pyFAI Documentation

Release 0.9.0

Jerome Kieffer

June 09, 2013

CONTENTS

Contents:

CHAPTER

ONE

AIM OF PYFAI

$2D$ 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 (xrpd)). These detectors have a large sensitive area of millions of pixels with high spatial resolution. The software package pyFAI 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*). 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.

CHAPTER

TWO

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 standalone 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.

CHAPTER

THREE

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, . . .).

3.1 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*.

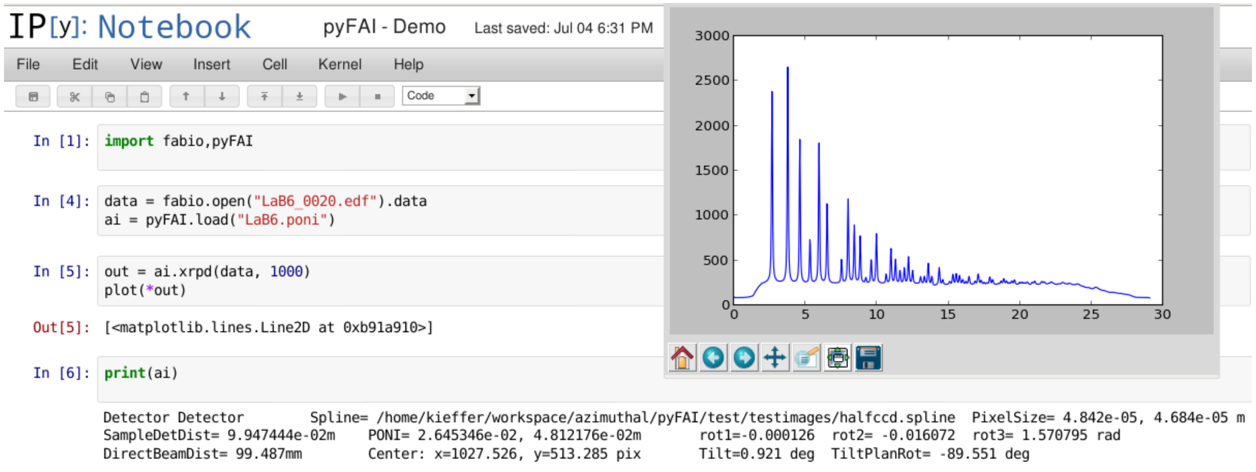
3.2 PyFAI executables

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. 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 Grqaphical interface based on Qt is under development

3.3 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.



```
In [1]: import fabio,pyFAI

In [4]: data = fabio.open("LaB6_0020.edf").data
        ai = pyFAI.load("LaB6.poni")

In [5]: out = ai.xrpd(data, 1000)
        plot(*out)

Out[5]: [<matplotlib.lines.Line2D at 0xb91a910>]

In [6]: print(ai)

        Detector Detector        Spline= /home/kieffer/workspace/azimuthal/pyFAI/test/testimages/halfccd.spline  PixelSize= 4.842e-05, 4.684e-05 m
        SampleDetDist= 9.947444e-02m    PONI= 2.645346e-02, 4.812176e-02m       rot1=-0.000126  rot2= -0.016072  rot3= 1.570795 rad
        DirectBeamDist= 99.487mm        Center: x=1027.526, y=513.285 pix       Tilt=0.921 deg  TiltPlanRot= -89.551 deg
```

CHAPTER

FOUR

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).

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.

