

7 February 2022

User Meeting 2022



Welcome

The tutorial will start shortly





| The European Synchrotron



Welcome to the T8a: Data reduction for scattering experiments

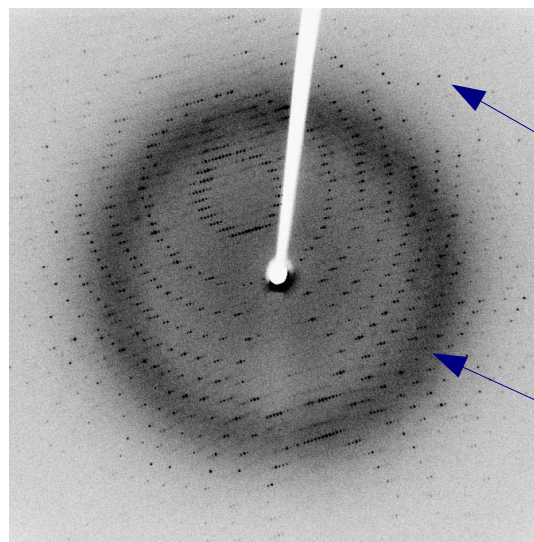
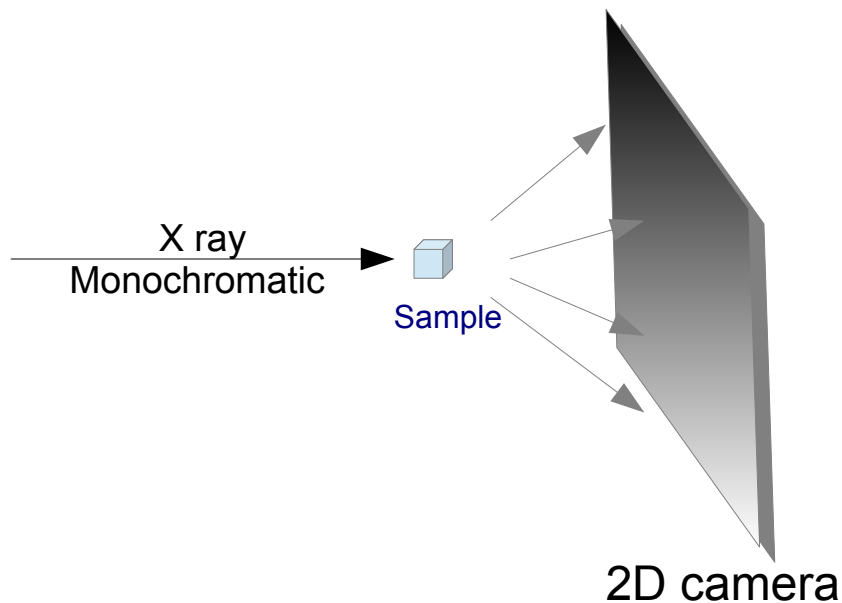
ESRF user meeting 2022, digital edition

Data reduction tools for scattering experiments

Jérôme Kieffer
Algorithms & scientific Data Analysis

- **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



Source: Wikipedia
CC-BY-SA: Jeff Dahl

Bragg spots:
diffraction from
single crystal

Ice ring: diffraction
from powder

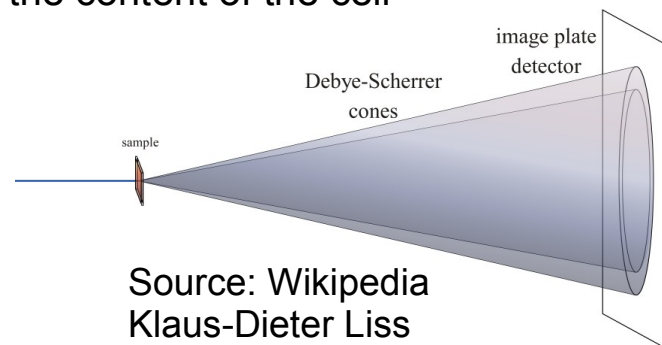
- **Light is reflected as on mirrors:**

- 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



Source: Wikipedia
Klaus-Dieter Liss

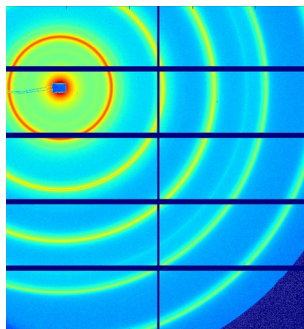
Application of powder diffraction:

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

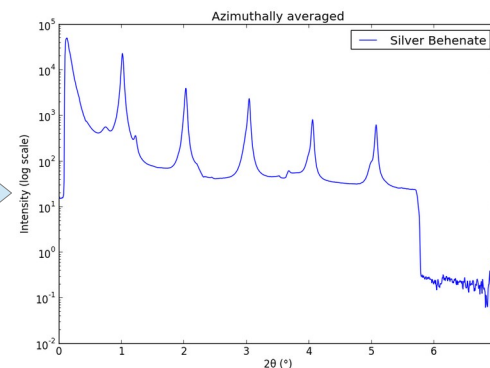
Application of small angle scattering

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

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

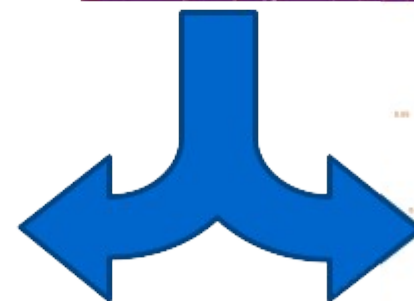
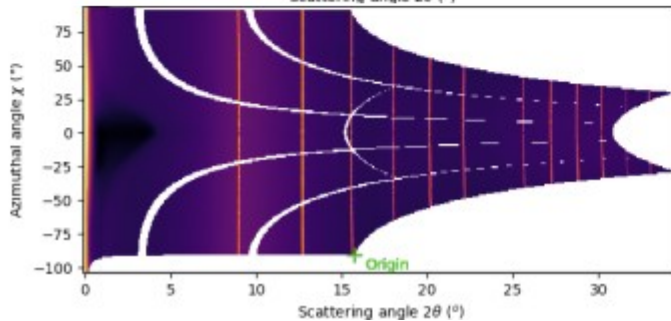
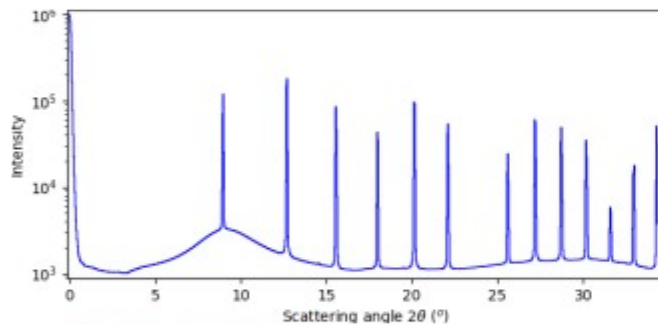
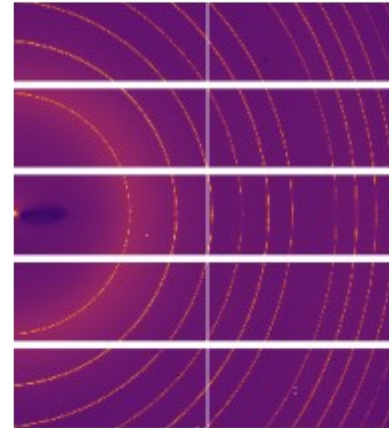


