ESRF | The European Synchrotron

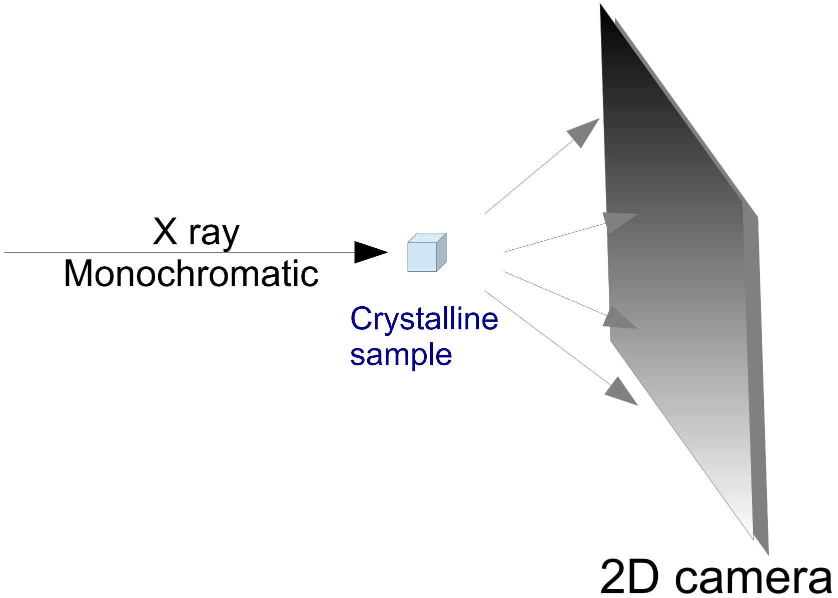
Data reduction tools for scattering experiments

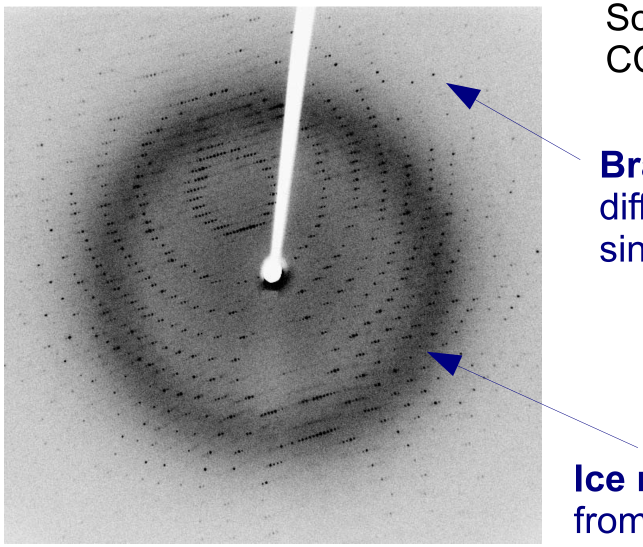
Jérôme Kieffer
Online data analysis @ ESRF

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
  - The *silx* collaboration
  - Latest developments: 3D view of the experimental setup
- **Conclusions**

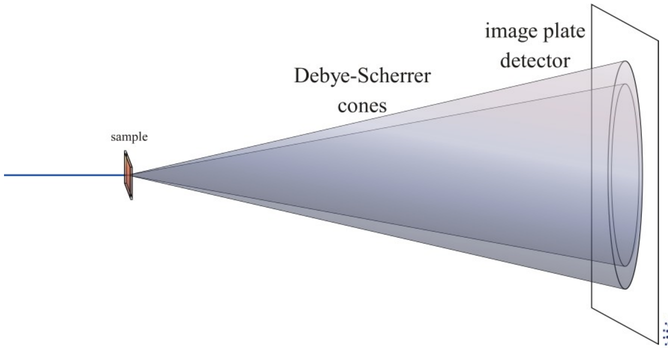
X-ray scattering experiments





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

    → Nobel price of Bragg (1915) $n\lambda = 2d\sin\theta\,,$
- **Multiple small crystals (powder)**
  - Isotropic cones giving conics (mainly ellipses) when intersected with the detector



Powder diffraction and small angle scattering

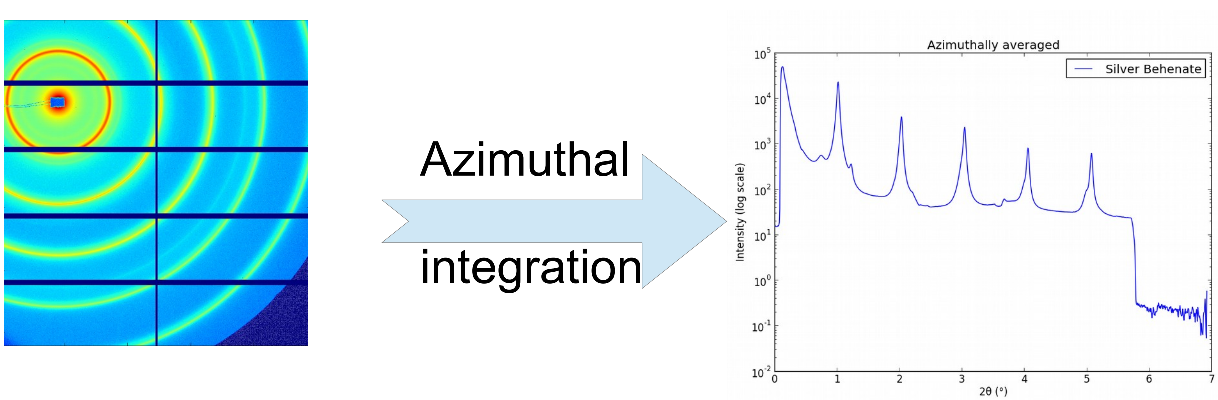
Application of powder diffraction:

- 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

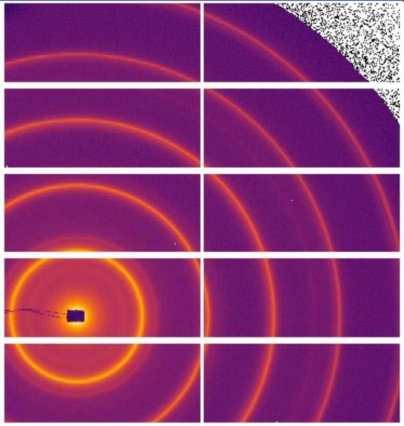
- **Both rely on the same transformation: 2D image → azimuthal average**



Many different tools exists …

- **FIT2D**
  - MIT licensed from ESRF, written in Fortran, now discontinued
- **XRDUA**
  - GPL licensed from U. Antwerp written in IDL, focuses of diffraction mapping
- **Dawn**
  - EPL license from Diamond Light Source, written in Java
- **DataSqueeze**
  - Freeware from U. Pennsylvania
- **Foxtrot**
  - Commercial, from Xenocs & synchrotron Soleil, written Java
- **Maud**
  - Freeware from U. Trento, written in Java
- **GSAS-II**
  - Freeware tool from U.Chicago & APS, written in Python
- **Scikit-beam**
  - BSD licensed from NSLS-II & BNL, written in Python.

