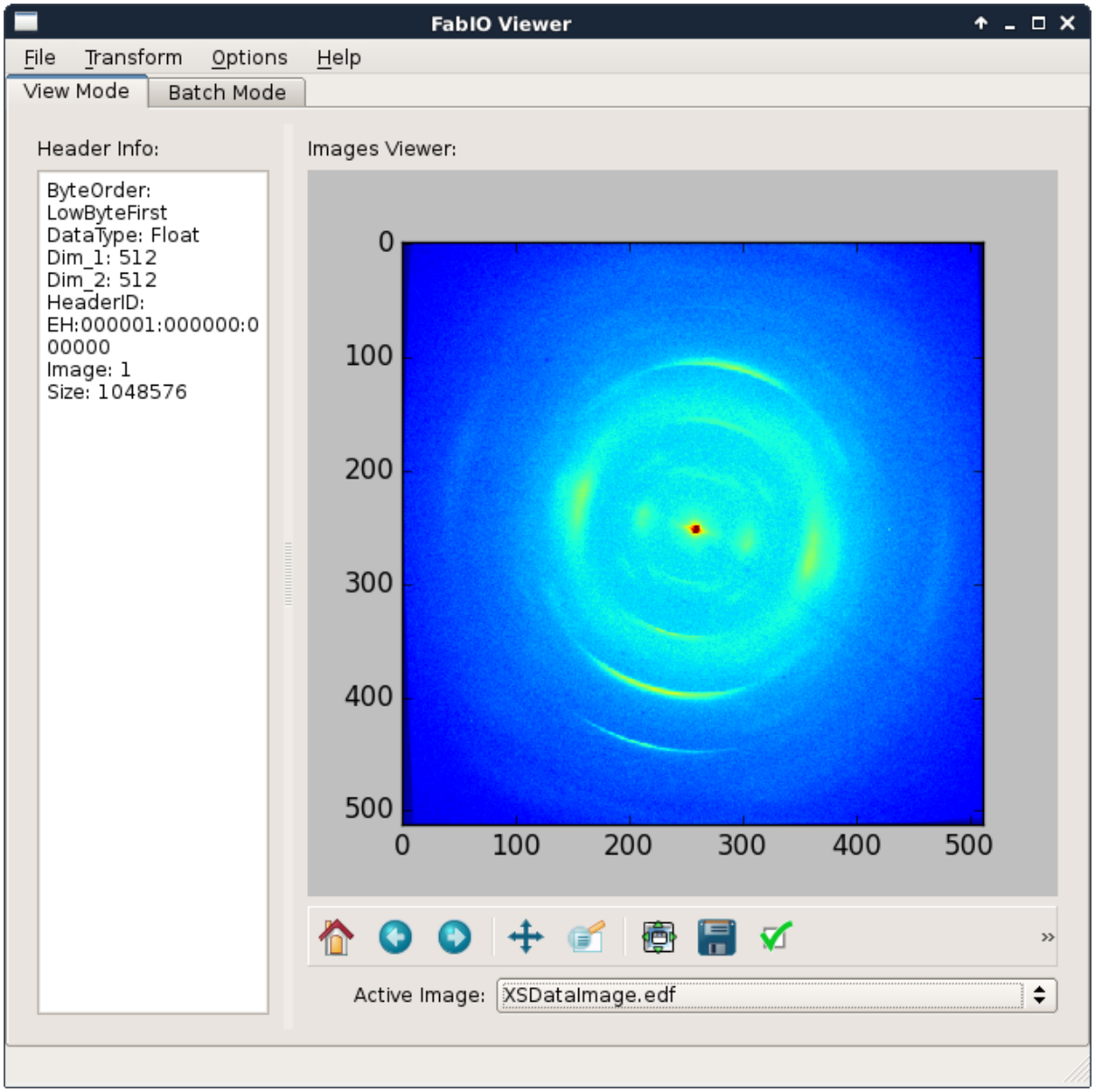


FabIO Documentation

Release 0.4.0

H. Sorensen, E. Knudsen, J. Wright, G. Goret, B. Pauw and J. Kieffer

July 15, 2016

Contents:

CHAPTER

ONE

GETTING STARTED

FabIO is a Python module for reading and handling data from two-dimensional X-ray detectors.

FabIO is a Python module written for easy and transparent reading of raw two-dimensional data from various X-ray detectors. The module provides a function for reading any image and returning a FabioImage object which contains both metadata (header information) and the raw data. All FabioImage object offer additional methods to extract information about the image and to open other detector images from the same data series.

1.1 Introduction

One obstacle when writing software to analyse data collected from a two-dimensional detector is to read the raw data into the program, not least because the data can be stored in many different formats depending on the instrument used. To overcome this problem we decided to develop a general module, FabIO (FABle I/O), to handle reading and writing of two-dimensional data. The code-base was initiated by merging parts of our fabian imageviewer and ImageD11 peak-search programs and has been developed since 2007 as part of the TotalCryst program suite for analysis of 3DXRD microscopy data. During integration into a range of scientific programs like the FABLE graphical interface, EDNA and the fast azimuthal integration library, pyFAI; FabIO has gained several features like handling multi-frame image formats as well as writing many of the file formats.

1.2 FabIO Python module

Python is a scripting language that is very popular among scientists and which also allows well structured applications and libraries to be developed.

1.2.1 Philosophy

The intention behind this development was to create a Python module which would enable easy reading of 2D data images, from any detector without having to worry about the file format. Therefore FabIO just needs a file name to open a file and it determines the file format automatically and deals with gzip and bzip2 compression transparently. Opening a file returns an object which stores the image in memory as a 2D NumPy array and the metadata, called header, in a Python dictionary. Beside the data and header attributes, some methods are provided for reading the previous or next image in a series of images as well as jumping to a specific file number. For the user, these auxiliary methods are intended to be independent of the image format (as far as is reasonably possible).

FabIO is written in an object-oriented style (with classes) but aims at being used in a scripting environment: special care has been taken to ensure the library remains easy to use. Therefore no knowledge of object-oriented programming is required to get full benefits of the library. As the development is done in a collaborative and decentralized way; a comprehensive test suite has been added to reduce the number of regressions when new features are added or old problems are repaired. The software is very modular and allows new classes to be added for handling other data formats easily. FabIO and its source-code are freely available to everyone on-line, licensed under the GNU General Public License version 3 (GPLv3). FabIO is also available directly from popular Linux distributions like Debian and Ubuntu.

1.2.2 Implementation

The main language used in the development of FabIO is Python; however, some image formats are compressed and require compression algorithms for reading and writing data. When such algorithms could not be implemented efficiently using Python or NumPy native modules were developed, in i.e. standard C code callable from Python (sometimes generated using Cython). This code has to be compiled for each computer architecture and offers excellent performance. FabIO is only dependent on the NumPy module and has extra features if two other optional Python modules are available. For reading XML files (that are used in EDNA) the Lxml module is required and the Python Image Library, PIL is needed for producing a PIL image for displaying the image in graphical user interfaces and several image-processing operations that are not re-implemented in FabIO. A variety of useful image processing is also available in the scipy.ndimage module and in scikits-image.

Images can also be displayed in a convenient interactive manner using matplotlib and an IPython shell , which is mainly used for developing data analysis algorithms. Reading and writing procedure of the various TIFF formats is based on the TiffIO code from PyMCA.

In the Python shell, the *fabio* module must be imported prior to reading an image in one of the supported file formats (see Table *Supported formats*, hereafter). The *fabio.open* function creates an instance of the Python class *fabioimage*, from the name of a file. This instance, named *img* hereafter, stores the image data in *img.data* as a 2D NumPy array. Often the image file contains more information than just the intensities of the pixels, e.g. information about how the image is stored and the instrument parameters at the time of the image acquisition, these metadata are usually stored in the file header. Header information, are available in *img.header* as a Python dictionary where keys are strings and values are usually strings or numeric values.

Information in the header about the binary part of the image (compression, endianness, shape) are interpreted however, other metadata are exposed as they are recorded in the file. FabIO allows the user to modify and, where possible, to save this information (the table *Supported formats* summarizes writable formats). Automatic translation between file-formats, even if desirable, is sometimes impossible because not all format have the capability to be extended with additional metadata. Nevertheless FabIO is capable of converting one image data-format into another by taking care of the numerical specifics: for example float arrays are converted to integer arrays if the output format only accepts integers.

1.2.3 FabIO methods

One strength of the implementation in an object oriented language is the possibility to combine functions (or methods) together with data appropriate for specific formats. In addition to the header information and image data, every *fabioimage* instance (returned by *fabio.open*) has methods inherited from *fabioimage* which provide information about the image minimum, maximum and mean values. In addition there are methods which return the file number, name etc. Some of the most important methods are specific for certain formats because the methods are related to how frames in a sequence are handled; these methods are *img.next()*, *img.previous()*, and *img.getframe(n)*. The behaviour of such methods varies depending on the image format: for single-frame format (like mar345), *img.next()* will return the image in next file; for multi-frame format (like GE), *img.next()* will return the next frame within the same file. For formats which are possibly multi-framed like EDF, the behaviour depends on the actual number of frames per file (accessible via the *img.nframes* attribute).

1.3 Usage

1.3.1 Examples

In this section we have collected some basic examples of how FabIO can be employed.

Opening an image:

```
import fabio
im100 = fabio.open('Quartz_0100.tif') # Open image file
print(im0.data[1024,1024])            # Check a pixel value
im101 = im100.next()                  # Open next image
im270 = im1.getframe(270)             # Jump to file number 270: Quartz_0270.tif
```

Normalising the intensity to a value in the header:

```
img = fabio.open('exampleimage0001.edf')
print(img.header)
{'ByteOrder': 'LowByteFirst',
 'DATE (scan begin)': 'Mon Jun 28 21:22:16 2010',
 'ESRFCurrent': '198.099',
...
}
# Normalise to beam current and save data
srcur = float(img.header['ESRFCurrent'])
img.data *= 200.0/srcur
img.write('normed_0001.edf')
```

Interactive viewing with matplotlib:

```
from matplotlib import pyplot        # Load matplotlib
pyplot.imshow(img.data)              # Display as an image
pyplot.show()                        # Show GUI window
```

1.4 Future and perspectives

The Hierarchical Data Format version 5 (*hdf5*) is a data format which is increasingly popular for storage of X-ray and neutron data. To name a few facilities the synchrotron Soleil and the neutron sources ISIS, SNS and SINQ already use HDF extensively through the NeXus format. For now, mainly processed or curated data are stored in this format but new detectors (Eiger from Dectris) are natively saving data in HDF5. FabIO will rely on H5Py, which already provides a good HDF5 binding for Python, as an external dependency, to be able to read and write such HDF5 files. This starts to be available in version 0.4.0.

1.5 Conclusion

FabIO gives an easy way to read and write 2D images when using the Python computer language. It was originally developed for X-ray diffraction data but now gives an easy way for scientists to access and manipulate their data from a wide range of 2D X-ray detectors. We welcome contributions to further improve the code and hope to add more file formats in the future.

