The Fast Azimuthal Integration Python library

Aurore Deschildre,[a] Giannis Ashiotis,[a] Zubair Nawaz,[b] Jonathan P. Wright,[a] Dimitrios Karkoulis,[a] Frédéric-Emmanuel Picca[c] and Jérôme Kieffer [a]*

[a]*European Synchrotron Radiation Facility, 71 Avenue des Martyrs, Grenoble, France.*

E-mail: jerome.kieffer@esrf,fr

Abstract

PyFAI is an open source software package designed to perform azimuthal and radial integration and, correspondingly, 2D regrouping on area detector frames for small and wide angle X-ray scattering experiments. It is written in Python, a language widely accepted and used by the scientific community today, which enables the users to easily incorporate the PyFAI library into their processing pipeline. This contribution focuses on recent work, especially the ease of calibration, its accuracy and the execution speed for integration.

1. Introduction

Azimuthal integration allows the usage of area detectors for recording powder diffraction patterns, which ensure larger solid angle coverage and hence a better harvest of X-ray photons. This data reduction step is often one of the most time consuming tasks in the processing pipeline and sometimes limits the productivity of modern synchrotron beamlines, where diffraction is used to probe samples with a point focussed beam in 2D raster scans or diffraction tomography experiments and very fast detectors.

This contribution describes the Python library pyFAI in its version 0.10 (released in october 2014) which can be used to calibrate the experimental setup of a powder diffraction experiment or a SAXS experiment using area detector using Debye-Scherrer rings collected from a reference compound. After describing how the geometry is internally represented in pyFAI, the various image analysis algorithm used to extract Debye-Scherrer rings are presented. Those peaks positions are combined with the knowledge of a calibrant and the wavelength of the radiation to perform the refinement of the detector position in space.

Once this geometry is known, azimuthal regrouping can be performed. As pyFAI implements various algorithm for integration, including multiple pixel splitting schemes, they will be exposed and compared for speed, precision and memory consumption. A scientific example is given on how pyFAI can be used to decompose diffraction images into amorphous and Bragg components and its application to serial crystallography.

As pyFAI is a library, and many users prefer an integrated graphical user interfaces (GUI), some other project related to pyFAI are described before concluding.

Appendices contains information about the project structure, tutorials on calibration and precision on the many-core implementation of azimuthal integration using OpenCL (Stone *et al.*, 2010).

2. Experiment description

In pyFAI, a basic experiment is defined by an area-detector whose position in space is defined from the sample position and the incident radiation.

2.1. Detector

Like most other diffraction processing packages, pyFAI allows the definition of 2D detectors by a constant pixel size (in meter) but this approach shows its limits with various types of detectors, especially multi-module detectors and fiber optic taper coupled detectors. Large area pixel detector are often composed from the assembly of smaller modules (Pilatus from Dectris, Maxipix from ESRF, . . .). By construction, such detector exhibit gaps between modules and pixels of various sizes within a single module, hence they require a specific mask. Optically coupled detectors should also be corrected for small spatial displacement of the pixel, often called geometric distortion.

2.1.1. Detectors classes are used to define a family of detectors. To take the specificities of each detector into account, pyFAI contains about 40 detectors class definitions which contain the mask (invalid pixels, gaps, . . .) and a method to calculate the pixel position in cartesian coordinates. For optically coupled CCD detectors, the geometrical distortion is often described by a bi-dimensional cubic spline which can be imported into the detector instance and used to calculate the actual pixel position in space.

2.1.2. Nexus Detectors: Any detector object in pyFAI, can be saved to a HDF5 file following the NeXus (Comittee, 2003) convention. Detector objects can subsequently be restored from the disk, making complex detector definition less error-prone. Pixels of an area detector are saved as a 4-dimensional dataset: 2D array of vertices pointing to every corner of each pixel, which makes an array of shape: (Ny, Nx, Nc, 3) where Nx and Ny are the dimension of the detector, Nc the number of corners of each pixel, usually 4, and the last dimension contains the vertex itself. This definition, while relying on large description file, is the only which can cope for complex detectors layout when the sensitive surface is no longer planar, for example curved imaging plates, tiled modules pixel detectors from ImXpad or the semi-cylindrical Pilatus12M from Dectris can be described.

2.2. Geometry

The experiment geometry is defined in pyFAI 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's main axis. The detector being a rigid body, its position in space is described by six parameters: 3 coordinates and 3 rotations. In pyFAI, we use the point, orthogonal projection of origin on the detector surface (Z=0 in detector's coordinate system), called PONI (for Point Of Normal Incidence, (Bösecke, 2007)). The detector distance is the sample-PONI distance and the PONI coordinates are measured in the detector's referential (origin at the lower left of the image). As pixel size could be non constant, all 3 distances are given in meter. The 3 rotations, stored in radians, correspond to the rotation along the 3 axes. When all rotations are zero, the detector is in transmission mode with the incident beam orthogonal to the detector's surface. The choice of S.I. units may look unadapted or odd to users familiar to other code like FIT2D (Hammersley *et al.*, 1996), therefore the geometry used in pyFAI can be exported to and imported from the parameter sets from other software. We are happy to integrate geometries used in other software to ease the comparison of results and cross-validate approaches.

2.2.1. Binning One of the strength of this geometry is capability to perform binning of the detector without having to re-calibrate or re-calculate the position in space. All pyFAI detector classes have a binning option which will increase the pixel size accordingly and divide detector shape accordingly. This even works for detectors with distortions (as the spline can be evaluated on various grid size).

3. Calibration

Calibration of the position of the detector is performed using Debye-Scherrer rings collected from a reference powder called "calibrant". Rings are extracted (see 3.2.1) and control points are located as maxima of those rings. The geometry of the experiment is obtained from a least-squares refinement of the 2θ angles. In this contribution we will call them "ring" even if, for planar detector, they are actually conics formed by the intersection of the Debye-Scherrer cones with the detector plan. PyFAI does not assume rings are circles, ellipses or parabolas and is able to fit the geometry of a wide range of experiments. The support for the geometry refinement of non planar detectors is still under development.

