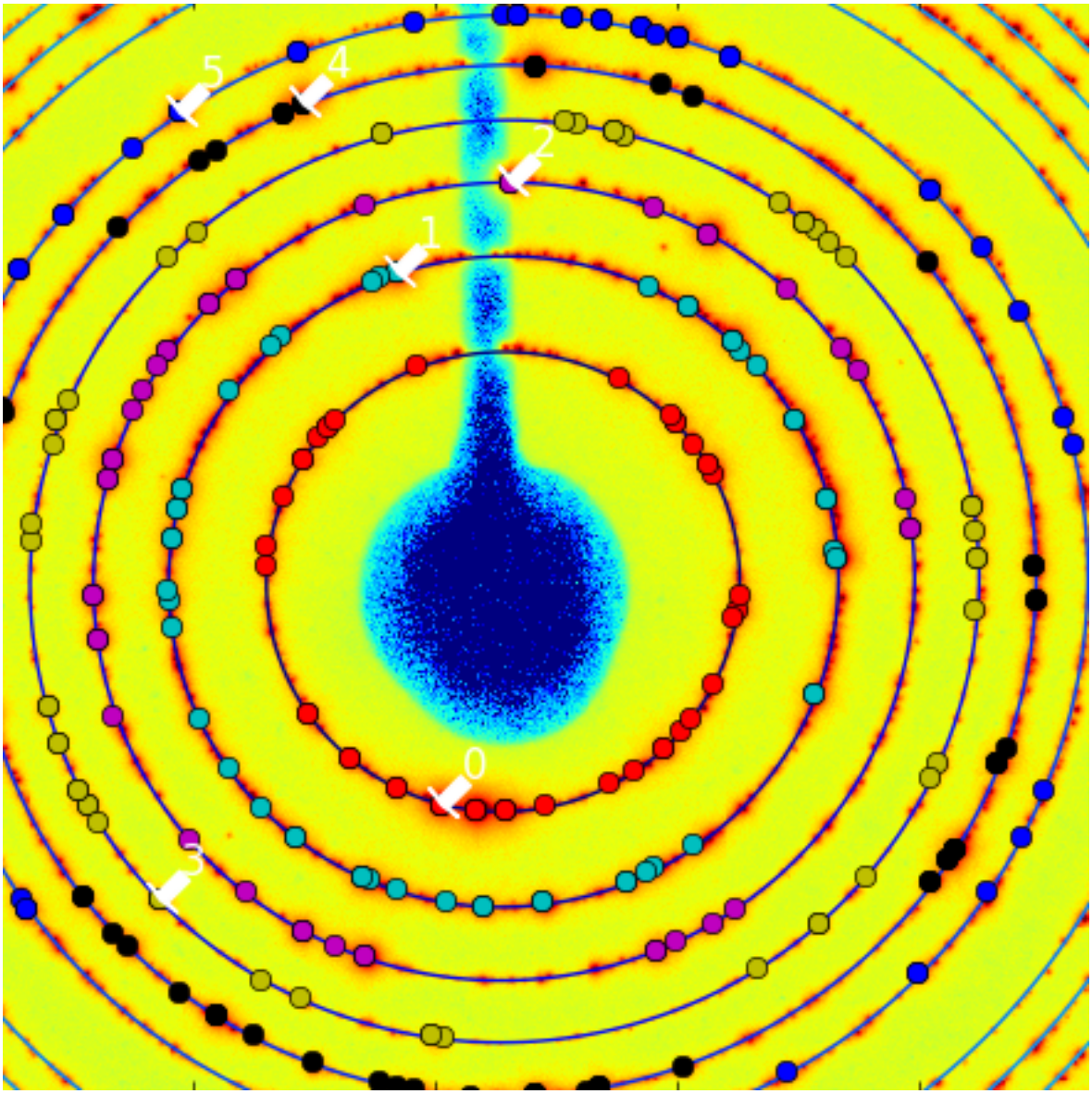


pyFAI Documentation

Release 0.10.3

Jérôme Kieffer

March 20, 2015

CONTENTS

PyFAI A Python libary for high performance azimuthal integration which can use on GPU, which was presented at EuroScipy 2014: the video is online as well as the proceedings.

This document starts with a general descriptions of the pyFAI library in the first chapter. This first chapter contains an introduction to pyFAI, what it is, what it aims at and how it works (for scientists). Especially, geometry, calibration, azimuthal integration algorithms are described and pixel splitting schemes are explained.

Follows cookbook, tutorials on how to use pyFAI scripts, then the manual pages of all scripts. Those are programs to be launched at the command line allowing the treatment of a diffraction experiment without knowing anything about Python.

The design of the programming interface is then exposed before a comprehensive description of most modules contained in pyFAI. Some minor submodules as well as the documentation of the Cython sub-modules are not included for concision purposes. The last chapter is an appendix giving some figures about the project and its management.

Installation procedures for Windows, MacOSX and Linux operating systems are then described. Finally other programs/projects relying on pyFAI are presented and the project is summarized from a developer's point of view.

CHAPTER

ONE

GENERAL INTRODUCTION TO PYFAI

1.1 Python Fast Azimuthal Integration

PyFAI is implemented in Python programming language, which is open source and already very popular for scientific data analysis ([PyMca], [PyNX], ...). It relies on the scientific stack of python composed of [NumPy], [SciPy] and [Matplotlib] plus the [OpenCL] binding [PyOpenCL] for performances.

$2D$ area detectors like CCD or pixel detectors have become popular in the last 15 years for diffraction experiments (e.g. for WAXS, SAXS, single crystal and powder diffraction). These detectors have a large sensitive area of millions of pixels with high spatial resolution. The software package pyFAI ([SRI2012], [EPDIC13]) has been designed to reduce SAXS, WAXS and XRPD images taken with those detectors into $1D$ curves (azimuthal integration) usable by other software for in-depth analysis such as Rietveld refinement, or $2D$ images (a radial transformation named *caking* in [FIT2D]). As a library, the aim of pyFAI is to be integrated into other tools like [PyMca] or [EDNA] or [LImA] with a clean pythonic interface. However pyFAI features also command line and graphical tools for batch processing, converting data into *q-space* (q being the momentum transfer) or 2θ-space (θ being the Bragg angle) and a calibration graphical interface for optimizing the geometry of the experiment using the Debye-Scherrer rings of a reference sample. PyFAI shares the geometry definition of SPD but can directly import geometries determined by the software FIT2D. PyFAI has been designed to work with any kind of detector and geometry (transmission or reflection) and relies on FabIO, a library able to read more than 20 image formats produced by detectors from 12 different manufacturers. During the transformation from cartesian space (x, y) to polar space $(2\theta, \chi)$, both local and total intensities are conserved in order to obtain accurate quantitative results. Technical details on how this integration is implemented and how it has been ported to native code and parallelized on graphic cards are discussed in this paper.

1.2 Introduction

With the advent of hyperspectral experiments like diffraction tomography in the world of synchrotron radiation, existing software tools for azimuthal integration like [FIT2D] and [SPD] reached their performance limits owing to the fast data rate needed by such experiments. Even when integrated into massively parallel frameworks like [EDNA] , such stand-alone programs, due to their monolithic nature, cannot keep the pace with the data flow of new detectors. Therefore we decided to implemente from scratch a novel azimuthal integration tool which is designed to take advantage of modern parallel hardware features. PyFAI assumes the setup does not change during the experiment and tries to reuse a maximum number of data (using memoization), moreover those calculation are performed only when needed (lazy_evaluation).

1.3 Experiment description

In pyFAI, the basic experiment is defined by a description of an area-detector whose position in space is defined through the sample position and the incident X-ray beam, and can be calibrated using Debye-Scherrer rings of a reference compound.

1.3.1 Detector

Simple detector

Like most other diffraction processing packages, pyFAI allows the definition of 2D detectors with a constant pixel size and recoded in S.I.. Typical pixel size are 50e-6 m and will be used as example in the numerical application.

Pixels of the detector are indexed from the *origin* located at the *lower left corner*. The pixel center is located at half integer index: * pixel 0 goes from position 0 to 50e-6 and is centered at 25e-6. * pixel 1 goes from position 50e-6 to 100e-6 and is centered at 75e-6m

Complex detectors

The *simple detector* approach reaches its limits with several detector types, such as multi-module and fiber optic taper coupled detectors. Large area pixel detectors are often composed of smaller modules (i.e. Pilatus from Dectris, Maxipix from ESRF,...).

By construction, such detectors exhibit gaps between modules along with pixels of various sizes within a single module, hence they require specific data masks. Optically coupled detectors need also to be corrected for small spatial displacements, often called geometric distortion. This is why detectors need more complex definitions than just that of a pixel size. To avoid complicated and error-prone sets of parameters, two tools have been introduced: either *detector* classes define programatically detector or Nexus saved detector setup.

Detectors classes

They are used to define families of detectors. In order to take the specificities of each detector into account, pyFAI contains about 40 detector class definitions (and twice a much with aliases) which contain a mask (invalid pixels, gaps, ...) and a method to calculate the pixel positions in Cartesian coordinates. Available detectors can be printed using:

```
import pyFAI
print(pyFAI.detectors.ALL_DETECTORS)
```

For optically coupled CCD detectors, the geometrical distortion is often described by a bi-dimensional cubic spline which can be imported into the detector instance and be used to calculate the actual pixel position in space.

Nexus Detectors

Any detector object in pyFAI, can be saved into a HDF5 file following the NeXus convention (http://nexusformat.org). Detector objects can subsequently be restored from the disk, making complex detector definitions less error-prone. Pixels of an area detector are saved as a 4D dataset: i.e. a 2D array of vertices pointing to every corner of each pixel, generating an array of shape: (*Ny*, *Nx*, *Nc*, 3) where *Nx* and *Ny* are the dimensions of the detector, *Nc* is the number of corners of each pixel, usually 4, and the last entry contains the coordinates of the vertex itself (z,y,x). This kind of definitions, while relying on large description files, can address some of the most complex detector layouts:

- hexagonal pixels (i.e. Pixirad detectors)
- curved/bent imaging plates (i.e. Rigaku)
- pixel detectors with tiled modular (i.e. Xpad detectors from ImXpad)
- semi-cylindrical pixel detectors (i.e. Pilatus12M from Dectris).

The detector instance can be saved as HDF5, either programmatically, either on the command line.

```
from pyFAI import detectors
frelon = detectors.FReLoN("halfccd.spline")
print(frelon)
frelon.save("halfccd.h5")
```

Using the *detector2nexus* script to convert a complex detector definition (multiple modules, possibly in 3D) into a single NeXus detector definition together with the mask:

detector2nexus -s halfccd.spline -o halfccd.h5

1.3.2 Geometry

PyFAI uses a 6-parameter geometry definition similar, while not rigorously identical to SPD: One distance, 2 coordinates to define the point of normal incidence and 3 rotations around the main axis; these parameters are saved in text files usually with the *.poni* extension.

Image representation in Python

PyFAI takes diffraction images as 2D numpy arrays, those are usually read using the FabIO library:

```
import fabio
data = fabio.open("image.edf").data
```

But data can also be extracted from HDF5 files with h5py and displayed using matplotlib:

```
%pylab
imshow(data, origin="lower")
```

Because Python is written in C language, data are stored lines by lines, this means to go from a pixel to the on its right, one offets the position by the pixel width. To go the pixel above the current one, on needs to offset by the length of the line. This is why, if one considers the pixel at position (x,y), its value can be retrieved by data[y,x] (note the order y,x, this is not a bug!). We usually refer the *x* axis as the fast dimension (because pixels are adjacent) and the *y* axis as the slow axis (as pixel are appart from each other by a line length). More information on how numpy array are stored can be found at: https://github.com/numpy/numpy/blob/master/doc/source/reference/arrays.ndarray.rst

Like most scientific packages, the origin of the image is considered to be at the lower-left corner of the image to have the polar angle growing from 0 along the x axis to 90 deg along the y axis. This is why we pass the *origin="lower"* option to imshow. Axis 1 and 2 on the image (like in poni1 & poni2) refer to the slow and fast dimension of the image, so usually to the y and x axis (and not the opposite)

Position of the observer

There are two (main) conventions when representing images:

- In imaging application, one can can replace the camera by the eye, the camera looks at the scene. In this convention, the origin is usually at the top of the image.
- In diffraction application, the observer is situated at the sample position and looks

at the detector, hence on the other side of the detector. Because we measure (signed) angles, the origin is ideally situated at the lower left of the image.

PyFAI being a diffraction application, it uses the later description.

Default geometry in pyFAI

In the (most common) case of *transmission diffraction setup* on synchrotrons (like ESRF, Soleil, Petra3, SLS...) this makes looks like:

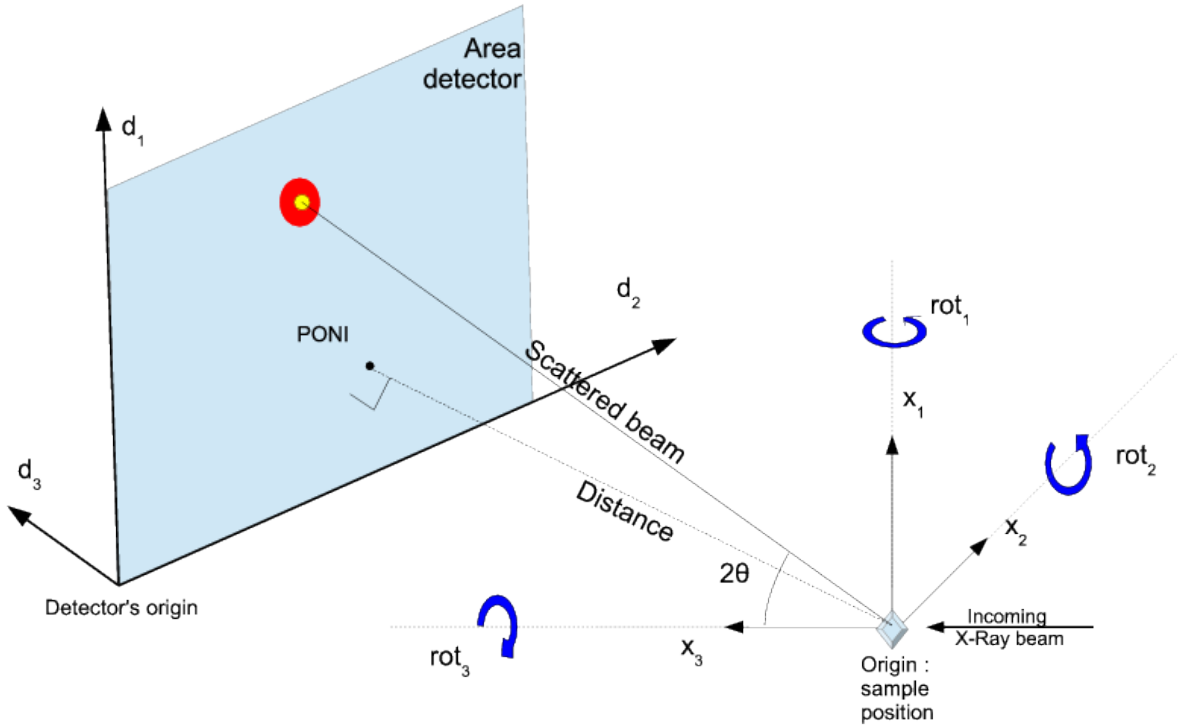
- Observer looking at the detector from the sample position:
- Origin at the lower left of the detector
- Axis 1 (i.e. *y*) being vertical, pointing upwards
- Axis 2 (i.e. *x*) being horizontal, pointing to the center of the storage ring

- Axis 3 (i.e. *z*) being horizontal, along the transmitted beam

Axis 3 is built in such a way to be orthogonal and (1,2,3) is a direct orientation. This makes the sample position at z<0.

Detector position

In pyFAI, the experiment geometry is defined by the position of the detector in space, the origin being located at the sample position, more precisely where the X-ray beam crosses the diffractometer main axis.



With the detector being a rigid body, its position in space is described by six parameters: 3 translations and 3 rotations. In pyFAI, the beam center is not directly used as it is ill-defined with highly tilted detectors. Like SPD, we use the orthogonal projection of origin on the detector surface called PONI (for Point Of Normal Incidence). For non planar detectors, the PONI is defined in the plan z=0 in detector's coordinate system.

Poni1 and Poni2 are distances in meter (along the *y* and *x* axis of the detector), like the sample-detector distance, letting the calibration parameters be independent of the pixel size hence stable regarding the binning factor of the detector.

In the same idea rot1, rot2 and rot3 are rotation along axis 1, 2 and 3, always expressed in radians. Rotations applied in the same order: *rot1* then *rot2* and finally *rot3*. Due to the axial symmetry of the Debye-Scherrer cones, *rot3* cannot be optimized but can be adjusted manually in some cases like if the detector is not mounted horizontally and/or one cares about polarization correction.

When all rotations are zero, the detector is in transmission mode with the incident beam orthogonal to the detector's surface.

1.3.3 calibration

The determination of the geometry of the experimental setup for the diffraction pattern of a reference sample is called calibration in pyFAI. A geometry setup is composed of a detector, the six refined parameters like the distance and fixed parameters like the wavelength (or the energy of the beam), they are all saved together into a text files named ".poni" (as a reference to the point of normal incidence) which is subsequently used for processing the experiment.

The program *pyFAI-calib* helps calibrating

the experimental setup using a constrained least squares optimization on the Debye-Scherrer rings of a reference sample (LaB_6, silver behenate, ...) and saves the results into a *.poni* file. Alternatively, geometries calibrated using fit2d can be imported into pyFAI, including geometric distortions (i.e. optical-fiber tapers distortion) described as *spline-files*.

By storing all parameters together in a single small file, the risk of mixing two parameters is highly reduced and we believe this helps performing better science with fewer mistakes.

While entering the geometry of the experiment in a poni-file is possible it is easier to perform a calibration, using the Debye-Sherrer rings of a reference sample called calibrant. About 10 calibrant description files are shipped with the default installation of pyFAI, like LaB6, silicon, ceria, corrundum or silver behenate. The user can choose to provide their own calibrant description files which are simple text-file containing the largest d-spacing (in Angstrom) for a set of Miller plans. A useful reference is the American Mineralogist database [AMD] or the Crystallographic Open database [COD].

The calibration is divided into 4 major steps:

Pre-processing of images:

The typical pre-processing consists of the averaging (or median filter) of darks images. Dark current images are then subtracted from data and corrected for flat.

If saturated pixels exists, the are likely to be treated like peaks but their positions will be wrong. It is advised to either mask them out or to desaturate them (pyFAI provides an option, but it is expensive in calculation time)

Peak-picking

The Peak-picking consists in the identification of peaks and groups of peaks belonging to same ring. It can be performed by two methods : blob detection or massif detection.

Massif detection

This method consists in making the difference of the original image and a blurred image. Then we look for a chain of positives values, corresponding to a single group of peak. The blurring parameter can be adjusted using the "-g" option in pyFAI-calib.

Blob detection

The approach is based on difference of gaussians (DoGs) as described in the blob_detection article of wikipedia.

It consists in blurring the image by convolution with a 2D gaussian kernel and making differences between two successive blurs (called Difference Of Gaussian). In theses DoGs, keypoints are defined as the maxima in the 3D space (y,x,size of the gaussian). After their localization, keypoints are refined by Savitzky Golay algorithm or by an interpolation at the second order which is equivalent but uses less points. At this step, if the estimation of the maximum is too far from the maximum, the keypoint will be considered as a fake maximum and removed.

Steepest ascent

This is very naive implementation which looks for the nearest local maximum. Subsequently a sub-pixel optimization is performed based on the local gradient and hessian.

Monte-Carlo sampling

Series of peaks can be extracted using the Steepest Ascent on randomly selected seeds.

Refinement of the parameters

After grouping of peaks, groups of peak are assigned to a Debye-Scherrer ring and to a d-spacing. PyFAI uses a least-squares refinement of the geometry parameters on peak position.

The optimization procedure is the Sequential Least SQuares Programming implemented in scipy.optimize.slsqp. The cost function is the sum of the square of the difference between the expected and calculated 2theta values for the various peaks. This sum is dependent on the number of control-points.

Validation of the calibration

Validation by an human being of the geometry is an essential step: pyFAI will overlay to the diffraction image, the lines corresponding to the various diffraction rings expected from the calibrant. Those lines should be in pretty good agreement with the rings of the scattering image.

Once the calibration is finished, one can use the validate option to check the offset between the input image and the generated on from the diffraction pattern.

1.3.4 PyFAI executables

PyFAI was designed to be used by scientists needing a simple and effective tool for azimuthal integration. Two command line programs *pyFAI-waxs* and *pyFAI-saxs* are provided with pyFAI for performing the integration of one or more images on the command line. The waxs version outputs result in $2\theta/I$, whereas the saxs version outputs result in $q/I(/\sigma)$. Options for these programs are parameter file (*poni-file*) describing the geometry and the mask file. They can also do some pre-processing like dark-noise subtraction and flat-field correction (solid-angle correction is done by default).

A new Graphical interface based on Qt called *pyFAI-integrate* is now available, offers all options possible for azimuthal integration (dark/flat/polarization, ...) in addition to a finer tuning for the computing device selection (CPU/GPU).

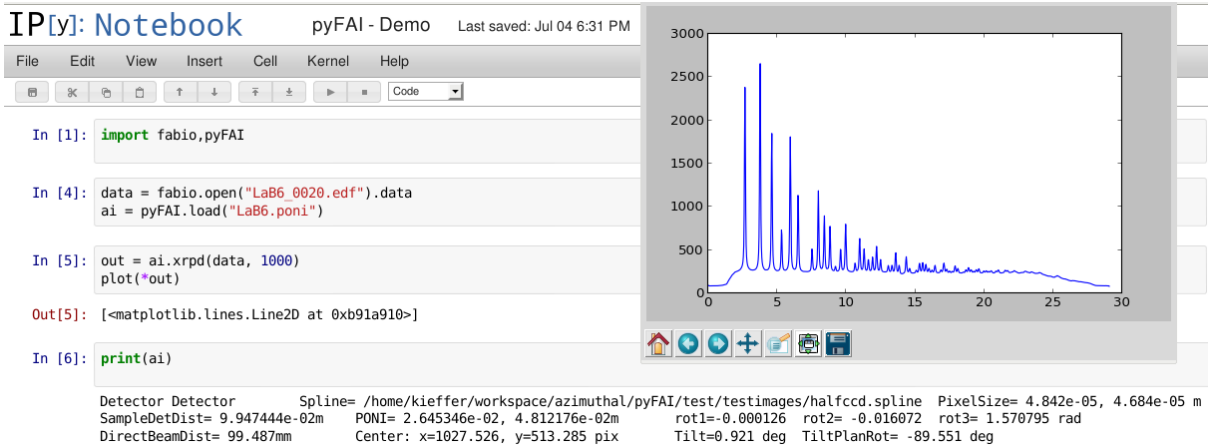
Finally a specialized tool called diff_tomo is available to reduce a mapping of 2D images into a 3D volume (math:*x, y, 2theta* for mapping or math:*rot, trans, 2theta* for tomography)

1.3.5 Python library

PyFAI is first and foremost a library: a tool of the scientific toolbox built around [IPython] and [NumPy] to perform data analysis either interactively or via scripts. Figure [notebook] shows an interactive session where an integrator is created, and an image loaded and integrated before being plotted.

1.4 Regrouping mechanism

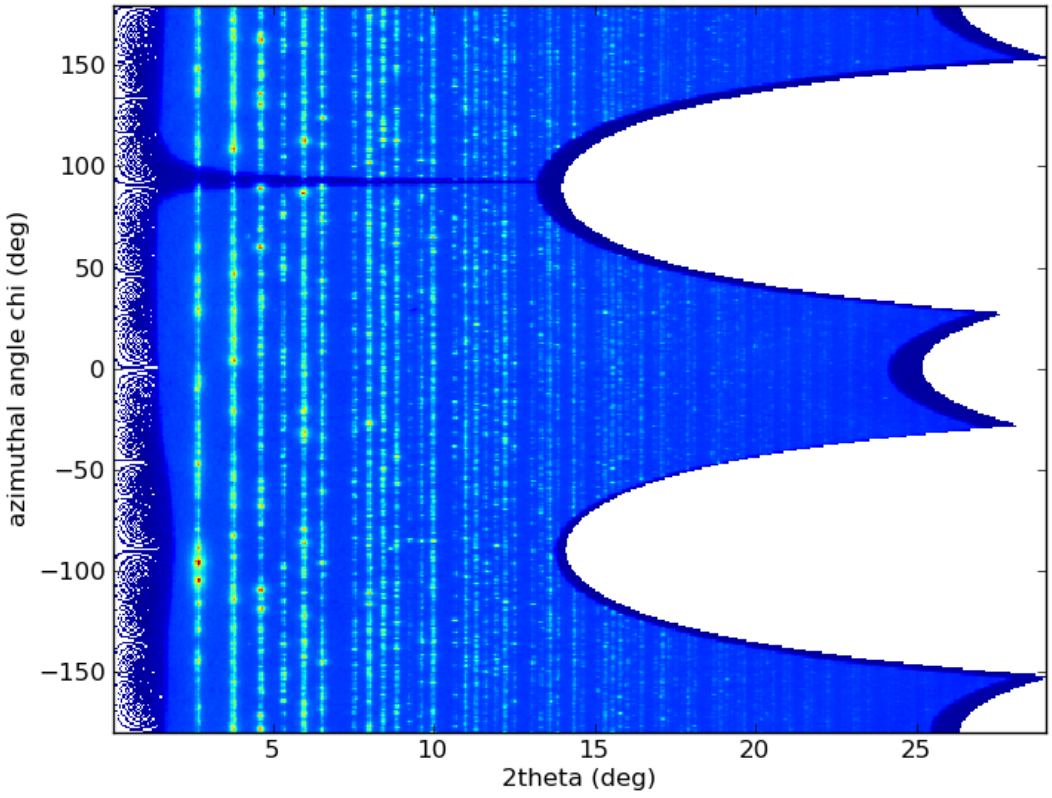
In pyFAI, regrouping is performed using a histogram-like algorithm. Each pixel of the image is associated to its polar coordinates $(2\theta, \chi)$ or (q, χ), then a pair of histograms versus 2θ (or q) are built, one non weighted for measuring the number of pixels falling in each bin and another weighted by pixel intensities (after dark-current



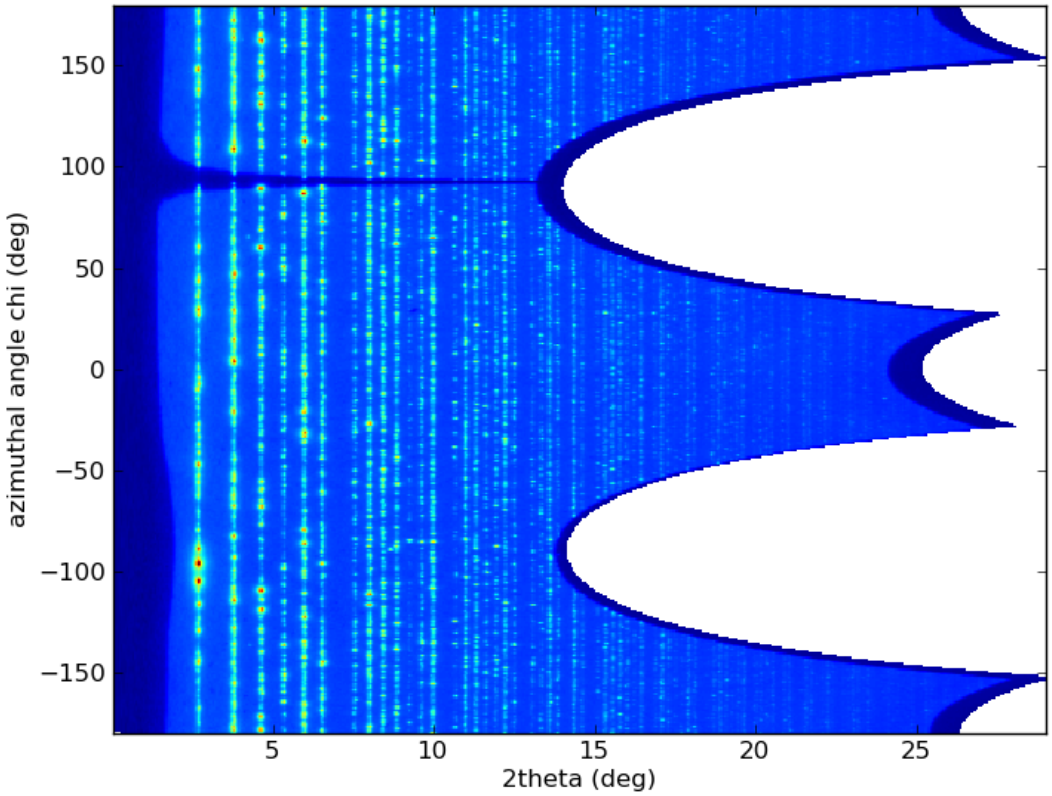
subtraction, and corrections for flat-field, solid-angle and polarization). The division of the weighted histogram by the number of pixels per bin gives the diffraction pattern. $2D$ regrouping (called *caking* in FIT2D) is obtained in the same way using two-dimensional histograms over radial (2θ or q) and azimuthal angles (χ).

