ESRF | The European Synchrotron

ESRF USER ORGANISATION

ESRF | The European Synchrotron

Welcome to the T5: Data reduction for scattering experiments

ESRF user meeting 2023

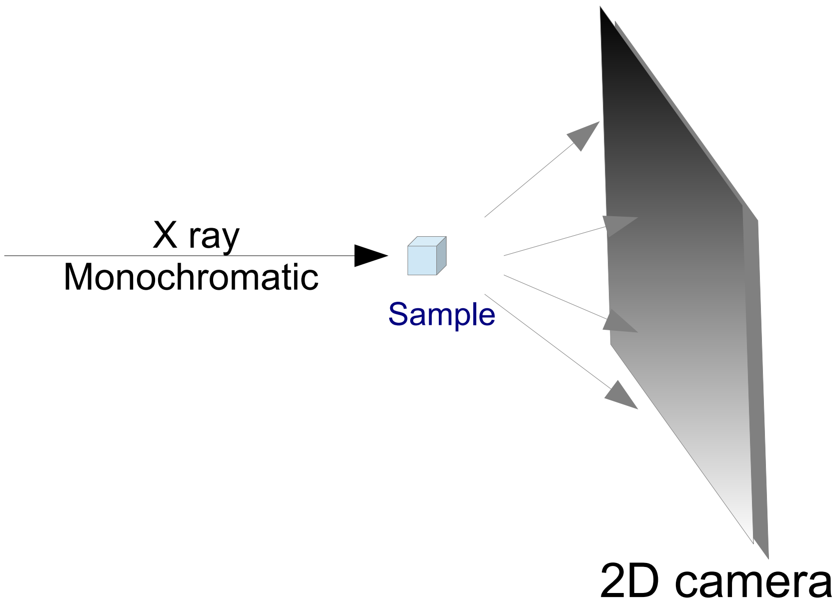
Data reduction tools for scattering experiments

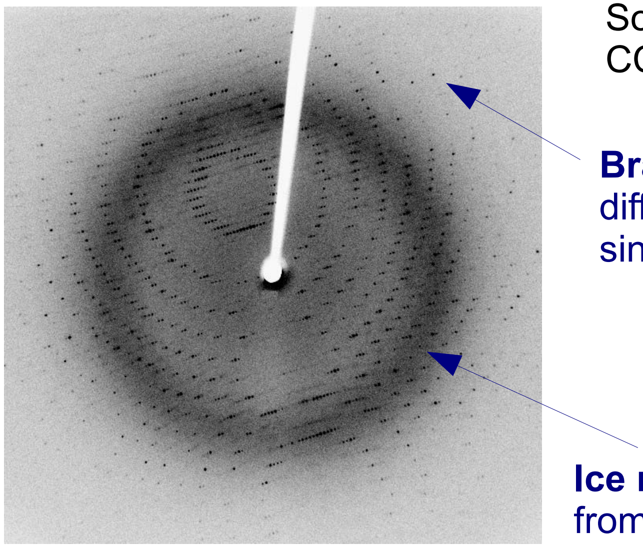
Jérôme Kieffer
Algorithms & scientific Data Analysis

Layout

- **Power diffraction and scattering of X-Rays**
- **What is azimuthal integration of 2D detector data ?**
- **The need for faster data processing …**
- **… without compromising quality**
- **PyFAI:**
  - Ecosystem and user community
  - Within the *silx* collaboration
- **Conclusions**

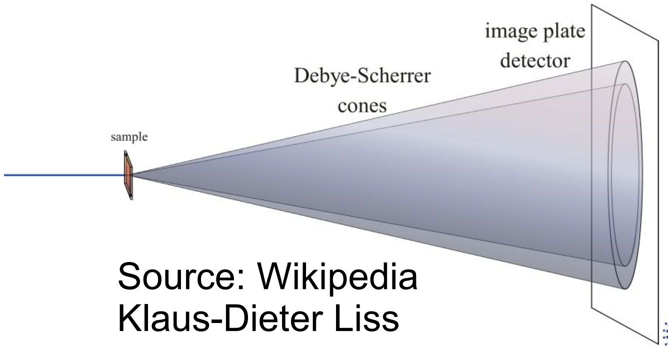
X-ray scattering experiments





- **Light is reflected as on mirrors:**
  - No energy change (elastic scattering)
  - Direction of diffracted beam depends on the crystalline cell and its orientation
  - Intensity of the diffracted beam depends on the the content of the cell

    → Bragg's Nobel price in 1915 $n\lambda = 2d\sin\theta\,,$
- **Multiple small crystals (powder)**
  - Isotropic cones gives ellipses
    when intersected by a flat detector



Powder diffraction and small angle scattering

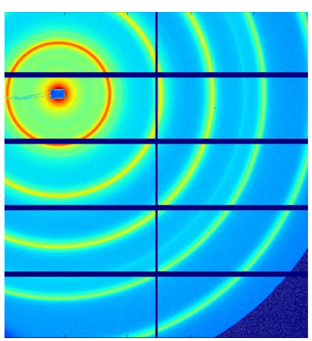
Application of powder diffraction:

- Phase identification (mapping)
- Crystallinity
- Lattice parameters
- Thermal expansion
- Phase transition
- Crystal structure
- Strain and crystallite size

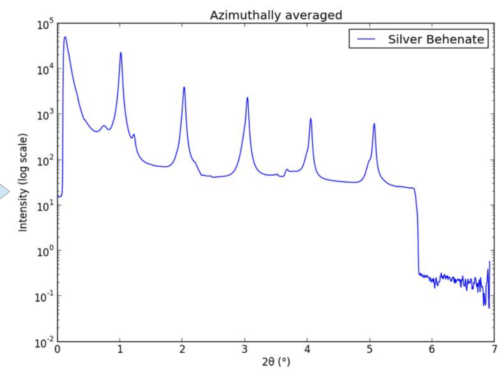
Application of small angle scattering

- Micro/nano-scale structure
- Particle shape
- Protein domains
- Protein folding
- Colloids
- Fiber orientation

- **Both rely on the same transformation: 2D image → azimuthal average**

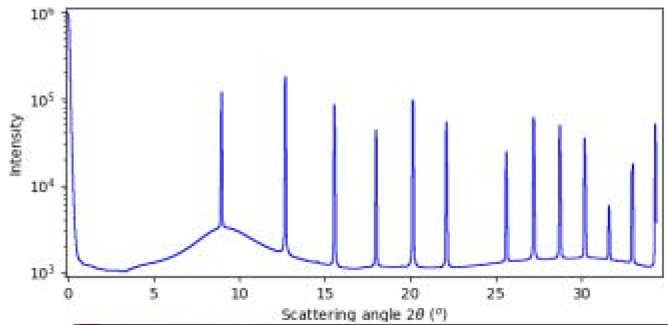


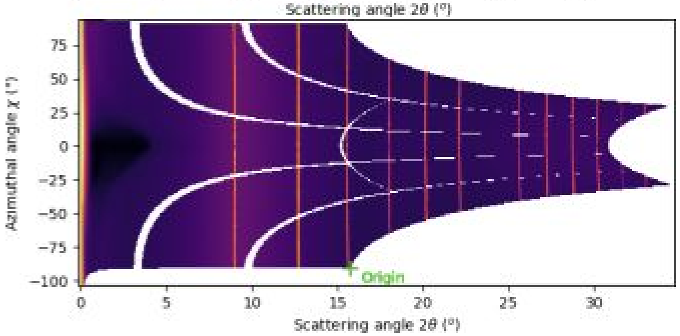
Azimuthal integration

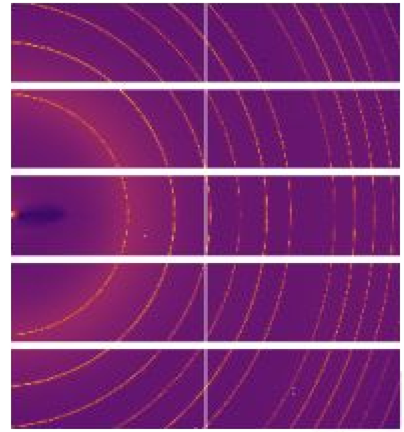


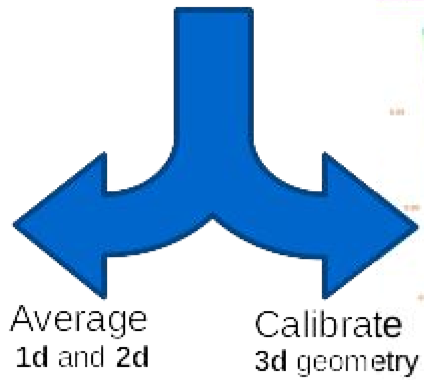
Fast Azimuthal Integration using Python

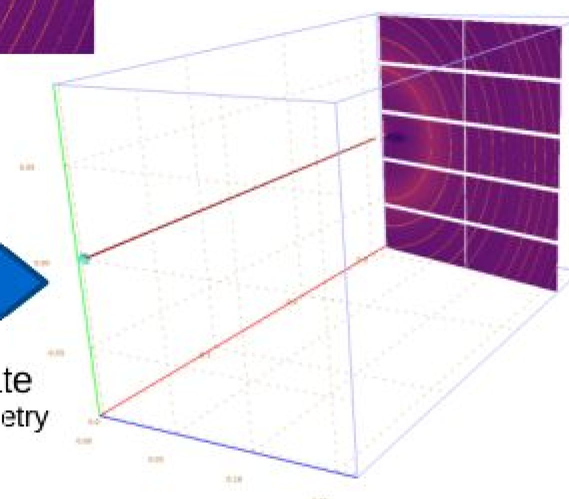
PyFAI
Fast Azimuthal Integration











- **Why Python ?**
  - It is the main programming language used in science and at ESRF: Bliss, PyMca, …
- **But isn’t Python slow ?**
  - Maybe … Python is just a convenient interface, what matters is written in C & compiled

