PyNIfTI Manual

Release 0.20090303.1

Michael Hanke

April 05, 2011

CONTENTS

The PDF version of the manual is available for download.

CHAPTER

ONE

WHAT IS NIFTI AND WHAT DO I NEED PYNIFTI FOR?

1.1 NIfTI

NIfTI is a new Analyze-style data format, proposed by the NIfTI Data Format Working Group as a *"short-term measure to facilitate inter-operation of functional MRI data analysis software packages"*. Meanwhile a number of toolkits are NIfTI-aware (e.g. FSL, AFNI, SPM, Freesurfer and a to a certain degree also Brainvoyager). Additionally, dicomnifti allows the direct conversion from DICOM images into the NIfTI format.

With libnifti there is a reference implementation of a C library to read, write and manipulate NIfTI images. The library source code is put into the public domain and a corresponding project is hosted at SourceForge.

In addition to the C library, there is also an IO library written in Java and Matlab functions to make use of NIfTI files from within Matlab.

1.2 Python

Unfortunately, it is not that trivial to read NIfTI images with Python. This is particularly sad, because there is a large number of easy-to-use, high-quality libraries for signal processing available for Python (e.g. SciPy).

Moreover Python has bindings to almost any important language/program in the fields of maths, statistics and/or engineering. If you want to use R to calculate some stats in a Python script, simply use RPy and pass any data to R. If you don't care about R, but Matlab is your one and only friend, there are at least two different Python modules to control Matlab from within Python scripts. Python is the glue between all those helpers and the Python user is able to combine as many tools as necessary to solve a given problem – the easiest way.

1.3 PyNIfTI

PyNIfTI aims to provide easy access to NIfTI images from within Python. It uses SWIG-generated wrappers for the NIfTI reference library and provides the `NiftiImage` class for Python-style access to the image data.

While PyNIfTI is not yet complete (i.e. doesn't support everything the C library can do), it already provides access to the most important features of the NIfTI-1 data format and *libniftiio* capabilities. The following features are currently implemented:

- PyNIfTI can read and write any file format supported by libniftiio. This includes NIfTI (single and pairs) as well as ANALYZE files, both also in gzipped versions.
- PyNIfTI provides fast and convenient access to the image data via NumPy arrays. This should enable users to process image data with most (if not all) numerical routines available for Python. The NumPy array automatically uses a datatype corresponding to the NIfTI image data – no unnecessary upcasting is performed.
- PyNIfTI provides full read and write access to the NIfTI header data. Header information can be exported to a Python dictionary and can also be updated by using information from a dictionary.
- Besides accessing NIfTI data from files, PyNIfTI is able to create NIfTI images from NumPy arrays. The appropriate NIfTI header information is determined from the array properties. Additional header information can be optionally specified – making it easy to clone NIfTI images if necessary, but with minor modifications.
- Most properties of NIfTI images are accessible via attributes and/or accessor functions of the `NiftiImage`. Inter-dependent properties are automatically updated if necessary (e.g. modifying the Q-Form matrix also updates the pixdim properties and quaternion representation).
- All properties are accessible via Python-style datatypes: A 4x4 matrix is an array not 16 individual numbers.
- PyNIfTI should be resonably fast. Image data will only be loaded into the memory if necessary. Simply opening a NIfTI file to access some header data is performed with virtually no delay independent of the size of the image. Unless image resizing or datatype conversion must be performed the image data can be shared by the NIfTI image and accessing NumPy arrays, and therefore memory won't be wasted memory with redundant copies of the image data.
- Additionally PyNIfTI can access uncompressed NIfTI or ANALYZE files by providing memory-mapped access to them via NumPy's memmap arrays. In this mode it is possible to modified existing files of any size without having to load them in memory first.
- PyNIfTI allows to embed arbitrary additional information into the NIfTI file header.

1.4 Scripts

Some functions provided by PyNIfTI also might be useful outside the Python environment and it might be useful to export them via some command line scripts.

Currently there is only one: `pynifti_pst` (pst: peristimulus timecourse). Using this script one can compute the signal timecourse for a certain condition for all voxels in a volume at once. Although it is done by simply averaging the timecourses of the involved events (nothing fancy), this might nevertheless be useful for exploring a dataset and accompanies similar tools like FSL's `tsplot`. The output of `pynifti_pst` can be loaded into FSLView to simultaneously look at statistics and signal timecourses. Please see the corresponding example below.

1.5 Known issues

PyNIfTI currently ignores the origin field of ANALYZE files - it is neither read nor written. A possible workaround is to convert ANALYZE files into the NIfTI format using FSL's `fslchfiletype`.

1.6 Things to know

When accessing NIfTI image data through NumPy arrays the order of the dimensions is reversed. If the *x, y, z, t* dimensions of a NIfTI image are 64, 64, 32, 456 (as for example reported by `nifti_tool`), the shape of the NumPy array (e.g. as returned by `NiftiImage.data`) will be: 456, 32, 64, 64.

This is done to be able to slice the data array much easier in the most common cases. For example, if you are interested in a certain volume of a timeseries it is much easier to write `data[2]` instead of `data[:,:,:,2]`, right?

CHAPTER

TWO

INSTALLATION

It should be fairly easy to get PyNIfTI running on any system. For the most popular platforms and operating systems there are binary packages provided in the respective native packaging format (DEB, RPM or installers). On all other systems PyNIfTI has to be compiled from source – which should also be pretty straightforward.

2.1 Binary packages

2.1.1 GNU/Linux

PyNIfTI is available in recent versions of the Debian (since lenny) and Ubuntu (since gutsy in universe) distributions. The name of the binary package is `python-nifti` in both cases.

- PyNIfTI versions in Debian
- PyNIfTI versions in Ubuntu

Binary packages for some additional Debian and (K)Ubuntu versions are also available. Please visit Michael Hanke's APT repository to read about how you have to setup your system to retrieve the PyNIfTI package via your package manager and stay in sync with future releases.

If you are using Debian lenny (or later) or Ubuntu gutsy (or later) or you have configured your system for Michael Hanke's APT repository all you have to do to install PyNIfTI is this:

```
apt-get update
apt-get install python-nifti
```

This should pull all necessary dependencies. If it doesn't, it's a bug that should be reported. Additionally, there are binary packages for several RPM-based distributions, provided through the OpenSUSE Build Service. To install one of these packages first download it from the OpenSUSE software website. Please note, that this site does not only offer OpenSUSE packages, but also binaries for other distributions, including: CentOS 5, Fedora 9-10, Mandriva 2007-2008, RedHat Enterprise Linux 5, SUSE Linux Enterprise 10, OpenSUSE 10.2 up to 11.0. Once downloaded, open a console and invoke (the example command refers to PyMVPA 0.3.1):

```
rpm -i python-nifti-0.20080710.1-4.1.i386.rpm
```

The OpenSUSE website also offers 1-click-installations for distributions supporting it.

A more convenient way to install PyNIfTI and automatically receive software updates is to included one of the **'RPM-package repositories'_** in the system's package management configuration. For e.g. OpenSUSE 11.0, simply use Yast to add another repository, using the following URL:

http://download.opensuse.org/repositories/home:/hankem/openSUSE_11.0/

For other distributions use the respective package managers (e.g. Yum) to setup the repository URL. The repositories include all core dependencies of PyNIfTI, if they are not available from other repositories of the respective distribution. There are two different repository groups, one for Suse and Mandriva-related packages and another one for Fedora, Redhat and CentOS-related packages.

2.1.2 Windows

A binary installer for a recent Python version is available from the nifticlibs Sourceforge project site.

There are a few Python distributions for Windows. In theory all of them should work equally well. However, I only tested the standard Python distribution from www.python.org (with version 2.5.2).

First you need to download and install Python. Use the Python installer for this job. Yo do not need to install the Python test suite and utility scripts. From now on we will assume that Python was installed in *C:\Python25* and that this directory has been added to the *PATH* environment variable.

In addition you'll need NumPy. Download a matching NumPy windows installer for your Python version (in this case 2.5) from the SciPy download page and install it.

