ESRF User Meeting 2024
PyFAI tutorial

Edgar Gutierrez Fernandez

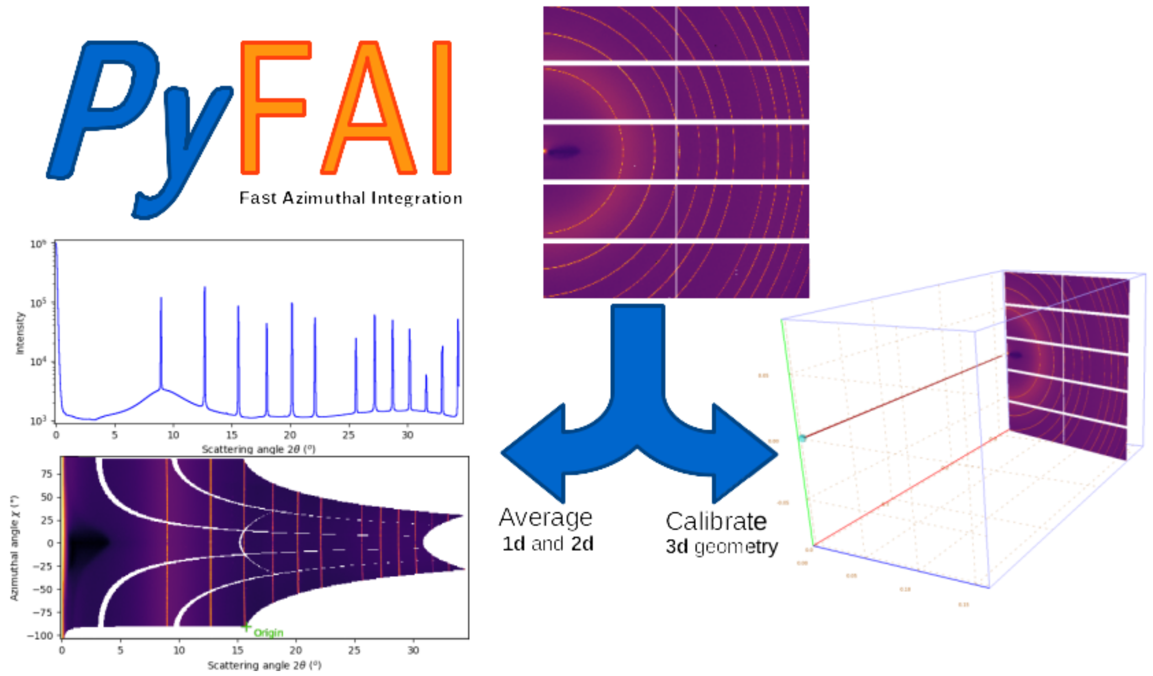
PIONEERING SYNCHROTRON SCIENCE

ESRF
The European Synchrotron

30 years of science

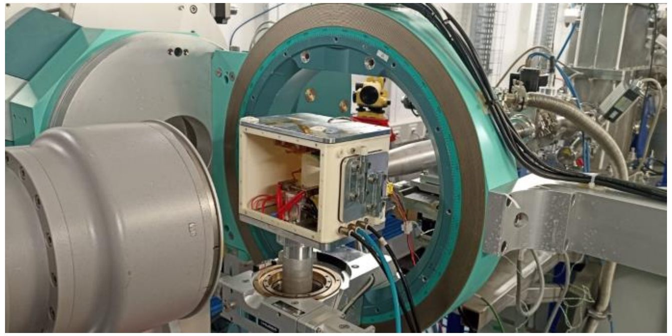
PyFAI tutorial - overview

- 05/02/2024 / 14:00 – 17:00
- Each user should use its own computer (Windows, Linux, MacOS)
- 1 half: concepts of PyFAI
  - Motivations
  - Applications
  - Working philosophy
- Coffe break (~15 minutes)
- 2 half: tutorial with jupyter notebook
  - Installation of python: venv/conda
  - Installation of pyFAI and dependencies
  - Calibration GUI
  - AzimuthalIntegration
  - Other pyFAI applications: integration/diff-map/waxs/saxs/worker…



PyFAI = Python Fast Azimuthal Integration

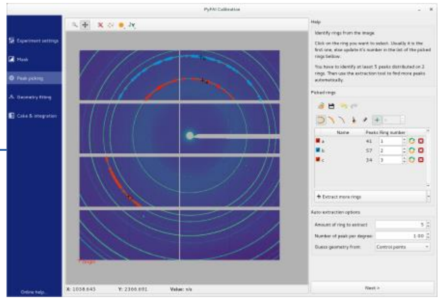
**BM28*



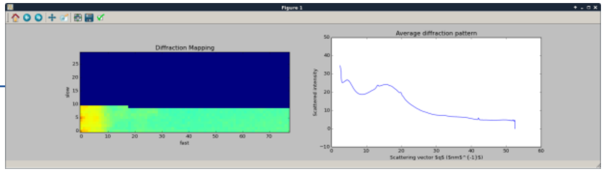
PyFAI
Fast Azimuthal Integration

Setup experiment → Data collection → Data reduction

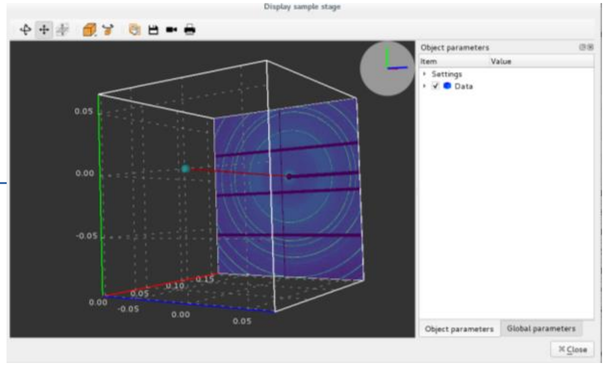
Calibration



Mapping

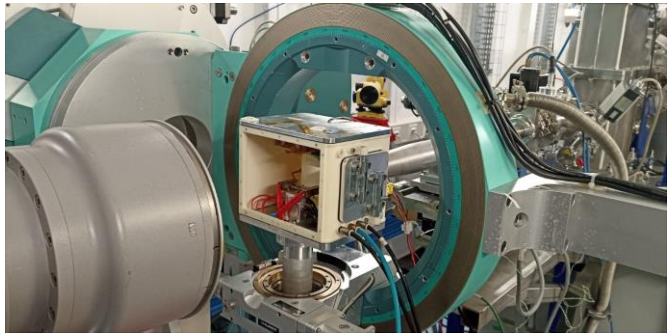


3D geometry

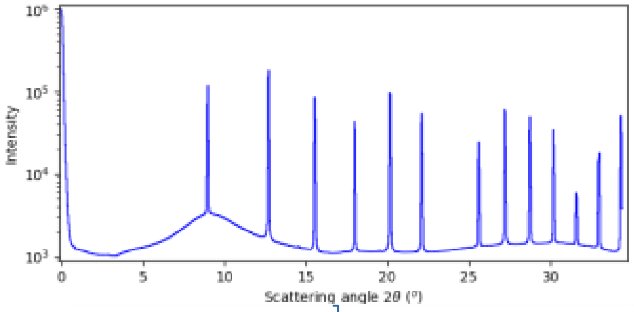


PyFAI = Python Fast Azimuthal Integration

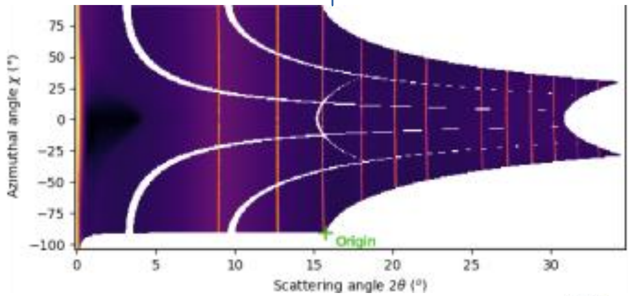
*BM28



PyFAI
Fast Azimuthal Integration

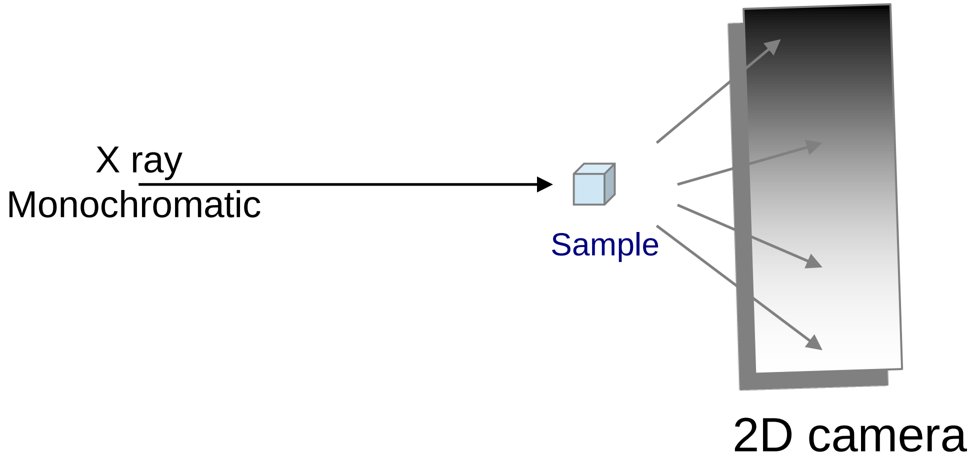


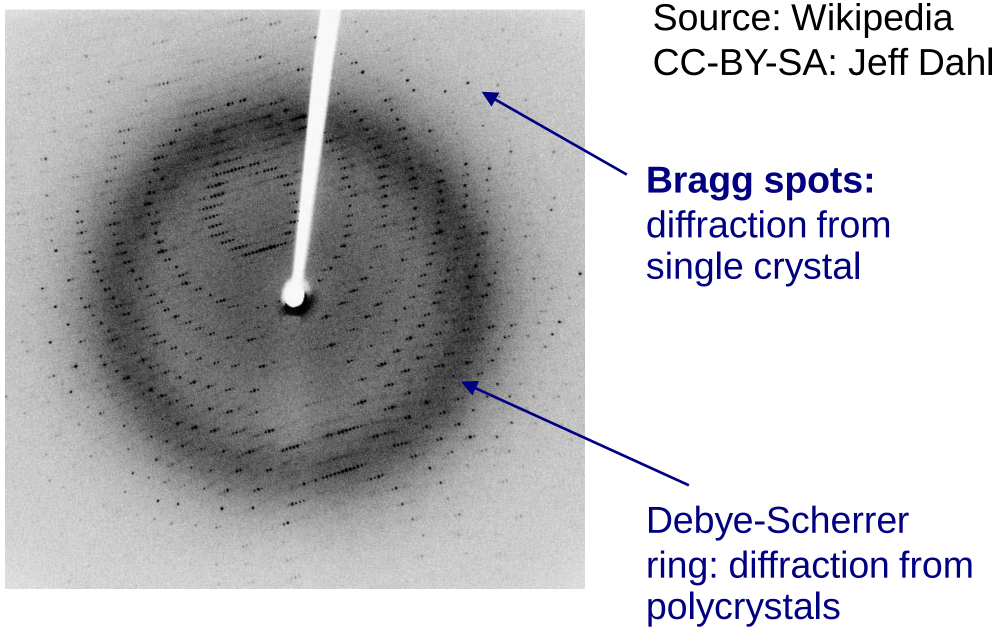
Setup experiment → Data collection → Data reduction → Data analysis



X-ray scattering techniques

- **Scattering** is the deflection (bending of the trajectory) of photons upon interaction with matter.

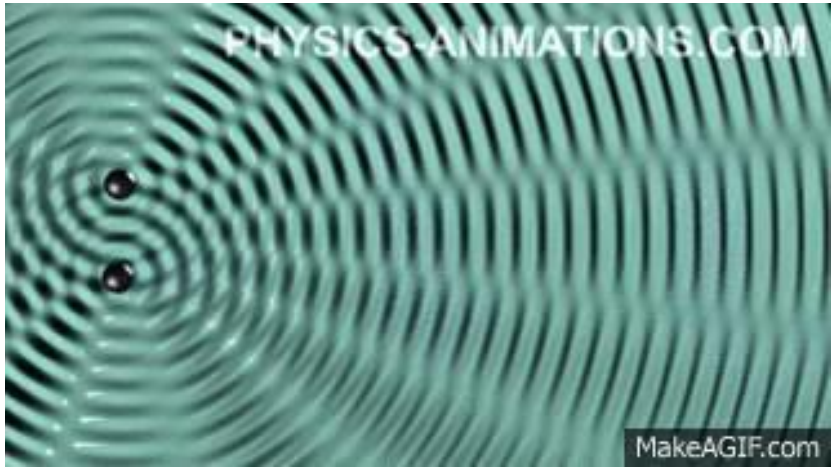




- **Monochromatic** beam.
- **2-dimensional detectors.**
- **Elastic** scattering (no change of the energy).

X-ray scattering techniques: diffraction/scattering

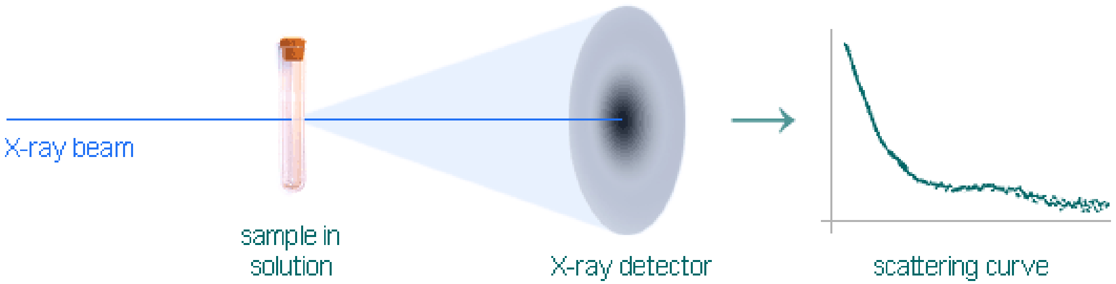
- If the material is **crystalline**, the scattered photons create interference, like water waves.



