pyFAI Documentation

Release 0.9.4

Jerome Kieffer

June 04, 2014

CONTENTS

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.

The second chapter is the concatenation of the manual pages of all (relevant) scripts. Those are programs to be launched at the command line allowing the treatment of a diffraction experiment without knowing anything about Python.

The third chapter contains a comprehensive description of most Python 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.

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 Geometry and calibration

PyFAI and spd share the same 6-parameter geometry definition: distance, point of normal incidence (2 coordinates) and 3 rotations around the main axis; these parameters are saved in text files usually with the *.poni* extension. 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, ...). Alternatively, geometries calibrated using fit2d can be imported into pyFAI, including geometric distortions (i.e. optical-fiber tapers distortion) described as *spline-files*.

1.3.1 Geometry

PyFAI takes diffraction images as 2D numpy arrays. The origin of the image is considered to be at the lower-left corner of the image. 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)

(p1,2) = (y, x)

In the same idea rot1, rot2 and rot3 are rotation along axis 1, 2 and 3, always expressed in radians. Axis 3 is built in such a way to be orthogonal and (1,2,3) is a direct orientation Rotations applied in the same order: rot1 then rot2 then rot3.

Poni1 amd poni2 are distances in meter, like the distance, letting the calibration be independent to pixel size hence to binning.

1.3.2 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.

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.

1.3.3 PyFAI executables

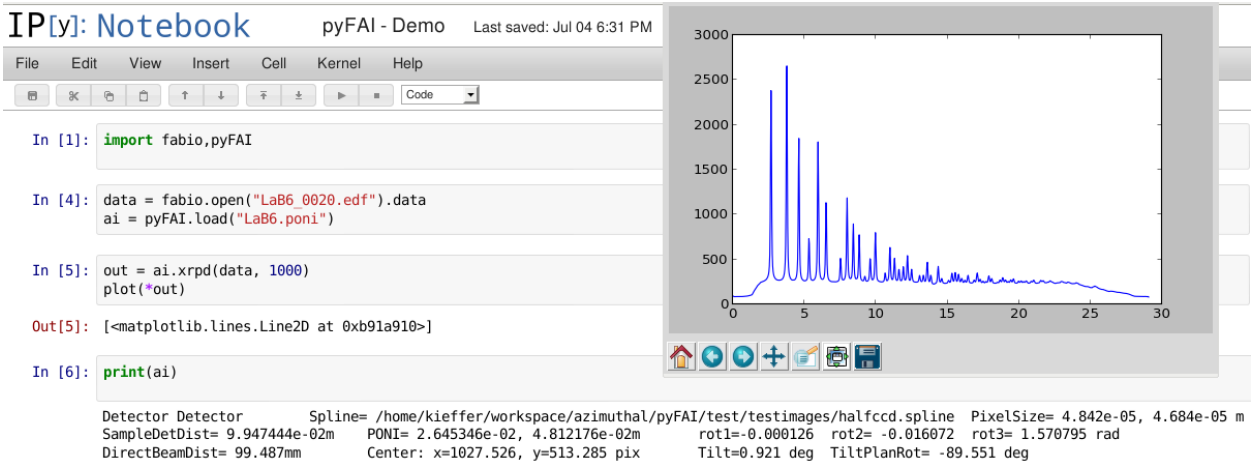
PyFAI was designed to be used by scientists needing a simple and effective tool for azimuthal integration. Two command line programs *pyFAI-waxs* and *pyFAI-saxs* are provided with pyFAI for performing the integration of one or more images on the command line. The waxs version outputs result in $2\theta/I$, 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)

1.3.4 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.



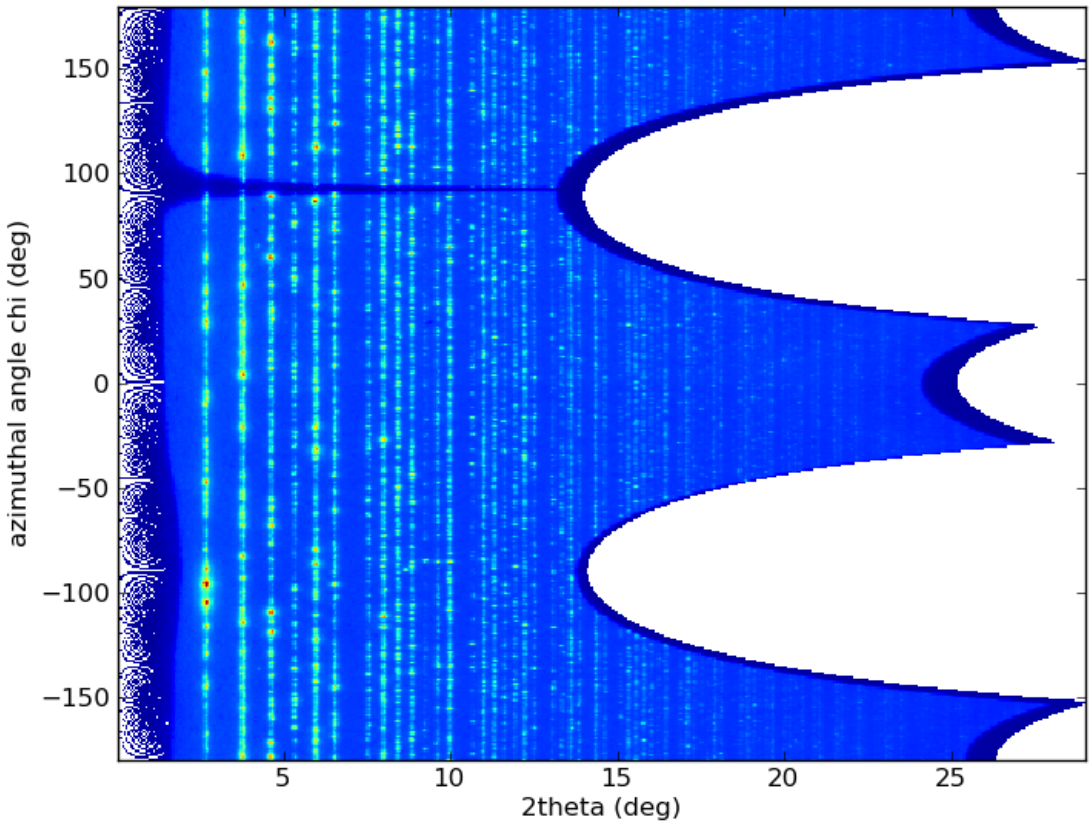
1.4 Regrouping mechanism

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.