3.1. Calibrant

PyFAI provides ten calibrant descriptions among the most used ones: ceria, corundum, gold, lanthanum hexaboride and silicon for powder diffraction; silver behenate, tetradecanol and para-bromobenzoic acid for small angle scattering. Any file containing d-spacing in angstrom can be used as calibrant for a geometry calibration and can be loaded by the *Calibrant* class instance. This *calibrant* object is in charge of calculating the reference 2θ cone aperture against which the geometry will be refined (provided the wavelength/energy is known).

3.2. Peak-picking

With micro and nano-focused beams available on modern synchrotron facilities (Riekel *et al.*, 2010), fewer crystals get hit by the beam going through the sample making Debye-Scherrer ring obtained from the diffraction of reference powders spotty. As grinding reference powder is not advised (it would at least broaden peaks and may introduce strain), we decided to address this issue by image analysis and reconstruction of Debye-Scherrer rings. An alternative approach is to use single crystal indexation techniques for example using the Fable software (Oddershede *et al.*, 2010) as demonstrated for diffraction tomography experiment (Bonnin *et al.*, 2014).

3.2.1. "Massif" extraction allows a clear separation between regions containing large photon counts (rings) and the background. It uses a difference of the image with itself, Gaussian blurred with a given width, σ. Border of the regions with high intensity ($massif$) are negative in this difference image, so positive regions are labeled and represent (a fraction of) a ring. Peaks, which are local maxima, are sampled within the same region and belong to the same ring. The width of the Gaussian, in pixel units, has to be larger than the typical distance between two peaks within a ring and smaller than the distance between two rings. PyFAI includes some heuristics to guess an acceptable parameter in most cases but they can be overridden by the command line argument –gaussian=XX, where XX is the size of the gaussian in per-mille of the image size.

3.2.2. Sub-pixel precision on the peak position is obtained using a second order development of the intensity on the neighborhood of the peak position $\overrightarrow{x_0}$:

$$I(\overrightarrow{x}) = I(\overrightarrow{x_0}) + \nabla I(\overrightarrow{x_0}) \cdot (\overrightarrow{x} - \overrightarrow{x_0}) + \frac{1}{2}(\overrightarrow{x} - \overrightarrow{x_0})^T \cdot \mathcal{H}I(\overrightarrow{x_0}) \cdot (\overrightarrow{x} - \overrightarrow{x_0})$$

which can be derived into:

$$\nabla I(\overrightarrow{x}) = \nabla I(\overrightarrow{x_0}) + \mathcal{H}I(\overrightarrow{x_0}) \cdot (\overrightarrow{x} - \overrightarrow{x_0})$$

The position of the actual maximum $\overrightarrow{x}$ is define by $\nabla I(\overrightarrow{x}) = 0$, hence:

$$\overrightarrow{x} = \overrightarrow{x_0} - (\mathcal{H}I(\overrightarrow{x_0}))^{-1} \cdot \nabla I(\overrightarrow{x_0})$$

where I, ∇I and $\mathcal{H}I$ are the scalar field of intensity, its gradient (vector) and Hessian (matrix), respectively, measured at the maximum pixel position. Those derivatives are numerically assessed on a 3x3 neighborhood, so with noisy data, it happens that $\overrightarrow{x}$ is far away from $\overrightarrow{x_0}$ (more than a pixel). In such cases, $\overrightarrow{x}$ is taken as the center of mass of the 3x3 neighborhood around $\overrightarrow{x_0}$ (less precise, but more robust).

3.2.3. Blob detection is a computer vision method which allows to perform peak-picking without *a priori* knowledge on the level of the image. This feature is essential to us as diffraction images exhibit a very large dynamic range.

The diffraction image is sequentially blurred with Gaussian filters which width σ follows the geometric series: $\frac{1}{2}$, $\frac{\sqrt{2}}{2}$, 1, $\sqrt{2}$, 2, $2\sqrt{2}$, $\ldots$ For each blurred image at scale σ, the subsequent blurred (at $\sigma' = \sigma \cdot \sqrt{(}2)$ is subtracted to create a difference of Gaussian image (called *DoG*) which highlights the features of the image which typical size is σ. A 3D scale-space (x, y, σ) representation is created from those DoG images.

This method provides us not only the peaks location, as local maxima in scale-space, but also the typical size of the peak. Peak position, scales and intensity are refined as described in 3.2.2, extended to the 3D scale space.

To keep the computation time reasonable, the implementation of the blob detection relies on Gaussian convolution in real space (i.e. without Fourier transform), separated in horizontal and vertical direction, with small convolution kernels of width $8\sigma + 1$. To prevent the growth of the window with larger σ, a pyramid of Gaussians is built by binning the blurred image by a factor 2 when it reaches a $\sigma = 2$.

The drawback of this algorithm, beside the calculation time, is its very high sensitivity to noise in flat regions. This is why blob detection is only used in the re-calibration procedure to extract all peaks in a region of interest, determined form an approximative geometry. Moreover blob detection cannot find peaks wich width is smaller than $\sigma = 0.7$ which correspond to 3 pixels.

3.3. Graphical user interface for calibration

Only a minimalistic graphical user interface (called pyFAI-calib, figure 4) is provided for calibration, with visual assignment of the ring number. An rough estimate of the geometry is usually obtained from a mouse click on two of the inner-most rings. The pink and yellow dots correspond to the control points (peaks) extracted using the algorithm described in 3.2.1 (with only two mouse clicks). The refinement is performed on the error in 2θ (squared) using the Sequential Least SQuares Programming from SciPy (function scipy.optimize.min$_s lsqp).Afterrefinementofthegeometry,theiso-contourforrefined2\theta$ array is overlaid to the diffraction image. Those are the four thin plain lines drawn on the image to mark where Debye-Scherrer rings are expected, allowing a visual validation of the calibration.