PyNIfTI does not come with the required *zlib* library, so you also need to download and install it. A binary installer is available from the GnuWin32 project. Install it in some arbitrary folder (just the binaries nothing else), find the *zlib1.dll* file in the *bin* subdirectory and move it in the Windows *system32* directory.

Now, you can use the PyNIfTI windows installer to install PyNIfTI on your system. As always: click *Next* as long as necessary and finally *Finish*. If done, verify that everything went fine by opening a command promt and start Python by typing *python* and hit enter. Now you should see the Python prompt. Import the nifti module, which should cause no error messages:

```
>>> import nifti
>>>
```

2.1.3 MacOS X

The easiest installation method for OSX is via MacPorts. MacPorts is a package management system for MacOS, which is in some respects very similiar to RPM or APT which are used in most GNU/Linux distributions. However, rather than installing binary packages, it compiles software from source on the target machine.

The MacPort of PyNIfTI is kindly maintained by James Kyle <jameskyle@ucla.edu>.

In the context of PyNIfTI MacPorts is much easier to handle than the previously available installer for Macs. Although the initial overhead to setup MacPorts on a machine is higher than simply installing PyNIfTI using the former installer, MacPorts saves the user a significant amount of time (in the long run). This is due to the fact that this framework will not only take care of updating a PyNIfTI installation automatically whenever a new release is available. It will also provide many of the optional dependencies of PyNIfTI (e.g. NumPy, nifticlibs) in the same environment and therefore abolishes the need to manually check dozens of websites for updates and deal with an unbelievable number of different installation methods.

MacPorts provides a universal binary package installer that is downloadable at http://www.macports.org/install.php

After downloading, simply mount the dmg image and double click *MacPorts.pkg*.

By default, MacPorts installs to */opt/local*. After the installation is completed, you must ensure that your paths are set up correctly in order to access the programs and utilities installed by MacPorts. For exhaustive details on editing shell paths please see:

http://www.debian.org/doc/manuals/reference/ch-install.en.html#s-bashconf

A typical *.bash_profile* set up for MacPorts might look like:

```
> export PATH=/opt/local/bin:/opt/local/sbin:$PATH
> export DYLD_LIBRARY_PATH=/opt/local/lib:$DYLD_LIBRARY_PATH
```

Be sure to source your .bash_profile or close Terminal.app and reopen it for these changes to take effect.

Once MacPorts is installed and your environment is properly configured, PyNIfTI is installed using a single command:

```
> $ sudo port install py25-pynifti
```

If this is your first time using MacPorts Python 2.5 will be automatically installed for you. However, an additional step is needed:

```
$ sudo port install python_select
$ sudo python_select python25
```

MacPorts has the ability of installing several Python versions at a time, the *python_select* utility ensures that the default Python (located at */opt/local/bin/python*) points to your preferred version.

Upon success, open a terminal window and start Python by typing *python* and hit return. Now try to import the PyNIfTI module by doing:

```
>>> import nifti
>>>
```

If no error messages appear, you have succesfully installed PyNIfTI.

2.2 Compile from source

If no binary packages are provided for your platfom, you can build PyNIfTI from source. It needs a few things to build and run properly:

- Python 2.4 or greater
- NumPy
- SWIG 1.3.29 (or later)

  NIfTI C libraries
  : Proper developer packages are prefered, but for convenience reasons a minimal copy is included in the PyNIfTI source package.

2.2.1 Get the sources

Since June 2007 PyNIfTI is part of the niftilibs family. The source code of PyNIfTI releases can be obtained from the corresponding Sourceforge project site. Alternatively, one can also download a tarball of the latest development snapshot (i.e. the current state of the *master* branch of the PyNIfTI source code repository).

If you want to have access to both, the full PyNIfTI history and the latest development code, you can use the PyNIfTI Git repository on the Alioth server, a service kindly provided by the Debian project. To view the repository, please point your web browser to gitweb:

- http://git.debian.org/?p=pkg-exppsy/pynifti.git

The gitweb browser also allows to download arbitrary development snapshots of PyNIfTI. For a full clone (aka checkout) of the PyNIfTI repository simply do:

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

2.2.2 Compiling: General instructions

Make sure that the compiled nifticlibs and the corresponding headers are available to your compiler. If they are located in a custom directory, you might have to specify `--include-dirs` and `--library-dirs` options to the build command below. In case, you want to build and use the nifticlibs copy that is shipped with PyNIfTI, this is automatically done for you.

Once you have downloaded the sources, extract the tarball and enter the root directory of the extracted sources. If you *do not* have the nifticlibs installed, run:

```
make
```

in the root of the extracted source tarball. If you have system-wide installed nifticlibs available on your system, instead simply do:

```
python setup.py build
```

That should build the SWIG wrappers. If this has been done successfully, all you need to do is install the modules by invoking:

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

If sudo is not configured (or even installed) you might have to use `su` instead.

Now fire up Python and try importing the module to see if everything is fine. It should look similar to this:

```
Python 2.4.4 (#2, Oct 20 2006, 00:23:25)
[GCC 4.1.2 20061015 (prerelease) (Debian 4.1.1-16.1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import nifti
>>>
```

2.2.3 Building on Windows Systems

On Windows the whole situation is a little more tricky, as the system doesn't come with a compiler by default. Nevertheless, it is easily possible to build PyNIfTI from source. One could use the Microsoft compiler that comes with Visual Studio to do it, but as this is commercial software and not everybody has access to it, I will outline a way that exclusively involves free and open source software.

First one needs to install the Python and NumPy, if not done yet. Please refer to the installation intructions for the Windows binary package below.

Next we need to obtain and install the MinGW compiler collection. Download the *Automated MinGW Installer* from the MinGW project website. Now, run it and choose to install the *current* package. You will need the *MinGW base tools*, *gcc* **and** *g++* compiler and *MinGW Make*. For the remaining parts of the section, we will assume that MinGW got installed in *C:\MinGW* and the directory *C:\MinGW\bin* has been added to the *PATH* environment variable, to be able to easily access all MinGW tools. Note, that it is not necessary to install MSYS to build PyNIfTI, but it might handy to have it.

In addition, PyNIfTI needs the developer version of the *zlib* library, so you also need to download and install it. A binary installer is available from the GnuWin32 project. It is best to install it into the same directory as MinGW (i.e. *C:\MinGW* in this example), as all paths will be automatically configured properly.

You also need to download SWIG (actually *swigwin*, the distribution for Windows). SWIG does not have to be installed, just unzip the file you downloaded and add the root directory of the extracted sources to the *PATH* environment variable (make sure that this directory contains *swig.exe*, if not, you haven't downloaded *swigwin*).

Now, we are ready to build PyNIfTI. The easiest way to do this, is to make use of the *Makefile.win* that is shipped with PyNIfTI to build a binary installer package (*.exe*). Make sure, that the settings at the top of *Makefile.win* (the file is located in the root directory of the source distribution) correspond to your Python installation – if not, first adjust them accordingly before your proceed. When everything is set, do:

```
mingw32-make -f Makefile.win installer
```

Upon success you can find the installer in the *dist* subdirectory. Install it as described below.

2.2.4 MacOS X

Since the MacPorts system basically compiles from source there should be no need to perform this step manually. However, if one intends to compile without MacPorts the XCode developer tools, have to be installed first, as the operating system does not come with a compiler by default. If you want to use or even work on the latest development code, you should also install Git. There is a MacOS installer for Git, that make this step very easy.

Otherwise follow the *general build instructions*.

2.2.5 MacOS X and MacPython

When you are comiling PyNIfTI on MacOS X and want to use it with MacPython, please make sure that the NIfTI C libraries are compiled as fat binaries (compiled for both *ppc* and *i386*). Otherwise PyNIfTI extensions will not compile.

