NTNU – Trondheim
Norwegian University of
Science and Technology

A workflow for Diffraction/Scattering Computed Tomography using the XRDtoolkit.

Presenting the easy to use Python module for online XRD processing.

Amund Hov

Master of Science in Physics and Mathematics
Submission date: September 2013
Supervisor: Ragnvald H. Mathiesen, IFY
Co-supervisor: Olof Svensson, ESRF
Veijo Honkimaki, ESRF

Norwegian University of Science and Technology
Department of Physics

Abstract

We present an easy to use data analysis framework for X-ray diffraction data. It is aimed at users and scientists for online analysis at the beam line or offline anywhere. As an example application we present and demonstrate a workflow for Diffraction/Scattering Computed Tomography data, covering all steps from detector tilt calibration to sinogram assembly, filtering and reconstruction. Finally we discuss the effects of sample self-absorption on quantitative analysis and propose some solutions that could be investigated in further work.

Foreword

The work for this master thesis which is the final requirement for the Master's degree of Technical Physics at the Institute of Physics at the Norwegian University of Science and Technology was started during the summer of 2012 as I stayed three months at the ESRF as an intern.

The work has consisted of planning and implementing a module in python for providing the necessary functions and methods for analysing X-ray diffraction experiments. As a demonstration of the module's capability an experiment was done at the the end of june. The experiment was performed with a dummy sample specially prepared for Diffraction/Scattering Computed Tomography as done by ID15. Finally the last two months have been spent on the presentation of the work in this thesis.

Before continuing I would like to thank my supervisor at NTNU **Ragnvald Mathiesen** and **Veijo Honkimaki** for providing me with the opportunity to return to the ESRF in march of 2013 as the final part of my master program.

Many thanks goes also to beamline scientist Marco di Michiel as operator of ID15. His suggestions for samples and help with the practical execution of the experiment itself is very much appreciated.

I owe Simon Jacques my gratitude for the use of some of the valuable beamtime, but also for their encouragement and interest in the data processing i was doing.

In addition i would like to extend my thanks to Jerome Kieffer, Olof Svensson, Claudio Ferrero and the rest of the ESRF software group for their feedback and suggestions on the software technical side.

Finally I would like to thank **Natouk Kurdjian** for her support and understanding while writing this report - as well as for keeping me fed.

Contents

Glossary

CPU Central Processing Unit. 28

Diffraction/Scattering Computed Tomography Term coined by Bleuet[4] to disambiguate from the technique of Diffraction contrast Computed Tomography (XRD-CT) as performed by e.g. Ludwig et al.[13]. iv, 1, 8, 9, 21, 23, 25, 34, 42, 57, 58, 64

ESRF European Synchrotron Radiation Facility. 43

FEM Finite Element Method. 7

GIL Global Interpreter Lock. 34

GPU Graphics Processing Unit. 28, 55

Chapter 1

Introduction

1.1 Motivation

When doing X-ray diffraction experiments it is often necessary to perform several steps of data reduction and analysis in order to evaluate a result. In some cases data could even be found to be completely useless upon analysis after the experiment has concluded. Online analysis at the beamline can be useful in such cases, but for this to be feasible the software used must be easy to operate and reasonably fast.

Furthermore, users of X-ray radiation are not necessarily specialists in neither X-ray diffraction or scientific computing. It would be useful for sychrotrons and other facilities to be able to provide their users not only their data, but also the tools to process them. The python modules proposed for this thesis would be a modest attempt at starting to covering some of these use cases.

Its aim is to be easy to install, use and extend for users and scientist having to process x-ray diffraction data in the line of their research. To that end we have spent considerable time making sure that the software is independent of the computing infratructure of the ESRF, easy to operate and reasonably fast.

Chapter 2 gives a short introduction to X-ray diffraction and tomography, with a focus on the exciting new technique of Diffraction/Scattering Computed Tomography .

In chapter 3 we establish the necessary theory underlying our least squares fitting, estimation of uncertainties, sinogram reconstruction and sinogram correction filters. Finally we discuss the problem of sample self-absorption in Diffraction/Scattering Computed Tomography and propose some solutions to try for later work.

In chapter 4 we start to present the software that was written over the course of the spring and summer before finaly in chapter 5 and 6 we describe our experiment and the results we obtained there.

At last we discuss the performance of our workflow and what might be expanded upon in further work.

Chapter 2

X-ray diffraction and Tomography

X-rays are able to penetrate bulk material that are otherwise opaque at optical wavelengths. Systems using coupled X-ray sources and detectors are therefore common in a lot of dfferent applications ranging from security and customs checkpoints for looking inside packaging, clothing and other materials in search of contraband and illegal objects, to medical imaging of the human body, industrial applications such as non-destructiv tesing and of course materials research. In this chapter we will concern ourselves with X-ray scattering and absorption and how it is exploited in computed tomography.

Assuming neglegible effects from dynamical diffraction theory the interaction of X-ray with matter can be viewed as a series of scattering and absorption events. As a beam of X-rays passes through matter its intensity is thus reduced as radiation is absorbed by the atoms or scattered in other directions. Normally we describe this attenuation by its mass attenuation coefficient (μ/ρ) where ρ is the material density. In general the coefficient is a function of beam energy and chemical composition and combines a range of different physical processes, each of which have a certain probability to occur governed by their scattering cross sections σ.

> **Photoelectric absorption**. The energy of the photon is sufficient to eject an electron from the scattering atom. Remaining energy is converted to kinetic energy of the electron.
>
> **Rayleigh scattering**. The electric field of the radiation couples with that of the electrons and is deflected in all possible directions without any change in energy. The process is said to be elastic.
>
> **Compton scattering** makes the photon impart some of its energy

and momentum to a weakly bound electron. The photon is both shifted in energy and direction.

Pair production. At energies above 1.02 MeV (twice the rest mass of the electron) the photons have enough energy to produce a virtual electron-positron pair. Alone the photon cannot produce this pair due to momentum considerations (a electron-posittron annihilation requires two photons to carry momentum), but close to a nucleus it can interact with virtual photons of the coulomb field imparting some momentum.

Of these only rayleigh scattering produces coherent radiation. With coherency we mean that there is a fixed relationship between the phase of incoming and outgoing radiation. This means that when summing up the contributions from multiple scattering centers the combined intensity is a vector sum; the individual waves can interfere with another in a constructive and/or destructive manner. Significant effects only present themselves when the wavelength of the radiation is comparable to the repetition distance in the scattering geometry. Typical interatomic distance is of the order of 1Å with repeating units every 10Å in a typical crystalline solid. When a large number of scattering atoms are arranged in a diffracting geometry the result are sharp peaks of high intensity punctuated by large regions of destructive interference. These peaks are called bragg peaks and occur only when the path difference of each scattering center is a whole multiple of the wavelength. This is called the bragg condition.

Amorphous materials, such as liquids and glasses exhibit only short-range order and scatter X-rays over a wide range of 2θ angles. This means the bragg condition is seldom satisfied and there are noe bragg peaks, but a smoothly oscillating scattering intensity. Instead the amorphous phase is described by its *pair distribution function* which is a measure of the probability to find an atom in the vicinity of another as a function of radial distance.

Returning to our discussion about attenuation, it will be useful to decompose the attenuation coefficient into two parts

$$\mu_{eff} = \mu_{abs} + \mu_{scat} \tag{2.1}$$

where μ_{abs} represents all effects which would absorb radiation and keep it in the sample and μ_{scat} refers to radiation that is lost to the direct beam, but which can be recorded coming from the sample at some angle 2θ. To relate the intensity of our beam before and after passing through a sample we postulate that the attenuation at a point is directly proportional to the beam intensity and attenuation coefficient, that is, the change in intensity dI when passing through a slice of thickness dx is

$$dI = -I(x)\mu(x)dx.$$

The solution to this ordinary differential equation is known as the Beer-Lambert law and is given by

$$I = I_0 \exp(-\int \mu dx) \tag{2.2}$$

for a monochromatic, narrow beam.

2.0.1 Synchrotron radiation

Whenever charges are accelereated they give of radiation. Synchrotron radiation referes to the radiation produced when charges are accelerated in a direction normal to their velocity, that is, when they are made to change direction. In a synchrotron this effect is exploited to generate bright beams of radiation which can be used to probe materials.

In simple terms the operation of a synchrotron is as follow. Bunches of electrons are produced and subsequently accelereated by a linear accelerator (LINAC). The bunches are typically further accelerated in a smaller synchrotron before being injected into the storage ring.

The storage ring has been evacuated for air and can hold the charges for extended periods of time, usually hours, while giving off synchrotron radiation. In fact, the ring is not really a ring, but a sequence of segments connected by sections where bending magnets direct the charges to the next segment. It is at these bending sections that radiation is produced. In addiation 3rd generation synchrotrons typically employ devices called wigglers and undulators which forces the electrons to accelerate up and down on parts of the otherwise straight segments. The force with which the electrons are

Relativistic collimation of beam. Strong relativistic effect ($\gamma >> 1$). In the reference system of the electron it is accelerated up and down, radiating with power distributed like the idealized dipol antenna. Due to the large lorentz factor γ the synchrotron ring is contracted in the direction of travel. Equivalently, in the laboratory system, the radiated beam is elongated in the direction of travel, collimating it in the plane of the ring.

Synchrotron facilities are big, complex and expensive machines, but the unique properties of its radiation more than pays for it, as is evident by their commonplace in the world. The benefits of Synchrotron radiation can be summarised as

- **Coherent radiation**
- **Low emittance** The electron beam is confined leading to a concentrated x-ray beam, high brightness.
- **High luminosity**. High photon count, enabling shorter exposures and/or better statistics.
- **Flexbility** White beam can be used for max luminosity, or tuned to desired wavelength using crystal monochromators.

In addiation the pulsed operation with electron bunches enables one to freeze the sample in time at the nanosecond, much like the operation of a flash on a camera affords high-speed photography.

The work for this master thesis was performed at the ID15A[2] beamline at the European Synchrotron Radiation Facility in Grenoble. This particular beamline specialises in high-energy and high-resolution, ultra fast μ-tomography. High-energy in this context refers to radiation in the range $30 - 500\,\mathrm{keV}$, which is quite exotic considering X-rays are said to be *hard* (highly penetrating) from $10\,\mathrm{keV}$ onwards. A collection of monochromators, slits and refractive lenses allow tuning the wavelength and shaping the beam for each experiment.

2.1 Computed tomography

Computed tomography is a quite recent technique enabled by modern computation which has found great use in medicine and industry since it allows non-invasive characterisation of bulk materials in three dimensions. Other techniques for studying the chemical and structural properties of materials generally requires either preparing the sample by slicing and polishing, possibly damaging the microstructure, or they are restricted to look at global features and surfaces.

Computed tomography can be used for characterisation of materials on a wide range of scales depending on the nature of the material and properties to be studied. On the very small end, μ-tomography is useful in materials science where the relation between micro-structure and macroscopic properties often is essential.

Finally, using synchrotron radiation which have near perfect parallel and monochromatic beams, the reconstructions are free from geometrical or hardening effects and can be used quantitively as there is a direct correspondence between the reconstruction and corresponding physical property in the material[15]. The reconstructed volume can then be used to perform local characterisation or as input for Finite Element Method (FEM)-routines to solve for mechanical, thermal and other properties of the material.

2.1.1 Absorption tomography

Since the refractive index of most materials is virtually equal to unity at X-ray energies one usually works with the assumption that the beam follow straight lines in the sample. The attenuation in the material depends on material density and composition. In hospitals large machines producing X-ray radiation irradiate the body from different angles and records the attenuation as the beam travels through it. Having recorded the attenuation from different angles reconstruction algorithms are used to compute the density inside the body which would produce the registered pattern. There are several methods for doing the reconstructions which we describe briefly in the analysis section.

2.1.2 Phase contrast tomography

With the highly coherent radiation afforded by synchrotron facilities it has become possible to record the difference in phase that X-rays xperience when passing through a sample. Phase contrast tomography can provide extra contrast in materials which are not that absorptive or which provide poor absorption contrast. Internal structures of micro-sized fossils have been studied in this manner.

The sample is assumed to consists of phases with refractive indices

$$n_i = 1 - \delta_i + i\beta_i. \tag{2.3}$$

A plane monochromatic wave

$$\exp(i\frac{2\pi}{\lambda}z)$$

traveling in free space will be modified to

$$\exp(i\,n_i\frac{2\pi}{\lambda}z)$$

when made to travel in the phase corresponding to n_i. Written out this leads to the the following xpression for the amplitude

$$\exp(i(1-\delta_i)\frac{2\pi}{\lambda}z)\exp(-\beta_i\frac{2\pi}{\lambda}z).$$

The first part is a pure retardation $(-\delta)$ of the phase as it travels in the material compared to the virtually free propagation in air. The second part is a attenuation of the beam which in intensity corresponds to

$$|\exp(-\beta_i \frac{2\pi}{\lambda} z)|^2 = \exp(-\beta_i \frac{4\pi}{\lambda} z). \tag{2.4}$$

Equation 2.4 gives us a direct relation to Beer-Lambert's law 2.2 that we used in absorption tomography. The relation to the attenuation coefficient is thus

$$\mu = \beta \frac{4\pi}{\lambda}. \tag{2.5}$$

The phase retardation δ is essentially proportional to electron density (p. $30 - 31$)[6].

2.2 Diffraction/Scattering tomography

The combination of diffraction/scattering and tomography is a very recent development made possible by advances in X-ray optics which enables probing samples with micro- or even nano-focused beams.

Unlike absorption and phase contrast tomography our signal is not the direct beam, but the scattered or diffracted part that we defined for μ_{scat} in equation 2.1. The technique is sensitive to the local crystalline structure of the sample, so one could say that it provides 2θ or d-spacing contrast. This makes it particularily suited for studying complex, poorly ordered materials composed of various amorphous and poly-crystalline phases. Furthermore, no a-priori knowledge is required for analysis.

We still use a $2D$-area detector to record the data, but since the diffraction patterns from different locations in the sample would be overlapping we need to scan the sample with a pencil-beam one projection at a time. The resolution of the final reconstruction is then directly associated with the size of the probe. If a resolution of $100x100$ is desired for instance, the beam should not be much larger than one hundredth of the sample size. To record one slice in a Diffraction/Scattering Computed Tomography experiment it is necessary to perform a full scan across the sample for each projection which can take hours depending on the available flux and how well the sample is diffracting. The large amount of data is not a problem for storage in these modern times, but it means loading and processing the data can be cumbersome.

In contrast the typical resolution of a absorption contrast tomograph is dependent on the effective pixel size of the imaging detector and a full $3D$

volume can be reconstructed after a 180° scan of the sample. In certain ultra fast tomography setups this can be done in less than a second.

However, the Diffraction/Scattering Computed Tomography probe can easily be combined with for instance fluorescence enabling simultaneous multimodal tomography.

Assuming that each voxel provides the same scattering power for each angle we can reuse the principles of absorption tomography after an azimuthal integration of the diffraction pattern. In practice this requires that the crystallites in the sample be much smaller than the probe size so that a statistically meaningful number of them are available at different orientations in the volume. This in effect puts a practical limit on the technique for poly-crystalline samples.

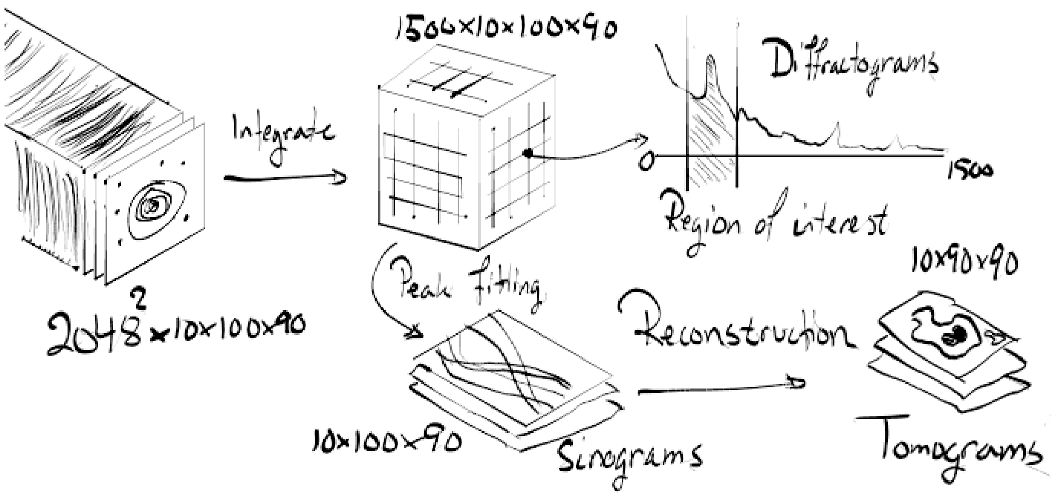


Figure 2.1: Overview of the data reduction workflow. A series of diffraction images taken with a 2048^2 resolution is reduced to a stack of density tomograms through several steps: First azimuthal integration, then assembly of sinograms from selected phase or region of interest and finally reconstruction of the assembled sinograms.

A demonstration of feasability of the technique for medical imaging was done by Kleuker et al.[12] in 1998 using a sample of soft tissue with bones, muscle and fat. But already the the technique has been applied to various samples such as Portland cements [18], pigments [7], teeth [8] and catalyst bodies [14].

The name itself was coined by Álvarez-Murga, Bleuet and Hodeau [4] to distinguish it from techniques such as diffraction computed tomography as perfomed by Ludwig et al. Parts of this section are based on their paper

"Diffraction/scattering computed tomography for three-dimensional charactierization of multi-phase crystalline and amorphous materials", and the reader is encouraged to read their paper in full for a thorough treatment of the technique and its advantages compared to more traditional imaging and characterisation techniques. Its most attractive feature is perhaps the ability to perform in-situ analysis in $3D$ combined with the power of diffraction based techniques.

2.2.1 Reverse analysis

If the sinograms are properly corrected for air scattering and normalized so that the levels outside the field-of-view of the object are consistent it is possible to do what is called reverse analysis. Instead of fitting single peaks and doing reconstructions from there as we have done one performs reconstructions on $(r, \omega)_{2\theta}$ for each and every 2θ angle. This then explains each contribution to this scattering angle as a function of the voxel (x, y) in sample. Another way to look at this is to look at all the contributions $(x, y)_{2\theta}$, which is in effect a reconstructed powder $1D$ pattern at the voxel level.

The quality of the $1D$ patterns extracted from reverse analysis can be good enough to perform crystallographic analysis [5]. This opens up all sorts of possibilities to use established diffraction techniques such as rietvald refinement, pair distribution function analysis and le Bail fitting on a voxel level or averaged over a region of interest, something which can not be done through X-ray diffraction tehcniques or computed tomography alone. A nice example of this is the paper "Non-invasive imaging of the crystalline structure within a human tooth" by Egan et al. which uses reverse analysis to map variations in lattice parameters, preferred orientations, organic content, chemical composition and other parameters.

Chapter 3

Data analysis

This chapter presents the theory and methods used in the software section. First we introduce the method of least squares which is the most common method of solving optimisation problems. Then we go through different strategies of reconsctructing tomographs from a set of projections.