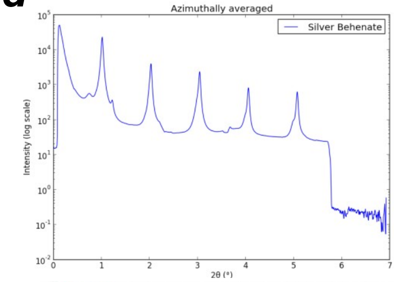
Concepts in PyFAI



- **Image**

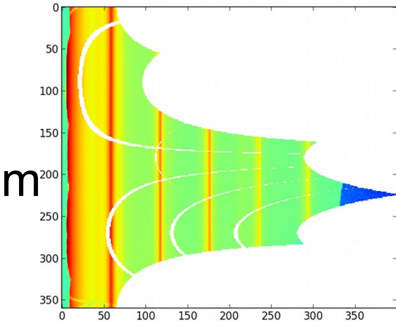
  2D array of pixels as *numpy* array
  read using *silx, fabio, h5py*, ...

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



- **Detector**
  - **Calculates the pixel position (center and corners)**
  - **Calculate or store the mask**

  → 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

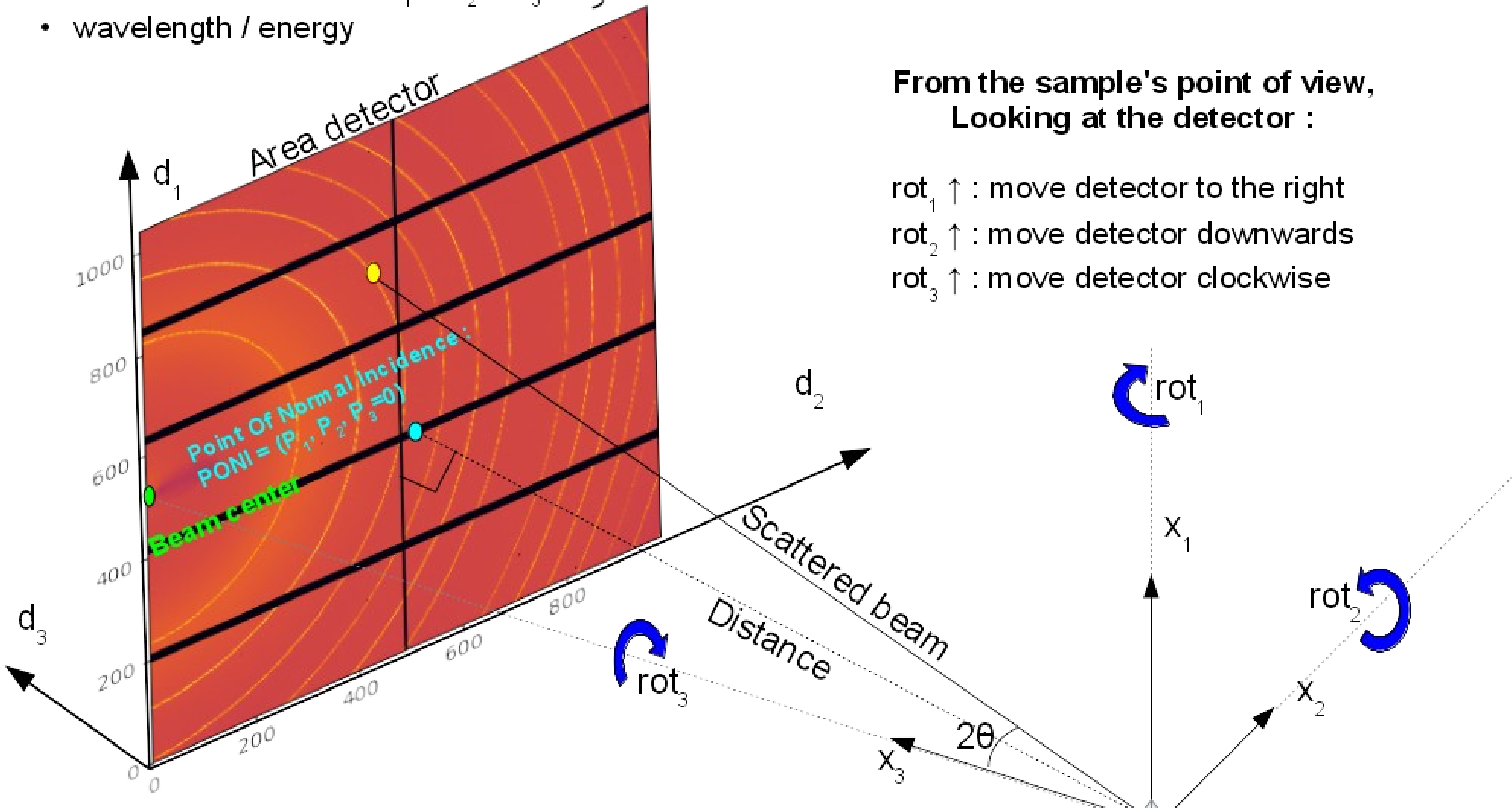
Geometry in pyFAI

- 3 distances in meter: dist, $poni_1$, $poni_2$
- 3 rotation in radians: rot_1, rot_2, rot_3
- wavelength / energy

$\}$ *PONI*-file

From the sample's point of view, Looking at the detector :

$rot_1 \uparrow$: move detector to the right
$rot_2 \uparrow$: move detector downwards
$rot_3 \uparrow$: move detector clockwise



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

Calibration in pyFAI

- **The determination of the geometry is also known as calibration**
  - 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 an human being of the geometry
- **PyFAI assumes this setup does not change during the experiment**
- **Tutorial:**

  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.**

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*:**

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

Where c_i is the fraction of the pixel i contributing to bin_r

- **Associated error propagated:**

  - Assuming there is no correlation between pixels
  - Can create correlation between bins