One can achieve this by adding both architectures to the `CFLAGS` definition in the toplevel Makefile of the NIfTI C library source code or in the file *3rd/nifticlibs/Makefile* if you are using the nifticlibs copy that is shipped with the PyNIfTI sources. Like this:

```
CFLAGS=-Wall -O2 -I. -DHAVE_ZLIB -arch ppc -arch i386
```

2.3 Troubleshooting

If you get an error when importing the `nifti` module in Python complaining about missing symbols your niftiio library contains references to some unresolved symbols. Try adding `znzlib` and `zlib` to the linker options the PyNIfTI `setup.py`, like this:

```
libraries = [ 'niftiio', 'znz', 'z' ],
```

CHAPTER

THREE

EXAMPLES

The next sections contains some examples showing ways to use PyNIfTI to read and write imaging data from within Python to be able to process it with some random Python library.

All examples assume that you have imported the PyNIfTI module by invoking:

```
>>> from nifti import *
```

3.1 Loading and saving NIfTI files

First we will open the tiny example NIfTI file that is included in the PyNIfTI source tarball. No filename extension is necessary as libniftiio determines it automatically:

```
>>> nim = NiftiImage('example4d')
```

The filename is available via the 'filename' attribute:

```
>>> print nim.filename
example4d.nii.gz
```

This indicates a compressed NIfTI image. If you want to save this image as an uncompressed image simply do:

```
>>> nim.save('something.nii')
```

The filetype is determined from the filename. If you want to save to gzipped ANALYZE file pairs instead the following would be an alternative to calling the `save()` with a new filename:

```
>>> nim.setFilename('analyze.img.gz')
>>> nim.save()
```

Please see the documentation of `setFilename()` to learn how the filetypes are determined from the filenames.

3.2 NIfTI files from array data

The next code snipped demonstrates how to create a 4d NIfTI image containing gaussian noise. First we need to import the NumPy module

```
>>> import numpy as N
```

Now we generate the noise dataset. Let's generate noise for 100 volumes with 16 slices and a 32x32 inplane matrix.

```
>>> noise = N.random.randn(100,16,32,32)
```

Please notice the order in which the dimensions are specified: (t, z, y, x).

The datatype of the array is by default *float64*, which can be verified by:

```
>>> noise.dtype
dtype('float64')
```

Converting this dataset into a NIfTI image is done by invoking the `NiftiImage` constructor with the noise dataset as argument:

```
>>> nim = NiftiImage(noise)
```

The relevant header information is extracted from the NumPy array. If you query the header information about the dimensionality of the image, it returns the desired values:

```
>>> print nim.header['dim']
[4, 32, 32, 16, 100, 1, 1, 1]
```

First value shows the number of dimensions in the datset: 4 (good, that's what we wanted). The following numbers are dataset size on the x, y, z, t, u, v, w axis (NIfTI files can handle up to 7 dimensions). Please notice, that the order of dimensions is now 'correct': We have 32x32 inplane resolution, 16 slices in z direction and 100 volumes.

Also the datatype was set appropriately:

```
>>> import nifti.clib as ncl
>>> nim.header['datatype'] == ncl.NIFTI_TYPE_FLOAT64
True
```

To save the noise file to disk, we can simply call the `save()` method:

```
>>> nim.save('noise.nii.gz')
```

3.3 Select ROIs

Suppose you want to have the first ten volumes of the noise dataset we have previously created in a separate file. First, we open the file:

```
>>> nim = NiftiImage('noise.nii.gz')
```

Now we select the first ten volumes and store them to another file, while preserving as much header information as possible

```
>>> nim2 = NiftiImage(nim.data[:10], nim.header)
>>> nim2.save('part.hdr.gz')
```

The `NiftiImage` constructor takes a dictionary with header information as an optional argument. Settings that are not determined by the array (e.g. size, datatype) are taken from the dictionary and stored to the new NIfTI image.

3.4 Linear detrending of timeseries (SciPy module is required for this example)

Let's load another 4d NIfTI file and perform a linear detrending, by fitting a straight line to the timeseries of each voxel and substract that fit from the data. Although this might sound complicated at first, thanks to the excellent SciPy module it is just a few lines of code. For this example we will first create a NIfTI image with just a single voxel and 50 timepoints (basically a linear function with some noise):

```
>>> nim = NiftiImage(
...             (N.linspace(0,100) + N.random.randn(50)).reshape(50,1,1,1))
>>> nim.timepoints
50
>>> nim.volextent
(1, 1, 1)
```

Depending on the datatype of the input image the detrending process might change the datatype from integer to float. As operations that change the (binary) size of the NIfTI image are not supported, we need to make a copy of the data and later create a new NIfTI image. Remember that the array has the time axis as its first dimension (in contrast to the NIfTI file where it is the 4th).

```
>>> from scipy import signal
>>> data_detrended = signal.detrend(nim.data, axis=0)
```

Finally, create a new NIfTI image using header information from the original source image.

```
>>> nim_detrended = NiftiImage( data_detrended, nim.header)
```

3.5 Make a quick plot of a voxel's timeseries (matplotlib module is required)

Plotting is essential to get a 'feeling' for the data. The Matlab-style plotting via matplotlib make it really easy to plot something with (e.g. when running Python interactively via IPython).

Please note, that there are many other possibilities for plotting, e.g. R via RPy or Gnuplot via the Gnuplot python bindings

However, using matplotlib is really easy. For this example we will plot the two timeseries from the previous example, i.e. the raw and the detrended one. First we import the pylab module:

```
>>> import pylab as P
```

Now we can easly plot both timeseries of the single voxel in our artifical image:

```
>>> line1 = P.plot(nim.data[:, 0, 0, 0])
>>> line2 = P.plot(nim_detrended.data[:, 0, 0, 0,])
```

A *P.show()* call would render the plot on the screen.

3.6 Show a slice of a 3d volume (Matplotlib module is required)

This example demonstrates howto use the Matlab-style plotting of matplotlib to view a slice from a 3d volume. We will actually use a 4D image as data source and limit us to the first volume:

```
>>> nim = NiftiImage('example4d')
>>> volume = nim.data[0]
```

If everything went fine, we can now view a slice (x,y):

```
>>> xyplot = P.imshow(volume[16],
...                   interpolation='nearest',
...                   cmap=P.cm.gray)
```

Again a call to the *P.show()* function would render the plot on the screen.

When you want to have a look at a yz-slice, NumPy array magic comes into play.

```
>>> yzplot = P.imshow(volume[::-1,:,18],
...                   interpolation='nearest',
...                   cmap=P.cm.gray)
```

The `::-1` notation causes the z-axis to be flipped in the images. This makes a much nicer view, if the used example volume has the z-axis originally oriented upsidedown.

3.7 Compute and display peristimulus signal time-course of multiple conditions

Sometimes one wants to look at the signal timecourse of some voxel after a certain stimulation onset. An easy way would be to have some fMRI data viewer that displays a statistical map and one could click on some activated voxel and the peristimulus signal timecourse of some condition in that voxel would be displayed.

This can easily be done by using `pynifti_pst` and `FSLView`.

`pynifti_pst` comes with a manpage that explains all options and arguments. Basically `pynifti_pst` needs a 4d image (e.g. an fMRI timeseries; possibly preprocessed/filtered) and some stimulus onset information. This information can either be given directly on the command line or is read from files. Additionally one can specify onsets as volume numbers or as onset times.

`pynifti_pst` understands the FSL custom EV file format so one can easily use those files as input.

An example call could look like this:

```
pynifti_pst --times --nvols 5 -p uf92.feat/filtered_func_data.nii.gz \
    pst_cond_a.nii.gz uf92.feat/custom_timing_files/ev1.txt \
    uf92.feat/custom_timing_files/ev2.txt
```

This computes a peristimulus timeseries using the preprocessed fMRI from a FEAT output directory and two custom EV files that both together make up condition A. `--times` indicates that the EV files list onset times (not volume ids) and `--nvols` requests the mean peristimulus timecourse for 4 volumes after stimulus onset (5 including onset). `-p` recodes the peristimulus timeseries into percent signalchange, where the onset is always zero and any following value is the signal change with respect to the onset volume.