- Constructive interference between scattered X-rays takes place if **Bragg relation** is fulfilled:

$$n\lambda = 2d\,\sin\Theta$$

- If the material is **disordered**, the scattered photons create broad distributions of intensity.



https://biosaxs.com/technique.html

- Information from broad distributions is generally more ambiguous and harder to analyze (usually need complementary techniques).

X-ray scattering techniques: diffraction/scattering

- The study of highly **crystalline** materials (metals, ceramics, oxides…) is named '**diffraction**'.
- **Powder Diffraction**: isotropic.
  - Phase identification
  - Crystallinity
  - Lattice parameters
  - Thermal expansion
  - Phase transition
  - Strain/crystallite size

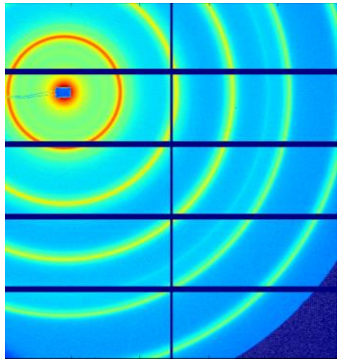
- The study of largely/inherently **disordered** materials (polymers, proteins, colloids…) is named '**scattering**'.
- Wide-Angle X-ray Scattering (**WAXS**): analog to diffraction:
  - Phase identification
  - Crystallinity/orientation
- Small-Angle X-ray Scattering (**SAXS**): micro/nano scale prove:
  - Particle shape/surface
  - Protein domains
  - Protein folding
  - Colloid parameters
  - Fiber orientation

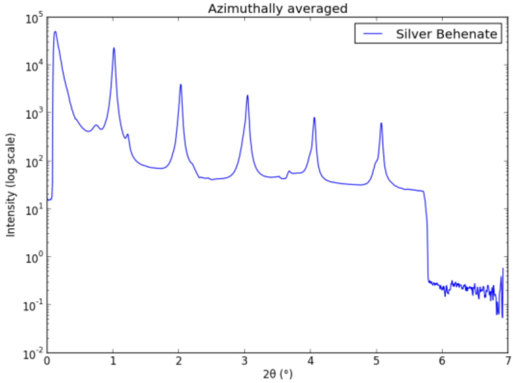
X-ray scattering techniques: diffraction/scattering

- The study of highly **crystalline** materials (metals, ceramics, oxides…) is named '**diffraction**'.
- **Powder Diffraction**: isotropic.

- The study of largely/inherently **disordered** materials (polymers, proteins, colloids…) is named '**scattering**'.
- **WAXS**: analog to diffraction.
- **SAXS**: micro/nano scale prove.

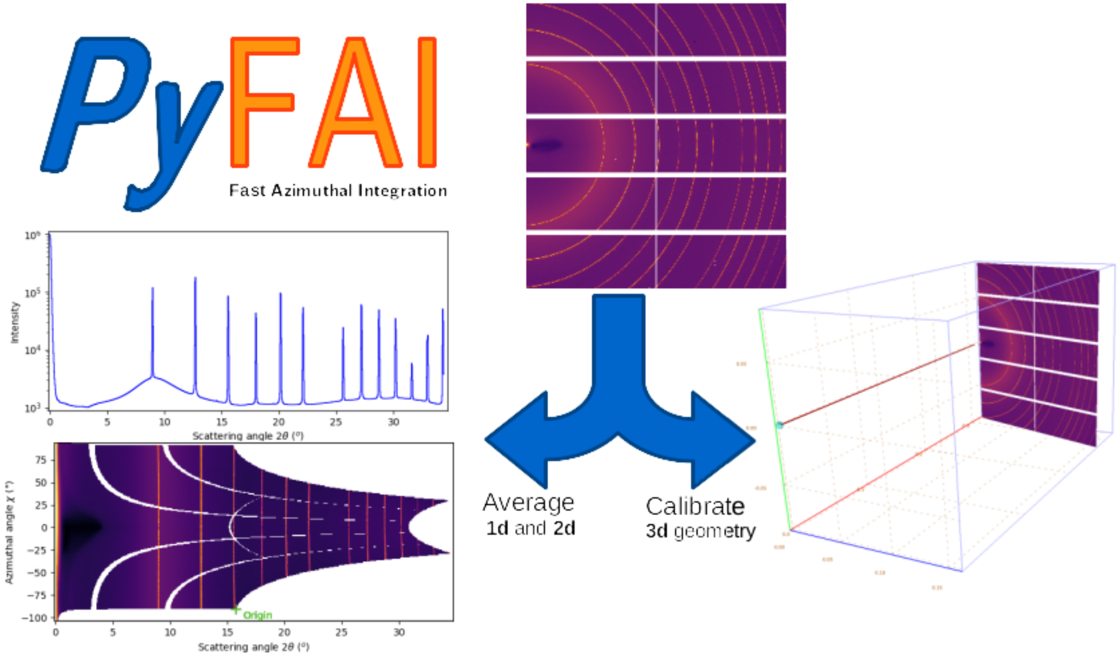
- Both rely on the same transformation: **2D image to azimuthal average.**





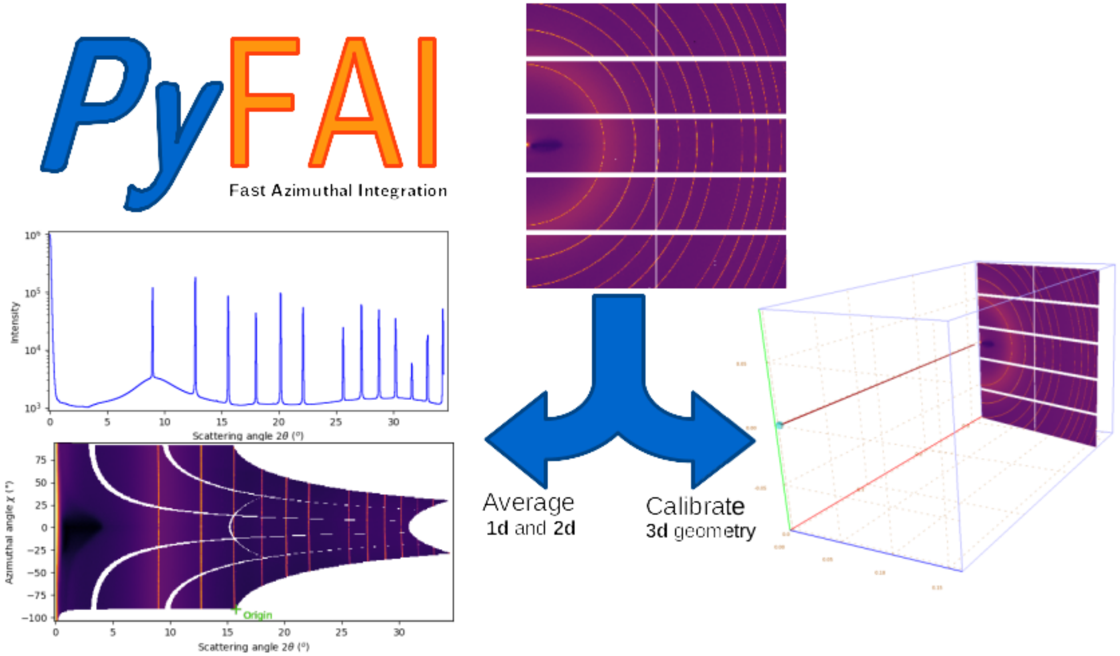
- **PyFAI is the first tool to be used after data collection.**

PyFAI = The reference pythonic tool to integrate 2D patterns



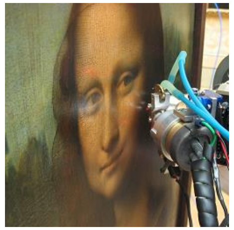
Why PyFAI?

PyFAI = The reference pythonic tool to integrate 2D patterns



- **Python** is considered the most accessible and widespread language in science.

- It's the main language, used and developed, at the ESRF:
  - Data acquisition (**BLISS**)
  - Data visualization (**silx**)
  - Data analysis (**PyMCA**)



- PyFAI combines python API with fast algorithms **written in C**:

Alternatives to pyFAI

- Fit2D
  - MIT licensed from ESRF, written in Fortran, now discontinued
- XRDUA
  - GPL licensed from U. Antwerp, written in IDL, focuses on diffraction mapping
- DAWN
  - EPL licensed from Diamond Light Source, written in Java
- DataSqueeze
  - Freeware from U. Pennsylvania
- Foxtrot
  - Commercial, from XENOCS & SOLEIL synchrotron, written in 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, written in Python

Concepts (class instances) in pyFAI

- **Detector**
  - Calculates the pixel position: center and corners.
  - Calculates and stores the mask of invalid pixels.
  - Registered detectors / Custom defined.

- **Image**
  - Numpy 2D array
  - Read using FabIO, silx, h5py…

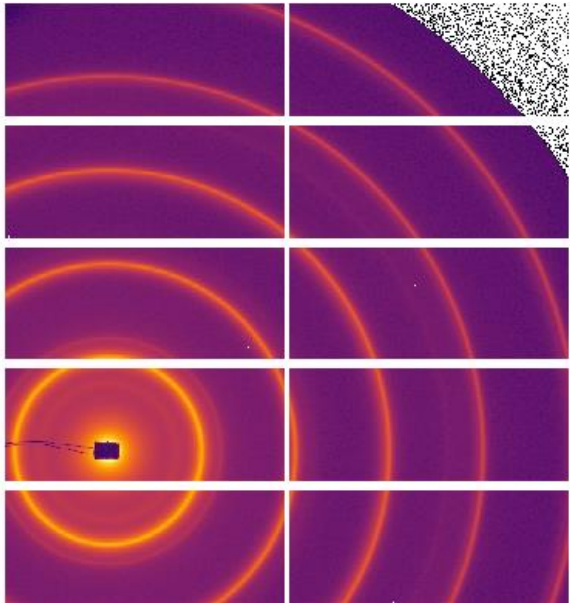
- **Geometry**
  - Position of the detector from the sample and incoming beam.
  - Full characterization of the beam-sample-detector setup.

- **Azimuthal Integrator**
  - Contains the methods to integrate the image.
  - integrate1d: average a double-arc section (cake) of the pattern into an intensity profile.
  - integrate2d: reshaping of the image into meaningful units (angle or momentum transfer).

Concepts (class instances) in pyFAI

- ## Detector
  - Calculates the pixel position: center and corners.
  - Calculates and stores the mask of invalid pixels.
  - Registered detectors / Custom defined.

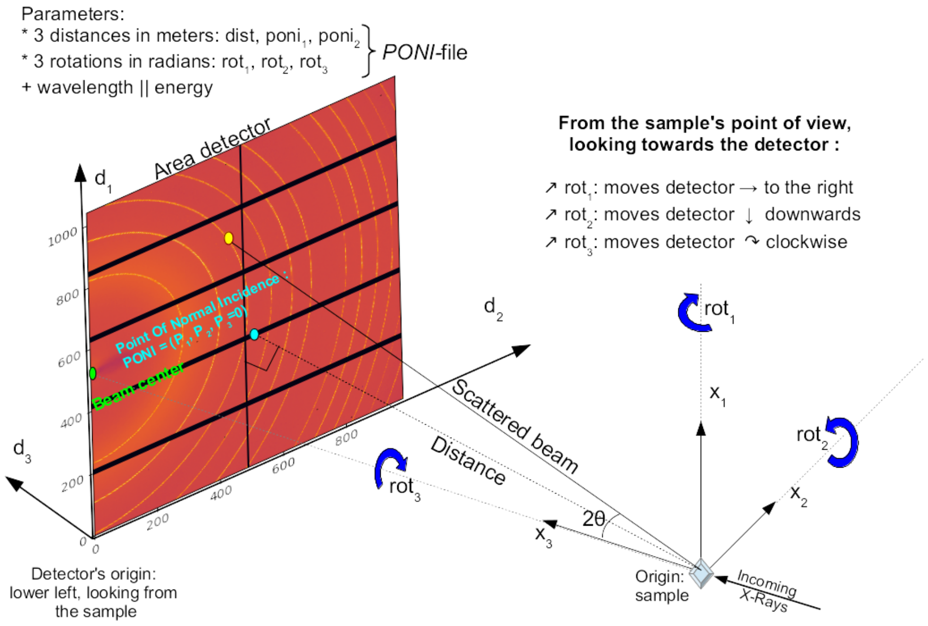
- ## Image
  - Numpy 2D array
  - Read using FabIO, silx, h5py…

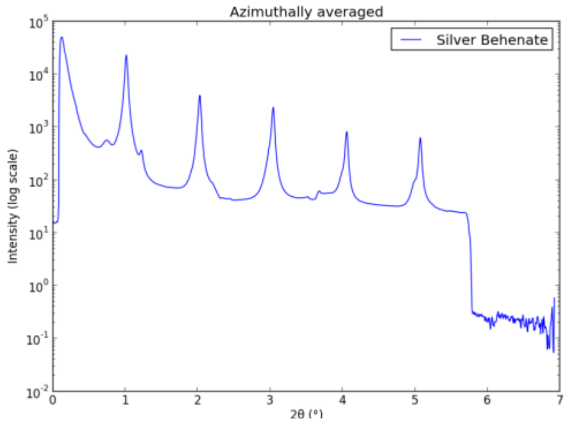


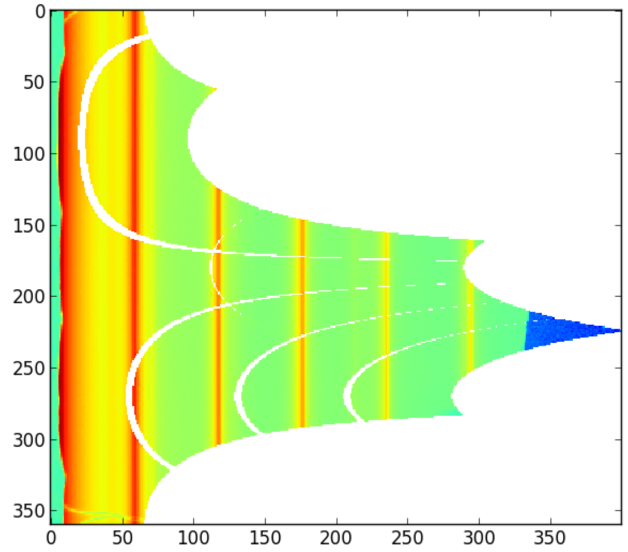
Concepts (class instances) in pyFAI

- ## Geometry
  - Position of the detector from the sample and incoming beam.
  - Full characterization of the beam-sample-detector setup.

