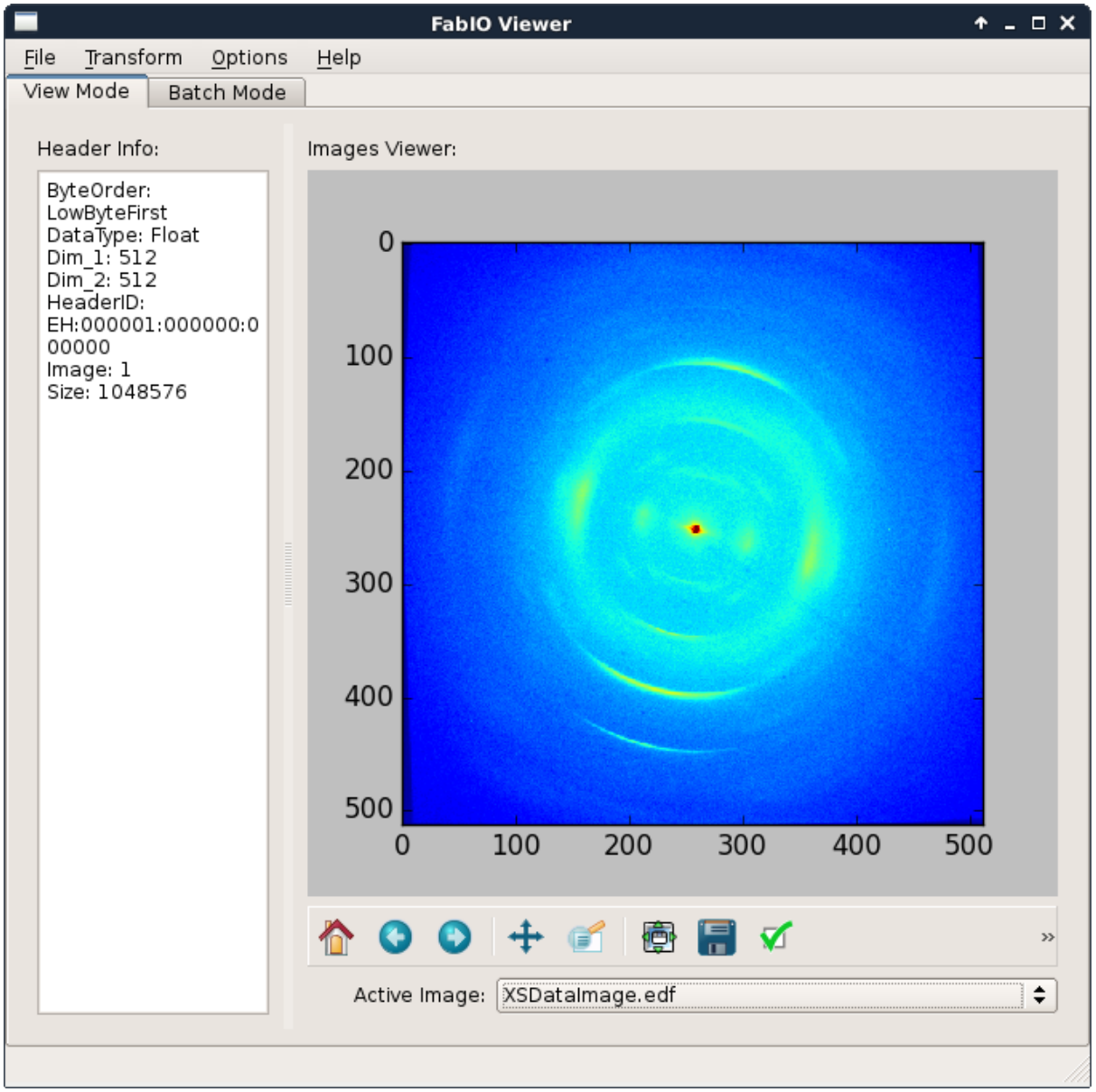


FabIO Documentation

Release 0.2.2

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

July 21, 2015

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.

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.