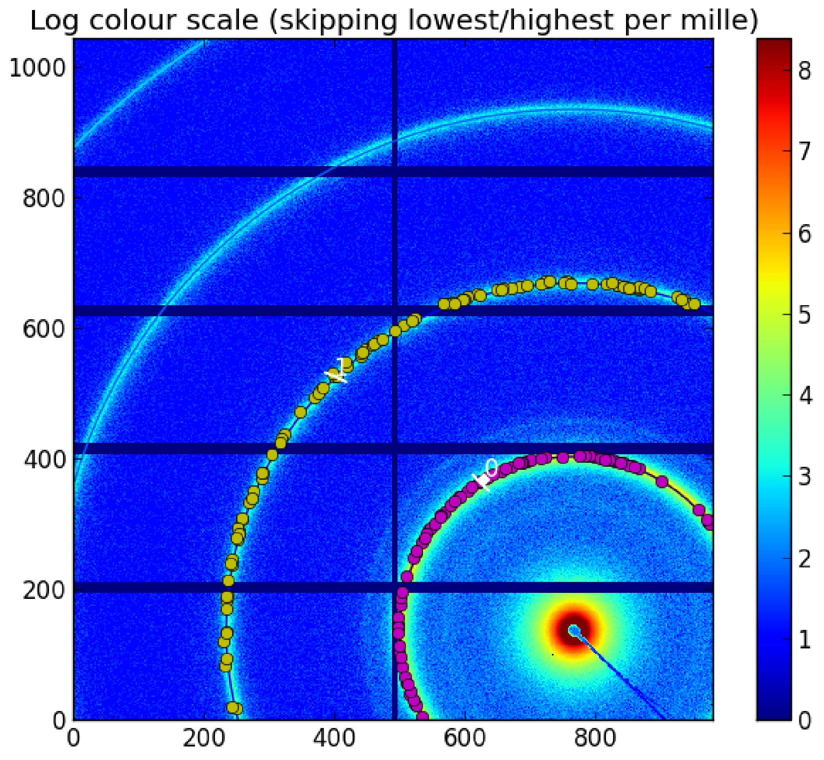

Fig. 1. The calibration window allows manual peak-picking and ring assignment. The data correspond to a silver behenate sample, used as a calibrant on the BioSAXS beamline BM29 at the synchrotron ESRF (detector: Pilatus 1M, $\lambda = 1.0$Å)

From this initial rough calibration, pyFAI allows to perform many operations in Command Line Interface (CLI) mode, like setting, constaining, fixing and refining parameters, extracting a new set of keypoints or performing the integration. The complete description of options is described in annex B.

4. Azimuthal Integration

The core of pyFAI is to be able to perform azimuthal integration in 1D or 2D as fast as possible, using Python binary extensions, often multi-threaded or even mani-core accelerated (i.e. Graphics Processors Units, GPU and Intel Xeon Phi accelerators), while offering a common interface at the Python level. Many details on the techniques used to speed up the code, especially porting it to GPU have been described in (Kieffer & Ashiotis, 2014).

4.1. Programming interface for azimuthal integration

The initial idea behind pyFAI is to provide a easy way to perform azimuthal integration, ideally in a single command. In the snippet of code we present how this is done:

```python
import pyFAI, fabio, pylab
img = fabio.open("imagefile.tif").data
ai = pyFAI.load("geometry.poni")
tth, I = ai.integrate1d(img, 1000, unit="2th_deg", method="splitpixel")
pylab.plot(tth, I)
pylab.show()
```

The fist line load three important libraries: fabio (Knudsen *et al.*, 2013) to read images, pylab (Hunter, 2007) to display the result and pyFAI. The second and third line load the image and the geometry. the two last lines are about displaying the curve.

In this snippet, the image *img* is azimuthaly integrated into 1000 bins with an out, eventhly spaced in 2θ. Other units like the scattering vector length q or the radius r are available. The method keyword is a switch to select the algorithm used.

4.2. Pixel splitting schemes and implementation

PyFAI implementents a dozen of azimuthal integration procedures which can be classified according to they way integration is performed and the pixel splitting scheme used.

4.2.1. Histogram vs Look-Up Table. The naive way to integrate data is to treat an image pixel after pixel, implemented like an histogram. This is a scatter operation which is hard to parallelize but cheap in memory. Using a scatter to gather transformation, the azimuthal integration for a given geometry can be stored into a look-up table (LUT) and applied like a sparse matrix, dense vector multiplication. While much more expensive in memory, this implementation is effective in parallelization and speed. The Compressed Row Storage (CSR) matrix representation is now used in place of the LUT and provides a lower memory footprint.

4.2.2. Three pixel splitting schemes are available in pyFAI and define the way photons counted by a pixel are assigned to the various histogram bins, especially when the pixels are large (Pilatus detectors):

- No splitting: the full intensity is assigned into a single bin (dirac like shape)
- Bounding box splitting: the pixel is abstracted by a simpler shape oriented parallel to the radial and azimuthal directions.
- Tight/full pixel splitting: the only assumption made is that pixel edges are lines.

The figure **??** presents the way a single pixel is split on a large number of bins using the three schemes exposed previously. The way Fit2D splits pixels has been added for comparison: it looks pretty similar to bounding box pixel splitting but the abstracted shape is likely to be smaller.

