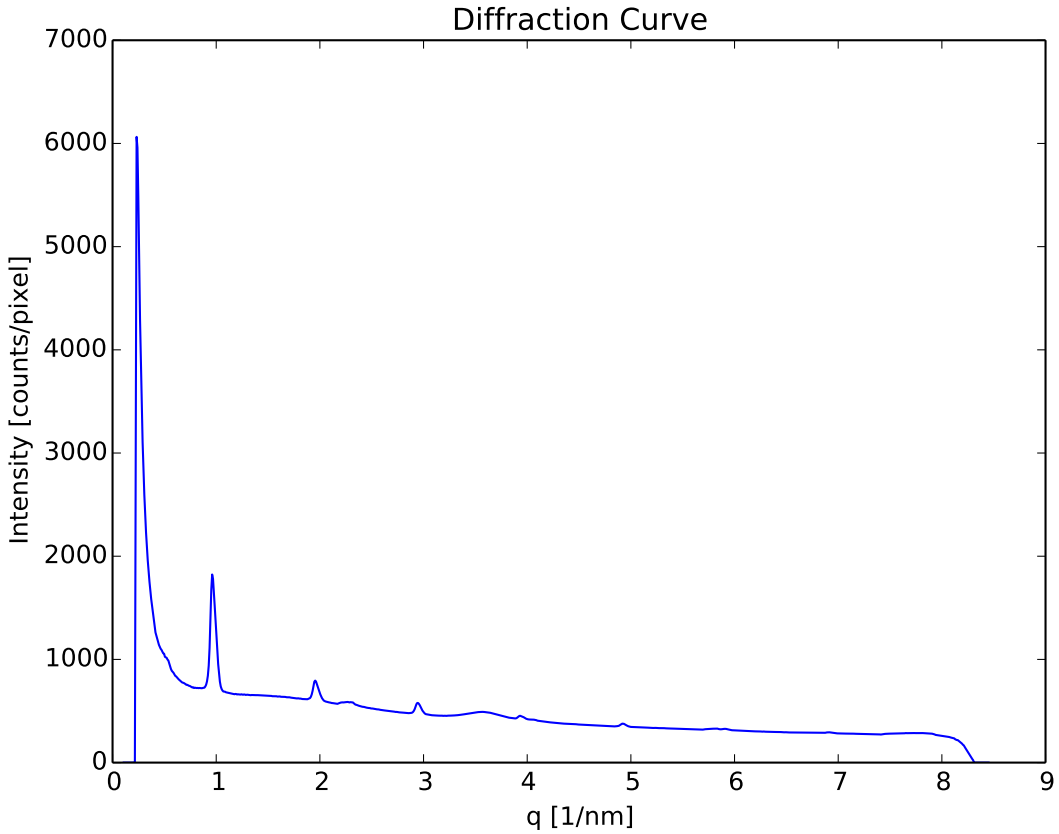
SAXS Documentation

Release 0.2.1

Christian Meisenbichler

July 03, 2014

The SAXS python package implements analysis tools for Small Angle X-Ray Scattering (SAXS) data analysis. The first and most important one is to efficiently integrate 2d sensor data to an angle dependent diffraction curve.



The SAXS module consists of a Pyhon library and 3 command line tools: *The Saxsdog*, *Plotchi* and *The Converter*

CHAPTER

ONE

INSTALL

The SAXS Package is distributes as a Python package. So in order to use it, you need a Python system installed. It depends on following Python modules that don't come with the standard Python:

```
numpy scipy matplotlib jsonschema bitarray watchdog sphinxcontrib-programoutput
```

they are all available through "pip" so the command:

```
>>pip install numpy scipy matplotlib jsonschema bitarray watchdog sphinxcontrib-programoutput
```

Should get all the modules. For Windows, use the Anaconda Python distribution which includes pip.

The code can be obtained on github: https://github.com/ChristianMeisenbichler/SAXS where you would also find a "Download Zip" button. After unpacking or cloning with git you end up with a directory called "SAXS" containing the files. Go there, and type into the commad line:

```
python setup.py install
```

This installs the Python module where it is found by Python, creates the command line tools and installs them on the system. Where that is, depends on the Python installation.

CHAPTER

TWO

THE TOOLS

2.1 The Saxsdog

The saxsdog is a script that converts directories with images to curves. It can use multiple threads and watch the file system for changes.

For Help on the usage type:

```
$ saxsdog --help
Usage: saxsdog [options] directory/to/watch

Options:
  -h, --help            Show this help message and exit.
  -c FILE, --calibration=FILE
                        Path to calibration file (JSON).
  -t THREADS, --threads=THREADS
                        Number of concurrent threads.
  -m, --plotmonitor     show a live updating plot window.
  -w, --watch           Watch directory for changes, using file system events
                        recursively for all sub directories.
  -r, --resume          Skip files that are already converted.
  -o OUTDIR, --out=OUTDIR
                        Specify output directory. Default is './out'.
  -i, --inplace         Files are written, in place, in the directory of the
                        image.
  -s, --svg             Write plot to svg file.
  -p, --png             Write png of original.
  -P, --profile         Make a time Profile and print it.
  -S, --silent          Less output.
```

The calibration file must be a valid *JSON Configuration File*

2.2 The Converter

The converter extracts information from the calibration.txt generated by fit2d and adds them to a SAXS.calibration configuration file. (*JSON Configuration File*)

```
$ saxsconverter  --help
Usage: saxsconverter [options] calibration.txt ouput.json

Options:
  -h, --help            show this help message and exit
  -t FILE, --template=FILE
                        Path to calibration file which serves as template.
```

If there is a target file and it is a valid *JSON Configuration File*, the parsed values are added or updated in place.

2.3 Plotchi

The tool “plotchi” plots a list of ”.chi”-files:

```
$ plotchi  --help
Usage: plotchi [options] CHIFILE [List of more ".chi" files]

Options:
  -h, --help            show this help message and exit
  -o FILE, --out=FILE   Write the plot to FILE. The format is derived from the
                        suffix, e.g. '.svg','.pdf'.
  -c, --compare         Compare datasets to first one.
  -l, --log             Use log scale.
  -n, --no-legend       Hide legend.
  -t TITLE, --title=TITLE
                        Give plot title.
  -s N, --skip=N        Skip first N points.
  -k N, --clip=N        Clip last N points.
```

CHAPTER

THREE

THE CALIBRATION FILE

The `SAXS.calibration` class and the saxsdog tool acept a input file with the calibration data. This input file is written as a JSON file. This is a common syntax to express structured data as text. You might want to read a bit about it bevore moving on.

3.1 JSON Configuration File

The '*' signifies a required Field.

The SAXS configuration file specifies the parameters of a SAXS sensor calibration. It is written in the JSON format which governs the general syntax.

Type object

Contains *Title*, *Tilt**, *BeamCenter**, *DedectorDistanceMM**, *Imagesize**, *MaskFile**, *Oversampling**, *PixelSizeMicroM**, *PixelPerRadialElement**, *Wavelength**, *PolarizationCorrection*

Required True

JSON Path *#*

Example JSON:

```
{
  "PixelSizeMicroM": [
    172.0
  ],
  "Imagesize": [
    1043,
    981
  ],
  "PixelPerRadialElement": 1,
  "Tilt": {
    "TiltRotDeg": 0,
    "TiltAngleDeg": 0
  },
  "MaskFile": "AAA_integ.msk",
  "Oversampling": 3,
  "Wavelength": 1.54,
  "BeamCenter": [
    808.37,
    387.772
  ],
  "DedectorDistanceMM": 1031.657
}
```

3.1.1 Title

Type string

Required False

JSON Path *#* ['*Title*']

Example JSON:

```
{"Title": ""}
```

3.1.2 Tilt

The sensor, usually is not perfectly perpenticular to the ray direction. The tilt angle can be specified by giving the following paramters.

Type object

Contains *TiltRotDeg**, *TiltAngleDeg**

Required True

JSON Path *#* ['*Tilt*']

Example JSON:

```
{"Tilt": {"TiltRotDeg": 0,"TiltAngleDeg": 0}}
```

3.1.3 TiltRotDeg

This gives the angel of the tilt direction.

Type number in degree

Required True

Default 0

JSON Path *#* ['*Tilt*']['*TiltRotDeg*']

Example JSON:

```
{"TiltRotDeg": 0}
```

3.1.4 TiltAngleDeg

This gives the angle between the ray direction and the normal to the sensor plane.

Type number in degree

Required True

Default 0

JSON Path *#* ['*Tilt*']['*TiltAngleDeg*']

Example JSON:

```
{"TiltAngleDeg": 0}
```

3.1.5 BeamCenter

Gives the beam center in pixel coorinates.

Type array(2) items: number number

Required True

Default [808.37, 387.772]

JSON Path # ['*BeamCenter*']

Example JSON:

```
{"BeamCenter": [808.37,387.772]}
```

3.1.6 DedectorDistanceMM

Distance between diffraction center and sensor.

Type number in Millimeters

Required True

Default 1031.657

JSON Path # ['*DedectorDistanceMM*']

Example JSON:

```
{"DedectorDistanceMM": 1031.657}
```

3.1.7 Imagesize

Size of sensor image in pixel.