- ## Azimuthal Integrator
  - Contains the methods to integrate the image.
  - integrate1d: average a double-arc section (cake) of the pattern into an intensity profile.
  - integrate2d: reshaping of the image into meaningful units (angle or momentum transfer).







Detector instance

- Import detector from **detector_factory** (e.g. Pilatus1M from Dectris)

```
[1]: from pyFAI import detector_factory
     p1m = detector_factory("pilatus1m")
```

```
[2]: p1m
```

```
[2]: Detector Pilatus 1M      PixelSize= 1.720e-04, 1.720e-04 m       BottomRight (3)
```

```
[3]: print(f"""
     The detector {p1m.name} has a shape of {p1m.shape}
     The size of the pixels is {p1m.pixel1*1e6} microns x {p1m.pixel2*1e6} microns
     """)
```

```
The detector Pilatus 1M has a shape of (1043, 981)
The size of the pixels is 172.0 microns x 172.0 microns
```

```
[4]: p1m.set_binning((2,2))
     print(f"""
     With a binning of {p1m.binning}:
     The detector {p1m.name} has a shape of {p1m.shape}
     The size of the pixels is {p1m.pixel1*1e6} microns x {p1m.pixel2*1e6} microns
     """)
```

```
With a binning of (2, 2):
The detector Pilatus 1M has a shape of (521, 490)
The size of the pixels is 344.0 microns x 344.0 microns
```

Detector instance

- Import detector from **detector_factory** (e.g. Pilatus1M from Dectris)

- **Customized** detector

```
[5]: det = detector_factory(
         name="Detector",
         config={
             "max_shape" : (2000,2000),
             "pixel1" : 50e-6,
             "pixel2" : 50e-6,
             "orientation" : 3,
     })
```

```
[6]: print(f"""
     The detector {det.name} has a shape of {det.shape}
     The size of the pixels is {det.pixel1*1e6} microns x {det.pixel2*1e6} microns
     """)
```

```
The detector Detector has a shape of (2000, 2000)
The size of the pixels is 50.0 microns x 50.0 microns
```

Detector instance

- Import detector from **detector_factory** (e.g. Pilatus1M from Dectris)

- **Customized** detector

```
[5]: det = detector_factory(
         name="Detector",
         config={
             "max_shape" : (2000,2000),
             "pixel1" : 50e-6,
             "pixel2" : 50e-6,
             "orientation" : 3,
     })
```

```
[6]: print(f"""
     The detector {det.name} has a shape of {det.shape}
     The size of the pixels is {det.pixel1*1e6} microns x {det.pixel2*1e6} microns
     """)
```

```
The detector Detector has a shape of (2000, 2000)
The size of the pixels is 50.0 microns x 50.0 microns
```

- Import during calibration and included in **.poni file** (most common approach)

Image instance (2D array)

- Imported through different modules: **FabIO, silx, h5py**
- Common file formats: **.edf, .tiff, .cbf, .h5**
- Example: single frame **.edf file**

```
[30]: import fabio
      import matplotlib.pyplot as plt

[31]: img = fabio.open('LaB6_r00005_n0125_p031.edf')

[32]: data = img.data

[35]: data.shape

[35]: (2048, 2048)
```

```
[34]: plt.imshow(data, vmin=0, vmax=1e2)
      plt.show()
```

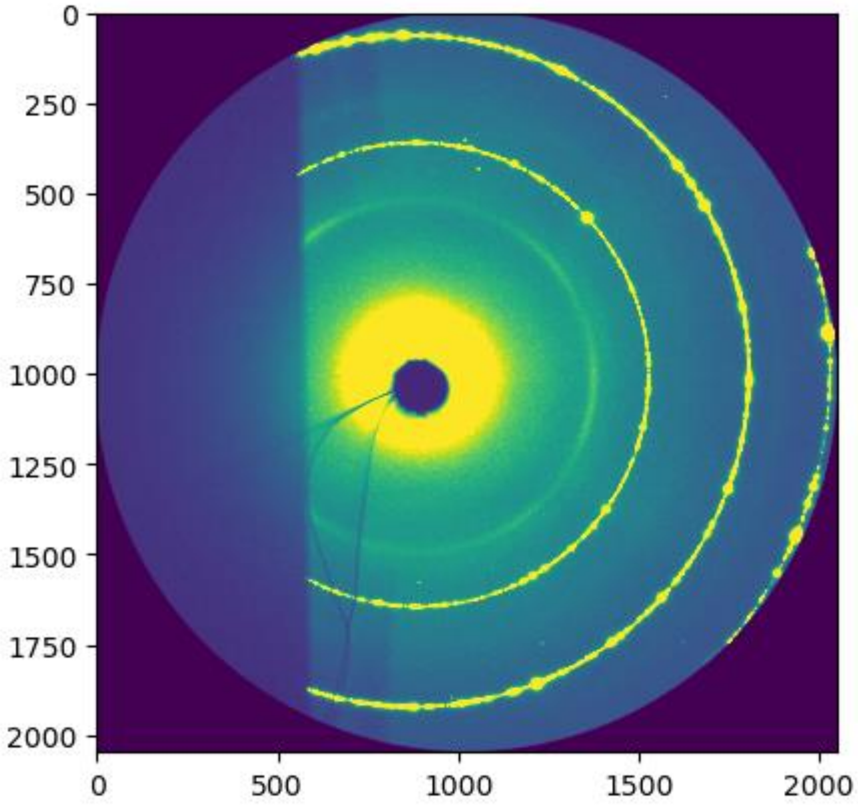


Image instance (2D array)

- Imported through different modules: **FabIO, silx, h5py**
- Common file formats: **.edf, .tiff, .cbf, .h5**
- Example: multiple frame **.h5 file (scan)**

```
[13]: import fabio
      import matplotlib.pyplot as plt

[14]: im_series = fabio.open('rayonix_0000.h5')

[15]: im_series.nframes

[15]: 10

[16]: img = im_series.get_frame(0)

[17]: data = img.data

[18]: data.shape

[18]: (1920, 1920)
```

```
[29]: plt.imshow(data, vmin=0, vmax=1e3)
      plt.show()
```

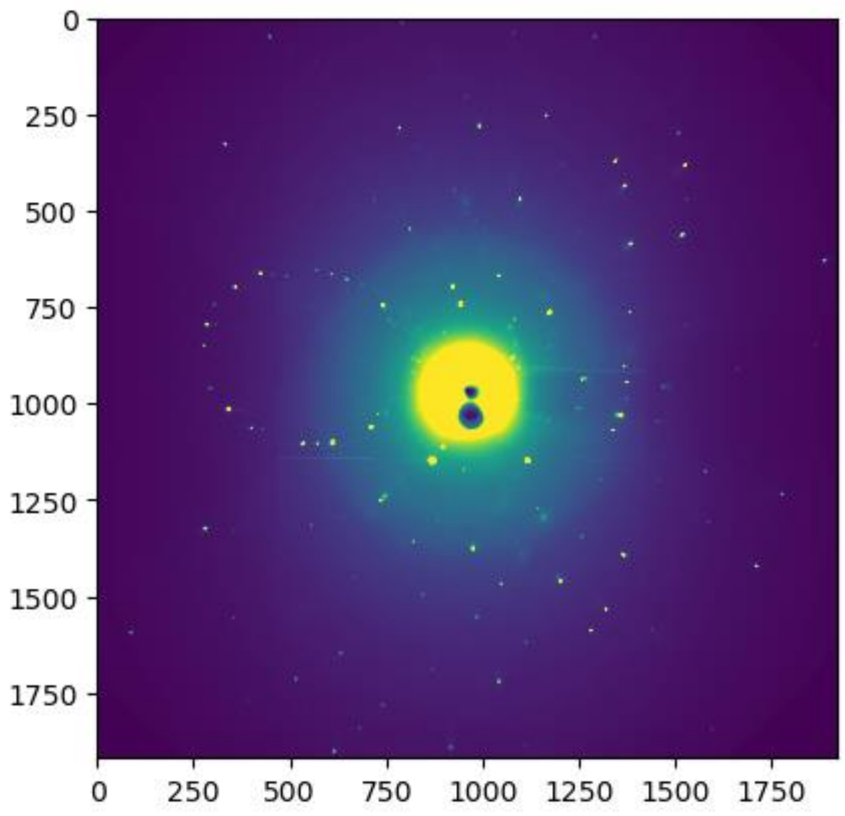


Image instance (2D array)

- Imported through different modules: **FabIO, silx, h5py**
- Common file formats: **.edf, .tiff, .cbf, .h5**
- Example: multiple frame **.h5 file (scan)**

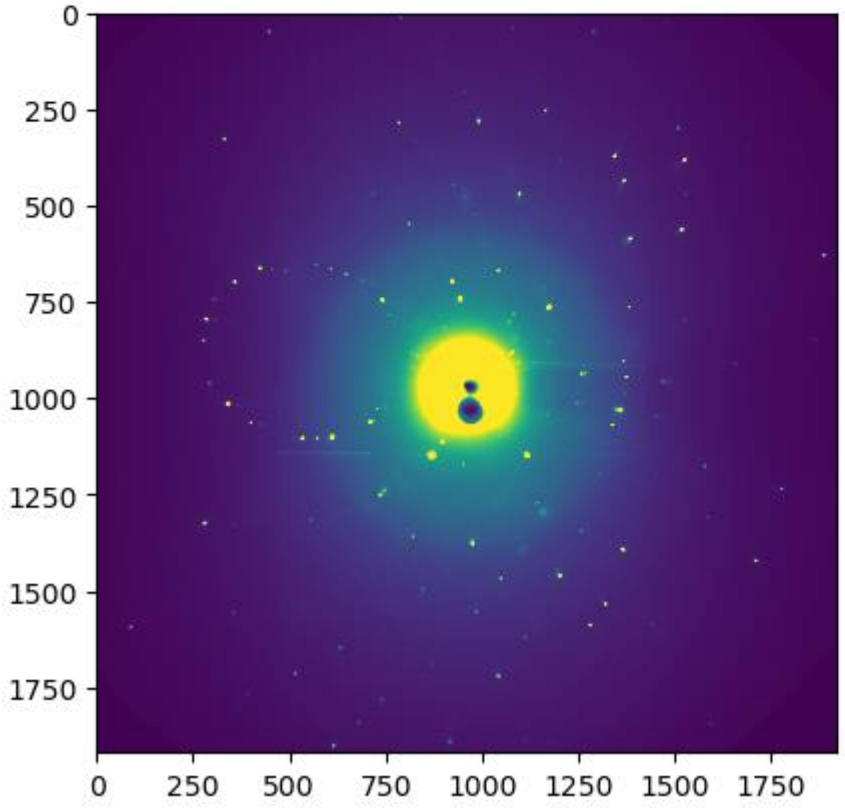
```
[20]: import h5py
```

```
[24]: with h5py.File('rayonix_0000.h5') as f:
          data_set = f['entry_0000']['measurement']['data'][()]
```

```
[26]: data_set.shape
```

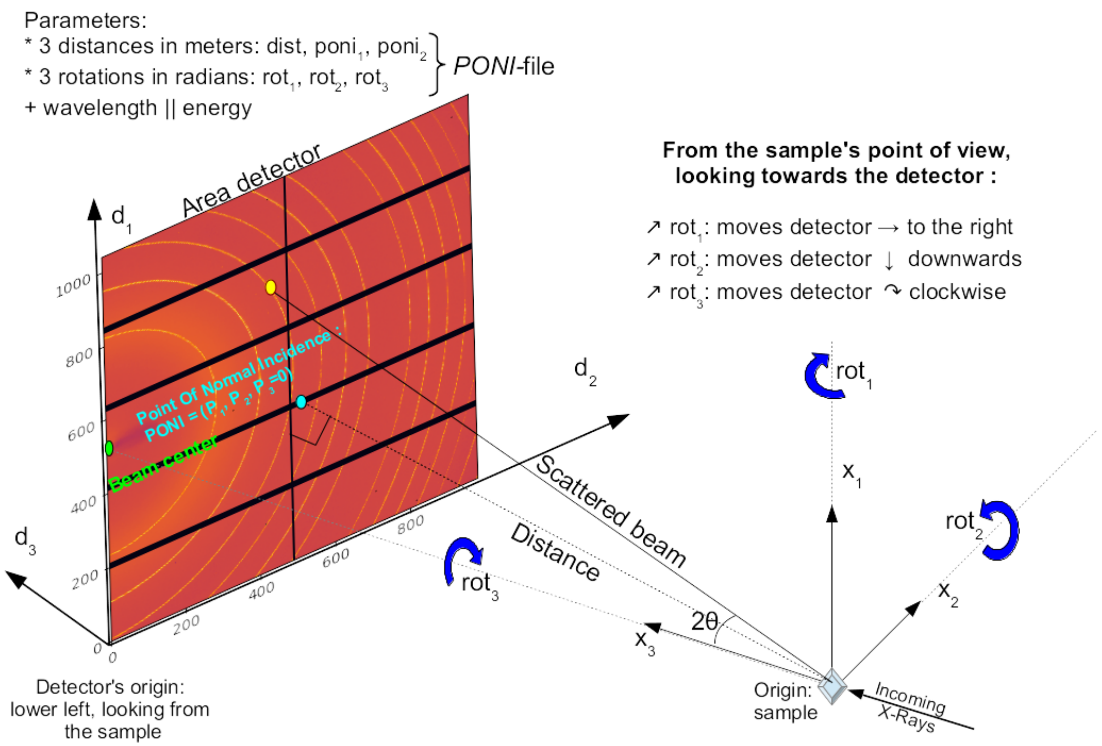
```
[26]: (10, 1920, 1920)
```

```
[37]: data = data_set[0,:,:]
      plt.imshow(data, vmin=0, vmax=1e3)
      plt.show()
```



Geometry instance

- A geometry is fully defined by:
  - **Detector** instance (+ orientation of detector)
  - **Sample-to-detector distance** (in meters)
  - **Wavelength** of the beam (in meters)
  - **Three rotations of the detector**
  - Coordinates of the **point-of-normal-incidence** (PONI), from the sample to the detector plane



- The geometry instance could be initialized **manually**.
- The common approach is through the **calibration GUI of pyFAI**. The user has to provide:
  - **Calibration file (+choosing a calibrant)**
  - **Detector**
  - **Wavelength** of the beam

Calibration of the setup: pyFAI-calib2

- Calibration is made after measuring a **standard sample**:
  - LaB_6, Cr_2O_3, AgBh…
- Choosing the correct **detector** (+ orientation, binning, mask…)
- Selecting the **Debye-Scherrer rings** associated to the standard.
- **Fitting** the rings, refinement, validation.
- **Saving of .poni file.**
  - The .poni file is valid as long as the setup does not change (beam energy, detector movements, distance shifting).