1.4.1 Pixel splitting algorithm

Powder diffraction patterns obtained by histogramming have a major weakness where pixel statistics are low. A manifestation of this weakness becomes apparent in the $2D$-regrouping where most of the bins close to the beam-stop are not populated by any pixel. In this figure, many pixels are missing in the low 2θ region, due to the arbitrary discretization of the space in pixels as intensities were assigned to each pixel center which does not reflect the physical reality of the scattering experiment.



PyFAI solves this problem by pixel splitting : in addition to the pixel position, its spatial extension is calculated and each pixel is then split and distributed over the corresponding bins, the intensity being considered as homogeneous within a pixel and spread accordingly. The drawback of this is the correlation introduced between two adjacent bins. To simplify calculations, this was initially done by abstracting the pixel shape with a bounding box that circumscribes the pixel. In an effort to better the quality of the results this method was dropped in favoor of a full pixel splitting scheme that actually uses the actual pixel geometry for its calculations.



1.4.2 Performances and migration to native code

Originally, regrouping was implemented using the histogram provided by [NumPy], then re-implemented in [Cython] with pixel splitting to achieve a four-fold speed-up. The computation time scales like O(N) with the size of the input image. The number of output bins shows only little influence; overall the single threaded [Cython] implementation has been stated at 30 Mpix/s (on a 3.4 GHz Intel core i7-2600).

1.4.3 Parallel implementation

The method based on histograms works well on a single processor but runs into problems requiring so called "atomic operations" when run in parallel. Processing pixels in the input data order causes write access conflicts which become less efficient with the increase of number of computing units (need of atomic_operation)_. This is the main limit of the method exposed previously; especially on GPU where hundreds of threads are executed simultaneously.

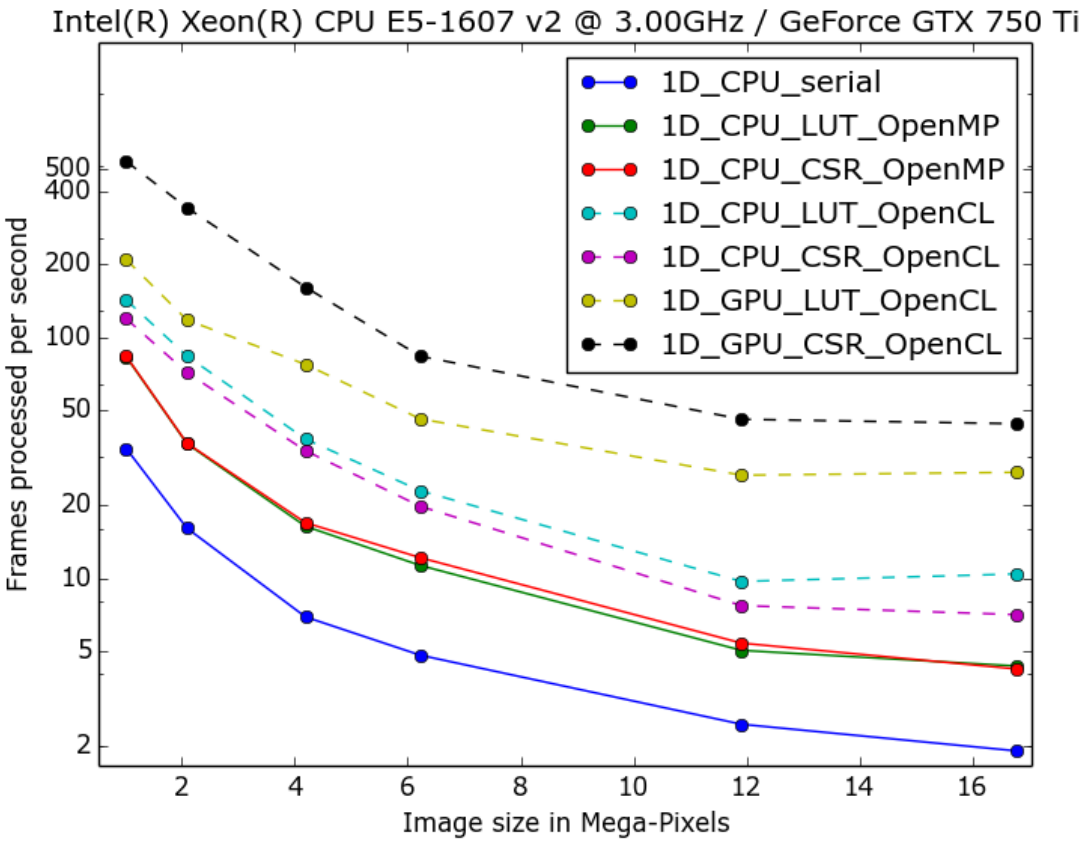
To overcome this limitation; instead of looking at where input pixels GO TO in the output image, we instead look at where the output pixels COME FROM in the input image. This transformation is called a "scatter to gather" transformation in parallel programming.

The correspondence between pixels and output bins can be stored in a look-up table (LUT) together with the pixel weight which make the integration look like a simple (if large and sparse) matrix vector product. This look-up table size depends on whether pixels are split over multiple bins and to exploit the sparse structure, both index and weight of the pixel have to be stored. We measured that 500 Mbytes are needed to store the LUT to integrate a 16 megapixels image, which fits onto a reasonable quality graphics card nowadays but can still be too large to fit on an entry-level graphics card.

By making this change we switched from a "linear read / random write" forward algorithm to a "random read / linear write" backward algorithm which is more suitable for parallelization. As a farther improvement on the algorithm, the use of compressed sparse row (CSR) format was introduced, to store the LUT data. This algorithm was implemented both in [Cython]-OpenMP and OpenCL. The CSR approach has a double benefit: first, it reduces the size of the storage needed compared to the LUT by a factor two to three, offering the opportunity of working with larger images on the same hardware. Secondly, the CSR implementation in OpenCL is using an algorithm based on multiple parallel reductions where many execution threads are collaborating to calculate the content of a single bin. This makes it very well suited to run on GPUs and accelerators where hundreds to thousands of simultaneous threads are available.

When using OpenCL for the GPU we used a compensated (or Kahan_summation), to reduce the error accumulation in the histogram summation (at the cost of more operations to be done). This allows accurate results to be obtained on cheap hardware that performs calculations in single precision floating-point arithmetic (32 bits) which are available on consumer grade graphic cards. Double precision operations are currently limited to high price and performance computing dedicated GPUs. The additional cost of Kahan summation, 4x more arithmetic operations, is hidden by smaller data types, the higher number of single precision units and that the GPU is usually limited by the memory bandwidth anyway.

The performances of the parallel implementation based on a LUT, stored in CSR format, can reach 750 MPix/s on recent multi-core computer with a mid-range graphics card. On multi-socket server featuring high-end GPUs like Tesla cards, the performances are similar with the additional capability to work on multiple detector simultaneously.



1.5 Related Work

We report here scientific software which are using pyFAI for azimuthal integration:

- "Dioptas" is a graphical user interface for calibrating and processing high pressure Xray diffraction experiment, developped by Clemens Perscher (APS, USA).
- "NanoPeakCell" allows the pre-treatement of serial crystallography written by Nicolas Coquelle (IBS, France)
- "Dpdak" is an open source tool for (online) analyzing large sequences of small angle scattering data [Dpdak]

1.6 Conclusion

The library pyFAI was developed with two main goals:

- Performing azimuthal integration with a clean programming interface.
- No compromise on the quality of the results is accepted: a careful management of the geometry and precise pixel splitting ensures total and local intensity preservation.

PyFAI is the first implementation of an azimuthal integration algorithm on a GPUs as far as we are aware of, and the stated twenty-fold speed up opens the door to a new kind of analysis, not even considered before. With a good interface close to the camera, we believe PyFAI is able to sustain the data streams from the next generation high-speed detectors.

1.6.1 Acknowledgments

Porting pyFAI to GPU would have not been possible without the financial support of LinkSCEEM-2 (RI-261600).

CHAPTER

TWO

COOKBOOK RECIPES

Cookbook are short tutorials: 1 page, 5 minutes to read.

2.1 Calibration of a diffraction setup

Author: Jérôme Kieffer

Date: 20/01/2015

Keywords: Calibration

Target: Scientists

Associated video: http://www.edna-site.org/pub/calibration/calib.flv

2.1.1 Review your calibration image

As viewer, try fabio_viewer from the FabIO package If you need to pre-process your data, look at pyFAI-average. In this example we have used a “max filter” over 20 frames using pyFAI-average.

2.1.2 Get all additional data

- calibrant used, here LaB6
- the energy or the wavelength
- detector geometry
- masks, ...

2.1.3 Start pyFAI-calib

Use the man page (or –help) to see all options

2.1.4 Pick peaks

- A few (5) points in the most inner
- Increase the counter to indicate the ring number
- Pick some extra point in outer ring
- right click to pick a point !

2.1.5 Review the group of peaks

Press Enter to do so... and check the ring assignment

The position of the expected rings is overlaid to the image Un-zoom to view them !

2.1.6 Acquire some more control points

- Use the recalib to extract new data-points
- free/fix/bound then refine

2.1.7 Visualize the integrated patterns

- integrate to view the integrated pattern
- then extract a few extra rings ...
- the geometry is displayed on the screen

2.1.8 Quit

All different geometries have been saved into the .poni file. It can directly be used for integration

That's all.

2.2 Azimuthal integration using the graphical user interface

Author: Jérôme Kieffer

Date: 20/01/2015

Keywords: Integration

Target: Scientists

Associated video: http://www.edna-site.org/pub/calibration/integration.flv

2.2.1 Look at your integrated patterns

PyFAI can perform 1D or 2D integration. To view 1D patterns, I will use *grace* Let's look at the integrated patterns obtained during calibration.

As you can see, only the 10 rings used for calibration are well defined.

This file is a text file containing as header all metadata needed to determine the geometry.

2D integrated pattern (aka cake images) are EDF images. Try *fabio_viewer* to see them.

Once again check the header of the file and the associated metadata.

2.2.2 Integrate a bunch of images

We will work with the 20 images used for the calibration.

2.2.3 Start pyFAI-intgrate

Either select files to process using the file-dialog or provide them on the command line.

2.2.4 Set the geometry

Simply load the PONI file and check the populated fields.

2.2.5 Azimuthal integration options

Check the dark/flat/ ... options Use the check-box to activate the option.

Do *NOT* forget to specify the number of radial bins !

2.2.6 Select the device for processing

Unless the processing will be done on the CPU using OpenMP.

Press OK to start the processing.

The generation of the Look-Up table takes a few seconds then all files get processed quickly

2.2.7 Run it again to perform caking

Same a previously ... but provide a number of azimuthal bins !

2.2.8 Visualize the integrated patterns

Once again I used *grace* and *fabio_viewer* to display the result.

That's all.

TODO

TODO

For more in depth explanation, see the tutorials section.

more in depth explanation ... To come

CHAPTER

THREE

PYFAI SCRIPTS MANUAL

While pyFAI is first and foremost a Python library to be used by developers, a set of scripts is provided to process a full diffraction experiment on the command line without knowing anything about Python. Those scipts can be divided into 3 categories: pre-processing tools which prepare the dataset for the calibration tool. The calibration is the determination of the geometry of the experimental setup using Debye-Scherrer rings of a reference compound (or calibrant). Finally a full dataset can be integrated using different tools targeted at different experiments.

Pre-processing tools:

- drawMask_pymca: tool for drawing a mask on top of an image
- pyFAI-average: tool for averaging/median/... filtering images (i.e. for dark current)

Calibration tools:

- pyFAI-calib: manually select the rings and refine the geometry
- pyFAI-recalib: automatic ring extraction to refine the geometry (deprecated: see "recalib" option in pyFAI-calib)
- MX-calibrate: Calibrate automatically a set of images taken at various detector distances
- check_calib: checks the calibration of an image at the sub-pixel level (deprecated: see "validate" option in pyFAI-calib)

Azimuthal integration tools:

- pyFAI-integrate: the only graphical interface for integration
- pyFAI-saxs: command line interface for small-angle scattering
- pyFAI-waxs: command line interface for powder difration
- diff_tomo: diffraction mapping&tomography tool

3.1 Preprocessing tool: pyFAI-average

3.1.1 Purpose

This tool can be used to average out a set of dark current images using mean or median filter (along the image stack). One can also reject outliers be specifying a cutoff (remove cosmic rays / zingers from dark)

It can also be used to merge many images from the same sample when using a small beam and reduce the spottyness of Debye-Scherrer rings. In this case the "max-filter" is usually recommended.

3.1.2 Options:

Usage: pyFAI-average [options] -o output.edf file1.edf file2.edf ...

Options:

--version show program's version number and exit

-h, --help show help message and exit

-o OUTPUT, --output=OUTPUT Output/ destination of average image

-m METHOD, --method=METHOD Method used for averaging, can be 'mean'(default) or 'median', 'min' or 'max'

-c CUTOFF, --cutoff=CUTOFF Take the mean of the average +/- cutoff * std_dev.

-f FORMAT, --format=FORMAT Output file/image format (by default EDF)

-v, --verbose switch to verbose/debug mode

3.2 Mask generation tool: drawMask_pymca

3.2.1 Purpose

Draw a mask, i.e. an image containing the list of pixels which are considered invalid (no scintillator, module gap, beam stop shadow, ...).

This will open a PyMca window and let you draw on the first image (provided) with different tools (brush, rectangle selection, ...). When you are finished, come back to the console and press enter. The mask image is saved into file1-masked.edf. Optionally the script will print the number of pixel masked and the intensity masked (as well on other files provided in input)

Usage: drawMask_pymca [options] file1.edf file2.edf ...

3.2.2 Options:

--version show program's version number and exit

-h, --help show help message and exit

Optionally the script will print the number of pixel masked and the intensity masked (as well on other files provided in input)

3.3 Calibration tool: pyFAI-calib

3.3.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images without a priori knowledge of your setup. You will need to provide a calibrant or a "d-spacing" file containing the spacing of Miller plans in Angstrom (in decreasing order). If you are using a standard calibrant, look at https://github.com/kif/pyFAI/tree/master/calibration or search in the American Mineralogist database: [AMD] or in the [COD]. The –calibrant option is mandatory !

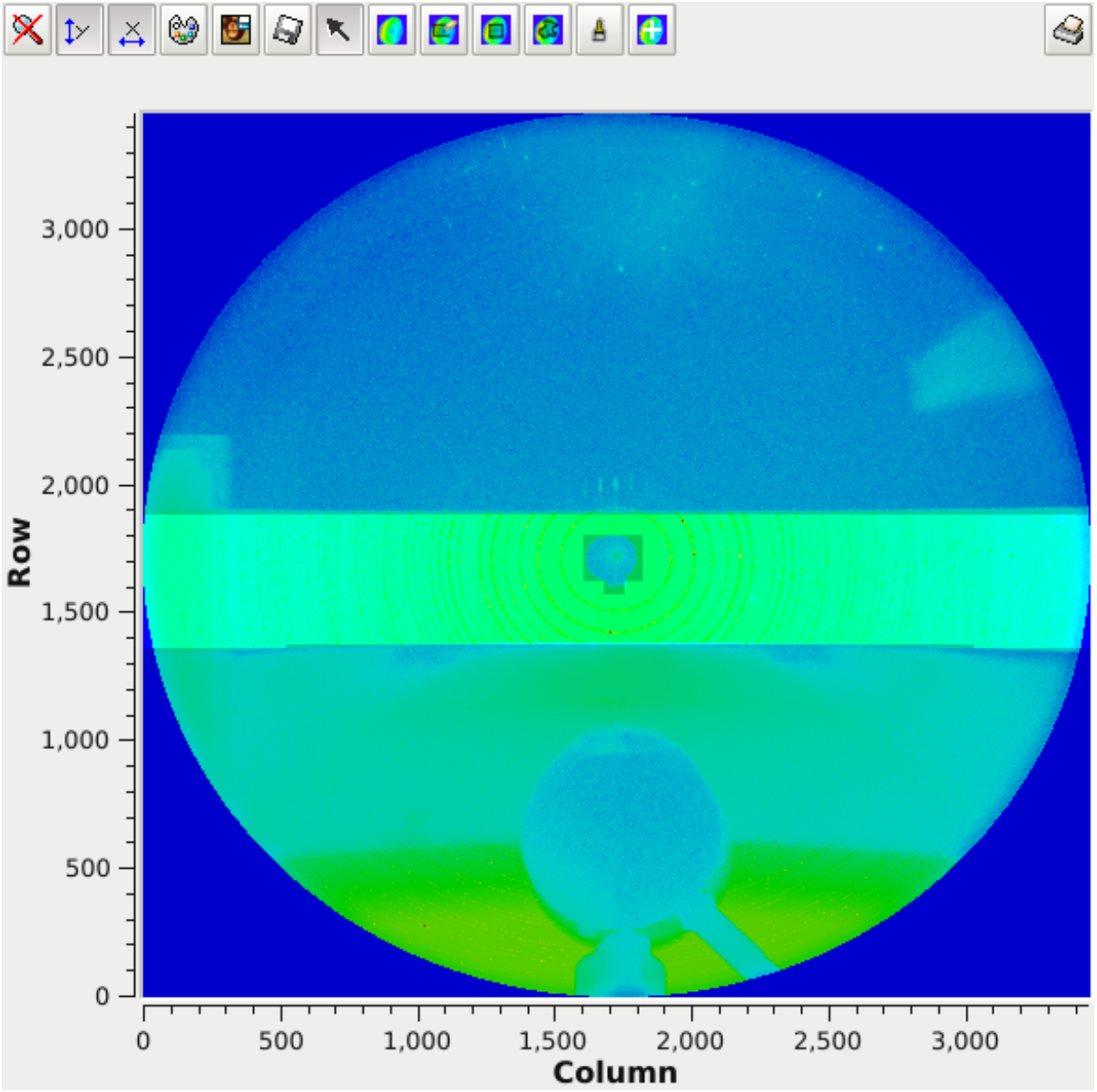
You will need in addition:

- The radiation energy (in keV) or its wavelength (in A)
- The description of the detector: it name or it's pixel size or the spline

file describing its distortion

Many option are available among those:

- dark-current / flat field corrections
- Masking of bad regions


3,000
2,500
2,000
1,500
1,000
500
0
Row
0
500
1,000
1,500
2,000
2,500
3,000
Column

- reconstruction of missing region (module based detectors)
- Polarization correction
- Automatic desaturation (time consuming!)
- Intensity weighted least-squares refinements

The output of this program is a "PONI" file containing the detector description and the 6 refined parameters (distance, center, rotation) and wavelength. An 1D and 2D diffraction patterns are also produced. (.dat and .azim files)

3.3.2 Usage:

pyFAI-calib [options] -w 1 -D detector -c calibrant.D imagefile.edf

3.3.3 Options:

--version show program's version number and exit

-h, --help show this help message and exit

-o FILE, --out=FILE Filename where processed image is saved

-v, --verbose switch to debug/verbose mode

-c FILE, --calibrant=FILE Calibrant name or file containing d-spacing of the reference sample (MANDATORY)

-w WAVELENGTH, --wavelength=WAVELENGTH wavelength of the X-Ray beam in Angstrom

-e ENERGY, --energy=ENERGY energy of the X-Ray beam in keV (hc=12.398419292keV.A)

-P POLARIZATION_FACTOR, --polarization=POLARIZATION_FACTOR polarization factor, from -1 (vertical) to +1 (horizontal), default is None (no correction), synchrotrons are around 0.95

-b BACKGROUND, --background=BACKGROUND Automatic background subtraction if no value are provided

-d DARK, --dark=DARK list of dark images to average and subtract

-f FLAT, --flat=FLAT list of flat images to average and divide

-s SPLINE, --spline=SPLINE spline file describing the detector distortion

-D DETECTOR_NAME, --detector=DETECTOR_NAME Detector name (instead of pixel size+spline)

-m MASK, --mask=MASK file containing the mask (for image reconstruction)

-n NPT, --pt=NPT file with datapoints saved. Default: basename.npt

--filter=FILTER select the filter, either mean(default), max or median

-l DISTANCE, --distance=DISTANCE sample-detector distance in millimeter

--poni1=PONI1 poni1 coordinate in meter

--poni2=PONI2 poni2 coordinate in meter

--rot1=ROT1 rot1 in radians

--rot2=ROT2 rot2 in radians

--rot3=ROT3 rot3 in radians

--fix-dist fix the distance parameter

--free-dist free the distance parameter

--fix-poni1 fix the poni1 parameter

--free-poni1 free the poni1 parameter

--fix-poni2 fix the poni2 parameter

--free-poni2 free the poni2 parameter

--fix-rot1 fix the rot1 parameter

--free-rot1 free the rot1 parameter

--fix-rot2 fix the rot2 parameter

--free-rot2 free the rot2 parameter

--fix-rot3 fix the rot3 parameter

--free-rot3 free the rot3 parameter

--fix-wavelength fix the wavelength parameter

--free-wavelength free the wavelength parameter

--saturation=SATURATION consider all pixel>max*(1-saturation) as saturated and reconstruct them

--weighted weight fit by intensity, by default not.

--npt=NPT_1D Number of point in 1D integrated pattern, Default: 1024

--npt-azim=NPT_2D_AZIM Number of azimuthal sectors in 2D integrated images. Default: 360

--npt-rad=NPT_2D_RAD Number of radial bins in 2D integrated images. Default: 400

--unit=UNIT Valid units for radial range: 2th_deg, 2th_rad, q_nm^-1, q_A^-1, r_mm. Default: 2th_deg

--no-gui force the program to run without a Graphical interface

--no-interactive force the program to run and exit without prompting for refinements

-r, --reconstruct Reconstruct image where data are masked or <0 (for Pilatus detectors or detectors with modules)

-g GAUSSIAN, --gaussian=GAUSSIAN Size of the gaussian kernel. Size of the gap (in pixels) between two consecutive rings, by default 100 Increase the value if the arc is not complete; decrease the value if arcs are mixed together.

--square Use square kernel shape for neighbor search instead of diamond shape

-p PIXEL, --pixel=PIXEL size of the pixel in micron

3.3.4 Example of usage:

Pilatus 1M image of Silver Behenate taken at ESRF-BM26:

```
pyFAI-calib -D Pilatus1M -c AgBh -r -w 1.0 test/testimages/Pilatus1M.edf
```

We use the parameter -r to reconstruct the missing part between the modules of the Pilatus detector.

Half a FReLoN CCD image of Lantanide hexaboride taken at ESRF-ID11:

```
pyFAI-calib -s test/testimages/halfccd.spline -c LaB6 -w 0.3 test/testimages/halfccd.edf -g 250
```

This image is rather spotty. We need to blur a lot to get the continuity of the rings. This is achieved by the -g parameter. While the sample is well diffracting and well known, the wavelength has been guessed. One should refine the wavelength when the peaks extracted are correct

All those images are part of the test-suite of pyFAI. To download them from internet, run

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

Downloaded test images are located in tests/testimages

3.4 Calibration tool: pyFAI-recalib

3.4.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images with a priori knowledge of your setup (an input PONI-file). You will need to provide a calibrant or a "d-spacing" file containing the spacing of Miller plans in Angstrom (in decreasing order). If you are using a standard calibrant, look at https://github.com/kif/pyFAI/tree/master/calibration Else, you will need a "d-spacing" file containing the spacing of Miller plans in Angstrom (in decreasing order), they can be found on the American Mineralogist database [AMD] or in the [COD]. The –calibrant option is mandatory !

You will need in addition:

- The radiation energy (in keV) or its wavelength (in A)

Many option are available among those:

- dark-current / flat field corrections
- Masking of bad regions
- Polarization correction
- Automatic desaturation (time consuming!)
- Intensity weighted least-squares refinements

The output of this program is a "PONI" file containing the detector description and the 6 refined parameters (distance, center, rotation) and wavelength. An 1D and 2D diffraction patterns are also produced. (.dat and .azim files)

The main difference with pyFAI-calib is the way control-point hence Debye-Sherrer rings are extracted. While pyFAI-calib relies on the contiguity of a region of peaks called massif; pyFAI-recalib knows approximatly the geometry and is able to select the region where the ring should be. From this region it selects automatically the various peaks; making pyFAI-recalib able to run without graphical interface and without human intervention (–no-gui –no-interactive options).