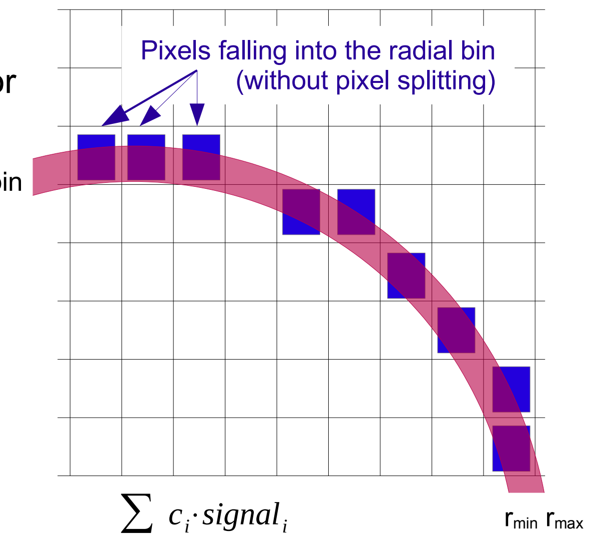
How it works

- **Pixel-wise corrections:**

$$I_{cor} = \frac{I_{raw} - I_{dark}}{F \cdot \Omega \cdot P \cdot A \cdot I_0} = \frac{signal}{normalization}$$

Where: I_0 is the incoming flux (pixel independent)

  - I_{raw} and I_{dark} are the signal measured from the detector
  - F is the flat-field correction
  - Ω is the solid angle for this pixel
  - P is the polarization factor
  - A is the parallax correction factor



- **Averaging over a bin defined by the radius *r*:**
  - Need for pixel splitting?
  - c_i being the fraction of the pixel i contributing to bin_r

$$\langle I \rangle_r = \frac{\sum_{i \in bin_r} c_i \cdot signal_i}{\sum_{i \in bin_r} c_i \cdot normalization_i}$$

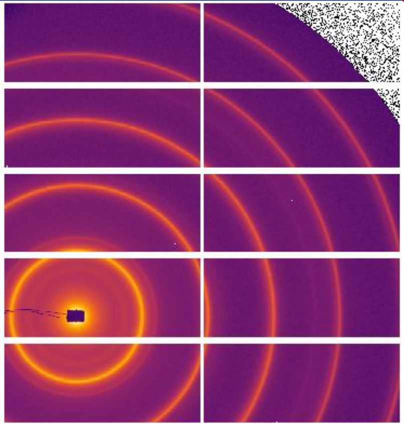
- **Associated uncertainty propagation:**
  - Assuming there is no correlation between pixels
  - Pixel splitting can create correlation between bins...

$$\sigma(I_r) = \sqrt{\frac{\sum_{i \in bin_r} c_i^2 \cdot variance_i}{\sum_{i \in bin_r} c_i^2 \cdot normalization_i^2}}$$

$$\sigma(\langle I \rangle_r) = \frac{\sqrt{\sum_{i \in bin_r} c_i^2 \cdot variance_i}}{\sum_{i \in bin_r} c_i \cdot normalization_i}$$

Math from Kieffer et al.; *J. Synch. Radiation* (2020)
https://doi.org/10.1107/S1600577520000776

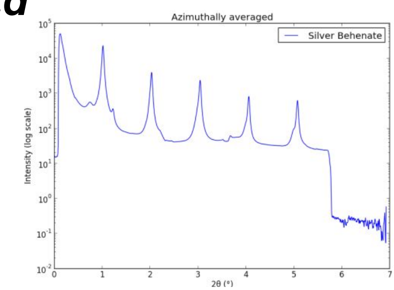
Concepts in PyFAI



- **Image**

  2D array of pixels as *numpy* array
  read using *silx, fabio, h5py*, ...

- **Azimuthal integrator: core object**
  - **powder diagram using *integrate1d***
  - **“cake” image, azimuthally regrouped using *integrate2d***



- **Detector**

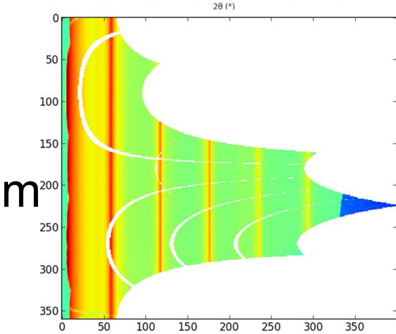
  - **Calculates the pixel position (center and corners)**
  - **Calculates and stores the mask of invalid pixels.**

  → saved as a HDF5 file

- **Geometry**

  Position of the detector from the sample & incoming beam

  → saved as *PONI*-file



http://www.silx.org/doc/pyFAI/dev/geometry.html#detector-position

Geometry in pyFAI

Parameters:

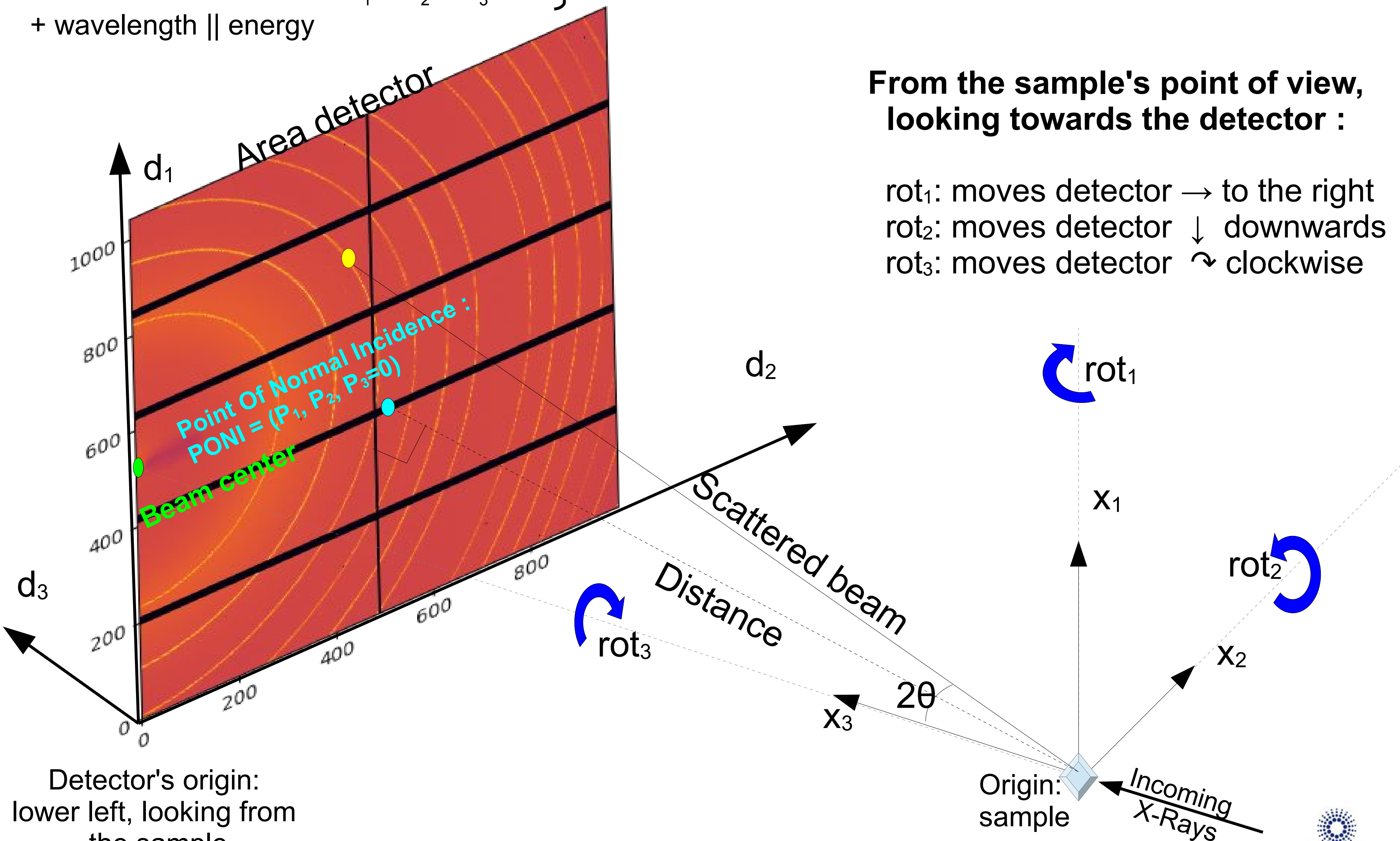
* 3 distances in meters: dist, $poni_1$, $poni_2$
* 3 rotations in radians: rot_1, rot_2, rot_3

} *PONI*-file

+ wavelength || energy

From the sample's point of view, looking towards the detector :

rot_1: moves detector → to the right
rot_2: moves detector ↓ downwards
rot_3: moves detector ↷ clockwise



Calibration in pyFAI

- **Geometry is best determined from the analysis of a known reference sample**
- **This calibration step is preferred to measuring distances and beam center position**
  - The prerequisite is:
    - **detector geometry and mask,**
    - **calibrant (LaB_6, CeO_2, AgBh, …)**
    - **wavelength or energy used**
  - Only the position of the detector and the rotation needs to be refined:
    - **3 translations: dist, $poni_1$ and $poni_2$**
    - **3 rotations: rot_1, rot_2, ~~rot_3~~**
- **It is divided into 4 major steps:**
  1) Extraction of groups of peaks
  2) Identification of peaks and groups of peaks belonging to same ring
  3) Least-squares refinement of the geometry parameters on peak position
  4) Validation by a human being of the geometry
- **PyFAI assumes this setup does not change during the experiment**

Tutorial 1:

http://www.silx.org/doc/pyFAI/dev/usage/cookbook/calib-gui/index.html

What happens during an integration

1) **Get the pixel coordinates from the detector, in meter.**

   There are 3 coordinates per pixel corner, and usually 4 corners per pixel.