Azimuthal integrator instance

- After calibration, the *.poni* file is enough to create an instance of the **azimuthal** integrator.

```
[10]: from pyFAI import load

[11]: ai = load('detx_50mm_2.poni')

[12]: ai

[12]: Detector Rayonix MX170   PixelSize= 8.854e-05, 8.854e-05 m
      Wavelength= 7.306081e-11 m
      SampleDetDist= 4.928280e-02 m   PONI= 8.602623e-02, 8.526832e-02 m      rot1=0.000000  rot2=0.000000  rot3=0.000000 rad
      DirectBeamDist= 49.283 mm       Center: x=963.030, y=971.590 pix       Tilt= 0.000° tiltPlanRotation= 0.000° λ= 0.731Å
```

- The azimuthal integrator instance contains both integrate1d and integrate2d functions to average the patterns.
  - integrate1d needs the data array and the size of the outcoming vector (radial bins)
  - integrate2d needs the data array and the radial bins (by default, it uses 360 azimuthal bins)

What happens during an integration

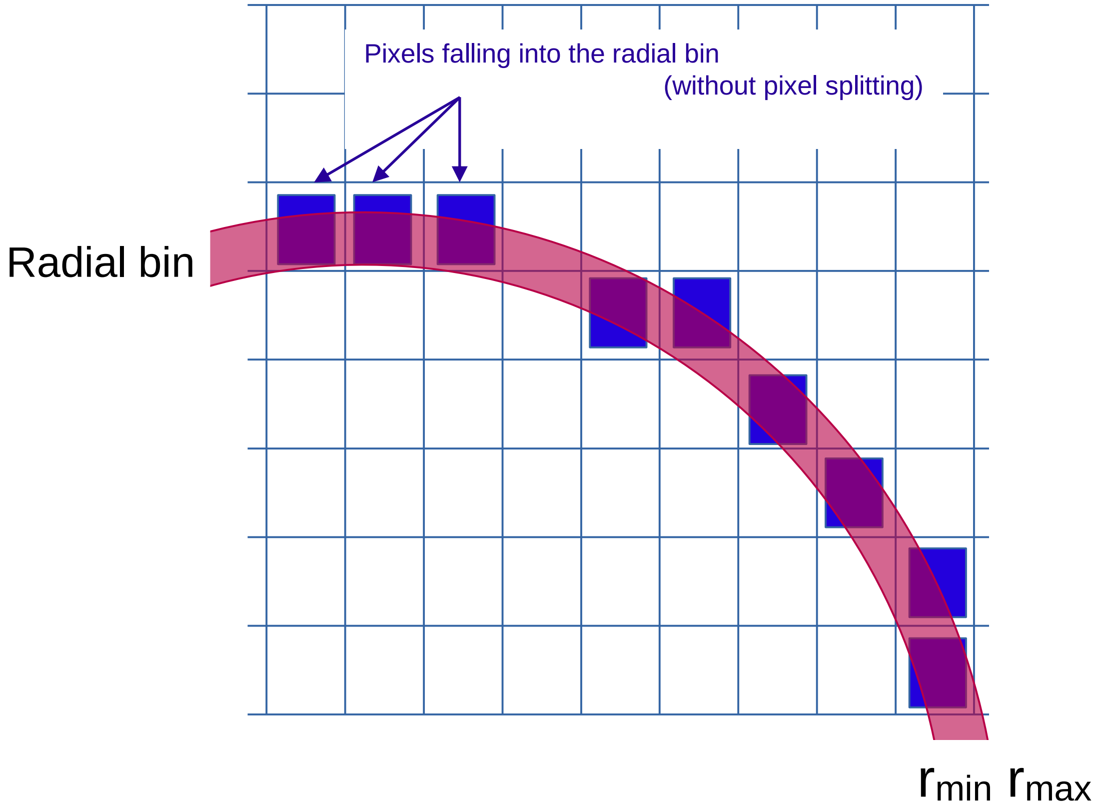
- 1) Get the coordinates of every **corner of every pixel of the detector** (in meters).
  - 3 coordinates per corner, 4 corners per pixel
  - Detector of 1000x1000 = 10^6 pixels = 1Mpix * 4 (corners) * 3 (coordinates) * 4 (bytes) = 48 Mbytes
- 2) Offset the **detector's origin** to the PONI and **rotate** around the sample.
- 3) Calculate the **radial (2theta) and azimuthal (chi) positions of each corner.**
- 4) Calculate **normalization** matrix: polarization, solid-angle, flat-field…
- 5) Assign each pixel to one or multiple **bins**.
- 6) Histogram bin position with **associated intensities**
- 7) Histogram bin position with **associated normalizations**
- 8) Return bin position and the ratio of **sum of intensities / sum of norm.**

Pixel-wise corrections

- Pixel-wise corrections:

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

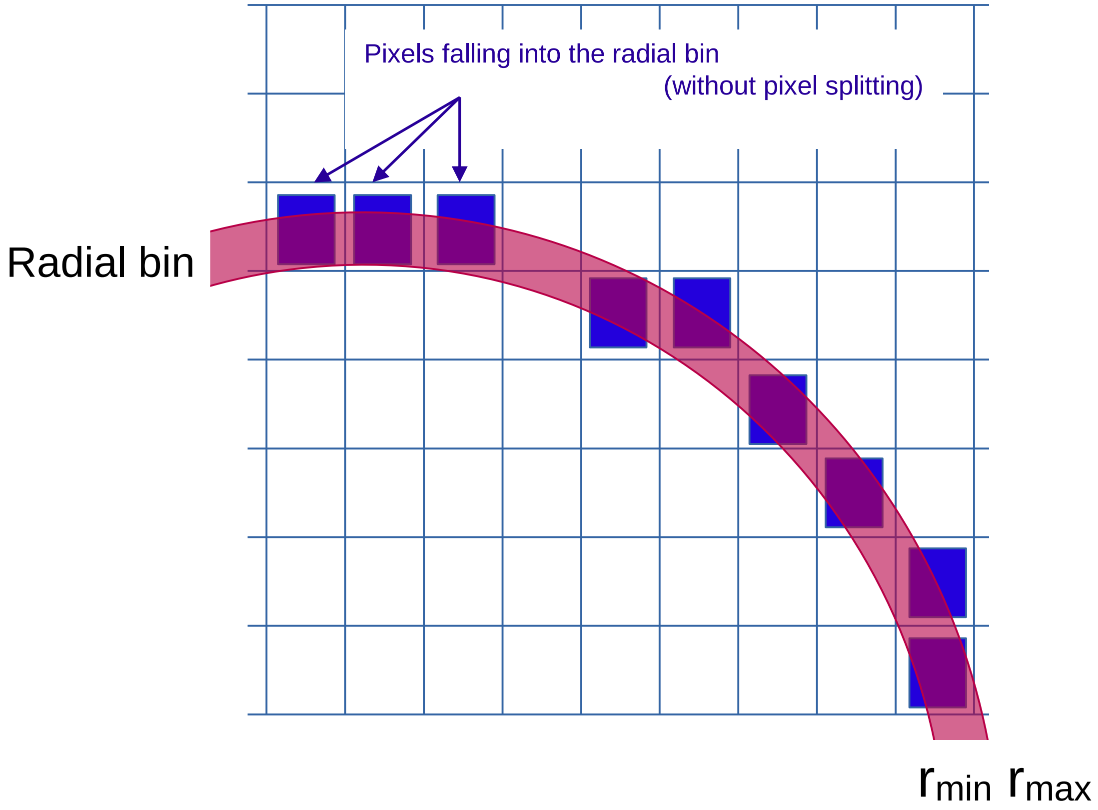
  - Dark current: electronic signal
  - Flat field: pixel efficiency under different illumination
  - Polarization: angle-related scattering
  - Parallax: peak shifting under oblique angle
  - Flux: drifting of beam intensity



- Pixel-wise corrections:

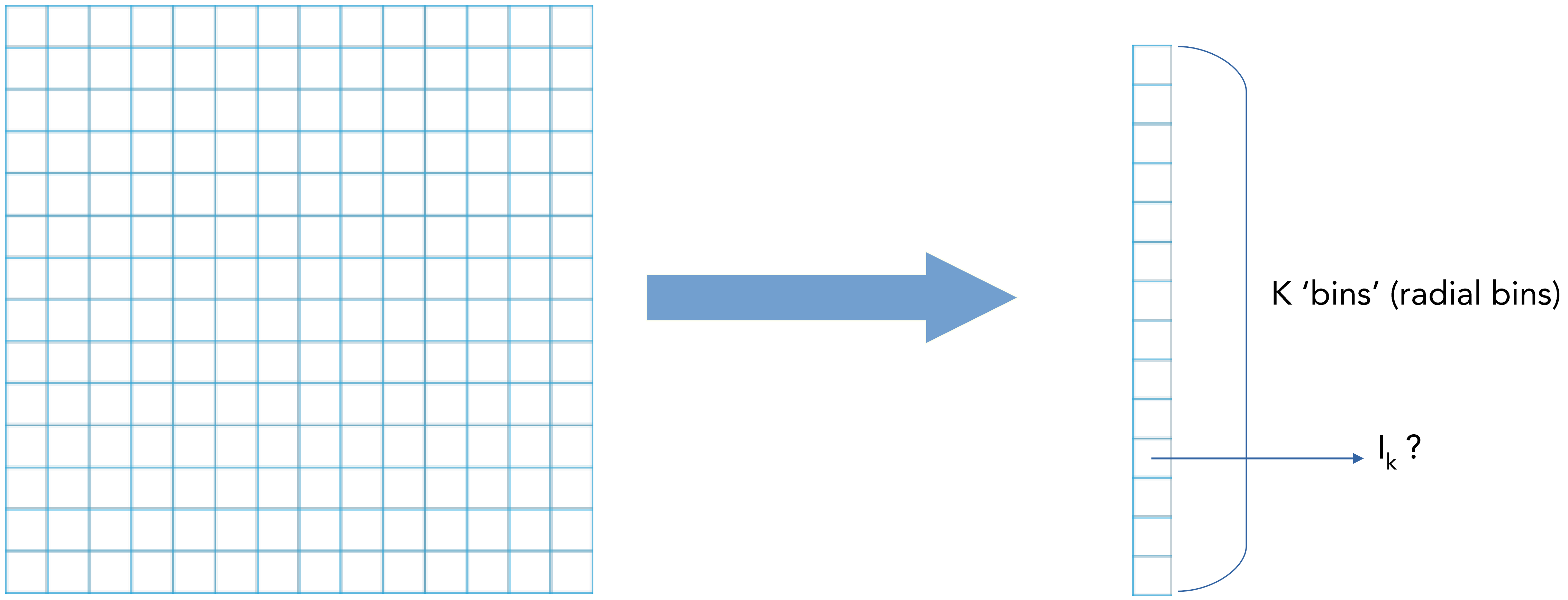
$$I_{cor} = \frac{I_{raw} - I_{dark}}{F \cdot \Omega \cdot P \cdot A \cdot I_0} = \frac{signal}{normalization}$$

- Averaging over a bin defined by the radius r:
  - Need pixel splitting?
  - Calculates ci: fraction of pixel intensity (i) associated to bin (r).
- Associated uncertainty propagation:
  - Assuming there is no correlation between pixels.
  - Pixel splitting can create correlations between bins.