Type array(2) items: number number

Required True

Default [1043, 981]

JSON Path # ['*Imagesize*']

Example JSON:

```
{"Imagesize": [1043,981]}
```

3.1.8 MaskFile

Path of Maskfile

Type string

Required True

Default AAA_integ.msk

JSON Path # ['*MaskFile*']

Example JSON:

```
{"MaskFile": "AAA_integ.msk"}
```

3.1.9 Oversampling

Oversampling factor for radial integration. The higher, the longer the setup but the higher the accuracy. More then 3 is probably overkill.

Type number

Required True

Default 3

JSON Path *#* ['*Oversampling*']

Example JSON:

```
{"Oversampling": 3}
```

3.1.10 PixelSizeMicroM

The pixel size on the sensor.

Type array(2) items: number

Required True

Default [172.0]

JSON Path *#* ['*PixelSizeMicroM*']

Example JSON:

```
{"PixelSizeMicroM": [172.0]}
```

3.1.11 PixelPerRadialElement

Expresses the width of a radial step in terms of pixels. '1' means $\delta R \approx 1$ *PixelSizeMicroM*.

Type number

Required True

Default 1

JSON Path *#* ['*PixelPerRadialElement*']

Example JSON:

```
{"PixelPerRadialElement": 1}
```

3.1.12 Wavelength

Refined wavelength.

Type number in Angstrom

Required True

Default 1.54

JSON Path *#* ['*Wavelength*']

Example JSON:

```
{"Wavelength": 1.54}
```

3.1.13 PolarizationCorrection

The scattering direction id dependend on the light polarization. This may be acconted for with the polarization correction.

Type object

Contains *Fraction**, *Angle**

Required False

Default OrderedDict([(u'Fraction', 0.95), (u'Angle', 0)])

JSON Path # ['*PolarizationCorrection*']

Example JSON:

```
{"PolarizationCorrection": {"Angle": 0.0,"Fraction": 0.95}}
```

3.1.14 Fraction

Fraction of light polarized in the given (*Angle*) direction.

Type number

Required True

Default 0.95

JSON Path # ['*PolarizationCorrection*']['*Fraction*']

Example JSON:

```
{"Fraction": 0.95}
```

3.1.15 Angle

Angle of the polarization plane.

Type number in degree

Required True

Default 0.0

JSON Path # ['*PolarizationCorrection*']['*Angle*']

Example JSON:

```
{"Angle": 0.0}
```

CHAPTER

FOUR

THE TECHNOLOGY

4.1 Integration as Matrix-Vector Multiplication

Every SAXS image $\mathbf{p}$ is a list of pixels that have an intensity value. This 2d array might as well be addressed as a vector with all the pixels addressable with one index $\mathbf{p}_i$.

The integration over pixels that are within a certain radial interval is in any case a weighted sum of some of the pixels.

This weighted sum is a scalar product with another vector containing the weight factors. As only the pixels in a radius interval are counted, most of these factors are 0.

$$r = \mathbf{c} \cdot \mathbf{p}$$

As we intend to do all the radial intervals at once, we write it as a matrix vector product.

$$\mathbf{r} = \mathbf{X} \cdot \mathbf{p}$$

The columns are the weight factors for the i^{th} radial element. Rearranged in the order of the image, this looks like the ring element relevant for the radial Point.

This matrix would be quite big as it has the dimensions len($\mathbf{r}$)*len($\mathbf{p}$). Fortunately most of the entries are 0 and we can use a sparse matrix representation which uses only about ~len($\mathbf{p}$) of memory, as every pixel is counted only once, or, as we will see, about once.

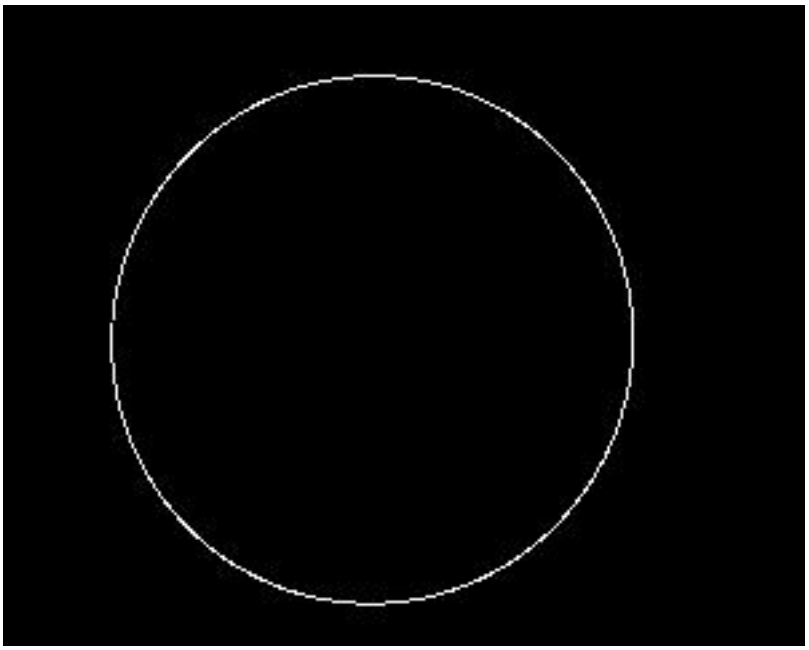


Figure 4.1: The vector $\mathbf{c}$ displayed as image.

Figure *CircleNoAA* Scows the data of such a matrix column.

4.2 Oversampling

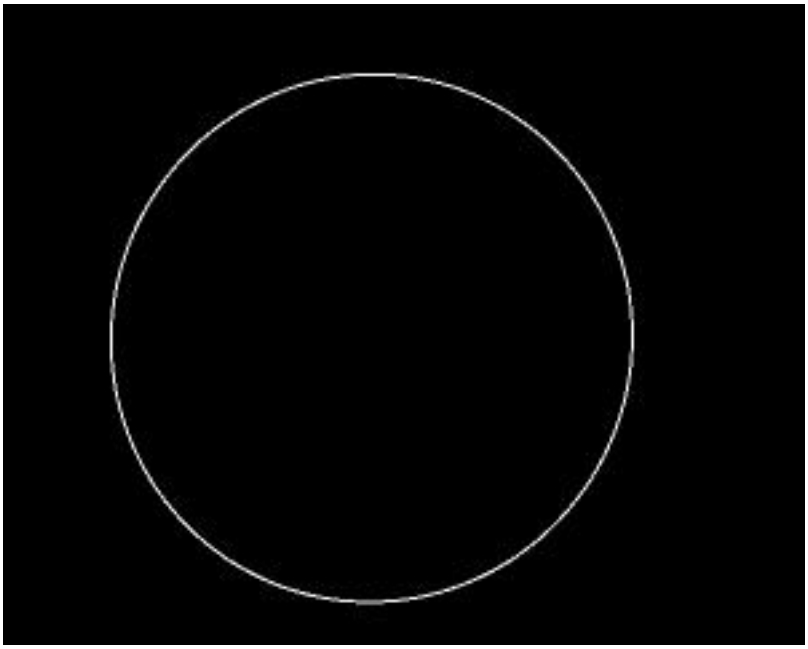


Figure 4.2: Ring with antialiasing / oversampling.

A pixel might lie on the border of two radial intervals, making it unclear to which one it should be added. By only choosing the nearest one, one may get artifacts in the resulting curve especially when only few pixels contribute. So, how could we calculate to which fraction a pixel should account to one radial interval?

The idea here is to use an algorithm comparable to antialiasing in computer graphics. We will divide a much larger picture into the radial intervals and downsample it to the real pixels. Which results in nicely balanced factors for the border pixels that add up nicely over joining intervals such that the intensity is conserved. If one looks closer at image *Ring with antialiasing / oversampling.* , one sees that the ring has soft edges. Quite as it would have through antialiasing.

4.3 The Geometry

The plane of the sensor is not perfectly normal to the beam. So in order to calculate witch pixel is on witch cone in the diffracted light, we need to express the geometry somehow.