Azimuthal
integration



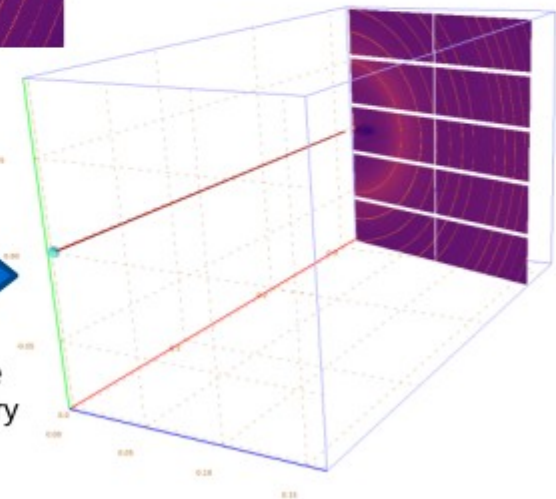
Fast Azimuthal Integration using Python

PyFAI
Fast Azimuthal Integration



Average
1d and 2d

Calibrate
3d geometry



- **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

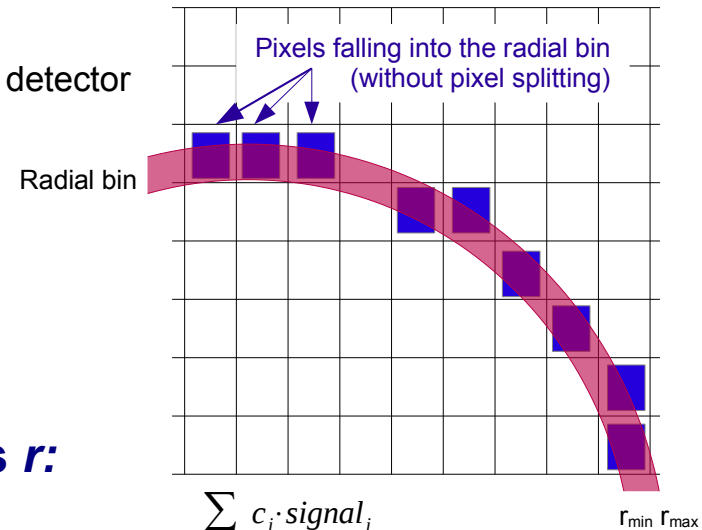
How it works

- Pixel-wise corrections:**

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

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 $_r$

$$\langle I \rangle_r = \frac{\sum_{i \in \text{bin}_r} c_i \cdot \text{signal}_i}{\sum_{i \in \text{bin}_r} c_i \cdot \text{normalization}_i}$$

- Associated uncertainty propagation:**

- Assuming there is no correlation between pixels
- Pixel splitting can create correlation between bins...

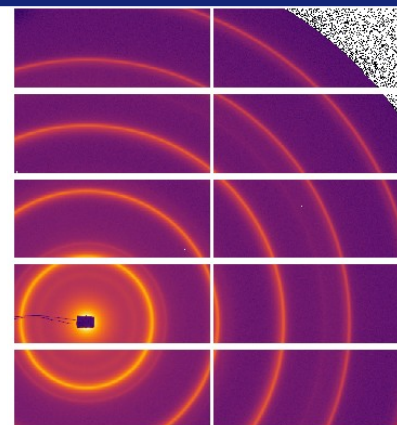
$$\sigma(I_r) = \sqrt{\frac{\sum_{i \in \text{bin}_r} c_i^2 \cdot \text{variance}_i}{\sum_{i \in \text{bin}_r} c_i \cdot \text{normalization}_i}}$$

$$\sigma(\langle I \rangle_r) = \sigma(I_r) \frac{\sqrt{\sum_{i \in \text{bin}_r} c_i^2 \cdot \text{normalization}_i^2}}{\sum_{i \in \text{bin}_r} c_i \cdot \text{normalization}_i}$$



- **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**

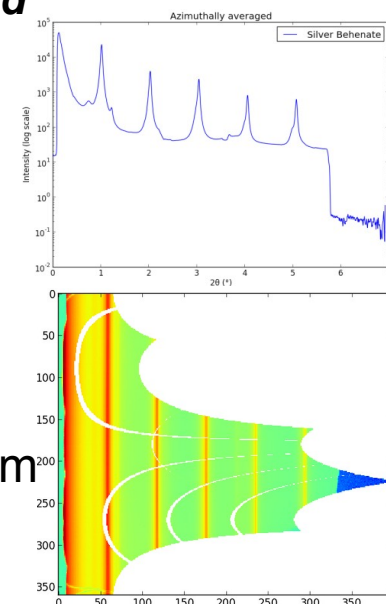
- Calculates the pixel position (center and corners)
- Calculates and stores the mask of invalid pixels.
→ saved as a HDF5 file