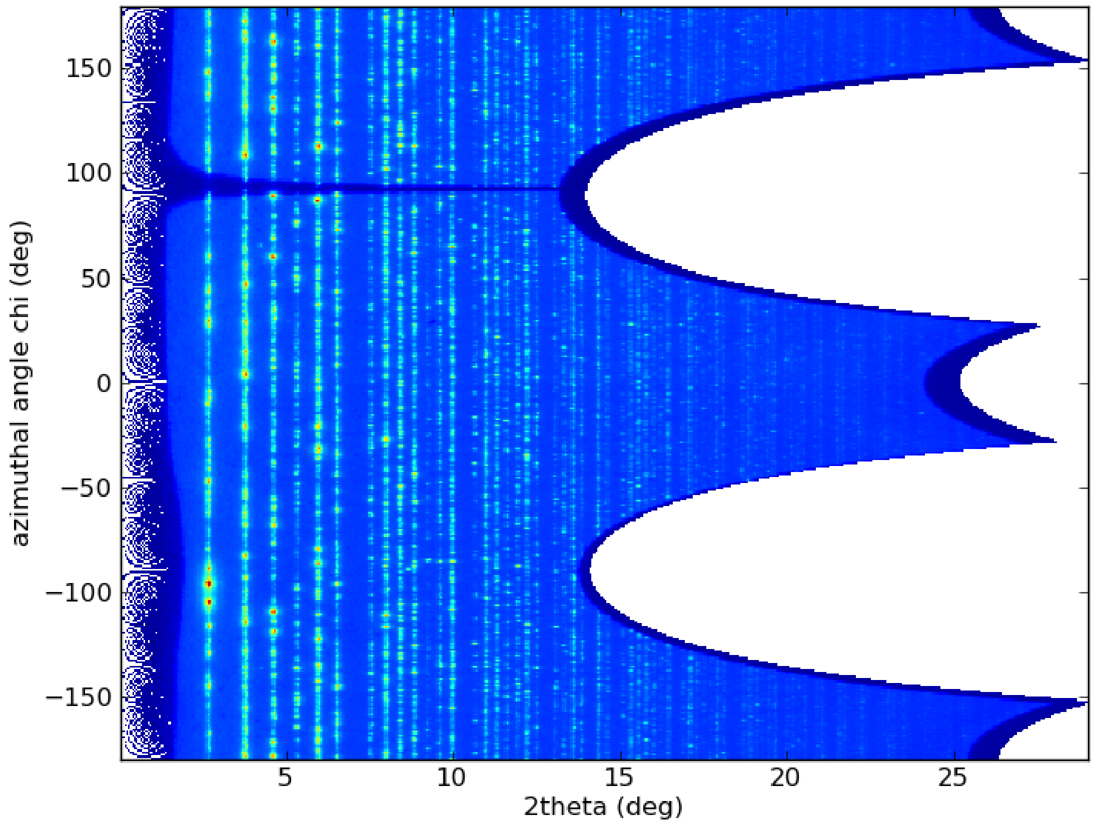
4.2 Performances and migration to native code