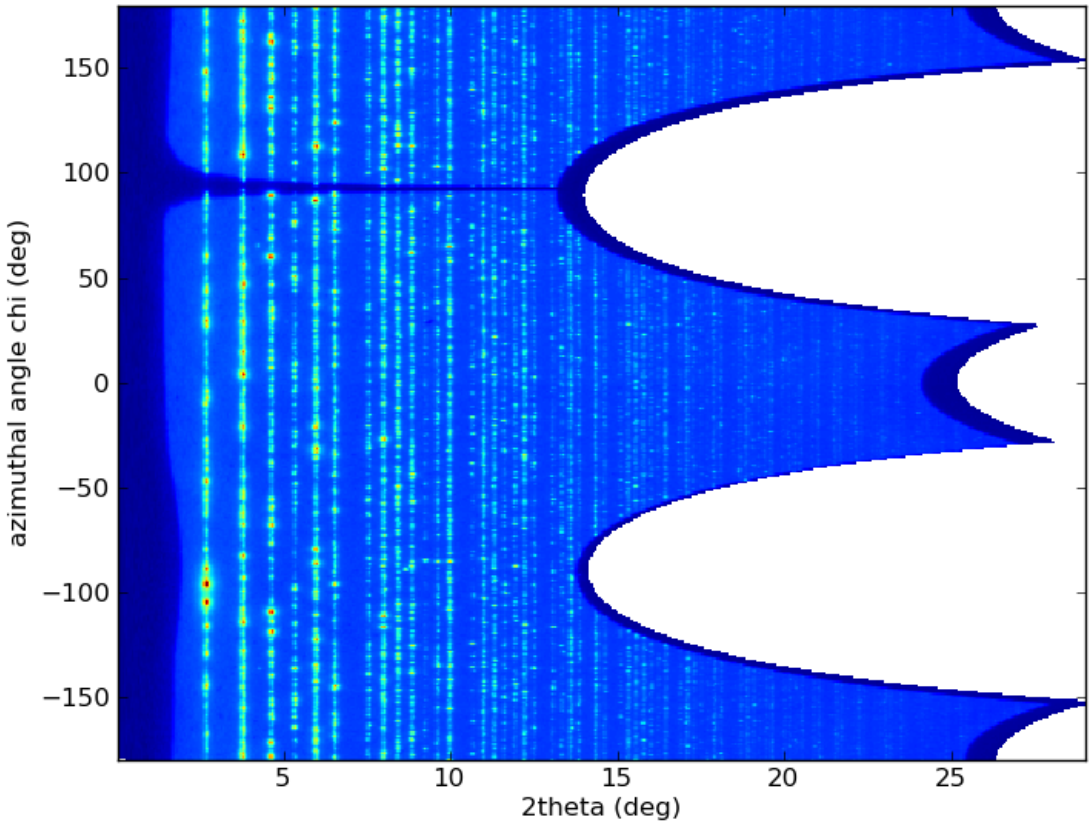
PyFAI solves this problem by pixel splitting : in addition to the pixel position, its spatial extension is calculated and each pixel is then split and distributed over the corresponding bins, the intensity being considered as homogeneous within a pixel and spread accordingly. The drawback of this is 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 favo2r 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


150
100
50
0
−50
−100
−150
azimuthal angle chi (deg)
5
10
15
20
25
2theta (deg)

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 megapixel 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 Conclusion

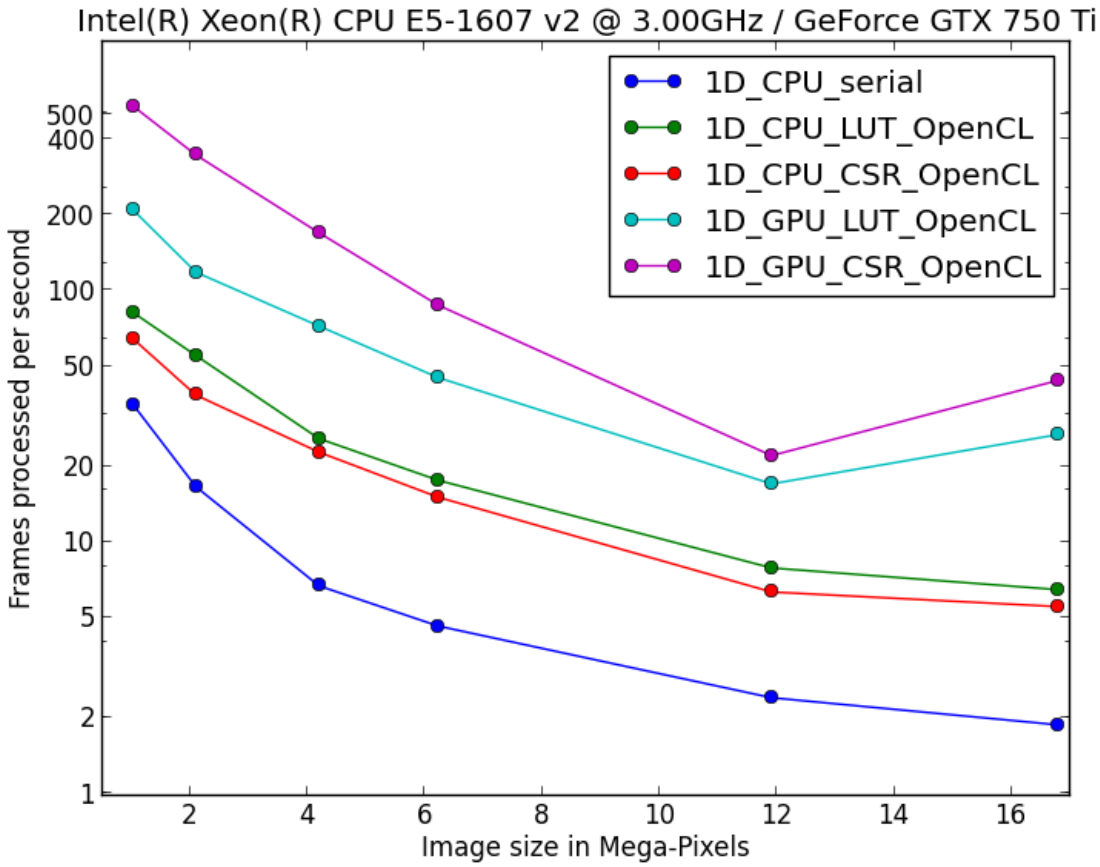
The library pyFAI was developed with two main goals:

- 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.

1.5.1 Acknowledgments

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


Intel(R) Xeon(R) CPU E5-1607 v2 @ 3.00GHz / GeForce GTX 750 Ti
1D_CPU_serial
1D_CPU_LUT_OpenCL
1D_CPU_CSR_OpenCL
1D_GPU_LUT_OpenCL
1D_GPU_CSR_OpenCL
Frames processed per second
500
400
200
100
50
20
10
5
2
1
2
4
6
8
10
12
14
16
Image size in Mega-Pixels

CHAPTER

TWO

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
- 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

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

2.1 Preprocessing tool: pyFAI-average

2.1.1 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 spotty-ness of Debye-Scherrer rings. In this case the “max-filter” is usually recommended.

