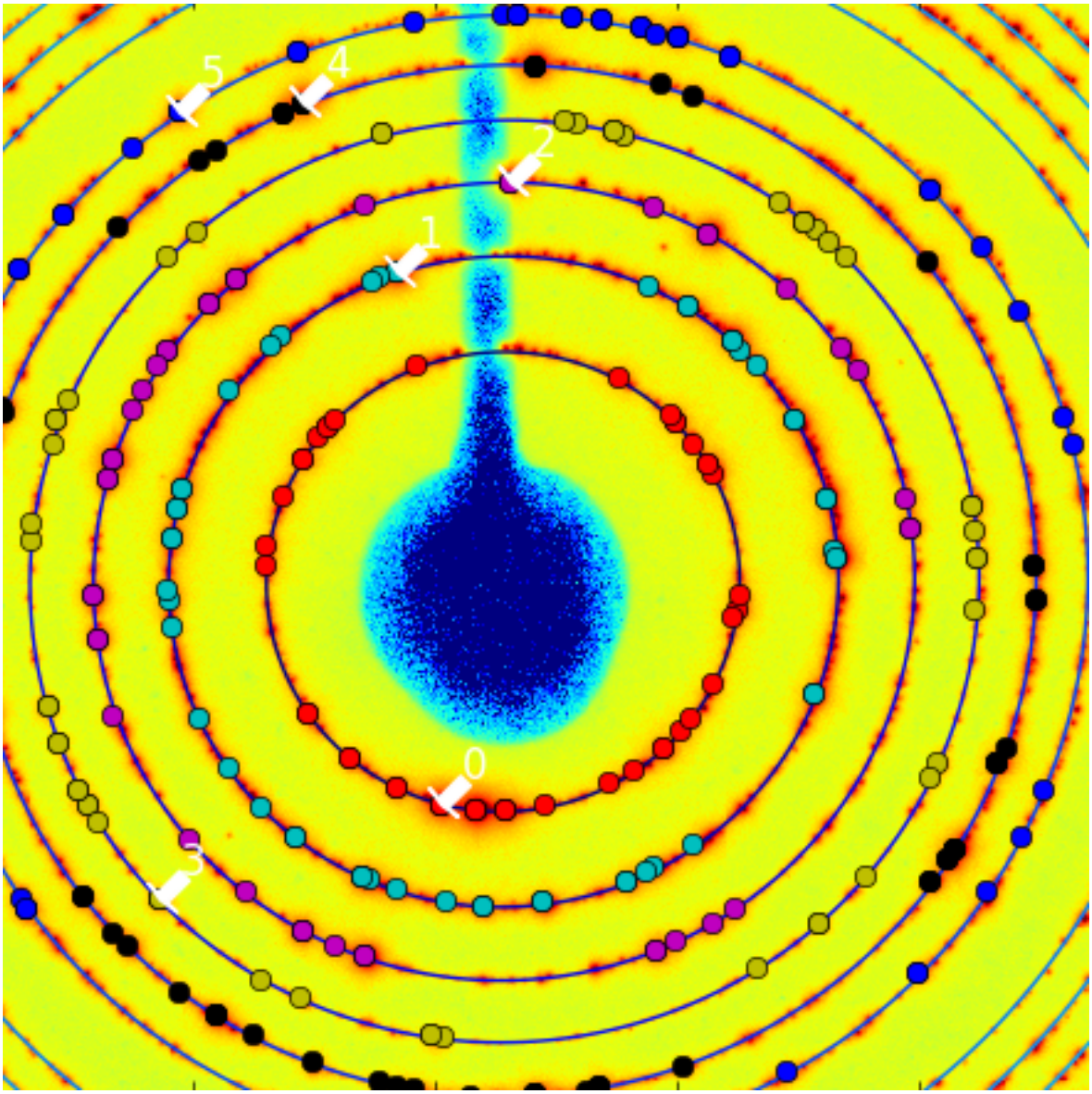


pyFAI Documentation

Release 0.13.0

Jérôme Kieffer

December 02, 2016

PyFAI is a python libary for azimuthal integration of diffraction data acquired with 2D detectors, providing high performance thanks to GPU computing. Most concepts have been presented at EuroScipy 2014 in this video as well as the proceedings.

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

Follows cookbook and tutorials on how to use pyFAI: Cookbooks focus on how to use pyFAI using the various graphical interfaces or from scripts on the command linee. Tutorials use the *Jupyter* notebook (formerly ipython) and present the Python interface. The first tutorial start with a quick general introduction of the notebooks, how to process data in the general case, subsequent tutorials are more advanced and require already a good knowledge both in Python and pyFAI.. After the tutorials, all manual pages of pyFAI programs, both graphical interfaces and scripts are described in the documentation.

The design of the programming interface (API) is then exposed before a comprehensive description of most modules contained in pyFAI. Some minor submodules as well as the documentation of the Cython sub-modules are not included for concision purposes.

Installation procedures for Windows, MacOSX and Linux operating systems are then described.

Finally other programs/projects relying on pyFAI are presented and the project is summarized from a developer's point of view.

In appendix there are some figures about the project and its management and a list of publication on pyFAI.

CHAPTER

ONE

GENERAL INTRODUCTION TO PYFAI

1.1 Python Fast Azimuthal Integration

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

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

1.2 Introduction

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

1.3 Experiment description

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

1.3.1 Detector

Simple detector

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

Pixels of the detector are indexed from the *origin* located at the **lower left corner** when looking from the sample. The pixel's center is located at half integer index:

- pixel 0 goes from position 0 m to 50e-6m and is centered at 25e-6 m.
- pixel 1 goes from position 50e-6 m to 100e-6 m and is centered at 75e-6 m

Nota: Most of the time you will need to pass the optional argument *origin="lower"* to matplotlib's *imshow* function when displaying the image to avoid confusion.

Complex detectors

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

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

Detectors classes

They are used to define families of detectors. In order to take the specificities of each detector into account, pyFAI contains about 58 detector class definitions (and 168 with aliases) which contain a mask (invalid pixels, gaps, ...) and a method to calculate the pixel positions in Cartesian coordinates. Available detectors can be printed using:

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

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

Nexus Detectors

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

- hexagonal pixels (i.e. Pixirad detectors, still under development)
- curved/bent imaging plates (i.e. Rigaku, Aarhus detector)
- pixel detectors with tiled modular (i.e. Xpad detectors from ImXpad)
- semi-cylindrical pixel detectors (i.e. Pilatus12M from Dectris or CirPad from Soleil).

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

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

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

```
detector2nexus -s halfccd.spline -o halfccd.h5
```

Conclusion

Detector definition in pyFAI is very versatile. Fortunately, most detectors are already described, making the usage transparent for most users. There are a couple of *Tutorials* on detector definition which will help you understanding the underlying mechanism:

- **Distortion** which explains how to correct images for geometric distortion
- **CCD-calibration** which explains how to calibrate such detectors for geometric distortion.

1.3.2 Geometry

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

Image representation in Python

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

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

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

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

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

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

Position of the observer

There are two (main) conventions when representing images:

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

- In diffraction application, the observer is situated at the sample position and looks at the detector, hence on the other side of the detector.

Because we measure (signed) angles, the origin is ideally situated at the lower left of the image. PyFAI being a diffraction application, it uses the later description. Changing the point of view behind the detector changes the sign of the azimuthal angle.

Default geometry in pyFAI

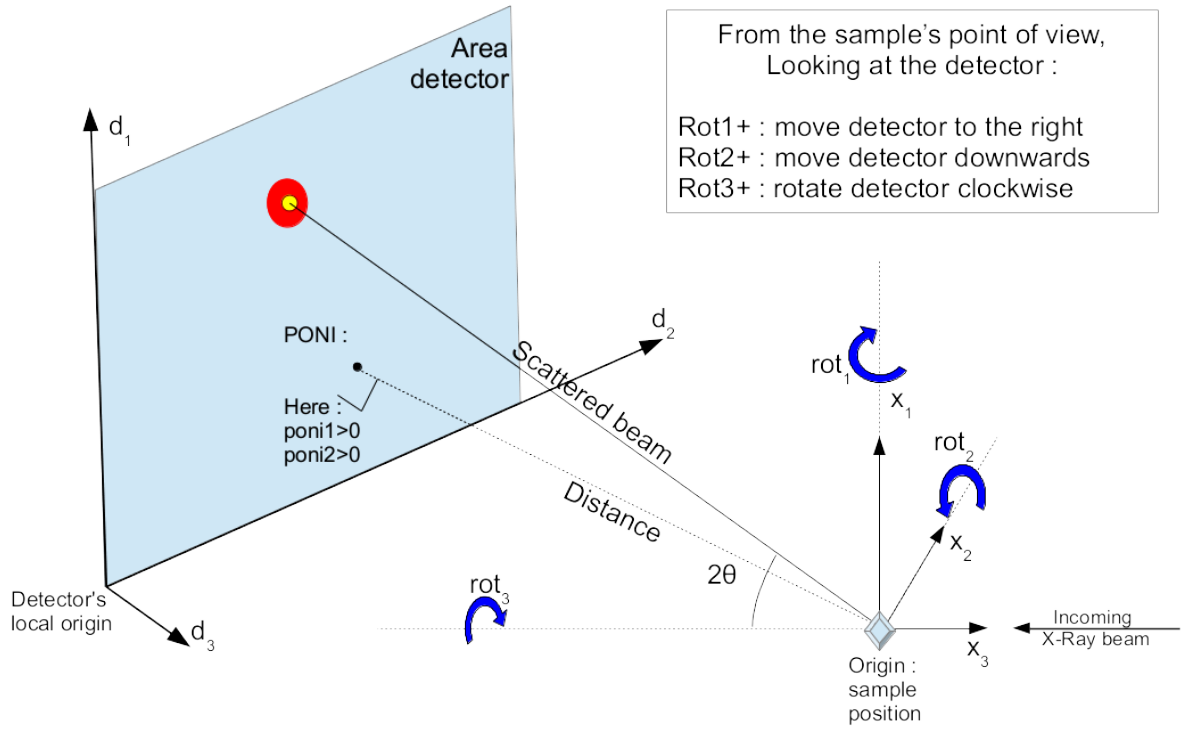
In the (most common) case of *transmission diffraction setup* on synchrotrons (like ESRF, Soleil, Petra3, SLS...) this makes looks like:

- Observer looking at the detector from the sample position:
- Origin at the lower left of the detector
- Axis 1 (i.e. y) being vertical, pointing upwards
- Axis 2 (i.e. x) being horizontal, pointing to the center of the storage ring
- Axis 3 (i.e. z) being horizontal, along the incident beam

Axis 3 is built in such a way to be orthogonal to the plane (1,2). Due to constraints on the origin and orientation of the azimuthal angle, chi, (1, 2, 3) is indirect orientation. This makes usually the PONI position with negative third coordiante (at z<0).

Detector position

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



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

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

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

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

There is also a tutorial *Tutorials* on the geometry which explains in detail the orientations of the different rotations used by pyFAI.

1.3.3 Calibration

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

The program **pyFAI-calib** is used for calibrating the experimental setup using a constrained least squares optimization on the Debye-Scherrer rings of a reference sample (LaB_6, CeO_2, silver behenate $AgBh$, ...) and saves the results into a *.poni* file. Alternatively, geometries calibrated using Fit2D can be imported into pyFAI, including geometric distortions (i.e. optical-fiber tapers distortion) described as *spline-files*. For Fit2D compatibility, please refer to the *Tutorials* on basic usage of pyFAI.

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

While entering the geometry of the experiment in a poni-file is possible, it is easier to perform a calibration, using the Debye-Sherrer rings of a reference sample called calibrant. About 30 calibrants are provided by pyFAI like LaB_6, ceria CeO_2, silicon, corrundum or silver behenate. Among other simple compound, all of the NIST Standard Reference Materials have been are tabulated and are directly available as calibrant. One can alternatively provide its own calibrant description files which is a simple text-file containing the largest d-spacing (in Angstrom) for a set of Miller plans. A useful reference to generate this file is the American Mineralogist database [AMD] or the Crystallographic Open database [COD].

The calibration procedure is divided into 4 major steps:

Pre-processing of images:

The typical pre-processing consists of the averaging (or better median filtering) of darks images. Dark current images are then subtracted from data and corrected for flat. The pre-processing is best performed using the *pyFAI-average* tool, which documentation is available in the *pyFAI scripts manual*.

If saturated pixels exists, the are likely to be treated like peaks but their positions will be wrong. It is advised to either mask them out or to desaturate them (pyFAI provides an option, but it is expensive in calculation time). A Mask drawing tool, called *pyFAI-drawmask*, is installed together with pyFAI and its documentation available in the *pyFAI scripts manual*.

To start the calibration the *pyFAI-calib* tool will need:

- an image with Debye-Sherrer rings
- the energy or the wavelength

- the calibrant name or the d-spacing file of the calibrant
- the detector description.

Peak-picking

Once started, *pyFAI-calib* will ask you to select rings. The Peak-picking consists in the identification of peaks and groups of peaks belonging to same ring. It can be performed by 4 methods described hereafter.

Massif detection

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

Blob detection

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

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

Steepest ascent

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

Monte-Carlo sampling

Series of peaks can be extracted using the Steepest Ascent on randomly selected seeds. This method can be biased towards an already known geometry by starting from points which are supposed to be on the ring.

Refinement of the parameters

After selecting groups of peaks, each of them is assigned to a Debye-Scherrer ring number (0-based numbering in python) and associated to a d-spacing value hence a theoretical 2theta value. A supervised least-squares refinement, performed on the difference of peak position's 2-theta values versus the expected ones from calibrant provides the 6-geometry parameters fitted.

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

Validation of the calibration

Validation by an human being of the geometry is an essential step: pyFAI will overlaps to the diffraction image, the lines corresponding to the various diffraction rings expected from the calibrant. Those lines should be in pretty good agreement with the rings of the scattering image. The average error per control point (delta 2theta error in radian) is printed out and offers a quantitative measurement of the relative quality of the fit for similar setups/experiment. Nevertheless its absolute value has no meaning, except the lower, the better.

Subsequently, pyFAI offers some validation options in to check the quality of the fit. some of them global, some of them limited to given rings.

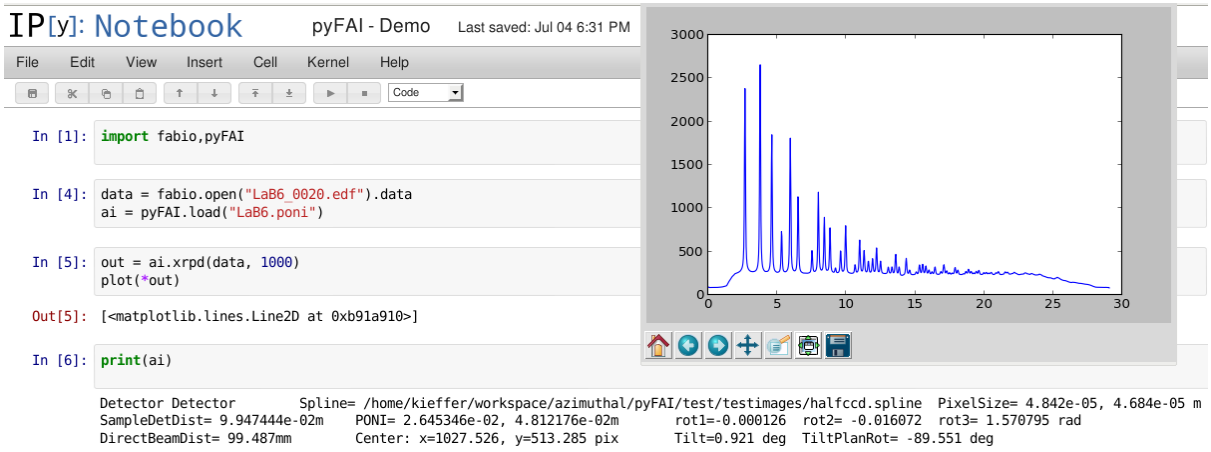
1.3.4 PyFAI executables

PyFAI was designed to be used by scientists needing a simple and effective tool for azimuthal integration. There are a certain number of scripts which will help you in preprocessing images (dark current, flat-field, averaging, ...), calibrating the geometry, performing the integration. Finally couple of specialized tool called diff_tomo and diff_map are available to reduce a 2D/3D-mapping experiment of 2D images into a 3D volume ($x, y, 2\theta$ for mapping or $rot, trans, 2\theta$ for tomography)

There are cookbooks on these scripts in *Cookbook recipes* and their complete manual pages are available in the *pyFAI scripts manual* section.

1.3.5 Python library

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



The *Tutorials* section makes heavy use of *IPython notebooks*, now called *Jupyter notebooks* to process data using *pyFAI*. The first tutorial also explains a bit how *Python* and *Jupyter* works to be able to perform basic processing efficiently with pyFAI.

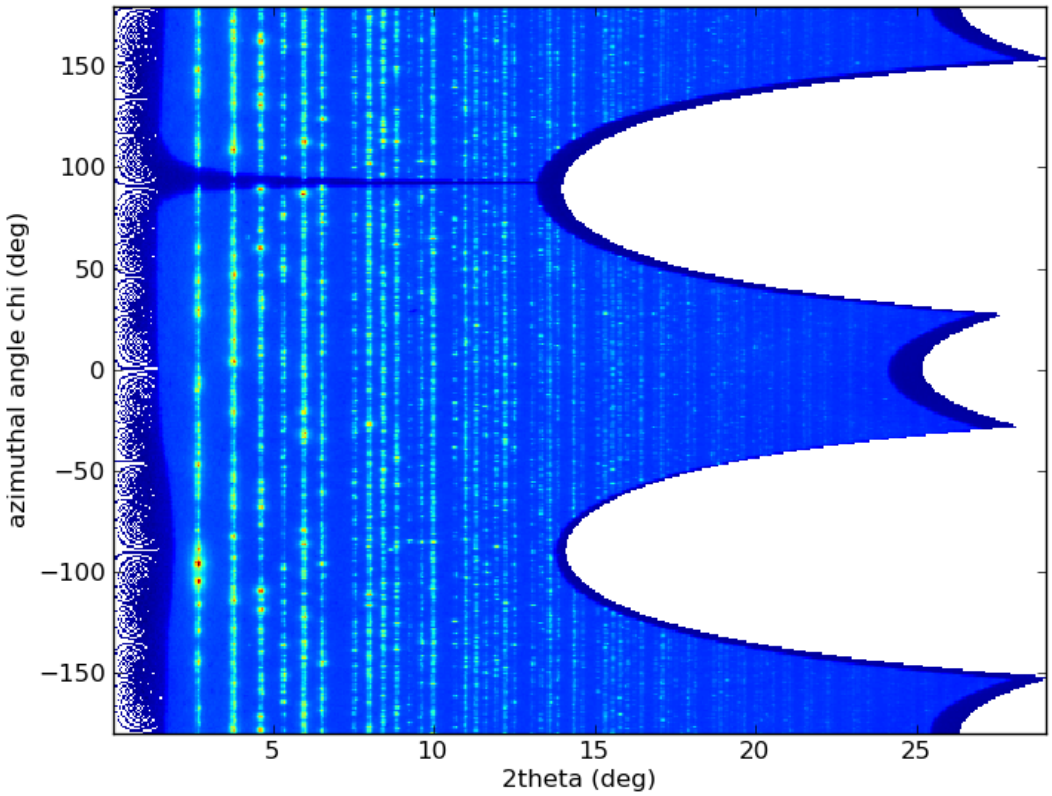
1.4 Regrouping mechanism

In pyFAI, regrouping is performed using a histogram-like algorithm. Each pixel of the image is associated to its polar coordinates $(2\theta, \chi)$ or (q, χ), then a pair of histograms versus 2θ (or q) are built, one non weighted for measuring the number of pixels falling in each bin and another weighted by pixel intensities (after dark-current subtraction, and corrections for flat-field, solid-angle and polarization). The division of the weighted histogram by the number of pixels per bin gives the average signal over the given corona which provides the diffraction pattern. $2D$ regrouping (called *caking* in FIT2D) is obtained in the same way using two-dimensional histograms over radial (2θ or q) and azimuthal angles (χ).

1.4.1 Pixel splitting algorithm

Powder diffraction patterns obtained by histogramming have a major weakness where pixel statistics are low. A manifestation of this weakness becomes apparent in the $2D$-regrouping where most of the bins close to the beam-stop are not populated by any pixel. In this figure, many pixels are missing in the low 2θ region, due to

the arbitrary discretization of the space in pixels as intensities were assigned to each pixel center which does not reflect the physical reality of the scattering experiment.



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

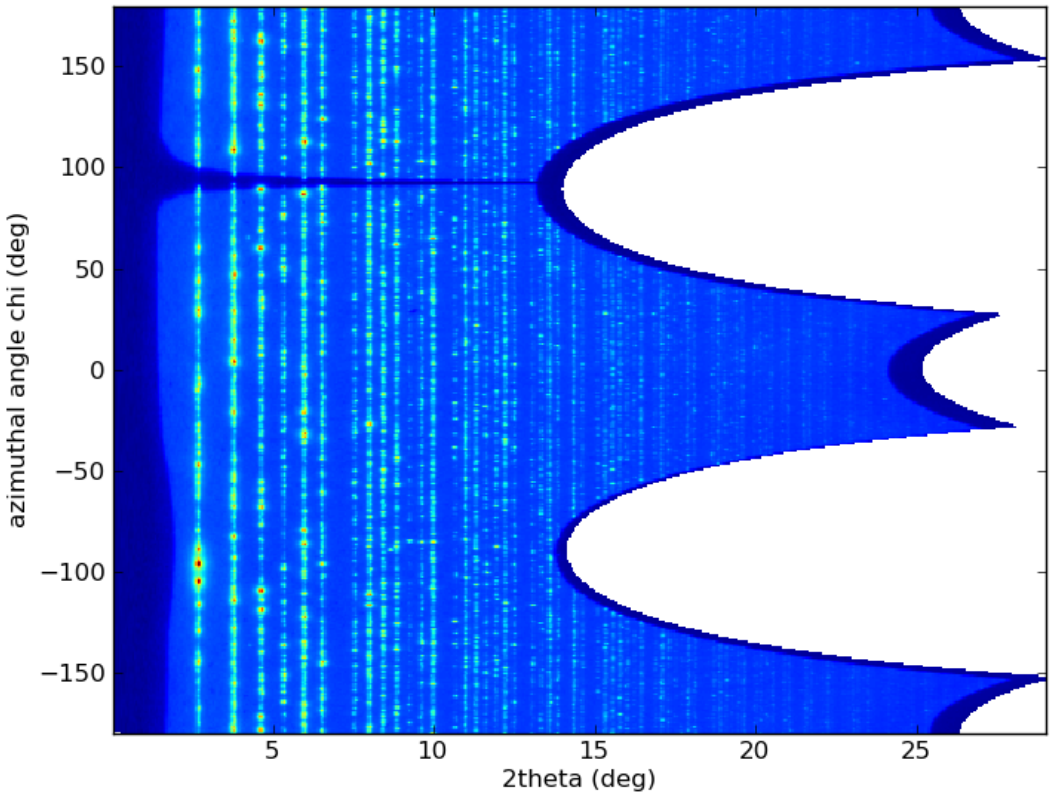
1.4.2 Performances and migration to native code

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

1.4.3 Parallel implementation

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

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



transformation in parallel programming.

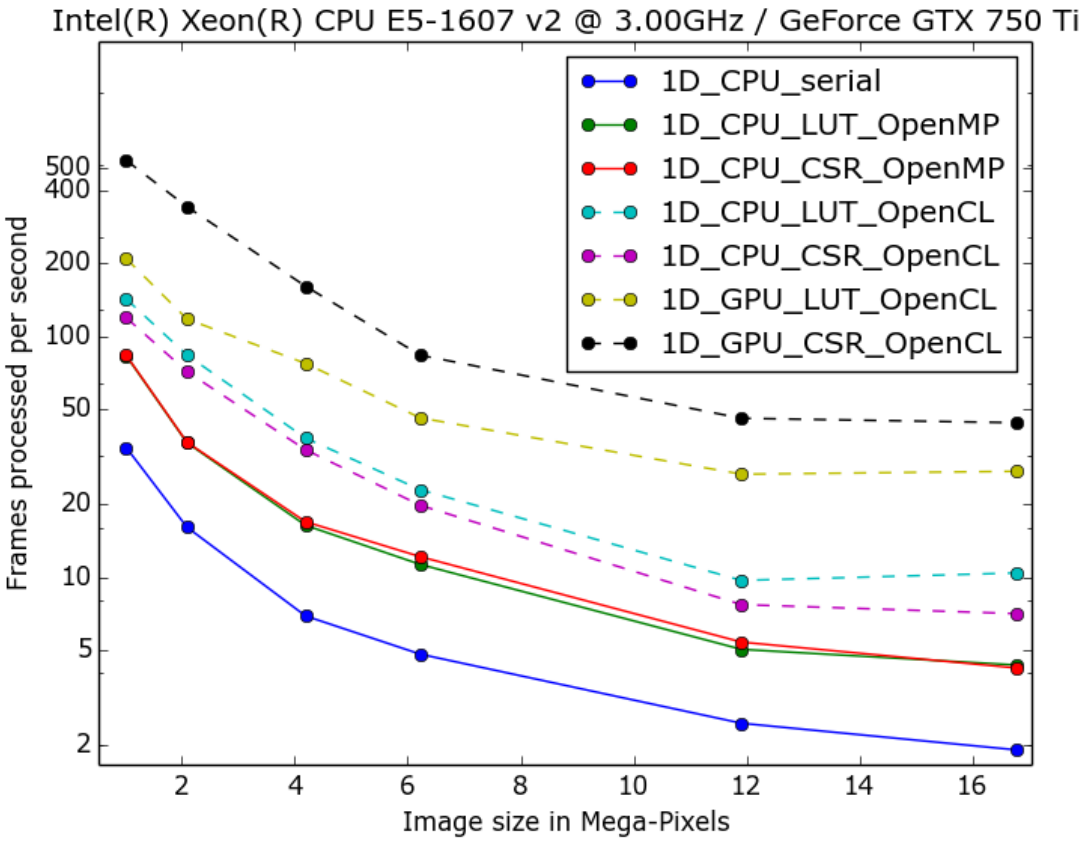
The correspondence between pixels and output bins can be stored in a look-up table (LUT) together with the pixel weight which make the integration look like a simple (if large and sparse) matrix vector product. This look-up table size depends on whether pixels are split over multiple bins and to exploit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

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

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

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

neously.



1.5 Related Work

There are many projects which are already relying on pyFAI: Dioptas, NanoPeakCell, Dpdak, PySAXS, xPDFSuite ... There is a list of *Program using pyFAI as a library* on the ecosystem page.

1.6 Conclusion

The library pyFAI was developed with two main goals:

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

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

1.6.1 Acknowledgments

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

CHAPTER

TWO

COOKBOOK RECIPES

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

2.1 Calibration of a diffraction setup

The files needed for this cookbook can be downloaded on: http://www.silx.org/pub/pyFAI/cookbook/calibration/
You can download them to practice:

```
$ mkdir calibration
$ cd calibration
$ wget http://www.silx.org/pub/pyFAI/cookbook/calibration/LaB6_29.4keV.tif
$ wget http://www.silx.org/pub/pyFAI/cookbook/calibration/F_K4320T_Cam43_30012013_distorsion.spli
$ ls
F_K4320T_Cam43_30012013_distorsion.spline  LaB6_29.4keV.tif
```

The associated video is here

2.1.1 Review your calibration image

As viewer, try fabio_viewer from the FabIO package

```
$ fabio_viewer LaB6_29.4keV.tif
```

You may need to pre-process your data, *pyFAI-average* is a tool to perform pixel-wise transformation. In this example, the image file was obtained using a "max filter" over 20 frames with pyFAI-average:

```
$ pyFAI-average -m max -F tif -o LaB6_29.4keV.tif ref_lab6_00??.edf
```

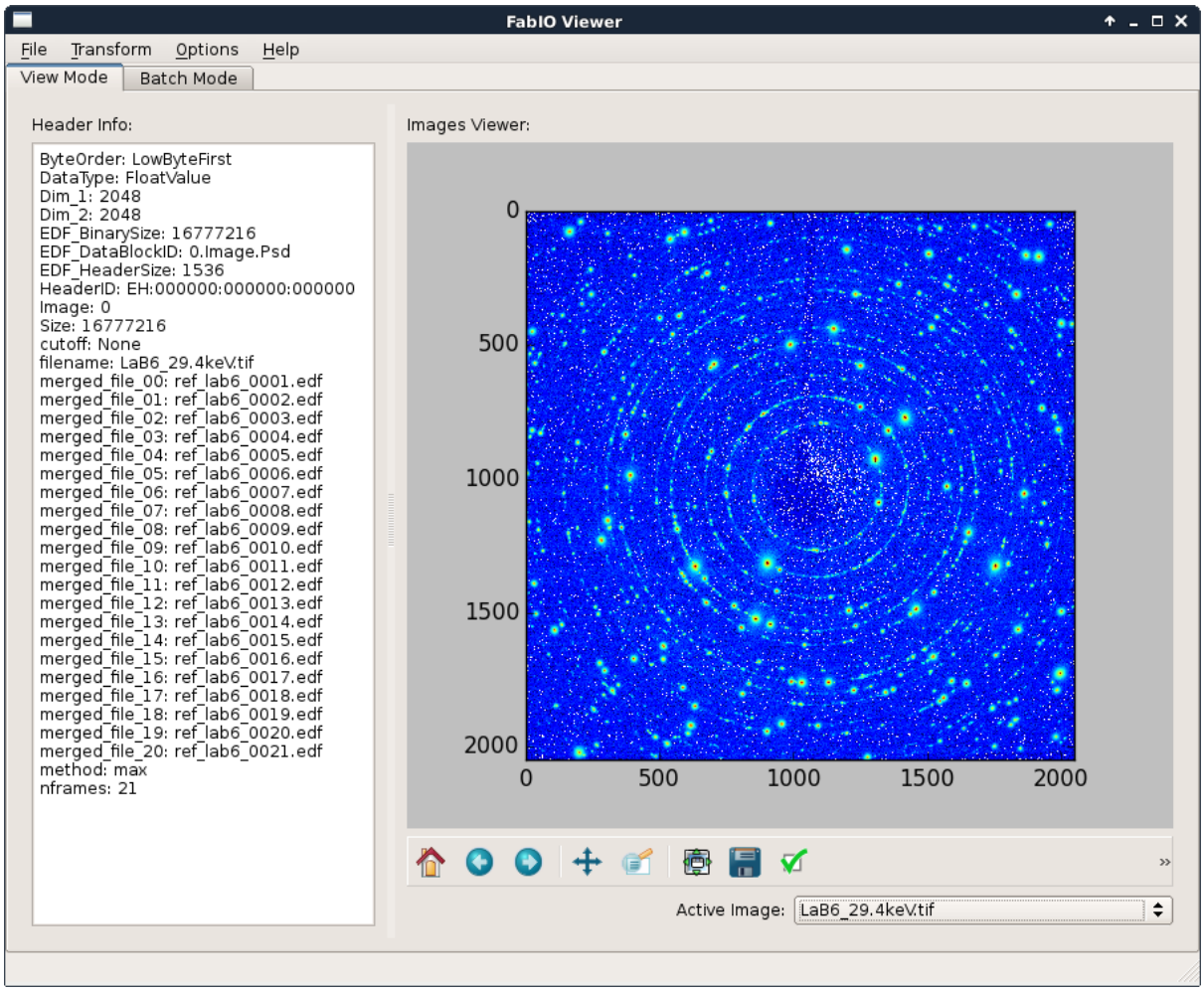
2.1.2 Get all additional data

- calibrant used: LaB6 according to the name of the file
- the energy or the wavelength, 29.4keV according to the name of the file
- detector geometry: there is a spline file along with the file
- masks, dark, flat ... are not going to be used to keep things simple.

2.1.3 Start pyFAI-calib

Use the man page (or –help) to see all options, or go to the *pyFAI scripts manual* section.

Here we just provide the energy, the detector disortion file and the calibrant options in addition to the image file.

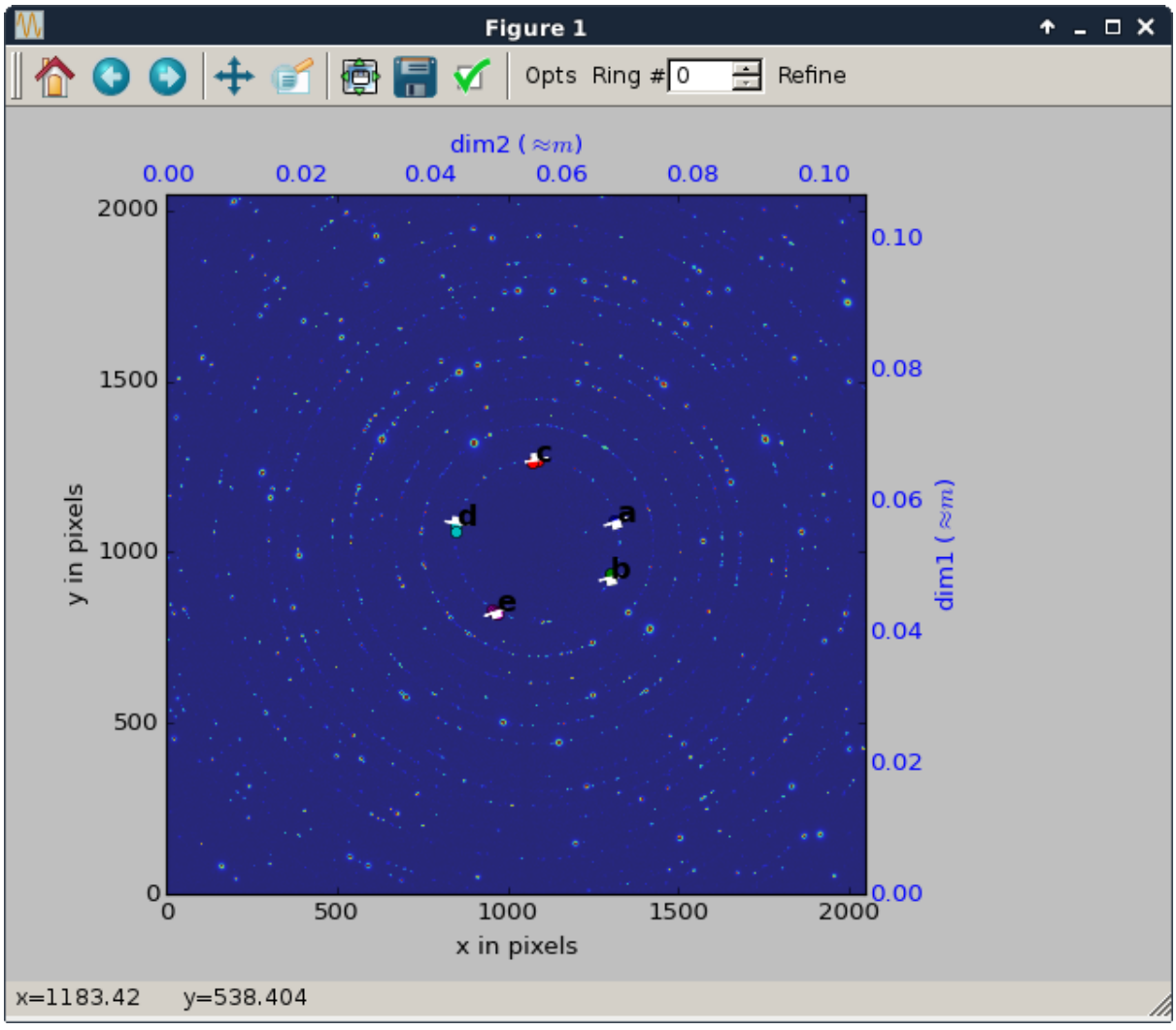

FabIO Viewer
File
Transform
Options
Help
View Mode
Batch Mode
Header Info:
ByteOrder: LowByteFirst
DataType: FloatValue
Dim_1: 2048
Dim_2: 2048
EDF_BinarySize: 16777216
EDF_DataBlockID: 0.Image.Psd
EDF_HeaderSize: 1536
HeaderID: EH:000000:000000:000000
Image: 0
Size: 16777216
cutoff: None
filename: LaB6_29.4keV.tif
merged_file_00: ref_lab6_0001.edf
merged_file_01: ref_lab6_0002.edf
merged_file_02: ref_lab6_0003.edf
merged_file_03: ref_lab6_0004.edf
merged_file_04: ref_lab6_0005.edf
merged_file_05: ref_lab6_0006.edf
merged_file_06: ref_lab6_0007.edf
merged_file_07: ref_lab6_0008.edf
merged_file_08: ref_lab6_0009.edf
merged_file_09: ref_lab6_0010.edf
merged_file_10: ref_lab6_0011.edf
merged_file_11: ref_lab6_0012.edf
merged_file_12: ref_lab6_0013.edf
merged_file_13: ref_lab6_0014.edf
merged_file_14: ref_lab6_0015.edf
merged_file_15: ref_lab6_0016.edf
merged_file_16: ref_lab6_0017.edf
merged_file_17: ref_lab6_0018.edf
merged_file_18: ref_lab6_0019.edf
merged_file_19: ref_lab6_0020.edf
merged_file_20: ref_lab6_0021.edf
method: max
nframes: 21
Images Viewer:
0
500
1000
1500
2000
0
500
1000
1500
2000
Active Image:
LaB6_29.4keV.tif

```
$ pyFAI-calib -e 29.4 -s F_K4320T_Cam43_30012013_distorsion.spline -c LaB6 LaB6_29.4keV.tif
FixedParameters(['wavelength'])
ERROR:pyFAI.peak_picker:ControlPoint.load: No such file LaB6_29.4keV.npt
INFO:pyFAI.massif:Image size is (2048, 2048)
INFO:pyFAI.massif:Binning size is [2, 2]
INFO:pyFAI.massif:Labeling found 7272 massifs.
INFO:pyFAI.massif:Labeling found 7272 massifs.
INFO:root:Please select rings on the diffraction image. In parenthesis, some modified shortcuts fo
 * Right-click (click+n):          try an auto find for a ring
 * Right-click + Ctrl (click+b):   create new group with one point
 * Right-click + Shift (click+v): add one point to current group
 * Right-click + m (click+m):      find more points for current group
 * Center-click or (click+d):      erase current group
 * Center-click + 1 or (click+1): erase closest point from current group
Please press enter when you are happy with your selection
```

2.1.4 Pick peaks

To perform the calibration one needs to create control points and assign them to rings.

- Right click on a few (5) points in the inner-most ring which has the index number #0
- Increase the counter on the top right to change the ring number and pick a few more points on the corresponding ring (again with right click).

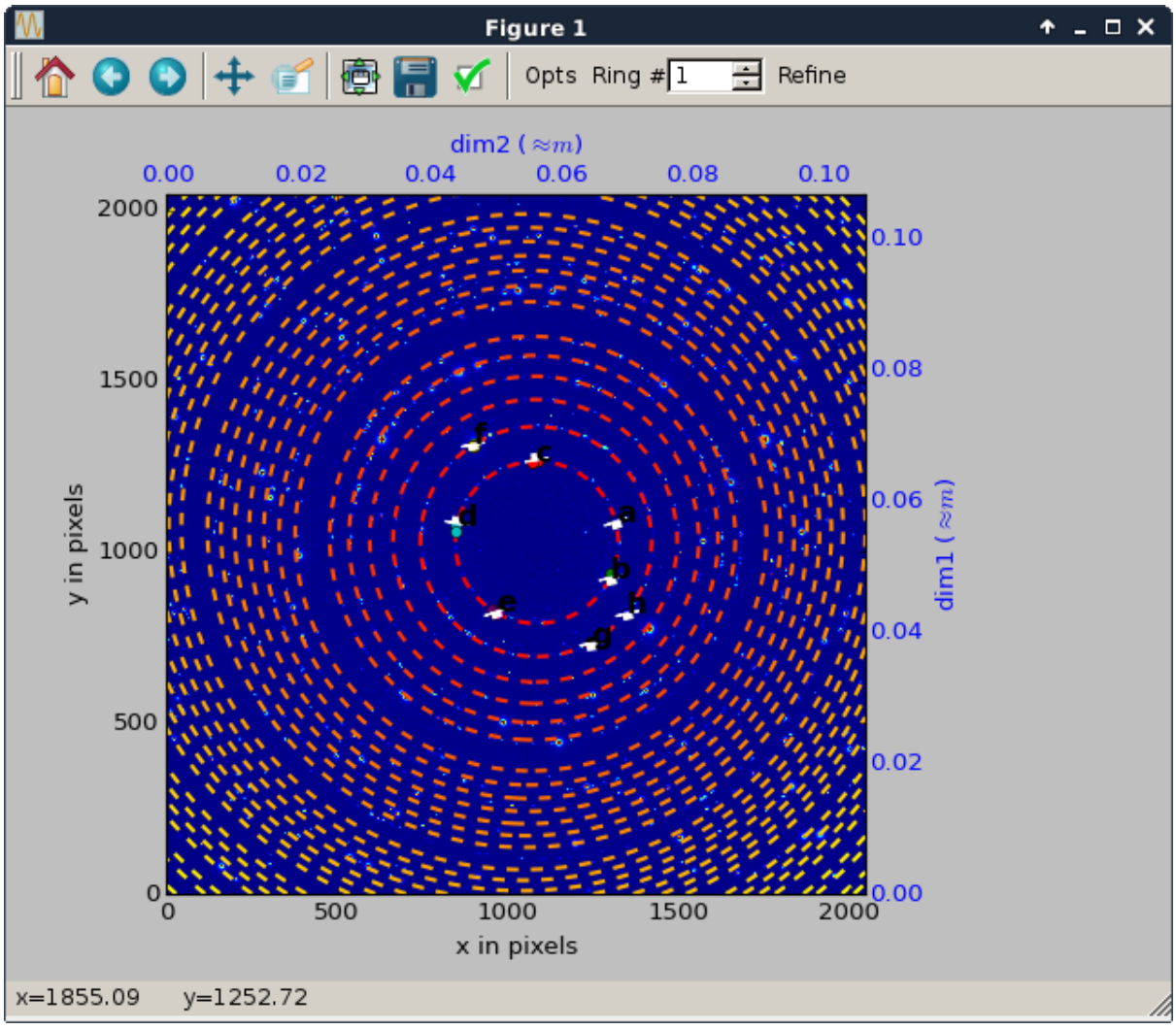


2.1.5 Review the group of peaks

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

Once done with all groups, the position of the expected rings is overlaid to the diffraction image. You may need to unzoom to view them !

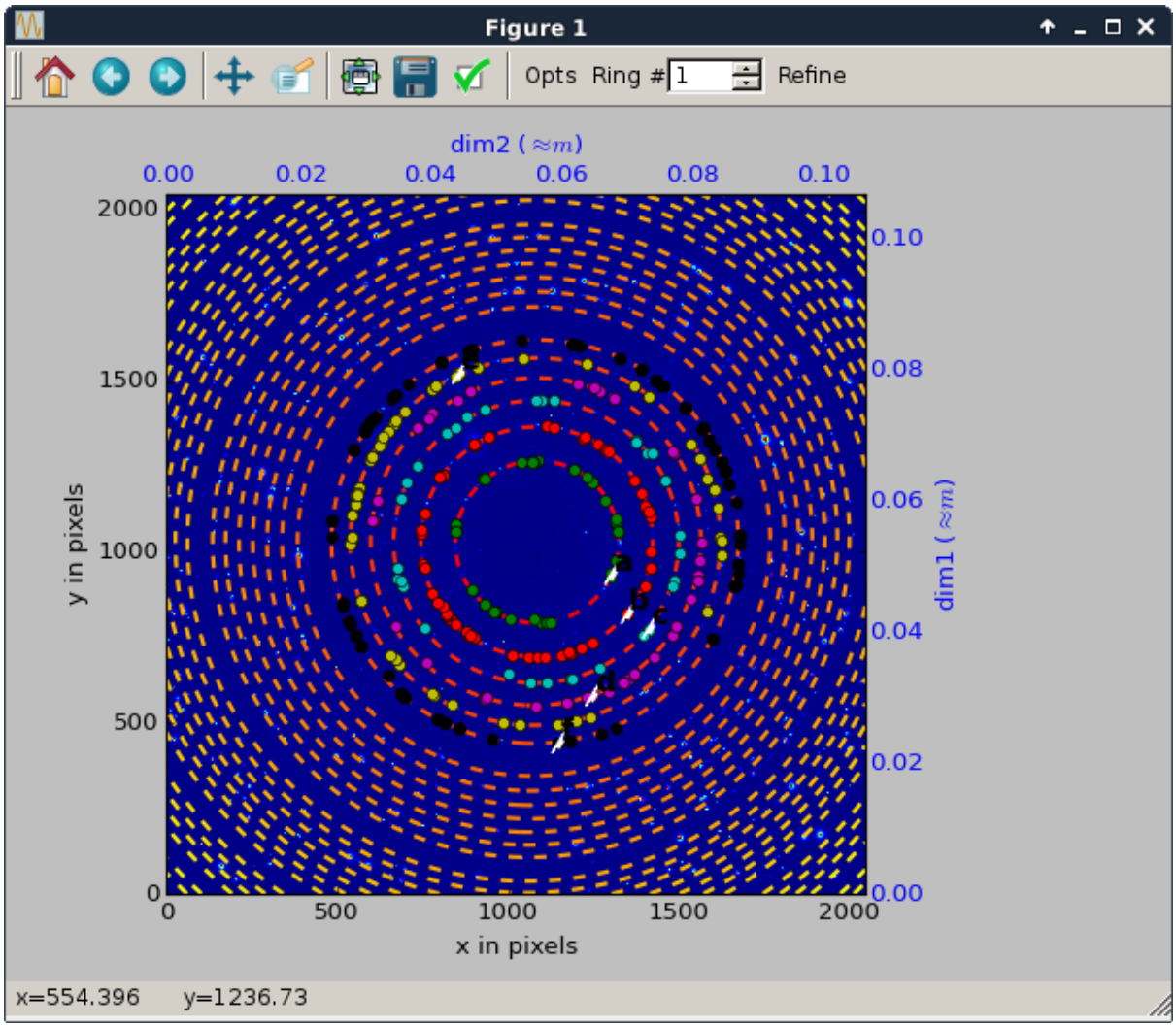
```
Now fill in the ring number. Ring number starts at 0, like point-groups.
Point group # a (4 points)  (1315.9,1090.1)   [default=0] Ring#
Point group # b (5 points)  (1302.0, 926.0)   [default=0] Ring#
Point group # c (2 points)  (1085.3,1268.1)   [default=0] Ring#
Point group # d (2 points)  ( 850.1,1083.3)   [default=0] Ring#
Point group # e (5 points)  ( 965.1, 825.7)   [default=0] Ring#
Point group # f (4 points)  ( 898.2,1315.9)   [default=1] Ring#
Point group # g (2 points)  (1244.6, 733.6)   [default=1] Ring#
Point group # h (2 points)  (1350.1, 821.9)   [default=1] Ring#
Optimization terminated successfully.    (Exit mode 0)
```



2.1.6 Acquire some more control points

- Use *recalib* to extract a new set of control points, specify the number of rings, first a few of them then more
- You may want to free/fix/bound some parameter then *refine* again

```
Fixed: wavelength
Modify parameters (or ? for help)?   recalib 6
[...]
Fixed: wavelength
Modify parameters (or ? for help)?   recalib 15
[...]
Fixed: wavelength
Modify parameters (or ? for help)?   recalib 25
```



2.1.7 Visualize the integrated patterns

- integrate to view the integrated pattern
- then extract a few extra rings ...
- the geometry is displayed on the screen, and saved automatically in the poni-file

```
Fixed: wavelength
Modify parameters (or ? for help)?   integrate
```

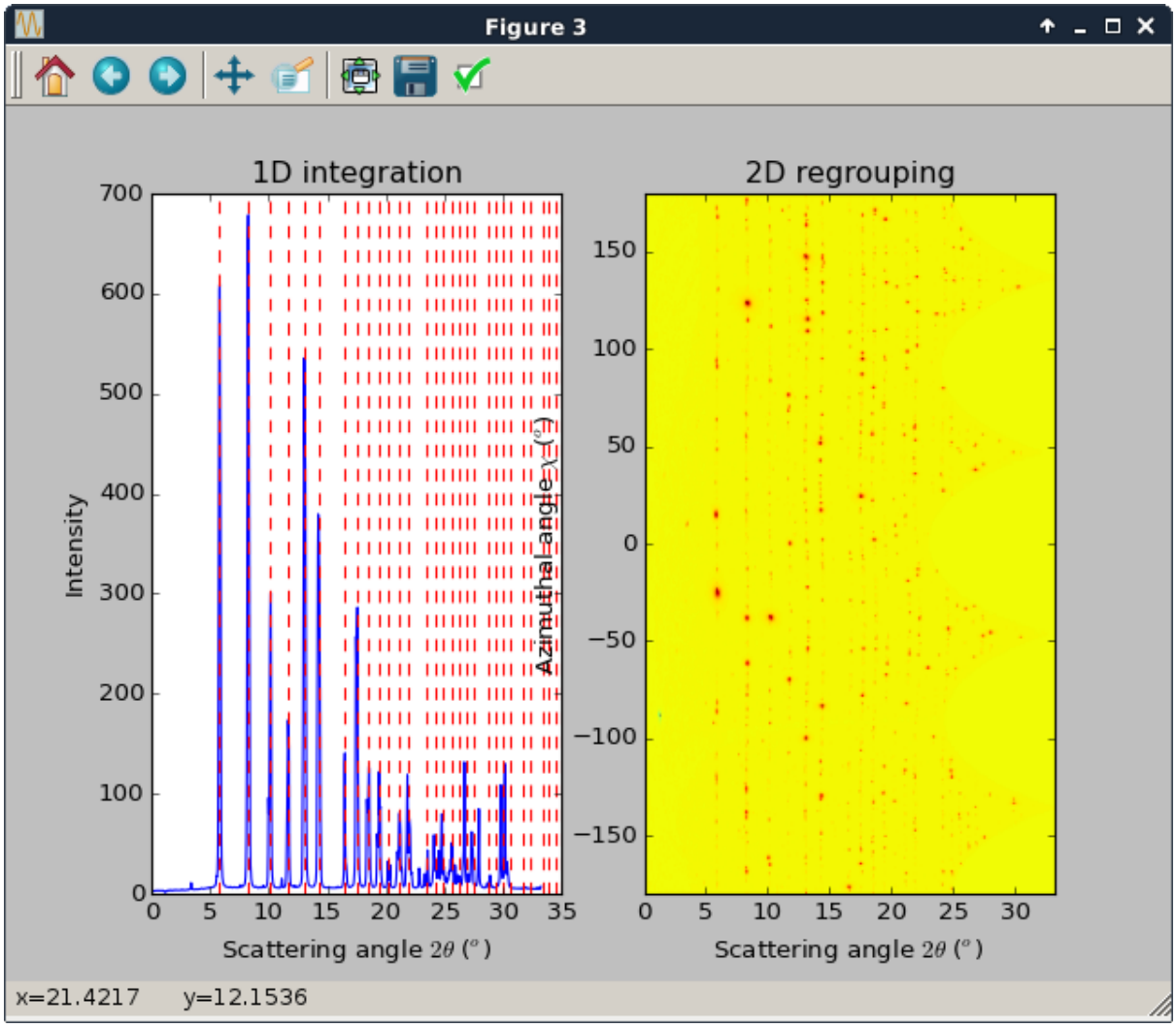


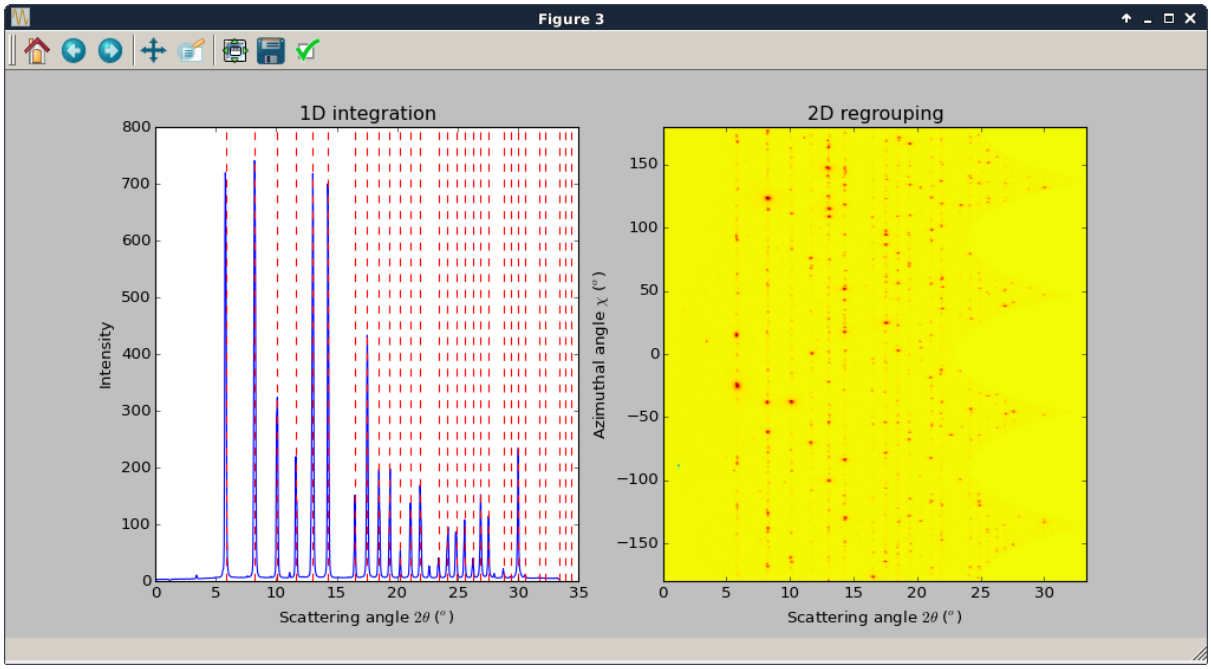
Figure 3
1D integration
2D regrouping
Intensity
Azimuthal angle χ (^{o})
Scattering angle 2θ (^{o})
Scattering angle 2θ (^{o})
x=21.4217
y=12.1536

2.1.8 Quit

```
Fixed: wavelength
Modify parameters (or ? for help)?   quit
$

ls
F_K4320T_Cam43_30012013_distorsion.spline  LaB6_29.4keV.npt   LaB6_29.4keV.tif
LaB6_29.4keV.azim                          LaB6_29.4keV.poni  LaB6_29.4keV.xy
```

All different geometries have been saved into the LaB6_29.4keV.poni file and can directly be used with *pyFAI-integrate*. All control points are saved into LaB6_29.4keV.npt.

Final notes: In this case the calibration is far from being good and it is likely the distortion file applied is not the proper one according to many the waves on the 2D integration pattern. If one flips up-down the spline file prior to the calibration, the residual error is twice lower but the goes far beyond this cookbook. Here is the diffraction pattern from a properly calibrated setup:




```
from pyFAI.spline import Spline
s = Spline("F_K4320T_Cam43_30012013_distorsion.spline")
ud = s.flipud()
ud.write("flipud.spline")
```

2.2 Azimuthal integration using the graphical user interface

Associated video

2.2.1 Look at your integrated patterns

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

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

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

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

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

2.2.2 Integrate a bunch of images

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

2.2.3 Start pyFAI-intgrate

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

2.2.4 Set the geometry

Simply load the PONI file and check the populated fields.

2.2.5 Azimuthal integration options

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

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

2.2.6 Select the device for processing

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

Press OK to start the processing.

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

2.2.7 Run it again to perform caking

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

2.2.8 Visualize the integrated patterns

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

That's all.

2.3 Azimuthal integration using scripts

2.4 Azimuthal integration using Python

This cookbook is now available a a tutorial using jupyter on the *Tutorials* section.

For more in depth explanation, see the *Tutorials* section.

CHAPTER

THREE

TUTORIALS

Tutorials explain the Python interface of *pyFAI* and use the jupyter notebook interface, formerly known as ipython notebooks. The two first tutorials are an introduction to the usage of pyFAI from Python for diffraction data reduction. The subsequent tutorials are more in depth explanation and require a good Python fluency and to a certain extent, of the pyFAI library.

3.1 Introduction to the tutorials

3.1.1 The iPython/Jupyter notebook

The document you are seeing it may be a PDF or a web page or another format, but what is important is how it has been made ...

According to this article in Nature the Notebook, invented by iPython, and now part of the Jupyter project, is the revolution for data analysis which will allow reproducable science.

This tutorial will introduce you to how to access to the notebook, how to use it and perform some basic data analysis with it and the pyFAI library.

3.1.2 Getting access to the notebook

There are many cases ...

Inside ESRF

The simplest case as the data analysis unit offers you a notebook server: just connect your web browser to http://scisoft13:8000 and authenticate with your ESRF credentials.

Outside ESRF with an ESRF account.

The JupyterHub server is not directly available on the internet, but you can login into the firewall to forward the web server:

ssh -XC -p5022 -L8000:scisoft13:8000 user@firewall.esrf.fr

Once logged in ESRF, keep the terminal opened and browse on your local computer to http://localhost:8000 to authenticate with your ESRF credentials. Do not worry about confidentiality as the connection from your computer to the ESRF firewall is encrypted by the SSH connection.

Other cases

In the most general case you will need to install the notebook on your local computer in addition to pyFAI and FabIO to follow the tutorial. WinPython provides it under windows. Please refer to the installation procedure of pyFAI to install locally pyFAI on your computer

3.1.3 Getting trained in using the notebook

There are plenty of good tutorials on how to use the notebook. This one presents a quick overview of the Python programming language and explains how to use the notebook. Reading it is strongly encouraged before proceeding to the pyFAI itself.

Anyway, the most important information is to use **Control-Enter** to evaluate a cell.

In addition to this, we will need to download some files from the internet. The following cell contains a piece of code to download files. You do not need to understand what it does, but you may have to adjust the proxy settings to be able to connect to internet, especially at ESRF.

```
import os, sys
os.environ["http_proxy"] = "http://proxy.site.com:3128"

def download(url):
    """download the file given in URL and return its local path"""
    if sys.version_info[0]<3:
        from urllib2 import urlopen, ProxyHandler, build_opener
    else:
        from urllib.request import urlopen, ProxyHandler, build_opener
    dictProxies = {}
    if "http_proxy" in os.environ:
        dictProxies['http'] = os.environ["http_proxy"]
        dictProxies['https'] = os.environ["http_proxy"]
    if "https_proxy" in os.environ:
        dictProxies['https'] = os.environ["https_proxy"]
    if dictProxies:
        proxy_handler = ProxyHandler(dictProxies)
        opener = build_opener(proxy_handler).open
    else:
        opener = urlopen
    target = os.path.split(url)[-1]
    with open(target,"wb") as dest, opener(url) as src:
        dest.write(src.read())
    return target
```

3.1.4 Introduction to diffraction image analysis using the notebook

All the tutorials in pyFAI are based on the notebook and if you wish to practice the exercises, you can download the notebook files (.ipynb) from Github

Load and display diffraction images

First of all we will download an image and display it. Displaying it the right way is important as the orientation of the image imposes the azimuthal angle sign.

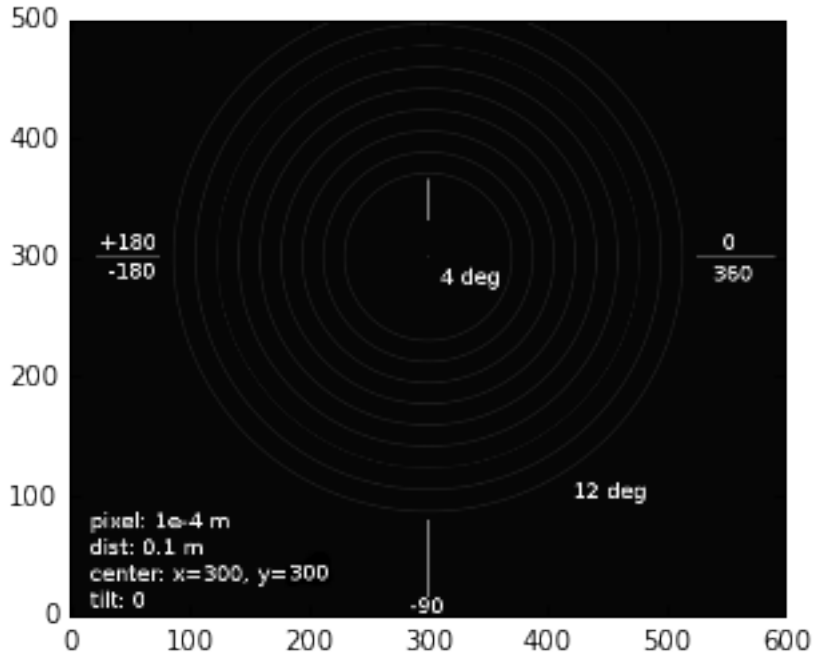
```
#initializes the visualization module
%pylab inline

Populating the interactive namespace from numpy and matplotlib

moke = download("http://www.silx.org/pub/pyFAI/testimages/moke.tif")
print(moke)

moke.tif
```

The *moke.tif* image we just downloaded is not a real diffraction image but it is a test pattern used in the tests of pyFAI.