3.4.2 Usage:

pyFAI-recalib [options] -w 1 -p imagefile.poni -S calibrant.D imagefile.edf

3.4.3 Options:

-h, --help	show help message and exit
-V, --version	print version of the program and quit
-o FILE, --out=FILE	Filename where processed image is saved

-v, --verbose switch to debug/verbose mode

-S FILE, --spacing=FILE file containing d-spacing of the reference sample (MANDATORY)

-w WAVELENGTH, --wavelength=WAVELENGTH wavelength of the X-Ray beam in Angstrom

-e ENERGY, --energy=ENERGY energy of the X-Ray beam in keV (hc=12.398419292keV.A)

-P POLARIZATION_FACTOR, --polarization=POLARIZATION_FACTOR polarization factor, from -1 (vertical) to +1 (horizontal), default is None (no correction), synchrotrons are around 0.95

-b BACKGROUND, --background=BACKGROUND Automatic background subtraction if no value are provided

-d DARK, --dark=DARK list of dark images to average and subtract

-f FLAT, --flat=FLAT list of flat images to average and divide

-s SPLINE, --spline=SPLINE spline file describing the detector distortion

-D DETECTOR_NAME, --detector=DETECTOR_NAME Detector name (instead of pixel size+spline)

-m MASK, --mask=MASK file containing the mask (for image reconstruction)

-n NPT, --pt=NPT file with datapoints saved. Default: basename.npt

--filter=FILTER select the filter, either mean(default), max or median

-l DISTANCE, --distance=DISTANCE sample-detector distance in millimeter

--poni1=PONI1 poni1 coordinate in meter

--poni2=PONI2 poni2 coordinate in meter

--rot1=ROT1 rot1 in radians

--rot2=ROT2 rot2 in radians

--rot3=ROT3 rot3 in radians

--fix-dist fix the distance parameter

--free-dist free the distance parameter

--fix-poni1 fix the poni1 parameter

--free-poni1 free the poni1 parameter

--fix-poni2 fix the poni2 parameter

--free-poni2 free the poni2 parameter

--fix-rot1 fix the rot1 parameter

--free-rot1 free the rot1 parameter

--fix-rot2 fix the rot2 parameter

--free-rot2 free the rot2 parameter

--fix-rot3 fix the rot3 parameter

--free-rot3 free the rot3 parameter

--fix-wavelength fix the wavelength parameter

--free-wavelength free the wavelength parameter

--saturation=SATURATION consider all pixel>max*(1-saturation) as saturated and reconstruct them

--weighted weight fit by intensity, by default not.

--npt=NPT_1D Number of point in 1D integrated pattern, Default: 1024

--npt-azim=NPT_2D_AZIM Number of azimuthal sectors in 2D integrated images. Default: 360

--npt-rad=NPT_2D_RAD Number of radial bins in 2D integrated images. Default: 400

--unit=UNIT Valid units for radial range: 2th_deg, 2th_rad, q_nm^-1, q_A^-1, r_mm. Default: 2th_deg

--no-gui force the program to run without a Graphical interface

--no-interactive force the program to run and exit without prompting for refinements

-r MAX_RINGS, --ring=MAX_RINGS maximum number of rings to extract. Default: all accessible

-p FILE, --poni=FILE file containing the diffraction parameter (poni-file). MANDATORY

-k, --keep Keep existing control point and append new

3.4.4 Tips & Tricks

PONI files are ASCII files and each new refinement adds an entry int the file. So if you are unhappy with the last step, just edit this file and remove the last entry (timestamps will help you).

3.5 Calibration tool: check_calib

3.5.1 Purpose

Check_calib is a research tool aiming at validating both the geometric calibration and everything else like flat-field correction, distortion correction, at a sub-pixel level.

Note that *check_calib* program is obsolete as the same functionnality is available from within pyFAI-calib, using the *validate* command in the refinement process.

3.5.2 Usage:

check_calib [options] -p param.poni image.edf

3.5.3 Options:

-h, --help show this help message and exit

-V, –version -v, –verbose switch to debug mode -d FILE, –dark FILE file containing the dark images to subtract -f FILE, –flat FILE file containing the flat images to divide -m FILE, –mask FILE file containing the mask -p FILE, –poni FILE file containing the diffraction parameter (poni-file) -e ENERGY, –energy ENERGY

energy of the X-Ray beam in keV (hc=12.398419292keV.A)

-w WAVELENGTH, --wavelength WAVELENGTH wavelength of the X-Ray beam in Angstrom

3.5.4 Arguments:

FILE Image file to check calibration for

3.6 Calibration tool: MX-calibrate

3.6.1 Purpose

Calibrate automatically a set of frames taken at various sample-detector distance.

This tool has been developed for ESRF MX-beamlines where an acceptable calibration is usually present is the header of the image. PyFAI reads it and does a "recalib" on each of them before exporting a linear regression of all parameters versus this distance.

Most standard calibrants are directly installed together with pyFAI. If you prefer using your own, you can provide a "d-spacing" file containing the spacing of Miller plans in Angstrom (in decreasing order). Most crystal powders used for calibration are available in the American Mineralogist database [AMD] or in the [COD].

3.6.2 Usage:

MX-Calibrate -w 1.54 -c CeO2 file1.cbf file2.cbf ...

3.6.3 Options:

-h, --help show this help message and exit

-V, --version print version of the program and quit

-v, --verbose switch to debug/verbose mode

-c FILE, --calibrant FILE calibrant name or file containing d-spacing of the calibrant reference sample (MANDATORY)

-w WAVELENGTH, --wavelength=WAVELENGTH wavelength of the X-Ray beam in Angstrom

-e ENERGY, --energy=ENERGY energy of the X-Ray beam in keV (hc=12.398419292keV.A)

-P POLARIZATION_FACTOR, --polarization=POLARIZATION_FACTOR polarization factor, from -1 (vertical) to +1 (horizontal), default is 0, synchrotrons are around 0.95

-b BACKGROUND, --background=BACKGROUND Automatic background subtraction if no value are provided

-d DARK, --dark=DARK list of dark images to average and subtract

-f FLAT, --flat=FLAT list of flat images to average and divide

-s SPLINE, --spline=SPLINE spline file describing the detector distortion

-p PIXEL, --pixel=PIXEL size of the pixel in micron

-D DETECTOR_NAME, --detector=DETECTOR_NAME Detector name (instead of pixel size+spline)

-m MASK, --mask=MASK file containing the mask (for image reconstruction)

--filter=FILTER select the filter, either mean(default), max or median

--saturation=SATURATION consider all pixel>max*(1-saturation) as saturated and reconstruct them

-r MAX_RINGS, --ring=MAX_RINGS maximum number of rings to extract

--weighted weight fit by intensity

-l DISTANCE, --distance=DISTANCE sample-detector distance in millimeter

--no-tilt refine the detector tilt

--poni1=PONI1 poni1 coordinate in meter

--poni2=PONI2 poni2 coordinate in meter

--rot1=ROT1 rot1 in radians

--rot2=ROT2 rot2 in radians

--rot3=ROT3 rot3 in radians

--fix-dist fix the distance parameter

--free-dist free the distance parameter

--fix-poni1 fix the poni1 parameter

--free-poni1 free the poni1 parameter

--fix-poni2 fix the poni2 parameter

--free-poni2 free the poni2 parameter

--fix-rot1 fix the rot1 parameter

--free-rot1 free the rot1 parameter

--fix-rot2 fix the rot2 parameter

--free-rot2 free the rot2 parameter

--fix-rot3 fix the rot3 parameter

--free-rot3 free the rot3 parameter

--fix-wavelength fix the wavelength parameter

--free-wavelength free the wavelength parameter

--no-gui force the program to run without a Graphical interface

--gui force the program to run with a Graphical interface

--no-interactive force the program to run and exit without prompting for refinements

--interactive force the program to prompt for refinements

--peak-picker PEAKPICKER Uses the 'massif' or the 'blob' peak-picker algorithm (default: blob)

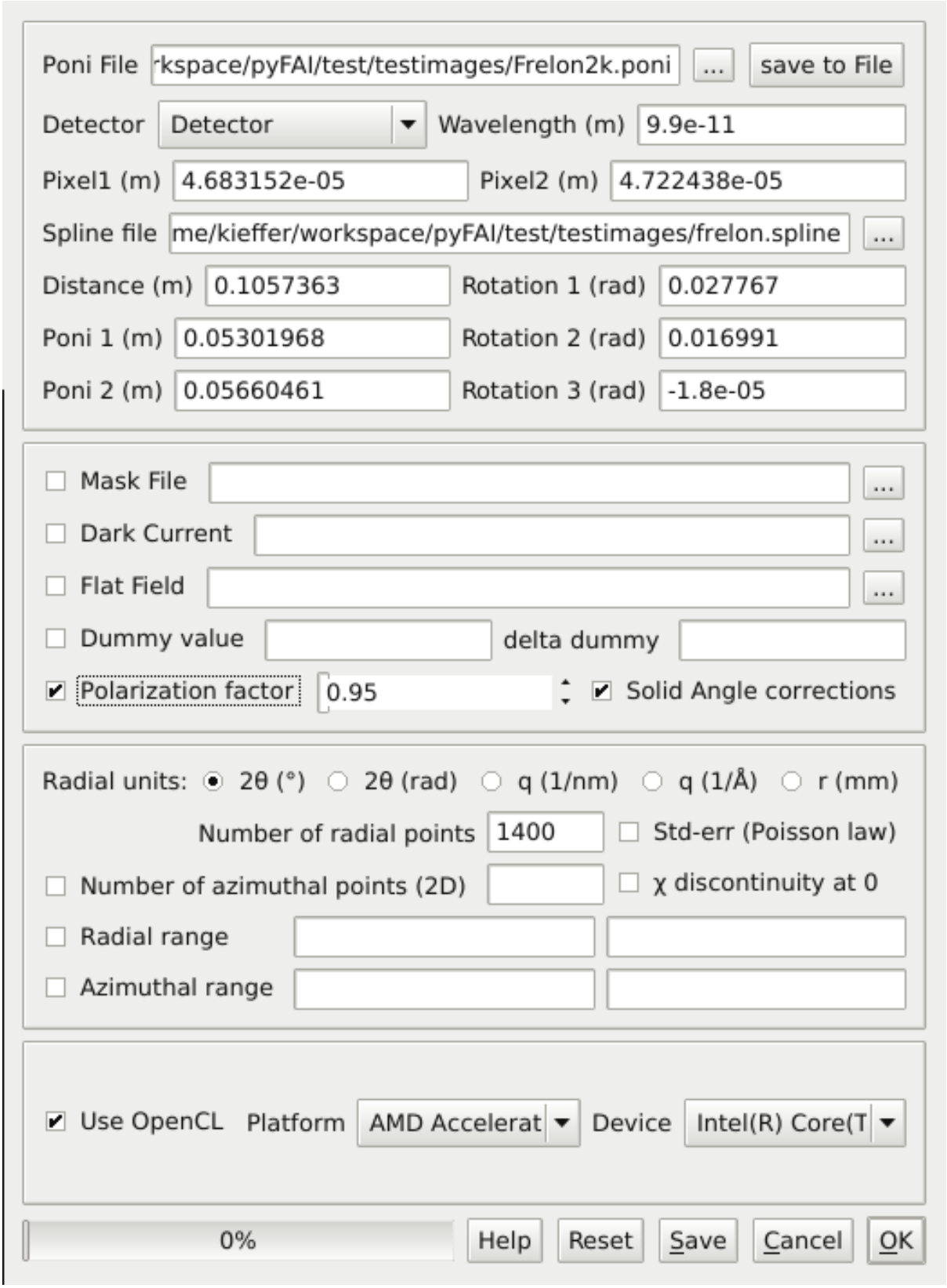
3.7 Integration tool: pyFAI-integrate

3.7.1 Purpose

PyFAI-integrate is a graphical interface (based on Python/Qt4) to perform azimuthal integration on a set of files. It exposes most of the important options available within pyFAI and allows you to select a GPU (or an openCL platform) to perform the calculation on.

3.7.2 Usage

pyFAI-integrate [options] file1.edf file2.edf ...


Poni File
kspace/pyFAI/test/testimages/Frelon2k.poni
...
save to File
Detector
Detector
Wavelength (m)
9.9e-11
Pixel1 (m)
4.683152e-05
Pixel2 (m)
4.722438e-05
Spline file
me/kieffer/workspace/pyFAI/test/testimages/frelon.spline
Distance (m)
0.1057363
Rotation 1 (rad)
0.027767
Poni 1 (m)
0.05301968
Rotation 2 (rad)
0.016991
Poni 2 (m)
0.05660461
Rotation 3 (rad)
-1.8e-05
Mask File
Dark Current
Flat Field
Dummy value
delta dummy
Polarization factor
0.95
Solid Angle corrections
Radial units:
2θ (°)
2θ (rad)
q (1/nm)
q (1/Å)
r (mm)
Number of radial points
1400
Std-err (Poisson law)
Number of azimuthal points (2D)
χ discontinuity at 0
Radial range
Azimuthal range
Use OpenCL
Platform
AMD Accelerat
Device
Intel(R) Core(T
0%
Help
Reset
Save
Cancel
OK

3.7.3 Options:

--version show program's version number and exit

-h, --help show help message and exit

-v, --verbose switch to verbose/debug mode

-o OUTPUT, --output=OUTPUT Directory or file where to store the output data

3.7.4 Tips & Tricks:

PyFAI-integrate saves all parameters in a .azimint.json (hidden) file. This JSON file is an ascii file which can be edited and used to configure online data analysis using the LImA plugin of pyFAI.

Nota: there is bug in debian6 making the GUI crash (to be fixed inside pyqt) http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=697348

3.8 Integration tool: diff_tomo

3.8.1 Purpose

Azimuthal integration for diffraction tomography.

Diffraction tomography is an experiment where 2D diffraction patterns are recorded while performing a 2D scan, one (the slowest) in rotation around the sample center and the other (the fastest) along a translation through the sample. Diff_tomo is a script (based on pyFAI and h5py) which allows the reduction of this 4D dataset into a 3D dataset containing the rotations angle (hundreds), the translation step (hundreds) and the many diffraction angles (thousands). The resulting dataset can be opened using PyMca roitool where the 1d dataset has to be selected as last dimension. This file is not (yet) NeXus compliant.

This tool can be used for mapping experiments if one considers the slow scan direction as the rotation.

tips: If the number of files is too large, use double quotes around "*.edf"

3.8.2 Usage:

diff_tomo [options] -p ponifile imagefiles*

3.8.3 Options:

--version show program's version number and exit

-h, --help show help message and exit

-o FILE, --output=FILE HDF5 File where processed sinogram was saved

-v, --verbose switch to verbose/debug mode

-P FILE, --prefix=FILE Prefix or common base for all files

-e EXTENSION, --extension=EXTENSION Process all files with this extension

-t NTRANS, --nTrans=NTRANS number of points in translation

-r NROT, --nRot=NROT number of points in rotation

-c NDIFF, --nDiff=NDIFF number of points in diffraction powder pattern

-d FILE, --dark=FILE list of dark images to average and subtract

-f FILE, --flat=FILE list of flat images to average and divide

-m FILE, --mask=FILE file containing the mask

-p FILE, --poni=FILE file containing the diffraction parameter (poni-file)

-O OFFSET, --offset=OFFSET do not process the first files

-g, --gpu process using OpenCL on GPU

Most of those options are mandatory to define the structure of the dataset.

3.9 Integration tool: pyFAI-saxs

3.9.1 Purpose

Azimuthal integration for SAXS users.

pyFAI-saxs is the SAXS script of pyFAI that allows data reduction (azimuthal integration) for Small Angle Scattering with output axis in q space.

3.9.2 Usage:

pyFAI-saxs -p=param.poni -w1.54e-9 file.edf file2.edf file3.edf

Options:

--version show program's version number and exit

-h, --help show this help message and exit

-p PONIFILE PyFAI parameter file (.poni)

-n NPT Number of points in radial dimension

-w WAVELENGTH, --wavelength=WAVELENGTH wavelength of the X-Ray beam in Angstrom

-e ENERGY, --energy=ENERGY energy of the X-Ray beam in keV (hc=12.398419292keV.A)

-u DUMMY, --dummy=DUMMY dummy value for dead pixels

-U DELTA_DUMMY, --delta_dummy=DELTA_DUMMY delta dummy value

-m MASK, --mask=MASK name of the file containing the mask image

-d DARK, --dark=DARK name of the file containing the dark current

-f FLAT, --flat=FLAT name of the file containing the flat field

-P POLARIZATION_FACTOR, --polarization=POLARIZATION_FACTOR Polarization factor, from -1 (vertical) to +1 (horizontal), default is None for no correction, synchrotrons are around 0.95

--error-model=ERROR_MODEL Error model to use. Currently on 'poisson' is implemented

--unit=UNIT unit for the radial dimension: can be q_nm^-1, q_A^-1, 2th_deg, 2th_rad or r_mm

--ext=EXT extension of the regrouped filename (.dat)

3.10 Integration tool: pyFAI-saxs

3.10.1 Purpose

Azimuthal integration for WAXS users.

pyFAI-waxs is the script of pyFAI that allows data reduction (azimuthal integration) for Wide Angle Scattering to produce X-Ray Powder Diffraction Pattern with output axis in 2-theta space.

3.10.2 Usage:

pyFAI-waxs -p param.poni [options] file1.edf file2.edf ...

3.10.3 Options:

--version show program's version number and exit

-h, --help show this help message and exit

-p PONIFILE PyFAI parameter file (.poni) MANDATORY

-n NPT Number of points in radial dimension

-w WAVELENGTH, --wavelength=WAVELENGTH wavelength of the X-Ray beam in Angstrom

-e ENERGY, --energy=ENERGY energy of the X-Ray beam in keV (hc=12.398419292keV.A)

-u DUMMY, --dummy=DUMMY dummy value for dead pixels

-U DELTA_DUMMY, --delta_dummy=DELTA_DUMMY delta dummy value

-m MASK, --mask=MASK name of the file containing the mask image

-d DARK, --dark=DARK name of the file containing the dark current

-f FLAT, --flat=FLAT name of the file containing the flat field

-P POLARIZATION_FACTOR, --polarization=POLARIZATION_FACTOR Polarization factor, from -1 (vertical) to +1 (horizontal), default is None for no correction, synchrotrons are around 0.95

--error-model=ERROR_MODEL Error model to use. Currently on 'poisson' is implemented

--unit=UNIT unit for the radial dimension: can be q_nm^-1, q_A^-1, 2th_deg, 2th_rad or r_mm

--ext=EXT extension of the regrouped filename (.xy)

--multi Average out all frame in a file before integrating

--average AVERAGE Method for averaging out: can be 'mean' (default), 'min', 'max' or 'median'

--do-2D Perform 2D integration in addition to 1D

pyFAI-waxs is the script of pyFAI that allows data reduction (azimuthal integration) for Wide Angle Scattering to produce X-Ray Powder Diffraction Pattern with output axis in 2-theta space.

CHAPTER

FOUR

DESIGN OF THE PYTHON FAST AZIMUTHAL INTEGRATION LIBRARY

Author: Jérôme Kieffer

Date: 18/12/2014

Keywords: Design

Target: Developers interested in using the library

Reference: API documentation

4.1 Design of the Python Fast Azimuthal Integrator

Author: Jérôme Kieffer

Date: 20/03/2015

Keywords: Design

Target: Developers interested in using the library

Reference: API documentation

4.1.1 Abstract

The core part of pyFAI is the AzimuthalIntegator objects, named *ai* hereafter. This document describes the two importants methods of the class, how it is related to Detector, Geometry, and integration engines.

One of the core idea is to have a complete representation of the geometry and perform the azimuthal integration as a single geometrical re-binning which take into account all effects like:

- Detector distortion
- Polar transformation
- assignment to the output space

This document focuses on the core of pyFAI while peripheral code dealing with graphical user interfaces, image analysis online data analysis integration are not covered.

4.1.2 AzimuthalIntegrator

This class is the core of pyFAI, and it is the only one likely to be used by external developers/users. It is usually instantiated via a function of the module to load a poni-file:

As one can see, the *ai* contains the detector geometry (type, pixel size, distortion) as well as the geometry of the experimental setup. The geometry is given in two equivalent forms: the internal representation of pyFAI (second line) and the one used by FIT2D.

The *ai* is responsible for azimuthal integration, either the integration along complete ring, called full-integration, obtained via *ai.integrate1d* method. The sector-wise integration is obtained via the *ai.integrate2d* method. The options for those two methods are really similar and differ only by the parameters related to the azimuthal dimension of the averaging for *ai.integrate2d*.

Azimuthal integration methods

Both integration method take as first argument the image coming from the detector as a numpy array. This is the only mandatory parameter.

Important parameters are the number of bins in radial and azimuthal dimensions. Other parameters are the preprocessing information like dark and flat pixel wise correction (as array), the polarization factor and the solid-angle correction to be applied.

Because multiple radial output space are possible (q, r, 2theta) each with multiple units, if one wants to avoid interpolation, it is important to export directly the data in the destination space, specifying the unit="2th_deg" or "q_nm^-1"

Many more option exists, please refer to the documentation of AzimuthalIntegration **integrate_**

The AzimuthalIntegration class inherits from the Geometry class and hold references to configured rebinning engines.

4.1.3 Geometry

The Geometry class contains a reference to the detector (composition) and the logic to calculate the position in space of the various pixels. All arrays in the class are cached and calculated on demand.

The Geometry class relies on the detector to provide the pixel position in space and subsequently transforms it in 2theta coordinates, or q, chi, r ... This can either be performed in the class itself or by calling function in the parallel implemented Cython module _geometry. Those transformation could be GPU-ized in the future.

4.1.4 Detector

PyFAI deals only with area detector, indexed in 2 dimension but can handle pixel located in a 3D space.

The *pyFAI.detectors* module contains the master *Detector* class which is capable of describing any detector. About 40 types of detectors, inheriting and specializing the *Detector* class are provided, offering convienient access to most commercial detectors. A factory is provided to easily instantiate a detector from its name.

A detector class is responsible for two main tasks:

- provide the coordinate in space of any pixel position (center, corner, ...)
- Handle the mask: some detector feature automatic mask calculation (i.e. module based detectors).

The disortion of the detector is handled here and could be GPU-ized in the future.

4.1.5 Rebinning engines

Once the geometry (radial and azimuthal coordinates) calculated for every pixel on the detector, the image from the detector is rebinned into the output space. Two types of rebinning engines exists:

Histograms They take each single pixel from the image and transfer it to the destination bin, like histograms do. This family of algorithms is rather easy to implement and provides good single threaded performances, but it is hard to parallelize (efficiently) due to the need of atomic operations.

Sparse matrix multiplication By recording where every single ends one can transform the previous histogram into a large sparse matrix multiplication which is either stored as a Look-Up Table (actually an array of struct, also called LIL) or more efficiently in the **CSR_** format. Those rebinning engines are trivially parallel and provide the best performances.

4.1.6 Pixel splitting

Three levels of pixel splitting schemes are available within pyFAI:

No splitting The whole intensity is assigned to the center of the pixel and rebinned using a simple histogram

Bounding box pixel splitting The pixel is abstracted by a box surrounding it with, making calculation easier but blurring a bit the image

Tight pixel splitting The pixel is represented by its actual corner position, offering a very precise positionning in space.

The main issue with pixel splitting arose from 2D integration and the habdling of pixel laying on the chi-discontinuity.

4.1.7 References:

:: _integrate: http://pythonhosted.org/pyFAI/api/pyFAI.html#pyFAI.azimuthalIntegrator.AzimuthalIntegrator.integrate1d

:: _CSR: http://en.wikipedia.org/wiki/Sparse_matrix

CHAPTER

FIVE

PYFAI API

This chapter describes the programming interface of pyFAI, so what you can expect after having launched ipython and typed: ..

    import pyFAI

The most important class is AzimuthalIntegrator which is an object containing both the geometry (it inherits from Geometry, another class) and exposes important methods (functions) like integrate1d and integrate2d.

5.1 pyFAI Package

pyFAI.__init__.**tests**()

5.2 azimuthalIntegrator Module

class pyFAI.azimuthalIntegrator.**AzimuthalIntegrator**(*dist=1*, *poni1=0*, *poni2=0*, *rot1=0*, *rot2=0*, *rot3=0*, *pixel1=None*, *pixel2=None*, *splineFile=None*, *detector=None*, *wavelength=None*)

Bases: pyFAI.geometry.Geometry

This class is an azimuthal integrator based on P. Boesecke's geometry and histogram algorithm by Manolo S. del Rio and V.A Sole

All geometry calculation are done in the Geometry class

main methods are:

```
>>> tth, I = ai.integrate1d(data, npt, unit="2th_deg")
>>> q, I, sigma = ai.integrate1d(data, npt, unit="q_nm^-1", error_model="poisson")
>>> regrouped = ai.integrate2d(data, npt_rad, npt_azim, unit="q_nm^-1")[0]
```

DEFAULT_METHOD = 'splitbbox'

array_from_unit(*shape*, *typ='center'*, *unit=2th_deg*)

Generate an array of position in different dimentions (R, Q, 2Theta)

Parameters

- **shape** (*ndarray.shape*) – shape of the expected array
- **typ** (*str*) – "center", "corner" or "delta"
- **unit** (*pyFAI.units.Enum*) – can be Q, TTH, R for now

Returns R, Q or 2Theta array depending on unit

Return type ndarray

create_mask(*data*, *mask=None*, *dummy=None*, *delta_dummy=None*, *mode='normal'*)
Combines various masks into another one.

Parameters

- **data** (*ndarray*) – input array of data
- **mask** (*ndarray*) – input mask (if none, self.mask is used)
- **dummy** (*float*) – value of dead pixels
- **delta_dumy** – precision of dummy pixels
- **mode** (*str*) – can be "normal" or "numpy" (inverted) or "where" applied to the mask

Returns the new mask

Return type ndarray of bool

This method combine two masks (dynamic mask from *data & dummy* and *mask*) to generate a new one with the 'or' binary operation. One can adjust the level, with the *dummy* and the *delta_dummy* parameter, when you consider the *data* values needs to be masked out.

This method can work in two different *mode*:

•"normal": False for valid pixels, True for bad pixels

•"numpy": True for valid pixels, false for others

This method tries to accomodate various types of masks (like valid=0 & masked=-1, ...) and guesses if an input mask needs to be inverted.

dark_correction(*data*, *dark=None*)
Correct for Dark-current effects. If dark is not defined, correct for a dark set by "set_darkfiles"

Parameters

- **data** – input ndarray with the image
- **dark** – ndarray with dark noise or None

Returns 2tuple: corrected_data, dark_actually used (or None)

darkcurrent

empty

flat_correction(*data*, *flat=None*)
Correct for flat field. If flat is not defined, correct for a flat set by "set_flatfiles"

Parameters

- **data** – input ndarray with the image
- **dark** – ndarray with dark noise or None

Returns 2tuple: corrected_data, flat_actually used (or None)

flatfield

get_darkcurrent()

get_empty()

get_flatfield()

integrate1d(*data*, *npt*, *filename=None*, *correctSolidAngle=True*, *variance=None*, *error_model=None*, *radial_range=None*, *azimuth_range=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*, *method='lut'*, *unit=q_nm^-1*, *safe=True*, *normalization_factor=None*, *block_size=32*, *profile=False*)
Calculate the azimuthal integrated Saxs curve in q(nm^-1) by default

Multi algorithm implementation (tries to be bullet proof), suitable for SAXS, WAXS, ... and much more

Parameters

- **data** (*ndarray*) – 2D array from the Detector/CCD camera
- **npt** (*int*) – number of points in the output pattern
- **filename** (*str*) – output filename in 2/3 column ascii format
- **correctSolidAngle** (*bool*) – correct for solid angle of each pixel if True
- **variance** (*ndarray*) – array containing the variance of the data. If not available, no error propagation is done
- **error_model** (*str*) – When the variance is unknown, an error model can be given: "poisson" (variance = I), "azimuthal" (variance = (I-<I>)^2)
- **radial_range** ((*float, float), optional*) – The lower and upper range of the radial unit. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **azimuth_range** ((*float, float), optional*) – The lower and upper range of the azimuthal angle in degree. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **mask** (*ndarray*) – array (same size as image) with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor between -1 (vertical) and +1 (horizontal). 0 for circular polarization or random, None for no correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image
- **method** (*str*) – can be "numpy", "cython", "BBox" or "splitpixel", "lut", "csr", "nosplit_csr", "full_csr", "lut_ocl" and "csr_ocl" if you want to go on GPU. To Specify the device: "csr_ocl_1,2"
- **unit** (*pyFAI.units.Enum*) – Output units, can be "q_nm^-1", "q_A^-1", "2th_deg", "2th_rad", "r_mm" for now
- **safe** (*bool*) – Do some extra checks to ensure LUT/CSR is still valid. False is faster.
- **normalization_factor** (*float*) – Value of a normalization monitor

Returns q/2th/r bins center positions and regrouped intensity (and error array if variance or variance model provided).