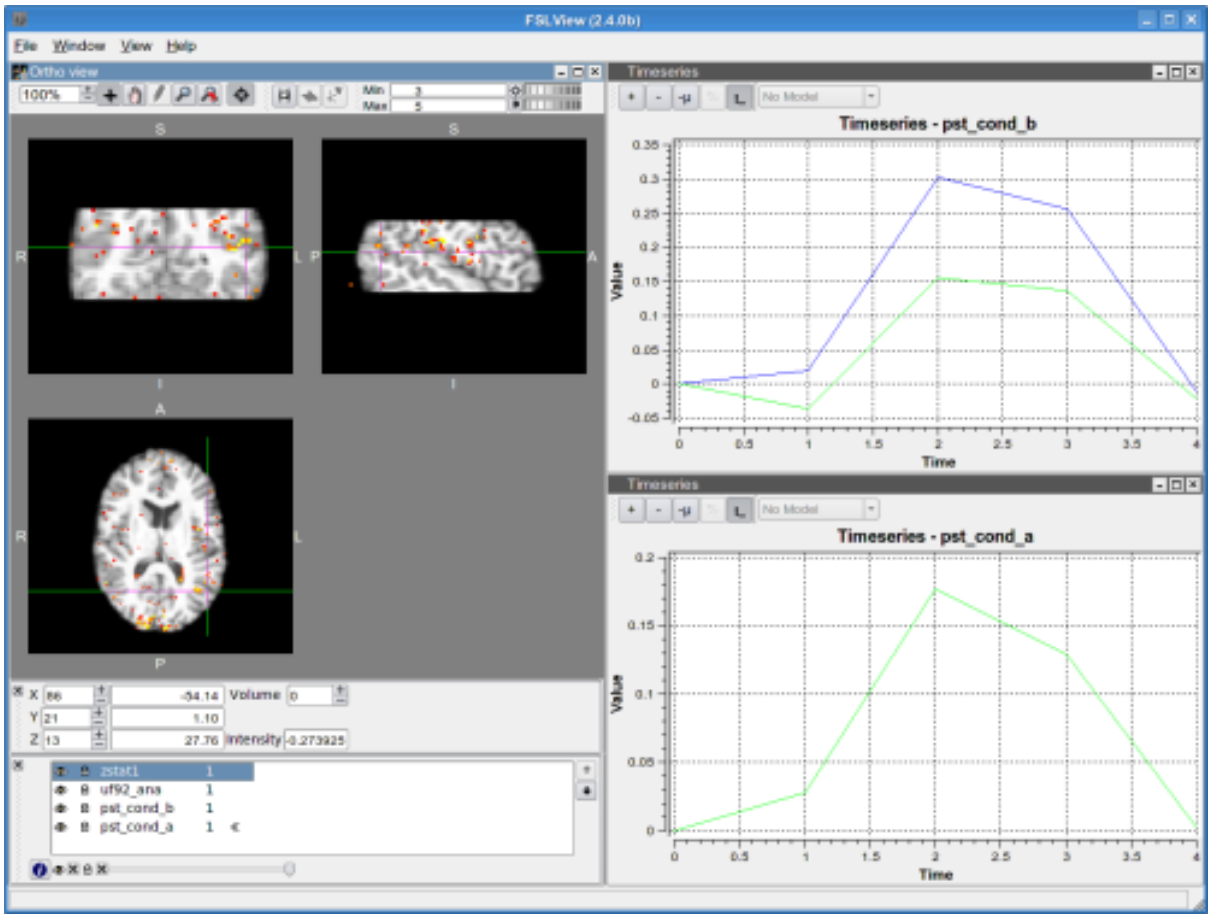

Figure 3.1: FSLView with `pynifti_pst` example.

This call produces a simple 4d NIfTI image that can be loaded into FSLView as any other timeseries. The following call can be used to display an FSL zmap from the above results path on top of some anatomy. Additionally the peristimulus timeseries of two conditions are loaded. The figure shows how it could look like. One of the nice features of FSLView is that its timeseries window can remember selected curves, which can be useful to compare signal timecourses from different voxels (blue and green line in the figure).

CHAPTER

FOUR

MODULE REFERENCE

This module provides Python bindings to the NIfTI data format.

The PyNIfTI module is a Python interface to the NIfTI I/O libraries. Using PyNIfTI, one can easily read and write NIfTI and ANALYZE images from within Python. The `NiftiImage` class provides pythonic access to the full header information and for a maximum of interoperability the image data is made available via NumPy arrays.

4.1 Module `format`

This modules provides a class representation of a NIfTI image header. The interface provides pythonic access to NIfTI properties using Python datatypes.

4.1.1 `NiftiFormat`

class `nifti.format.`**`NiftiFormat`**(*source*, *header=None*, *loadmeta=False*)

Bases: `object`

NIfTI header representation.

NIfTI header can be created by loading information from an existing NIfTI file or by creating a matching NIfTI header for a ndarray.

In addition, a number of methods to manipulate the header information are provided. However, this class is not able to write a NIfTI header back to disk. Please refer to the NIfTIImage class for this functionality.

Note: Handling of NIfTI header extensions is provided by the `NiftiExtensions` class (see its documentation for more information). Access to an instance of this class is available through the *NiftiFormat.extensions* attribute.

The constructor decides whether to load a nifti image header from file or create one from ndarray data, depending on the datatype of *source*.

Parameters

- **source** (*str | ndarray*) – If source is a string, it is assumed to be a filename and an attempt will be made to open the corresponding NIfTI file. Filenames might be provided as unicode strings. However, as the underlying library does not support unicode, they must be ascii-encodable, i.e. must not contain pure unicode characters. In case of an ndarray the array data will be used for the to be created nifti image and a matching nifti header is generated. If an object of a different type is supplied as 'source' a ValueError exception will be thrown.
- **header** (*dict*) – Additional header data might be supplied if image data is not loaded from a file. However, dimensionality and datatype are determined from the ndarray and not taken from a header dictionary.

asDict()

Returns the header data of the *NiftiImage* in a dictionary.

Return type
dict

Returns
The dictionary contains all NIfTI header information. Additionally, it might also contain a special 'meta' item that contains the meta data currently assigned to this instance.

Note: Modifications done to the returned dictionary do not cause any modifications in the NIfTI image itself. Please use `updateFromDict()` to apply changes to the image.

See Also:

`updateFromDict()`, `header`

description

extent

Returns the shape of the dataimage.

Return type
tuple

Returns
The order of dimensions is (x,y,z,t,u,v,w). If the image has less dimensions than 7 the return tuple will be shortened accordingly. Please note that the order of dimensions is different from the tuple returned by calling *NiftiImage.data.shape*!

See Also:

`getVolumeExtent()`, `getTimepoints()`, `extent`

filename

Returns the filename.

To distinguish ANALYZE from 2-file NIfTI images the image filename is returned for ANALYZE images while the header filename is returned for NIfTI files.

See Also:

`filename`

getExtent()
Returns the shape of the dataimage.

Return type
tuple

Returns
The order of dimensions is (x,y,z,t,u,v,w). If the image has less dimensions than 7 the return tuple will be shortened accordingly. Please note that the order of dimensions is different from the tuple returned by calling *NiftiImage.data.shape*!

See Also:

`getVolumeExtent()`, `getTimepoints()`, `extent`

getFilename()
Returns the filename.

To distinguish ANALYZE from 2-file NIfTI images the image filename is returned for ANALYZE images while the header filename is returned for NIfTI files.

See Also:

`filename`

getInverseQForm()
Returns the inverse qform matrix.

Note: The inverse qform matrix cannot be modified in-place. One needs to set a new qform matrix instead. The corresponding inverse matrix is then re-calculated automatically.

See Also:

`getQForm()`, `qform`, `qform_inv`,

getInverseSForm()
Returns the inverse sform matrix.

Note: The inverse sform matrix cannot be modified in-place. One needs to set a new sform matrix instead. The corresponding inverse matrix is then re-calculated automatically.

See Also:

`getSForm()`, `sform`, `sform_inv`,

getPixDims()
Returns the pixel dimensions on all 7 dimensions.