Finally we discuss corrections for improving the quality of our sinograms and the effect of sample self-absorption.

3.0.2 The method of least squares

In the method of least squares one tries to fit a series of observations $\{y_0, \ldots, y_n\}$ to a model $A(c_0, \ldots, c_m)$ by minimizing their squared error. The observations are modeled as stochastic variables

$$y_i = A \cdot c + \epsilon_i$$

where c is the vector of model parameters to be estimated and ϵ_i is the observational error. There needs to be at least as many observations n as there are parameters m. If the variance-covariance matrix Σ_y of the observations is known one can write the generalized distance between model and observations as

$$S = (\mathbf{y} - \mathbf{A} \cdot \mathbf{c})^\intercal \mathbf{\Sigma}_y (\mathbf{y} - \mathbf{A} \cdot \mathbf{c})$$

The estimator that best minimizes this is given by

$$\hat{\mathbf{c}} = min_c(S) = \mathbf{\Sigma}_c \cdot \mathbf{A}^\intercal \mathbf{\Sigma}_y^{-1} \cdot \mathbf{y} \tag{3.1}$$

where

$$\mathbf{\Sigma}_c = (\mathbf{A}^\intercal \cdot \mathbf{\Sigma}_y^{-1} \cdot \mathbf{A})^{-1} \tag{3.2}$$

is the estimated variance-covariance matrix of the parameters $\hat{\mathbf{c}}$. The ordinary least squares estimator

$$\hat{\mathbf{c}} = \mathbf{A}^\intercal \mathbf{\Sigma}_y^{-1} \cdot \mathbf{y}$$

most commonly cited is recovered by setting all off-diagonal elements of $\mathbf{\Sigma}_C$ to zero and all variances equal unity (the identity matrix). A special case called weighted least squares is obtained when all off-diagonal elements (the covariances) are set to zero. The inverted matrix is then

$$\mathbf{\Sigma}_c^{-1} = \begin{bmatrix} 1/\sigma_0^2 & 0 & 0 & \dots & 0 \\ 0 & 1/\sigma_1^2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1/\sigma_i^2 \end{bmatrix}$$

weighting each observation in the regression according to its uncertainty where lower variance leads to higher weights. This is the form we have used in our peak fitting and calibration routines.

3.0.3 Estimates of variance

For a lot of our methods we use diffraction images subtracted for dark current. We can model the stocasthic process of registering photons on the detector as a poisson process. The uncertainty of such a process is given by

$$\Delta I = \sqrt{n}$$

. We modify this to

$$\Delta I = \sqrt{n+1}$$

for our use since an uncertainty of 0 for $n = 0$ is not helpful when using the inverse uncertainty as weights in the weighted least square regression.

Each corrected image is then the combination of two poisson processes and must be combined using Gauss' law of propagation of uncertainty. Suppose we have a function $S(c)$ where the variance-covariance matrix Σ_c is known.

Defining the jacobian

$$J = (\frac{\partial s}{\partial c_1}, \frac{\partial s}{\partial c_2}, \dots)$$

the propagated uncertainty is given by

$$(\Delta S)^2 = J\,\Sigma_c J^\intercal. \tag{3.3}$$

For the background subtraction $S = n_{img} - n_{dark}$ and

$$(\Delta S)^2 = (\Delta n_{img})^2 + (\Delta n_{dark})^2 = n_{img} + n_{dark} + 2$$

which we modify to

$$(\Delta S)^2 = n_{img} + n_{dark} + 1$$

.

When dark current counts are not provided or in the case of an integrated profile were all the original counts would have to be taken into account we fall back to a constant bias being the median value of the background image. In our code this has been hand-coded for the Perkin-Elmer detector, which should be extended for other detectors.

3.1 Reconstruction algorithms

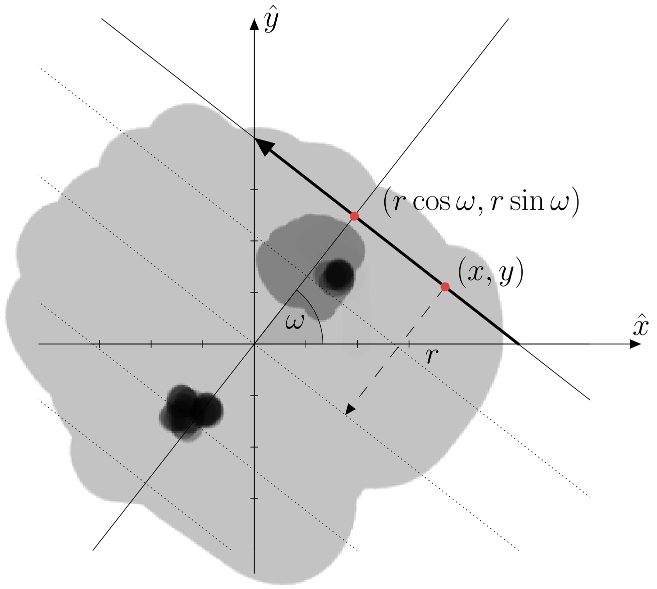


Figure 3.1: Coordinate system of rays going through the sample at an angle ω and displacement r.

First we define our coordinate system as we rotate and scan the sample. We assume that the sample is fixed and aligned to the coordinate axes $\hat{x}$ and $\hat{y}$ as in figure 3.1. Assuming that we rotate about the center of the object, the rays penetrate the object with their normal vector $\hat{r}$ forming an angle ω with the x-axis. In the figure we have hilighted one of these rays which has been translated a distance r from the center of the object.

The equation of the line l can be written on vector form as

$$l: \ \mathbf{r} \cdot ((x, y) - (r \cos \theta, r \sin \theta) = 0$$

since any segment on the line should be normal to the translation vector $\mathbf{r}$. Writing out $\mathbf{r} = (r \cos \theta, r \sin \theta$

$$(r \cos \theta, r \sin \theta) \cdot ((x, y) - (r \cos \theta, r \sin \theta) = 0$$

which implies that

$$(r \cos \theta x - r^2 \cos^2 \theta) + (r \sin \theta y - r^2 \sin^2 \theta) = 0$$

. Collecting terms

$$rx \cos \theta + ry \sin \theta = r^2(\cos^2 \theta + \sin^2 \theta)$$

and due to the identity $\cos^2 \theta + \sin^2 \theta \equiv 1$

$$x \cos \theta + y \sin \theta = r \tag{3.4}$$

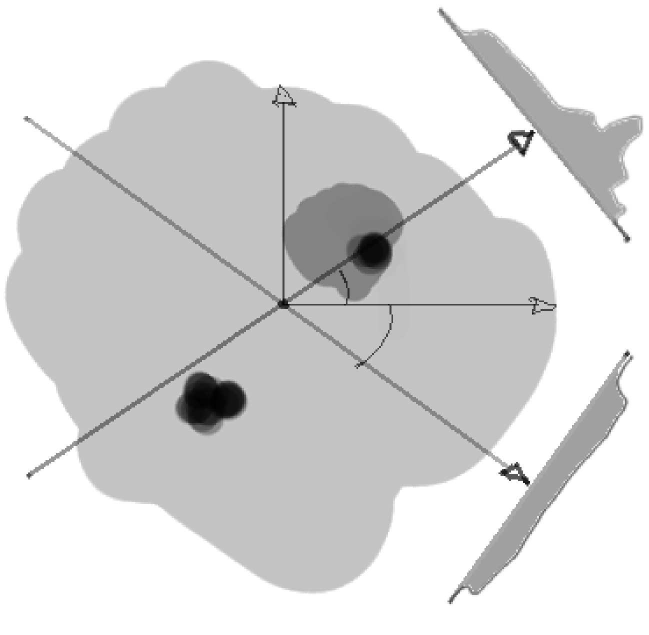


Figure 3.2: Rays passing through sample $f(x, y)$ at differnt angles and their profile $f_{(0,\omega)}$ displayed.

Figure 3.2 shows the profile of two rays as they pass through the center of the sample $f(x, y)$ at some angles ω_1 and ω_2. The projection of $f(x, y)$ along the ray is given by the line integral

$$p(r, \omega) = \int_{l \in (r,\omega)} f(x, y) dl \tag{3.5}$$

where l takes us through all the coordinates (x, y) of the line. Using our relation in 3.4 we can rewrite this to

$$p(r, \omega) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} \delta(x \cos\theta + y \sin\theta - r) f(x, y) \mathrm{d}x \mathrm{d}y \tag{3.6}$$

This equation is called the Radon transform of $f(x, y)$ and is the mapping between the cartesian system (x, y) of the sample and the parameters space (r, ω) of the sinogram system. If one had a perfect knowledge of the projected values one could in theory perform the inverse radon transform on the sinogram to recover f.

By the projection of a sample one usually means all the line integrals $p(r)_\omega$ for a fixed angle. The fourier slice theorem[6] tells us that there is a direct correspondence between the fourier transform of the projection $p(r)_\omega$ and the fourier transform of the original function $f(x, y)$ or more specifically that

$$\mathfrak{F}\{p(r)_\omega\} = \mathfrak{F}\{f(r \cos\omega, r \sin\omega)\}.$$

Each fourier transformed projection thus corresponds to a line in the fourier spectrum of the function $f(x, y)$. The fourier method of reconstruction builds on this fact. If one uses all the projections to paint the fourier spectrum of the original function and interpolates the missing values one can simply perform the inverse fourier transform to do the reconstruction. This is intuitively appealing, and fast, but interpolation can lead to a large artifacts, especially the interpolation of phases can be problematic. However, the method remains a nice option as a quick initial solution for feeding into iterative methods such as **SART**.

Another method influenced by the fourier slice theorem is the filtered backprojection. Backprojection refers to summing all the projections (r, ω) that includes the pixel in question. This will include all the relevant contributions, but overemphasize the contribution from the closest pixels, and what you end up with is a very blurry version of the original function.

The filtered backprojection method explores the connection between the radon transform and the fourier transform showing that the function $f(x, y)$ can be reconstructed by performing backprojection, but with a filter on the projections applied in fourier-space. Normally the filter is the ramp function $|R|$ which increses with frequency. In practice noise considerations make other filters that cuts off the high frequencies more usable.

Lastly there are the algebraic methods which sets up the sinogram as a discrete linear system of simultaneous equations

$$\mathbf{p} = \mathbf{M} \cdot \mathbf{x}$$

where **p** are all the measured projections from the sinogram, **x** is the reconstructed function $f(x, y)$ and **M** is the projection matrix that described the relationship of the radon transform. The system is then solved in the least square sense of minimizing the quadratic error. The system of equations is usually to large to be inverted directly so an iterative approach is taken where blocks of the projection matrix are solved one at the time. When the shape of these blocks coincide with projections in the sinogram the method is known as **SART**, or "Simultaneous algebraic reconstruction technique". This is the method we have depended on in conjunction with the Filtered Backprojection algorithm (**FBP**.

3.2 Sinogram correction

This section describes the filters implemented in `xrdtoolkit/tomo.py` for correcting sinograms assembled by peak fitting. The code itself is listed in appendix C.3.

3.2.1 Deinterlacing

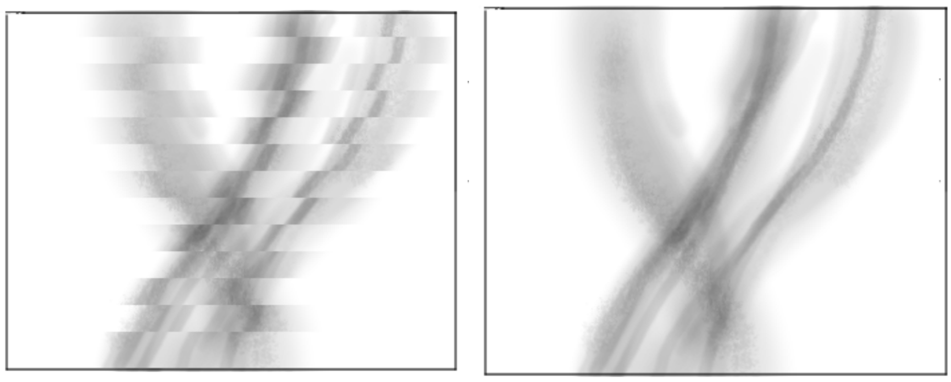


Figure 3.3: When the sample is scanned up and down alternately there might be a slight shift between even and odd rows due to timing differences in the acquisition. Here the effect has been exaggerated for illustration.

In order to save time when performing the experiment, scans are performed both going up and down. Subtle timing differences can give shifts in the projections after flipping back every other line in the sinogram. This effect is shown exaggerated in figure 3.3. The de-interlacing filter attempts to find this shift and applies the inverse to every odd row. The procedure is as follows:

- We split the sinogram into two parts. One with the even rows and another with the odd.
- We cross-correlate the two images through the relation $(f \star g)(t) = \mathfrak{F}^{-1}\{F^*G\}$ and use the coordinates of the maximum as inital shift.
- We construct a cost function with the correlation coefficient with the initial shift applied to th odd rows and refine the shift by optimization.
- The sinogram is reassembled with the refined shift applied to the odd rows.

3.2.2 Rotation center

Reconstruction algorithms usually assume the center of rotation to be running down the center of the sinogram. If this is not the case then artifacts are introduced.In practice it can be difficult to center the scan accurately when doing the experiment so we shift the sinograms after the fact.

The method we use is to correlate the first and last projections which are spaced 180° apart from each other. When the sinogram is perfectly centered the relation

$$p(r, \omega) = p(-r, \omega + \pi)$$

since we are in a parallel ray geometry. After flipping the last projection we correlate them as we did with the subsinograms in the de-interlacing filter and find find the shift that makes overlap best. The rotation center needs to be shifted half of that.

3.2.3 Bragg peak filtering

For some orientations large crystallites or pollutants will be highly diffracting completely saturating the signal compared to poly-crystalline and amorphous phases of the sample. We encountered this problem in our dummy sample as can be see on some of the reconstructions in our results.

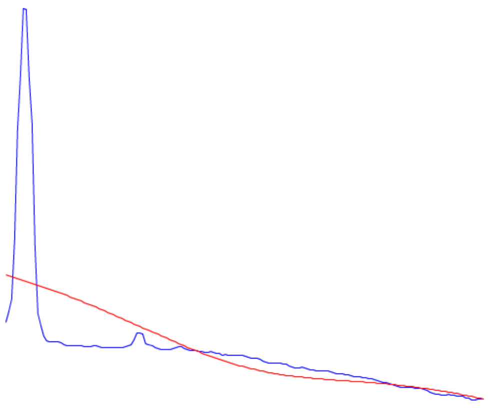


Figure 3.4: Integrated powder profile (blue) and fitted peak (red). At the left edge the contributions from a bragg peak causes a negativ area for the center peak in the least square fit.

The diffracting peaks may cause very high or very low values in the sinogram depending on the location of the peak relative to the gaussian which is fitted. If the bragg peak is not overlapping with the gaussian it will be fitted to the linear background with a very high positive or negative slope, while the gaussian function will try to compensate with a large negative area as shown in figure 3.4.

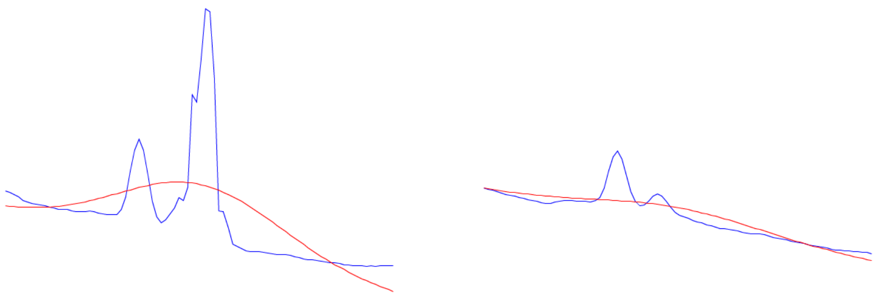


Figure 3.5: Integrated powder profile (blue) and fitted peak (red).

If the bragg peak coincides with the peak being fitted our signal is dominated and a artificially high value is found 3.5.

The problem with large values in the sinogram is that they are backprojected in the reconstructed image as dark or bright streaks which masks otherwise sensible values. This means that even if we are not able to remove the bragg peak from the diffractogram used to assemble our sinogram we can at least improve the reconstruction by reassigning the value to something less extreme.

We have developed a simple filtering procedure that works directly on the sinogram. Qualitatively this has worked satisfactory without manual user input as demonstrated in figure 6.8 in our results section.

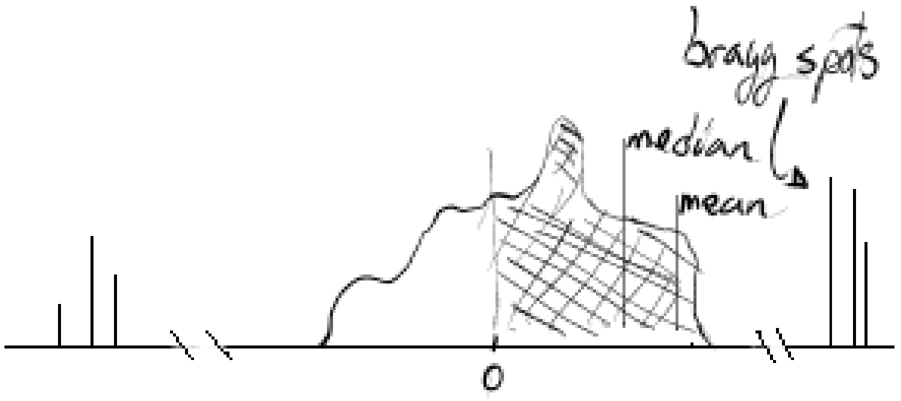


Figure 3.6: Typical histogram of sinogram with bragg spots. The positive part has been shaded and annotated with mean and median. The bragg spots are not to scale, but far off-figure, skewing the arithmetic mean.

Initially we tried an approach using the sobel filter in the ω direction looking at big changes from one angle to another. This brings out the spots really well, but as an edge detector it does not mark the outliers themselves, only their edges. Instead we ended up with adaptive thresholding which mean selecting values which are much larger than the local median. The Numpy modules in python already provides such a function `threshold_adaptive` defined by

$$pixel > [median(blocksize) - const]$$

where the median is performed in a block with `blocksize`$\times$`blocksize` number of elements about the candidate pixel. We want to make the constant term so big that we do not select any values in the sinogram itself, but small enough that we include all the values which are caused by bragg spots. To do this we consider a hypothetical sinogram with a histogram as drawn in figures 3.6 and 3.7. Since the bragg spots may have very high or very low values we split the procedure in two, considering high and low parts of the sinogram separately. For some angles 2θ we found that the assembled sinograms were biased with negative values due to the assumption of a gaussian diffraction peak on linear background. For instance when the background

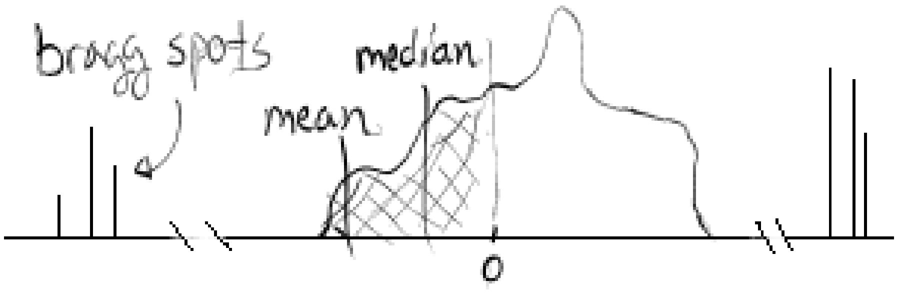


Figure 3.7: Typical histogram of sinogram with bragg spots. The negative part has been shaded and annotated with mean and median. The bragg spots are not to scale, but far off-figure, skewing the arithmetic mean.