1.5.1 Acknoledgements

We acknowledge Andy Götz and Kenneth Evans for extensive testing when including the FabIO reader in the Fable image viewer (Götz et al., 2007).We also thank V. Armando Solé for assistance with his TiffIO reader and Carsten Gundlach for deployment of FabIO at the beamlines i711 and i811, MAX IV, and providing bug reports. We finally acknowledge our colleagues who have reported bugs and helped to improve FabIO. Financial support was granted by the EU 6th Framework NEST/ADVENTURE project TotalCryst (Poulsen et al., 2006).

1.5.2 Citation

Knudsen, E. B., Sørensen, H. O., Wright, J. P., Goret, G. & Kieffer, J. (2013). J. Appl. Cryst. 46, 537-539.

http://dx.doi.org/10.1107/S0021889813000150

1.5.3 List of file formats that FabIO can read and write

In alphabetical order. The listed filename extensions are typical examples. FabIO tries to deduce the actual format from the file itself and only uses extensions as a fallback if that fails.

1.5.4 Adding new file formats

We hope it will be relatively easy to add new file formats to FabIO in the future. Please refere at the *fabio/templateimage.py* file in the source which describes how to add a new format.

CHAPTER

TWO

INSTALLATION

FabIO can, as any Python module, be installed from its sources, available on the Python cheese shop but we advice to use binary wheels packages provided for the most common platforms: Windows, MacOSX. For Debian Linux and its derivatives (Ubuntu, Mint, ...), FabIO is part of the distributions and itss package is named *python-fabio* and can be installed via:

```
sudo apt-get install python-fabio
```

If you are using MS Windows or MacOSX; binary version (as wheel packages) are PIP-installable. PIP is the Python Installer Program, similar to `apt-get` for Python. It runs under any architecture. Since Python 2.7.10 and 3.4, PIP is installed together with Python itself. If your Python is elder, PIP can be simply downloaded and executed using your standard Python:

2.1 Installation under windows

Install Python from the official web page. I would recommend Python 2.7 in 64 bits version if your operating system allows it. Python3 (>=3.4) is OK.

If you are looking for an integrated scientific Python distribution on Windows, WinPython is a good one, the Python2.7, 64 bit

version is advised.

It comes with pip pre-installed and configured.

2.1.1 Installation using PIP:

If you use a Python2 elder than 2.7.10 or a Python3 <3.4, you will need Download PIP. Then install the wheel package manager and all dependencies for :

```
python get-pip.py
pip install setuptools
pip install wheel
pip install fabio
```

Note: for now, PyQt4 is not yet pip-installable. you will need to get it from riverbankcomputing: http://www.riverbankcomputing.co.uk/software/pyqt/download

2.1.2 Manual installation under windows

You will find all the scientific Python stack packaged for Windows on Christoph Gohlke' page (including FabIO):

Pay attention to the Python version (both number and architecture). **DO NOT MIX 32 and 64 bits version**. To determine the version and architecture width of the Python interpreter:

2.1.3 Installation from sources

- Retrieve the sources from github:
  - The master development branch
  - The latest release
- unzip the file in a directory
- open a console (cmd.exe) in this directory.
- install the required dependencies using PIP:

```
pip install -r ci/requirements_appveyor.txt --trusted-host www.silx.org
```

Get the compiler and install it

The version of the compiler and the version of the Microsoft SDK have to match the Python version you are using. Here are a couple of examples:

- Python2.7: Microsoft compiler 2008
- Python 3.5: Microsoft Visual studio community edition 2015

Compile the sources

```
python setup.py build
python setup.py test
pip install .
```

2.1.4 Testing version of FabIO

Continuous integration runs the complete test suite on multiple operating systems and python version. Under Windows, this is done using the AppVeyor cloud service Select the environment which matches your setup like **Environment: PYTHON=C:Python34-x64, PYTHON_VERSION=3.4.3, PYTHON_ARCH=64** and go to **artifacts** where wheels and MSI-installers are available.

2.2 Installation on MacOSX

Python 2.7, 64 bits and numpy are natively available on MacOSX.

2.2.1 Install via PIP

Since MacOSX 10.11 (El-Captain), PIP is available as part of the standard python installation. For elder MacOSX, download PIP and run. Then install FabIO directly:

Note: for now, PyQt4, used by the *fabio-viewer* is not yet pip-installable. You will need to get it from riverbankcomputing.

2.2.2 Compile from sources

Get the compiler

Apple provides for free Xcode which contains the compiler needed to build binary extensions. Xcode can be installed from the App-store.

Compile the sources

Once done, follow the classical procedure (similar to Windows or Linux):

- Retrieve the sources from github:
  - The master development branch
  - The latest release
- unzip the file in a directory
- open a terminal in the unzipped archive directory
- run:

```
sudo pip install -r ci/requirements_travis.txt --trusted-host www.silx.org
python setup.py build
python setup.py test
sudo pip install .
```

2.3 Manual Installation for any operating system

2.3.1 Install the dependencies

Most Linux distribution come with a Python environment configured. Complete it with the needed dependencies.

- Python 2.7 or 3.4+
- numpy - http://www.numpy.org

For full functionality of FabIO the following modules need to be installed:

- Pillow (python imaging library) - http://www.pythonware.com
- lxml (library for reading XSDimages)
- PyQt4 for the fabio_viewer program

Once done, follow the classical procedure (similar to Windows or MacOSX):

- Retrieve the sources from github:
  - The master development branch
  - The latest release
- unzip the file in a directory
- open a terminal in the unzipped archive directory
- run:

```
sudo pip install -r ci/requirements_travis.txt --trusted-host www.silx.org
python setup.py build
python setup.py test
sudo pip install .
```

Most likely you will need to gain root privileges (with sudo in front of the command) to install packages.

2.4 Development versions

The newest development version can be obtained by checking it out from the git repository:

```
git clone https://github.com/silx-kit/fabio
cd fabio
python setup.py build test
sudo pip install .
```

For Ubuntu/Debian users, you will need:

- python-imaging
- python-imaging-tk
- python-numpy
- python-dev

```
sudo apt-get install python-imaging python-imaging-tk python-numpy
```

2.4.1 Automatic debian packaging

Debian 6 and 7:

We provide a debian-package builder based on stdeb, building a package for Python2:

```
sudo apt-get install python-stdeb
./build-deb7.sh
```

which builds a debian package and installs them in a single command. Handy for testing, but very clean, see hereafter

2.5 Debian 8 and newer

There is also a script which builds a bunch of *real* debian packages: *build-deb8.sh* It will build a bunch of 6 debian packages:

```
* *fabio-viewer*: the GUI for visualizing diffraction images
* *fabio-doc*: the documumentation package
* *python3-fabio*: library built for Python3
* *python3-fabio-dbg*: debug symbols for Python3
* *python-fabio*: library built for Python2
* *python-fabio-dbg*: debug symbols for Python2
```

For this, you need a complete debian build environment:

```
sudo apt-get install cython cython-dbg cython3 cython3-dbg debhelper dh-python \
python-all-dev python-all-dbg python-h5py \
python-lxml python-lxml-dbg python-matplotlib python-matplotlib-dbg python-numpy\
python-numpy-dbg python-qt4 python-qt4-dbg python-sphinx \
python-sphinxcontrib.programoutput python-tk python-tk-dbg python3-all-dev python3-all-dbg \
python3-lxml python3-lxml-dbg python3-matplotlib \
python3-matplotlib-dbg python3-numpy python3-numpy-dbg python3-pyqt4 python3-pyqt4-dbg \
 python3-sphinx python3-sphinxcontrib.programoutput \
python3-tk python3-tk-dbg

./build-debian-full.sh
```

2.6 Test suite

FabIO has a comprehensive test-suite to ensure non regression. When you run the test for the first time, many test images will be download and converted into various compressed format like gzip and bzip2 (this takes a lot of

time).

Be sure you have an internet connection and your proxy setting are correctly defined in the environment variable *http_proxy*. For example if you are behind a firewall/proxy:

Under Linux and MacOSX:

```
export http_proxy=http://proxy.site.org:3128
```

Under Windows:

```
set http_proxy=http://proxy.site.org:3128
```

Where *proxy.site.org* and *3128* correspond to the proxy server and port on your network. At ESRF, you can get this piece of information by phoning to the hotline: 24-24.

Many tests are there to deal with malformed files, don't worry if the programs complains in warnings about "bad files", it is done on purpose to ensure robustness in FabIO.

2.6.1 Run test suite from installation directory

To run the test:

```
python setup.py build test
```

2.6.2 Run test suite from installed version

Within Python (or ipython):

```
>>> import fabio
>>> fabio.tests()
```

2.6.3 Test coverage

FabIO comes with 33 test-suites (145 tests in total) representing a coverage of 60%. This ensures both non regression over time and ease the distribution under different platforms: FabIO runs under Linux, MacOSX and Windows (in each case in 32 and 64 bits) with Python versions 2.7, 3.4 and 3.5. Under linux it has been tested on i386, x86_64, arm, ppc, ppc64le. FabIO may run on other untested systems but without warranty.

Test coverage report for fabio

Measured on *fabio* version 0.4.0, 15/07/2016

Table 2.1: Test suite coverage

Name	Stmts	Exec	Cover
GEimage	103	56	54.4 %
HiPiCimage	59	9	15.3 %
OXDimage	344	274	79.7 %
TiffIO	821	560	68.2 %
__init__	31	24	77.4 %
_version	32	26	81.2 %
adscimage	75	46	61.3 %
binaryimage	53	17	32.1 %
bruker100image	247	165	66.8 %
brukerimage	202	169	83.7 %
Continued on next page			

Table 2.1 – continued from previous page

Name	Stmts	Exec	Cover
cbfimage	520	221	42.5 %
compression	255	191	74.9 %
converters	18	15	83.3 %
directories	21	16	76.2 %
dm3image	135	18	13.3 %
edfimage	584	398	68.2 %
eigerimage	101	54	53.5 %
ext/__init__	0	0	0.0 %
fabioimage	325	241	74.2 %
fabioutils	341	267	78.3 %
file_series	141	62	44.0 %
fit2dimage	90	75	83.3 %
fit2dmaskimage	75	72	96.0 %
fit2dspreadsheetimage	43	8	18.6 %
hdf5image	89	49	55.1 %
kcdimage	92	72	78.3 %
mar345image	268	244	91.0 %
marccdimage	65	58	89.2 %
nexus	188	96	51.1 %
numpyimage	63	50	79.4 %
openimage	118	85	72.0 %
pilatusimage	36	31	86.1 %
pixiimage	98	25	25.5 %
pnmimage	132	84	63.6 %
raxisimage	102	93	91.2 %
speimage	169	163	96.4 %
tifimage	181	62	34.3 %
xsdimage	93	70	75.3 %
fabio total	6310	4166	66.0 %

CHAPTER

THREE

TUTORIALS ON FABIO

3.1 Nexus -> CBF

In this tutorial we will see how to export a Nexus archive produced by the Eiger detector from Dectris into a bunch of CBF files, similar to the one generated by Pilatus detectors, using the FabIO library to write images (and h5py which actually reads them).