The function is similar to *getVoxDims()*, but instead of the 3d spatial dimensions of a voxel it returns the dimensions of an image pixel on all 7 dimensions supported by the NIfTI dataformat.

See Also:

getVoxDims(), setPixDims(), pixdim

getQForm()
Returns the qform matrix.

Note: The returned qform matrix is not bound to the object. Therefore it cannot be successfully modified in-place. Modifications to the qform matrix can only be done by setting a new qform matrix

---

See Also:

setQForm(), setQFormCode(), getQFormCode(), getQuaternion(), getQOffset(), setQuaternion(), setQOffset(), setQFac(), qform, qform_inv, qform_code, quatern, qoffset, qfac

getQFormCode(*as_string=False*)
Return the qform code.

By default NIfTI xform codes are returned, but if *as_string* is set to true a string representation ala 'talairach' is returned instead.

See Also:

getQFormCode(), qform_code

getQOffset()
Returns a 3-tuple containing (qx, qy, qz).

See Also:

setQOffset(), qform, qoffset

getQOrientation(*as_string=False*)
Returns to orientation of the i, j and k axis as stored in the qform matrix.

By default NIfTI orientation codes are returned, but if *as_string* is set to true a string representation ala 'Left-to-right' is returned instead.

Return type
list

Returns
orientations fo the x, y and z axis respectively.

See Also:

qform

getQuaternion()
Returns a 3-tuple containing (qb, qc, qd).

See Also:

setQuaternion(), qform, quatern

getRepetitionTime()
Returns the temporal distance between the volumes in a timeseries.

See Also:

setRepetitionTime(), rtime

getSForm()
Returns the sform matrix.

Note: The returned sform matrix is not bound to the object. Therefore it cannot be successfully modified in-place. Modifications to the sform matrix can only be done by setting a new sform matrix

---

See Also:

setSForm(), setSFormCode(), getSFormCode(), sform, sform_inv, sform_code

getSFormCode(*as_string=False*)
Return the sform code.

By default NIfTI xform codes are returned, but if *as_string* is set to true a string representation ala 'talairach' is returned instead.

See Also:

getSFormCode(), sform_code

getSOrientation(*as_string=False*)
Returns to orientation of the i, j and k axis as stored in the sform matrix.

By default NIfTI orientation codes are returned, but if *as_string* is set to true a string representation ala 'Left-to-right' is returned instead.

Return type
list

Returns
orientations fo the x, y and z axis respectively.

See Also:

sform

getTimeUnit(*as_string=False*)
Return unit of temporal (4th) axis.

By default NIfTI unit codes are returned, but if *as_string* is set to true a string representation ala 's' is returned instead.

See Also:

`setTimeUnit()`, `time_unit`

getTimepoints()
Returns the number of timepoints in the image.

In case of a 3d (or less dimension) image this method returns 1.

See Also:

`timepoints`

getVolumeExtent()
Returns the size/shape of the volume(s) in the image as a tuple.

Return type
tuple

Returns
Either a 3-tuple or 2-tuple or 1-tuple depending on the available dimensions in the image. The order of dimensions in the tuple is (x [, y [, z]]).

See Also:

`getExtent()`, `volextent`

getVoxDims()
Returns a 3-tuple a voxel dimensions/size in (x,y,z).

See Also:

`setVoxDims()`, `voxdim`

getXYZUnit(*as_string=False*)
Return 3D-space unit.

By default NIfTI unit codes are returned, but if *as_string* is set to true a string representation ala 'mm' is returned instead.

See Also:

`setXYZUnit()`, `xyz_unit`

header
Access to a dictionary version of the NIfTI header data.

Note: This property cannot be used like this:

```
nimg.header['something'] = 'new value'
```

Instead one has to get the header dictionary, modify and later reassign it:

```
h = nimg.header
h['something'] = 'new value'
nimg.header = h
```

See Also:

asDict(), updateFromDict()

intercept

max

min

nvox

pixdim

Returns the pixel dimensions on all 7 dimensions.

The function is similar to *getVoxDims()*, but instead of the 3d spatial dimensions of a voxel it returns the dimensions of an image pixel on all 7 dimensions supported by the NIfTI dataformat.

See Also:

getVoxDims(), setPixDims(), pixdim

qfac

qform

Returns the qform matrix.

Note: The returned qform matrix is not bound to the object. Therefore it cannot be successfully modified in-place. Modifications to the qform matrix can only be done by setting a new qform matrix

---

See Also:

setQForm(), setQFormCode(), getQFormCode(), getQuaternion(), getQOffset(), setQuaternion(), setQOffset(), setQFac(), qform, qform_inv, qform_code, quatern, qoffset, qfac

qform_code

Return the qform code.

By default NIfTI xform codes are returned, but if *as_string* is set to true a string representation ala 'talairach' is returned instead.

See Also:

getQFormCode(), qform_code

qform_inv

Returns the inverse qform matrix.

Note: The inverse qform matrix cannot be modified in-place. One needs to set a new qform matrix instead. The corresponding inverse matrix is then re-calculated automatically.

See Also:

`getQForm()`, `qform`, `qform_inv`,

qoffset
Returns a 3-tuple containing (qx, qy, qz).

See Also:

`setQOffset()`, `qform`, `qoffset`

quatern
Returns a 3-tuple containing (qb, qc, qd).

See Also:

`setQuaternion()`, `qform`, `quatern`

raw_nimg

rtime
Returns the temporal distance between the volumes in a timeseries.

See Also:

`setRepetitionTime()`, `rtime`

setDescription(*value*)
Set the description element in the NIfTI header.

Parameters
value (*str*) – Description – must not be longer than 79 characters.

See Also:

`description`

setIntercept(*value*)
Set the intercept attribute in the NIfTI header.

The intercept is only considered for scaling in case of a non-zero slope value.

See Also:

`slope`, `intercept`

setPixDims(*value*)
Set the pixel dimensions.

Parameters
value (*sequence*) – Up to 7 values (max. number of dimensions supported by the NIfTI format) are allowed in the sequence. The supplied

sequence can be shorter than seven elements. In this case only present values are assigned starting with the first dimension (spatial: x).

Note: Calling *setPixDims()* with a length-3 sequence equals calling *setVoxDims()*.

See Also:

`setVoxDims()`, `getPixDims()`, `pixdim`

setQFac(*value*, *code='scanner'*)

Set qfac (scaling factor of qform matrix).

The qform matrix and its inverse are re-computed automatically.

Besides reading it is also possible to set the qfac by assigning to the *qfac* property.

Parameters

- **value** (*float*) – Scaling factor.
- **code** (str | *NIFTI_XFORM_CODE* | int (0..4)) – The type of the coordinate system the corresponding qform matrix is describing. By default this coordinate system is assumed to be the scanner space. Please refer to the `setXFormCode()` method for a full list of possible codes and their meaning.

See Also:

`qform`, `qfac`

setQForm(*m*, *code='scanner'*)

Sets the qform matrix.

The supplied value has to be a 4x4 matrix. The matrix will be converted to float.

The inverse qform matrix and the quaternion representation will be automatically recalculated.

Parameters

- **m** (*4x4 ndarray*) – The qform matrix.
- **code** (str | *NIFTI_XFORM_CODE* | int (0..4)) – The type of the coordinate system the qform matrix is describing. By default this coordinate system is assumed to be the scanner space. Please refer to the `setXFormCode()` method for a full list of possible codes and their meaning.

See Also:

`getQForm()`, `setQFormCode()`, `getQFormCode()`, `getQuaternion()`, `getQOffset()`, `setQuaternion()`, `setQOffset()`, `setQFac()`, `qform`, `qform_inv`, `qform_code`, `quatern`, `qoffset`, `qfac`

setQFormCode(*code*)
Set the qform code.

Note: This is a convenience frontend for `setXFormCode()`. Please see its documentation for more information.

---

setQOffset(*value*, *code='scanner'*)
Set QOffset from 3-tuple (qx, qy, qz).

The qform matrix and its inverse are re-computed automatically.

Besides reading it is also possible to set the qoffset by assigning to the *qoffset* property.