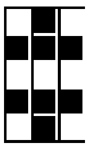
followed a exponential decay the gaussian fit would compensate and even go negative when there was not a big signal. In order to be robust against a bias such as this we use the global median as a measure of the zero level in the sinogram and split the histogram based on this value.

The bragg spots may be of different orders of magnitude, so we want to use the median for our constant in the adaptive threshold routine. In some extreme cases we saw that values which were clearly supposed to be caught in the filter did not get above the treshold on account of the very skewed arithmetic mean. The median is more robust to outliers and so will give us a value closer to the actual sinogram in the histogram.

To control the number of points that we select in the filter we scale the median by a number that we have chosen to call the upper and lower `sensitivity` coefficients. They simply serve to scale the threshold value beyond the values making up the sinogram. For our sinograms we found a lower sensitivity of 1.5 and higher sensitivity of 0.2 to give good results, so these serve as defaults, but can be changed at will when calling the filter function.

Having made a decision about which pixels to mask we need to find some better suited values to replace them with. If we look at the uncorrected sinograms in figure 6.8 we can clearly see that the bragg spots appear as horizontal lines, usually no more than one pixel in height and a few pixels in width. We can therefore be fairly certain that the values just above and below the one we are looking to replace are not affected by bragg peaks, but can be used. The projections on the immediate left and right on the other hand, are most likely suffering from the same diffracting condition and should under no circumstance be used.

In the end we chose to use a 2D median filter with the following footprint



as a compromise between getting enough values to be robust while not sampling too far away from the current location.

As a fail-safe there is a configurable tolerance for the high and low part. If the filter finds that more than for instance 5% of the values are too bright or too dark the filter panics and should do nothing for that part except write a warning to the user. The motivation for doing this is that a sinogram which has had a significant part replaced by that of the median filter described above will lose a lot of information. Since Diffraction/Scattering Computed Tomography sinograms already are low in resolution this is a situation that needs to be avoided.

3.3 Sample self-absorption correction

In Diffraction/Scattering Computed Tomography we have particular problem with absorption in the sample. Our signal is in effect the the line integral of μ_{scat} from equation 2.1 which we defined as the attenuation of the beam due to scattering effects. These are the rays we are regisering on the diffraction detector. However voxels further along the gauge volume see a reduced beam intensity, which is *not* due to μ_{scat} alone, but also μ_{abs}. In addition, any radiation that is scattered in the gauge volume is further attenuated as it exits the sample.In order to perform quantitive tomography, we really need to correct for this self-absorption if the sample contains any heavier elements.

Looking at figure 3.8 we will try to derive an expression for the effective line integral that we will record on the detector.

When performing our reconstructions we are already assuming that scattering from different depths in the sample contribute to the same pattern on the detector, meaning the gauge volume is considered point-like. When the sample-size is much smaller than the sample-detector distance this is a fair approximation.

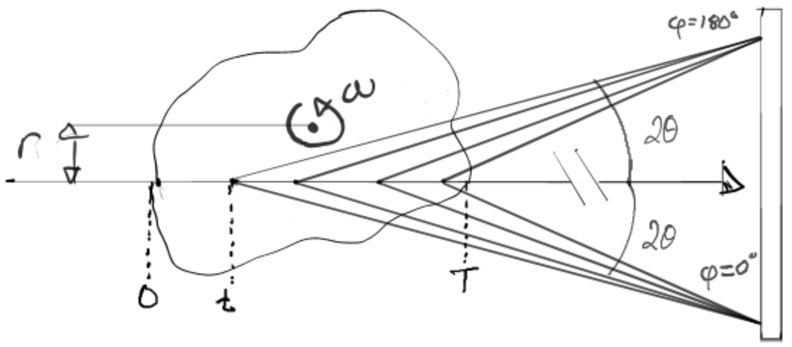


Figure 3.8: Illustration of ray (r, ω) scattered at point point t towards detector at the two azimuthal angles in the plane. T is the length that would be traversed by the beam in transmission tomography.

The scattered intensity at a point t along the path should be proportional to the density at that point. If the intensity of the beam was originally I_0 we know due to Beer-Lambert that the intensity seen by the scattering point is

$$I = I_0 \exp(-\int_0^t \mu(t')dt').$$

To get the corrected value at the detector we would need to perform the line integral

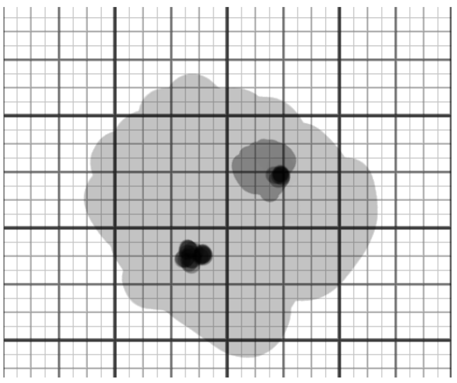
$$\int_t^T \mu(l(t'))dl(t')).$$

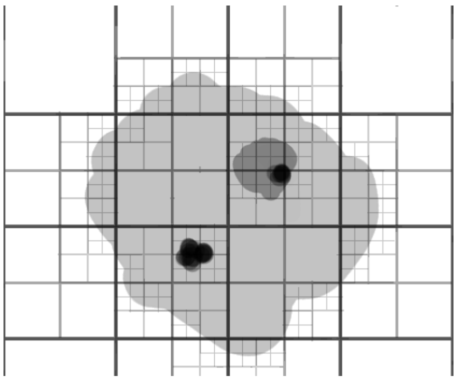
where $l(t')$ is the line from the point t' to the detector. In theory we could calculate these integrals using a reconstructed volume from absorption contrast tomography covering the necessary region and scale accordingly. However it means that a proper correction involves integrating over no less than three variables 2θ, ϕ, t for each and every diffraction image $(r\omega)$. A brute force solution is clearly not compatible with online-analysis.

3.3.1 Acceleration of absorption correction

The fundamental difficulty in the absorption correction problem is the number of "nested" integrals when calculating the corrected projection values. However if we can decrease the number of operations necessary for integrating each path through the sample that might at least help some.

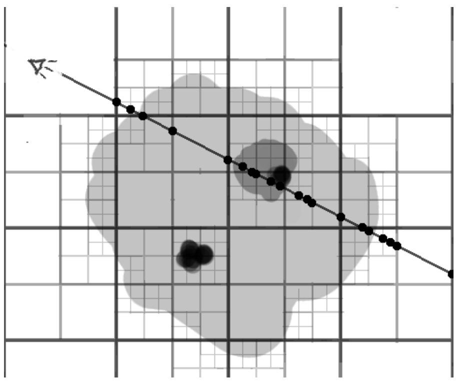
We propose using an octree-based approach for accelerating the integration of attenuation in the sample. The octree is a commonly used acceleration structure in the video game and visualisation industries. In computer science literature the process of sampling and integrating from such a volume is called “Volume ray casting” and it has been the subject of much research lately as the basis for next-generation rendering. High-performance solutions are likely to be available. The octree is at heart a hierarchical representation where the further up in the hierarchy you go the larger regions are covered. Figure 3.9a shows an example absorption slice after it has been split into a hierarchical grid. Weingartner et. al.[17] describes in their paper how a split and merge procedure can be used to segment an image into homogeneous regions. In summary adjecent cells in the hierarchy are merged together if satisfy some homogeinity parameter. Figure 3.9b shows how the result might look when the parameter has been set to a suitable scale.


(a) Absorption slice once split procedure has built an octree covering every pixel.

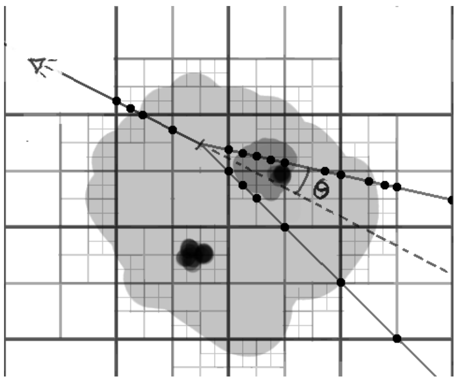

(b) Hierarchical representation of absorption slice after merge procedure. Each square contains the average of the corresonding region.

The structure only needs to be built once, and can then be used for all projections in the experiment, since we need to cover the whole sample due to the azimuthal dependency anyway. The time spent building the acceleration structure should thus be neglegible.

The power of the structure is that it allows us to skip large homogenous regions, and provides a simple mechanism to tune performance against precision. In addition, the resolution of a typical absorption sinogram is on the order of 2K pixels, or the resolution of the detector, whereas the slow 1D-scanning of the Diffraction/Scattering Computed Tomography technique makes it unpractical to go above a few hundred pixels per slice. The split-merge procedure naturally reduces the data of the absorption contrast volume and the octree hierarchy makes it efficient to query. This is shown in figure 3.10a where we only need to sample when entering a new cell and homogenous regions are covered by larger cells up the hierarchy.



(a) Sampling through a slice of the absorption volume by querying the octree representation. Each dot marks the .



(b) Snapshot of the absorption routine as it steps through the sample and accumulates the contributions as a function of 2θ. The stapled line represents the distance still to be stepped through.

Method 3.1 Pseudocode of the self-absorption correction using an octree for volume ray casting. The attenuation() function is assumed to implement volume ray casting using the given octree volume.

```
Input: octree_volume , $(r, \theta)$
Output: flat_field
  ACC ← 0
  LUT ← constructLookUpTable()
  repeat
    STEP along $t$
    ACC ← ACC $*$ attenuation(t,octree_volume)
    for all pixels in detector do
      $(2\theta, \varphi)$ ← LUT(pixel)
      flat_field$(2\theta, \varphi)$ ← ACC * attenuation($t, 2\theta, \varphi$,octree_volume)
    end for
  until end of sample
```

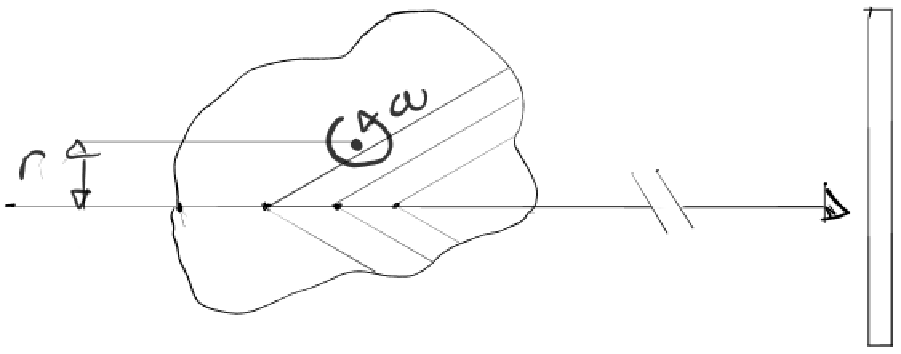


Figure 3.11: Illustration of different paths taken through the sample when in incoming ray is scattered at different points in the interior. The scattering angles are all equal and in the far-field approximation the rays will converge in a ring on the detector.

3.3.2 Zero order approximation

An alternative to doing the full self-absorption correction as suggested above we can make some simplifications in order to at least try to solve a zero-order approximation of the problem.

The expression we found for the recorded intensity including self-absorption was

$$\propto \int_0^T \rho(t)\exp(-\int_0^t \mu(t')dt')\exp(-\int_t^T \mu(l(t'))dl(t'))dt$$

We start of by assuming a roughly spherical and homogenous sample. If we further assume that the total path length as seen in figure 3.11 are all equal to T then we can eventually separate the factors so that

$$\propto \exp(-\int_0^T \mu(t')dt')\int \rho(t)dt.$$

This last assumption should hold pretty well in the case of small-angle scattering as the paths taken out of the sample are not too different.

The first term in the last equation is simply the projected value from the absorption sinogram, so a zero order absorption correction could involve simply using the absorption sinogram as a flat field for the Diffraction/Scattering Computed Tomography sinogram. This does not even require a reconstruction of the absorption volume since we use the projected values.

In practice this would involve registering the sinograms that they are aligned and match the same region. This may be problematic unless care is taken to note the alignment when performing the experiment. It is expected some form of correlation routine might be able to align the sinograms, but since our sample was not really absorbing, software routines for this was not tested nor developed.

Chapter 4

xrdtoolkit

4.1 The framework

In this chapter we describe the software developed for processing X-ray diffraction data, and how it has been applied in practice with a computed tomography workflow. At the core is the python module *xrdtoolkit* which contains routines and functions for

- Atomic cross sections.
- Sample mass attenuation by stoichiometry
- Performing file IO.
- Peak fitting.
- Detector tilt calibration.
- Azimuthal regrouping/integration of diffraction patterns.
- Averaging of images.

With these operations covered the toolkit can already be useful since most workflows perform detector tilt calibration and integration as first steps of the data reduction.

4.1.1 Dependencies

As is usually the case, common problems already have solutions. One of the motivations for choosing Python as the implementation language was the adoption it has seen in the scientific computing community lately. The numpy and scipy [3] libraries cover a lot of what is needed for scientific computing such as linear algebra, signal and image analysis, various solvers and integrators and so on, making it in many instances a drop-in replacement for Matlab scripts commonly used by scientists. An added benefit is that this software is licenced freely for modification and use. These routines are for the most part wrappers around efficient C or Fortran code so you do not necessarily have to sacrifice performance for the convenience of using a high-level programming language.

At the ESRF this combination of python and numpy has been adopted by the software group. One of their active developments, pyFAI [11] provides fasth azimuthal integration of diffraction data as a python module. It can be made to run on the Graphics Processing Unit (GPU)or Central Processing Unit (CPU)using OpenCL and OpenMP and bundles with detector calibration functionality. In xrdtoolkitwe have used its integration functionality. For X-ray fluorescence there is the PyMca package, also based on python and developed by the software group at the ESRF. We briefly used their interactive peak fitting dialog when looking for regions of interests in the powder profiles.

4.1.2 File handling

By default we store datasets in the `hdf5` format. This is a hierarchical container for datasets where you reference each resource by a path. For example the toolkit will place all its output datasets in the group `/xrdtoolkit/`. This makes it possible to store datasets that relate to each other in a single file and to keep things tidy and organized.

After performing the data reduction steps as illustrated in figure 2.1 on the dummy sample data, our results file `sinogram.h5` has the following structure

```
xrdtoolkit
├── fitted_peaks
│   └── [datasets with peaks]...
├── sinogram
├── sinogram_corrected
└── reconstructed
```

The `h5py` python module provides easy access to the datasets, compatible with numpy arrays and for all intents and purposes it operates like a python dictionary.

Working with diffraction data you can encounter a variety of file formats. We support quite a few of them through the use of the `fabio` module. In `files.py` we have abstracted access to files and sequence of files so that it is possible to simply pass a list of filepaths to our generator

```
from xrdtools import files
for image in files.ImageSequence(filePaths):
        #do something with the image
```

to iterate over every image. The `ImageSequence` generator even supports multiframe `EDF` files and the `fastReadData` functionality in fabio for efficient reading.

4.1.3 The common tables

Included in the xrdtoolkittoolkit are a collection of tables tabulating information such as absorption edges, atomic mass and so on. For a complete overview issue the following in an interactive python shell:

```
from xrdtoolkit import common
common.XrayTable
```

To use any of the properties you index `XrayTable` as a dictionary with the atomic number of the element you need. As an example, to get the atomic mass of carbon you would write

```
from xrdtoolkit import common
common.XrayTable[common.Elements['C']]['AtomicMass']
```

Finally `common.Constants` contains some of the most common physical constants.

4.1.4 The sample class

The sample class is provided as a starting point and can currently be used to compute the mass attenuation coefficient. To get the mass attenuation of water for 40 keV X-rays as an example simply do the following

```
from xrdtoolkit import sample
water_sample  = sample.Sample(1, 0.999868, 'H2O', 0)
water_sample.mass_attenuation('40')
    -> 0.27340376617108358
print(water_sample)
    -> Thickness: 1cm
    -> Density: 0.999868g/cm^3
    -> Compound: ('H2O', {('H', 1): 2, ('O', 8): 1})
    -> Chi: 0
```

4.1.5 Testing

The module has support for unit testing and already provides a few tests to make sure essential calculations match those of the reference implementation by Veijo in matlab[10].

To run the tests go to the project directory and type

```
:~$ python setup.py test
```

The test files themselves serve as further examples of how to use the python module.

4.1.6 Installation

Scripts for setting up xraylib for use *and* development are provided. In theory all you should need to do if you have system-wide access is

```
:~$ git clone https://github.com/amundhov/xrdtoolkit.git
:~$ cd xrdtoolkit
:~$ ./install.sh or ./develop.shp
```

The python library dependencies are maintained in **requirements.txt** and can be parsed by standard python package tools such as **pip** and **distutils**. More information is available in the code repository itself.

4.2 xrdtoolkit scripts

The scripts described here all builds on the functionality of the python module and makes it available to run on the command line.

For example usage of the scripts we refer the reader to appendix A where we have listed some example code for doing process level parallelisation with the xrdtoolkitscripts. The scripts themselves also provide an overivew of the accepted options through the `--help` option.