   1Mpix image → 48 Mbyte !

2) **Offset the detector's origin to the PONI and rotate around the sample**

3) **Calculate the radial (2θ) and azimuthal (χ) positions of each corner**

4) **Assign each pixel to one or multiple bins.**

   If a look-up table is used, just store the fraction of the pixel.

   Then for each bin sum the content of all contributing pixels.

5) **Histogram bin position with associated intensities**

6) **Histogram bin position with associated normalizations (i.e. solid angle)**

7) **Return bin position and the ratio of sum of intensities / sum of norm.**

→ Tutorial 2

Example of simplified implementation in Python

Common initialization step:

In [1]:
```python
import numpy
npt = 1024
y,x  = numpy.ogrid[-512:512,-512:512]
radius = (x*x+y*y)**0.5
rmax = radius.max()+0.1
data = numpy.random.random((1024,1024))
```

Naive approach integration using corona extraction with masks:

In [2]:
```python
%%time
res_loop = numpy.zeros(npt)
for i in range(npt):
    rinf = rmax * i / npt
    rsup = rinf + rmax / npt
    mask = numpy.logical_and((rinf <= radius),(radius < rsup))
    res_loop[i] = data[mask].mean()
```

```
CPU times: user 1.04 s, sys: 0 ns, total: 1.04 s
Wall time: 1.04 s
```

Vectorized version using histograms:

In [3]:
```python
%%time
count_of_pixels = numpy.histogram(radius, npt, range=[0,rmax] )[0]
sum_of_intensities = numpy.histogram(radius, npt, weights=data, range=[0,rmax])[0]
res_vec = sum_of_intensities / count_of_pixels
```

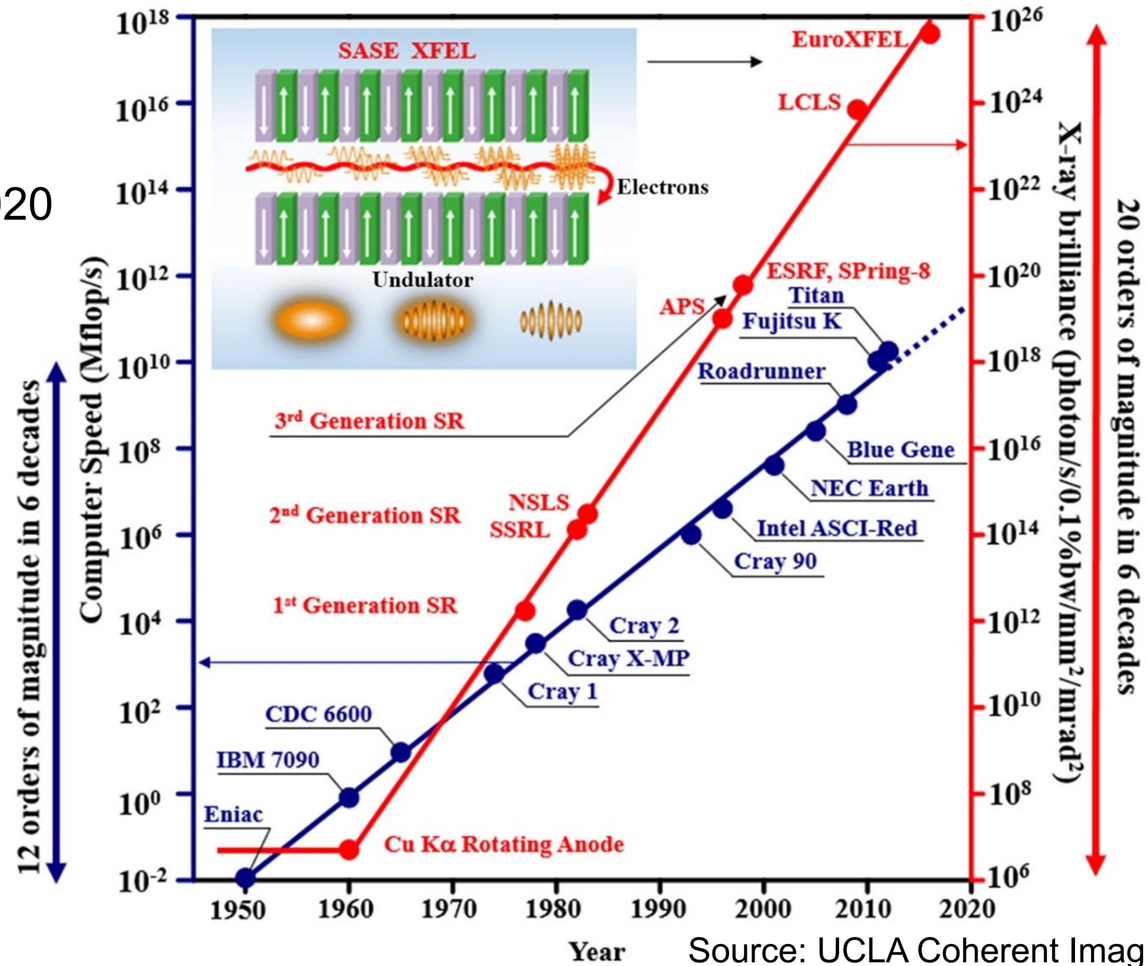
```
CPU times: user 19.5 ms, sys: 1.44 ms, total: 20.9 ms
Wall time: 19.4 ms
```

In [4]:
```python
# Speed-up: 50x, validation:
numpy.allclose(res_loop, res_vec)
```

Out[4]: True

But speed does matters ...

- **New EBS source**
  - 50x brighter
  - User mode since 2020



Source: UCLA Coherent Imaging

- **Faster detectors**
  - Eiger2 detector (2-20 kHz)
  - Jungfrau detector (2 kHz)

→ Stream limited to 2 GB/s/detector !

The gap between computing and acquisition grows

- **Most other codes use the same algorithm based on histograms … … and reach the same speed:**
  - Fit2D written in Fortran
  - SPD written in C
  - Foxtrot written in Java

- **The algorithm needs to be changed !**
  - Histograms **cannot** easily/efficiently be parallelized !
  - Re-develop based on parallel algorithms
    → CSR sparse matrix dot product is many-core friendly
    Described in https://arxiv.org/abs/1412.6367v1 (2014)
  - Several other program copied this idea:
    - **Saxsdog https://arxiv.org/abs/2007.02022 (2020),**
    - **MatFRAIA https://doi.org/10.1107/S1600577522008232 (2022)**

Look-up table integration using only Python

Using a Sparse matrix multiplication

Those multiplication can take advantage of parallel hardware unlike histogram which require costly *atomic* operations. This trick is called "scatter to gather" transformation in parallel programming.

In [5]:
```
%%time
from scipy.sparse import csc_matrix
positions = numpy.histogram(radius, npt, range=[0,rmax] )[1]
row = numpy.digitize(radius.ravel(), positions) - 1
size = row.size
col = numpy.arange(size)
dat = numpy.ones(size, dtype=float)
csr = csc_matrix((dat, (row, col)), shape = (npt, data.size))
print(csr.shape)
```

```
(1024, 1048576)
CPU times: user 60.5 ms, sys: 6.21 ms, total: 66.7 ms
Wall time: 69.7 ms
```

In [6]:
```
%%time
count_csr = csr.dot(numpy.ones(data.size))
sum_csr = csr.dot(data.ravel())
res_csr = sum_csr / count_csr
```

```
CPU times: user 3.11 ms, sys: 3.1 ms, total: 6.21 ms
Wall time: 4.69 ms
```

In [7]:
```
# Speed-up: 5x vs histograms, validation:
numpy.allclose(res_csr, res_vec)
```