Nota: HDF5 files produced by Nexus detector use a specific LZ4/bitshuffle plugin for reading/writing. They require recent version of hdf5 (>= 1.8.10), h5py (>= 2.5.0) and those plugins installed:

- https://github.com/nexusformat/HDF5-External-Filter-Plugins/tree/master/LZ4
- https://github.com/kiyo-masui/bitshuffle

Under Windows, those plugins can easily be installed via this

repository which provides binary DLLs: | https://github.com/silx-kit/hdf5plugin

```
#Run as first to set the plugin path, very important to handle Eiger data:
import os
os.environ["HDF5_PLUGIN_PATH"]="/usr/lib/x86_64-linux-gnu/hdf5/plugins"

import fabio
```

In this example we will use the Eiger 4M dataset which can be obtained from Dectris: https://www.dectris.com/datasets.html. You may register to get download access.

```
images = fabio.open("collect_01_00001_master.h5")
print(images)
```

```
Eiger dataset with 1800 frames from collect_01_00001_master.h5
```

Each "EigerImage" object contains a list to the corresponding HDF5 opened with h5py. So one can retrieve all metadata associated:

```
header = {}
for key, value in images.h5["entry/instrument/detector"].items():
    try:
        val = value[()]
    except:
        print("%s: unprintable"%key)
    else:
        print("%s: %s"%(key, val))
        header[key] = val
```

```
beam_center_x: 1051.0
beam_center_y: 1001.0
bit_depth_readout: 32
```

```
count_time: 0.099996
countrate_correction_applied: 0
description: b'Dectris Eiger 4M'
detectorSpecific: unprintable
detector_distance: 0.00733
detector_number: b'E-08-0102'
detector_readout_time: 3.78e-06
efficiency_correction_applied: 0
flatfield_correction_applied: 1
frame_time: 0.1
geometry: unprintable
pixel_mask_applied: 0
sensor_material: b'Si'
sensor_thickness: 0.00032
threshold_energy: 5635.65
virtual_pixel_correction_applied: 1
x_pixel_size: 7.5e-05
y_pixel_size: 7.5e-05
```

Now we can translate every single image into a CBF file, here we do only a dozen of then:

```
for idx, frame in enumerate(images):
    cbf = fabio.cbfimage.cbfimage(header=header,data=frame.data)
    fname = "collect_01_00001_%04i.cbf"%idx
    cbf.write(fname)
    print(fname)
    if idx>10:
        break
```

```
collect_01_00001_0000.cbf
collect_01_00001_0001.cbf
collect_01_00001_0002.cbf
collect_01_00001_0003.cbf
collect_01_00001_0004.cbf
collect_01_00001_0005.cbf
collect_01_00001_0006.cbf
collect_01_00001_0007.cbf
collect_01_00001_0008.cbf
collect_01_00001_0009.cbf
collect_01_00001_0010.cbf
collect_01_00001_0011.cbf
```

This is how to display an image using the notebook backend:

```
%pylab nbagg
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
imshow(numpy.log(fabio.open("collect_01_00001_0010.cbf").data))
```

```
/scisoft/users/jupyter/jupy34/lib/python3.4/site-packages/ipykernel/__main__.py:1: RuntimeWarning
  if __name__ == '__main__':
```

```
<IPython.core.display.Javascript object>
```

```
<matplotlib.image.AxesImage at 0x7f47531e26a0>
```

3.1.1 Conclusion

FabIO offers a versatile way for manipulating image files and convert them.

```
#For info:
print(fabio.version)
print(fabio.eigerimage.h5py.version.version)
print(fabio.eigerimage.h5py.version.hdf5_version)

0.4.0-dev6
2.6.0
1.8.13
```

CHAPTER

FOUR

BENCHMARKS

Those benchmarks have been done with all data already in cache using a Intel Xeon E5520 @ 2.27GHz running Debian 7 and FabIO 0.2.2... Those data are now out-dated.

Table 4.1: Execution time for reading a file, benchmarked using the ‘‘timeit” module.

FabIO module	Filename	Size	Py2.7 v0.1.4	Py2.7 v0.2.0	Py3.2 v0.2.0
adscimage	mb_LP_1_001.img	9 Mpix	1.16 s	1.12 s	2.67 s
brukerimage	Cr8F8140k103.0026	256kpix	796 µs	795 µs	1.43 ms
cbfimage	run2_1_00148.cbf	6 Mpix	173 ms	70.8 ms	70.8 ms
edfimage	F2K_Seb_Lyso0675.edf	4 Mpix	512 µs	597 µs	595 µs
fit2dmaskimage	fit2d_click.msk	1 Mpix	30.4 ms	30.5 ms	28.4 ms
GEimage	GE_aSI_detector_image_1529	4 Mpix	5.37 ms	5.44 ms	4.3 ms
kcdimage	i01f0001.kcd	360kpix	130 ms	121 ms	292 ms
mar345image	example.mar2300	5 Mpix	78 ms	77 ms	77 ms
marccdimage	corkcont2_H_0089.mccd	4 Mpix	9.28 ms	9.01 ms	17.2 ms
OXDimage	b191_1_9_1.img	256kpix	5.75 ms	5.67 ms	8.35 ms
pnmimage	image0001.pgm	1 Mpix	3.29 ms	3.25 ms	5.71 ms
raxisimage	mgzn-20hpt.img	3 Mpix	53.5 ms	57.1 ms	69.2 ms
tifimage	oPPA_5grains_0001.tif	4 Mpix	59.9 ms	58.4 ms	105 ms
xsdimage	XSDataImage.xml	256kpix	13.3 ms	12.9 ms	18.4 ms

The Python3 version is sometimes twice slower then the Python2 version. As the codebase is the same this regression is not due to FabIO but to the programming language itself. The Performances obtained using Python3.4 is now again close to the one of Python2.7.

CHAPTER

FIVE

CHANGELOG

5.1 FabIO-0.4.0 (07/2016):

- Write support for Bruker100 (contribution from Sigmund Neher)
- Read support for Princeton instrumentation cameras (contribution from Clemens Percher)
- Read support for FIT2D binary format
- Read support for Eiger detector (Dectris) and generic HDF5 (partial)
- Switch ESRF-contributed file formats to MIT license (more liberal)
- Drop support for python 2.6, 3.2 and 3.3. Supports only 2.7 and 3.4+
- Include TiffIO into core of FabIO (no more third-party)
- Refactor mar345 (contributed by Henri Payno)
- Enhanced byte-offset compression using Cython
- Move master repository to silx-kit (https://github.com/silx-kit)

5.2 FabIO-0.3.0 (12/2015):

- Migrate to PEP8 for class names.
- Use a factory & registry instead of fiddling in sys.modules for instance creation
- dim1, dim2, bpp and bytecode are properties. Use their private version while reading.
- FabioImage.header["filename"] has disappeared. Use FabioImage.filename instead.
- Automatic backported package down to debian-8
- Compatibility checked with 2.6, 2.7, 3.2, 3.3, 3.4 and 3.5
- Continuous integration based on appveyor (windows) and travis (linux)
- Support for numpy 2d-array and PNM saving
- Move away from Sourceforge -> Github.

5.3 FabIO-0.2.2 (07/2015):

- work on the auto-documentation on ReadTheDocs (see http://fabio.readthedocs.org)
- fix regression when reading BytesIO
- Python3 compatibility
- prepare multiple package for debian

5.4 FabIO-0.2.1 (02/2015):

- Fix issues with variable endianness (tested PPC, PPC64le, i386, x86-64, ARM processors)
- Optimization of byte-offset reader (about 20% faster on some processors)

5.5 FabIO-0.2.0 (01/2015):

- Compatibility with Python3 (tested on Python 2.6, 2.7, 3.2 and 3.4)
- Support for Mar555 flat panel
- Optimization of CBF reader (about 2x faster)
- include tests into installed module (and download in /tmp)

5.6 FabIO-0.1.4:

- Work on compatibility with Python3
- Specific debian support with test images included but no auto-generated files
- Image viewer (fabio_viewer) based on Qt4 (Thanks for Gaël Goret)
- Reading images from HDF5 datasets
- Read support for "MRC" images
- Read support for "Pixi detector (Thanks for Jon Wright)
- Read support for "Raxis" images from Rigaku (Thanks to Brian Pauw)
- Write support for fit2d mask images
- Drop support for python 2.5 + Cythonization of other algorithms

5.7 FabIO-0.1.3:

- Fixed a memory-leak in mar345 module
- Improved support for bruker format (writer & reader)
- Fixed a bug in EDF headers (very long headers)
- Provide template for new file-formats
- Fix a bug related to PIL in new MacOSX
- Allow binary-images to be read from end

5.8 FabIO-0.1.2 (04/2013):

- Fixed a bug in fabioimage.write (impacted all writers)
- added Sphinx documentation "python setup.py build_doc"
- PyLint compliance of some classes (rename, ...)
- tests from installer with "python setup.py build test"

5.9 FabIO-0.1.1:

- Merged Mar345 image reader and writer with cython bindings (towards python3 compliance)
- Improve CBF image writing under windows
- Bz2, Gzip and Flat files are managed through a common way ... classes are more (python v2.5) or less (python v2.7) overloaded
- Fast EDF reading if one assumes offsets are the same between files, same for ROIs

5.10 FabIO-0.1.0:

- OXD reader improved and writer implemented
- Mar345 reader improved and writer implemented
- CBF writer implemented
- Clean-up of the code & bug fixes
- Move towards python3
- Make PIL optional dependency
- Preliminary Python3 support (partial).

