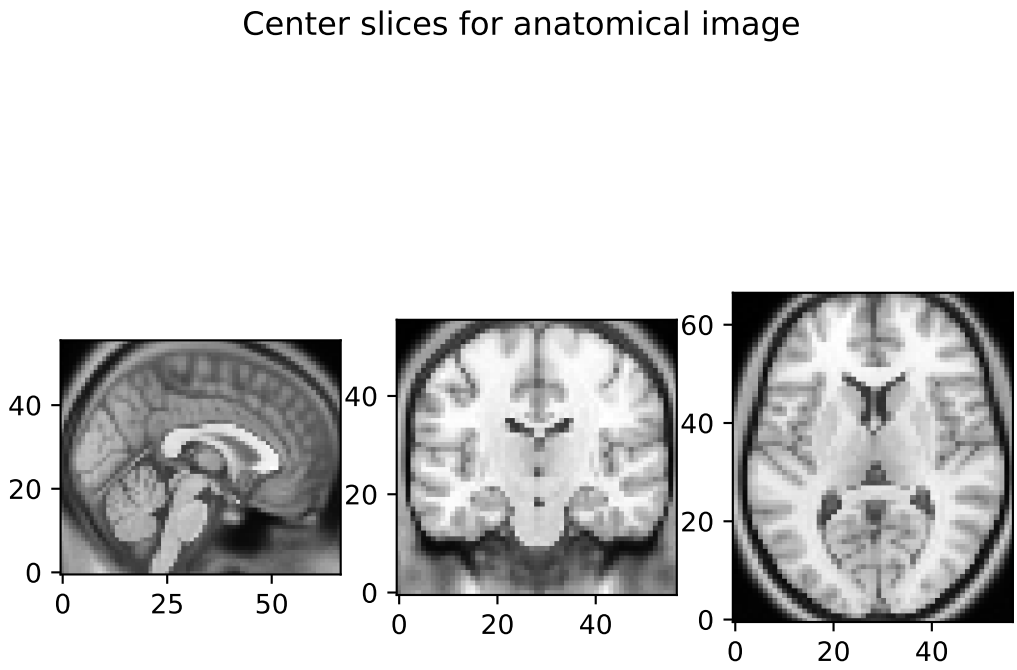
Neurological / radiological voxel layout

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_data()
>>> 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")
```

Center slices for EPI image





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 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.

[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.

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.

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.

[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)

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)		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)		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)		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

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.

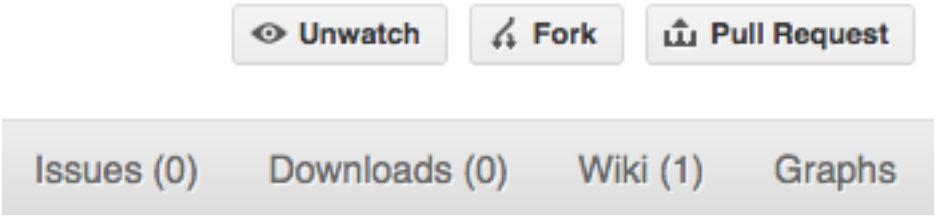
Set up and configure a github account

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"
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]
| |\
| |/
```

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)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

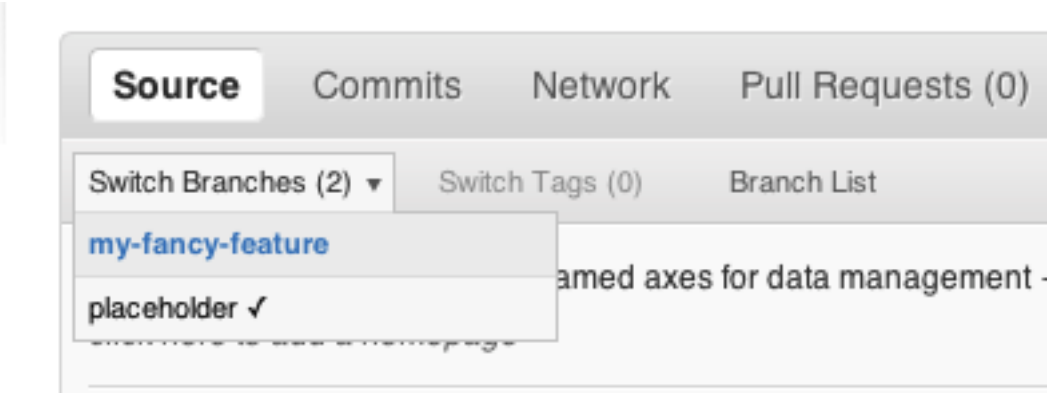
3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just 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