Prior to displaying it, we will use the Fable Input/Output library to read the content of the file:

```
import fabio
img = fabio.open(moke).data
imshow(img, origin="lower", cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fd7aaa975c0>
```

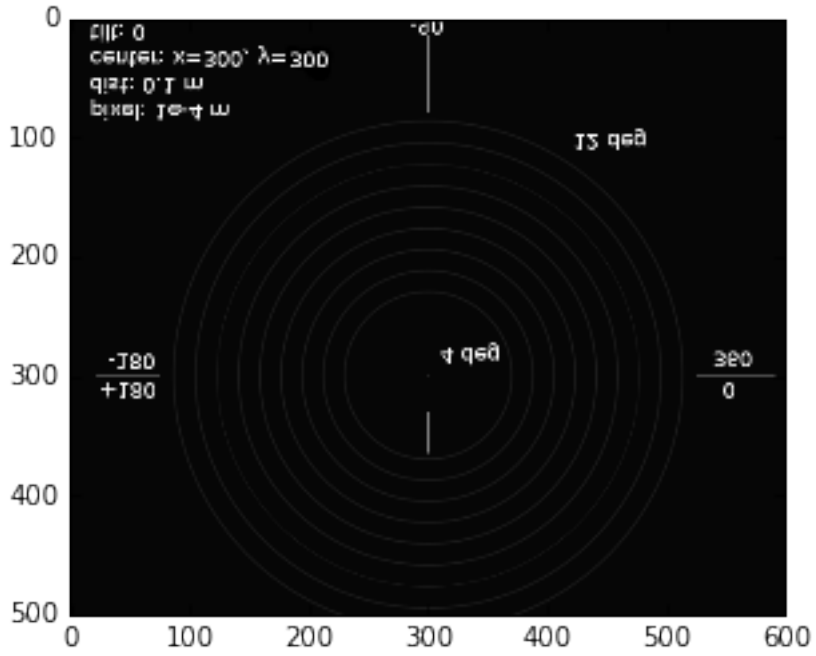


As you can see, the image looks like an archery target. The option *origin="lower"* of *imshow* allows to display the image with the origin at the **lower left** of the image.

Displaying the image without this option ends with having the azimuthal angle (which angles are displayed in degrees on the image) to turn clockwise, so the inverse of the trigonometric order.

```
imshow(img, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fd7aaa70b00>
```



Nota: Displaying the image properly or not does not change the content of the image or its representation in memory, it only changes its representation, which is important only for the user. **DO NOT USE** *numpy.flipud* or other array-manipulation which changes the memory representation of the image. This is likely to mess-up all your subsequent calculation.

1D azimuthal integration

To perform an azimuthal integration of this image, we need to create an **AzimuthalIntegrator** object we will call *ai*. Fortunately, the geometry is explained on the image.

```
import pyFAI
ai = pyFAI.AzimuthalIntegrator(dist=0.1, pixel1=1e-4, pixel2=1e-4)
print(ai)

Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
SampleDetDist= 1.000000e-01m        PONI= 0.000000e+00, 0.000000e+00m       rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=0.000, y=0.000 pix    Tilt=0.000 deg  tiltPlanRotation= 0.0
```

Printing the *ai* object displays 3 lines:

- The detector definition, here a simple detector with square, regular pixels with the right size
- The detector position in space using the *pyFAI* coordinate system
- The detector position in space using the *FIT2D* coordinate system

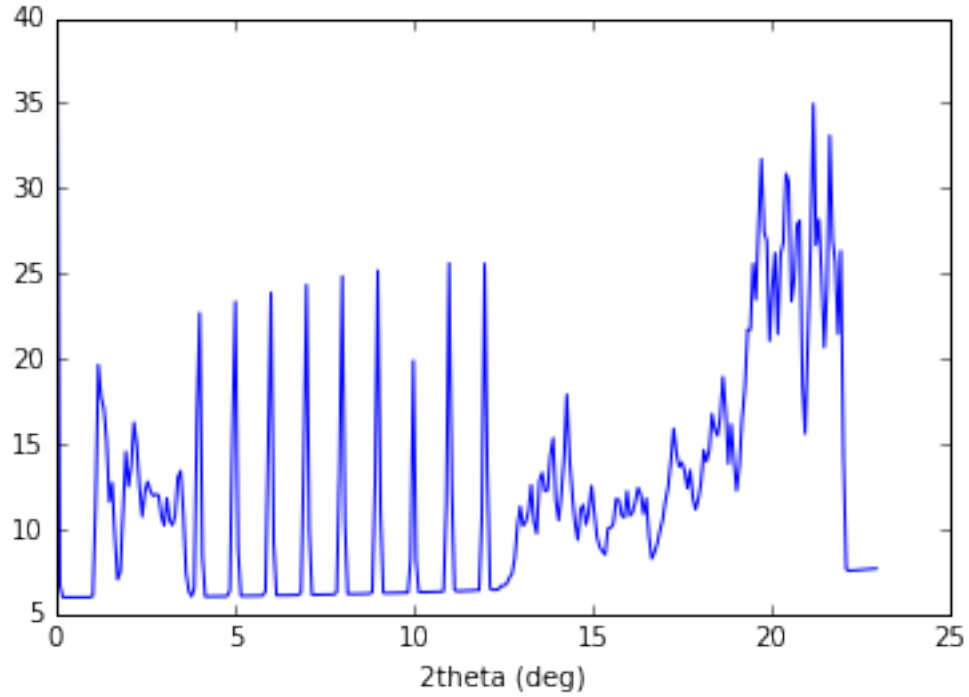
Right now, the geometry in the *ai* object is wrong. It may be easier to define it correctly using the *FIT2D* geometry which uses pixels for the center coordinates (but the sample-detector distance is in millimeters).

```
help(ai.setFit2D)

Help on method setFit2D in module pyFAI.geometry:

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

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

    @param direct: direct distance from sample to detector along the incident beam (in millimeter
    @param tilt: tilt in degrees
    @param 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

    @param pixelX,pixelY: as in fit2d they ar given in micron, not in meter
    @param centerX, centerY: pixel position of the beam center
    @param splineFile: name of the file containing the spline

ai.setFit2D(100, 300, 300)
print(ai)

Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
SampleDetDist= 1.000000e-01m        PONI= 3.000000e-02, 3.000000e-02m       rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=300.000, y=300.000 pix        Tilt=0.000 deg  tiltPlanRotati
```

With the *ai* object properly setup, we can perform the azimuthal integration using the *intergate1d* method. This methods takes only 2 mandatory parameters: the image to integrate and the number of bins. We will provide a few other to enforce the calculations to be performed in 2theta-space and in degrees:

```
tth, I = ai.integrate1d(img, 300, unit="2th_deg")
plot(tth, I, label="moke")
xlabel("2theta (deg)")
```

```
<matplotlib.text.Text at 0x7fd7aab3e908>
```



As you can see, the 9 rings gave 9 sharp peaks at 2theta position regularly ranging from 4 to 12 degrees as expected from the image annotation.

Nota: the default unit is "q_nm^1", so the scattering vector length expressed in inverse nanometers. To be able to calculate q, one needs to specify the wavelength used (here we didn't). For example: ai.wavelength = 1e-10

To save the content of the integrated pattern into a 2 column ASCII file, one can either save the (tth, I) arrays, or directly ask pyFAI to do it by providing an output filename:

```
ai.integrate1d(img, 30, unit="2th_deg", filename="moke.dat")
!cat moke.dat
```

```
# == pyFAI calibration ==
# SplineFile: None
# PixelSize: 1.000e-04, 1.000e-04 m
# PONI: 3.000e-02, 3.000e-02 m
# Distance Sample to Detector: 0.1 m
# Rotations: 0.000000 0.000000 0.000000 rad
#
# == Fit2d calibration ==
# Distance Sample-beamCenter: 100.000 mm
# Center: x=300.000, y=300.000 pix
# Tilt: 0.000 deg  TiltPlanRot: 0.000 deg
#
# Polarization factor: None
# Normalization factor: 1.0
# --> moke.dat
#       2th_deg              I
  3.831631e-01    6.384597e+00
  1.149489e+00    1.240657e+01
  1.915815e+00    1.222277e+01
  2.682141e+00    1.170348e+01
```

```
3.448468e+00    9.964798e+00
4.214794e+00    8.913503e+00
4.981120e+00    9.104074e+00
5.747446e+00    9.242975e+00
6.513772e+00    6.136262e+00
7.280098e+00    9.039030e+00
8.046424e+00    9.203415e+00
8.812750e+00    9.324570e+00
9.579076e+00    6.470130e+00
1.034540e+01    7.790757e+00
1.111173e+01    9.410036e+00
1.187805e+01    9.464832e+00
1.264438e+01    7.749060e+00
1.341071e+01    1.151200e+01
1.417703e+01    1.324891e+01
1.494336e+01    1.038730e+01
1.570969e+01    1.069764e+01
1.647601e+01    1.056094e+01
1.724234e+01    1.286720e+01
1.800866e+01    1.323239e+01
1.877499e+01    1.548398e+01
1.954132e+01    2.364553e+01
2.030764e+01    2.537154e+01
2.107397e+01    2.512984e+01
2.184029e+01    2.191267e+01
2.260662e+01    7.605135e+00
```

Here the excalmation mark indicates the notebook to call the *cat* command from UNIX to print the content of the file. This "moke.dat" file contains in addition to the 2th/I value, a header commented with "#" with the geometry used to perform the calculation.

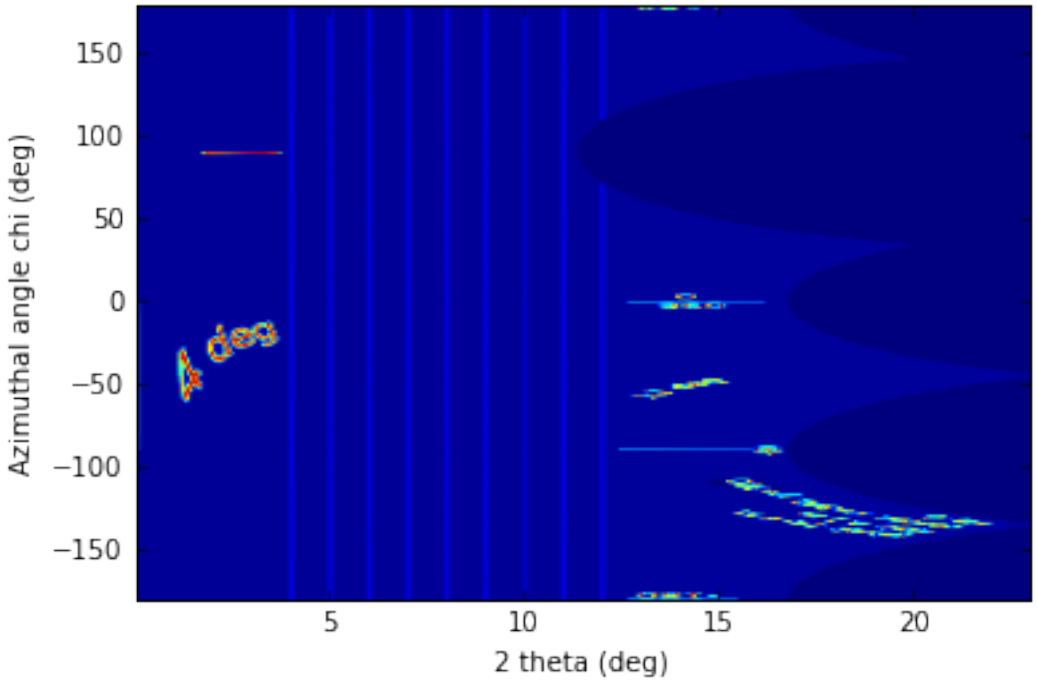
Nota: The *ai* object has initialized the geometry on the first call and re-uses it on subsequent calls. This is why it is important to re-use the geometry in performance critical applications.

2D integration or Caking

One can perform the 2D integration which is called caking in FIT2D by simply calling the *intrgate2d* method with 3 mandatroy parameters: the data to integrate, the number of radial bins and the number of azimuthal bins.

```
I, tth, chi = ai.integrate2d(img, 300, 360, unit="2th_deg")
imshow(I, origin="lower", extent=[tth.min(), tth.max(), chi.min(), chi.max()], aspect="auto")
xlabel("2 theta (deg)")
ylabel("Azimuthal angle chi (deg)")
```

```
<matplotlib.text.Text at 0x7fd7aaa0aa58>
```



The displayed image presents the "caked" image with the radial and azimuthal angles properly set on the axes. Search for the -180, -90, 360/0 and 180 mark on the transformed image.

Like *integrate1d*, *integrate2d* offers the ability to save the intgrated image into an image file (EDF format by default) with again all metadata in the headers.

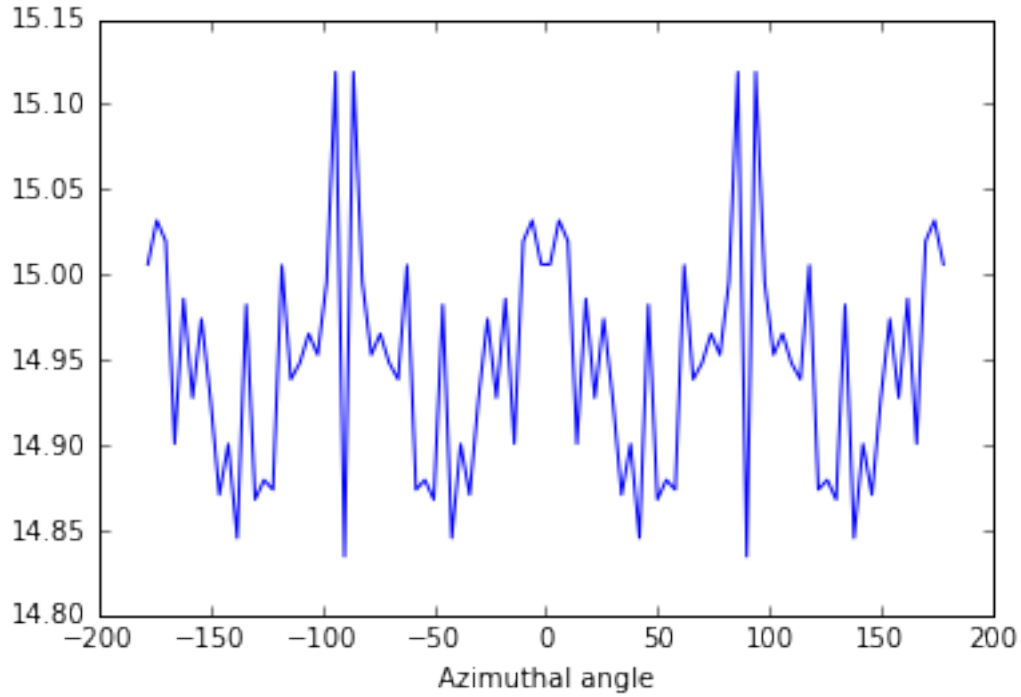
Radial integration

Radial integration can directly be obtained from Caked images:

```
target = 8 #degrees
#work on fewer radial bins in order to have an actual averaging:
I, tth, chi = ai.integrate2d(img, 100, 90, unit="2th_deg")
column = argmin(abs(tth-target))
print("Column number %s"%column)
plot(chi, I[:,column])
xlabel("Azimuthal angle")
```

```
Column number 34
```

```
<matplotlib.text.Text at 0x7fd7a833dcc0>
```



Nota: the pattern with higher noise along the diagonals is typical from the pixel splitting scheme employed. Here this scheme is a "bounding box" which makes digonal pixels look a bit larger (+40%) than the ones on the horizontal and vertical axis, explaining the variation of the noise.

Integration of a bunch of files using pyFAI

Once the processing for one file is established, one can loop over a bunch of files. A convienient way to get the list of files matching a pattern is with the *glob* module.

Most of the time, the azimuthal integrator is obtained by simply loading the *poni-file* into pyFAI and use it directly.

```
import glob
all_files = glob.glob("al2o*.edf.bz2")
all_files.sort()
print(len(all_files))
```

```
51
```

```
ai = pyFAI.load("al2o3_00_max_51_frames.poni")
print(ai)
```

```
Detector Detector    Spline= /users/kieffer/workspace-400/pyFAI/doc/source/usage/tutorial/Introduc
Wavelength= 7.084811e-11m
SampleDetDist= 1.168599e-01m        PONI= 5.295653e-02, 5.473342e-02m       rot1=0.015821  rot2= 
DirectBeamDist= 116.880mm   Center: x=515.795, y=522.995 pix        Tilt=1.055 deg  tiltPlanRotati
```

```
%%time
for one_file in all_files:
    destination = os.path.splitext(one_file)[0]+".dat"
    image = fabio.open(one_file).data
    ai.integrate1d(image, 1000, filename=destination)
```

```
CPU times: user 35.6 s, sys: 272 ms, total: 35.8 s
Wall time: 22 s
```

This was a simple integration of 50 files, saving the result into 2 column ASCII files.

3.1.5 Conclusion

Using the notebook is rather simple as it allows to mix comments, code, and images for visualization of scientific data.

The basic use pyFAI's AzimuthalIntgrator has also been presented and may be adapted to you specific needs.

3.2 Geometries in pyFAI

This notebook demonstrates the different orientations of axes in the geometry used by pyFAI.

3.2.1 Demonstration

The tutorial uses the ipython notebook (aka Jupyter).

```
%pylab inline
```

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

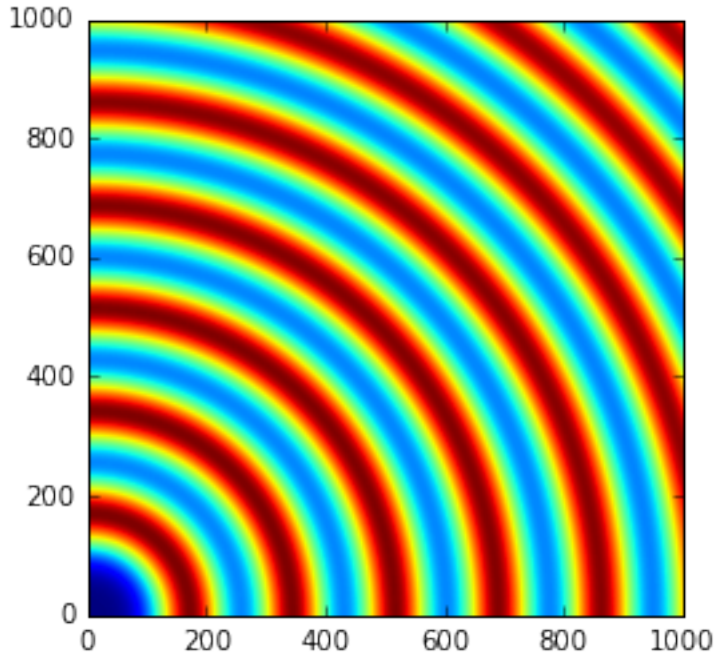
```
import pyFAI
from pyFAI.calibrant import ALL_CALIBRANTS
```

```
WARNING:pyFAI.utils:Exception No module named 'fftw3': FFTw3 not available. Falling back on Scipy
```

We will use a fake detector of 1000x1000 pixels of 100_µm each. The simulated beam has a wavelength of 0.1_nm and the calibrant chose is silver behenate which gives regularly spaced rings. The detector will originally be placed at 1_m from the sample.

```
wl = 1e-10
cal = ALL_CALIBRANTS["AgBh"]
cal.wavelength=wl

detector = pyFAI.detectors.Detector(100e-6, 100e-6)
detector.max_shape=(1000,1000)

ai = pyFAI.AzimuthalIntegrator(dist=1, detector=detector)
ai.wavelength = wl

img = cal.fake_calibration_image(ai)
imshow(img, origin="lower")
```

```
<matplotlib.image.AxesImage at 0x7f997becf7b8>
```



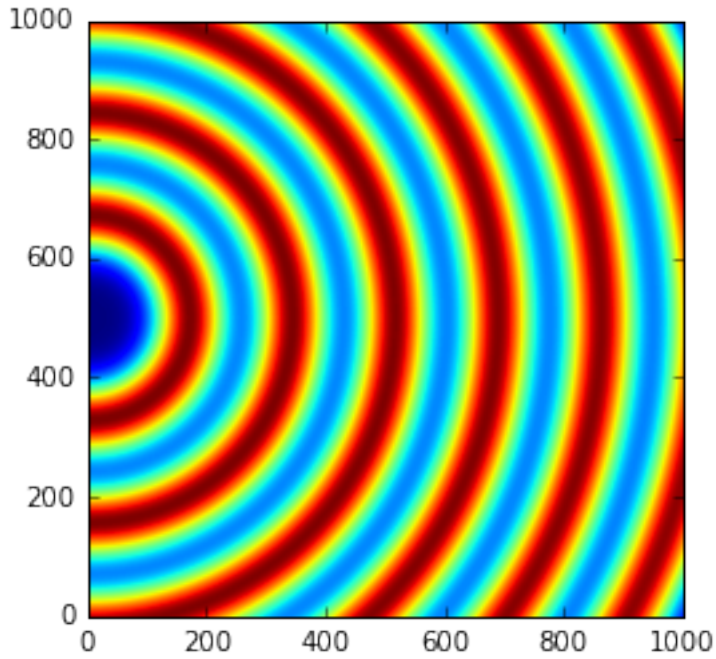
3.2.2 Translation orthogonal to the beam: poni1 and poni2

We will now set the first dimension (vertical) offset to the center of the detector: 100e-6 * 1000 / 2

```
p1 = 100e-6 * 1000 / 2
print(p1)
ai.poni1 = p1
img = cal.fake_calibration_image(ai)
imshow(img, origin="lower")
```

```
0.05
```

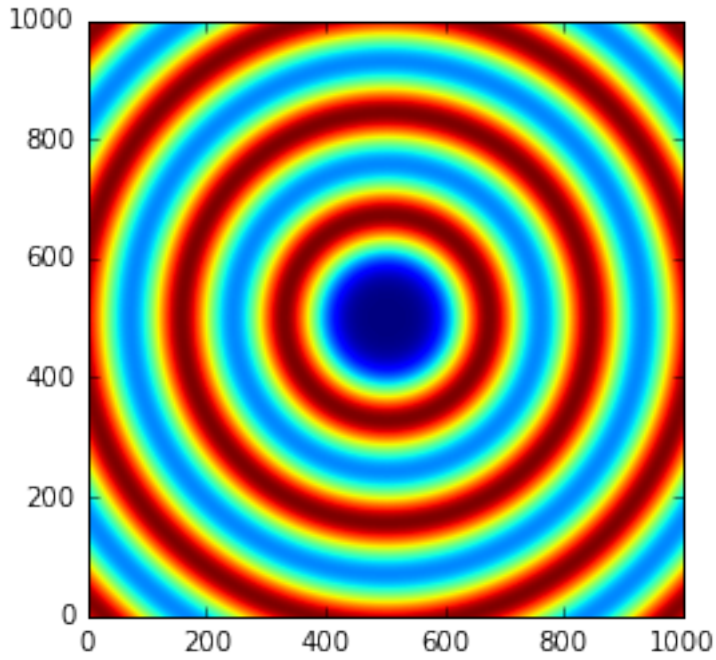
```
<matplotlib.image.AxesImage at 0x7f997ae09be0>
```



Let's do the same in the second dimensions: along the horizontal axis

```
p2 = 100e-6 * 1000 / 2
print(p2)
ai.poni2 = p2
print(ai)
img = cal.fake_calibration_image(ai)
imshow(img, origin="lower")
```

```
0.05
Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e+00m        PONI= 5.000000e-02, 5.000000e-02m       rot1=0.000000  rot2= 0
DirectBeamDist= 1000.000mm  Center: x=500.000, y=500.000 pix        Tilt=0.000 deg  tiltPlanRotati

<matplotlib.image.AxesImage at 0x7f997ade7a20>
```



The image is now properly centered. We will now investigate the rotation along the different axes.

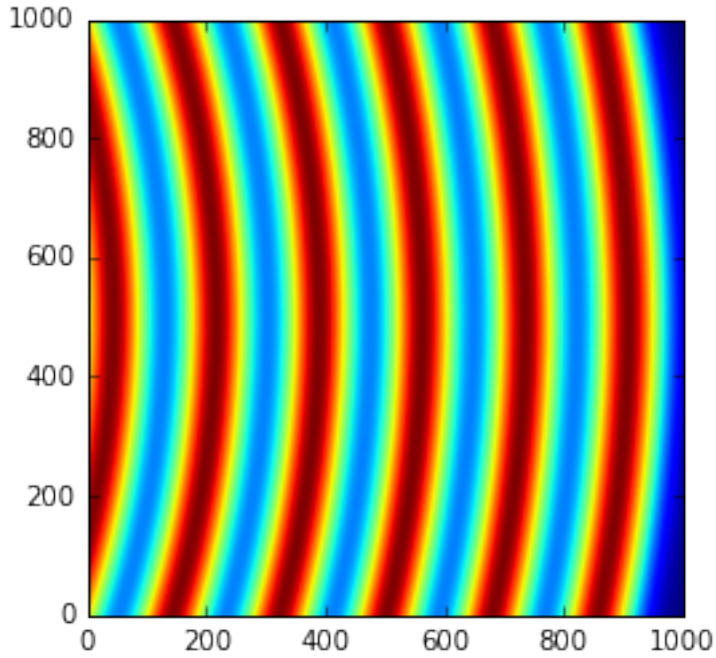
3.2.3 Investigation on the rotations:

Any rotations of the detector apply after the 3 translations (*dist*, *poni1* and *poni2*)

The first axis is the vertical one and a rotation around it ellongates ellipses along the orthogonal axis:

```
rotation = +0.2
ai.rot1 = rotation
print(ai)
img = cal.fake_calibration_image(ai)
imshow(img, origin="lower")
```

```
Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e+00m        PONI= 5.000000e-02, 5.000000e-02m       rot1=0.200000  rot2= 0
DirectBeamDist= 1020.339mm  Center: x=-1527.100, y=500.000 pix      Tilt=11.459 deg  tiltPlanRotat

<matplotlib.image.AxesImage at 0x7f997ad42ef0>
```

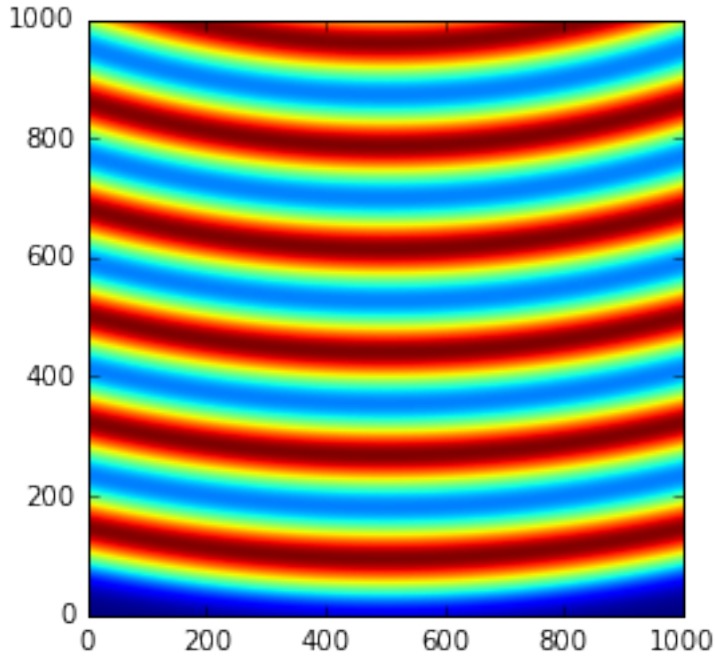


So a positive rot1 is equivalent to turning the detector to the right, around the sample position (where the observer is).

Let's consider now the rotation along the horizontal axis, rot2:

```
rotation = +0.2
ai.rot1 = 0
ai.rot2 = rotation
print(ai)
img = cal.fake_calibration_image(ai)
imshow(img, origin="lower")
```

```
Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e+00m        PONI= 5.000000e-02, 5.000000e-02m       rot1=0.000000  rot2= 
DirectBeamDist= 1020.339mm  Center: x=500.000, y=2527.100 pix       Tilt=11.459 deg  tiltPlanRotat

<matplotlib.image.AxesImage at 0x7f997ad26710>
```

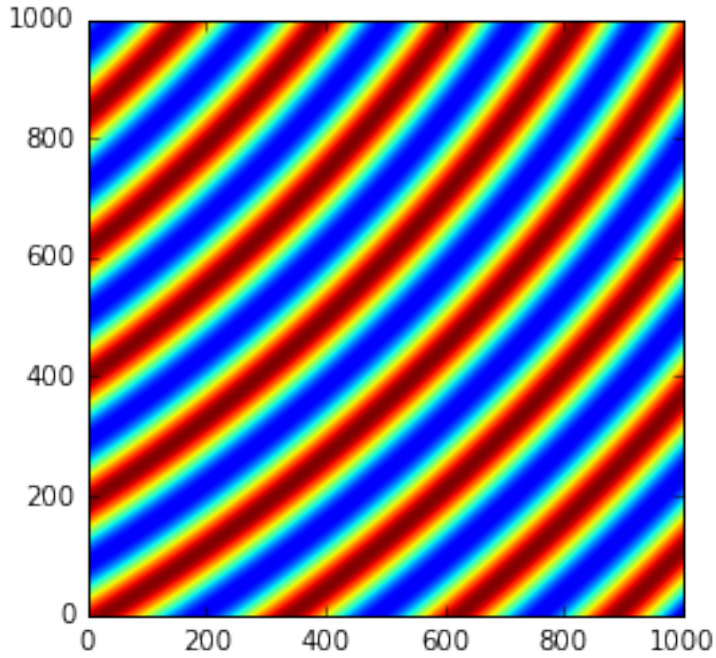


So a positive rot2 is equivalent to turning the detector to the down, around the sample position (where the observer is).

Now we can combine the two first rotations and check for the effect of the third rotation.

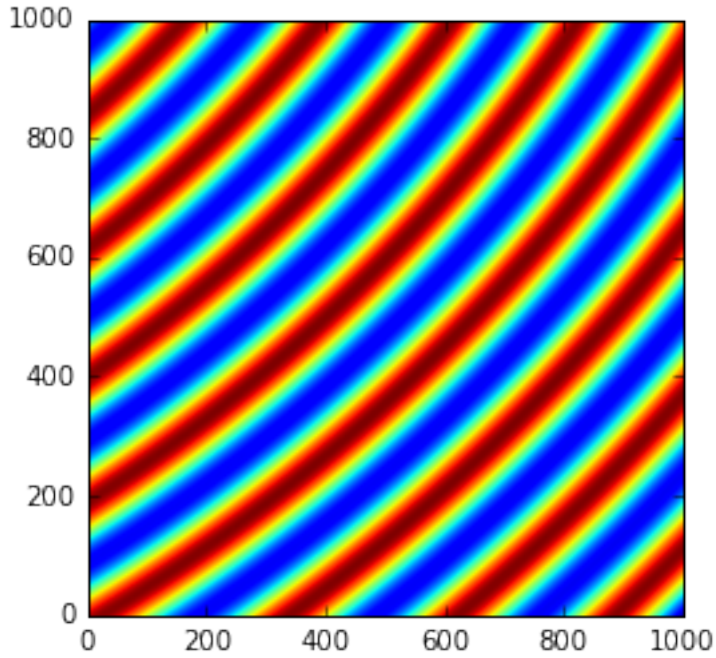
```
rotation = +0.2
ai.rot1 = rotation
ai.rot2 = rotation
ai.rot3 = 0
print(ai)
img = cal.fake_calibration_image(ai)
imshow(img, origin="lower")
```

```
Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e+00m        PONI= 5.000000e-02, 5.000000e-02m       rot1=0.200000  rot2= 
DirectBeamDist= 1041.091mm  Center: x=-1527.100, y=2568.329 pix     Tilt=16.151 deg  tiltPlanRotat

<matplotlib.image.AxesImage at 0x7f997ac835c0>
```




```
rotation = +0.2
import copy
ai2 = copy.copy(ai)
ai2.rot1 = rotation
ai2.rot2 = rotation
ai2.rot3 = rotation
print(ai2)
img2 = cal.fake_calibration_image(ai2)
imshow(img2, origin="lower")
```

```
Detector Detector    Spline= None    PixelSize= 1.000e-04, 1.000e-04 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e+00m        PONI= 5.000000e-02, 5.000000e-02m       rot1=0.200000  rot2= 
DirectBeamDist= 1041.091mm  Center: x=-1527.100, y=2568.329 pix     Tilt=16.151 deg  tiltPlanRotat

<matplotlib.image.AxesImage at 0x7f997ac63f60>
```



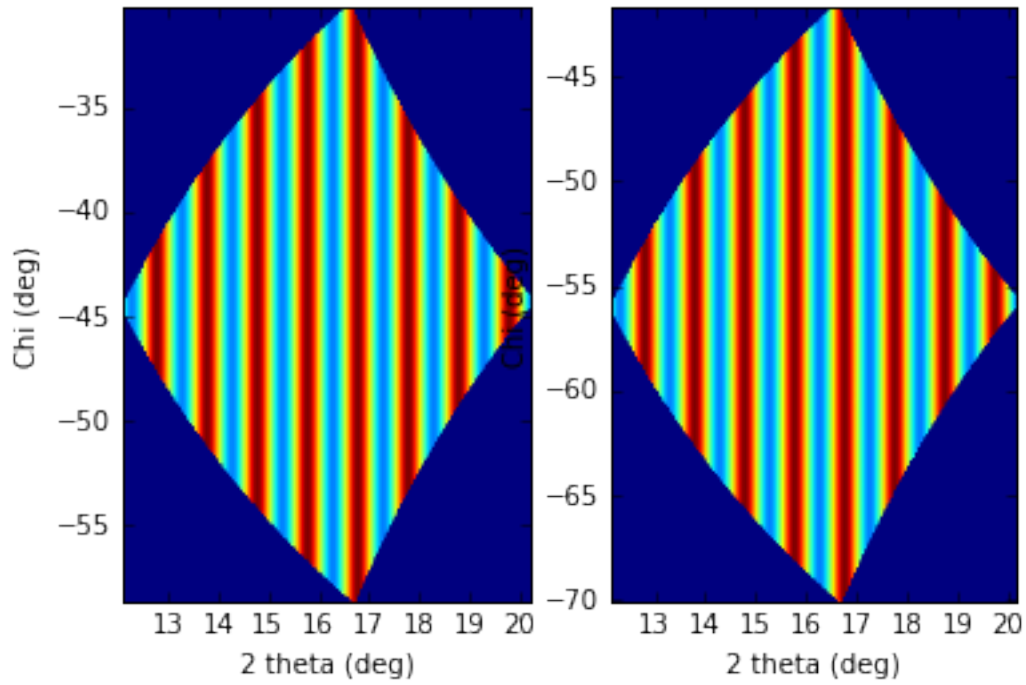
If one considers the rotation along the incident beam, there is no visible effect on the image as the image is invariant along this transformation.

To actually see the effect of this third rotation one needs to perform the azimuthal integration and display the result with properly labeled axes.

```
subplot(1,2,1)
I, tth, chi = ai.integrate2d(img, 300, 360, unit="2th_deg")
imshow(I, origin="lower", extent=[tth.min(), tth.max(), chi.min(), chi.max()], aspect="auto")
xlabel("2 theta (deg)")
ylabel("Chi (deg)")
subplot(1,2,2)
I, tth, chi = ai2.integrate2d(img2, 300, 360, unit="2th_deg")
imshow(I, origin="lower", extent=[tth.min(), tth.max(), chi.min(), chi.max()], aspect="auto")
xlabel("2 theta (deg)")
ylabel("Chi (deg)")
```

```
WARNING:pyFAI.geometry:No fast path for space: None
WARNING:pyFAI.geometry:No fast path for space: None
```

```
<matplotlib.text.Text at 0x7f997abd99e8>
```



So the increasing *rot3* creates more negative azimuthal angles: it is like rotating the detector clockwise around the incident beam.

3.2.4 Conclusion

All 3 translations and all 3 rotations can be summarized in the following figure:

It may appear strange to have (x_1, x_2, x_3) indirect but this has been made in such a way chi, the azimuthal angle, is 0 along x_2 and 90_deg along x_1 (and not vice-versa).

3.3 Detector distortion corrections

This tutorial shows how to correct images for spatial distortion. Some tutorial examples rely on files available in http://www.silx.org/pub/pyFAI/testimages/ and will be downloaded during this tutorial. The requiered minimum version of pyFAI is 0.12.0 (currently dev5)

3.3.1 Detector definitions

PyFAI features an impressive list of 55 detector definitions contributed often by manufacturers and some other reverse engineerd by scientists. Each of them is defined as an invividual class which contains a way to calculate the mask (invalid pixels, gaps,...) and a method to calculate the pixel positions in Cartesian coordinates.

```
import pyFAI
all_detectors = list(set(pyFAI.detectors.ALL_DETECTORS.values()))
#Sort detectors according to their name
all_detectors.sort(key=lambda i:i.__name__)
nb_det = len(all_detectors)
print("Number of detectors registered:", nb_det)
for i in all_detectors:
    print(i())
```

```
WARNING:xsdimage:lxml library is probably not part of your python installation: disabling xsdimage
WARNING:pyFAI.utils:Exception No module named 'fftw3': FFTw3 not available. Falling back on Scipy
```

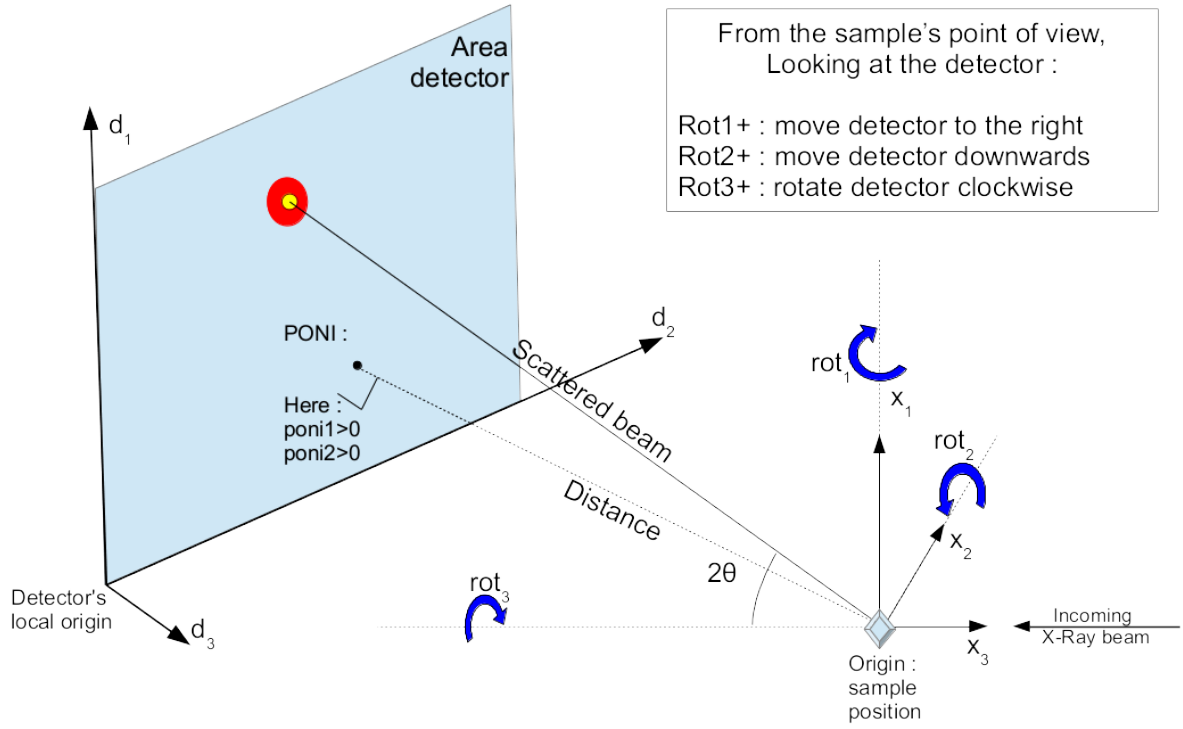


Figure 3.1: PONI figure

```
Number of detectors registered: 55
Detector Quantum 210         Spline= None    PixelSize= 5.100e-05, 5.100e-05 m
Detector Quantum 270         Spline= None    PixelSize= 6.480e-05, 6.480e-05 m
Detector Quantum 315         Spline= None    PixelSize= 5.100e-05, 5.100e-05 m
Detector Quantum 4   Spline= None    PixelSize= 8.200e-05, 8.200e-05 m
Detector Aarhus      Spline= None    PixelSize= 2.500e-05, 2.500e-05 m
Detector ApexII      PixelSize= 6.000e-05, 6.000e-05 m
Detector aca1300     PixelSize= 3.750e-06, 3.750e-06 m
Undefined detector
Detector Dexela 2923         PixelSize= 7.500e-05, 7.500e-05 m
Detector Eiger16M    PixelSize= 7.500e-05, 7.500e-05 m
Detector Eiger1M     PixelSize= 7.500e-05, 7.500e-05 m
Detector Eiger4M     PixelSize= 7.500e-05, 7.500e-05 m
Detector Eiger9M     PixelSize= 7.500e-05, 7.500e-05 m
Detector Fairchild   PixelSize= 1.500e-05, 1.500e-05 m
Detector HF-130k     Spline= None    PixelSize= 1.500e-04, 1.500e-04 m
Detector HF-1M       Spline= None    PixelSize= 1.500e-04, 1.500e-04 m
Detector HF-262k     Spline= None    PixelSize= 1.500e-04, 1.500e-04 m
Detector HF-2.4M     Spline= None    PixelSize= 1.500e-04, 1.500e-04 m
Detector HF-4M       Spline= None    PixelSize= 1.500e-04, 1.500e-04 m
Detector HF-9.4M     Spline= None    PixelSize= 1.500e-04, 1.500e-04 m
Detector Imxpad S10  PixelSize= 1.300e-04, 1.300e-04 m
Detector Imxpad S140         PixelSize= 1.300e-04, 1.300e-04 m
Detector Imxpad S70  PixelSize= 1.300e-04, 1.300e-04 m
Detector MAR 345     PixelSize= 1.000e-04, 1.000e-04 m
Detector Pixium 4700 detector        PixelSize= 1.540e-04, 1.540e-04 m
Detector Perkin detector     PixelSize= 2.000e-04, 2.000e-04 m
Detector Pilatus100k         PixelSize= 1.720e-04, 1.720e-04 m
Detector Pilatus1M   PixelSize= 1.720e-04, 1.720e-04 m
Detector Pilatus200k         PixelSize= 1.720e-04, 1.720e-04 m
Detector Pilatus2M   PixelSize= 1.720e-04, 1.720e-04 m
Detector Pilatus300k         PixelSize= 1.720e-04, 1.720e-04 m
Detector Pilatus300kw        PixelSize= 1.720e-04, 1.720e-04 m
Detector Pilatus6M   PixelSize= 1.720e-04, 1.720e-04 m
Detector PilatusCdTe1M       PixelSize= 1.720e-04, 1.720e-04 m
Detector PilatusCdTe2M       PixelSize= 1.720e-04, 1.720e-04 m
Detector PilatusCdTe300k     PixelSize= 1.720e-04, 1.720e-04 m
Detector PilatusCdTe300kw    PixelSize= 1.720e-04, 1.720e-04 m
Detector Rayonix     PixelSize= 3.200e-05, 3.200e-05 m
Detector MAR133      PixelSize= 6.400e-05, 6.400e-05 m
Detector Rayonix lx170       PixelSize= 4.427e-05, 4.427e-05 m
Detector Rayonix lx255       PixelSize= 4.427e-05, 4.427e-05 m
Detector Rayonix mx170       PixelSize= 4.427e-05, 4.427e-05 m
Detector Rayonix mx225       PixelSize= 7.324e-05, 7.324e-05 m
Detector Rayonix mx225hs     PixelSize= 7.813e-05, 7.813e-05 m
Detector Rayonix mx300       PixelSize= 7.324e-05, 7.324e-05 m
Detector Rayonix mx300hs     PixelSize= 7.813e-05, 7.813e-05 m
Detector Rayonix mx325       PixelSize= 7.935e-05, 7.935e-05 m
Detector Rayonix mx340hs     PixelSize= 8.854e-05, 8.854e-05 m
Detector Rayonix mx425hs     PixelSize= 4.427e-05, 4.427e-05 m
Detector MAR165      PixelSize= 3.950e-05, 3.950e-05 m
Detector Rayonix sx200       PixelSize= 4.800e-05, 4.800e-05 m
Detector Rayonix Sx30hs      PixelSize= 1.563e-05, 1.563e-05 m
Detector Rayonix Sx85hs      PixelSize= 4.427e-05, 4.427e-05 m
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Detector Xpad S540 flat      PixelSize= 1.300e-04, 1.300e-04 m
```

3.3.2 Defining a detector from a spline file

For optically coupled CCD detectors, the geometrical distortion is often described by a two-dimensional cubic spline (as in FIT2D) which can be imported into the relevant detector instance and used to calculate the actual

pixel position in space (and masked pixels).

At the ESRF, mainly FReLoN detectors [J.-C. Labiche, ESRF Newsletter 25, 41 (1996)] are used with spline files describing the distortion of the fiber optic taper.

Let's download such a file and create a detector from it.

```
import os, sys
os.environ["http_proxy"] = "http://proxy.site.com:3128"
def download(url):
    """download the file given in URL and return its local path"""
    if sys.version_info[0]<3:
        from urllib2 import urlopen, ProxyHandler, build_opener
    else:
        from urllib.request import urlopen, ProxyHandler, build_opener
    dictProxies = {}
    if "http_proxy" in os.environ:
        dictProxies['http'] = os.environ["http_proxy"]
        dictProxies['https'] = os.environ["http_proxy"]
    if "https_proxy" in os.environ:
        dictProxies['https'] = os.environ["https_proxy"]
    if dictProxies:
        proxy_handler = ProxyHandler(dictProxies)
        opener = build_opener(proxy_handler).open
    else:
        opener = urlopen
    target = os.path.split(url)[-1]
    with open(target,"wb") as dest, opener(url) as src:
        dest.write(src.read())
    return target

spline_file = download("http://www.silx.org/pub/pyFAI/testimages/halfccd.spline")

hd = pyFAI.detectors.FReLoN(splineFile=spline_file)
print(hd)
print("Shape: %i, %i"% hd.shape)
```

```
Detector FReLoN      Spline= /users/kieffer/workspace-400/pyFAI/doc/source/usage/tutorial/Distort
Shape: 1025, 2048
```

Note the unusual shape of this detector. This is probably a human error when calibrating the detector distortion in FIT2D.

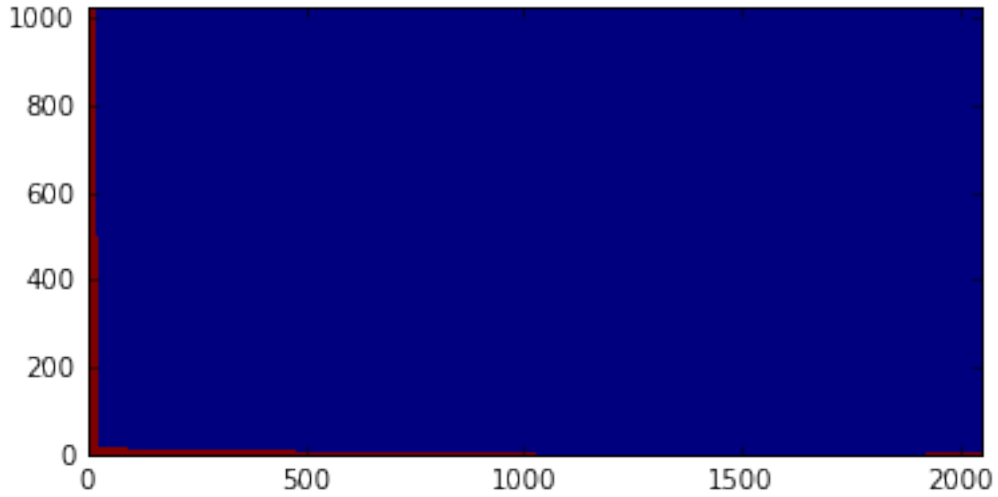
Visualizing the mask

Every detector object contains a mask attribute, defining pixels which are invalid. For FReLoN detector (a spline-files-defined detectors), all pixels having an offset such that the pixel falls out of the initial detector are considered as invalid.

Masked pixel have non-null values can be displayed like this:

```
%pylab inline
imshow(hd.mask, origin="lower", interpolation="nearest")
```

```
Populating the interactive namespace from numpy and matplotlib

<matplotlib.image.AxesImage at 0x7f2304a8cd68>
```



3.3.3 Detector definition files as NeXus files

Any detector object in pyFAI can be saved into an HDF5 file following the NeXus convention [Könnecke et al., 2015, J. Appl. Cryst. 48, 301-305.]. Detector objects can subsequently be restored from disk, making complex detector definitions less error prone.

```
h5_file = "halfccd.h5"
hd.save(h5_file)
new_det = pyFAI.detector_factory(h5_file)
print(new_det)
print("Mask is the same: ", numpy.allclose(new_det.mask, hd.mask))
print("Pixel positions are the same: ", numpy.allclose(new_det.get_pixel_corners(), hd.get_pixel_
print("Number of masked pixels", new_det.mask.sum())
```

```
FReLoN detector from NeXus file: halfccd.h5  PixelSize= 4.842e-05, 4.684e-05 m
Mask is the same:  True
Pixel positions are the same:  True
Number of masked pixels 34382
```

Pixels of an area detector are saved as a four-dimensional dataset: i.e. a two-dimensional array of vertices pointing to every corner of each pixel, generating an array of dimension (Ny, Nx, Nc, 3), where Nx and Ny are the dimensions of the detector, Nc is the number of corners of each pixel, usually four, and the last entry contains the coordinates of the vertex itself (in the order: Z, Y, X).

This kind of definition, while relying on large description files, can address some of the most complex detector layouts. They will be presented a bit later in this tutorial.

```
print("Size of Spline-file:", os.stat('halfccd.spline').st_size)
print("Size of Nexus-file:", os.stat('halfccd.h5').st_size)
```

```
Size of Spline-file: 1183
Size of Nexus-file: 21451707
```

The HDF5 file is indeed much larger than the spline file.

3.3.4 Modify a detector and saving

One may want to define a new mask (or flat-field) for its detector and save the mask with the detector definition. Here, we create a copy of the detector and reset its mask to enable all pixels in the detector and save the new detector instance into another file.

```
import copy
nomask_file = "nomask.h5"
nomask = copy.deepcopy(new_det)
nomask.mask = numpy.zeros_like(new_det.mask)
nomask.save(nomask_file)
nomask = pyFAI.detector_factory("nomask.h5")
print("No pixels are masked",nomask.mask.sum())
```

```
No pixels are masked 0
```

Wrap up

In this section we have seen how detectors are defined in pyFAI, how they can be created, either from the list of the parametrized ones, or from spline files, or from NeXus detector files. We have also seen how to save and subsequently restore a detector instance, preserving the modifications made.

3.3.5 Distortion correction

Once the position of every single pixel in space is known, one can benefit from the regridding engine of pyFAI adapted to image distortion correction tasks. The *pyFAI.distortion.Distortion* class is the equivalent of the *pyFAI.AzimuthalIntegrator* for distortion. Provided with a detector definition, it enables the correction of a set of images by using the same kind of look-up tables as for azimuthal integration.

```
from pyFAI.distortion import Distortion
dis = Distortion(nomask)
print(dis)
```

```
Distortion correction lut on device None for detector shape (1025, 2048):
NexusDetector detector from NeXus file: nomask.h5    PixelSize= 4.842e-05, 4.684e-05 m
```

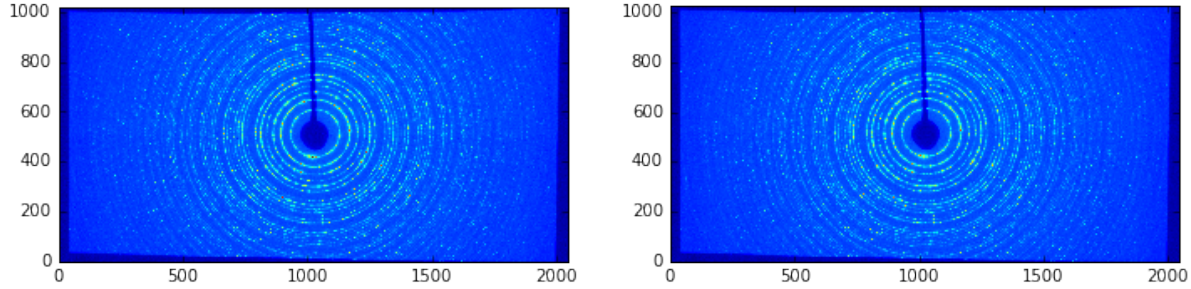
FReLoN detector

First load the image to be corrected, then correct it for geometric distortion.

```
halfccd_img = download("http://www.silx.org/pub/pyFAI/testimages/halfccd.edf")
import fabio
raw = fabio.open(halfccd_img).data
cor = dis.correct(raw)

#Then display images side by side
numpy.seterr(divide="ignore") #remove warning messages from numpy
figure(figsize=(12,6))
subplot(1,2,1)
imshow(numpy.log(raw), interpolation="nearest", origin="lower")
subplot(1,2,2)
imshow(numpy.log(cor), interpolation="nearest", origin="lower")
```

```
ERROR:pyFAI.distortion:The image shape ((1024, 2048)) is not the same as the detector ((1025, 204
```

```
<matplotlib.image.AxesImage at 0x7f22fb6d48d0>
```



Nota: in this case the image size (1024 lines) does not match the detector's number of lines (1025) hence pyFAI complains about it. Here, pyFAI patched the image on an empty image of the right size so that the processing can occur.

In this example, the size of the pixels and the shape of the detector are preserved, discarding all pixels falling outside the detector's grid.

One may want all pixels' intensity to be preserved in the transformation. By allowing the output array to be large enough to accomodate all pixels, the total intensity can be kept. For this, just enable the "resize" option in the constructor of *Distortion*:

```
dis1 = Distortion(hd, resize=True)
print(dis1)
cor = dis1.correct(raw)
print(dis1)
print("After correction, the image has a different shape", cor.shape)
```

```
ERROR:pyFAI.distortion:The image shape ((1024, 2048)) is not the same as the detector ((1025, 204

Distortion correction lut on device None for detector shape None:
Detector FReLoN      Spline= /users/kieffer/workspace-400/pyFAI/doc/source/usage/tutorial/Distorti
Distortion correction lut on device None for detector shape (1045, 2052):
Detector FReLoN      Spline= /users/kieffer/workspace-400/pyFAI/doc/source/usage/tutorial/Distorti
After correction, the image has a different shape (1045, 2052)
```

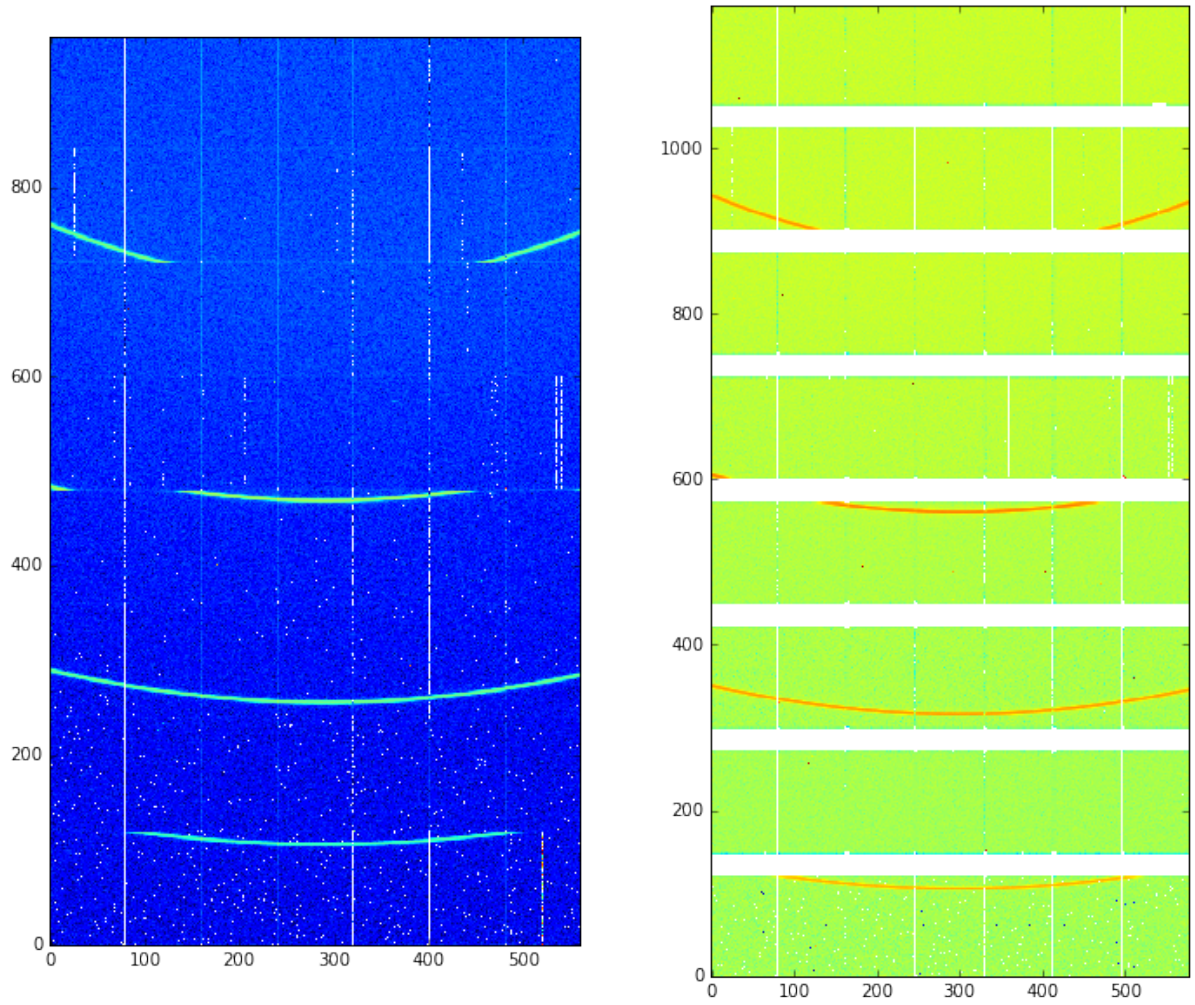
Example of Pixel-detectors:

XPad Flat detector

There is a striking example in the cover image of this article: http://scripts.iucr.org/cgi-bin/paper?S1600576715004306 where a detector made of multiple modules is *eating up* some rings. The first example will be about the regeneration of an "eyes friendly" version of this image.

```
xpad_url = "http://www.silx.org/pub/pyFAI/testimages/LaB6_18.57keV_frame_13.edf"
xpad_file = download(xpad_url)
xpad = pyFAI.detector_factory("Xpad_flat")
print(xpad)
xpad_dis = Distortion(xpad, resize=True)

raw = fabio.open(xpad_file).data
cor = xpad_dis.correct(raw)
print("Shape as input and output:", raw.shape, cor.shape)

#then display images side by side
figure(figsize=(12,10))
subplot(1,2,1)
imshow(numpy.log(raw), interpolation="nearest", origin="lower")
subplot(1,2,2)
imshow(numpy.log(cor), interpolation="nearest", origin="lower")

print("Conservation of the total intensity:", raw.sum(), cor.sum())
```

```
Detector Xpad S540 flat      PixelSize= 1.300e-04, 1.300e-04 m
Shape as input and output: (960, 560) (1174, 578)
Conservation of the total intensity: 11120798 1.11208e+07
```



WOS XPad detector

This is a new **WAXS opened for SAXS** pixel detector from ImXPad (available at ESRF-BM02/D2AM CRG beamline). It looks like two of *XPad_flat* detectors side by side with some modules shifted in order to create a hole to accomodate a flight-tube which gathers the SAXS photons to a second detector further away.