2.1.2 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

2.2 Mask generation tool: drawMask_pymca

2.2.1 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 ...

2.2.2 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)

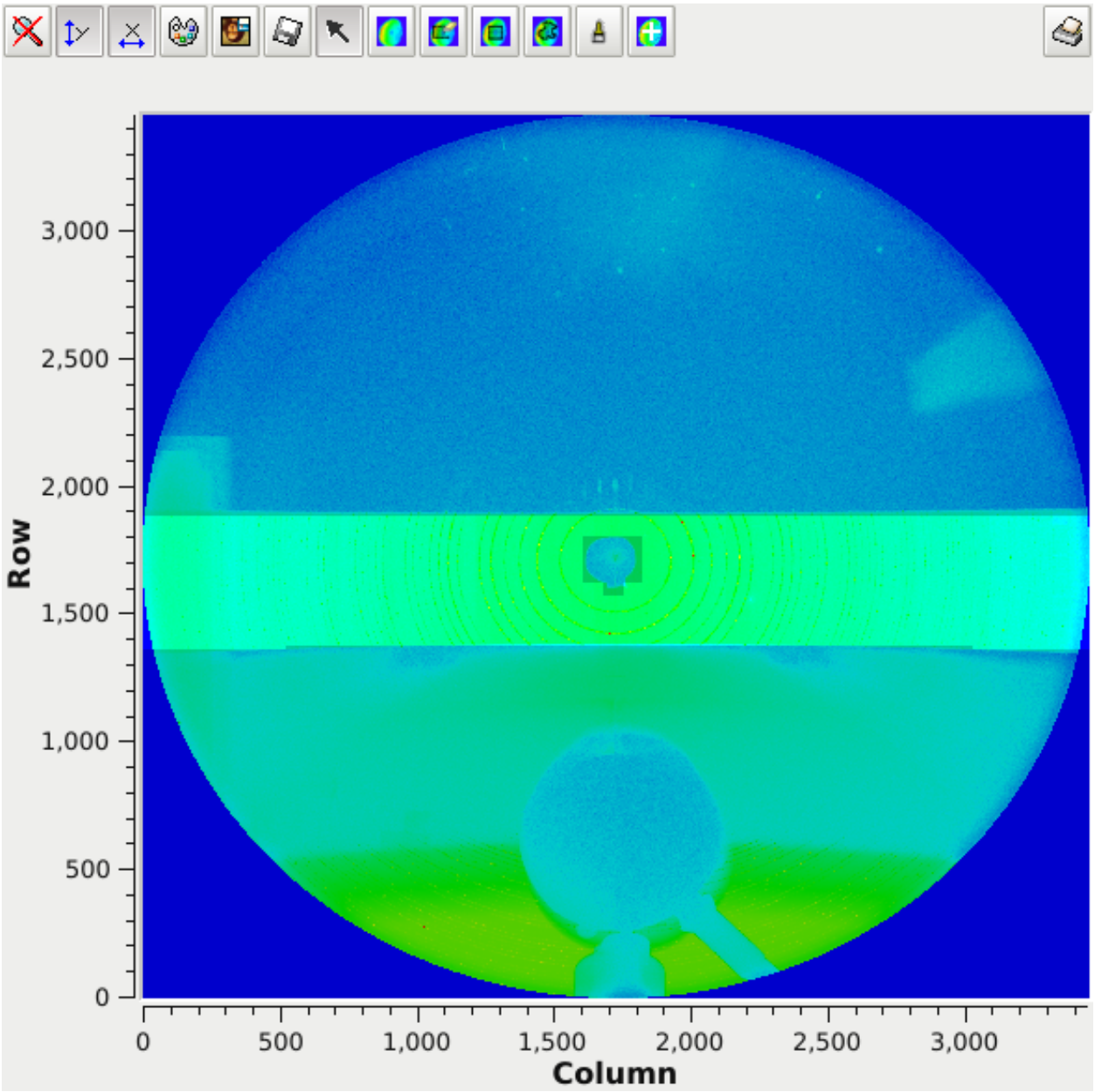
2.3 Calibration tool: pyFAI-calib

2.3.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images without a priori knowledge of your setup. 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].

You will need in addition:

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


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

- 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
- 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)

2.3.2 Usage:

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

2.3.3 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

2.3.4 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

2.4 Calibration tool: pyFAI-recalib

2.4.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images with a priori knowledge of your setup (an input PONI-file). If you are using a standard calibrant, look at the list provided by pyFAI 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].

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).

2.4.2 Usage:

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

2.4.3 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

2.4.4 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).

2.5 Calibration tool: check_calib

2.5.1 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.

2.5.2 Usage:

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

2.5.3 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

2.5.4 Arguments:

FILE Image file to check calibration for

2.6 Calibration tool: MX-calibrate

2.6.1 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].

2.6.2 Usage:

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

2.6.3 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)

2.7 Integration tool: pyFAI-integrate

2.7.1 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.