Every pixel has the polar coordinates r, ϕ with the projected diffraction center in the origin. For each pixel (P) the triangle S,C,P (Sample, Center, Pixel, θ, β, γ) can be fully expressed with the law of cosines.

l is the distance the light travels from the diffraction center to the sensor.

$$l^2 = d^2 + r^2 - 2dr\cos(\pi/2 + \alpha)$$

r is the radial coordinate of the pixel P.

$$r^2 = l^2 + d^2 - 2ld\cos(\theta)$$

from these two formulas the diffraction angle (here) θ can be computed.

$$\theta = \arccos(-r^2 - l^2 - d^2 2ld)$$

α comes from the following relation in figure *Angle between two planes.*

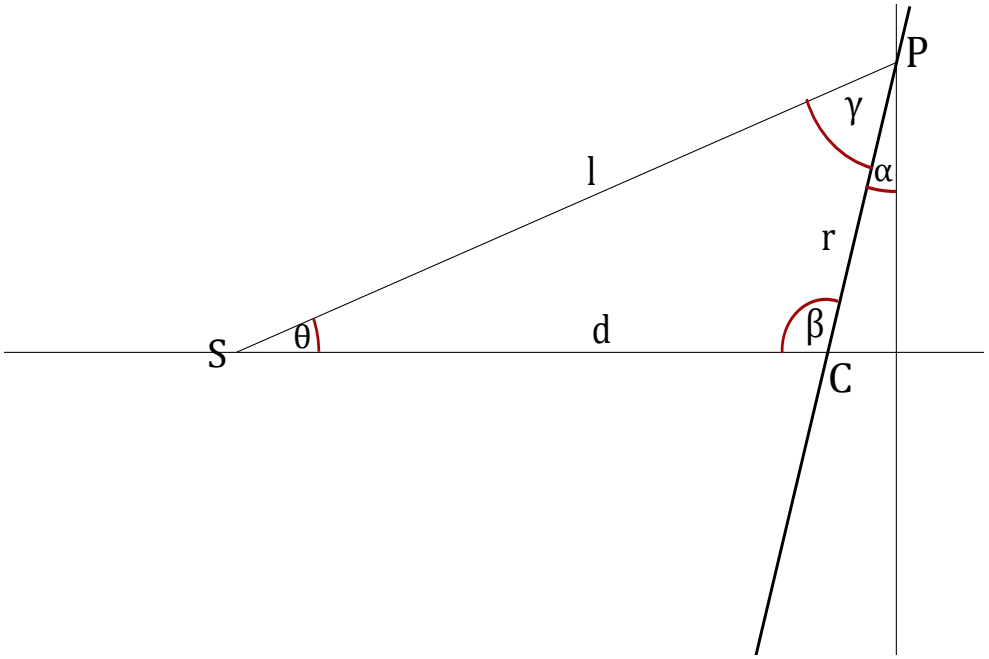


Figure 4.3: The SCP triangle.

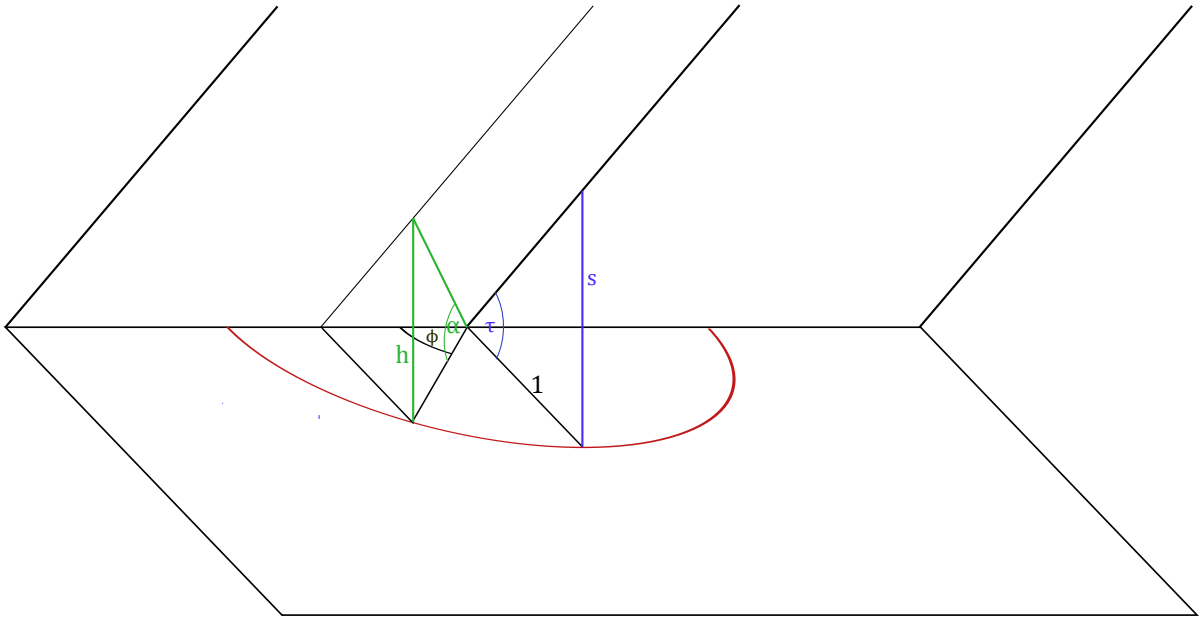


Figure 4.4: Angle between two planes.

The angle between the sensor plane and the normal plane to the ray is given by τ. The slope s derived from τ is

$$s = \sin(\tau)$$

On the (red) unit circle in the plane of the sensor the distance to the plane normal to the ray is expressed as

$$h = \sin(\phi)s$$

The angle α is therefore:

$$\alpha = \arcsin(\sin(\tau)\sin(\phi))$$

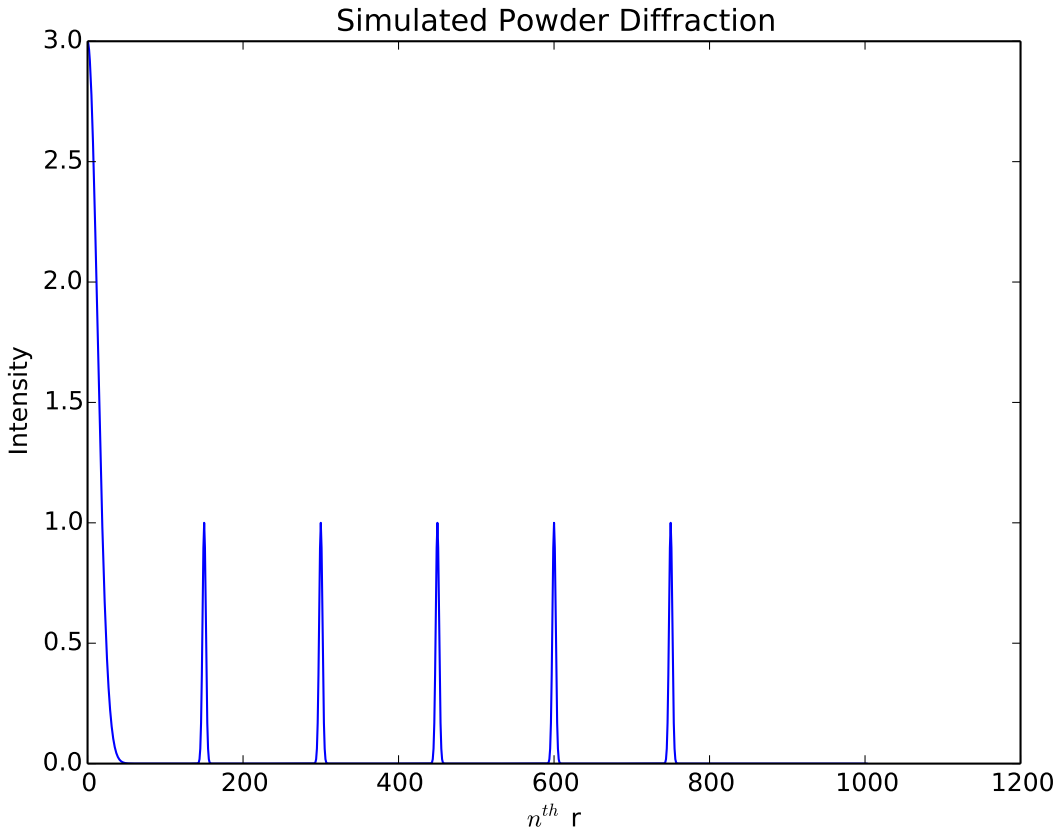
in python code this is `SAXS.calc_theta()`:

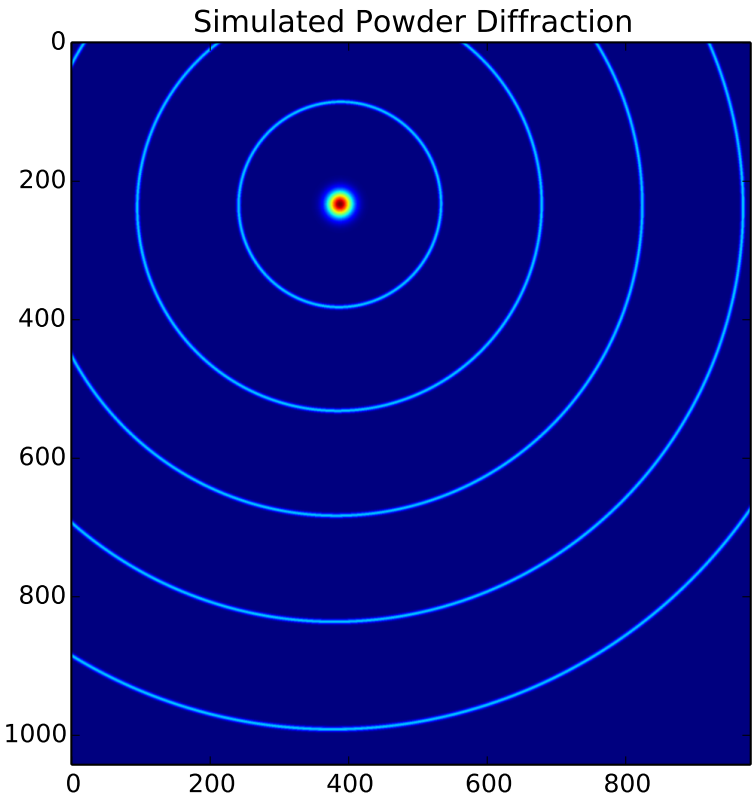
```
def calc_theta(r,theta,d,tilt,tiltdir):
    alpha=np.arcsin(np.sin(tilt)*np.sin(theta+tiltdir))
    lsquared=d**2 +r**2 -2*d*r*np.cos(np.pi/2+alpha)
    return np.arccos(-(r**2-lsquared-d**2)/(2*np.sqrt(lsquared)*d))
```

This angle θ then is calculated for every sub pixel in the sensor. This number then can be rescaled and rounded to the nearest integer in order to get unique integer labels for all the pixels. This labels are the index of the radial interval.