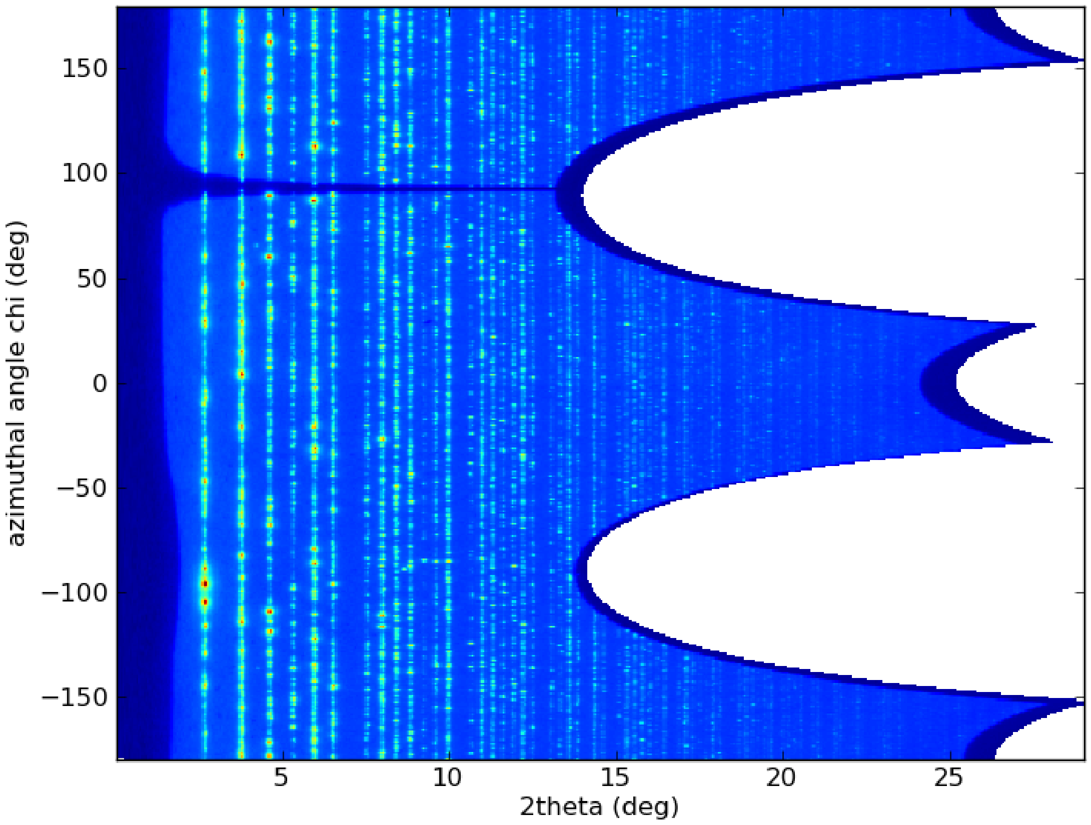
Originally, regrouping was implemented using the histogram provided by NumPy, then re-implemented in Cython with pixel splitting to achieve a four-fold speed-up. The computation time scales like O(N) with the size of the input image. The number of output bins shows only little influence; overall the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

4.3 Parallel implementation

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

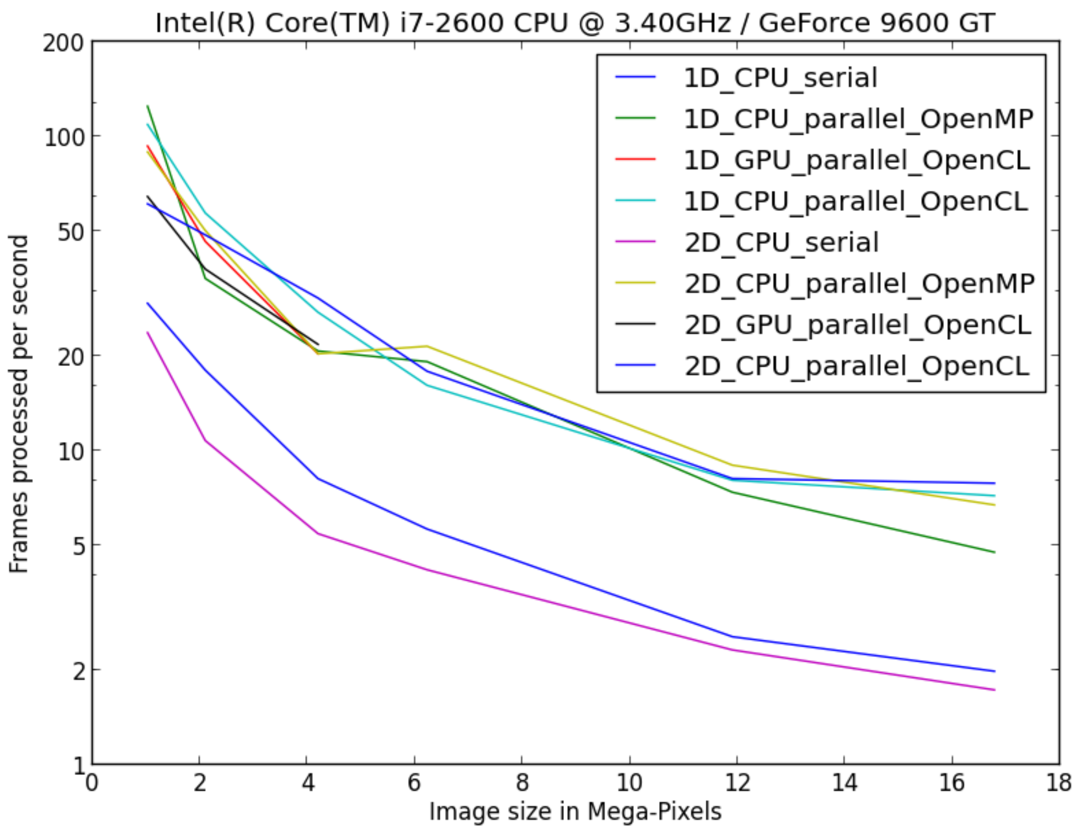
To overcome this limitation; instead of looking at where input pixels GO TO in the output image, we instead look at where the output pixels COME FROM in the input image. The correspondence between pixels and output bins can be stored in a look-up table (LUT) together with the pixel weight which make the integration look like a simple (if large