- **Geometry**

Position of the detector from the sample & incoming beam

→ saved as *PONI*-file

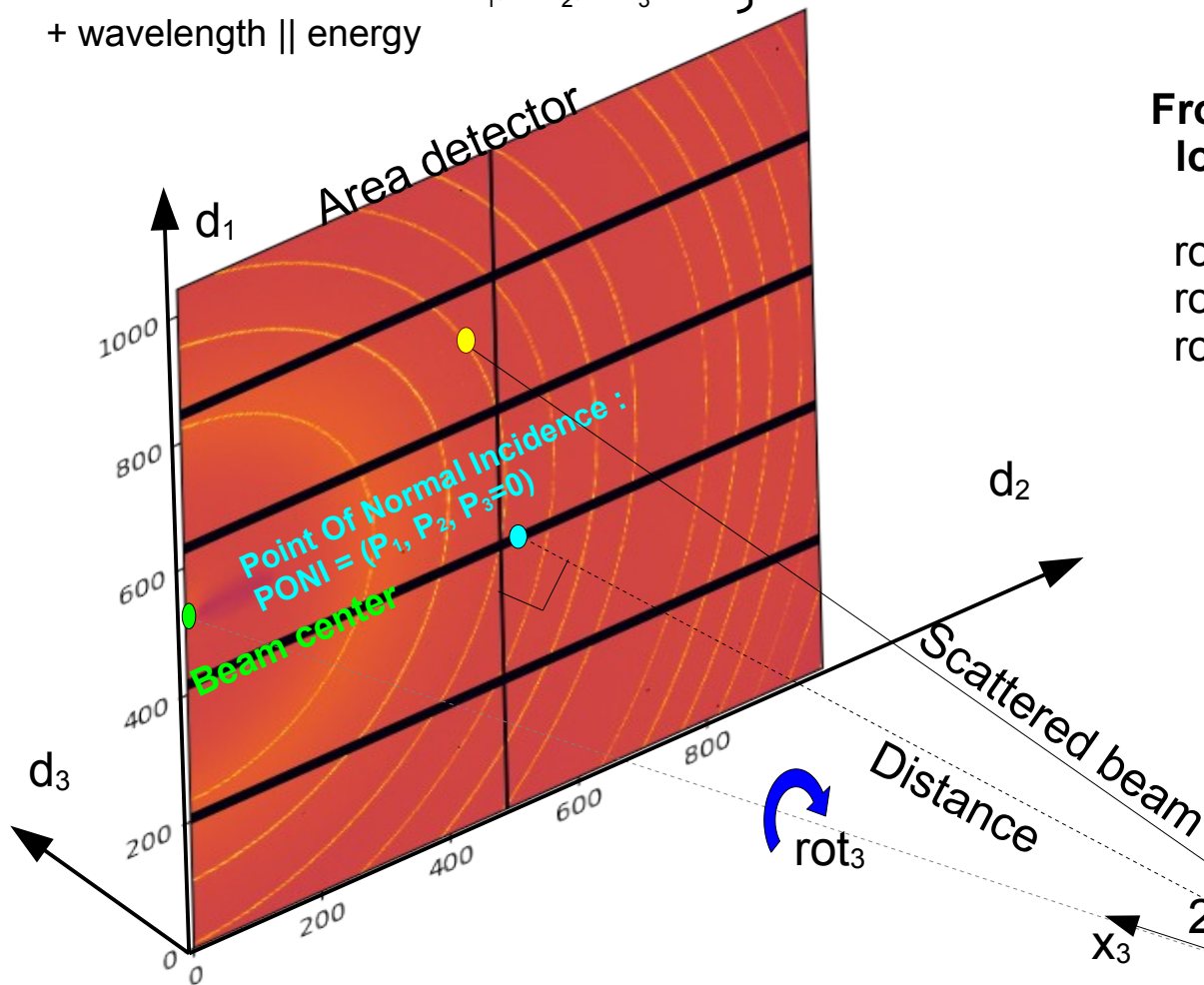


<http://www.silx.org/doc/pyFAI/dev/geometry.html#detector-position>

Geometry in pyFAI

Parameters:

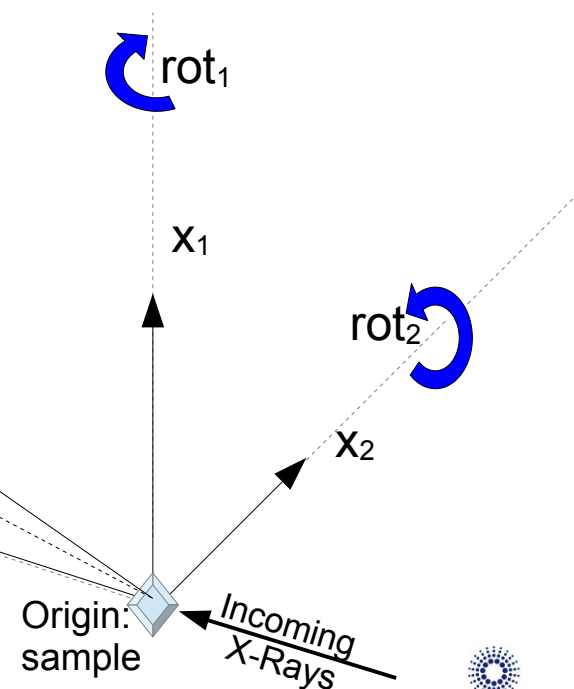
- * 3 distances in meters: dist , poni_1 , poni_2
 - * 3 rotations in radians: rot_1 , rot_2 , rot_3
 - + wavelength || energy
- } *PONI*-file



Detector's origin:
lower left, looking from
the sample

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



- **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 (LaB₆, CeO₂, AgBh, ...)**
 - **wavelength or energy used**
 - Only the position of the detector and the rotation needs to be refined:
 - **3 translations: dist, poni₁ and poni₂**
 - **3 rotations: rot₁, rot₂, rot₃**
- **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]: 1 import numpy
2 npt = 1024
3 y,x = numpy.ogrid[-512:512, -512:512]
4 radius = (x*x+y*y)**0.5
5 rmax = radius.max()+0.1
6 data = numpy.random.random((1024,1024))
```

Naive approach integration using corona extraction with masks:

```
In [2]: 1 %%time
2 res_loop = numpy.zeros(npt)
3 for i in range(npt):
4     rinf = rmax * i / npt
5     rsup = rinf + rmax / npt
6     mask = numpy.logical_and((rinf <= radius), (radius < rsup))
7     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]: 1 %%time
2 count_of_pixels = numpy.histogram(radius, npt, range=[0,rmax] )[0]
3 sum_of_intensities = numpy.histogram(radius, npt, weights=data, range=[0,rmax])[0]
4 res_vec = sum_of_intensities / count_of_pixels
```

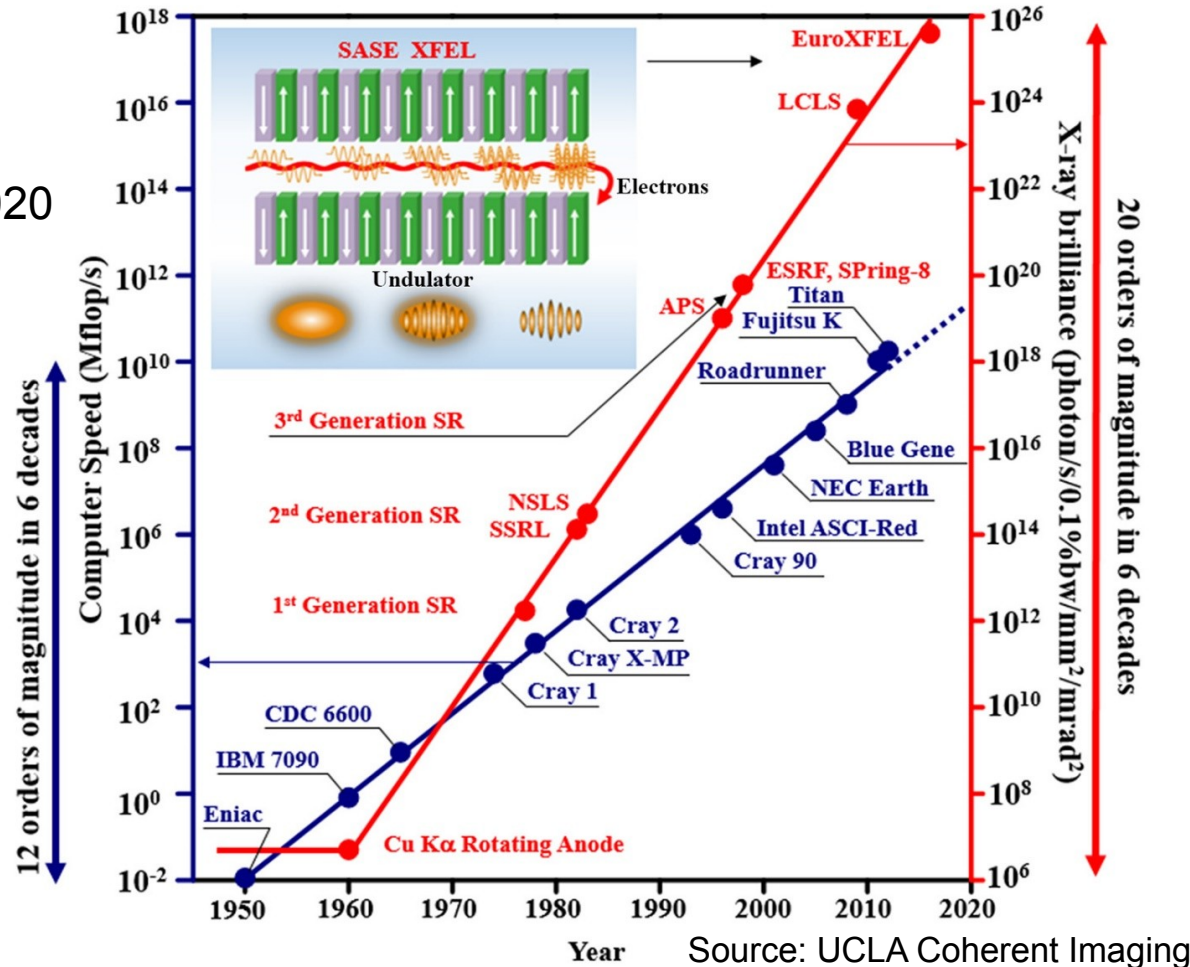
CPU times: user 19.5 ms, sys: 1.44 ms, total: 20.9 ms
Wall time: 19.4 ms