5.11 FabIO-0.0.8:

- Support for Tiff using TiffIO module from V.A.Solé
- Clean-up of the code & bug fixes

5.12 FabIO-0.0.7 (03/2011):

- Support for multi-frames EDF files
- Support for XML images/2D arrays used in EDNA
- new method: fabio.open(filename) that is an alias for fabio.openimage.openimage(filename)

5.13 FabIO-0.0.6 (01/2011):

- Support for CBF files from Pilatus detectors
- Support for KCD files from Nonius Kappa CCD images
- write EDF with their native data type (instead of uint16 by default)

5.14 FabIO-0.0.4 (2009):

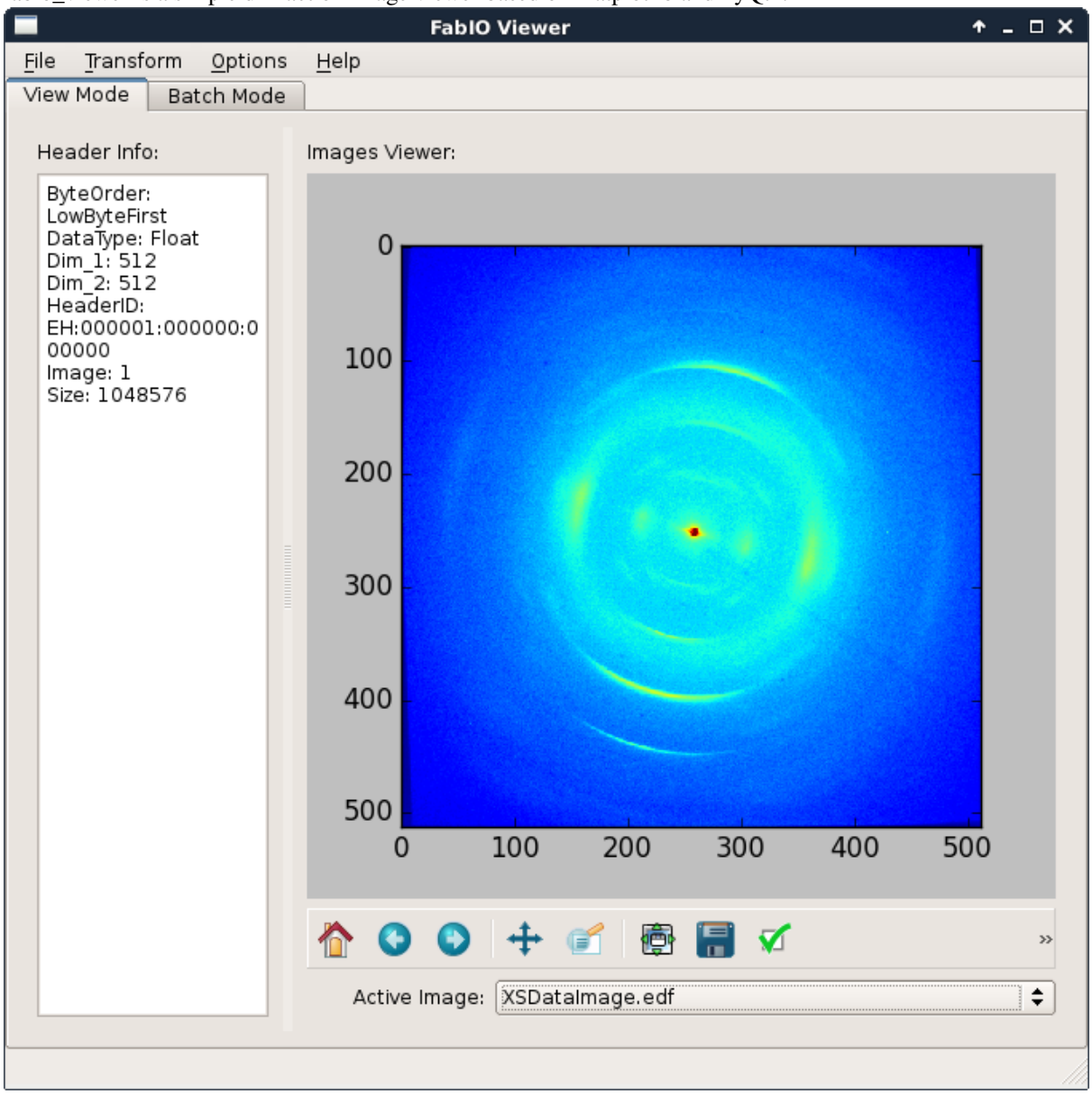
- Support for EDF and many other formats

CHAPTER

SIX

FABIO VIEWER

fabio_viewer is a simple diffraction image viewer based on matplotlib and PyQt4.




```
$ fabio_viewer --help
usage: fabio_viewer img1 img2 ... imgn

FabIO_viewer is a portable diffraction images viewer/converter: * Written in
Python, it combines the functionalities of the I/O library fabIO with a user
friendly Qt4 GUI. * Image converter is also a light viewer based on the
```

```
visualization tool provided by the module matplotlib.

positional arguments:
  images

optional arguments:
  -h, --help     show this help message and exit
  -V, --version  Print version & quit

Based on FabIO version 0.4.0-dev6
```

CHAPTER

SEVEN

FABIO PACKAGE

7.1 fabio Package

FabIO module

fabio.**benchmarks**()
Run the benchmarks

fabio.**tests**()
Run the FabIO test suite.

If the test-images are not already installed (via the debian package for example), they need to be downloaded from sourceforge.net, which make take a while. Ensure your network connection is operational and your proxy settings are correct, for example:

export http_proxy=http://proxy.site.com:3128

7.2 fabio.fabioimage Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

and Jon Wright, Jerome Kieffer: ESRF

class fabio.fabioimage.**FabioImage**(*data=None*, *header=None*)
Bases: object

A common object for images in fable

Contains a numpy array (.data) and dict of meta data (.header)

add(*other*)
Add another Image - warning, does not clip to 16 bit images by default

bpp
Getter for bpp: data superseeds _bpp

bytecode
Getter for bpp: data superseeds _bytecode

static check_data(*data=None*)
Empty for fabioimage but may be populated by others classes, especially for format accepting only integers

Parameters **data** – array like

Returns numpy array or None

static check_header(*header=None*)
Empty for fabioimage but may be populated by others classes

Parameters **header** – dict like object

Returns Ordered dict

classname
Retrieves the name of the class :return: the name of the class

convert(*dest*)
Convert a fabioimage object into another fabioimage object (with possible conversions) :param dest: destination type "EDF", "edfimage" or the class itself :return: instance of the new class

dim1
Getter for dim1: data superseeds _dim1

dim2
Getter for dim2: data superseeds _dim2

classmethod factory(*name*)
A kind of factory... for image_classes

Parameters **name** (*str*) – name of the class to instantiate

Returns an instance of the class

Return type fabioimage

get_bpp()
Getter for bpp: data superseeds _bpp

get_bytecode()
Getter for bpp: data superseeds _bytecode

get_dim1()
Getter for dim1: data superseeds _dim1

get_dim2()
Getter for dim2: data superseeds _dim2

get_header_keys()

getclassname()
Retrieves the name of the class :return: the name of the class

getframe(*num*)
returns the file numbered 'num' in the series as a fabioimage

getheader()
returns self.header

getmax()
Find max value in self.data, caching for the future

getmean()
return the mean

getmin()
Find min value in self.data, caching for the future

getstddev()
return the standard deviation

header_keys

integrate_area(*coords*)
Sums up a region of interest if len(coords) == 4 -> convert coords to slices if len(coords) == 2 -> use as slices floor -> ? removed as unused in the function.

load(**arg*, ***kwarg*)
Wrapper for read

make_slice(*coords*)
Convert a len(4) set of coords into a len(2) tuple (pair) of slice objects the latter are immutable, meaning the roi can be cached

next()
returns the next file in the series as a fabioimage

previous()
returns the previous file in the series as a fabioimage

read(*filename*, *frame=None*)
To be overridden - fill in self.header and self.data

readROI(*filename*, *frame=None*, *coords=None*)
Method reading Region of Interest. This implementation is the trivial one, just doing read and crop

readheader(*filename*)
Call the _readheader function...

rebin(*x_rebin_fact*, *y_rebin_fact*, *keep_I=True*)
Rebin the data and adjust dims :param x_rebin_fact: x binning factor :param y_rebin_fact: y binning factor :param keep_I: shall the signal increase ? :type x_rebin_fact: int :type y_rebin_fact: int :type keep_I: boolean

registry = OrderedDict([('fabioimage', <class 'fabio.fabioimage.FabioImage'>), ('edfimage', <class 'fabio.edfimage

resetvals()
Reset cache - call on changing data

save(*fname*)
wrapper for write

set_bpp(*value*)
Setter for bpp

set_bytecode(*value*)
Setter for bpp

set_dim1(*value*)
Setter for dim1

set_dim2(*value*)
Setter for dim2

set_header_keys(*value*)

toPIL16(*filename=None*)
Convert to Python Imaging Library 16 bit greyscale image

update_header(***kwds*)
update the header entries by default pass in a dict of key, values.

write(*fname*)
To be overwritten - write the file

class fabio.fabioimage.**FabioMeta**(*name*, *bases*, *dct*)
Bases: type

Metaclass used to register all image classes inheriting from fabioImage

fabio.fabioimage.**fabioimage**
alias of FabioImage

7.3 fabio.fabioutils Module

General purpose utilities functions for fabio

class fabio.fabioutils.**BZ2File**(*name*, *mode='r'*, *buffering=0*, *compresslevel=9*)
Bases: bz2.BZ2File

Wrapper with lock

getSize()

setSize(*value*)

size

class fabio.fabioutils.**BytesIO**(*data*, *fname=None*, *mode='r'*)
Bases: StringIO.StringIO

just an interface providing the name and mode property to a BytesIO

BugFix for MacOSX mainly

getSize()

setSize(*size*)

size

class fabio.fabioutils.**DebugSemaphore**(**arg*, ***kwarg*)
Bases: threading._Semaphore

threading.Semaphore like class with helper for fighting dead-locks

acquire(**arg*, ***kwarg*)

blocked = []

release(**arg*, ***kwarg*)

write_lock = <threading._Semaphore object at 0x7fae12af46d0>

class fabio.fabioutils.**File**(*name*, *mode='rb'*, *buffering=0*, *temporary=False*)
Bases: file

wrapper for "file" with locking

close()

getSize()

setSize(*size*)

size

class fabio.fabioutils.**FilenameObject**(*stem=None*, *num=None*, *directory=None*, *format_=None*, *extension=None*, *postnum=None*, *digits=4*, *filename=None*)
Bases: object

The 'meaning' of a filename ...

deconstruct_filename(*filename*)
Break up a filename to get image type and number

str()
Return a string representation

tostring()
convert yourself to a string

class fabio.fabioutils.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*)
Bases: gzip.GzipFile

Just a wrapper forgzip.GzipFile providing the correct seek capabilities for python 2.5

measure_size()

exception fabio.fabioutils.**NotGoodReader**
Bases: exceptions.RuntimeError

The reader used is probably not the good one

class fabio.fabioutils.**UnknownCompressedFile**(*name*, *mode='rb'*, *buffering=0*)
Bases: fabio.fabioutils.File

wrapper for "File" with locking

fabio.fabioutils.**construct_filename**(*filename*, *frame=None*)
Try to construct the filename for a given frame

fabio.fabioutils.**deconstruct_filename**(*filename*)
Function for backward compatibility. Deprecated

fabio.fabioutils.**deprecated**(*func*)
used to deprecate a function/method: prints a lot of warning messages to enforce the modification of the code

fabio.fabioutils.**exists**(*path*)
Test whether a path exists.

Replaces os.path.exists and handles in addition "::" based URI as defined in http://odo.pydata.org/en/latest/uri.html#separating-parts-with

Parameters **path** – string

Returns boolean

fabio.fabioutils.**extract_filenumber**(*name*)
extract file number

fabio.fabioutils.**getnum**(*name*)
try to figure out a file number # guess it starts at the back

fabio.fabioutils.**isAscii**(*name*, *listExcluded=None*)

Parameters

- **name** – string to check
- **listExcluded** – list of char or string excluded.