Parameters

- **value** (*length-3 sequence*) – qx, qy and qz offsets.
- **code** (str | *NIFTI_XFORM_CODE* | int (0..4)) – The type of the coordinate system the corresponding qform matrix is describing. By default this coordinate system is assumed to be the scanner space. Please refer to the `setXFormCode()` method for a full list of possible codes and their meaning.

See Also:

`getQOffset()`, `qform`, `qoffset`

setQuaternion(*value*, *code='scanner'*)
Set Quaternion from 3-tuple (qb, qc, qd).

The qform matrix and it's inverse are re-computed automatically.

Parameters

- **value** (*length-3 sequence*) – qb, qc and qd quaternions.
- **code** (str | *NIFTI_XFORM_CODE* | int (0..4)) – The type of the coordinate system the corresponding qform matrix is describing. By default this coordinate system is assumed to be the scanner space. Please refer to the `setXFormCode()` method for a full list of possible codes and their meaning.

See Also:

`getQForm()`, `setQForm()`, `setQFormCode()`, `getQFormCode()`, `getQuaternion()`, `getQOffset()`, `setQOffset()`, `setQFac()`, `qform`, `qform_inv`, `qform_code`, `quatern`, `qoffset`, `qfac`

setRepetitionTime(*value*)
Set the repetition time of a NIfTI image (dt).

See Also:

getRepetitionTime(), rtime

setSForm(*m*, *code='mni152'*)
Sets the sform matrix.

The supplied value has to be a 4x4 matrix. The matrix elements will be converted to floats. By definition the last row of the sform matrix has to be (0,0,0,1). However, different values can be assigned, but will not be stored when the NIfTI image is saved to a file.

The inverse sform matrix will be automatically recalculated.

Parameters

- **m** (*4x4 ndarray*) – The sform matrix.
- **code** (str | *NIFTI_XFORM_CODE* | int (0..4)) – The type of the coordinate system the sform matrix is describing. By default this coordinate system is assumed to be the MNI152 space. Please refer to the setXFormCode() method for a full list of possible codes and their meaning.

See Also:

getSForm(), setSFormCode(), getSFormCode(), sform, sform_code

setSFormCode(*code*)
Set the sform code.

Note: This is a convenience frontend for setXFormCode(). Please see its documentation for more information.

---

setSlope(*value*)
Set the slope attribute in the NIfTI header.

Setting the slope to zero, will disable scaling.

See Also:

slope, intercept

setTimeUnit(*value*)
Set the unit of the temporal axis (4th).

Parameters
value (*int* | *str*) – The unit can either be given as a NIfTI unit code or as any of the plain text abbrevations returned by :meth:'~nifti.format.NiftiFormat.getTimeUnit'

See Also:

getTimeUnit(), time_unit

setVoxDims(*value*)
Set voxel dimensions/size.

The qform matrix and its inverse will be recalculated automatically.

Parameters
value (*3-tuple of floats*) – Have to be given in (x,y,z) order.

See Also:

`getVoxDims()`, `voxdim`

setXFormCode(*xform*, *code*)
Set the type of space described by the NIfTI transformations.

The NIfTI format defines five coordinate system types which are used to describe the target space of a transformation (qform or sform). Please note, that the last four transformation types are only available in the NIfTI format and not when saving into ANALYZE.

'unkown', *NIFTI_XFORM_UNKNOWN*, 0:
Transformation is arbitrary. This is the ANALYZE compatibility mode. In this case no *sform* matrix will be written, even when stored in NIfTI and not in ANALYZE format. Additionally, only the pixdim parts of the *qform* matrix will be saved (upper-left 3x3).

'scanner', *NIFTI_XFORM_SCANNER_ANAT*, 1:
Scanner-based anatomical coordinates.

'aligned', *NIFTI_XFORM_ALIGNED_ANAT*, 2:
Coordinates are aligned to another file's coordinate system.

'talairach', *NIFTI_XFORM_TALAIRACH*, 3:
Coordinate system is shifted to have its origin (0,0,0) at the anterior commissure, as in the Talairach-Tournoux Atlas.

'mni152', *NIFTI_XFORM_MNI_152*, 4:
Coordinates are in MNI152 space.

Parameters

- **xform** (*'qform'* | *'q'* | *'sform'* | *'s'*) – Which of the two NIfTI transformations to set.
- **code** (str | *NIFTI_XFORM_CODE* | int (0..4)) – The Transformation code can be specified either by a string, the *NIFTI_XFORM_CODE* defined in the nifti1.h header file (accessible via the *nifti.clib* module, or the corresponding integer value.

See Also:

`setQFormCode()`, `getQFormCode()`, `setSFormCode()`, `getSFormCode()`, `qform_code`, `sform_code`

setXYZUnit(*value*)
Set the unit of the spatial axes.

Parameters
value (*int* | *str*) – The unit can either be given as a NIfTI unit code or as any of the plain text abbrevations returned by :meth:'~nifti.format.NiftiFormat.getXYZUnit'

See Also:

getXYZUnit(), xyz_unit

sform
Returns the sform matrix.

Note: The returned sform matrix is not bound to the object. Therefore it cannot be successfully modified in-place. Modifications to the sform matrix can only be done by setting a new sform matrix

---

See Also:

setSForm(), setSFormCode(), getSFormCode(), sform, sform_inv, sform_code

sform_code
Return the sform code.

By default NIfTI xform codes are returned, but if *as_string* is set to true a string representation ala 'talairach' is returned instead.

See Also:

getSFormCode(), sform_code

sform_inv
Returns the inverse sform matrix.

Note: The inverse sform matrix cannot be modified in-place. One needs to set a new sform matrix instead. The corresponding inverse matrix is then re-calculated automatically.

See Also:

getSForm(), sform, sform_inv,

slope

time_unit
Return unit of temporal (4th) axis.

By default NIfTI unit codes are returned, but if *as_string* is set to true a string representation ala 's' is returned instead.

See Also:

setTimeUnit(), time_unit

timepoints
Returns the number of timepoints in the image.

In case of a 3d (or less dimension) image this method returns 1.

See Also:

timepoints

updateFromDict (*hdrdict*)
Update NIfTI header information.

Updated header data is read from the supplied dictionary. One cannot modify dimensionality and datatype of the image data. If such information is present in the header dictionary it is removed before the update. If resizing or datatype casting are required one has to convert the image data into a separate array and perform resize and data manipulations on this array. When finished, the array can be converted into a nifti file by calling the NiftiImage constructor with the modified array as 'source' and the nifti header of the original NiftiImage object as 'header'.

Note: If the provided dictionary contains a 'meta' item its content is used to overwrite any potentially existing meta data. dictionary.

The same behavior will be used for 'extensions'. If extensions are defined in the provided dictionary all currently existing extensions will be overwritten.

See Also:

asDict(), header

updateQFormFromQuaternion()
Only here for backward compatibility.

volextent
Returns the size/shape of the volume(s) in the image as a tuple.

Return type
tuple

Returns
Either a 3-tuple or 2-tuple or 1-tuple depending on the available dimensions in the image. The order of dimensions in the tuple is (x [, y [, z]]).

See Also:

getExtent(), volextent

voxdim
Returns a 3-tuple a voxel dimensions/size in (x,y,z).

See Also:

setVoxDims(), voxdim

vx2q(*coord*)
Transform a voxel's index into coordinates (qform-defined).

Parameters
coord (*3-tuple*) – A voxel's index in the volume fiven as three positive integers (i, j, k).

Return type
vector

See Also:

`setQForm()`, `getQForm()` `qform`

vx2s(*coord*)
Transform a voxel's index into coordinates (sform-defined).

Parameters
coord (*3-tuple*) – A voxel's index in the volume fiven as three positive integers (i, j, k).

Return type
vector

See Also:

`setSForm()`, `getSForm()` `sform`

xyz_unit
Return 3D-space unit.

By default NIfTI unit codes are returned, but if *as_string* is set to true a string representation ala 'mm' is returned instead.