Return type 2 or 3-tuple of ndarrays

integrate2d(*data*, *npt_rad*, *npt_azim=360*, *filename=None*, *correctSolidAngle=True*, *variance=None*, *error_model=None*, *radial_range=None*, *azimuth_range=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*, *method='bbox'*, *unit=q_nm^-1*, *safe=True*, *normalization_factor=None*)

Calculate the azimuthal regrouped 2d image in q(nm^-1)/chi(deg) by default

Multi algorithm implementation (tries to be bullet proof)

Parameters

- **data** (*ndarray*) – 2D array from the Detector/CCD camera
- **npt_rad** (*int*) – number of points in the radial direction

- **npt_azim** (*int*) – number of points in the azimuthal direction
- **filename** (*str*) – output image (as edf format)
- **correctSolidAngle** (*bool*) – correct for solid angle of each pixel if True
- **variance** (*ndarray*) – array containing the variance of the data. If not available, no error propagation is done
- **error_model** (*str*) – When the variance is unknown, an error model can be given: "poisson" (variance = I), "azimuthal" (variance = (I-<I>)^2)
- **radial_range** ((*float, float), optional*) – The lower and upper range of the radial unit. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **azimuth_range** ((*float, float), optional*) – The lower and upper range of the azimuthal angle in degree. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **mask** (*ndarray*) – array (same size as image) with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor between -1 (vertical) and +1 (horizontal). 0 for circular polarization or random, None for no correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image
- **method** (*str*) – can be "numpy", "cython", "BBox" or "splitpixel", "lut", "csr; "lut_ocl" and "csr_ocl" if you want to go on GPU. To Specify the device: "csr_ocl_1,2"
- **unit** (*pyFAI.units.Enum*) – Output units, can be "q_nm^-1", "q_A^-1", "2th_deg", "2th_rad", "r_mm" for now
- **safe** (*bool*) – Do some extra checks to ensure LUT is still valid. False is faster.
- **normalization_factor** (*float*) – Value of a normalization monitor

Returns azimuthaly regrouped intensity, q/2theta/r pos. and chi pos.

Return type 3-tuple of ndarrays (2d, 1d, 1d)

makeHeaders(*hdr='#'*, *dark=None*, *flat=None*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

- **hdr** (*str*) – string used as comment in the header
- **dark** – save the darks filenames (default: no)
- **flat** – save the flat filenames (default: no)
- **polarization_factor** (*float*) – the polarization factor

Returns the header

Return type str

reset()

Reset azimuthal integrator in addition to other arrays.

save1D(*filename*, *dim1*, *I*, *error=None*, *dim1_unit=2th_deg*, *dark=None*, *flat=None*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

- **filename** (*str*) – the filename used to save the 1D integration
- **dim1** (*numpy.ndarray*) – the x coordinates of the integrated curve
- **I** (*numpy.mdarray*) – The integrated intensity
- **error** (*numpy.ndarray or None*) – the error bar for each intensity
- **dim1_unit** (*pyFAI.units.Unit*) – the unit of the dim1 array
- **dark** – save the darks filenames (default: no)
- **flat** – save the flat filenames (default: no)
- **polarization_factor** (*float*) – the polarization factor
- **normalization_factor** (*float*) – the monitor value

This method save the result of a 1D integration.

save2D (*filename*, *I*, *dim1*, *dim2*, *error=None*, *dim1_unit=2th_deg*, *dark=None*, *flat=None*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

- **filename** (*str*) – the filename used to save the 2D histogram
- **dim1** (*numpy.ndarray*) – the 1st coordinates of the histogram
- **dim1** – the 2nd coordinates of the histogram
- **I** (*numpy.mdarray*) – The integrated intensity
- **error** (*numpy.ndarray or None*) – the error bar for each intensity
- **dim1_unit** (*pyFAI.units.Unit*) – the unit of the dim1 array
- **dark** – save the darks filenames (default: no)
- **flat** – save the flat filenames (default: no)
- **polarization_factor** (*float*) – the polarization factor
- **normalization_factor** (*float*) – the monitor value

This method save the result of a 2D integration.

saxs (**arg*, ***kw*)

decorator that deprecates the use of a function

separate (*data*, *npt_rad=1024*, *npt_azim=512*, *unit='2th_deg'*, *percentile=50*, *mask=None*, *restore_mask=True*)

Separate bragg signal from powder/amorphous signal using azimuthal integration, median filering and projected back before subtraction.

Parameters

- **data** – input image as numpy array
- **npt_rad** – number of radial points
- **npt_azim** – number of azimuthal points
- **unit** – unit to be used for integration
- **percentile** – which percentile use for cutting out
- **mask** – masked out pixels array
- **restore_mask** – masked pixels have the same value as input data provided

Returns bragg, amorphous

set_darkcurrent (*dark*)

set_darkfiles (*files=None*, *method='mean'*)

Parameters

- **files** (*str or list(str) or None*) – file(s) used to compute the dark.
- **method** (*str*) – method used to compute the dark, “mean” or “median”

Set the dark current from one or mutliple files, avaraged according to the method provided

set_empty(*value*)

set_flatfield(*flat*)

set_flatfiles(*files*, *method='mean'*)

Parameters

- **files** (*str or list(str) or None*) – file(s) used to compute the dark.
- **method** (*str*) – method used to compute the dark, “mean” or “median”

Set the flat field from one or mutliple files, averaged according to the method provided

setup_CSR(*shape*, *npt*, *mask=None*, *pos0_range=None*, *pos1_range=None*, *mask_checksum=None*, *unit=2th_deg*, *split='bbox'*)

Prepare a look-up-table

Parameters

- **shape** (*(int, int)*) – shape of the dataset
- **npt** (*int or (int, int)*) – number of points in the the output pattern
- **mask** (*ndarray*) – array with masked pixel (1=masked)
- **pos0_range** (*(float, float)*) – range in radial dimension
- **pos1_range** (*(float, float)*) – range in azimuthal dimension
- **mask_checksum** (*int (or anything else ...)*) – checksum of the mask buffer
- **unit** (*pyFAI.units.Enum*) – use to propagate the LUT object for further checkings
- **split** – Splitting scheme: valid options are “no”, “bbox”, “full”

This method is called when a look-up table needs to be set-up. The *shape* parameter, correspond to the shape of the original datatset. It is possible to customize the number of point of the output histogram with the *npt* parameter which can be either an integer for an 1D integration or a 2-tuple of integer in case of a 2D integration. The LUT will have a different shape: (npt, lut_max_size), the later parameter being calculated during the instanciation of the splitBBoxLUT class.

It is possible to prepare the LUT with a predefine *mask*. This operation can speedup the computation of the later integrations. Instead of applying the patch on the dataset, it is taken into account during the histogram computation. If provided the *mask_checksum* prevent the re-calculation of the mask. When the mask changes, its checksum is used to reset (or not) the LUT (which is a very time consuming operation !)

It is also possible to restrain the range of the 1D or 2D pattern with the *pos1_range* and *pos2_range*.

The *unit* parameter is just propagated to the LUT integrator for further checkings: The aim is to prevent an integration to be performed in 2th-space when the LUT was setup in q space.

setup_LUT(*shape*, *npt*, *mask=None*, *pos0_range=None*, *pos1_range=None*, *mask_checksum=None*, *unit=2th_deg*)

Prepare a look-up-table

Parameters

- **shape** (*(int, int)*) – shape of the dataset
- **npt** (*int or (int, int)*) – number of points in the the output pattern
- **mask** (*ndarray*) – array with masked pixel (1=masked)

- **pos0_range** ((*float, float)*) – range in radial dimension
- **pos1_range** ((*float, float)*) – range in azimuthal dimension
- **mask_checksum** (*int (or anything else ...)*) – checksum of the mask buffer
- **unit** (*pyFAI.units.Enum*) – use to propagate the LUT object for further checkings

This method is called when a look-up table needs to be set-up. The *shape* parameter, correspond to the shape of the original datatset. It is possible to customize the number of point of the output histogram with the *npt* parameter which can be either an integer for an 1D integration or a 2-tuple of integer in case of a 2D integration. The LUT will have a different shape: (npt, lut_max_size), the later parameter being calculated during the instanciation of the splitBBoxLUT class.

It is possible to prepare the LUT with a predefine *mask*. This operation can speedup the computation of the later integrations. Instead of applying the patch on the dataset, it is taken into account during the histogram computation. If provided the *mask_checksum* prevent the re-calculation of the mask. When the mask changes, its checksum is used to reset (or not) the LUT (which is a very time consuming operation !)

It is also possible to restrain the range of the 1D or 2D pattern with the *pos1_range* and *pos2_range*.

The *unit* parameter is just propagated to the LUT integrator for further checkings: The aim is to prevent an integration to be performed in 2th-space when the LUT was setup in q space.

xrpd(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd2(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd2_histogram(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd2_numpy(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd2_splitBBox(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd2_splitPixel(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_CSR_OCL(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_LUT(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_LUT_OCL(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_OpenCL(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_cython(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_numpy(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_splitBBox(**arg*, ***kw*)
decorator that deprecates the use of a function

xrpd_splitPixel(**arg*, ***kw*)
decorator that deprecates the use of a function

5.3 integrate_widget Module

class pyFAI.integrate_widget.**AIWidget**(*input_data=None*)
Bases: PyQt4.QtGui.QWidget

detector_changed()

die()

dump(*filename='.azimint.json'*)
Dump the status of the current widget to a file in JSON

Parameters **filename** (*str*) – path where to save the config

help()

openCL_changed()

platform_changed()

proceed()

restore(*filename='.azimint.json'*)
restore from JSON file the status of the current widget

Parameters **filename** (*str*) – path where the config was saved

select_darkcurrent()

select_flatfield()

select_maskfile()

select_ponifile()

select_splinefile()

setBackgroundImage(*dark=None*)
PyMca Plugin specific

Parameters **dark** – 2D array with the dark-current

setSelectionMask(*mask=None*)
PyMca Plugin specific

Parameters **mask** – 2D array with the masked region

setStackDataObject(*stack*, *stack_name=None*)

set_ai()

set_input_data(*stack*, *stack_name=None*)

set_ponifile(*ponifile=None*)

set_validators()
Set all validators for text entries

class pyFAI.integrate_widget.**Browser**(*default_url='http://google.com'*)
Bases: PyQt4.QtGui.QMainWindow

browse()
Make a web browse on a specific url and show the page on the Webview widget.

5.4 geometry Module

class pyFAI.geometry.**Geometry**(*dist=1*, *poni1=0*, *poni2=0*, *rot1=0*, *rot2=0*, *rot3=0*, *pixel1=None*, *pixel2=None*, *splineFile=None*, *detector=None*, *wavelength=None*)

Bases: object

This class is an azimuthal integrator based on P. Boesecke's geometry and histogram algorithm by Manolo S. del Rio and V.A Sole

Detector is assumed to be corrected from "raster orientation" effect. It is not addressed here but rather in the Detector object or at read time. Considering there is no tilt:

- Detector fast dimension (dim2) is supposed to be horizontal (dimension X of the image)
- Detector slow dimension (dim1) is supposed to be vertical, upwards (dimension Y of the image)
- The third dimension is chose such as the referential is orthonormal, so dim3 is along incoming X-ray beam

Axis 1 is along first dimension of detector (when not tilted), this is the slow dimension of the image array in C or Y x1={1,0,0}

Axis 2 is along second dimension of detector (when not tilted), this is the fast dimension of the image in C or X x2={0,1,0}

Axis 3 is along the incident X-Ray beam x3={0,0,1}

We define the 3 rotation around axis 1, 2 and 3:

rotM1 = RotationMatrix[rot1,x1] = {{1,0,0},{0,cos[rot1],-sin[rot1]},{0,sin[rot1],cos[rot1]}} rotM2 = RotationMatrix[rot2,x2] = {{cos[rot2],0,sin[rot2]},{0,1,0},{-sin[rot2],0,cos[rot2]}} rotM3 = RotationMatrix[rot3,x3] = {{cos[rot3],-sin[rot3],0},{sin[rot3],cos[rot3],0},{0,0,1}}

Rotations of the detector are applied first Rot around axis 1, then axis 2 and finally around axis 3:

R = rotM3.rotM2.rotM1

R = {{cos[rot2] cos[rot3],cos[rot3] sin[rot1] sin[rot2]-cos[rot1] sin[rot3],cos[rot1] cos[rot3] sin[rot2]+sin[rot1] sin[rot3] {cos[rot2] sin[rot3],cos[rot1] cos[rot3]+sin[rot1] sin[rot2] sin[rot3],-cos[rot3] sin[rot1]+cos[rot1] sin[rot2] sin[rot3]}, {-sin[rot2],cos[rot2] sin[rot1],cos[rot1] cos[rot2]}}

In Python notation:

R.x1 = [cos(rot2)*cos(rot3),cos(rot2)*sin(rot3),-sin(rot2)]

R.x2 = [cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3),cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3), cos(rot2)*sin(rot1)]

R.x3 = [cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3),-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3), cos(rot1)*cos(rot2)]

- Coordinates of the Point of Normal Incidence:

  PONI = R.{0,0,L}

  PONI = [L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3)), L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)),L*cos(rot1)*cos(rot2)]

- Any pixel on detector plan at coordinate (d1, d2) in meters. Detector is at z=L

  P={d1,d2,L}

  R.P = [t1, t2, t3] t1 = R.P.x1 = d1*cos(rot2)*cos(rot3) + d2*(cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3)) + L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3)) t2 = R.P.x2 = d1*cos(rot2)*sin(rot3) + d2*(cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3)) + L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)) t3 = R.P.x3 = d2*cos(rot2)*sin(rot1) - d1*sin(rot2) + L*cos(rot1)*cos(rot2)

- Distance sample (origin) to detector point (d1,d2)

|R.P| = sqrt(pow(Abs(L*cos(rot1)*cos(rot2) + d2*cos(rot2)*sin(rot1) - d1*sin(rot2)),2) +
pow(Abs(d1*cos(rot2)*cos(rot3) + d2*(cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3)) + L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3))),2) + pow(Abs(d1*cos(rot2)*sin(rot3) + L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)) + d2*(cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3))),2))

•cos(2theta) is defined as (R.P component along x3) over the distance from origin to data point **|R.P|**

tth = ArcCos [-(R.P).x3/**|R.P|**]

tth = Arccos((-(L*cos(rot1)*cos(rot2)) - d2*cos(rot2)*sin(rot1) + d1*sin(rot2))/

sqrt(pow(Abs(L*cos(rot1)*cos(rot2) + d2*cos(rot2)*sin(rot1) - d1*sin(rot2)),2) +

pow(Abs(d1*cos(rot2)*cos(rot3) + d2*(cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3)) +

L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3))),2) + pow(Abs(d1*cos(rot2)*sin(rot3) + L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)) +

d2*(cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3))),2)))

•tan(2theta) is defined as sqrt(t1**2 + t2**2) / t3

tth = ArcTan2 [sqrt(t1**2 + t2**2) , t3]

Getting 2theta from it's tangeant seems both more precise (around beam stop very far from sample) and faster by about 25% Currently there is a swich in the method to follow one path or the other.

•Tangeant of angle chi is defined as (R.P component along x1) over (R.P component along x2). Arctan2 should be used in actual calculation

chi = ArcTan[((R.P).x1) / ((R.P).x2)]

chi = ArcTan2(d1*cos(rot2)*cos(rot3) + d2*(cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3)) +

L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3)),

d1*cos(rot2)*sin(rot3) + L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)) +
d2*(cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3)))

calc_pos_zyx(*d0=None*, *d1=None*, *d2=None*, *param=None*)

Allows you to calculate the position of a set of points in space in the sample re

Parameters

- **d0** – altitude on the point compared to the detector (i.e. z)
- **d1** – position on the detector along the slow dimention (i.e. y)
- **d2** – position on the detector along the fastest dimention (i.e. x)

:return zyx array, so 3D array with dim0=along the beam, dim1=along slowest dimension dim2=along fastest dimension unless rotations are too large

calc_transmission(*t0*, *shape=None*)

Defines the absorption correction for a phosphor screen or a scintillator from t0, the normal transmission of the screen.

Icor = Iobs(1-t0)/(1-exp(ln(t0)/cos(incidence))) 1-exp(ln(t0)/cos(incidence)

let t = ———————— 1 - t0

See reference on: J. Appl. Cryst. (2002). 35, 356–359 G. Wu et al. CCD phosphor

Parameters

- **t0** – value of the normal transmission (from 0 to 1)
- **shape** – shape of the array

Returns actual

calcfrom1d(*tth*, *I*, *shape=None*, *mask=None*, *dim1_unit=2th_deg*, *correctSolidAngle=True*)
Computes a 2D image from a 1D integrated profile

Parameters

- **tth** – 1D array with 2theta in degrees
- **I** – scattering intensity

Returns 2D image reconstructed

chi(*d1*, *d2*, *path='cython'*)
Calculate the chi (azimuthal angle) for the centre of a pixel at coordinate d1,d2 which in the lab ref has coordinate:

X1 = p1*cos(rot2)*cos(rot3) + p2*(cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3)) - L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3)) X2 = p1*cos(rot2)*sin(rot3) - L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)) + p2*(cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3)) X3 = -(L*cos(rot1)*cos(rot2)) + p2*cos(rot2)*sin(rot1) - p1*sin(rot2) hence tan(Chi) = X2 / X1

Parameters

- **d1** (*float or array of them*) – pixel coordinate along the 1st dimention (C convention)
- **d2** (*float or array of them*) – pixel coordinate along the 2nd dimention (C convention)
- **path** – can be "tan" (i.e via numpy) or "cython"

Returns chi, the azimuthal angle in rad

chiArray(*shape*)
Generate an array of the given shape with chi(i,j) (azimuthal angle) for all elements.

Parameters **shape** (*ndarray.shape*) – the shape of the chi array

Returns the chi array

Return type ndarray

chi_corner(*d1*, *d2*)
Calculate the chi (azimuthal angle) for the corner of a pixel at coordinate d1,d2 which in the lab ref has coordinate:

X1 = p1*cos(rot2)*cos(rot3) + p2*(cos(rot3)*sin(rot1)*sin(rot2) - cos(rot1)*sin(rot3)) - L*(cos(rot1)*cos(rot3)*sin(rot2) + sin(rot1)*sin(rot3)) X2 = p1*cos(rot2)*sin(rot3) - L*(-(cos(rot3)*sin(rot1)) + cos(rot1)*sin(rot2)*sin(rot3)) + p2*(cos(rot1)*cos(rot3) + sin(rot1)*sin(rot2)*sin(rot3)) X3 = -(L*cos(rot1)*cos(rot2)) + p2*cos(rot2)*sin(rot1) - p1*sin(rot2) hence tan(Chi) = X2 / X1

Parameters

- **d1** (*float or array of them*) – pixel coordinate along the 1st dimention (C convention)
- **d2** (*float or array of them*) – pixel coordinate along the 2nd dimention (C convention)

Returns chi, the azimuthal angle in rad

chia
chi array in cache

cornerArray(*shape*)
Generate a 3D array of the given shape with (i,j) (radial angle 2th, azimuthal angle chi) for all elements.

Parameters **shape** (*ndarray.shape*) – expected shape

Returns 3d array with shape=(*shape,2) the two elements are (radial angle 2th, azimuthal angle chi)

cornerQArray(*shape*)
Generate a 3D array of the given shape with (i,j) (azimuthal angle) for all elements.

cornerRArray(*shape*)
Generate a 3D array of the given shape with (i,j) (azimuthal angle) for all elements.

correct_SA_spline

cosIncidance(*d1*, *d2*, *path='cython'*)
Calculate the incidence angle (alpha) for current pixels (P). The poni being the point of normal incidence, it's incidence angle is ${alpha} = 0$ hence $cos({alpha}) = 1$

Parameters

- **d1** – 1d or 2d set of points in pixel coord
- **d2** – 1d or 2d set of points in pixel coord

Returns cosine of the incidence angle

del_chia()

del_dssa()

del_qa()

del_ttha()

delta2Theta(*shape*)
Generate a 3D array of the given shape with (i,j) with the max distance between the center and any corner in 2 theta

Parameters **shape** – The shape of the detector array: 2-tuple of integer

Returns 2D-array containing the max delta angle between a pixel center and any corner in 2theta-angle (rad)

deltaChi(*shape*)
Generate a 3D array of the given shape with (i,j) with the max distance between the center and any corner in chi-angle (rad)

Parameters **shape** – The shape of the detector array: 2-tuple of integer

Returns 2D-array containing the max delta angle between a pixel center and any corner in chi-angle (rad)

deltaQ(*shape*)
Generate a 2D array of the given shape with (i,j) with the max distance between the center and any corner in q_vector unit (nm^-1)

Parameters **shape** – The shape of the detector array: 2-tuple of integer

Returns array 2D containing the max delta Q between a pixel center and any corner in q_vector unit (nm^-1)

deltaR(*shape*)
Generate a 2D array of the given shape with (i,j) with the max distance between the center and any corner in radius unit (mm)

Parameters **shape** – The shape of the detector array: 2-tuple of integer

Returns array 2D containing the max delta Q between a pixel center and any corner in q_vector unit (nm^-1)

diffSolidAngle(*d1*, *d2*)
Calculate the solid angle of the current pixels (P) versus the PONI (C)

Omega(P) A cos(a) SC^2 3 SC^3

dOmega = ——— = ——— x ——— = cos (a) = —— Omega(C) SP^2 A cos(0) SP^3

cos(a) = SC/SP

Parameters

- **d1** – 1d or 2d set of points
- **d2** – 1d or 2d set of points (same size&shape as d1)

Returns solid angle correction array

dist

dssa
solid angle array in cache

getFit2D()
Export geometry setup with the geometry of Fit2D

Returns dict with parameters compatible with fit2D geometry

getPyFAI()
Export geometry setup with the geometry of PyFAI

Returns dict with the parameter-set of the PyFAI geometry

getSPD()
get the SPD like parameter set: For geometry description see Peter Boesecke J.Appl.Cryst.(2007).40, s423–s427

Basically the main difference with pyFAI is the order of the axis which are flipped

Returns dictionnary with those parameters: SampleDistance: distance from sample to detector at the PONI (orthogonal projection) Center_1, pixel position of the PONI along fastest axis Center_2: pixel position of the PONI along slowest axis Rot_1: rotation around the fastest axis (x) Rot_2: rotation around the slowest axis (y) Rot_3: rotation around the axis ORTHOGONAL to the detector plan PSize_1: pixel size in meter along the fastest dimention PSize_2: pixel size in meter along the slowst dimention splineFile: name of the file containing the spline BSize_1: pixel binning factor along the fastest dimention BSize_2: pixel binning factor along the slowst dimention WaveLength: wavelength used in meter

get_chia()

get_correct_solid_angle_for_spline()

get_dist()

get_dssa()

get_mask()

get_maskfile()

get_pixel1()

get_pixel2()

get_poni1()

get_poni2()

get_qa()

get_rot1()

get_rot2()

get_rot3()

get_spline()

get_splineFile()

get_ttha()

get_wavelength()

load(*filename*)
Load the refined parameters from a file.

Parameters **filename** (*string*) – name of the file to load

mask

maskfile

oversampleArray(*myarray*)

pixel1

pixel2

polarization(*shape=None*, *factor=None*, *axis_offset=0*)
Calculate the polarization correction accoding to the polarization factor:

•If the polarization factor is None, the correction is not applied (returns 1)

•If the polarization factor is 0 (circular polarization), the correction correspond to (1+(cos2θ)^2)/2

•If the polarization factor is 1 (linear horizontal polarization), there is no correction in the vertical plane and a node at 2th=90, chi=0

•If the polarization factor is -1 (linear vertical polarization), there is no correction in the horizontal plane and a node at 2th=90, chi=90

•If the polarization is elliptical, the polarization factor varies between -1 and +1.

The axis_offset parameter allows correction for the misalignement of the polarization plane (or ellipse main axis) and the the detector's X axis.

Parameters

- **factor** – (Ih-Iv)/(Ih+Iv): varies between 0 (no polarization) and 1 (where division by 0 could occure at 2th=90, chi=0)
- **axis_offset** – Angle between the polarization main axis and detector X direction (in radians !!!)

Returns 2D array with polarization correction array (intensity/polarisation)

poni1

poni2

qArray(*shape*)
Generate an array of the given shape with q(i,j) for all elements.

qCornerFunct(*d1*, *d2*)
Calculate the q_vector for any pixel corner (in nm^-1)

qFunction(*d1*, *d2*, *param=None*, *path='cython'*)
Calculates the q value for the center of a given pixel (or set of pixels) in nm-1

q = 4pi/lambda sin(2theta / 2)

Parameters

- **d1** (*scalar or array of scalar*) – position(s) in pixel in first dimension (c order)
- **d2** (*scalar or array of scalar*) – position(s) in pixel in second dimension (c order)

Returns q in in nm^(-1)

Return type float or array of floats.

qa
Q array in cache

rArray(*shape*)
Generate an array of the given shape with r(i,j) for all elements; r in mm.

Parameters **shape** – expected shape

Returns 2d array of the given shape with radius in mm from beam stop.

rCornerFunct(*d1*, *d2*)
Calculate the radius array for any pixel corner (in mm)

rFunction(*d1*, *d2*, *param=None*, *path='numpy'*)
Calculates the radius value for the center of a given pixel (or set of pixels) in mm

r = direct_distance * tan(2theta)

Parameters

- **d1** (*scalar or array of scalar*) – position(s) in pixel in first dimension (c order)
- **d2** (*scalar or array of scalar*) – position(s) in pixel in second dimension (c order)

Returns r in in mm

Return type float or array of floats.

read(*filename*)
Load the refined parameters from a file.