Table 1.1: Supported formats

Python Module	Detector / Format	Extension	Read	Multi-image	Write
ADSC	ADSC Quantum	.img	Yes	No	Yes
Bruker	Bruker formats	.sfrm	Yes	No	Yes
DM3	Gatan Digital Micrograph	.dm3	Yes	No	No
EDF	ESRF data format	.edf	Yes	Yes	Yes
EDNA-XML	Used by EDNA	.xml	Yes	No	No
CBF	CIF binary files	.cbf	Yes	No	Yes
kcd	Nonius KappaCCD	.kccd	Yes	No	No
fit2d mask	Used by Fit2D	.msk	Yes	No	Yes
fit2d spreadsheet	Used by Fit2D	.spr	Yes	No	Yes
GE	General Electric	No	Yes	Yes	No
HiPiC	Hamamatsu CCD	.tif	Yes	No	No
HDF5	Hierachical data dormat	.h5	Yes	No	No
marccd	MarCCD/Mar165	.mccd	Yes	No	No
mar345	Mar345 image plate	.mar3450	Yes	No	Yes
OXD	Oxford Diffraction	.img	Yes	No	Yes
Pixi	pixi	.	Yes	No	No
pilatus	Dectris Pilatus Tiff	.tif	Yes	No	Yes
PNM	Portable aNy Map	.pnm	Yes	No	No
Raxis	Rigaku Saxs format	.img	Yes	No	No
TIFF	Tagged Image File Format	.tif	Yes	No	Yes

1.5.4 Adding new file formats

We hope it will be relatively easy to add new file formats to fabio in the future. The basic idea is the following:

1. inherit from fabioimage overriding the methods _readheader, read and optionally write. Name your new module XXXimage where XXX means something (eg tifimage).
2. readheader fills in a dictionary of “name”:”value” pairs in self.header. No one expects to find anything much in there.
3. read fills in self.data with a numpy array holding the image. Some redundant info which also appears are self.dim1 and self.dim2: the image dimensions, self.bpp is the bytes per pixel and self.bytecode is the numpy.dtype.type of the data.
4. The member variables “_need_a_seek_to_read” and “_need_a_real_file” are there in case you have trouble with the transparent handling of bz2 and gz files.
5. Register the file type (extension naming) in fabioutils.py:FILETYPES
6. Add your new module as an import into fabio.openimage
7. Fill out the magic numbers for your format in fabio.openimage if you know them (the characteristic first few bytes in the file)
8. Upload a testimage to the file release system and create a unittest testcase which opens an example of your new format, confirming the image has actually been read in successfully (eg check the mean, max, min and esd are all correct, perhaps orientation too)
9. Run pylint on your code and then please go clean it up. Have a go at mine while you are at it.
10. Bask in the warm glow of appreciation when someone unexpectedly learns they don’t need to convert their data into another format

CHAPTER

TWO

INSTALLATION

FabIO can, as any Python module, be installed from its sources, available on sourceforge but we advice to use binary packages provided for the most common platforms on sourceforge: Windows, MacOSX and Linux. Moreover FabIO is part of the common Linux distributions Ubuntu (since 11.10) and Debian7 where the 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 have been packaged and should be PIP-installable. PIP is the Python Installer Program, similar to `apt-get` for Python. It runs under any architecture and can simply be installed from:

https://bootstrap.pypa.io/get-pip.py

then

```
pip install fabio
```

2.1 Installation under windows

Install Python from http://python.org. I would recommend Python 2.7 in 64 bits version if your operating system allows it. Python3 (>=3.2) is OK while less tested.

If you are looking for an integrated distribution of Python on Windows, WinPython is a good one, the Python2.7, 64 bit version is advised. https://winpython.github.io/ It comes with pip pre-installed and configured.