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


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

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 Mb are needed to store the LUT to integrate a 16 megapixel image, which fits onto a reasonable quality graphics card nowadays. 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. This algorithm was implemented in Cython-OpenMP and OpenCL. 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 perfomances of the parallel implementation based on a LUT are above 125 MPix/s (on a 3.4 GHz Intel core i7-2600) and can reach 200 MPix/s on recent multi-socket, multi-core computer or on high-end GPUs like Tesla cards.



CHAPTER

FIVE

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 gpu 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.

5.1 Acknowledgments

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

5.2 References:

- The philosophy of pyFAI is described in the proceedings of SRI2012: doi:10.1088/1742-6596/425/20/202012 http://iopscience.iop.org/1742-6596/425/20/202012/
- The LUT implementation (ported to GPU) is described in the proceedings of EPDIC13: http://epdic13.grenoble.cnrs.fr/spip.php?article43 (to be published)

CHAPTER

SIX

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 anythong about Python:

- pyFAI-average to merge a set of files like dark-current files or diffracton images using various filters
- drawMask_pymca to mask out some region of the detector
- pyFAI-calib to select the rings and refine the geometry
- pyFAI-recalib with an automatic ring extraction followed by the refinement.
- pyFAI-integrate offers a graphical interface to configure the integration of an experiment
- pyFAI-waxs text interface for integration of an experiment in 2theta
- pyFAI-saxs text interface for integration of an experiment in q-space

Few other scripts are also available, most of them are very specific to one experiment or are highly experimental.

- diff_tomo is a tool to generate a 3D sinogram as an HDF5 dataset for a diffracton tomography experiment
- check-calib is an experimental tool to validate a full calibration of the complete image (not only on the peaks)
- MX-calibrate refines the calibration from a set of images and exports the parameters interpolated as function of the detector distance

6.1 Preprocessing tool: pyFAI-average

6.1.1 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-Sherrer rings. In this case the "max-filter" is usually recommended.

6.1.2 Options:

Usage: pyFAI-average [options] -o option.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)

6.2 Mask generation tool: drawMask_pymca

6.2.1 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.

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

6.2.2 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)

6.3 Calibration tool: pyFAI-calib

6.3.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images without a priori knowledge of your setup. You will need a "d-spacing" file containing the spacing of Miller plans in Angstrom (in decreasing order). If you are using a standart calibrant, look at https://github.com/kif/pyFAI/tree/master/calibration or search in the American Mineralogist database: http://rruff.geo.arizona.edu/AMS/amcsd.php

You will need in addition:

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

file describing its distortion

Many option are available among those:

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

- 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)

6.3.2 Usage:

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

6.3.3 Options:

-h, --help show the 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, --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.
-c, --square	Use square kernel shape for neighbor search instead of diamond shape
-p PIXEL, --pixel=PIXEL	size of the pixel in micron

6.3.4 Example of usage:

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

:: pyFAI-calib -D Pilatus1M -S calibration/AgBh.D -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 -S calibration/LaB6.D -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.