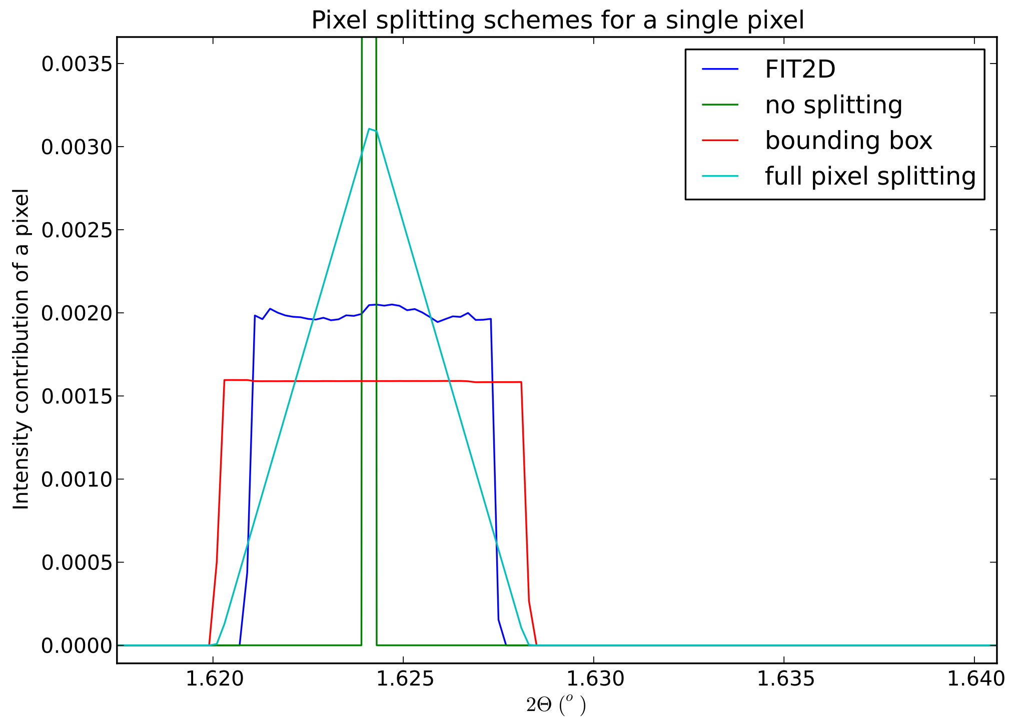

Fig. 2. Contribution to a powder diffraction pattern of a single pixel to highlight the pixel splitting algorithm underneath. PyFAI's implementations are compared to FIT2D.

4.2.3. Summary about speed and memory consumption. The table table:methods summarizes the various implementation available with their execution speed and the memory footprint for integrating a 2048x2048 pixels image into 1000 bins.

Table 1. *Various "methods" available within pyFAI for azimuthal integration featuring their speed and memory footprint. Measured on a 3 GHz quad-core computer with a 2048x2048 pixels image.*

Pixel split	No splitting	Bounding box	Tight pixel
Direct histogram	numpy (889 ms, 336 MB) cython (361 ms, 323 MB)	splitbbox (129 ms, 343 MB)	splitpixel(516 ms, 480 MB)
Look-up table	CSR nosplit (48 ms, 330 MB)	splitBBoxLUT (59 ms, 327 MB) CSR bbox (52 ms, 330 MB)	CSR full (51 ms, 502 MB)

It is worth mentioning that while pixel splitting provides smoother results, any pixel splitting scheme introduces some serial-correlation between neighboring bins, introducing an overestimation of errors, as described in (Yang X.and Juhas & Billinge, 2014).

4.3. Graphical user interface for azimuthal integration

TODO

5. Example of applications

Azimuthal regrouping and its opposite transformation (assuming uniform-distribution over the azimuthal angle) can be performed using pyFAI which offers many opportunities for processing.

5.1. Diffraction image generation

Once the geometry defined (i.e. by loading a poni-file), the 2θ and χ position of every single pixel of the detector are known. If one assumes the isotropy of signal (real powder without prefered orientation), $2D$ diffraction patterns can be generated as exposed in this example:

```python
import numpy, scipy.signal, pyFAI
N = 1000
# generate the powder curve as a single gaussian
tth = numpy.linspace(0, 60, N)
I = scipy.signal.gaussian(N, 5)

det = pyFAI.detectors.detector_factory("Pilatus1M")
ai = pyFAI.AzimuthalIntegrator(dist=0.1, poni1=0.1, poni2=0.1, detector=det)

img = ai.calcfrom1d(tth, I)
```

The method, calcfrom1d is available from any AzimuthalIntegrator or Geometry instance. It is used together with a calibrant object to generate a fake diffraction image suitable for testing pyFAI or other calibration codes (for example to validate the geometry translation from one program to another).

```
import pyFAI.calibrant
lab6 = pyFAI.calibrant.ALL_CALIBRANTS["LaB6"]
lab6.set_wavelength(1e-10)
img_lab6 = lab6.fake_calibration_image(ai)
```

In this snippet of code, a reference sample LaB_6 is defined on the second line from the known calibrants before the the wavelength is set. Combined with the geometry, this calibrant is able to generate a 2D numpy array containing the simulated Debye-Scherrer diffraction rings which can be saved or displayed on the screen. The fake_calibration_image contains further option to set U, V and W parameters from Caglioti's formula (Caglioti *et al.*, 1958) to inlude the broadening of peaks according to this simple resolution function. Calibrants in pyFAI contain only their d-spacing, so the reconstructed image will have all rings with the same intensity (once integrated).

5.2. Image offset and validation of the calibration

By regenerating a 2D diffraction image from the integrated powder pattern one can assess the quality of the calibration used for the integration. The calibration tool, pyFAI-calib, offers a "validate" command which measures the offset on the image (x, y) between the 2D diffraction image and the one regenerated from the integrated patern, using a phase correlation algorithm. This allows a measurement of the precision of localization of the PONI, which can be better than a tenth of a pixel, when calibrating images with continuous rings (i.e. not spotty) with a mask large enough to remove the beam stop and all parasitic scattering.