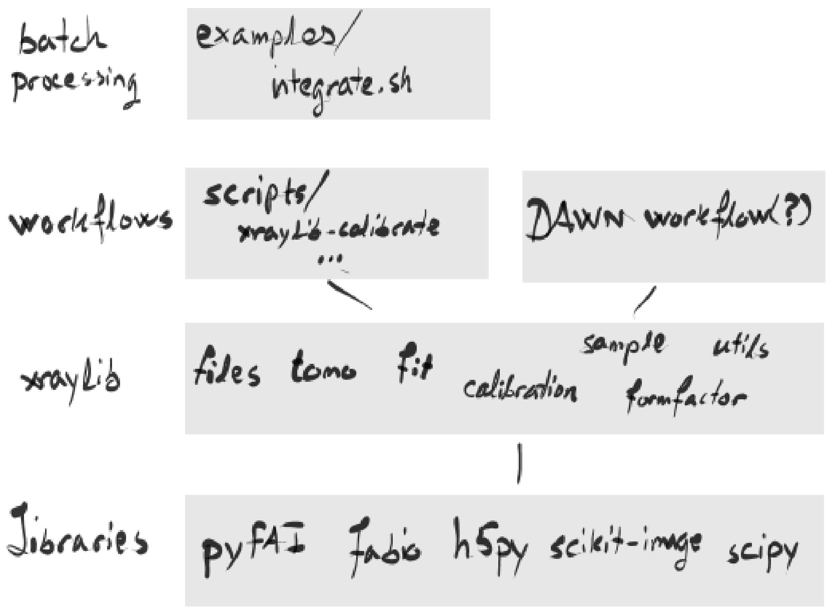
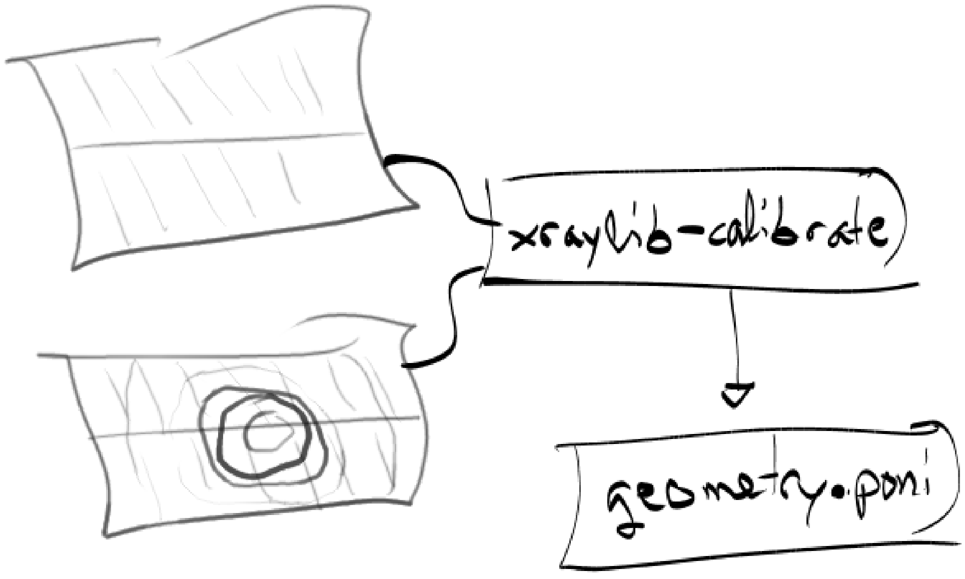

batch processing
examples/ integrate.sh
workflows
scripts/ xraylib-calibrate ...
DAWN workflow(?)
xraylib
files tomo fit calibration sample utils formfactor
libraries
pyFAI fabio h5py scikit-image scipy

Figure 4.1: Overview of the software stack from supporting python libraries to files for batch processing.

4.2.1 xrdtoolkit-calibrate



The xrdtoolkit-calibrate script attempts to find the detector tilt and beam position on the detector without any user input. It uses the diffraction pattern As optons it accepts one or more darkcurrent images to be subtracted, detector binning mode, initial conditions for tilt and origin, detector distance and limits on radial distance.

The script builds on the `f2w.py` submodule which contains azimuthal integration code due to Veijo. The integration code can split the integration into a number of sectors. These sectors are compared against each other in trying to make the diffraction rings appear perfectly circular, which would mean that the correct tilt and origin has been found.

In working with the code we found some errors in the way that the stopping criteria was calculated. The vector $c = [c_1, c_2]$ gives us the change in origin or angle that was done for the current iteration. We define then the distance

$$s = \sqrt{c_1^2 + c_2^2}$$

and let our stopping criteria be

$$stp = \frac{s}{\Delta s} < 0.001.$$

that is, when our step is less than a percent of the uncertainty of the step. The jacobi vector

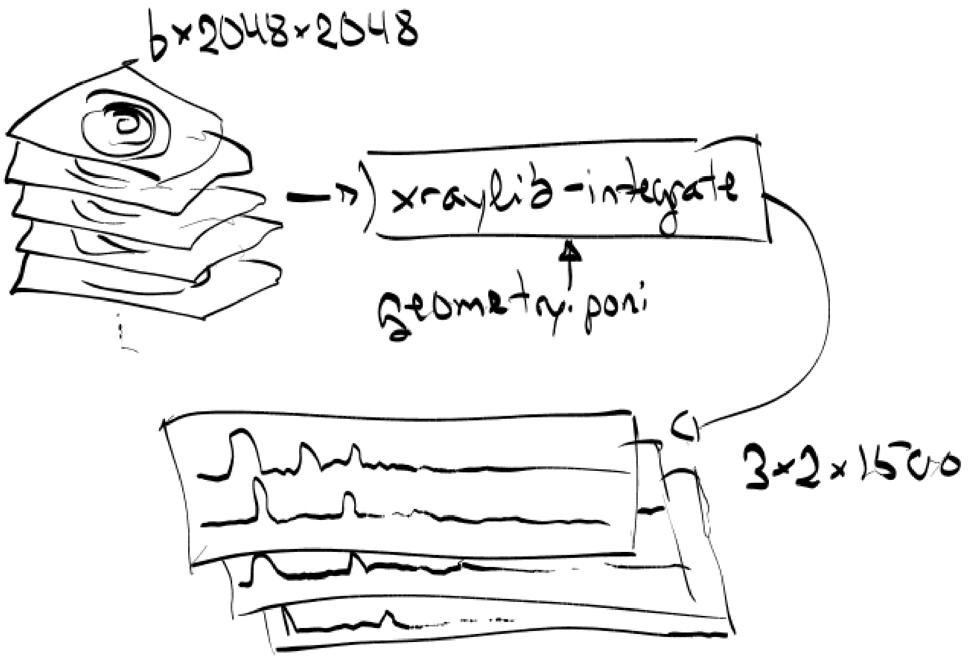
$$J = (\frac{\partial s}{\partial c_1}, \frac{\partial s}{\partial c_2}) = \left(\frac{c_1}{\sqrt{c_1^2 + c_2^2}}, \frac{c_2}{\sqrt{c_1^2 + c_2^2}} \right)$$

and Σ_c is provided by the least squares regression in `f2w.py`. The uncertainty Δs is now given by equation 3.3.

A final modification we made was to make the calibration routine start with only searching the origin when starting. The justification for this is that tilt angles usually are very small and so do not need to be considered until the very end of the search, whereas the position of the origin may be far from the center of the detector. In a few cases the tilt angles were observed to converge towards very high angles of what was probably a local minima when optimizing angles and origin simultaneously. Instead we lock optimization of tilt until the stopping criteria goes below 1, at which point, we hope, the beam origin is sufficiently close for full optimisation to converge on a good geometry.

The detector geometry is written to disk in the file format required by `pyFAI`.

4.2.2 xrdtoolkit-integrate



The xrdtoolkit-integrate script is a wrapper around `pyFAI` for performing azimuthal integration on a series of diffraction images. It requires the geometry file produced by `xrdtoolkit-calibrate` or from its own calibration routines. To specify the files to be calibrated you give it the prefix of the path that contains your diffraction images, for instance `/mnt/data/.../EXPERIMENT/dummy_` will match every file in the `EXPERIMENT` directory starting with file name `dummy_`. It will attempt to parse the filename to extract the experiment parameters that each file belongs to. E.g. the file `dummy_0_0_50.edf` will be interpreted as having index $[0, 0, 50]$ in a three dimensional parameter space, and stored accordingly in a volume of diffractograms. The dimensions of the resulting dataset is inferred automatically after parsing all the matching files. If there are any multiframe `EDF` files they will be treated as a separate dimension. This might sound complicated, but as long as the diffraction image indices are given at the end of the file name separated by underscores the script should do the right thing. For tomography data this means assembling a stack of sinograms/diffractograms.

If the dark current is given as a stack of images it will check to see if they match the number of frames in the files to be integrated. If that is the case the diffraction images to be integrated will be corrected by the correspoind dark current image in the series.

It was immediately apparent that loading all the diffraction data for integration would take a considerable amount of time. In order to hide this latency from the user we use one python thread for loading the diffraction images into a queue and another one to performing the integration using `pyFAI`. This is known as the consumer-producer pattern. In the most common imeplementation of Python only one thread can execute in the python context at a time due to what is know as the Global Interpreter Lock (GIL). Almost every action performed by the python interpreter requires that it holds the GIL, but luckily threads will release the lock when entering external pieces of code. This is the case `pyFAI` which means we can benefit from threading despite the GIL.

4.2.3 xrdtoolkit-assemble

The xrdtoolkit-assemble script is particular to our Diffraction/Scattering Computed Tomography workflow. It takes a list of diffractograms and a list of regions of interests and performs peak fitting on regions of interest to assemble sinogram. If given more than one diffractogram the srcipt will construct a stack with each sinogram as slices. Our reconstruction script can later reconstruct a *volume* from the stack of sinograms. The peaks to be fitted are assumed to be constant in shape and position over all projections with only the area varying. We can then write our model as a set of equations

$$y_i = c_0 + c_1 x_i + c_2 f_i(x_i - x_0, \sigma) \tag{4.1}$$

where y_i is the signal that we are reading from the diffractogram, c_0 and c_1 are coefficients for our linear background, c_2 is the area of our peak and x_i is our x variable, or channel number. Our peak function f defaults to the

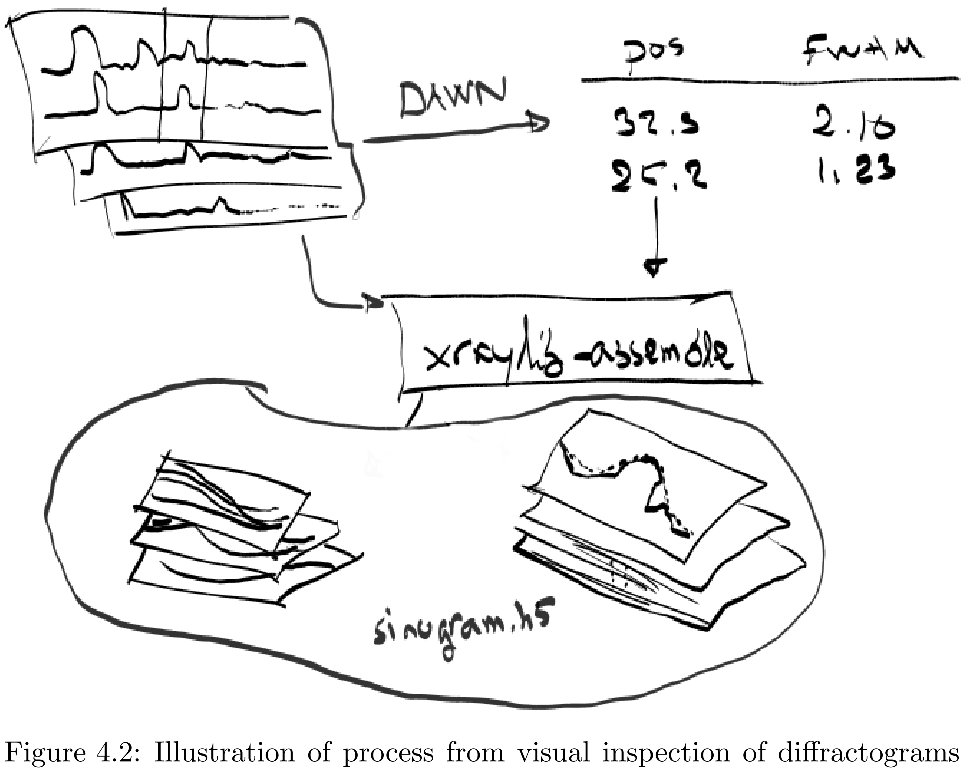


Figure 4.2: Illustration of process from visual inspection of diffractograms in e.g. Dawn, specification of interesting peaks and eventual assembly of sinograms by xrdtoolkit-asemble. The `sinogram.h5` file also gets written the fitted peaks for reference.

gaussian function. However, if the peak is very narrow a better fit is obtained if we define the guassian through its cumulative distribution function erf

$$f(x_i - x_0, \sigma) = \frac{\operatorname{erf}\left(\frac{x_i + \Delta x/2 - x_0}{\sqrt{2}\sigma}\right) - \operatorname{erf}\left(\frac{x_i + \Delta x/2 - x_0}{\sqrt{2}\sigma}\right)}{\Delta x} \tag{4.2}$$

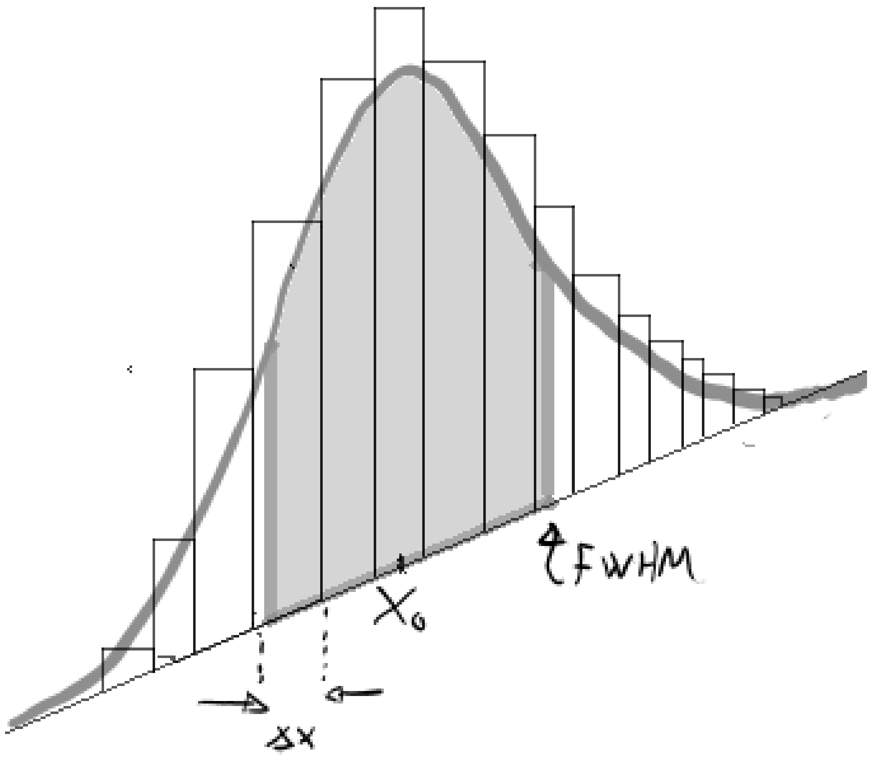


Figure 4.3: Example data fitted to a gaussian curve on a linear background. The bars represent counts in the integrated diffraction data. The shaded area shows the section which is within the FWHM of the fitted gaussian curve.

To convert between FWHM and σ we can use the following relation for gaussian peaks.

$$\text{gaussian} : \text{FWHM} = 2\sqrt{2ln2}\sigma \approx 2.3548\sigma. \tag{4.3}$$

In matrix form our system can be written as

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{c} \tag{4.4}$$

with

$$\mathbf{A} = [\text{ones(n)}, \mathbf{x}, \mathbf{f}(x - x_0, \sigma)]\,. \tag{4.5}$$

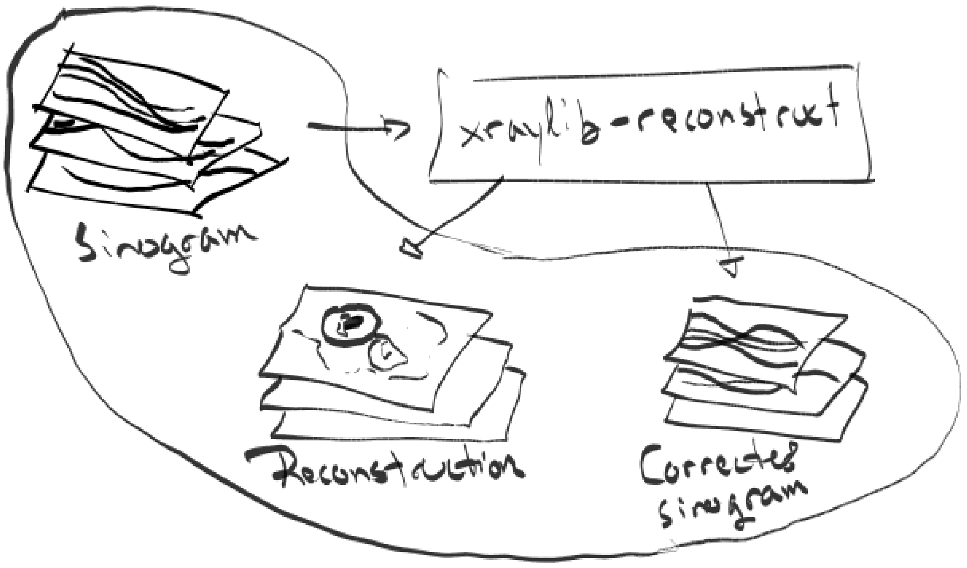
Since **A** does not depend on anything but the peak shape we can reuse the the matrix for all our peak fitting. The system 4.4 we solved in equation 3.1 so our solution is given by

$$\hat{\mathbf{c}} = \mathbf{\Sigma}_c \cdot \mathbf{A}^\intercal \mathbf{\Sigma}_y^{-1} \cdot \mathbf{y}.$$

For uncertainty we use the signal and dark current as we discussed in section 3.0.2. Every observation y_i is independent, so the covariances in Σ_y^{-1} are all zero. For further performance gains we can perform the multiplication elementwise instead of by matrix multiplication.

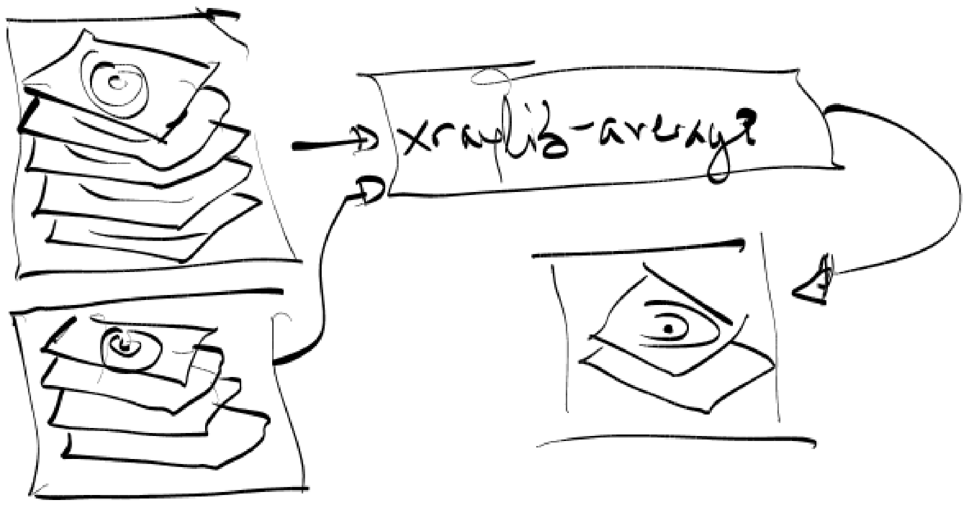
For the sinogram the value we use in the end is c_2 which corresponds to the area of our peak function. In addiation we store all the signals and fitted peaks in a separate dataset in the same sinogram file, which can be useful for explaining value in the sinogram.