```
In [4]: 1 # Speed-up: 50x, validation:
2 numpy.allclose(res_loop, res_vec)
```

Out[4]: True

But speed does matters ...

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



- **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>

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]:

```
1 %%time
2 from scipy.sparse import csc_matrix
3 positions = numpy.histogram(radius, npt, range=[0,rmax] )[1]
4 row = numpy.digitize(radius.ravel(), positions) - 1
5 size = row.size
6 col = numpy.arange(size)
7 dat = numpy.ones(size, dtype=float)
8 csr = csc_matrix((dat, (row, col)), shape = (npt, data.size))
9 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]:

```
1 %%time
2 count_csr = csr.dot(numpy.ones(data.size))
3 sum_csr = csr.dot(data.ravel())
4 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]:

```
1 # Speed-up: 5x vs histograms, validation:
2 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

- Pro
- **Easier to understand**
 - **Low memory consumption**
 - **Fast initialization**

Sparse matrix multiplication

- **Faster, even on a single core**
- **Many-core friendly**
 - OpenMP and OpenCL

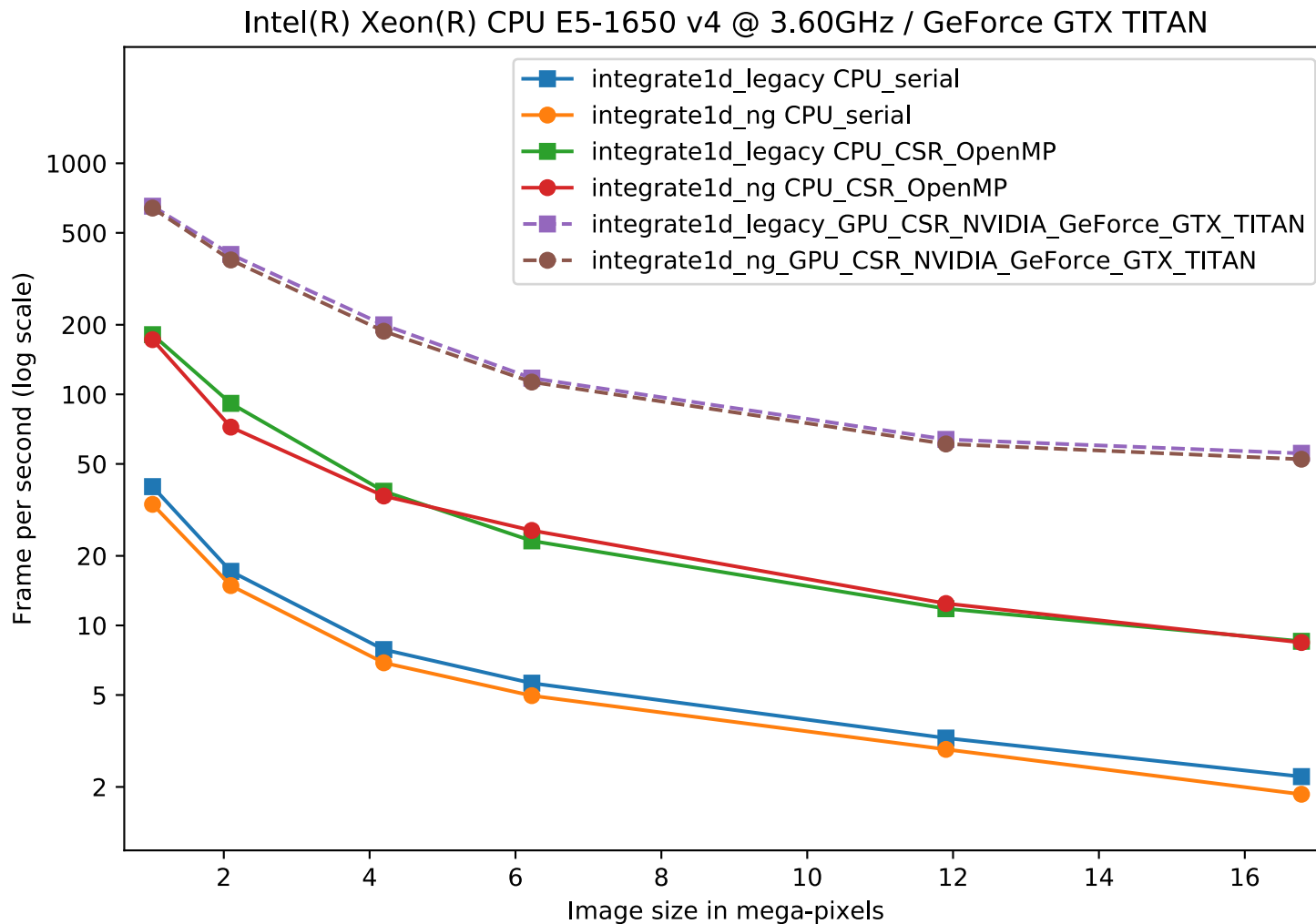
- Con
- **Pretty slow**
 - **Hardly parallelizable**

- **Slower initialization**
- **The sparse matrix can be large**

Rule of thumb: < 5 frames

≥ 5 frames

Benchmark: Let's speak about speed !