4.3.1 Tilt Angle Correction Test

To check if the tilt angle correction is working, lets create some fake calibration data, with the following peaks in the diffraction curve:





This was done with this configuration file:

```
{
    "PixelPerRadialElement": 1,
    "Imagesize": [
        1043,
        981
    ],
    "Tilt": {
        "TiltAngleDeg": -10,
        "TiltRotDeg": 73.569
    },
    "MaskFile": "emptymask.tif",
    "PixelSizeMicroM": [
        172.0,
        172.0
    ],
    "Wavelength": 1.54,
    "DedectorDistanceMM": 1031.657,
    "BeamCenter": [
        808.37,
        387.772
    ],
    "Oversampling": 2
}
```

Which amounts to a large tilt, lets see what Fit2d makes of it

```
INFO: SOLUTION 2
INFO: Best fit beam centre (X/Y mm) =   66.78356      138.9544
INFO: Best fit beam centre (X/Y pixels) =   388.2765      807.8745
INFO: Cone  1 best fit 2 theta angle (degrees) =   1.392646
```

```
INFO: Cone  2 best fit 2 theta angle (degrees) =   2.780810
INFO: Cone  3 best fit 2 theta angle (degrees) =   4.168858
INFO: Cone  4 best fit 2 theta angle (degrees) =   5.556975
INFO: Cone  5 best fit 2 theta angle (degrees) =   6.944765
INFO: Best fit angle of tilt plane rotation (degrees) =   73.59509
INFO: Best fit angle of tilt (degrees) =  -10.00601
INFO: Estimated coordinate radial position error (mm) =  0.7136476E-02
INFO: Estimated coordinate radial position error (X pixels) =  0.4149114E-01
```

Seems OK.

4.4 Polarization Correction

The polarization correction is expected to be small at small angles, but it is deemed important.

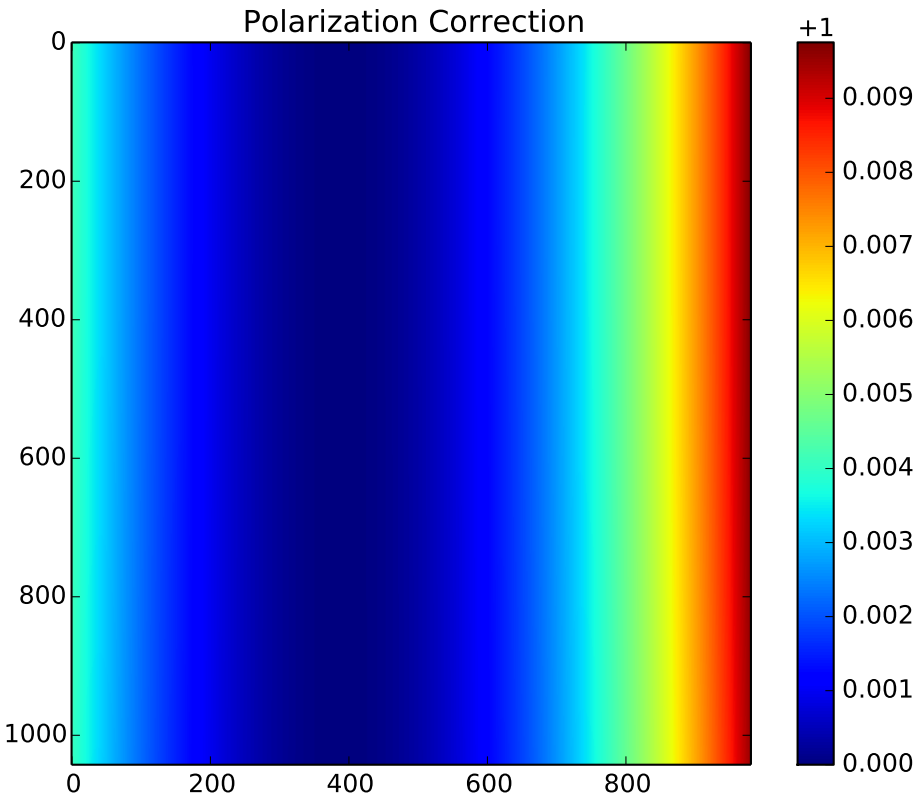
$$I_{cor} = I_j \left[P(1 - (sin(\phi)sin(2\theta))^2)(1 - P)(1 - (cos(\phi)sin(2\theta))^2) \right]$$

where ϕ is the azimuthal angle on the detector surface (defined here clockwise, 0 at 12 o'clock) 2θ the scattering angle, and P the fraction of incident radiation polarized in the horizontal plane (azimuthal angle of 90°) The polarization correction is configured by two parameters in *PolarizationCorrection*. Its factors are included in the integration matrix (operator).

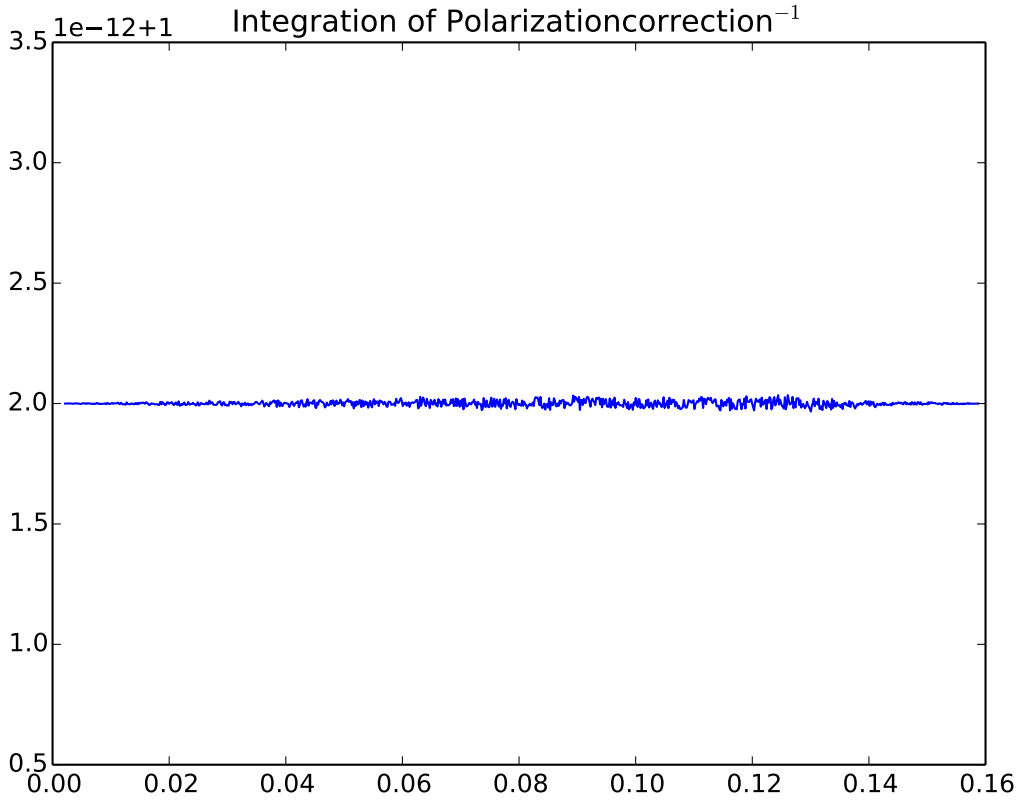
This input:

```
{
    "PixelPerRadialElement": 1,
    "Imagesize": [
        1043,
        981
    ],
    "Tilt": {
        "TiltAngleDeg": -0.56,
        "TiltRotDeg": 73.569
    },
    "MaskFile": "../data/AAA_integ.msk",
    "PolarizationCorrection": {
        "Angle": 0,
        "Fraction":1
    },
    "Oversampling": 2,
    "Wavelength": 1.54,
    "DedectorDistanceMM": 1031.657,
    "BeamCenter": [
        808.37,
        387.772
    ],
    "PixelSizeMicroM": [
        172.0,
        172.0
    ]
}
```