2.1.1 Installation using PIP:

Download PIP and run: https://bootstrap.pypa.io/get-pip.py

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 Christopher Gohlke' page (including FabIO):

http://www.lfd.uci.edu/~gohlke/pythonlibs/

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

This gives you the architecture width of the Python interpreter

2.1.3 Installation from sources

Install the required dependencies (via PIP or a repository), then retrieve the Microsoft compiler and install it from: http://aka.ms/vcpython27

Once done, follow the classical procedure (similar to MacOSX or Linux): * download sources of FabIO from fable.sourceforge.net. * unzip the archive * run `python setup.py build install`

2.2 Installation on MacOSX

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

2.2.1 Install PIP

Download PIP and run: https://bootstrap.pypa.io/get-pip.py

Then install the wheel package manager:

```
pip install setuptools
pip install wheel
pip install PIL
pip install lxml
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

2.2.2 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.

2.3 Manual Installation for any operating system

2.3.1 Install the dependencies

- Python 2.6 - 2.7 or 3.2+
- numpy - http://www.numpy.org

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

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

FabIO can be downloaded from the fable download page on sourceforge.net. Presently the source code has been distributed as a zip package and a compressed tarball. Download either one and unpack it.

```
http://sourceforge.net/projects/fable/files/fabio/
```

e.g.

```
tar xvzf fabio-0.2.2.tar.gz
```

or

```
unzip fabio-0.2.2.zip
```

all files are unpacked into the directory fabio-0.2.2. To install these do

```
cd fabio-0.2.2
```

and install fabio: build it, run the tests and build the wheel package and install it.

```
python setup.py build
python setup.py bdist_wheel
sudo pip install dist/fabio-0.2.2*.whl
```

most likely you will need to gain root privileges (with sudo in front of the command) to install the built package.

2.4 Development versions

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

```
git clone https://github.com/kif/fabio
cd fabio
python setup.py build bdist_wheel
sudo pip install dist/fabio-0.2.2*.whl
```

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
```

We provide also a debian-package builder based on stdeb:

```
sudo apt-get install python-stdeb
./build-deb.sh 3
```

which builds a couple of debian packages (actually one for python2 and another for python3) and installs them in a single command. Handy for testing, but very clean, see hereafter

2.5 Debian packaging

FabIO features some helper function to make debian packaging easier:

```
#to create the orig.tar.gz without cython generated C files for Sphinx built documentation:
python setup.py debian_src

# to create a tarball of all images needed to test the library
python setup.py debian_testimages
```

Two tarball are created, one with all source code (and only source code) and the other one with all test-data.

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 environment variable http_proxy is correctly set-up. For example if you are behind a firewall/proxy:

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

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 25 test-suites (113 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 2.6, 2.7, 3.2 and 3.4. Under linux it has been tested on i386, x86_64, arm, ppc, ppc64le.

Table 2.1: Test suite coverage

Name	Stmts	Exec	Cover
fabio/GEimage	94	48	51%
fabio/HiPiCimage	55	7	12%
fabio/OXDimage	285	271	95%
fabio/TiffIO	794	534	67%
fabio/__init__	15	15	100%
fabio/adscimage	79	37	46%

Continued on next page

Table 2.1 – continued from previous page

Name	Stmts	Exec	Cover
fabio/binaryimage	50	15	30%
fabio/bruker100image	60	13	21%
fabio/brukerimage	212	171	80%
fabio/cbfimage	441	219	49%
fabio/compression	223	136	60%
fabio/converters	17	14	82%
fabio/dm3image	133	16	12%
fabio/edfimage	596	397	66%
fabio/fabioimage	306	193	63%
fabio/fabioutils	322	256	79%
fabio/file_series	140	61	43%
fabio/fit2dmaskimage	75	71	94%
fabio/fit2dspreadsheetimage	47	7	14%
fabio/hdf5image	146	25	17%
fabio/kcdimage	80	65	81%
fabio/mar345image	244	215	88%
fabio/marccdimage	63	56	88%
fabio/mrcimage	96	0	0%

Continued on next page

Table 2.1 – continued from previous page

Name	Stmts	Exec	Cover
fabio/openimage	104	69	66%
fabio/pilatusimage	34	5	14%
fabio/pixiimage	95	22	23%
fabio/pnmimage	109	21	19%
fabio/raxisimage	98	88	89%
fabio/readbytestream	26	5	19%
fabio/tifimage	167	60	35%
fabio/xsdimage	94	68	72%

CHAPTER

THREE

BENCHMARKS

Those benchmarks have been done with all data already in cache using a Intel Xeon E5520 @ 2.27GHz running Debian 7.