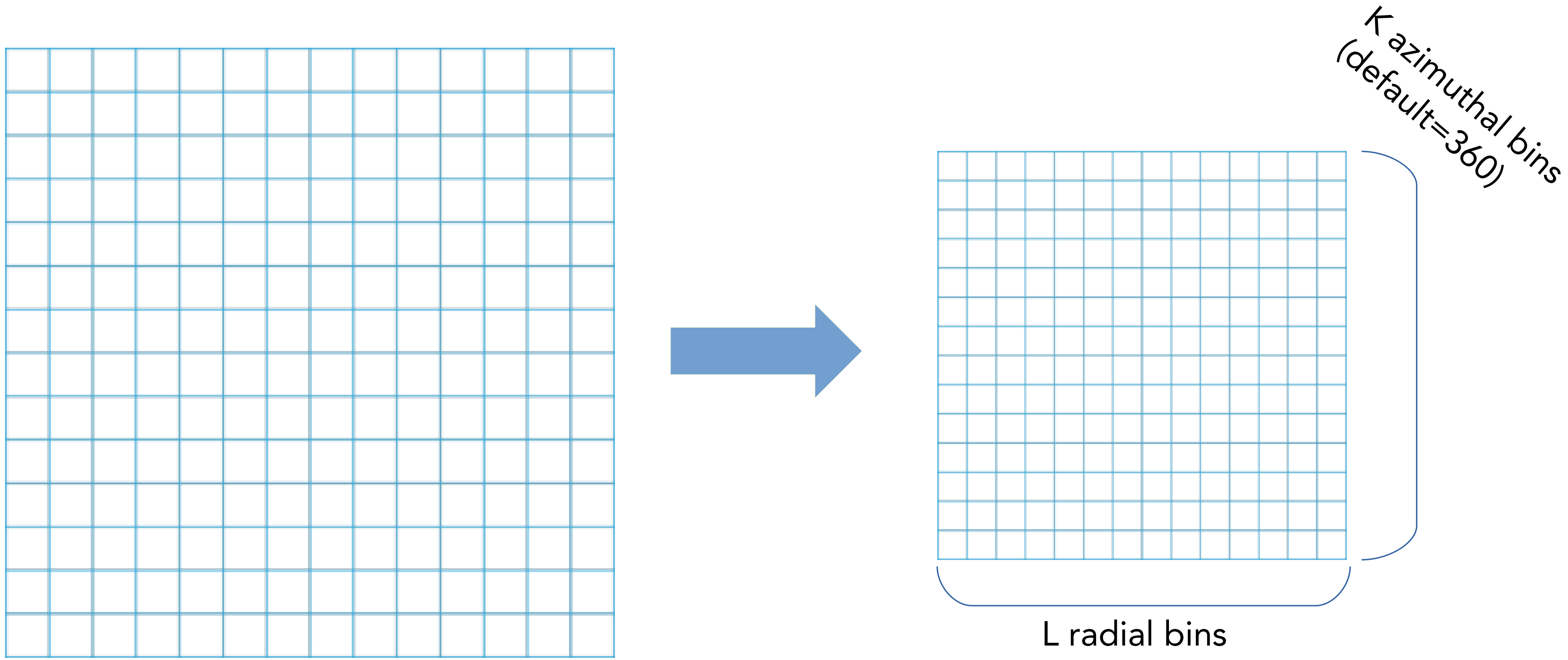
Integration is averaging + rearranging

- 2D array: m x n elements
- After integrate1d: K x 1 elements



Integration is averaging + rearranging

- 2D array: m x n elements
- After integrate2d: K x L elements

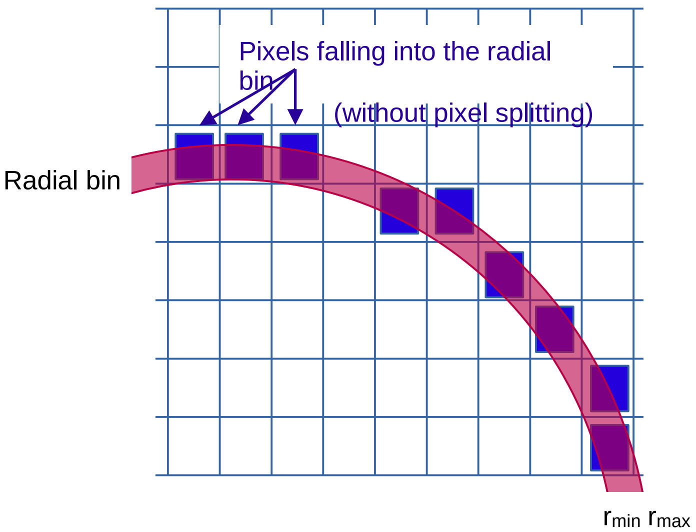


Pixel-wise corrections

- Pixel-wise corrections:

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

- Averaging over a bin defined by the radius r:
  - Need pixel splitting?
  - Calculates ci: fraction of pixel intensity (i) associated to bin (r).
- Associated uncertainty propagation:
  - Assuming there is no correlation between pixels.
  - Pixel splitting can create correlations between bins.



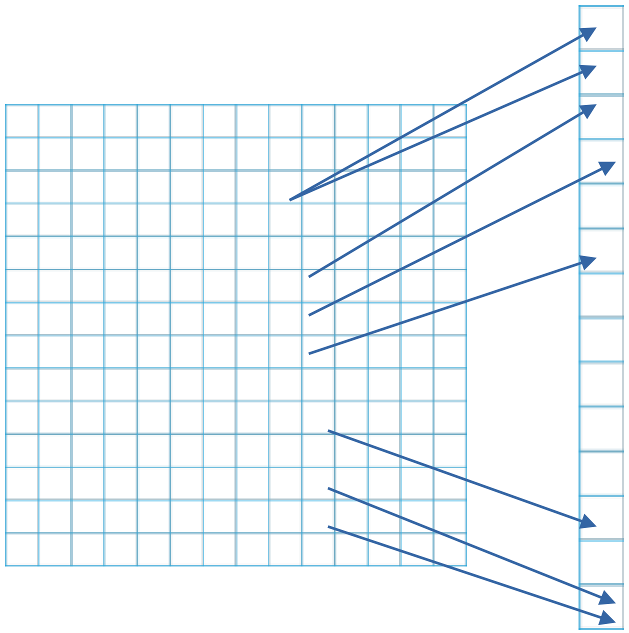
$$\langle I \rangle_r = \frac{\sum_{i \in bin_r} c_i \cdot signal_i}{\sum_{i \in bin_r} c_i \cdot normalization_i}$$

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

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

Methods to integrate the pattern

- Pixel splitting:
  - No splitting
  - Bounding Box
  - Pseudo (only 2D)
  - Full splitting

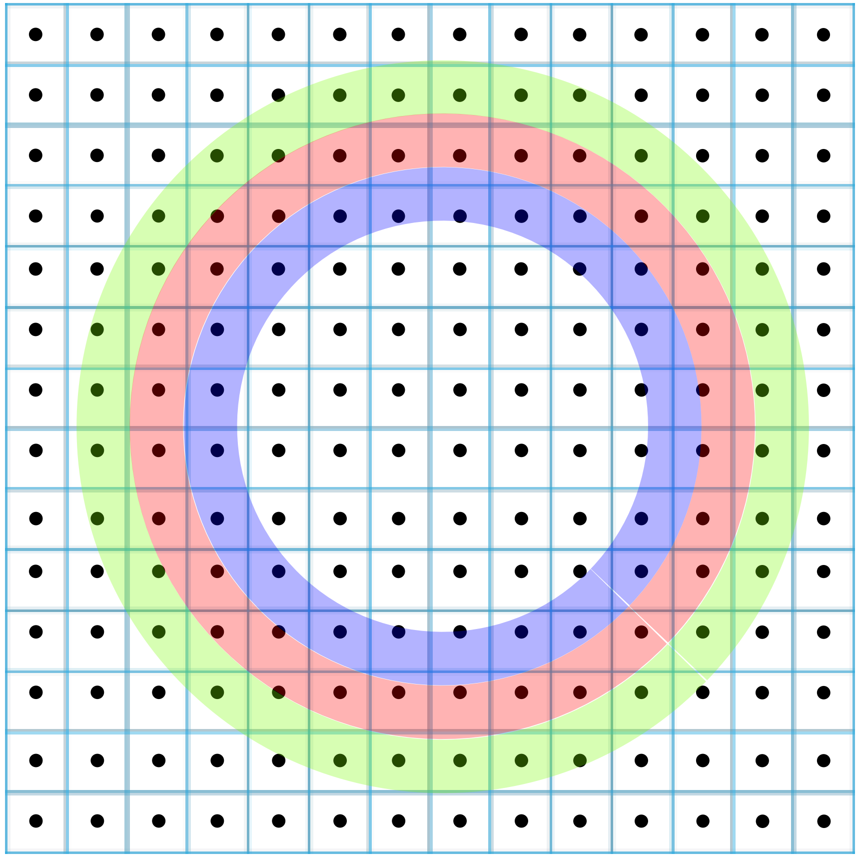


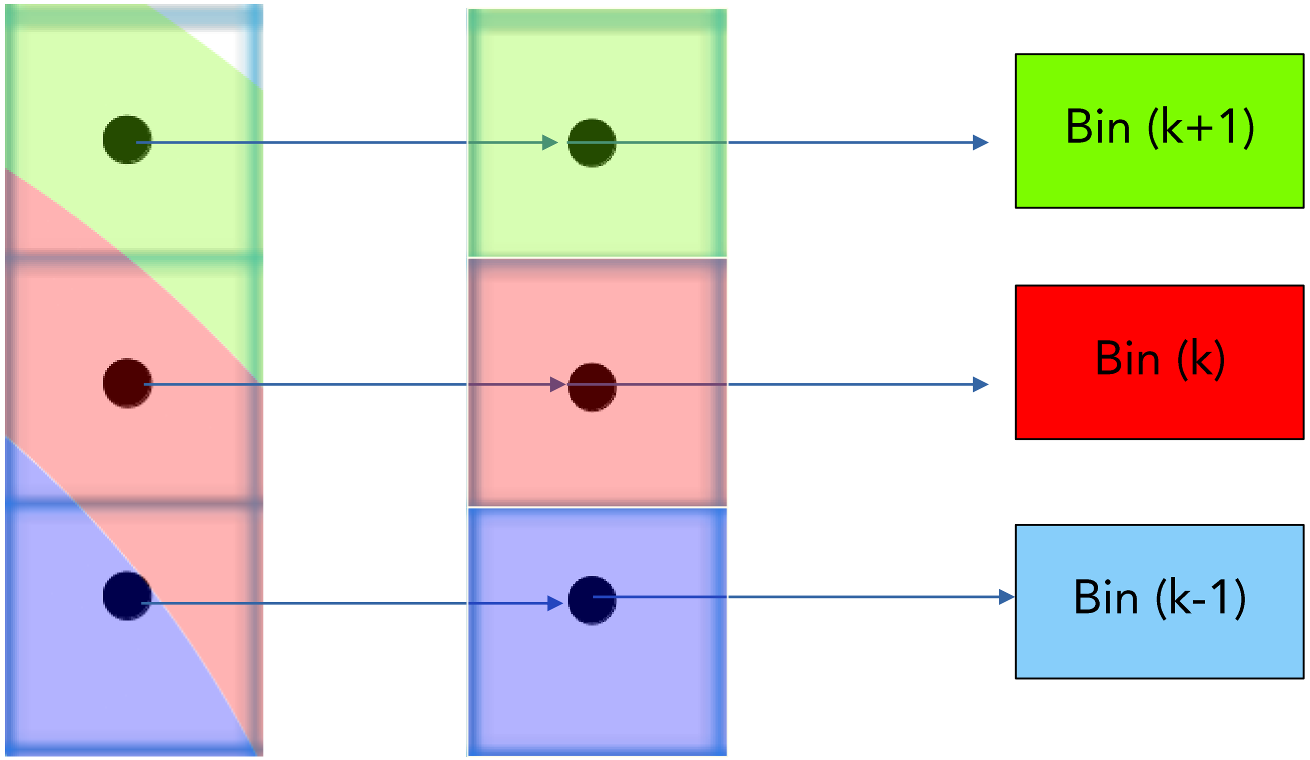
- Algorithm
  - Histogram
  - Sparse matrix multiplication:
    - CSR
    - CSC
    - LUT

- Implementation:
  - Python
  - Cython
  - OpenCL

Pixel splitting

- **No splitting:** one pixel to one single bin upon in which bin the center of the pixel is falling.