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

Math from Kieffer et al.; *J. Synch. Radiation* (2020) accepted

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
```

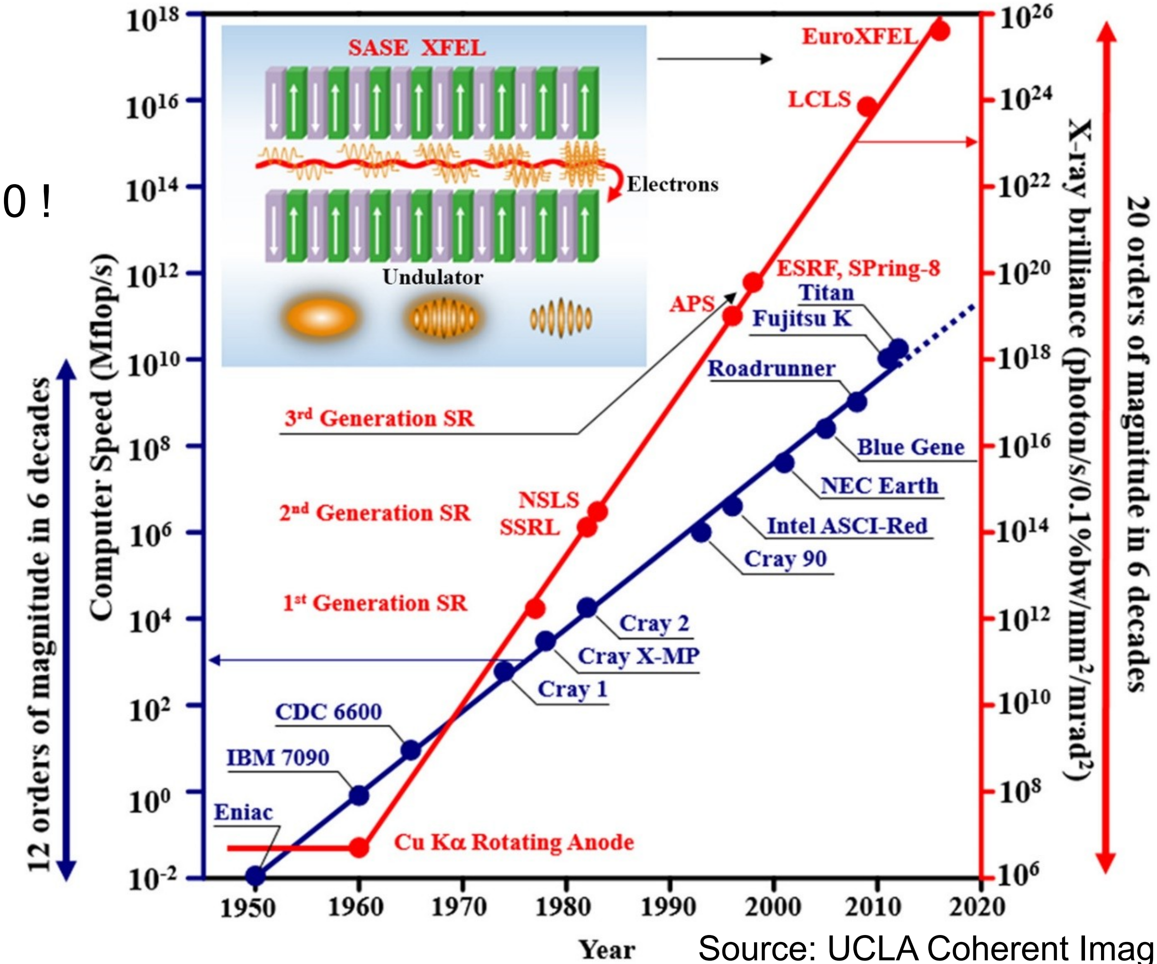
```
CPU times: user 19.5 ms, sys: 1.44 ms, total: 20.9 ms
Wall time: 19.4 ms
```

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

Out[4]: True

Speed matters ...

- **New EBS source**
  - 50x brighter
  - Starts in March 2020 !



Source: UCLA Coherent Imaging

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

→ Stream limited to 2 Gigabyte/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 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]:

```
%%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)
```

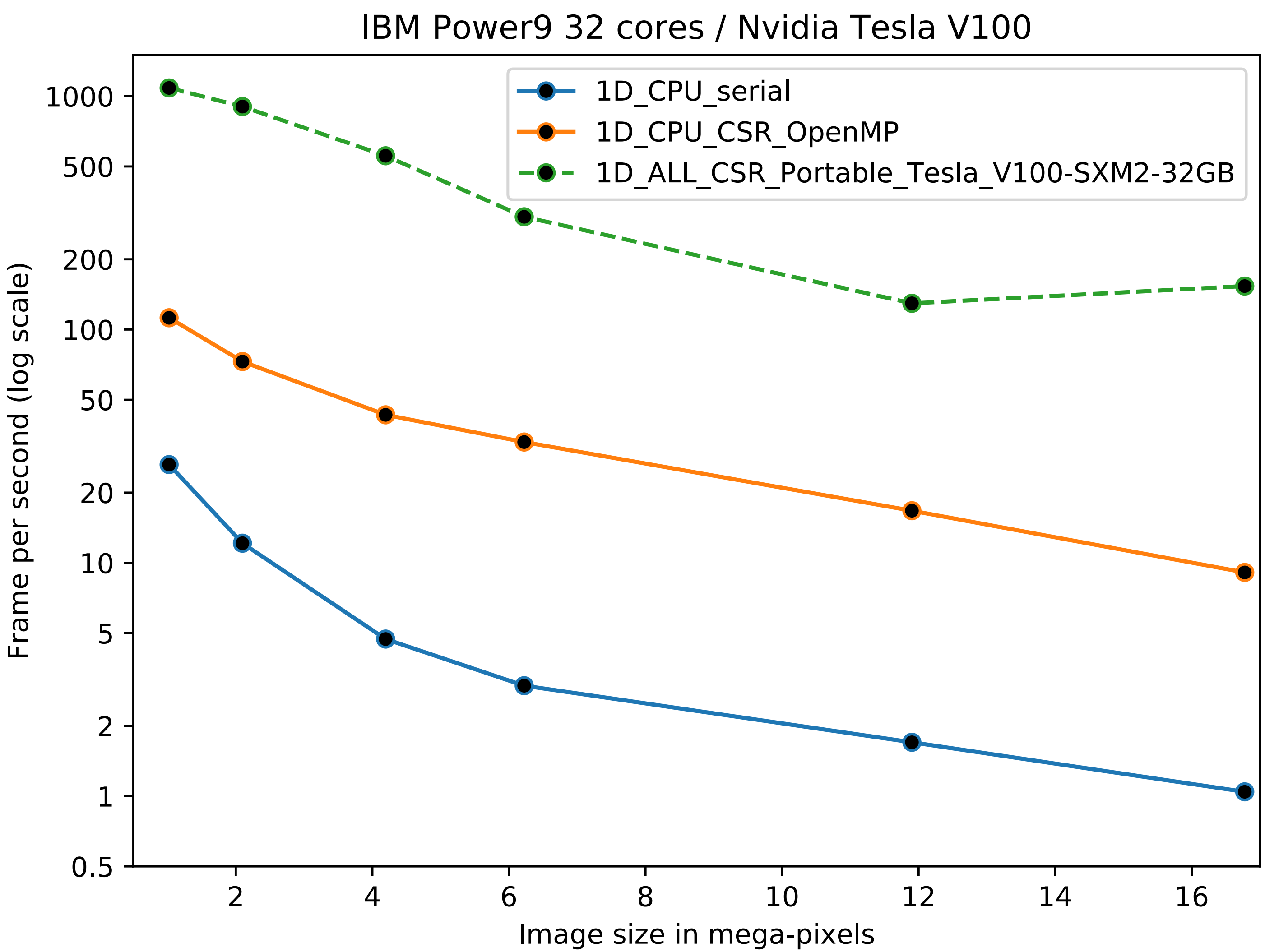
Out[7]: True

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

Advantages of histograms vs sparse 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 !

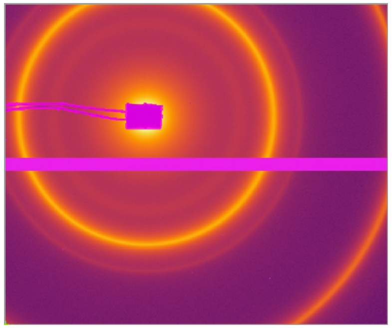


GPU cluster foreseen for ESRF's restart and online data analysis, up to 4x V100 per computer

Moiré effect

High frequency noise issue

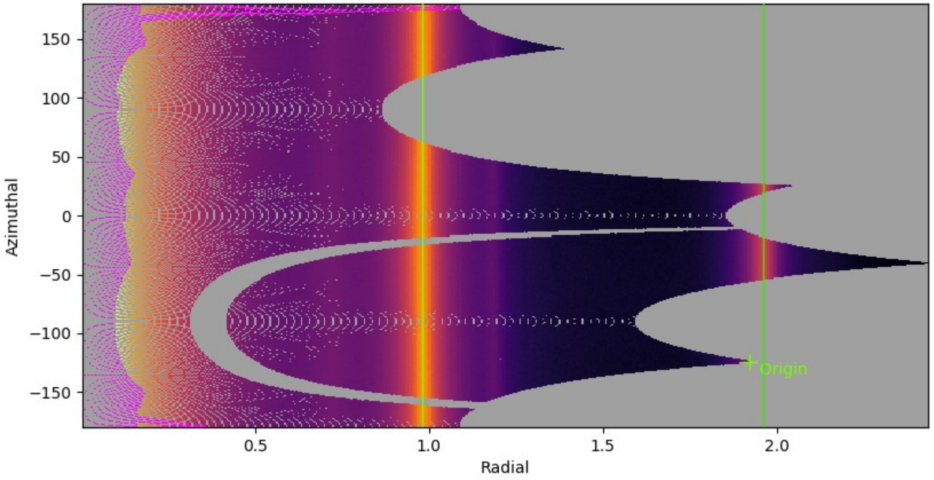
Example with SAXS data integrated in 2D



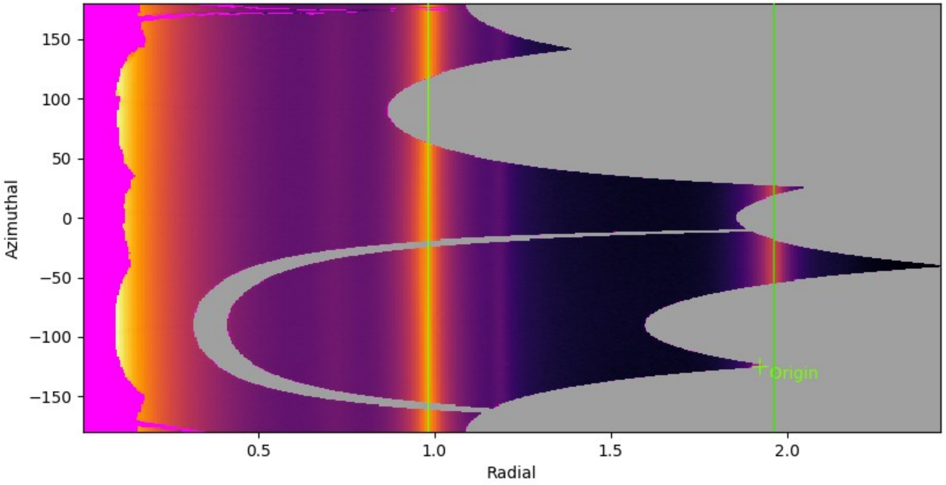
Pilatus 200k:
~500 x 400 pixels

2D averaging over 512x360 bins

Without pixel splitting



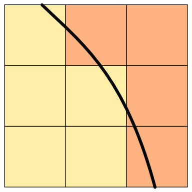
With pixel splitting



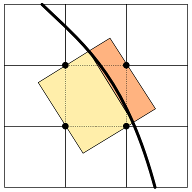
⚠ creates bin cross-correlation ⚠

Pixel splitting schemes available in pyFAI

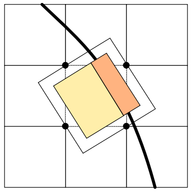
- **No pixel splitting: default histograms**
  - Each pixel contributes to a single bin of the result
  - No bin correlation but more noisy
  - The pixel has no surface: sharpest peaks



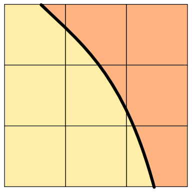
- **Bounding-box pixel splitting**
  - The smoothest integrated curve