All documentation should be written using Numpy documentation conventions:

https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt#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.

10.3.2 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')

@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.

10.3.3 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.

10.3.4 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.
 ____________________________________________________________________________

  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
    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)
>> 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 =

    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_data()
data[:] = 0
img2 = nibabel.load('some_image.nii')
assert not np.all(img2.get_data() == img.get_data())
```

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_data()`, 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.

10.3.5 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):

```
nosetests --with-doctest 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'}
/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)](http://dx.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.

10.3.6 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 `nosetests`.

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
- http://fixunix.com/dicom/50449-image-position-patient-image-orientation-patient.html

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:

$$\begin{pmatrix} T^N \\ 1 \end{pmatrix} = A \begin{pmatrix} 0 \\ 0 \\ -1+N \\ 1 \end{pmatrix}$$

From this it follows that:

$$\left\{ k_1 : \frac{T_1^1 - T_1^N}{1-N}, \quad k_2 : \frac{T_2^1 - T_2^N}{1-N}, \quad k_3 : \frac{T_3^1 - T_3^N}{1-N} \right\}$$

and therefore:

3D affine formulae

$$A_{multi} = \begin{pmatrix} F_{11}\Delta r & F_{12}\Delta c & \frac{T_1^1 - T_1^N}{1-N} & T_1^1 \\ F_{21}\Delta r & F_{22}\Delta c & \frac{T_2^1 - T_2^N}{1-N} & T_2^1 \\ F_{31}\Delta r & F_{32}\Delta c & \frac{T_3^1 - T_3^N}{1-N} & T_3^1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_{single} = \begin{pmatrix} F_{11}\Delta r & F_{12}\Delta c & \Delta s n_1 & T_1^1 \\ F_{21}\Delta r & F_{22}\Delta c & \Delta s n_2 & T_2^1 \\ F_{31}\Delta r & F_{32}\Delta c & \Delta s n_3 & T_3^1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

See `derivations/spm_dicom_orient.py` for the derivations and some explanations.