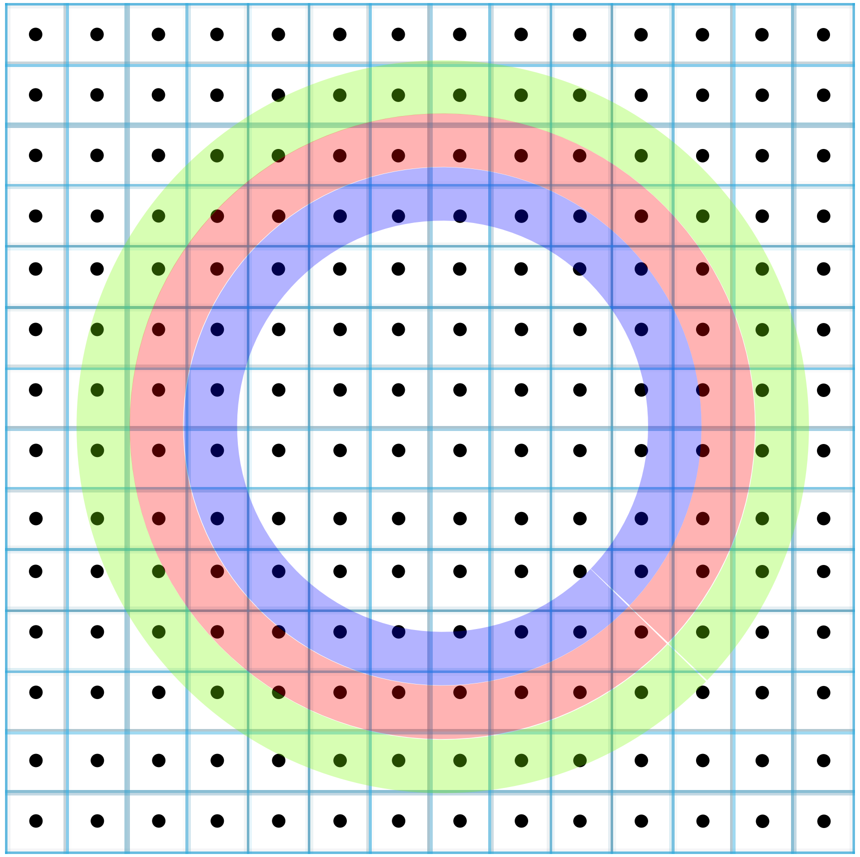




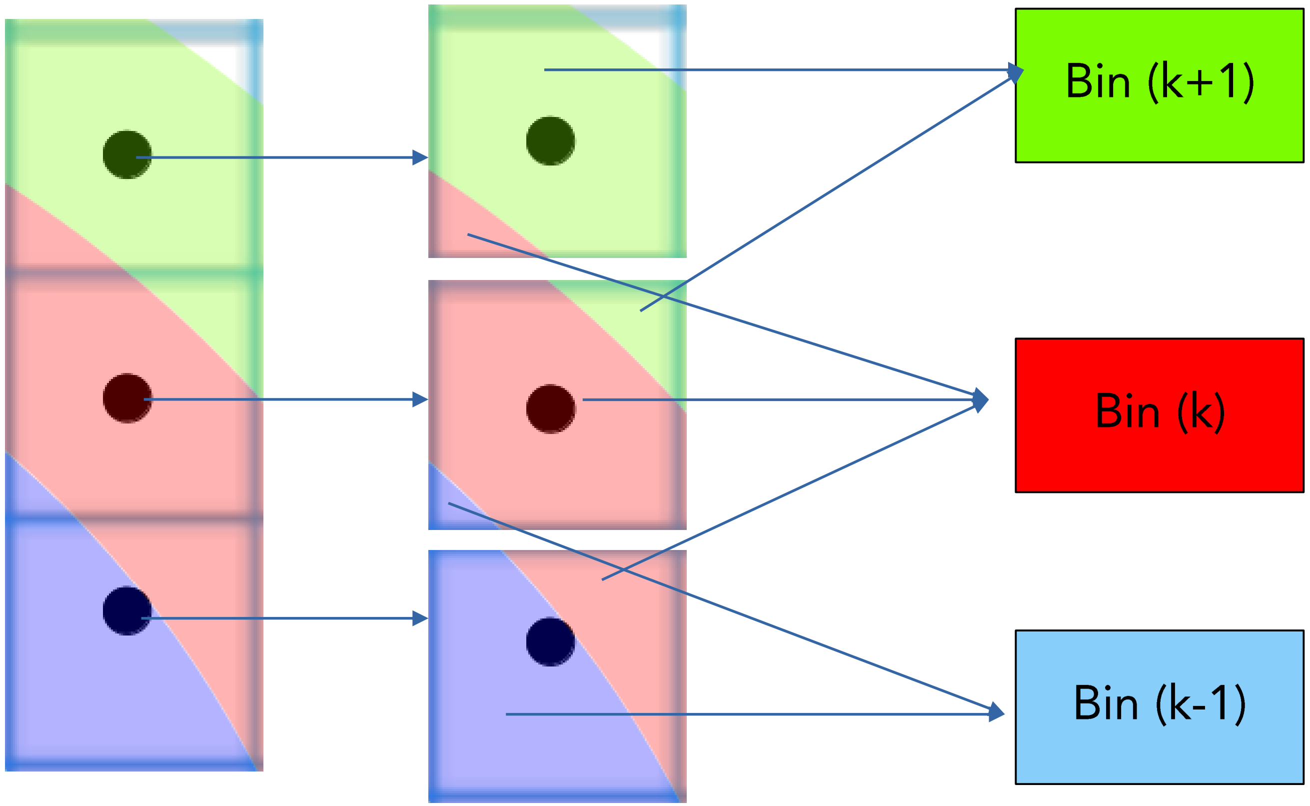
The intensity of each bin is the sum of the intensity of the pixels whose center falls into the radius bin.

Pixel splitting

- **Splitting:** each pixel intensity is shared between consecutive bins.



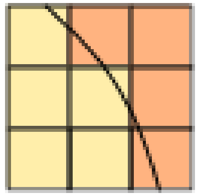
*Full pixel splitting



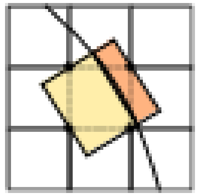
Each bin is the sum of the intensities of different pixels multiplied by a weight component between 0.0 and 1.0.

Pixel splitting methods

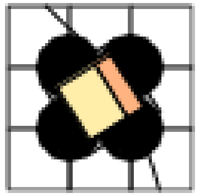
- **No pixel splitting**
  - Each pixel contributes to a single bin
  - No bin correlation but noisy
  - The pixel has no surface: sharpest peaks



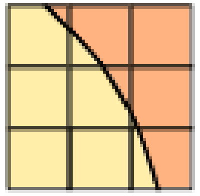
- **Bounding box** splitting (default)
  - Smooth integrated curve
  - Blurs a bit the signal



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

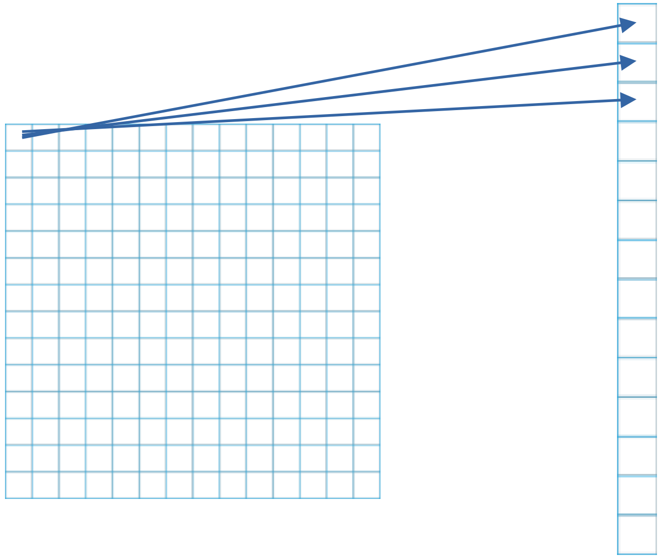


- **Full pixel** splitting
  - Split each pixel as a polygon
  - Costly high-precision choice

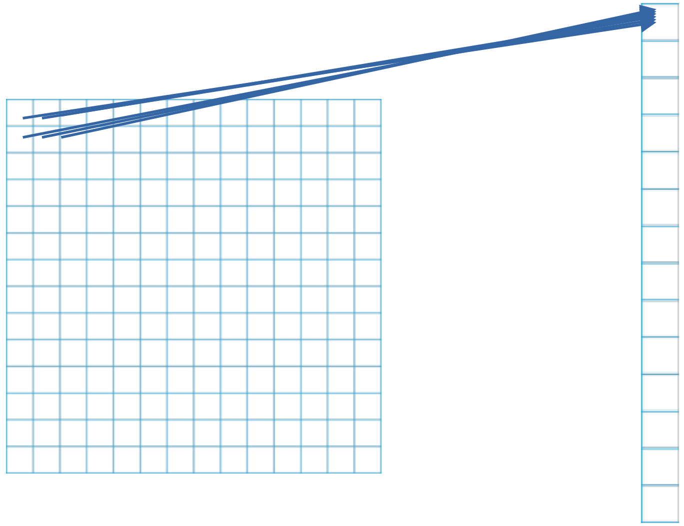


Algorithms

- ## Histogram
  - Pixel by pixel procedure.
  - Each pixel is split over the bins it covers.
  - Corner coordinates have to be calculated (4x slower initialization).
  - The slow down is function of the oversampling factor, for every image.
  - Serial read → Random write



- ## Sparse Matrix Multiplication
  - Bin by bin procedure.
  - Creates and stores a sparse matrix with all the integration information.
  - The sparse matrix can be huge: longer initialization related to the oversampling factor.
  - No performance penalty on the integration itself.
  - Serial write ← Random read



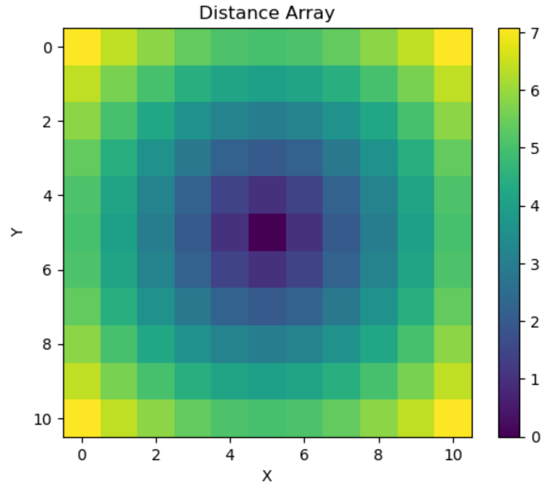
Algorithms

Import the modules

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from pyFAI.azimuthalIntegrator import AzimuthalIntegrator
```

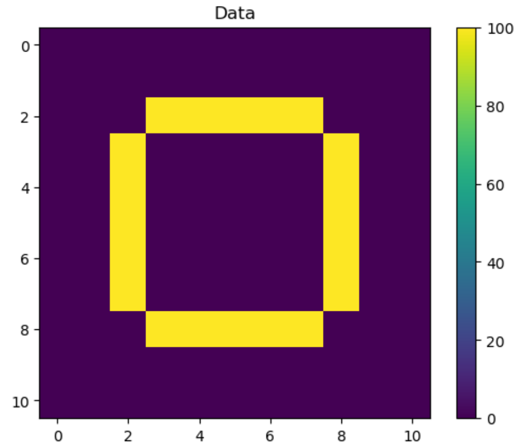
We will define a data matrix 11x11 with a diffraction ring in the middle

[2]:
```
SIZE = 11
# First, we will define a matrix of distances from the center of the matrix
x = np.linspace(-5, 5, SIZE)
y = np.copy(x)
X, Y = np.meshgrid(x,y)
array_d = np.sqrt(X**2 + Y**2)
plt.imshow(array_d)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Distance Array')
plt.colorbar()
pass
```



[3]:
```
# Data array with a diffraction ring
RING_LIMITS = [3.0, 4.0]
# Dummy array filled with the value 100
data = np.empty(shape=(SIZE, SIZE))
data.fill(100)

# Define a mask to draw a ring
mask = np.logical_and(array_d >= RING_LIMITS[0], array_d < RING_LIMITS[1])
data *= mask
plt.imshow(data)
plt.title('Data')
plt.colorbar()
pass
```

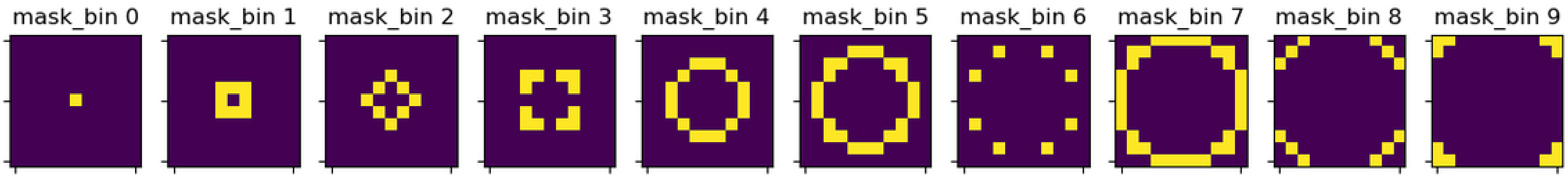


Now, we want to integrate this data array so the result intensity profile will have 10 data points (bins)