  - Blurs a bit the signal



- **Pseudo pixel splitting**
  - 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:

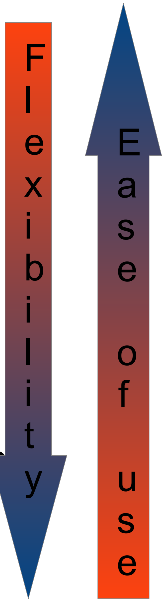
- compiled code like C, C++, Fortran, ...
- JIT compiled code like Java, Julia or numba

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 with OpenMP or OpenCL)



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

  It is up to you !

- In this tutorial, only applications in **bold** will be demonstrated

Project management

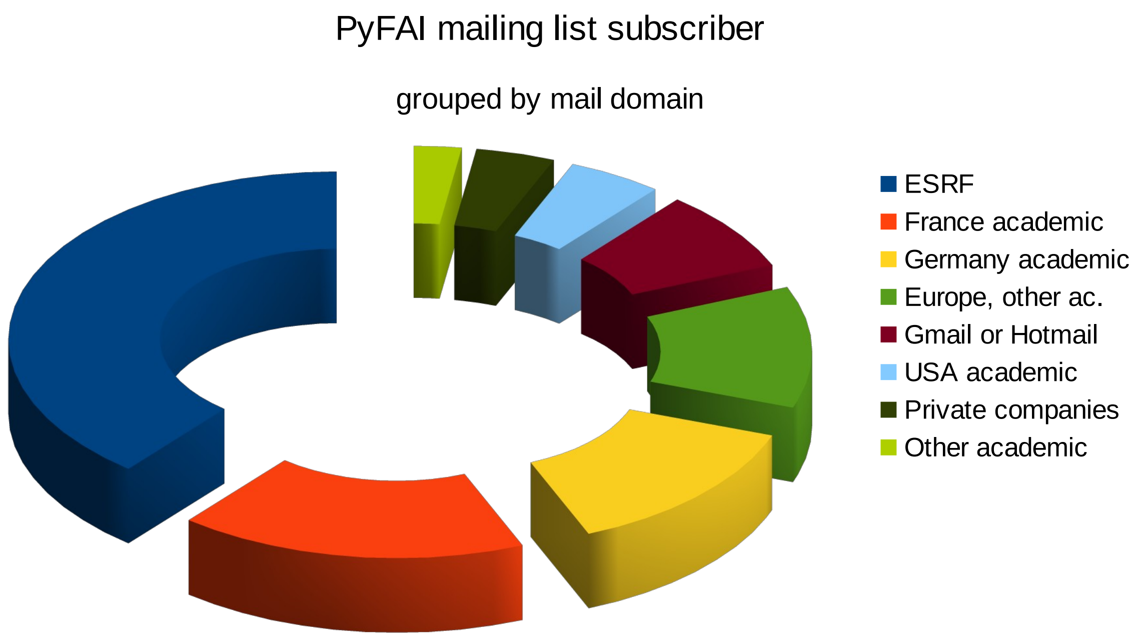
Silx & pyFAI

PyFAI is yet another azimuthal integration tool

- **Written in Python (compatible with ~~2.7, 3.4,~~ 3.5, 3.6, 3.7 & 3.8)**
  - 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 > 40 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**

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**
  - Archived on http://www.silx.org/lurker/list/pyfai.en.html
  - 137 people subscribed to the list (Jan 2020; 112 in 2018, 132 in 2019)
  - limited activity (~1 thread/month)

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

Reasons to chose pyFAI

- **Faster than others**