4.2.4 xrdtoolkit-reconstruct



xrdtoolkit-reconstruct is a pretty straight forward script. It takes sinograms assembled by xrdtoolkit-assemble, performs sinogram filtering as discussed in section 3.1 and does the reconstruction slice for slice. For the filters it is possible to disable each one individually. The reconstruction is first done with the filtered backprojection method before going through one iteration of SART, however the number of iterations can be increased if desired. The script places the reconstructions in a separate group in the hdf5 file of the sinogram, along with the corrected sinograms for reference.

4.2.5 xrdtoolkit-average



The xrdtoolkit-average script is a very thin wrapper around the `files.averageImages()` method in the toolkit. Its principal arguments are what kind of averaging method that should be used (mean or median), and whether multiframe files should be flattened before averaging or if average should be taken across the each file preserving the number of frames in the outpuf file.

Chapter 5

Experiment

5.1 Setup

To test the tools that had been developed

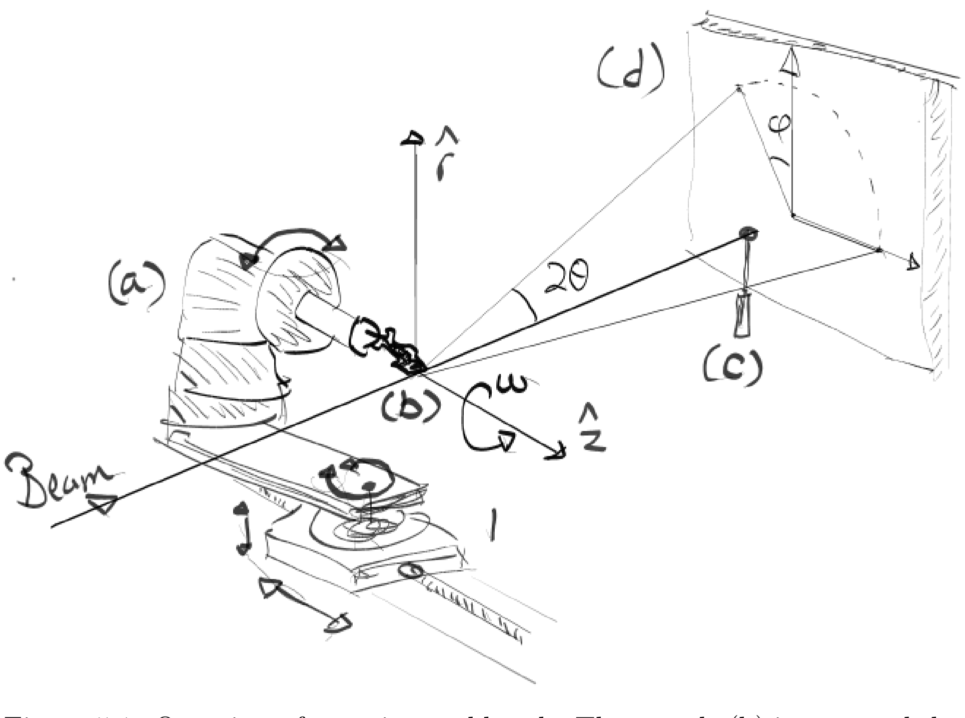


Figure 5.1: Overview of experimental hutch. The sample (b) is mounted the diffractormeter (a) which is free to rotate and translate. The beam is scattered by the sample and registered on the detetcor (d). A beam stop (c) prevents the direct beam which is not scattered by the sample from burning out the detector.

Only a small fraction of the X-rays are scattered by the sample, especially micrometer and nanometer sized samples. Most of the beam goes straight through and would completely saturate the diffraction detector, or most likely destroy it. A set of beam stops are therefore used to

1. Stop the direct beam from reaching the detector.
2. Reduce the amount of background noise registered on the diffraction detector from air scattering between sample and detector.

To demonstrate the tomography workflow we wanted to prepare a sample that would produce a nice and clear diffraction pattern, with nice and even Debye-Scherrer rings. This rules out metals or anything with too big crystalline grains or preferred orientations that induces texture in the diffraction rings. Previously ID15 has used a mixture of glass spheres and wax for testing the DSCT setup and knew its characteristics, so we opted for the same composition.

Initial attempts at filling micrometer capillaries with wax proved difficult. Since the sample was going to be rotated horizontally it was important that there were no loose parts that would change position or deform under gravity. A capillary with only glass spheres was therefore out of the question. Luckily we found that the cohesive force of the wax was enough to keep itself and the glass spheres fixed in place without any other support.

5.2 Sample preparation

For the glass spheres we used a mixture of $106 - 125\,\mu$m glass microspheres (GP0116) produced by Whitehouse Scientific Ltd. Using a scalpel we placed a small edge knife of spheres on a piece of aluminium folded as in figure 5.2 below.

The sharp fold in the bottom of the foil was intended to keep the glass spheres in a single line and provide a gradient for the melting wax to flow down. After a gentle shake to make the spheres settle wax was added to the top of the aluminium slide and put in a oven at 90 °C. After being baked for a few minutes we examined the unfolded aluminium foil in a microscope at 2x−5x magnification. As the reader can see in figure 5.3 the wax successfully wet the glass forming a nice wax linewith embedded spheres. Finally a few spheres were extracted with a scalpel and assembled on the edge of a cut tailor's pin. The low volume to surface ratio meant that the wax provided enough cohesive force to keep the glass spheres attached to the pin and fixed without bending under the effect of gravity. No change was seen in the sample from the time it was made until the experiment was performed a couple of weeks later, making us confident that the sample would stay intact

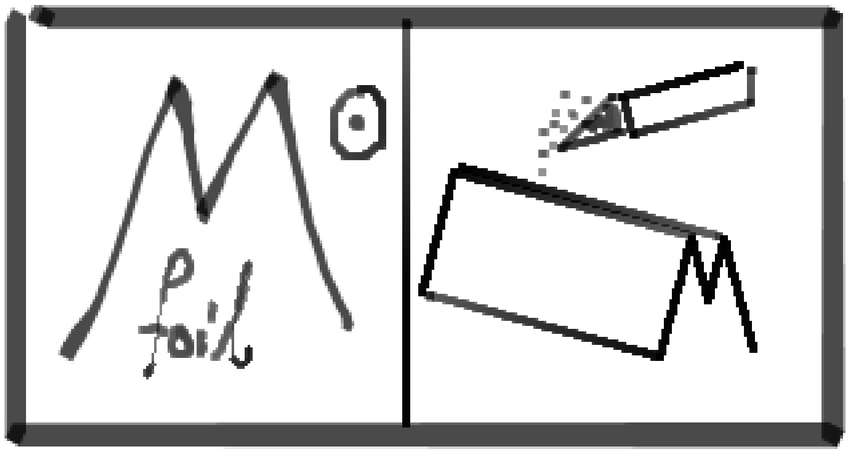


Figure 5.2: Illustration of aluminium foil used as a slide to mix wax and the micrometer glass spheres. The sharp fold held the spheres in a line while a small gradient made sure the wax flowed down and covered the spheres.

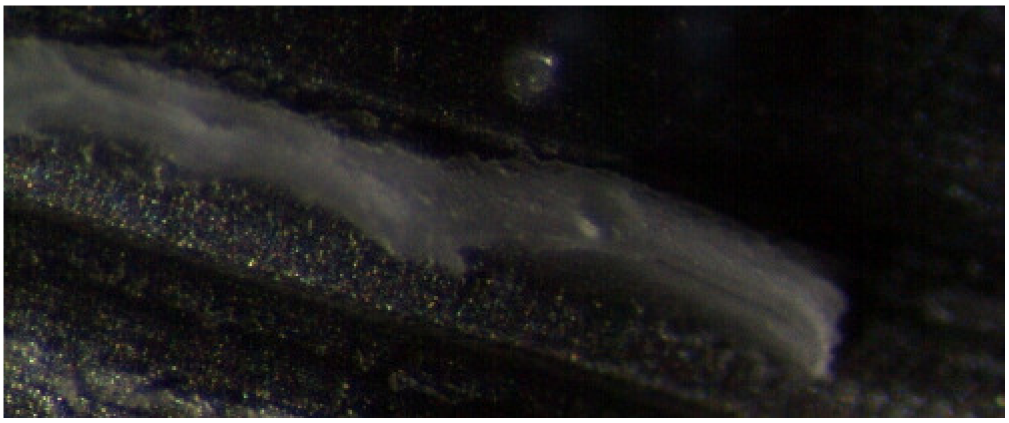


Figure 5.3: Aluminium foil with line of wax at 6x magnification. Several glass spheres can be seen contained within the wax in addition to a single sphere on the foil at the top.

during the experiment. The pin was already fed through a metal cylinder suitable for mounting on the goniometer and fixed with another piece of wax.

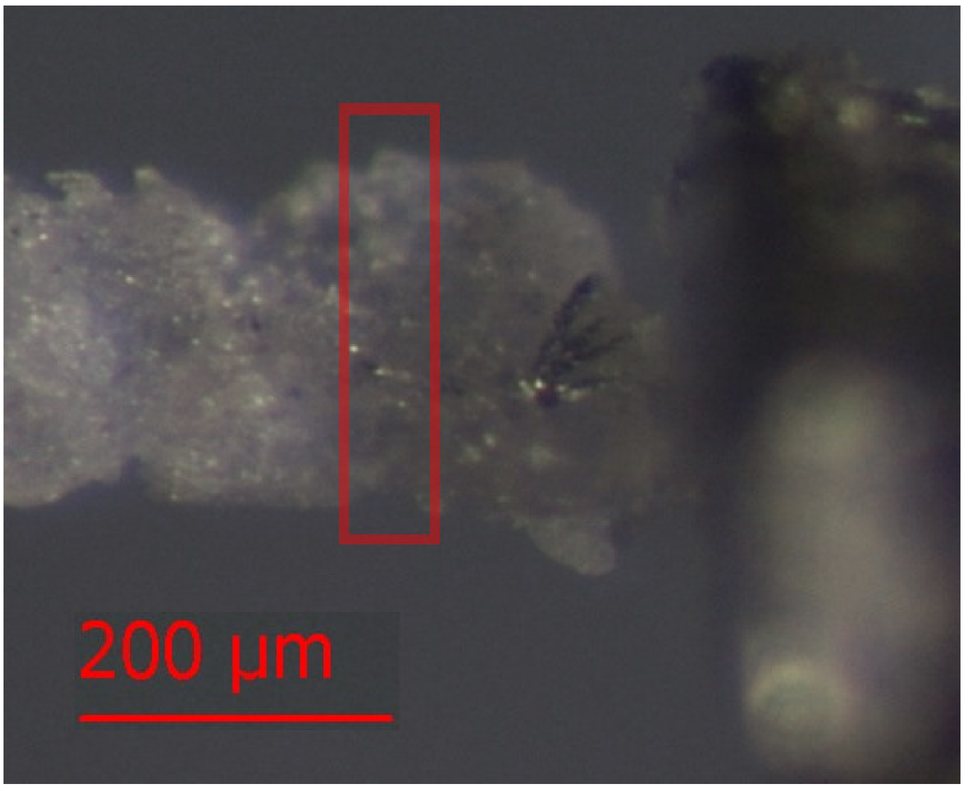


Figure 5.4: Image of dummy sample fixed to the end of a cut tailor's pin. The red rectangle show the approximate region where the tomographic scan was made. The reflecting material to the right of the rectangle is a piece of aluminium foil left over from sample preparation.

5.3 Data collection

The beam was aligned by beamline scientist Marco Di Michiel to be suitable for both absorption contrast imaging and Diffraction/Scattering Computed Tomography by using slits to shape the beam into a box-profile of 2.5 μm for the pencil beam. The monochromators were tuned to give a beam with energy 46.45 keV.

The prepared sample was mounted on the goniometer and rotated so that the rotation axis $\hat{z}$ of the diffractometer (a) in figure 5.1 aligned with the the sample. This way the spatial extent of the sample as it rotates is minimised. Using the FReLoN imaging detector developed at the European Synchrotron Radiation Facility (ESRF)the sample was measured to occupy 253 μm. We decided to scan $\pm 150\,\mu$m ($300\,\mu$m) to be sure to not clip off the sample at any point during data collection. While the original intention was to collect absorption data as well as diffraction data, to see how our zero-order self-absorption correction would perform. The imaging detector showed virtually no contrast however, except around the edges of the glass spheres, so we switched to the pencil-beam without doing the absorption data.

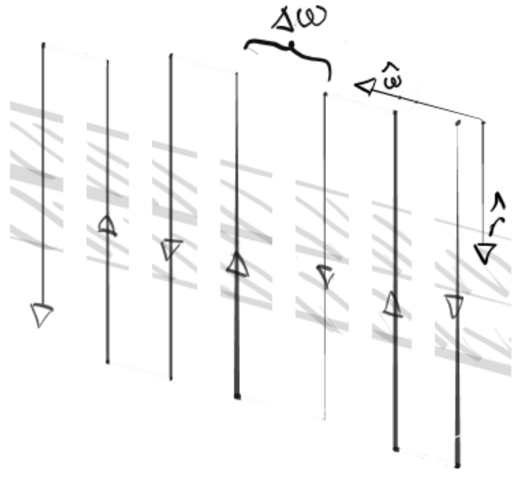


Figure 5.5: Illustration of scanning pattern where data is collected alternating going up and down as the sample is rotated. For each scan the sample is rotated through an angle $\Delta\omega$.

The sample was measured in a continuous scan along the $\hat{r}$ axis with a velocity so that the probe would cover a distance of $2.5\,\mu$m and generate 121 diffraction images with a 100 ms exposure. The sample was scanned as illustrated in figure 5.5 going up and down with a stepwise rotation of the sample in between in order to save time. A total of 90 projections were chosen, with a final one at 180° to make it possible to calculate the true rotation center of the sample. To increase the sensitivity of the images the Perkin-Elmer area detector was set to the $(2, 2)$ binning mode. Even in this mode and with the detector fairly close at 180 mm we still had enough radial resolution in our diffraction pattern to perform subsequent peak fitting. The dark current is aqcuired in the same way as a full scan, with 121 dark current frames in series. The motivation behind this is that the Perkin-Elmer has some memory of preceding frames when doing the readout so there is some change in the response of the detector as the scan progresses. When the signal is small this effect can be significant, so performing the azimuthal integration one needs make sure to use the correct dark current.

During collection we discovered that part of the detector was not shielded properly from scatter before the sample. Luckily this was outside the small-angle scattering that we used for our reconstructions. However a misplaced flashlight used to illuminate the sample during alignment was left in the path of some of the amorphous scatter from our glass phase and cast a shadow on the detector. Monitoring the detector readouts we also observed the occasional saturated spots in the some of the diffraction images. We can recognise some of these as isolated sharp, peaks in the integrated data shown in figure 6.2.

5.4 Data reduction

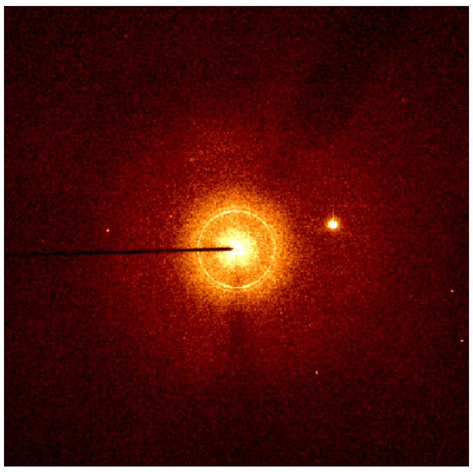
A series of 12 images were taken of the CeO2 reference for detector tilt calibration in addition to another series of 12 images for dark current. Both were averaged intra-frame using `xrdtoolkit-average`. Since the detector was positioned so that the beam origin was far off to one of the corners we did a quick inspection of the calibration image to provide an initial estimate for the calibration script.

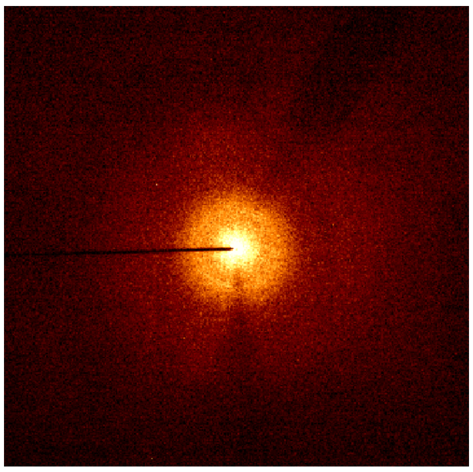
After calibration we proceeded with the data reduction as epxlained in appendix A

Chapter 6

Results

We now present the results that were obtained from our data collection. Figures 6.1a and 6.1b show the kind of diffraction conditions we were faced with. The region depicted is less than a quarter of the full detector.


(a) Selected small angle diffraction image with bragg spots completely saturating the detector (maximum value registered).


(b) Clean diffraction image showing a diffuse glow of the amorphous glass phase without the rings of the wax phase.

After azimuthal integration of the diffraction images we start to see patterns emerg in the powder profile as is evident in stack plots 6.2, 6.3a and 6.3b. That the broad scattering contributions corresponds to amorphous glass is evident from their reconstructions in figure 6.4 and 6.7 even though the latter is a bit influenced by the overlapping

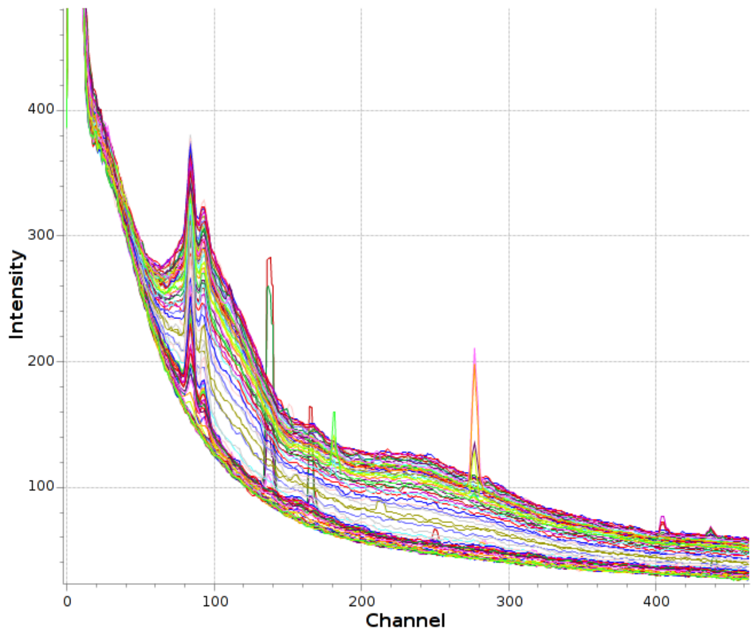
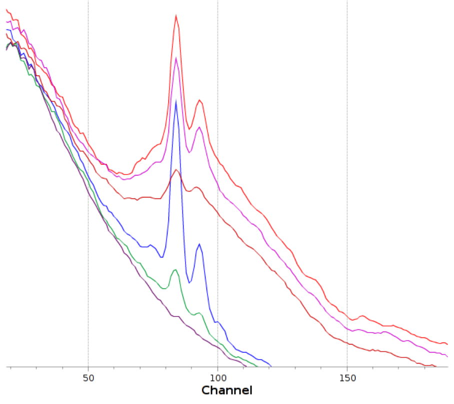
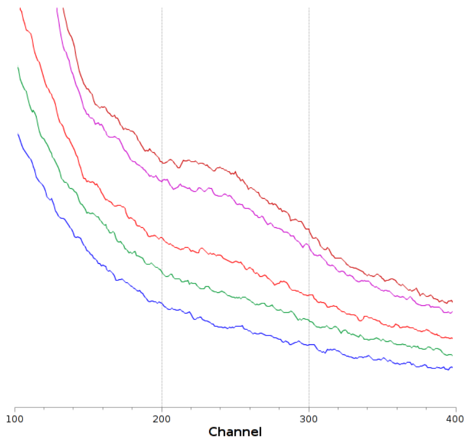


Figure 6.2: Stacked plot of azimuthally integrated diffraction patterns in the same small-angle region as shown in figures 6.1a and 6.1b.