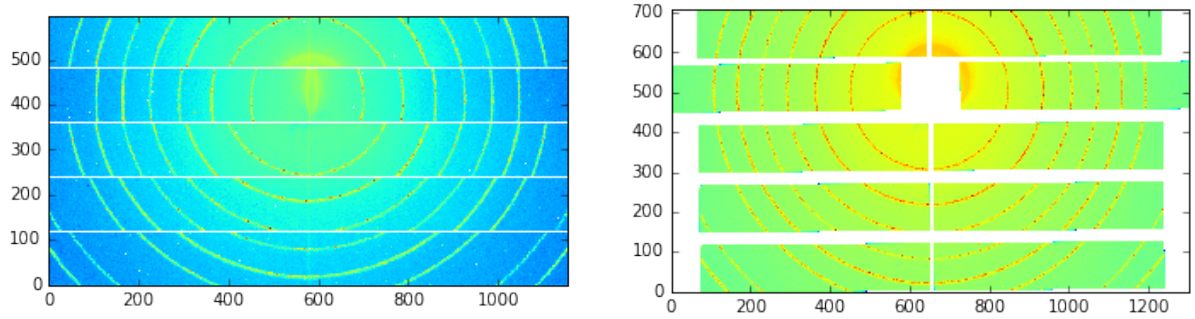
The detector definition for this specific detector has directly been put down using the metrology informations from the manufacturer and saved as a NeXus detector definition file.

```
wos_det = download("http://www.silx.org/pub/pyFAI/testimages/WOS.h5")
wos_img = download("http://www.silx.org/pub/pyFAI/testimages/WOS.edf")
wos = pyFAI.detector_factory(wos_det)
print(wos)
wos_dis = Distortion(wos, resize=True)

raw = fabio.open(wos_img).data
cor = wos_dis.correct(raw)
print("Shape as input and output:", raw.shape, cor.shape)

#then display images side by side
figure(figsize=(12,10))
subplot(1,2,1)
imshow(numpy.log(raw), interpolation="nearest", origin="lower")
subplot(1,2,2)
imshow(numpy.log(cor), interpolation="nearest", origin="lower")
```

```
print("Conservation of the total intensity:", raw.sum(), cor.sum())
```

```
NexusDetector detector from NeXus file: WOS.h5       PixelSize= 1.300e-04, 1.300e-04 m
Shape as input and output: (598, 1154) (710, 1302)
Conservation of the total intensity: 444356428 4.44363e+08
```

```
/scisoft/users/jupyter/jupy34/lib/python3.4/site-packages/ipykernel/__main__.py:14: RuntimeWarnin
```



Nota: Do not use this detector definition file to process data from the WOS@D2AM as it has not (yet) been fully validated and may contain some errors in the pixel positioning.

3.3.6 Conclusion

PyFAI provides a very comprehensive list of detector definitions, is versatile enough to address most area detectors on the market, and features a powerful regridding engine, both combined together into the distortion correction tool which ensures the conservation of the signal during the transformation (the number of photons counted is preserved during the transformation)

Distortion correction should not be used for pre-processing images prior to azimuthal integration as it re-bins the image, thus induces a broadening of the peaks. The AzimuthalIntegrator object performs all this together with integration, it has hence a better precision.

This tutorial did not answer the question *how to calibrate the distortion of a given detector ?* which is addressed in another tutorial called **detector calibration**.

3.4 Selection of a calibrant

In this tutorial we will see how to select a calibrant for a given experimental setup.

3.4.1 Experimental setup

The experimental setup is a classical protein crystallography setup with:

- Large Pilatus 6M detector on a translation table
- The small and intense beam of ~50 microns in size has a wavelength of 1 Angstrom
- The detector is in *normal* condition: orthogonal to the beam and centered in the middle of the detector.

The scientist in charge of this beamline want to ensure all encoders are working properly and needs to validate the setup for distances between 10cm and 80cm. He will buy reference material from NIST but he does not know which calibrant is best suited for his setup. We will assume all reference material sold by NIST are equally suitable for ray position (no issue with grain size, ...).

The calibration works best in pyFAI if more than one Debye-Scherrer ring is seen on the detector.

3.4.2 Define the detector

```
import pyFAI
dete = pyFAI.detectors.Pilatus6M()
print(dete)

Detector Pilatus6M   PixelSize= 1.720e-04, 1.720e-04 m
```

3.4.3 Select reference materials

The NIST sells different Standard Refrence Materials, among them Silicon (SRM640), Lanthanum hexaboride (SRM660), Alumina (SRM676) and Ceria (SRM674) are commonly used. Many other exists: Cr203, TiO2, Zn0, SiO2, ... Evaluating them is left as an exercise.

```
import pyFAI.calibrant
print(pyFAI.calibrant.ALL_CALIBRANTS)

Calibrants available: Ni, CrOx, NaCl, Si_SRM640e, Si_SRM640d, Si_SRM640a, Si_SRM640c, alpha_Al2O3,
```

You may wonder where the names of the calibrant came from and how they have been established.

The name of all calibrant available in your version of pyFAI can be listed by just printing out the content of ALL_CALIBRANTS. New calibrant may have been added in pyFAI in more recent releases, just have a look at the developent web page.

Most of those calibrant files, which contain the *d-spacing* in Angstrom between Miller plans, have been prepared from the unit cell of the compound, found in publication. This publication is referenced in the header of the file. If one wishes to regenerate those files, the *pyFAI.calibrant.Cell* class may be used for.

We will now focus on a subset of calibrant, instanciate them and put them into a dictionnary. The Python construct used here is called *dict-comprehension* and allows the creation and population of a dictionnary in a single line.

```
cals = dict((name,pyFAI.calibrant.ALL_CALIBRANTS(name)) for name in ("Si", "LaB6", "CeO2", "alpha_
print(cals)

{'Si': Si Calibrant , 'alpha_Al2O3': alpha_Al2O3 Calibrant , 'LaB6': LaB6 Calibrant , 'CeO2': CeO
```

To be able to use those calibrants, one needs to define the wavelength used, here 1 Angstrom.

```
wl = 1e-10
for cal in cals.values():
    cal.wavelength = wl
print(cals)

{'Si': Si Calibrant at wavelength 1e-10, 'alpha_Al2O3': alpha_Al2O3 Calibrant at wavelength 1e-10,
```

3.4.4 Short distance images

The shortest the detector can come to the sample is about 10cm (to leave space for the beamstop). We will generate images of diffraction at this distance.

For the display of images we will use *matplotlib* inlined.

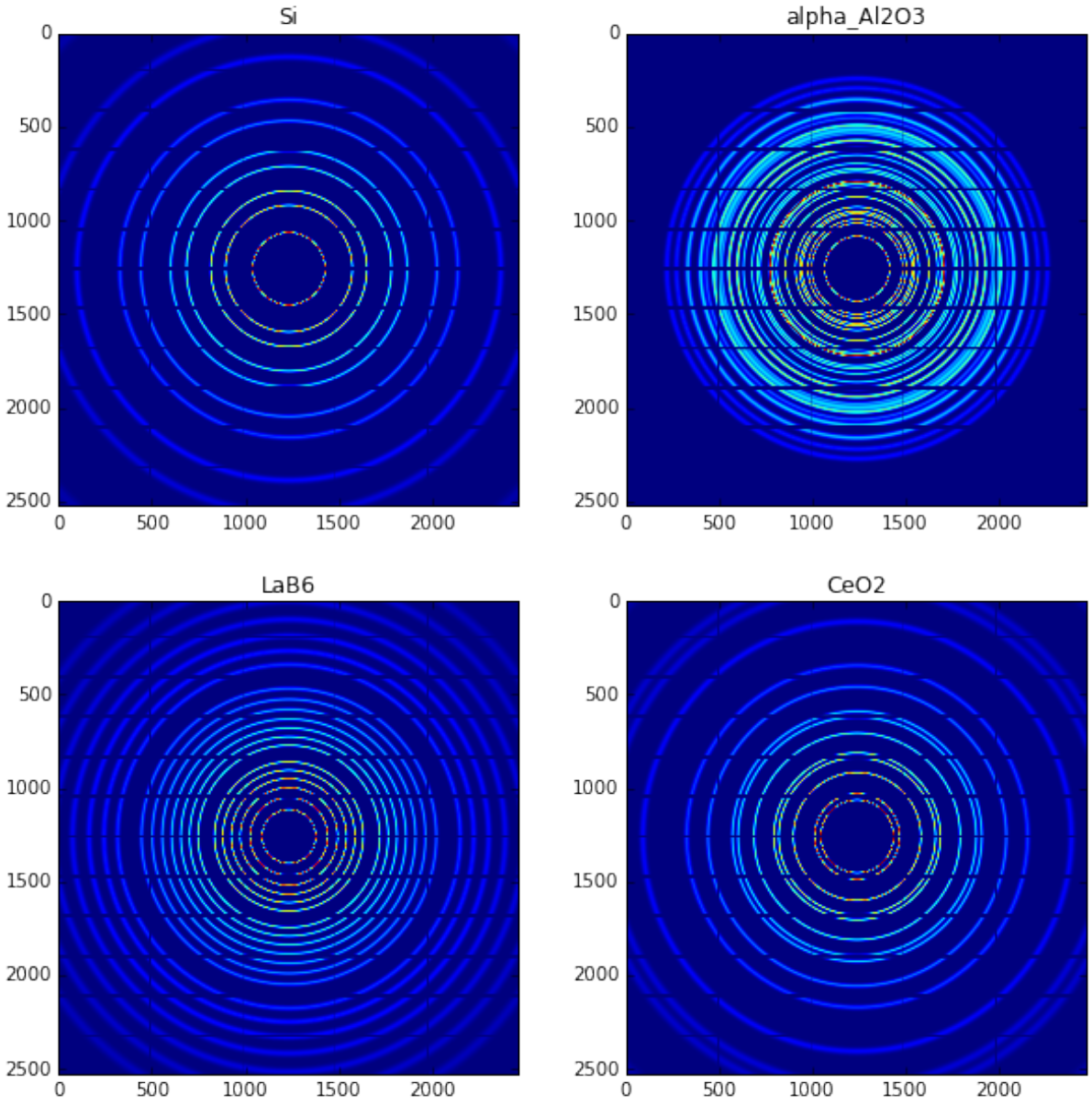
```
p1, p2, p3 = dete.calc_cartesian_positions()
poni1 = p1.mean()
poni2 = p2.mean()
print("Detector center at %s, %s"%(poni1, poni2))
ai_short = pyFAI.AzimuthalIntegrator(dist=0.1, poni1=poni1, poni2=poni2,detector=dete)
print(ai_short)
```

```
Detector center at 0.217322000001, 0.211818
Detector Pilatus6M   PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 1.000000e-01m        PONI= 2.173220e-01, 2.118180e-01m       rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=1231.500, y=1263.500 pix      Tilt=0.000 deg  tiltPlanRotati
```

```
%pylab inline
```

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

```
fig = figure(figsize=(10,10))
for idx, key in enumerate(cals):
    cal = cals[key]
    subplot(2,2,idx+1)
    img = cal.fake_calibration_image(ai_short)
    imshow(img)
    title(key)
```



As one can see, there are plenty of rings on the image: it should be easy to calibrate. By moving further away the detector, the number of rings will decrease.

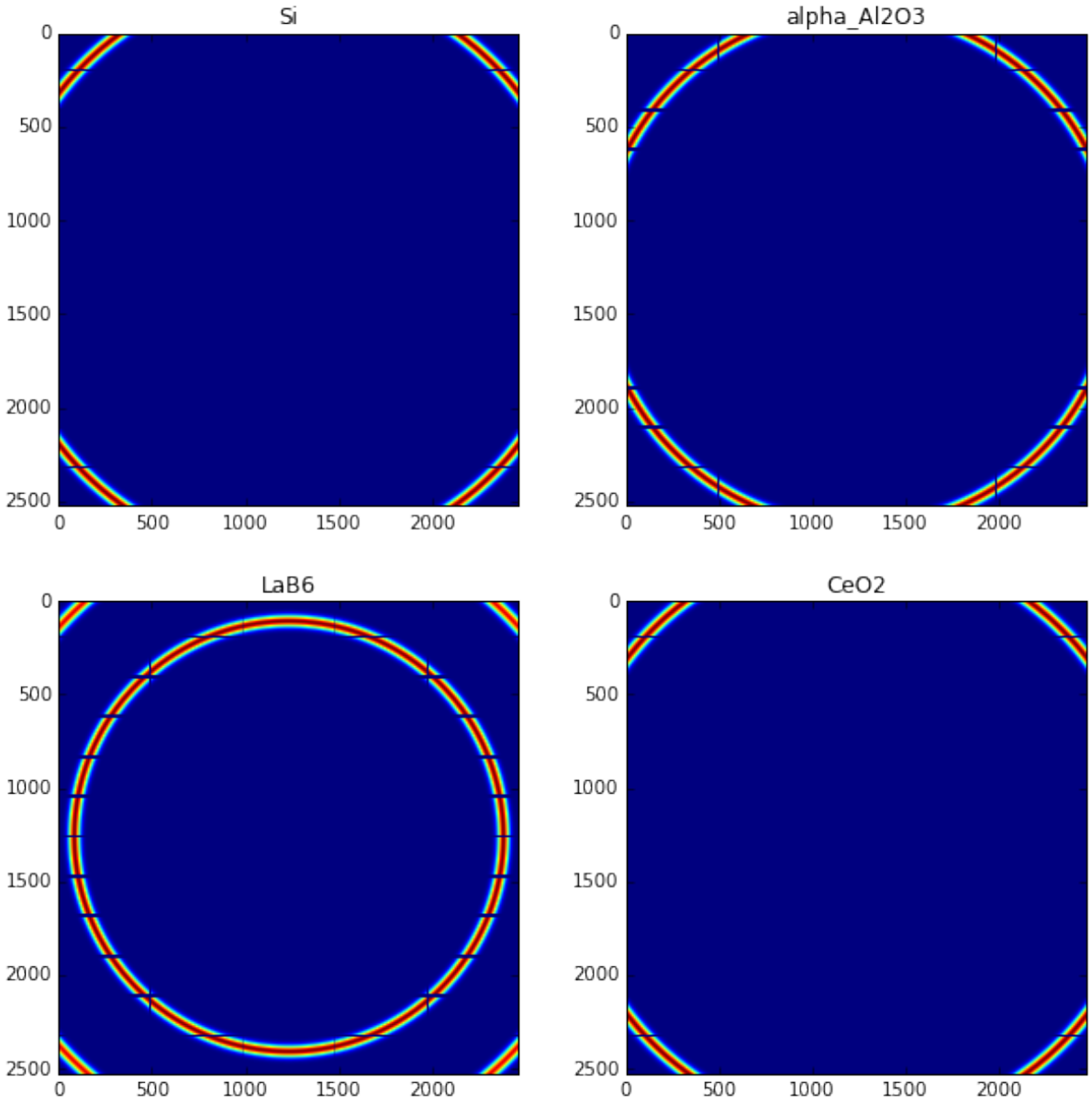
3.4.5 Long distance images

To keep good calibration one should have at lease two rings for the calibration. The longest distance from sample to the detector is 80cm.

```
ai_long = pyFAI.AzimuthalIntegrator(dist=0.8, poni1=poni1, poni2=poni2,detector=dete)
print(ai_long)
```

```
Detector Pilatus6M   PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 8.000000e-01m        PONI= 2.173220e-01, 2.118180e-01m       rot1=0.000000  rot2= 
DirectBeamDist= 800.000mm   Center: x=1231.500, y=1263.500 pix      Tilt=0.000 deg  tiltPlanRotati
```

```
fig = figure(figsize=(10,10))
for idx, key in enumerate(cals):
    cal = cals[key]
    subplot(2,2,idx+1)
    img = cal.fake_calibration_image(ai_long)
    imshow(img)
    title(key)
```



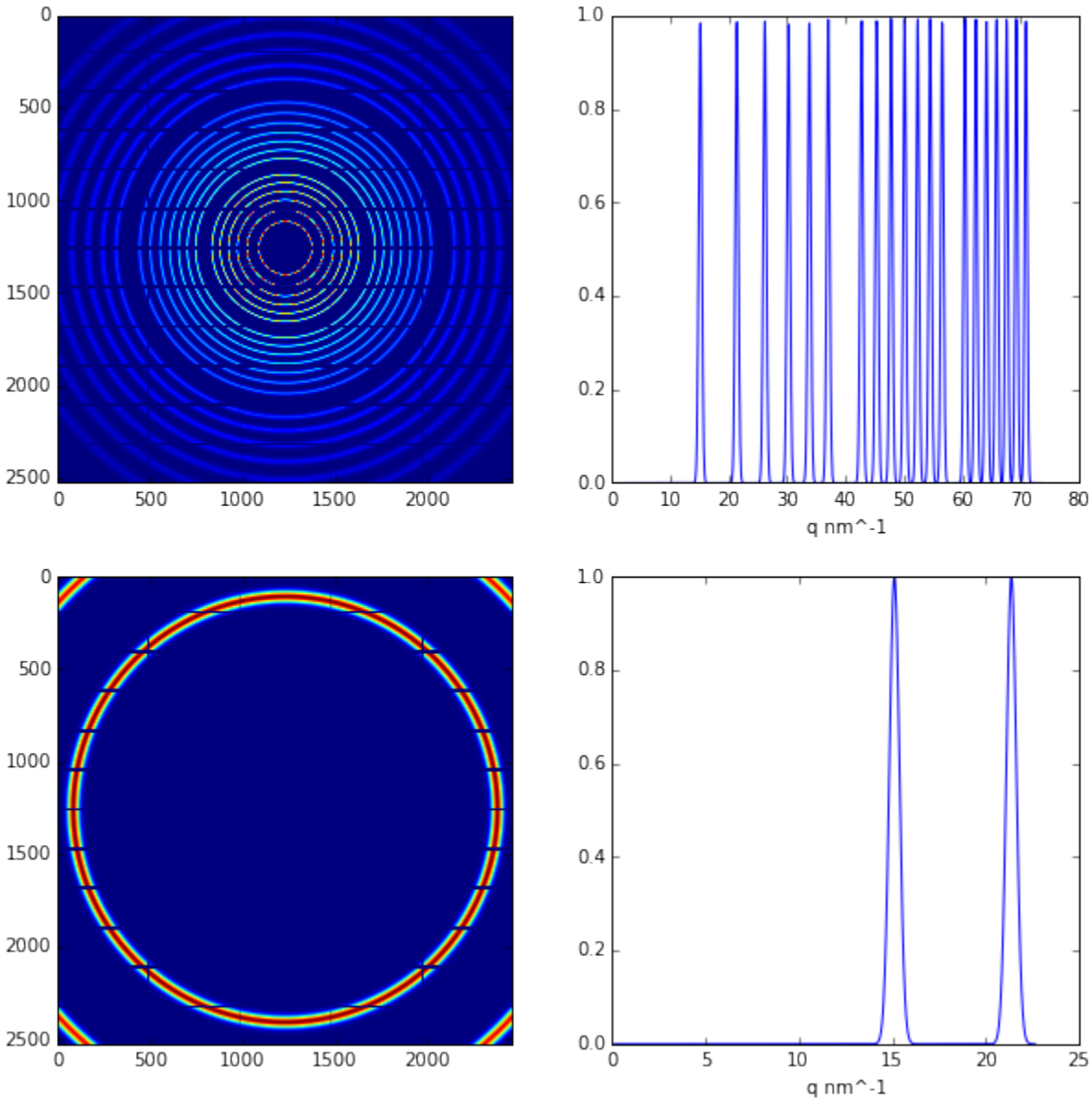
The most adapted calibrant is probably the *LaB6* as 2 rings are still visible at 80 cm from the detector.

3.4.6 Integration of the pattern for the two extreme cases

We can integrate the image for the two extreme cases:

```
lab6 = cals["LaB6"]
ai_short.wavelength = ai_long.wavelength = wl

fig = figure(figsize=(10,10))
ax = subplot(2,2,1)
img_short = lab6.fake_calibration_image(ai_short)
ax.imshow(img_short)
ax = subplot(2,2,2)
ax.plot(*ai_short.integrate1d(img_short,1000))
ax.set_xlabel("q nm^-1")
ax = subplot(2,2,3)
img_long = lab6.fake_calibration_image(ai_long)
ax.imshow(img_long)
ax = subplot(2,2,4)
ax.plot(*ai_long.integrate1d(img_long,1000))
ax.set_xlabel("q nm^-1")
```

```
<matplotlib.text.Text at 0x7f1b7da2de50>
```



3.4.7 Conclusion

The best calibrant in this case is probably LaB6.

3.5 Multi-geometry azimuthal integration

Or how it is possible to perform an azimuthal regrouping using multiple detectors or by moving a single (small) detector to cover more solid angle.

3.5.1 Idea

Azimuthal integration or azimuthal regrouping is (roughly) the averaging of all pixel intensities with the same Q value (or 2theta), as described in this publication chapter 3.2 and 3.3.

By taking multiple images at various places in space one covers more solid angle, allowing either a better statistics or a larger Q-range coverage.

As described in the publication, the average is calculated by the ratio of the (intensity-) weighted histogram by the unweighted histogram. By enforcing a same bin position over multiple geometries, one can create a combined weighted and unweighted histograms by simply summing all partial histograms from each geometry.

The resulting pattern is obtained as usual by the ration of weighted/unweighted

3.5.2 How it works

Lets assume you are able to know where your detector is in space, either calibrated, either calculated from the goniometer position. A diffrection image (img_i) has been acquired using a geometry which is stored in a poni-file (poni_i) useable by pyFAI.

To define a multi-geometry integrator, one needs all poni-files and one needs to define the output space so that all individual integrators use the same bins.

```
import glob
import fabio
from pyFAI.multi_geometry import MultiGeometry

img_files = glob.glob("*.cbf")
img_data = [fabio.open(i).data for i in img_files]
ais = [i[:-4]+".poni" for i in img_files]
mg = MultiGeometry(ais, unit="q_A^-1", radial_range=(0, 50), wavelength=1e-10)
q, I = mg.integrate1d(img_data, 10000)
```

What is automatic

- MultiGeometry takes care of defining the same output space with the same bin position,
- It performs the normalization of the solid angle (absolute solid angle, unlike AzimuthalIntegrator !)
- It can handle polarization correction if needed
- It can normalize by a monitor (I1 normalization) or correct for exposure time

What is not

For PDF measurement, data needs to be properly prepared, especially:

- Dark current subtraction
- Flat-field correction

- Exposure time correction (if all images are not taken with the same exposure time)

3.5.3 Examples

Demo of usage of the MultiGeometry class of pyFAI

For this tutorial, we will use the ipython notebook, now known as *Jypyter*, an take advantage of the integration of matplotlib with the *inline*:

```
%pylab inline
```

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

The multi_geometry module of pyFAI allows you to integrate multiple images taken at various image position, all togeather. This tutorial will explain you how to perform azimuthal integration in three use-case: translation of the detector, rotation of the detector around the sample and finally how to fill gaps of a pixel detector. But before, we need to know how to generate fake diffraction image.

Generation of diffraction images

PyFAI knows about 20 different reference sample called calibrants. We will use them to generate fake diffraction images knowing the detector and its position in space

```
import pyFAI
import pyFAI.calibrant
print("Number of known calibrants: %s"%len(pyFAI.calibrant.ALL_CALIBRANTS))
print(" ".join(pyFAI.calibrant.ALL_CALIBRANTS.keys()))
```

```
Number of known calibrants: 27
Ni CrOx NaCl Si_SRM640e Si_SRM640d Si_SRM640a Si_SRM640c alpha_Al2O3 Cr2O3 AgBh Si_SRM640 CuO PBB
```

```
wl = 1e-10
LaB6 = pyFAI.calibrant.ALL_CALIBRANTS("LaB6")
LaB6.set_wavelength(wl)
print(LaB6)
```

```
LaB6 Calibrant at wavelength 1e-10
```

We will start with a "simple" detector called *Titan* (build by *Oxford Diffraction* but now sold by *Agilent*). It is a CCD detector with scintilator and magnifying optics fibers. The pixel size is constant: 60µm

```
det = pyFAI.detectors.Titan()
print(det)
p1, p2, p3 = det.calc_cartesian_positions()
print("Detector is flat, P3= %s"%p3)
poni1 = p1.mean()
poni2 = p2.mean()
print("Center of the detector: poni1=%s poni2=%s"%(poni1, poni2))
```

```
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Detector is flat, P3= None
Center of the detector: poni1=0.06144 poni2=0.06144
```

The detector is placed orthogonal to the beam at 10cm. This geometry is saved into an *AzimuthalIntegrator* instance:

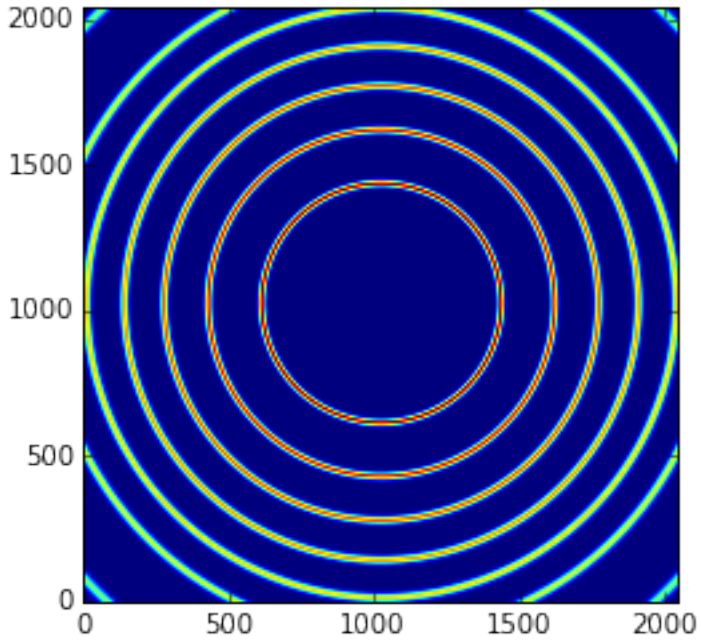
```
ai = pyFAI.AzimuthalIntegrator(dist=0.1, poni1=poni1, poni2=poni2, detector=det)
ai.set_wavelength(wl)
print(ai)
```

```
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e-01m        PONI= 6.144000e-02, 6.144000e-02m       rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=1024.000, y=1024.000 pix      Tilt=0.000 deg  tiltPlanRotati
```

Knowing the calibrant, the wavelength, the detector and the geometry, one can simulate the 2D diffraction pattern:

```
img = LaB6.fake_calibration_image(ai)
imshow(img, origin="lower")
```

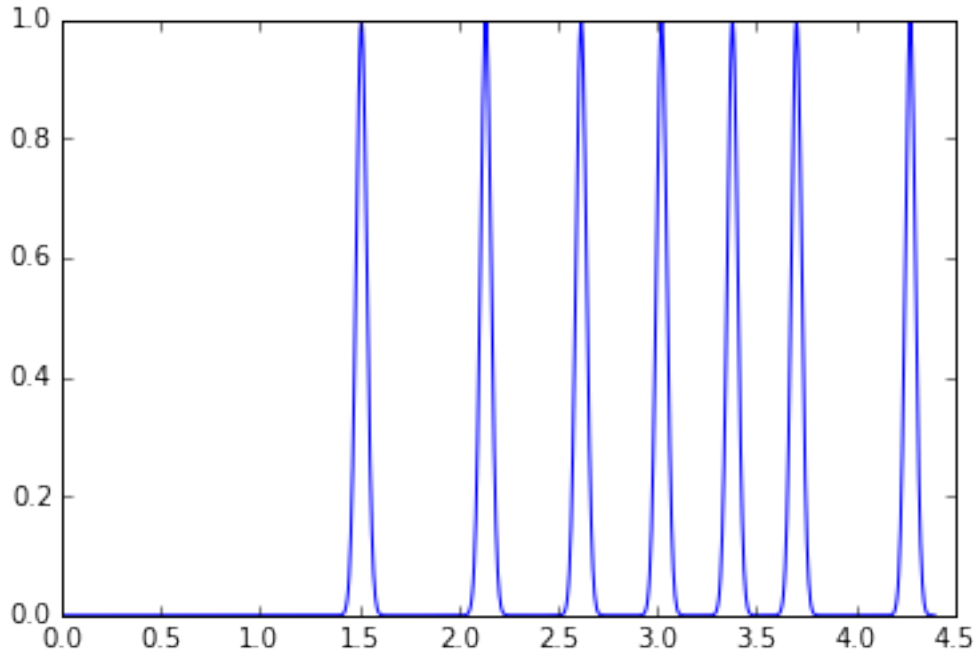
```
<matplotlib.image.AxesImage at 0x7fbf946e68d0>
```



This image can be integrated in q-space and plotted:

```
plot(*ai.integrate1d(img, 1000, unit="q_A^-1"))
```

```
[<matplotlib.lines.Line2D at 0x7fbf92da9d90>]
```



Note pyFAI does now about the ring position but nothing about relative intensities of rings.

Translation of the detector along the vertical axis

The vertical axis is defined along the *poni1*. If one moves the detector higher, the poni will appear at lower coordinates. So lets define 5 upwards verical translations of half the detector size.

For this we will duplicate 5x the AzimuthalIntegrator object, but instances of *AzimuthalIntegrator* are mutable, so it is important to create an actual *copy* and not an *view* on them. In Python, one can use the *copy* function of the *copy* module:

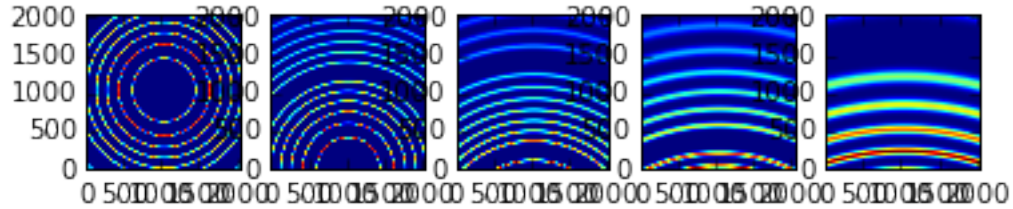
```
import copy
```

We will now offset the *poni1* value of each AzimuthalIntegratoe which correspond to a vertical translation. Each subsequent image is offsetted by half a detector width (stored as *poni1*).

```
ais = []
imgs = []
fig, plots = subplots(1,5)
for i in range(5):
    my_ai = copy.deepcopy(ai)
    my_ai.poni1 -= i*poni1
    my_img = LaB6.fake_calibration_image(my_ai)
    plots[i].imshow(my_img, origin="lower")
    ais.append(my_ai)
    imgs.append(my_img)
    print(my_ai)
```

```
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e-01m        PONI= 6.144000e-02, 6.144000e-02m       rot1=0.000000  rot2= 
DirectBeamDist= 100.000mm   Center: x=1024.000, y=1024.000 pix      Tilt=0.000 deg  tiltPlanRotati
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e-01m        PONI= 0.000000e+00, 6.144000e-02m       rot1=0.000000  rot2= 
DirectBeamDist= 100.000mm   Center: x=1024.000, y=0.000 pix Tilt=0.000 deg  tiltPlanRotation= 0.0
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Wavelength= 1.000000e-10m
```

```
SampleDetDist= 1.000000e-01m        PONI= -6.144000e-02, 6.144000e-02m      rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=1024.000, y=-1024.000 pix     Tilt=0.000 deg  tiltPlanRotati
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e-01m        PONI= -1.228800e-01, 6.144000e-02m      rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=1024.000, y=-2048.000 pix     Tilt=0.000 deg  tiltPlanRotati
Detector Titan 2k x 2k       PixelSize= 6.000e-05, 6.000e-05 m
Wavelength= 1.000000e-10m
SampleDetDist= 1.000000e-01m        PONI= -1.843200e-01, 6.144000e-02m      rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=1024.000, y=-3072.000 pix     Tilt=0.000 deg  tiltPlanRotati
```



MultiGeometry integrator

The *MultiGeometry* instance can be created from any list of *AzimuthalIntegrator* instances or list of *poni-files*. Here we will use the former method.

The main difference of a *MultiIntegrator* with a "normal" *AzimuthalIntegrator* comes from the definition of the output space in the constructor of the object. One needs to specify the unit and the integration range.

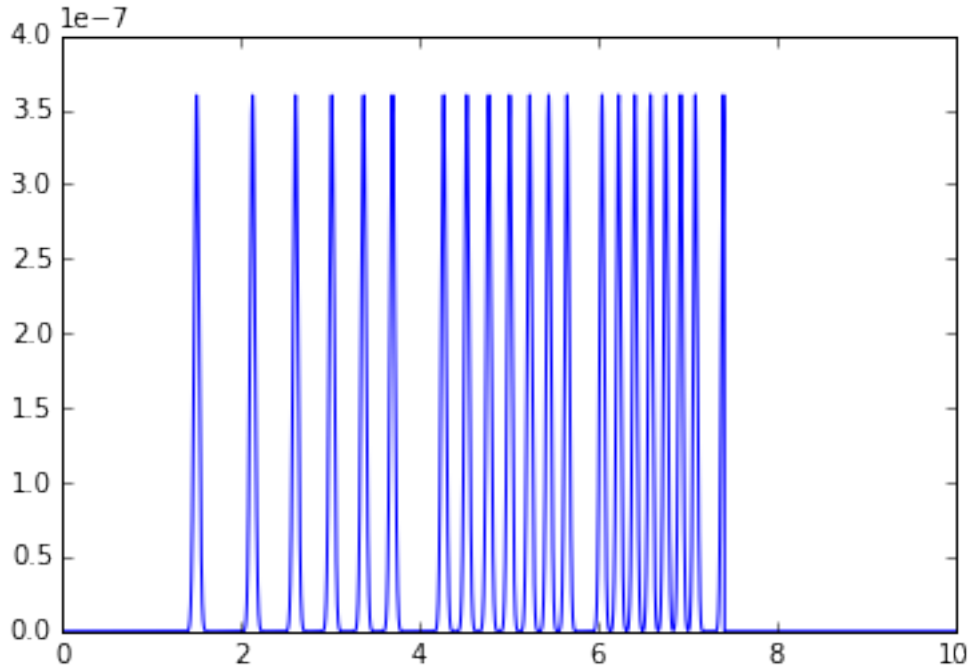
```
from pyFAI.multi_geometry import MultiGeometry

mg = MultiGeometry(ais, unit="q_A^-1", radial_range=(0, 10))
print(mg)
```

```
MultiGeometry integrator with 5 geometries on (0, 10) radial range (q_A^-1) and (-180, 180) azimut
```

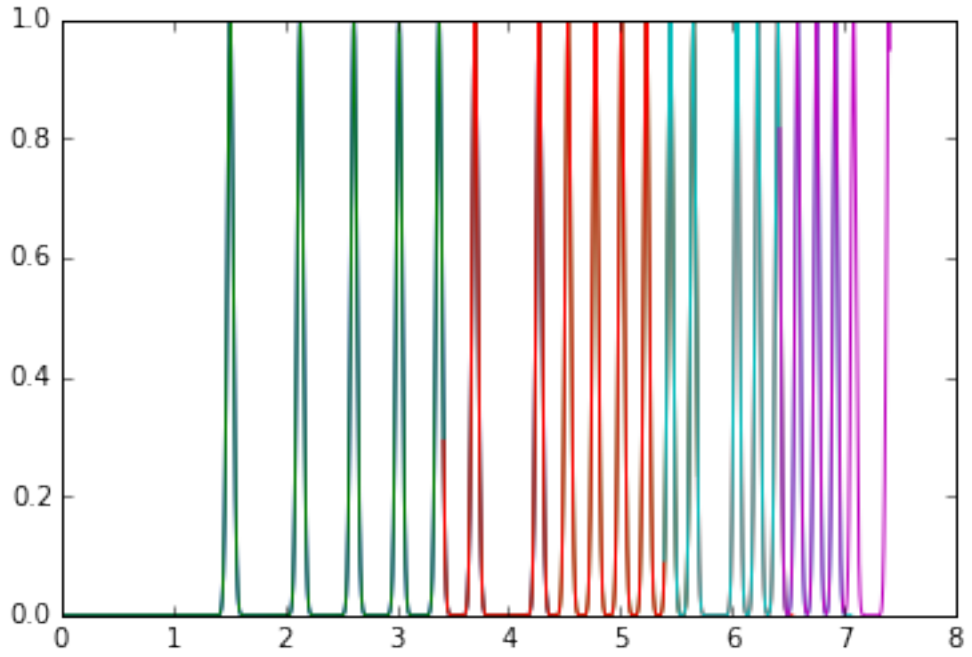
MultiGeometry integrators can be used in a similar way to "normal" *AzimuthalIntegrator*s. Keep in mind the output intensity is always scaled to absolute solid angle.

```
plot(*mg.integrate1d(imgs, 10000))
```

```
[<matplotlib.lines.Line2D at 0x7fbf90a4a210>]
```




```
for i, a in zip(imgs, ais):
    plot(*a.integrate1d(i, 1000, unit="q_A^-1"))
```



Rotation of the detector

The strength of translating the detector is that it simulates a larger detector, but this approach reaches its limit quikly as the higher the detector gets, the smallest the solid angle gets and induces artificial noise. One solution is to keep the detector at the same distance and rotate the detector.

Creation of diffraction images In this example we will use a Pilatus 200k with 2 modules. It has a gap in the middle of the two detectors and we will see how the *MultiGeometry* can help.

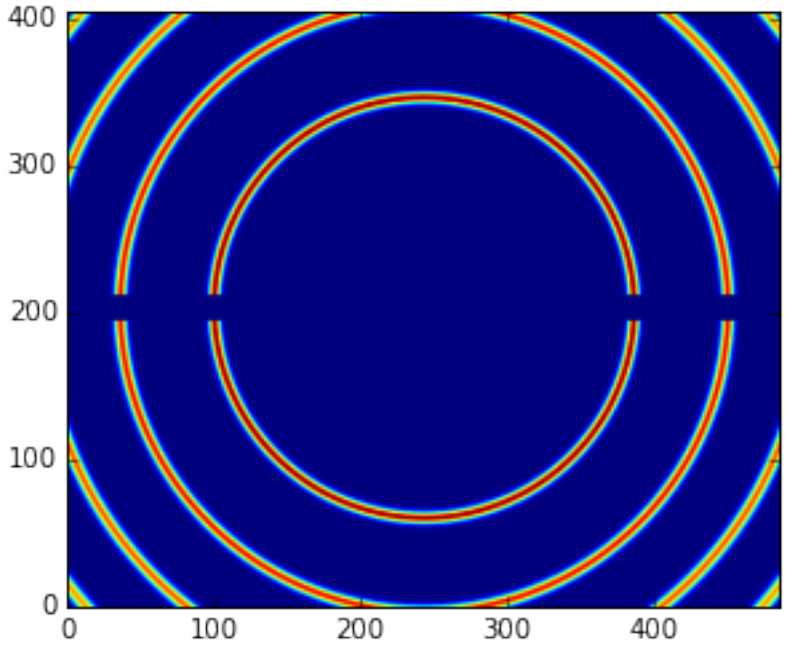
As previously, we will use LaB6 but instead of translating the images, we will rotate them along the second axis:

```
det = pyFAI.detectors.detector_factory("pilatus200k")
p1, p2, p3 = det.calc_cartesian_positions()
print(p3)
poni1 = p1.mean()
poni2 = p2.mean()
print(poni1)
print(poni2)
```

```
None
0.035002
0.041882
```

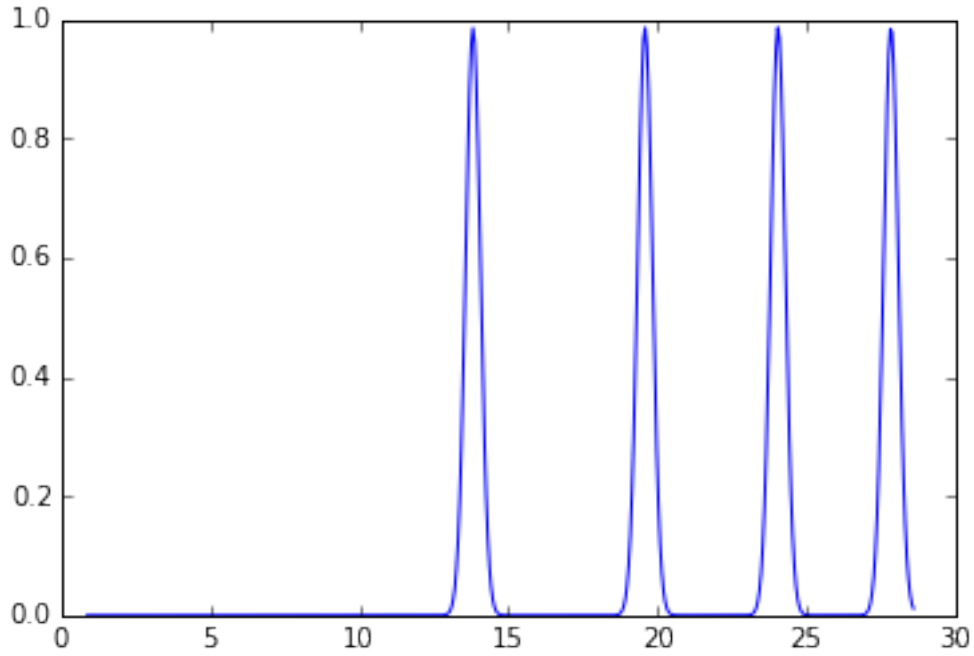
```
ai = pyFAI.AzimuthalIntegrator(dist=0.1, poni1=poni1, poni2=poni2, detector=det)
img = LaB6.fake_calibration_image(ai)
imshow(img, origin="lower")
#imshow(log(ai.integrate2d(img, 500, 360, unit="2th_deg")[0]))
```

```
<matplotlib.image.AxesImage at 0x7fbf90923790>
```




```
plot(*ai.integrate1d(img, 500,unit="2th_deg"))
```

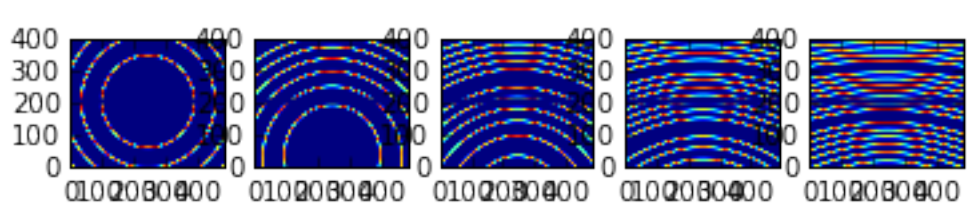
```
[<matplotlib.lines.Line2D at 0x7fbf90847490>]
```



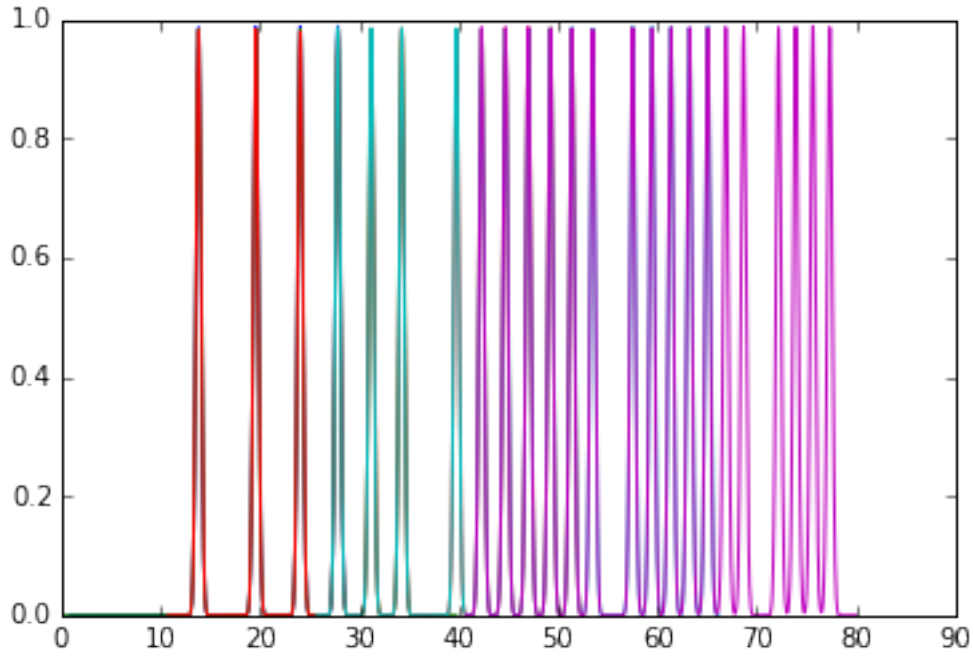
We will rotate the detector with a step size of 15 degrees

```
step = 15*pi/180
ais = []
imgs = []
fig, plots = subplots(1,5)
for i in range(5):
    my_ai = copy.deepcopy(ai)
    my_ai.rot2 -= i*step
    my_img = LaB6.fake_calibration_image(my_ai)
    plots[i].imshow(my_img, origin="lower")
    ais.append(my_ai)
    imgs.append(my_img)
    print(my_ai)
```

```
Detector Pilatus200k         PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 1.000000e-01m        PONI= 3.500200e-02, 4.188200e-02m       rot1=0.000000  rot2= 0
DirectBeamDist= 100.000mm   Center: x=243.500, y=203.500 pix        Tilt=0.000 deg  tiltPlanRotati
Detector Pilatus200k         PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 1.000000e-01m        PONI= 3.500200e-02, 4.188200e-02m       rot1=0.000000  rot2= -
DirectBeamDist= 103.528mm   Center: x=243.500, y=47.716 pix Tilt=15.000 deg  tiltPlanRotation= -90
Detector Pilatus200k         PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 1.000000e-01m        PONI= 3.500200e-02, 4.188200e-02m       rot1=0.000000  rot2= -
DirectBeamDist= 115.470mm   Center: x=243.500, y=-132.169 pix       Tilt=30.000 deg  tiltPlanRotat
Detector Pilatus200k         PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 1.000000e-01m        PONI= 3.500200e-02, 4.188200e-02m       rot1=0.000000  rot2= -
DirectBeamDist= 141.421mm   Center: x=243.500, y=-377.895 pix       Tilt=45.000 deg  tiltPlanRotat
Detector Pilatus200k         PixelSize= 1.720e-04, 1.720e-04 m
SampleDetDist= 1.000000e-01m        PONI= 3.500200e-02, 4.188200e-02m       rot1=0.000000  rot2= -
DirectBeamDist= 200.000mm   Center: x=243.500, y=-803.506 pix       Tilt=60.000 deg  tiltPlanRotat
```



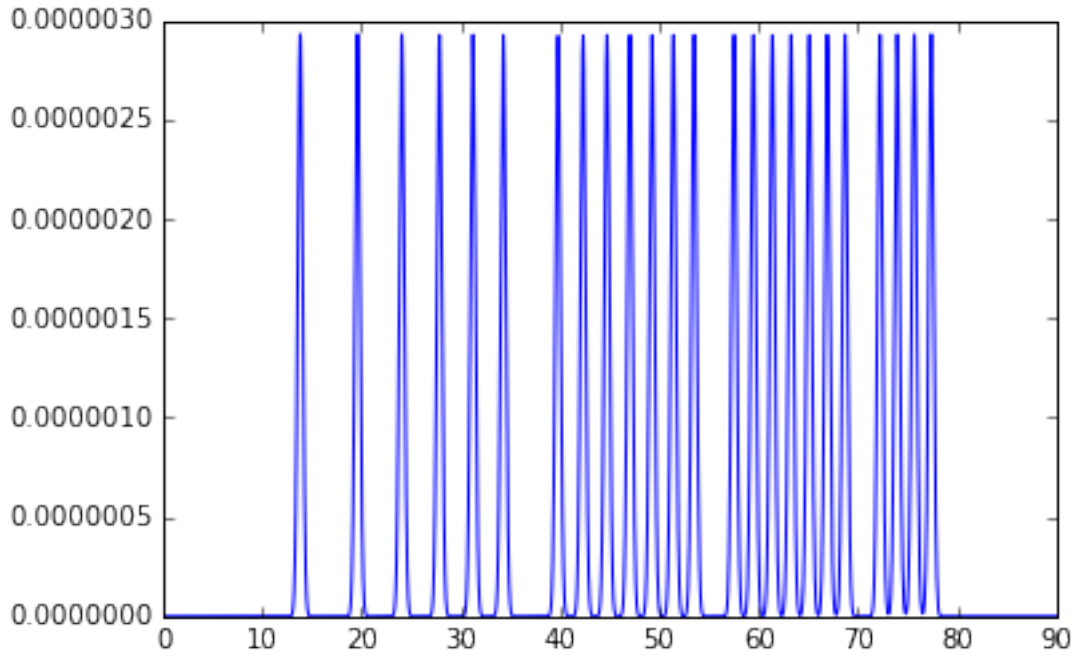
```
for i, a in zip(imgs, ais):
    plot(*a.integrate1d(i, 1000, unit="2th_deg"))
```



Creation of the MultiGeometry This time we will work in 2theta angle using degrees:

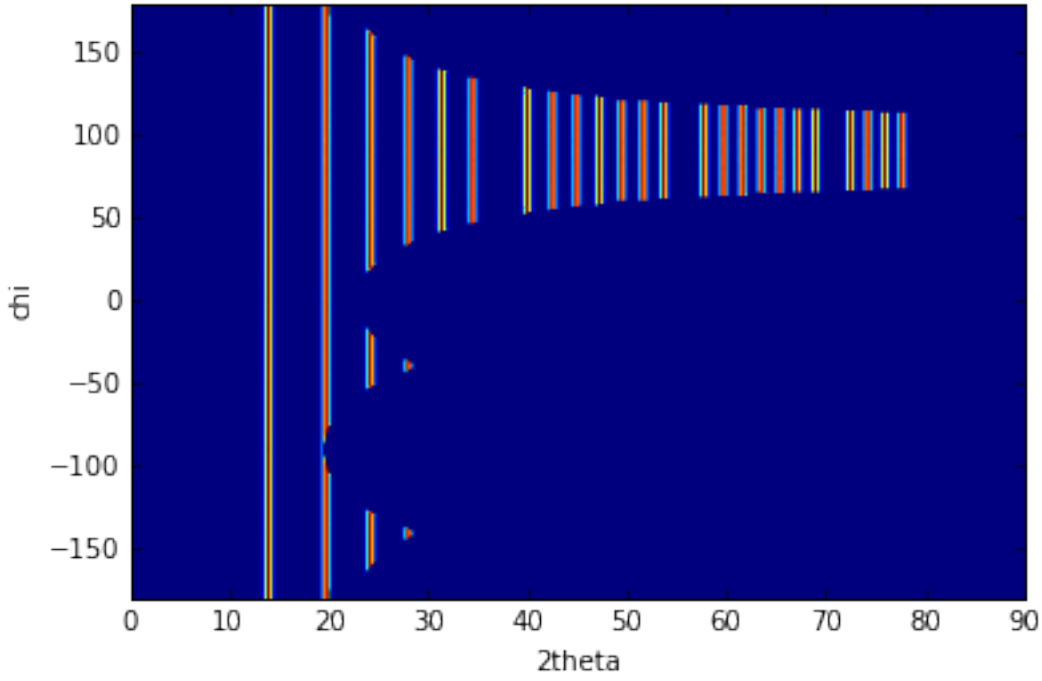
```
mg = MultiGeometry(ais, unit="2th_deg", radial_range=(0, 90))
print(mg)
plot(*mg.integrate1d(imgs, 10000))
```

```
MultiGeometry integrator with 5 geometries on (0, 90) radial range (2th_deg) and (-180, 180) azimu
area_pixel=1.32053624453 area_sum=2.69418873745, Error= -1.04022324159

[<matplotlib.lines.Line2D at 0x7fbf903e2650>]
```



```
I,tth, chi = mg.integrate2d(imgs, 1000,360)
imshow(I, origin="lower",extent=[tth.min(), tth.max(), chi.min(), chi.max()], aspect="auto")
xlabel("2theta")
ylabel("chi")
```

```
<matplotlib.text.Text at 0x7fbf90330350>
```



How to fill-up gaps in arrays of pixel detectors during 2D integration

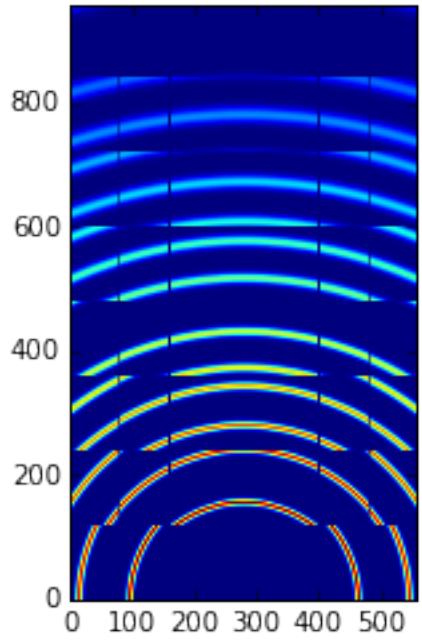
We will use ImXpad detectors which exhibits large gaps.

```
det = pyFAI.detectors.detector_factory("Xpad_flat")
p1, p2, p3 = det.calc_cartesian_positions()
print(p3)
poni1 = p1.mean()
poni2 = p2.mean()
print(poni1)
print(poni2)
```

```
None
0.076457
0.0377653
```

```
ai = pyFAI.AzimuthalIntegrator(dist=0.1, poni1=0, poni2=poni2, detector=det)
img = LaB6.fake_calibration_image(ai)
imshow(img, origin="lower")
```

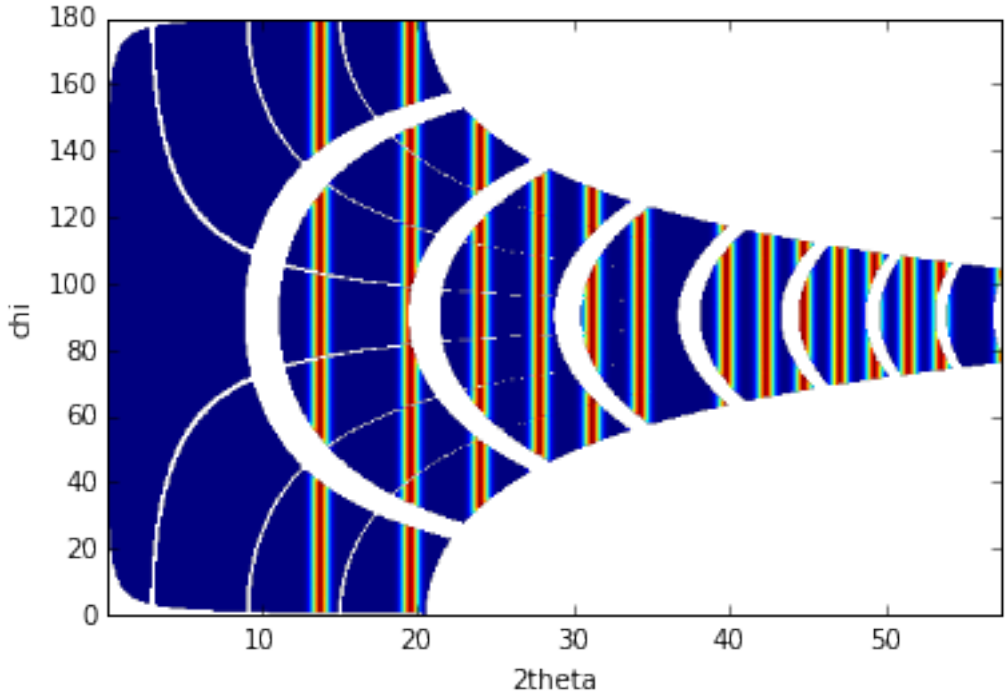
```
<matplotlib.image.AxesImage at 0x7fbf909b8210>
```




```
I, tth, chi=ai.integrate2d(img, 500, 360, azimuth_range=(0,180), unit="2th_deg", dummy=-1)
imshow(sqrt(I),origin="lower",extent=[tth.min(), tth.max(), chi.min(), chi.max()], aspect="auto")
xlabel("2theta")
ylabel("chi")
```

```
-c:2: RuntimeWarning: invalid value encountered in sqrt
```

```
<matplotlib.text.Text at 0x7fbf90a67850>
```

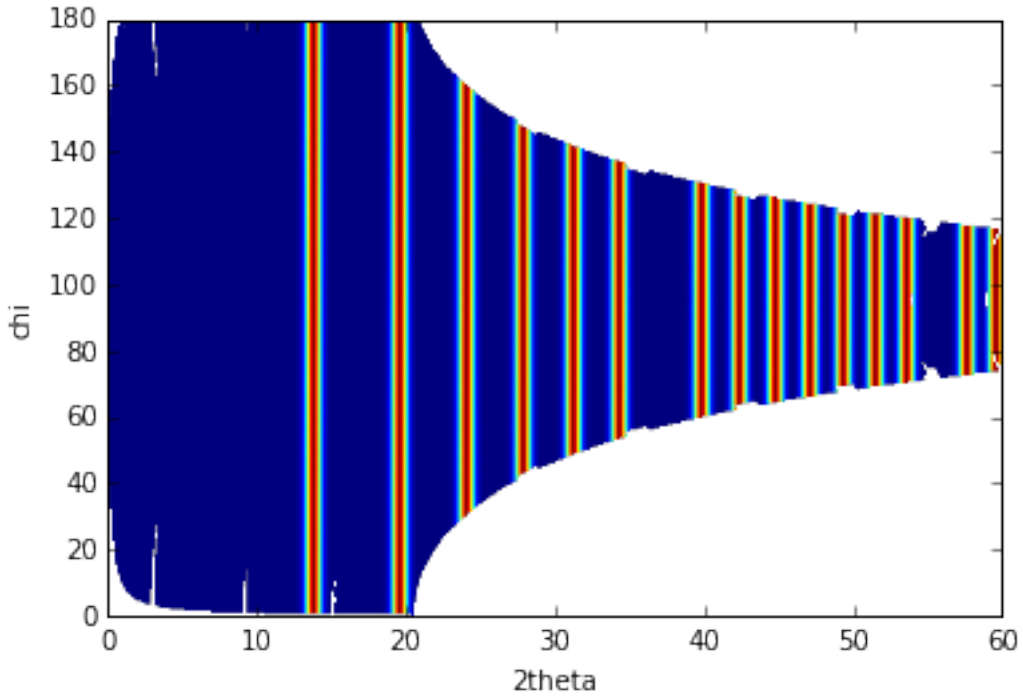


To observe textures, it is mandatory to fill the large empty space. This can be done by tilting the detector by a few degrees to higher 2theta angle (yaw 2x5deg) and turn the detector along the azimuthal angle (roll 2x5deg):

```
step = 5*pi/180
nb_geom = 3
```

```
ais = []
imgs = []
for i in range(nb_geom):
    for j in range(nb_geom):
        my_ai = copy.deepcopy(ai)
        my_ai.rot2 -= i*step
        my_ai.rot3 -= j*step
        my_img = LaB6.fake_calibration_image(my_ai)
        ais.append(my_ai)
        imgs.append(my_img)
mg = MultiGeometry(ais, unit="2th_deg", radial_range=(0, 60), azimuth_range=(0, 180), empty=-1)
print(mg)
I, tth, chi = mg.integrate2d(imgs, 1000, 360)
imshow(sqrt(I),origin="lower",extent=[tth.min(), tth.max(), chi.min(), chi.max()], aspect="auto")
xlabel("2theta")
ylabel("chi")
```

```
MultiGeometry integrator with 9 geometries on (0, 60) radial range (2th_deg) and (0, 180) azimutha

-c:16: RuntimeWarning: invalid value encountered in sqrt

<matplotlib.text.Text at 0x7fbf92c74710>
```



As on can see, the gaps have disapeared and the statistics is much better, except on the border were only one image contributes to the integrated image.

Conclusion

The multi_geometry module of pyFAI makes powder diffraction experiments with small moving detectors much easier.

Some people would like to stitch input images together prior to integration. There are plenty of good tools to do this: generalist one like Photoshop or more specialized ones like AutoPano. More seriously this can be using the distortion module of a detector to re-sample the signal on a regular grid but one will have to store on one side the number of actual pixel contributing to a regular pixels and on the other the total intensity contained in the

regularized pixel. Without the former information, doing science with a rebinned image is as meaningful as using Photoshop.

3.5.4 Conclusion

MultiGeometry is a unique feature of PyFAI ... While extremely powerful, it need careful understanding of the numerical treatement going on underneath.

3.6 Creation of a calibrant file

In this tutorial we will see how to generate a file describing a *reference material* used as calibrant to refine the geometry of the experimental setup, especially the position of the detector.

In pyFAI, calibrant are defined in a bunch of files available from github. Those files are automatically installed with pyFAI and readily available from the programming interface, as described in the **Calibrant** tutorial.

This tutorials focuses on the content of the file and how to generate other calibrant files and how to exended existing ones.

3.6.1 Content of the file

Calibrant files from *pyFAI* are heavily inspired from the ones used in *Fit2D*: they simply store the *d-spacing* between Miller plans and are simply loaded using the *numpy.loadtxt* function. In pyFAI we have improved with plent of comment (using **#**) to provide in addition the familly of Miller plan associated and the multiplicity, which, while not proportionnal to *Fobs* is somewhat related.

One may think it would have been better/simpler to describe the cell of the material. Actually some calibrant, especially the ones used in SAXS like silver behenate (AgBh), but not only, are not well crystallized compounds and providing the *d-spacing* allows to use them as calibrant. Neverthless this tutorial will be focused on how to generate calibrant files fron crystal structures.