See Also:

`setXYZUnit()`, `xyz_unit`

4.2 Module `image`

This module provides two classes for accessing NIfTI files.

- `NiftiImage` (traditional load-as-much-as-you-can approach)
- `MemMappedNiftiImage` (memory-mapped access to uncompressed NIfTI files)

4.2.1 `NiftiImage`

class `nifti.image.`**`NiftiImage`**(*source*, *header=None*, *load=False*, ***kwargs*)
Bases: `nifti.format.NiftiFormat`

Wrapper class for convenient access to NIfTI images.

An image can either be loaded from a file or created from a NumPy ndarray. Either way is automatically determined by the type of the 'source' argument. If *source* is a string, it is assumed to be a filename an ndarray is treated as such.

All NIfTI header information is conveniently exposed via Python data types. This functionality is provided by the `NiftiFormat` base class. Please refer to its documentation for the full list of its methods and properties.

See Also:

`NiftiFormat`, `MemMappedNiftiImage`

Parameters

- **source** (*str* | *ndarray*) – If source is a string, it is assumed to be a filename and an attempt will be made to open the corresponding NIfTI file. In case of an ndarray the array data will be used for the to be created nifti image and a matching nifti header is generated. If an object of a different type is supplied as 'source' a ValueError exception will be thrown.
- **header** (*dict*) – Additional header data might be supplied. However, dimensionality and datatype are determined from the ndarray and not taken from a header dictionary.
- **load** (*Boolean*) – If set to True the image data will be loaded into memory. This is only useful if loading a NIfTI image from file. This flag is almost useless, as the data will be loaded automatically whenever it is accessed.
- ****kwargs** – Additional stuff is passed to `NiftiFormat`.

asarray(*copy=True*)

Convert the image data into a ndarray.

Parameters

copy (*Boolean*) – If set to False the array only wraps the image data, while True will return a copy of the data array.

bbox

Get the bounding box an image.

The bounding box is the smallest box covering all non-zero elements.

Return type

tuple(2-tuples) | None

Returns

It returns as many (min, max) tuples as there are image dimensions. The order of dimensions is identical to that in the data array. *None* is returned of the images does not contain non-zero elements.

Examples:

```
>>> from nifti import NiftiImage
>>> nim = NiftiImage(N.zeros((12, 24, 32)))
>>> nim.bbox is None
True

>>> nim.data[3,10,13] = 1
>>> nim.data[6,20,26] = 1
>>> nim.bbox
((3, 6), (10, 20), (13, 26))

>>> nim.crop()
>>> nim.data.shape
(4, 11, 14)
>>> nim.bbox
((0, 3), (0, 10), (0, 13))
```

See Also:

nifti.image.NiftiImage.bbox, nifti.imgfx.crop()

copy()
Return a copy of the image.

crop(*nim*, *bbox=None*)
Crop an image.

Parameters
bbox (*list(2-tuples)* | *None*) – Each tuple has the (min,max) values for a particular image dimension. If *None*, the images actual bounding box is used for cropping.

See Also:

nifti.image.NiftiImage.bbox, nifti.imgfx.getBoundingBox()

data
Return the NIfTI image data wrapped into a NumPy array.

See Also:

data

filename
Please see nifti.format.NiftiFormat.getFilename() for the documentation.

getDataArray()
Return the NIfTI image data wrapped into a NumPy array.

See Also:

data

getFilename()
Please see nifti.format.NiftiFormat.getFilename() for the documentation.

getScaledData()
Returns a scaled copy of the data array.

Scaling is done by multiplying with the slope and adding the intercept that is stored in the NIfTI header. In compliance with the NIfTI standard scaling is only performed in case of a non-zero slope value. The original data array is returned otherwise.

Return type
ndarray

load()
Load the image data into memory, if it is not already accessible.

It is save to call this method several times successively.

save(*filename=None*, *filetype='NIFTI'*, *update_minmax=True*)
Save the image to a file.

If the image was created using array data (i.e., not loaded from a file) a filename has to be specified.

If not yet done already, the image data will be loaded into memory before saving the file.

Parameters

- **filename** (*str | None*) – The name of the target file (typically including its extension). Filenames might be provided as unicode strings. However, as the underlying library does not support unicode, they must be ascii-encodable, i.e. must not contain pure unicode characters. Usually setting the filename also determines the filetype (NIfTI/ANALYZE). Please see `setFilename()` for some more details. If None, an image loaded from a file will cause the original image to be overwritten.
- **filetype** (*str*) – Provide intented filetype. Please see the documentation of the *setFilename()* method for some more details.
- **update_minmax** (*bool*) – Whether the image header's min and max values should be updated according to the current image data.

Warning: There will be no exception if writing fails for any reason, as the underlying function nifti_write_hdr_img() from libniftiio does not provide any feedback. Suggestions for improvements are appreciated.

setDataArray(*data*)

setFilename(*filename*, *filetype='NIFTI'*)
Set the filename for the NIfTI image.

Setting the filename also determines the filetype. If the filename ends with '.nii' the type will be set to NIfTI single file. A '.hdr' extension can be used for NIfTI file

pairs. If the desired filetype is ANALYZE the extension should be '.img'. However, one can use the '.hdr' extension and force the filetype to ANALYZE by setting the filetype argument to ANALYZE. Setting filetype if the filename extension is '.nii' has no effect, the file will always be in NIFTI format.

If the filename carries an additional '.gz' the resulting file(s) will be compressed.

Uncompressed NIfTI single files are the default filetype that will be used if the filename has no valid extension. The '.nii' extension is appended automatically. The 'filetype' argument can be used to force a certain filetype when no extension can be used to determine it. 'filetype' can be one of the nifticlibs filtetypes or any of 'NIFTI', 'NIFTI_GZ', 'NIFTI_PAIR', 'NIFTI_PAIR_GZ', 'ANALYZE', 'ANALYZE_GZ'.

Setting the filename will cause the image data to be loaded into memory if not yet done already. This has to be done, because without the filename of the original image file there would be no access to the image data anymore. As a side-effect a simple operation like setting a filename may take a significant amount of time (e.g. for a large 4d dataset).

By passing an empty string or none as filename one can reset the filename and detach the NiftiImage object from any file on disk.

Examples:

Filename	Output of save()
exmpl.nii	exmpl.nii (NIfTI)
exmpl.hdr	exmpl.hdr, exmpl.img (NIfTI)
exmpl.img	exmpl.hdr, exmpl.img (ANALYZE)
exmpl	exmpl.nii (NIfTI)
exmpl.hdr.gz	exmpl.hdr.gz, exmpl.img.gz (NIfTI)
exmpl.gz	exmpl.gz.nii (uncompressed NIfTI)

Setting the filename is also possible by assigning to the 'filename' property.

See Also:

`getFilename()`, `filename`

unload()
Unload image data and free allocated memory.

This methods does nothing in case of memory mapped files.

updateCalMinMax()
Update the image data maximum and minimum value in the nifti header.

updateHeader(*hdrdict*)
Deprecated method only here for backward compatibility.

Please refer to NiftiFormat.updateFromDict()

4.2.2 MemMappedNiftiImage

class nifti.image.**MemMappedNiftiImage**(*source*)
Bases: nifti.image.NiftiImage

Memory mapped access to uncompressed NIfTI files.

This access mode might be the prefered one whenever only a small part of the image data has to be accessed or the memory is not sufficient to load the whole dataset.

Please note, that memory-mapping is not required when exclusively header information shall be accessed. The NiftiFormat class and by default also the NiftiImage class will not load any image data into memory.

Note: The class is mostly useful for read-only access to the NIfTI image data. It currently neither supports saving changed header fields nor storing meta data.

Create a NiftiImage object.

This method decides whether to load a nifti image from file or create one from ndarray data, depending on the datatype of *source*.