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

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

They will be located in tests/testimages

6.4 Calibration tool: pyFAI-recalib

6.4.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images with a priori knowledge of your setup (an input PONI-file). You will need a "d-spacing" file containing the spacing of Miller plans in Angstrom (in decreasing order). If you are using a standart calibrant, look at https://github.com/kif/pyFAI/tree/master/calibration or search in the American Mineralogist database: http://rruff.geo.arizona.edu/AMS/amcsd.php

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).

6.4.2 Usage:

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

6.4.3 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

6.4.4 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).

6.5 Integration tool: pyFAI-integrate

6.5.1 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.

TODO: add an image of the GUI

6.5.2 Usage

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

6.5.3 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

6.5.4 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

CHAPTER

SEVEN

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.

7.1 pyFAI Package

7.1.1 `pyFAI` Package

7.1.2 `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

darkcurrent

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*)

Calculate the azimuthal integrated Saxs curve in q(nm^-1) by default

Multi algorithm implementation (tries to be bullet proof)

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 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", "lut", "lut_ocl" if you want to go on GPU, ...
- **unit** (*pyFAI.units.Enum*) – can be Q, TTh, R for now
- **safe** (*bool*) – Do some extra checks to ensure LUT is still valid. False is faster.

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

Return type 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*)

Calculate the azimuthal regrouped 2d image in q(nm^-1)/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 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", "lut", "lut_ocl" if you want to go on GPU, ...
- **unit** (*pyFAI.units.Enum*) – can be Q, TTH, R for now
- **safe** (*bool*) – Do some extra checks to ensure LUT is still valid. False is faster.

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

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

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

Parameters **hdr** (*str*) – string used as comment in the header

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*)

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

`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'*)
Set the dark current from one or mutliple files, avaraged according to the method provided

`set_flatfield`(*flat*)

`set_flatfiles`(*files*, *method='mean'*)
Set the flat field from one or mutliple files, avaraged according to the method provided

`set_mask`(*mask*)

`set_maskfile`(*maskfile*)

`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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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_LUT` (*data*, *nbPt*, *filename=None*, *correctSolidAngle=True*, *tthRange=None*, *chiRange=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *safe=True*)

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** (*boolean*) – solid angle correction
- **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*)

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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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** (*boolean*) – solid angle correction
- **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.

7.1.3 `detectors` Module

class `pyFAI.detectors.`**`Basler`**(*pixel=3.75e-06*)

Bases: `pyFAI.detectors.Detector`

Basler camera are simple CCD camara over GigaE

`force_pixel` = **True**

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!

`force_pixel` = **False**

`getFit2D`()

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

Returns representation of the detector easy to serialize

Return type dict

`getPyFAI`()

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

Returns representation of the detector easy to serialize

Return type dict

get_binning()

get_mask()

get_maskfile()

get_pixel1()

get_pixel2()

get_splineFile()

mask

maskfile

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_mask(*mask*)

set_maskfile(*maskfile*)

set_pixel1(*value*)

set_pixel2(*value*)

set_splineFile(*splineFile*)

splineFile

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

Dexela CMOS family detector

force_pixel = True

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

`force_pixel` **= True**

class `pyFAI.detectors.`**`Perkin`**(*pixel=0.0002*)

Bases: `pyFAI.detectors.Detector`

Perkin detector

`force_pixel` **= True**

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

Bases: `pyFAI.detectors.Detector`

Pilatus detector: generic description containing mask algorithm

Sub-classed by Pilatus1M, Pilatus2M and Pilatus6M

`MODULE_GAP` **= (17, 7)**

`MODULE_SIZE` **= (195, 487)**

`calc_cartesian_positions`(*d1=None*, *d2=None*)

Calculate the position of each pixel center in cartesian coordinate and in meter of a couple of coordinates. The half pixel offset is taken into account here !!!