```
[4]: NPT = 10
```

Therefore, we have to chop the data array in 10 portions according to the distance from the center (the direct beam)

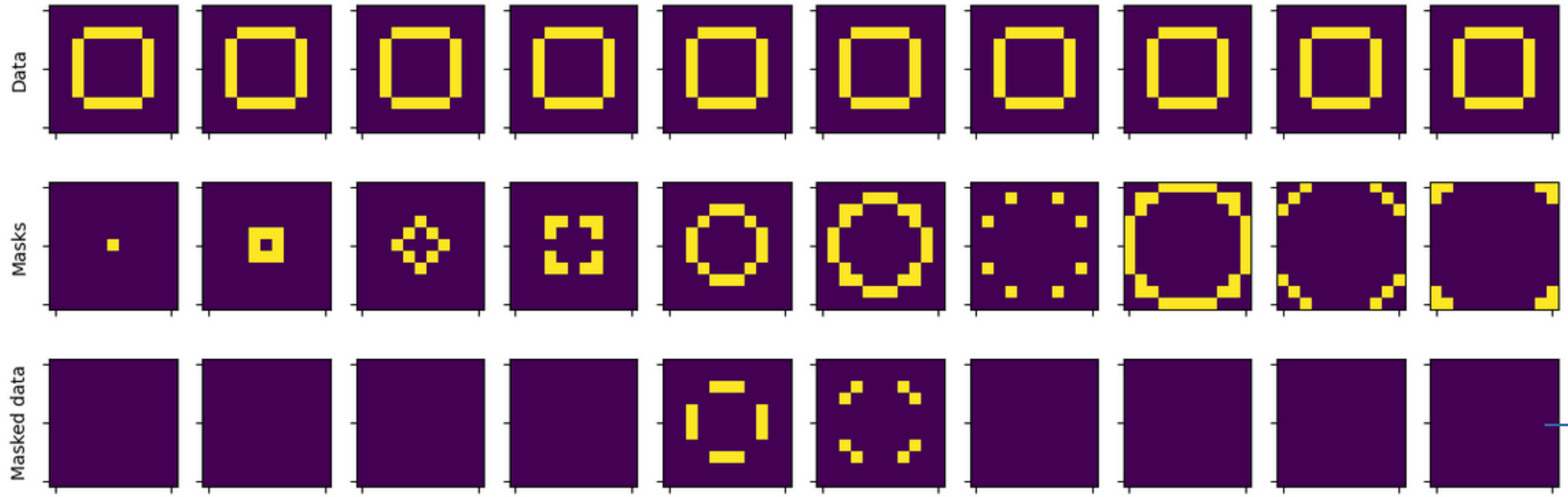
```
[5]: fig, axes = plt.subplots(ncols=NPT, figsize=(15,5))
     step = (array_d.max() + array_d.min()) / NPT
     masks = []
     for ii in range(NPT):
         min_threshold = array_d.min() + (ii) * step
         max_threshold = min_threshold + step
         new_mask = np.logical_and(array_d >= min_threshold, array_d <= max_threshold)
         masks.append(new_mask)
         axes[ii].imshow(masks[ii])
         axes[ii].set_title(f'mask_bin {ii}')
         axes[ii].set_xticklabels([])
         axes[ii].set_yticklabels([])
```

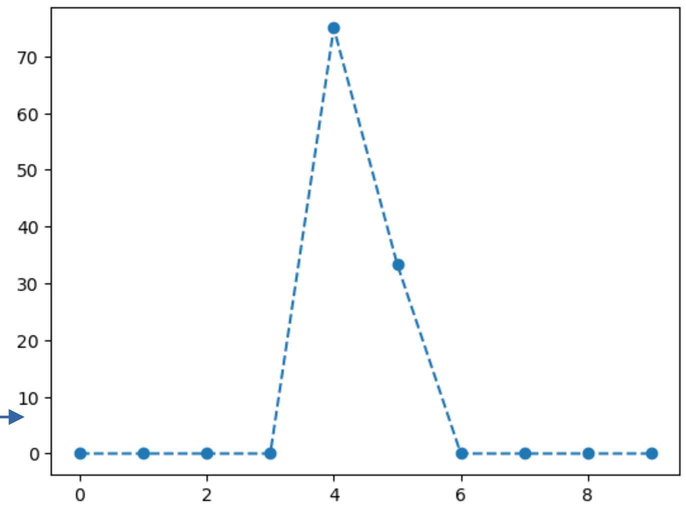


Algorithms

Each mask will multiply the data, and the result will be associated to each bin

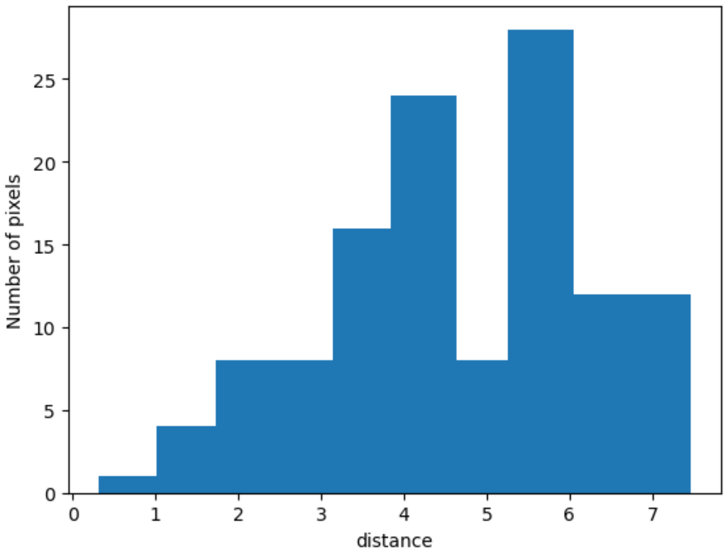
[6]:
```
fig, axes = plt.subplots(ncols=NPT, nrows=3, figsize=(15,5))
for ii in range(NPT):
    axes[0,ii].imshow(data)
    axes[1,ii].imshow(masks[ii])
    axes[2,ii].imshow(masks[ii]*data)
for ax in axes.ravel():
    ax.set_xticklabels([])
    ax.set_yticklabels([])
axes[0,0].set_ylabel('Data')
axes[1,0].set_ylabel('Masks')
axes[2,0].set_ylabel('Masked data')
pass
```



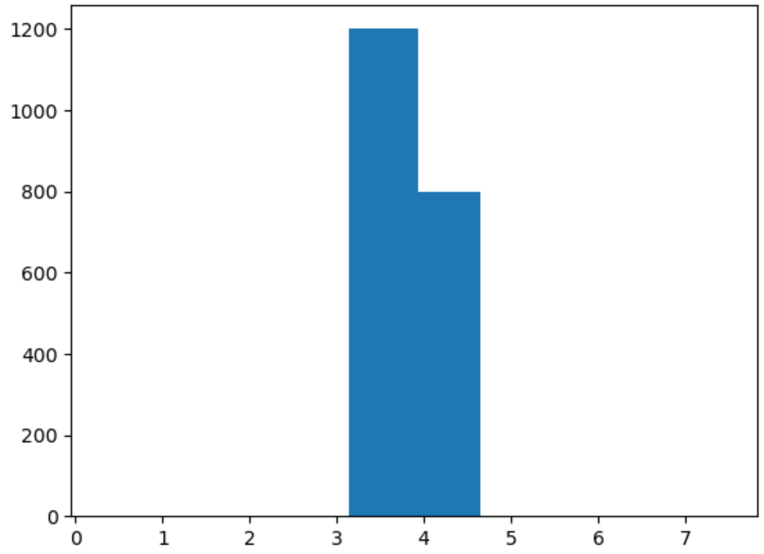


Algorithms: histogram

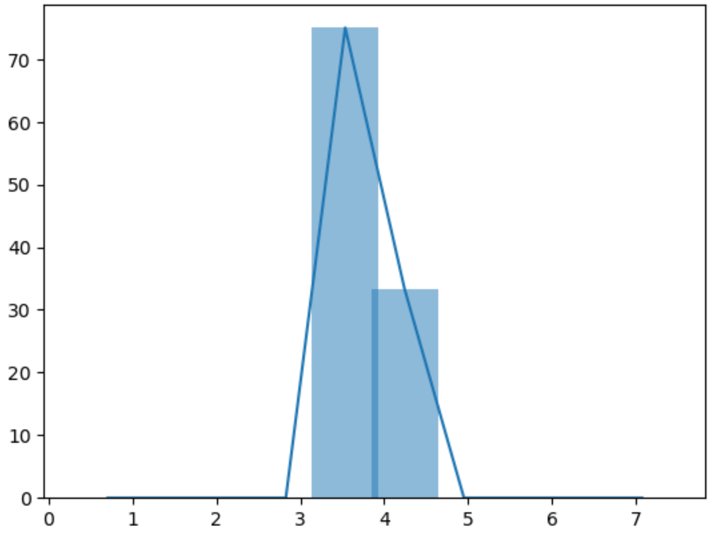
[8]:
```
number_of_pixels, d_positions = np.histogram(a=array_d, bins=NPT)
plt.bar(d_positions[1:], number_of_pixels)
plt.xlabel("distance")
plt.ylabel("Number of pixels")
pass
```



[9]:
```
data_hist, d_positions = np.histogram(a=array_d, bins=NPT, weights=data)
plt.bar(d_positions[1:], data_hist)
pass
```

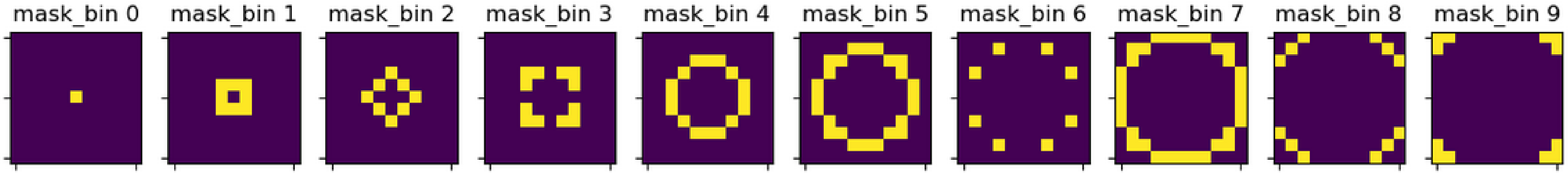


[28]:
```
data_norm = data_hist / number_of_pixels
plt.bar(d_positions[1:], data_norm, alpha=0.5)
plt.plot(d_positions[1:], data_norm)
pass
```

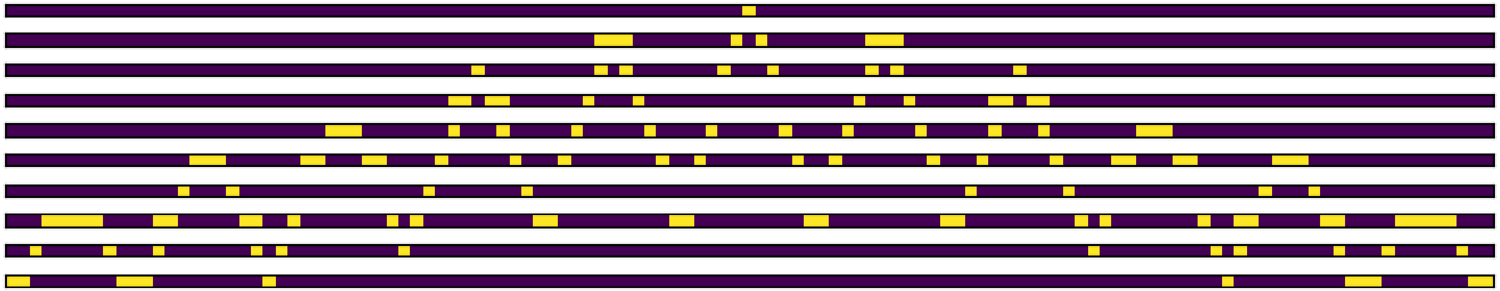


Algorithms: sparsification

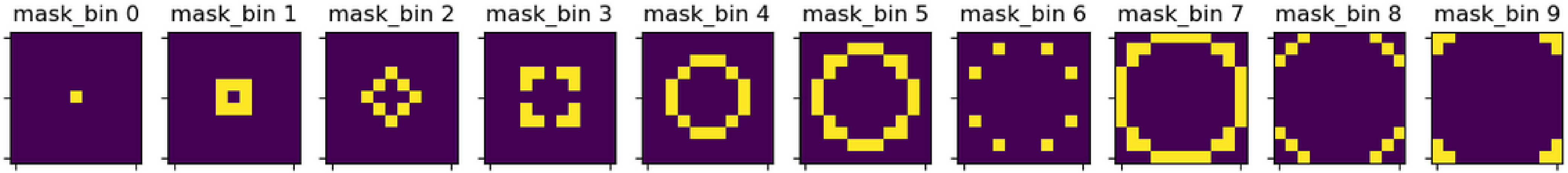
The other way is sparsification: the integration code is stored already in the masks images



[13]:
```
fig, axes = plt.subplots(nrows=NPT, figsize=(10,2))
masks_ravel = []
for ii in range(NPT):
    mask_ravel = masks[ii].ravel()
    masks_ravel.append(mask_ravel)
    axes[ii].imshow(mask_ravel[np.newaxis,:])
    axes[ii].set_xticklabels([])
    axes[ii].set_yticklabels([])
    axes[ii].set_yticks([])
    axes[ii].set_xticks([])
```



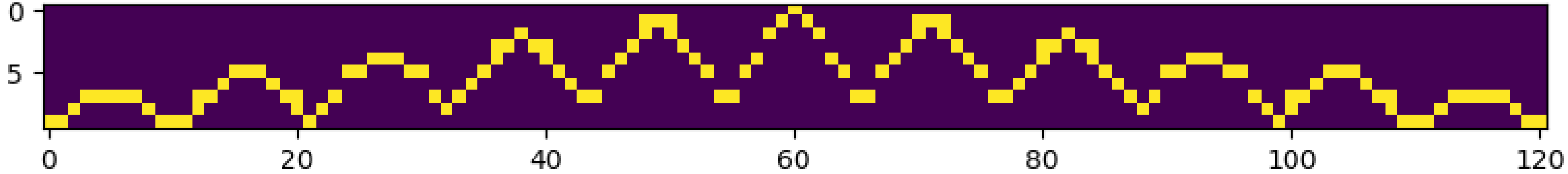
The other way is sparsification: the integration code is stored already in the masks images



Now, we can build up a dense matrix with the encoded instructions of integration

[14]:
```
dense_matrix = np.stack(masks_ravel, axis=0)
fig, ax = plt.subplots(figsize=(10,5))
ax.imshow(dense_matrix)
print(f'Shape of dense matrix: {dense_matrix.shape}: rows as bins and columns as matrix size')
pass
```