  - First tool using sparse matrix multiplication to perform integration
  - All computation intensive kernel 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**
  - It’s geometry is so generic it matches all configuration
    - **SAXS, WAXS …**
  - 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

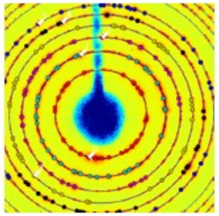
- silx on GitHub
- Wheels and source on PyPi
- Installation instructions

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

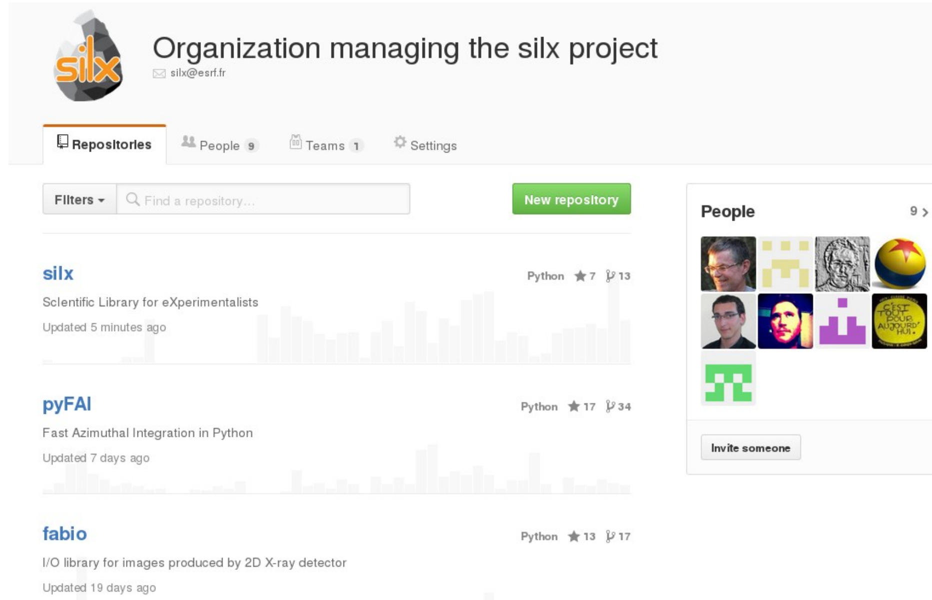
- Latest release
- Nightly build

silx-kit: Shared development around:

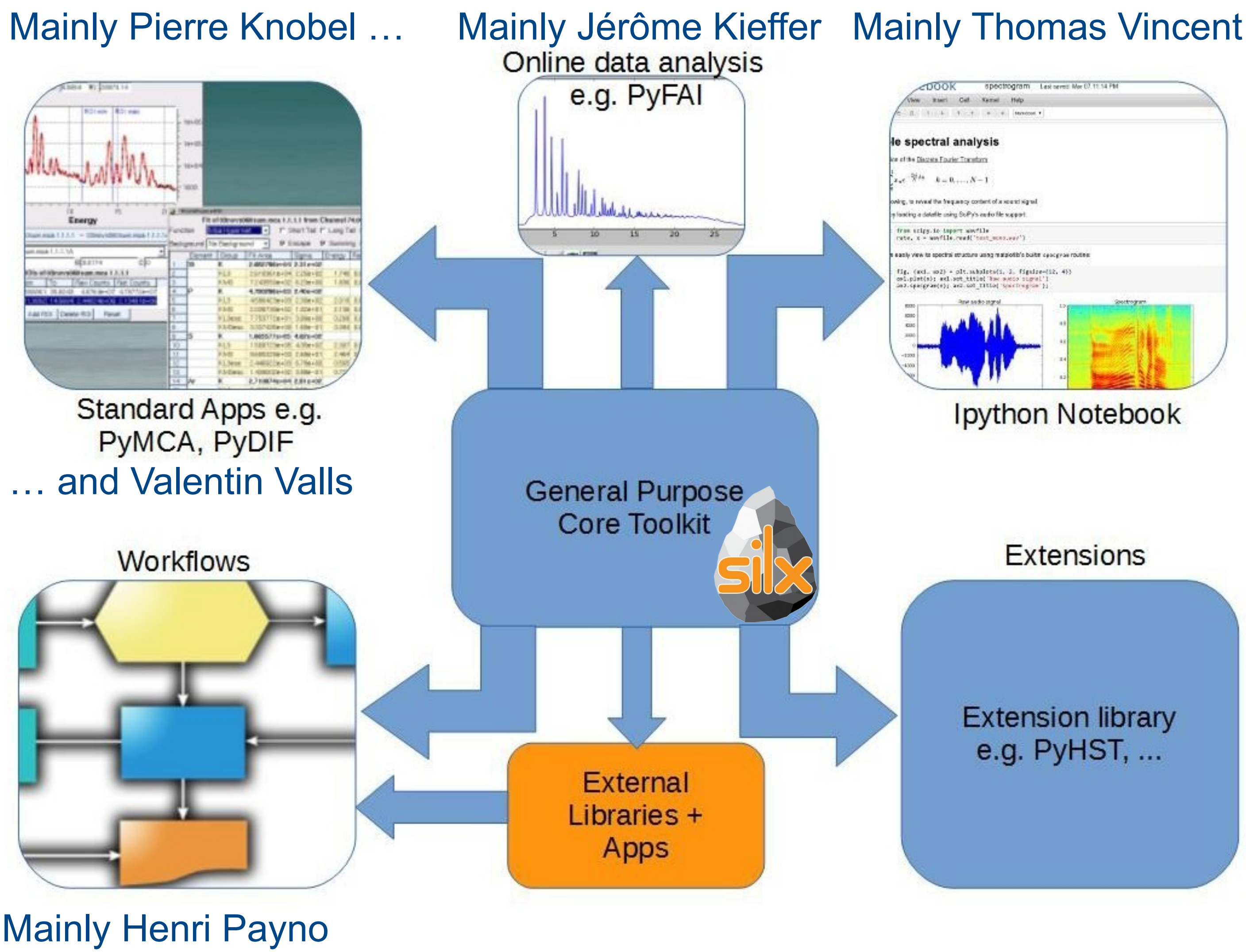
- **User interface**
  - Common interface to Qt
  - Common visualization widgets
- **GPU computing**
  - Common initialization
- **Scientific data analysis**
  - Multi-threaded implementation of core algorithms

Management of the *silx-kit* project

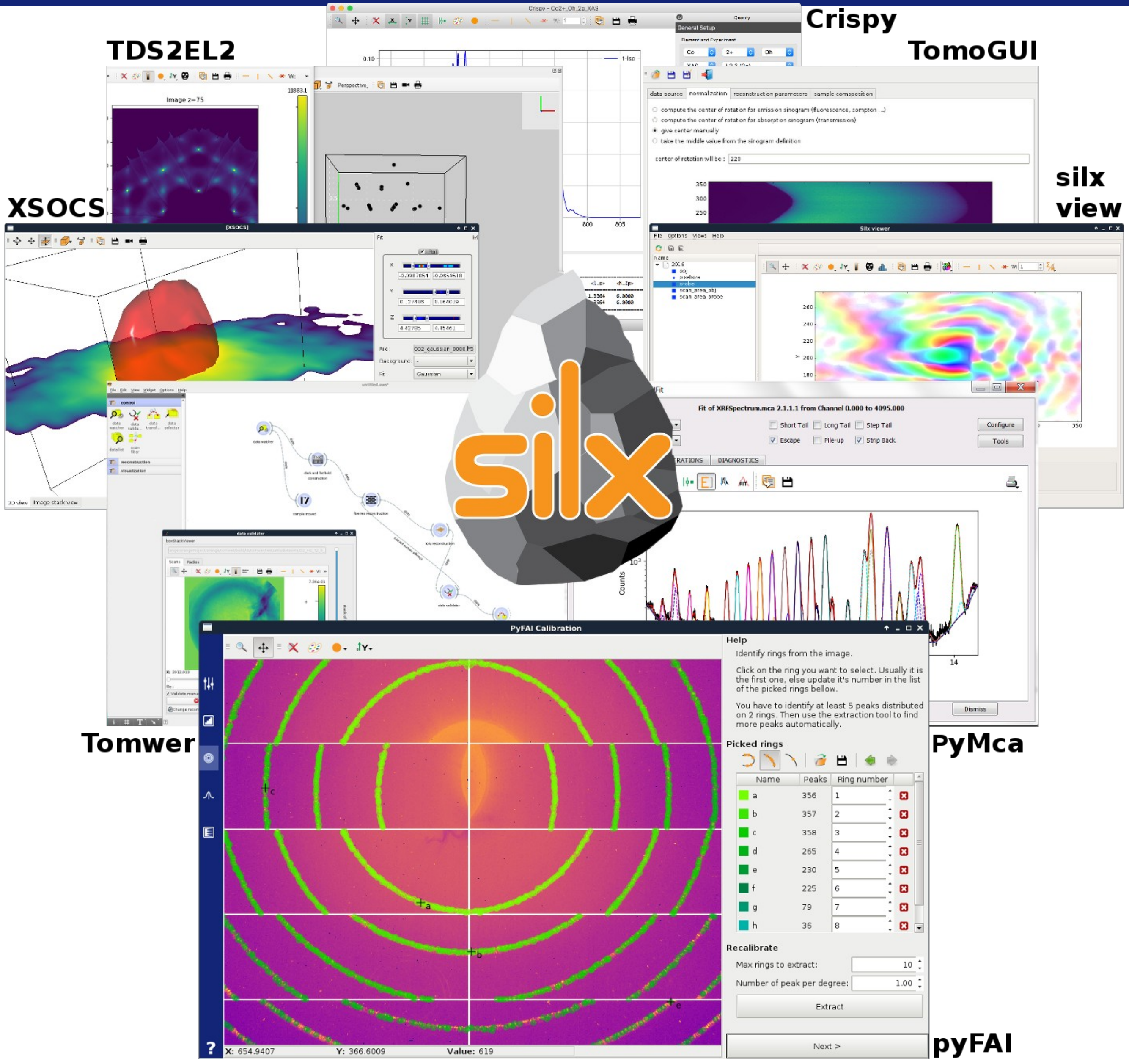
- **Public project hosted at github**
  - https://github.com/silx-kit/silx
- **Continuous testing**
  - Linux, Windows and macOS
- **Nightly builds**
  - Debian packages
- **Weekly meetings**
- **Quarterly releases**
- **Code camps before release**
- **Continuous documentation**