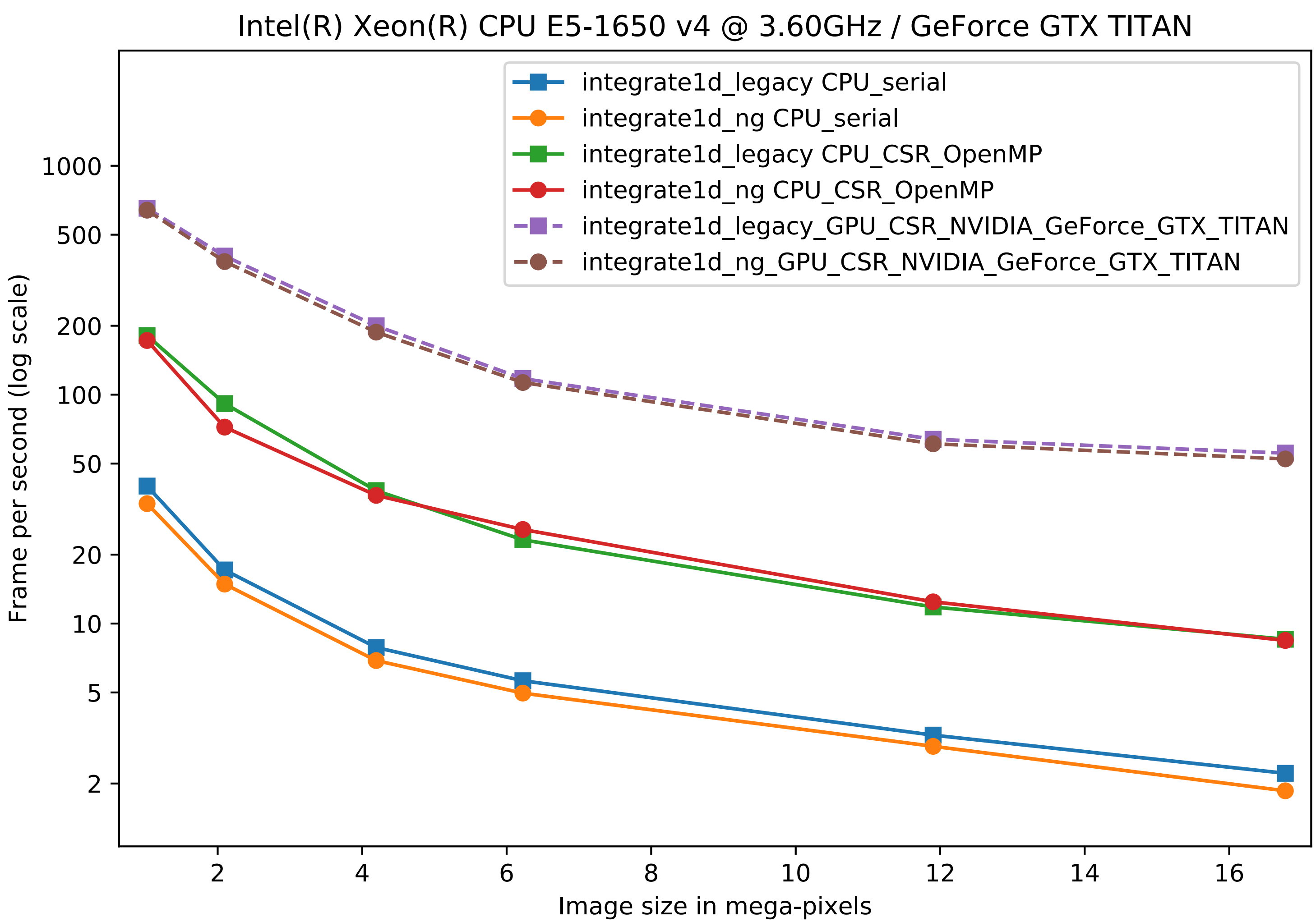
Out[7]: True

Sources of this demo available on:
https://gist.github.com/kif/ab37c61351d8238f90245b0afb56192e

Advantages of *histograms* vs *CSR* matrix multiplication

	Histograms	Sparse matrix multiplication
Pro	• **Easier to understand** • **Low memory consumption** • **Fast initialization**	• **Faster, even on a single core** • **Many-core friendly** – OpenMP and OpenCL
Con	• **Pretty slow** • **Hardly parallelizable**	• **Slower initialization** • **The sparse matrix can be large**
Rule of thumb:	< 5 frames	≥ 5 frames

Benchmark: Let's speak about speed !



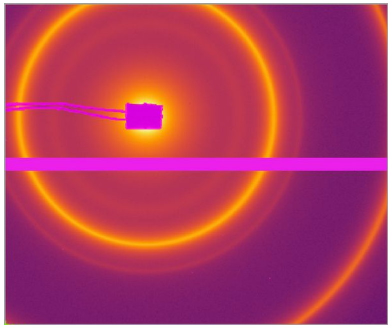
6-year-old workstation: CPU from 2016, GPU from 2013

Moiré effect

High frequency noise issue

Where pixel splitting comes back

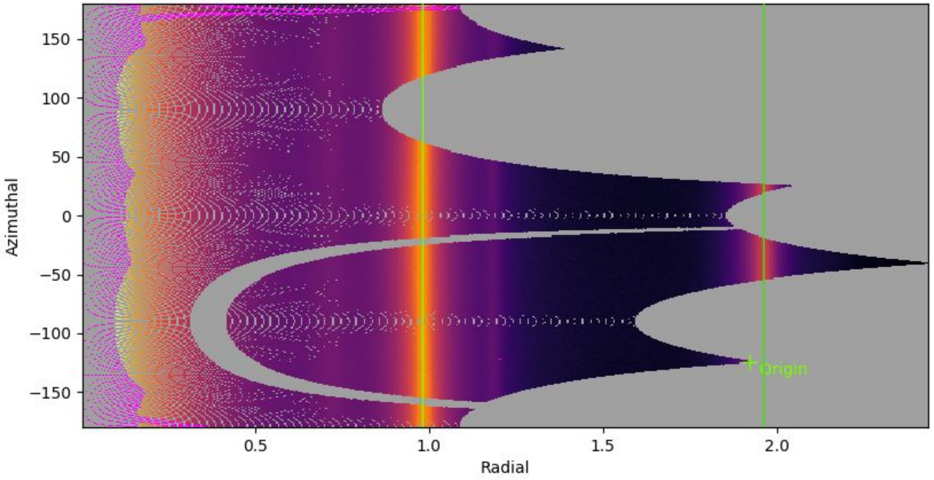
Example with SAXS data integrated in 2D



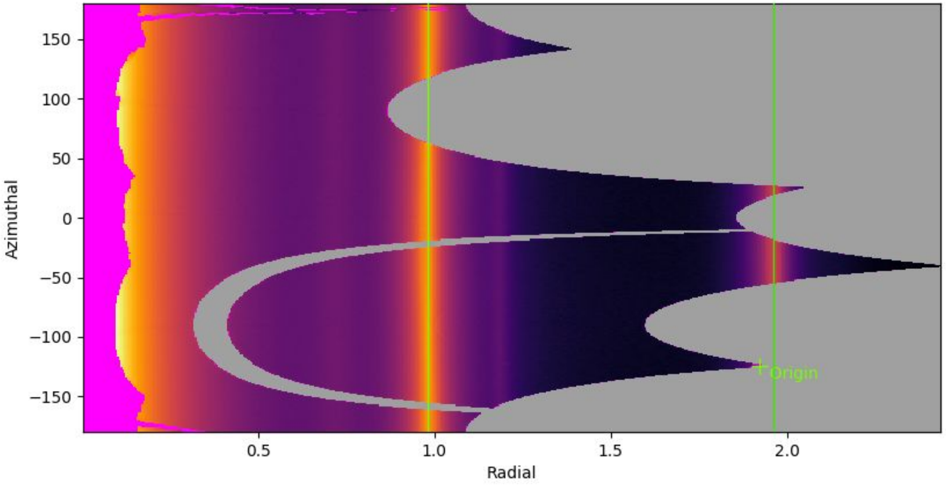
Pilatus 200k:
~500 x 400 pixels

2D averaging over 512x360 bins

Without pixel splitting



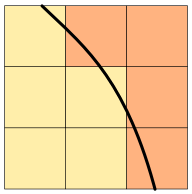
With pixel splitting



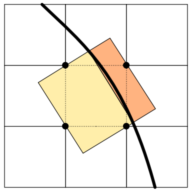
creates bin cross-correlation

Pixel splitting schemes available in pyFAI

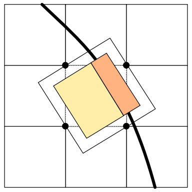
- **No pixel splitting: default histograms**
  - Each pixel contributes to a single bin of the result
  - No bin correlation but noisy
  - The pixel has no surface: sharpest peaks



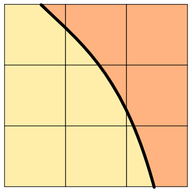
- **Bounding-box pixel splitting**
  - The smoothest integrated curve
  - Blurs a bit the signal



- **Pseudo pixel splitting (deprecated)**
  - Scale down the bounding box to the pixel area, before splitting.
  - Good cost/precision compromise, similar to FIT2D



- **Full pixel splitting**
  - Split each pixel as a polygon on the output bins.
  - Costly high-precision choice



Impact of pixel splitting on integration speed

- **Histogram based algorithms:**
  - Each pixel is split over the bins it covers.
  - The corner coordinates have to be calculated (4x slower initialization)
  - The slow down is function of the oversampling factor, for every image

- **Sparse matrix multiplication based algorithms**
  - The sparse matrix contains already the pixel splitting scheme
  - Longer initialization time related to the oversampling factor
  - There are *NO* performance penalty on the integration itself