2.7.2 Usage

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

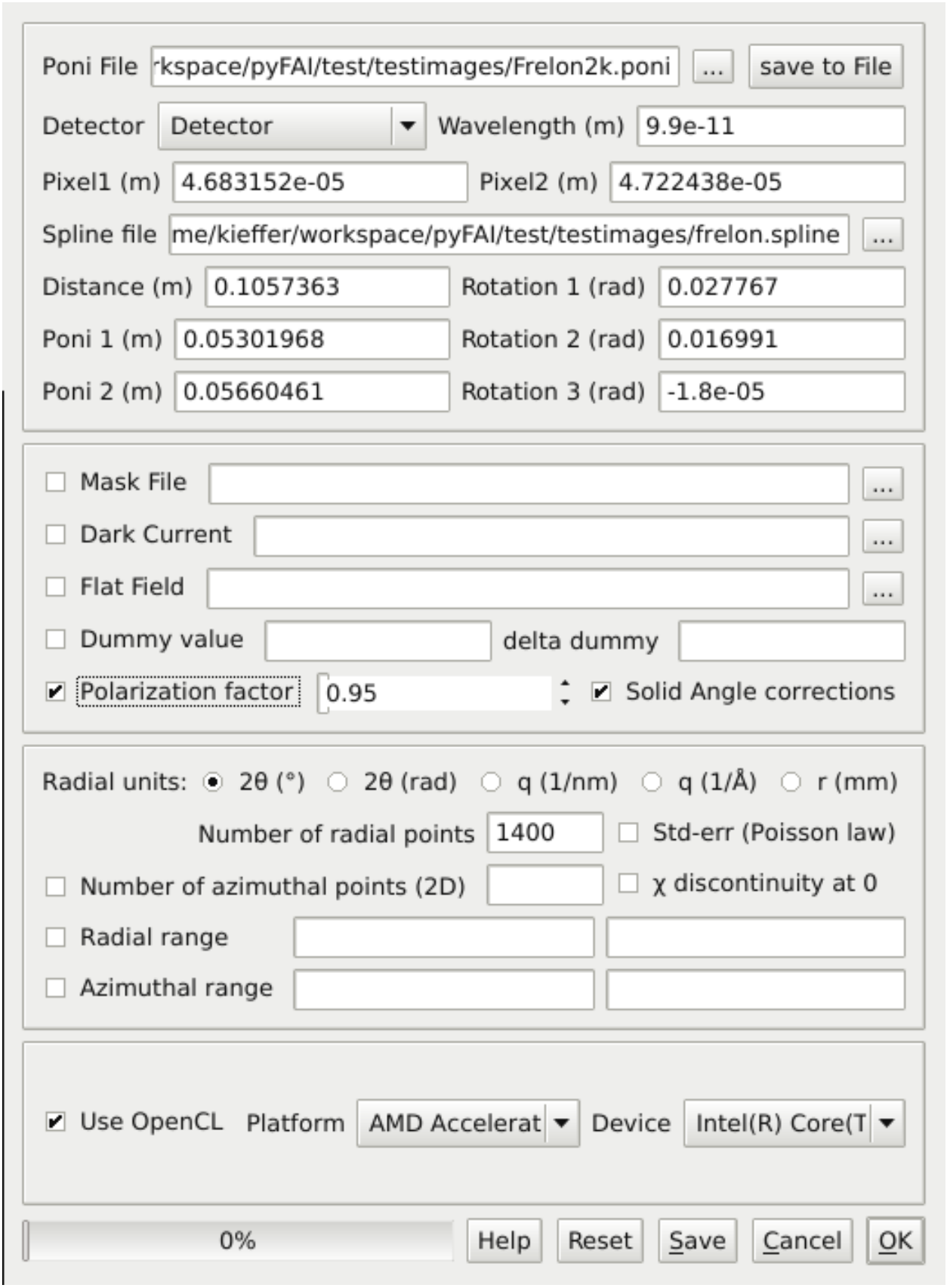
2.7.3 Options:

--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


Poni File
rkspace/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

2.7.4 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

2.8 Integration tool: diff_tomo

2.8.1 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"

2.8.2 Usage:

diff_tomo [options] -p ponifile imagefiles*

2.8.3 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.

2.9 Integration tool: pyFAI-saxs

2.9.1 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.

2.9.2 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)

2.10 Integration tool: pyFAI-saxs

2.10.1 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.

2.10.2 Usage:

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

2.10.3 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

THREE

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.