Finally the *calibrant file* has a few lines of headers containing the name, the crystal cell parameters (if available) and quite importantly a reference to the source of information like a publication to allow the re-generation of the file if needed.

```
# Calibrant: Silver Behenate (AgBh)
# Pseudocrystal a=inf b=inf c=58.380
# Ref: doi:10.1107/S0021889899012388
5.83800000e+01 # (0,0,1)
2.91900000e+01 # (0,0,2)
1.94600000e+01 # (0,0,3)
1.45950000e+01 # (0,0,4)
1.16760000e+01 # (0,0,5)
9.73000000e+00 # (0,0,6)
8.34000000e+00 # (0,0,7)
7.29750000e+00 # (0,0,8)
6.48666667e+00 # (0,0,9)
(...)
```

3.6.2 The Cell class

To generate a *calibrant file* from a crystal structure, the easiest is to use the **pyFAI.calibrant.Cell** class.

```
from pyFAI.calibrant import Cell
print(Cell.__doc__)
print(Cell.__init__.__doc__)
```

```
    This is a cell object, able to calculate the volume and d-spacing according to formula from:

    http://geoweb3.princeton.edu/research/MineralPhy/xtalgeometry.pdf

Constructor of the Cell class:

        Crystalographic units are Angstrom for distances and degrees for angles !

        @param a,b,c: unit cell length in Angstrom
        @param alpha, beta, gamma: unit cell angle in degrees
        @param lattice: "cubic", "tetragonal", "hexagonal", "rhombohedral", "orthorhombic", "mono
        @param lattice_type: P, I, F, C or R
```

The constructor of the class is used to build and well suited to triclinic crystal.

Specific constructors

Nevertheless, most used calibrants are of much higher symmetry, like cubic which takes only **one** parameter.

Here is an example for defining Polonium which is a simple cubic cell (Primitive) with a cell parameter of 335.9pm. This example was chosen as Polonium is apparently the only element with such primitive cubic packing.

```
Po = Cell.cubic(3.359)
print(Po)
```

```
Primitive cubic cell a=3.3590 b=3.3590 c=3.3590 alpha=90.000 beta=90.000 gamma=90.000
```

```
print(Po.volume)
```

```
37.899197279
```

```
print("Po.d_spacing?")
print(Po.d_spacing.__doc__)
print("Po.save?")
print(Po.save.__doc__)
```

```
Po.d_spacing?
Calculate all d-spacing down to dmin

        applies selection rules

        @param dmin: minimum value of spacing requested
        @return: dict d-spacing as string, list of tuple with Miller indices
                preceded with the numerical value

Po.save?
Save informations about the cell in a d-spacing file, usable as Calibrant

        @param name: name of the calibrant
        @param doi: reference of the publication used to parametrize the cell
        @param dmin: minimal d-spacing
        @param dest_dir: name of the directory where to save the result
```

To generate a *d-spacing* file usable as calibrant, one simply has to call the *save* method of the *Cell* instance.

Nota: the ".D" suffix is automatically appended.

```
Po.save("Po",doi="http://www.periodictable.com/Elements/084/data.html", dmin=1)
```

```
!cat Po.D
```

```
# Calibrant: Po
# Primitive cubic cell a=3.3590 b=3.3590 c=3.3590 alpha=90.000 beta=90.000 gamma=90.000
# Ref: http://www.periodictable.com/Elements/084/data.html
3.35900000 # (1, 0, 0) 6
2.37517168 # (1, 1, 0) 12
1.93931955 # (1, 1, 1) 8
1.67950000 # (2, 0, 0) 6
1.50219047 # (2, 1, 0) 24
1.37130601 # (2, 1, 1) 24
1.18758584 # (2, 2, 0) 12
1.11966667 # (3, 0, 0) 30
1.06220907 # (3, 1, 0) 24
1.01277661 # (3, 1, 1) 24
```

Other Examples: LaB6

Lantanide Hexaboride, probably my favorite calibrant as it has a primitive cubic cell hence all reflections are valid and intense. The cell parameter is available from the NIST at a=4.156826.

The number of reflections in a file is controled by the *dmin* parameter. The lower it is, the more lines there are and the more time-consuming this will be:

```
LaB6 = Cell.cubic(4.156826)
%time d=LaB6.d_spacing(0.1)
print("Number of reflections: %s"%len(d))

CPU times: user 976 ms, sys: 76 ms, total: 1.05 s
Wall time: 964 ms
Number of reflections: 1441

LaB6.save("LaB6",doi="https://www-s.nist.gov/srmors/certificates/660C.pdf", dmin=0.1)

!head LaB6.D

# Calibrant: LaB6
# Primitive cubic cell a=4.1568 b=4.1568 c=4.1568 alpha=90.000 beta=90.000 gamma=90.000
# Ref: https://www-s.nist.gov/srmors/certificates/660C.pdf
4.15682600 # (1, 0, 0) 6
2.93931985 # (1, 1, 0) 12
2.39994461 # (1, 1, 1) 8
2.07841300 # (2, 0, 0) 6
1.85898910 # (2, 1, 0) 24
1.69701711 # (2, 1, 1) 24
1.46965993 # (2, 2, 0) 12
```

Other Examples: Silicon

Silicon is easy to obtain **pure** thanks to microelectronics industry. Its cell is a face centered cubic with a diamond like structure. The cell parameter is available from the NIST: a=5.431179 A.

Let's compare the difference between the silicon structure and the equivalent FCC structure:

```
Si = Cell.diamond(5.431179)
print(Si)

Face centered cubic cell a=5.4312 b=5.4312 c=5.4312 alpha=90.000 beta=90.000 gamma=90.000

FCC = Cell.cubic(5.431179,"F")
print(FCC)
```

```
Face centered cubic cell a=5.4312 b=5.4312 c=5.4312 alpha=90.000 beta=90.000 gamma=90.000
```

Apparently, there is not difference. But to check it, let's generate all lines down to 1A and compare them:

```
sid = Si.d_spacing(1)
for key, val in sid.items():
    print("%s: %s"%(sorted(val[1:][-1]),key))
```

```
[2, 2, 4]: 1.10863477e+00
[1, 1, 3]: 1.63756208e+00
[0, 2, 2]: 1.92021175e+00
[1, 1, 1]: 3.13569266e+00
[1, 1, 5]: 1.04523089e+00
[1, 3, 3]: 1.24599792e+00
[0, 0, 4]: 1.35779475e+00
```

```
fccd = FCC.d_spacing(1)
for key, val in fccd.items():
    print("%s: %s"%(sorted(val[1:][-1]),key))
```

```
[2, 2, 4]: 1.10863477e+00
[1, 1, 3]: 1.63756208e+00
[0, 2, 2]: 1.92021175e+00
[1, 1, 1]: 3.13569266e+00
[1, 1, 5]: 1.04523089e+00
[0, 2, 4]: 1.21444854e+00
[2, 2, 2]: 1.56784633e+00
[1, 3, 3]: 1.24599792e+00
[0, 0, 2]: 2.71558950e+00
[0, 0, 4]: 1.35779475e+00
```

So there are many more reflection in the FCC structure compared to the diamond-like structure: (4 2 0), (2 2 2) are extinct as h+k+l=4n and all even.

Selection rules

Cell object contain *selection_rules* which tells if a reflection is allowed or not. Those *selection_rules* can be introspected:

```
print(Si.selection_rules)
print(FCC.selection_rules)
```

```
[<function <lambda> at 0x7f8080fad848>, <function <lambda> at 0x7f8080fad938>, <function <lambda>
[<function <lambda> at 0x7f8080fadaa0>, <function <lambda> at 0x7f8080fada28>]
```

The *Si* object has one additionnal selection rule which explains the difference: A specific rule telling that reflection with h+k+l=4n is forbidden when (h,k,l) are all even.

We will now have a look at the source code of those selection rules using the *inspect* module.

Nota: This is advanced Python hacking but useful for the demonstration

```
import inspect
for rule in Si.selection_rules:
    print(inspect.getsource(rule))
```

```
self.selection_rules = [lambda h, k, l: not(h == 0 and k == 0 and l == 0)]

    self.selection_rules.append(lambda h, k, l: (h % 2 + k % 2 + l % 2) in (0, 3))

    lambda h, k, l:not((h % 2 == 0) and (k % 2 == 0) and (l % 2 == 0) and ((h + k + l) % 4 != 0))
```

Actually the last line correspond to an anonymous function (lambda) which implements this rule.

As we have seen previously one can simply adapt the Cell instance by simply appending extra selection rules to this list.

A selection rule is a function taking 3 integers as input and returning *True* if the reflection is allowed and *False* when it is forbidden by symmetry. By this way one can simply adapt existing object to generate the right *calibrant file*.

3.6.3 Conclusion

In this tutorial we have seen the structure of a *calibrant file*, how to generate crystal structure cell object to write such calibrant files automatically, including all metadata needed for redistribution. Most advanced programmers can now modify the selection rules to remove forbidden reflection for a given cell.

3.7 CCD calibration

This tutorial presents how to calibrate the distortion from a CCD camera coupled with a taper of optic fibers. If your camera is already calibrated using Fit2D and you have access to the corresponding *spline* file, this tutorial is not for you: simply create your detector object like this `pyFAI.detectors.Detector(splineFile="example.spline")` and you are done. This tutorial uses the image of a regular grid on the detector.

It uses a procedure described in: "Calibration and correction of spatial distortions in 2D detector systems" from Hammersley, A. P.; Svensson, S. O.; Thompson, A. published in Nuclear Instruments and Methods in Physics Research Section A, Volume 346, Issue 1-2, p. 312-321. DOI:10.1016/0168-9002(94)90720-X

The procedure is performed in 4 steps:

1. peak picking
2. grid assignment
3. distortion fitting
4. interpolation of the fitted data
5. saving into a detector definition file

The picture used is the one of a regular metallic grid of holes (spaced by 5mm), just in front of the detector. We will assume holes are circular what looks correct in first approximation. Parallax error will be ignored in a first time.

3.7.1 Peak picking

Lets start with peak picking, for this, we will use the *FabIO* library able to read the image and *matplotlib* to display the image. The distortion is assumed to be minimal in the middle of the detector, so we first focus on one spot in the middle:

```
#For final rendering switch from nbagg -> inline
#%pylab nbagg
%pylab inline
```

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

```
# search for the image containing the grid
!ls *.edf
```

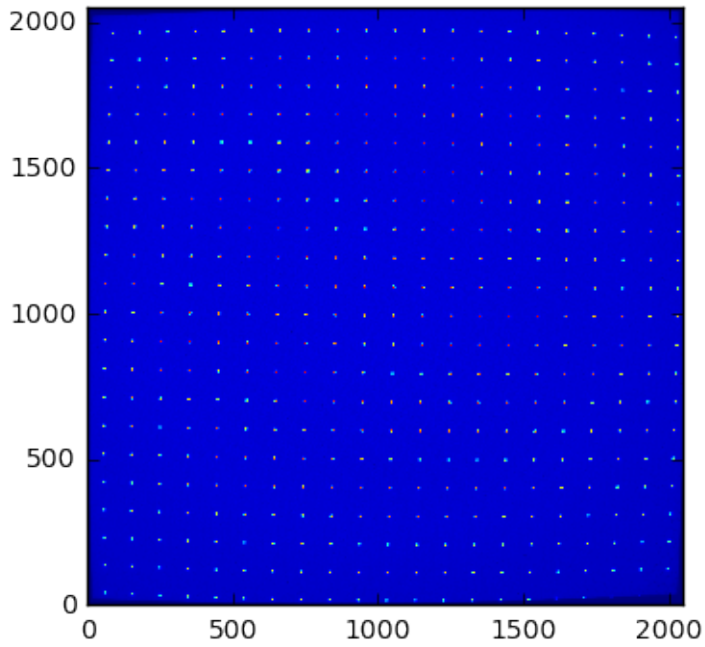
```
corrected.edf  frelonID22_grid.edf
```

```
fname = "frelonID22_grid.edf"
#fname = "corrected.edf"
import fabio
img = fabio.open(fname).data
```

```
WARNING:fabioimage:PIL is not installed ... trying to do without
WARNING:tifimage:PIL is not installed ... trying to do without
WARNING:bruker100image:PIL is not installed ... trying to do without
WARNING:xsdimage:lxml library is probably not part of your python installation: disabling xsdimage
```

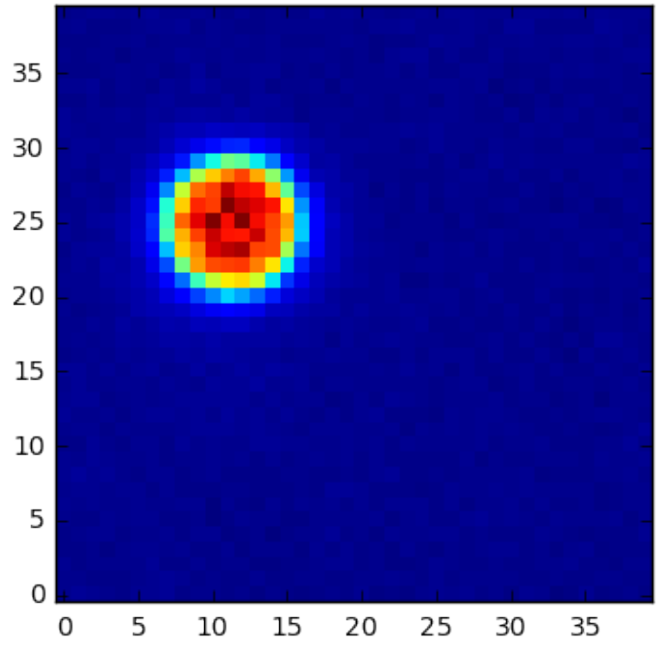
```
imshow(img, interpolation="nearest", origin="lower")
```

```
<matplotlib.image.AxesImage at 0x7f2c70f1add8>
```



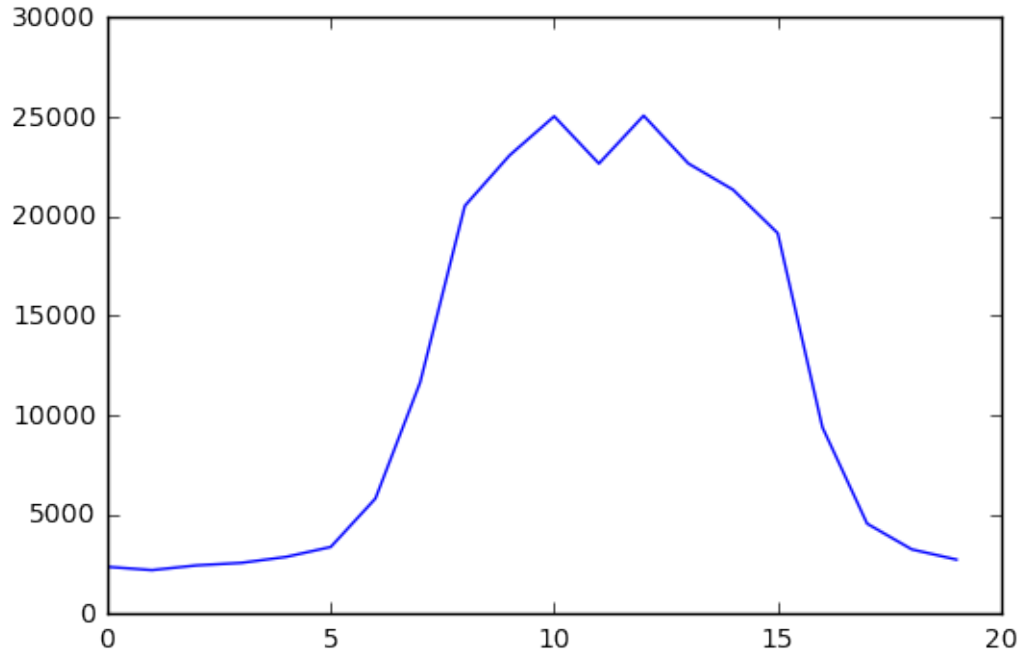

```
#Zoom into a spot in the middle of the image, where the distortion is expected to be minimal
imshow(img[1060:1100,1040:1080], interpolation="nearest", origin="lower")
```

```
<matplotlib.image.AxesImage at 0x7f2c70ebbba8>
```




```
#Look at the profile of the peak to measure the width (it is expected to be a crenel)
plot(img[1060+25,1040:1060])
```

```
[<matplotlib.lines.Line2D at 0x7f2c70e383c8>]
```



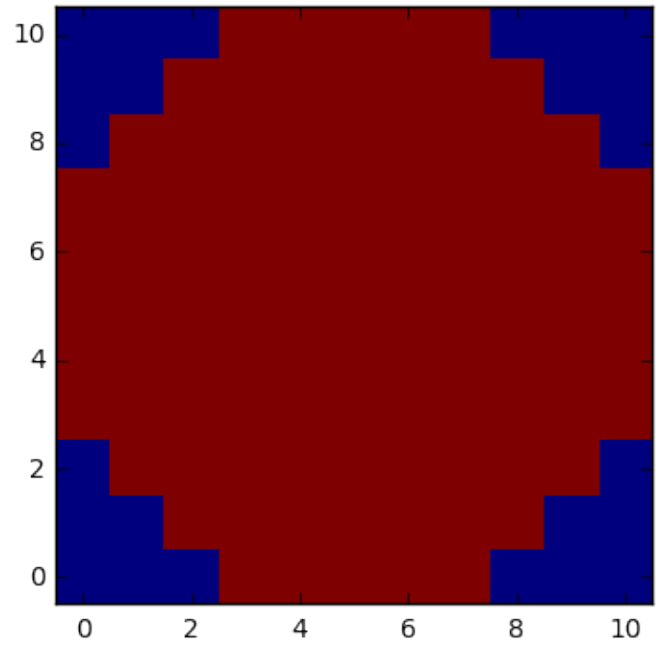
Let's look at one spot, in the center of the image: it is circular and is slightly larger than 10 pixels. We will define a convolution kernel of size 11x11 of circular shape with sharp borders as this is what a perfect spot is expected to look like. The kernel is normalized in such a way it does not modify the average intensity of the image

Now convolve the image with this circular kernel using scipy.signal (in direct space: the kernel is small and performance does not really matter here).

It is important to have an odd size for the kernel for convolution as an even shape would induce an offset of 1/2 pixel in the located peak-position.

```
size = 11 #Odd of course
center = (size-1)//2
y, x = numpy.ogrid[-center:center+1,-center:center+1]
r2 = x*x + y*y
kernel = (r2<=(center+0.5)**2).astype(float)
kernel /= kernel.sum()
imshow(kernel, interpolation="nearest", origin="lower")
```

```
<matplotlib.image.AxesImage at 0x7f2c70d9c438>
```

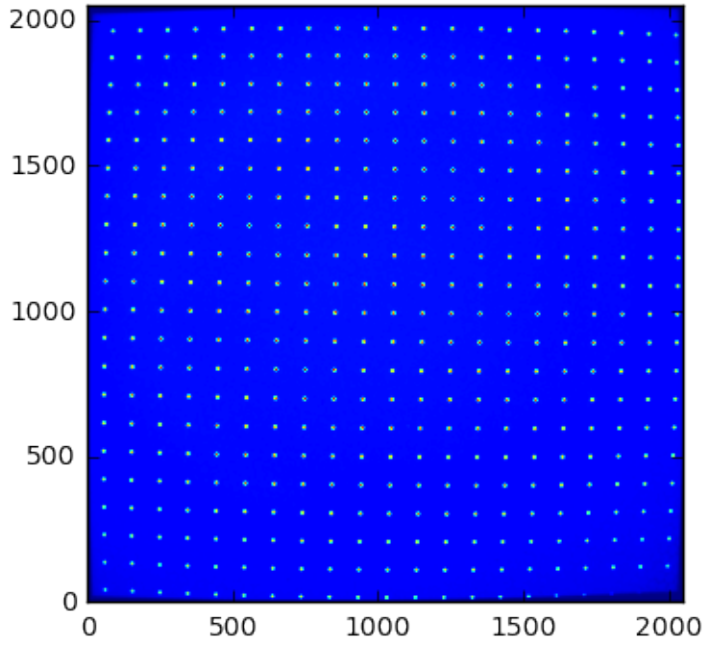



```
from scipy import ndimage, signal

cnv = signal.convolve2d(img, kernel, mode="same")

#Check that size is unchanged.
print(img.shape)
print(cnv.shape)
```

```
(2048, 2048)
(2048, 2048)
```

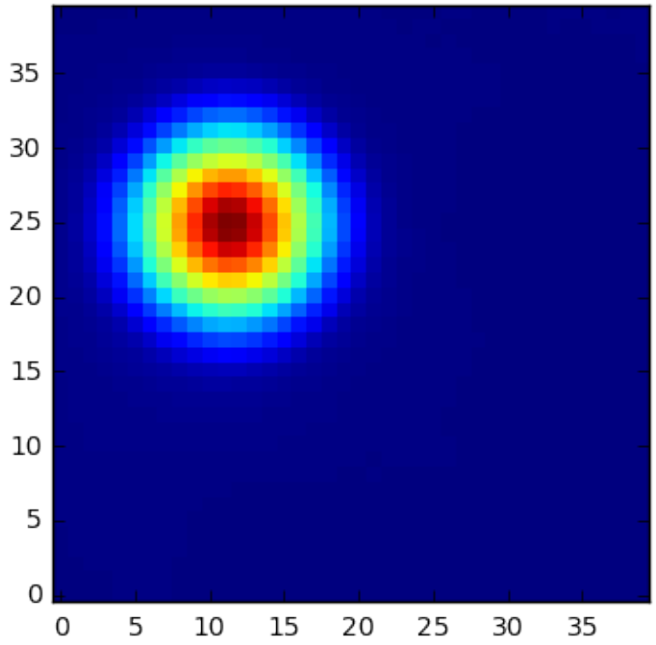
```
#Check the image still looks the same. it is just supposed to be smoother.
imshow(cnv, origin="lower", interpolation="nearest")
```

```
<matplotlib.image.AxesImage at 0x7f2c5e259550>
```



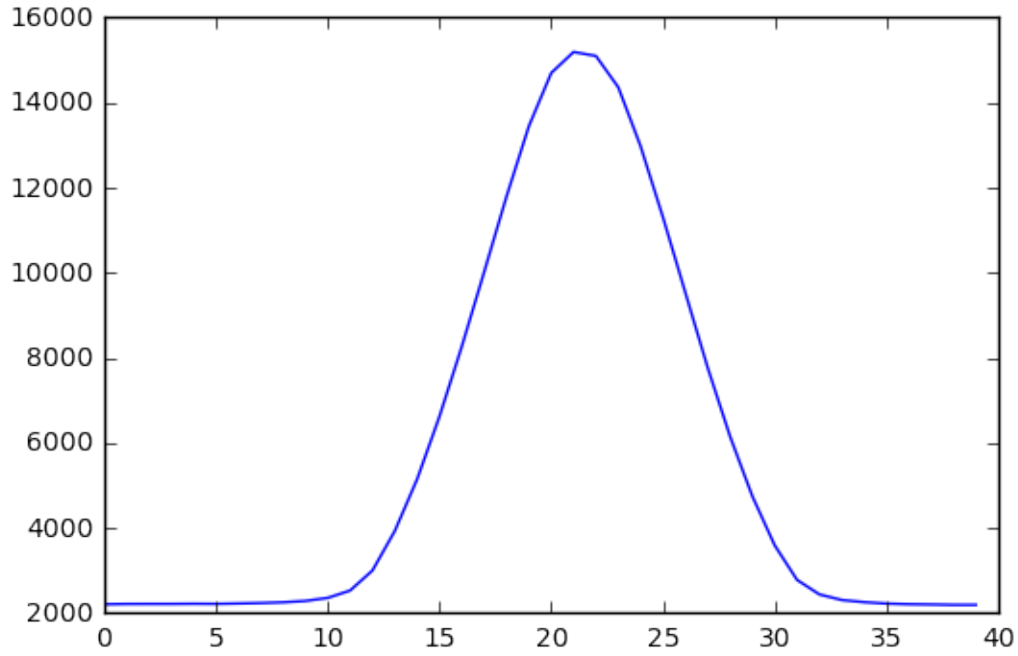

```
#Zoom into the very same spot to ensure it is smoother
imshow(cnv[1060:1100,1040:1080], interpolation="nearest", origin="lower")
```

```
<matplotlib.image.AxesImage at 0x7f2c5e1f8e80>
```




```
# and here again the same profile:
plot(cnv[1060+25,1030:1070])
# the peak got broader (2x) but much smoother on the top: this is what we are interrested in.
```

```
[<matplotlib.lines.Line2D at 0x7f2c5e160358>]
```



After convolution with a pattern of the same shape as the hole, the peak center is located with a sub-pixel resolution. The peak has a full size of 30 pixels in 1 dimension.

All peak positions will be extracted using the pyFAI inverse watershed algorithm. Once all regions are segmented, the ones too small are sieved out and the remaining ones are classifies according to their peak intensity using an histogram. As intensity vary a lot, this histogram it is done on the log-scale of the intensity.

```
mini = (kernel>0).sum()
print("Number of points in the kernel: %s"%mini)
```

```
Number of points in the kernel: 97
```

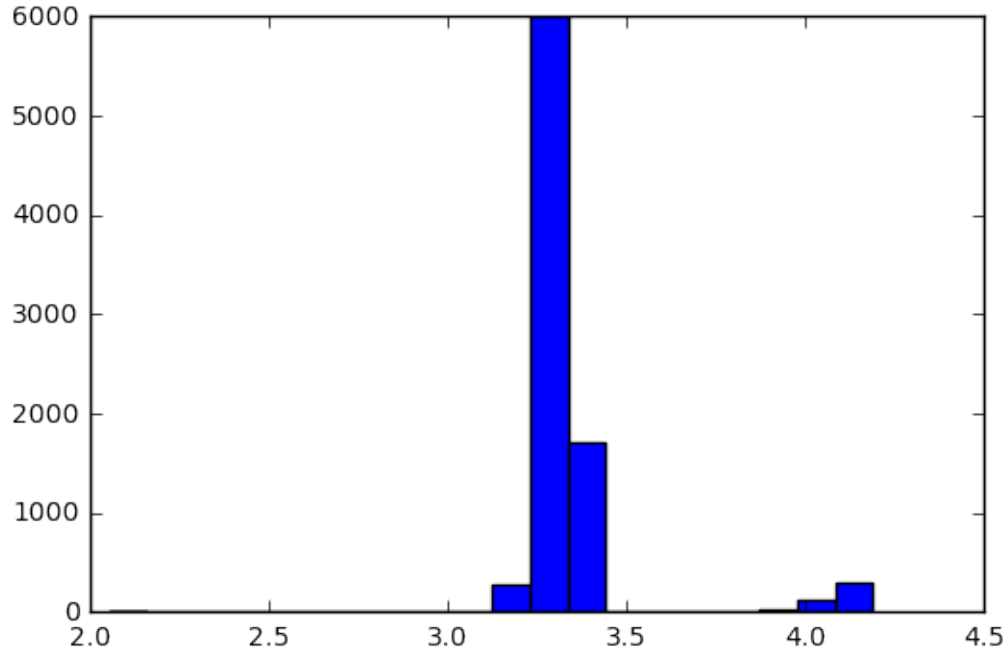
```
try: #depends if the version of pyFAI you are using
    from pyFAI.watershed import InverseWatershed
except:
    from pyFAI.ext.watershed import InverseWatershed
    #Version of pyFAI newer than feb 2016
iw = InverseWatershed(cnv)
iw.init()
iw.merge_singleton()
all_regions = set(iw.regions.values())
regions = [i for i in all_regions if i.size>mini]

print("Number of region segmented: %s"%len(all_regions))
print("Number of large enough regions : %s"%len(regions))
```

```
WARNING:pyFAI.utils:Exception No module named 'fftw3': FFTw3 not available. Falling back on Scipy
WARNING:pyFAI.opencl:Unable to import pyOpenCl. Please install it from: http://pypi.python.org/pyp
WARNING:pyFAI.timeit:init_labels took 1.104s
WARNING:pyFAI.timeit:init_borders took 0.048s
WARNING:pyFAI.timeit:init_regions took 0.450s
WARNING:pyFAI.timeit:init_pass took 0.143s
WARNING:pyFAI.timeit:merge_singleton took 0.033s

Number of region segmented: 79513
Number of large enough regions : 8443
```

```
s = [i.maxi for i in regions]
hist(numpy.log10(s), 20)
```

```
#Look for the maximum value in each region to be able to segment accordingly
```

```
(array([  1.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          1.00000000e+00,   2.85000000e+02,   5.99900000e+03,
          1.71900000e+03,   1.00000000e+00,   0.00000000e+00,
          1.00000000e+00,   3.00000000e+00,   2.30000000e+01,
          1.17000000e+02,   2.93000000e+02]),
 array([ 2.05537045,  2.1621182 ,  2.26886594,  2.37561369,  2.48236143,
         2.58910918,  2.69585692,  2.80260467,  2.90935241,  3.01610016,
         3.1228479 ,  3.22959565,  3.33634339,  3.44309114,  3.54983888,
         3.65658663,  3.76333437,  3.87008212,  3.97682986,  4.08357761,
         4.19032535]),
 <a list of 20 Patch objects>)
```



There are clearly 3 groups of very different intensity, well segregated:

- around $10^{2.1}$ (~125), those are the peaks where no tapper brings light
- around $10^{3.4}$ (~2500), those are segmented region in the background
- above $10^{3.9}$ (~8000), those are actual peaks, we are looking for.

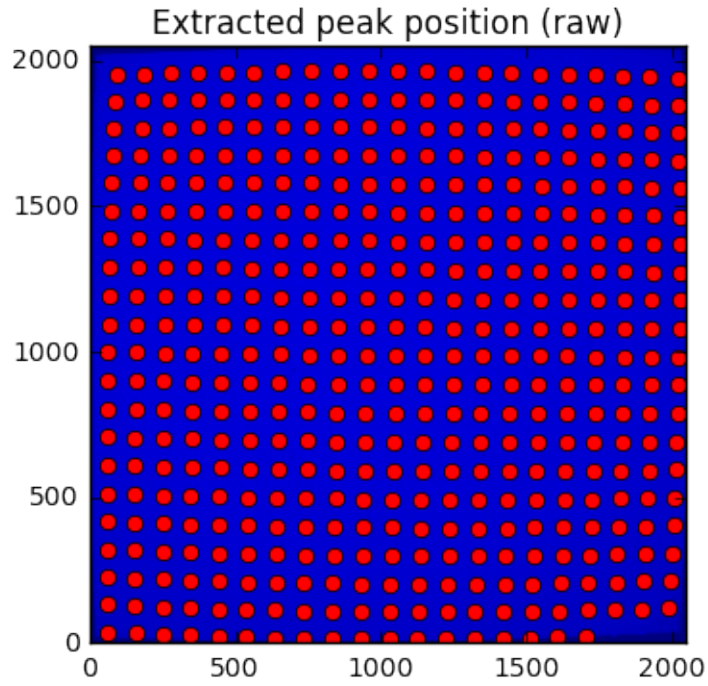
We retain all peaks > $10^{3.5}$

```
peaks = [(i.index//img.shape[-1], i.index%img.shape[-1]) for i in regions if (i.maxi)>10**3.5]
print("Number of remaining peaks: %s"%len(peaks))
```

```
Number of remaining peaks: 438
```

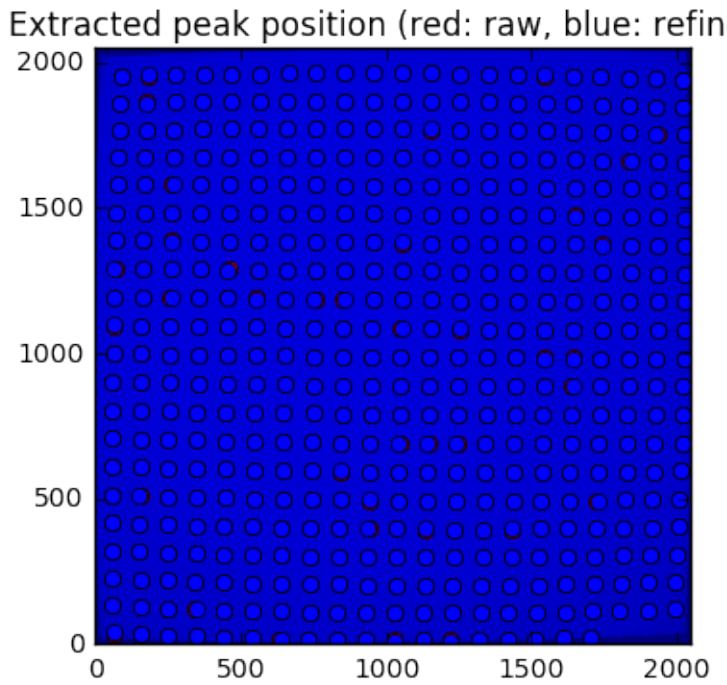
```
imshow(img, interpolation="nearest", origin="lower")
peaks_raw = numpy.array(peaks)
plot(peaks_raw[:,1], peaks_raw[:, 0], "or")
xlim(0,2048)
ylim(0,2048)
title("Extracted peak position (raw)")
print("Raw peak coordinate:")
print(peaks[:10])
```

```
Raw peak coordinate:
[(1273, 2027), (1664, 1742), (1666, 1646), (1866, 1155), (1274, 1933), (1867, 466), (1867, 563),
```



3.7.2 Precise peak extraction is performed using a second order tailor expansion

```
try:
    from pyFAI.bilinear import Bilinear
except:
    from pyFAI.ext.bilinear import Bilinear
bl = Bilinear(cnv)

ref_peaks = [bl.local_maxi(p) for p in peaks]
imshow(img, interpolation="nearest", origin="lower")
peaks_ref = numpy.array(ref_peaks)
plot(peaks_raw[:,1], peaks_raw[:, 0], "or")
plot(peaks_ref[:,1],peaks_ref[:, 0], "ob")
xlim(0,2048)
ylim(0,2048)
title("Extracted peak position (red: raw, blue: refined)")
print("Refined peak coordinate:")
print(ref_peaks[:10])

Refined peak coordinate:
[(1272.9463423714042, 2026.5502902269363), (1664.0545781441033, 1742.1054049506783), (1666.296777
```



At this stage, a visual inspection of the grid confirms all peaks have been properly segmented. If this is not the case, one can adapt:

- the size of the kernel
- the threshold coming out of the histogramming

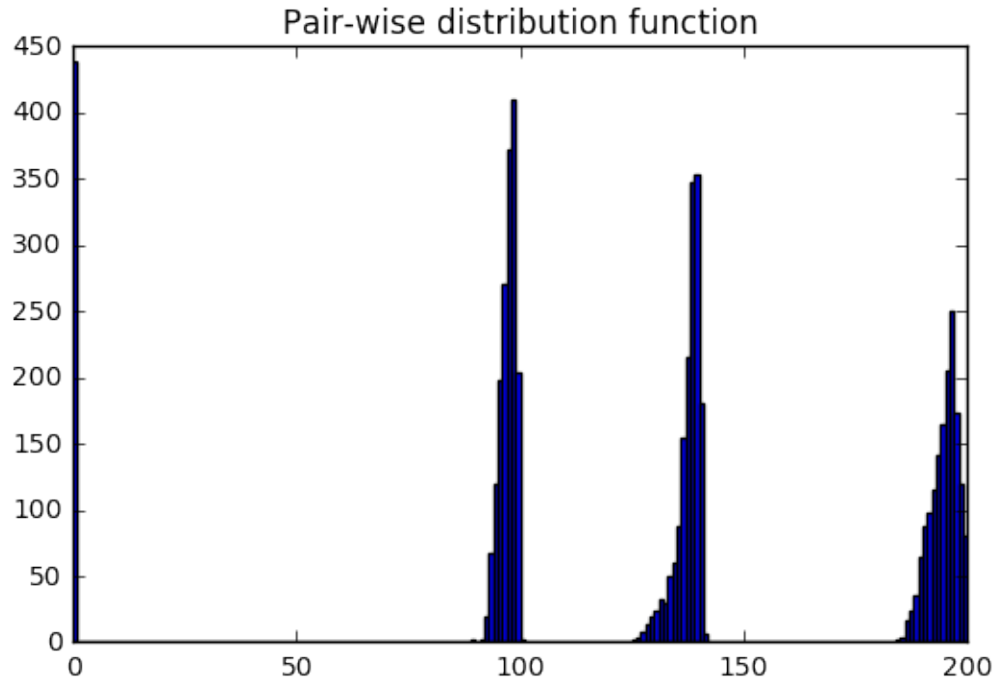
3.7.3 Pair-wise distribution function

We will now select the (4-) first neighbours for every single peak. For this we calculate the distance_matrix from any point to any other:

```
# Nota, pyFAI uses **C-coordinates** so they come out as (y,x) and not the usual (x,y).
# This notation helps us to remind the order
yx = numpy.array(ref_peaks)

# pairwise distance calculation using scipy.spatial.distance_matrix
from scipy.spatial import distance_matrix
dist = distance_matrix(peaks_ref, peaks_ref)
```

Let's have a look at the pairwise distribution function for the first neighbors

```
hist(dist.ravel(), 200, range=(0,200))
title("Pair-wise distribution function")

<matplotlib.text.Text at 0x7f2c4e3468d0>
```



This histogram provides us:

- At 0, the 438 peaks with 0-distance to themselves.
- between 85 and 105 the first neighbours
- between 125 and 150 the second neighbours.
- ... and so on.

We now focus on the first neighbours which are all located between 70 and 110 pixels apart.

```
#We define here a data-type for each peak (called center) with 4 neighbours (called north, east,
point_type = np.dtype([('center_y', float), ('center_x', float),
                       ('east_y', float), ('east_x', float),
                       ('west_y', float), ('west_x', float),
                       ('north_y', float), ('north_x', float),
                       ('south_y', float), ('south_x', float)])

neig = np.logical_and(dist>70.0, dist<110.0)
valid = (neig.sum(axis=-1)==4).sum()
print("There are %i control point with exactly 4 first neigbours"%valid)
# This initializes an empty structure to be populated
point = numpy.zeros(valid, point_type)

There are 359 control point with exactly 4 first neigbours

#Populate the structure: we use a loop as it loops only over 400 points
h=-1
for i, center in enumerate(peaks_ref):
    if neig[i].sum()!=4: continue
    h+=1
    point[h]["center_y"],point[h]["center_x"] = center
    for j in ((0,1),(0,-1),(1,0),(-1,0)):
        tmp = []
        for k in numpy.where(neig[i]):
            curr = yx[k]
            tmp.append(dot(curr-center,j))
            l = argmax(tmp)
            y, x = peaks_ref[numpy.where(neig[i])][l]
```

```
            if j==(0,1):point[h]["east_y"], point[h]["east_x"] = y, x
            elif j==(0,-1):point[h]["west_y"], point[h]["west_x"] = y, x
            elif j==(1,0): point[h]["north_y"],point[h]["north_x"] = y, x
            elif j==(-1,0):point[h]["south_y"],point[h]["south_x"] = y, x
```

We will need to define an *origin* but taking it on the border of the image is looking for trouble as this is where distortions are likely to be the most important. The center of the detector is an option but we prefer to take the peak the nearest to the centroid of all other peaks.

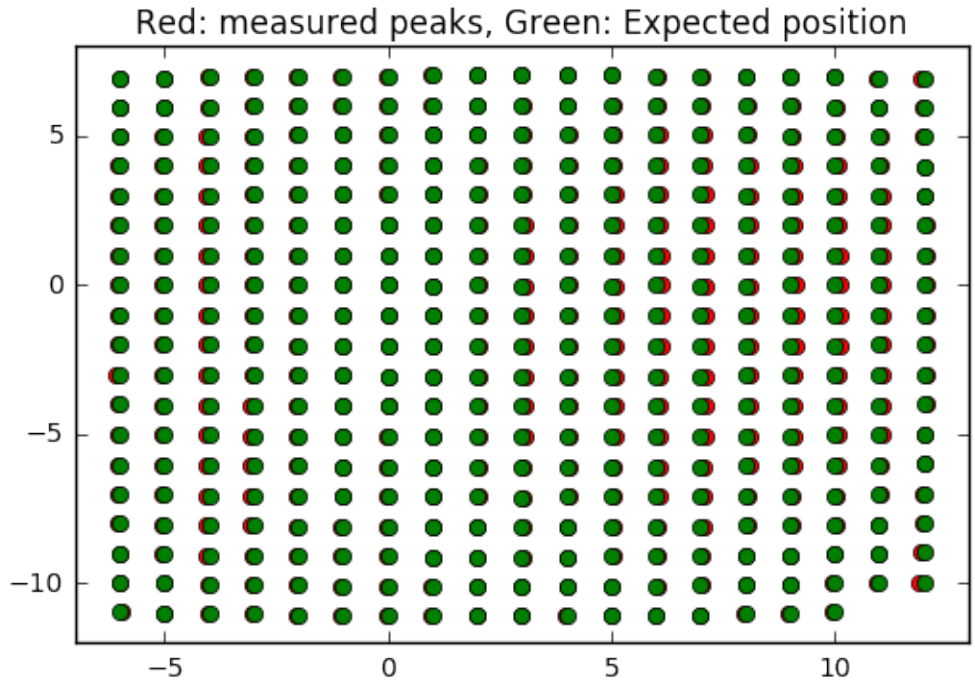
```
#Select the initial guess for the center:

#Most intense peak:
#m = max([i for i in regions], key=lambda i:i.maxi)
#Cx, Cy = m.index%img.shape[-1],m.index//img.shape[-1]
#Cx, Cy = point["center_x"].mean(), point["center_y"].mean() #Centroid of all points
Cx, Cy = 734, 1181 #beam center
#Cx, Cy = tuple(i//2 for i in cnv.shape) #detector center
print("The guessed center is at (%s, %s)"%(Cx, Cy))

#Get the nearest point from centroid:
d2 = ((point["center_x"]-Cx)**2+(point["center_y"]-Cy)**2)
best = d2.argmin()
Op = point[best]
Ox, Oy = Op["center_x"], Op["center_y"]

print("The center is at (%s, %s)"%(Ox, Oy))
#Calculate the average vector along the 4 main axes
Xx = (point[:]["east_x"] - point[:]["center_x"]).mean()
Xy = (point[:]["east_y"] - point[:]["center_y"]).mean()
Yx = (point[:]["north_x"] - point[:]["center_x"]).mean()
Yy = (point[:]["north_y"] - point[:]["center_y"]).mean()

print("The X vector is is at (%s, %s)"%(Xx, Xy))
print("The Y vector is is at (%s, %s)"%(Yx, Yy))

The guessed center is at (734, 1181)
The center is at (753.703500152, 1186.18798503)
The X vector is is at (97.7197301826, -0.787977117653)
The Y vector is is at (1.38218579497, 97.0826990758)

print("X has an angle of %s deg"%rad2deg(arctan2(Xy, Xx)))
print("Y has an angle of %s deg"%rad2deg(arctan2(Yy, Yx)))
print("The XY angle is %s deg"%rad2deg(arctan2(Yy, Yx)-arctan2(Xy, Xx)))

X has an angle of -0.462002756355 deg
Y has an angle of 89.1843236418 deg
The XY angle is 89.6463263982 deg

x = point[:]["center_x"] - Ox
y = point[:]["center_y"] - Oy
xy = numpy.vstack((x,y))
R = numpy.array([[Xx,Yx],[Xy,Yy]])
iR = numpy.linalg.inv(R)
IJ = dot(iR,xy).T

Xmin = IJ[:,0].min()
Xmax = IJ[:,0].max()
Ymin = IJ[:,1].min()
Ymax = IJ[:,1].max()
print("Xmin/max", Xmin, Xmax)
print("Ymin/max", Ymin, Ymax)
print("Maximum error versus integrer: %s * pitch size (5mm)"%(abs(IJ-IJ.round()).max()))
```

```
Xmin/max -6.07394212848 12.060721056
Ymin/max -11.0890545732 7.04060363671
Maximum error versus integrer: 0.117211354675 * pitch size (5mm)
```

At this point it is important to check the correct rounding to integers: The maximum error should definitely be better than 0.2*pitch ! If not, try to change the origin (Cx and Cy). This criteria will be used for the optimization later on.

```
plot(IJ[:,0],IJ[:,1],"or")
idx = numpy.round(IJ).astype(int)
plot(idx[:,0],IJ[:,1],"og")
xlim(floor(Xmin), ceil(Xmax))
ylim(floor(Ymin), ceil(Ymax))
title("Red: measured peaks, Green: Expected position")
```

```
<matplotlib.text.Text at 0x7f2c4e313278>
```



Estimation of the pixel size:

The pixel size is obtained from the pitch of the grid, in vectorial:

$$pitch^2 = (Px \cdot Xx)^2 + (Py \cdot Xy)^2$$

$$pitch^2 = (Px \cdot Yx)^2 + (Py \cdot Yy)^2$$

```
pitch = 5e-3 #mm distance between holes
Py = pitch*sqrt((Yx**2-Xx**2)/((Xy*Yx)**2-(Xx*Yy)**2))
Px = sqrt((pitch**2-(Xy*Py)**2)/Xx**2)
print("Pixel size in average: x:%.3f micron, y: %.3f microns"%(Px*1e6, Py*1e6))
```

```
Pixel size in average: x:51.165 micron, y: 51.497 microns
```

At this stage, we have:

- A list of control points placed on a regular grid with a sub-pixel precision

- The center of the image, located on a control point
- the average X and Y vector to go from one control point to another

3.7.4 Optimization of the pixel position

The optimization is obtained by minimizing the mis-placement of the control points on the regular grid. For a larger coverage we include now the peaks on the border with less than 4 neighbours.

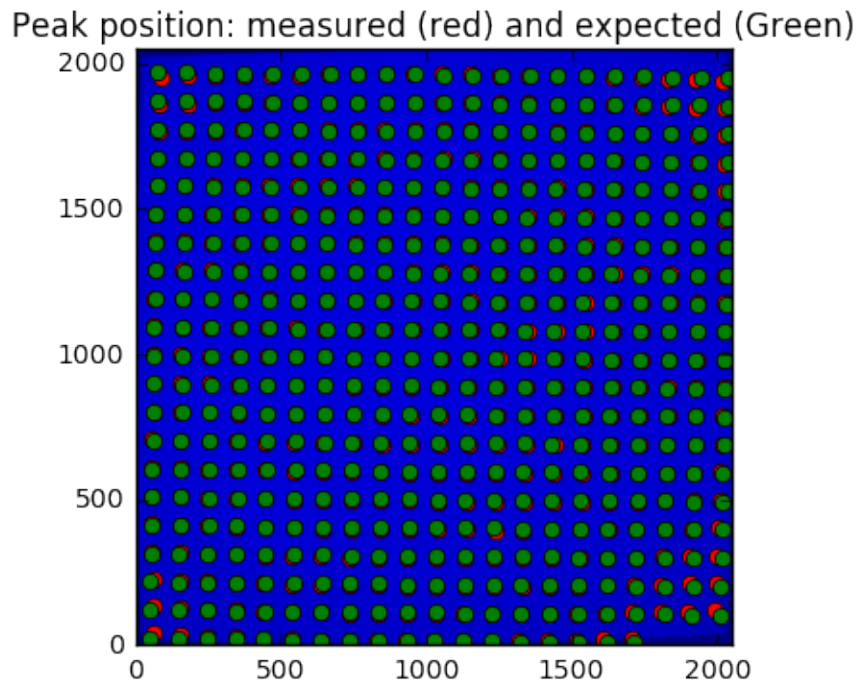
```
#Measured peaks (all!), needs to flip x<->y
peaks_m = numpy.empty_like(peaks_ref)
peaks_m[:,1] = peaks_ref[:,0]
peaks_m[:,0] = peaks_ref[:,1]

#parameter set for optimization:
P0 = [Ox, Oy, Xx, Yx, Xy, Yy]

P = numpy.array(P0)

def to_hole(P, pixels):
    "Translate pixel -> hole"
    T = numpy.atleast_2d(P[:2])
    R = P[2:].reshape((2,2))
    #Transformation matrix from pixel to holes:
    hole = dot(numpy.linalg.inv(R), (pixels - T).T).T
    return hole

def to_pix(P, holes):
    "Translate hole -> pixel"
    T = numpy.atleast_2d(P[:2])
    R = P[2:].reshape((2,2))
    #Transformation from index points (holes) to pixel coordinates:
    pix = dot(R,holes.T).T + T
    return pix

def error(P):
    "Error function"
    hole_float = to_hole(P, peaks_m)
    hole_int = hole_float.round()
    delta = hole_float-hole_int
    delta2 = (delta**2).sum()
    return delta2

print("Total inital error ", error(P), P0)
holes = to_hole(P, peaks_m)
print("Maximum initial error versus integrer: %s * pitch size (5mm)"%(abs(holes-holes.round()).ma
from scipy.optimize import minimize
res = minimize(error, P)
print(res)
print("total Final error ", error(res.x),res.x)
holes = to_hole(res.x, peaks_m)
print("Maximum final error versus integrer: %s * pitch size (5mm)"%(abs(holes-holes.round()).max(
```

```
Total inital error  2.5995763607 [753.70350015163422, 1186.1879850327969, 97.719730182623479, 1.38
Maximum initial error versus integrer: 0.199838456433 * pitch size (5mm)
      fun: 2.123772842169884
 hess_inv: array([[  1.41698853e+01,   5.02981780e-01,  -8.67450996e-01,
          5.65400698e-01,  -2.23588556e-02,   3.62469793e-02],
       [  5.02981780e-01,   1.44432486e+01,  -6.17043562e-03,
          3.18737250e-02,  -8.80159842e-01,   5.53478243e-01],
       [ -8.67450996e-01,  -6.17043562e-03,   2.99705132e-01,
         -4.12312169e-03,   2.39113093e-03,  -1.79968692e-03],
```

```
       [  5.65400698e-01,   3.18737250e-02,  -4.12312169e-03,
          3.01702833e-01,  -1.78715958e-03,   3.83867286e-03],
       [ -2.23588556e-02,  -8.80159842e-01,   2.39113093e-03,
         -1.78715958e-03,   2.97818929e-01,  -3.46536500e-03],
       [  3.62469793e-02,   5.53478243e-01,  -1.79968692e-03,
          3.83867286e-03,  -3.46536500e-03,   2.93190623e-01]])
     jac: array([ -2.98023224e-08,   5.66244125e-07,   1.19209290e-07,
         3.57627869e-07,   8.34465027e-07,   1.10268593e-06])
 message: 'Optimization terminated successfully.'
    nfev: 160
     nit: 15
    njev: 20
  status: 0
 success: True
       x: array([  7.53021133e+02,   1.18519693e+03,   9.81143528e+01,
        1.47509462e+00,  -8.04478941e-01,   9.73166902e+01])
total Final error  2.12377284217 [  7.53021133e+02   1.18519693e+03   9.81143528e+01   1.47509462e
  -8.04478941e-01   9.73166902e+01]
Maximum final error versus integrer: 0.234645015537 * pitch size (5mm)
```

```
clf()
peaks_c = to_pix(res.x,to_hole(res.x,peaks_m).round())
imshow(img, interpolation="nearest", origin="lower")
plot(peaks_m[:,0],peaks_m[:, 1], "or")
plot(peaks_c[:,0], peaks_c[:, 1], "og")
xlim(0,2048)
ylim(0,2048)
title("Peak position: measured (red) and expected (Green)")
```

```
<matplotlib.text.Text at 0x7f2c4e3de668>
```




```
pitch = 5e-3 #mm distance between holes
Ox, Oy, Xx, Yx, Xy, Yy = res.x
Py = pitch*sqrt((Yx**2-Xx**2)/((Xy*Yx)**2-(Xx*Yy)**2))
Px = sqrt((pitch**2-(Xy*Py)**2)/Xx**2)
print("Optimized pixel size in average: x:%.3f micron, y: %.3f microns"%(Px*1e6, Py*1e6))
```

```
Optimized pixel size in average: x:50.959 micron, y: 51.373 microns
```

Few comments:

- The maximum error grow during optimization without explanations
- The outer part of the detector is the most distorted

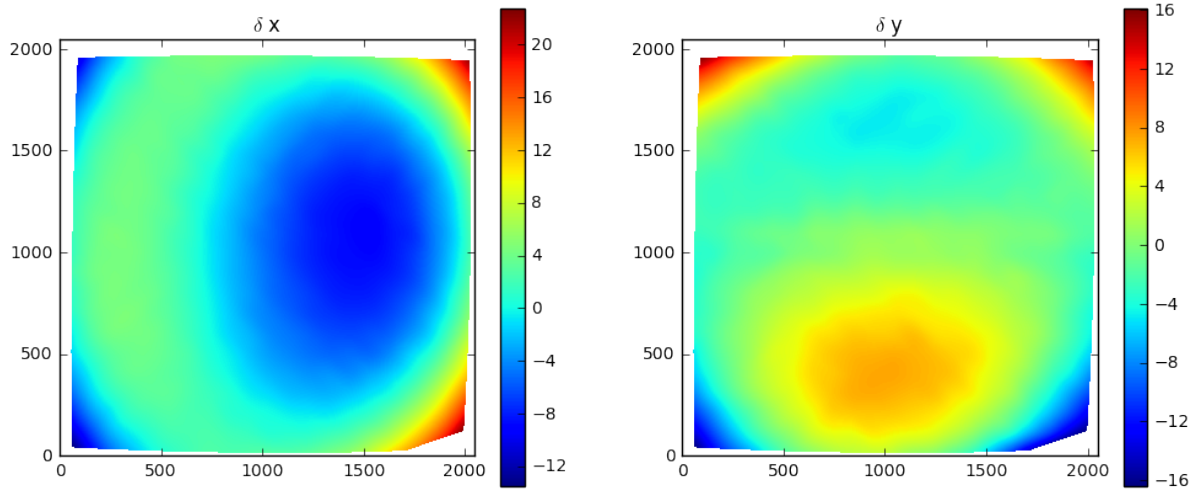
3.7.5 Interpolation of the fitted data

Multivariate data interpolation (griddata)

Correction arrays are built slightly larger (+1) to be able to manipulate corners instead of centers of pixels As coordinates are needed as y,x (and not x,y) we use p instead of peaks_m

```
from scipy.interpolate import griddata
grid_x, grid_y = np.mgrid[0:img.shape[0]+1, 0:img.shape[1]+1]
delta = peaks_c - peaks_m
#we use peaks_res instead of peaks_m to be in y,x coordinates, not x,y
delta_x = griddata(peaks_ref, delta[:,0], (grid_x, grid_y), method='cubic')
delta_y = griddata(peaks_ref, delta[:,1], (grid_x, grid_y), method='cubic')

figure(figsize=(12,5))
subplot(1,2,1)
imshow(delta_x,origin="lower", interpolation="nearest")
title(r"$\delta$ x")
colorbar()
subplot(1,2,2)
imshow(delta_y, origin="lower", interpolation="nearest")
title(r"$\delta$ y")
colorbar()
#Nota: the arrays are filled with "NaN" outside the convex Hull
```

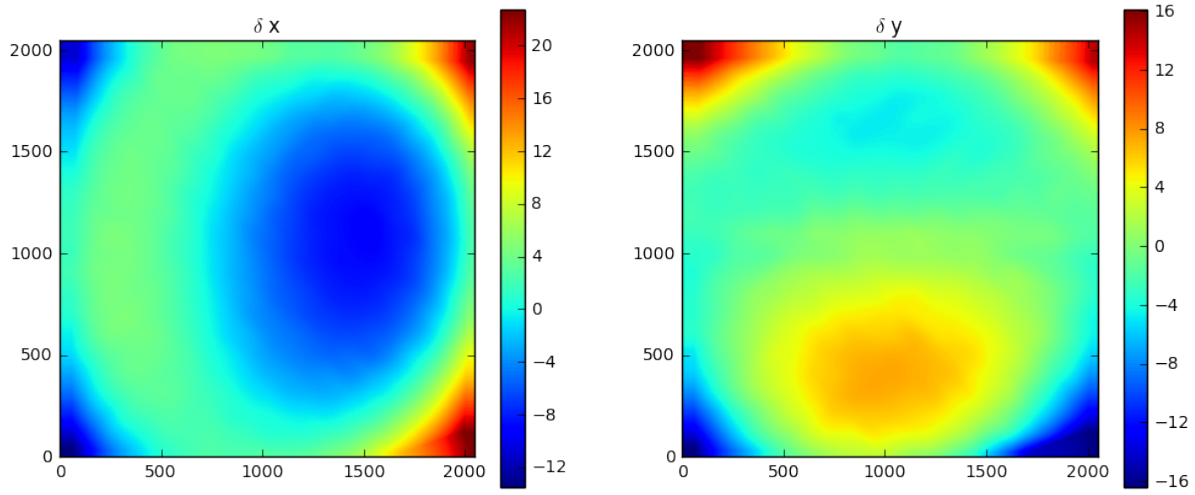
```
<matplotlib.colorbar.Colorbar at 0x7f2c47ed3d30>
```




```
#From http://stackoverflow.com/questions/3662361/fill-in-missing-values-with-nearest-neighbour-in
def fill(data, invalid=None):
    """
    Replace the value of invalid 'data' cells (indicated by 'invalid')
    by the value of the nearest valid data cell

    Input:
        data:    numpy array of any dimension
        invalid: a binary array of same shape as 'data'. True cells set where data
```

```
                value should be replaced.
                If None (default), use: invalid  = np.isnan(data)

    Output:
        Return a filled array.
    """

    if invalid is None:
        invalid = numpy.isnan(data)

    ind = ndimage.distance_transform_edt(invalid, return_distances=False, return_indices=True)
    return data[tuple(ind)]

figure(figsize=(12,5))
subplot(1,2,1)
imshow(fill(delta_x),origin="lower", interpolation="nearest")
title(r"$\delta$ x")
colorbar()
subplot(1,2,2)
imshow(fill(delta_y), origin="lower", interpolation="nearest")
title(r"$\delta$ y")
colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7f2c46c55f98>
```



It is important to understand the extrapolation outside the convex hull has no justification, it is there just to prevent numerical bugs.

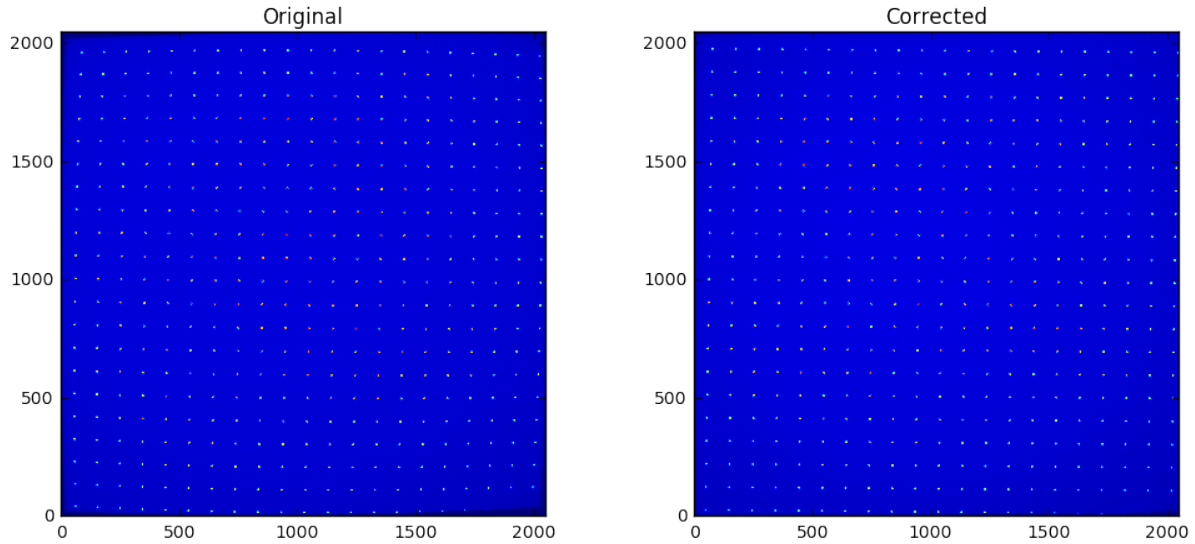
3.7.6 Saving the distortion correction arrays to a detector

```
from pyFAI.detectors import Detector
detector = Detector(Py,Px)
detector.max_shape = detector.shape = img.shape
detector.set_dx(fill(delta_x))
detector.set_dy(fill(delta_y))
detector.mask = numpy.isnan(delta_x).astype(numpy.int8)[:img.shape[0], :img.shape[1]]
detector.save("testdetector.h5")
```

3.7.7 Validation of the distortion correction

```
from pyFAI.distortion import Distortion
dis = Distortion(detector)
```

```
cor = dis.correct(img)
figure(figsize=(12,5))
subplot(1,2,1)
imshow(img, interpolation="nearest", origin="lower")
title("Original")
subplot(1,2,2)
imshow(cor, origin="lower", interpolation="nearest")
title("Corrected")
fabio.edfimage.EdfImage(data=cor).save("corrected.edf")
```



3.8 Conclusion

This procedure describes how to measure the detector distortion and how to create a detector description file directly usable in pyFAI. Only the region inside the convex hull of the grid data-points is valid and the region of the detector which is not calibrated has been masked out to prevent accidental use of it.