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

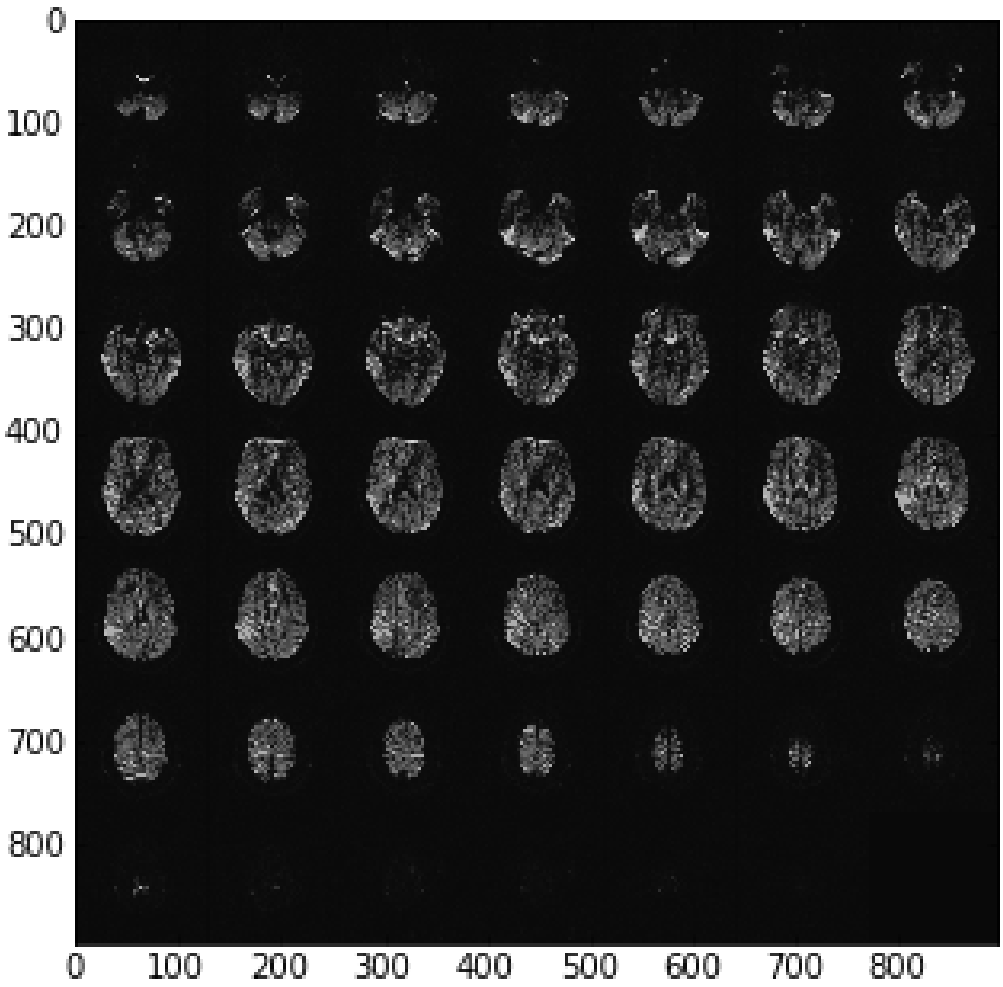
- Number of Frames (0028,0008) - Number of frames in a Multi-frame Image.
- Frame Increment Pointer (0028,0009) - Contains the Data Element Tag of the attribute that is used as the frame increment in Multi-frame pixel data.

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 imagqes 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:
   (a) Seeks to beginning of file
   (b) Reads two unsigned short values into `group` and `tag`
   (c) If the (`group`, `element`) pair exist in `spm_dicom_dict.mat`, then set file pointer to 0 and continue read with `read_dicom` subfunction..
   (d) 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.
   (e) 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:
   (a) SeriesNumber
   (b) Rows
   (c) Columns
   (d) ImageOrientationPatient (to tolerance of sum squared difference 1e-4)
   (e) PixelSpacing (to tolerance of sum squared difference 1e-4)
   (f) ICE dims as defined above
   (g) ImageType (iff imagetype exists in both)zv
   (h) SequenceName (iff sequencename exists in both)
   (i) SeriesInstanceUID (iff exists in both)
   (j) 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`:
   (a) Find first header for which corresponding `is_processed` is False - call this `hdr_to_check`
   (b) Collect indices (in `work_list`) of headers which have the same z coordinate as `hdr_to_check`, call this list `z_same_indices`.
   (c) Sort `work_list[z_same_indices]` by 'InstanceNumber'
   (d) 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.
   (e) 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:
   (a) StudyDate/Time
   (b) SeriesNumber
   (c) AcquisitionNumber
   (d) 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 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
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
trackvis	Read and write trackvis files (old interface)

10.5.2 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

10.5.3 Float / integer conversion

arraywriters	Array writer objects
casting	Utilties for casting numpy values in various ways

10.5.4 System utilities

data	Utilities to find files from NIPY data packages
environment	Settings from the system environment relevant to NIPY

10.5.5 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

10.5.6 Alphabetical API reference