Parameters **filename** (*string*) – name of the file to load

reset()
reset most arrays that are cached: used when a parameter changes.

rot1

rot2

rot3

save(*filename*)
Save the refined parameters.

Parameters **filename** (*string*) – name of the file where to save the parameters

setChiDiscAtPi()
Set the position of the discontinuity of the chi axis between -pi and +pi. This is the default behavour

setChiDiscAtZero()
Set the position of the discontinuity of the chi axis between 0 and 2pi. By default it is between pi and -pi

setFit2D(*directDist*, *centerX*, *centerY*, *tilt=0.0*, *tiltPlanRotation=0.0*, *pixelX=None*, *pixelY=None*, *splineFile=None*)
Set the Fit2D-like parameter set: For geometry description see HPR 1996 (14) pp-240

Warning: Fit2D flips automatically images depending on their file-format. By reverse engineering we noticed this behavour for Tiff and Mar345 images (at least). To obtaine correct result you will have to flip images using numpy.flipud.

Parameters

- **direct** – direct distance from sample to detector along the incident beam (in millimeter as in fit2d)
- **tilt** – tilt in degrees
- **tiltPlanRotation** – Rotation (in degrees) of the tilt plan arround the Z-detector axis * 0deg -> Y does not move, +X goes to Z<0 * 90deg -> X does not move, +Y goes to

Z<0 * 180deg -> Y does not move, +X goes to Z>0 * 270deg -> X does not move, +Y goes to Z>0

- **pixelX,pixelY** – as in fit2d they ar given in micron, not in meter
- **centerY** (*centerX,*) – pixel position of the beam center
- **splineFile** – name of the file containing the spline

setOversampling(*iOversampling*)
set the oversampling factor

setPyFAI(***kwargs*)
set the geometry from a pyFAI-like dict

setSPD(*SampleDistance*, *Center_1*, *Center_2*, *Rot_1=0*, *Rot_2=0*, *Rot_3=0*, *PSize_1=None*, *PSize_2=None*, *splineFile=None*, *BSize_1=1*, *BSize_2=1*, *WaveLength=None*)
Set the SPD like parameter set: For geometry description see Peter Boesecke J.Appl.Cryst.(2007).40, s423–s427

Basically the main difference with pyFAI is the order of the axis which are flipped

Parameters **SampleDistance** – distance from sample to detector at the PONI (orthogonal projection)

:param Center_1, pixel position of the PONI along fastest axis :param Center_2: pixel position of the PONI along slowest axis :param Rot_1: rotation around the fastest axis (x) :param Rot_2: rotation around the slowest axis (y) :param Rot_3: rotation around the axis ORTHOGONAL to the detector plan :param PSize_1: pixel size in meter along the fastest dimention :param PSize_2: pixel size in meter along the slowst dimention :param splineFile: name of the file containing the spline :param BSize_1: pixel binning factor along the fastest dimention :param BSize_2: pixel binning factor along the slowst dimention :param WaveLength: wavelength used

set_chia(*_*)

set_correct_solid_angle_for_spline(*value*)

set_dist(*value*)

set_dssa(*_*)

set_mask(*mask*)

set_maskfile(*maskfile*)

set_pixel1(*pixel1*)

set_pixel2(*pixel2*)

set_poni1(*value*)

set_poni2(*value*)

set_qa(*_*)

set_rot1(*value*)

set_rot2(*value*)

set_rot3(*value*)

set_spline(*spline*)

set_splineFile(*splineFile*)

set_ttha(*_*)

set_wavelength(*value*)

classmethod **sload**(*filename*)
A static method combining the constructor and the loader from a file

Parameters **filename** (*string*) – name of the file to load

Returns instance of Gerometry of AzimuthalIntegrator set-up with the parameter from the file.

solidAngleArray (*shape*, *order=3*, *absolute=False*)
Generate an array for the solid angle correction given the shape of the detector.

solid_angle = cos(incidence)^3

Parameters

- **shape** – shape of the array expected
- **order** – should be 3, power of the formula just obove
- **absolute** – the absolute solid angle is calculated as:

SA = pix1*pix2/dist^2 * cos(incidence)^3

spline

splineFile

tth (*d1*, *d2*, *param=None*, *path='cython'*)
Calculates the 2theta value for the center of a given pixel (or set of pixels)

Parameters

- **d1** (*scalar or array of scalar*) – position(s) in pixel in first dimension (c order)
- **d2** (*scalar or array of scalar*) – position(s) in pixel in second dimension (c order)
- **path** – can be "cos", "tan" or "cython"

Returns 2theta in radians

Return type floar or array of floats.

tth_corner (*d1*, *d2*)
Calculates the 2theta value for the corner of a given pixel (or set of pixels)

Parameters

- **d1** (*scalar or array of scalar*) – position(s) in pixel in first dimension (c order)
- **d2** (*scalar or array of scalar*) – position(s) in pixel in second dimension (c order)

Returns 2theta in radians

Return type floar or array of floats.

ttha
2theta array in cache

twoThetaArray (*shape*)
Generate an array of the given shape with two-theta(i,j) for all elements.

wavelength

write (*filename*)
Save the refined parameters.

Parameters **filename** (*string*) – name of the file where to save the parameters

5.5 geometryRefinement Module

class pyFAI.geometryRefinement.**GeometryRefinement**(*data*, *dist=1*, *poni1=None*, *poni2=None*, *rot1=0*, *rot2=0*, *rot3=0*, *pixel1=None*, *pixel2=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)

Bases: pyFAI.azimuthalIntegrator.AzimuthalIntegrator

anneal(*maxiter=1000000*)

calc_2th(*rings*, *wavelength=None*)

Parameters

- **rings** – indices of the rings. starts at 0 and self.dSpacing should be long enough !!!
- **wavelength** – wavelength in meter

chi2(*param=None*)

chi2_wavelength(*param=None*)

confidence(*with_rot=True*)

Confidence interval obtained from the second derivative of the error function next to its minimum value.

Note the confidence interval increases with the number of points which is "surprizing"

Parameters **with_rot** – if true include rot1 & rot2 in the parameter set.

Returns std_dev, confidence

curve_fit(*with_rot=True*)

Refine the geometry and provide confidence interval Use curve_fit from scipy.optimize to not only refine the geometry (unconstrained fit)

Parameters **with_rot** – include rotation intro error measurment

Returns std_dev, confidence

dist_max

dist_min

get_dist_max()

get_dist_min()

get_poni1_max()

get_poni1_min()

get_poni2_max()

get_poni2_min()

get_rot1_max()

get_rot1_min()

get_rot2_max()

get_rot2_min()

get_rot3_max()

get_rot3_min()

get_wavelength_max()

get_wavelength_min()

guess_poni()
Poni can be guessed by the centroid of the ring with lowest 2Theta

poni1_max

poni1_min

poni2_max

poni2_min

refine1()

refine2(*maxiter=1000000, fix=['wavelength']*)

refine2_wavelength(*maxiter=1000000, fix=['wavelength']*)

residu1(*param*, *d1*, *d2*, *rings*)

residu1_wavelength(*param*, *d1*, *d2*, *rings*)

residu2(*param*, *d1*, *d2*, *rings*)

residu2_wavelength(*param*, *d1*, *d2*, *rings*)

residu2_wavelength_weighted(*param*, *d1*, *d2*, *rings*, *weight*)

residu2_weighted(*param*, *d1*, *d2*, *rings*, *weight*)

roca()
run roca to optimise the parameter set

rot1_max

rot1_min

rot2_max

rot2_min

rot3_max

rot3_min

set_dist_max(*value*)

set_dist_min(*value*)

set_poni1_max(*value*)

set_poni1_min(*value*)

set_poni2_max(*value*)

set_poni2_min(*value*)

set_rot1_max(*value*)

set_rot1_min(*value*)

set_rot2_max(*value*)

set_rot2_min(*value*)

set_rot3_max(*value*)

set_rot3_min(*value*)

set_tolerance(*value=10*)
Set the tolerance for a refinement of the geometry; in percent of the original value

Parameters **value** – Tolerance as a percentage

set_wavelength_max(*value*)

set_wavelength_min(*value*)

simplex (*maxiter=1000000*)

wavelength_max

wavelength_min

5.6 detectors Module

Module containing the description of all detectors with a factory to instanciate them

class pyFAI.detectors.**Basler** (*pixel=3.75e-06*)
Bases: pyFAI.detectors.Detector

Basler camera are simple CCD camara over GigaE

MAX_SHAPE = (966, 1296)

aliases = ['aca1300']

force_pixel = True

class pyFAI.detectors.**Detector** (*pixel1=None*, *pixel2=None*, *splineFile=None*)
Bases: object

Generic class representing a 2D detector

aliases = []

binning

calc_cartesian_positions (*d1=None*, *d2=None*)
Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!!

Parameters

- **d1** (*ndarray (1D or 2D)*) – the Y pixel positions (slow dimension)
- **d2** (*ndarray (1D or 2D)*) – the X pixel positions (fast dimension)

Returns position in meter of the center of each pixels.

Return type ndarray

d1 and d2 must have the same shape, returned array will have the same shape.

calc_mask ()
Detectors with gaps should overwrite this method with something actually calculating the mask!

classmethod factory (*name*, *config=None*)
A kind of factory...

Parameters

- **name** (*str*) – name of a detector
- **config** (*dict or JSON representation of it.*) – configuration of the detector

Returns an instance of the right detector, set-up if possible

Return type pyFAI.detectors.Detector

force_pixel = False

getFit2D ()
Helper method to serialize the description of a detector using the Fit2d units

Returns representation of the detector easy to serialize

Return type dict

getPyFAI()
Helper method to serialize the description of a detector using the pyFAI way with everything in S.I units.

Returns representation of the detector easy to serialize

Return type dict

get_binning()

get_mask()

get_maskfile()

get_name()
Get a meaningful name for detector

get_pixel1()

get_pixel2()

get_pixel_corners()
Calculate the position of the corner of the pixels

This should be overwritten by class representing non-contiguous detector (Xpad, ...)

Returns 4D array containing: pixel index (slow dimension) pixel index (fast dimension) corner index (A, B, C or D), triangles or hexagons can be handled the same way vertex position (z,y,x)

get_splineFile()

guess_binning(*data*)
Guess the binning/mode depending on the image shape :param data: 2-tuple with the shape of the image or the image with a .shape attribute.

mask

maskfile

name
Get a meaningful name for detector

pixel1

pixel2

registry = {'imxpads70': <class 'pyFAI.detectors.ImXPadS70'>, 'rayonix_mx225': <class 'pyFAI.detectors.Rayoni

save(*filename*)
Saves the detector description into a NeXus file, adapted from: http://download.nexusformat.org/sphinx/classes/base_classes/NXdetector.html Main differences:

•differentiate pixel center from pixel corner offsets

•store all offsets are ndarray according to slow/fast dimention (not x, y)

Parameters **filename** – name of the file on the disc

setFit2D(***kwarg*)
Twin method of getFit2D: setup a detector instance according to a description

Parameters **kwarg** – dictionary containing pixel1, pixel2 and splineFile

setPyFAI(***kwarg*)
Twin method of getPyFAI: setup a detector instance according to a description

Parameters **kwarg** – dictionary containing detector, pixel1, pixel2 and splineFile

set_binning(*bin_size=(1, 1)*)
Set the "binning" of the detector,

Parameters **bin_size** (*(int, int)*) – binning as integer or tuple of integers.

set_config(*config*)
Sets the configuration of the detector. This implies: - Orientation: integers - Binning - ROI

The configuration is either a python dictionary or a JSON string or a file containing this JSON configuration

keys in that dictionary are : "orientation": integers from 0 to 7 "binning": integer or 2-tuple of integers. If only one integer is provided, "offset": coordinate (in pixels) of the start of the detector

set_dx(*dx=None*)
set the pixel-wise displacement along X (dim2):

set_dy(*dy=None*)
set the pixel-wise displacement along Y (dim1):

set_mask(*mask*)

set_maskfile(*maskfile*)

set_pixel1(*value*)

set_pixel2(*value*)

set_splineFile(*splineFile*)

splineFile

uniform_pixel = True

class pyFAI.detectors.**DetectorMeta**(*name*, *bases*, *dct*)
Bases: type

Metaclass used to register all detector classes inheriting from Detector

class pyFAI.detectors.**Dexela2923**(*pixel1=7.5e-05*, *pixel2=7.5e-05*)
Bases: pyFAI.detectors.Detector

Dexela CMOS family detector

MAX_SHAPE = (3888, 3072)

aliases = ['Dexela 2923']

force_pixel = True

class pyFAI.detectors.**Eiger**(*pixel1=7.5e-05*, *pixel2=7.5e-05*)
Bases: pyFAI.detectors.Detector

Eiger detector: generic description containing mask algorithm

MODULE_GAP = (37, 10)

MODULE_SIZE = (514, 1030)

calc_cartesian_positions(*d1=None*, *d2=None*)
Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!!

Parameters

- **d1** (*ndarray (1D or 2D)*) – the Y pixel positions (slow dimension)
- **d2** (*ndarray (1D or 2D)*) – the X pixel positions (fast dimension)

Returns position in meter of the center of each pixels.

Return type ndarray

d1 and d2 must have the same shape, returned array will have the same shape.

calc_mask()
Returns a generic mask for Pilatus detectors...

force_pixel = True

class pyFAI.detectors.**Eiger16M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 16M detector

MAX_SHAPE = (4371, 4150)

class pyFAI.detectors.**Eiger1M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 1M detector

MAX_SHAPE = (1065, 1030)

class pyFAI.detectors.**Eiger4M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 4M detector

MAX_SHAPE = (2167, 2070)

class pyFAI.detectors.**Eiger9M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 9M detector

MAX_SHAPE = (3269, 3110)

class pyFAI.detectors.**FReLoN**(*splineFile=None*)
Bases: pyFAI.detectors.Detector

FReLoN detector: The spline is mandatory to correct for geometric distortion of the taper

TODO: create automatically a mask that removes pixels out of the "valid reagion"

calc_mask()
Returns a generic mask for Frelon detectors... All pixels which (center) turns to be out of the valid region are by default discarded

class pyFAI.detectors.**Fairchild**(*pixel1=1.5e-05*, *pixel2=1.5e-05*)
Bases: pyFAI.detectors.Detector

Fairchild Condor 486:90 detector

MAX_SHAPE = (4096, 4096)

aliases = ['Fairchild', 'Condor', 'Fairchild Condor 486:90']

force_pixel = True

uniform_pixel = True

class pyFAI.detectors.**ImXPadS10**(*pixel1=0.00013*, *pixel2=0.00013*)
Bases: pyFAI.detectors.Detector

ImXPad detector: ImXPad s10 detector with 1x1modules

BORDER_SIZE_RELATIVE = 2.5

MAX_SHAPE = (120, 80)

MODULE_SIZE = (120, 80)

PIXEL_SIZE = (0.00013, 0.00013)

aliases = ['Imxpad S10']

calc_cartesian_positions(*d1=None*, *d2=None*)
Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!!

Parameters

- **d1** (*ndarray (1D or 2D)*) – the Y pixel positions (slow dimension)
- **d2** (*ndarray (1D or 2D)*) – the X pixel positions (fast dimension)

Returns position in meter of the center of each pixels.

Return type ndarray

d1 and d2 must have the same shape, returned array will have the same shape.

calc_mask()
Calculate the mask

calc_pixels_edges()
Calculate the position of the pixel edges

force_pixel = True

uniform_pixel = False

class pyFAI.detectors.**ImXPadS140**(*pixel1=0.00013*, *pixel2=0.00013*)
Bases: pyFAI.detectors.ImXPadS10

ImXPad detector: ImXPad s140 detector with 2x7modules

BORDER_PIXEL_SIZE_RELATIVE = 2.5

MAX_SHAPE = (240, 560)

MODULE_SIZE = (120, 80)

PIXEL_SIZE = (0.00013, 0.00013)

aliases = ['Imxpad S140']

force_pixel = True

class pyFAI.detectors.**ImXPadS70**(*pixel1=0.00013*, *pixel2=0.00013*)
Bases: pyFAI.detectors.ImXPadS10

ImXPad detector: ImXPad s70 detector with 1x7modules

BORDER_SIZE_RELATIVE = 2.5

MAX_SHAPE = (120, 560)

MODULE_SIZE = (120, 80)

PIXEL_EDGES = None

PIXEL_SIZE = (0.00013, 0.00013)

aliases = ['Imxpad S70']

force_pixel = True

class pyFAI.detectors.**Mar345**(*pixel1=0.0001*, *pixel2=0.0001*)
Bases: pyFAI.detectors.Detector

Mar345 Imaging plate detector

In this detector, pixels are always square The valid image size are 2300, 2000, 1600, 1200, 3450, 3000, 2400, 1800

MAX_SHAPE = (3450, 3450)

VALID_SIZE = {2000: 0.00015, 1600: 0.00015, 3000: 0.0001, 2400: 0.0001, 3450: 0.0001, 1200: 0.00015, 2300: 0.00015

aliases = ['MAR 345', 'Mar3450']

calc_mask()

force_pixel = True

guess_binning(*data*)
Guess the binning/mode depending on the image shape :param data: 2-tuple with the shape of the image or the image with a .shape attribute.

class pyFAI.detectors.**NexusDetector**(*filename=None*)
Bases: pyFAI.detectors.Detector

Class representing a 2D detector loaded from a NeXus file

calc_cartesian_positions(*d1=None*, *d2=None*, *center=True*, *use_cython=True*)
Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!! Adapted to Nexus detector definition

Parameters

- **d1** (*ndarray (1D or 2D)*) – the Y pixel positions (slow dimension)
- **d2** (*ndarray (1D or 2D)*) – the X pixel positions (fast dimension)
- **center** – retrieve the coordinate of the center of the pixel
- **use_cython** – set to False to test Python implementeation

Returns position in meter of the center of each pixels.

Return type ndarray

d1 and d2 must have the same shape, returned array will have the same shape.

get_pixel_corners(*use_cython=True*)
Calculate the position of the corner of the pixels

This should be overwritten by class representing non-contiguous detector (Xpad, ...)

Returns 4D array containing: pixel index (slow dimension) pixel index (fast dimension) corner index (A, B, C or D), triangles or hexagons can be handled the same way vertex position (z,y,x)

load(*filename*)
Loads the detector description from a NeXus file, adapted from: http://download.nexusformat.org/sphinx/classes/base_classes/NXdetector.html

Parameters **filename** – name of the file on the disc

class pyFAI.detectors.**Perkin**(*pixel1=0.0002*, *pixel2=0.0002*)
Bases: pyFAI.detectors.Detector

Perkin detector

MAX_SHAPE = (4096, 4096)

aliases = ['Perkin detector']

force_pixel = True

class pyFAI.detectors.**Pilatus**(*pixel1=0.000172*, *pixel2=0.000172*, *x_offset_file=None*, *y_offset_file=None*)
Bases: pyFAI.detectors.Detector

Pilatus detector: generic description containing mask algorithm

Sub-classed by Pilatus1M, Pilatus2M and Pilatus6M

MODULE_GAP = (17, 7)

MODULE_SIZE = (195, 487)

calc_cartesian_positions(*d1=None*, *d2=None*)
Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!!

Parameters

- **d1** (*ndarray (1D or 2D)*) – the Y pixel positions (slow dimension)
- **d2** (*ndarray (1D or 2D)*) – the X pixel positions (fast dimension)

Returns position in meter of the center of each pixels.

Return type ndarray

d1 and d2 must have the same shape, returned array will have the same shape.

calc_mask()
Returns a generic mask for Pilatus detectors...

force_pixel = True

get_splineFile()

set_splineFile(*splineFile=None*)
In this case splinefile is a couple filenames

splineFile

class pyFAI.detectors.**Pilatus100k**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 100k detector

MAX_SHAPE = (195, 487)

class pyFAI.detectors.**Pilatus1M**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 1M detector

MAX_SHAPE = (1043, 981)

class pyFAI.detectors.**Pilatus200k**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 200k detector

MAX_SHAPE = (407, 487)

class pyFAI.detectors.**Pilatus2M**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 2M detector

MAX_SHAPE = (1679, 1475)

class pyFAI.detectors.**Pilatus300k**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 300k detector

MAX_SHAPE = (619, 487)

class pyFAI.detectors.**Pilatus300kw**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 300k-wide detector

MAX_SHAPE = (195, 1475)

class pyFAI.detectors.**Pilatus6M**(*pixel1=0.000172*, *pixel2=0.000172*)
Bases: pyFAI.detectors.Pilatus

Pilatus 6M detector

MAX_SHAPE = (2527, 2463)

class pyFAI.detectors.**Rayonix**(*pixel1=None*, *pixel2=None*)
Bases: pyFAI.detectors.Detector

BINNED_PIXEL_SIZE = {}

binning

force_pixel = True

get_binning()

guess_binning(*data*)
Guess the binning/mode depending on the image shape :param data: 2-tuple with the shape of the image or the image with a .shape attribute.

set_binning(*bin_size=(1, 1)*)
Set the "binning" of the detector,

Parameters **bin_size** (*int or (int, int)*) – set the binning of the detector

class pyFAI.detectors.**Rayonix133**
Bases: pyFAI.detectors.Rayonix

Rayonix 133 2D CCD detector detector also known as mar133

Personnal communication from M. Blum

What should be the default binning factor for those cameras ?

Circular detector

BINNED_PIXEL_SIZE = {8: 0.000256, 1: 3.2e-05, 2: 6.4e-05, 4: 0.000128}

MAX_SHAPE = (4096, 4096)

aliases = ['MAR133']

calc_mask()
Circular mask

force_pixel = True

class pyFAI.detectors.**RayonixLx170**
Bases: pyFAI.detectors.Rayonix

Rayonix lx170 2d CCD Detector (2x1 CCDs).

Nota: this is the same for lx170hs

BINNED_PIXEL_SIZE = {1: 4.42708e-05, 2: 8.85417e-05, 3: 0.0001328125, 4: 0.0001770833, 5: 0.0002213542, 6: 0.00

MAX_SHAPE = (1920, 3840)

aliases = ['Rayonix lx170']

force_pixel = True

class pyFAI.detectors.**RayonixLx255**
Bases: pyFAI.detectors.Rayonix

Rayonix lx255 2d Detector (3x1 CCDs)

Nota: this detector is also called lx255hs

BINNED_PIXEL_SIZE = {1: 4.42708e-05, 2: 8.85417e-05, 3: 0.0001328125, 4: 0.0001770833, 5: 0.0002213542, 6: 0.00

MAX_SHAPE = (1920, 5760)

aliases = ['Rayonix lx225']

class pyFAI.detectors.**RayonixMx170**
Bases: pyFAI.detectors.Rayonix

Rayonix mx170 2d CCD Detector (2x2 CCDs).

Nota: this is the same for mx170hs

BINNED_PIXEL_SIZE = {1: 4.42708e-05, 2: 8.85417e-05, 3: 0.0001328125, 4: 0.0001770833, 5: 0.0002213542, 6: 0.00

MAX_SHAPE = (3840, 3840)

aliases = ['Rayonix mx170']

class pyFAI.detectors.**RayonixMx225**
Bases: pyFAI.detectors.Rayonix

Rayonix mx225 2D CCD detector detector

Nota: this is the same definition for mx225he Personnal communication from M. Blum

BINNED_PIXEL_SIZE = {8: 0.000292969, 1: 3.6621e-05, 2: 7.3242e-05, 3: 0.000109971, 4: 0.000146484}

MAX_SHAPE = (6144, 6144)

aliases = ['Rayonix mx225']

force_pixel = True

class pyFAI.detectors.**RayonixMx225hs**
Bases: pyFAI.detectors.Rayonix

Rayonix mx225hs 2D CCD detector detector

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {1: 3.90625e-05, 2: 7.8125e-05, 3: 0.0001171875, 4: 0.00015625, 5: 0.0001953125, 6: 0.00023

MAX_SHAPE = (5760, 5760)

aliases = ['Rayonix mx225hs']

force_pixel = True

class pyFAI.detectors.**RayonixMx300**
Bases: pyFAI.detectors.Rayonix

Rayonix mx300 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {8: 0.000292969, 1: 3.6621e-05, 2: 7.3242e-05, 3: 0.000109971, 4: 0.000146484}

MAX_SHAPE = (8192, 8192)

aliases = ['Rayonix mx300']

force_pixel = True

class pyFAI.detectors.**RayonixMx300hs**
Bases: pyFAI.detectors.Rayonix

Rayonix mx300hs 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {1: 3.90625e-05, 2: 7.8125e-05, 3: 0.0001171875, 4: 0.00015625, 5: 0.0001953125, 6: 0.00023

MAX_SHAPE = (7680, 7680)

aliases = ['Rayonix mx300hs']

force_pixel = True

class pyFAI.detectors.**RayonixMx325**
Bases: pyFAI.detectors.Rayonix

Rayonix mx325 and mx325he 2D detector (4x4 CCD chips)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {8: 0.000317383, 1: 3.9673e-05, 2: 7.9346e-05, 3: 0.000119135, 4: 0.000158691}

MAX_SHAPE = (8192, 8192)

aliases = ['Rayonix mx325']

class pyFAI.detectors.**RayonixMx340hs**
Bases: pyFAI.detectors.Rayonix

Rayonix mx340hs 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {1: 4.42708e-05, 2: 8.85417e-05, 3: 0.0001328125, 4: 0.0001770833, 5: 0.0002213542, 6: 0.00

MAX_SHAPE = (7680, 7680)

aliases = ['Rayonix mx340hs']

force_pixel = True

class pyFAI.detectors.**RayonixMx425hs**
Bases: pyFAI.detectors.Rayonix

Rayonix mx425hs 2D CCD camera (5x5 CCD chip)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {1: 4.42708e-05, 2: 8.85417e-05, 3: 0.0001328125, 4: 0.0001770833, 5: 0.0002213542, 6: 0.00

MAX_SHAPE = (9600, 9600)

aliases = ['Rayonix mx425hs']

class pyFAI.detectors.**RayonixSx165**
Bases: pyFAI.detectors.Rayonix

Rayonix sx165 2d Detector also known as MAR165.

Circular detector

BINNED_PIXEL_SIZE = {8: 0.000316, 1: 3.95e-05, 2: 7.9e-05, 3: 0.000118616, 4: 0.000158}

MAX_SHAPE = (4096, 4096)

aliases = ['MAR165', 'Rayonix Sx165']

calc_mask()
Circular mask

force_pixel = True

class pyFAI.detectors.**RayonixSx200**
Bases: pyFAI.detectors.Rayonix

Rayonix sx200 2d CCD Detector.

Pixel size are personnal communication from M. Blum.