(a) Integrated profile of selected projections. The two narrow peaks belong to the wax phase and overlap with broad scattering from the amorphous glass.



(b) Amorphous glass phase as can be seen on the tail of the complete small-angle region of figure 6.2.

Peak center	FWHM [channel]	Fit width [FWHM]	
254.88	81.72	2.0	pyMCA
83.96	4.82	- (3.0)	DAWN
92.86	3.50	2.5	DAWN
97.80	43.77	2.0	pyMCA

Table 6.1: Regions of interest passed to `xrdtoolkit-assemble`. The peaks were either fitted using DAWN's gaussian peak fit or pyMCA's gaussian fit corrected with internal background. The fit width parameter is used by the sinogram assembly script to determine how many samples should be included in the least square fitting; It defaults to include 3 times the FWHM of the peak.

6.1 Sinogram reconstruction

The regions that we chose to investigate and assemble are given in table 6.1. Their resulting sinograms

Significant streaking is clear in the case of peaks 254.88, 92.86 and 97.80. However they are all strongly reduced after being filtered and reconstructed again without bragg spots.

In order to get a clearer picture of which values are being masked by our bragg spot filter we have compiled all the sinograms into one figure 6.8 along with the mask that decides which projections will be discarded. In the case where there are no bragg spots, peak 92.86 quite a few spots are still masked.

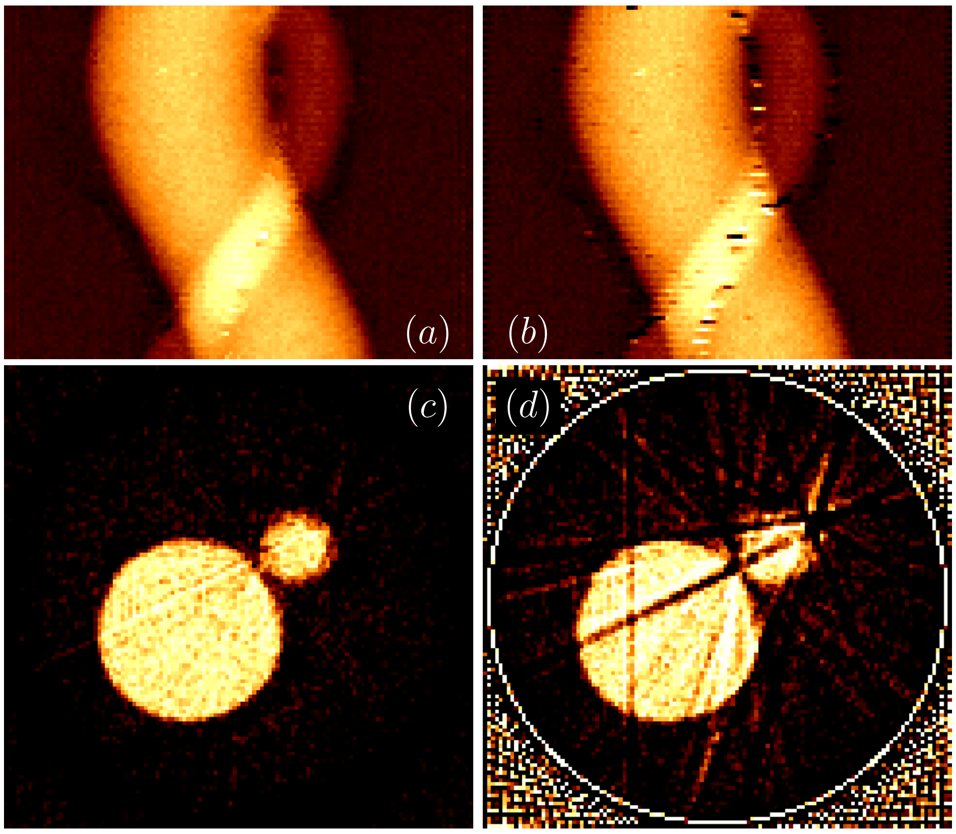


Figure 6.4: Sinograms and reconstructions of amorphous glass phase around channel 254, before and after diffracting spots have been removed by our filter as described in 3.2.3.

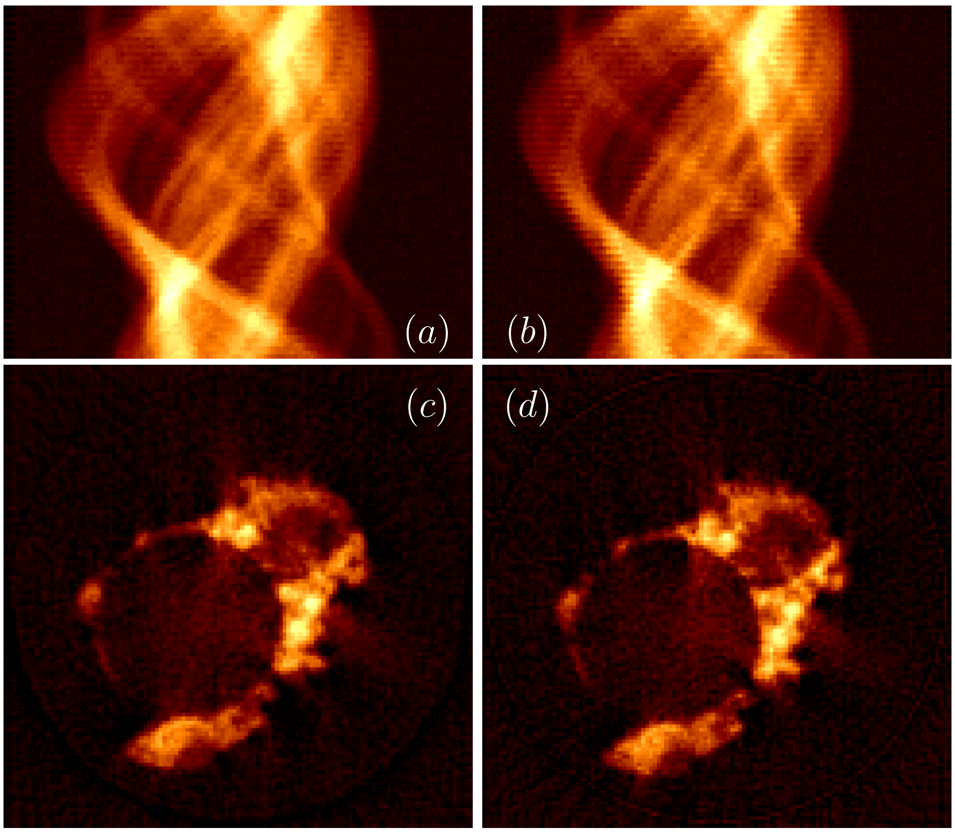


Figure 6.5

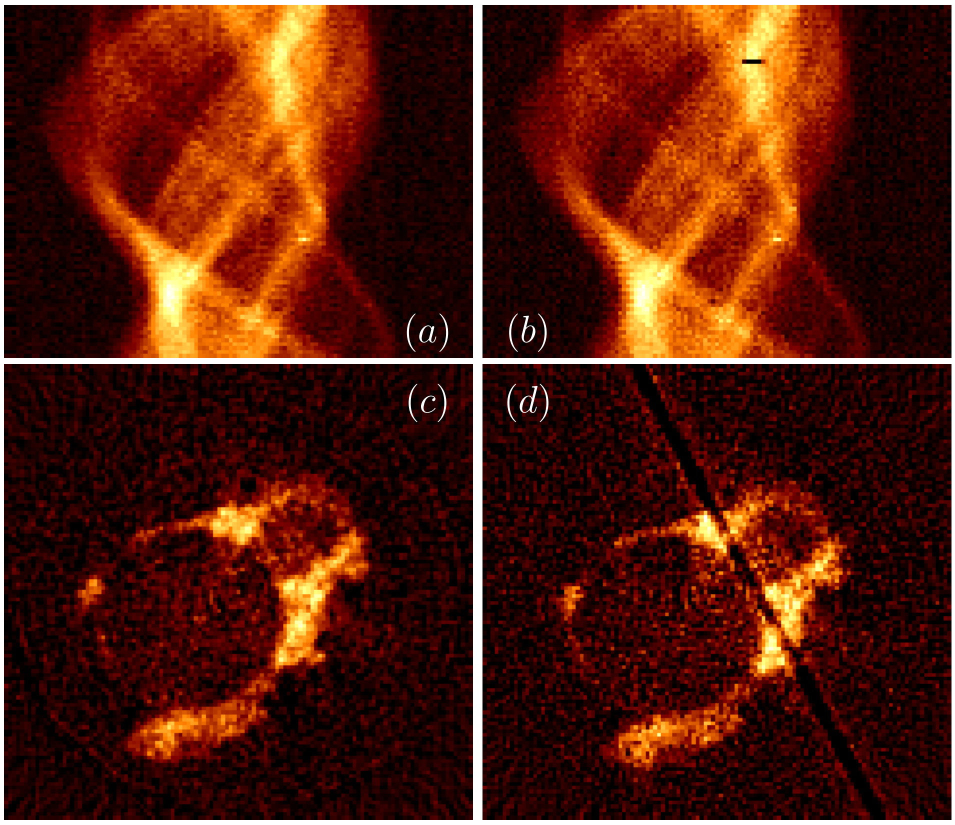


Figure 6.6

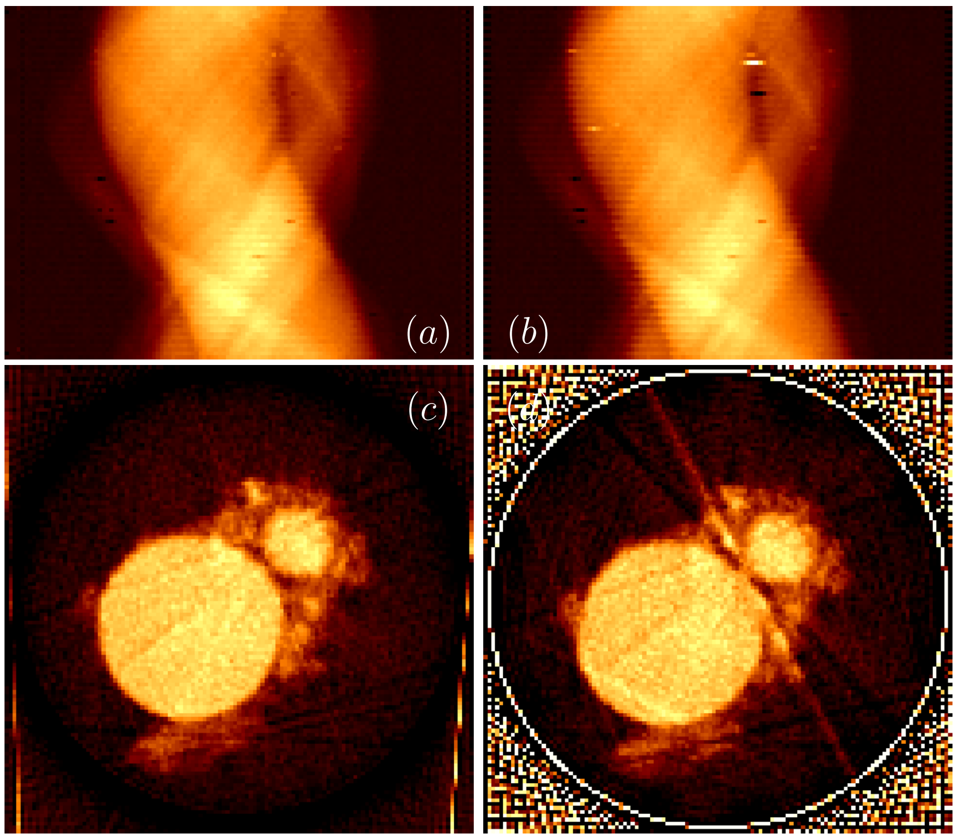


Figure 6.7: Sinograms and reconstructions of broad peak centered on channel 97.80 before and after going through our sinogram filters.

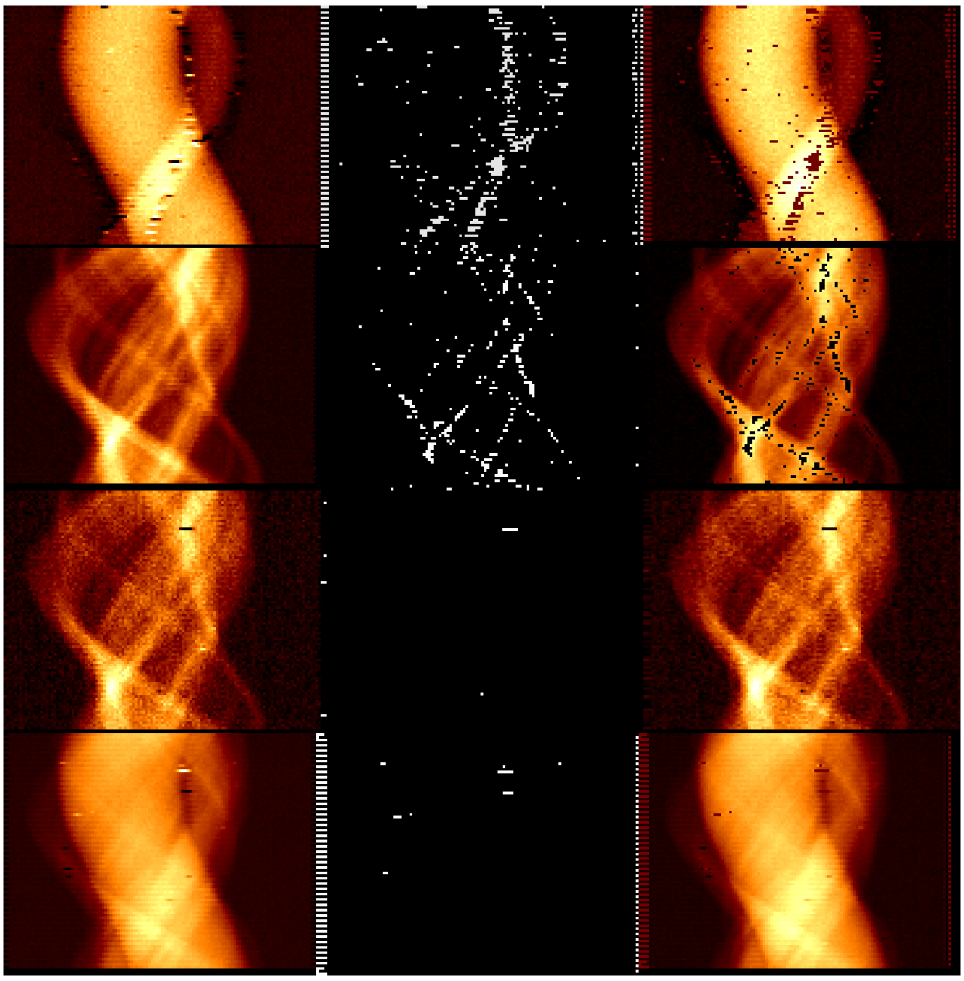


Figure 6.8: Performance of our thresholding filter 3.2.3. First column (left) shows sinograms as output by xrdtoolkit-assemble. Second column (middle) is the mask generated with default sensistivity and tolerances and (right) is the original sinogram with masked values set to 0.

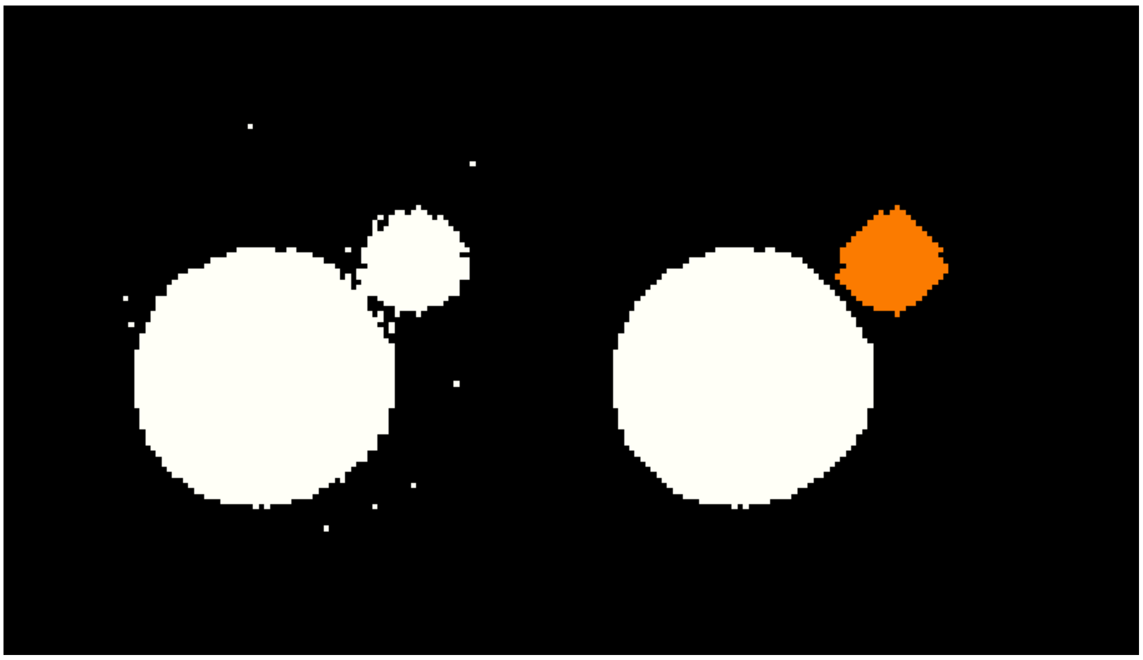


Figure 6.9: Labeling of glass phase from channel 254.88. Reconstruction was thresholded (left), then morphologically opened and labeled (right) by the `get_disc_parameters()` method in **image.py**

In figure 6.9 we see the glass phase from channel 254.88 after it has been thresholded and automatically labeled by the `get_disc_parameters()`. The two spherical shapes were found to cover 1817 and 269 pixels . The resulting shapes look nicely circular. Assuming perfect circles they correspond to diameters 18.5 and 48.1 respectively. As our resolution is $2.5\,\mu$m this corresponds to $46.2\,\mu$m and $120.2\,\mu$m.