5.3. Amorphous background removal

PyFAI's azimuthal integrator features a *separate* method able to separate the background with an azimuthal symmetry (amorphous scattering or powder's ring), from Bragg peaks automatically.

Based on what was described in (Kieffer & Wright, 2013), a bidimensional azimuthal integration is performed on the input image. The output 2D image is filtered along the azimuthal χ axis using a percentile (often median) filter to reconstruct the powder diffraction curve without the sharp Bragg spots. The number of points in azimuthal and radial direction as well as the percentile value can be adjusted but the default values are reasonably good.

The reconstructed 2D image corresponds to the amorphous/powder/isotropic component of the input image and the subtraction of this generated image from the raw data contains only the signal coming from large crystals. Figure 3 (left) presents a close-up of protein single crystal data recorded on a Pilatus3-2M detectors (image taken at the ID23-2 beamline from ESRF). A diffuse amorphous halo is clearly visible. After using the automatic amorphous background removal, which takes into account the mask needed for such pixel detectors, only Bragg peaks remain (right of the image).

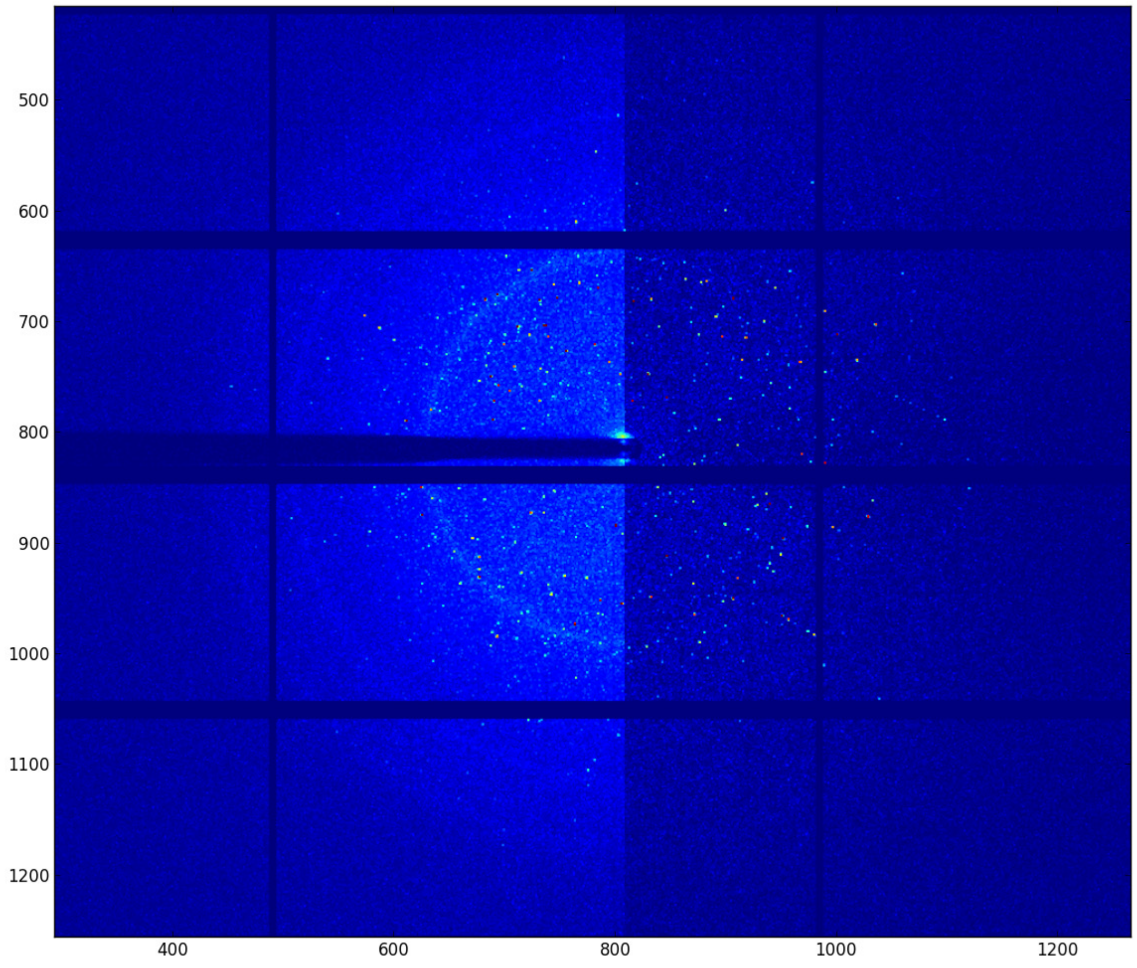

Fig. 3. Automatic removal of amorphous signal (ice ring) from Bragg peaks in a protein crystallography experiment (data from beamline ID23 at synchrotron ESRF).

5.3.1. Application to serial crystallography In these experiments, tiny crystals in their solvent are sent in front of the X-ray beam (using a jet or moving a motor) and data are acquired continuously, using a fast detector (from dozens of Hertz to kHz). Those experiments produce a huge amount of data and only a small fraction of the frames contain diffraction signal. PyFAI has been integrated into the processing software NanoPeakCell which provides a graphical interface for frame selection in serial crystallography. But pyFAI has also been integrated into the LImA data acquisition system (Homs *et al.*, 2012), where it assesses the amount of single crystal diffraction data within each frame and decides to save it or not. In this way, a huge amount of disk space and network bandwidth can be saved.

6. Related Work

As mentionned earlier, the pyFAI software library has been integrated into other pieces of code, mainly integrated at the different beamlines from synchrotron ESRF to perform azimuthal integration online, either using LImA or dedicated online data analysis server: EDNA(Incardona *et al.*, 2009) on the BioSaxs(Pernot *et al.*, 2013) and DAHU at beamline TRUSAXS (ID02).