The distortion corrected image can now be used to check how "good" the calibration actually is. This file can be injected in the third cell, and follow the same procedure (left as exercise). This gives a maximum mis-placement of 0.003, the average error is then of 0.0006 and correction-map exhibit a displacement of pixels in the range +/- 0.2 pixels which is acceptable and validates the whole procedure.

CHAPTER

FOUR

PYFAI SCRIPTS MANUAL

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

Pre-processing tools:

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

Calibration tools:

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

Azimuthal integration tools:

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

4.1 Preprocessing tool: pyFAI-average

4.1.1 Purpose

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

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

4.1.2 Options:

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

positional arguments: FILE Files to be processed

optional arguments:

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

-V, --version show program's version number and exit

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

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

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

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

-d DARK, --dark DARK Dark noise to be subtracted

-f FLAT, --flat FLAT Flat field correction

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

-q QUANTILES, --quantiles QUANTILES average out between two quantiles -q 0.20-0.90

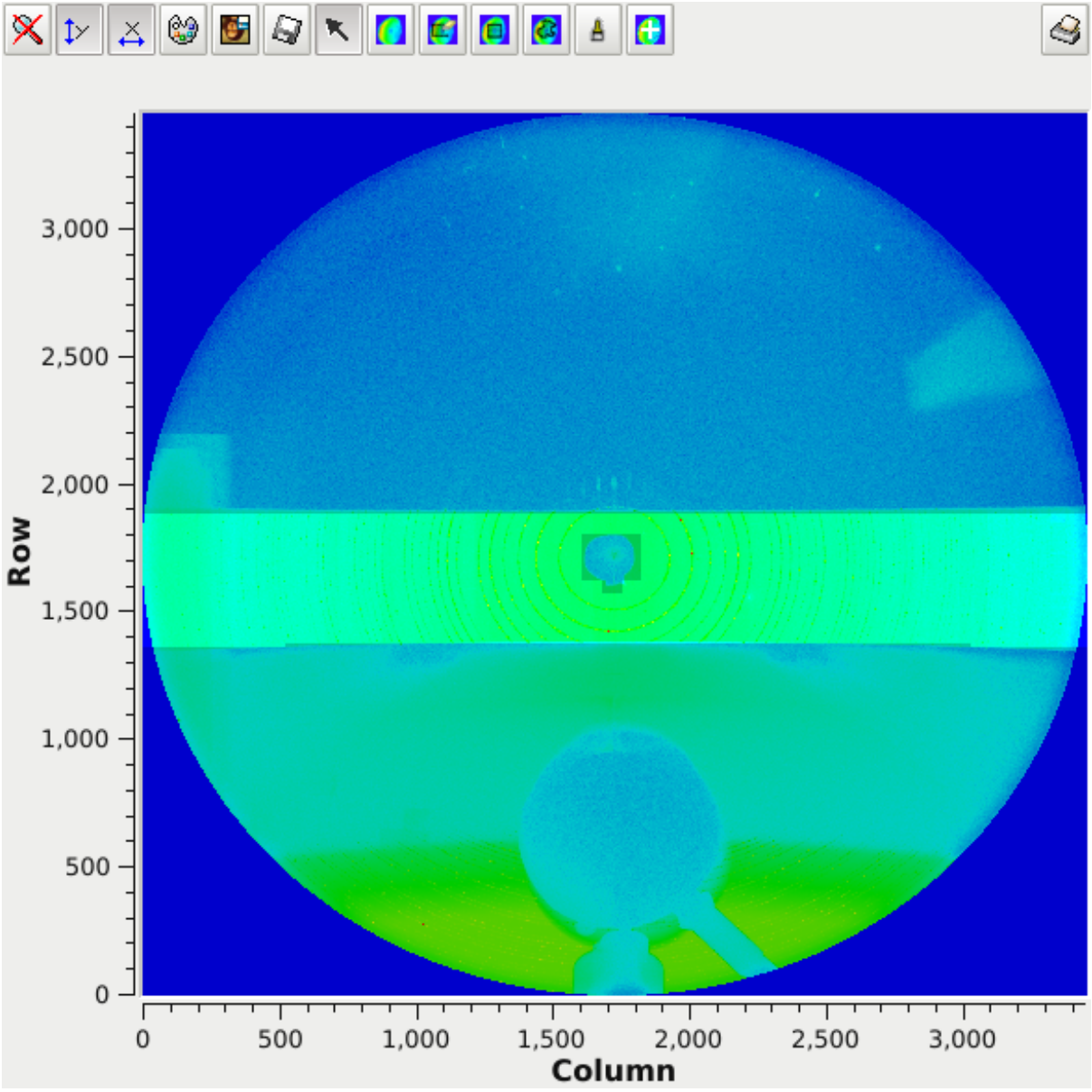
```
$ pyFAI-average --help
usage: pyFAI-average [options] [options] -o output.edf file1.edf file2.edf ...

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)

positional arguments:
  FILE                  Files to be processed

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -o OUTPUT, --output OUTPUT
                        Output/ destination of average image
  -m METHOD, --method METHOD
                        Method used for averaging, can be 'mean' (default) or
                        'min', 'max', 'median', 'sum', 'quantiles' , 'cutoff',
                        'std'. Multiple filters can be defined with ','
                        separator.
  -c CUTOFF, --cutoff CUTOFF
                        Take the mean of the average +/- cutoff * std_dev.
  -F FORMAT, --format FORMAT
                        Output file/image format (by default EDF)
  -d DARK, --dark DARK  Dark noise to be subtracted
  -f FLAT, --flat FLAT  Flat field correction
  -v, --verbose         switch to verbose/debug mode
  -q QUANTILES, --quantiles QUANTILES
                        average out between two quantiles -q 0.20-0.90
  --monitor-name MONITOR_KEY
                        Name of the monitor in the header of each input files.
                        If defined the contribution of each input file is
                        divided by the monitor. If the header does not contain
                        or contains a wrong value, the contribution of the
                        input file is ignored. On EDF files, values from
                        'counter_pos' can accessed by using the expected
                        mnemonic. For example 'counter/bmon'.
  --quiet               Only error messages are printed out

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.
```

4.2 Mask generation tool: pyFAI-drawmask

4.2.1 Purpose

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



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

Usage: pyFAI-drawmask [options] file1.edf file2.edf ...

4.2.2 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)

```
$ pyFAI-drawmask --help
usage: pyFAI-drawmask file1.edf file2.edf ...

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
window and let you draw on the first image (provided) with different tools
(brush, rectangle selection...) When you are finished, click on the "Save and
quit" button.

positional arguments:
  FILE           Files to be processed

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  show program's version number and exit

The mask image is saved into file1-masked.edf. Optionally the script will
print the number of pixel masked and the intensity masked (as well on other
files provided in input)
```

4.3 Preprocessing tool: detector2nexus

Convert a complex detector definition (multiple modules, possibly in 3D) into a single NeXus detector definition together with the mask (and much more in the future)

4.3.1 Purpose

Convert a detector to NeXus detector definition for pyFAI.

4.3.2 Usage:

detector2nexus [options] [options] -o nxs.h5

4.3.3 Options:

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

-V, --version show program's version number and exit

-o OUTPUT, --output OUTPUT Output nexus file, unless detector_name.h5

-n NAME, --name NAME name of the detector

-m MASK, --mask MASK mask corresponding to the detector

-D DETECTOR, --detector DETECTOR Base detector name (see documentation of pyFAI.detectors

-s SPLINEFILE, --splinefile SPLINEFILE Geometric distortion file from FIT2D

-dx DX, –x-corr DX Geometric correction for pilatus -dy DY, –y-corr DY Geometric correction for pilatus -p PIXEL, –pixel PIXEL

pixel size (comma separated): x,y

-S SHAPE, --shape SHAPE shape of the detector (comma separated): x,y

-d DARK, --dark DARK Dark noise to be subtracted

-f FLAT, --flat FLAT Flat field correction

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

```
$ detector2nexus --help
usage: detector2nexus [options] [options] -o nxs.h5

Convert a complex detector definition (multiple modules, possibly in 3D) into
a single NeXus detector definition together with the mask (and much more in
the future)

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -o OUTPUT, --output OUTPUT
                        Output nexus file, unless detector_name.h5
  -n NAME, --name NAME  name of the detector
  -m MASK, --mask MASK  mask corresponding to the detector
  -D DETECTOR, --detector DETECTOR
                        Base detector name (see documentation of
                        pyFAI.detectors
  -s SPLINEFILE, --splinefile SPLINEFILE
                        Geometric distortion file from FIT2D
  -dx DX, --x-corr DX   Geometric correction for pilatus
  -dy DY, --y-corr DY   Geometric correction for pilatus
  -p PIXEL, --pixel PIXEL
                        pixel size (comma separated): x,y
  -S SHAPE, --shape SHAPE
                        shape of the detector (comma separated): x,y
  -d DARK, --dark DARK  Dark noise to be subtracted
  -f FLAT, --flat FLAT  Flat field correction
  -v, --verbose         switch to verbose/debug mode

This summarizes detector2nexus
```

4.4 Calibration tool: pyFAI-calib

4.4.1 Purpose

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images without a priori knowledge of your setup. You will need to provide a calibrant or a “d-spacing” file containing the spacing of Miller plans in Angstrom (in decreasing order).

If you are using a standard calibrant, look at https://github.com/kif/pyFAI/tree/master/calibration or search in the American Mineralogist database: [AMD] or in the [COD]. The –calibrant option is mandatory !

Calibrants available: Ni, CrOx, NaCl, Si_SRM640e, Si_SRM640d, Si_SRM640a, Si_SRM640b, Cr2O3, AgBh, Si_SRM640, CuO, PBBA, alpha_Al2O3, SI_SRM640c, quartz, C14H30O, cristobaltite, Si, LaB6, CeO2, LaB6_SRM660a, LaB6_SRM660b, LaB6_SRM660c, TiO2, ZnO, Al, Au

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 or
- the NeXus file describing the distortion

Many option are available among those:

- dark-current / flat field corrections
- Masking of bad regions
- reconstruction of missing region (module based detectors), see option -r
- 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)

4.4.2 Usage:

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

4.4.3 Options:

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

-V, --version show program's version number and exit

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

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

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

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

-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

-i FILE, --poni FILE file containing the diffraction parameter (poni-file). MANDATORY for pyFAI-recalib!

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

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

-f FLAT, --flat FLAT list of comma separated 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. Default: 100mm

--dist DIST sample-detector distance in meter. Default: 0.1m

--poni1 PONI1 poni1 coordinate in meter. Default: center of detector

--poni2 PONI2 poni2 coordinate in meter. Default: center of detector

--rot1 ROT1 rot1 in radians. default: 0

--rot2 ROT2 rot2 in radians. default: 0

--rot3 ROT3 rot3 in radians. default: 0

--fix-dist fix the distance parameter

--free-dist free the distance parameter. Default: Activated

--fix-poni1 fix the poni1 parameter

--free-poni1 free the poni1 parameter. Default: Activated

--fix-poni2 fix the poni2 parameter

--free-poni2 free the poni2 parameter. Default: Activated

--fix-rot1 fix the rot1 parameter

--free-rot1 free the rot1 parameter. Default: Activated

--fix-rot2 fix the rot2 parameter

--free-rot2 free the rot2 parameter. Default: Activated

--fix-rot3 fix the rot3 parameter

--free-rot3 free the rot3 parameter. Default: Activated

--fix-wavelength fix the wavelength parameter. Default: Activated

--free-wavelength free the wavelength parameter. Default: Deactivated

--tilt Allow initially detector tilt to be refined (rot1, rot2, rot3). Default: Activated

--no-tilt Deactivated tilt refinement and set all rotation to 0

--saturation SATURATION consider all pixel>max*(1-saturation) as saturated and reconstruct them, default: 0 (deactivated)

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

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

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

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

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

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

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

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

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

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

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

4.4.4 Tips & Tricks

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

4.4.5 Example of usage:

```
$ pyFAI-calib --help
usage: pyFAI-calib [options] -w 1 -D detector -c calibrant.D imagefile.edf

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images
without a priori knowledge of your setup. You will need to provide a calibrant
or a "d-spacing" file containing the spacing of Miller plans in Angstrom (in
decreasing order). Calibrants available: Ni, CrOx, NaCl, Si_SRM640e,
Si_SRM640d, Si_SRM640a, Si_SRM640c, alpha_Al2O3, Cr2O3, AgBh, Si_SRM640, CuO,
PBBA, Si_SRM640b, mock, quartz, C14H30O, cristobaltite, Si, LaB6, CeO2,
LaB6_SRM660a, LaB6_SRM660b, LaB6_SRM660c, TiO2, ZnO, Al, Au or search in the
American Mineralogist database: http://rruff.geo.arizona.edu/AMS/amcsd.php The
--calibrant option is mandatory !

positional arguments:
  FILE                  List of files to calibrate

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -o FILE, --out FILE   Filename where processed image is saved
  -v, --verbose         switch to debug/verbose mode
  -c FILE, --calibrant FILE
                        Calibrant name or file containing d-spacing of the
                        reference sample (MANDATORY, case sensitive !)
  -w WAVELENGTH, --wavelength WAVELENGTH
                        wavelength of the X-Ray beam in Angstrom. Mandatory
  -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
  -i FILE, --poni FILE  file containing the diffraction parameter (poni-file).
                        MANDATORY for pyFAI-recalib!
  -b BACKGROUND, --background BACKGROUND
                        Automatic background subtraction if no value are
                        provided
```

```
-d DARK, --dark DARK  list of comma separated dark images to average and
                      subtract
-f FLAT, --flat FLAT  list of comma separated 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. Default: 100mm
--dist DIST           sample-detector distance in meter. Default: 0.1m
--poni1 PONI1         poni1 coordinate in meter. Default: center of detector
--poni2 PONI2         poni2 coordinate in meter. Default: center of detector
--rot1 ROT1           rot1 in radians. default: 0
--rot2 ROT2           rot2 in radians. default: 0
--rot3 ROT3           rot3 in radians. default: 0
--fix-dist            fix the distance parameter
--free-dist           free the distance parameter. Default: Activated
--fix-poni1           fix the poni1 parameter
--free-poni1          free the poni1 parameter. Default: Activated
--fix-poni2           fix the poni2 parameter
--free-poni2          free the poni2 parameter. Default: Activated
--fix-rot1            fix the rot1 parameter
--free-rot1           free the rot1 parameter. Default: Activated
--fix-rot2            fix the rot2 parameter
--free-rot2           free the rot2 parameter. Default: Activated
--fix-rot3            fix the rot3 parameter
--free-rot3           free the rot3 parameter. Default: Activated
--fix-wavelength      fix the wavelength parameter. Default: Activated
--free-wavelength     free the wavelength parameter. Default: Deactivated
--tilt                Allow initially detector tilt to be refined (rot1,
                      rot2, rot3). Default: Activated
--no-tilt             Deactivated tilt refinement and set all rotation to 0
--saturation SATURATION
                      consider all pixel>max*(1-saturation) as saturated and
                      reconstruct them, default: 0 (deactivated)
--weighted            weight fit by intensity, by default not.
--npt NPT_1D          Number of point in 1D integrated pattern, Default:
                      1024
--npt-azim NPT_2D_AZIM
                      Number of azimuthal sectors in 2D integrated images.
                      Default: 360
--npt-rad NPT_2D_RAD  Number of radial bins in 2D integrated images.
                      Default: 400
--unit UNIT           Valid units for radial range: 2th_deg, 2th_rad,
                      q_nm^-1, q_A^-1, r_mm. Default: 2th_deg
--no-gui              force the program to run without a Graphical interface
--no-interactive      force the program to run and exit without prompting
                      for refinements
-r, --reconstruct     Reconstruct image where data are masked or <0 (for
                      Pilatus detectors or detectors with modules)
-g GAUSSIAN, --gaussian GAUSSIAN
                      Size of the gaussian kernel. Size of the gap (in
                      pixels) between two consecutive rings, by default 100
                      Increase the value if the arc is not complete;
                      decrease the value if arcs are mixed together.
--square              Use square kernel shape for neighbor search instead of
                      diamond shape
-p PIXEL, --pixel PIXEL
                      size of the pixel in micron
```

```
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)
```

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

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

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

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

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

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

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

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

Downloaded test images are located in tests/testimages

4.5 Calibration tool: pyFAI-recalib

pyFAI-recalib is now obsolete. All feature provided by it are now available as part of pyFAI-calib.

```
$ pyFAI-recalib --help
usage: pyFAI-recalib [options] -i ponifile -w 1 -c calibrant.D imagefile.edf

Calibrate the diffraction setup geometry based on Debye-Sherrer rings images
with a priori knowledge of your setup (an input PONI-file). You will need to
provide a calibrant or a "d-spacing" file containing the spacing of Miller
plans in Angstrom (in decreasing order). Calibrants available: Ni, CrOx, NaCl,
Si_SRM640e, Si_SRM640d, Si_SRM640a, Si_SRM640c, alpha_Al2O3, Cr2O3, AgBh,
Si_SRM640, CuO, PBBA, Si_SRM640b, mock, quartz, C14H30O, cristobaltite, Si,
LaB6, CeO2, LaB6_SRM660a, LaB6_SRM660b, LaB6_SRM660c, TiO2, ZnO, Al, Au or
search in the American Mineralogist database:
http://rruff.geo.arizona.edu/AMS/amcsd.php The --calibrant option is mandatory
!

positional arguments:
  FILE                  List of files to calibrate

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -o FILE, --out FILE   Filename where processed image is saved
  -v, --verbose         switch to debug/verbose mode
  -c FILE, --calibrant FILE
                        Calibrant name or file containing d-spacing of the
                        reference sample (MANDATORY, case sensitive !)
  -w WAVELENGTH, --wavelength WAVELENGTH
                        wavelength of the X-Ray beam in Angstrom. Mandatory
  -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
-i FILE, --poni FILE  file containing the diffraction parameter (poni-file).
                      MANDATORY for pyFAI-recalib!
-b BACKGROUND, --background BACKGROUND
                      Automatic background subtraction if no value are
                      provided
-d DARK, --dark DARK  list of comma separated dark images to average and
                      subtract
-f FLAT, --flat FLAT  list of comma separated 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. Default: 100mm
--dist DIST           sample-detector distance in meter. Default: 0.1m
--poni1 PONI1         poni1 coordinate in meter. Default: center of detector
--poni2 PONI2         poni2 coordinate in meter. Default: center of detector
--rot1 ROT1           rot1 in radians. default: 0
--rot2 ROT2           rot2 in radians. default: 0
--rot3 ROT3           rot3 in radians. default: 0
--fix-dist            fix the distance parameter
--free-dist           free the distance parameter. Default: Activated
--fix-poni1           fix the poni1 parameter
--free-poni1          free the poni1 parameter. Default: Activated
--fix-poni2           fix the poni2 parameter
--free-poni2          free the poni2 parameter. Default: Activated
--fix-rot1            fix the rot1 parameter
--free-rot1           free the rot1 parameter. Default: Activated
--fix-rot2            fix the rot2 parameter
--free-rot2           free the rot2 parameter. Default: Activated
--fix-rot3            fix the rot3 parameter
--free-rot3           free the rot3 parameter. Default: Activated
--fix-wavelength      fix the wavelength parameter. Default: Activated
--free-wavelength     free the wavelength parameter. Default: Deactivated
--tilt                Allow initially detector tilt to be refined (rot1,
                      rot2, rot3). Default: Activated
--no-tilt             Deactivated tilt refinement and set all rotation to 0
--saturation SATURATION
                      consider all pixel>max*(1-saturation) as saturated and
                      reconstruct them, default: 0 (deactivated)
--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
  -k, --keep            Keep existing control point and append new

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 and --no-
interactive options). Note that `pyFAI-recalib` program is obsolete as the
same functionality is available from within pyFAI-calib, using the `recalib`
command in the refinement process. Two option are available for recalib: the
numbe of rings to extract (similar to the -r option of this program) and a new
option which lets you choose between the original `massif` algorithm and newer
ones like `blob` and `watershed` detection.
```

4.6 Calibration tool: check_calib

4.6.1 Purpose

Check_calib is a deprecated tool aiming at validating both the geometric calibration and everything else like flat-field correction, distortion correction, at a sub-pixel level. Pleas use the *validate*, *validate2* and *chiplot* commands in pyFAI-calib, during the refinement process to obtain the same output with more options.

```
$ check_calib --help
usage: check_calib [options] -p param.poni image.edf

Check_calib is a research tool aiming at validating both the geometric
calibration and everything else like flat-field correction, distortion
correction, at a sub-pixel level. Note that `check_calib` program is obsolete
as the same functionality is available from within pyFAI-calib, using the
`validate` command in the refinement process. :returns: True if the parsing
succeed, else False

positional arguments:
  FILE                  Image file to check calibration for

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -v, --verbose         switch to debug mode
  -d FILE, --dark FILE  file containing the dark images to subtract
  -f FILE, --flat FILE  file containing the flat images to divide
  -m FILE, --mask FILE  file containing the mask
  -p FILE, --poni FILE  file containing the diffraction parameter (poni-file)
  -e ENERGY, --energy ENERGY
                        energy of the X-Ray beam in keV (hc=12.398419292keV.A)
  -w WAVELENGTH, --wavelength WAVELENGTH
                        wavelength of the X-Ray beam in Angstrom
```

4.7 Calibration tool: MX-calibrate

4.7.1 Purpose

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

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

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

4.7.2 Usage:

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

4.7.3 Options:

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

Calibrate automatically a set of frames taken at various sample-detector distance. Return the linear regression of the fit in funtion of the sample- setector distance.

positional arguments: FILE List of files to calibrate

optional arguments:

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

-V, --version show program's version number and exit

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

--weighted weight fit by intensity

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

--tilt	Allow initially detector tilt to be refined (rot1, rot2, rot3). Default: Activated
--no-tilt	Deactivated tilt refinement and set all rotation to 0
--dist DIST	sample-detector distance in meter
--poni1 PONI1	poni1 coordinate in meter
--poni2 PONI2	poni2 coordinate in meter
--rot1 ROT1	rot1 in radians
--rot2 ROT2	rot2 in radians
--rot3 ROT3	rot3 in radians
--fix-dist	fix the distance parameter
--free-dist	free the distance parameter
--fix-poni1	fix the poni1 parameter
--free-poni1	free the poni1 parameter
--fix-poni2	fix the poni2 parameter
--free-poni2	free the poni2 parameter
--fix-rot1	fix the rot1 parameter
--free-rot1	free the rot1 parameter
--fix-rot2	fix the rot2 parameter
--free-rot2	free the rot2 parameter
--fix-rot3	fix the rot3 parameter
--free-rot3	free the rot3 parameter
--fix-wavelength	fix the wavelength parameter
--free-wavelength	free the wavelength parameter
--no-gui	force the program to run without a Graphical interface
--gui	force the program to run with a Graphical interface
--no-interactive	force the program to run and exit without prompting for refinements
--interactive	force the program to prompt for refinements
--peak-picker PEAKPICKER	Uses the 'massif', 'blob' or 'watershed' peak-picker algorithm (default: blob)

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

4.7.4 Example:

```
$ MX-calibrate --help
usage: MX-Calibrate -w 1.54 -c CeO2 file1.cbf file2.cbf ...

Calibrate automatically a set of frames taken at various sample-detector
distance. Return the linear regression of the fit in funtion of the sample-
setector distance.

positional arguments:
  FILE                  List of files to calibrate
```

```
optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -v, --verbose         switch to debug/verbose mode
  -c FILE, --calibrant FILE
                        file containing d-spacing of the calibrant reference
                        sample (MANDATORY)
  -w WAVELENGTH, --wavelength WAVELENGTH
                        wavelength of the X-Ray beam in Angstrom
  -e ENERGY, --energy ENERGY
                        energy of the X-Ray beam in keV (hc=12.398419292keV.A)
  -P POLARIZATION_FACTOR, --polarization POLARIZATION_FACTOR
                        polarization factor, from -1 (vertical) to +1
                        (horizontal), default is 0, synchrotrons are around
                        0.95
  -b BACKGROUND, --background BACKGROUND
                        Automatic background subtraction if no value are
                        provided
  -d DARK, --dark DARK  list of dark images to average and subtract
  -f FLAT, --flat FLAT  list of flat images to average and divide
  -s SPLINE, --spline SPLINE
                        spline file describing the detector distortion
  -p PIXEL, --pixel PIXEL
                        size of the pixel in micron
  -D DETECTOR_NAME, --detector DETECTOR_NAME
                        Detector name (instead of pixel size+spline)
  -m MASK, --mask MASK  file containing the mask (for image reconstruction)
  --filter FILTER       select the filter, either mean(default), max or median
  --saturation SATURATION
                        consider all pixel>max*(1-saturation) as saturated and
                        reconstruct them
  -r MAX_RINGS, --ring MAX_RINGS
                        maximum number of rings to extract
  --weighted            weight fit by intensity
  -l DISTANCE, --distance DISTANCE
                        sample-detector distance in millimeter
  --tilt                Allow initially detector tilt to be refined (rot1,
                        rot2, rot3). Default: Activated
  --no-tilt             Deactivated tilt refinement and set all rotation to 0
  --dist DIST           sample-detector distance in meter
  --poni1 PONI1         poni1 coordinate in meter
  --poni2 PONI2         poni2 coordinate in meter
  --rot1 ROT1           rot1 in radians
  --rot2 ROT2           rot2 in radians
  --rot3 ROT3           rot3 in radians
  --fix-dist            fix the distance parameter
  --free-dist           free the distance parameter
  --fix-poni1           fix the poni1 parameter
  --free-poni1          free the poni1 parameter
  --fix-poni2           fix the poni2 parameter
  --free-poni2          free the poni2 parameter
  --fix-rot1            fix the rot1 parameter
  --free-rot1           free the rot1 parameter
  --fix-rot2            fix the rot2 parameter
  --free-rot2           free the rot2 parameter
  --fix-rot3            fix the rot3 parameter
  --free-rot3           free the rot3 parameter
  --fix-wavelength      fix the wavelength parameter
  --free-wavelength     free the wavelength parameter
  --no-gui              force the program to run without a Graphical interface
  --gui                 force the program to run with a Graphical interface
  --no-interactive      force the program to run and exit without prompting
```

```
                        for refinements
  --interactive         force the program to prompt for refinements
  --peak-picker PEAKPICKER
                        Uses the 'massif', 'blob' or 'watershed' peak-picker
                        algorithm (default: blob)

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

4.8 Integration tool: pyFAI-integrate

4.8.1 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.

4.8.2 Usage

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

4.8.3 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

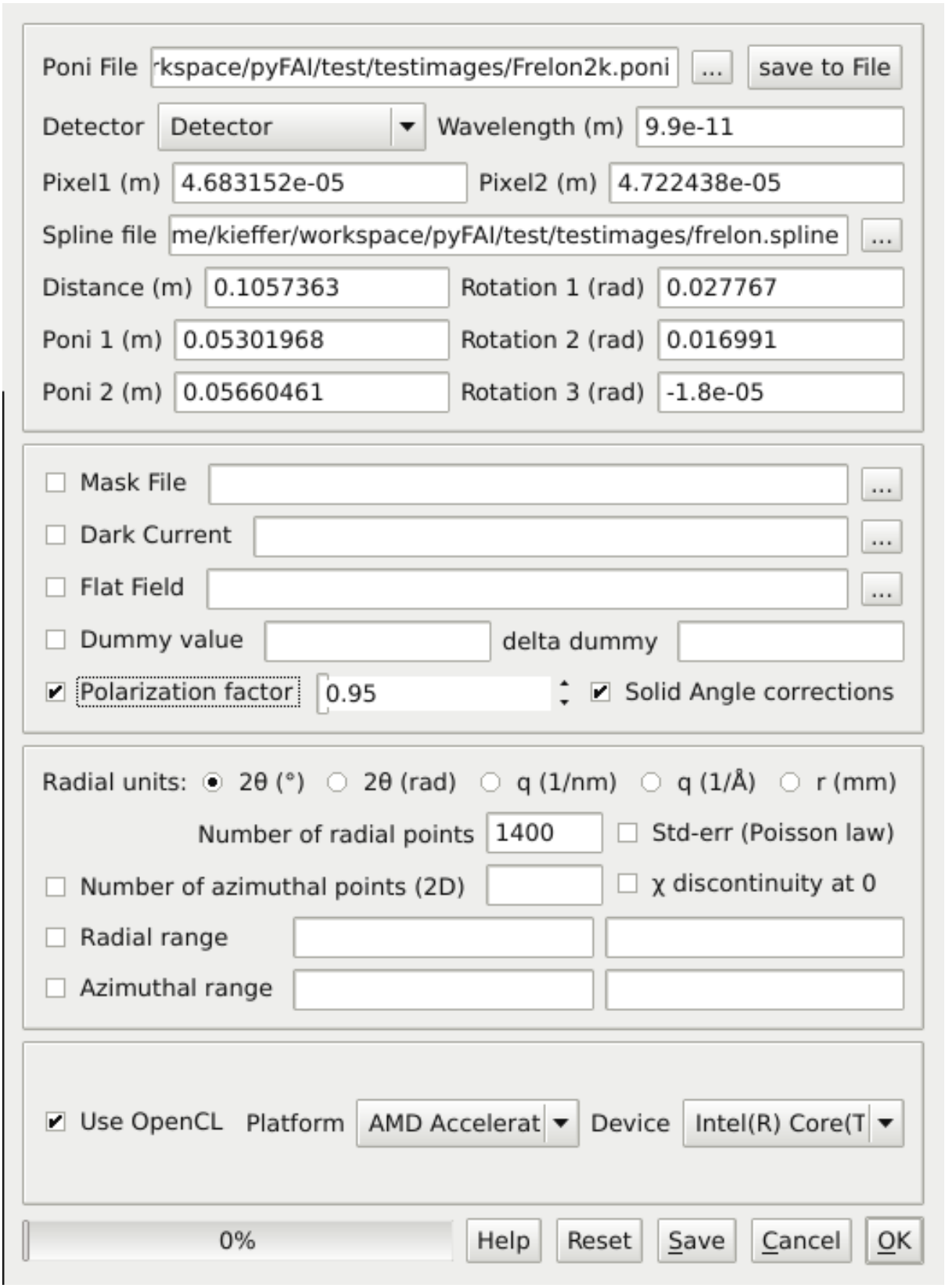
4.8.4 Tips & Tricks:

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

4.8.5 Example:

```
$ pyFAI-integrate --help
usage: pyFAI-integrate [options] file1.edf file2.edf ...

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.

positional arguments:
  FILE                  Files to be integrated

optional arguments:
```


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

```
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -v, --verbose         switch to verbose/debug mode
  -o OUTPUT, --output OUTPUT
                        Directory or file where to store the output data
  -f FORMAT, --format FORMAT
                        output data format (can be HDF5)
  -s SLOW, --slow-motor SLOW
                        Dimension of the scan on the slow direction (makes
                        sense only with HDF5)
  -r RAPID, --fast-motor RAPID
                        Dimension of the scan on the fast direction (makes
                        sense only with HDF5)
  --no-gui              Process the dataset without showing the user
                        interface.
  -j JSON, --json JSON  Configuration file containing the processing to be
                        done

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
```

4.9 Integration tool: diff_map

4.9.1 Purpose

Azimuthal integration for diffraction imaging

Diffraction mapping is an imaging experiment where 2D diffraction patterns are recorded while performing a 2D scan along two axes, one slower and the other fastest.

Diff_map provides a Graphical User Interface (based on top of PyQt, pyFAI and h5py) which allows the reduction of this 4D dataset into a 3D dataset containing the two dimension of movement and the many diffraction angles (the output can be q-space for PDF-CT).

On the left-hand side, the user can select a bunch of files to be processed. For now, any image format supported by FabIO is possible, including multi-frame images, but not ye NeXus files (work ongoing).

On the right-hand side, the motor range can be specified togeather with their names. The diffraction parameters can be setup in a similar way to *pyFAI-integrate*. The output name can also be specified.

The processing is launched using the *Run* button which opens a matplotlib figure to display the plot of the diffraction pattern and of the map image under construction. During this processing, one can select a scatering range to highlight the regions of interest.

Further analysis can be performed on the resulting dataset using the PyMca roitool where the 1d dataset has to be selected as last dimension. The result file aims at NeXus compliance.

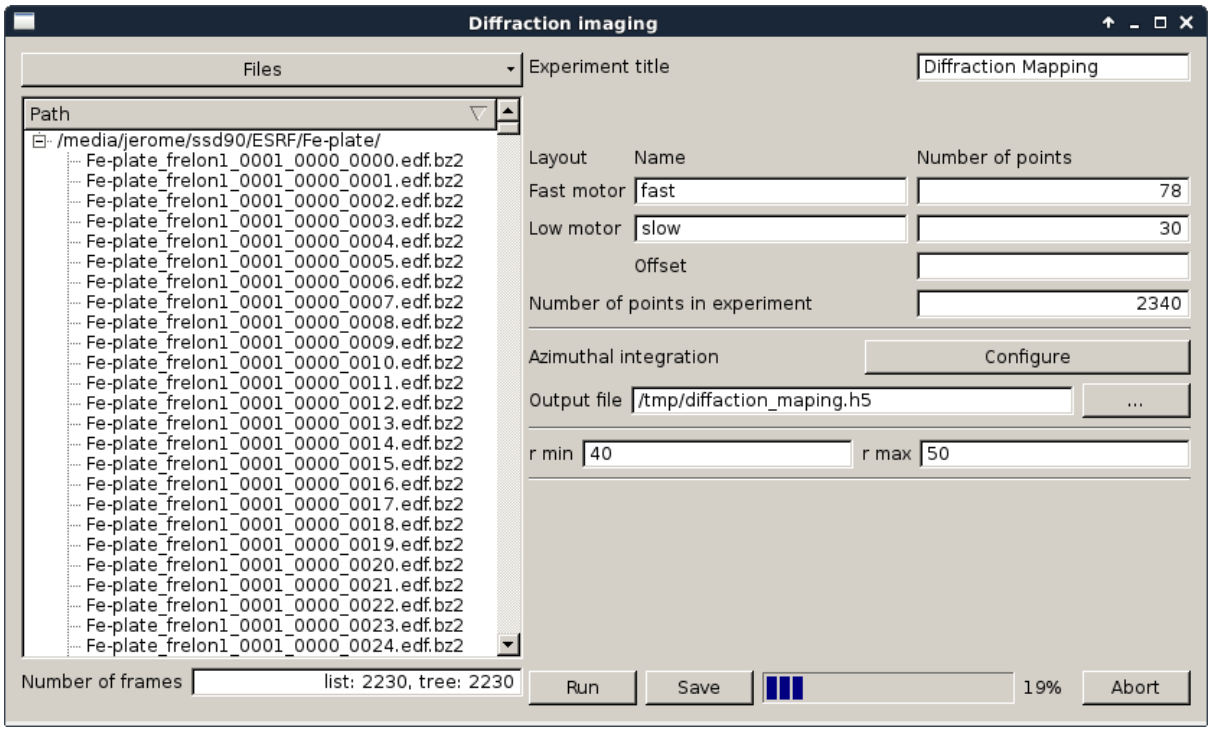
This tool can be used for tomography experiments if one considers the slow scan direction as the rotation.

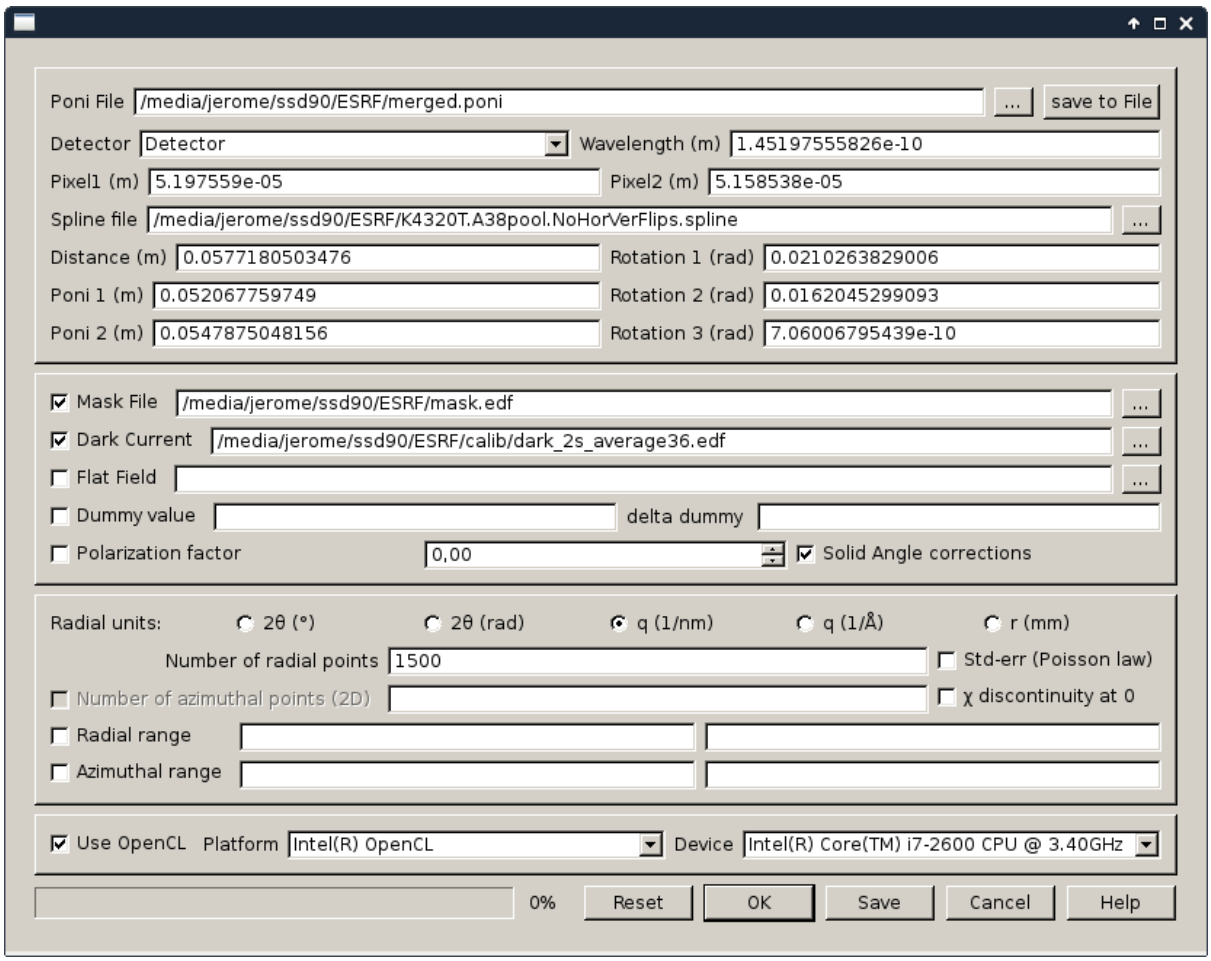
The user interface can be disabled to allow scripting interface and providing a JSON configuration file (or all the options).

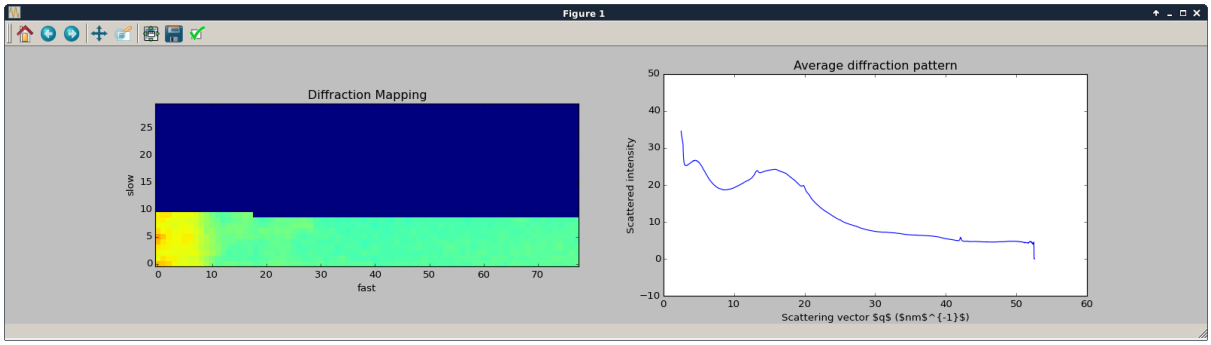
4.9.2 Usage:

diff_map [options] imagefiles*

positional arguments: FILE List of files to integrate


Diffraction imaging
Files
Path
/media/jerome/ssd90/ESRF/Fe-plate/
Fe-plate_frelon1_0001_0000_0000.edf.bz2
Fe-plate_frelon1_0001_0000_0001.edf.bz2
Fe-plate_frelon1_0001_0000_0002.edf.bz2
Fe-plate_frelon1_0001_0000_0003.edf.bz2
Fe-plate_frelon1_0001_0000_0004.edf.bz2
Fe-plate_frelon1_0001_0000_0005.edf.bz2
Fe-plate_frelon1_0001_0000_0006.edf.bz2
Fe-plate_frelon1_0001_0000_0007.edf.bz2
Fe-plate_frelon1_0001_0000_0008.edf.bz2
Fe-plate_frelon1_0001_0000_0009.edf.bz2
Fe-plate_frelon1_0001_0000_0010.edf.bz2
Fe-plate_frelon1_0001_0000_0011.edf.bz2
Fe-plate_frelon1_0001_0000_0012.edf.bz2
Fe-plate_frelon1_0001_0000_0013.edf.bz2
Fe-plate_frelon1_0001_0000_0014.edf.bz2
Fe-plate_frelon1_0001_0000_0015.edf.bz2
Fe-plate_frelon1_0001_0000_0016.edf.bz2
Fe-plate_frelon1_0001_0000_0017.edf.bz2
Fe-plate_frelon1_0001_0000_0018.edf.bz2
Fe-plate_frelon1_0001_0000_0019.edf.bz2
Fe-plate_frelon1_0001_0000_0020.edf.bz2
Fe-plate_frelon1_0001_0000_0021.edf.bz2
Fe-plate_frelon1_0001_0000_0022.edf.bz2
Fe-plate_frelon1_0001_0000_0023.edf.bz2
Fe-plate_frelon1_0001_0000_0024.edf.bz2
Number of frames
list: 2230, tree: 2230
Experiment title
Diffraction Mapping
Layout
Name
Number of points
Fast motor
fast
78
Low motor
slow
30
Offset
Number of points in experiment
2340
Azimuthal integration
Configure
Output file
/tmp/diffraction_maping.h5
r min
40
r max
50
Run
Save
19%
Abort


Poni File
/media/jerome/ssd90/ESRF/merged.poni
save to File
Detector
Detector
Wavelength (m)
1.45197555826e-10
Pixel1 (m)
5.197559e-05
Pixel2 (m)
5.158538e-05
Spline file
/media/jerome/ssd90/ESRF/K4320T.A38pool.NoHorVerFlips.spline
Distance (m)
0.0577180503476
Rotation 1 (rad)
0.0210263829006
Poni 1 (m)
0.052067759749
Rotation 2 (rad)
0.0162045299093
Poni 2 (m)
0.0547875048156
Rotation 3 (rad)
7.06006795439e-10
Mask File
/media/jerome/ssd90/ESRF/mask.edf
Dark Current
/media/jerome/ssd90/ESRF/calib/dark_2s_average36.edf
Flat Field
Dummy value
delta dummy
Polarization factor
0,00
Solid Angle corrections
Radial units:
2θ (°)
2θ (rad)
q (1/nm)
q (1/Å)
r (mm)
Number of radial points
1500
Std-err (Poisson law)
Number of azimuthal points (2D)
χ discontinuity at 0
Radial range
Azimuthal range
Use OpenCL
Platform
Intel(R) OpenCL
Device
Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
0%
Reset
OK
Save
Cancel
Help



optional arguments:

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

-V, --version show program's version number and exit

-o FILE, --output FILE HDF5 File where processed map will be saved

-v, --verbose switch to verbose/debug mode, defaut: quiet

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

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

-t FAST, --fast FAST number of points for the fast motion. Mandatory without GUI

-r SLOW, --slow SLOW number of points for slow motion. Mandatory without GUI

-c NPT_RAD, --npt NPT_RAD number of points in diffraction powder pattern, Mandatory without GUI

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

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

-m FILE, --mask FILE file containing the mask, no mask by default

-p FILE, --poni FILE file containing the diffraction parameter (poni-file), Mandatory without GUI

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

-g, --gpu process using OpenCL on GPU

-S, --stats show statistics at the end

--gui Use the Graphical User Interface

--no-gui Do not use the Graphical User Interface

--config CONFIG provide a JSON configuration file

Bugs:

1. If the number of files is too large, use double quotes "*.edf"
2. There is a known bug on Debian7 where importing a large number of file can take much longer than the integration itself: consider passing files in the command line

```
$ diff_map --help
usage: diff_map [options] -p ponifile imagefiles*
If the number of files is too large, use double quotes like "*.edf"

Azimuthal integration for diffraction imaging. Diffraction mapping is an
experiment where 2D diffraction patterns are recorded while performing a 2D
scan. Diff_map is a graphical application (based on pyFAI and h5py) which
```

```
allows the reduction of this 4D dataset into a 3D dataset containing the two
motion dimensions and the many diffraction angles (thousands). The resulting
dataset can be opened using PyMca roitool where the 1d dataset has to be
selected as last dimension. This result file aims at being NeXus compliant.
This tool can be used for diffraction tomography experiment as well,
considering the slow scan direction as the rotation.

positional arguments:
  FILE                  List of files to integrate. Mandatory without GUI

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -o FILE, --output FILE
                        HDF5 File where processed map will be saved. Mandatory
                        without GUI
  -v, --verbose         switch to verbose/debug mode, default: quiet
  -P FILE, --prefix FILE
                        Prefix or common base for all files
  -e EXTENSION, --extension EXTENSION
                        Process all files with this extension
  -t FAST, --fast FAST  number of points for the fast motion. Mandatory
                        without GUI
  -r SLOW, --slow SLOW  number of points for slow motion. Mandatory without
                        GUI
  -c NPT_RAD, --npt NPT_RAD
                        number of points in diffraction powder pattern.
                        Mandatory without GUI
  -d FILE, --dark FILE  list of dark images to average and subtract (comma
                        separated list)
  -f FILE, --flat FILE  list of flat images to average and divide (comma
                        separated list)
  -m FILE, --mask FILE  file containing the mask, no mask by default
  -p FILE, --poni FILE  file containing the diffraction parameter (poni-file),
                        Mandatory without GUI
  -O OFFSET, --offset OFFSET
                        do not process the first files
  -g, --gpu             process using OpenCL on GPU
  -S, --stats           show statistics at the end
  --gui                 Use the Graphical User Interface
  --no-gui              Do not use the Graphical User Interface
  --config CONFIG       provide a JSON configuration file

Bugs: Many, see hereafter: 1)If the number of files is too large, use double
quotes "*.edf" 2)There is a known bug on Debian7 where importing a large
number of file can take much longer than the integration itself: consider
passing files in the command line
```

4.10 Integration tool: diff_tomo

4.10.1 Purpose

Azimuthal integration for diffraction tomography.

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

This tool can be used for mapping experiments if one considers the slow scan direction as the rotation, but a tool named *diff_map* operates in a similar way, provides a Graphical interface and is more flexible.

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

4.10.2 Usage:

diff_tomo [options] -p ponifile imagefiles*

4.10.3 Options:

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

-V, --version show program's version number and exit

-o FILE, --output FILE HDF5 File where processed sinogram was saved, by default diff_tomo.h5

-v, --verbose switch to verbose/debug mode, defaut: quiet

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

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

-t NTRANS, --nTrans NTRANS number of points in translation. Mandatory

-r NROT, --nRot NROT number of points in rotation. Mandatory

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

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

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

-m FILE, --mask FILE file containing the mask, no mask by default

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

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

-g, --gpu process using OpenCL on GPU

-S, --stats show statistics at the end

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

```
$ diff_tomo --help
usage: diff_tomo [options] -p ponifile imagefiles*

Azimuthal integration for diffraction tomography. Diffraction tomography is an
experiment where 2D diffraction patterns are recorded while performing a 2D
scan, one (the slowest) in rotation around the sample center and the other
(the fastest) along a translation through the sample. Diff_tomo is a script
(based on pyFAI and h5py) which allows the reduction of this 4D dataset into a
3D dataset containing the rotations angle (hundreds), the translation step
(hundreds) and the many diffraction angles (thousands). The resulting dataset
can be opened using the PyMca ROItool where the 1d dataset has to be selected
as last dimension. The output file aims at being NeXus compliant. This tool
can be used for mapping experiments if one considers the slow scan direction
as the rotation. but the *diff_map* tool provides in addition a graphical user
interface.

positional arguments:
  FILE                  List of files to calibrate

optional arguments:
```

```
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -o FILE, --output FILE
                        HDF5 File where processed sinogram was saved, by
                        default diff_tomo.h5
  -v, --verbose         switch to verbose/debug mode, defaut: quiet
  -P FILE, --prefix FILE
                        Prefix or common base for all files
  -e EXTENSION, --extension EXTENSION
                        Process all files with this extension
  -t NTRANS, --nTrans NTRANS
                        number of points in translation. Mandatory
  -r NROT, --nRot NROT  number of points in rotation. Mandatory
  -c NDIFF, --nDiff NDIFF
                        number of points in diffraction powder pattern,
                        Mandatory
  -d FILE, --dark FILE  list of dark images to average and subtract
  -f FILE, --flat FILE  list of flat images to average and divide
  -m FILE, --mask FILE  file containing the mask, no mask by default
  -p FILE, --poni FILE  file containing the diffraction parameter (poni-file),
                        Mandatory
  -O OFFSET, --offset OFFSET
                        do not process the first files
  -g, --gpu             process using OpenCL on GPU
  -S, --stats           show statistics at the end

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

4.11 Integration tool: pyFAI-saxs

4.11.1 Purpose

Azimuthal integration for SAXS users.

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

4.11.2 Usage:

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

Options:

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

-v, --version show program's version number and exit

-p PONIFILE PyFAI parameter file (.poni)

-n NPT, --npt NPT Number of points in radial dimension

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

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

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

-U DELTA_DUMMY, --delta_dummy DELTA_DUMMY delta dummy value

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

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

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

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

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

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

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

--method METHOD Integration method

4.11.3 Example:

```
$ pyFAI-saxs --help
usage: pyFAI-saxs [options] -n 1000 -p ponifile file1.edf file2.edf ...

Azimuthal integration for SAXS users.

positional arguments:
  FILE                  Image files to integrate

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -p PONIFILE           PyFAI parameter file (.poni)
  -n NPT, --npt NPT     Number of points in radial dimension
  -w WAVELENGTH, --wavelength WAVELENGTH
                        wavelength of the X-Ray beam in Angstrom
  -e ENERGY, --energy ENERGY
                        energy of the X-Ray beam in keV (hc=12.398419292keV.A)
  -u DUMMY, --dummy DUMMY
                        dummy value for dead pixels
  -U DELTA_DUMMY, --delta_dummy DELTA_DUMMY
                        delta dummy value
  -m MASK, --mask MASK  name of the file containing the mask image
  -d DARK, --dark DARK  name of the file containing the dark current
  -f FLAT, --flat FLAT  name of the file containing the flat field
  -P POLARIZATION_FACTOR, --polarization POLARIZATION_FACTOR
                        Polarization factor, from -1 (vertical) to +1
                        (horizontal), default is None for no correction,
                        synchrotrons are around 0.95
  --error-model ERROR_MODEL
                        Error model to use. Currently on 'poisson' is
                        implemented
  --unit UNIT           unit for the radial dimension: can be q_nm^-1, q_A^-1,
                        2th_deg, 2th_rad or r_mm
  --ext EXT             extension of the regrouped filename (.dat)
  --method METHOD       Integration method

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

4.12 Integration tool: pyFAI-waxs

4.12.1 Purpose

Azimuthal integration for powder diffraction.

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

4.12.2 Usage:

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

4.12.3 Options:

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

-v, --version show program's version number and exit

-p PONIFILE PyFAI parameter file (.poni)

-n NPT, --npt NPT Number of points in radial dimension

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

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

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

-U DELTA_DUMMY, --delta_dummy DELTA_DUMMY delta dummy value

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

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

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

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

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

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

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

--method METHOD Integration method

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

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

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

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

4.12.4 Example:

```
$ pyFAI-waxs --help
usage: pyFAI-waxs [options] -p ponifile file1.edf file2.edf ...

Azimuthal integration for powder diffraction.

positional arguments:
  FILE                  Image files to integrate

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -p PONIFILE           PyFAI parameter file (.poni)
  -n NPT, --npt NPT     Number of points in radial dimension
  -w WAVELENGTH, --wavelength WAVELENGTH
                        wavelength of the X-Ray beam in Angstrom
  -e ENERGY, --energy ENERGY
                        energy of the X-Ray beam in keV (hc=12.398419292keV.A)
  -u DUMMY, --dummy DUMMY
                        dummy value for dead pixels
  -U DELTA_DUMMY, --delta_dummy DELTA_DUMMY
                        delta dummy value
  -m MASK, --mask MASK  name of the file containing the mask image
  -d DARK, --dark DARK  name of the file containing the dark current
  -f FLAT, --flat FLAT  name of the file containing the flat field
  -P POLARIZATION_FACTOR, --polarization POLARIZATION_FACTOR
                        Polarization factor, from -1 (vertical) to +1
                        (horizontal), default is None for no correction,
                        synchrotrons are around 0.95
  --error-model ERROR_MODEL
                        Error model to use. Currently on 'poisson' is
                        implemented
  --unit UNIT           unit for the radial dimension: can be q_nm^-1, q_A^-1,
                        2th_deg, 2th_rad or r_mm
  --ext EXT             extension of the regrouped filename (.xy)
  --method METHOD       Integration method
  --multi               Average out all frame in a file before integrating
                        extracting variance, otherwise treat every single
                        frame
  --average AVERAGE     Method for averaging out: can be 'mean' (default),
                        'min', 'max' or 'median
  --do-2D               Perform 2D integration in addition to 1D

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

CHAPTER

FIVE

DESIGN OF THE PYTHON FAST AZIMUTHAL INTEGRATION LIBRARY

Author: Jérôme Kieffer

Date: 18/12/2014

Keywords: Design

Target: Developers interested in using the library

Reference: API documentation

5.1 Design of the Python Fast Azimuthal Integrator

Author: Jérôme Kieffer

Date: 20/03/2015

Keywords: Design

Target: Developers interested in using the library

Reference: API documentation

5.1.1 Abstract

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

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

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

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

5.1.2 AzimuthalIntegrator

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

```
>>> import pyFAI
>>> ai = pyFAI.load("Pilatus1M.poni")
>>> print(ai)

Detector Detector        Spline= None    PixelSize= 1.720e-04, 1.720e-04 m

SampleDetDist= 1.583231e+00m    PONI= 3.341702e-02, 4.122778e-02m       rot1=0.006487  rot2= 0.00

DirectBeamDist= 1583.310mm      Center: x=179.981, y=263.859 pix        Tilt=0.571 deg  tiltPlanR
```

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

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

Azimuthal integration methods

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

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

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

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

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

5.1.3 Geometry

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

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

5.1.4 Detector

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

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

A detector class is responsible for two main tasks:

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

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

5.1.5 Rebinning engines

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

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

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

5.1.6 Pixel splitting

Three levels of pixel splitting schemes are available within pyFAI:

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

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

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

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

5.1.7 References:

CHAPTER

SIX

PYFAI API

This chapter describes the programming interface of pyFAI, so what you can expect after having launched *Jupyter notebook* (or 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.

6.1 pyFAI Package

pyFAI.__init__.**benchmarks**(**arg*, ***kwarg*)
Run the integrated benchmarks.

See the documentation of pyFAI.benchmark.run_benchmark

pyFAI.__init__.**tests**(*deprecation=False*)
Runs the test suite of the installed version

Parameters **deprecation** – enable/disables deprecation warning in the tests

6.2 average Module

Utilities, mainly for image treatment

exception pyFAI.average.**AlgorithmCreationError**
Bases: exceptions.RuntimeError

Exception returned if creation of an ImageReductionFilter is not possible

class pyFAI.average.**Average**
Bases: object

Process images to generate an average using different algorithms.

__init__()
Constructor

add_algorithm(*algorithm*)
Defines another algorithm which will be computed on the source.

Parameters **algorithm** (*ImageReductionFilter*) – An averaging algorithm.

get_counter_frames()
Returns the number of frames used for the process.

Return type int

get_fabio_images()
Returns source images as fabio images.

Return type list(fabio.fabioimage.FabioImage)

get_image_reduction(*algorithm*)
Returns the result of an algorithm. The *process* must be already done.

Parameters **algorithm** (*ImageReductionFilter*) – An averaging algorithm

Return type numpy.ndarray

process()
Process source images to all defined averaging algorithms defined using defined parameters. To access to the results you have to define a writer (*AverageWriter*). To follow the process forward you have to define an observer (*AverageObserver*).

set_correct_flat_from_dark(*correct_flat_from_dark*)
Defines if the dark must be applied on the flat.

Parameters **correct_flat_from_dark** (*bool*) – If true, the dark is applied.

set_dark(*dark_list*)
Defines images used as dark.

Parameters **dark_list** (*list*) – List of dark used

set_flat(*flat_list*)
Defines images used as flat.

Parameters **flat_list** (*list*) – List of dark used

set_images(*image_list*)
Defines the set set of source images to used to process an average.

Parameters **image_list** (*list*) – List of filename, numpy arrays, fabio images used as source for the computation.

set_monitor_name(*monitor_name*)
Defines the monitor name used to correct images before processing the average. This monitor must be part of the file header, else the image is skipped.

Parameters **monitor_name** (*str*) – Name of the monitor available on the header file

set_observer(*observer*)
Set an observer to the average process.

Parameters **observer** (*AverageObserver*) – An observer

set_pixel_filter(*threshold*, *minimum*, *maximum*)
Defines the filter applied on each pixels of the images before processing the average.

Parameters

- **threshold** – what is the upper limit? all pixel > max*(1-threshold) are discareded.
- **minimum** – minimum valid value or True
- **maximum** – maximum valid value

set_writer(*writer*)
Defines the object write which will be used to store the result.

Parameters **writer** (*AverageWriter*) – The writer to use.

class pyFAI.average.**AverageDarkFilter**(*filter_name*, *cut_off*, *quantiles*)
Bases: pyFAI.average.ImageStackFilter

Filter based on the algorithm of average_dark

TODO: Must be split according to each filter_name, and removed

__init__(*filter_name*, *cut_off*, *quantiles*)

get_parameters()
Return a dictionary containing filter parameters

name

class pyFAI.average.**AverageObserver**
Bases: object

algorithm_finished(*algorithm*)
Called when an algorithm is finished

algorithm_started(*algorithm*)
Called when an algorithm is started

frame_processed(*algorithm*, *frame_index*, *frames_count*)
Called after providing a frame to an algorithm

image_loaded(*fabio_image*, *image_index*, *images_count*)
Called when an input image is loaded

process_finished()
Called when the full process is finished

process_started()
Called when the full processing is started

result_processing(*algorithm*)
Called before the result of an algorithm is computed

class pyFAI.average.**AverageWriter**
Interface for using writer in *Average* process.

close()
Close the writer. Must not be used anymore.

write_header(*merged_files*, *nb_frames*, *monitor_name*)
Write the header of the average

Parameters

- **merged_files** (*list*) – List of files used to generate this output
- **nb_frames** (*int*) – Number of frames used
- **monitor_name** (*str*) – Name of the monitor used. Can be None.

write_reduction(*algorithm*, *data*)
Write one reduction

Parameters

- **algorithm** (*ImageReductionFilter*) – Algorithm used
- **data** (*object*) – Data of this reduction

class pyFAI.average.**ImageAccumulatorFilter**
Bases: pyFAI.average.ImageReductionFilter

Filter applied in a set of images in which it is possible to reduce data step by step into a single merged image.

add_image(*image*)
Add an image to the filter.

Parameters **image** (*numpy.ndarray*) – image to add

get_result()
Get the result of the filter.

Returns result filter

Return type numpy.ndarray

init(*max_images=None*)

class pyFAI.average.**ImageReductionFilter**
Bases: object

Generic filter applied in a set of images.

add_image(*image*)
Add an image to the filter.

Parameters **image** (*numpy.ndarray*) – image to add

get_parameters()
Return a dictionary containing filter parameters

Return type dict

get_result()
Get the result of the filter.

Returns result filter

init(*max_images=None*)
Initialize the filter before using it.

Parameters **max_images** (*int*) – Max images supported by the filter

class pyFAI.average.**ImageStackFilter**
Bases: pyFAI.average.ImageReductionFilter

Filter creating a stack from all images and computing everything at the end.

add_image(*image*)
Add an image to the filter.