Returns True of False whether name is pure ascii or not

fabio.fabioutils.**jump_filename**(*name*, *num*, *padding=True*)
jump to number

fabio.fabioutils.**next_filename**(*name*, *padding=True*)
increment number

fabio.fabioutils.**nice_int**(*s*)
Workaround that int('1.0') raises an exception

Parameters **s** – string to be converted to integer

fabio.fabioutils.**numstem**(*name*)
cant see how to do without reversing strings Match 1 or more digits going backwards from the end of the string

fabio.fabioutils.**pad**(*mystr*, *pattern=' '*, *size=80*)
Performs the padding of the string to the right size with the right pattern

Parameters

- **mystr** – input string
- **pattern** – the filling pattern
- **size** – the size of the block

Returns the padded string to a multiple of size

fabio.fabioutils.**previous_filename**(*name*, *padding=True*)
decrement number

fabio.fabioutils.**toAscii**(*name*, *excluded=None*)

Parameters

- **name** – string to check
- **excluded** – tuple of char or string excluded (not list: they are mutable).

Returns the name with all non valid char removed

7.4 fabio.file_series Module

7.4.1 Authors:

- Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk
- Jon Wright, ESRF

class fabio.file_series.**file_series**(*list_of_strings*)
Bases: list

Represents a series of files to iterate has an idea of a current position to do next and prev

You also get from the list python superclass: append count extend insert pop remove reverse sort

current()
Current position in a sequence

current_image()
Current image in sequence

Returns fabioimage

current_object()
Current image in sequence

Returns file_object

first()
First image in series

first_image()
First image in a sequence

Returns fabioimage

first_object()
First image in a sequence

Returns file_object

jump(*num*)
Goto a position in sequence

jump_image(*num*)
Jump to and read image

Returns fabioimage

jump_object(*num*)
Jump to and read image

Returns file_object

last()
Last in series

last_image()
Last image in a sequence

Returns fabioimage

last_object()
Last image in a sequence

Returns file_object

len()
Number of files

next()
Next in a sequence

next_image()
Return the next image

Returns fabioimage

next_object()
Return the next image

Returns file_object

previous()
Prev in a sequence

previous_image()
Return the previous image

Returns fabioimage

previous_object()
Return the previous image

Returns file_object

class fabio.file_series.**filename_series**(*filename*)
Much like the others, but created from a string filename

current()
return current filename string

current_image()
returns the current image as a fabioimage

current_object()
returns the current filename as a fabio.FilenameObject

jump(*num*)
jump to a specific number

jump_image(*num*)
returns the image number as a fabioimage

jump_object(*num*)
returns the filename num as a fabio.FilenameObject

next()
increment number

next_image()
returns the next image as a fabioimage

next_object()
returns the next filename as a fabio.FilenameObject

prev_image()
returns the previos image as a fabioimage

previous()
decrement number

previous_object()
returns the previous filename as a fabio.FilenameObject

fabio.file_series.**new_file_series**(*first_object*, *nimages=0*, *step=1*, *traceback=False*)
A generator function that creates a file series starting from a a fabioimage. Iterates through all images in a file (if more than 1), then proceeds to the next file as determined by fabio.next_filename.

Parameters

- **first_object** – the starting fabioimage, which will be the first one yielded in the sequence
- **nimages** – the maximum number of images to consider step: step size, will yield the first and every step'th image until nimages is reached. (e.g. nimages = 5, step = 2 will yield 3 images (0, 2, 4)
- **traceback** – if True causes it to print a traceback in the event of an exception (missing image, etc.). Otherwise the calling routine can handle the exception as it chooses
- **yields** – the next fabioimage in the series. In the event there is an exception, it yields the sys.exec_info for the exception instead. sys.exec_info is a tuple: (exceptionType, exceptionValue, exceptionTraceback) from which all the exception information can be obtained.

Suggested usage:

```
for obj in new_file_series( ... ):
  if not isinstance(obj, fabio.fabioimage.fabioimage ):
    # deal with errors like missing images, non readable files, etc
    # e.g.
    traceback.print_exception(obj[0], obj[1], obj[2])
```

fabio.file_series.**new_file_series0**(*first_object*, *first=None*, *last=None*, *step=1*)
Created from a fabio image first and last are file numbers

class fabio.file_series.**numbered_file_series**(*stem*, *first*, *last*, *extension*, *digits=4*, *padding='Y'*, *step=1*)
Bases: fabio.file_series.file_series

mydata0001.edf = "mydata" + 0001 + ".edf" mydata0002.edf = "mydata" + 0002 + ".edf" mydata0003.edf = "mydata" + 0003 + ".edf"

7.5 fabio.openimage Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:henning.sorensen@risoe.dk

mods for fabio by JPW modification for HDF5 by Jérôme Kieffer

fabio.openimage.**do_magic**(*byts*, *filename*)
Try to interpret the bytes starting the file as a magic number

fabio.openimage.**openheader**(*filename*)
return only the header

fabio.openimage.**openimage**(*filename*, *frame=None*)
Try to open an image

7.6 fabio.adscimage Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

- mods for fabio by JPW

class fabio.adscimage.**AdscImage**(**args*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

Read an image in ADSC format (quite similar to edf?)

read(*fname*, *frame=None*)
read in the file

swap_needed()

write(*fname*)
Write adsc format

fabio.adscimage.**adscimage**
alias of AdscImage

7.7 fabio.binaryimage Module

Authors: Gael Goret, Jerome Kieffer, ESRF, France

Emails: gael.goret@esrf.fr, jerome.kieffer@esrf.fr Brian Richard Pauw <brian@stack.nl>

Binary files images are simple none-compressed 2D images only defined by their : data-type, dimensions, byte order and offset

This simple library has been made for manipulating exotic/unknown files format.

class fabio.binaryimage.**BinaryImage**(**args*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

This simple library has been made for manipulating exotic/unknown files format.

Binary files images are simple none-compressed 2D images only defined by their: data-type, dimensions, byte order and offset

if offset is set to a negative value, the image is read using the last data but n data in the file, skipping any header.

estimate_offset_value(*fname*, *dim1*, *dim2*, *bytecode='int32'*)
Estimates the size of a file

read(*fname*, *dim1*, *dim2*, *offset=0*, *bytecode='int32'*, *endian='<'*)
Read a binary image

Parameters

- **fname** (*string*) – file name
- **dim1** – image dimensions (Fast index)
- **dim2** – image dimensions (Slow index)
- **offset** – starting position of the data-block. If negative, starts at the end.
- **bytecode** – can be "int8","int16","int32","int64","uint8","uint16","uint32","uint64","float32","float64",...
- **endian** – among short or long endian ("<" or ">")

static swap_needed(*endian*)
Decide if we need to byteswap

write(*fname*)

fabio.binaryimage.**binaryimage**
alias of BinaryImage

7.8 fabio.bruker100image Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

Jérôme Kieffer, ESRF, Grenoble, France Sigmund Neher, GWDG, Göttingen, Germany

class fabio.bruker100image.**Bruker100Image**(*data=None*, *header=None*)
Bases: fabio.brukerimage.BrukerImage

bpp_to_numpy = {1: <type 'numpy.uint8'>, 2: <type 'numpy.uint16'>, 4: <type 'numpy.int32'>}

gen_header()
Generate headers (with some magic and guesses) format is Bruker100

gen_overflow()
Generate an overflow table, including the underflow, marked as 65535 .

gen_underflow100()
Generate an underflow table

overflows_one_byte()
Generate one-byte overflow table

overflows_two_byte()
Generate two byte overflow table

read(*fname*, *frame=None*)
data is stored in three blocks: data (uint8), overflow (uint32), underflow (int32). The blocks are zero paded to a multiple of 16 bits

toPIL16(*filename=None*)

underflows()
Generate underflow table

version = 100

write(*fname*)
Write a bruker image

fabio.bruker100image.**bruker100image**
alias of Bruker100Image

7.9 fabio.brukerimage Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

Based on: openbruker,readbruker, readbrukerheader functions in the opendata module of ImageD11 written by Jon Wright, ESRF, Grenoble, France

Writer by Jérôme Kieffer, ESRF, Grenoble, France

class fabio.brukerimage.**BrukerImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

Read and eventually write ID11 bruker (eg smart6500) images

TODO: int32 -> float32 conversion according to the "linear" keyword. This is done and works but we need to check with other program that we are appliing the right formula and not the reciprocal one.

HEADERS_KEYS = ['FORMAT', 'VERSION', 'HDRBLKS', 'TYPE', 'SITE', 'MODEL', 'USER', 'SAMPLE', 'SETN

SPACER = '\x1a\x04'

basic_translate (*fname=None*)
Does some basic population of the headers so that the writing is possible

bpp_to_numpy = {1: <type 'numpy.uint8'>, 2: <type 'numpy.uint16'>, 4: <type 'numpy.uint32'>}

calc_bpp (*data=None*, *max_entry=4096*)
Calculate the number of byte per pixel to get an optimal overflow table.

Returns byte per pixel

gen_header ()
Generate headers (with some magic and guesses) :param format can be 86 or 100

gen_overflow ()
Generate an overflow table

read (*fname*, *frame=None*)
Read in and unpack the pixels (including overflow table

version = 86

write (*fname*)
Write a bruker image

fabio.brukerimage.**brukerimage**
alias of BrukerImage

7.10 fabio.cbfimage Module

Authors: Jérôme Kieffer, ESRF email:jerome.kieffer@esrf.fr

Cif Binary Files images are 2D images written by the Pilatus detector and others. They use a modified (simplified) byte-offset algorithm.

CIF is a library for manipulating Crystallographic information files and tries to conform to the specification of the IUCR

class fabio.cbfimage.**CIF** (*_strFilename=None*)
Bases: dict

This is the CIF class, it represents the CIF dictionary; and as a a python dictionary thus inherits from the dict built in class.

keys are always unicode (str in python3) values are bytes

BINARY_MARKER = '–CIF-BINARY-FORMAT-SECTION–'

BLANK = [' ', '\t', '\r', '\n', '\r\n', '\n\r']

DATA = 'data_'

DOUBLE_QUOTE = '"'

EOL = ['\r', '\n', '\r\n', '\n\r']

GLOBAL = 'global_'

HASH = '#'

LOOP = 'loop_'

static LoopHasKey(*loop*, *key*)
Returns True if the key (string) exist in the array called loop

QUESTIONMARK = '?'