3.1 pyFAI Package

3.2 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, nbPt, unit="2th_deg")
>>> q, I, sigma = ai.integrate1d(data, nbPt, unit="q_nm^-1", error_model="poisson")
>>> regrouped = ai.integrate2d(data, nbPt_rad, nbPt_azim, unit="q_nm^-1")[0]
```

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

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

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_flatfield()

get_mask()

get_maskfile()

integrate1d(*data*, *nbPt*, *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
- **nbPt** (*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; "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*, *nbPt_rad*, *nbPt_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
- **nbPt_rad** (*int*) – number of points in the radial direction
- **nbPt_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

makeMask(*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 adjuste the level, with the *dummy* and the *delta_dummy* parameter, when you considere 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.

mask

maskfile

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(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *variance=None*, *error_model=None*, *qRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*, *method='bbox'*, *unit=q_nm^-1*)
Calculate the azimuthal integrated Saxs curve in q in nm^-1.

Wrapper for integrate1d emulating behavour of old saxs method

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera

- **nbPt** (*int*) – number of points in the output pattern
- **filename** (*str*) – file to save data to
- **correctSolidAngle** (*bool*) – if True, the data are devided by the solid angle of each pixel
- **variance** (*ndarray*) – array containing the variance of the data, if you know it
- **error_model** (*str*) – When the variance is unknown, an error model can be given: "poisson" (variance = I), "azimuthal" (variance = (I-<I>)^2)
- **qRange** ((*float, float), optional*) – The lower and upper range of the sctter vector q. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **chiRange** ((*float, float), optional*) – The lower and upper range of the chi angle. 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 and +1. 0 for no correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image
- **method** (*str*) – can be "numpy", "cython", "BBox" or "splitpixel"

Returns azimuthaly regrouped data, 2theta pos. and chi pos.

Return type 3-tuple of ndarrays

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_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

set_mask(*mask*)

set_maskfile(*maskfile*)

setup_CSR(*shape*, *nbPt*, *mask=None*, *pos0_range=None*, *pos1_range=None*, *mask_checksum=None*, *unit=2th_deg*, *padding=1*)

Prepare a look-up-table

Parameters

- **shape** (*(int, int)*) – shape of the dataset
- **nbPt** (*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 *nbPt* 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: (nbPt, 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*, *nbPt*, *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
- **nbPt** (*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 *nbPt* 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: (nbPt, 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(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the powder diffraction pattern from a set of data, an image.

Cython implementation

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float), optional*) – The lower and upper range of the 2theta
- **chiRange** ((*float, float), optional, disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns (2theta, I) in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image as also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern. you just need to set the range with the *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same). Pixels can also be maseked by seting them to an

Bad pixels can be masked out by setting them to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

xrpd2(*data*, *nbPt2Th*, *nbPtChi=360*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the 2D powder diffraction pattern (2Theta,Chi) from a set of data, an image

Split pixels according to their coordinate and a bounding box

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt2Th** – number of bin of the Radial (horizontal) axis (2Theta)
- **nbPtChi** (*int*) – number of bin of the Azimuthal (vertical) axis (chi)
- **filename** (*str*) – file to save data in
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float)*) – The lower and upper range of 2theta
- **chiRange** ((*float, float), disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns azimuthaly regrouped data, 2theta pos. and chi pos.

Return type 3-tuple of ndarrays

This method convert the *data* image from the pixel coordinates to the 2theta, chi coordinates. This is similar to a rectangular to polar conversion. The number of point of the new image is given by *nbPt2Th* and *nbPtChi*. If you give a *filename*, the new image is also saved as an edf file.

It is possible to correct the 2theta/chi pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta and range of the new image can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has a 2theta and a chi coordinate. So it is possible to restrain on any of those ranges ; you just need to set the range with the *tthRange* or thee *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Masking can also be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

the polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

xrpd2_histogram(*data*, *nbPt2Th*, *nbPtChi=360*, *filename=None*, *correctSolidAngle=True*, *dark=None*, *flat=None*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*)

Calculate the 2D powder diffraction pattern (2Theta,Chi) from a set of data, an image

Cython implementation: fast but incaccurate

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt2Th** – number of bin of the Radial (horizontal) axis (2Theta)
- **nbPtChi** (*int*) – number of bin of the Azimuthal (vertical) axis (chi)
- **filename** (*str*) – file to save data in
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** (*(float, float)*) – The lower and upper range of 2theta
- **chiRange** (*(float, float), disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value

Returns azimuthaly regrouped data, 2theta pos and chipos

Return type 3-tuple of ndarrays

This method convert the *data* image from the pixel coordinates to the 2theta, chi coordinates. This is simular to a rectangular to polar conversion. The number of point of the new image is given by *nbPt2Th* and *nbPtChi*. If you give a *filename*, the new image is also saved as an edf file.

It is possible to correct the 2theta/chi pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta and range of the new image can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has a 2theta and a chi coordinate. So it is possible to restrain on any of those ranges ; you just need to set the range with the *tthRange* or thee *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Masking can also be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

xrpd2_numpy(*data*, *nbPt2Th*, *nbPtChi=360*, *filename=None*, *correctSolidAngle=True*, *dark=None*, *flat=None*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*)

Calculate the 2D powder diffraction pattern (2Theta, Chi) from a set of data, an image

Pure numpy implementation (VERY SLOW !!!)

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt2Th** – number of bin of the Radial (horizontal) axis (2Theta)
- **nbPtChi** (*int*) – number of bin of the Azimuthal (vertical) axis (chi)
- **filename** (*str*) – file to save data in
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** (*(float, float)*) – The lower and upper range of 2theta
- **chiRange** (*(float, float), disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value

Returns azimuthaly regrouped data, 2theta pos and chipos

Return type 3-tuple of ndarrays

This method convert the *data* image from the pixel coordinates to the 2theta, chi coordinates. This is simular to a rectangular to polar conversion. The number of point of the new image is given by *nbPt2Th* and *nbPtChi*. If you give a *filename*, the new image is also saved as an edf file.

It is possible to correct the 2theta/chi pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta and range of the new image can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has a 2theta and a chi coordinate. So it is possible to restrain on any of those ranges ; you just need to set the range with the *tthRange* or thee *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Masking can also be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

xrpd2_splitBBox(*data*, *nbPt2Th*, *nbPtChi=360*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the 2D powder diffraction pattern (2Theta,Chi) from a set of data, an image

Split pixels according to their coordinate and a bounding box

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt2Th** – number of bin of the Radial (horizontal) axis (2Theta)
- **nbPtChi** (*int*) – number of bin of the Azimuthal (vertical) axis (chi)
- **filename** (*str*) – file to save data in
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** (*(float, float)*) – The lower and upper range of 2theta

- **chiRange** ((*float, float), disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns azimuthaly regrouped data, 2theta pos. and chi pos.

Return type 3-tuple of ndarrays

This method convert the *data* image from the pixel coordinates to the 2theta, chi coordinates. This is similar to a rectangular to polar conversion. The number of point of the new image is given by *nbPt2Th* and *nbPtChi*. If you give a *filename*, the new image is also saved as an edf file.

It is possible to correct the 2theta/chi pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta and range of the new image can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has a 2theta and a chi coordinate. So it is possible to restrain on any of those ranges ; you just need to set the range with the *tthRange* or thee *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Masking can also be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

the polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

xrpd2_splitPixel(*data*, *nbPt2Th*, *nbPtChi=360*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the 2D powder diffraction pattern (2Theta,Chi) from a set of data, an image

Split pixels according to their corner positions

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt2Th** – number of bin of the Radial (horizontal) axis (2Theta)
- **nbPtChi** (*int*) – number of bin of the Azimuthal (vertical) axis (chi)
- **filename** (*str*) – file to save data in
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float)*) – The lower and upper range of 2theta

- **chiRange** ((*float, float), disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns azimuthaly regrouped data, 2theta pos. and chi pos.

Return type 3-tuple of ndarrays

This method convert the *data* image from the pixel coordinates to the 2theta, chi coordinates. This is similar to a rectangular to polar conversion. The number of point of the new image is given by *nbPt2Th* and *nbPtChi*. If you give a *filename*, the new image is also saved as an edf file.

It is possible to correct the 2theta/chi pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta and range of the new image can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has a 2theta and a chi coordinate. So it is possible to restrain on any of those ranges ; you just need to set the range with the *tthRange* or thee *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Masking can also be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

the polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

xrpd_CSR_OCL(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *dark=None*, *flat=None*, *chiRange=None*, *safe=True*, *devicetype='all'*, *platformid=None*, *deviceid=None*, *block_size=32*, *padded=False*)

Calculate the powder diffraction pattern from a set of data, an image.

PyOpenCL implementation using a CSR version of the Look-Up Table (OpenCL). The look-up table is a Cython module.

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float)*) – The lower and upper range of 2theta

- **chiRange** ((*float, float)*) – The lower and upper range of the chi angle in degrees.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value

LUT specific parameters:

Parameters **safe** (*bool*) – set to False if your LUT & GPU is already set-up correctly

OpenCL specific parameters:

Parameters

- **devicetype** (*str*) – can be "all", "cpu", "gpu", "acc" or "def"
- **platformid** (*int*) – platform number
- **deviceid** (*int*) – device number