  - http://www.silx.org/doc/silx/



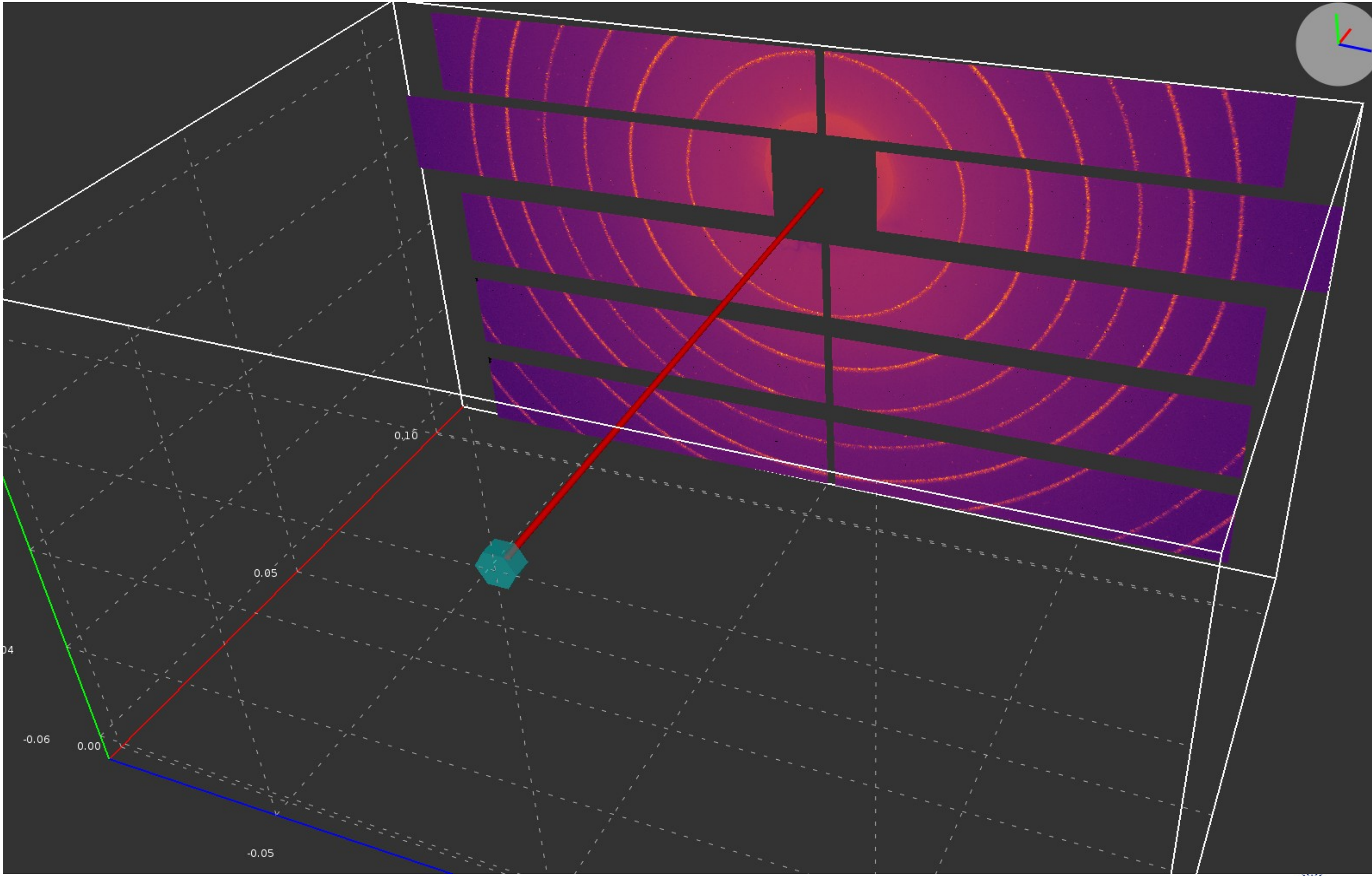
silx-kit project and the *silx* library



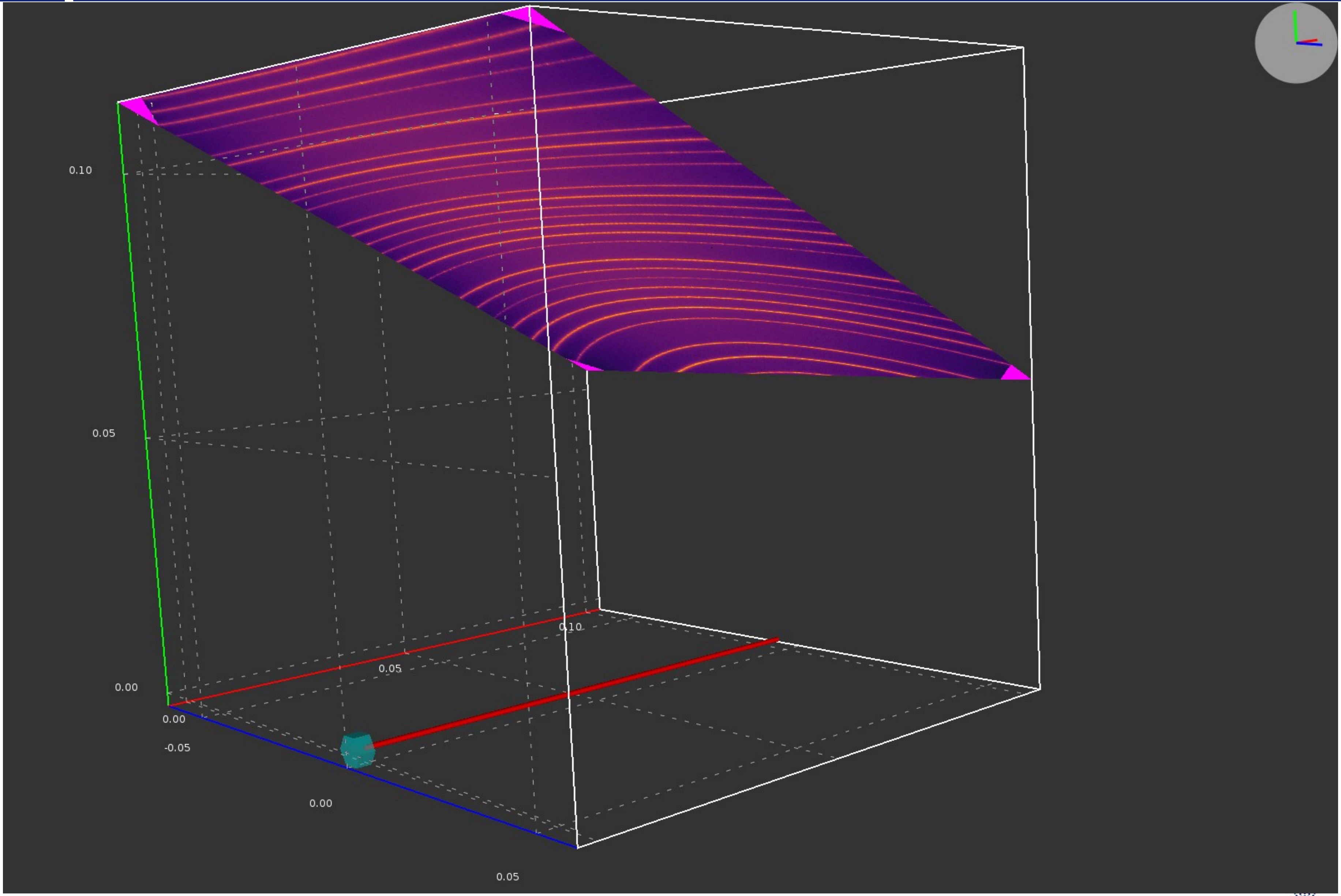
Outcome of the *silx* toolkit after 3 years:



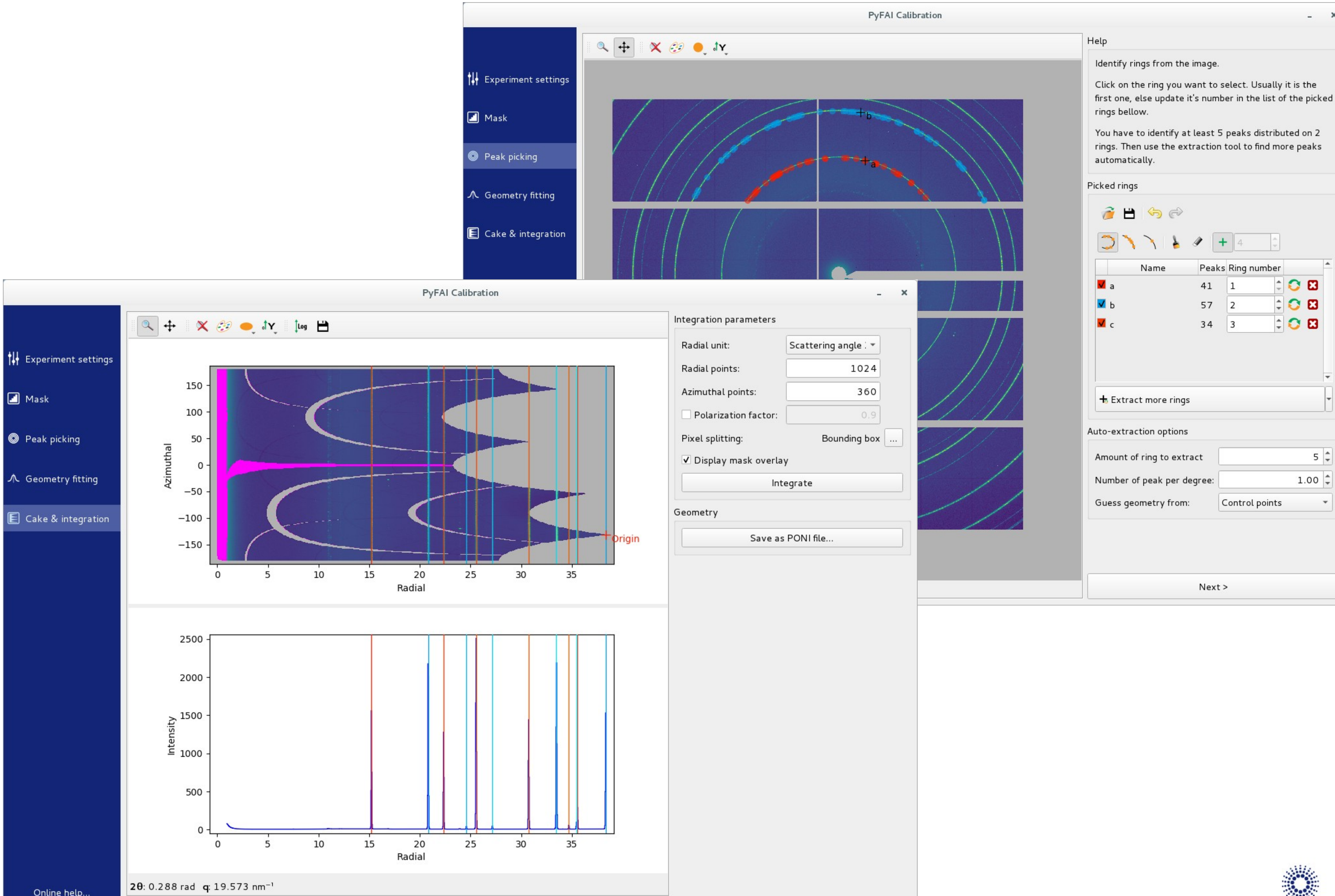
3D view of the diffraction setup



3D view of the diffraction setup



Calibration tools



Acknowledgments

- **Data analysis unit staff:**
  - Valentin Valls
  - 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
  - APS: Clemens Prescher
  - NSLS-II: scikit-beam project
  - ALS: Camera project
- **International Grants:**
  - LinkSCEEM-2 grant
    - **Dimitris Karkoulis**
    - **Giannis Ashiotis**
    - **Zubair Nawaz**

Questions ?

Installation procedure on MacOS

- **Download all data needed**
  - From `http://www.silx.org/pub/pyFAI/pyFAI_UM_2020.zip`
  - Unzip the content of this archive
- **Install Python3.7**
  - Double click on the `dmg` file found in the `macos` folder
  - Drag-and-drop to the `Applications` folder
- **Install pyFAI into a virtual environment**
  - `python3.7 -m venv pyfai`
  - `source pyfai/bin/activate`
  - `pip install -f macos/wheelhouse --pre --no-index pyFAI[gui]`