Table 3.1: 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.

CHAPTER

FOUR

CHANGELOG

4.1 From FabIO-0.2.1 to FabIO-0.2.2:

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

4.2 From FabIO-0.2.0 to FabIO-0.2.1:

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

4.3 From FabIO-0.1.4 to FabIO-0.2.0:

- 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)

4.4 From FabIO-0.1.3 to 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

4.5 From FabIO-0.1.2 to 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

4.6 From FabIO-0.1.1 to FabIO-0.1.2:

- 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"

4.7 From FabIO-0.1.0 to 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

4.8 From FabIO-0.0.8 to 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

Python3 is not yet tested but some blocking points have been identified and some fixed.

4.9 From FabIO-0.0.7 to FabIO-0.0.8:

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

4.10 From FabIO-0.0.6 to FabIO-0.0.7:

- 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)

4.11 From FabIO-0.0.4 to FabIO-0.0.6:

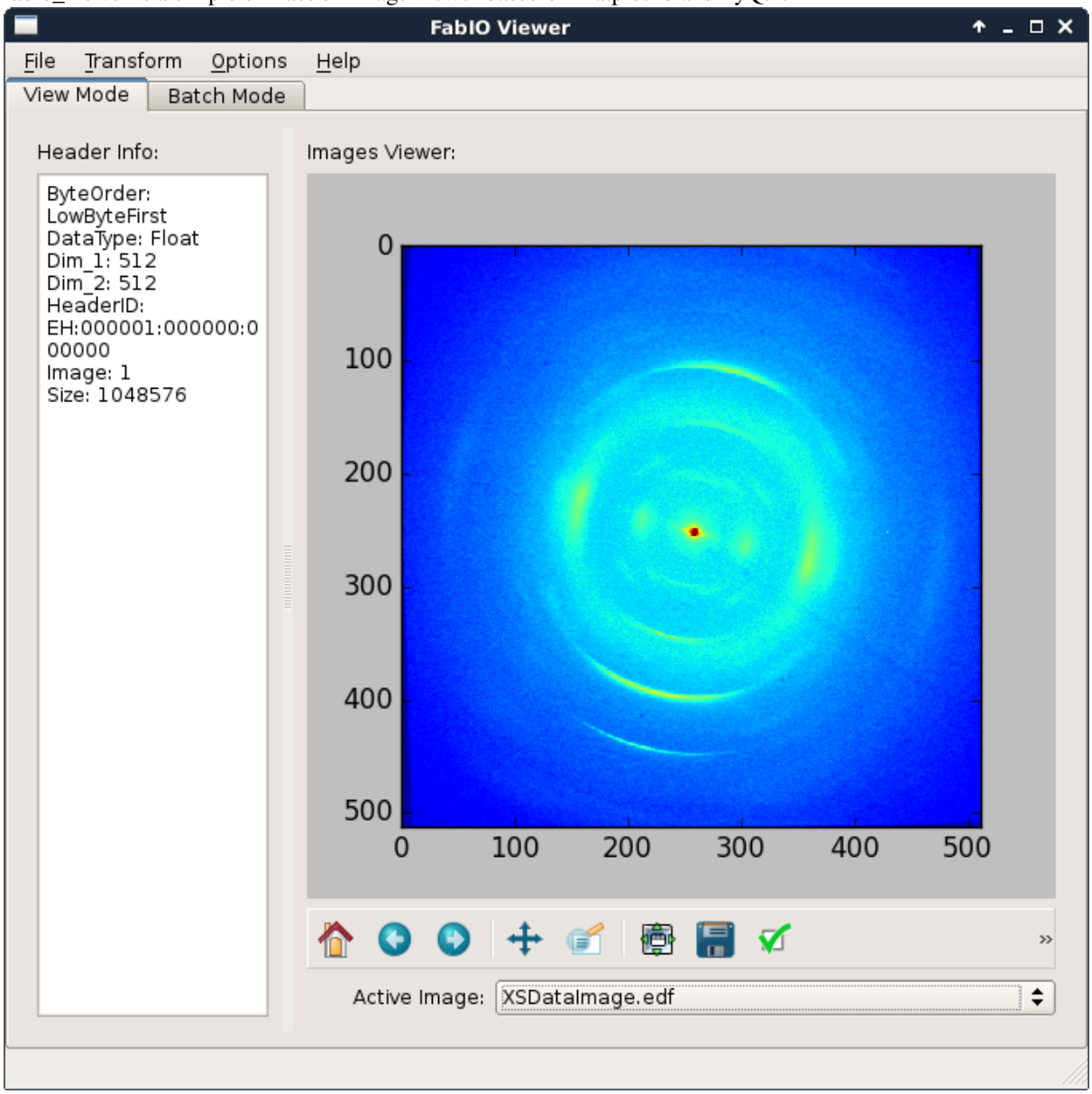
- Support for CBF files from Pilatus detectors
- Support for KCD files from Nonius Kappa CCD images
- write EDF with their native data type (instead of uint16 by default)