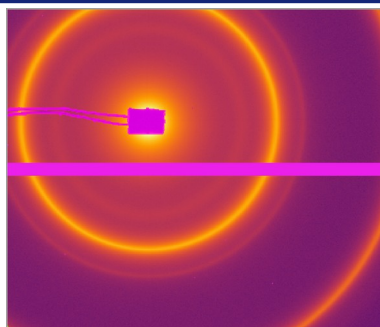


Workstation with CPU & GPU from 2012 !

High frequency noise issue

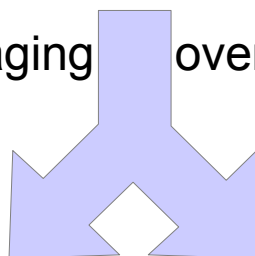
Where pixel splitting comes back

Example with SAXS data integrated in 2D



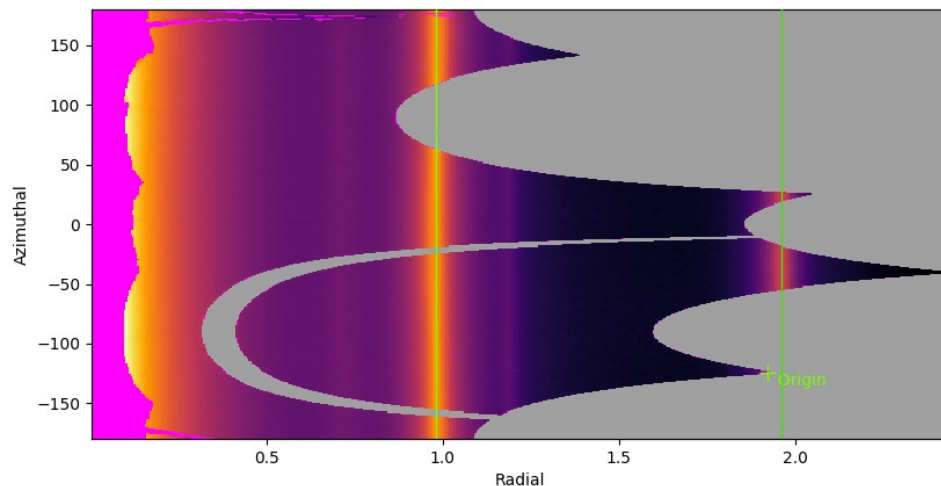
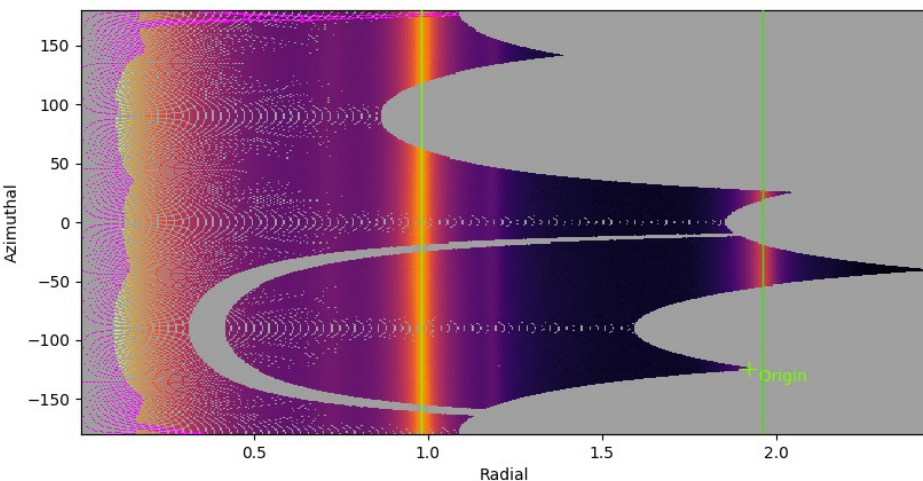
Pilatus 200k:
~500 x 400 pixels

2D averaging over 512x360 bins



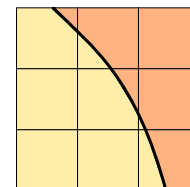
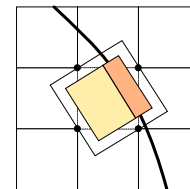
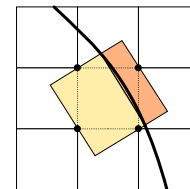
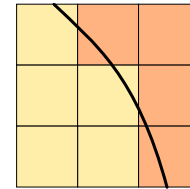
Without pixel splitting

With pixel splitting



creates bin cross-correlation

- **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



- **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:**

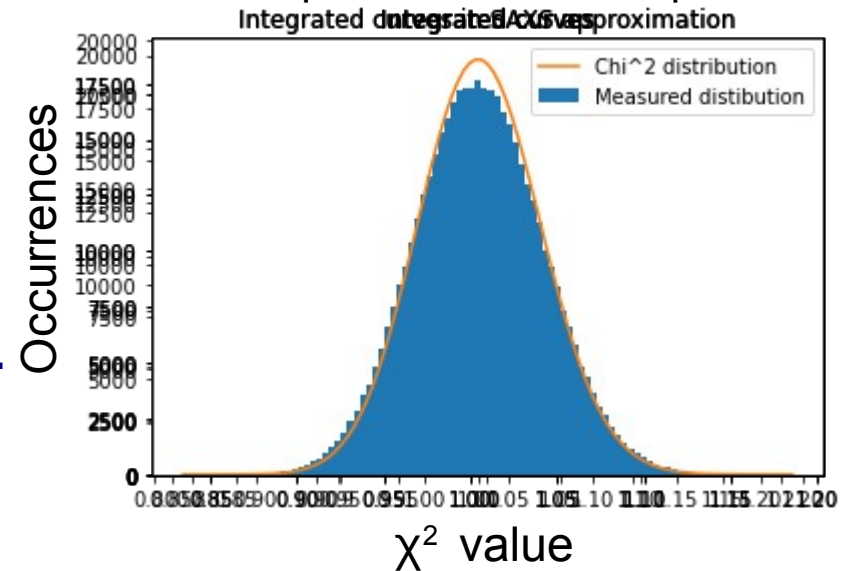
- SAXS-like data, 1000 frames with synthetic distribution, 5e5 pairs of curves compared.

- **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
 - **Fixed since PyFAI v0.20.0 (1d integration), v0.21.0 (2d integration)**
- 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>



- **Applications level:**

- GUI applications: **pyFAI-calib2**, **pyFAI-integrate**, **diff_map**
- Scriptable applications: **pyFAI-average**, pyFAI-saxs, pyFAI-waxs, diff_tomo, .