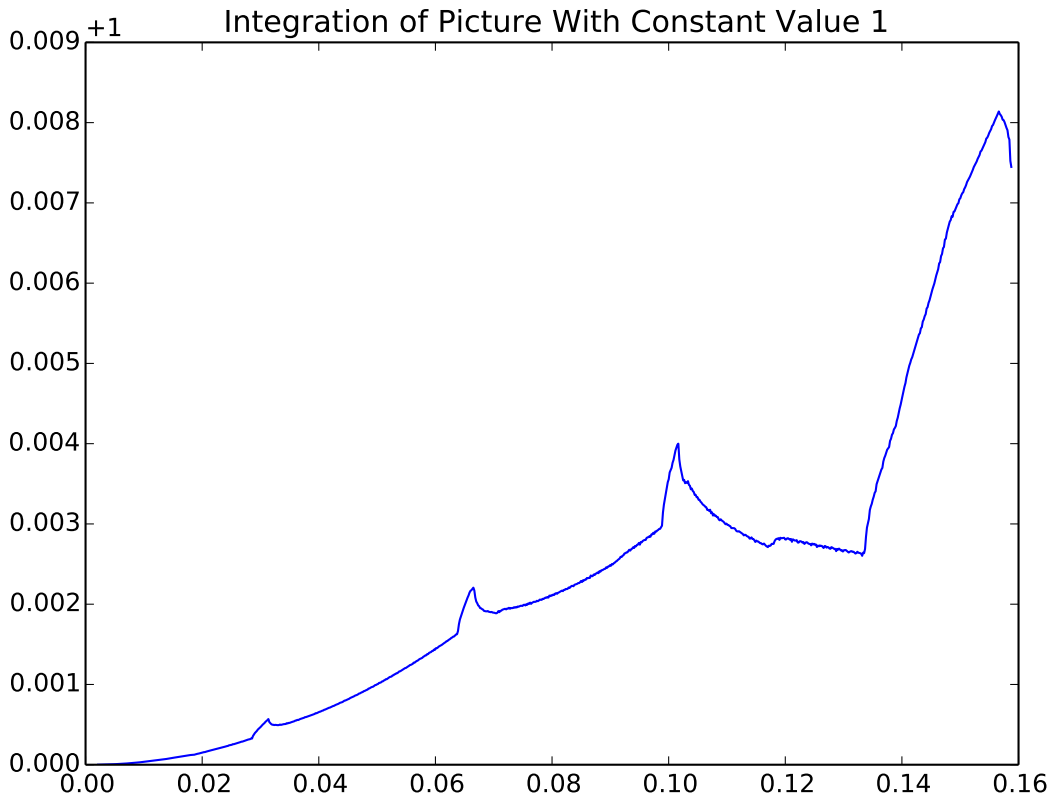
Gives:



If the correction factors are all correctly in the algorithm, the integration of an image containing $1/I_{corr}$ should give constant 1.0.



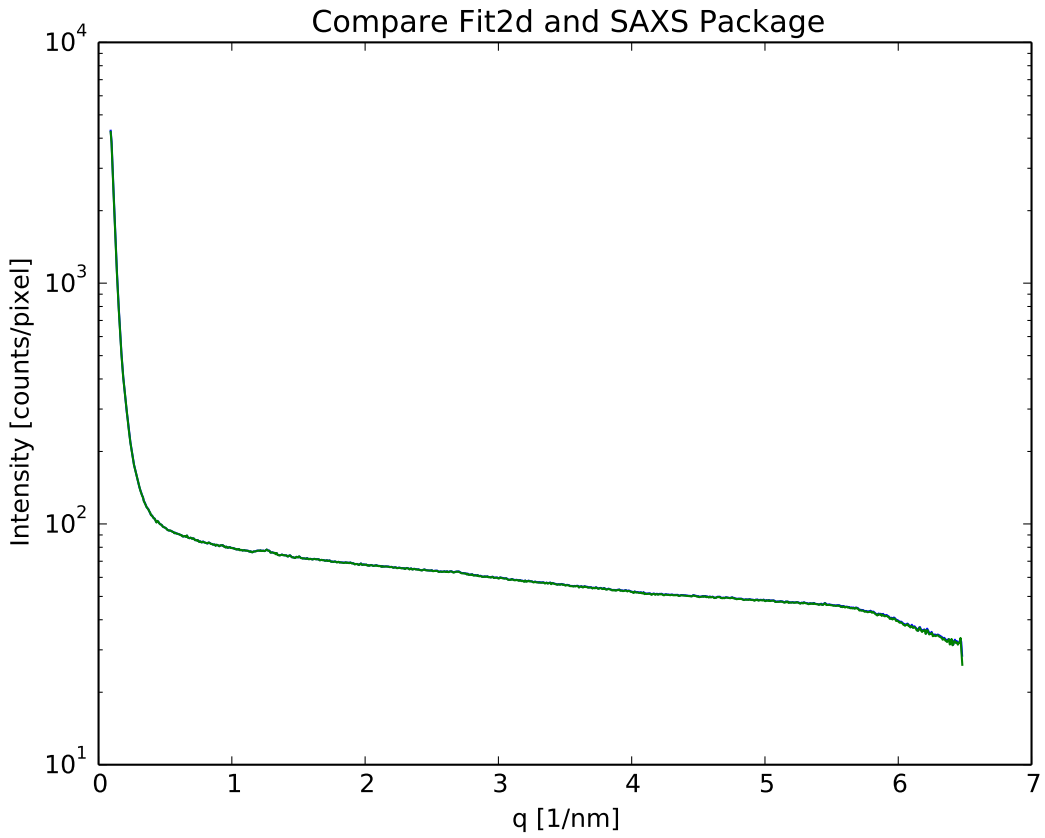
Just for checking: integrating a picture with only ones gives something different:



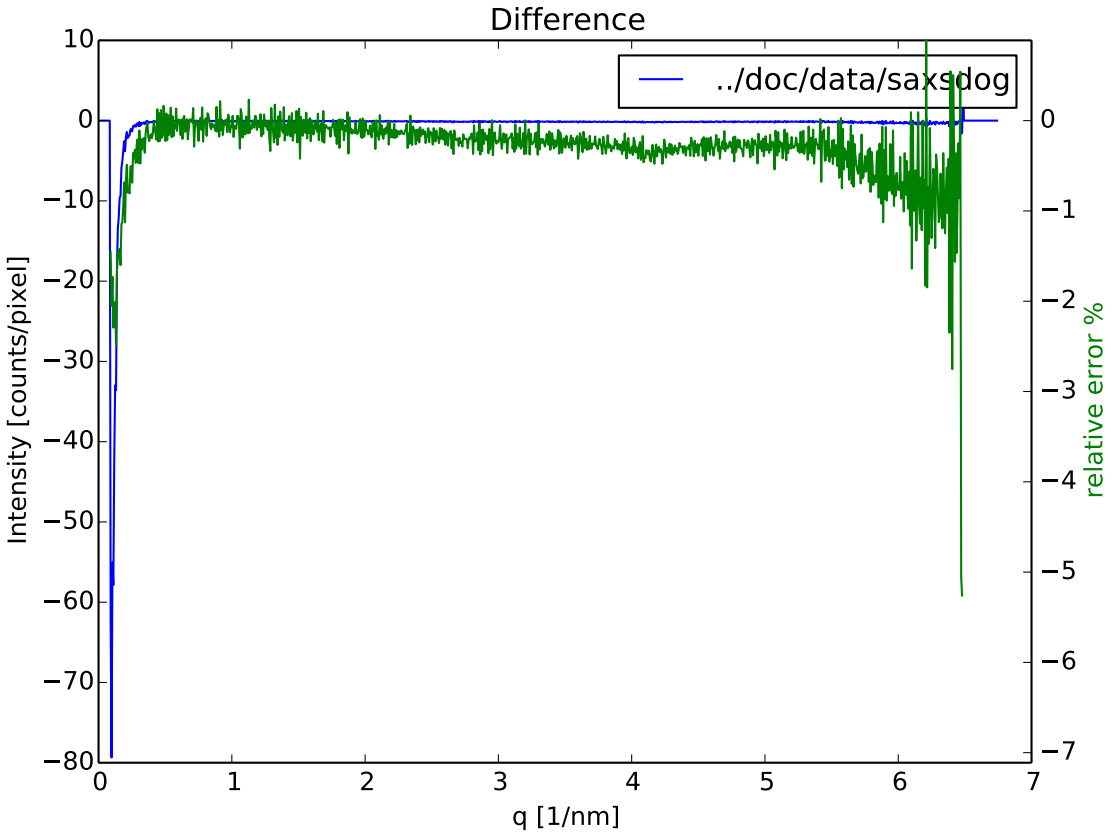
This are the wiggles that come from the polarisation corection pattern