CHAPTER

FIVE

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 0.2.2
```

CHAPTER

SIX

FABIO PACKAGE

6.1 fabio Package

6.2 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

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

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

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

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

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

resetvals()
Reset cache - call on changing data

save(*fname*)
wrapper for write

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

fabio.fabioimage.**test**()
check some basic fabioimage functionality

6.3 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 0x7fc9de832ad0>

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

wrapper for "file" with locking

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()

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 modifaction of the code

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

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

fabio.fabioutils.**to_str**(*s*)

6.4 fabio.file_series Module

6.5 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*)
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

6.6 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.**test**()
testcase

6.7 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*)

6.8 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

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

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

toPIL16(*filename=None*)

write(*fname*)

6.9 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

write (*fname*)
Write a bruker image

fabio.brukerimage.**test**()
a testcase

6.10 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

static checkData(*data=None*)

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

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

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

6.11 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*)

6.12 fabio.edfimage Module

License: GPLv2+

6.12.1 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.**Frame**(*data=None*, *header=None*, *header_keys=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

class fabio.edfimage.**edfimage**(*data=None*, *header=None*, *header_keys=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 checkHeader(*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

delHeaderKeys()
Deleter for edf header_keys

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,

getHeaderKeys()
Getter for edf header_keys

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

header_keys
property: header_keys 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

setHeaderKeys(*_listtHeader*)
Enforces the propagation of the header_keys to the list of frames :param _listtHeader: list of the (ordered) keys in the header :type _listtHeader: python list

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

6.13 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 checkData(*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

6.14 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

6.15 fabio.GEimage Module

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**()

6.16 fabio.hdf5mage Module

HDF5 image for FabIO

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

Specifications: input should being the form:

hdf5:///filename?path#slice=[:,:,1]

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

FabIO image class for Images from an HDF file

get_slice()

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 as hdf5:///filename?path#slice=[:,:,1]

set_url(*url*)
set the url of the data

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

6.17 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

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

6.18 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

6.19 fabio.mar345image Module

6.19.1 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

6.20 fabio.mrcimage Module

6.21 fabio.marccdimage Module

6.21.1 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: 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?)

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

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

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

6.22 fabio.OXDimage Module

Reads Oxford Diffraction Sapphire 3 images

6.22.1 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)

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*)

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!)

6.23 fabio.pilatusimage Module

6.23.1 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)

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

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

6.24 fabio.pixiimage Module

Author: Jon Wright, ESRF.

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

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'>*)

6.25 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 checkData(*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(*filename*)

6.26 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

6.27 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

6.27.1 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: GPLv3+

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.**Tiff_header**(*string*)

Bases: object

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?)...

6.28 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*)

6.29 fabio.compression Module

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

FabIO library containing compression and decompression algorithm for various

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_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*)

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*)

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*)

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 Python or alternatives if possible

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...

6.30 `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

6.31 `fabio.datIO` Module

6.32 `fabio.third_party.TiffIO` Module

6.33 `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

SEVEN

INDICES AND TABLES

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

f

A

B

C

D

E

F

G

H

I

S

T

U

W

X