- **Python interface:**

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

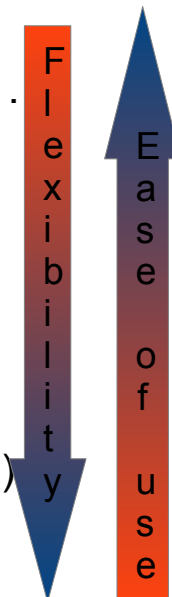
Ideally used from  jupyter

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

It is up to you !



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

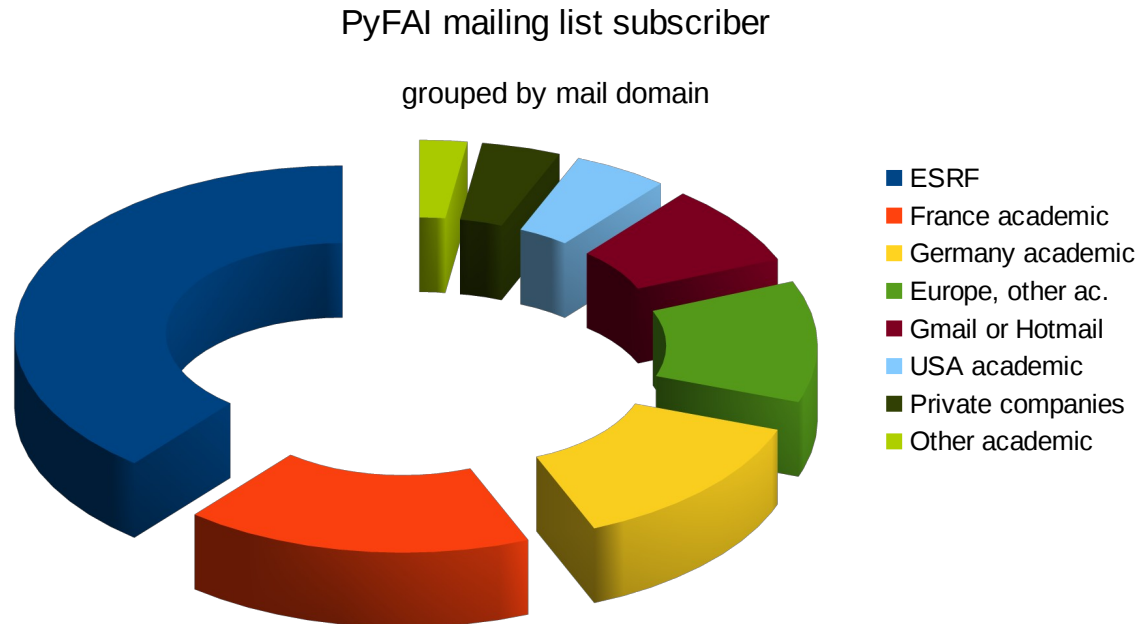


Silx & pyFAI

PyFAI is yet another azimuthal integration tool

- **Written in Python (compatible with ~~2.7, 3.5~~, 3.6, 3.7, 3.8 & 3.9 & 3.10)**
 - Free, fast and portable
 - 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 > 50 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**
 - **science does not suffer approximations**

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



- **User support is provided via the mailing list: pyFAI@esrf.fr**
 - 154 people subscribed to the list 2022 (was 142, 137, 132, 112)
 - limited activity (~1 thread/month)
- **Bugs are discussed via the Github issue tracker:**
 - <https://github.com/silx-kit/pyFAI/issues>

- **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

Scientific Library for
eXperimentalists



Resources

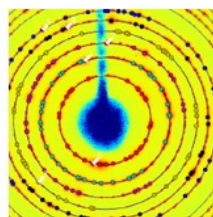
- [silx on GitHub](#)
- [Wheels and source on PyPi](#)
- [Installation instructions](#)

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](#)

- **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~~

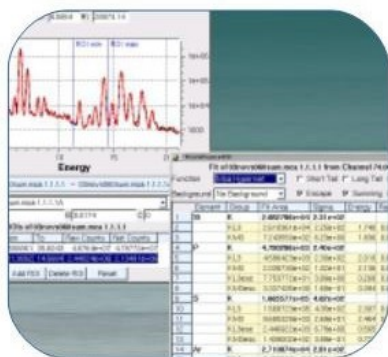
- ~~Code camps before release~~

- **Continuous documentation**

<http://www.silx.org/doc/silx/>

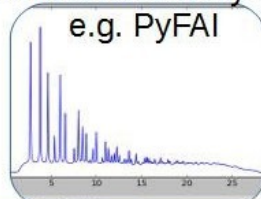


Was Pierre Knobel

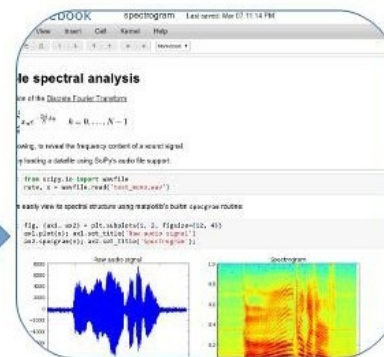


Standard Apps e.g.
PyMCA, PyDIF
... and Valentin Valls

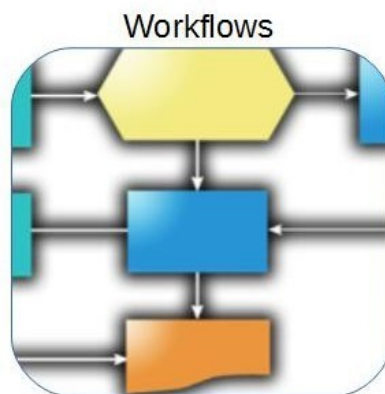
Mainly Jérôme Kieffer
Online data analysis
e.g. PyFAI



Loïc Huder & Axel Bocciarelli



Ipython Notebook



Workflows

Henri Payno & Wout de Nolf



General Purpose
Core Toolkit



External
Libraries +
Apps

Mainly Thomas Vincent

Extensions



Extension library
e.g. PyHST, ...