Parameters **image** (*numpy.ndarray*) – image to add

get_result()

init(*max_images=None*)

class pyFAI.average.**MaxAveraging**
Bases: pyFAI.average.ImageAccumulatorFilter

name = 'max'

class pyFAI.average.**MeanAveraging**
Bases: pyFAI.average.SumAveraging

get_result()

name = 'mean'

class pyFAI.average.**MinAveraging**
Bases: pyFAI.average.ImageAccumulatorFilter

name = 'min'

exception pyFAI.average.**MonitorNotFound**
Bases: exceptions.Exception

Raised when monitor information in not found or is not valid.

class pyFAI.average.**MultiFilesAverageWriter**(*file_name_pattern*, *file_format*, *dry_run=False*)
Bases: pyFAI.average.AverageWriter

Write reductions into multi files. File headers are duplicated.

__init__(*file_name_pattern*, *file_format*, *dry_run=False*)

Parameters

- **file_name_pattern** (*str*) – File name pattern for the output files. If it contains "{method_name}", it is updated for each reduction writing with the name of the reduction.
- **file_format** (*str*) – File format used. It is the default extension file.
- **dry_run** (*bool*) – If dry_run, the file is created on memory but not saved on the file system at the end

close()
Close the writer. Must not be used anymore.

get_fabio_image(*algorithm*)
Get the constructed fabio image

Return type fabio.fabioimage.FabioImage

write_header(*merged_files*, *nb_frames*, *monitor_name*)

write_reduction(*algorithm*, *data*)

class pyFAI.average.**SumAveraging**
Bases: pyFAI.average.ImageAccumulatorFilter

name = 'sum'

pyFAI.average.**average_dark**(*lstimg*, *center_method='mean'*, *cutoff=None*, *quantiles=(0.5, 0.5)*)
Averages a serie of dark (or flat) images. Centers the result on the mean or the median ... but averages all frames within cutoff*std

Parameters

- **lstimg** – list of 2D images or a 3D stack
- **center_method** (*str*) – is the center calculated by a "mean", "median", "quantile", "std"
- **cutoff** (*float or None*) – keep all data where (I-center)/std < cutoff
- **quantiles** (*tuple(float, float) or None*) – 2-tuple of floats average out data between the two quantiles

Returns 2D image averaged

pyFAI.average.**average_images**(*listImages*, *output=None*, *threshold=0.1*, *minimum=None*, *maximum=None*, *darks=None*, *flats=None*, *filter_='mean'*, *correct_flat_from_dark=False*, *cutoff=None*, *quantiles=None*, *fformat='edf'*, *monitor_key=None*)

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 "min", "max", "median", "mean", "sum", "quantiles" (default='mean')
- **correct_flat_from_dark** – shall the flat be re-corrected ?
- **cutoff** – keep all data where (I-center)/std < cutoff

- **quantiles** – 2-tuple containing the lower and upper quantile (0<q<1) to average out.
- **fformat** – file format of the output image, default: edf
- **str** (*monitor_key*) – Key containing the monitor. Can be none.

Returns filename with the data or the data ndarray in case format=None

pyFAI.average.**bounding_box**(*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.average.**common_prefix**(*string_list*)
Return the common prefix of a list of strings

TODO: move it into utils package

Parameters **string_list** (*list(str)*) – List of strings

Return type str

pyFAI.average.**create_algorithm**(*filter_name*, *cut_off=None*, *quantiles=None*)
Factory to create algorithm according to parameters

Parameters

- **cutoff** (*float or None*) – keep all data where (I-center)/std < cutoff
- **quantiles** (*tuple(float, float) or None*) – 2-tuple of floats average out data between the two quantiles

Returns An algorithm

Return type ImageReductionFilter

Raises **AlgorithmCreationError** If it is not possible to create the algorithm

pyFAI.average.**f**
alias of SumAveraging

pyFAI.average.**is_algorithm_name_exists**(*filter_name*)
Return true if the name is a name of a filter algorithm

pyFAI.average.**remove_saturated_pixel**(*ds*, *threshold=0.1*, *minimum=None*, *maximum=None*)
Remove saturated fixes from an array inplace.

Parameters

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

Returns the input dataset

6.3 azimuthalIntegrator Module

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

Bases: pyFAI.geometry.Geometry

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

All geometry calculation are done in the Geometry class

main methods are:

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

DEFAULT_METHOD = 'splitbbox'

__init__(*dist=1*, *poni1=0*, *poni2=0*, *rot1=0*, *rot2=0*, *rot3=0*, *pixel1=None*, *pixel2=None*, *splineFile=None*, *detector=None*, *wavelength=None*)

Parameters

- **dist** (*float*) – distance sample - detector plan (orthogonal distance, not along the beam), in meter.
- **poni1** (*float*) – coordinate of the point of normal incidence along the detector's first dimension, in meter
- **poni2** (*float*) – coordinate of the point of normal incidence along the detector's second dimension, in meter
- **rot1** (*float*) – first rotation from sample ref to detector's ref, in radians
- **rot2** (*float*) – second rotation from sample ref to detector's ref, in radians
- **rot3** (*float*) – third rotation from sample ref to detector's ref, in radians
- **pixel1** (*float*) – Deprecated. Pixel size of the fist dimension of the detector, in meter. If both pixel1 and pixel2 are not None, detector pixel size is overwritten. Prefer defining the detector pixel size on the provided detector object. Prefer defining the detector pixel size on the provided detector object (`detector.pixel1 = 5e-6`).
- **pixel2** (*float*) – Deprecated. Pixel size of the second dimension of the detector, in meter. If both pixel1 and pixel2 are not None, detector pixel size is overwritten. Prefer defining the detector pixel size on the provided detector object (`detector.pixel2 = 5e-6`).
- **splineFile** (*str*) – Deprecated. File containing the geometric distortion of the detector. If not None, pixel1 and pixel2 are ignored and detector spline is overwritten. Prefer defining the detector spline manually (`detector.splineFile = "file.spline"`).
- **detector** (*str or pyFAI.Detector*) – name of the detector or Detector instance. String description is deprecated. Prefer using the result of the detector factory: `pyFAI.detector_factory("eiger4m")`
- **wavelength** (*float*) – Wave length used in meter

create_mask(*data*, *mask=None*, *dummy=None*, *delta_dummy=None*, *mode='normal'*)

Combines various masks into another one.

Parameters

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

Returns the new mask

Return type ndarray of bool

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

This method can work in two different *mode*:

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

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

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

dark_correction(*data*, *dark=None*)

Correct for Dark-current effects. If dark is not defined, correct for a dark set by "set_darkfiles"

Parameters

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

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

darkcurrent

empty

flat_correction(*data*, *flat=None*)

Correct for flat field. If flat is not defined, correct for a flat set by "set_flatfiles"

Parameters

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

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

flatfield

get_darkcurrent()

get_empty()

get_flatfield()

integrate1d(*data*, *npt*, *filename=None*, *correctSolidAngle=True*, *variance=None*, *error_model=None*, *radial_range=None*, *azimuth_range=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*, *method='csr'*, *unit=q_nm^-1*, *safe=True*, *normalization_factor=1.0*, *block_size=32*, *profile=False*, *all=False*)

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

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

Parameters

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

- **npt** (*int*) – number of points in the output pattern
- **filename** (*str*) – output filename in 2/3 column ascii format
- **correctSolidAngle** (*bool*) – correct for solid angle of each pixel if True
- **variance** (*ndarray*) – array containing the variance of the data. If not available, no error propagation is done
- **error_model** (*str*) – When the variance is unknown, an error model can be given: "poisson" (variance = I), "azimuthal" (variance = (I-<I>)^2)
- **radial_range** ((*float, float), optional*) – The lower and upper range of the radial unit. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **azimuth_range** ((*float, float), optional*) – The lower and upper range of the azimuthal angle in degree. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **mask** (*ndarray*) – array (same size as image) with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor between -1 (vertical) and +1 (horizontal). 0 for circular polarization or random, None for no correction, True for using the former correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image
- **method** (*str*) – can be "numpy", "cython", "BBox" or "splitpixel", "lut", "csr", "nosplit_csr", "full_csr", "lut_ocl" and "csr_ocl" if you want to go on GPU. To Specify the device: "csr_ocl_1,2"
- **unit** (*pyFAI.units.Enum*) – Output units, can be "q_nm^-1", "q_A^-1", "2th_deg", "2th_rad", "r_mm" for now
- **safe** (*bool*) – Do some extra checks to ensure LUT/CSR is still valid. False is faster.
- **normalization_factor** (*float*) – Value of a normalization monitor
- **block_size** – size of the block for OpenCL integration (unused?)
- **profile** – set to True to enable profiling in OpenCL
- **all** (*bool*) – if true return a dictionary with many more parameters (deprecated, please refer to the documentation of Integrate1dResult).

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

Return type Integrate1dResult, dict

integrate2d(*data*, *npt_rad*, *npt_azim=360*, *filename=None*, *correctSolidAngle=True*, *variance=None*, *error_model=None*, *radial_range=None*, *azimuth_range=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *polarization_factor=None*, *dark=None*, *flat=None*, *method='bbox'*, *unit=q_nm^-1*, *safe=True*, *normalization_factor=1.0*, *all=False*)

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

Multi algorithm implementation (tries to be bullet proof)

Parameters

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

- **npt_rad** (*int*) – number of points in the radial direction
- **npt_azim** (*int*) – number of points in the azimuthal direction
- **filename** (*str*) – output image (as edf format)
- **correctSolidAngle** (*bool*) – correct for solid angle of each pixel if True
- **variance** (*ndarray*) – array containing the variance of the data. If not available, no error propagation is done
- **error_model** (*str*) – When the variance is unknown, an error model can be given: "poisson" (variance = I), "azimuthal" (variance = (I-<I>)^2)
- **radial_range** ((*float, float), optional*) – The lower and upper range of the radial unit. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **azimuth_range** ((*float, float), optional*) – The lower and upper range of the azimuthal angle in degree. If not provided, range is simply (data.min(), data.max()). Values outside the range are ignored.
- **mask** (*ndarray*) – array (same size as image) with 1 for masked pixels, and 0 for valid pixels
- **dummy** (*float*) – value for dead/masked pixels
- **delta_dummy** (*float*) – precision for dummy value
- **polarization_factor** (*float*) – polarization factor between -1 (vertical) and +1 (horizontal). 0 for circular polarization or random, None for no correction
- **dark** (*ndarray*) – dark noise image
- **flat** (*ndarray*) – flat field image
- **method** (*str*) – can be "numpy", "cython", "BBox" or "splitpixel", "lut", "csr; "lut_ocl" and "csr_ocl" if you want to go on GPU. To Specify the device: "csr_ocl_1,2"
- **unit** (*pyFAI.units.Enum*) – Output units, can be "q_nm^-1", "q_A^-1", "2th_deg", "2th_rad", "r_mm" for now
- **safe** (*bool*) – Do some extra checks to ensure LUT is still valid. False is faster.
- **normalization_factor** (*float*) – Value of a normalization monitor
- **all** (*bool*) – if true, return many more intermediate results as a dict (deprecated, please refer to the documentation of Integrate2dResult).

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

Return type Integrate2dResult, dict

reset()

Reset azimuthal integrator in addition to other arrays.

save1D(*filename*, *dim1*, *I*, *error=None*, *dim1_unit=2th_deg*, *has_dark=False*, *has_flat=False*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

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

- **has_flat** (*bool*) – save the flat filenames (default: no)
- **polarization_factor** (*float*) – the polarization factor
- **normalization_factor** (*float*) – the monitor value

This method save the result of a 1D integration.

save2D (*filename*, *I*, *dim1*, *dim2*, *error=None*, *dim1_unit=2th_deg*, *has_dark=False*, *has_flat=False*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

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

This method save the result of a 2D integration.

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

separate (*data*, *npt_rad=1024*, *npt_azim=512*, *unit='2th_deg'*, *method='splitpixel'*, *percentile=50*, *mask=None*, *restore_mask=True*)
Separate bragg signal from powder/amorphous signal using azimuthal integration, median filering and projected back before subtraction.

Parameters

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

Returns bragg, amorphous

set_darkcurrent (*dark*)

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

Parameters

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

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

set_empty (*value*)

set_flatfield(*flat*)

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

Parameters

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

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

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

Prepare a look-up-table

Parameters

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

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

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

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

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

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

Prepare a look-up-table

Parameters

- **shape** (*(int, int)*) – shape of the dataset
- **npt** (*int or (int, int)*) – number of points in the the output pattern
- **mask** (*ndarray*) – array with masked pixel (1=masked)
- **pos0_range** (*(float, float)*) – range in radial dimension
- **pos1_range** (*(float, float)*) – range in azimuthal dimension
- **mask_checksum** (*int (or anything else ...)*) – checksum of the mask buffer
- **unit** (*pyFAI.units.Enum*) – use to propagate the LUT object for further checkings

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4 multi_geometry Module

Module for treating simultaneously multiple detector configuration within a single integration

class pyFAI.multi_geometry.**MultiGeometry**(*ais*, *unit='2th_deg'*, *radial_range=(0, 180)*, *azimuth_range=(-180, 180)*, *wavelength=None*, *empty=0.0*, *chi_disc=180*)

Bases: object

This is an Azimuthal integrator containing multiple geometries (when the detector is on a goniometer arm)

__init__(*ais*, *unit='2th_deg'*, *radial_range=(0, 180)*, *azimuth_range=(-180, 180)*, *wavelength=None*, *empty=0.0*, *chi_disc=180*)

Constructor of the multi-geometry integrator :param ais: list of azimuthal integrators :param radial_range: common range for integration :param azimuthal_range: common range for integration :param empty: value for empty pixels :param chi_disc: if 0, set the chi_discontinuity at

integrate1d(*lst_data*, *npt=1800*, *correctSolidAngle=True*, *lst_variance=None*, *error_model=None*, *polarization_factor=None*, *monitors=None*, *all=False*, *lst_mask=None*, *lst_flat=None*)

Perform 1D azimuthal integration

Parameters

- **lst_data** – list of numpy array
- **npt** – number of points int the integration
- **correctSolidAngle** – correct for solid angle (all processing are then done in absolute solid angle !)
- **lst_variance** (*list of ndarray*) – list of 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)
- **polarization_factor** – Apply polarization correction ? is None: not applies. Else provide a value from -1 to +1
- **monitors** – normalization monitors value (list of floats)
- **all** – return a dict with all information in it (deprecated, please refer to the documentation of Integrate1dResult).
- **lst_mask** – numpy.Array or list of numpy.array which mask the lst_data.
- **lst_flat** – numpy.Array or list of numpy.array which flat the lst_data.

Returns 2th/I or a dict with everything depending on "all"

Return type Integrate1dResult, dict

integrate2d(*lst_data*, *npt_rad=1800*, *npt_azim=3600*, *correctSolidAngle=True*, *lst_variance=None*, *error_model=None*, *polarization_factor=None*, *monitors=None*, *all=False*, *lst_mask=None*, *lst_flat=None*)

Performs 2D azimuthal integration of multiples frames, one for each geometry

Parameters

- **lst_data** – list of numpy array
- **npt** – number of points int the integration
- **correctSolidAngle** – correct for solid angle (all processing are then done in absolute solid angle !)
- **lst_variance** (*list of ndarray*) – list of 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)
- **polarization_factor** – Apply polarization correction ? is None: not applies. Else provide a value from -1 to +1

- **monitors** – normalization monitors value (list of floats)
- **all** – return a dict with all information in it (deprecated, please refer to the documentation of Integrate2dResult).
- **lst_mask** – numpy.Array or list of numpy.array which mask the lst_data.
- **lst_flat** – numpy.Array or list of numpy.array which flat the lst_data.

Returns I/2th/chi or a dict with everything depending on "all"

Return type Integrate2dResult, dict

set_wavelength(*value*)
Changes the wavelength of a group of azimuthal integrators

6.5 integrate_widget Module

pyFAI-integrate

A graphical tool for performing azimuthal integration on series of files.

class pyFAI.integrate_widget.**AIWidget**(*input_data=None*, *output_path=None*, *output_format=None*, *slow_dim=None*, *fast_dim=None*, *json_file='.azimint.json'*)

Bases: PyQt4.QtGui.QWidget

URL = 'http://pyfai.readthedocs.org/en/latest/man/pyFAI-integrate.html'

__init__(*input_data=None*, *output_path=None*, *output_format=None*, *slow_dim=None*, *fast_dim=None*, *json_file='.azimint.json'*)

assign_unit()
assign unit to the corresponding widget

detector_changed()

die()

dump(*filename=None*)
Dump the status of the current widget to a file in JSON

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

Returns dict with configuration

get_config()
Read the configuration of the plugin and returns it as a dictionary

Returns dict with all information.

get_method()
Return the method name for azimuthal intgration

help()

openCL_changed()

platform_changed()

proceed()

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

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

save_config()

select_darkcurrent()

select_flatfield()

select_maskfile()

select_ponifile()

select_splinefile()

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

set_config(*dico*)
Setup the widget from its description

Parameters **dico** (*dict*) – dictionary with description of the widget

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

set_ponifile(*ponifile=None*)

set_validators()
Set all validators for text entries

6.6 geometry Module

This modules contrains only one (large) class in charge of:

- calculating the geometry, i.e. the position in the detector space of each pixel of the detector
- manages caches to store intermediate results

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

Demonstration of the equation done using Mathematica:

```
Axis 1 is allong first dimension of detector (when not tilted), this is the slow dimension of
In[5]:= x1={1,0,0}
Out[5]= {1,0,0}
 Axis 2 is allong second dimension of detector (when not tilted), this is the fast dimension
In[6]:= x2={0,1,0}
Out[6]= {0,1,0}
Axis 3 is along the incident X-Ray beam
In[7]:= x3={0,0,1}
Out[7]= {0,0,1}
In[9]:= id3={x1,x2,x3}
Out[9]= {{1,0,0},{0,1,0},{0,0,1}}
In[10]:= {{1,0,0},{0,1,0},{0,0,1}}
Out[10]= {{1,0,0},{0,1,0},{0,0,1}}
In[11]:= rotM1=RotationMatrix[rot1,x1]
Out[11]= {{1,0,0},{0,Cos[rot1],-Sin[rot1]},{0,Sin[rot1],Cos[rot1]}}
In[12]:= rotM2 =  RotationMatrix[rot2,x2]
Out[12]= {{Cos[rot2],0,Sin[rot2]},{0,1,0},{-Sin[rot2],0,Cos[rot2]}}
```

```
In[13]:= rotM3 =  RotationMatrix[rot3,x3]
Out[13]= {{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
In[14]:= R=rotM3.rotM2.rotM1
Out[14]= {{Cos[rot2] Cos[rot3],Cos[rot3] Sin[rot1] Sin[rot2]-Cos[rot1] Sin[rot3],Cos[rot1] Co
In[15]:= CForm[R.x1]

Out[15]//CForm=
List(Cos(rot2)*Cos(rot3),Cos(rot2)*Sin(rot3),-Sin(rot2))
In[16]:= CForm[R.x2]

Out[16]//CForm=
List(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3),Cos(rot1)*Cos(rot3) + Sin(rot1)*Sin(
In[17]:= CForm[R.x3]
Out[17]//CForm=
List(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3),-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*S
In[18]:= CForm[Det[R]]
Out[18]//CForm=
Power(Cos(rot1),2)*Power(Cos(rot2),2)*Power(Cos(rot3),2) + Power(Cos(rot2),2)*Power(Cos(rot3)
   Power(Cos(rot3),2)*Power(Sin(rot1),2)*Power(Sin(rot2),2) + Power(Cos(rot1),2)*Power(Cos(ro
   Power(Cos(rot1),2)*Power(Sin(rot2),2)*Power(Sin(rot3),2) + Power(Sin(rot1),2)*Power(Sin(ro
In[13]:=
Any pixel on detector plan at coordianate (d1, d2) in meters. Detector is at z=L

In[22]:= P={d1,d2,L}
CForm[R.P]

Out[22]= {d1,d2,L}
Out[23]//CForm=
List(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) + L*(C
   d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3)) + d2*(
In[24]:= t1 = R.P.x1
CForm[t1]
Out[24]= d1 Cos[rot2] Cos[rot3]+d2 (Cos[rot3] Sin[rot1] Sin[rot2]-Cos[rot1] Sin[rot3])+L (Cos
Out[25]//CForm=
d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) + L*(Cos(ro
In[26]:= t2 = R.P.x2
CForm[t2]
Out[26]= d1 Cos[rot2] Sin[rot3]+L (-Cos[rot3] Sin[rot1]+Cos[rot1] Sin[rot2] Sin[rot3])+d2 (Co
Out[27]//CForm=
d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3)) + d2*(Cos
In[28]:= t3=R.P.x3
CForm[t3]
Out[28]= L Cos[rot1] Cos[rot2]+d2 Cos[rot2] Sin[rot1]-d1 Sin[rot2]
Out[29]//CForm=
L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2)
Distance sample to detector point (d1,d2)
(no Mathematica translations)
GraphicsBox[
TagBox[RasterBox[CompressedData["
1:eJxtUstqU1EUDY2iUdSOnBcERwGHDgQf+YNGOrAQSEBBBwlWQfsDcVKsI4sf
IAQnGSQtCQ4yyfvVvO69SW7J+9W8GtIkg8SwXfuQSpFu2Pdxzl5rr7XP2bDY
Nt+saTSaDzfx2DR/er6zY941XsPPi92Pry1afNxdJS9ejuFwqJNl+XMsFstE
o1HijEQio1Qq9bPZbD7SXBHtdnsjGAxK+Xyeut0uTadTms1mNBgMqFwuUyKR
+APOd5cxo9HoHjBqqVSis7Mzms/nlMlkyOfziX/GdzodSiaTpCjKqwscar6o
qkq9Xo/Oz89pb2+PjEYj2Ww22t7epn6/L/DcF/wdt9u9Di4tPAxarZbgZKzd
bmde4jCZTORyuUTPWq1G8Erwb0HdQ2gn+KZ6vU6np6eCm8PhcNDW1hZVKhWx
xvusH/P6jronwVCIcpJE2VxOvLvQ5Q8EyGAwkMfjERilUBCYCGrj8fiv8Xj8
IIAa1iVJMsnIIWZ4dHRE+1/3hW72pUgK5cALTzyfbzwX4Kt5WSb15IRU8M5w
BgcHB2S1WqkL3TxntVgU3CH0g7+XjGs0Gm/D4bDgZS/QQH6/n5xOp+iH/X8z
QY+0Xq9fZ9xyudRKknTIGpibZzOZTMQMcYeoWq1SNptlzMzr9T4G5PbFGS4W
C12hUHDwbNlHEbr4TMEn7hvW69DwDKV3kGv/3zfUPU2n0z+g6Rj+j/H+jXxv
Npvvr/por7qnq+C9G8hbq9Qhr18u+AttYAMa
```

```
"], {{0, 14}, {14, 0}}, {0, 255},
ColorFunction->RGBColor],
BoxForm`ImageTag["Byte", ColorSpace -> "RGB", Interleaving -> True],
Selectable->False],
BaseStyle->"ImageGraphics",
ImageSize->Magnification[1],
ImageSizeRaw->{14, 14},
PlotRange->{{0, 14}, {0, 14}}]


In[30]:= dist = Norm[R.P]
CForm[dist]
Out[30]= (Abs[L Cos[rot1] Cos[rot2]+d2 Cos[rot2] Sin[rot1]-d1 Sin[rot2]]^2+Abs[d1 Cos[rot2] C
Out[31]//CForm=
Sqrt(Power(Abs(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2)),2) + Power(Abs(
       L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3))),2) + Power(Abs(d1*Cos(rot2)*S
     2))
cos(2theta) can be defined as (R.P component along x3) over the distance |R.P|
In[32]:= tthc = ArcCos [-(R.P).x3/Norm[R.P]]
CForm[tthc]

Out[32]= ArcCos[(-L Cos[rot1] Cos[rot2]-d2 Cos[rot2] Sin[rot1]+d1 Sin[rot2])/((Abs[L Cos[rot1
Out[33]//CForm=
ArcCos((-(L*Cos(rot1)*Cos(rot2)) - d2*Cos(rot2)*Sin(rot1) + d1*Sin(rot2))/
    Sqrt(Power(Abs(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2)),2) + Power(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3))),2) + Power(Abs(d1*Cos(rot2)
         d2*(Cos(rot1)*Cos(rot3) + Sin(rot1)*Sin(rot2)*Sin(rot3))),2)))



In[41]:= ttht = ArcTan[t3,Sqrt[t1^2 + t2^2]]

CForm[ttht]


Out[41]= ArcTan[L Cos[rot1] Cos[rot2]+d2 Cos[rot2] Sin[rot1]-d1 Sin[rot2],((d1 Cos[rot2] Cos[
Out[42]//CForm=
ArcTan(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),Sqrt(Power(d1*Cos(rot2)*
      2) + Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin
Tangeant of angle chi is defined as (R.P component along x1) over (R.P component along x2). A
In[36]:= chi =ArcTan[t1  , t2]
CForm[chi]
Out[36]= ArcTan[d1 Cos[rot2] Cos[rot3]+d2 (Cos[rot3] Sin[rot1] Sin[rot2]-Cos[rot1] Sin[rot3])
Out[37]//CForm=
ArcTan(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) + L*
   d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3)) + d2*(
Coodinates of the Point of Normal Incidence

In[38]:= PONI = R.{0,0,L}
CForm[PONI]
Out[38]= {L (Cos[rot1] Cos[rot3] Sin[rot2]+Sin[rot1] Sin[rot3]),L (-Cos[rot3] Sin[rot1]+Cos[r
Out[39]//CForm=
List(L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),L*(-(Cos(rot3)*Sin(rot1)) + Cos(
Derivatives of 2Theta
In[43]:= CForm[D[ttht,d1]]
Out[43]//CForm=
((L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2))*(2*Cos(rot2)*Cos(rot3)*(d1*C
          L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3))) + 2*Cos(rot2)*Sin(rot3)*
        (d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3))
    (2.*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin
       Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
      (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin
```

```
     + (Sin(rot2)*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
    (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(ro
       2) + Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Si
In[44]:= CForm[D[ttht,d2]]

Out[44]//CForm=
((L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2))*(2*(Cos(rot3)*Sin(rot1)*Sin(
         (d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) +
        2*(Cos(rot1)*Cos(rot3) + Sin(rot1)*Sin(rot2)*Sin(rot3))*(d1*Cos(rot2)*Sin(rot3) + L*(
    (2.*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
      (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin
     - (Cos(rot2)*Sin(rot1)*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(r
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
    (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(ro
       2) + Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Si
In[47]:= CForm[D[ttht,L]]
Out[47]//CForm=
((L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2))*(2*(Cos(rot1)*Cos(rot3)*Sin(
         (d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) +
        2*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3))*(d1*Cos(rot2)*Sin(rot3) +
    (2.*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
      (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin
     - (Cos(rot1)*Cos(rot2)*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(r
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
    (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(ro
       2) + Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Si
In[48]:= CForm[D[ttht,rot1]]
Out[48]//CForm=
((L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2))*(2*(L*(-(Cos(rot3)*Sin(rot1)
         (d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) +
        2*(d2*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3)) + L*(-(Cos(rot1)*Cos(r
         (d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3))
    (2.*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
      (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin
     - ((d2*Cos(rot1)*Cos(rot2) - L*Cos(rot2)*Sin(rot1))*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) +
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
    (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(ro
       2) + Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Si
In[49]:= CForm[D[ttht,rot2]]
Out[49]//CForm=
((L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2))*(2*(L*Cos(rot1)*Cos(rot2)*Co
         (d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) +
        2*(L*Cos(rot1)*Cos(rot2)*Sin(rot3) + d2*Cos(rot2)*Sin(rot1)*Sin(rot3) - d1*Sin(rot2)*
         (d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3))
    (2.*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(ro
      (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin
     - ((-(d1*Cos(rot2)) - L*Cos(rot1)*Sin(rot2) - d2*Sin(rot1)*Sin(rot2))*Sqrt(Power(d1*Cos(
         L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin
     /(Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(
       L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin(r
In[50]:= CForm[D[ttht,rot3]]
Out[50]//CForm=
((L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2))*(2*(d1*Cos(rot2)*Cos(rot3) +
        (-(d1*Cos(rot2)*Sin(rot3)) + L*(Cos(rot3)*Sin(rot1) - Cos(rot1)*Sin(rot2)*Sin(rot3))
```

```
        2*(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(rot3)) +
         (d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot3))
    (2.*Sqrt(Power(d1*Cos(rot2)*Cos(rot3) + d2*(Cos(rot3)*Sin(rot1)*Sin(rot2) - Cos(rot1)*Sin(
        Power(d1*Cos(rot2)*Sin(rot3) + L*(-(Cos(rot3)*Sin(rot1)) + Cos(rot1)*Sin(rot2)*Sin(rot
      (Power(L*Cos(rot1)*Cos(rot2) + d2*Cos(rot2)*Sin(rot1) - d1*Sin(rot2),2) + Power(d1*Cos(r
          L*(Cos(rot1)*Cos(rot3)*Sin(rot2) + Sin(rot1)*Sin(rot3)),2) + Power(d1*Cos(rot2)*Sin(
```

__init__ (*dist=1*, *poni1=0*, *poni2=0*, *rot1=0*, *rot2=0*, *rot3=0*, *pixel1=None*, *pixel2=None*, *splineFile=None*, *detector=None*, *wavelength=None*)

Parameters

- **dist** – distance sample - detector plan (orthogonal distance, not along the beam), in meter.
- **poni1** – coordinate of the point of normal incidence along the detector's first dimension, in meter
- **poni2** – coordinate of the point of normal incidence along the detector's second dimension, in meter
- **rot1** – first rotation from sample ref to detector's ref, in radians
- **rot2** – second rotation from sample ref to detector's ref, in radians
- **rot3** – third rotation from sample ref to detector's ref, in radians
- **pixel1** (*float*) – Deprecated. Pixel size of the fist dimension of the detector, in meter. If both pixel1 and pixel2 are not None, detector pixel size is overwritten. Prefer defining the detector pixel size on the provided detector object. Prefer defining the detector pixel size on the provided detector object (`detector.pixel1 = 5e-6`).
- **pixel2** (*float*) – Deprecated. Pixel size of the second dimension of the detector, in meter. If both pixel1 and pixel2 are not None, detector pixel size is overwritten. Prefer defining the detector pixel size on the provided detector object (`detector.pixel2 = 5e-6`).
- **splineFile** (*str*) – Deprecated. File containing the geometric distortion of the detector. If not None, pixel1 and pixel2 are ignored and detector spline is overwritten. Prefer defining the detector spline manually (`detector.splineFile = "file.spline"`).
- **detector** (*str or pyFAI.Detector*) – name of the detector or Detector instance. String description is deprecated. Prefer using the result of the detector factory: `pyFAI.detector_factory("eiger4m")`
- **wavelength** (*float*) – Wave length used in meter

array_from_unit (*shape=None*, *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

calc_pos_zyx (*d0=None*, *d1=None*, *d2=None*, *param=None*, *corners=False*, *use_cython=True*)

Calculate the position of a set of points in space in the sample's centers referential.

This is usually used for calculating the pixel position in space.

Parameters

- **d0** – altitude on the point compared to the detector (i.e. z), may be None
- **d1** – position on the detector along the slow dimention (i.e. y)
- **d2** – position on the detector along the fastest dimention (i.e. x)
- **corners** – return positions on the corners (instead of center)

Returns 3-tuple of nd-array, with dim0=along the beam, dim1=along slowest dimension dim2=along fastest dimension

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

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

$$Icor = \frac{Iobs(1-t0)}{1-exp(ln(t0)/cos(incidence))}$$
$$let_t = \frac{1-exp(ln(t0)/cos(incidence))}{1-t0}$$

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

Parameters

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

Returns actual

calcfrom1d(*tth*, *I*, *shape=None*, *mask=None*, *dim1_unit=2th_deg*, *correctSolidAngle=True*, *dummy=0.0*, *polarization_factor=None*, *dark=None*, *flat=None*)

Computes a 2D image from a 1D integrated profile

Parameters

- **tth** – 1D array with radial unit
- **I** – scattering intensity
- **shape** – shape of the image (if not defined by the detector)
- **dim1_unit** – unit for the "tth" array
- **correctSolidAngle** –
- **dummy** – value for masked pixels
- **polarization_factor** – set to true to use previously used value
- **dark** – dark current correction
- **flat** – flatfield corrction

Returns 2D image reconstructed

center_array(*shape=None*, *unit='2th'*)

Generate a 2D array of the given shape with (i,j) (radial angle) for all elements.

Parameters **shape** (*2-tuple of integer*) – expected shape

Returns 3d array with shape=(*shape,4,2) the two elements are: - dim3[0]: radial angle 2th, q, r... - dim3[1]: azimuthal angle chi

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=None*)
Generate an array of azimuthal angle chi(i,j) for all elements in the detector.

Azimuthal angles are in radians

Nota: Refers to the pixel centers !

Parameters **shape** – the shape of the chi array

Returns the chi array as numpy.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(**arg*, ***kw*)
decorator that deprecates the use of a function

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

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

cornerRd2Array(**arg*, ***kw*)
decorator that deprecates the use of a function

corner_array(*shape=None*, *unit=None*, *use_cython=True*)
Generate a 3D array of the given shape with (i,j) (radial angle 2th, azimuthal angle chi) for all elements.

Parameters **shape** (*2-tuple of integer*) – expected shape

Returns 3d array with shape=(*shape,4,2) the two elements are: - dim3[0]: radial angle 2th, q, r... - dim3[1]: azimuthal angle chi

correct_SA_spline

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

Parameters

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

Returns cosine of the incidence angle

del_chia()

del_dssa()

del_qa()

del_ra()

del_ttha()

delta2Theta(*shape=None*)

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=None*)

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=None*)

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=None*)

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)

deltaRd2(*shape=None*)

Generate a 2D array of the given shape with (i,j) with the max distance between the center and any corner in unit: reciprocal spacing squarred (1/nm^2)

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

Returns array 2D containing the max delta (d*)^2 between a pixel center and any corner in reciprocal spacing squarred (1/nm^2)

delta_array(*shape=None*, *unit='2th'*)

Generate a 2D array of the given shape with (i,j) (delta-radial angle) for all elements.

Parameters **shape** (*2-tuple of integer*) – expected shape

Returns

3d array with shape=(*shape,4,2) the two elements are:

- dim3[0]: radial angle 2th, q, r...
- dim3[1]: azimuthal angle chi

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

$$dOmega = \frac{Omega(P)}{Omega(C)} = \frac{A \cdot cos(a)}{SP^2} \cdot \frac{SC^2}{A \cdot cos(0)} = \frac{3}{cos(a)} = \frac{SC^3}{SP^3}$$
$$cos(a) = \frac{SC}{SP}$$

Parameters

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

Returns solid angle correction array

dist

dssa
solid angle array in cache

getFit2D()
Export geometry setup with the geometry of Fit2D

Returns dict with parameters compatible with fit2D geometry

getPyFAI()
Export geometry setup with the geometry of PyFAI

Returns dict with the parameter-set of the PyFAI geometry

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

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

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

get_chia()

get_correct_solid_angle_for_spline()

get_dist()

get_dssa()

get_mask()

get_maskfile()

get_pixel1()

get_pixel2()

get_poni1()

get_poni2()

get_qa()

get_ra()

get_rot1()

get_rot2()

get_rot3()

get_shape(*shape=None*)
Guess what is the best shape ....

Parameters **shape** – force this value (2-tuple of int)

Returns 2-tuple of int

get_spline()

get_splineFile()

get_ttha()

get_wavelength()

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

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

mask

maskfile

oversampleArray(*myarray*)

pixel1

pixel2

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

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

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

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

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

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

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

Parameters

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

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

poni1

poni2

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

position_array(*shape=None*, *corners=False*, *dtype=<type 'numpy.float64'>*, *use_cython=True*)
Generate an array for the pixel position given the shape of the detector.

if corners is False, the coordinates of the center of the pixel is returned in an array of shape: (shape[0], shape[1], 3) where the 3 coordinates are: * z: along incident beam, * y: to the top/sky, * x: towards the center of the ring

If is True, the corner of each pixels are then returned. the output shape is then (shape[0], shape[1], 4, 3)

Parameters

- **shape** – shape of the array expected
- **corners** – set to true to receive a (...,4,3) array of corner positions
- **dtype** – output format requested. Double precision is needed for fitting the geometry
- **use_cython** (*(bool)*) – set to false to test the Python path (slower)

Returns 3D coodinates as nd-array of size (...,3) or (...,3) (default)

Nota: this value is not cached and actually generated on demand (costly)

qArray(*shape=None*)
Generate an array of the given shape with q(i,j) for all elements.

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

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=None*)
Generate an array of the given shape with r(i,j) for all elements; The radius r being in meters.

Parameters **shape** – expected shape of the detector

Returns 2d array of the given shape with radius in m from beam center on detector.

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

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

r = distance to the incident beam

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 m

Return type float or array of floats.

ra
R array in cache

rd2Array(*shape=None*)
Generate an array of the given shape with (d*(i,j))^2 for all pixels.

d*^2 is the reciprocal spacing squared in inverse nm squared

Parameters **shape** – expected shape of the detector

Returns 2d array of the given shape with reciprocal spacing squared

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 geometry parameters.

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

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

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

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

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

Parameters

- **direct** – direct distance from sample to detector along the incident beam (in millimeter as in fit2d)
- **tilt** – tilt in degrees
- **tiltPlanRotation** – Rotation (in degrees) of the tilt plan arround the Z-detector axis * 0deg -> Y does not move, +X goes to Z<0 * 90deg -> X does not move, +Y goes to Z<0 * 180deg -> Y does not move, +X goes to Z>0 * 270deg -> X does not move, +Y goes to Z>0
- **pixelX,pixelY** – as in fit2d they ar given in micron, not in meter
- **centerY** (*centerX,*) – pixel position of the beam center
- **splineFile** – name of the file containing the spline

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

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

setSPD(*SampleDistance*, *Center_1*, *Center_2*, *Rot_1=0*, *Rot_2=0*, *Rot_3=0*, *PSize_1=None*, *PSize_2=None*, *splineFile=None*, *BSize_1=1*, *BSize_2=1*, *WaveLength=None*)

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

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

Parameters

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

set_chia(_)

set_correct_solid_angle_for_spline(*value*)

set_dist(*value*)

set_dssa(_)

set_mask(*mask*)

set_maskfile(*maskfile*)

set_pixel1(*pixel1*)

set_pixel2(*pixel2*)

set_poni1(*value*)

set_poni2(*value*)

set_qa(_)

set_ra(_)

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=None*, *order=3*, *absolute=False*)
Generate an array for the solid angle correction given the shape of the detector.

solid_angle = cos(incidence)^3

Parameters

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

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

spline

splineFile

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

Parameters

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

Returns 2theta in radians

Return type float or array of floats.

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

ttha
2theta array in cache

twoThetaArray (*shape=None*)
Generate an array of two-theta(i,j) in radians for each pixel in detector

the 2theta array values are in radians

Parameters **shape** – shape of the detector

Returns array of 2theta position in radians

wavelength

write (*filename*)
Save the geometry parameters.

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

6.7 geometryRefinement Module

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

__init__(*data*, *dist=1*, *poni1=None*, *poni2=None*, *rot1=0*, *rot2=0*, *rot3=0*, *pixel1=None*, *pixel2=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)

Parameters

- **data** – ndarray float64 shape = n, 3 col0: pos in dim0 (in pixels) col1: pos in dim1 (in pixels) col2: ring index in calibrant object
- **dist** – guessed sample-detector distance (optional, in m)
- **poni1** – guessed PONI coordinate along the Y axis (optional, in m)
- **poni2** – guessed PONI coordinate along the X axis (optional, in m)
- **rot1** – guessed tilt of the detector around the Y axis (optional, in rad)
- **rot2** – guessed tilt of the detector around the X axis (optional, in rad)
- **rot3** – guessed tilt of the detector around the incoming beam axis (optional, in rad)
- **pixel1** – Pixel size along the vertical direction of the detector (in m), almost mandatory
- **pixel2** – Pixel size along the horizontal direction of the detector (in m), almost mandatory
- **splineFile** – file describing the detector as 2 cubic splines. Replaces pixel1 & pixel2
- **detector** – name of the detector or Detector instance. Replaces splineFile, pixel1 & pixel2
- **wavelength** – wavelength in m (1.54e-10)
- **calibrant** – instance of pyFAI.calibrant.Calibrant containing the d-Spacing

anneal(*maxiter=1000000*)

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

Parameters

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

chi2(*param=None*)

chi2_wavelength(*param=None*)

confidence(*with_rot=True*)

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

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

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

Returns std_dev, confidence

curve_fit(*with_rot=True*)

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

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

Returns std_dev, confidence

dist_max

dist_min

get_dist_max()

get_dist_min()

get_poni1_max()

get_poni1_min()

get_poni2_max()

get_poni2_min()

get_rot1_max()

get_rot1_min()

get_rot2_max()

get_rot2_min()

get_rot3_max()

get_rot3_min()

get_wavelength_max()

get_wavelength_min()

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

poni1_max

poni1_min

poni2_max

poni2_min

refine1()

refine2(*maxiter=1000000*, *fix=None*)

refine2_wavelength(*maxiter=1000000*, *fix=None*)

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

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

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

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

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

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

roca()
run roca to optimise the parameter set

rot1_max

rot1_min

rot2_max

rot2_min

rot3_max

rot3_min

set_dist_max(*value*)

set_dist_min(*value*)

set_poni1_max(*value*)

set_poni1_min(*value*)

set_poni2_max(*value*)

set_poni2_min(*value*)

set_rot1_max(*value*)

set_rot1_min(*value*)

set_rot2_max(*value*)

set_rot2_min(*value*)

set_rot3_max(*value*)

set_rot3_min(*value*)

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

Parameters **value** – Tolerance as a percentage

set_wavelength_max(*value*)

set_wavelength_min(*value*)

simplex(*maxiter=1000000*)

wavelength_max

wavelength_min

6.8 detectors Module

Description of all detectors with a factory to instantiate them

class pyFAI.detectors.**ADSC_Q210**(*pixel1=5.1e-05*, *pixel2=5.1e-05*)
Bases: pyFAI.detectors.Detector

ADSC Quantum 210r detector, 2x2 chips

Informations from http://www.adsc-xray.com/products/ccd-detectors/q210r-ccd-detector/

Question: how are the gaps handled ?

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=5.1e-05*, *pixel2=5.1e-05*)

aliases = ['Quantum 210']

force_pixel = True

class pyFAI.detectors.**ADSC_Q270**(*pixel1=6.48e-05*, *pixel2=6.48e-05*)
Bases: pyFAI.detectors.Detector

ADSC Quantum 270r detector, 2x2 chips

Informations from http://www.adsc-xray.com/products/ccd-detectors/q270-ccd-detector/

Question: how are the gaps handled ?

MAX_SHAPE = (4168, 4168)

__init__(*pixel1=6.48e-05*, *pixel2=6.48e-05*)

aliases = ['Quantum 270']

force_pixel = True

class pyFAI.detectors.**ADSC_Q315**(*pixel1=5.1e-05*, *pixel2=5.1e-05*)
Bases: pyFAI.detectors.Detector

ADSC Quantum 315r detector, 3x3 chips

Informations from http://www.adsc-xray.com/products/ccd-detectors/q315r-ccd-detector/

Question: how are the gaps handled ?

MAX_SHAPE = (6144, 6144)

__init__ (*pixel1=5.1e-05*, *pixel2=5.1e-05*)

aliases = ['Quantum 315']

force_pixel = True

class pyFAI.detectors.**ADSC_Q4** (*pixel1=8.2e-05*, *pixel2=8.2e-05*)
Bases: pyFAI.detectors.Detector

ADSC Quantum 4r detector, 2x2 chips

Informations from http://proteincrystallography.org/detectors/adsc.php

Question: how are the gaps handled ?

MAX_SHAPE = (2304, 2304)

__init__ (*pixel1=8.2e-05*, *pixel2=8.2e-05*)

aliases = ['Quantum 4']

force_pixel = True

class pyFAI.detectors.**Aarhus** (*pixel1=2.5e-05*, *pixel2=2.5e-05*, *radius=0.3*)
Bases: pyFAI.detectors.Detector

Cylindrical detector made of a bent imaging-plate. Developped at the Danish university of Aarhus r = 1.2m or 0.3m

The image has to be laid-out horizontally

Nota: the detector is bending towards the sample, hence reducing the sample-detector distance. This is why z<0 (or p3<0)

TODO: update cython code for 3d detectors use expand2d instead of outer product with ones

IS_FLAT = False

MAX_SHAPE = (1000, 16000)

__init__ (*pixel1=2.5e-05*, *pixel2=2.5e-05*, *radius=0.3*)

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

Parameters

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

Returns position in meter of the center of each pixels.

Return type ndarray

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

force_pixel = True

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

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

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

class pyFAI.detectors.**Apex2**(*pixel1=0.00012*, *pixel2=0.00012*)
Bases: pyFAI.detectors.Detector

BrukerApex2 detector

Actually a derivative from the Fairchild detector with higher binning

DEFAULT_PIXEL1 = 6e-05

DEFAULT_PIXEL2 = 6e-05

MAX_SHAPE = (1024, 1024)

__init__(*pixel1=0.00012*, *pixel2=0.00012*)
Defaults to 2x2 binning

aliases = ['ApexII', 'Bruker']

force_pixel = True

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

Basler camera are simple CCD camara over GigaE

MAX_SHAPE = (966, 1296)

__init__(*pixel=3.75e-06*)

aliases = ['aca1300']

force_pixel = True

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

Generic class representing a 2D detector

API_VERSION = '1.0'

IS_CONTIGUOUS = True

IS_FLAT = True

__init__(*pixel1=None*, *pixel2=None*, *splineFile=None*, *max_shape=None*)

Parameters

- **pixel1** (*float*) – size of the pixel in meter along the slow dimension (often Y)
- **pixel2** (*float*) – size of the pixel in meter along the fast dimension (often X)
- **splineFile** (*str*) – path to file containing the geometric correction.
- **max_shape** (*2-tuple of integrers*) – maximum size of the detector

aliases = []

binning

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

Parameters

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

- **center** – retrieve the coordinate of the center of the pixel, unless gives one corner
- **use_cython** – set to False to test Python implementation

Returns position in meter of the center of each pixels.

Return type 3xndarray, the later being None if IS_FLAT

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

pos_z is None for flat detectors

calc_mask()
Method calculating the mask for a given detector

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

Returns the mask with valid pixel to 0

Return type numpy ndarray of int8 or None

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

Parameters

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

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

Return type pyFAI.detectors.Detector

force_pixel = False

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

Returns representation of the detector easy to serialize

Return type dict

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

Returns representation of the detector easy to serialize

Return type dict

get_binning()

get_mask()

get_maskfile()

get_name()
Get a meaningful name for detector

get_pixel1()

get_pixel2()

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

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

Precision float32 is ok: precision of 1µm for a detector size of 1m

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

get_splineFile()

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

mask

maskfile

name
Get a meaningful name for detector

pixel1

pixel2

registry = {'pilatus1m_cdte': <class 'pyFAI.detectors.PilatusCdTe1M'>, 'imxpads10': <class 'pyFAI.detectors.ImX

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

•differentiate pixel center from pixel corner offsets

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

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

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

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

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

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

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

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

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

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

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

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

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

set_mask(*mask*)

set_maskfile(*maskfile*)

set_pixel1(*value*)

set_pixel2(*value*)

set_splineFile(*splineFile*)

splineFile

uniform_pixel = True

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

Metaclass used to register all detector classes inheriting from Detector

__init__(*name*, *bases*, *dct*)

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

Dexela CMOS family detector

MAX_SHAPE = (3888, 3072)

__init__(*pixel1=7.5e-05*, *pixel2=7.5e-05*)

aliases = ['Dexela 2923']

force_pixel = True

class pyFAI.detectors.**Eiger**(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Detector

Eiger detector: generic description containing mask algorithm

MODULE_GAP = (37, 10)

MODULE_SIZE = (514, 1030)

__init__(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)

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

Parameters

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

Returns p1, p2 position in meter of the center of each pixels.

Return type 2-tuple of numpy.ndarray

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

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

force_pixel = True

class pyFAI.detectors.**Eiger16M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Eiger

Eiger 16M detector

MAX_SHAPE = (4371, 4150)

aliases = ['Eiger 16M']

class pyFAI.detectors.**Eiger1M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Eiger

Eiger 1M detector

MAX_SHAPE = (1065, 1030)

aliases = ['Eiger 1M']

class pyFAI.detectors.**Eiger4M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Eiger

Eiger 4M detector

MAX_SHAPE = (2167, 2070)

aliases = ['Eiger 4M']

class pyFAI.detectors.**Eiger500k**(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Eiger

Eiger 1M detector

MAX_SHAPE = (512, 1030)

aliases = ['Eiger 500k']

class pyFAI.detectors.**Eiger9M**(*pixel1=7.5e-05*, *pixel2=7.5e-05*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Eiger

Eiger 9M detector

MAX_SHAPE = (3269, 3110)

aliases = ['Eiger 9M']

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"

__init__(*splineFile=None*)

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

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

Fairchild Condor 486:90 detector

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=1.5e-05*, *pixel2=1.5e-05*)

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

force_pixel = True

uniform_pixel = True

class pyFAI.detectors.**HF_130K**(*pixel1=0.00015*, *pixel2=0.00015*)
Bases: pyFAI.detectors.Detector

ADSC HF-130K 1 module

Informations from http://www.adsc-xray.com/products/pixel-array-detectors/hf-130k/

MAX_SHAPE = (256, 512)

__init__(*pixel1=0.00015*, *pixel2=0.00015*)

aliases = ['HF-130k']

force_pixel = True

class pyFAI.detectors.**HF_1M**(*pixel1=0.00015*, *pixel2=0.00015*)
Bases: pyFAI.detectors.Detector

ADSC HF-1M 2x4 modules

Informations from http://www.adsc-xray.com/products/pixel-array-detectors/hf-1m/

Nota: gaps between modules is not known/described

MAX_SHAPE = (1024, 1024)

__init__(*pixel1=0.00015*, *pixel2=0.00015*)

aliases = ['HF-1M']

force_pixel = True

class pyFAI.detectors.**HF_262k**(*pixel1=0.00015*, *pixel2=0.00015*)
Bases: pyFAI.detectors.Detector

ADSC HF-262k 2 module

Informations from http://www.adsc-xray.com/products/pixel-array-detectors/hf-262k/

Nota: gaps between modules is not known/described

MAX_SHAPE = (512, 512)

__init__(*pixel1=0.00015*, *pixel2=0.00015*)

aliases = ['HF-262k']

force_pixel = True

class pyFAI.detectors.**HF_2M**(*pixel1=0.00015*, *pixel2=0.00015*)
Bases: pyFAI.detectors.Detector

ADSC HF-1M 3x6 modules

Informations from http://www.adsc-xray.com/products/pixel-array-detectors/hf-2.4m/

Nota: gaps between modules is not known/described

MAX_SHAPE = (1536, 1536)

__init__(*pixel1=0.00015*, *pixel2=0.00015*)

aliases = ['HF-2.4M']

force_pixel = True

class pyFAI.detectors.**HF_4M**(*pixel1=0.00015*, *pixel2=0.00015*)
Bases: pyFAI.detectors.Detector

ADSC HF-4M 4x8 modules

Informations from http://www.adsc-xray.com/products/pixel-array-detectors/hf-4m/

MAX_SHAPE = (2048, 2048)

__init__(*pixel1=0.00015*, *pixel2=0.00015*)

aliases = ['HF-4M']

force_pixel = True

class pyFAI.detectors.**HF_9M**(*pixel1=0.00015*, *pixel2=0.00015*)
Bases: pyFAI.detectors.Detector

ADSC HF-130K 1 module

Informations from http://www.adsc-xray.com/products/pixel-array-detectors/hf-9-4m/

MAX_SHAPE = (3072, 3072)

__init__(*pixel1=0.00015*, *pixel2=0.00015*)

aliases = ['HF-9.4M']

force_pixel = True

class pyFAI.detectors.**ImXPadS10**(*pixel1=0.00013*, *pixel2=0.00013*, *max_shape=None*, *module_size=None*)
Bases: pyFAI.detectors.Detector

ImXPad detector: ImXPad s10 detector with 1x1modules

BORDER_SIZE_RELATIVE = 2.5

MAX_SHAPE = (120, 80)

MODULE_SIZE = (120, 80)

PIXEL_SIZE = (0.00013, 0.00013)

__init__(*pixel1=0.00013*, *pixel2=0.00013*, *max_shape=None*, *module_size=None*)

aliases = ['Imxpad S10']

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

Parameters

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

Returns position in meter of the center of each pixels.

Return type ndarray

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

calc_mask()
Calculate the mask

calc_pixels_edges()
Calculate the position of the pixel edges

force_pixel = True

get_pixel_corners(*d1=None*, *d2=None*)
Calculate the position of the corner of the pixels

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

Precision float32 is ok: precision of 1µm for a detector size of 1m

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

uniform_pixel = False

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

ImXPad detector: ImXPad s140 detector with 2x7modules

BORDER_PIXEL_SIZE_RELATIVE = 2.5

MAX_SHAPE = (240, 560)

MODULE_SIZE = (120, 80)

PIXEL_SIZE = (0.00013, 0.00013)

__init__(*pixel1=0.00013*, *pixel2=0.00013*)

aliases = ['Imxpad S140']

force_pixel = True

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

ImXPad detector: ImXPad s70 detector with 1x7modules

BORDER_SIZE_RELATIVE = 2.5

MAX_SHAPE = (120, 560)

MODULE_SIZE = (120, 80)

PIXEL_EDGES = None

PIXEL_SIZE = (0.00013, 0.00013)

__init__(*pixel1=0.00013*, *pixel2=0.00013*)

aliases = ['Imxpad S70']

force_pixel = True

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

Mar345 Imaging plate detector

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

MAX_SHAPE = (3450, 3450)

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

__init__(*pixel1=0.0001*, *pixel2=0.0001*)

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

calc_mask()

force_pixel = True

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

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

Class representing a 2D detector loaded from a NeXus file

__init__(*filename=None*)

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

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

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

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

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

Returns Detector instance

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

Perkin detector

DEFAULT_PIXEL1 = 0.0002

DEFAULT_PIXEL2 = 0.0002

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=0.0002*, *pixel2=0.0002*)

aliases = ['Perkin detector', 'Perkin Elmer']

force_pixel = True

class pyFAI.detectors.**Pilatus**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *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)

__init__(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

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

Parameters

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

Returns position in meter of the center of each pixels.

Return type ndarray

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

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

force_pixel = True

get_splineFile()

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

splineFile

class pyFAI.detectors.**Pilatus100k**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)
Bases: pyFAI.detectors.Pilatus

Pilatus 100k detector

MAX_SHAPE = (195, 487)

aliases = ['Pilatus 100k']

class pyFAI.detectors.**Pilatus1M**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus 1M detector

MAX_SHAPE = (1043, 981)

aliases = ['Pilatus 1M']

class pyFAI.detectors.**Pilatus200k**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus 200k detector

MAX_SHAPE = (407, 487)

aliases = ['Pilatus 200k']

class pyFAI.detectors.**Pilatus2M**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus 2M detector

MAX_SHAPE = (1679, 1475)

aliases = ['Pilatus 2M']

class pyFAI.detectors.**Pilatus300k**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus 300k detector

MAX_SHAPE = (619, 487)

aliases = ['Pilatus 300k']

class pyFAI.detectors.**Pilatus300kw**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus 300k-wide detector

MAX_SHAPE = (195, 1475)

aliases = ['Pilatus 300kw']

class pyFAI.detectors.**Pilatus6M**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus 6M detector

MAX_SHAPE = (2527, 2463)

aliases = ['Pilatus 6M']

class pyFAI.detectors.**PilatusCdTe**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)

Bases: pyFAI.detectors.Pilatus

Pilatus CdTe detector: Like the Pilatus with an extra 3 pixel in the middle of every module (vertically)

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

class pyFAI.detectors.**PilatusCdTe1M**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)
Bases: pyFAI.detectors.PilatusCdTe

Pilatus CdTe 1M detector

MAX_SHAPE = (1043, 981)

aliases = ['Pilatus CdTe 1M', 'Pilatus 1M CdTe', 'Pilatus1M CdTe', 'Pilatus1MCdTe']

class pyFAI.detectors.**PilatusCdTe2M**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)
Bases: pyFAI.detectors.PilatusCdTe

Pilatus CdTe 2M detector

MAX_SHAPE = (1679, 1475)

aliases = ['Pilatus CdTe 2M', 'Pilatus 2M CdTe', 'Pilatus2M CdTe', 'Pilatus2MCdTe']

class pyFAI.detectors.**PilatusCdTe300k**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)
Bases: pyFAI.detectors.PilatusCdTe

Pilatus CdTe 300k detector

MAX_SHAPE = (619, 487)

aliases = ['Pilatus CdTe 300k', 'Pilatus 300k CdTe', 'Pilatus300k CdTe', 'Pilatus300kCdTe']

class pyFAI.detectors.**PilatusCdTe300kw**(*pixel1=0.000172*, *pixel2=0.000172*, *max_shape=None*, *module_size=None*, *x_offset_file=None*, *y_offset_file=None*)
Bases: pyFAI.detectors.PilatusCdTe

Pilatus CdTe 300k-wide detector

MAX_SHAPE = (195, 1475)

aliases = ['Pilatus CdTe 300kw', 'Pilatus 300kw CdTe', 'Pilatus300kw CdTe', 'Pilatus300kwCdTe']

class pyFAI.detectors.**Pixium**(*pixel1=0.000308*, *pixel2=0.000308*)
Bases: pyFAI.detectors.Detector

PIXIUM 4700 detector

High energy X ray diffraction using the Pixium 4700 flat panel detector J E Daniels, M Drakopoulos, et al.; Journal of Synchrotron Radiation 16(Pt 4):463-8 · August 2009

DEFAULT_PIXEL1 = 0.000154

DEFAULT_PIXEL2 = 0.000154

MAX_SHAPE = (1910, 2480)

__init__(*pixel1=0.000308*, *pixel2=0.000308*)
Defaults to 2x2 binning

aliases = ['Pixium 4700 detector', 'Thales Electronics']

force_pixel = True

class pyFAI.detectors.**RaspberryPi5M**(*pixel1=1.4e-06*, *pixel2=1.4e-06*)
Bases: pyFAI.detectors.Detector

5 Mpix detector from Raspberry Pi

MAX_SHAPE = (1944, 2592)

__init__(*pixel1=1.4e-06*, *pixel2=1.4e-06*)

aliases = ['Picam v1']

force_pixel = True

class pyFAI.detectors.**RaspberryPi8M**(*pixel1=1.12e-06*, *pixel2=1.12e-06*)
Bases: pyFAI.detectors.Detector

8 Mpix detector from Raspberry Pi

MAX_SHAPE = (2464, 3280)

__init__(*pixel1=1.12e-06*, *pixel2=1.12e-06*)

aliases = ['Picam v2']

force_pixel = True

class pyFAI.detectors.**Rayonix**(*pixel1=3.2e-05*, *pixel2=3.2e-05*)
Bases: pyFAI.detectors.Detector

BINNED_PIXEL_SIZE = {1: 3.2e-05}

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=3.2e-05*, *pixel2=3.2e-05*)

binning

force_pixel = True

get_binning()

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

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

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

class pyFAI.detectors.**Rayonix133**(*pixel1=6.4e-05*, *pixel2=6.4e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix 133 2D CCD detector detector also known as mar133

Personnal communication from M. Blum

What should be the default binning factor for those cameras ?