BINNED_PIXEL_SIZE = {8: 0.000384, 1: 4.8e-05, 2: 9.6e-05, 3: 0.000144, 4: 0.000192}

MAX_SHAPE = (4096, 4096)

aliases = ['Rayonix sx200']

class pyFAI.detectors.**RayonixSx30hs**
Bases: pyFAI.detectors.Rayonix

Rayonix sx30hs 2D CCD camera (1 CCD chip)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {1: 1.5625e-05, 2: 3.125e-05, 3: 4.6875e-05, 4: 6.25e-05, 5: 7.8125e-05, 6: 9.375e-05, 8: 0.000

MAX_SHAPE = (1920, 1920)

aliases = ['Rayonix Sx30hs']

class pyFAI.detectors.**RayonixSx85hs**
Bases: pyFAI.detectors.Rayonix

Rayonix sx85hs 2D CCD camera (1 CCD chip)

Pixel size from a personnal communication from M. Blum

BINNED_PIXEL_SIZE = {1: 4.42708e-05, 2: 8.85417e-05, 3: 0.0001328125, 4: 0.0001770833, 5: 0.0002213542, 6: 0.00

MAX_SHAPE = (1920, 1920)

aliases = ['Rayonix Sx85hs']

class pyFAI.detectors.**Titan**(*pixel1=6e-05*, *pixel2=6e-05*)
Bases: pyFAI.detectors.Detector

Titan CCD detector from Agilent. Mask not handled

MAX_SHAPE = (2048, 2048)

aliases = ['Titan 2k x 2k', 'OXD Titan', 'Agilent Titan']

force_pixel = True

uniform_pixel = True

class pyFAI.detectors.**Xpad_flat**(*pixel1=0.00013*, *pixel2=0.00013*)
Bases: pyFAI.detectors.ImXPadS10

Xpad detector: generic description for ImXPad detector with 8x7modules

BORDER_PIXEL_SIZE_RELATIVE = 2.5

MAX_SHAPE = (960, 560)

MODULE_GAP = (0.00357, 0)

MODULE_SIZE = (120, 80)

PIXEL_SIZE = (0.00013, 0.00013)

aliases = ['Xpad S540 flat']

calc_cartesian_positions(*d1=None*, *d2=None*, *center=True*, *use_cython=True*)
Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!! Adapted to Nexus detector definition

Parameters

- **d1** (*ndarray (1D or 2D)*) – the Y pixel positions (slow dimension)
- **d2** (*ndarray (1D or 2D)*) – the X pixel positions (fast dimension)
- **center** – retrieve the coordinate of the center of the pixel

@parm use_cython: set to False to test Numpy implementation :return: position in meter of the center of each pixels. :rtype: ndarray

d1 and d2 must have the same shape, returned array will have the same shape.

calc_mask()
Returns a generic mask for Xpad detectors... discards the first line and raw form all modules: those are 2.5x bigger and often mis - behaving

force_pixel = True

get_pixel_corners()
Calculate the position of the corner of the pixels

Returns 4D array containing: pixel index (slow dimension) pixel index (fast dimension) corner index (A, B, C or D), triangles or hexagons can be handled the same way vertex position (z,y,x)

uniform_pixel = False

5.7 spline Module

This is piece of software aims at manipulating spline files describing for geometric corrections of the 2D detectors using cubic-spline.

Mainly used at ESRF with FReLoN CCD camera.

class pyFAI.spline.**Spline**(*filename=None*)
Bases: object

This class is a python representation of the spline file

Those file represent cubic splines for 2D detector distortions and makes heavy use of fitpack (dierckx in netlib) — A Python-C wrapper to FITPACK (by P. Dierckx). FITPACK is a collection of FORTRAN programs for curve and surface fitting with splines and tensor product splines. See _http://www.cs.kuleuven.ac.be/cwis/research/nalag/research/topics/fitpack.html or _http://www.netlib.org/dierckx/index.html

array2spline(*smoothing=1000*, *timing=False*)
Calculates the spline coefficients from the displacements matrix using fitpack.

Parameters

- **smoothing** (*float*) – the greater the smoothing, the fewer the number of knots remaining
- **timing** (*bool*) – print the profiling of the calculation

bin(*binning=None*)
Performs the binning of a spline (same camera with different binning)

Parameters **binning** – binning factor as integer or 2-tuple of integers

Type int or (int, int)

comparison(*ref*, *verbose=False*)
Compares the current spline distortion with a reference

Parameters

- **ref** (*Spline instance*) – another spline file
- **verbose** (*bool*) – print or not pylab plots

Returns True or False depending if the splines are the same or not

Return type bool

correct(*pos*)

fliplr()
Flip the spline :return: new spline object

fliplrud()
Flip the spline left-right and up-down :return: new spline object

flipud()
Flip the spline up-down :return: new spline object

getPixelSize()
Return the size of the pixel from as a 2-tuple of floats expressed in meters.

Returns the size of the pixel from a 2D detector

Return type 2-tuple of floats expressed in meter.

read(*filename*)
read an ascii spline file from file

Parameters **filename** (*str*) – file containing the cubic spline distortion file

setPixelSize(*pixelSize*)
Sets the size of the pixel from a 2-tuple of floats expressed in meters.

Param pixel size in meter

spline2array(*timing=False*)
Calculates the displacement matrix using fitpack bisplev(x, y, tck, dx = 0, dy = 0)

Parameters **timing** (*bool*) – profile the calculation or not

Returns Nothing !

Return type float or ndarray

Evaluate a bivariate B-spline and its derivatives. Return a rank-2 array of spline function values (or spline derivative values) at points given by the cross-product of the rank-1 arrays x and y. In special cases, return an array or just a float if either x or y or both are floats.

splineFuncX(*x*, *y*, *list_of_points=False*)
Calculates the displacement matrix using fitpack for the X direction on the given grid.

Parameters

- **x** (*ndarray*) – points of the grid in the x direction
- **y** (*ndarray*) – points of the grid in the y direction
- **list_of_points** – if true, consider the zip(x,y) instead of the of the square array

Returns displacement matrix for the X direction

Return type ndarray

splineFuncY(*x*, *y*, *list_of_points=False*)
calculates the displacement matrix using fitpack for the Y direction

Parameters

- **x** (*ndarray*) – points in the x direction
- **y** (*ndarray*) – points in the y direction
- **list_of_points** – if true, consider the zip(x,y) instead of the of the square array

Returns displacement matrix for the Y direction

Return type ndarray

tilt(*center=(0.0, 0.0)*, *tiltAngle=0.0*, *tiltPlanRot=0.0*, *distanceSampleDetector=1.0*, *timing=False*)
The tilt method apply a virtual tilt on the detector, the point of tilt is given by the center

Parameters

- **center** (*2-tuple of floats*) – position of the point of tilt, this point will not be moved.
- **tiltAngle** (*float in the range [-90:+90] degrees*) – the value of the tilt in degrees
- **tiltPlanRot** (*Float in the range [-180:180]*) – the rotation of the tilt plan with the Ox axis (0 deg for y axis invariant, 90 deg for x axis invariant)
- **distanceSampleDetector** (*float*) – the distance from sample to detector in meter (along the beam, so distance from sample to center)

Returns tilted Spline instance

Return type Spline

write(*filename*)
save the cubic spline in an ascii file usable with Fit2D or SPD

Parameters **filename** (*str*) – name of the file containing the cubic spline distortion file

writeEDF(*basename*)
save the distortion matrices into a couple of files called basename-x.edf and basename-y.edf

Parameters **basename** (*str*) – base of the name used to save the data

zeros(*xmin=0.0*, *ymin=0.0*, *xmax=2048.0*, *ymax=2048.0*, *pixSize=None*)
Defines a spline file with no (zero) displacement.

Parameters

- **xmin** (*float*) – minimum coordinate in x, usually zero
- **xmax** (*float*) – maximum coordinate in x (+1) usually 2048
- **ymin** (*float*) – minimum coordinate in y, usually zero
- **ymax** (*float*) – maximum coordinate y (+1) usually 2048
- **pixSize** (*float*) – size of the pixel

zeros_like(*other*)
Defines a spline file with no (zero) displacement with the same shape as the other one given.

Parameters **other** (*Spline instance*) – another Spline instance

pyFAI.spline.**main**()
Some tests ....

5.8 opencl Module

class pyFAI.opencl.**Device**(*name='None'*, *dtype=None*, *version=None*, *driver_version=None*, *extensions=''*, *memory=None*, *available=None*, *cores=None*, *frequency=None*, *flop_core=None*, *idx=0*, *workgroup=1*)
Bases: object

Simple class that contains the structure of an OpenCL device

pretty_print()
Complete device description

Returns string

class pyFAI.opencl.**OpenCL**
Bases: object

Simple class that wraps the structure ocl_tools_extended.h

This is a static class. ocl should be the only instance and shared among all python modules.

comput_cap = (5, 0)

context_cache = {}

create_context(*devicetype='ALL'*, *useFp64=False*, *platformid=None*, *deviceid=None*, *cached=True*)
Choose a device and initiate a context.

Devicetypes can be GPU,gpu,CPU,cpu,DEF,ACC,ALL. Suggested are GPU,CPU. For each setting to work there must be such an OpenCL device and properly installed. E.g.: If Nvidia driver is installed, GPU will succeed but CPU will fail. The AMD SDK kit is required for CPU via OpenCL. :param devicetype: string in [”cpu”,”gpu”, “all”, “acc”] :param useFp64: boolean specifying if double precision will be used :param platformid: integer :param devid: integer :return: OpenCL context on the selected device

flop_core = 4

get_platform(*key*)
Return a platform according

Parameters **key** (*int or str*) – identifier for a platform, either an Id (int) or it's name

idd = 0

idx = 2

nb_devices = 4

platforms = [NVIDIA CUDA, AMD Accelerated Parallel Processing, Intel(R) OpenCL]

select_device(*dtype='ALL'*, *memory=None*, *extensions=[]*, *best=True*, ***kwargs*)
Select a device based on few parameters (at the end, keep the one with most memory)

Parameters

- **type** – "gpu" or "cpu" or "all" ....
- **memory** – minimum amount of memory (int)
- **extensions** – list of extensions to be present
- **best** – shall we look for the

workgroup = 8192

class pyFAI.opencl.**Platform**(*name='None'*, *vendor='None'*, *version=None*, *extensions=None*, *idx=0*)
Bases: object

Simple class that contains the structure of an OpenCL platform

add_device(*device*)
Add new device to the platform

Parameters **device** – Device instance

get_device(*key*)
Return a device according to key

Parameters **key** (*int or str*) – identifier for a device, either it's id (int) or it's name

pyFAI.opencl.**allocate_cl_buffers**(*buffers*, *device*, *context*)

Parameters **buffers** – the buffers info use to create the pyopencl.Buffer

Returns a dict containing the instanciated pyopencl.Buffer

Return type dict(str, pyopencl.Buffer)

This method instanciate the pyopencl.Buffer from the buffers description.

pyFAI.opencl.**release_cl_buffers**(*cl_buffers*)

Parameters **cl_buffer** (*dict(str, pyopencl.Buffer)*) – the buffer you want to release

This method release the memory of the buffers store in the dict

5.9 ocl_azim Module

C++ less implementation of Dimitris' code based on PyOpenCL

TODO and trick from dimitris still missing:

- dark-current subtraction is still missing
- In fact you might want to consider doing the conversion on the GPU when possible. Think about it, you have a uint16 to float which for large arrays was slow.. You load on the graphic card a uint16 (2x transfer speed) and you convert to float inside so it should be blazing fast.

class pyFAI.ocl_azim.**Integrator1d**(*filename=None*)
Bases: object

Attempt to implements ocl_azim using pyopencl

BLOCK_SIZE = 128

clean(*preserve_context=False*)
Free OpenCL related resources allocated by the library.

clean() is used to reinitiate the library back in a vanilla state. It may be asked to preserve the context created by init or completely clean up OpenCL. Guard/Status flags that are set will be reset.

Parameters **preserve_context** (*bool*) – preserves or destroys all OpenCL resources

configure(*kernel=None*)
The method configure() allocates the OpenCL resources required and compiled the OpenCL kernels. An active context must exist before a call to configure() and getConfiguration() must have been called at least once. Since the compiled OpenCL kernels carry some information on the integration parameters, a change to any of the parameters of getConfiguration() requires a subsequent call to configure() for them to take effect.

If a configuration exists and configure() is called, the configuration is cleaned up first to avoid OpenCL memory leaks

Parameters **kernel_path** – is the path to the actual kernel

execute(*image*)
Perform a 1D azimuthal integration

execute() may be called only after an OpenCL device is configured and a Tth array has been loaded (at least once) It takes the input image and based on the configuration provided earlier it performs the 1D integration. Notice that if the provided image is bigger than N then only N points will be taked into account, while if the image is smaller than N the result may be catastrophic. set/unset and loadTth methods have a direct impact on the execute() method. All the rest of the methods will require at least a new configuration via configure().

Takes an image, integrate and return the histogram and weights

Parameters **image** – image to be processed as a numpy array

Returns tth_out, histogram, bins

TODO: to improve performances, the image should be casted to float32 in an optimal way: currently using numpy machinery but would be better if done in OpenCL

getConfiguration(*Nimage*, *Nbins*, *useFp64=None*)
getConfiguration gets the description of the integrations to be performed and keeps an internal copy

Parameters

- **Nimage** – number of pixel in image
- **Nbins** – number of bins in regrouped histogram
- **useFp64** – use double precision. By default the same as init!

get_status()
return a dictionnary with the status of the integrator: for compatibilty with former implementation

init(*devicetype='GPU'*, *useFp64=True*, *platformid=None*, *deviceid=None*)
Initial configuration: Choose a device and initiate a context. Devicetypes can be GPU, gpu, CPU, cpu, DEF, ACC, ALL. Suggested are GPU,CPU. For each setting to work there must be such an OpenCL device and properly installed. E.g.: If Nvidia driver is installed, GPU will succeed but CPU will fail. The AMD SDK kit (AMD APP) is required for CPU via OpenCL.

Parameters

- **devicetype** – string in [”cpu”,”gpu”, “all”, “acc”]

- **useFp64** – boolean specifying if double precision will be used
- **platformid** – integer
- **devid** – integer

loadTth(*tth*, *dtth*, *tth_min=None*, *tth_max=None*)
Load the 2th arrays along with the min and max value.

loadTth maybe be recalled at any time of the execution in order to update the 2th arrays.

loadTth is required and must be called at least once after a configure()

log(***kwarg*)
log in a file all opencl events

setDummyValue(*dummy*, *delta_dummy*)
Enables dummy value functionality and uploads the value to the OpenCL device.

Image values that are similar to the dummy value are set to 0.

Parameters

- **dummy** – value in image of missing values (masked pixels?)
- **delta_dummy** – precision for dummy values

setMask(*mask*)
Enables the use of a Mask during integration. The Mask can be updated by recalling setMask at any point.

The Mask must be a PyFAI Mask. Pixels with 0 are masked out. TODO: check and invert!

Parameters **mask** – numpy.ndarray of integer.

setRange(*lowerBound*, *upperBound*)
Instructs the program to use a user - defined range for 2th values

setRange is optional. By default the integration will use the tth_min and tth_max given by loadTth() as integration range. When setRange is called it sets a new integration range without affecting the 2th array. All values outside that range will then be discarded when interpolating. Currently, if the interval of 2th (2th + -d2th) is not all inside the range specified, it is discarded. The bins of the histogram are RESCALED to the defined range and not the original tth_max - tth_min range.

setRange can be called at any point and as many times required after a valid configuration is created.

Parameters

- **lowerBound** (*float*) – lower bound of the integration range
- **upperBound** (*float*) – upper bound of the integration range

setSolidAngle(*solidAngle*)
Enables SolidAngle correction and uploads the suitable array to the OpenCL device.

By default the program will assume no solidangle correction unless setSolidAngle() is called. From then on, all integrations will be corrected via the SolidAngle array.

If the SolidAngle array needs to be changes, one may just call setSolidAngle() again with that array

Parameters **solidAngle** (*ndarray*) – the solid angle of the given pixel

unsetDummyValue()
Disable a dummy value. May be re-enabled at any time by setDummyValue

unsetMask()
Disables the use of a Mask from that point. It may be re-enabled at any point via setMask

unsetRange()
Disable the use of a user-defined 2th range and revert to tth_min,tth_max range

unsetRange instructs the program to revert to its default integration range. If the method is called when no user-defined range had been previously specified, no action will be performed

unsetSolidAngle()
Instructs the program to not perform solidangle correction from now on.

SolidAngle correction may be turned back on at any point

5.10 ocl_azim_lut Module

class pyFAI.ocl_azim_lut.**OCL_LUT_Integrator**(*lut*, *image_size*, *devicetype='all'*, *platformid=None*, *deviceid=None*, *checksum=None*, *profile=False*, *empty=None*)

Bases: object

BLOCK_SIZE = 16

integrate(*data*, *dummy=None*, *delta_dummy=None*, *dark=None*, *flat=None*, *solidAngle=None*, *polarization=None*, *dark_checksum=None*, *flat_checksum=None*, *solidAngle_checksum=None*, *polarization_checksum=None*, *preprocess_only=False*, *safe=True*)
Before performing azimuthal integration, the preprocessing is :

data = (data - dark) / (flat*solidAngle*polarization)

Integration is performed using the CSR representation of the look-up table

Parameters

- **dark** – array of same shape as data for pre-processing
- **flat** – array of same shape as data for pre-processing
- **solidAngle** – array of same shape as data for pre-processing
- **polarization** – array of same shape as data for pre-processing
- **dark_checksum** – CRC32 checksum of the given array
- **flat_checksum** – CRC32 checksum of the given array
- **solidAngle_checksum** – CRC32 checksum of the given array
- **polarization_checksum** – CRC32 checksum of the given array
- **safe** – if True (default) compares arrays on GPU according to their checksum, unless, use the buffer location is used
- **preprocess_only** – return the dark subtracted; flat field & solidAngle & polarization corrected image, else

:return averaged data, weighted histogram, unweighted histogram

log_profile()
If we are in profiling mode, prints out all timing for every single OpenCL call

5.11 ocl_azim_csr Module

class pyFAI.ocl_azim_csr.**OCL_CSR_Integrator**(*lut*, *image_size*, *devicetype='all'*, *block_size=32*, *platformid=None*, *deviceid=None*, *checksum=None*, *profile=False*, *empty=None*)

Bases: object

integrate(*data*, *dummy=None*, *delta_dummy=None*, *dark=None*, *flat=None*, *solidAngle=None*, *polarization=None*, *dark_checksum=None*, *flat_checksum=None*, *solidAngle_checksum=None*, *polarization_checksum=None*, *preprocess_only=False*, *safe=True*)

Before performing azimuthal integration, the preprocessing is :

data = (data - dark) / (flat*solidAngle*polarization)

Integration is performed using the CSR representation of the look-up table

Parameters

- **dark** – array of same shape as data for pre-processing
- **flat** – array of same shape as data for pre-processing
- **solidAngle** – array of same shape as data for pre-processing
- **polarization** – array of same shape as data for pre-processing
- **dark_checksum** – CRC32 checksum of the given array
- **flat_checksum** – CRC32 checksum of the given array
- **solidAngle_checksum** – CRC32 checksum of the given array
- **polarization_checksum** – CRC32 checksum of the given array
- **safe** – if True (default) compares arrays on GPU according to their checksum, unless, use the buffer location is used
- **preprocess_only** – return the dark subtracted; flat field & solidAngle & polarization corrected image, else

:return averaged data, weighted histogram, unweighted histogram

log_profile()

If we are in profiling mode, prints out all timing for every single OpenCL call

5.12 ocl_azim_csr_dis Module

class pyFAI.ocl_azim_csr_dis.**OCL_CSR_Integrator**(*lut*, *image_size*, *devicetype='all'*, *block_size=32*, *platformid=None*, *deviceid=None*, *checksum=None*, *profile=False*, *empty=None*)

Bases: object

integrate(*data*, *dummy=None*, *delta_dummy=None*, *dark=None*, *flat=None*, *solidAngle=None*, *polarization=None*, *dark_checksum=None*, *flat_checksum=None*, *solidAngle_checksum=None*, *polarization_checksum=None*)

log_profile()

If we are in profiling mode, prints out all timing for every single OpenCL call

5.13 worker Module

This module contains the Worker class:

A tool able to perform azimuthal integration with: additional saving capabilities like - save as 2/3D structure in a HDF5 File - read from HDF5 files

Aims at being integrated into a plugin like LImA or as model for the GUI

The configuration of this class is mainly done via a dictionary transmitted as a JSON string: Here are the valid keys:

"dist", "poni1", "poni2", "rot1" "rot3" "rot2" "pixel1" "pixel2"

"splineFile" "wavelength"

"poni" #path of the file

"chi_discontinuity_at_0" "do_mask" "do_dark" "do_azimuthal_range" "do_flat" "do_2D" "azimuth_range_min" "azimuth_range_max"

"polarization_factor" "nbpt_rad" "do_solid_angle" "do_radial_range" "do_poisson" "delta_dummy" "nbpt_azim" "flat_field" "radial_range_min" "dark_current" "do_polarization" "mask_file" "detector" "unit" "radial_range_max" "val_dummy" "do_dummy" "method"

}

class pyFAI.worker.**DistortionWorker**(*detector=None*, *dark=None*, *flat=None*, *solidangle=None*, *polarization=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *device=None*)

Bases: object

Simple worker doing dark, flat, solid angle and polarization correction

process(*data*, *normalization=None*)

Process the data and apply a normalization factor :param data: input data :param normalization: normalization factor :return processed data

class pyFAI.worker.**PixelwiseWorker**(*dark=None*, *flat=None*, *solidangle=None*, *polarization=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *device=None*)

Bases: object

Simple worker doing dark, flat, solid angle and polarization correction

process(*data*, *normalization=None*)

Process the data and apply a normalization factor :param data: input data :param normalization: normalization factor :return processed data

class pyFAI.worker.**Worker**(*azimuthalIntgrator=None*, *shapeIn=(2048, 2048)*, *shapeOut=(360, 500)*, *unit='r_mm'*)

Bases: object

do_2D()

get_config()

return configuration as a dictionary

get_json_config()

return configuration as a JSON string

get_normalization_factor()

get_unit()

normalization_factor

process(*data*)

Process a frame #TODO: dark, flat, sa are missing

Param data: numpy array containing the input image

reconfig(*shape=(2048, 2048)*, *sync=False*)

This is just to force the integrator to initialize with a given input image shape

Parameters

- **shape** – shape of the input image
- **sync** – return only when synchronized

reset()

this is just to force the integrator to initialize

save_config(*filename=None*)

setDarkcurrentFile(*imagefile*)

setExtension(*ext*)
enforce the extension of the processed data file written

setFlatfieldFile(*imagefile*)

setJsonConfig(*jsonconfig*)

setSubdir(*path*)
Set the relative or absolute path for processed data

set_normalization_factor(*value*)

set_unit(*value*)

unit

warmup(*sync=False*)
Process a dummy image to ensure everything is initialized

Parameters **sync** – wait for processing to be finished

5.14 io Module

Module for "high-performance" writing in either 1D with Ascii , or 2D with FabIO or even nD with n varying from 2 to 4 using HDF5

Stand-alone module which tries to offer interface to HDF5 via H5Py and capabilities to write EDF or other formats using fabio.

Can be imported without h5py but then limited to fabio & ascii formats.

TODO: * add monitor to HDF5

class pyFAI.io.**AsciiWriter**(*filename=None*, *prefix='fai_'*, *extension='.dat'*)
Bases: pyFAI.io.Writer

Ascii file writer (.xy or .dat)

init(*fai_cfg=None*, *lima_cfg=None*)
Creates the directory that will host the output file(s)

write(*data*, *index=0*)

class pyFAI.io.**FabioWriter**(*filename=None*)
Bases: pyFAI.io.Writer

Image file writer based on FabIO

TODO !!!

init(*fai_cfg=None*, *lima_cfg=None*)
Creates the directory that will host the output file(s)

write(*data*, *index=0*)

class pyFAI.io.**HDF5Writer**(*filename*, *hpath='data'*, *fast_scan_width=None*)
Bases: pyFAI.io.Writer

Class allowing to write HDF5 Files.

CONFIG = 'pyFAI'

DATASET_NAME = 'data'

close()

flush(*radial=None*, *azimuthal=None*)
Update some data like axis units and so on.

Parameters

- **radial** – position in radial direction
- **azimuthal** – position in azimuthal direction

init(*fai_cfg=None*, *lima_cfg=None*)
Initializes the HDF5 file for writing :param fai_cfg: the configuration of the worker as a dictionary

write(*data*, *index=0*)
Minimalistic method to limit the overhead. :param data: array with intensities or tuple (2th,I) or (I,2th,chi)

class pyFAI.io.**Nexus**(*filename*, *mode='r'*)
Bases: object

Writer class to handle Nexus/HDF5 data Manages: entry

pyFAI-subentry detector

#TODO: make it thread-safe !!!

close()
close the filename and update all entries

deep_copy(*name*, *obj*, *where='/'*, *toplevel=None*, *excluded=None*, *overwrite=False*)
perform a deep copy: create a "name" entry in self containing a copy of the object

Parameters

- **where** – path to the toplevel object (i.e. root)
- **toplevel** – firectly the top level Group
- **excluded** – list of keys to be excluded
- **overwrite** – replace content if already existing

find_detector(*all=False*)
Tries to find a detector within a NeXus file, takes the first compatible detector

Parameters **all** – return all detectors found as a list

get_class(*grp*, *class_type='NXcollection'*)
return all sub-groups of the given type within a group

Parameters

- **grp** – HDF5 group
- **class_type** – name of the NeXus class

get_data(*grp*, *class_type='NXdata'*)
return all dataset of the the NeXus class NXdata

Parameters

- **grp** – HDF5 group
- **class_type** – name of the NeXus class

get_entries()
retrieves all entry sorted the latest first.

Returns list of HDF5 groups

get_entry(*name*)
Retrieves an entry from its name

Parameters **name** – name of the entry to retrieve

Returns HDF5 group of NXclass == NXentry

new_class(*grp*, *name*, *class_type='NXcollection'*)
create a new sub-group with type class_type :param grp: parent group :param name: name of the sub-group :param class_type: NeXus class name :return: subgroup created

new_detector(*name='detector'*, *entry='entry'*, *subentry='pyFAI'*)
Create a new entry/pyFAI/Detector

Parameters

- **detector** – name of the detector
- **entry** – name of the entry
- **subentry** – all pyFAI description of detectors should be in a pyFAI sub-entry

new_entry(*entry='entry'*, *program_name='pyFAI'*, *title='description of experiment'*, *force_time=None*)
Create a new entry

Parameters

- **entry** – name of the entry
- **program_name** – value of the field as string
- **title** – value of the field as string

@force_time: enforce the start_time (as string!) :return: the corresponding HDF5 group

new_instrument(*entry='entry'*, *instrument_name='id00'*)
Create an instrument in an entry or create both the entry and the instrument if

class pyFAI.io.**Writer**(*filename=None*, *extension=None*)
Bases: object

Abstract class for writers.