Returns (2theta, I) in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern by setting the *chiRange* parameter. Like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Dynamic masking (i.e recalculated for each image) can be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Dynamic masking is computed at integration whereas static masking is done at LUT-generation, hence faster.

Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The *safe* parameter is specific to the OpenCL/LUT implementation, you can set it to false if you think the LUT calculated is already the correct one (setup, mask, 2theta/chi range) and the device set-up is the expected one.

devicetype, *platformid* and *deviceid*, parameters are specific to the OpenCL implementation. If you set *devicetype* to 'all', 'cpu', or 'gpu' you can force the device used to perform the computation. By providing the *platformid* and *deviceid* you can chose a specific device (computer specific).

xrpd_LUT(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *safe=True*, *dark=None*, *flat=None*)

Calculate the powder diffraction pattern from an image.

Parallel Cython implementation using a Look-Up Table (OpenMP).

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** (*(float, float), optional*) – The lower and upper range of the 2theta angle
- **chiRange** (*(float, float), optional*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value

LUT specific parameters:

Parameters **safe** (*bool*) – set to False if your LUT is already set-up correctly (mask, ranges, ...).

Returns (2theta, I) with 2theta angle in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image as also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern by setting the range with the *chiRange* parameter. Like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Dynamic masking (i.e recalculated for each image) can be achieved by setting masked pixels to an impossible value (-1) and calling this value the "dummy value". Dynamic masking is computed at integration whereas static masking is done at LUT-generation, hence faster.

Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The *safe* parameter is specific to the LUT implementation, you can set it to false if you think the LUT calculated is already the correct one (setup, mask, 2theta/chi range).

TODO: replace with inegrate1D

xrpd_LUT_OCL(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *safe=True*, *devicetype='all'*, *platformid=None*, *deviceid=None*, *dark=None*, *flat=None*)

Calculate the powder diffraction pattern from a set of data, an image.

PyOpenCL implementation using a Look-Up Table (OpenCL). The look-up table is a Cython module.

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float)*) – The lower and upper range of 2theta
- **chiRange** ((*float, float)*) – The lower and upper range of the chi angle in degrees.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value

LUT specific parameters:

Parameters **safe** (*bool*) – set to False if your LUT & GPU is already set-up correctly

OpenCL specific parameters:

Parameters

- **devicetype** (*str*) – can be “all”, “cpu”, “gpu”, “acc” or “def”
- **platformid** (*int*) – platform number
- **deviceid** (*int*) – device number

Returns (2theta, I) in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern by setting the *chiRange* parameter. Like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Dynamic masking (i.e recalculated for each image) can be achieved by setting masked pixels to an impossible value (-1) and calling this value the “dummy value”. Dynamic masking is computed at integration whereas static masking is done at LUT-generation, hence faster.

Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The *safe* parameter is specific to the OpenCL/LUT implementation, you can set it to false if you think the LUT calculated is already the correct one (setup, mask, 2theta/chi range) and the device set-up is the expected one.

devicetype, *platformid* and *deviceid*, parameters are specific to the OpenCL implementation. If you set *devicetype* to 'all', 'cpu', or 'gpu' you can force the device used to perform the computation. By providing the *platformid* and *deviceid* you can chose a specific device (computer specific).

xrpd_OpenCL(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *dark=None*, *flat=None*, *tthRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *devicetype='gpu'*, *useFp64=True*, *platformid=None*, *deviceid=None*, *safe=True*)

Calculate the powder diffraction pattern from a set of data, an image.

This is (now) a pure pyopencl implementation so it just needs pyopencl which requires a clean OpenCL installation. This implementation is not slower than the previous Cython and is less problematic for compilation/installation.

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float), optional*) – The lower and upper range of the 2theta
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value

OpenCL specific parameters:

Parameters

- **devicetype** (*str*) – possible values "cpu", "gpu", "all" or "def"
- **useFp64** (*bool*) – shall histogram be done in double precision (strongly adviced)
- **platformid** (*int*) – platform number
- **deviceid** (*int*) – device number
- **safe** (*bool*) – set to False if your GPU is already set-up correctly

Returns (2theta, I) angle being in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file. The powder diffraction is computed internally using an histogram which by default use should be done in 64bits. One can switch to 32 bits with the *useFp64* parameter set to False. In 32bit mode; do not expect better than 1% error and one can even observe overflows ! 32 bits is only left for testing hardware capabilities and should NEVER be used in any real experiment analysis.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image has also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern. You just need to set the range with the *chiRange* parameter; like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Bad pixels can also be masked by setting them to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

devicetype, *platformid* and *deviceid*, parameters are specific to the OpenCL implementation. If you set *devicetype* to 'all', 'cpu', 'gpu', 'def' you can force the device used to perform the computation; the program will select the device accordinly. By setting *platformid* and *deviceid*, you can directly address a specific device (which is computer specific).

The *safe* parameter is specific to the integrator object, located on the OpenCL device. You can set it to False if you think the integrator is already setup correcty (device, geometric arrays, mask, 2theta/chi range). Unless many tests will be done at each integration.

xrpd_cython(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*, *pixelSize=None*)

Calculate the powder diffraction pattern from a set of data, an image.

Cython multithreaded implementation: fast but still lacks pixels splitting as numpy implementation. This method should not be used in production, it remains to explain why histograms are hard to implement in parallel. Use xrpd_splitBBox instead

xrpd_numpy(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the powder diffraction pattern from a set of data, an image.

Numpy implementation: slow and without pixels splitting. This method should not be used in production, it remains to explain how other more sophisticated algorithms works. Use xrpd_splitBBox instead

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float), optional*) – The lower and upper range of the 2theta
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns (2theta, I) in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Bad pixels can be masked out by setting them to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

xrpd_splitBBox(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the powder diffraction pattern from a set of data, an image.

Cython implementation

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float), optional*) – The lower and upper range of the 2theta
- **chiRange** ((*float, float), optional, disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns (2theta, I) in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image as also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern. you just need to set the range with the *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same). Pixels can also be maseked by seting them to an

Bad pixels can be masked out by setting them to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

xrpd_splitPixel(*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*)

Calculate the powder diffraction pattern from a set of data, an image.

Cython implementation (single threaded)

Parameters

- **data** (*ndarray*) – 2D array from the CCD camera
- **nbPt** (*integer*) – number of points in the output pattern
- **filename** (*str*) – file to save data in ascii format 2 column
- **correctSolidAngle** (*bool or int*) – solid angle correction, order 1 or 3 (like fit2d)
- **tthRange** ((*float, float), optional*) – The lower and upper range of the 2theta
- **chiRange** ((*float, float), optional, disabled for now*) – The lower and upper range of the chi angle.
- **mask** (*ndarray*) – array with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels (dynamic mask)
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image

Returns (2theta, I) in degrees

Return type 2-tuple of 1D arrays

This method compute the powder diffraction pattern, from a given *data* image. The number of point of the pattern is given by the *nbPt* parameter. If you give a *filename*, the powder diffraction is also saved as a two column text file.

It is possible to correct or not the powder diffraction pattern using the *correctSolidAngle* parameter. The weight of a pixel is ponderate by its solid angle.

The 2theta range of the powder diffraction pattern can be set using the *tthRange* parameter. If not given the maximum available range is used. Indeed pixel outside this range are ignored.

Each pixel of the *data* image as also a chi coordinate. So it is possible to restrain the chi range of the pixels to consider in the powder diffraction pattern. you just need to set the range with the *chiRange* parameter. like the *tthRange* parameter, value outside this range are ignored.

Sometimes one needs to mask a few pixels (beamstop, hot pixels, ...), to ignore a few of them you just need to provide a *mask* array with a value of 1 for those pixels. To take a pixel into account you just need to set a value of 0 in the mask array. Indeed the shape of the mask array should be idential to the data shape (size of the array _must_ be the same).

Bad pixels can be masked out by setting them to an impossible value (-1) and calling this value the "dummy value". Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

Some Pilatus detectors are setting non existing pixel to -1 and dead pixels to -2. Then use dummy=-2 & delta_dummy=1.5 so that any value between -3.5 and -0.5 are considered as bad.

The polarisation correction can be taken into account with the *polarization_factor* parameter. Set it between [-1, 1], to correct your data. If set to 0 there is no correction at all.

The *dark* and the *flat* can be provided to correct the data before computing the radial integration.

3.3 `detectors` Module

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

class `pyFAI.detectors.`**`Basler`**(*pixel=3.7500000000000001e-06*)
Bases: `pyFAI.detectors.Detector`

Basler camera are simple CCD camara over GigaE

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

Generic class representing a 2D detector

`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

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_splineFile()

mask

maskfile

name
Get a meaningful name for detector

pixel1

pixel2

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 dictionnary or a JSON string or a file containing this JSON configuration

keys in that dictionnary 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_mask(*mask*)

set_maskfile(*maskfile*)

set_pixel1(*value*)

set_pixel2(*value*)

set_splineFile(*splineFile*)

splineFile

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

Metaclass used to register all detector classes inheriting from Detector

class pyFAI.detectors.**Dexela2923**(*pixel1=7.4999999999999993e-05*, *pixel2=7.4999999999999993e-05*)
Bases: pyFAI.detectors.Detector

Dexela CMOS family detector

class pyFAI.detectors.**Eiger**(*pixel1=7.4999999999999993e-05*, *pixel2=7.4999999999999993e-05*)
Bases: pyFAI.detectors.Detector

Eiger detector: generic description containing mask algorithm

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...

class pyFAI.detectors.**Eiger16M**(*pixel1=7.4999999999999993e-05*, *pixel2=7.4999999999999993e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 16M detector