4.5 Compare With Fit2d

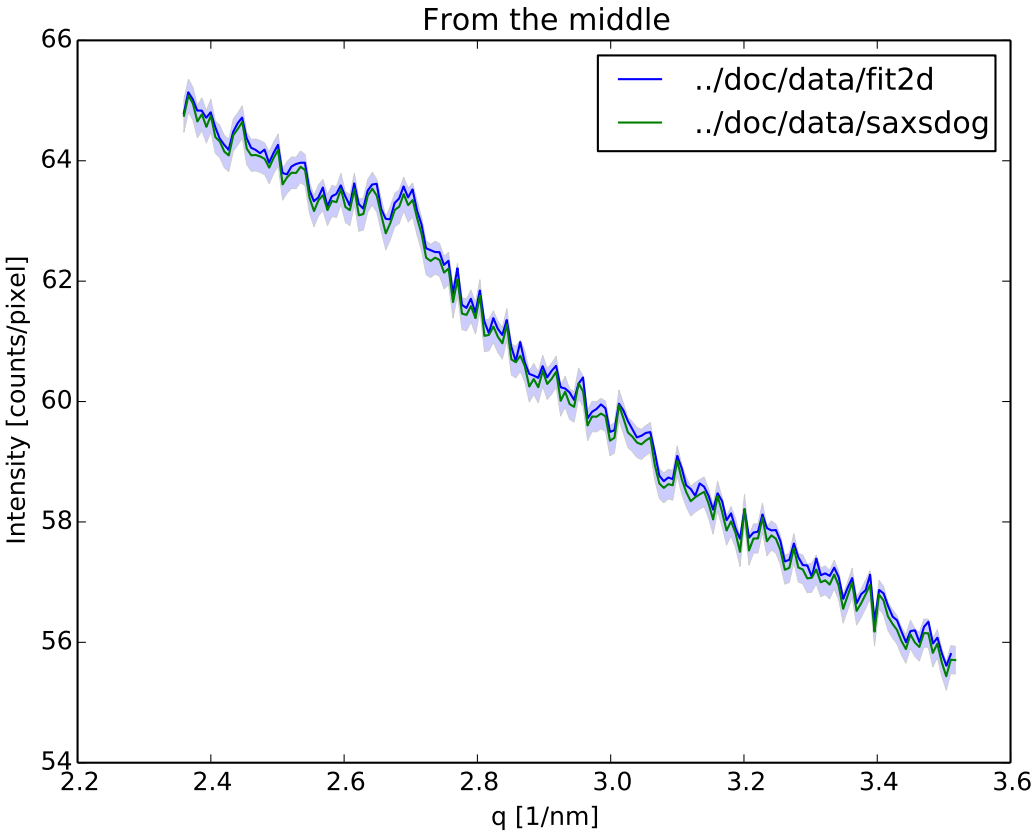
The program fit2d, which this package aims to partly replace, is the standard, so we better include a comparison plot here:

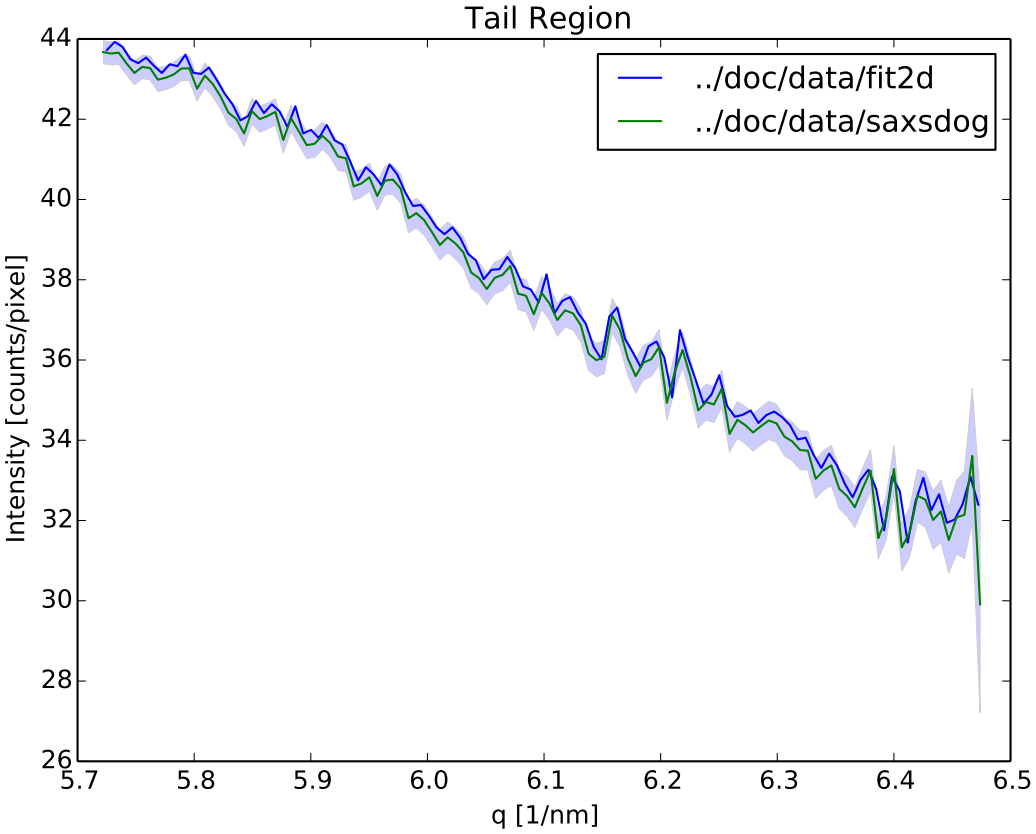


Okay, this doesn't show much but if we plot the difference:

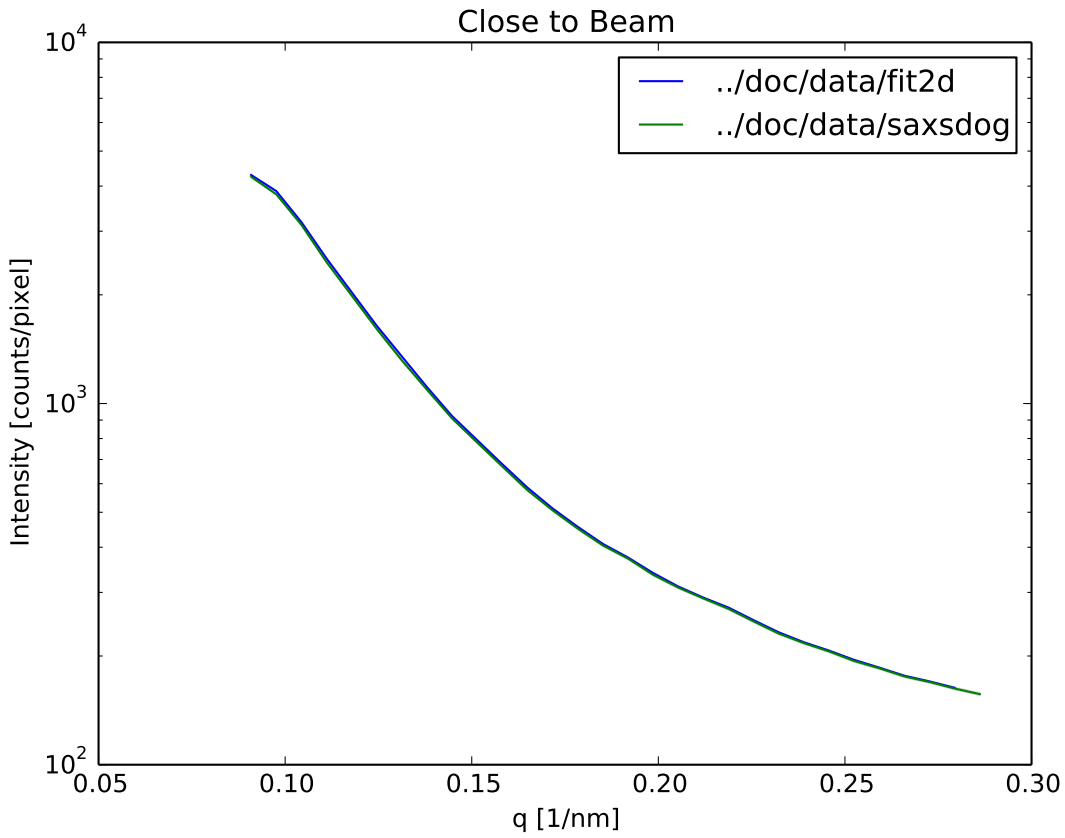


Still looks okay.





In the Tail region the blue halo (Poisson error) signifies that there are not enough counts to make good statistics.



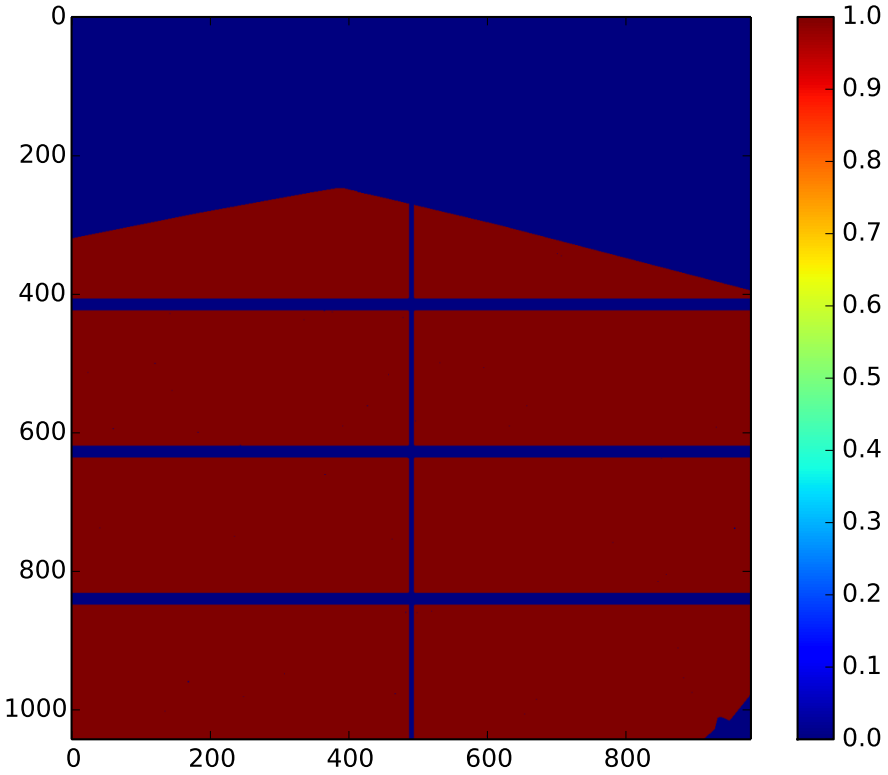
4.6 Integrating a Constant Image With Masked Values