CONFIG_ITEMS = ['filename', 'dirname', 'extension', 'subdir', 'hpath']

flush(**arg*, ***kwarg*)
To be implemented

init(*fai_cfg=None*, *lima_cfg=None*)
Creates the directory that will host the output file(s) :param fai_cfg: configuration for worker :param lima_cfg: configuration for acquisition

setJsonConfig(*json_config=None*)
Sets the JSON configuration

write(*data*)
To be implemented

pyFAI.io.**from_isotime**(*text*, *use_tz=False*)

Parameters **text** – string representing the time is iso format

pyFAI.io.**get_isotime**(*forceTime=None*)

Parameters **forceTime** (*float*) – enforce a given time (current by default)

Returns the current time as an ISO8601 string

Return type string

pyFAI.io.**is_hdf5**(*filename*)
Check if a file is actually a HDF5 file

Parameters **filename** – this file has better to exist

5.15 calibration Module

pyFAI-calib

A tool for determining the geometry of a detector using a reference sample.

class pyFAI.calibration.**AbstractCalibration**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)

Bases: object

Everything that is common to Calibration and Recalibration

HELP = {'reset': 'Reset the geometry to the initial guess (rotation to zero, distance to 0.1m, poni at the center of the im

PARAMETERS = ['dist', 'poni1', 'poni2', 'rot1', 'rot2', 'rot3', 'wavelength']

UNITS = {'poni1': 'meter', 'poni2': 'meter', 'rot1': 'radian', 'rot3': 'radian', 'rot2': 'radian', 'wavelength': 'meter',

VALID_URL = ['', 'file', 'hdf5', 'nxs', 'h5']

analyse_options(*options=None*, *args=None*)
Analyse options and arguments

Returns option,arguments

configure_parser(*version='calibration from pyFAI version 0.10.3: 20/03/2015'*, *usage='pyFAI-calib [options] input_image.edf'*, *description=None*, *epilog=None*)
Common configuration for parsers

extract_cpt(*method='massif'*)
Performs an automatic keypoint extraction: Can be used in recalib or in calib after a first calibration has been performed

get_pixelSize(*ans*)
convert a comma separated sting into pixel size

postProcess()
Common part: shows the result of the azimuthal integration in 1D and 2D

preprocess()
Common part: do dark, flat correction thresholding, ... and read missing data from keyboard if needed

prompt()
prompt for commands to guide the calibration process

Returns True when the user is happy with what he has, False to request another refinement

read_dSpacingFile(*verbose=True*)
Read the name of the calibrant / file with d-spacing

read_pixelsSize()
Read the pixel size from prompt if not available

read_wavelength()
Read the wavelength

refine()
Contains the common geometry refinement part

set_data(*data*)
call-back function for the peak-picker

validate_calibration()
Validate the calivration and calculate the offset in the diffraction image

win_error = 'We are under windows, matplotlib is not able to display too many images without crashing, this is why

class pyFAI.calibration.**Calibration**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *gaussianWidth=None*, *wavelength=None*, *calibrant=None*)

Bases: pyFAI.calibration.AbstractCalibration

class doing the calibration of frames

gui_peakPicker()

parse()
parse options from command line

preprocess()
do dark, flat correction thresholding, ...

refine()
Contains the geometry refinement part specific to Calibration

class pyFAI.calibration.**CheckCalib**(*poni=None*, *img=None*, *unit='2th_deg'*)

Bases: object

get_1dsize()

integrate()

parse()

rebuild()
Rebuild the diffraction image and measures the offset with the reference :return: offset

show()
Show the image with the the errors

size1d

smooth_mask(*hwhm=5*)
smooth out around the mask to avoid aligning on the mask

class pyFAI.calibration.**MultiCalib**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*)

Bases: object

get_pixelSize(*ans*)
convert a comma separated sting into pixel size

parse()
parse options from command line

process()

read_dSpacingFile()
Read the name of the calibrant or the file with d-spacing

read_pixelsSize()
Read the pixel size from prompt if not available

read_wavelength()
Read the wavelength

regression()

class pyFAI.calibration.**Recalibration**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)

Bases: pyFAI.calibration.AbstractCalibration

class doing the re-calibration of frames

parse()
parse options from command line

preprocess()
do dark, flat correction thresholding, ...

read_dSpacingFile()
Read the name of the file with d-spacing

refine()
Contains the geometry refinement part specific to Recalibration

pyFAI.calibration.**calib**(*img*, *calibrant*, *detector*, *basename='from_ipython'*, *reconstruct=False*, *dist=0.1*, *interactive=True*)
Procedural interfact for calibration

Parameters

- **img** – 2d array representing the imagence setup with mask
- **calibrant** – Instance of Calibrant, set-up with wavelength
- **detector** – Detector instance containing the mask
- **recontruct** – perform image reconstruction of masked pixel ?
- **interactive** – set to false for testing

pyFAI.calibration.**get_detector**(*detector*, *datafiles=None*)
Detector factory taking into account the binning knowing the datafiles :param detector: string or detector or other junk :param datafiles: can be a list of images to be opened and their shape used. :return pyFAI.detector.Detector instance

5.16 peak_picker Module

class pyFAI.peak_picker.**ControlPoints**(*filename=None*, *calibrant=None*, *wavelength=None*)
Bases: object

This class contains a set of control points with (optionally) their ring number hence d-spacing and diffraction 2Theta angle ...

append(*points*, *ring=None*, *annotate=None*, *plot=None*)

Parameters

- **point** – list of points
- **ring** – ring number
- **annotate** – matplotlib.annotate reference
- **plot** – matplotlib.plot reference

Returns PointGroup instance

append_2theta_deg(*points*, *angle=None*, *ring=None*)

Parameters

- **point** – list of points
- **angle** – 2-theta angle in degrees

check()
check internal consistency of the class

dSpacing

get(*ring=None*)
retireves the last set of points for a given ring (by default the last)

Parameters **ring** – index of ring to search for

getList()
Retrieve the list of control points suitable for geometry refinement with ring number

getList2theta()
Retrieve the list of control points suitable for geometry refinement

getListRing()
Retrieve the list of control points suitable for geometry refinement with ring number

getWeightedList(*image*)
Retrieve the list of control points suitable for geometry refinement with ring number and intensities :param image: :return: a (x,4) array with pos0, pos1, ring nr and intensity

#TODO: refine the value of the intensity using 2nd order polynomia

get_dSpacing()

get_wavelength()

load(*filename*)
load all control points from a file

pop(*ring=None*)
Remove the set of points for a given ring (by default the last)

Parameters **ring** – index of ring of which remove the last group

readRingNrFromKeyboard()
Ask the ring number values for the given points

reset()
remove all stored values and resets them to default

save(*filename*)
Save a set of control points to a file :param filename: name of the file :return: None

setWavelength_change2th(*value=None*)

setWavelength_changeDs(*value=None*)
This is probably not a good idea, but who knows !

set_dSpacing(*lst*)

set_wavelength(*value=None*)

wavelength

class pyFAI.peak_picker.**PeakPicker**(*data*, *reconst=False*, *mask=None*, *pointfile=None*, *calibrant=None*, *wavelength=None*, *method='massif'*)
Bases: object

This class is in charge of peak picking, i.e. find bragg spots in the image Two methods can be used : massif or blob

VALID_METHODS = ['massif', 'blob', 'watershed']

closeGUI()

contour(*data*, *cmap='autumn'*, *linewidths=2*, *linestyles='dashed'*)
Overlay a contour-plot

Parameters **data** – 2darray with the 2theta values in radians...

display_points(*minIndex=0*)
display all points and their ring annotations :param minIndex: ring index to start with

finish(*filename=None*, *callback=None*)
Ask the ring number for the given points

Parameters **filename** – file with the point coordinates saved

gui(*log=False*, *maximize=False*, *pick=True*)

Parameters **log** – show z in log scale

help = ['Please select rings on the diffraction image. In parenthesis, some modified shortcuts for single button mouse

init(*method*, *sync=True*)
Unified initializer

load(*filename*)
load a filename and plot data on the screen (if GUI)

massif_contour(*data*)
Overlays a mask over a diffraction image

Parameters **data** – mask to be overlaid

on_minus_pts_clicked(**args*)
callback function

on_option_clicked(**args*)
callback function

on_plus_pts_clicked(**args*)
callback function

on_refine_clicked(**args*)
callback function

onclick(*event*)
Called when a mouse is clicked

peaks_from_area(*mask*, *Imin*, *keep=1000*, *refine=True*, *method=None*, *ring=None*)
Return the list of peaks within an area

Parameters

- **mask** – 2d array with mask.
- **Imin** – minimum of intensity above the background to keep the point
- **keep** – maximum number of points to keep
- **method** – enforce the use of detection using "massif" or "blob"

Returns list of peaks [y,x], [y,x], ...]

reset()
Reset control point and graph (if needed)

sync_init()

class pyFAI.peak_picker.**PointGroup**(*points=None*, *ring=None*, *annotate=None*, *plot=None*, *force_label=None*)
Bases: object

Class contains a group of points ... They all belong to the same Debye-Scherrer ring

code
Numerical value for the label: mainly for sorting

classmethod **get_label**()
return the next label

get_ring()

label

last_label = 0

classmethod **reset_label**()
reset intenal counter

ring

classmethod **set_label**(*label*)
update the internal counter if needed

set_ring(*value*)

5.17 massif Module

class pyFAI.massif.**Massif**(*data=None*)
Bases: object

A massif is defined as an area around a peak, it is used to find neighbouring peaks

calculate_massif(*x*)
defines a map of the massif around x and returns the mask

delValleySize()

find_peaks(*x*, *nmax=200*, *annotate=None*, *massif_contour=None*, *stdout=<open file '<stdout>', mode 'w' at 0x7fa68d5f11e0>*)
All in one function that finds a maximum from the given seed (x) then calculates the region extension and extract position of the neighboring peaks. :param x: seed for the calculation, input coordinates :param nmax: maximum number of peak per region :param annotate: call back method taking number of points + coordinate as input. :param massif_contour: callback to show the contour of a massif with the given index. :param stdout: this is the file where output is written by default. :return: list of peaks

getBinnedData()
:return binned data

getBluredData()

Returns a blurred image

getLabeledMassif(*pattern=None*)

Returns an image composed of int with a different value for each massif

getMedianData()

Returns a spacial median filtered image

getValleySize()

initValleySize()

nearest_peak(*x*)

Parameters **x** – coordinates of the peak

:returns the coordinates of the nearest peak

peaks_from_area(*mask*, *Imin=None*, *keep=1000*, ***kwarg*)
Return the list of peaks within an area

Parameters

- **mask** – 2d array with mask.
- **Imin** – minimum of intensity above the background to keep the point
- **keep** – maximum number of points to keep
- **kwarg** – ignored parameters

Returns list of peaks [y,x], [y,x], ...]

setValleySize(*size*)

valley_size
Defines the minimum distance between two massifs

5.18 blob_detection Module

class pyFAI.blob_detection.**BlobDetection**(*img*, *cur_sigma=0.25*, *init_sigma=0.5*, *dest_sigma=1*, *scale_per_octave=2*, *mask=None*)

Bases: object

Performs a blob detection: http://en.wikipedia.org/wiki/Blob_detection using a Difference of Gaussian + Pyramid of Gaussians

direction()
Perform and plot the two main directions of the peaks, considering their previously calculated scale ,by calculating the Hessian at different sizes as the combination of gaussians and their first and second derivatives

nearest_peak(*p*, *refine=True*, *Imin=None*)
Return the nearest peak from a position

Parameters

- **p** – input position (y,x) 2-tuple of float
- **refine** – shall the position be refined on the raw data
- **Imin** – minimum of intensity above the background

peaks_from_area(*mask*, *keep=None*, *refine=True*, *Imin=None*, ***kwargs*)
Return the list of peaks within an area

Parameters

- **mask** – 2d array with mask.
- **refine** – shall the position be refined on the raw data
- **Imin** – minimum of intensity above the background
- **kwarg** – ignored parameters

Returns list of peaks [y,x], [y,x], ...]

process(*max_octave=None*)
Perform the keypoint extraction for max_octave cycles or until all octaves have been processed. :param max_octave: number of octave to process

refine_Hessian(*kpx*, *kpy*, *kps*)
Refine the keypoint location based on a 3 point derivative, and delete uncoherent keypoints

Parameters

- **kpx** – x_pos of keypoint
- **kpy** – y_pos of keypoint
- **kps** – s_pos of keypoint

:return arrays of corrected coordinates of keypoints, values and locations of keypoints

refine_Hessian_SG(*kpx*, *kpy*, *kps*)
Savitzky Golay algorithm to check if a point is really the maximum :param kpx: x_pos of keypoint :param kpy: y_pos of keypoint :param kps: s_pos of keypoint :return array of corrected keypoints

refinement()

show_neighboor()

show_stats()
Shows a window with the repartition of keypoint in function of scale/intensity

tresh = 0.6

pyFAI.blob_detection.**image_test**()

pyFAI.blob_detection.**local_max**(*dogs*, *mask=None*, *n_5=True*)

Parameters **dogs** – 3d array with (sigma,y,x) containing difference of gaussians

@parm mask: mask out keypoint next to the mask (or inside the mask) :param n_5: look for a larger neighborhood

pyFAI.blob_detection.**make_gaussian**(*im*, *sigma*, *xc*, *yc*)

5.19 calibrant Module

Calibrant

A module containing classical calibrant and also tools to generate d-spacing.

class pyFAI.calibrant.**Calibrant**(*filename=None*, *dSpacing=None*, *wavelength=None*)
Bases: object

A calibrant is a reference compound where the d-spacing (interplanar distances) are known. They are expressed in Angstrom (in the file)

append_2th(*value*)

append_dSpacing(*value*)

dSpacing

fake_calibration_image(*ai*, *shape=None*, *Imax=1.0*, *U=0*, *V=0*, *W=0.0001*)
Generates a fake calibration image from an azimuthal integrator

Parameters

- **ai** – azimuthal integrator
- **Imax** – maximum intensity of rings
- **V, W** (*U,*) – width of the peak from Caglioti's law (FWHM^2 = Utan(th)^2 + Vtan(th) + W)

get_2th()

get_2th_index(*angle*)
return the index in the 2theta angle index

get_dSpacing()

get_wavelength()

load_file(*filename=None*)

save_dSpacing(*filename=None*)
save the d-spacing to a file

setWavelength_change2th(*value=None*)

setWavelength_changeDs(*value=None*)
This is probably not a good idea, but who knows !

set_dSpacing(*lst*)

set_wavelength(*value=None*)

wavelength

class pyFAI.calibrant.**calibrant_factory**(*basedir=None*)
Bases: object

Behaves like a dict but is actually a factory: Each time one retrieves an object it is a new geniune new calibrant (unmodified)

get(*what*, *notfound=None*)

has_key(*k*)

items()

keys()

values()

5.20 distortion Module

class pyFAI.distortion.**Distortion**(*detector='detector'*, *shape=None*, *method='LUT'*, *device=None*, *workgroup=8*)

Bases: object

This class applies a distortion correction on an image.

New version compatible both with CSR and LUT...

calc_LUT(**arg*, ***kw*)

calc_init()
initialize all arrays

calc_pos(**arg*, ***kw*)

calc_size(**arg*, ***kw*)
Considering the "half-CCD" spline from ID11 which describes a (1025,2048) detector, the physical location of pixels should go from: [-17.48634 : 1027.0543, -22.768829 : 2028.3689] We chose to discard pixels falling outside the [0:1025,0:2048] range with a lose of intensity

correct(*image*)
Correct an image based on the look-up table calculated ...

Parameters **image** – 2D-array with the image

Returns corrected 2D image

reset(*method=None*, *device=None*, *workgroup=None*)
reset the distortion correction and re-calculate the look-up table

Parameters

- **method** – can be "lut" or "csr", "lut" looks faster
- **device** – can be None, "cpu" or "gpu" or the id as a 2-tuple of integer
- **worgroup** – enforce the workgroup size for CSR.

uncorrect(*image*)
Take an image which has been corrected and transform it into it's raw (with loss of information)

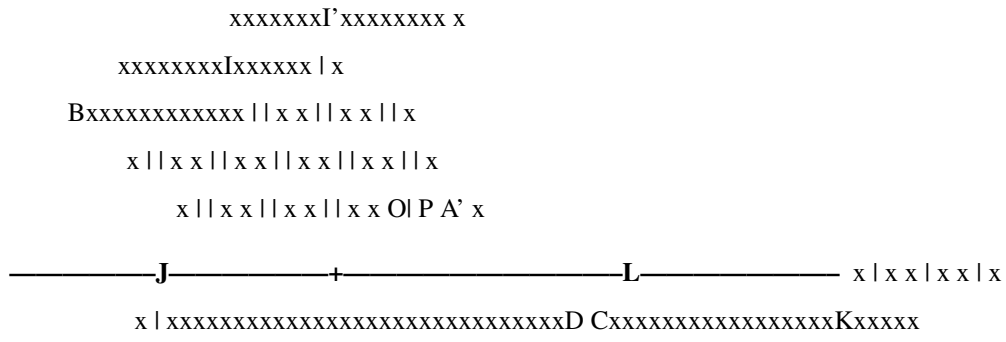
Parameters **image** – 2D-array with the image

Returns uncorrected 2D image and a mask (pixels in raw image

class pyFAI.distortion.**Quad**(*buffer*)

Bases: object

xxxxxA



calc_area()

calc_area_AB(*I1*, *I2*)

calc_area_BC(*J1*, *J2*)

calc_area_CD(*K1*, *K2*)

calc_area_DA(*L1*, *L2*)

calc_area_old()

calc_area_vectorial()

get_box(*i*, *j*)

get_box_size0()

get_box_size1()

get_idx(*i*, *j*)

get_offset0()

get_offset1()

init_slope()

integrateAB(*start*, *stop*, *calc_area*)

populate_box()

reinit(*A0*, *A1*, *B0*, *B1*, *C0*, *C1*, *D0*, *D1*)

pyFAI.distortion.**test**()

5.21 units Module

class pyFAI.units.**Enum**

Bases: dict

Simple class half way between a dict and a class, behaving as an enum

pyFAI.units.**to_unit**(*obj*)

5.22 `utils` Module

Utilities, mainly for image treatment

class pyFAI.utils.**FixedParameters**
Bases: set

Like a set, made for FixedParameters in geometry refinement

add_or_discard(*key*, *value=True*)
Add a value to a set if value, else discard it :param key: element to added or discared from set :type value: boolean. If None do nothing ! :return: None

pyFAI.utils.**averageDark**(*lstimg*, *center_method='mean'*, *cutoff=None*, *quantiles=(0.5, 0.5)*)
Averages a serie of dark (or flat) images. Centers the result on the mean or the median ... but averages all frames within cutoff*std

Parameters

- **lstimg** – list of 2D images or a 3D stack
- **center_method** – is the center calculated by a "mean" or a "median", or "quantile"
- **cutoff** – keep all data where (I-center)/std < cutoff
- **quantiles** – 2-tuple of floats average out data between the two quantiles

Returns 2D image averaged

pyFAI.utils.**averageImages**(*listImages*, *output=None*, *threshold=0.1*, *minimum=None*, *maximum=None*, *darks=None*, *flats=None*, *filter_='mean'*, *correct_flat_from_dark=False*, *cutoff=None*, *format='edf'*)
Takes a list of filenames and create an average frame discarding all saturated pixels.

Parameters

- **listImages** – list of string representing the filenames
- **output** – name of the optional output file
- **threshold** – what is the upper limit? all pixel > max*(1-threshold) are discareded.
- **minimum** – minimum valid value or True
- **maximum** – maximum valid value
- **darks** – list of dark current images for subtraction
- **flats** – list of flat field images for division
- **filter** – can be maximum, mean or median (default=mean)
- **correct_flat_from_dark** – shall the flat be re-corrected ?
- **cutoff** – keep all data where (I-center)/std < cutoff

Returns filename with the data or the data ndarray in case format=None

pyFAI.utils.**binning**(*input_img*, *binsize*, *norm=True*)

Parameters

- **input_img** – input ndarray
- **binsize** – int or 2-tuple representing the size of the binning
- **norm** – if False, do average instead of sum

Returns binned input ndarray

pyFAI.utils.**boundingBox**(*img*)
Tries to guess the bounding box around a valid massif

Parameters **img** – 2D array like

Returns 4-typle (d0_min, d1_min, d0_max, d1_max)

pyFAI.utils.**calc_checksum**(*ary*, *safe=True*)
Calculate the checksum by default (or returns its buffer location if unsafe)

pyFAI.utils.**center_of_mass**(*img*)
Calculate the center of mass of of the array. Like scipy.ndimage.measurements.center_of_mass :param img: 2-D array :return: 2-tuple of float with the center of mass

pyFAI.utils.**concatenate_cl_kernel**(*filenames*)

Parameters **filenames** – filenames containing the kernels

this method concatenates all the kernel from the list

pyFAI.utils.**convert_CamelCase**(*name*)
convert a function name in CamelCase into camel_case

pyFAI.utils.**deg2rad**(*dd*)
Convert degrees to radian in the range -pi->pi

Parameters **dd** – angle in degrees

Nota: depending on the platform it could be 0<2pi A branch is cheaper than a trigo operation

pyFAI.utils.**deprecated**(*func*)

pyFAI.utils.**dog**(*s1*, *s2*, *shape=None*)
2D difference of gaussian typically 1 to 10 parameters

pyFAI.utils.**dog_filter**(*input_img*, *sigma1*, *sigma2*, *mode='reflect'*, *cval=0.0*)
2-dimensional Difference of Gaussian filter implemented with FFTw

Parameters

- **input_img** (*array-like*) – input_img array to filter
- **sigma** (*scalar or sequence of scalars*) – standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **mode** – {'reflect','constant','nearest','mirror', 'wrap'}, optional The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'
- **cval** – scalar, optional Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

pyFAI.utils.**expand**(*input_img*, *sigma*, *mode='constant'*, *cval=0.0*)
Expand array a with its reflection on boundaries

Parameters

- **a** – 2D array
- **sigma** – float or 2-tuple of floats.

:param mode:"constant", "nearest", "reflect" or mirror :param cval: filling value used for constant, 0.0 by default

Nota: sigma is the half-width of the kernel. For gaussian convolution it is adviced that it is 4*sigma_of_gaussian

pyFAI.utils.**expand_args**(*args*)
Takes an argv and expand it (under Windows, cmd does not convert *.tif into a list of files. Keeps only valid files (thanks to glob)

Parameters **args** – list of files or wilcards

Returns list of actual args

pyFAI.utils.**float_**(*val*)
Convert anything to a float ... or None if not applicable

pyFAI.utils.**gaussian**(*M*, *std*)
Return a Gaussian window of length M with standard-deviation std.

This differs from the scipy.signal.gaussian implementation as: - The default for sym=False (needed for gaussian filtering without shift) - This implementation is normalized

Parameters

- **M** – length of the windows (int)
- **std** – standatd deviation sigma

The FWHM is 2*numpy.sqrt(2 * numpy.pi)*std

pyFAI.utils.**gaussian_filter**(*input_img*, *sigma*, *mode='reflect'*, *cval=0.0*)
2-dimensional Gaussian filter implemented with FFTw

Parameters

- **input_img** (*array-like*) – input array to filter
- **sigma** (*scalar or sequence of scalars*) – standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **mode** – {'reflect','constant','nearest','mirror', 'wrap'}, optional The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'
- **cval** – scalar, optional Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

pyFAI.utils.**get_calibration_dir**()
get the full path of a calibration directory

Returns the full path of the calibrant file

pyFAI.utils.**get_cl_file**(*filename*)
get the full path of a openCL file

Returns the full path of the openCL source file

pyFAI.utils.**get_ui_file**(*filename*)
get the full path of a user-interface file

Returns the full path of the ui

pyFAI.utils.**int_**(*val*)
Convert anything to an int ... or None if not applicable

pyFAI.utils.**lazy_property**
meant to be used for lazy evaluation of an object attribute. property should represent non-mutable data, as it replaces itself.

pyFAI.utils.**maximum_position**(*img*)
Same as scipy.ndimage.measurements.maximum_position: Find the position of the maximum of the values of the array.

Parameters **img** – 2-D image

Returns 2-tuple of int with the position of the maximum

pyFAI.utils.**measure_offset**(*img1*, *img2*, *method='numpy'*, *withLog=False*, *withCorr=False*)
Measure the actual offset between 2 images :param img1: ndarray, first image :param img2: ndarray, second image, same shape as img1 :param withLog: shall we return logs as well ? boolean :return: tuple of floats with the offsets

pyFAI.utils.**readFloatFromKeyboard**(*text*, *dictVar*)
Read float from the keyboard ....

Parameters

- **text** – string to be displayed
- **dictVar** – dict of this type: {1: [set_dist_min],3: [set_dist_min, set_dist_guess, set_dist_max]}

pyFAI.utils.**read_cl_file**(*filename*)

Parameters **filename** – read an OpenCL file and apply a preprocessor

Returns preprocessed source code

pyFAI.utils.**relabel**(*label*, *data*, *blured*, *max_size=None*)
Relabel limits the number of region in the label array. They are ranked relatively to their max(I0)-max(blur(I0)

Parameters

- **label** – a label array coming out of scipy.ndimage.measurement.label
- **data** – an array containing the raw data
- **blured** – an array containing the blured data
- **max_size** – the max number of label wanted

:return array like label

pyFAI.utils.**removeSaturatedPixel**(*ds*, *threshold=0.1*, *minimum=None*, *maximum=None*)

Parameters

- **ds** – a dataset as ndarray
- **threshold** – what is the upper limit? all pixel > max*(1-threshold) are discareded.
- **minimum** – minumum valid value (or True for auto-guess)
- **maximum** – maximum valid value

Returns another dataset

pyFAI.utils.**shift**(*input_img*, *shift_val*)
Shift an array like scipy.ndimage.interpolation.shift(input_img, shift_val, mode="wrap", order=0) but faster :param input_img: 2d numpy array :param shift_val: 2-tuple of integers :return: shifted image

pyFAI.utils.**shiftFFT**(*input_img*, *shift_val*, *method='fftw'*)
Do shift using FFTs Shift an array like scipy.ndimage.interpolation.shift(input, shift, mode="wrap", order="infinity") but faster :param input_img: 2d numpy array :param shift_val: 2-tuple of float :return: shifted image

pyFAI.utils.**str_**(*val*)
Convert anything to a string ... but None -> ""

pyFAI.utils.**timeit**(*func*)

pyFAI.utils.**unBinning**(*binnedArray*, *binsize*, *norm=True*)

Parameters

- **binnedArray** – input ndarray
- **binsize** – 2-tuple representing the size of the binning
- **norm** – if True (default) decrease the intensity by binning factor. If False, it is non-conservative

Returns unBinned input ndarray

5.23 gui_utils Module

gui_utils

Module to handle matplotlib and the Qt backend

class pyFAI.gui_utils.**Event**(*width*, *height*)
Bases: object

Dummy class for dummy things

pyFAI.gui_utils.**maximize_fig**(*fig=None*)
Try to set the figure fullscreen

pyFAI.gui_utils.**update_fig**(*fig=None*)
Update a matplotlib figure with a Qt4 backend

Parameters **fig** – pylab figure

CHAPTER

SIX

INSTALLATION OF PYTHON FAST AZIMUTHAL INTEGRATION LIBRARY

Author: Jérôme Kieffer

Date: 20/03/2015

Keywords: Installation procedure

Target: System administrators

Reference:

6.1 Abstract

Installation procedure

6.2 Hardware requirement

PyFAI has been tested on various hardware: i386, x86_64, PPC64le, ARM. The main constrain may be the memory requirement: 2GB of memory is a minimal requirement to run the tests. The program may run with less but "MemoryError" are expected (appearing sometimes as segmentation faults). As a consequence, a 64-bits operating system is strongly advised.