SAVE = 'save_'

SEMICOLUMN = ';'

SINGLE_QUOTE = "'"

START_COMMENT = ("'", '"')

STOP = 'stop_'

UNDERSCORE = '_'

exists(*sKey*)
Check if the key exists in the CIF and is non empty. :param sKey: CIF key :type sKey: string :param cif: CIF dictionary :return: True if the key exists in the CIF dictionary and is non empty :rtype: boolean

existsInLoop(*sKey*)
Check if the key exists in the CIF dictionary. :param sKey: CIF key :type sKey: string :param cif: CIF dictionary :return: True if the key exists in the CIF dictionary and is non empty :rtype: boolean

i = '\t'

static isAscii(*text*)
Check if all characters in a string are ascii,

Parameters **_strIn** (*python string*) – input string

Returns boolean

Return type boolean

loadCHIPLOT(*_strFilename*)
Load the powder diffraction CHIPLOT file and returns the pd_CIF dictionary in the object

Parameters **_strFilename** (*string*) – the name of the file to open

Returns the CIF object corresponding to the powder diffraction

Return type dictionary

loadCIF(*_strFilename*, *_bKeepComment=False*)
Load the CIF file and populates the CIF dictionary into the object :param _strFilename: the name of the file to open :type _strFilename: string :param _strFilename: the name of the file to open :type _strFilename: string :return: None

pop(*key*, *default=None*)

popitem(*key*, *default=None*)

readCIF(*_strFilename*, *_bKeepComment=False*)
Load the CIF file and populates the CIF dictionary into the object :param _strFilename: the name of the file to open :type _strFilename: string :param _strFilename: the name of the file to open :type _strFilename: string :return: None

saveCIF(*_strFilename='test.cif'*, *linesep='\n'*, *binary=False*)
Transforms the CIF object in string then write it into the given file :param _strFilename: the of the file to be written :param linesep: line separation used (to force compatibility with windows/unix) :param binary: Shall we write the data as binary (True only for imageCIF/CBF) :return: None

tostring(*_strFilename=None*, *linesep='\n'*)

Converts a cif dictionnary to a string according to the CIF syntax

param _strFilename the name of the filename to be appended in the header of the CIF file

type _strFilename string

param linesep default line separation: can be “

” or ” “

return a string that corresponds to the content of the CIF-file.

class fabio.cbfimage.**CbfImage**(*data=None*, *header=None*, *fname=None*)
Bases: fabio.fabioimage.FabioImage

Read the Cif Binary File data format

BINARAY_SECTION = '–CIF-BINARY-FORMAT-SECTION–'

CIF_BINARY_BLOCK_KEY = '_array_data.data'

PADDING = 512

STARTER = '\x0c\x1a\x04\xd5'

static checkData(*data=None*)

read(*fname*, *frame=None*, *check_MD5=True*, *only_raw=False*)
Read in header into self.header and the data into self.data

Param fname: name of the file

Returns fabioimage instance

read_raw_data(*infile*)
Read and return the raw data chunk

Parameters **infile** – opened file are correct position

Returns raw compressed stream

write(*fname*)
write the file in CBF format :param fname: name of the file :type: string

fabio.cbfimage.**cbfimage**
alias of CbfImage

7.11 fabio.dm3image Module

Authors: Henning O. Sorensen & Erik Knudsen

Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

- Jon Wright, ESRF

class fabio.dm3image.**Dm3Image**(**args*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

Read and try to write the dm3 data format

read(*fname*, *frame=None*)

read_data()

read_tag_entry()

read_tag_group()

read_tag_type()

readbytes(*bytes_to_read*, *format*, *swap=True*)

fabio.dm3image.**dm3image**
alias of Dm3Image

7.12 fabio.edfimage Module

License: GPLv3+

7.12.1 Authors:

- Henning O. Sorensen & Erik Knudsen: Center for Fundamental Research: Metal Structures in Four Dimensions; Risoe National Laboratory; Frederiksborgvej 399; DK-4000 Roskilde; email:erik.knudsen@risoe.dk
- Jon Wright & Jérôme Kieffer: European Synchrotron Radiation Facility; Grenoble (France)

class fabio.edfimage.**EdfImage**(*data=None*, *header=None*, *frames=None*)
Bases: fabio.fabioimage.FabioImage

Read and try to write the ESRF edf data format

appendFrame(*frame=None*, *data=None*, *header=None*)
Method used add a frame to an EDF file :param frame: frame to append to edf image :type frame: instance of Frame :return: None

bpp

bytecode

capsHeader
property: capsHeader of EDF file, i.e. the keys of the header in UPPER case.

static check_header(*header=None*)
Empty for FabioImage but may be populated by others classes

data
property: data of EDF file

delCapsHeader()
deleter for edf capsHeader

delData()
deleter for edf Data

delHeader()
Deleter for edf header

deleteFrame(*frameNb=None*)
Method used to remove a frame from an EDF image. by default the last one is removed. :param frameNb: frame number to remove, by default the last. :type frameNb: integer :return: None

dim1

dim2

dims

fastReadData(*filename=None*)
This is a special method that will read and return the data from another file ... The aim is performances, ... but only supports uncompressed files.

Returns data from another file using positions from current EdfImage

fastReadROI(*filename*, *coords=None*)
Method reading Region of Interest of another file based on metadata available in current EdfImage. The aim is performances, ... but only supports uncompressed files.

Returns ROI-data from another file using positions from current EdfImage

Return type numpy 2darray

getBpp()

getByteCode()

getCapsHeader()
getter for edf headers keys in upper case :return: data for current frame :rtype: dict

getData()
getter for edf Data :return: data for current frame :rtype: numpy.ndarray

getDim1()

getDim2()

getDims()

getHeader()
Getter for the headers. used by the property header,

getNbFrames()
Getter for number of frames

getframe(*num*)
returns the file numbered 'num' in the series as a FabioImage

header
property: header of EDF file

next()
returns the next file in the series as a FabioImage

nframes
Getter for number of frames

previous()
returns the previous file in the series as a FabioImage

read(*fname*, *frame=None*)

Read in header into self.header and the data into self.data

setBpp(*_iVal*)

setByteCode(*_iVal*)

setCapsHeader(*_data*)
Enforces the propagation of the header_keys to the list of frames :param _data: numpy array representing data

setData(*_data*)
Enforces the propagation of the data to the list of frames :param _data: numpy array representing data

setDim1(*_iVal*)

setDim2(*_iVal*)

setHeader(*_dictHeader*)
Enforces the propagation of the header to the list of frames

setNbFrames(*val*)
Setter for number of frames ... should do nothing. Here just to avoid bugs

swap_needed()
Decide if we need to byteswap

:return True if needed, False else and None if not understood

unpack()
Unpack a binary blob according to the specification given in the header and return the dataset

Returns dataset as numpy.ndarray

write (*fname*, *force_type=None*, *fit2dMode=False*)
Try to write a file check we can write zipped also mimics that fabian was writing uint16 (we sometimes want floats)

Parameters **force_type** – can be numpy.uint16 or simply "float"

Returns None

class fabio.edfimage.**Frame** (*data=None*, *header=None*, *number=None*)
Bases: object

A class representing a single frame in an EDF file

bytecode

data
Unpack a binary blob according to the specification given in the header

Returns dataset as numpy.ndarray

getByteCode ()

getData ()
Unpack a binary blob according to the specification given in the header

Returns dataset as numpy.ndarray

getEdfBlock (*force_type=None*, *fit2dMode=False*)

Parameters

- **force_type** (*string or numpy.dtype*) – type of the dataset to be enforced like "float64" or "uint16"
- **fit2dMode** (*boolean*) – enforce compatibility with fit2d and starts counting number of images at 1

Returns ascii header block + binary data block

Return type python bytes with the concatenation of the ascii header and the binary data block

parseheader (*block*)
Parse the header in some EDF format from an already open file

Parameters **block** (*string, should be full ascii*) – string representing the header block.

Returns size of the binary blob

setByteCode (*_iVal*)

setData (*npa=None*)
Setter for data in edf frame

swap_needed ()
Decide if we need to byteswap

fabio.edfimage.**edfimage**
alias of EdfImage

7.13 fabio.eigerimage Module

Eiger data/master file reader for FabIO

Eiger data files are HDF5 files with one group called "entry" and a dataset called "data" in it (now in a data group).

Those dataset are usually compressed using LZ4 and/or bitshuffle compression:

- https://github.com/nexusformat/HDF5-External-Filter-Plugins/tree/master/LZ4

- https://github.com/kiyo-masui/bitshuffle

H5py (>2.5) and libhdf5 (>1.8.10) with the corresponding compression plugin are needed to actually read the data. Under windows, those plugins can easily be installed via this repository: https://github.com/silx-kit/hdf5plugin

class fabio.eigerimage.**EigerImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

FabIO image class for Images from Eiger data files (HDF5)

getframe(*num*)
returns the frame numbered 'num' in the stack if applicable

next()
returns the next frame in the series as a fabioimage

previous()
returns the previous frame in the series as a fabioimage

read(*fname*, *frame=None*)
try to read image :param fname: name of the file

write(*fname*)
try to write image :param fname: name of the file

fabio.eigerimage.**eigerimage**
alias of EigerImage

7.14 fabio.fit2dimage Module

FabIO reader for Fit2D binary images

TODO: handle big-endian files

class fabio.fit2dimage.**Fit2dImage**(**arg*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

FabIO image class for Images for XXX detector

BUFFER_SIZE = 512

ENC = 'ascii'

PIXELS_PER_CHUNK = 128

read(*fname*, *frame=None*)
try to read image

Parameters **fname** – name of the file

fabio.fit2dimage.**fit2dimage**
alias of Fit2dImage

fabio.fit2dimage.**hex_to**(*stg*, *type_='int'*)
convert a 8-byte-long string (bytes) into an int or a float :param stg: bytes string :param type_: "int" or "float"

7.15 fabio.fit2dmaskimage Module

Author: Andy Hammersley, ESRF Translation into python/fabio: Jon Wright, ESRF. Writer: Jérôme Kieffer

class fabio.fit2dmaskimage.**Fit2dMaskImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

Read and try to write Andy Hammersley's mask format

static check_data (*data=None*)

read (*fname*, *frame=None*)

Read in header into self.header and the data into self.data

write (*fname*)
Try to write a file

fabio.fit2dmaskimage.**fit2dmaskimage**
alias of Fit2dMaskImage

7.16 fabio.fit2dspreadsheetimage Module

Read the fit2d ascii image output

- Jon Wright, ESRF

class fabio.fit2dspreadsheetimage.**Fit2dSpreadsheetImage** (*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

Read a fit2d ascii format

read (*fname*, *frame=None*)

Read in header into self.header and the data into self.data

fabio.fit2dspreadsheetimage.**fit2dspreadsheetimage**
alias of Fit2dSpreadsheetImage

7.17 fabio.GEimage Module

fabio.GEimage.**GEimage**
alias of GeImage

class fabio.GEimage.**GeImage** (*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

getframe (*num*)
Returns a frame as a new FabioImage object

next ()
Get the next image in a series as a fabio image

previous ()
Get the previous image in a series as a fabio image

read (*fname*, *frame=None*)
Read in header into self.header and the data into self.data

write (*fname*, *force_type=<type 'numpy.uint16'>*)
Not yet implemented

fabio.GEimage.**demo** ()