All those consideration are independent of the programming language

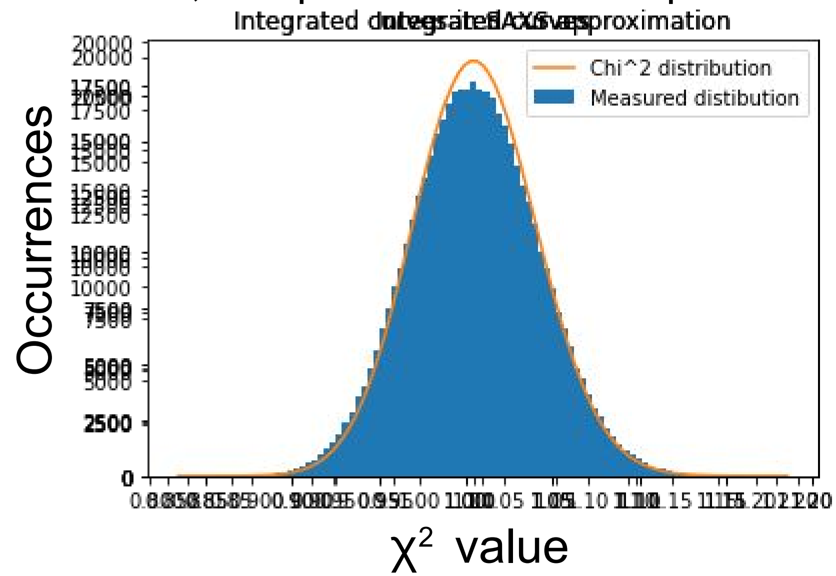
Nevertheless, Python which is interpreted is expected to be 1000x slower than:

- compiled code like C, C++, Fortran, ...
- JIT compiled code like Java, Julia or numba

Impact of averaging & pixel splitting on precision

- **Test case:**
  - SAXS-like data, 1000 frames with synthetic distribution, 5e5 pairs of curves compared.

- **No splitting / No intensity correction**
- **No splitting / intensity correction prior int.**
- **No splitting / intensity correction while int.**
- **BBox splitting / intensity correction while…**
- **Full splitting / intensity correction while…**



- **This demonstrates that:**
  - Intensity correction needs to be performed together with integration, not before!
  - Pixel splitting
    - **Actually creates bin-correlation**
    - **Affects precision of the propagated uncertainties**
- **Full demonstration at: http://www.silx.org/doc/pyFAI/0.20.0/usage/tutorial/Variance/Variance.html**

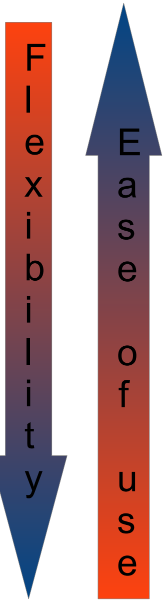
Layers in pyFAI

- **Applications level:**
  - GUI applications: **pyFAI-calib2, pyFAI-integrate, diff_map**
  - Scriptable applications:**pyFAI-average**, pyFAI-saxs, pyFAI-waxs, diff_tomo, …

- **Python interface:**

  Ideally used from jupyter

  - Top level: azimuthal integrator
  - Mid level: calibrant, detector, geometry, calibration
  - Low level: rebinning/histogramming engines (Cython + OpenMP or OpenCL)



- **Question: how to define the right balance ?**

  It is up to you !

- Applications in **bold** will be demonstrated in the introduction tutorial.

LATEST NEWS FROM PYFAI

- **High speed *sigma-clipping***
  - Enforce normal distribution in every azimuthal bin :
    - Remove single crystal contribution from powder diffraction
    - Several error models: *poissonian, azimuthal, hybrid*
  - Enables:
    - Single crystal frame compression (10x-200x, lossy compression)
    - Peak-finding: 250 Hz / GPU
  - Sponsored by serial crystallography (ESRF ID29, MX)
- **Square out all integration engines:**
  - Any type of integration: 1d (averaging) and 2d (caking)
  - Any type of pixel-splitting: without, bounding-box or full splitting
  - Any type of algorithm: histogram or sparse matrix multiplication
  - Any type of implementation: Python, Cython (C++) and OpenCL (GPU)

Project management

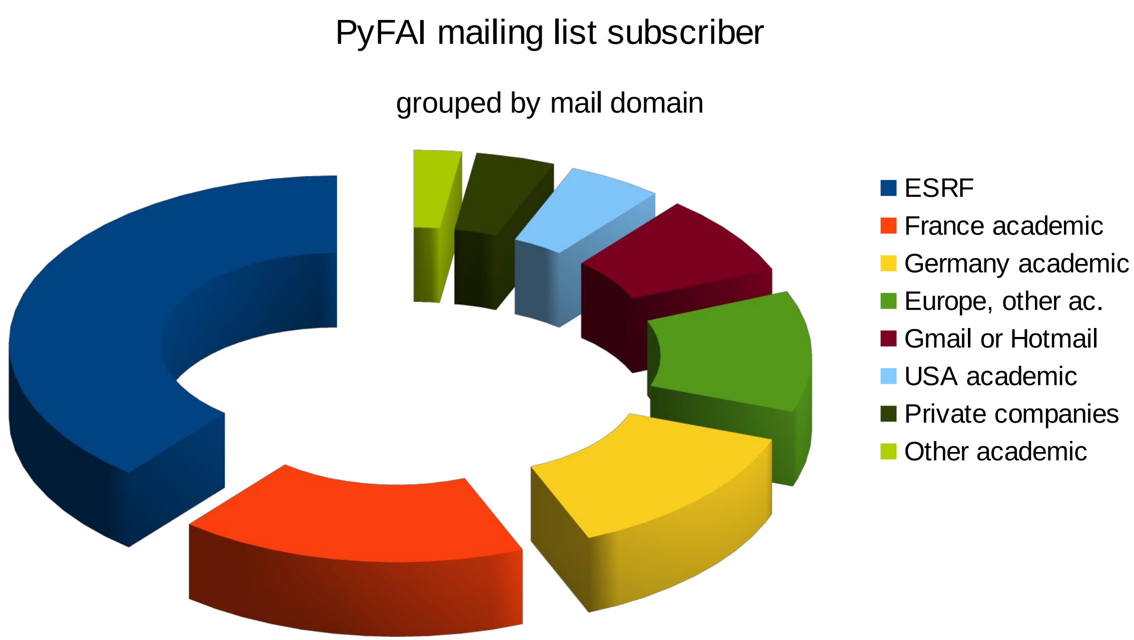
Silx & pyFAI

PyFAI is yet another azimuthal integration tool

- **Written in Python (compatible with ~~2.7, 3.6~~, 3.7, 3.8, 3.9, 3.10, 3.11)**
  - Free, fast and portable: (Windows, MacOS, Linux)
  - MIT licensed: compatible with both science & business
  - Part of the *silx* collaboration on data analysis initiated by ESRF
  - Graphical user interface using Qt5
- **Open to collaboration**
  - About 20 direct contributors,
    - **Mainly from ESRF**
    - **Also from other synchrotrons and XFELs:**
      - Soleil, NSLS-II, Petra-III, Eu-XFEL
    - **Industrial contributions from Xenocs**
  - Used by ~90 other projects from all the largest X-ray sources in the world
    - EuXFEL, SLAC, ALS, APS, NSLS-II, Petra-III, Soleil, Diamond, SLS, Max-IV, …
- **Avoid compromises:**
  - no difficulty is hidden

User community of pyFAI

- **PyFAI is used in most European and American synchrotons/FELs**



- **User support is provided via the mailing list: pyFAI@esrf.fr**
  - 166 people subscribed to the list 2023 (154, 142, 137, 132, 112)
  - limited activity (~1 thread/month)
- **Bugs are discussed via the Github issue tracker:**
  - https://github.com/silx-kit/pyFAI/issues