- **Run the application of your choice:**
  - `pyFAI-calib2`
  - `pyFAI-integrate`
  - `pyFAI-benchmark`

Installation procedure on Windows

- **Download all data needed**
  - From `http://www.silx.org/pub/pyFAI/pyFAI_UM_2020.zip`
  - Unzip the content of this archive
- **Install Python3.7**
  - Double click on the `exe` file found in the windows folder
  - Install `winpython` to the root of the archive
  - Launch the “`WinPython Command Prompt.exe"`
- **Install pyFAI and the missing dependencies**
  - `pip install -f windows\wheelhouse --pre --no-index pyFAI[gui]`
- **Run the application of your choice:**
  - `pyFAI-calib2`
  - `pyFAI-integrate`
  - `pyFAI-benchmark`

Installation procedure on Linux

- **Download all data needed**
  - From `http://www.silx.org/pub/pyFAI/pyFAI_UM_2020.zip`
  - Unzip the content of this archive
- **Install Python 3.x (x≥5) and create a virtual environment**
  - Follow the procedure of your distribution
  - `python3 -m venv pyfai`
  - `source pyfai/bin/activate`
- **Install pyFAI and the missing dependencies**
  - `pip install --pre pyFAI[gui]`
- **Run the application of your choice:**
  - `pyFAI-calib2`
  - `pyFAI-integrate`
  - `pyFAI-benchmark`