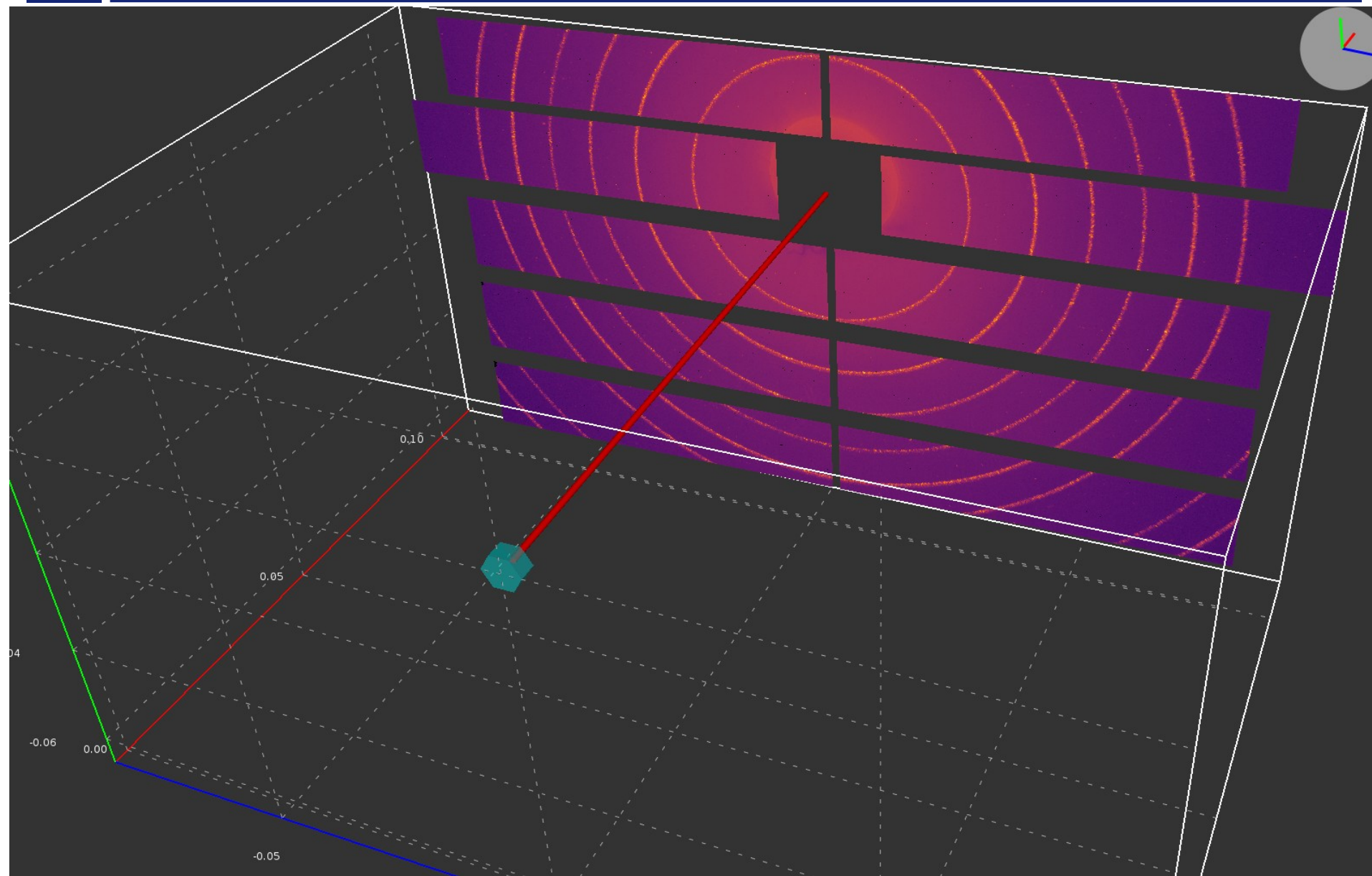
Pierre Paleo & Alessandro Mirone

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

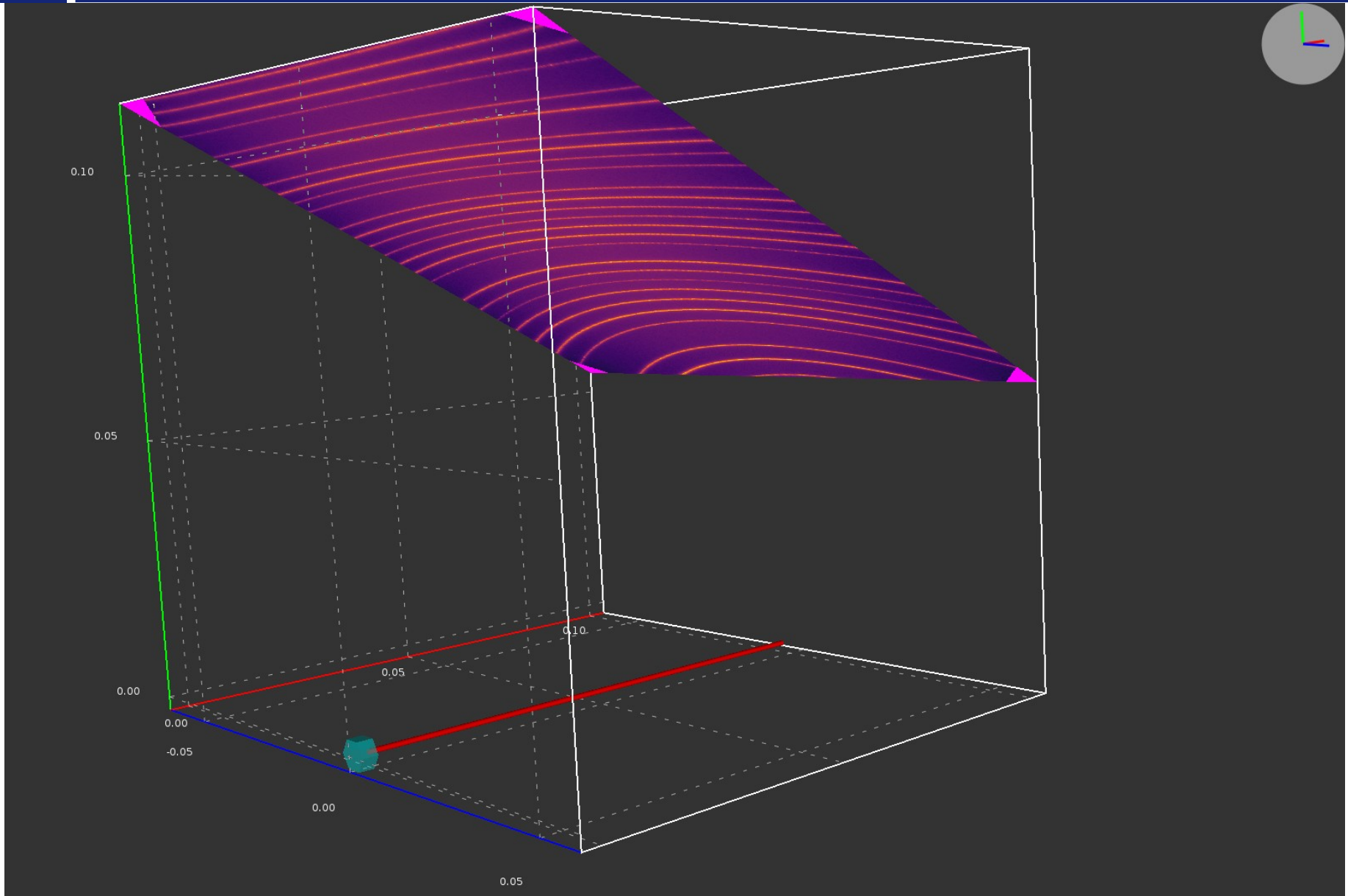
The image displays a central 3D model of a crystal structure with the **silx** logo overlaid. Surrounding this are several software windows from the silx toolkit:

- TDS2EL2**: A window showing a 2D image of a crystal lattice with a color scale and a 3D perspective view of the lattice.
- XSOCS**: A window showing a 3D visualization of a crystal structure with various parameters and a data table.
- TomoGUI**: A window for tomographic reconstruction, showing reconstruction parameters and a 2D slice of the reconstructed volume.
- silx view**: A window showing a 2D slice of the reconstructed volume with a color scale.
- Tomwer**: A window showing a 2D image of a crystal lattice with a color scale and a 3D perspective view of the lattice.
- PyMca**: A window showing a 2D image of a crystal lattice with a color scale and a 3D perspective view of the lattice.
- pyFAI**: A window showing a 2D image of a crystal lattice with a color scale and a 3D perspective view of the lattice.