The `get_disc_parameters()` procedure is implemented in `xrdtoolkit/image.py` and works by first thresholding the sinogram according to

$$pixel > median(image) + image.std() * 0.85.$$

The median value of the reconsctruction is assumed to be close to the null-level so we define any pixel that exceeds this value by more than 0.85 standard deviations as part of the sphere. In order to remove isolated pixels and separate the spheres we perform morphological opening on the thresholded image a number of times until desired result is obtained (default= 5). The spheres in the picture are now separated and can be labeled by numpy's `ndimage.label()` method and summed individually.

6.2 Performance

In this section we decribe the performance of the scripts developed for xrd-toolkit. All benchmarks were performed on a dedicated machine scisoft11. `scisoft11` has eight hyperthreaded cores running at 3.30 GHz which means it can service up to 16 threads of execution. Experimental data were mounted over NFS on the internal network.

6.2.1 xrdtoolkit-average

To test the performance of xrdtoolkit-average we asked it to perform median filtering on a series of 51 frames having a resolution of 2048×2048. At number of runs were made in order to make sure the dataset was in caches and we obtained consistent number. Three runs were then averaged to give a running time of 40 seconds.

At the time we were using numpy's `array()` method to assemble the outputarray. The computations were done in a list comprehension which allocates memory on the fly. This is not very efficient so we switched over to pre-allocating the final buffer and set each sub-computation manually in a `for` loop. At the same time we made sure not to promote data types to `float` if we were given less demanding data types [1]. These alterations led to a run time of 11 seconds using the same setup as above. This is still not very fast, but nearly all the time is now spent in the `median()` method of numpy.

6.2.2 xrdtoolkit-calibrate

The time needed to make a calibration is very dependent on each individual geometry. In some degenerate cases the origin of the beam might not be found even in a 100 iterations. For very off-center beams it is essential to specify an approximate initial position. As this was our case we gave xrdtoolkit-calibrate the initial coordinates $(300, 300)$ mm) and obtained a good geometry after 73 seconds and … iterations.

6.2.3 xrdtoolkit-integrate

During profiling it was found that our integration script was continuously writing to disk to update the dataset each time a new diffraction image was integrated. This lead to a lot of overhead in the `write` method of `h5py`.

[1] Our `edf` files are typically stored as 16-bit unsigned integers.

	disable-threads		disable-gpu		disable thread
	yes	no	yes	no	& gpu
coalesced	42.66	33.85	51.36	33.85	37.50
multiple writes	63.93	52.09	103.0	59.09	56.70

Table 6.2: Final benchmark of xrdtoolkit-integrate performed with local storage on 10 files with 75 frames each. All figures are in seconds.

Integrating 750 frames the original script would call this method 751 times, once for each frame and once when closing the file. After coalescing all the writes so that it only touched the file at the end we reduced the number of calls to 1 and time spent in the `write` method from $20.7s$ to $0.06s$. For this particular case that time was almost half of the total execution time. The effect of coalesced write is summarized in table 6.2.

In general this change meant that we could perform 14.6 frames per second (68.6 ms per frame) instead of 9.05 with multiple writes.

After compiling this table it became clear that our file buffer was not large enough when dealing with multi-frame `edf` files. As default we had used a few images, but with `edf` files containing 75 frames the buffer would get starved while reading the next 75 frames. We set the size of the new buffer so that at least two files would fit, regardless if they contained single or multiple frames. With this IO was no longer a bottleneck as local storage managed to keep the buffer filled at all times. GPUutilisation at this point was approaching 80% as reported by `nvidia-smi`, while there was still one python thread saturing one thread of execution. We are now possibly limited by overhead in python when invoking `pyFAI`. In a test run of $89 \times 81 = 7209$ frames we now obtained 16.6 frames per second, or 60.2 ms per frame.

6.2.4 xrdtoolkit-assemble

For testing the performance of xrdtoolkit-assemble we passed it a peak file with 9 gaussian peaks to fit. With the `--flip` and sinogram dimensions of 91×121 it took a total of 1 minute and 28 seconds, or 9 seconds per sinogram.

Chapter 7

Discussion and conclusion

7.1 experiment

We have demonstrated that the xrdtoolkit module is capable of running a Diffraction/Scattering Computed Tomography workflow giving sensible results. The diameters we found for the glass spheres of 46.2 μm and 120.2 μm are consistent with the sieve fraction $106 - 125\,\mu$m. The smaller sphere is likely a cross-section of a sphere towards the edge where the effective size is smaller.

Assembling the sinograms by peak fitting enabled us to separate phases, even with changing background and with broad amorphous scattering overlapping with the wax phase.

7.2 Corrections

Parameters of filters and were used with their default values, to demonstrate their ability to work witout user input. With some manual tweaking even higher quality reconstructions should be possible. While we can see qualitatively that the sinograms have improved by applying the de-interlacing filter it would be interesting to investigate what effect each of the centering, de-interlacing and bragg filter have on the reconstructions in isoloation.

We have shown in figures 6.4 through 6.7 that when there are a limited amount of large crystalline grains in the sample it is possible and sufficient to perform bragg peak filtering at the sinogram level. Usually each crystallite is only in a diffracting orientation through a small angle, so adjecent projections in the sinogram can be used to recover a sensible value and hence preventing bragg peaks from being backprojected and masking whole lines in the reconstruction. Moreover our filter was able to identify the majority of the anamalous projections.

For samples where some of the crystalline material matches the size of the probe another approach is likely needed. There has been some work in separating the contributions of bragg peaks from amorphous and poly-crystalline scattering by Voltoline et al. [18]. This is rather more robust, however this method work on the raw diffraction data, making it rather slow and necessary to redo all of the data reduction steps to evaluate the results.

7.2.1 Possible contamination

The bragg spots in sinogram 6.4(b) follow sinusoids that we managed to fit to equation 3.4. They are apparent as two lines in the topmost mask of the bragg peak filter in figure 6.8. This means that the disturbance corresponds to two fixed points (x, y) in the sample. We can even see this in the reconstruction 6.4(d) where the backprojected streaks seem to converge in two points on either side of the smaller sphere. It is possible that some of the aluminium foil we see in figure 5.4 has been included in our scanned area, which would account for the intense bragg diffraction we have seen.

7.3 The toolkit

Alvarez-Murga notes that Diffraction/Scattering Computed Tomography is rather simple experiment and fairly easy to handle for non-expert users. Making the processing and analysis of the experimental data easy to do should be a priority as well, since the ability to perform in-situ $3D$-resolved studies of complex materials will be of great interest to a broad community of researchers.

The Diffraction/Scattering Computed Tomography workflow we have presented is largely automated and should with some documentation be fairly easy to use to non-expert users. However, the calibration script still has some problems converging to a correct detector geometry if the beam origin is far from the center of the detector and no initial conditions or calibration limits are specified. This is currently the biggest obstacle to an unsupervised, complete analysis.

Batch-files were created to automatically average darkcurrent and integrate directories of diffraction data. The only argument to these batch files were the locations of the detector calibration file, peak file and the location of the directories containing diffraction data to be integrated. Each beamline could help by providing batch scripts that are specific to their particular types of experiment and file layout.

For the dummy experiment described in this thesis there was no need for automated scripts for batch processing, as it was only a single slice with one directory of data. Howevever, the batch processing enabled easy integration of xrd-data as they were collected for Simon Jacques who used the same setup immediately following our experiment and collected data for multiple days. In the end several terabytes of data was processed resulting in more than 4 gigabytes of reduced 1D diffraction patterns.

7.4 Script performance

The time to properly calibrate for detector tilt is very variable since it is a non-linear process. For good geometry only a few iterations might be needed, while for very off-center beams the search for origin may require many iterations. In this case the time spent searching can be minimized by providing closer initial conditions on the command line. The calibration time also heavily depends on the resolution,binning mode of the detector and calibration limits.

As we have seen in our benchmarks the azimuthal integration is more than fast enough to keep up with acquisition when the exposure is on the order of 100 ms. The time taken to average the dark current is surprisingly high due to the poor performance of numpy's median filter and ends up taking a significant amount of time. This is why we have implemented a crude queue system in the batch processing which performs the averaging and integration in parallel. Replacing the median function should bring a lot savings. The python module `Bottleneck` have an alternative implementation which is already 2.57 times faster for 1000×1000 images.

7.5 Further work

For the future of xrdtoolkitwe should look how it relates to other software. We do not aim to provide any graphical interface, plotting or as this is already provided by a several projects. Our framework of choise has been DAWN which provides a very nice and powerful interface for visualization and data exploration, with slicing and good controls for color-mapping having been very useful.

Parts of the module could be integrated in a number of different frameworks:

1. **ImageJ**[16]. The image processing suite. Corrections and filtering of sinograms could be made more interactive by making them available

as plugins for ImageJ. It uses Jython just like DAWN, so Python code can be used directly.

2. **pyMCA** is another Python based solution where calculations and filters could be integrated.
3. **DAWN** provides a mechanism for linking together workflows in a workbench and exhange data seamlessly with Python scripts, so this seems a promising direction for providing access to the xrdtoolkit-scripts in a graphical interface integrated with data exploration.
4. **pyFAI** Exchange of utility functions?

It would be useful to be able to refine the wavelength in experiments. `pyFAI` already has method for refining wavelengths from a collection of PONIfiles using constrained least squares, so given a couple of calibrations at different distances can already be done.

Another consideration is the implementation of NX-classes to conform with the Nexus format. The Nexus format is a specification for metadata built on top of `hdf5` which more and more scientific software is following. For instance is it possible to make DAWN use a dataset for plotting the diffractograms as a function of 2θ instead of channel numbers that we have used in this thesis.

The 2θ values are already included as dataset when integrating using xrdtoolkit-integrate. All that is needed is some attribute following the nexus format to specify that this should be used for the x-axis when plotting.

Appendices

Appendix A

Detector coordinate systems

Here we attempt to explain the relation between detector tilt coordinates as they are used in `f2w.py`, fit2D [9] and pyFAI [11].

The ingration routine in `f2w.py` due to Veijo Honkimaki uses projected angles α and β in radians to calculate the effective radial distance of each pixel in the diffraction image. The modified distance is given by

$$R' = R(1 - a \cdot y - b \cdot x)$$

where a and b are directly proportional to α and β.

pyFAI allows us to specify the detector tilt in fit2D notation, which we have chosen to do. The coordinates here are ξ and φ_0 which defines a tilt-plane rotated φ_0 degrees from the x-axis with a tilt of ξ degrees.

Doing the coordinate transformation we get

$$R' = R(1 - a \cdot r \sin\varphi - b \cdot r \cos\varphi).$$

with our new angles

$$a = \xi \sin\varphi_0 \tag{A.1}$$

$$b = \xi \cos\varphi_0 \tag{A.2}$$

The trigonometric identity $\cos^2\varphi_0 + \sin^2\varphi_0 \equiv 1$ then leads us to

$$a \sin\varphi_0 + a \cos\varphi_0 = \xi$$

.

Using this equation and

$$\varphi_0 = \tan^{-1}\frac{a}{b}$$

we write the detector geometry for use with pyFAI. The code doing the conversion is listed in D.3

Appendix B

Batch processing of Diffraction/Scattering Computed Tomography data

This appendix lists *bash*-scripts used for batch processing of experimental data for the Diffraction/Scattering Computed Tomography workflow as described in this report. It uses the built-in job control facilities in bash to provide process-level parallelisation.

Once the calibration file (e.g. `calibration.poni`) has been written by `xrdtoolkit-calibrate` and specified in the header of `integrate.sh` one can simply invoke the script with the paths of the directories where your diffraction images are saved. In our case we write

```
:~$ ./integrate.sh ../data/xrd/xrdtomo/dummyA1_*
```

The integrated data is then put into a separate folder `diffractograms` which can be visualised and explored using for instance DAWN [1]. The averaged dark currents are put in the `darkcurrent/` directory ready for reuse.

Once integration is done you need to specify regions of interest for `xrdtoolkit-assemble` to produce sinograms. The script accepts files on the following format:

Listing B.1: dummy_A1.peak

```
# position      fwhm     fit_width
  83.96         4.82                # from DAWN
  92.86         3.50     2.5        # from DAWN
  97.80         43.77    2.0      # from pyMCA with Internal background.
  254.88        81.72    2.0      # from pyMCA with Internal background.
```

The header on the first line needs to specify the names of the columns that follow. At least position and fwhm needs to be specified. The function shape is assumed to be gaussian if not given. For simple peaks it is sufficient to do the peak fitting in DAWN. The format DAWN uses when exporting peaks is the same as above. After creating the peak specification file and changing the name in the header of `assemble.sh` it is sufficient to invoke it with

```
:~$ ./assemble.sh
```

to perform peak fitting on all the available diffractograms. The `xrdtoolkit-assemble` script will skip any sinograms that have already been assembled. In addition to all the sinograms it will also attach all the fitted peaks along with the signal it used for the least square fitting so that the user can inspect and verify that the peaks are a good match.

For our dummy sample this is the resulting directory structure after batch processing has completed

```
results
 ├─ assemble.sh
 ├─ average_dark.sh
 ├─ calibration.poni
 ├─ integrate.sh
 ├─ peaks_dummyA1
 ├─ darkcurrent/
 │   └─ DARK_dummyA1_151.h5
 ├─ diffractograms/
 │   └─ DIFFTOMO-dummyA1_151.h5
 └─ sinograms/
     └─ SINOGRAM-dummyA1_151.h5
```

If we had more slices they would each have its own `.h5` results file in the `sinograms/` directory.

Listing B.2: xrdtoolkit/examples/batch_processing/integrate.sh

```bash
#!/bin/bash

NR_JOBS=1
PONI_FILE='180mm.poni'

[ -d diffractograms ] || mkdir diffractograms


trap control_c SIGINT

control_c()
# run if user hits control-c
```

```
{
  echo -en "\n*** Ouch! Exiting ***\n"
  exit $?
}

# Start parallell averaging of dark current
./average_dark.sh $@ &

for dir in "$@"; do
  name=$(basename $dir)
  dark=darkcurrent/DARK_$name.h5

  while [ ! -f $dark ]; do
      # Wait for dark current to be averaged
      echo $name' waiting for '$dark
      sleep 10
  done

  diffractogram=diffractograms/DIFFTOMO-$name.h5
  if [ ! -f $diffractogram ]; then
      echo "$name -- Integrating diffraction patterns"
      xrdtoolkit-integrate -p $PONI_FILE --data-path $dir/$name --dark da
  fi
done
```

Listing B.3: average_dark.sh

```
#!/bin/bash

NR_JOBS=2

[ -d darkcurrent ] || mkdir darkcurrent

control_c()
# run if user hits control-c
{
  echo -en "\n*** Ouch! Exiting ***\n"
  exit $?
}

count=0
for dir in "$@"; do
  name=$(basename $dir)
  dark=darkcurrent/DARK_$name.h5
```

```
    if [ ! -f $dark ]; then
        xrdtoolkit-average $dir/DARK_*.edf -o $dark &
    fi
    let count+=1
    [[ $((count%NR_JOBS)) -eq 0 ]] && wait # Limit to NR_JOBS concurrent jobs
done
```

Listing B.4: assemble.sh

```
#!/bin/bash

# Constructs sinograms from a list of integrated diffraction patterns
# (diffractograms).

NR_JOBS=2

PEAK_FILE=$1 #'peak_730.dat'
DO_FLIP=true

[ -d sinograms ] || mkdir sinograms

trap control_c SIGINT

control_c()
# run if user hits control-c
{
  echo -en "\n*** Ouch! Exiting ***\n"
  exit $?
}

count=0
for diffractogram in "$@"; do
  echo $diffractogram
  basename=$(basename $diffractogram .h5)
  name=SINOGRAM-$(echo $basename | sed s/DIFFTOMO-//)
  args="-o sinograms/$name $diffractogram.h5"
  if [ $DO_FLIP ]; then
      args="--flip $args"
  fi
  echo "Assembling $basename using $PEAK_FILE"
  xrdtoolkit-assemble --peak-file $PEAK_FILE $args &
  let count+=1
  [[ $((count%NR_JOBS)) -eq 0 ]] && wait # Limit to NR_JOBS concurrent jobs
done
```

```
36
37  wait
```

Appendix C

The xrdtoolkit module

Listing C.1: xrdtoolkit/__init__.py

```
IMAGE_PATH = '/entry/image' # default path used by DAWN
AVERAGE_DATA_SET = '/xrdtoolkit/average'
DARKCURRENT_DATA_SET = '/xrdtoolkit/darkcurrent'
TWO_THETA_DATA_SET = '/xrdtoolkit/two_theta'

# Calibration
CALIBRATION_IMAGE = '/xrdtoolkit/calibration_image'
CALIBRATION_PROFILE = '/xrdtoolkit/calibration_profile'

# DIffractogram
DIFFRACTOGRAM_DATA_SET = '/xrdtoolkit/diffractogram'

# Sinograms
SINOGRAM_GROUP = '/xrdtoolkit/sinogram'
SINOGRAM_PEAK_GROUP = '/xrdtoolkit/fitted_peaks'
CORRECTED_SINOGRAM_GROUP = '/xrdtoolkit/sinogram_corrected'
RECONSTRUCTION_GROUP = '/xrdtoolkit/reconstructed'
```

Listing C.2: xrdtoolkit/image.py

```
from scipy.fftpack import fft, ifft, fft2, ifft2
from scipy import optimize
from scipy import fftpack
from scipy import ndimage

import numpy as np

def get_disc_parameters(image, iterations=5, std=0.85):
    image = image > np.median(image) + image.std()*std

    # Perform morphological opening of image. This has the effect of sepa
    # discs, turning them more spherical and removing outliers.
    image = ndimage.binary_opening(image, iterations=iterations)
    labels,circles = ndimage.label(image)
    # Calculate diameter of each circle from its area, assuming perfect
    # circular shapes.
    area =     [ (labels == i).sum() for i in xrange(1,circles+1) ]
    diameter = [ np.sqrt(float(a) / np.pi) * 2.0 for a in area ]

    return dict({ 'area':area, 'diameter': diameter, 'circles': circles, '

# Code taken and adapted from
# https://github.com/eddam/python-esrf/blob/master/rotation_axis.py
# due to Emmanuelle Gouillart

def _correlate_images(im1, im2, method='brent'):
    shape = im1.shape
    f1 = fft2(im1)
    f1[0, 0] = 0
    f2 = fft2(im2)
    f2[0, 0] = 0
    ir = np.real(ifft2((f1 * f2.conjugate())))
    t0, t1 = np.unravel_index(np.argmax(ir), shape)
    if t0 >= shape[0]/2:
        t0 -= shape[0]
    if t1 >= shape[1]/2:
        t1 -= shape[1]

    median2 = np.median(im2)

    def cost_function(s, im1, im2):
        return - np.corrcoef([im1[3:-3, 3:-3].ravel(),
                              ndimage.shift(im2, (0, s),mode='nearest',cval=
    if method == 'brent':
        newim2 = ndimage.shift(im2, (t0, t1),mode='nearest',cval=median2)
        refine = optimize.brent(cost_function, args=(im1, newim2),
                          brack=[-1, 1], tol=1.e-2)
    return t1 + refine

def _correlate_projections(proj1, proj2, method='brent'):
    shape = proj1.shape
    f1 = fft(proj1)
    f1[0] = 0
    f2 = fft(proj2)
    f2[0] = 0
    ir = np.real(ifft((f1 * f2.conjugate())))
    (t0,) = np.unravel_index(np.argmax(ir), shape)
    if t0 >= shape[0]/2:
        t0 -= shape[0]

    median2 = np.median(proj2)

    def cost_function(s, proj1, proj2):
        cost = - np.corrcoef([proj1, ndimage.shift(proj2,s,mode='nearest',cval=me
        return cost

    if method == 'brent':
        newproj2 = ndimage.shift(proj2, (t0,),mode='nearest',cval=median2)
        refine = optimize.brent(cost_function, args=(proj1, newproj2),
                                brack=[-1, 1], tol=1.e-5)

    return t0 + refine
```