This test shows that nothing wrong happens at mask borders. For thos we want to integrate an image that is one everywhere except for the masked regions

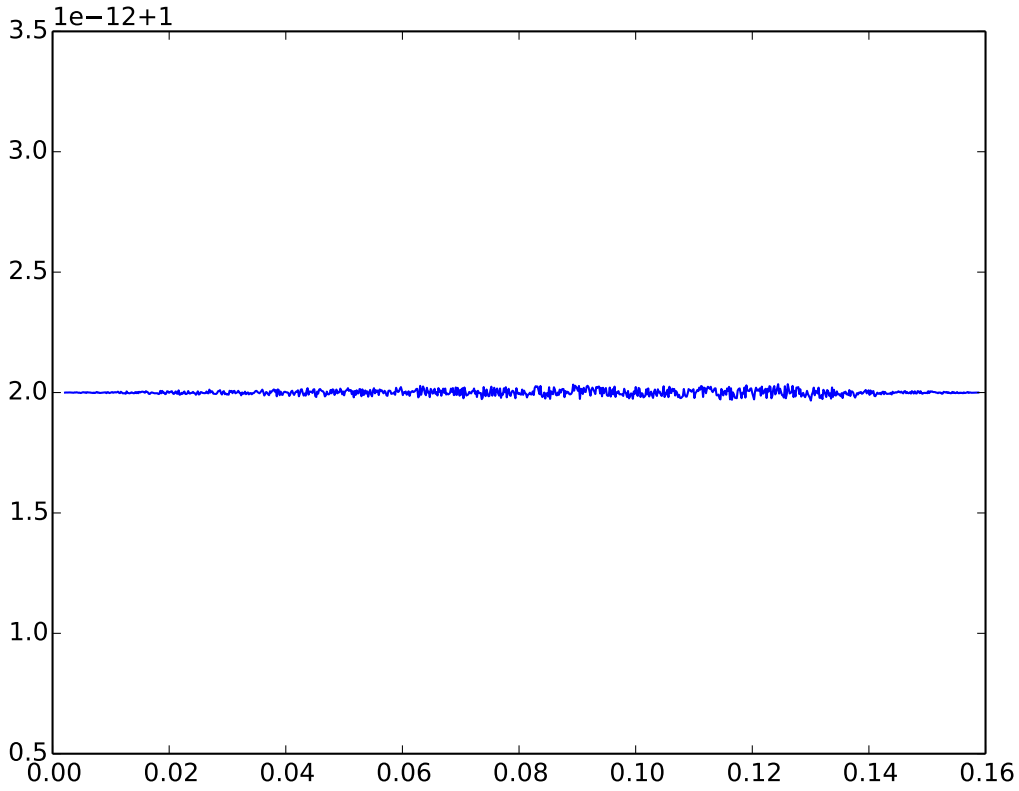
We use the following calibration without Polarization correction and mask:

```
{
    "PixelPerRadialElement": 1,
    "Imagesize": [
        1043,
        981
    ],
    "Tilt": {
        "TiltAngleDeg": -0.56,
        "TiltRotDeg": 73.569
    },
    "MaskFile": "../data/AAA_integ.msk",
    "PixelSizeMicroM": [
        172.0,
        172.0
    ],
    "Wavelength": 1.54,
    "DedectorDistanceMM": 1031.657,
    "BeamCenter": [
        808.37,
        387.772
    ],
    "Oversampling": 2
}
```

The image we are going to integrate is exactly the array the `SAXS.openmask()` returns:



The result is constant 1 (wher the intensity is not 0), save 2e-12.



Doing the same with Fit2d,

```
$ fit2d -svar\#IN=mask.tif -dim2000x2000 -svar\#OUT=data/const.chi -mac../data/AAA_integ_Pilatus1M
```

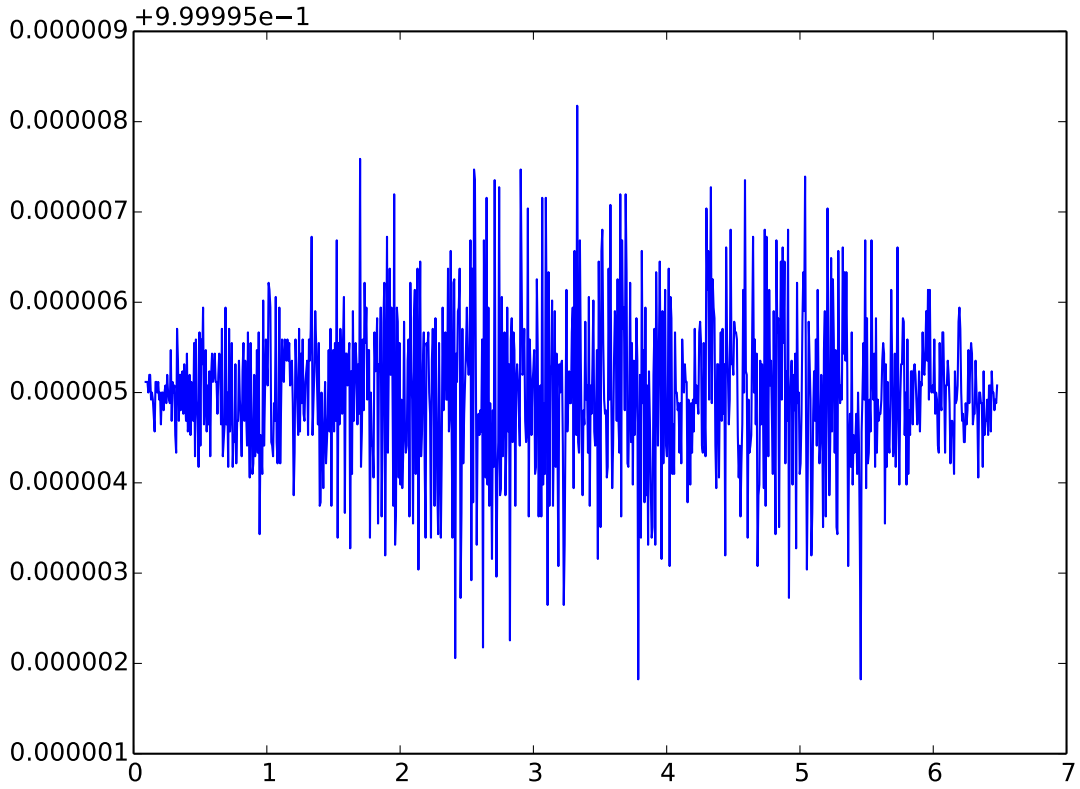
```
                        ---------------------------------
                        PROGRAM  FIT2D  Version: V12.081
                        _________________________________

        Copyright 1987-2005 Andy Hammersley / ESRF (hammersley@esrf.fr)

FIT2D: 2-D Detector Calibration/Correction; File re-formatting; 2-D Fitting

                        YOU CAN ALWAYS ENTER:   ?
...
```

results in something similar, just with less precision, about 10e-7. Probably because of single precission arithmetics.



4.7 Statistics

4.7.1 Poisson Statistics

The most important error is the statistical fluctuation that stems from the randomness of the scattering events. Counts of such events follow the Poison distribution. Such, the error (σ) is $\sqrt{n}$ for a count of n. The result of which is, that the relative error $\frac{\sqrt{n}}{n}$ rapidly gets small for larger counts.