```
Shape of dense matrix: (10, 121): rows as bins and columns as matrix size
```

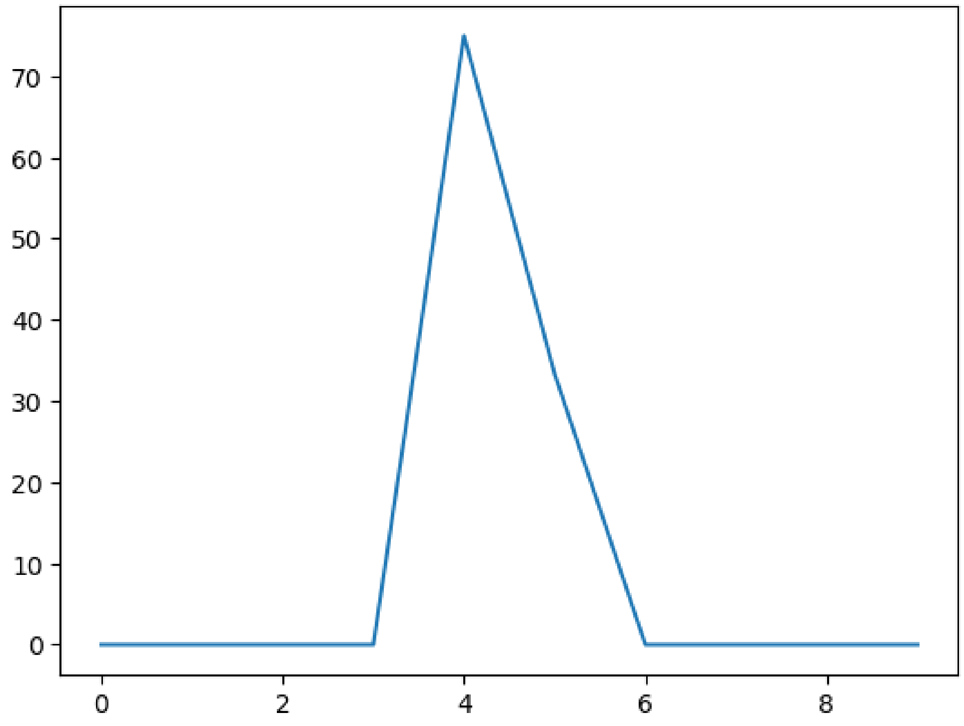


Dense matrix (10x121) x data_flattened (121x1) = result (10x1)

Algorithms: sparsification

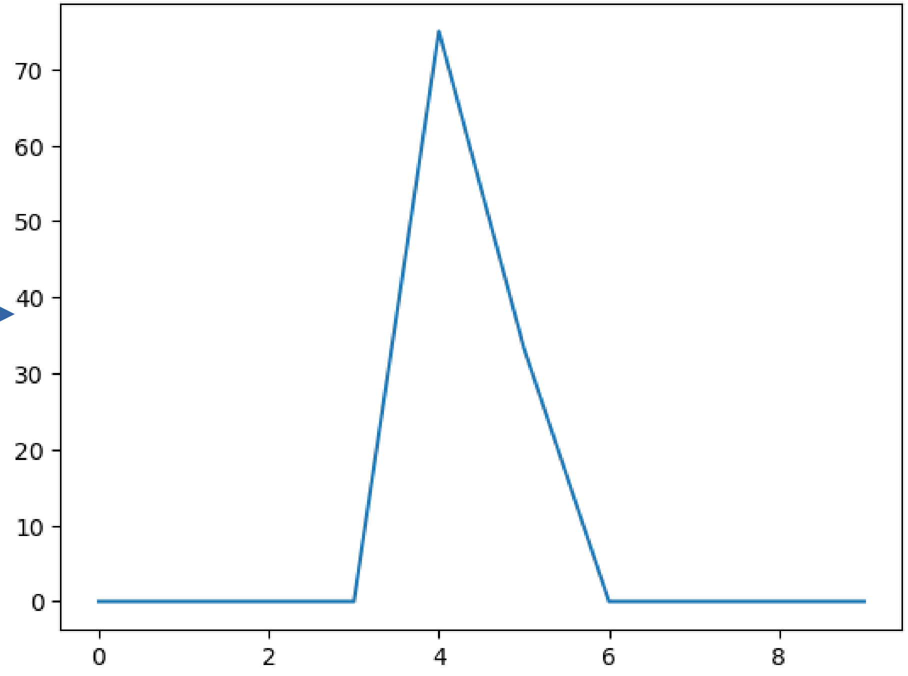
Multiplication of dense matrix

[17]:
```
counts = np.dot(dense_matrix, np.ones_like(data_ravel))
result_norm = result / counts[:,np.newaxis]
plt.plot(result_norm)
pass
```



Multiplication of compressed sparse matrix

[18]:
```
from scipy.sparse import csr_array
csr = csr_array(dense_matrix)
result = csr.dot(data_ravel[:,np.newaxis])
counts = csr.dot(np.ones_like(data_ravel))
result_norm = result / counts[:,np.newaxis]
plt.plot(result_norm)
pass
```



Layers in pyFAI

- Applications level:
  - GUI applications: pyFAI-calib2, pyFAI-integrate, diff_map, worker
  - Scriptable applications: pyFAI-average, pyFAI-saxs, pyFAI-waxs, diff_tomo
- Python interface:
  - Top level: azimuthal integrator (through importing the poni file) – Jupyter Notebooks
  - Mid level: calibrant, detector, geometry, calibration
  - Low level: rebinning/histogramming engines (Cython + OpenMP/OpenCL)

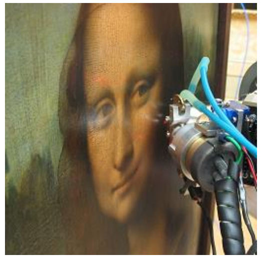
- It is up to the user to choose the right balance.

Latest news from pyFAI

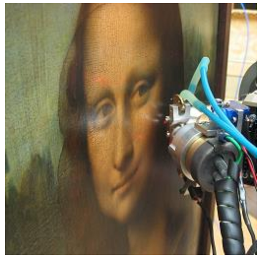
- **PyFAI-2024.1.0: release on 18/01/2024 + PyFAI-2024.1.1: released on 01/02/2024**
- Orientation-tag in detector instances (compatibility with Dioptas)
- New functionalities in pyFAI-calib2 GUI
- Guessing of bins (avoid oversampling)
- Support python 3.7 – 3.12 (3.12 needs silx 2.0, check release)
- Drop setup.py build system (use meson-python)
- Support XRDML files, pathlib instances for .poni files
- Grazing-Incidence capabilities: new units q_in-plane and q_out-of-plane
- Consistent azimuthal errors between methods
- Integrated dynamic mask on pyFAI-calib2

Project management: Silx & pyFAI

- Compatible with Windows, MacOS, Linux
- MIT licensed: compatible with both science and business
- **PyFAI** is embedded in the silx-kit project: https://github.com/silx-kit/
- **Silx-kit** project is python-based, developed at the ESRF and includes:
  - **Silx** toolkit: multifunctional library specially focused on i/o data files and GUI widgets
  - **FabIO**: I/O library to handle 2D detector files
  - **PyMCA**: X-ray fluorescence toolkit
  - **H5web/myhdf5**: web-based widget to browse through .h5 files
  - **FreeSAS**: small-angle scattering toolkit
  - **DAHU**: online data analysis server

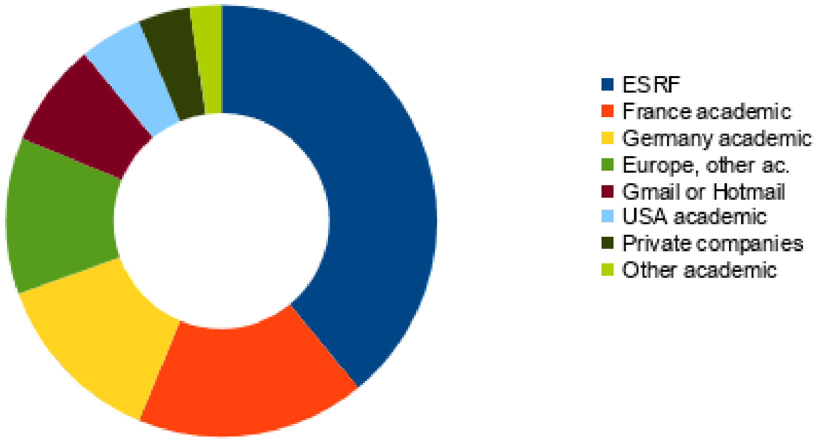


- Compatible with Windows, MacOS, Linux
- MIT licensed: compatible with both science and business
- **PyFAI** is embedded in the silx-kit project: https://github.com/silx-kit/
- **Silx-kit** project is python-based, developed at the ESRF and includes:
- Open to collaborations:
  - About 20 direct contributors from ESRF, from other synchrotrons, XFELs (Soleil, NSLS-II, Petra-III, Eu-XFEL) and companies (Xenocs)
  - Used by ~90 other projects from many other X-ray sources in the world (SLAC, ALS, APS, ALBA, NSLS-II, Petra-III, Soleil, Diamond, SLS, MaxIV…)



PyFAI user community

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



- User support is provided via the mailing list: pyFAI@esrf.fr (183 subscribers)
- Bugs are discussed via Github issue tracker
  - https://github.com/silx-kit/pyFAI/issues

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

- **Versatile** (increasing with every version)
  - Wide space to vary the integration protocol
  - Generic geometry
  - Most detectors already defined (+ custom detector through Nexus file)
  - Graphical user interface alternatives (thanks to Valentin Valls)

Silx-kit: join efforts, share the maintenance

Fork me on GitHub

silx-kit

silx

Scientific Library for eXperimentalists

Resources

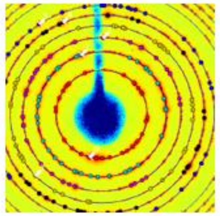
- silx on GitHub
- Wheels and source on PyPi
- Installation instructions

Documentation

- Latest release
- Nightly build
- ...

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

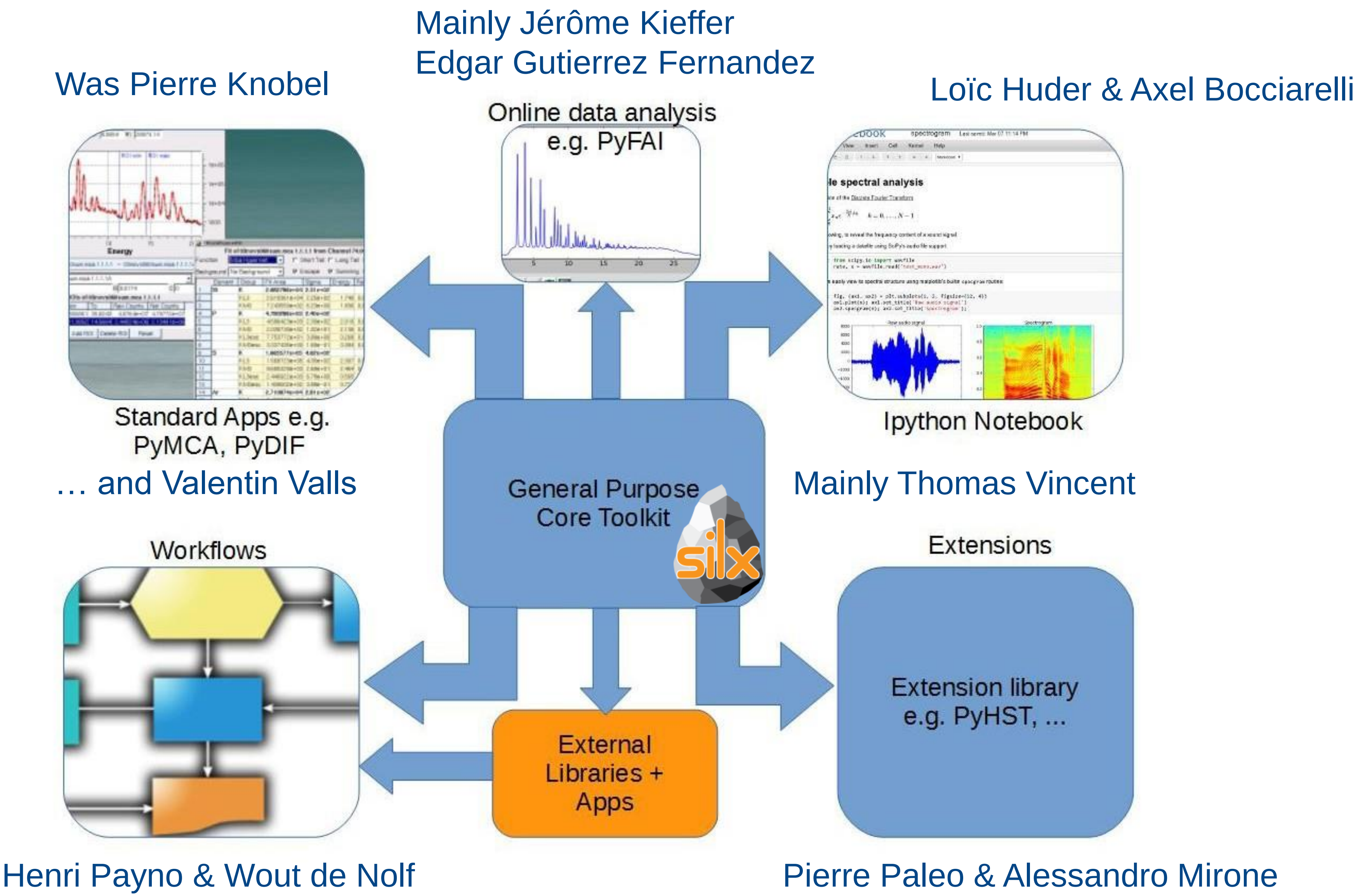
Resources

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

Documentation

- Latest release
- Nightly build
- ...

Silx-kit project and library



Acknowledgements

- Former data analysis unit colleagues:
  - Valentin Valls
  - Loic Huder
  - Thomas Vincent
  - Claudio Ferrero
- ESRF Beamlines:
  - BM01, BM02, ID02, ID11, ID13, ID15a, ID15b, ID21, ID22, ID23, BM26, ID27, BM28, 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

PyFAI-tutorial overview

- Install python environments + pyFAI
- Learn how to perform a **setup calibration**:
  - Through pyFAI-calib2
  - Through Jupyter Notebook
- Learn how to perform an **azimuthal integration**
  - Through python shell
  - Through Jupyter Notebooks
  - Through pyFAI-integrate
- Azimuthal integrator attributes:
  - 1D/2D integration, caking, different methods (pixel splitting, algorithms, GPUs…)
  - Low/Mid level tools: unit arrays
- Other tools:
  - pyFAI-benchmark
  - pyFAI-integrate
  - pyFAI-diffmap
  - pyFAI-average
  - pyFAI-waxs / pyFAI-saxs
  - PyFAI.worker

ESRF User Meeting 2024
PyFAI tutorial

Edgar Gutierrez Fernandez

COFFEE BREAK (15')

Installing a Python environment

- Virtual Environment
- Anaconda
- Miniconda (light option) / micromamba (lightest option)
  - Download the installer from https://docs.conda.io/projects/miniconda/en/latest/
  - Install miniconda3:
    - >sh ..file.sh for Linux/MacOS users
    - Execute .exe file for Windows users
  - Activate the conda prompt:
    - > source ~/miniconda3/bin/activate (Linux/MacOS users)
    - >~miniconda3\\Scripts\\activate (Windows users)
  - Create our conda environment:
    - (base) > conda create -n pyfai_tutorial python==3.11
  - Activate our conda environment:
    - (base) > conda activate pyfai_tutorial
  - Install packages (could take a while):
    - (pyfai_tutorial) >conda install pyFAI[gui] jupyterlab -c conda-forge
    - Alternative: (pyfai_tutorial) >pip install pyFAI[gui] jupyterlab

Download data files from:
http://www.silx.org/pub/pyFAI/pyFAI_UM_2024/