Other institutes have independently integrated pyFAI into their processing pipeline: NanoPeakCell (developped at IBS by Nicolas Coquelle), PySAXS (developped at CEA by Olivier Taché), DpDak (developped at synchrotrons Petra III by Gunthard Benecke and co-workers) (Benecke *et al.*, 2014) and Dioptas (developped at synchrotron APS by Clemens Prescher). Most of those software packages propose a graphical interface to ease the data processing for a specific type of experiment.

7. Conclusion and Future Work

Graphical user interface for calibration

re-assignement of group of points to rings, especially useful for off-centered detectors where the ring number is not clear.

The support for the geometry refinement of non planar detectors is still under development.

References

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. & Smith, K. (2011). *Comput. Sci. Eng.* **13**(2), 31–39.

Benecke, G., Wagermaier, W., Li, C., Schwartzkopf, M., Flucke, G., Hoerth, R., Zizak, I., Burghammer, M., Metwalli, E., Müller-Buschbaum, P., Trebbin, M., Förster, S., Paris, O., Roth, S. V. & Fratzl, P. (2014). *Journal of Applied Crystallography,* **47**(5), 1797–1803.
URL: *http://dx.doi.org/10.1107/S1600576714019773*

Bonnin, A., Wright, J. P., Tucoulou, R. & Palancher, H. (2014). *Applied Physics Letters,* **105**(8), –.
URL: *http://scitation.aip.org/content/aip/journal/apl/105/8/10.1063/1.4894009*

Bösecke, P. (2007). *J. Appl. Cryst.* **40**(s1), s423–s427.
URL: *http://dx.doi.org/10.1107/S0021889807001100*

Caglioti, G., Paoletti, A. & Ricci, F. P. (1958). *Nuclear Instruments,* **3**, 223–228.

Comittee, N. I. A., (2003). A common data format for neutron, x-ray and muon science.
URL: *http://www.nexusformat.org/*

Hammersley, A. P., Svensson, S. O., Hanfland, M., Fitch, A. N. & Hausermann, D. (1996). *High Press. Res.* **14**(4-6), 235–248.
URL: *http://www.tandfonline.com/doi/abs/10.1080/08957959608201408*

Homs, A., Claustre, L., Kirov, A. & Papillon, E.and Petitdemange, S. (2012). In *Contributions to the Proceedings of ICALEPCS 2011*, pp. 676–679.

Hunter, J. D. (2007). *Comput. Sci. Eng.* **9**(3), 90–95.

Incardona, M.-F., Bourenkov, G., Levik, K., Pieritz, R., Popov, A. & Svensson, O. (2009). *J. Synchrotron Rad.* **16**(6), 872–879.

Jones, E., Oliphant, T. E. & Peterson, P., (2001). SciPy: Open source scientific tools for Python.
URL: *http://www.scipy.org/*

Kieffer, J. & Ashiotis, G. (2014). In *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*, edited by P. de Buyl & N. Varoquaux.

Kieffer, J. & Wright, J. (2013). *Powder Diffraction,* **28**, S339–S350.
URL: *http://journals.cambridge.org/article*$_S$0885715613000924

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. & Fasih, A. (2012). *Parallel Computing,* **38**(3), 157–174.

Knudsen, E. B., Sørensen, H. O., Wright, J. P., Goret, G. & Kieffer, J. (2013). *J. Appl. Cryst.* **46**, 537–539.

Oddershede, J., Schmidt, S., Poulsen, H. F., Sørensen, H. O., Wright, J. & Reimers, W. (2010). *Journal of Applied Crystallography,* **43**(3), 539–549.
URL: *http://dx.doi.org/10.1107/S0021889810012963*

Oliphant, T. E. (2007). *Comput. Sci. Eng.* **9**(3), 10–20.

Pernot, P., Round, A., Barrett, R., De Maria Antolinos, A., Gobbo, A., Gordon, E., Huet, J., Kieffer, J., Lentini, M., Mattenet, M., Morawe, C., Mueller-Dieckmann, C., Ohlsson, S., Schmid, W., Surr, J., Theveneau, P., Zerrad, L. & McSweeney, S. (2013). *Journal of Synchrotron Radiation,* **20**(4), 660–664.
URL: *http://dx.doi.org/10.1107/S0909049513010431*

Riekel, C., Burghammer, M. & Davies, R. (2010). *IOP Conference Series: Materials Science and Engineering,* **14**(1), 012013.
URL: *http://stacks.iop.org/1757-899X/14/i=1/a=012013*

Stone, J. E., Gohara, D. & Shi, G. (2010). *Computing in Science and Engineering,* **12**(3), 66–73.

Yang X.and Juhas, P. & Billinge, S. J. L. (2014). *J. Appl. Cryst.* **47**, 1273–1283.

Appendix A

Project structure

PyFAI is an open source project licensed under the GPL mainly written in Python (v2.6 or 2.7) and heavily relying on the python scientific ecosystem: NumPy(Oliphant, 2007), SciPy (Jones *et al.*, 2001) and Matplotlib (Hunter, 2007). It provides high performances image treatment and azimuthal integration thanks to Cython (Behnel *et al.*, 2011) and PyOpenCL (Klöckner *et al.*, 2012). PyOpenCL remains an optional dependency, so all OpenCL code has a Python or Cython implementation.

The project is hosted on GitHub (https://github.com/kif/pyFAI) which provides the issue tracker in addition to code hosting. To ease the distribution, the software is available as official Debian package and included in some of the most famous Linux distribution like Ubuntu and Debian. The program is also tested on other operating systems like Windows and MacOSX.

Anybody can fork the project and adapt it to their own needs: CEA-saclay, Synchrotrons Soleil, Desy and APS did. Collaboration is encouraged and new developments can be submitted and merged into the main branch via pull-requests.

While there are a couple of official releases each year (better tested versions), continuous integration is used to ensure any snapshot of the master branch provides correct results.