7.18 fabio.hdf5image Module

HDF5 image for FabIO

Authors: Jerome Kieffer email: Jerome.Kieffer@terre-adelie.org

Specifications: input should being the form:

filename::path

Only supports ndim=2 or 3 (exposed as a stack of images

class fabio.hdf5image.**Hdf5Image**(**arg*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

FabIO image class for Images from an HDF file

filename::dataset

getframe(*num*)
Returns a frame as a new FabioImage object :param num: frame number

next()
Get the next image in a series as a fabio image

previous()
Get the previous image in a series as a fabio image

read(*fname*, *frame=None*)
try to read image :param fname: filename::datasetpath

write(*fname*, *force_type=<type 'numpy.uint16'>*)

fabio.hdf5image.**hdf5image**
alias of Hdf5Image

7.19 fabio.HiPiCimage Module

Authors: Henning O. Sorensen & Erik Knudsen

Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

- Jon Wright, ESRF

Information about the file format from Masakatzu Kobayashi is highly appreciated

fabio.HiPiCimage.**HiPiCimage**
alias of HipicImage

class fabio.HiPiCimage.**HipicImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

Read HiPic images e.g. collected with a Hamamatsu CCD camera

read(*fname*, *frame=None*)

Read in header into self.header and the data into self.data

7.20 fabio.kcdimage Module

Authors: Jerome Kieffer, ESRF email:jerome.kieffer@esrf.fr

kcd images are 2D images written by the old KappaCCD diffractometer built by Nonius in the 1990's Based on the edfimage.py parser.

class fabio.kcdimage.**KcdImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

Read the Nonius kcd data format

static checkData(*data=None*)

read(*fname*, *frame=None*)

Read in header into self.header and the data into self.data

fabio.kcdimage.**kcdimage**
alias of KcdImage

7.21 fabio.mar345image Module

7.21.1 Authors:

- Henning O. Sorensen & Erik Knudsen: Center for Fundamental Research: Metal Structures in Four Dimensions; Risoe National Laboratory; Frederiksborgvej 399; DK-4000 Roskilde; email:erik.knudsen@risoe.dk
- Jon Wright, Jérôme Kieffer & Gaël Goret: European Synchrotron Radiation Facility; Grenoble (France)

Supports Mar345 imaging plate and Mar555 flat panel

Documentation on the format is available from: http://rayonix.com/site_media/downloads/mar345_formats.pdf

class fabio.mar345image.**Mar345Image**(**args*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

ascii_header(*linesep='\n'*, *size=4096*)
Generate the ASCII header for writing

Parameters

- **linesep** – end of line separator
- **size** – size of the header (without the binary header)

Returns string (unicode) containing the mar345 header

binary_header()

Returns Binary header of mar345 file

static checkData(*data=None*)

nb_overflow_pixels()

read(*fname*, *frame=None*)
Read a mar345 image

write(*fname*)
Try to write mar345 file. This is still in beta version. It uses CCP4 (LGPL) PCK1 algo from JPA

fabio.mar345image.**mar345image**
alias of Mar345Image

7.22 fabio.mrcimage Module

MRC image for FabIO

Authors: Jerome Kieffer email: Jerome.Kieffer@terre-adelie.org

Specifications from: http://ami.scripps.edu/software/mrctools/mrc_specification.php

class fabio.mrcimage.**MrcImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

FabIO image class for Images from a mrc image stack

KEYS = ('NX', 'NY', 'NZ', 'MODE', 'NXSTART', 'NYSTART', 'NZSTART', 'MX', 'MY', 'MZ', 'CELL_A', 'CELL_

getframe(*num*)
Returns a frame as a new FabioImage object :param num: frame number

next()
Get the next image in a series as a fabio image

previous()
Get the previous image in a series as a fabio image

read(*fname*, *frame=None*)
try to read image :param fname: name of the file :param frame:

write(*fname*, *force_type=<type 'numpy.uint16'>*)

fabio.mrcimage.**mrcimage**
alias of MrcImage

7.23 fabio.marccdimage Module

7.23.1 Authors:

- Henning O. Sorensen & Erik Knudsen: Center for Fundamental Research: Metal Structures in Four Dimensions; Risoe National Laboratory; Frederiksborgvej 399; DK-4000 Roskilde; email:erik.knudsen@risoe.dk
- Jon Wright: European Synchrotron Radiation Facility; Grenoble (France)

marccdimage can read MarCCD and MarMosaic images including header info.

JPW : Use a parser in case of typos (sorry?)

class fabio.marccdimage.**MarccdImage**(**args*, ***kwds*)
Bases: fabio.tifimage.TifImage

Read in data in mar ccd format, also MarMosaic images, including header info

fabio.marccdimage.**interpret_header**(*header*, *fmt*, *names*)
given a format and header interpret it

fabio.marccdimage.**make_format**(*c_def_string*)
Reads the header definition in c and makes the format string to pass to struct.unpack

fabio.marccdimage.**marccdimage**
alias of MarccdImage

7.24 fabio.numpyimage Module

Generic numpy file reader for FabIO

class fabio.numpyimage.**NumpyImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

FabIO image class for Images for numpy array dumps

Source: http://docs.scipy.org/doc/numpy/neps/npy-format.html

The first 6 bytes are a magic string: exactly "x93NUMPY".

The next 1 byte is an unsigned byte: the major version number of the file format, e.g. x01.

The next 1 byte is an unsigned byte: the minor version number of the file format, e.g. x00. Note: the version of the file format is not tied to the version of the numpy package.

The next 2 bytes form a little-endian unsigned short int: the length of the header data HEADER_LEN.

The next HEADER_LEN bytes form the header data describing the array's format. It is an ASCII string which contains a Python literal expression of a dictionary. It is terminated by a newline ('n') and padded

with spaces ('x20') to make the total length of the magic string + 4 + HEADER_LEN be evenly divisible by 16 for alignment purposes.

The dictionary contains three keys:

"descr" [dtype.descr] An object that can be passed as an argument to the numpy.dtype() constructor to create the array's dtype.

"fortran_order" [bool] Whether the array data is Fortran-contiguous or not. Since Fortran-contiguous arrays are a common form of non-C-contiguity, we allow them to be written directly to disk for efficiency.

"shape" [tuple of int] The shape of the array.

For repeatability and readability, this dictionary is formatted using pprint.pformat() so the keys are in alphabetic order.

Following the header comes the array data. If the dtype contains Python objects (i.e. dtype.hasobject is True), then the data is a Python pickle of the array. Otherwise the data is the contiguous (either C- or Fortran-, depending on fortran_order) bytes of the array. Consumers can figure out the number of bytes by multiplying the number of elements given by the shape (noting that shape=() means there is 1 element) by dtype.itemsize.

The version 1.0 format only allowed the array header to have a total size of 65535 bytes. This can be exceeded by structured arrays with a large number of columns. The version 2.0 format extends the header size to 4 GiB. numpy.save will automatically save in 2.0 format if the data requires it, else it will always use the more compatible 1.0 format.

The description of the fourth element of the header therefore has become:

The next 4 bytes form a little-endian unsigned int: the length of the header data HEADER_LEN.

getframe(*num*)
returns the frame numbered 'num' in the stack if applicable

next()
returns the next frame in the series as a fabioimage

previous()
returns the previous frame in the series as a fabioimage

read(*fname*, *frame=None*)
try to read image :param fname: name of the file

slice_dataset(*frame=None*)

write(*fname*)
try to write image :param fname: name of the file

fabio.numpyimage.**numpyimage**
alias of NumpyImage

7.25 fabio.OXDimage Module

Reads Oxford Diffraction Sapphire 3 images

7.25.1 Authors:

- Henning O. Sorensen & Erik Knudsen: Center for Fundamental Research: Metal Structures in Four Dimensions; Risoe National Laboratory; Frederiksborgvej 399; DK-4000 Roskilde; email:erik.knudsen@risoe.dk
- Jon Wright, Jérôme Kieffer & Gaël Goret: European Synchrotron Radiation Facility; Grenoble (France)

fabio.OXDimage.**OXDimage**
alias of OxdImage

class fabio.OXDimage.**OxdImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

Oxford Diffraction Sapphire 3 images reader/writer class

Note: We assume the binary format is alway little-endian, is this True ?

static checkData(*data=None*)

dec_TY5(*stream*)
Attempt to decode TY5 compression scheme

Parameters **stream** – input stream

Returns 1D array with data

getCompressionRatio()
calculate the compression factor obtained vs raw data

read(*fname*, *frame=None*)

Read in header into self.header and the data into self.data

write(*fname*)
Write Oxford diffraction images: this is still beta Only TY1 compressed images is currently possible :param fname: output filename

class fabio.OXDimage.**Section**(*size*, *dictHeader*)
Bases: object

Small helper class for writing binary headers

getSize(*dtype*)

setData(*key*, *offset*, *dtype*, *default=None*)

Parameters

- **offset** – int, starting position in the section
- **key** – name of the header key
- **dtype** – type of the data to insert (defines the size!)

7.26 fabio.pilatusimage Module

7.26.1 Authors:

- Henning O. Sorensen & Erik Knudsen: Center for Fundamental Research: Metal Structures in Four Dimensions; Risoe National Laboratory; Frederiksborgvej 399; DK-4000 Roskilde; email:erik.knudsen@risoe.dk
- Jon Wright: European Synchrotron Radiation Facility; Grenoble (France)

class fabio.pilatusimage.**PilatusImage**(**args*, ***kwds*)
Bases: fabio.tifimage.TifImage

Read in Pilatus format, also pilatus images, including header info

fabio.pilatusimage.**pilatusimage**
alias of PilatusImage

7.27 fabio.pixiimage Module

Author: Jon Wright, ESRF.

class fabio.pixiimage.**PixiImage**(*data=None*, *header=None*)
Bases: fabio.fabioimage.FabioImage

getframe(*num*)
Returns a frame as a new FabioImage object

next()
Get the next image in a series as a fabio image

previous()
Get the previous image in a series as a fabio image

read(*fname*, *frame=None*)

write(*fname*, *force_type=<type 'numpy.uint16'>*)

fabio.pixiimage.**demo**(*fname*)

fabio.pixiimage.**pixiimage**
alias of PixiImage

7.28 fabio.pnmimage Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:henning.sorensen@risoe.dk

- Jérôme Kieffer: European Synchrotron Radiation Facility; Grenoble (France)

License: GPLv3+

class fabio.pnmimage.**PnmImage**(**arg*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

P1dec(*buf*, *bytecode*)

P2dec(*buf*, *bytecode*)

P3dec(*buf*, *bytecode*)

P4dec(*buf*, *bytecode*)

P5dec(*buf*, *bytecode*)

P6dec(*buf*, *bytecode*)