Parameters

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

Returns position in meter of the center of each pixels.

Return type ndarray

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

`calc_mask`()

Returns a generic mask for Pilatus detectors...

`force_pixel` **= True**

`get_splineFile`()

`set_splineFile`(*splineFile=None*)

In this case splinefile is a couple filenames

`splineFile`

class `pyFAI.detectors.`**`Pilatus1M`**(*pixel1=0.000172*, *pixel2=0.000172*)

Bases: `pyFAI.detectors.Pilatus`

Pilatus 1M detector

class `pyFAI.detectors.`**`Pilatus2M`**(*pixel1=0.000172*, *pixel2=0.000172*)

Bases: `pyFAI.detectors.Pilatus`

Pilatus 2M detector

`force_pixel` **= True**

class `pyFAI.detectors.`**`Pilatus6M`**(*pixel1=0.000172*, *pixel2=0.000172*)

Bases: `pyFAI.detectors.Pilatus`

Pilatus 6M detector

`force_pixel` = **True**

class `pyFAI.detectors.`**`Xpad_flat`**(*pixel1=0.00013*, *pixel2=0.00013*)

Bases: `pyFAI.detectors.Detector`

Xpad detector: generic description for ImXPad detector with 8x7modules

`MODULE_GAP` = **(30.46153846153846, 3)**

`MODULE_SIZE` = **(120, 80)**

`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

`force_pixel` = **True**

`pyFAI.detectors.`**`detector_factory`**(*name*)

A kind of factory... :param name: name of a detector :type name: str

Returns an instance of the right detector

Return type pyFAI.detectors.Detector

7.1.4 `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)))

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

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*)

Calulate 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

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

`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

7.1.5 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*, *dSpacing=None*)
Bases: pyFAI.azimuthalIntegrator.AzimuthalIntegrator

anneal(*maxiter=1000000*)

calc_2th(*rings*, *wavelength*)

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*)

Parameters **value** – Tolerance as a percentage

set_wavelength_max(*value*)

set_wavelength_min(*value*)

simplex(*maxiter=1000000*)

wavelength_max

wavelength_min

7.1.6 `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

7.1.7 `ocl_azim_lut` Module

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

Bases: `object`

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

7.1.8 `opencl` Module

class `pyFAI.opencl.`**`Device`**(*name='None'*, *type=None*, *version=None*, *driver_version=None*, *extensions=''*, *memory=None*, *available=None*, *cores=None*, *frequency=None*, *flop_core=None*, *id=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

`comput_cap` = (1, 1)

`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

`flop_core` = 4

`get_platform`(*key*)

Return a platform according

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

`id` = 3

`idd` = 0

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

`select_device`(*type='ALL'*, *memory=None*, *extensions=[ ]*, *best=True*)

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*, *id=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

7.1.9 peakPicker Module

class pyFAI.peakPicker.**ControlPoints**(*filename=None*, *dSpacing=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

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

getWavelength()

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

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

load_dSpacing(*filename*)
Load a d-spacing file containing the inter-reticular plan distance in Angstrom

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(*filename*)
save the d-spacing to a file

setWavelength(*value=None*)

setWavelength_change2th(*value=None*)

setWavelength_changeDs(*value=None*)
This is probably not a good idea, but who knows !

wavelength

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

Dummy class for dumm 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 0x7f91c94011e0>*)
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*)
:returns the coordinates of the nearest peak

setValleySize(*size*)

valley_size
Defines the minimum distance between two massifs

class `pyFAI.peakPicker.`**`PeakPicker`**(*strFilename*, *reconst=False*, *mask=None*, *pointfile=None*, *dSpacing=None*, *wavelength=None*)

Bases: `object`

`closeGUI`()

`contour`(*data*)

Overlay a contour-plot

Parameters **data** – 2darray with the 2theta values in radians...