Appendix B
Calibration tool

The calibration tool of pyFAI, called pyFAI-calid has been available along with the integration tool since the very beginning of the project as a core idea is to use the same code to calculate the geometry and to perform the integration. The geometry file (also called PONI-file) is updated at each refinement step and contains the whole description of the experiment (together with time-stamps), this avoids copy-paste errors of coordinates.

B.1. Calibration user interface

The user interface for calibration has improved a lot recently: the assignment of the ring number is now possible graphically, as shown in figure 4. An rough estimate of the geometry is usually obtained from a mouse click on two of the innermost rings. The pink and yellow dots correspond to the control points extracted using the algorithm described in 3.2.1 (two mouse clicks). The refinement is performed on the error in 2θ (squared) using the Sequential Least SQuares Programming from SciPy (function scipy.optimize.min$_s lsqp).Afterrefinementofthegeometry, theiso -$ $contourforrefined2\theta$ array is overlaid to the diffraction image. Those are the four thin plain lines drawn on the image to mark where Debye-Scherrer rings are expected, allowing a visual validation of the calibration.

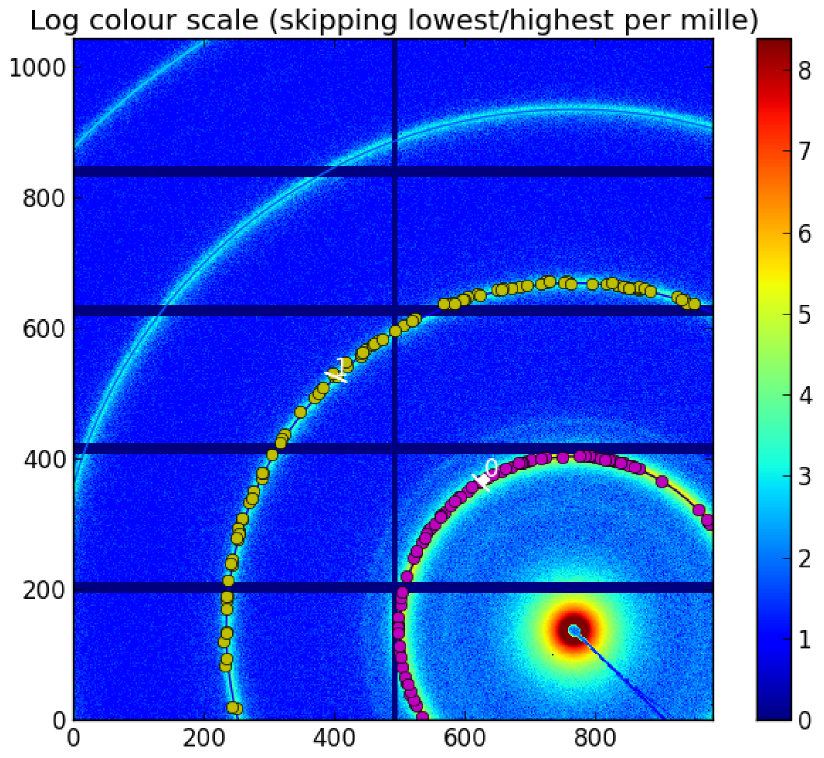

Fig. 4. The calibration window allows manual peak-picking and ring assignment. The data correspond to a silver behenate sample, used as a calibrant on the BioSAXS beamline BM29 at the synchrotron ESRF (detector: Pilatus 1M, $\lambda = 1.0$Å)

From an initial rough calibration, pyFAI allows to perform many operation to refine all parameters: distance, center position, rotation of the detector and optionally the wavelength (all units are S.I). The available commands are:

- *help*: show the help message
- *abort*: quit the program directly
- *done*: perform the azimuthal integration and quit
- *get*: print out the value of a parameter get wavelength
- *set*: define the value of a parameter, set wavelength 1.54e-10
- *bound*: select the region of validity for a parameter, bound dist 0.1 0.5

- *bounds*: review and modify all region of validity for all parameters
- fix: prevent the parameter to be refined fix wavelength
- $free$: allow the parameter to be refined free rot1
- $refine$: re-runs the least squares refinement
- *validate*: estimate the precision of the calibration on the whole image by overlaying the raw image with the integrated pattern retro-projected
- *recalib n*: extract a new set of control points from the n
- *reset*: set the geometry to its default values (centered orthogonal detector)
- *show*: print out to the screen the current geometry parameters
- *integrate*: perform the 1D and 2D integration and display it in a separate window to validate the quality of the calibration.

If the initial calibration is correct, like in 4, the procedure to get a perfect calibration should be "*recalib*, *done*".

If the theoretical rings are becoming too large or too small compared to the actual ones, the wavelength may be refined. For this extract as many rings as reasonable using *"recalib n"* then *"free wavelength"* and *"refine"* to check if it is better. Distance and wavelength are heavily correlated, it is important to use as many rings as possible at high angle.

The validate option allows us to measure the offset between the actual diffraction image and an image generated from the refined geometry using phase correlation. In most cases it is possible to obtain a precision better the 0.1 pixel in all directions.

B.2. Automatic distance and center calibration

The previous procedure has been automated for Macromolecular Crystallography beamlines at the synchrotron ESRF (MX), as they need to input the sample distance together with the beam center automatically in the header of their collected images to be able to process them automatically. As the area detector is on a moving stage, distance and center position are changing from data-collection to data-collection. The MX-calibrate tool from pyFAI allows the automatic calibration of a set of images taken at various distances using Debye-Scherrer rings of a reference compound. After performing subsequently peak-picking based on blob-detection (as described in 3.2.3) and least square refinement of the geometry on every input frame, distances and beam-center position are returned as function of the detector motor position without human intervention.