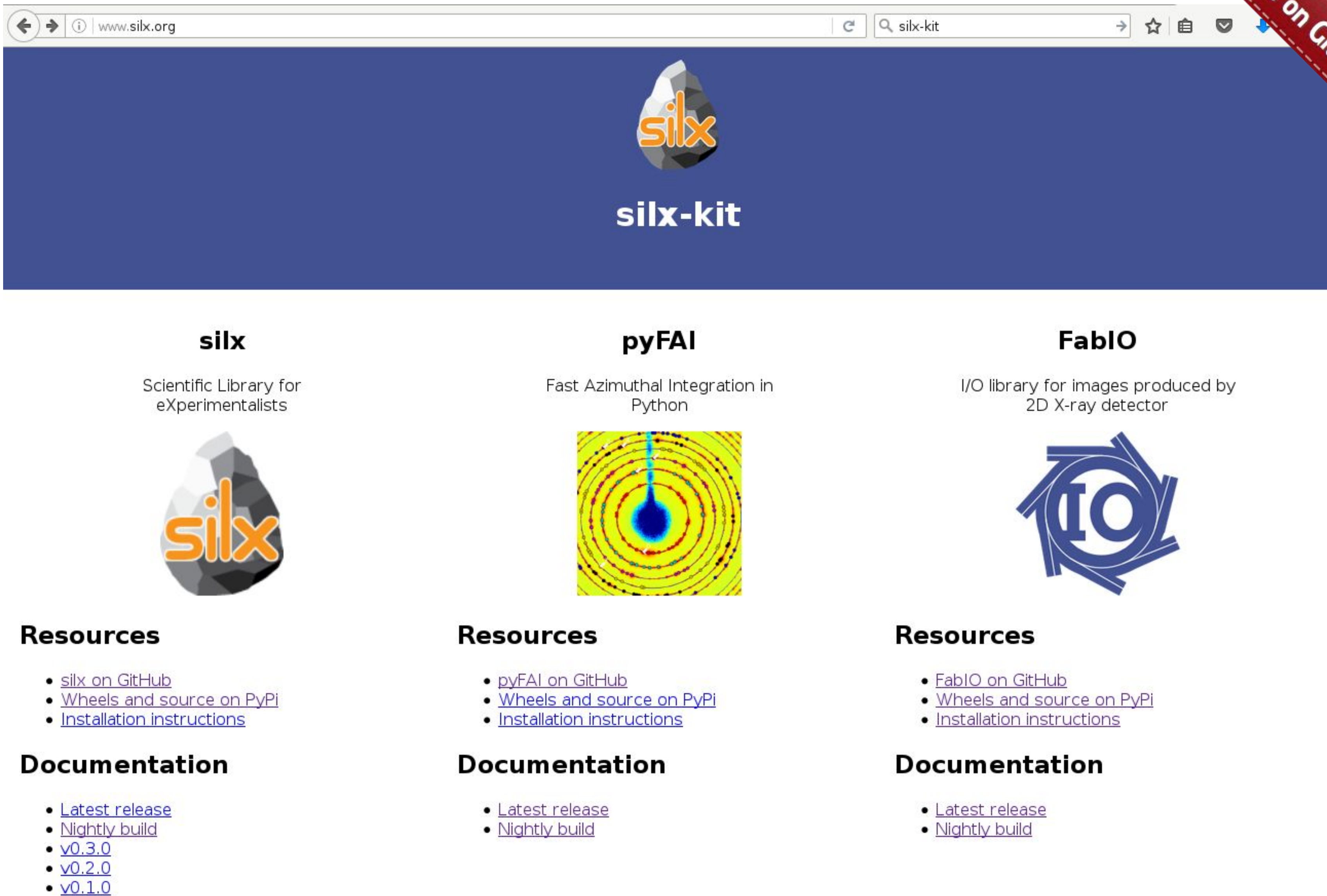
http://www.silx.org/doc/pyFAI/dev/project.html#getting-help

Reasons to choose pyFAI

- **Faster than others**
  - First tool using sparse matrix multiplication to perform integration
  - All computation intensive kernels are ported to C, C++ or OpenCL
  - PyFAI is the only azimuthal integration tool benefiting from GPU
- **More versatile (hackable) than other**
  - Many integration space already exists ...
    - **you can add your own easily**
  - Its geometry is so generic it matches any configurations
    - **SAXS, WAXS, moving detectors …**
  - Most detectors are already defined
    - **Each detector can be adapted, and saved in a Nexus file**
  - It has a nice user interface thanks to Valentin !
- **Part of the *silx* collaboration**
  - Bus-count slightly larger than one !

silx-kit: join efforts, share the maintenance

Fork me on GitHub



www.silx.org

silx-kit

silx-kit

silx

Scientific Library for eXperimentalists

Resources

- silx on GitHub
- Wheels and source on PyPi
- Installation instructions

Documentation

- Latest release
- Nightly build
- v0.3.0
- v0.2.0
- v0.1.0

pyFAI

Fast Azimuthal Integration in Python

Resources

- pyFAI on GitHub
- Wheels and source on PyPi
- Installation instructions

Documentation

- Latest release
- Nightly build

FabIO

I/O library for images produced by 2D X-ray detector

Resources

- FabIO on GitHub
- Wheels and source on PyPi
- Installation instructions

Documentation

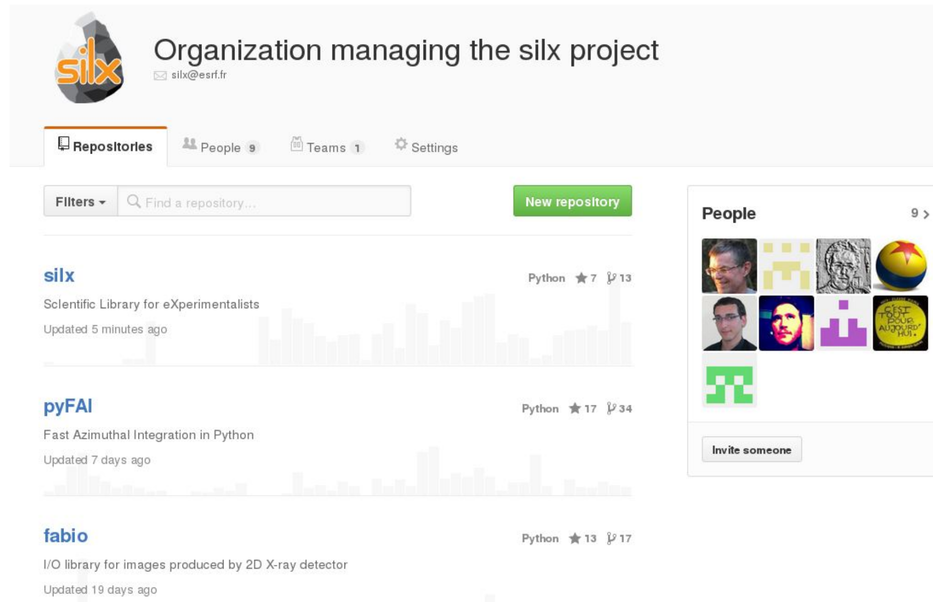
- Latest release
- Nightly build

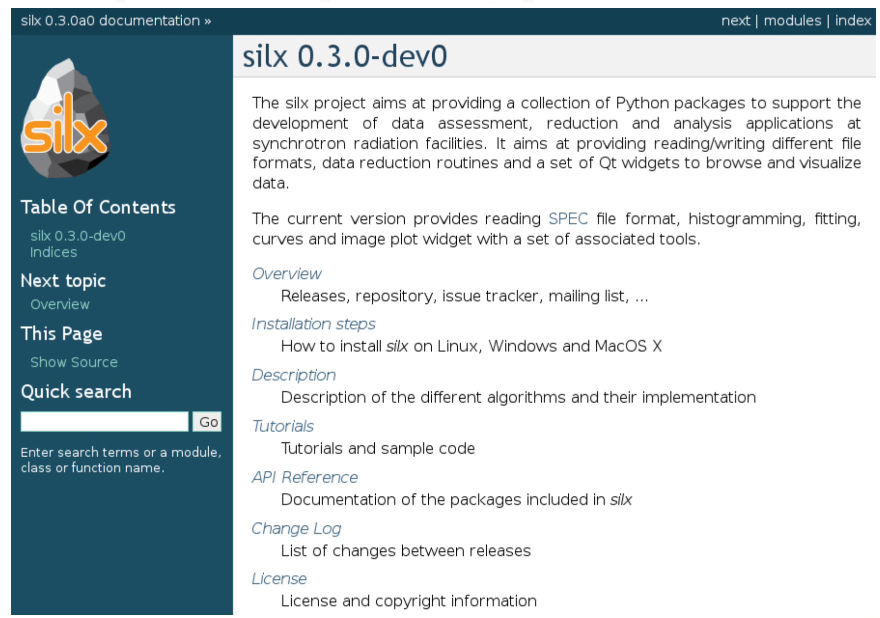
silx-kit: Shared development around:

- **User interface**
  - Common interface to Qt and soon jupyter-lab
  - Common visualization widgets
- **GPU computing**
  - Common initialization
- **Scientific data analysis**
  - Multi-threaded implementation of core algorithms