P7dec(*buf*, *bytecode*)

static check_data(*data=None*)

read(*fname*, *frame=None*)
try to read PNM images :param fname: name of the file :param frame: not relevant here! PNM is always single framed

write(*fname*)
try to write image. For now, limited to :param fname: name of the file

fabio.pnmimage.**pnmimage**
alias of PnmImage

7.29 fabio.raxisimage Module

Authors: Brian R. Pauw email: brian@stack.nl

Written using information gleaned from the ReadRAXISImage program written by T. L. Hendrixson, made available by Rigaku Americas. Available at: http://www.rigaku.com/downloads/software/readimage.html

class fabio.raxisimage.**RaxisImage**(**arg*, ***kwargs*)
Bases: fabio.fabioimage.FabioImage

FabIO image class to read Rigaku RAXIS image files. Write functions are not planned as there are plenty of more suitable file formats available for storing detector data. In particular, the MSB used in Rigaku files is used in an uncommon way: it is used as a *multiply-by* flag rather than a normal image value bit. While it is said to multiply by the value specified in the header, there is at least one case where this is found not to hold, so YMMV and be careful.

read(*fname*, *frame=None*)
try to read image :param fname: name of the file :param frame:

rigakuKeys()

swap_needed()
not sure if this function is needed

fabio.raxisimage.**raxisimage**
alias of RaxisImage

7.30 fabio.tifimage Module

FabIO class for dealing with TIFF images. In facts wraps TiffIO from V. Armando Solé (available in PyMca) or falls back to PIL

7.30.1 Authors:

- Henning O. Sorensen & Erik Knudsen: Center for Fundamental Research: Metal Structures in Four Dimensions; Risoe National Laboratory; Frederiksborgvej 399; DK-4000 Roskilde; email:erik.knudsen@risoe.dk
- Jérôme Kieffer: European Synchrotron Radiation Facility; Grenoble (France)

License: MIT

class fabio.tifimage.**Image_File_Directory**(*instring=None*, *offset=-1*)
Bases: object

unpack(*instring*, *offset=-1*)

class fabio.tifimage.**Image_File_Directory_entry**(*tag=0*, *tag_type=0*, *count=0*, *offset=0*)
Bases: object

extract_data(*full_string*)

unpack(*strInput*)

class fabio.tifimage.**TifImage**(**args*, ***kwds*)
Bases: fabio.fabioimage.FabioImage

Images in TIF format Wraps TiffIO

read(*fname*, *frame=None*)
Wrapper for TiffIO.

write(*fname*)
Overrides the FabioImage.write method and provides a simple TIFF image writer. :param fname: name of the file to save the image to @tag_type fname: string or unicode (file?)...

class fabio.tifimage.**Tiff_header**(*string*)
Bases: object

fabio.tifimage.**tifimage**
alias of TifImage

7.31 fabio.xsdimage Module

Authors: Jérôme Kieffer, ESRF email:jerome.kieffer@esrf.fr

XSDimge are XML files containing numpy arrays

class fabio.xsdimage.**XsdImage**(*data=None*, *header=None*, *fname=None*)
Bases: fabio.fabioimage.FabioImage

Read the XSDataImage XML File data format

read(*fname*, *frame=None*)

fabio.xsdimage.**xsdimage**
alias of XsdImage

7.32 fabio.compression Module

Compression and decompression algorithm for various formats

Authors: Jérôme Kieffer, ESRF email:jerome.kieffer@esrf.fr

class fabio.compression.**ExternalCompressors**
Bases: object

Class to handle lazy discovery of external compression programs

COMMANDS = {'.bz2': ['bzip2-dcf'], '.gz': ['gzip', '-dcf']}

fabio.compression.**compByteOffset**(*data*)
Compress a dataset into a string using the byte_offet algorithm

Parameters **data** – ndarray

Returns string/bytes with compressed data

test = numpy.array([0,1,2,127,0,1,2,128,0,1,2,32767,0,1,2,32768,0,1,2,2147483647,0,1,2,2147483648,0,1,2,128,129,130,327

fabio.compression.**compByteOffset_cython**(*data*)
Compress a dataset into a string using the byte_offet algorithm

Parameters **data** – ndarray

Returns string/bytes with compressed data

test = numpy.array([0,1,2,127,0,1,2,128,0,1,2,32767,0,1,2,32768,0,1,2,2147483647,0,1,2,2147483648,0,1,2,128,129,130,327

fabio.compression.**compByteOffset_numpy**(*data*)
Compress a dataset into a string using the byte_offet algorithm

Parameters **data** – ndarray

Returns string/bytes with compressed data

test = numpy.array([0,1,2,127,0,1,2,128,0,1,2,32767,0,1,2,32768,0,1,2,2147483647,0,1,2,2147483648,0,1,2,128,129,130,327

fabio.compression.**compPCK**(*data*)
Modified CCP4 pck compressor used in MAR345 images

Parameters **data** – numpy.ndarray (square array)

Returns compressed stream

fabio.compression.**compTY1**(*data*)
Modified byte offset compressor used in Oxford Diffraction images

Parameters **data** – numpy.ndarray with the input data (integers!)

Returns 3-tuple of strings: raw_8,raw_16,raw_32 containing raw data with integer of the given size

fabio.compression.**decByteOffset**(*stream*, *size=None*, *dtype='int64'*)

Analyze a stream of char with any length of exception: 2, 4, or 8 bytes integers

Parameters

- **stream** – string representing the compressed data
- **size** – the size of the output array (of longInts)

Returns 1D-ndarray

fabio.compression.**decByteOffset_cython**(*stream*, *size=None*, *dtype='int64'*)

Analyze a stream of char with any length of exception: 2, 4, or 8 bytes integers

Parameters

- **stream** – string representing the compressed data
- **size** – the size of the output array (of longInts)

Returns 1D-ndarray

fabio.compression.**decByteOffset_numpy**(*stream*, *size=None*, *dtype='int64'*)

Analyze a stream of char with any length of exception: 2, 4, or 8 bytes integers

Parameters

- **stream** – string representing the compressed data
- **size** – the size of the output array (of longInts)

Returns 1D-ndarray

fabio.compression.**decBzip2**(*stream*)
Decompress a chunk of data using the bzip2 algorithm from Python

fabio.compression.**decGzip**(*stream*)
Decompress a chunk of data using the gzip algorithm from system or from Python

Parameters **stream** – compressed data

Returns uncompressed stream

fabio.compression.**decKM4CCD**(*raw_8*, *raw_16=None*, *raw_32=None*)
Modified byte offset decompressor used in Oxford Diffraction images

Note: Always expect little endian data on the disk

Parameters

- **raw_8** – strings containing raw data with integer 8 bits
- **raw_16** – strings containing raw data with integer 16 bits
- **raw_32** – strings containing raw data with integer 32 bits

Returns numpy.ndarray

fabio.compression.**decPCK**(*stream*, *dim1=None*, *dim2=None*, *overflowPix=None*, *version=None*, *normal_start=None*, *swap_needed=None*)
Modified CCP4 pck decompressor used in MAR345 images

Parameters

- **raw** – input string (bytes in python3)

- **dim1,dim2** – optional parameters size
- **overflowPix** – optional parameters: number of overflowed pixels
- **version** – PCK version 1 or 2
- **normal_start** – position of the normal value section (can be auto-guessed)
- **swap_needed** – set to True when reading data from a foreign endianness (little on big or big on little)

:return : ndarray of 2D with the right size

`fabio.compression.`**`decTY1`**(*raw_8*, *raw_16=None*, *raw_32=None*)
Modified byte offset decompressor used in Oxford Diffraction images

Note: Always expect little endian data on the disk

Parameters

- **raw_8** – strings containing raw data with integer 8 bits
- **raw_16** – strings containing raw data with integer 16 bits
- **raw_32** – strings containing raw data with integer 32 bits

Returns numpy.ndarray

`fabio.compression.`**`decZlib`**(*stream*)
Decompress a chunk of data using the zlib algorithm from Python

`fabio.compression.`**`endianness`**()
Return the native endianness of the system

`fabio.compression.`**`md5sum`**(*blob*)
returns the md5sum of an object...

7.33 `fabio.converters` Module

Converter module. This is for the moment empty (populated only with almost pass through anonymous functions) but aims to be populated with more sofisticated translators ...

`fabio.converters.`**`convert_data`**(*inp*, *outp*, *data*)
Return data converted to the output format ... over-simplistic implementation for the moment ... :param inp,outp: input/output format like “cbfimage” :param data(ndarray): the actual dataset to be transformed

`fabio.converters.`**`convert_data_integer`**(*data*)
convert data to integer

`fabio.converters.`**`convert_header`**(*inp*, *outp*, *header*)
return header converted to the output format :param inp,outp: input/output format like “cbfimage” :param header(dict):the actual set of headers to be transformed

7.34 `fabio.datIO` Module

Authors: Henning O. Sorensen & Erik Knudsen Center for Fundamental Research: Metal Structures in Four Dimensions Risoe National Laboratory Frederiksborgvej 399 DK-4000 Roskilde email:erik.knudsen@risoe.dk

and Jon Wright, ESRF

class `fabio.datIO.`**`columnfile`**(*data=None*, *clabels=None*, *rlabels=None*, *fname=None*)
Bases: `fabio.datIO.fabiodata`

Concrete fabiodata class

read(*fname*, *frame=None*)

class fabio.datIO.**fabiodata**(*data=None*, *clabels=None*, *rlabels=None*, *fname=None*)
Bases: object

A common class for dataIO in fable Contains a 2d numpy array for keeping data, and two lists (clabels and rlabels) containing labels for columns and rows respectively

read(*fname=None*, *frame=None*)
To be overridden by format specific subclasses

7.35 fabio.TiffIO Module

class fabio.TiffIO.**TiffIO**(*filename*, *mode=None*, *cache_length=20*, *mono_output=False*)
Bases: object

close()

getData(*nImage*, ***kw*)

getImage(*nImage*)

getImageFileDirectories(*fd=None*)

getInfo(*nImage*, ***kw*)

getNumberOfImages()

writeImage(*image0*, *info=None*, *software=None*, *date=None*)

7.36 fabio.readbytestream Module

Reads a bytestream

Authors: Jon Wright Henning O. Sorensen & Erik Knudsen ESRF Risoe National Laboratory

fabio.readbytestream.**readbytestream**(*fil*, *offset*, *x*, *y*, *nbytespp*, *datatype='int'*, *signed='n'*, *swap='n'*, *typeout=<type 'numpy.uint16'>*)
Reads in a bytestream from a file (which may be a string indicating a filename, or an already opened file (should be "rb")) offset is the position (in bytes) where the pixel data start nbytespp = number of bytes per pixel type can be int or float (4 bytes pp) or double (8 bytes pp) signed: normally signed data 'y', but 'n' to try to get back the right numbers when unsigned data are converted to signed (python once had no unsigned numeric types.) swap, normally do not bother, but 'y' to swap bytes typeout is the numpy type to output, normally uint16, but more if overflows occurred x and y are the pixel dimensions

TODO : Read in regions of interest

PLEASE LEAVE THE STRANGE INTERFACE ALONE - IT IS USEFUL FOR THE BRUKER FORMAT

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

f

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X