6.3 Dependencies

PyFAI is a Python library which relies on the scientific stack (numpy, scipy, matplotlib)

- Python: version 2.6, 2.7 and 3.2, 3.3 and 3.4
- NumPy: version 1.4 or newer
- SciPy: version 0.7 or newer
- Matplotlib: verson 0.99 or newer
- FabIO: version 0.08 or newer

There are plenty of optional dependencies which will not prevent pyFAI from working by may impair performances or prevent tools from properly working:

- h5py (to access HDF5 files)
- pyopencl (for GPU computing)
- fftw (for image analysis)

- pymca (for mask drawing)
- PyQt4 or PySide (for the graphical user interface)

6.4 Build dependencies:

In addition to the run dependencies, pyFAI needs a C compiler.

C files are generated from **cython_** source and distributed. Cython is only needed for developing new binary modules. If you want to generate your own C files, make sure your local Cython version supports memory-views (available from Cython v0.17 and newer).

6.5 Building procedure

```
python setup.py build install
```

There are few specific options to setup.py:

- –no-cython: do not use cython (even if present) and use the C source code provided by the development team
- –no-openmp: if you compiler lacks OpenMP support (MacOSX)
- –with-testimages: build the source distribution including all test images. Download 200MB of test images to create a self consistent tar-ball.

Author: Jérôme Kieffer

Date: 29/01/2015

Keywords: Installation procedure on Linux

Target: System administrators

6.5.1 Installation procedure on Linux

Installation procedure on Debian/Ubuntu

PyFAI has been designed and originally developed on Ubuntu 10.04 and debian6. Now it is included into debian7, 8 and any recent Ubuntu distribution. To install it, simply use the package provided by the distribution.

:: sudo apt-get install pyfai

To build a more recent version, pyFAI provides you a small scripts which builds a debian package and installs it. It relies on stdeb:

::

sudo apt-get install python-stdeb cython python-fabio ./build-deb.sh

If you are interested in programming in Python3, use

:: sudo apt-get install cython3 python3-fabio ./build-deb.sh 3

Installation procedure on other linux distibution

:: python setup.py build build_doc sudo python setup.py install

Author: Jérôme Kieffer

Date: 29/01/2015

Keywords: Installation procedure on MacOSX

Target: System administrators

6.5.2 Installation procedure on MacOSX

Using PIP

To install pyFAI on an Apple computer you will need a scientific Python stack. MacOSX provides by default Python2.7 with Numpy which is a good basis.

:: sudo pip install matplotlib –upgrade sudo pip install scipy –upgrade sudo pip install fabio –upgrade sudo pip install pyFAI –upgrade

If you get an error about the local “UTF-8”, try to:

:: export LC_ALL=C

Before the installation

Installation from sources

Get the sources from Github:

:: git clone https://github.com/pyFAI/pyFAI.git cd pyFAI

To build pyFAI from sources, a compiler is needed. Apple provides Xcode for free: https://developer.apple.com/xcode/

Another option is to use GCC which provides supports for multiprocessing via OpenMP (see below)

Optional build dependencies: Cython (>v0.17) is needed to translate the source files into C code. If Cython is present on your system, the source code will be re-generated and compiled.

:: sudo pip install cython –upgrade

About OpenMP

There is an issue with MacOSX (v10.8 onwards) where the default compiler (Xcode 5 or 6) switched from gcc 4.2 to clang and dropped the support for OpenMP. This is why OpenMP is by default deactivated under MacOSX. If you have installed an OpenMP-able compiler like GCC, you can re-activate it using the flag –openmp for setup.py

:: LC_ALL=C python setup.py build –openmp sudo LC_ALL=C python setup.py install

Author: Jérôme Kieffer

Date: 20/03/2015

Keywords: Installation procedure

Target: System administrators

6.5.3 Installation procedure on Windows

PyFAI is a Python library. Even if you are only interested in some tool like pyFAI-calib or pyFAI-integrate, you need to install the complete library (for now). This is usually performed in 3 steps: install Python, the scientific python stack and finally the pyFAI itself.

Get Python

Unlike on Unix computers, Python is not available by default on Windows computers. We recommend you to install the 64 bit version of Python from http://python.org, preferably the latest version from the 2.7 series. Any version between 2.6, 2.7, 3.2, 3.3 or 3.4 should be OK but 2.7 is the most tested.

The 64bits version is strongly advised if your hardware and operating system supports it, as the 32 bits versions is limited to 2GB of memory, hence unable to treat large images (4096 x 4096). The test suite is not passing on Windows 32 bits due to the limited amount of memory available to the Python process, nevertheless, pyFAI is running on Winodws32 bits (but not as well).

Alternative Scientific Python stacks exists, like Enthought Python Distribution, Canopy, Anaconda, PythonXY or WinPython. They all offer most of the scientific packages already installed which makes the installation of dependencies much easier. On the other hand, they all offer different packaging system and we cannot support all of them. Moreover, distribution from Enthought and Continuum are not free so you should be able to get support from those companies.

Install PIP

PIP is the package management system for Python, it connects to http://pypi.python.org, download and install software packages from there.

PIP has revolutionize the way Python libraries are installed as it is able to select the right build for your system, or compile from the sources (Which could be tricky).

To install it, download: https://bootstrap.pypa.io/get-pip.py and run it:

:: python get-pip.py

Assuming python.exe is already in your PATH.

Install the scientific stack

The strict dependencies for pyFAI are:

- NumPy
- SciPy
- matplotlib
- FabIO

Recommanded dependencies are:

- cython
- h5py
- pyopencl
- PyQt4
- pymca
- rfoo
- pyfftw3
- lxml

The ways

Using PIP

Most of the dependencies are available via PIP:

:: pip install numpy pip install scipy pip install matplotlib pip install fabio pip install PyQt4

Note that numpy/scipy/matplotlib are already installed in most "Scientific Python distribution"

If one of the dependency is not available as a Wheel (i.e. binary package) but only as a source package, a compiler will be required. In this case, see the next paragraph The generalization of Wheel packages should help and the installation of binary modules should become easier.

Using Christoph Gohlke repository

Christoph Gohlke, Laboratory for Fluorescence Dynamics, University of California, Irvine. He is maintaining a repository for various Python extension (actually, all we need :) for Windows. Check twice the Python version and the Windows version (win32 or win_amd64) before downloading and installing them

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

Moreover the libraries he provides are linked against the MKL library from Intel which makes his packages faster then what you would get by simply recompiling them.

Install pyFAI via PIP

The latest stable release of pyFAI should also be PIP-installable (starting at version 0.11)

:: pip install pyFAI

6.5.4 Install pyFAI from sources

The sources of pyFAI are available at https://github.com/pyFAI/pyFAI/releases

In addition to the Python interpreter, you will need the C compiler compatible with your Python interpreter, for example you can find the one for Python2.7 at: http://aka.ms/vcpython27

To upgrade the C-code in pyFAI, one needs in addition Cython:

:: pip install cython

6.6 Test suites

PyFAI comes with a test suite to ensure all core functionalities are working as expected and numerical results are correct:

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

Nota: to run the test an internet connection is needed as 200MB of test images need to be download.

6.6.1 Project structure

PyFAI is a library to deal with diffraction images for data reduction. This chapter describes the project from the computer engineering point of view.

PyFAI is an open source project licensed under the GPL mainly written in Python (v2.6, 2.7) and heavily relying on the python scientific ecosystem: numpy, scipy and matplotlib. It provides high performances image treatment thanks to cython and OpenCL... but only a C-compiler is needed to build that.

Programming language

PyFAI is a Python project but uses many programming languages:

- 15000 lines of Python (plus 5000 for the test)
- 11000 lines of Cython which are converted into ... C (about a million lines)
- 5000 lines of OpenCL kernels

The OpenCL code has been tested using:

- Nvidia OpenCL v1.1 on Linux, Windows (GPU device)
- Intel OpenCL v1.2 on Linux and Windows (CPU and ACC (Phi) devices)
- AMD OpenCL v1.2 on Linux and Windows (CPU and GPU device)
- Apple OpenCL v1.2 on MacOSX (CPU and GPU)
- Beignet OpenCL v1.2 on Linux (GPU device)
- Pocl OpenCL v1.2 on Linux (CPU device)

Repository:

The project is hosted by GitHub: https://github.com/pyFAI/pyFAI

Which provides the issue tracker in addition to Git hosting. Collaboration is done via Pull-Requests in github's web interface:

Everybody is welcome to fork the project and adapt it to his own needs: CEA-Saclay, Synchrotrons Soleil, Desy and APS have already done so. Collaboration is encouraged and new developments can be submitted and merged into the main branch via pull-requests.

Getting help

A mailing list: pyfai@esrf.fr is publicly available. To subscribe, send an email to sympa@esrf.fr with "subscribe pyfai" as subject. On this mailing list, you will have information about release of the software, new features available and meet experts to help you solve issues related to azimuthal integration. This mailing list is archived and can be consulted at: http://www.edna-site.org/lurker

Run dependencies

- Python version 2.6, 2.7 3.2 or newer
- NumPy
- SciPy
- Matplotlib
- FabIO
- h5py (optional)
- pyopencl (optional)
- fftw (optional)
- pymca (optional)
- PyQt4 or PySide (for the graphical user interface)

Build dependencies:

In addition to the run dependencies, pyFAI needs a C compiler. There is an issue with MacOSX (v10.8 onwards) where the default compiler (Xcode5 or 6) switched from gcc 4.2 to clang which dropped the support for OpenMP (clang v3.5 supports OpenMP under linux but not directly under MacOSX). Multiple solution exist, pick any of those:

- Install a recent version of GCC (>=4.2)
- Use Xcode without OpenMP, using the –no-openmp flag for setup.py.

C files are generated from cython source and distributed. Cython is only needed for developing new binary modules. If you want to generate your own C files, make sure your local Cython version supports memory-views (available from Cython v0.17 and newer), unless your Cython files will not be compiled or used.

Building procedure

As most of the python projects:

```
python setup.py build install
```

There are few specific options to setup.py:

- –no-cython: do not use cython (even if present) and use the C source code provided by the development team
- –no-openmp: if you compiler lacks OpenMP support, like Xcode on MacOSX
- –with-testimages: build the source distribution including all test images. Download 200MB of test images to create a self consistent tar-ball.

Test suites

To run the test an internet connection is needed as 200MB of test images will be downloaded.

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

Setting the environment variable http_proxy can be necessary (depending on your network):

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

PyFAI comes with 28 test-suites (163 tests in total) representing a coverage of 65%. This ensures both non regression over time and ease the distribution under different platforms: pyFAI runs under Linux, MacOSX and Windows (in each case in 32 and 64 bits). Test may not pass on computer featuring less than 2GB of memory.

Table 6.1: Test suite coverage

Name	Stmts	Miss	Cover
pyFAI.__init__	10	3	70%
pyFAI.azimuthalIntegrator	1205	330	73%
pyFAI.blob_detection	521	323	38%
pyFAI.calibrant	196	69	65%
pyFAI.detectors	993	248	75%
pyFAI.geometry	768	182	76%
pyFAI.geometryRefinement	371	205	45%
pyFAI.gui_utils	53	33	38%
pyFAI.io	421	189	55%
pyFAI.massif	187	59	68%
pyFAI.ocl_azim	307	91	70%
pyFAI.ocl_azim_csr	261	55	79%
pyFAI.ocl_azim_lut	258	55	79%
pyFAI.opencl	151	44	71%
pyFAI.peak_picker	592	439	26%
pyFAI.spline	329	220	33%
pyFAI.units	40	5	88%
pyFAI.utils	664	300	55%

Note that the test coverage tool does not count lines of Cython, nor those of OpenCL

Continuous integration is made by a home-made scripts which checks out the latest release and builds and runs the test every night. Nightly builds are available for debian6-64 bits in: http://www.edna-site.org/pub/debian/binary/

List of contributors in code

```
$ git log  --pretty='%aN##%s' | grep -v 'Merge pull' | grep -Po '^[^#]+' | sort | uniq -c | sort
```

As of 03/2015:

- Jérôme Kieffer (ESRF)
- Aurore Deschildre (ESRF)
- Frédéric-Emmanuel Picca (Soleil)
- Giannis Ashiotis (ESRF)
- Dimitrios Karkoulis (ESRF)
- Jon Wright (ESRF)
- Zubair Nawaz (Sesame)
- Amund Hov (ESRF)
- Dodogerstlin @github
- Gunthard Benecke (Desy)
- Gero Flucke (Desy)

List of other contributors (ideas or code)

- Peter Boesecke (geometry)
- Manuel Sanchez del Rio (histogramming)
- Armando Solé (masking widget + PyMca plugin)

- Sebastien Petitdemange (Lima plugin)

List of supporters

- LinkSCEEM project: porting to OpenCL
- ESRF ID11: Provided manpower in 2012 and 2013 and beamtime
- ESRF ID13: Provided manpower in 2012, 2013, 2014, 2015 and beamtime
- ESRF ID29: provided manpower in 2013 (MX-calibrate)
- ESRF ID02: provided manpower 2014
- ESRF ID15: provide manpower 2015

6.7 Environment variables

PyFAI can use a certain number of environment variable to modify its default behavior:

- PYFAI_OPENCL: set to “0” to disable the use of OpenCL
- PYFAI_DATA: path with gui, calibrant, ...
- PYFAI_TESTIMAGES: path wit test images (if absent, they get downloaded from the internet)

6.8 References:

:: _cython: http://cython.org

Author: Jérôme Kieffer

Date: 05/02/2015

Keywords: Other software related to pyFAI

CHAPTER

SEVEN

PYFAI ECOSYSTEM

7.1 Software pyFAI is relying on

7.1.1 FabIO

PyFAI is using FabIO everywhere access to a 2D images is needed. The fabio-viewer is also a lightweight convenient viewer for diffraction images.

7.1.2 PyMca

The X-ray Fluorescence Toolkit provides convenient tools for HDF5 file browsing and mask drawing.

7.2 Program using pyFAI as a library

7.2.1 Bubble

Developed for the SNBL and Dubble beamlines by Vadim DIADKIN.

7.2.2 Dahu

Dahu is a lightweight plugin based framework. Lighter then EDNA, it is technically a JSON-RPC server over Tango. Used on TRUSAXS beamline at ESRF (ID02), dahu uses pyFAI to process data up to the kHz range.

7.2.3 Dioptas

TODO ... Developed at the APS synchrotron by C. Prescher

7.2.4 Dpdak

TODO ... Developed at the Petra III synchrotron by G. Benecke and co-workers

7.2.5 EDNA

EDNA is a framework for developing plugin-based applications especially for online data analysis in the X-ray experiments field (http://edna-site.org) A EDNA data analysis server is using pyFAI as an integration engine (on the GPU) on the ESRF BioSaxs beamline, BM29. The server is running 24x7 with a processing frequency from 0.1 to 10 Hz.

7.2.6 LImA

The Library for Image Acquisition is used at many European synchrotrons to control various types of camera. A pyFAI plugin is available to integrate images on the fly without saving them.

7.2.7 NanoPeakCell

TODO ... Developed at IBS (Grenoble) by N. Coquelle

7.2.8 PySAXS

TODO ... Developed at CEA by O. Taché

CHAPTER

EIGHT

PROJECT STRUCTURE

PyFAI is a library to deal with diffraction images for data reduction. This chapter describes the project from the computer engineering point of view.

PyFAI is an open source project licensed under the GPL mainly written in Python (v2.6, 2.7) and heavily relying on the python scientific ecosystem: numpy, scipy and matplotlib. It provides high performances image treatment thanks to cython and OpenCL... but only a C-compiler is needed to build that.

8.1 Programming language

PyFAI is a Python project but uses many programming languages:

- 15000 lines of Python (plus 5000 for the test)
- 11000 lines of Cython which are converted into ... C (about a million lines)
- 5000 lines of OpenCL kernels

The OpenCL code has been tested using:

- Nvidia OpenCL v1.1 on Linux, Windows (GPU device)
- Intel OpenCL v1.2 on Linux and Windows (CPU and ACC (Phi) devices)
- AMD OpenCL v1.2 on Linux and Windows (CPU and GPU device)
- Apple OpenCL v1.2 on MacOSX (CPU and GPU)
- Beignet OpenCL v1.2 on Linux (GPU device)
- Pocl OpenCL v1.2 on Linux (CPU device)

8.2 Repository:

The project is hosted by GitHub: https://github.com/pyFAI/pyFAI

Which provides the issue tracker in addition to Git hosting. Collaboration is done via Pull-Requests in github's web interface:

Everybody is welcome to fork the project and adapt it to his own needs: CEA-Saclay, Synchrotrons Soleil, Desy and APS have already done so. Collaboration is encouraged and new developments can be submitted and merged into the main branch via pull-requests.

8.3 Getting help

A mailing list: pyfai@esrf.fr is publicly available. To subscribe, send an email to sympa@esrf.fr with "subscribe pyfai" as subject. On this mailing list, you will have information about release of the software, new features

available and meet experts to help you solve issues related to azimuthal integration. This mailing list is archived and can be consulted at: http://www.edna-site.org/lurker

8.4 Run dependencies

- Python version 2.6, 2.7 3.2 or newer
- NumPy
- SciPy
- Matplotlib
- FabIO
- h5py (optional)
- pyopencl (optional)
- fftw (optional)
- pymca (optional)
- PyQt4 or PySide (for the graphical user interface)

8.5 Build dependencies:

In addition to the run dependencies, pyFAI needs a C compiler. There is an issue with MacOSX (v10.8 onwards) where the default compiler (Xcode5 or 6) switched from gcc 4.2 to clang which dropped the support for OpenMP (clang v3.5 supports OpenMP under linux but not directly under MacOSX). Multiple solution exist, pick any of those:

- Install a recent version of GCC (>=4.2)
- Use Xcode without OpenMP, using the –no-openmp flag for setup.py.

C files are generated from cython source and distributed. Cython is only needed for developing new binary modules. If you want to generate your own C files, make sure your local Cython version supports memory-views (available from Cython v0.17 and newer), unless your Cython files will not be compiled or used.

8.6 Building procedure

8.6.1 As most of the python projects:

python setup.py build install

There are few specific options to setup.py:

- –no-cython: do not use cython (even if present) and use the C source code provided by the development team
- –no-openmp: if you compiler lacks OpenMP support, like Xcode on MacOSX
- –with-testimages: build the source distribution including all test images. Download 200MB of test images to create a self consistent tar-ball.

8.7 Test suites

8.7.1 To run the test an internet connection is needed as 200MB of test images will be downloaded.

python setup.py build test

Setting the environment variable http_proxy can be necessary (depending on your network):

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

PyFAI comes with 28 test-suites (163 tests in total) representing a coverage of 65%. This ensures both non regression over time and ease the distribution under different platforms: pyFAI runs under Linux, MacOSX and Windows (in each case in 32 and 64 bits). Test may not pass on computer featuring less than 2GB of memory.

Table 8.1: Test suite coverage

Name	Stmts	Miss	Cover
pyFAI/__init__	12	3	75%
pyFAI/_version	31	1	97%
pyFAI/azimuthalIntegrator	1193	311	74%
pyFAI/blob_detection	520	323	38%
pyFAI/calibrant	197	60	70%
pyFAI/detectors	1034	274	74%
pyFAI/directories	30	8	73%
pyFAI/geometry	808	203	75%
pyFAI/geometryRefinement	477	304	36%
pyFAI/gui_utils	66	41	38%
pyFAI/io	453	212	53%
pyFAI/massif	188	60	68%
pyFAI/ocl_azim	269	78	71%
pyFAI/ocl_azim_csr	225	50	78%
pyFAI/ocl_azim_lut	219	45	79%
pyFAI/opencl	191	52	73%
pyFAI/peak_picker	707	516	27%
pyFAI/spline	397	249	37%
pyFAI/test/__init__	19	2	89%
pyFAI/test/test_all	77	7	91%
pyFAI/test/test_azimuthal_integrator	241	67	72%
pyFAI/test/test_bilinear	80	8	90%
pyFAI/test/test_bispev	66	16	76%
pyFAI/test/test_blob_detection	54	5	91%
pyFAI/test/test_bug_regression	41	5	88%
pyFAI/test/test_calibrant	84	25	70%
pyFAI/test/test_convolution	54	6	89%
pyFAI/test/test_csr	88	23	74%
pyFAI/test/test_detector	137	12	91%
pyFAI/test/test_distortion	56	8	86%
pyFAI/test/test_dummy	27	6	78%
pyFAI/test/test_export	87	9	90%
pyFAI/test/test_flat	112	9	92%
pyFAI/test/test_geometry	91	6	93%
pyFAI/test/test_geometry_refinement	64	7	89%
pyFAI/test/test_histogram	228	17	93%
pyFAI/test/test_integrate	139	12	91%
pyFAI/test/test_io	108	30	72%

Continued on next page

Table 8.1 – continued from previous page

Name	Stmts	Miss	Cover
pyFAI/test/test_marchingsquares	42	9	79%
pyFAI/test/test_mask	137	29	79%
pyFAI/test/test_openCL	196	22	89%
pyFAI/test/test_peak_picking	88	11	88%
pyFAI/test/test_polarization	57	6	89%
pyFAI/test/test_saxs	105	31	70%
pyFAI/test/test_sparse	44	5	89%
pyFAI/test/test_split_pixel	74	6	92%
pyFAI/test/test_utils	96	6	94%
pyFAI/test/utilstest	281	164	42%
pyFAI/third_party/__init__	0	0	100%
pyFAI/third_party/six	393	184	53%
pyFAI/units	41	5	88%
pyFAI/utils	718	316	56%
TOTAL	11142	3864	65%

Note that the test coverage tool does not count lines of Cython, nor those of OpenCL

Continuous integration is made by a home-made scripts which checks out the latest release and builds and runs the test every night. Nightly builds are available for debian6-64 bits in: http://www.edna-site.org/pub/debian/binary/

8.8 List of contributors in code

```
$ git log  --pretty='%aN##%s' | grep -v 'Merge pull' | grep -Po '^[^#]+' | sort | uniq -c | sort
```

As of 03/2015:

- Jérôme Kieffer (ESRF)
- Aurore Deschildre (ESRF)
- Frédéric-Emmanuel Picca (Soleil)
- Giannis Ashiotis (ESRF)
- Dimitrios Karkoulis (ESRF)
- Jon Wright (ESRF)
- Zubair Nawaz (Sesame)
- Amund Hov (ESRF)
- Dodogerstlin @github
- Gunthard Benecke (Desy)
- Gero Flucke (Desy)

8.9 List of other contributors (ideas or code)

- Peter Boesecke (geometry)
- Manuel Sanchez del Rio (histogramming)
- Armando Solé (masking widget + PyMca plugin)
- Sebastien Petitdemange (Lima plugin)

8.10 List of supporters

- LinkSCEEM project: porting to OpenCL
- ESRF ID11: Provided manpower in 2012 and 2013 and beamtime
- ESRF ID13: Provided manpower in 2012, 2013, 2014, 2015 and beamtime
- ESRF ID29: provided manpower in 2013 (MX-calibrate)
- ESRF ID02: provided manpower 2014
- ESRF ID15: provide manpower 2015

CHAPTER

NINE

INDICES AND TABLES

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

BIBLIOGRAPHY

[SRI2012] PyFAI, a versatile library for azimuthal regrouping J. Kieffer & D. Karkoulis J. Phys.: Conf. Ser. 425 202012 http://dx.doi.org/10.1088/1742-6596/425/20/202012

[EPDIC13] PyFAI: a Python library for high performance azimuthal integration on GPU J. Kieffer & J. P. Wright, Powder Diffraction / Volume 28 / Supplement S2 / September 2013, pp S339-S350 http://dx.doi.org/10.1017/S0885715613000924

[FIT2D] Hammersley A. P., Svensson S. O., Hanfland M., Fitch A. N. and Hausermann D. 1996 High Press. Res. vol14 p235–248

[SPD] Bösecke P. 2007 J. Appl. Cryst. vol40 s423–s427

[EDNA] Incardona M. F., Bourenkov G. P., Levik K., Pieritz R. A., Popov A. N. and Svensson O. 2009 J. Synchrotron Rad. vol16 p872–879

[PyMca] Solé V. A., Papillon E., Cotte M., Walter P. and Susini J. 2007 Spectrochim. Acta Part B vol vol62 p63 – 68

[PyNX] Favre-Nicolin V., Coraux J., Richard M. I. and Renevier H. 2011 J. Appl. Cryst. vol44 p635–640

[IPython] Pérez F and Granger B E 2007 Comput. Sci. Eng. vol9 p21–29 URL http://ipython.org

[NumPy] Oliphant T E 2007 Comput. Sci. Eng. vol9 p10–20

[Cython] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn D and Smith K 2011 Comput. Sci. Eng. vol13 p31 –39

[OpenCL] Khronos OpenCL Working Group 2010 The OpenCL Specification, version 1.1 URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[FabIO] Sorensen H O, Knudsen E, Wright J, Kieffer J et al. 2007–2013 FabIO: I/O library for images produced by 2D X-ray detectors URL http://fable.sf.net/

[Matplotlib] Hunter J D 2007 Comput. Sci. Eng. vol9 p90–95 ISSN 1521-9615

[SciPy] Jones E, Oliphant T, Peterson P et al. 2001– SciPy: Open source scientific tools for Python URL http://www.scipy.org/

[FFTw] Frigo M and Johnson S G 2005 Proceedings of the IEEE 93 p 216–231

[LImA] The LIMA Project Update S. Petitdemange, L. Claustre, A. Homs, R. Homs Regojo, E. Papillon Proceedings of ICALEPCS2013 http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/html/auth1084.htm

[PyOpenCL] PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, Parallel Computing Vol 38, 3, March 2012, Pages 157–174 http://dx.doi.org/10.1016/j.parco.2011.09.001

[AMD] The American Mineralogist Crystal Structure Database. Downs, R.T. and Hall-Wallace, M. (2003) American Mineralogist 88, 247-250 http://rruff.geo.arizona.edu/AMS/amcsd.php

[COD] Crystallography Open Database: an open-access collection of crystal structures and platform for world-wide collaboration Saulius Grazulis et al. Nucl. Acids Res. (2012) 40 (D1): D420-D427. http://dx.doi.org/10.1093/nar/gkr900 http://www.crystallography.net/

[Dpdak] A customizable software for fast reduction and analysis of large X-ray scattering data sets: applications of the new DPDAK package to small angle X-ray scattering and grazing-incidence small angle X-ray scattering, Benecke, G. et al., (2014) J. Appl. Cryst. 47, http://dx.doi.org/10.1107/S1600576714019773

PYTHON MODULE INDEX

p