class pyFAI.detectors.**Eiger1M**(*pixel1=7.4999999999999993e-05*, *pixel2=7.4999999999999993e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 1M detector

class pyFAI.detectors.**Eiger4M**(*pixel1=7.4999999999999993e-05*, *pixel2=7.4999999999999993e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 4M detector

class pyFAI.detectors.**Eiger9M**(*pixel1=7.4999999999999993e-05*, *pixel2=7.4999999999999993e-05*)
Bases: pyFAI.detectors.Eiger

Eiger 9M detector

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

class pyFAI.detectors.**ImXPadS140**(*pixel1=0.00012999999999999999*, *pixel2=0.00012999999999999999*)
Bases: pyFAI.detectors.Detector

ImXPad detector: ImXPad s140 detector with 2x7modules

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.

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

Mar345 Imaging plate detector

calc_mask()

class pyFAI.detectors.**Perkin**(*pixel=0.00020000000000000001*)
Bases: pyFAI.detectors.Detector

Perkin detector

class pyFAI.detectors.**Pilatus**(*pixel1=0.00017200000000000001*, *pixel2=0.00017200000000000001*, *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

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...

get_splineFile()

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

splineFile

class pyFAI.detectors.**Pilatus100k**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 100k detector

class pyFAI.detectors.**Pilatus1M**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 1M detector

class pyFAI.detectors.**Pilatus200k**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 200k detector

class pyFAI.detectors.**Pilatus2M**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 2M detector

class pyFAI.detectors.**Pilatus300k**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 300k detector

class pyFAI.detectors.**Pilatus300kw**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 300k-wide detector

class pyFAI.detectors.**Pilatus6M**(*pixel1=0.0001720000000000001*, *pixel2=0.0001720000000000001*)
Bases: pyFAI.detectors.Pilatus

Pilatus 6M detector

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

binning

get_binning()

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

calc_mask()
Circular mask

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

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

Nota: this is the same for lx170hs

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

Rayonix lx255 2d Detector (3x1 CCDs)

Nota: this detector is also called lx255hs

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

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

Nota: this is the same for mx170hs

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

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

Rayonix mx225hs 2D CCD detector detector

Pixel size from a personnal communication from M. Blum

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

Rayonix mx300 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

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

Rayonix mx300hs 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

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

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

Rayonix mx340hs 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

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

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

Rayonix sx165 2d Detector also known as MAR165.

Circular detector

calc_mask()
Circular mask

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

Rayonix sx200 2d CCD Detector.

Pixel size are personnal communication from M. Blum.

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

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

class pyFAI.detectors.**Titan**(*pixel1=6.0000000000000002e-05*, *pixel2=6.0000000000000002e-05*)
Bases: pyFAI.detectors.Detector

Titan CCD detector from Agilent. Mask not handled

class pyFAI.detectors.**Xpad_flat**(*pixel1=0.00012999999999999999*, *pixel2=0.00012999999999999999*)
Bases: pyFAI.detectors.Detector

Xpad detector: generic description for ImXPad detector with 8x7modules

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 Xpad detectors... discards the first line and raw form all modules: those are 2.5x bigger and often mis - behaving

3.4 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_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*)
Calculate the incidence angle (alpha) for current pixels (P). The poni is at incidence angle=1 so cos(alpha) = 1

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

get_chia()

get_correct_solid_angle_for_spline()

get_dist()

get_dssa()

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

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

set_chia(*_*)

set_correct_solid_angle_for_spline(*value*)

set_dist(*value*)

set_dssa(*_*)

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

Parameters

- **shape** – shape of the array expected

- **order** –

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

3.5 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*)

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

3.6 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

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

3.7 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*)
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

3.8 ocl_azim_csr Module

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

3.9 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*)
Bases: object

Simple class that contains the structure of an OpenCL device

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

Simple class that wraps the structure ocl_tools_extended.h

create_context(*devicetype='ALL'*, *useFp64=False*, *platformid=None*, *deviceid=None*)
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

get_platform(*key*)
Return a platform according

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

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

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*)

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

3.10 peakPicker Module

class pyFAI.peakPicker.**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*, *angle=None*, *ring=None*)

Parameters

- **point** – list of points
- **angle** – 2-theta angle in radians

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

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

get_dSpacing()

get_wavelength()

load(*filename*)
load all control points from a file

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

pop(*idx=None*)
Remove the set of points at given index (by default the last) :param idx: position of the point to remove

readAngleFromKeyboard()
Ask the 2theta values for the given points

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

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

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.peakPicker.**Event**(*width*, *height*)
Bases: object

Dummy class for dummy things

class pyFAI.peakPicker.**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 0x7fc8adee9150>*)
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()

getLabeledMassif(*pattern=None*)

getMedianData()

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

class pyFAI.peakPicker.**PeakPicker**(*strFilename*, *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

closeGUI()

contour(*data*)
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*)
Ask the ring number for the given points

Parameters **filename** – file with the point coordinates saved

gui(*log=False*, *maximize=False*)

Parameters **log** – show z in log scale

init(*method*, *sync=True*)
Unified initializer

init_blob(*sync=True*)
Initialize PeakPicker for blob based detection

init_massif(*sync=True*)
Initialize PeakPicker for massif based detection

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

onclick(*event*)

peaks_from_area(*mask*, *Imin*, *keep=1000*, *refine=True*, *method=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()

3.11 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(**arg*, ***kw*)
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

show_stats()
Shows a window with the repartition of keypoint in function of scale/intensity

pyFAI.blob_detection.**image_test**()

pyFAI.blob_detection.**make_gaussian**(*im*, *sigma*, *xc*, *yc*)

3.12 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.**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_unit()

process(*data*)
Process a frame

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

3.13 `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.

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.**Writer**(*filename=None*, *extension=None*)
Bases: object

Abstract class for writers.

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.**getIsoTime**(*forceTime=None*)

Parameters **forceTime** (*float*) – enforce a given time (current by default)

Returns the current time as an ISO8601 string

Return type string

3.14 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

analyse_options(*options=None*, *args=None*)
Analyse options and arguments

Returns option,arguments

configure_parser(*version='calibration from pyFAI version 0.9.4: 2014-06'*, *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_detector(*detector*)

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

validate_calibration()
Validate the calivration and calculate the offset in the diffraction image

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

read_dSpacingFile()
Read the name of the file with d-spacing

refine()
Contains the geometry refinement part specific to Recalibration

3.15 distortion Module

pyFAI.distortion.**test**()

3.16 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*)

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*)
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

Returns displacement matrix for the X direction

Return type ndarray

splineFuncY(*x*, *y*)
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

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 ....

3.17 utils Module

Utilities, mainly for image treatment

pyFAI.utils.**averageDark**(*lstimg*, *center_method='mean'*, *cutoff=None*)
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"
- **cutoff** – keep all data where (I-center)/std < cutoff

Returns 2D image averaged

pyFAI.utils.**averageImages**(*listImages*, *output=None*, *threshold=0.10000000000000001*, *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*)

Parameters

- **input_img** – input ndarray
- **binsize** – int or 2-tuple representing the size of the binning

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.**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.**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_cl_file`**(*filename*)

`pyFAI.utils.`**`get_ui_file`**(*filename*)

`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.`**`percentile`**(*a*, *q*, *axis=None*, *out=None*, *overwrite_input=False*)
Compute the qth percentile of the data along the specified axis.

Returns the qth percentile of the array elements.

a [array_like] Input array or object that can be converted to an array.

q [float in range of [0,100] (or sequence of floats)] Percentile to compute which must be between 0 and 100 inclusive.

axis [int, optional] Axis along which the percentiles are computed. The default (None) is to compute the median along a flattened version of the array.

out [ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

overwrite_input [bool, optional] If True, then allow use of memory of input array *a* for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if *overwrite_input* is True and the input is not already an array, an error will be raised.

pcntile [ndarray] A new array holding the result (unless *out* is specified, in which case that array is returned instead). If the input contains integers, or floats of smaller precision than 64, then the output data-type is float64. Otherwise, the output data-type is the same as that of the input.

mean, median

Given a vector V of length N, the qth percentile of V is the qth ranked value in a sorted copy of V. A weighted average of the two nearest neighbors is used if the normalized ranking does not match q exactly. The same as the median if `q=0.5`, the same as the minimum if `q=0` and the same as the maximum if `q=1`.

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.percentile(a, 50)
3.5
>>> np.percentile(a, 0.5, axis=0)
array([ 6.5,  4.5,  2.5])
>>> np.percentile(a, 50, axis=1)
array([ 7.,  2.])

>>> m = np.percentile(a, 50, axis=0)
>>> out = np.zeros_like(m)
>>> np.percentile(a, 50, axis=0, out=m)
array([ 6.5,  4.5,  2.5])
>>> m
array([ 6.5,  4.5,  2.5])

>>> b = a.copy()
>>> np.percentile(b, 50, axis=1, overwrite_input=True)
array([ 7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.percentile(b, 50, axis=None, overwrite_input=True)
3.5
```