Parameters
source (*str* | *ndarray*) – If source is a string, it is assumed to be a filename and an attempt will be made to open the corresponding NIfTI file. In case of an ndarray the array data will be used for the to be created nifti image and a matching nifti header is generated. If an object of a different type is supplied as 'source' a ValueError exception will be thrown.

data
Please see nifti.format.NiftiImage.getDataArray() for the documentation.

filename
Please see nifti.format.NiftiFormat.getFilename() for the documentation.

getDataArray()
Please see nifti.format.NiftiImage.getDataArray() for the documentation.

getFilename()
Please see nifti.format.NiftiFormat.getFilename() for the documentation.

load()
Does nothing for memory mapped images.

save()
Save the image.

This methods does nothing except for syncing the file on the disk.

Please note that the NIfTI header might not be completely up-to-date. For example, the min and max values might be outdated, but this class does not automatically update them, because it would require to load and search through the whole array.

setFilename (*filename*, *filetype='NIFTI'*)
Does not work for memory mapped images and therefore raises an exception.

unload()
Does nothing for memory mapped images.

4.3 Module extensions

This module provides a container-like interface to NIfTI1 header extensions.

4.3.1 NiftiExtensions

class nifti.extensions.**NiftiExtensions**(*raw_nimg*, *source=None*)
Bases: object

NIfTI1 header extension handler.

This class wraps around a NIfTI1 struct and provides container-like access to NIfTI1 header extensions. It is basically a hibrid between a list and a dictionary. The reason for this is that the NIfTI header allows for a *list* of extensions, but additionally each extension is associated with some type (*ecode* or extension code). This is some form of *mapping*, however, the ecodes are not necessarily unique (e.g. multiple comments).

The current list of known extensions is documented here:

http://nifti.nimh.nih.gov/nifti-1/documentation/faq#Q21

The usage is best explained by a few examples. All examples assume a NIfTI image to be loaded:

```
>>> from nifti import NiftiImage
>>> nim = NiftiImage('example4d.nii.gz')
```

Access to the extensions is provided through the *extensions* attribute of the NiftiImage class:

```
>>> type(nim.extensions)
<class 'nifti.extensions.NiftiExtensions'>
```

How many extensions are avialable?

```
>>> len(nim.extensions)
2
```

How many comments? Any AFNI extension?

```
>>> nim.extensions.count('comment')
2
```

Show me all *ecodes* of all extensions:

```
>>> nim.extensions.ecodes
[6, 6]
```

Add an *AFNI* extension:

```
>>> nim.extensions += ('afni', '<xml>Some voodoo</xml>')
>>> nim.extensions.ecodes
[6, 6, 4]
```

Delete superfluous comment extension:

```
>>> del nim.extensions[1]
```

Access the last extension, which should be the *AFNI* one:

```
>>> nim.extensions[-1]
'<xml>Some voodoo</xml>'
```

Wipe them all:

```
>>> nim.extensions.clear()
>>> len(nim.extensions)
0
```

Parameters

- **raw_nimg** (*nifti_image struct*) – This is the raw NIfTI image struct pointer. It is typically provided by `nifti.format.NiftiFormat.raw_nimg`.
- **source** (*list(2-tuple)*) – This is an optional list for extension tuples (ecode, edata). Each element of this list will be appended as a new extension.

append(*extension*)

Append a new extension.

Parameters

extension (*2-tuple*) – An extension is given by a *(ecode, edata)* tuple, where *ecode* can be either literal or numerical and *edata* is any kind of data.

Note:

Currently, *edata* can only be stuff whos len(edata) matches its size in bytes, e.g. str.

clear()
Remove all extensions.

count(*code*)
Returns the number of extensions matching a given *ecode*.

Parameters
code (*int* | *str*) – The ecode can be specified either literal or as numerical value.

ecodes
Returns a list of ecodes for all extensions.

iteritems()
A generator method that returns a 2-tuple (ecode, edata) on each iteration. It can be used in the same fashion as *dict.iteritems()*.

4.4 Useful Functions

4.4.1 Utilities from `nifti.utils`

Utility functions for PyNifti

nifti.utils.**applyFxToVolumes**(*ts*, *vols*, *fx*, ***kwargs*)
Apply a function on selected volumes of a timeseries.

'ts' is a 4d timeseries. It can be a NiftiImage or a ndarray. In case of a ndarray one has to make sure that the time is on the first axis. 'ts' can actually be of any dimensionality, but datasets aka volumes are assumed to be along the first axis.

'vols' is either a sequence of sequences or a 2d array indicating which volumes fx should be applied to. Each row defines a set of volumes.

'fx' is a callable function to get an array of the selected volumes as argument. Additonal arguments may be specified as keyword arguments and are passed to 'fx'.

The output will be a 4d array with one computed volume per row in the 'vols' array.

nifti.utils.**getPeristimulusTimeseries**(*ts*, *onsetvols*, *nvols=10*, *fx=<function mean at 0x89bc80c>*)
Returns 4d array with peristimulus timeseries.

Parameters

- **ts** – source 4d timeseries
- **onsetvols** – sequence of onsetvolumes to be averaged over
- **nvols** – length of the peristimulus timeseries in volumes (starting from onsetvol)

- **fx** – function to be applied to the list of corresponding volumes. Typically this will be mean(), so it is default, but it could also be var() or something different. The supplied function is to be able to handle an 'axis=0' argument similiar to NumPy's mean(), var(), ...

nifti.utils.**splitFilename**(*filename*)

Split a NIfTI filename into basename and extension.

Parameters
filename (*str*) – Filename to be split.

Return type
tuple

Returns
The function returns a tuple of basename and extension. If no valid NIfTI filename extension is found, the whole string is returned as basename and the extension string will be empty.

nifti.utils.**time2vol**(*t*, *tr*, *lag=0.0*, *decimals=0*)

Translates a time 't' into a volume number. By default function returns the volume number that is closest in time. Volumes are assumed to be recorded exactly (and completely) after tr/2, e.g. if 'tr' is 2 secs the first volume is recorded at exactly one second.

't' might be a single value, a sequence or an array.

The repetition 'tr' might be specified directly, but can also be a NiftiImage object. In the latter case the value of 'tr' is determined from the 'rtime' property of the NiftiImage object.

't' and 'tr' can be given in an arbitrary unit (but both have to be in the same unit).

The 'lag' argument can be used to shift the times by constant offset.

Please note that numpy.round() is used to round to interger value (rounds to even numbers). The 'decimals' argument will be passed to numpy.round().

4.4.2 Image functions in `nifti.imgfx`

Functions operating on images

nifti.imgfx.**getBoundingBox**(*nim*)

Get the bounding box an image.

The bounding box is the smallest box covering all non-zero elements.

Return type
tuple(2-tuples) | None

Returns
It returns as many (min, max) tuples as there are image dimensions. The order of dimensions is identical to that in the data array. *None* is returned of the images does not contain non-zero elements.

Examples:

```
>>> from nifti import NiftiImage
>>> nim = NiftiImage(N.zeros((12, 24, 32)))
>>> nim.bbox is None
True

>>> nim.data[3,10,13] = 1
>>> nim.data[6,20,26] = 1
>>> nim.bbox
((3, 6), (10, 20), (13, 26))

>>> nim.crop()
>>> nim.data.shape
(4, 11, 14)
>>> nim.bbox
((0, 3), (0, 10), (0, 13))
```

See Also:

nifti.image.NiftiImage.bbox, nifti.imgfx.crop()

nifti.imgfx.**crop**(*nim*, *bbox=None*)

Crop an image.

Parameters

bbox (*list(2-tuples) | None*) – Each tuple has the (min,max) values for a particular image dimension. If *None*, the images actual bounding box is used for cropping.

See Also:

nifti.image.NiftiImage.bbox, nifti.imgfx.getBoundingBox()

CHAPTER

FIVE

PYNIFTI DEVELOPMENT CHANGELOG

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>`

The full VCS changelog is available here:

http://git.debian.org/?p=pkg-exppsy/pynifti.git;a=shortlog;h=upstream

Unreleased changes

Changes described here are not yet released, but available from VCS repository.

- Added support for COMPLEX64 datatype.
- Bugfix: Ensure consistent mapping between NIfTI and NumPy datatypes.

5.1 Releases

- 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.

PYTHON MODULE INDEX

n

INDEX

T

U

V

X