Circular detector

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

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=6.4e-05*, *pixel2=6.4e-05*)

aliases = ['MAR133']

calc_mask()
Circular mask

force_pixel = True

class pyFAI.detectors.**RayonixLx170**(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)
Bases: pyFAI.detectors.Rayonix

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

Nota: this is the same for lx170hs

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

MAX_SHAPE = (1920, 3840)

__init__(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)

aliases = ['Rayonix LX170', 'Rayonix LX170-HS', 'Rayonix LX170 HS', 'RayonixLX170HS']

force_pixel = True

class pyFAI.detectors.**RayonixLx255**(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix lx255 2d Detector (3x1 CCDs)

Nota: this detector is also called lx255hs

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

MAX_SHAPE = (1920, 5760)

__init__(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)

aliases = ['Rayonix LX255', 'Rayonix LX255-HS', 'Rayonix LX 255HS', 'RayonixLX225HS']

class pyFAI.detectors.**RayonixMx170**(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)
Bases: pyFAI.detectors.Rayonix

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

Nota: this is the same for mx170hs

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

MAX_SHAPE = (3840, 3840)

__init__(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)

aliases = ['Rayonix MX170', 'Rayonix MX170-HS', 'RayonixMX170HS', 'Rayonix MX170 HS']

class pyFAI.detectors.**RayonixMx225**(*pixel1=7.3242e-05*, *pixel2=7.3242e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix mx225 2D CCD detector detector

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

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

MAX_SHAPE = (6144, 6144)

__init__(*pixel1=7.3242e-05*, *pixel2=7.3242e-05*)

aliases = ['Rayonix MX225']

force_pixel = True

class pyFAI.detectors.**RayonixMx225hs**(*pixel1=7.8125e-05*, *pixel2=7.8125e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix mx225hs 2D CCD detector detector

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (5760, 5760)

__init__(*pixel1=7.8125e-05*, *pixel2=7.8125e-05*)

aliases = ['Rayonix MX225HS', 'Rayonix MX225 HS']

force_pixel = True

class pyFAI.detectors.**RayonixMx300**(*pixel1=7.3242e-05*, *pixel2=7.3242e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix mx300 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (8192, 8192)

__init__(*pixel1=7.3242e-05*, *pixel2=7.3242e-05*)

aliases = ['Rayonix mx300']

force_pixel = True

class pyFAI.detectors.**RayonixMx300hs**(*pixel1=7.8125e-05*, *pixel2=7.8125e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix mx300hs 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (7680, 7680)

__init__(*pixel1=7.8125e-05*, *pixel2=7.8125e-05*)

aliases = ['Rayonix MX300HS', 'Rayonix MX300 HS']

force_pixel = True

class pyFAI.detectors.**RayonixMx325**(*pixel1=7.9346e-05*, *pixel2=7.9346e-05*)
Bases: pyFAI.detectors.Rayonix

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

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (8192, 8192)

__init__(*pixel1=7.9346e-05*, *pixel2=7.9346e-05*)

aliases = ['Rayonix MX325']

class pyFAI.detectors.**RayonixMx340hs**(*pixel1=8.85417e-05*, *pixel2=8.85417e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix mx340hs 2D detector (4x4 CCDs)

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (7680, 7680)

__init__(*pixel1=8.85417e-05*, *pixel2=8.85417e-05*)

aliases = ['Rayonix MX340HS', 'Rayonix MX340HS']

force_pixel = True

class pyFAI.detectors.**RayonixMx425hs**(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)
Bases: pyFAI.detectors.Rayonix

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

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (9600, 9600)

__init__(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)

aliases = ['Rayonix MX425HS', 'Rayonix MX425 HS']

class pyFAI.detectors.**RayonixSx165**(*pixel1=3.95e-05*, *pixel2=3.95e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix sx165 2d Detector also known as MAR165.

Circular detector

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

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=3.95e-05*, *pixel2=3.95e-05*)

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

calc_mask()
Circular mask

force_pixel = True

class pyFAI.detectors.**RayonixSx200**(*pixel1=4.8e-05*, *pixel2=4.8e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix sx200 2d CCD Detector.

Pixel size are personnal communication from M. Blum.

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

MAX_SHAPE = (4096, 4096)

__init__(*pixel1=4.8e-05*, *pixel2=4.8e-05*)

aliases = ['Rayonix sx200']

class pyFAI.detectors.**RayonixSx30hs**(*pixel1=1.5625e-05*, *pixel2=1.5625e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix sx30hs 2D CCD camera (1 CCD chip)

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (1920, 1920)

__init__(*pixel1=1.5625e-05*, *pixel2=1.5625e-05*)

aliases = ['Rayonix SX30HS', 'Rayonix SX30 HS']

class pyFAI.detectors.**RayonixSx85hs**(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)
Bases: pyFAI.detectors.Rayonix

Rayonix sx85hs 2D CCD camera (1 CCD chip)

Pixel size from a personnal communication from M. Blum

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

MAX_SHAPE = (1920, 1920)

__init__(*pixel1=4.42708e-05*, *pixel2=4.42708e-05*)

aliases = ['Rayonix SX85HS', 'Rayonix SX85 HS']

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

Bases: pyFAI.detectors.Detector

Titan CCD detector from Agilent. Mask not handled

MAX_SHAPE = (2048, 2048)

__init__(*pixel1=6e-05*, *pixel2=6e-05*)

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

force_pixel = True

uniform_pixel = True

class pyFAI.detectors.**Xpad_flat**(*pixel1=0.00013*, *pixel2=0.00013*, *max_shape=None*, *module_size=None*)

Bases: pyFAI.detectors.ImXPadS10

Xpad detector: generic description for ImXPad detector with 8x7modules

BORDER_PIXEL_SIZE_RELATIVE = 2.5

IS_CONTIGUOUS = False

MAX_SHAPE = (960, 560)

MODULE_GAP = (0.00357, 0)

MODULE_SIZE = (120, 80)

PIXEL_SIZE = (0.00013, 0.00013)

__init__(*pixel1=0.00013*, *pixel2=0.00013*, *max_shape=None*, *module_size=None*)

aliases = ['Xpad S540 flat', 'd5']

calc_cartesian_positions(*d1=None*, *d2=None*, *center=True*, *use_cython=True*)

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

Parameters

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

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

calc_pixels_edges()

Calculate the position of the pixel edges, specific to the S540, d5 detector

force_pixel = True

get_pixel_corners()

Calculate the position of the corner of the pixels

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

uniform_pixel = False

6.9 spline Module

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

Mainly used at ESRF with FReLoN CCD camera.

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

This class is a python representation of the spline file

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

__init__(*filename=None*)
This is the constructor of the Spline class.

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

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

Parameters

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

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

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

Type int or (int, int)

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

Parameters

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

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

Return type bool

correct(*pos*)

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

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

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

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

Returns the size of the pixel from a 2D detector

Return type 2-tuple of floats expressed in meter.

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

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

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

Param pixel size in meter

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

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

Returns Nothing !

Return type float or ndarray

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

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

Parameters

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

Returns displacement matrix for the X direction

Return type ndarray

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

Parameters

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

Returns displacement matrix for the Y direction

Return type ndarray

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

Parameters

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

Returns tilted Spline instance

Return type Spline

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

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

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

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

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

Parameters

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

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

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

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

6.10 opencl Module

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

Simple class that contains the structure of an OpenCL device

__init__(*name='None'*, *dtype=None*, *version=None*, *driver_version=None*, *extensions=''*, *memory=None*, *available=None*, *cores=None*, *frequency=None*, *flop_core=None*, *idx=0*, *workgroup=1*)
Simple container with some important data for the OpenCL device description:

Parameters

- **name** – name of the device
- **dtype** – device type: CPU/GPU/ACC...
- **version** – driver version
- **driver_version** –
- **extensions** – List of opencl extensions
- **memory** – maximum memory available on the device
- **available** – is the device desactivated or not
- **cores** – number of SM/cores
- **frequency** – frequency of the device
- **flop_cores** – Flopating Point operation per core per cycle
- **idx** – index of the device within the platform

- **workgroup** – max workgroup size

pretty_print()
Complete device description

Returns string

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

Simple class that wraps the structure ocl_tools_extended.h

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

comput_cap = (5, 0)

context_cache = {}

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

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

device_from_context(*context*)
Retrieves the Device from the context

Parameters **context** – OpenCL context

Returns instance of Device

flop_core = 4

get_platform(*key*)
Return a platform according

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

idd = 0

idx = 2

nb_devices = 4

platforms = [Portable Computing Language, NVIDIA CUDA, Intel(R) OpenCL]

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

Parameters

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

workgroup = 8192

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

Simple class that contains the structure of an OpenCL platform

__init__ (*name='None'*, *vendor='None'*, *version=None*, *extensions=None*, *idx=0*)
Class containing all descriptions of a platform and all devices description within that platform.

Parameters

- **name** – platform name
- **vendor** – name of the brand/vendor
- **version** –
- **extension** – list of the extension provided by the platform to all of its devices
- **idx** – index of the platform

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

Parameters **device** – Device instance

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

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

pyFAI.opencl.**allocate_cl_buffers** (*buffers*, *device=None*, *context=None*)

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

Returns a dict containing the instanciated pyopencl.Buffer

Return type dict(str, pyopencl.Buffer)

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

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

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

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

6.11 ocl_azim Module

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

TODO and trick from dimitris still missing:

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

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

Attempt to implements ocl_azim using pyopencl

BLOCK_SIZE = 128

__init__ (*filename=None*)

Parameters **filename** – file in which profiling information are saved

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

6.12 ocl_azim_lut Module

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

Bases: object

BLOCK_SIZE = 16

__init__(*lut*, *image_size*, *devicetype='all'*, *platformid=None*, *deviceid=None*, *checksum=None*, *profile=False*, *empty=None*)

Parameters

- **lut** – array of int32 - float32 with shape (nbins, lut_size) with indexes and coefficients
- **image_size** – Expected image size: image.shape.prod()
- **devicetype** – can be "cpu","gpu","acc" or "all"
- **platformid** (*int*) – number of the platform as given by clinfo
- **deviceid** (*int*) – number of the device as given by clinfo
- **checksum** – pre - calculated checksum to prevent re - calculating it :)
- **profile** – store profiling elements
- **empty** – value to be assigned to bins without contribution from any pixel

integrate(*data*, *dummy=None*, *delta_dummy=None*, *dark=None*, *flat=None*, *solidAngle=None*, *polarization=None*, *dark_checksum=None*, *flat_checksum=None*, *solidAngle_checksum=None*, *polarization_checksum=None*, *preprocess_only=False*, *safe=True*, *normalization_factor=1.0*)

Before performing azimuthal integration, the preprocessing is:

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

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

Parameters

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

Returns averaged data, weighted histogram, unweighted histogram

log_profile()

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

6.13 ocl_azim_csr Module

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

Bases: object

__init__(*lut*, *image_size*, *devicetype='all'*, *block_size=32*, *platformid=None*, *deviceid=None*, *checksum=None*, *profile=False*, *empty=None*)

Parameters

- **lut** – 3-tuple of arrays data: coefficient of the matrix in a 1D vector of float32 - size of nnz indices: Column index position for the data (same size as data) indptr: row pointer indicates the start of a given row. len nbin+1
- **image_size** – size of the image (for pre-processing)
- **devicetype** – can be "cpu","gpu","acc" or "all"
- **block_size** – the chosen size for WORKGROUP_SIZE
- **platformid** (*int*) – number of the platform as given by clinfo
- **deviceid** (*int*) – number of the device as given by clinfo
- **checksum** – pre - calculated checksum to prevent re - calculating it :)
- **profile** – store profiling elements
- **empty** – value to be assigned to bins without contribution from any pixel

integrate(*data*, *dummy=None*, *delta_dummy=None*, *dark=None*, *flat=None*, *solidAngle=None*, *polarization=None*, *dark_checksum=None*, *flat_checksum=None*, *solidAngle_checksum=None*, *polarization_checksum=None*, *preprocess_only=False*, *safe=True*, *normalization_factor=1.0*)

Before performing azimuthal integration, the preprocessing is:

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

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

Parameters

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

Returns averaged data, weighted histogram, unweighted histogram

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

6.14 ocl_azim_csr_dis Module

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

Bases: object

__init__(*lut*, *image_size*, *devicetype='all'*, *block_size=32*, *platformid=None*, *deviceid=None*, *checksum=None*, *profile=False*, *empty=None*)

Parameters

- **lut** – 3-tuple of arrays data: coefficient of the matrix in a 1D vector of float32 - size of nnz indices: Column index position for the data (same size as data) indptr: row pointer indicates the start of a given row. len nbin+1
- **image_size** –
- **devicetype** – can be "cpu","gpu","acc" or "all"
- **block_size** – the chosen size for WORKGROUP_SIZE
- **platformid** (*int*) – number of the platform as given by clinfo
- **deviceid** (*int*) – number of the device as given by clinfo
- **checksum** – pre - calculated checksum to prevent re - calculating it :)
- **profile** – store profiling elements
- **empty** – value to be assigned to bins without contribution from any pixel

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

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

6.15 io Module

Module for "high-performance" writing in either 1D with Ascii , or 2D with FabIO or even nD with n varying from 2 to 4 using HDF5

Stand-alone module which tries to offer interface to HDF5 via H5Py and capabilities to write EDF or other formats using fabio.

Can be imported without h5py but then limited to fabio & ascii formats.

TODO:

- Add monitor to HDF5

class pyFAI.io.**AsciiWriter**(*filename=None*, *prefix='fai_'*, *extension='.dat'*)

Bases: pyFAI.io.Writer

Ascii file writer (.xy or .dat)

__init__(*filename=None*, *prefix='fai_'*, *extension='.dat'*)

init(*fai_cfg=None*, *lima_cfg=None*)
Creates the directory that will host the output file(s)

write(*data*, *index=0*)

class pyFAI.io.**DefaultAiWriter**(*filename*, *ai*)
Bases: pyFAI.io.Writer

__init__(*filename*, *ai*)
Constructor of the historical writer of azimuthalIntegrator.

close()

flush()

make_headers(*hdr='#'*, *has_dark=False*, *has_flat=False*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

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

Returns the header

Return type str

save1D(*filename*, *dim1*, *I*, *error=None*, *dim1_unit='2th_deg'*, *has_dark=False*, *has_flat=False*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

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

This method save the result of a 1D integration.

save2D(*filename*, *I*, *dim1*, *dim2*, *error=None*, *dim1_unit='2th_deg'*, *has_dark=False*, *has_flat=False*, *polarization_factor=None*, *normalization_factor=None*)

Parameters

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

- **normalization_factor** (*float, None*) – the monitor value

This method save the result of a 2D integration.

set_filename (*filename*)
Define the filename while will be used

write (*data*)
Minimalistic method to limit the overhead.

Parameters data – array with intensities or tuple (2th,I) or (I,2th,chi) :type data: Integrate1dResult, Integrate2dResult

class pyFAI.io.**FabioWriter** (*filename=None*)
Bases: pyFAI.io.Writer

Image file writer based on FabIO

TODO !!!

__init__ (*filename=None*)

init (*fai_cfg=None*, *lima_cfg=None*)
Creates the directory that will host the output file(s)

write (*data*, *index=0*)

class pyFAI.io.**HDF5Writer** (*filename*, *hpath='data'*, *fast_scan_width=None*)
Bases: pyFAI.io.Writer

Class allowing to write HDF5 Files.

CONFIG = 'pyFAI'

DATASET_NAME = 'data'

__init__ (*filename*, *hpath='data'*, *fast_scan_width=None*)
Constructor of an HDF5 writer:

Parameters

- **filename** – name of the file
- **hpath** – name of the group: it will contain data (2-4D dataset), [tth|q|r] and pyFAI, group containing the configuration
- **fast_scan_width** – set it to define the width of

close ()

flush (*radial=None*, *azimuthal=None*)
Update some data like axis units and so on.

Parameters

- **radial** – position in radial direction
- **azimuthal** – position in azimuthal direction

init (*fai_cfg=None*, *lima_cfg=None*)
Initializes the HDF5 file for writing :param fai_cfg: the configuration of the worker as a dictionary

write (*data*, *index=0*)
Minimalistic method to limit the overhead. :param data: array with intensities or tuple (2th,I) or (I,2th,chi)

class pyFAI.io.**Nexus** (*filename*, *mode='r'*)
Bases: object

Writer class to handle Nexus/HDF5 data

Manages:

- entry
  - pyFAI-subentry
    - detector

TODO: make it thread-safe !!!

__init__(*filename*, *mode='r'*)
Constructor

Parameters

- **filename** – name of the hdf5 file containing the nexus
- **mode** – can be r or a

close()
close the filename and update all entries

deep_copy(*name*, *obj*, *where='/'*, *toplevel=None*, *excluded=None*, *overwrite=False*)
perform a deep copy: create a "name" entry in self containing a copy of the object

Parameters

- **where** – path to the toplevel object (i.e. root)
- **toplevel** – firectly the top level Group
- **excluded** – list of keys to be excluded
- **overwrite** – replace content if already existing

find_detector(*all=False*)
Tries to find a detector within a NeXus file, takes the first compatible detector

Parameters **all** – return all detectors found as a list

classmethod **get_attr**(*dset*, *name*, *default=None*)
Return the attribute of the dataset

Handles the ascii -> unicode issue in python3 #275

Parameters

- **dset** – a HDF5 dataset (or a group)
- **name** – name of the attribute
- **default** – default value to be returned

Returns attribute value decoded in python3 or default

get_class(*grp*, *class_type='NXcollection'*)
return all sub-groups of the given type within a group

Parameters

- **grp** – HDF5 group
- **class_type** – name of the NeXus class

get_data(*grp*, *class_type='NXdata'*)
return all dataset of the the NeXus class NXdata

Parameters

- **grp** – HDF5 group
- **class_type** – name of the NeXus class

get_entries()
retrieves all entry sorted the latest first.

Returns list of HDF5 groups

get_entry(*name*)
Retrieves an entry from its name

Parameters **name** – name of the entry to retrieve

Returns HDF5 group of NXclass == NXentry

new_class(*grp*, *name*, *class_type='NXcollection'*)
create a new sub-group with type class_type :param grp: parent group :param name: name of the sub-group :param class_type: NeXus class name :return: subgroup created

new_detector(*name='detector'*, *entry='entry'*, *subentry='pyFAI'*)
Create a new entry/pyFAI/Detector

Parameters

- **detector** – name of the detector
- **entry** – name of the entry
- **subentry** – all pyFAI description of detectors should be in a pyFAI sub-entry

new_entry(*entry='entry'*, *program_name='pyFAI'*, *title='description of experiment'*, *force_time=None*)
Create a new entry

Parameters

- **entry** – name of the entry
- **program_name** – value of the field as string
- **title** – value of the field as string
- **force_time** – enforce the start_time (as string!)

Returns the corresponding HDF5 group

new_instrument(*entry='entry'*, *instrument_name='id00'*)
Create an instrument in an entry or create both the entry and the instrument if

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

Abstract class for writers.

CONFIG_ITEMS = ['filename', 'dirname', 'extension', 'subdir', 'hpath']

__init__(*filename=None*, *extension=None*)
Constructor of the class

flush(**arg*, ***kwarg*)
To be implemented

init(*fai_cfg=None*, *lima_cfg=None*)
Creates the directory that will host the output file(s) :param fai_cfg: configuration for worker :param lima_cfg: configuration for acquisition

setJsonConfig(*json_config=None*)
Sets the JSON configuration

write(*data*)
To be implemented

pyFAI.io.**from_isotime**(*text*, *use_tz=False*)

Parameters **text** – string representing the time is iso format

pyFAI.io.**get_isotime**(*forceTime=None*)

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

Returns the current time as an ISO8601 string

Return type string

pyFAI.io.**is_hdf5**(*filename*)
Check if a file is actually a HDF5 file

Parameters **filename** – this file has better to exist

6.16 calibration Module

pyFAI-calib

A tool for determining the geometry of a detector using a reference sample.

class pyFAI.calibration.**AbstractCalibration**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)
Bases: object

Everything that is common to Calibration and Recalibration

HELP = {'reset': 'Reset the geometry to the initial guess (rotation to zero, distance to 0.1m, poni at the center of the im

PARAMETERS = ['dist', 'poni1', 'poni2', 'rot1', 'rot2', 'rot3', 'wavelength']

PTS_PER_DEG = 0.3

UNITS = {'poni1': 'meter', 'poni2': 'meter', 'rot1': 'radian', 'rot3': 'radian', 'rot2': 'radian', 'wavelength': 'meter',

VALID_URL = ['', 'file', 'hdf5', 'nxs', 'h5']

__init__(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)
Constructor of AbstractCalibration

Parameters

- **dataFiles** – list of filenames containing data images
- **darkFiles** – list of filenames containing dark current images
- **flatFiles** – list of filenames containing flat images
- **pixelSize** – size of the pixel in meter as 2 tuple
- **splineFile** – file containing the distortion of the taper
- **detector** – Detector name or instance
- **wavelength** – radiation wavelength in meter
- **calibrant** – pyFAI.calibrant.Calibrant instance

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

Returns option,arguments

chiplot(*rings=None*)
plot delta_2theta/2theta = f(chi) and fit the curve.

Parameters **rings** – list of rings to consider

configure_parser(*version='calibration from pyFAI version 0.13.0: 02/12/2016'*, *usage='pyFAI-calib [options] input_image.edf'*, *description=None*, *epilog=None*)
Common configuration for parsers

extract_cpt(*method='massif'*, *pts_per_deg=1.0*)
Performs an automatic keypoint extraction: Can be used in recalib or in calib after a first calibration has been performed.

Parameters

- **method** – method for keypoint extraction
- **pts_per_deg** – number of control points per azimuthal degree (increase for better precision)

get_pixelSize(*ans*)
convert a comma separated sting into pixel size

postProcess()
Common part: shows the result of the azimuthal integration in 1D and 2D

preprocess()
Common part: do dark, flat correction thresholding, ... and read missing data from keyboard if needed

prompt()
prompt for commands to guide the calibration process

Returns True when the user is happy with what he has, False to request another refinement

read_dSpacingFile(*verbose=True*)
Read the name of the calibrant / file with d-spacing

read_pixelsSize()
Read the pixel size from prompt if not available

read_wavelength()
Read the wavelength

refine()
Contains the common geometry refinement part

reset_geometry(*how='center'*, *refine=False*)
Reset the geometry: no tilt in all cases

Parameters

- **how** – multiple options * center: set the PONI at the center of the detector * ring: center the poni at the middle of the inner-most ring * best: try both option and keeps the best (this option is not available)
- **refine** – launch the refinement (argument not used)

set_data(*data*)
call-back function for the peak-picker

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

validate_center(*slices=36*)
Validate the position of the center by cross-correlating two spectra 180 deg appart. Output values are in micron.

Designed for orthogonal setup with centered beam...

Parameters **slices** – number of slices on which perform

win_error = 'We are under windows with a 32 bit version of python, matplotlib is not able to display too many imag

class pyFAI.calibration.**Calibration**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *gaussianWidth=None*, *wavelength=None*, *calibrant=None*)
Bases: pyFAI.calibration.AbstractCalibration

class doing the calibration of frames

__init__(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *gaussianWidth=None*, *wavelength=None*, *calibrant=None*)
Constructor for calibration:

Parameters

- **dataFiles** – list of filenames containing data images
- **darkFiles** – list of filenames containing dark current images
- **flatFiles** – list of filenames containing flat images
- **pixelSize** – size of the pixel in meter as 2 tuple
- **splineFile** – file containing the distortion of the taper
- **detector** – Detector name or instance
- **wavelength** – radiation wavelength in meter
- **calibrant** – pyFAI.calibrant.Calibrant instance

gui_peakPicker()

initgeoRef()
Tries to initialise the GeometryRefinement (dist, poni, rot) Returns a dictionary of key value pairs

parse()
parse options from command line

preprocess()
do dark, flat correction thresholding, ...

refine()
Contains the geometry refinement part specific to Calibration Sets up the initial guess when starting pyFAI-calib

class pyFAI.calibration.**CheckCalib**(*poni=None*, *img=None*, *unit='2th_deg'*)
Bases: object

__init__(*poni=None*, *img=None*, *unit='2th_deg'*)

get_1dsize()

integrate()

parse()

rebuild()
Rebuild the diffraction image and measures the offset with the reference :return: offset

show()
Show the image with the the errors

size1d

smooth_mask(*hwhm=5*)
smooth out around the mask to avoid aligning on the mask

class pyFAI.calibration.**MultiCalib**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*)
Bases: object

__init__(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*)

get_pixelSize(*ans*)
convert a comma separated sting into pixel size

parse(*exe=None*, *description=None*, *epilog=None*)
parse options from command line :param exe: name of the program (MX-calibrate) :param description: Description of the program

process()

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

read_pixelsSize()
Read the pixel size from prompt if not available

read_wavelength()
Read the wavelength

regression()

class pyFAI.calibration.**Recalibration**(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)
Bases: pyFAI.calibration.AbstractCalibration

class doing the re-calibration of frames

__init__(*dataFiles=None*, *darkFiles=None*, *flatFiles=None*, *pixelSize=None*, *splineFile=None*, *detector=None*, *wavelength=None*, *calibrant=None*)
Constructor for Recalibration:

Parameters

- **dataFiles** – list of filenames containing data images
- **darkFiles** – list of filenames containing dark current images
- **flatFiles** – list of filenames containing flat images
- **pixelSize** – size of the pixel in meter as 2 tuple
- **splineFile** – file containing the distortion of the taper
- **detector** – Detector name or instance
- **wavelength** – radiation wavelength in meter
- **calibrant** – pyFAI.calibrant.Calibrant instance

parse()
parse options from command line

preprocess()
do dark, flat correction thresholding, ...

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

refine()
Contains the geometry refinement part specific to Recalibration

pyFAI.calibration.**calib**(*img*, *calibrant*, *detector*, *basename='from_ipython'*, *reconstruct=False*, *dist=0.1*, *gaussian=None*, *interactive=True*)
Procedural interfact for calibration

Parameters

- **img** – 2d array representing the calibration image
- **calibrant** – Instance of Calibrant, set-up with wavelength
- **detector** – Detector instance containing the mask
- **basename** – output file base
- **recontruct** – perform image reconstruction of masked pixel ?
- **dist** – initial distance

- **gaussian** – width of the gaussian used for difference of gaussian in the "massif" peak-picking algorithm
- **interactive** – set to False for testing

Returns AzimuthalIntegrator instance

pyFAI.calibration.**get_detector**(*detector*, *datafiles=None*)
Detector factory taking into account the binning knowing the datafiles

Parameters

- **detector** – string or detector or other junk
- **datafiles** – can be a list of images to be opened and their shape used.

:return pyFAI.detector.Detector instance :raise RuntimeError: If no detector found

6.17 peak_picker Module

class pyFAI.peak_picker.**ControlPoints**(*filename=None*, *calibrant=None*, *wavelength=None*)
Bases: object

This class contains a set of control points with (optionally) their ring number hence d-spacing and diffraction 2Theta angle ...

__init__(*filename=None*, *calibrant=None*, *wavelength=None*)

append(*points*, *ring=None*, *annotate=None*, *plot=None*)

Parameters

- **point** – list of points
- **ring** – ring number
- **annotate** – matplotlib.annotate reference
- **plot** – matplotlib.plot reference

Returns PointGroup instance

append_2theta_deg(*points*, *angle=None*, *ring=None*)

Parameters

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

check()
check internal consistency of the class

dSpacing

get(*ring=None*)
retireves the last set of points for a given ring (by default the last)

Parameters **ring** – index of ring to search for

getList()
Retrieve the list of control points suitable for geometry refinement with ring number

getList2theta()
Retrieve the list of control points suitable for geometry refinement

getListRing()
Retrieve the list of control points suitable for geometry refinement with ring number

getWeightedList(*image*)
Retrieve the list of control points suitable for geometry refinement with ring number and intensities :param image: :return: a (x,4) array with pos0, pos1, ring nr and intensity

#TODO: refine the value of the intensity using 2nd order polynomia

get_dSpacing()

get_wavelength()

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

pop(*ring=None*, *lbl=None*)
Remove the set of points, either from its code or from a given ring (by default the last)

Parameters

- **ring** – index of ring of which remove the last group
- **lbl** – code of the ring to remove

readRingNrFromKeyboard()
Ask the ring number values for the given points

reset()
remove all stored values and resets them to default

save(*filename*)
Save a set of control points to a file :param filename: name of the file :return: None

setWavelength_change2th(*value=None*)

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

set_dSpacing(*lst*)

set_wavelength(*value=None*)

wavelength

class pyFAI.peak_picker.**PeakPicker**(*data*, *reconst=False*, *mask=None*, *pointfile=None*, *calibrant=None*, *wavelength=None*, *detector=None*, *method='massif'*)
Bases: object

This class is in charge of peak picking, i.e. find bragg spots in the image Two methods can be used : massif or blob

VALID_METHODS = ['massif', 'blob', 'watershed']

__init__(*data*, *reconst=False*, *mask=None*, *pointfile=None*, *calibrant=None*, *wavelength=None*, *detector=None*, *method='massif'*)

Parameters

- **data** – input image as numpy array
- **reconst** – shall masked part or negative values be reconstructed (wipe out problems with pilatus gaps)
- **mask** – area in which keypoints will not be considered as valid
- **pointfile** –

closeGUI()

contour(*data*, *cmap='autumn'*, *linewidths=2*, *linestyles='dashed'*)
Overlay a contour-plot

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

display_points(*minIndex=0*, *reset=False*)
display all points and their ring annotations :param minIndex: ring index to start with :param reset: remove all point before re-displaying them

finish(*filename=None*, *callback=None*)
Ask the ring number for the given points

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

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

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

help = ['Please select rings on the diffraction image. In parenthesis, some modified shortcuts for single button mouse

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

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

massif_contour(*data*)
Overlays a mask over a diffraction image

Parameters **data** – mask to be overlaid

on_minus_pts_clicked(**args*)
callback function

on_option_clicked(**args*)
callback function

on_plus_pts_clicked(**args*)
callback function

on_refine_clicked(**args*)
callback function

onclick(*event*)
Called when a mouse is clicked

peaks_from_area(***kwargs*)
Return the list of peaks within an area

Parameters

- **mask** – 2d array with mask.
- **Imin** – minimum of intensity above the background to keep the point
- **keep** – maximum number of points to keep
- **method** – enforce the use of detection using "massif" or "blob" or "watershed"
- **ring** – ring number to which assign the points
- **dmin** – minimum distance between two peaks (in pixels)
- **seed** – good starting points.

Returns list of peaks [y,x], [y,x], ...]

remove_grp(*lbl*)
remove a group of points

Parameters **lbl** – label of the group of points

reset()
Reset control point and graph (if needed)

sync_init()

class pyFAI.peak_picker.**PointGroup**(*points=None*, *ring=None*, *annotate=None*, *plot=None*, *force_label=None*)

Bases: object

Class contains a group of points ... They all belong to the same Debye-Scherrer ring

__init__(*points=None*, *ring=None*, *annotate=None*, *plot=None*, *force_label=None*)

Constructor

Parameters

- **points** – list of points
- **ring** – ring number
- **annotate** – reference to the matplotlib annotate output
- **plot** – reference to the matplotlib plot
- **force_label** – allows to enforce the label

code

Numerical value for the label: mainly for sorting

classmethod get_label()

return the next label

get_ring()

label

last_label = 0

classmethod reset_label()

reset intenal counter

ring

classmethod set_label(*label*)

update the internal counter if needed

set_ring(*value*)

6.18 massif Module

class pyFAI.massif.**Massif**(*data=None*)

Bases: object

A massif is defined as an area around a peak, it is used to find neighboring peaks

TARGET_SIZE = 1024

__init__(*data=None*)

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 0x7f23416fe150>*)

All in one function that finds a maximum from the given seed (x) then calculates the region extension and extract position of the neighboring peaks. :param x: seed for the calculation, input coordinates :param nmax: maximum number of peak per region :param annotate: call back method taking number of points + coordinate as input. :param massif_contour: callback to show the contour of a massif with the given index. :param stdout: this is the file where output is written by default. :return: list of peaks

getBinnedData()

:return binned data

getBluredData()

Returns a blurred image

getLabeledMassif(*pattern=None*)

Returns an image composed of int with a different value for each massif

getMedianData()

Returns a spacial median filtered image

getValleySize()

initValleySize()

nearest_peak(*x*)

Parameters **x** – coordinates of the peak

Returns the coordinates of the nearest peak

peaks_from_area(*mask*, *Imin=None*, *keep=1000*, *dmin=0.0*, *seed=None*, ***kwarg*)
Return the list of peaks within an area

Parameters

- **mask** – 2d array with mask.
- **Imin** – minimum of intensity above the background to keep the point
- **keep** – maximum number of points to keep
- **kwarg** – ignored parameters
- **dmin** – minimum distance to another peak
- **seed** – list of good guesses to start with

Returns list of peaks [y,x], [y,x], ...]

setValleySize(*size*)

valley_size
Defines the minimum distance between two massifs

6.19 blob_detection Module

class pyFAI.blob_detection.**BlobDetection**(*img*, *cur_sigma=0.25*, *init_sigma=0.5*, *dest_sigma=1*, *scale_per_octave=2*, *mask=None*)

Bases: object

Performs a blob detection: http://en.wikipedia.org/wiki/Blob_detection using a Difference of Gaussian + Pyramid of Gaussians

__init__(*img*, *cur_sigma=0.25*, *init_sigma=0.5*, *dest_sigma=1*, *scale_per_octave=2*, *mask=None*)
Performs a blob detection: http://en.wikipedia.org/wiki/Blob_detection using a Difference of Gaussian + Pyramid of Gaussians

Parameters

- **img** – input image
- **cur_sigma** – estimated smoothing of the input image. 0.25 correspond to no interaction between pixels.
- **init_sigma** – start searching at this scale (sigma=0.5: 10% interaction with first neighbor)

- **dest_sigma** – sigma at which the resolution is lowered (change of octave)
- **scale_per_octave** – Number of scale to be performed per octave
- **mask** – mask where pixel are not valid

direction()
Perform and plot the two main directions of the peaks, considering their previously calculated scale ,by calculating the Hessian at different sizes as the combination of gaussians and their first and second derivatives

nearest_peak(*p*, *refine=True*, *Imin=None*)
Return the nearest peak from a position

Parameters

- **p** – input position (y,x) 2-tuple of float
- **refine** – shall the position be refined on the raw data
- **Imin** – minimum of intensity above the background

peaks_from_area(*mask*, *keep=None*, *refine=True*, *Imin=None*, *dmin=0.0*, ***kwargs*)
Return the list of peaks within an area

Parameters

- **mask** – 2d array with mask.
- **refine** – shall the position be refined on the raw data
- **Imin** – minimum of intensity above the background
- **kwarg** – ignored parameters

Returns list of peaks [y,x], [y,x], ...]

process(*max_octave=None*)
Perform the keypoint extraction for max_octave cycles or until all octaves have been processed. :param max_octave: number of octave to process

refine_Hessian(*kpx*, *kpy*, *kps*)
Refine the keypoint location based on a 3 point derivative, and delete non-coherent keypoints.

Parameters

- **kpx** – x_pos of keypoint
- **kpy** – y_pos of keypoint
- **kps** – s_pos of keypoint

Returns arrays of corrected coordinates of keypoints, values and locations of keypoints

refine_Hessian_SG(*kpx*, *kpy*, *kps*)
Savitzky Golay algorithm to check if a point is really the maximum :param kpx: x_pos of keypoint :param kpy: y_pos of keypoint :param kps: s_pos of keypoint :return array of corrected keypoints

refinement()

show_neighboor()

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

tresh = 0.6

pyFAI.blob_detection.**image_test**()

pyFAI.blob_detection.**local_max**(*dogs*, *mask=None*, *n_5=True*)

Parameters

- **dogs** – 3d array with (sigma,y,x) containing difference of gaussians

- **mask** – mask out keypoint next to the mask (or inside the mask)
- **n_5** – look for a larger neighborhood

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

6.20 calibrant Module

Calibrant

A module containing classical calibrant and also tools to generate d-spacing.

Interesting formula: http://geoweb3.princeton.edu/research/MineralPhy/xtalgeometry.pdf

class pyFAI.calibrant.**Calibrant**(*filename=None*, *dSpacing=None*, *wavelength=None*)

Bases: object

A calibrant is a reference compound where the d-spacing (interplanar distances) are known. They are expressed in Angstrom (in the file)

__init__(*filename=None*, *dSpacing=None*, *wavelength=None*)

append_2th(*value*)

append_dSpacing(*value*)

dSpacing

fake_calibration_image(*ai*, *shape=None*, *Imax=1.0*, *U=0*, *V=0*, *W=0.0001*)

Generates a fake calibration image from an azimuthal integrator

Parameters

- **ai** – azimuthal integrator
- **Imax** – maximum intensity of rings
- **V, W** (*U,*) – width of the peak from Caglioti's law (FWHM^2 = Utan(th)^2 + Vtan(th) + W)

get_2th()

get_2th_index(*angle*)

return the index in the 2theta angle index

get_dSpacing()

get_wavelength()

load_file(*filename=None*)

save_dSpacing(*filename=None*)

save the d-spacing to a file

setWavelength_change2th(*value=None*)

setWavelength_changeDs(*value=None*)

This is probably not a good idea, but who knows !

set_dSpacing(*lst*)

set_wavelength(*value=None*)

wavelength

class pyFAI.calibrant.**Cell**(*a=1*, *b=1*, *c=1*, *alpha=90*, *beta=90*, *gamma=90*, *lattice='triclinic'*, *lattice_type='P'*)

Bases: object

This is a cell object, able to calculate the volume and d-spacing according to formula from:

http://geoweb3.princeton.edu/research/MineralPhy/xtalgeometry.pdf

__init__(*a=1*, *b=1*, *c=1*, *alpha=90*, *beta=90*, *gamma=90*, *lattice='triclinic'*, *lattice_type='P'*)
Constructor of the Cell class:

Crystalographic units are Angstrom for distances and degrees for angles !

Parameters

- **a,b,c** – unit cell length in Angstrom
- **beta, gamma** (*alpha,*) – unit cell angle in degrees
- **lattice** – "cubic", "tetragonal", "hexagonal", "rhombohedral", "orthorhombic", "monoclinic", "triclinic"
- **lattice_type** – P, I, F, C or R

classmethod cubic(*a*, *lattice_type='P'*)
Factory for cubic lattices

Parameters **a** – unit cell length

d(*hkl*)
Calculate the actual d-spacing for a 3-tuple of integer representing a family of Miller plans

Parameters **hkl** – 3-tuple of integers

Returns the inter-planar distance

d_spacing(*dmin=1.0*)
Calculate all d-spacing down to dmin

applies selection rules

Parameters **dmin** – minimum value of spacing requested

Returns dict d-spacing as string, list of tuple with Miller indices preceded with the numerical value

classmethod diamond(*a*)
Factory for Diamond type FCC like Si and Ge

Parameters **a** – unit cell length

get_type()

classmethod hexagonal(*a*, *c*, *lattice_type='P'*)
Factory for hexagonal lattices

Parameters

- **a** – unit cell length
- **c** – unit cell length

lattices = ['cubic', 'tetragonal', 'hexagonal', 'rhombohedral', 'orthorhombic', 'monoclinic', 'triclinic']

classmethod monoclinic(*a*, *b*, *c*, *beta*, *lattice_type='P'*)
Factory for hexagonal lattices

Parameters

- **a** – unit cell length
- **b** – unit cell length
- **c** – unit cell length
- **beta** – unit cell angle

classmethod orthorhombic(*a*, *b*, *c*, *lattice_type='P'*)
Factory for orthorhombic lattices

Parameters

- **a** – unit cell length
- **b** – unit cell length
- **c** – unit cell length

classmethod rhombohedral(*a*, *alpha*, *lattice_type='P'*)
Factory for hexagonal lattices

Parameters

- **a** – unit cell length
- **alpha** – unit cell angle

save(*name*, *long_name=None*, *doi=None*, *dmin=1.0*, *dest_dir=None*)
Save informations about the cell in a d-spacing file, usable as Calibrant

Parameters

- **name** – name of the calibrant
- **doi** – reference of the publication used to parametrize the cell
- **dmin** – minimal d-spacing
- **dest_dir** – name of the directory where to save the result

selection_rules = None
contains a list of functions returning True(allowed)/False(forbiden)/None(unknown)

set_type(*lattice_type*)

classmethod tetragonal(*a*, *c*, *lattice_type='P'*)
Factory for tetragonal lattices

Parameters

- **a** – unit cell length
- **c** – unit cell length

type

types = {'I': 'Body centered', 'P': 'Primitive', 'C': 'Side centered', 'R': 'Rhombohedral', 'F': 'Face centered'}

volume

class pyFAI.calibrant.**calibrant_factory**(*basedir=None*)
Bases: object

Behaves like a dict but is actually a factory:

Each time one retrieves an object it is a new geniune new calibrant (unmodified)

__init__(*basedir=None*)
Constructor

Parameters **basedir** – directory name where to search for the calibrants

get(*what*, *notfound=None*)

has_key(*k*)

items()

keys()

values()

6.21 distortion Module

class pyFAI.distortion.**Distortion**(*detector='detector'*, *shape=None*, *resize=False*, *empty=0*, *mask=None*, *method='CSR'*, *device=None*, *workgroup=8*)

Bases: object

This class applies a distortion correction on an image.

New version compatible both with CSR and LUT...

__init__(*detector='detector'*, *shape=None*, *resize=False*, *empty=0*, *mask=None*, *method='CSR'*, *device=None*, *workgroup=8*)

Parameters

- **detector** – detector instance or detector name
- **shape** – shape of the output image
- **resize** – allow the output shape to be different from the input shape
- **empty** – value to be given for empty bins
- **method** – "lut" or "csr", the former is faster
- **device** – Name of the device: None for OpenMP, "cpu" or "gpu" or the id of the OpenCL device a 2-tuple of integer
- **workgroup** – workgroup size for CSR on OpenCL

calc_LUT(*use_common=True*)
Calculate the Look-up table

Returns look up table either in CSR or LUT format depending on serl.method

calc_init()
Initialize all arrays

calc_pos(*use_cython=True*)
Calculate the pixel boundary position on the regular grid

Returns pixel corner positions (in pixel units) on the regular grid

Return type ndarray of shape (nrow, ncol, 4, 2)

calc_size(*use_cython=True*)
Calculate the number of pixels falling into every single bin and

Returns max of pixel falling into a single bin

Considering the "half-CCD" spline from ID11 which describes a (1025,2048) detector, the physical location of pixels should go from: [-17.48634 : 1027.0543, -22.768829 : 2028.3689] We chose to discard pixels falling outside the [0:1025,0:2048] range with a lose of intensity

correct(*image*, *dummy=None*, *delta_dummy=None*)
Correct an image based on the look-up table calculated ...

Parameters

- **image** – 2D-array with the image
- **dummy** – value suggested for bad pixels
- **delta_dummy** – precision of the dummy value

Returns corrected 2D image

reset(*method=None*, *device=None*, *workgroup=None*, *prepare=True*)
reset the distortion correction and re-calculate the look-up table

Parameters

- **method** – can be “lut” or “csr”, “lut” looks faster
- **device** – can be None, “cpu” or “gpu” or the id as a 2-tuple of integer
- **worgroup** – enforce the workgroup size for CSR.
- **prepare** – set to false to only reset and not re-initialize

shape_out
Calculate/cache the output shape

:return output shape

uncorrect(*image*, *use_cython=False*)
Take an image which has been corrected and transform it into it’s raw (with loss of information)

Parameters **image** – 2D-array with the image

Returns uncorrected 2D image

Nota: to retrieve the input mask on can do:

```
>>> msk =  dis.uncorrect(numpy.ones(dis._shape_out)) <= 0
```

class pyFAI.distortion.**Quad**(*buffer*)
Bases: object

Quad modelisation.

__init__(*buffer*)

calc_area()

calc_area_AB(*I1*, *I2*)

calc_area_BC(*J1*, *J2*)

calc_area_CD(*K1*, *K2*)

calc_area_DA(*L1*, *L2*)

calc_area_old()

calc_area_vectorial()

get_box(*i*, *j*)

get_box_size0()

get_box_size1()

get_idx(*i*, *j*)

get_offset0()

get_offset1()

init_slope()

integrateAB(*start*, *stop*, *calc_area*)

populate_box()

reinit(*A0*, *A1*, *B0*, *B1*, *C0*, *C1*, *D0*, *D1*)

pyFAI.distortion.**test**()

6.22 worker Module

This module contains the Worker class:

A tool able to perform azimuthal integration with: additional saving capabilities like

- save as 2/3D structure in a HDF5 File
- read from HDF5 files

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

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

- “dist”
- “poni1”
- “poni2”
- “rot1”
- “rot3”
- “rot2”
- “pixel1”
- “pixel2”
- “splineFile”
- “wavelength”
- “poni” #path of the file
- “chi_discontinuity_at_0”
- “do_mask”
- “do_dark”
- “do_azimuthal_range”
- “do_flat”
- “do_2D”
- “azimuth_range_min”
- “azimuth_range_max”
- “polarization_factor”
- “nbpt_rad”
- “do_solid_angle”
- “do_radial_range”
- “do_poisson”
- “delta_dummy”
- “nbpt_azim”
- “flat_field”
- “radial_range_min”
- “dark_current”
- “do_polarization”
- “mask_file”
- “detector”
- “unit”
- “radial_range_max”

- "val_dummy"
- "do_dummy"
- "method"

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

Bases: object

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

__init__(*detector=None*, *dark=None*, *flat=None*, *solidangle=None*, *polarization=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *device=None*)

Constructor of the worker :param dark: array :param flat: array :param solidangle: solid-angle array :param polarization: numpy array with 2D polarization corrections :param device: Used to influance OpenCL behavour: can be "cpu", "GPU", "Acc" or even an OpenCL context

process(*data*, *normalization_factor=1.0*)

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

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

Bases: object

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

__init__(*dark=None*, *flat=None*, *solidangle=None*, *polarization=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *device=None*)

Constructor of the worker

Parameters

- **dark** – array
- **flat** – array
- **solidangle** – solid-angle array
- **polarization** – numpy array with 2D polarization corrections
- **device** – Used to influance OpenCL behavour: can be "cpu", "GPU", "Acc" or even an OpenCL context

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

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

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

Bases: object

__init__(*azimuthalIntegrator=None*, *shapeIn=(2048, 2048)*, *shapeOut=(360, 500)*, *unit='r_mm'*, *dummy=None*, *delta_dummy=None*, *azimuthalIntgrator=None*)

Parameters

- **AzimuthalIntegrator** (*azimuthalIntgrator*) – pyFAI.AzimuthalIntegrator instance
- **AzimuthalIntegrator** – pyFAI.AzimuthalIntegrator instance (deprecated)
- **shapeIn** – image size in input
- **shapeOut** – Integrated size: can be (1,2000) for 1D integration
- **unit** – can be "2th_deg, r_mm or q_nm^-1 ...

do_2D()

error_model

get_config()
return configuration as a dictionary

get_error_model()

get_json_config()
return configuration as a JSON string

get_normalization_factor()

get_unit()

normalization_factor

process(*data*, *normalization_factor=1.0*, *writer=None*)
Process a frame #TODO: dark, flat, sa are missing

Parameters

- **data** – numpy array containing the input image
- **writer** – An open writer in which 'write' will be called with the result of the integration

reconfig(*shape=(2048, 2048)*, *sync=False*)
This is just to force the integrator to initialize with a given input image shape

Parameters

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

reset()
this is just to force the integrator to initialize

save_config(*filename=None*)

setDarkcurrentFile(*imagefile*)

setExtension(*ext*)
enforce the extension of the processed data file written

setFlatfieldFile(*imagefile*)

setJsonConfig(*jsonconfig*)

setSubdir(*path*)
Set the relative or absolute path for processed data

set_error_model(*value*)

set_normalization_factor(*value*)

set_unit(*value*)

unit

warmup(*sync=False*)
Process a dummy image to ensure everything is initialized

Parameters **sync** – wait for processing to be finished

pyFAI.worker.**make_ai**(*config*)
Create an Azimuthal integrator from the configuration stand alone function !

Parameters **config** – dict with all parameters

Returns configured (but uninitialized) AzimuthalIntgrator

6.23 `units` Module

Manages the different units

Nota for developers: this module is used a singleton to store all units in a unique manner. This explains the number of top-level variables on the one hand and their CAPITALIZATION on the other.

class `pyFAI.units.`**`Unit`**(*name*, *scale=1*, *label=None*, *equation=None*, *center=None*, *corner=None*, *delta=None*)

Bases: `object`

Represents a unit.

It has at least a name and a scale (in SI-unit)

`__init__`(*name*, *scale=1*, *label=None*, *equation=None*, *center=None*, *corner=None*, *delta=None*)

Constructor of a unit.

Parameters

- **name** (*(str)*) – name of the unit
- **scale** (*(float)*) – scale of th unit to go to SI
- **label** (*(string)*) – label for nice representation in matplotlib, can use latex representation
- **equation** (*(funct)*) – equation to calculate the value from coordinates (x,y,z) in detector space. Parameters of the function are x, y, z, lambda
- **center** (*(str)*) – name of the fast-path function

`get`(*key*)

Mimic the dictionary interface

Parameters **key** (*(str)*) – key wanted

Returns self.key

`pyFAI.units.`**`eq_2th`**(*x*, *y*, *z*, *wavelength=None*)

Calculates the 2theta aperture of the cone

Parameters

- **x** – horizontal position, towards the center of the ring, from sample position
- **y** – Vertical position, to the roof, from sample position
- **z** – distance from sample along the beam
- **wavelength** – in meter

`pyFAI.units.`**`eq_q`**(*x*, *y*, *z*, *wavelength*)

Calculates the modulus of the scattering vector

Parameters

- **x** – horizontal position, towards the center of the ring, from sample position
- **y** – Vertical position, to the roof, from sample position
- **z** – distance from sample along the beam
- **wavelength** – in meter

`pyFAI.units.`**`eq_r`**(*x*, *y*, *z=None*, *wavelength=None*)

Calculates the radius

Parameters

- **x** – horizontal position, towards the center of the ring, from sample position

- **y** – Vertical position, to the roof, from sample position
- **z** – distance from sample along the beam
- **wavelength** – in meter

pyFAI.units.**register_radial_unit**(*name*, *scale=1*, *label=None*, *equation=None*, *center=None*, *corner=None*, *delta=None*)

pyFAI.units.**to_unit**(*obj*, *type_=None*)

6.24 utils Module

Utilities, mainly for image treatment

class pyFAI.utils.**FixedParameters**

Bases: set

Like a set, made for FixedParameters in geometry refinement

add_or_discard(*key*, *value=True*)

Add a value to a set if value, else discard it :param key: element to added or discared from set :type value: boolean. If None do nothing ! :return: None

pyFAI.utils.**binning**(*input_img*, *binsize*, *norm=True*)

Parameters

- **input_img** – input ndarray
- **binsize** – int or 2-tuple representing the size of the binning
- **norm** – if False, do average instead of sum

Returns binned input ndarray

pyFAI.utils.**calc_checksum**(*ary*, *safe=True*)

Calculate the checksum by default (or returns its buffer location if unsafe)

pyFAI.utils.**center_of_mass**(*img*)

Calculate the center of mass of of the array. Like scipy.ndimage.measurements.center_of_mass :param img: 2-D array :return: 2-tuple of float with the center of mass

pyFAI.utils.**concatenate_cl_kernel**(*filenames*)

Parameters **filenames** (*list of str which can be filename of kernel as a string.*) – filenames containing the kernels

this method concatenates all the kernel from the list

pyFAI.utils.**convert_CamelCase**(*name*)

convert a function name in CamelCase into camel_case

pyFAI.utils.**deg2rad**(*dd*)

Convert degrees to radian in the range -pi->pi

Parameters **dd** – angle in degrees

Nota: depending on the platform it could be 0<2pi A branch is cheaper than a trigo operation

pyFAI.utils.**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 FFT

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.
- **mode** – "constant", "nearest", "reflect" or "mirror"
- **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.**expand2d**(*vect*, *size2*, *vertical=True*)
This expands a vector to a 2d-array.

The result is the same as:

```
if vertical:
    numpy.outer(numpy.ones(size2), vect)
else:
    numpy.outer(vect, numpy.ones(size2))
```

This is a ninja optimization: replace *1 with a memcopy, saves 50% of time at the ms level.

Parameters

- **vect** – 1d vector
- **size2** – size of the expanded dimension
- **vertical** – if False the vector is expanded to the first dimension. If True, it is expanded to the second dimension.

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*, *use_scipy=True*)
2-dimensional Gaussian filter implemented with FFT

Parameters

- **input_img** (*array-like*) – input array to filter
- **sigma** (*scalar or sequence of scalars*) – standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **mode** – {'reflect','constant','nearest','mirror', 'wrap'}, optional The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'
- **cval** – scalar, optional Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

pyFAI.utils.**get_calibration_dir**()
get the full path of a calibration directory

Returns the full path of the calibrant file

pyFAI.utils.**get_cl_file**(*filename*)
get the full path of a openCL file

Returns the full path of the openCL source file

pyFAI.utils.**get_ui_file**(*filename*)
get the full path of a user-interface file

Returns the full path of the ui

pyFAI.utils.**int_**(*val*)
Convert anything to an int ... or None if not applicable

pyFAI.utils.**is_far_from_group**(*pt*, *lst_pts*, *d2*)
Tells if a point is far from a group of points, distance greater than d2 (distance squared)

Parameters

- **pt** – point of interest
- **lst_pts** – list of points
- **d2** – minimum distance squarred

Returns True If the point is far from all others.

class pyFAI.utils.**lazy_property**(*fget*)
Bases: `object`

meant to be used for lazy evaluation of an object attribute. property should represent non-mutable data, as it replaces itself.

__init__(*fget*)

pyFAI.utils.**maximum_position**(*img*)
Same as scipy.ndimage.measurements.maximum_position: Find the position of the maximum of the values of the array.

Parameters **img** – 2-D image

Returns 2-tuple of int with the position of the maximum

pyFAI.utils.**measure_offset**(*img1*, *img2*, *method='numpy'*, *withLog=False*, *withCorr=False*)
Measure the actual offset between 2 images :param img1: ndarray, first image :param img2: ndarray, second image, same shape as img1 :param withLog: shall we return logs as well ? boolean :return: tuple of floats with the offsets

pyFAI.utils.**readFloatFromKeyboard**(*text*, *dictVar*)
Read float from the keyboard ....

Parameters

- **text** – string to be displayed
- **dictVar** – dict of this type: {1: [set_dist_min],3: [set_dist_min, set_dist_guess, set_dist_max]}

pyFAI.utils.**read_cl_file**(*filename*)

Parameters **filename** – read an OpenCL file and apply a preprocessor

Returns preprocessed source code

pyFAI.utils.**relabel**(*label*, *data*, *blured*, *max_size=None*)
Relabel limits the number of region in the label array. They are ranked relatively to their max(I0)-max(blur(I0)

Parameters

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

Returns array like label

pyFAI.utils.**roundfft**(*N*)
This function returns the integer >=N for which size the Fourier analysis is faster (fron the FFT point of view) Credit: Alessandro Mirone, ESRF, 2012

Parameters **N** – interger on which one would like to do a Fourier transform

Returns integer with a better choice

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='fft'*)
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.**unBinning**(*binnedArray*, *binsize*, *norm=True*)

Parameters

- **binnedArray** – input ndarray
- **binsize** – 2-tuple representing the size of the binning
- **norm** – if True (default) decrease the intensity by binning factor. If False, it is non-conservative

Returns unBinned input ndarray

6.25 `gui.gui_utils` Module

Module providing gui util tools

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

Dummy class for dummy things

__init__(*width*, *height*)

pyFAI.gui.utils.**maximize_fig**(*fig=None*)
Try to set the figure fullscreen

pyFAI.gui.utils.**update_fig**(*fig=None*)
Update a matplotlib figure with a Qt4 backend

Parameters **fig** – pylab figure

6.26 ext.bilinear Module

This extension makes a discrete 2D-array appear like a continuous function thanks to bilinear interpolations.

class pyFAI.ext.bilinear.**Bilinear**
Bases: object

Bilinear interpolator for finding max.

Instance attribute defined in pxd file

cp_local_maxi(*self*, *size_t x*) → size_t

data

f_cy(*self*, *x*)
Function f((y,x)) where f is a continuous function (y,x) are pixel coordinates @param x: 2-tuple of float @return: Interpolated signal from the image (negative for minimizer)

height

local_maxi(*self*, *x*)
Return the local maximum ... with sub-pixel refinement

@param x: 2-tuple of integers @param w: half with of the window: 1 or 2 are advised @return: 2-tuple of float with the nearest local maximum

Sub-pixel refinement: Second order Taylor expansion of the function; first derivative is null delta = x-i = -Inverse[Hessian].gradient

if Hessian is singular or |delta|>1: use a center of mass.

maxi

mini

width

pyFAI.ext.bilinear.**calc_cartesian_positions**
calc_cartesian_positions(signatures, args, kwargs, defaults)

Calculate the Cartesian position for array of position (d1, d2) with pixel coordinated stored in array pos This is bilinear interpolation

Parameters

- **d1** – position in dim1
- **d2** – position in dim2
- **pos** – array with position of pixels corners

:return 3-tuple of position.

pyFAI.ext.bilinear.**convert_corner_2D_to_4D**
convert_corner_2D_to_4D(signatures, args, kwargs, defaults)

Convert 2 (or 3) arrays of corner position into a 4D array of pixel corner coordinates

Parameters

- **ndim** – 2d or 3D output
- **d1** – 2D position in dim1 (shape +1)
- **d2** – 2D position in dim2 (shape +1)
- **d3** – 2D position in dim3 (z) (shape +1)

Returns pos 4D array with position of pixels corners

6.27 ext._bispev Module

This extension is a re-implementation of bi-cubic spline evaluation from scipy Spline evaluation function

Created on Nov 4, 2013

@author: zubair, Jerome Kieffer

pyFAI.ext._bispev.**bisplev**(*x*, *y*, *tck*, *dx=0*, *dy=0*)
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. Based on BISPEV from FITPACK.

x, y [ndarray] Rank-1 arrays specifying the domain over which to evaluate the spline or its derivative.

tck [tuple] A sequence of length 5 returned by *bisplrep* containing the knot locations, the coefficients, and the degree of the spline: [tx, ty, c, kx, ky].

dx, dy [int, optional] The orders of the partial derivatives in *x* and *y* respectively. This version does bot implement derivatives.

vals [ndarray] The B-spline or its derivative evaluated over the set formed by the cross-product of *x* and *y*.

splprep, splrep, splint, sproot, splev UnivariateSpline, BivariateSpline

See *bisplrep* to generate the *tck* representation.

6.28 ext._blob Module

Blob detection is used to find peaks in images by performing subsequent blurs Some Cythonized function for blob detection function

pyFAI.ext._blob.**local_max**(*__Pyx_memviewslice dogs*, *mask=None*, *bool n_5=False*)
Calculate if a point is a maximum in a 3D space: (scale, y, x)

Parameters

- **dogs** – 3D array of difference of gaussian
- **mask** – mask with invalid pixels
- **N_5** – take a neighborhood of 5x5 pixel in plane

Returns 3d_array with 1 where is_max

6.29 ext.container Module

Container are a new uniform storage, optimized for the creation of both LUT and CSR. It has nothing to do with Docker.

6.30 ext._convolution Module

Convolutions in real space are used to blurs images, used in blob-detection algorithm Implementation of a separable 2D convolution

pyFAI.ext._convolution.**gaussian**(*sigma*, *width=None*)

Return a Gaussian window of length "width" with standard-deviation "sigma".

Parameters

- **sigma** – standard deviation sigma
- **width** – length of the windows (int) By default 8*sigma+1,

Width should be odd.

The FWHM is 2*sqrt(2 * pi)*sigma

pyFAI.ext._convolution.**gaussian_filter**(*img*, *sigma*)

Performs a gaussian bluring using a gaussian kernel.

Parameters

- **img** – input image
- **sigma** – width parameter of the gaussian

pyFAI.ext._convolution.**horizontal_convolution**(*__Pyx_memviewslice img*, *__Pyx_memviewslice filter*)

Implements a 1D horizontal convolution with a filter. The only implemented mode is "reflect" (default in scipy.ndimage.filter)

Parameters

- **img** – input image
- **filter** – 1D array with the coefficients of the array

Returns array of the same shape as image with

pyFAI.ext._convolution.**vertical_convolution**(*__Pyx_memviewslice img*, *__Pyx_memviewslice filter*)

Implements a 1D vertical convolution with a filter. The only implemented mode is "reflect" (default in scipy.ndimage.filter)

Parameters

- **img** – input image
- **filter** – 1D array with the coefficients of the array

Returns array of the same shape as image with

6.31 ext._distortion Module

Distortion correction are correction are applied by Look-up table (or CSR) Common Look-Up table datatypes

class pyFAI.ext._distortion.**Distortion**(*self*, *detector='detector'*, *shape=None*)
Bases: object

This class applies a distortion correction on an image.

It is also able to apply an inversion of the correction.

__init__(*self*, *detector='detector'*, *shape=None*)

Parameters **detector** – detector instance or detector name

calc_LUT(*self*)

calc_LUT_size(*self*)
Considering the "half-CCD" spline from ID11 which describes a (1025,2048) detector, the physical location of pixels should go from: [-17.48634 : 1027.0543, -22.768829 : 2028.3689] We chose to discard pixels falling outside the [0:1025,0:2048] range with a lose of intensity

We keep self.pos: pos_corners will not be compatible with systems showing non adjacent pixels (like some xpads)

calc_pos(*self*)

correct(*self*, *image*)
Correct an image based on the look-up table calculated ...