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.**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=0.10000000000000001*, *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

Returns unBinned input ndarray

CHAPTER

FOUR

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 or 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.

4.1 Programming language

PyFAI is a Python project but uses many programming languages:

- 12000 lines of Python (plus 3000 for the test)
- 8000 lines of Cython which are converted into ...
- 400000 lines of C
- 2000 lines of OpenCL kernels

4.2 Repository:

The project is hosted by GitHub: https://github.com/kif/pyFAI

Which provides the issue tracker in addition to Git hosting. Collaboration is done via Pull-Requests in github's web interface:

Anybody can fork the project and adapt it to his own needs (people from CEA-saclay, Synchrotron Soleil or APS did it). If developments are useful to other, new developments can be merged into the main branch.

4.3 Run dependencies

- Python2.6 or python2.7
- NumPy
- SciPy
- Matplotlib
- FabIO

- pyOpencl (optional)
- fftw (optional)

4.4 Build dependencies:

In addition to the run dependencies, pyFAI needs a C compiler which supports OpenMP (gcc>=4.2, msvc, ...) There is an issue with MacOSX (v10.8 and newer) where the default compiler switched from gcc 4.2 to clang which dropped the support for OpenMP. The best solution looks like to install any recent gcc and use it for compiling pyFAI.

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).

4.5 Building procedure

As most of the python projects: ..

    python setup.py build install

4.6 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 23 test-suites (132 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)

Table 4.1: Test suite coverage

Name	Stmts	Exec	Cover
pyFAI/__init__	10	7	70%
pyFAI/azimuthalIntegrator	1140	879	77%
pyFAI/detectors	441	289	65%
pyFAI/blob_detection	510	194	38%
pyFAI/calibrant	161	69	42%
pyFAI/calibration	770	0	0%
pyFAI/detectors	694	548	78%
pyFAI/distortion	454	0	0%
pyFAI/geometry	707	545	77%
pyFAI/geometryRefinement	371	221	59%
pyFAI/integrate_widget	402	0	0%
pyFAI/io	344	0	0%
pyFAI/ocl_azim	306	215	70%
pyFAI/ocl_azim_csr	242	171	70%
pyFAI/ocl_azim_csr_dis	239	0	0%
pyFAI/ocl_azim_lut	228	198	86%
pyFAI/opencl	140	102	72%
pyFAI/peakPicker	694	322	46%
pyFAI/spline	327	108	33%
pyFAI/units	40	35	87%
pyFAI/utils	658	377	57%
pyFAI/worker	183	0	0%
pyFAI	9061	4280	47%
tests	1890	1563	83
TOTAL	10951	5843	53%

Note that the test coverage tool does not count lines of Cython.

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/

4.7 List of contributors in code

```
$ git log  --pretty='%aN##%s' | grep -v 'Merge pull' | grep -Po '^[^#]+' | sort | uniq -c | sort -rn
```

As of 06/2014:

- Jérôme Kieffer (ESRF)
- Frédéric-Emmanuel Picca (Soleil)
- Dimitris Karkoulis (ESRF)
- Aurore Deschildre (ESRF)
- Giannis Ashiotis (ESRF)
- Zubair Nawaz (Sesame)
- Jon Wright (ESRF)
- Amund Hov (ESRF)
- Dodogerstlin @github

- Gunthard Benecke (Desy)
- Gero Flucke (Desy)

4.8 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)

4.9 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 and beamtime
- ESRF ID29: provided manpower in 2013 (MX-calibrate)
- ESRF ID02: provide manpower 2014

CHAPTER

FIVE

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 all. Nucl. Acids Res. (2012) 40 (D1): D420-D427. http://dx.doi.org10.1093/nar/gkr900 http://www.crystallography.net/

PYTHON MODULE INDEX

p

INDEX

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z