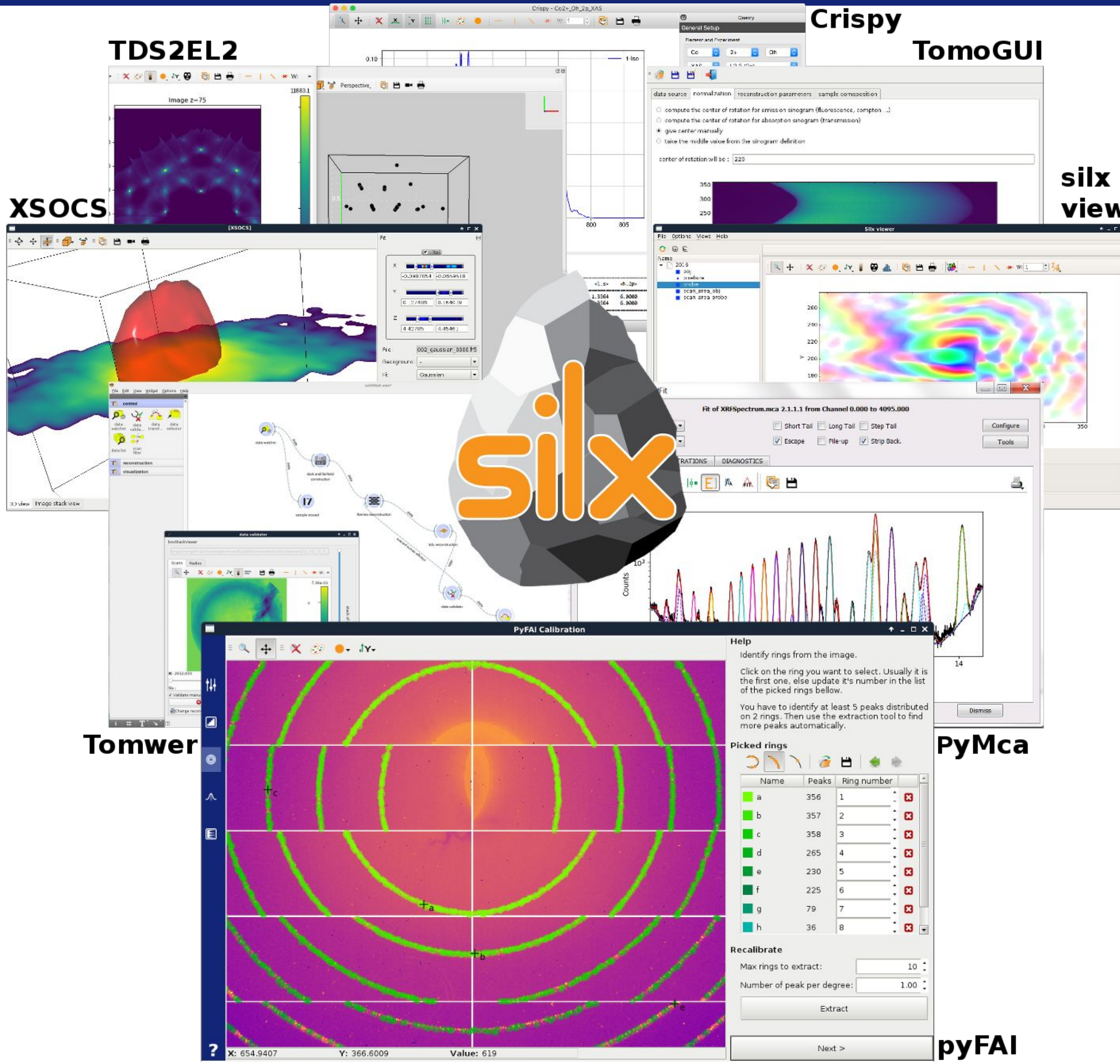
Management of the *silx-kit* project

- **Public project hosted at github**
  - https://github.com/silx-kit/silx
- **Continuous testing**
  - Linux, Windows and macOS
- **Nightly builds**
  - Debian packages
- ~~**Weekly meetings**~~
- ~~**Quarterly releases**~~
- ~~**Code camps before release**~~
- **Continuous documentation**
  - http://www.silx.org/doc/silx/

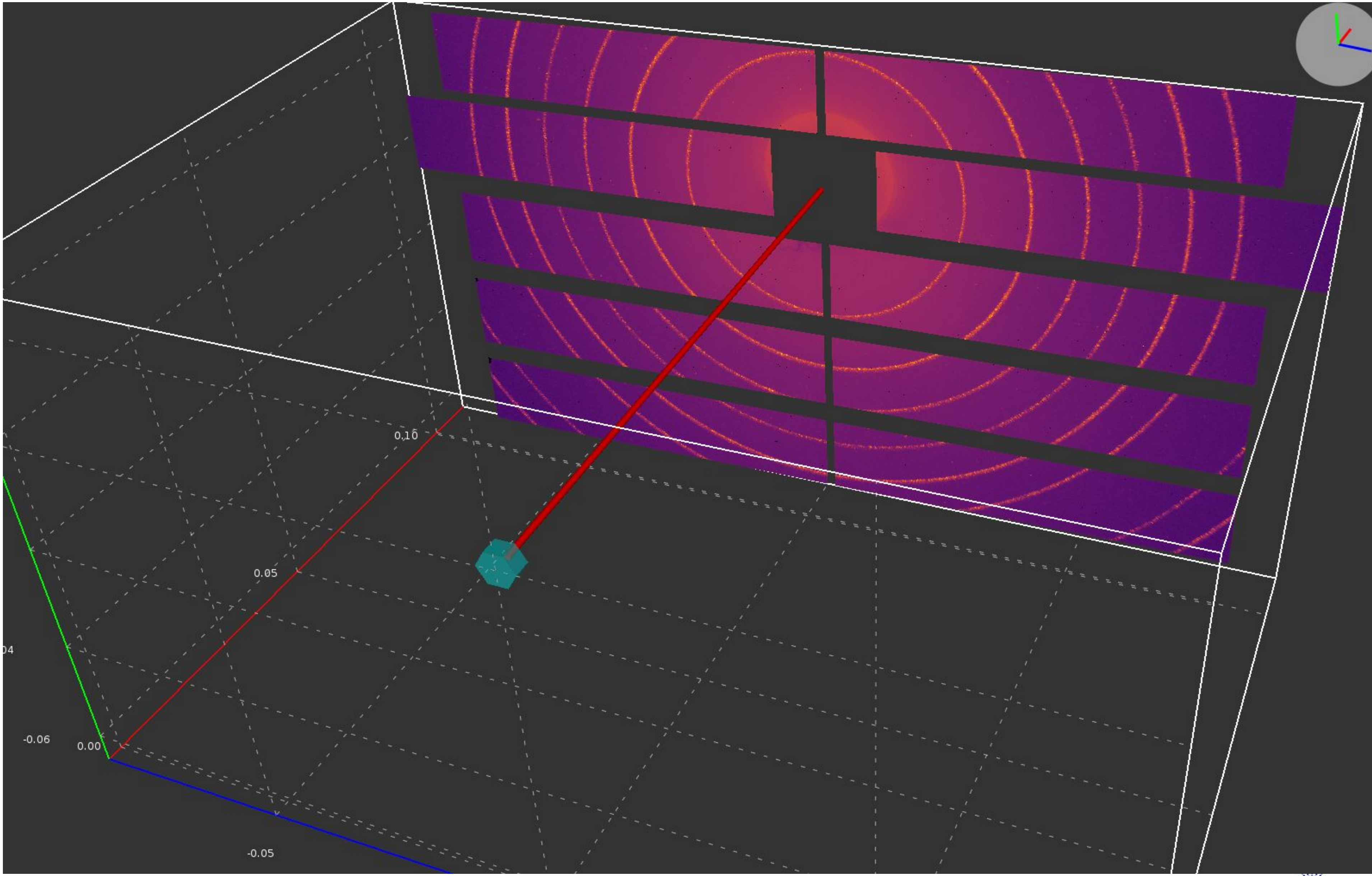




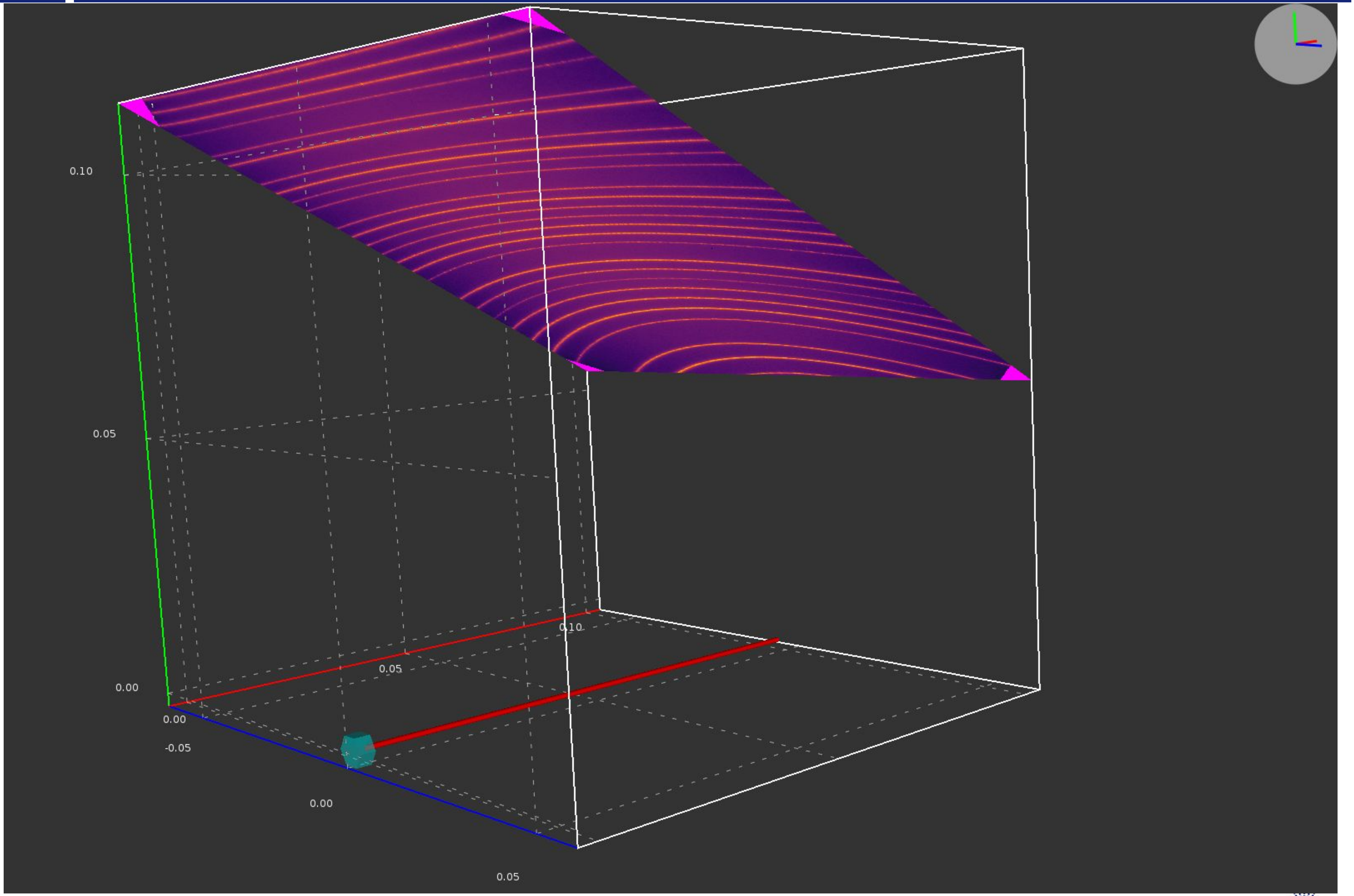
Outcome of the *silx* toolkit (2015-2018)