`display_points`()

`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

`load`(*filename*)

load a filename and plot data on the screen (if GUI)

`massif_contour`(*data*)

Parameters **data** –

`onclick`(*event*)

`readFloatFromKeyboard`(*text*, *dictVar*)

Read float from the keyboard .... :param text: string to be displayed :param dictVar: dict of this type: {1: [set_dist_min],3: [set_dist_min, set_dist_guess, set_dist_max]}

7.1.10 `refinment2D` Module

class `pyFAI.refinment2D.`**`Refinment2D`**(*img*, *ai=None*)

Bases: `object`

refine the parameters from image itself ... (Jerome est-ce que tu peux elaborer un petit peu plus ???)

`diff_Fit2D`(*axis='all'*, *dx=0.1*)

???

Parameters

- **axis** – ???
- **dx** – ???

Returns

???

Return type

???

`diff_tth_X`(*dx=0.1*)

Jerome peux-tu décrire de quoi il retourne ???

Parameters **dx** – ???

Type float ???

Returns

???

Return type

???

diff_tth_tilt(*dx=0.1*)

idem ici ???

Parameters **dx** (*float ???*) – ???

Returns

???

Return type

???

get_shape()

reconstruct(*tth*, *I*)

Reconstruct a perfect image according to 2th / I given in input

Parameters

- **tth** (*ndarray*) – 2 theta array
- **I** (*ndarray*) – intensity array

Returns a reconstructed image

Return type ndarray

scan_Fit2D(*width=1.0*, *points=10*, *axis='tilt'*, *dx=0.1*)

???

Parameters

- **width** (*float ???*) – ???
- **points** (*int ???*) – ???
- **axis** (*str ???*) – ???
- **dx** (*float ???*) – ???

Returns

???

Return type

???

scan_centerX(*width=1.0*, *points=10*)

???

Parameters

- **width** (*float ???*) – ???
- **points** (*int ???*) – ???

Returns

???

Return type

???

scan_tilt (*width=1.0*, *points=10*)
???

Parameters

- **width** (*float ???*) – ???
- **points** (*int ???*) – ???

Returns

???

Return type

???

shape

7.1.11 `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*)
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 ....

7.1.12 `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=0.1*, *minimum=None*, *maximum=None*, *darks=None*, *flats=None*, *filter_='mean'*, *correct_flat_from_dark=False*)
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 ?

`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.`**`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.`**`int_`**(*val*)

Convert anything to an int ... or None if not applicable

`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.`**`relabel`**(*label*, *data*, *blured*, *max_size=None*)

Relabel limits the number of region in the label array. They are ranked relatively to their max(I0)-max(blur(I0)

Parameters

- **label** – a label array coming out of scipy.ndimage.measurement.label
- **data** – an array containing the raw data
- **blured** – an array containing the blured data
- **max_size** – the max number of label wanted

:return array like label

`pyFAI.utils.`**`removeSaturatedPixel`**(*ds*, *threshold=0.1*, *minimum=None*, *maximum=None*)

Parameters

- **ds** – a dataset as ndarray
- **threshold** – what is the upper limit? all pixel > max*(1-threshold) are discareded.
- **minimum** – minumum valid value (or True for auto-guess)
- **maximum** – maximum valid value

Returns another dataset

`pyFAI.utils.`**`shift`**(*input_img*, *shift_val*)

Shift an array like scipy.ndimage.interpolation.shift(input_img, shift_val, mode="wrap", order=0) but faster :param input_img: 2d numpy array :param shift_val: 2-tuple of integers :return: shifted image

`pyFAI.utils.`**`shiftFFT`**(*input_img*, *shift_val*, *method='fftw'*)

Do shift using FFTs Shift an array like scipy.ndimage.interpolation.shift(input, shift, mode="wrap", order="infinity") but faster :param input_img: 2d numpy array :param shift_val: 2-tuple of float :return: shifted image

`pyFAI.utils.`**`str_`**(*val*)

Convert anything to a string ... but None -> ""

`pyFAI.utils.`**`timeit`**(*func*)

`pyFAI.utils.`**`unBinning`**(*binnedArray*, *binsize*, *norm=True*)

Parameters

- **binnedArray** – input ndarray
- **binsize** – 2-tuple representing the size of the binning

Returns unBinned input ndarray

PyFAI is a library to deal with diffraction images for data reduction. This chapter describes the project from the computer engineering point of view.