3D view of the diffraction setup



3D view of the diffraction setup



Calibration tools: Qt5 based

PyFAI Calibration

Experiment settings
Mask
Peak picking
Geometry fitting
Cake & integration

Help
Identify rings from the image.
Click on the ring you want to select. Usually it is the first one, else update it's number in the list of the picked rings below.
You have to identify at least 5 peaks distributed on 2 rings. Then use the extraction tool to find more peaks automatically.

Picked rings

Name	Peaks	Ring number
<input checked="" type="checkbox"/> a	41	1
<input checked="" type="checkbox"/> b	57	2
<input checked="" type="checkbox"/> c	34	3

Integrate

Auto-extraction options
Amount of ring to extract: 5
Number of peak per degree: 1.00
Guess geometry from: Control points

Next >

PyFAI Calibration

Experiment settings
Mask
Peak picking
Geometry fitting
Cake & integration

Integration parameters
Radial unit: Scattering angle
Radial points: 1024
Azimuthal points: 360
 Polarization factor: 0.9
Pixel splitting: Bounding box
 Display mask overlay
Integrate

Geometry
Save as PONI file...

Azimuthal

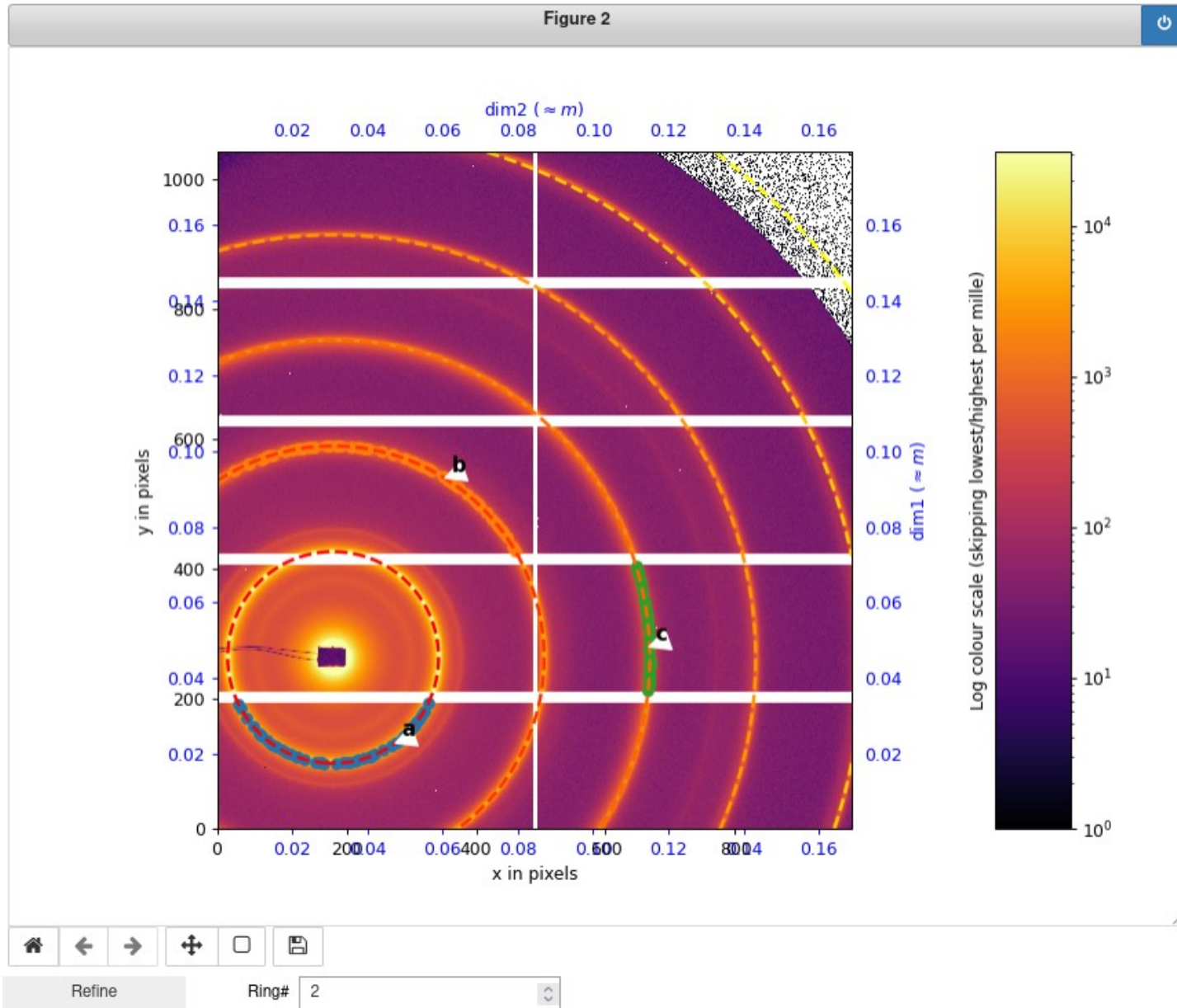
Radial

Intensity

Radial

2θ: 0.288 rad q: 19.573 nm⁻¹

Calibration tools: Jupyter-lab



- **Former data analysis unit staff:**

- Valentin Valls
- Loïc Huder
- Thomas Vincent
- V. Armando Solé
- Claudio Ferrero†

- **ESRF Beamlines:**

BM01, BM02, ID02, ID11,
ID13, ID15, ID21, ID22, ID23,
BM26, BM29, ID29, ID30, ID31
...

- **Trainees:**

- Aurore Deschildre
- Frederic Sulzmann
- Guillaume Bonamis

- **Other synchrotron/labs**

- Soleil: Fred Picca, Diffabs & Cristal beamlines
- Clemens Prescher (APS, EuXFEL, ...)
- Philipp Hans (U. Marseille, APS, ...)
- NSLS-II: scikit-beam project
- ALS: Camera project

- **International Grants:**

- LinkSCEEM-2 grant
 - **Dimitris Karkoulis**
 - **Giannis Ashiotis**
 - **Zubair Nawaz**

Questions ?



7 February 2022

User Meeting 2022



Coffee Break

The tutorial will resume shortly