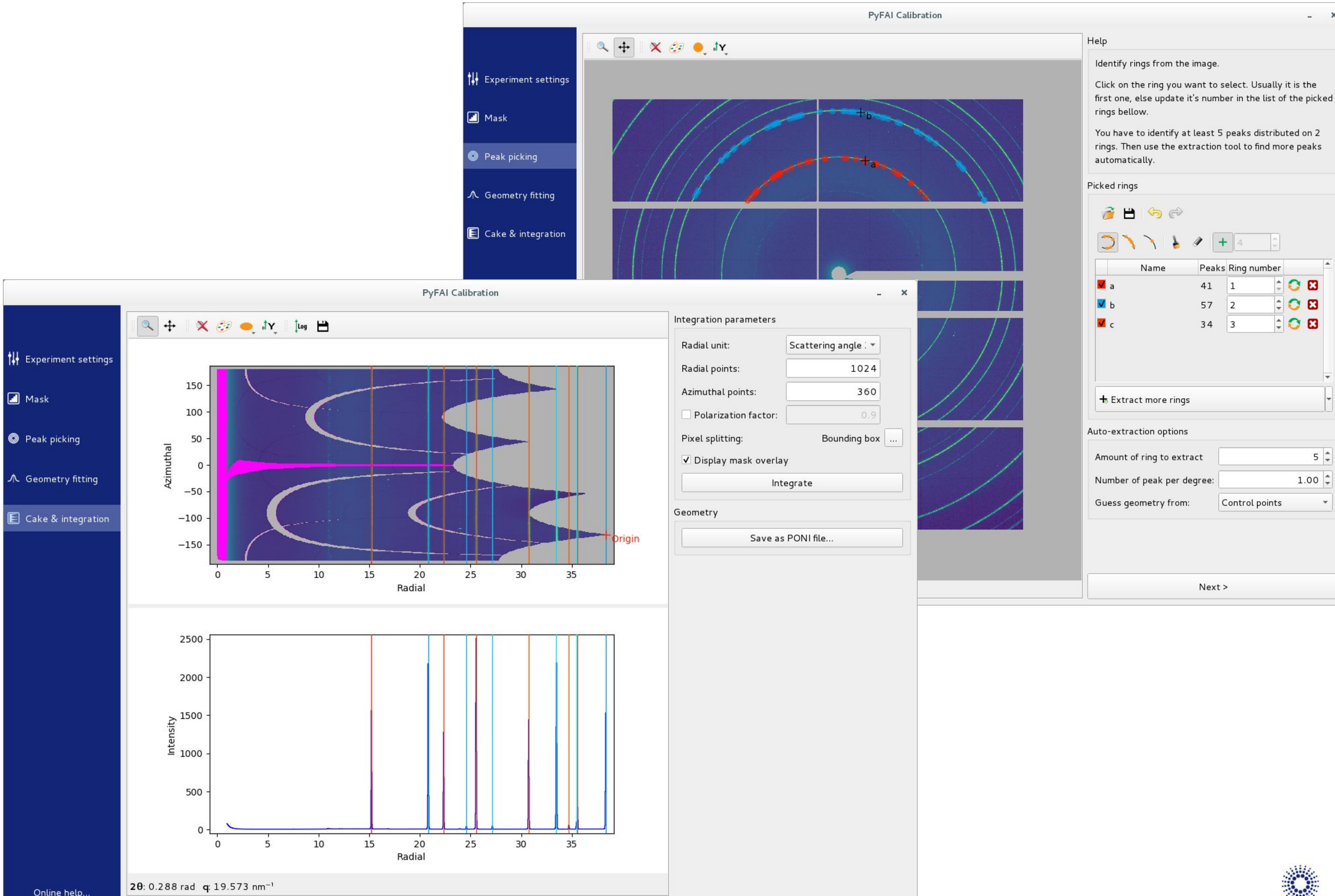
3D view of the diffraction setup



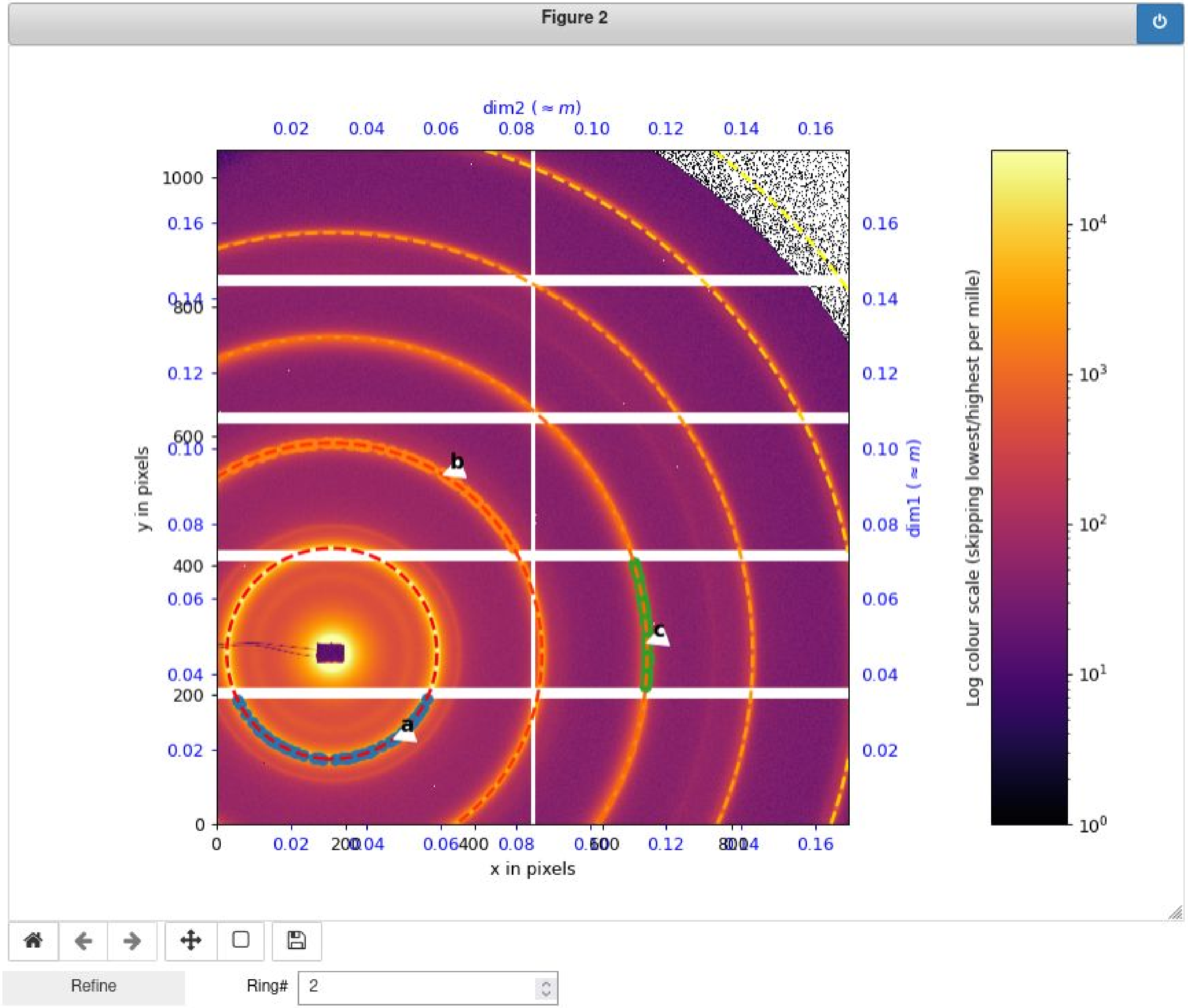
3D view of the diffraction setup



Calibration tools: Qt5 based



Calibration tools: Jupyter-lab



Acknowledgments

- **Former data analysis unit colleagues:**
  - Valentin Valls
  - Loïc Huder
  - Thomas Vincent
  - Claudio Ferrero†
- **ESRF Beamlines:**
  - BM01, BM02, ID02, ID11, ID13, ID15a, ID15b, ID21, ID22, ID23, BM26, ID27, ID28, BM29, ID29, ID30, ID31 ...
- **Trainees:**
  - Aurore Deschildre
  - Frederic Sulzmann
  - Guillaume Bonamis
- **Other synchrotron/labs**
  - Soleil: Fred Picca
  - Clemens Prescher (Dioptas)
  - Sesame: Philipp Hans
  - NSLS-II, ALS, APS, ...
- **International Grants:**
  - LinkSCEEM-2 grant
    - **Dimitris Karkoulis**
    - **Giannis Ashiotis**
    - **Zubair Nawaz**

Questions ?