Appendix C
Parallel implementations using OpenCL

Azimuthal integration, as many computing intensive parts in pyFAI were written in OpenCL kernels and interfaced using PyOpenCL (Klöckner *et al.*, 2012). PyOpenCL offers a shared execution model effective both on processors (CPU), graphics cards (GPU) and accelerators like the Intel's Xeon Phi.

C.1. Azimuthal Integration

The direct azimuthal integration is basically a scatter operation which requires large memory locked section. To overcome this limitation, pixels have been associated to the output bin of the histogram and stored in a look-up table (LUT), making the integration look like a simple (if large and sparse) matrix vector product. The sparse matrix "Compressed Row Storage" (CSR) format is now used in pyFAI, saving about half of the space of the LUT previously used. Secondly, all threads within a workgroup are collaborating to calculate the matrix-vector product via a so-called "parallel reduction", offering additional speed-up (especially on GPUs). The compensated algebra (Kahan summation) is kept to maintain the precision of the calculation while using single precision (32 bits) floating points arithmetic.

C.2. Performances

The figure 5 shows the performance of pyFAI in frames processed per second depending on the input image size (in semi-logarithmic scale). This benchmark has been performed on a dual Intel Xeon E5-2667 (2.9Ghz) computer with a Nvidia Tesla K20 GPU and an Intel Xeon Phi accelerator.

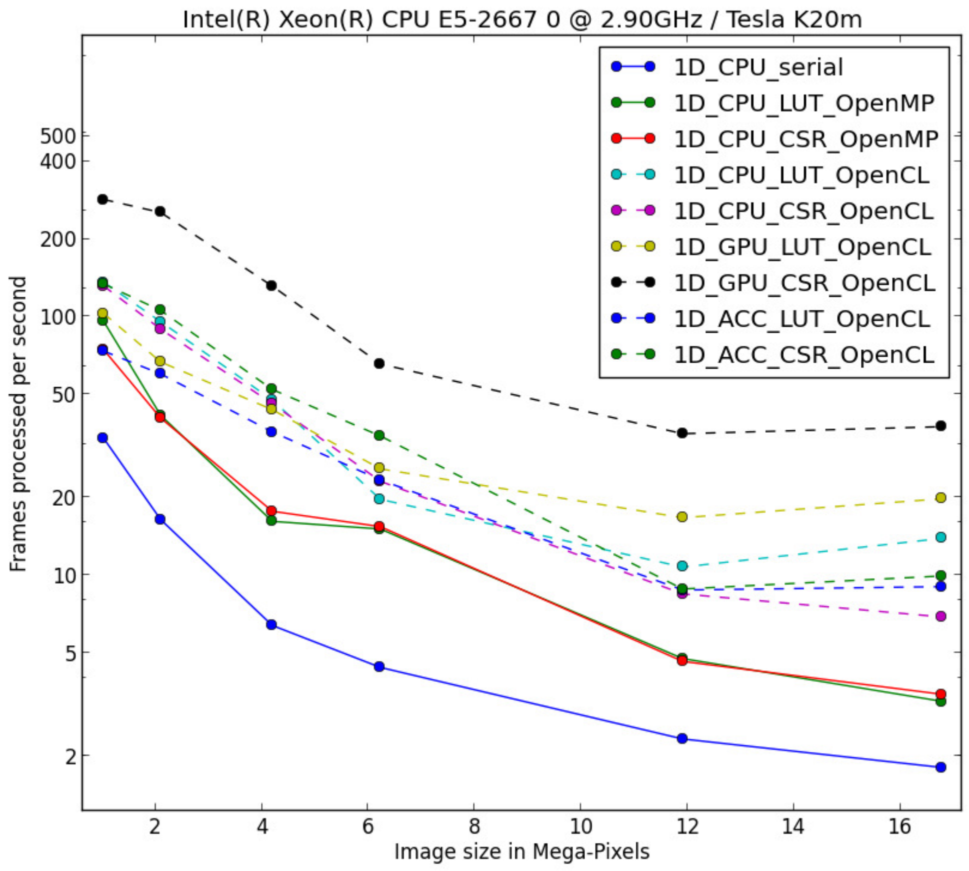

Fig. 5. Benchmark of the various algorithm to perform azimuthal integration in pyFAI

In this benchmark, four group of curves can be extracted.

- The lower plain blue curve presenting the serial Cython code using histograms (corresponding to the "splitbbox" method) which is the slowest implementation (even if it is 7x faster than a numpy implementation).
- The red and blue plain curves which correspond to the two parallel Cython implementation for look-up tables integration.
- the group of dashed curves which represent the OpenCL optimized code running on the 12 CPU cores, the 60 cores from the accelerator or the GPU (LUT implementation).
- The upper curve, in dashed black correspond the CSR sparse matrix multiplication implemented in OpenCL code and running on the Tesla card which is twice faster than any other implementation. This is obtained from the collaborative reduction from all threads within a workgroup.

Acknowledgements

The authors would like to thank all ESRF beamline teams for supporting the pyFAI development, especially BM01, ID02, ID11, ID13, ID15, ID23, BM26, ID29, BM29, ID30. In the instrumentation division (ISDD) we would like to thank Claudio Ferrero, head of data analysis unit, and Andy Götz, head of software group, for supporting the algorithmic work performed on pyFAI in addition to the features seen by the user. V. Armando Solé, the author of PyMca, is also acknowledged for providing us some important graphical user interface building blocks.

The huge parallelization work on the integration routines and their porting to many-core devices was mainly done by Dimitris Karkoulis, Zubair Nawaz and Giannis Ashiotis, thanks the EU-grant LinkSCEEM-2 (RI-261600).

We would like to thank David Flot for the protein diffraction image with ice-rings (fig. 3) and Martha Brennich for the silver behenate calibration image (fig. 4).

Synopsis

Details about the geometry, peak-picking, calibration and integration procedures on multi- and many-core devices implemented in the Python library for high performance azimuthal integration.