CHAPTER

EIGHT

PROJECT STRUCTURE

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 perfromances image treatement thanks to cython and OpenCL... but only a C-compiler is needed to build that.

8.1 Programming language

PyFAI is a python project but uses many programming languages:

- 8000 lines of Python (without the tests)
- 3500 lines of cython which are converted into
- 139055 lines of C
- 880 lines of OpenCL kernels

8.2 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 or Synchrotron Soleil did it). If developments are useful to other, new developments can be merged into the main branch.

8.3 Run dependencies

- Python2.6 or python2.7
- NumPy
- SciPy
- Matplotlib
- FabIO
- pyOpencl (optional)
- fftw (optional)

8.4 Build dependencies:

In addition to the run dependencies, pyFAI needs a C compiler which supports OpenMP (gcc>=4.2, msvc, ...)

C files are generated from cython source and distributed. Cython is only needed for developing new binary modules.

8.5 Building procedure

As most of the python projects: ..

> python setup.py build install

8.6 Test suites

To run the test an internet connection is needed as 200MB of test images will be downloaded ..

> python setup.py build test

PyFAI comes with 15 test-suites (98 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 8.1: Supported formats

Name	Stmts	Miss	Cover
pyFAI/__init__	9	3	67%
pyFAI/azimuthalIntegrator	1097	329	70%
pyFAI/detectors	271	105	61%
pyFAI/geometry	672	166	75%
pyFAI/geometryRefinement	353	198	44%
pyFAI/ocl_azim	304	90	70%
pyFAI/ocl_azim_lut	196	29	85%
pyFAI/opencl	134	42	69%
pyFAI/peakPicker	609	388	36%
pyFAI/spline	324	217	33%
pyFAI/units	36	2	94%
pyFAI/utils	494	196	60%
testAzimuthalIntegrator	189	75	60%
testBilinear	40	4	90%
testDistortion	47	3	94%
testExport	69	7	90%
testFlat	88	9	90%
testGeometry	50	4	92%
testGeometryRefinement	53	3	94%
testHistogram	153	14	91%
testIntegrate	136	10	93%
testMask	85	21	75%
testOpenCL	89	9	90%
testPeakPicking	70	6	91%
testPolarization	54	21	61%

Continued on next page

Table 8.1 – continued from previous page

Name	Stmts	Miss	Cover
testSaxs	102	29	72%
testUtils	80	4	95%
test_all	47	1	98%
utilstest	139	88	37%
TOTAL	5990	2073	65%

Continuous integration is made by a home-made scripts which checks out the latest release and builds and runs the test every night. Nightly builds are available for debian6-64 bits in: http://www.edna-site.org/pub/debian/binary/

8.7 List of contributors

As of 04/2013 (number of commits in parenthesis):

- Jérôme Kieffer (656)
- Dimitris Karkoulis (22)
- Frédéric-Emmanuel Picca (21)
- Jonathan Wright (6)
- Amund Hov (1)

8.8 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)

8.9 List of supporters

- LinkSCEEM project: initial porting to OpenCL
- ESRF ID11: Provided manpower in 2012 and 2013 and beamtime
- ESRF ID13: Provided manpower in 2012 and 2013 and beamtime
- ESRF ID29: will provide manpower in 2013

CHAPTER

NINE

INDICES AND TABLES

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

PYTHON MODULE INDEX