Each Pixel in the SAXS sensor counts the number of events, and thus follows the Poisson statistics. The error of a sum of pixels is calculated as.

$$\sigma_{sum} = \sqrt{\sum_i \sigma_i^2}$$

which means here

$$\sigma_{sum} = \sqrt{\sum_i n_i}$$

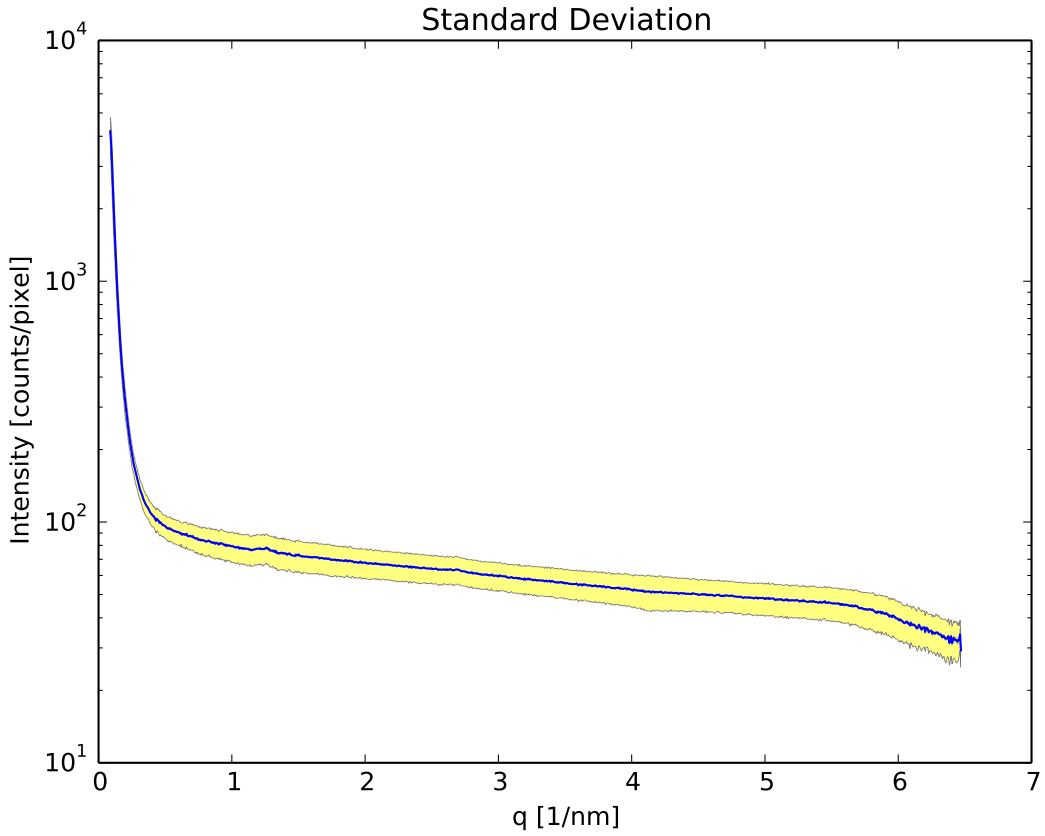
Rescaled over the number of pixels (P) in the sum this gives:

$$\sigma_{sum} = \frac{\sqrt{\sum_{i=1}^{P} n_i}}{P}$$

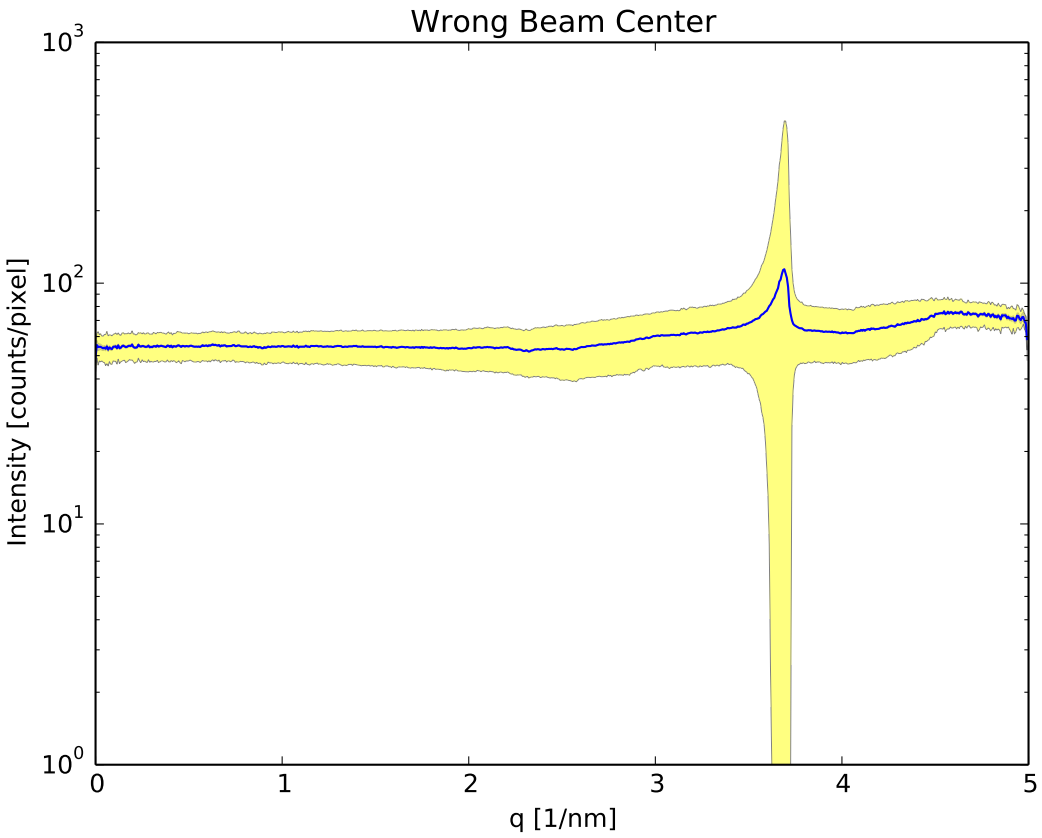
The `SAXS.calibration.plot()` method of the `SAXS.calibration` class will give you the Poisson error along with the standard deviation. So for regions, where the total number of counts is too small, you can see if there is a significant error. This might occur, if too few pixels are used for a data point or the intensity is just to small.

4.7.2 Standard Deviation

The standard deviation of the mean that is taken through the integration is not as such particularly useful to estimate the error of the resulting intensities because there are quite a few things that produce an angle dependence. In an optimal case, if the angle dependence can be corrected with the Polarization correction, the standard deviation of the integration might be very small. In an ordinary case the standard deviation gives you a measure of how spread the intensities within a radius interval are.



The standard deviation is bright yellow and the Poison error is blueisch



If the calibration is wrong you will for example see in the standard deviation. Like in this example. Here the beam center is wrong

CHAPTER

FIVE

SAXS MODULE API

class SAXS.**calibration**(*config*)

This class represents a calibration for SAXS diffraction. After initialization, the `integrate()` method can compute the radial intensity very fast.

Parameters **config** (*string*) – is the path to the *JSON Configuration File*:

integrate(*image*)

Integrate a picture.

Parameters **image** (*numpy.array(dim=2)*) – Sensor image to integrate as 2d *NumPy* array

Returns Returns Angle and intensity vector as a tuple (angle,intensity)

integratechi(*image*, *path*)

Integrate and save to file in "chi" format.

Parameters

- **image** (*np.array()*) – Image to integrate as numpy array
- **path** (*string*) – Path to save the file to

Returns Scattering curve data as numpy array

integrateerror(*image*)

Integrates an image and computes error estimates.

Parameters **image** (*np.array()*) – Image to integrate as numpy array

Returns The intensity the standard deviation and the Poisson statistics error in a numpy array.

plot(*image*, *outputfile=''*, *startplotat=0*, *fig=None*)

Plot integrated function for image in argument.

Parameters

- **image** (*numpy.array(dim=2)*) – Sensor image to integrate as 2d *NumPy* array
- **outputfile** (*string*) – File to write plot to. Might be any image format supported by matplotlib.
- **startplotat** (*integer*) – radial point from which to start the plot

SAXS.**calc_theta**(*r*, *phi*, *d*, *tilt*, *tiltdir*)

Calculates the difraction angle from pixel coordinates. It does work when called with arrays. See *The Geometry*

Parameters

- **r** (*float*) – Distance to beamcenter.
- **phi** (*float*) – Angle[rad] from polar sensor plane coordinates.
- **d** (*float*) – distance to difraction center.
- **tilt** (*float*) – Angle[rad] of sensor plane tilt.

- **tiltdir** (*float*) – Angle[rad] of direction of tilt.

Returns theta

SAXS.**scalemat**(*Xsize*, *Ysize*, *ov*)
Computes a scaling projection for use in computing the pixel weights for integration

Parameters

- **Xsize** (*int*) – Picture size in X direction.
- **Ysize** (*int*) – Picture size in Ydirection.
- **ov** (*int*) – Number of oversampling ticks in x ynd y direction
- **corr** (*array*) – Polarizationn an other correction factors

Returns sparce matrix toing the scaling

SAXS.**openmask**(*config*)
Open the mask file especialy the *.msk file. Unfortunately there is no library module for msk files available also no documentation. So, for the msk file, we have a very brittle hack it works for our sensor. Nevermind any other resolution or size.

Parameters **config** (*object*) – Calibration config object.

Returns Mask as logical numpy array.

SAXS.**convert**()
This implements the functionality oft *The Converter*. It parses the commandline options and converts the Fit2d info file to the JSON data used by the SAXS.callibration class.

SAXS.**saxsdog**()
This implements the functionality of *The Saxsdog*

class SAXS.**imagequeue**(*Cal*, *options*, *args*)
This class keeps a queue of images which may be worked on in threads.

Parameters

- **Cal** (*SAXS.calibration*) – The SAXS Calibration to use for the processing
- **options** (*optparser*) – The object with the comandline options of the saxsdog
- **args** (*list*) – List of command line options

fillqueuewithexistingfiles()
Fill the queue with the list of images that is already there.

start()
Start threads and directory observer.

C

F

I

O

P

S