Parameters **image** – 2D-array with the image

Returns corrected 2D image

uncorrect(*self*, *image*)
Take an image which has been corrected and transform it into it's raw (with loss of information)

Parameters **image** – 2D-array with the image

Returns uncorrected 2D image and a mask (pixels in raw image

pyFAI.ext._distortion.**calc_CSR**(*__Pyx_memviewslice pos*, *shape*, *bin_size*, *max_pixel_size*, *__Pyx_memviewslice mask=None*)
Calculate the Look-up table as CSR format

Parameters

- **pos** – 4D position array
- **shape** – output shape
- **bin_size** – number of input element per output element (as numpy array)
- **max_pixel_size** – (2-tuple of int) size of a buffer covering the largest pixel

Returns look-up table in CSR format: 3-tuple of array

pyFAI.ext._distortion.**calc_LUT**(*__Pyx_memviewslice pos*, *shape*, *bin_size*, *max_pixel_size*, *__Pyx_memviewslice mask=None*)

Parameters

- **pos** – 4D position array
- **shape** – output shape
- **bin_size** – number of input element per output element (numpy array)
- **max_pixel_size** – (2-tuple of int) size of a buffer covering the largest pixel
- **mask** – arry with bad pixels marked as True

Returns look-up table

pyFAI.ext._distortion.**calc_area**(*float I1*, *float I2*, *float slope*, *float intercept*) → float
Calculate the area between I1 and I2 of a line with a given slope & intercept

pyFAI.ext._distortion.**calc_openmp**(*__Pyx_memviewslice pos*, *shape*, *max_pixel_size=(8, 8)*, *__Pyx_memviewslice mask=None*, *format='csr'*, *int bins_per_pixel=8*)

Calculate the look-up table (or CSR) using OpenMP

Parameters

- **pos** – 4D position array
- **shape** – output shape
- **max_pixel_size** – (2-tuple of int) size of a buffer covering the largest pixel
- **format** – can be "CSR" or "LUT"
- **bins_per_pixel** – average splitting factor (number of pixels per bin)

Returns look-up table in CSR/LUT format

pyFAI.ext._distortion.**calc_pos**

calc_pos(signatures, args, kwargs, defaults) Calculate the pixel boundary position on the regular grid

param pixel_corners pixel corner coordinate as detector.get_pixel_corner()

param shape requested output shape. If None, it is calculated

param pixel1, pixel2 pixel size along row and column coordinates

return pos, delta1, delta2, shape_out, offset

pyFAI.ext._distortion.**calc_size**

calc_size(signatures, args, kwargs, defaults) Calculate the number of items per output pixel

param pos 4D array with position in space

param shape shape of the output array

param mask input data mask

param offset 2-tuple of float with the minimal index of

return number of input element per output elements

pyFAI.ext._distortion.**clip**(*int value*, *int min_val*, *int max_val*) → int

Limits the value to bounds

pyFAI.ext._distortion.**correct_CSR**(*image*, *shape_in*, *shape_out*, *LUT*, *dummy=None*, *delta_dummy=None*)

Correct an image based on the look-up table calculated ...

Parameters

- **image** – 2D-array with the image
- **shape_in** – shape of input image
- **shape_out** – shape of output image
- **LUT** – Look up table, here a 3-tuple array of ndarray
- **dummy** – value for invalid pixels
- **delta_dummy** – precision for invalid pixels

Returns corrected 2D image

pyFAI.ext._distortion.**correct_LUT**(*image*, *shape_in*, *shape_out*, *__Pyx_memviewslice LUT*, *dummy=None*, *delta_dummy=None*)

Correct an image based on the look-up table calculated ...

Parameters

- **image** – 2D-array with the image
- **shape_in** – shape of input image

- **shape_out** – shape of output image
- **LUT** – Look up table, here a 2D-array of struct
- **dummy** – value for invalid pixels
- **delta_dummy** – precision for invalid pixels

Returns corrected 2D image

pyFAI.ext._distortion.**uncorrect_CSR**(*image*, *shape*, *LUT*)

Take an image which has been corrected and transform it into it's raw (with loss of information)

Parameters

- **image** – 2D-array with the image
- **shape** – shape of output image
- **LUT** – Look up table, here a 3-tuple of ndarray

Returns uncorrected 2D image and a mask (pixels in raw image not existing)

pyFAI.ext._distortion.**uncorrect_LUT**(*image*, *shape*, *__Pyx_memviewslice LUT*)

Take an image which has been corrected and transform it into it's raw (with loss of information)

Parameters

- **image** – 2D-array with the image
- **shape** – shape of output image
- **LUT** – Look up table, here a 2D-array of struct

Returns uncorrected 2D image and a mask (pixels in raw image not existing)

6.32 ext._geometry Module

This extension is a fast-implementation for calculating the geometry, i.e. where every pixel of an array stays in space (x,y,z) or its (r, chi) coordinates.

pyFAI.ext._geometry.**calc_chi**(*double L*, *double rot1*, *double rot2*, *double rot3*, *ndarray pos1*, *ndarray pos2*, *ndarray pos3=None*)

Calculate the chi array (azimuthal angles) using OpenMP

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) tan(Chi) = X2 / X1

Parameters

- **L** – distance sample - PONI
- **rot1** – angle1
- **rot2** – angle2
- **rot3** – angle3
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)
- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector

Returns ndarray of double with same shape and size as pos1

pyFAI.ext._geometry.**calc_cosa**(*double L*, *ndarray pos1*, *ndarray pos2*, *ndarray pos3=None*)

Calculate the cosine of the incidence angle using OpenMP. Used for sensors thickness effect corrections

Parameters

- **L** – distance sample - PONI
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)
- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector

Returns ndarray of double with same shape and size as pos1

pyFAI.ext._geometry.**calc_pos_zyx**(*double L*, *double poni1*, *double poni2*, *double rot1*, *double rot2*, *double rot3*, *ndarray pos1*, *ndarray pos2*, *ndarray pos3=None*)

Calculate the 3D coordinates in the sample's referential

Parameters

- **L** – distance sample - PONI
- **poni1** – PONI coordinate along y axis
- **poni2** – PONI coordinate along x axis
- **rot1** – angle1
- **rot2** – angle2
- **rot3** – angle3
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)
- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector

Returns 3-tuple of ndarray of double with same shape and size as pos1

pyFAI.ext._geometry.**calc_q**(*double L*, *double rot1*, *double rot2*, *double rot3*, *ndarray pos1*, *ndarray pos2*, *double wavelength*, *pos3=None*)

Calculate the q (scattering vector) array using OpenMP

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) tan(Chi) = X2 / X1

Parameters

- **L** – distance sample - PONI
- **rot1** – angle1
- **rot2** – angle2
- **rot3** – angle3
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)
- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector
- **wavelength** – in meter to get q in nm-1

Returns ndarray of double with same shape and size as pos1

pyFAI.ext._geometry.**calc_r**(*double L*, *double rot1*, *double rot2*, *double rot3*, *ndarray pos1*, *ndarray pos2*, *ndarray pos3=None*)

Calculate the radius array (radial direction) in parallel

Parameters

- **L** – distance sample - PONI
- **rot1** – angle1
- **rot2** – angle2
- **rot3** – angle3
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)
- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector

Returns ndarray of double with same shape and size as pos1

pyFAI.ext._geometry.**calc_rad_azim**(*double L*, *double poni1*, *double poni2*, *double rot1*, *double rot2*, *double rot3*, *ndarray pos1*, *ndarray pos2*, *ndarray pos3=None*, *space='2th'*, *wavelength=None*)

Calculate the radial & azimutal position for each pixel from pos1, pos2, pos3.

Parameters

- **L** – distance sample - PONI
- **poni1** – PONI coordinate along y axis
- **poni2** – PONI coordinate along x axis
- **rot1** – angle1
- **rot2** – angle2
- **rot3** – angle3
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)
- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector
- **space** – can be "2th", "q" or "r" for radial units. Azimuthal units are radians

Returns ndarray of double with same shape and size as pos1 + (2,),

Raise KeyError when space is bad ! ValueError when wavelength is missing

pyFAI.ext._geometry.**calc_tth**(*double L*, *double rot1*, *double rot2*, *double rot3*, *ndarray pos1*, *ndarray pos2*, *ndarray pos3=None*)

Calculate the 2theta array (radial angle) in parallel

Parameters

- **L** – distance sample - PONI
- **rot1** – angle1
- **rot2** – angle2
- **rot3** – angle3
- **pos1** – numpy array with distances in meter along dim1 from PONI (Y)
- **pos2** – numpy array with distances in meter along dim2 from PONI (X)

- **pos3** – numpy array with distances in meter along Sample->PONI (Z), positive behind the detector

Returns ndarray of double with same shape and size as pos1

6.33 ext.histogram Module

Re-implementation of the numpy.histogram, optimized for azimuthal integration. Deprecated, will be replaced by silx.math.histogramnd Re-implementation of numpy histograms using OpenMP

pyFAI.ext.histogram.**histogram**(*ndarray pos*, *ndarray weights*, *int bins=100*, *bin_range=None*, *pixelSize_in_Pos=None*, *nthread=None*, *double empty=0.0*, *double normalization_factor=1.0*)

Calculates histogram of pos weighted by weights

@param pos: 2Theta array @param weights: array with intensities @param bins: number of output bins @param pixelSize_in_Pos: size of a pixels in 2theta: DESACTIVATED @param nthread: OpenMP is disabled. unused @param empty: value given to empty bins @param normalization_factor: divide the result by this value

@return 2theta, I, weighted histogram, raw histogram

pyFAI.ext.histogram.**histogram2d**(*ndarray pos0*, *ndarray pos1*, *bins*, *ndarray weights*, *split=False*, *nthread=None*, *double empty=0.0*, *double normalization_factor=1.0*)

Calculate 2D histogram of pos0,pos1 weighted by weights

@param pos0: 2Theta array @param pos1: Chi array @param weights: array with intensities @param bins: number of output bins int or 2-tuple of int @param split: pixel splitting is disabled in histogram @param nthread: maximum number of thread to use. By default: maximum available. @param empty: value given to empty bins @param normalization_factor: divide the result by this value

@return I, edges0, edges1, weighted histogram(2D), unweighted histogram (2D)

One can also limit this with OMP_NUM_THREADS environment variable

6.34 ext.marchingsquares Module

The marchingsquares algorithm is used for calculating an iso-contour curve (displayed on the screen while calibrating) but also to seed the points for the "massif" algoritm during recalib. Cythonized version of the marching square function for "isocontour" plot

pyFAI.ext.marchingsquares.**isocontour**(*img*, *isovalue=None*, *sorted=False*)

isocontour(img, isovalue=None)

Calculate the iso contours for the given 2D image. If isovalue is not given or None, a value between the min and max of the image is used.

@param img: 2D array representing the image @param isovalue: the value for which the iso_contour shall be calculated @param sorted: perform a sorting of the points to have them contiguous ?

Returns a pointset in which each two subsequent points form a line piece. This van be best visualized using "vv.plot(result, ls='+')".

pyFAI.ext.marchingsquares.**marching_squares**(*__Pyx_memviewslice img*, *double isovalue*, *__Pyx_memviewslice cellToEdge*, *__Pyx_memviewslice edgeToRelativePosX*, *__Pyx_memviewslice edgeToRelativePosY*)

pyFAI.ext.marchingsquares.**sort_edges**(*edges*)

Reorder edges in such a way they become contiguous

6.35 ext.morphology Module

The morphology extension provides a couple of binary morphology operations on images. They are also implemented in scipy.ndimage in the general case, but not as fast. A few binary morphology operation

pyFAI.ext.morphology.**binary_dilation**(*__Pyx_memviewslice image*, *float radius=1.0*)
Return fast binary morphological dilation of an image.

Morphological dilation sets a pixel at (i,j) to the maximum over all pixels in the neighborhood centered at (i,j). Dilation enlarges bright regions and shrinks dark regions.

:param image : ndarray :param radius: float :return: ndiamge

pyFAI.ext.morphology.**binary_erosion**(*__Pyx_memviewslice image*, *float radius=1.0*)
Return fast binary morphological erosion of an image.

Morphological erosion sets a pixel at (i,j) to the minimum over all pixels in the neighborhood centered at (i,j). Erosion shrinks bright regions and enlarges dark regions.

:param image : ndarray :param radius: float :return: ndiamge

6.36 ext.reconstruct Module

Very simple inpainting module for reconstructing the missing part of an image (masked) to be able to use more common algorithms. Cython module to reconstruct the masked values of an image

pyFAI.ext.reconstruct.**reconstruct**(*ndarray data*, *ndarray mask=None*, *dummy=None*, *delta_dummy=None*)
reconstruct missing part of an image (tries to be continuous)

Parameters

- **data** – the input image
- **mask** – where data should be reconstructed.
- **dummy** – value of the dummy (masked out) data
- **delta_dummy** – precision for dummy values

Returns reconstructed image.

6.37 ext.relabel Module

Relabel regions, used to flag from largest regions to the smallest A module to relabel regions

pyFAI.ext.relabel.**countThem**(*ndarray label*, *ndarray data*, *ndarray blured*)
Count :param label: 2D array containing labeled zones :param data: 2D array containing the raw data :param blured: 2D array containing the blured data :return: 2D arrays containing:

•count pixels in labelled zone: label == index).sum()

•max of data in that zone: data[label == index].max()

•max of blured in that zone: blured[label == index].max()

•data-blured where data is max.

6.38 ext.preproc Module

Contains a preprocessing function in charge of the dark-current subtraction, flat-field normalization, ... taking care of masked values and normalization.

pyFAI.ext.preproc.**preproc**(*data*, *dark=None*, *flat=None*, *solidangle=None*, *polarization=None*, *absorption=None*, *mask=None*, *dummy=None*, *delta_dummy=None*, *float normalization_factor=1.0*, *empty=None*)

Common preprocessing step for all

Parameters

- **data** – raw value, as a numpy array, 1D or 2D
- **mask** – array non null where data should be ignored
- **dummy** – value of invalid data
- **delta_dummy** – precision for invalid data
- **dark** – array containing the value of the dark noise, to be subtracted
- **flat** – Array containing the flatfield image. It is also checked for dummies if relevant.
- **solidangle** – the value of the solid_angle. This processing may be performed during the rebinning instead. left for compatibility
- **polarization** – Correction for polarization of the incident beam
- **absorption** – Correction for absorption in the sensor volume
- **normalization_factor** – final value is divided by this
- **empty** – value to be given for empty bins

All calculation are performed in single precision floating point.

NaN are always considered as invalid

if neither empty nor dummy is provided, empty pixels are 0

6.39 ext._tree Module

The tree is used in file hierarchy tree for the diff_map graphical user interface.

class pyFAI.ext._tree.**TreeItem**

Bases: object

TreeItem(str label=None, TreeItem parent=None)

Node of a tree ...

Each node contains:

•children: list

•parent: TreeItem parent

•label: str

•order: int

•type: str can be “dir”, “file”, “group” or “dataset”

•extra: any object

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

add_child(*self*, *TreeItem child*)

children

children: list

extra
extra: object

first (*self*) → TreeItem

get (*self*, *str label*) → TreeItem

has_child (*self*, *str label*) → bool

label
label: str

last (*self*) → TreeItem

name

next (*self*) → TreeItem

order
order: 'int'

parent
parent: pyFAI.ext._tree.TreeItem

previous (*self*) → TreeItem

size

sort (*self*)

type
type: str

update (*self*, *TreeItem new_root*)
Add new children in tree

6.40 ext.watershed Module

Peak peaking via inverse watershed for connecting region of high intensity Inverse watershed for connecting region of high intensity

class pyFAI.ext.watershed.**Bilinear**
Bases: object

Bilinear interpolator for finding max.

Instance attribute defined in pxd file

cp_local_maxi (*self*, *size_t x*) → size_t

data

f_cy (*self*, *x*)
Function f((y,x)) where f is a continuous function (y,x) are pixel coordinates @param x: 2-tuple of float @return: Interpolated signal from the image (negative for minimizer)

height

local_maxi (*self*, *x*)
Return the local maximum ... with sub-pixel refinement

@param x: 2-tuple of integers @param w: half with of the window: 1 or 2 are advised @return: 2-tuple of float with the nearest local maximum

Sub-pixel refinement: Second order Taylor expansion of the function; first derivative is null delta = x-i = -Inverse[Hessian].gradient

if Hessian is singular or |delta|>1: use a center of mass.

maxi

mini

width

class pyFAI.ext.watershed.**InverseWatershed**(*self*, *data*, *thres=1.0*)
Bases: object

Idea:

•label all peaks

•define region around those peaks which raise always to this peak

•define the border of such region

•search for the pass between two peaks

•merge region with high pass between them

NAME = 'Inverse watershed'

VERSION = '1.0'

__init__(*self*, *data*, *thres=1.0*)

Parameters **data** – 2d image as numpy array

init(*self*)

init_borders(*self*)

init_labels(*self*)

init_pass(*self*)

init_regions(*self*)

classmethod load(*cls*, *fname*)
Load data from a HDF5 file

merge_intense(*self*, *thres=1.0*)
Merge groups then (pass-mini)/(maxi-mini) >=thres

merge_singleton(*self*)
merge single pixel region

merge_twins(*self*)
Twins are two peak region which are best linked together: A -> B and B -> A

peaks_from_area(*self*, *mask*, *Imin=None*, *keep=None*, *bool refine=True*, *float dmin=0.0*, ***kwarg*)

Parameters

- **mask** – mask of data points valid
- **Imin** – Minimum intensity for a peak
- **keep** – Number of points to keep
- **refine** – refine sub-pixel position
- **dmin** – minimum distance from

save(*self*, *fname*)
Save all regions into a HDF5 file

class pyFAI.ext.watershed.**Region**
Bases: object

border

get_borders(*self*)

get_highest_pass(*self*)

get_index(*self*)

get_maxi(*self*)

get_mini(*self*)

get_neighbors(*self*)

get_pass_to(*self*)

get_size(*self*)

highest_pass

index

init_values(*self*, *__Pyx_memviewslice flat*)
Initialize the values : maxi, mini and pass both height and so on :param flat: flat view on the data (intensity) :return: True if there is a problem and the region should be removed

maxi

merge(*self*, *Region other*)
merge 2 regions

mini

neighbors

pass_to

peaks

size

CHAPTER

SEVEN

INSTALLATION OF PYTHON FAST AZIMUTHAL INTEGRATION LIBRARY

7.1 Abstract

Installation procedure for all main operating systems

7.2 Hardware requirement

PyFAI has been tested on various hardware: i386, x86_64, PPC64le, ARM. The main constrain may be the memory requirement: 2GB of memory is a minimal requirement to run the tests. The program may run with less but "MemoryError" are expected (appearing sometimes as segmentation faults). As a consequence, a 64-bits operating system with enough memory is strongly advised.

7.3 Dependencies

PyFAI is a Python library which relies on the scientific stack (numpy, scipy, matplotlib)

- Python: version 2.7, 3.4 and 3.5. Support for 2.6, 3.2 and 3.3 has been dropped in v0.12
- NumPy: version 1.4 or newer
- SciPy: version 0.7 or newer
- Matplotlib: verson 0.99 or newer
- FabIO: version 0.08 or newer
- h5py (to access HDF5 files)

There are plenty of optional dependencies which will not prevent pyFAI from working by may impair performances or prevent tools from properly working:

- pyopencl (for GPU computing)
- fftw (for image analysis)
- pymca (for mask drawing)
- PyQt4 or PySide (for the graphical user interface)

7.4 Build dependencies:

In addition to the run dependencies, pyFAI needs a C compiler to build extensions.

C files are generated from cython source and distributed. The distributed version correspond to OpenMP version. Non-OpenMP version needs to be built from cython source code (especially on MacOSX). If you want to generate your own C files, make sure your local Cython version is sufficiently recent (>0.20).

7.5 Building procedure

```
python setup.py build
pip install . --upgrade
```

There are few specific options to `setup.py`:

- `--no-cython`: Prevent Cython (even if present) to re-generate the C source code. Use the one provided by the development team.
- `--no-openmp`: Recompiles the Cython code without OpenMP support (default under MacOSX).
- `--openmp`: Recompiles the Cython code with OpenMP support (Default under Windows and Linux).
- `--with-testimages`: build the source distribution including all test images. Download 200MB of test images to create a self consistent tar-ball.

7.6 Detailed installation procedure on different operating systems

7.6.1 Installation procedure on Linux

We cover first Debian-like distribution, then a generic recipie for all other version is given.

Installation procedure on Debian/Ubuntu

PyFAI has been designed and originally developed on Ubuntu 10.04 and debian6. Now, the pyFAI library is included into debian7, 8 and any recent Ubuntu and Mint distribution. To install the package provided by the distribution, use:

```
sudo apt-get install pyfai
```

The issue with distribution based installation is the obsolescence of the version available.

Debian7 and Ubuntu 12.04

To build a more recent version, pyFAI provides you a small scripts which builds a *debian* package and installs it. It relies on *stdeb* and provides a single package with everything inside. You will be prompted for your password to gain root access in order to be able to install the freshly built package.

```
sudo apt-get install python-stdeb cython python-fabio
wget https://github.com/silx-kit/pyFAI/archive/master.zip
unzip master.zip
cd pyFAI-master
./build-deb7.sh
```

Debian8 and newer

Thanks to the work of Frédéric-Emmanuel Picca, the debian package of pyFAI provides a pretty good template which allows continuous builds.

From silx repository You can automatically install the latest nightly built of pyFAI with:

```
wget http://www.silx.org/pub/debian/silx.list
wget http://www.silx.org/pub/debian/silx.pref
sudo mv silx.list /etc/apt/sources.list.d/
sudo mv silx.pref /etc/apt/preferences.d/
sudo apt-get update
sudo apt-get install pyfai
```

Nota: The nightly built packages are not signed, hence you will be prompted to install non-signed packages.

Build from sources One can also built from sources:

```
sudo apt-get install cython cython-dbg cython3 cython3-dbg debhelper dh-python \
python-all-dev python-all-dbg python-fabio python-fabio-dbg python-fftw python-h5py \
python-lxml python-lxml-dbg python-matplotlib python-matplotlib-dbg python-numpy\
python-numpy-dbg python-qt4 python-qt4-dbg python-scipy python-scipy-dbg python-sphinx \
python-sphinxcontrib.programoutput python-tk python-tk-dbg python3-all-dev python3-all-dbg \
python3-fabio python3-fabio-dbg python3-lxml python3-lxml-dbg python3-matplotlib \
python3-matplotlib-dbg python3-numpy python3-numpy-dbg python3-pyqt4 python3-pyqt4-dbg \
python3-scipy python3-scipy-dbg python3-sphinx python3-sphinxcontrib.programoutput \
python3-tk python3-tk-dbg
wget https://github.com/silx-kit/pyFAI/archive/master.zip
unzip master.zip
cd pyFAI-master
./build-deb8.sh
```

The first line is really long and defines all the dependence tree for building *debian* package, including debug and documentation. The build procedure last for a few minutes and you will be prompted for your password in order to install the freshly built packages. The *deb-*files, available in the *package* directory are backports for your local installation.

Installation procedure on other linux distibution

If your distribution does not provide you pyFAI packages, using the **PIP** way is advised, via wheels packages. First install *pip* and *wheel*:

```
wget https://bootstrap.pypa.io/get-pip.py
sudo python get-pip.py
sudo pip install pyFAI
```

Or you can install pyFAI from the sources:

```
wget https://github.com/silx-kit/pyFAI/archive/master.zip
unzip master.zip
cd pyFAI-master
python setup.py build test
sudo pip install . --upgrade
```

Nota: The usage of "python setup.py install" is now deprecated. It causes much more trouble as there is no installed file tracking, hence no way to de-install properly the package.

7.6.2 Installation procedure on MacOSX

Using PIP

To install pyFAI on an *Apple* computer you will need a scientific Python stack. MacOSX provides by default Python2.7 with Numpy which is a good basis.

```
sudo pip install matplotlib --upgrade
sudo pip install scipy --upgrade
sudo pip install fabio --upgrade
sudo pip install h5py --upgrade
sudo pip install cython --upgrade
sudo pip install pyFAI --upgrade
```

If you get an error about the local "UTF-8", try to:

```
export LC_ALL=C
```

Before the installation.

Installation from sources

Get the sources from Github:

```
wget https://github.com/silx-kit/pyFAI/archive/master.zip
unzip master.zip
cd pyFAI-master
```

About OpenMP

OpenMP is a way to write multi-threaded code, running on multiple processors simultaneously. PyFAI makes heavy use of OpenMP, but there is an issue with recent versions of MacOSX (>v10.6) where the default compiler of Apple, *Xcode*, dropped the support for OpenMP.

There are two ways to compile pyFAI on MacOSX:

- Using *Xcode* and de-activating OpenMP in pyFAI
- Using another compiler which supports OpenMP

Using Xcode

To build pyFAI from sources, a C-compiler is needed. On an *Apple* computer, the default compiler is Xcode, and it is available for free on the **AppStore**. As pyFAI has by default OpenMP activated, and it needs to be de-activated, one needs to regenerate all Cython files without OpenMP.

```
sudo pip install cython --upgrade
rm pyFAI/ext/*.c
python setup.py build --no-openmp
python setup.py bdist_wheel
sudo pip install --find-links=dist/ --pre --no-index --upgrade pyFAI
```

Using gcc or clang

If you want to keep the OpenMP feature (which makes the processing slightly faster), the alternative is to install another compiler like gcc or clang on your *Apple* computer. As gcc & clang support OpenMP, there is no need to re-generate the cython files.

```
CC=gcc python setup.py build --openmp
python setup.py bdist_wheel
sudo pip install --find-links=dist/ --pre --no-index --upgrade pyFAI
```

Nota: The usage of "python setup.py install" is now deprecated. It causes much more trouble as there is no installed file tracking, hence no way to de-install properly a package.

7.6.3 Installation procedure on Windows

PyFAI is a Python library. Even if you are only interested in some tool like pyFAI-calib or pyFAI-integrate, you need to install the complete library (for now). This is usually performed in 3 steps:

1. install Python,
2. install the scientific Python stack
3. install pyFAI itself.

Get Python

Unlike on Unix computers, Python is not available by default on Windows computers. We recommend you to install the 64 bit version of Python, preferably the latest 64 bits version from the 2.7 series. But Python 3.4 and 3.5 are also very good candidates. Python 2.6, 3.2 and 3.3 are no more supported since pyFAI v0.12.

The 64 bits version is strongly advised if your hardware and operating system supports it, as the 32 bits versions is limited to 2 GB of memory, hence unable to treat large images (like 4096 x 4096). The test suite is not passing on Windows 32 bits due to the limited amount of memory available to the Python process, nevertheless, pyFAI is running on Winodws 32 bits (but not as well).

Alternative Scientific Python stacks exists, like Enthought Python Distribution , Canopy, Anaconda, PythonXY or WinPython. They all offer most of the scientific packages already installed which makes the installation of dependencies much easier. On the other hand, they all offer different packaging system and we cannot support all of them. Moreover, distribution from *Enthought* and *Continuum* are not free so you should be able to get support from those companies.

Nota: any flavor of those Python distribution is probably incompatible with any other due to change in compiler or Python compilation options. Mixing them is really looking for trouble, hence strongly discouraged. If you want an advice on which scientific python distribution for Windows to use, I would recommend WinPython.

Install PIP

PIP is the package management system for Python, it connects to the Python Package Index, download and install software packages from there.

PIP has revolutionize the way Python libraries are installed as it is able to select the right build for your system, or compile them from the sources, which could be extremely tricky otherwise. If you installed python 2.7.10, 3.4 or newer, PIP is already installed. If **pip** is not yet installed on your system, download get_pip.py and run it:

```
python get-pip.py
```

Assuming python.exe is already in your PATH.

Nota: Because PIP connects to the network, the *http_proxy* and *https_proxy* environment variable may need to be set-up properly. At ESRF, please get in contact with the hotline (24-24) to retrive those information.

Install the scientific stack

The strict dependencies for pyFAI are:

- NumPy
- SciPy
- matplotlib
- FabIO
- h5py

Recommended dependencies are:

- cython
- h5py
- pyopencl
- PyQt4
- pymca
- rfoo
- pyfftw3
- lxml

Using PIP

Most of the dependencies are available via PIP:

```
pip install numpy --upgrade
pip install scipy --upgrade
pip install matplotlib --upgrade
pip install fabio --upgrade
pip install PyQt4 --upgrade
```

Note that numpy/scipy/matplotlib are already installed in most "Scientific Python distribution"

If one of the dependency is not available as a Wheel (i.e. binary package) but only as a source package, a compiler will be required. In this case, see the next paragraph. The generalization of Wheel packages should help and the installation of binary modules should become easier.

Nota: This requires a network access and correct proxy settings. At ESRF, please get in contact with the hotline (24-24) to retrive those information.

```
set http_proxy=http://proxy.site.com:3128
set https_proxy=http://proxy.site.com:3128
```

Using Christoph Gohlke repository

Christoph Gohlke is a researcher at Laboratory for Fluorescence Dynamics, University of California, Irvine. He is maintaining a large repository Python extension (actually, all we need :) for Windows. Check twice the Python version and the Windows version (win32 or win_amd64) before downloading.

Moreover the libraries he provides are linked against the MKL library from Intel which makes his packages faster then what you would get by simply recompiling them.

Christoph now provides packages as wheels. To install them, download the wheels and use PIP:

```
pip install numpy*.whl
```

Alternatively, you can use the wheelhouse of the silx project:

```
pip install --trusted-host www.silx.org --find-links http://www.silx.org/pub/wheelhouse/ numpy sc
```

Install pyFAI via PIP

The latest stable release of pyFAI should also be PIP-installable (starting at version 0.10.3):

```
pip install pyFAI --upgrade
```

Install pyFAI from sources

The sources of pyFAI are available at https://github.com/silx-kit/pyFAI/releases the development is performed on https://github.com/kif/pyFAI

In addition to the Python interpreter, you will need *the* C compiler compatible with your Python interpreter, for example you can find the one for Python2.7 at: http://aka.ms/vcpython27

To upgrade the C-code in pyFAI, one needs in addition Cython:

```
pip install cython --upgrade
python setup.py bdist_wheel
pip install --pre --no-index --find-links dist/ pyFAI
```

Troubleshooting

This section contains some tips on windows.

Side-by-side error

When starting pyFAI you get a side-by-side error like:

```
ImportError: DLL load failed: The application has failed to start because its
side-by-side configuration is incorrect. Please see the application event log or
use the command-line sxstrace.exe tool for more detail.
```

This means you are using a version of pyFAI which was compiled using the MSVC compiler (maybe not on your computer) but the Microsoft Visual C++ Redistributable Package is missing. For Python2.7, 64bits the missing DLL can be downloaded from:

```
http://www.microsoft.com/en-us/download/confirmation.aspx?id=2092
```

7.7 Test suites

PyFAI comes with a test suite to ensure all core functionalities are working as expected and numerical results are correct:

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

Nota: to run the test, an internet connection is needed as 200MB of test images need to be download. You may have to set the environment variable *http_proxy* and *https_proxy* according to the networking environment you are in. Specifically at ESRF, please phone the hotline at 24-24 to get those information.

7.8 Environment variables

PyFAI can use a certain number of environment variable to modify its default behavior:

- PYFAI_OPENCL: set to “0” to disable the use of OpenCL
- PYFAI_DATA: path with gui, calibrant, ...
- PYFAI_TESTIMAGES: path wit test images (if absent, they get downloaded from the internet)

CHAPTER

EIGHT

PYFAI ECOSYSTEM

8.1 Software pyFAI is relying on

PyFAI is relying on the full Python scientific stack which includes [NumPy], [SciPy], [Matplotlib], [PyOpenCL] but also on some ESRF-developped code:

8.1.1 FabIO

PyFAI is using FabIO everywhere access to a 2D images is needed. The *fabio_viewer* is also a lightweight convenient viewer for diffraction images. It has been described in doi:10.1107/S0021889813000150

8.1.2 PyMca

The X-ray Fluorescence Toolkit provides convenient tools for HDF5 file browsing and mask drawing. It has been described in doi:10.1016/j.sab.2006.12.002

8.1.3 Silx

The silx toolkit is currently onging development. Future releases of pyFAI will use its input/output and graphical visualization capabilities

8.2 Program using pyFAI as a library

8.2.1 Bubble

Client-server program to perform azimuthal integration online. Developed for the SNBL and Dubble beamlines by Vadim DIADKIN and available from this mercurial repository.

8.2.2 Dahu

Dahu is a lightweight plugin based framework available from this git repository. Lighter then EDNA, it is technically a JSON-RPC server over Tango. Used on TRUSAXS beamline at ESRF (ID02), ID15 and ID31, dahu uses pyFAI to process data up to the kHz range.

8.2.3 Dioptas

Graphical user interface for high-pressure diffraction, developed at the APS synchrotron by C. Prescher and described in: doi:10.1080/08957959.2015.1059835

The amount of data collected during synchrotron X-ray diffraction (XRD) experiments is constantly increasing. Most of the time, the data are collected with image detectors, which necessitates the use of image reduction/integration routines to extract structural information from measured XRD patterns. This step turns out to be a bottleneck in the data processing procedure due to a lack of suitable software packages. In particular, fast-running synchrotron experiments require online data reduction and analysis in real time so that experimental parameters can be adjusted interactively. Dioptas is a Python-based program for on-the-fly data processing and exploration of two-dimensional X-ray diffraction area detector data, specifically designed for the large amount of data collected at XRD beamlines at synchrotrons. Its fast data reduction algorithm and graphical data exploration capabilities make it ideal for online data processing during XRD experiments and batch post-processing of large numbers of images.

8.2.4 Dpdak

Graphical user interface for small angle diffusion, developed at the Petra III synchrotron by G. Benecke and co-workers and described in doi:10.1107/S1600576714019773

X-ray scattering experiments at synchrotron sources are characterized by large and constantly increasing amounts of data. The great number of files generated during a synchrotron experiment is often a limiting factor in the analysis of the data, since appropriate software is rarely available to perform fast and tailored data processing. Furthermore, it is often necessary to perform online data reduction and analysis during the experiment in order to interactively optimize experimental design. This article presents an open-source software package developed to process large amounts of data from synchrotron scattering experiments. These data reduction processes involve calibration and correction of raw data, one- or two-dimensional integration, as well as fitting and further analysis of the data, including the extraction of certain parameters. The software, DPDAK (directly programmable data analysis kit), is based on a plug-in structure and allows individual extension in accordance with the requirements of the user. The article demonstrates the use of DPDAK for on- and offline analysis of scanning small-angle X-ray scattering (SAXS) data on biological samples and microfluidic systems, as well as for a comprehensive analysis of grazing-incidence SAXS data. In addition to a comparison with existing software packages, the structure of DPDAK and the possibilities and limitations are discussed.

8.2.5 EDNA

EDNA is a framework for developing plugin-based applications especially for online data analysis in the X-ray experiments field (http://edna-site.org) A EDNA data analysis server is using pyFAI as an integration engine (on the GPU) on the ESRF BioSaxs beamline, BM29. The server is running 24x7 with a processing frequency from 0.1 to 10 Hz.

8.2.6 LImA

The Library for Image Acquisition, developped at the European synchrotron is used worldwide to control any types of cameras. A pyFAI plugin has been written to integrate images on the fly without saving them. (no more tested).

8.2.7 NanoPeakCell

NanoPeakCell (NPC) is a python-software intended to pre-process your serial crystallography raw-data into ready-to-be-inedexed images with CrystFEL, cctbx.xfel and nXDS. NPC is able to process data recorded at SACLA and LCLS XFELS, as well as data recorded at any synchrotron beamline. A graphical interface is deployed to visualize your raw and pre-processed data.

Developed at IBS (Grenoble) by N. Coquelle

8.2.8 pygix

A Python library for reduction of 2D grazing-incidence X-ray scattering data developped at ESRF (ID13) by Thomas DANE.

Grazing-incidence X-ray scattering techniques (GISAXS, GIWAXS/GID) allow the study of thin films on surfaces that would otherwise be unmeasurable in standard transmission geometry experiments. The fixed incident X-ray angle gives rise to a distortion in the diffraction patterns, which is extreme at wide-angles. The pygix library provides routines for projecting 2D detector images into corrected reciprocal space maps, radial transformations and line profile extraction using pyFAI's regrouping functions.

8.2.9 PySAXS

Python for Small Angle X-ray Scattering data acquisition, treatment and computation of model SAXS intensities.

Developed at CEA Saclay by O. Taché and available on PyPI.

8.2.10 Xi-cam

Xi-cam is developed by Ronald Pandolfi, Dinesh Kumar, Singanallur Venkatakrishnan and Alexander Hexemer at ALS.

Xi-cam aims to provide a community driven platform for multimodal analysis in synchrotron science. The platform core provides a robust plugin infrastructure for extensibility, allowing continuing development to simply add further functionality. Current modules include tools for characterization with (GI)SAXS, Tomography, and XAS. This will continue to serve as a development base as algorithms for multimodal analysis develop.

Seamless remote data access, visualization and analysis are key elements of Xi-CAM, and will become critical to synchrotron data infrastructure as expectations for future data volume and acquisition rates rise with continuously increasing throughputs. The highly interactive design elements of Xi-cam will similarly support a generation of users which depend on immediate data quality feedback during high-throughput or burst acquisition modes.

8.2.11 xPDFsuite

Developed by the Billinge Group, this commercial software is described in arXiv 1402.3163 (2014)

xPDFsuite is an application that facilitates the workflow of atomic pair distribution function analysis of x-ray diffraction measurements from complex materials. It is specially designed to help the scientist visualize, handle and process large numbers of datasets that is common when working with high throughput modern synchrotron sources. It has a full-featured interactive graphical user interface (GUI) with 3D and 3D graphics for plotting data and it incorporates a number of powerful packages for integrating 2D powder diffraction images, analyzing the curves to obtain PDFs and then tools for assessing the data and modeling it. It is available from diffpy.org.

CHAPTER

NINE

PROJECT

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

PyFAI is an open source project licensed under the GPL (switching to MIT) mainly written in Python (v2.7 or newer, 3.4 or newer) and heavily relying on the Python scientific ecosystem: numpy, scipy and matplotlib. It provides high performances image treatment thanks to Cython and OpenCL... but only a C-compiler is needed to build it.

9.1 Programming language

PyFAI is a Python project but uses many programming languages:

- 23000 lines of Python (plus 5000 for the test)
- 8000 lines of Cython which are converted into ... C (about half a million lines)
- 5000 lines of OpenCL kernels

The OpenCL code has been tested using:

- Nvidia OpenCL v1.1 and v1.2 on Linux, Windows (GPU device)
- Intel OpenCL v1.2 on Linux and Windows (CPU and ACC (Phi) devices)
- AMD OpenCL v1.2 on Linux and Windows (CPU and GPU device)
- Apple OpenCL v1.2 on MacOSX (CPU and GPU)
- Beignet OpenCL v1.2 on Linux (GPU device)
- Pocl OpenCL v1.2 on Linux (CPU device)

9.2 Repository

The project is hosted on GitHub: https://github.com/silx-kit/pyFAI

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

Everybody is welcome to fork the project and adapt it to his own needs: CEA-Saclay, Synchrotrons Soleil, Desy and APS have already done so. Collaboration is encouraged and new developments can be submitted and merged into the main branch via pull-requests.

9.3 Getting help

A mailing list: pyfai@esrf.fr is publicly available, it is the best place to ask your questions: the author and many advanced users are there and willing to help you. To subscribe to this mailing list, send an email to sympa@esrf.fr with "subscribe pyfai" as subject.

On this mailing list, you will have information about release of the software, new features available and meet experts to help you solve issues related to azimuthal integration and diffraction in general. It also provides a knowledge-base of most frequently asked question which needs to be integrated into this documentation.

The volume of email on the list remains low, so you can subscribe without being too much spammed. As the mailing list is archived, and can be consulted at: http://www.edna-site.org/lurker, you can also check the volume of the list.

If you think you are facing a bug, the best is to create a new issue on the GitHub page (you will need a GitHub account for that).

Direct contact with authors is discouraged: pyFAI is open source software that we develop to aid the research community in doing what they do best. While we do enjoy doing this, we would not be able to dream of spending nearly as much time with pyFAI as we do if it wasn't for your support. Interest of the scientific community (via a lively mailing list) and citation in scientific publication for our software is one of the main criterion for ESRF management when deciding if they should continue funding development.

9.4 Run dependencies

- Python version 2.7, 3.4, 3.5
- NumPy
- SciPy
- Matplotlib
- FabIO
- h5py
- pyopencl (optional)
- PyQt4 or PySide (for the graphical user interface)

9.5 Build dependencies

In addition to the run dependencies, pyFAI needs a C compiler.

There is an issue with MacOS (v10.8 onwards) where the default compiler (Xcode5 or newer) dropped the support for OpenMP. On this platform pyFAI will enforce the generation of C-files from Cython sources (making Cython a build-dependency on MacOS) without support of OpenMP (options: –no-openmp –force-cython). On OSX, an alternative is to install a recent version of GCC (>=4.2) and to use it for compiling pyFAI. The options to be used then are * –force-cython –openmp*.

Otherwise, C files are which are provided with pyFAI sources are directly useable and Cython is only needed for developing new binary modules. If you want to generate your own C files, make sure your local Cython version is recent enough (v0.21 and newer), unless your Cython files will not be translated to C, nor used.

9.6 Building procedure

9.6.1 As most of the Python projects:

```
python setup.py build bdist_wheel
pip install dist/pyFAI-0.13.0*.whl --upgrade
```

There are few specific options to setup.py:

- –no-cython: do not use cython (even if present) and use the C source code provided by the development team
- –force-cython: enforce the regeneration of all C-files from cython sources
- –no-openmp: if you compiler lacks OpenMP support, like Xcode on MacOS.
- –openmp: enforce the use of OpenMP.
- –with-testimages: build the source distribution including all test images. Downloads 200MB of test images to create a self consistent tar-ball.

9.7 Test suites

To test the installed version of pyFAI:

```
python run_test.py
```

or from python:

```
import pyFAI
pyFAI.tests()
```

Some **Warning** messages are normal as the test procedure also tests corner cases.

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

Setting the environment variable http_proxy can be necessary (depending on your network):

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

Especially at ESRF, the configuration of the network proxy can be obtained by asking at the helpdesk: helpdesk@esrf.fr

To test the development version (built but not yet installed):

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

or

```
python setup.py build
python run_test.py -i
```

PyFAI comes with 40 test-suites (338 tests in total) representing a coverage of 60%. This ensures both non regression over time and ease the distribution under different platforms: pyFAI runs under Linux, MacOSX and Windows (in each case in 32 and 64 bits). Test may not pass on computer featuring less than 2GB of memory or 32 bit architectures.

Note:: The test coverage tool does not count lines of Cython, nor those of OpenCL. Anyway test coverage is a tool for developers about possible weaknesses of their code and not a management indicator.

Test coverage report for pyFAI

Measured on *pyFAI* version 0.13.0, 01/12/2016

Table 9.1: Test suite coverage

Name	Stmts	Exec	Cover
__init__.py	35	20	57.1 %
_version.py	41	39	95.1 %
average.py	526	389	74.0 %
azimuthalIntegrator.py	1207	926	76.7 %
blob_detection.py	504	194	38.5 %
calibrant.py	338	244	72.2 %
containers.py	79	72	91.1 %
decorators.py	33	23	69.7 %
detectors.py	1336	1050	78.6 %
directories.py	34	21	61.8 %
distortion.py	557	166	29.8 %
ext/__init__.py	0	0	0.0 %
geometry.py	959	759	79.1 %
geometryRefinement.py	485	176	36.3 %
gui/__init__.py	4	4	100.0 %
gui/matplotlib.py	24	17	70.8 %
gui/qt.py	100	36	36.0 %
gui/utils.py	34	14	41.2 %
integrate_widget.py	453	268	59.2 %
io.py	579	294	50.8 %
massif.py	201	132	65.7 %
multi_geometry.py	128	94	73.4 %
ocl_azim.py	269	191	71.0 %
ocl_azim_csr.py	240	184	76.7 %
ocl_azim_lut.py	231	177	76.6 %
ocl_sort.py	177	136	76.8 %
opencl.py	206	146	70.9 %
peak_picker.py	758	197	26.0 %
spline.py	432	165	38.2 %
units.py	80	76	95.0 %
utils/__init__.py	467	290	62.1 %
utils/bayes.py	92	60	65.2 %
utils/shell.py	44	41	93.2 %
utils/stringutil.py	25	24	96.0 %
worker.py	358	140	39.1 %
pyFAI total	11036	6765	61.3 %

9.8 Continuous integration

This software engineering practice consists in merging all developer working copies to a shared mainline several times a day and build the whole project for multiple targets.

9.8.1 On Debian 8 - Jessie

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 debian8-64 bits. To install them:

```
sudo apt-get update
sudo apt-get install pyfai
```

You have to accept non-signed packages because they are automatically built.

In addition some "cloud-based" tools are used to ensure a larger coverage of operating systems/environment. They rely on a "local wheelhouse".

Those wheels are optimized for Travis-CI, AppVeyor and ReadTheDocs, using them is not recommended as your Python configuration may differ (and those libraries could even crash your system).

9.8.2 Linux

Travis provides continuous integration on Linux, 64 bits computer with Python 2.7, 3.4 and 3.5.

The builds cannot yet be retrieved with Travis-CI, but manylinux-wheels are on the radar.

9.8.3 AppVeyor

AppVeyor provides continuous integration on Windows, 64 bits computer with Python 2.7 and 3.4. Successful builds provide installers for pyFAI as *wheels* and *msi*, they are anonymously available as *artifacts*. Due to the limitation of AppVeyor's build system, those installers have openMP disabled.

9.9 List of contributors in code

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

As of 06/2016:

- Jérôme Kieffer (ESRF)
- Frédéric-Emmanuel Picca (Soleil)
- Aurore Deschildre (ESRF)
- Giannis Ashiotis (ESRF)
- Dimitrios Karkoulis (ESRF)
- Valentin Valls (ESRF)
- Jon Wright (ESRF)
- Zubair Nawaz (Sesame)
- Amund Hov (ESRF)
- Dodogerstlin @github
- Gunthard Benecke (Desy)
- Gero Flucke (Desy)
- Vadim Dyadkin (ESRF)
- Sigmund Neher (GWDG)
- Thomas Vincent (ESRF)

9.10 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)

9.11 List of supporters

- LinkSCEEM project: porting to OpenCL
- ESRF ID11: Provided manpower in 2012 and 2013 and beamtime
- ESRF ID13: Provided manpower in 2012, 2013, 2014, 2015, 2016 and beamtime
- ESRF ID29: provided manpower in 2013 (MX-calibrate)
- ESRF ID02: provided manpower 2014, 2016
- ESRF ID15: provide manpower 2015, 2016
- ESRF ID21: provide manpower 2015, 2016
- ESRF ID31: provide manpower 2016

CHAPTER

TEN

CHANGELOG OF VERSIONS

10.1 0.13.0: 01/12/2016

- Global improvement of tests, packaging, code quality, documentation and project tools
- **Scripts**
  - Add support for multiframe formats on pyFAI-average
  - Add support for monitoring correction from header file (on pyFAI-average)
  - Add progressbar in the shell (on pyFAI-average and pyFAI-integrate)
  - Script drawMask_pymca is renamed into pyFAI-drawmask
  - Rework of the drawmask GUI using silx
  - pyFAI-drawmask do not have anymore hard dependency on PyMCA
  - pyFAI-integrate can now be used without qt dependency (–no-gui)
  - Fix the script to support both Python 2 and 3 (pyFAI-calib, pyFAI-benchmark)
  - Fix selection of units on diff-map (the user selection was not propagated)
- **For users**
  - More source code in MIT license
  - Update name and specification for cameras
  - Add cameras: Eiger500k, RaspberryPi5M, RaspberryPi8M
  - Fix Xpad S540 flat detector geometry
  - Fix definition of CeO2 calibrant
  - Add mask and flat on multi-geometry
  - Fix solid angle of the multi-geometry
  - Fix geometry processing for custom output space
  - Fix normalization factor and variance
  - Add support for Qt5
  - Add support for Debian 9 packaging
- **For developers**
  - Create common preprocessing for distortion correction
  - Create common image preprocessing using Cython (NaN filter, flatfield, dark, polarisation)
  - Refactoring of units module. It allows to register custom units.
  - Worker can now use Writer

  - Worker polarization argument is renamed into polarization_factor
  - Remove the dependency from python-fftw3, use numpy instead
  - Remove QtWebKit dependency
  - Fix un-correction of images using sparse matrix from scipy

10.2 0.12.0: 06/06/2016

- Continuous integration on linux, windows using Python 2.7 and 3.4+
- Drop support of Python 2.6, 3.2, 3.3 and debian6 packaging
- New radial output units: Reciprocal spacing squared and log(q) **ID02**
- GPU accelerate version of ai.separate (Bragg & amorphous) **ID13**
- Quantile filtering in pyFAI-average **ID02**
- New graphical application for diffraction imaging **ID21**
- Migrate to a common structure with *silx* (reorganize tests, benchmarks, ...)
- Extensions (binary sub-modules) have all been moved to *ext* directory
- Many improvements multigeometry integrators
- Compatibility with the copy module (copy.deepcopy) for azimuthal integrator **ID02**
- Distortion correction works also for non-contiguous detectors
- **Update documentation and provide advanced tutorials:**
  - Introduction to pyFAI using the jupyter notebook
  - detector calibration **ID15, BM02**
  - Correction of detector distortion, examples of pixel detectors.
  - calibrant calculation **ID30**
  - error handling **ID02, BM29**
- pyFAI-integrate can now be used with or without GUI
- **Many new detectors (ADSC, Pilatus CdTe, Apex II, Pixium):**
  - support for non-flat/curved detectors (Aarhus)
  - non-contiguous detectors (WOS Xpad)
- Include tests and benchmarking tools as part of the library
- Better testing.

10.3 0.11.0: 07/2015

- All calibrant from NIST are now available, + Nickel, Aluminum, ... with bibliographic references
- The Cell class helps defining new calibrants.
- OpenCL Bitonic sort (to be integrated into Bragg/Amorphous separation)
- Calib is available from the Python interface (procedural API), not only from the shell script.
- **Many new options in calib for reset/assign/delete/validate/validate2/chiplot.**
  - reset: set the detector, orthogonal, centered and at 10cm

  - assign: checks the assignment of groups of points to rings
  - delete: remove a group of peaks
  - validate: autocorrelation of images: error on the center
  - validate2: autocorrelation of patterns at 180° apart: error on the center function of chi
  - chiplot: assesses the quality of control points of one/multiple rings.
- Fix the regression of the initial guess in calib (Thanks Jon Wright)
- New peak picking algorithm named "watershed" and based on inverse watershed for ridge recognition
- start factorizing cython regridding engines (work ongoing)
- Add "–poni" option for pyFAI-calib (Thanks Vadim Dyakin)
- Improved "guess_binning", especially for Perkin Elmer flat panel detectors.
- Support for non planar detectors like Curved Imaging plate developped at Aarhus
- Support for Multi-geometry experiments (tested)
- Speed improvement for detector initialization
- better isotropy in peak picking (add penalization term)
- enhanced documentation on http://pyfai.readthedocs.org

10.4 0.10.3: 03/2015

- Image segmentation based on inverse watershed (only for recalib, not for calib)
- Python3 compatibility
- include testimages into distribution

10.5 0.10.2: 11/2014

- Update documentation
- Packaging for debian 8

10.6 0.10.1: 10/2014

- Fix issue in peak-picking
- Improve doc & manpages
- Compatibility with PyMca5

10.7 0.10.0: 10/2014

- Correct Caglioti's formula
- Update tests and OpenCL -> works with Beignet and pocl open source drivers
- Compatibility with MacOSX and windows

10.8 0.9.4: 06/2014

- include spec of Maxwell GPU
- fix issues with intel OpenCL icd v4.4
- introduce shape & max_shape in detectors
- work on marchingsquares/sorted controurplot for calibration
- Enforce the use the Qt4Agg for Matplotlib and other GUI stuff.
- Update shape of detector in case of binning
- unified distortion class: merge OpenCL & OpenMP implementation #108
- Benchmarks for distortion
- Raise the level to warning when inverting the mask
- set of new ImXpad detectors Related issue #111
- Fix issue with recalib within MX-calibrate
- saving detector description in Nexus files issue #110
- Update some calibrants: gold
- about to make peak-picking more user-friendly
- test for bragg separation
- work on PEP8 compliance
- Do not re-cythonize: makes debian package generation able to benefit from ccache
- conversion to SPD (rotation is missing)
- pixelwise worker
- correct both LUT & OCL for memory error
- replace os.linsep with "n" when file file opened in text mode (not binary)
- rework the Extension part to be explicit instead of "black magic" :)
- implement Kahan summation in Cython (default still use Doubles: faster)
- Preprocessing kernel containing all cast to float kernels #120
- update setup for no-openmp option related to issue #127
- Add read-out mode for mar345 as "guess_binning" method for detector. Also for MAR and Rayonix #125
- tool to benchmark HDF5 writing
- try to be compatible with both PySide and PyQt4 ... the uic stuff is untested and probably buggy #130
- Deactivate the automatic saturation correction by default. now it is opt-in #131

10.9 0.9.3: 02/2014

- Better control for peak-picking (Contribution from Gero Flucke, Desy)
- Precise Rayonix detectors description thanks to Michael Blum
- Start integrating blob-detection algorithm for peak-picking: #70
- Switch fron OptParse to ArgPrse: #83
- Provide some calibrant by default: #91

- Description of Mar345 detector + mask#92
- Auto-registration of detectors: #97
- Recalib and check-calib can be called from calib: #99
- Fake diffraction image from calibrant: #101
- Implementation of the CSR matrix representation to replace LUT
- Tight pixel splitting: #43
- Update documentation

10.10 0.9.2: (01/2014)

- Fix memory leak in Cython part of the look-up table generation
- Benchmarks with memory profiling

10.11 0.9: 10/2013

- Add detector S140 from ImXpad, Titan from Agilent, Rayonix
- Fix issues: 61, 62, 68, 76, 81, 82, 85, 86, 87
- Enhancement in LImA plugins (better structure)
- IO module with Ascii/EDF/HDF5 writers
- Switch some GUI to pyQtGraph in addition to Qt
- Correction for solid-angle formula

10.12 0.8: 10/2012

- Detector object is member of the geometry
- Binning of the detector, propagation to the spline if needed
- Detector object know about their masks.
- Automatic mask for some detectors like Pilatus or XPad
- Implementation of sub-pixel position correction for Pilatus detectors
- LUT implementation in 1D & 2D (fully tested) both with OpenMP and with OpenCL
- Switch from C++/Cython OpenCL framework to PyOpenCL
- Port opencl code to both Windows 32/64 bits and MacOSX
- Add polarization corrections
- Use fast-CRC checksum on x86 using SSE4 (when available) to track array change on GPU buffers
- Support for flat 7*8 modules Xpad detectors.
- Benchmark with live graphics (still a memory issue with python2.6)
- Fat source distribution (python setup.py sdist –with-test-images) for debian
- Enhanced tests, especially for Saxs and OpenCL
- Recalibration tool for refining automatically parameters
- Enhancement of peak picking (much faster, recoded in pure Cython)

- Easy calibration for pixel detector (reconstruction of inter-module space)
- Error-bar generation using Poisson law
- Unified programming interface for all integration methods in 2theta, q or radius unit
- Graphical interface for azimuthal integration (pyFAI-integrate)
- Lots of test to prevent non regression
- Tool for merging images using various method (mean, median) and with outlayer rejection
- LImA plugin which can perform azimuthal integration live during the acquisition
- Distortion correction is available alone and as LImA plugin
- Recalibration can refine the wavelength in addition to 6 other parameters
- Calibration always done vs calibrant's ring number, lots of new calibrant are available
- Selection by hand of single peaks for calibration
- New detectors: Dexela and Perkin-Elmer flat panel
- Automatic refinement of multiple images at various geometries (for MX)
- Many improvements requested by ID11 and ID13

10.13 0.7.2: 08/2012

- Add diff_tomo script
- Geometry calculation optimized in (parallel) cython

10.14 0.7: 07/2012

Implementation of look-up table based integration and OpenCL version of it

10.15 0.6: 07/2012

- OpenCL flavor works well on GPU in double precision with device selection

10.16 0.5: 06/2012

- Include OpenCL version of azimuthal integration (based on histograms)

10.17 0.4: 06/2012

- Global clean up of the code regarding options from command line and better design
- Correct the orientation of the azimuthal angle chi
- Rename scripts in pyFAI-calib, pyFAI-saxs and pyFAI-waxs

10.18 0.3: 11/2011

- Azimuthal integration splits pixels like fit2d

10.19 0.2: 07/2011

- Azimuthal integration using cython histogramming is working

10.20 0.1: 05/2011

- Geometry is OK

CHAPTER

ELEVEN

LIST OF PUBLICATION ABOUT PYFAI

- *PyFAI, a versatile library for azimuthal regrouping*, J Kieffer & D Karkoulis; **Journal of Physics: Conference Series** (2013) 425 (20), pp202012
- *PyFAI: a Python library for high performance azimuthal integration on GPU* J Kieffer & J.P. Wright; **Powder Diffraction** (2013) 28 (S2), pp339-350
- ***PyFAI: a Python library for high performance azimuthal integration on GPU*** Jérôme Kieffer, Giannis Ashiotis **PROC. OF THE 7th EUR. CONF. ON PYTHON IN SCIENCE** (EUROSCIPY 2014)
- *The fast azimuthal integration Python library: pyFAI* J. Kieffer, G. Ashiotis, A. Deschildre, Z. Nawaz, J. P. Wright, D. Karkoulis, F. E. Picca **Journal of Applied Crystallography** (2015) 48 (2), 510-519

The later publication provides a nice overview of the features introduced in version 0.11 and further developed in v0.12 and v0.13 and should be the one cited in publications using pyFAI (or one of the graphical application on top of it). There are already 72 publication referring to pyFAI, some of them in the most prestigious scientific journals (Nature, PNAS, ...).

CHAPTER

TWELVE

BIBLIOGRAPHY

CHAPTER

THIRTEEN

INDICES AND TABLES

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

[SRI2012] PyFAI, a versatile library for azimuthal regrouping J. Kieffer & D. Karkoulis J. Phys.: Conf. Ser. 425 202012 http://dx.doi.org/10.1088/1742-6596/425/20/202012

[EPDIC13] PyFAI: a Python library for high performance azimuthal integration on GPU J. Kieffer & J. P. Wright, Powder Diffraction / Volume 28 / Supplement S2 / September 2013, pp S339-S350 http://dx.doi.org/10.1017/S0885715613000924

[FIT2D] Hammersley A. P., Svensson S. O., Hanfland M., Fitch A. N. and Hausermann D. 1996 High Press. Res. vol14 p235–248

[SPD] Bösecke P. 2007 J. Appl. Cryst. vol40 s423–s427

[EDNA] Incardona M. F., Bourenkov G. P., Levik K., Pieritz R. A., Popov A. N. and Svensson O. 2009 J. Synchrotron Rad. vol16 p872–879

[PyMca] Solé V. A., Papillon E., Cotte M., Walter P. and Susini J. 2007 Spectrochim. Acta Part B vol vol62 p63 – 68

[PyNX] Favre-Nicolin V., Coraux J., Richard M. I. and Renevier H. 2011 J. Appl. Cryst. vol44 p635–640

[IPython] Pérez F and Granger B E 2007 Comput. Sci. Eng. vol9 p21–29 URL http://ipython.org

[NumPy] Oliphant T E 2007 Comput. Sci. Eng. vol9 p10–20

[Cython] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn D and Smith K 2011 Comput. Sci. Eng. vol13 p31 –39

[OpenCL] Khronos OpenCL Working Group 2010 The OpenCL Specification, version 1.1 URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[FabIO] Sorensen H O, Knudsen E, Wright J, Kieffer J et al. 2007–2013 FabIO: I/O library for images produced by 2D X-ray detectors URL http://fable.sf.net/

[Matplotlib] Hunter J D 2007 Comput. Sci. Eng. vol9 p90–95 ISSN 1521-9615

[SciPy] Jones E, Oliphant T, Peterson P et al. 2001– SciPy: Open source scientific tools for Python URL http://www.scipy.org/

[FFTw] Frigo M and Johnson S G 2005 Proceedings of the IEEE 93 p 216–231

[LImA] The LIMA Project Update S. Petitdemange, L. Claustre, A. Homs, R. Homs Regojo, E. Papillon Proceedings of ICALEPCS2013 http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/html/auth1084.htm

[PyOpenCL] PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, Parallel Computing Vol 38, 3, March 2012, Pages 157–174 http://dx.doi.org/10.1016/j.parco.2011.09.001

[AMD] The American Mineralogist Crystal Structure Database. Downs, R.T. and Hall-Wallace, M. (2003) American Mineralogist 88, 247-250 http://rruff.geo.arizona.edu/AMS/amcsd.php

[COD] Crystallography Open Database: an open-access collection of crystal structures and platform for world-wide collaboration Saulius Grazulis et al. Nucl. Acids Res. (2012) 40 (D1): D420-D427. http://dx.doi.org/10.1093/nar/gkr900 http://www.crystallography.net/

[Dpdak] A customizable software for fast reduction and analysis of large X-ray scattering data sets: applications of the new DPDAK package to small angle X-ray scattering and grazing-incidence small angle X-ray scattering, Benecke, G. et al., (2014) J. Appl. Cryst. 47, http://dx.doi.org/10.1107/S1600576714019773

p