Listing C.3: xrdtoolkit/tomo.py

```
from skimage import filter as filters
from scipy import ndimage

import numpy as np

import utils, image

def sino_remove_bragg_spots(sinogram, block_size=5, tolerance=0.05, sensitivity_lo
    """ If value is above some local threshold,
        replace by median. Removes dodgy highlights and shadows
        resulting from bragg peaks from large crystallites
        in diffracting orientations """

    # Footprint for median value to replace bragg spots.
    # Usually the spots are contained to one projection,
    # so we sample above and below for good values.
    footprint = np.array(
        [[  False, True, False ],
          [  True,  True,  True ],
          [  False, False, False ],
          [  True,  True,  True ],
              [   False,  True,  False  ]])

    # Only consider pixels which differ from the local median by this offs
    # Highlights and shadows will skew the arithmetic mean so use median.

    median_value = np.median(sinogram)
    offset_high  = np.median(sinogram[sinogram>median_value])
    offset_low   = np.median(sinogram[sinogram<median_value])

    utils.debug_print(median=median_value, offset_high=offset_high, offset_

    mask_low = ~filters.threshold_adaptive(
                    sinogram,
                    block_size,
                    method='median',
                    offset=-sensitivity_low*(offset_low-median_value),
                )
    mask_high = filters.threshold_adaptive(
                    sinogram,
                    block_size,
                    method='median',
                    offset=-sensitivity_high*(offset_high-median_value),
                )
    if float(mask_high.sum()) > tolerance * mask_high.size:
        # Too many values marked as spots. Ignoring hilights.
        print('Found more than %s%% of values as hilights' % (tolerance *
        mask_high = np.zeros(shape=sinogram.shape, dtype=bool)
    if float(mask_low.sum()) > tolerance * mask_low.size:
        # Too many values marked as spots. Ignoring shadows.
        print('Found more than %s%% of values as shadows' % (tolerance * 1
        mask_low = np.zeros(shape=sinogram.shape, dtype=bool)

    mask = mask_low + mask_high
    # FIXME, only calculate values in mask.
    median = ndimage.median_filter(sinogram, footprint=footprint)
    ret = sinogram.copy()
    ret[mask==True] = median[mask==True]
    return ret


def sino_deinterlace(sinogram):
    sino_deinterlaced = sinogram.copy()
    sino_even = sinogram[::2,...]
    sino_odd = sinogram[1::2,...]
    if sino_even.shape > sino_odd.shape:
        shift = image._correlate_images(sino_even[:-1,...], sino_odd)
    else:
        shift = image._correlate_images(sino_even, sino_odd)

    sino_deinterlaced[1::2,...] = ndimage.shift(sinogram[1::2,...],(0,shift),mode
    return sino_deinterlaced

def sino_center(sinogram):
    """ Finds rotation axis of sinogram by using first and last projections
    which are assumed to be 180 degrees apart. Last projection is reversed and
    correlated with the first and the shifted image with rotation axis in
    center is returned. """

    proj1 = sinogram[0,...]
    proj2 = sinogram[-1,::-1]
    shift = image._correlate_projections(proj1, proj2)
    return ndimage.shift(sinogram, (0,-shift), mode='nearest',cval=np.median(sino
```

Listing C.4: xrdtoolkit/fit.py

```
import numpy as np

import utils

SQRT2 = 1.41421356237309504880

GAUSSIAN  = 'gaussian'
DELTA     = 'delta'
GAUSS_ERF = 'erf'

def erf(x):
    a1 =  0.254829592; a2 = -0.284496736
    a3 =  1.421413741; a4 = -1.453152027
    a5 =  1.061405429; p  =  0.3275911

    if not type(x) == np.ndarray:
        x = np.array(x)
    sign = np.ones(x.shape)
    sign[x<0] = -1
    x = np.abs(x)

    # A & S 7.1.26
    t = 1.0/(1.0 + p*x)
    y = 1.0 - (((((a5*t + a4)*t) + a3)*t + a2)*t + a1)*t*np.exp(-x*x)

    return sign*y


PEAK_FUNCTIONS = {
        GAUSSIAN   : lambda x,x0,sigma: 1.0/(np.sqrt(2*np.pi)*sigma)*np.exp
        DELTA      : lambda x,x0: 1 if  x == x0 else 0,
        GAUSS_ERF  : lambda x,x0,sigma: 0.5*(erf( (x+0.5-x0) / (SQRT2*sigma
                                       erf( (x-0.5-x0) / (SQRT2*sigma)))
}

def get_peak_function(position=0, fwhm=1, shape=DELTA, **kwargs):
    if shape == GAUSSIAN:
        sigma = fwhm / 2.35482004503  # FWHM = 2 sqrt(2 ln(2) ) sigma
        if fwhm < 10:
            # Few points, so we need to use proper quadrature
            shape = GAUSS_ERF
    elif shape == DELTA:
        return lambda x: PEAK_FUNCTIONS[shape](x,position)

    return lambda x: PEAK_FUNCTIONS[shape](x,position,sigma)

def fit_peak_intensity(signal, peak, bias=1):
    """
    """

    try:
        assert(signal.shape == peak.shape)
    except:
        raise Exception("Arguments must be compatible vectors.")

    A = np.array([np.ones(signal.size), np.arange(0,signal.size), peak])
    w = 1.0 / (signal+bias) # Use signal as squared error. dy ~ sqrt(y)
    B = A * np.array([w,w,w])

    covC = np.linalg.inv(np.dot(B,A.T))
    c = np.dot(np.dot(covC, B), signal)

    print c

def test_signal(x,c,fun,rand=0):
        signal =np.dot(np.array([np.ones(len(x)), x, fun]).T,c)
        signal = signal + rand*np.random.normal(0,2,signal.size)
        return signal

def test(N=21,(xmin,xmax)=(-10,10),c=[13.45,1.23,49.0],peak_width=2.2):
        x = np.linspace(xmin,xmax,N)
        #c = np.array([13.45, 1.23, 49.0]) # Constant, slope and peak magnitude
        signal = test_signal(x, c, gaussian(x,0,peak_width),2)
        peak = gaussian(x,0,peak_width)
        fit_peak_intensity(signal,peak)
```

Listing C.5: xrdtoolkit/f2w.py

```
#
# f2w-package by V. Honkim\"aki
#
# Copyright 2012 European Synchrotron Ratiation Facility
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.

#  example:
#   >>> import edf                 # or however you load the edf-files
#   >>> import f2w                 # import the detector classes
#   >>> Det=f2w.Pixium()           # object for the Pixium detector
#   >>> print(Det)                 # ...just to see what parameters it holds
#   >>> P=edf.read("pix.edf")      # reading the 2d pattern from the edf-file
#   >>> Det.calibrate(P,[10,130]) # calibrating the center and the tilt using
#                                  # all the data between 10mm < radius < 130mm
#   >>> r,A=Det.integrate(P,1)     # integrates over the whole 2pi
#   >>> r,A=Det.integrate(P,N)     # integrates N pies
#
import numpy as np
from numpy import zeros, pi, meshgrid, arange, sqrt, arctan2, isnan, floor, prod,
                    trunc, nonzero, diff, hstack, vstack, int_, transpose,\
                    linalg, dot, sin, cos, cumsum, ones

```

```
import utils

class Detector(object):
    _T = []
    _R = []
    _dr = []
    _updated   = False
    """A general detector object"""
    def __init__(self, **kwargs):
        for key in kwargs:
            if key in ['distance','tilt', 'origin', 'binning']:
                self.__dict__.update({'_%s' % (key,) : utils.flatten(kwarg
        if 'binning' in kwargs:
            self._pixelsize = list(np.multiply(self._pixelsize, self._binn
            self._pixels    = list(np.divide(self._pixels, self._binning))
    def setdist(self,D):
        self._distance = D; self._updated = False;
    def setorigin(self,v):
        self._origin = v; self._updated = False;
    def settilt(self,v):
        self._tilt = v; self._updated = False;
    def __str__(self):
        s = 'Distance = ' + repr(self._distance) + 'mm\nOrigin
  = ' + repr(self._origin) \
          + 'mm\nTilt     = ' + repr(self._tilt) + 'deg\nPixels
  = ' + repr(self._pixels)\
          + '\nPixsize  = ' + repr(self._pixelsize) + 'mm'
        if 'binning' in self.__dict__:
            s = s + '\nBinning mode = ' + repr(self._binning)
        return(s)
    def _calcrt(self):
        if (not self._updated):
           r = pi/180.0/self._distance
           a = self._tilt[0]*r; b = self._tilt[1]*r;
           xv = arange(self._pixels[1])*self._pixelsize[1]-self._origin[1]
           yv = arange(self._pixels[0])*self._pixelsize[0]-self._origin[0]
           x,y = meshgrid(xv,yv)
           self._R = sqrt(x**2+y**2)*(1.0-a*y-b*x); self._T = arctan2(y,x)
           self._R.shape = prod(self._R.shape); self._T.shape = prod(self.
           self._Rind = self._R.argsort(axis=None);
           n = cumsum(ones(self._Rind.shape));
           self._dr = sqrt(self._pixelsize[0]*self._pixelsize[1])
           i = floor(self._R[self._Rind]/self._dr + 0.5)
           self._jind = 0 < diff(i);
```

```
            self._nR = i[self._jind]*self._dr;
            self._dcj = diff(hstack((0,n[self._jind])))
    def integrate(self,Im,n=1):
        if Im.shape != tuple(self._pixels):
            raise Exception("Diffraction image does not match detector resolution
        if (not self._updated):
            self._calcrt();
        Imc = Im[:]; Imc.shape = prod(Imc.shape);
        if n>1:
            R = self._R[:]; T = self._T[:]; dr = self._dr;
            tpi = 2.0*pi; dp = tpi/n; ip = int_(floor(T/dp+0.5)%n); A = zeros([0,
            for i in range(n):
                j = nonzero(ip == i); a = self._pie(Imc[j],R[j],dr); M = a.shape[0
                if (m < M):
                    A = vstack((A,zeros([M-m,n])))
                a.shape = M; A[:M,i] = a;
        else:
            A = zeros(self._nR.shape); c = Imc[self._Rind].cumsum()[self._jind];
            A[0] = c[0]; A[1:] = diff(c); A = A/self._dcj;
        return(self._nR,A);

    def _pie(self,Im,R,dr):
        mn = prod(Im.shape); j = R.argsort(axis=None); w = R[j]/dr;
        ir = int_(w); w0 = 1.0+ir-w; w1 = 1.0-w0; c0 = w0*Im[j]; c1 = w1*Im[j];
        i = isnan(Im[j]); w0[i] = 0; w1[i] = 0; c0[i] = 0; c1[i] = 0;
        w0 = w0.cumsum(); w1 = w1.cumsum(); c0 = c0.cumsum(); c1 = c1.cumsum();
        i = nonzero(diff(ir)); m = ir[-1]+2; A = zeros([m,1]); C = zeros([m,1]);
        ta = diff(hstack((0,c0[i]))); tc = diff(hstack((0,w0[i]))); j = ir[i]; A[j
        ta = diff(hstack((0,c1[i]))); tc = diff(hstack((0,w1[i]))); j += 1; A[j,0]
        j = nonzero(C); A[j] = A[j]/C[j]; return(A);
    def calibrate(self,Im,rg, drk=None):
        """
            Im  : Raw Image to use for calibration.
            rg  : Region (min,max) in mm to include in calibration.
            drk : If provided, the dark current image will be subtracted
                  from the raw image and its variance added to the error statistic

            N   - number of pies to integrate
            db  - covariance matrix of c, weighted for high q-counts and intensity
            rg  - Range [mm] to use for calibration
        """

        if drk is not None:
            self._bias = drk.mean()
```

```
             Im = Im.astype('float') - drk # Small negative values in uint16
        iterations = 0; full_iterations = 0;
        D = self._distance; stp = 1;  N = 36; dp = 2*pi/N; p = arange(N)*dp
        y = zeros([N,1]); z = zeros([N,1]); dy = zeros([N,1]); dz = zeros([
        sc = 2*pi/(sqrt(self._pixelsize[0]*self._pixelsize[1])*N);
        while (0.001 < stp and full_iterations < 20):
            r,A = self.integrate(Im,N); i = nonzero((rg[0] < r)*(r < rg[1]))
            d = diff(A[:,-1])/diff(r); C = vstack((A[:-1,-1],d,d*r[-1:]**2/D
            for j in range(N):
                w = sc*r[:-1]/(A[:-1,j]+self._bias); w[w<0] = 0;
                Cs = (C*w[:,[0,0,0]]).T; db = linalg.inv(dot(Cs,C));
                b = dot(db,dot(Cs,A[:-1,j])); y[j] = b[1]/b[0]; z[j] = b[2]/b
                c = hstack((1,-y[j]))/b[0]; c.shape = [1,2]; dy[j] = dot(c,do
                c = hstack((1,-z[j]))/b[0]; c.shape = [1,2]; dz[j] = dot(c,do
                d = diff(A[:,j])/diff(r); C = vstack((A[:-1,j],d,d*r[:-1]**2/

            y = y/dp; z = z/dp; dy = dy/dp**2; dz = dz/dp**2;

            # Update origin
            w = (1/dy);
            C = vstack((cos(p),-sin(p))).T; Cs = (C*w[:,[0,0]]).T;
            db = linalg.inv(dot(Cs,C)); c = dot(db,dot(Cs,y));
            stp = sum(c**2); q = c/sqrt(stp); stp = stp/dot(dot(q.T,db),q);
            self.setorigin(self._origin - c);

            # Update tilt.
            # Tilt is normally very small, so wait until origin is rather st
            stp2=0
            if stp < 1:
              w = (1/dz);
              C = vstack((cos(p),-sin(p))).T; Cs = (C*w[:,[0,0]]).T;
              db = linalg.inv(dot(Cs,C)); c = dot(db,dot(Cs,z));
              stp2 = sum(c**2); q = c/sqrt(stp2); stp2 = stp2/dot(dot(q.T,db
              self.settilt(self._tilt - c*180/pi);
              full_iterations += 1

            stp = sqrt(stp+stp2)
            iterations += 1
            print("Step = {0:.3f}".format(stp[0,0]))
            print('Iteration %s' % (iterations,))
            print self

class Pixium(Detector):
    """Pixium detector object"""
    def __init__(self, **kwargs):
       """ Set up default geometry and
           allow it to be overriden in base __init__ """
        self._distance  = 1000
        self._origin    = [147.84,203.28]
        self._tilt      = [0,0]
        self._pixels    = [1920,2640]
        self._pixelsize = [0.154,0.154]
        self._bias      = 3378.4108576774597 # Average dark current.
        super(Pixium,self).__init__(**kwargs)

class Perkin(Detector):
    """ detector object"""
    def __init__(self, **kwargs):
        self._distance  = 1000
        self._origin    = [204.8,204.8]
        self._tilt      = [0,0]
        self._pixels    = [2048,2048]
        self._pixelsize = [0.200,0.200]
        self._bias      = 3378.4108576774597 # Average dark current.
        super(Perkin,self).__init__(**kwargs)

def get_detector(name, **kwargs):
    """ Detector name to object translation.
        Based on similar method in pyFAI.  """
    detectors = {"perkin": Perkin,
                 "pixium": Pixium, }
    _name = name.lower()
    if _name in detectors:
        return detectors[_name](**kwargs)
    else:
        raise Exception('Detector %s not known.' % (name,))

class Calibrator(object):
    def __init__(self, image, dark_current, detector):
        self.image = image
        self.detector = detector
        self.dark_current = dark_current

    def calibrate(self, limits=[10,350]):
        """ Calibrate tilt and origin of detector using data in the interval
        [@lower,@upper]mm with a value exceeding @threshold.
        If pixel_limits=True, lower and upper limits are given in pixels. """
        # FIXME default limits to fraction of diffraction detector
        #self.image[self.image<100] = 0
        self.detector.calibrate(self.image, limits, drk=self.dark_current)

    def __str__(self):
        return str(self.detector)
```

Listing C.6: xrdtoolkit/common.py

```
import pickle
import numpy as np
from pkg_resources import resource_string

class Bunch:
    def __init__(self, **kwds):
        self.add(**kwds)
    def add(self, **kwds):
        self.__dict__.update(kwds)
    def __repr__(self):
        return unicode(self)
    def __unicode__(self):
        format_str = u": %s\n".join(self.__dict__.keys())
        format_str +=u": %s\n"
        return format_str % tuple(self.__dict__.values())

#
#>>> import xrdtoolkit
#>>> xrdtoolkit.XrayTable.dtype # list all available fields
#>>> xrdtoolkit.Xraytable['Density'][1] # density of Hydrogen
#>>> xrdtoolkit.Xraytable[1]['Density'] # also density of Hydrogen
#
# Please note that numpy's dot function for matrix multiplication does _no
# work with scipy sparse matrices (such as the JumpMatrix). Use jumpMatrix
# instead.

class XrayTable:
    """ XrayTable contains various data related to
    interactions with X-rays for a range of elements. """

    def __init__(self):
        self.loaded = False

    def _load(self):
        self.table = pickle.loads(resource_string(__name__,'data/xraytable

    def __getitem__(self,key):
        if not self.loaded:
            self._load()
        if not isinstance(key,int):
            raise IndexError('First index must be an atomic number')
        if key > self.table.shape[0] or key < 1:
            raise IndexError('Invalid atomic number')
        return self.table[key-1]

    def __repr__(self):
        """ Give a listing of available data """
        if not self.loaded:
            self._load()
        return '\n'.join(self.table.dtype.names)

    def __str__(self):
        if not self.loaded:
            self._load()
        return self.__repr__()

# Replace class by singleton instance
# Loads xray data once.
XrayTable = XrayTable()

Elements = pickle.loads(resource_string(__name__,'data/elements.pickle'))



Constants = Bunch(
    c  = 299792458,        # (m/s) speed of light in vacuum (exact)
    mu =   4.0e-7*np.pi,   # (N/A^2) magnetic constant mu_0 (exact)
    Na =   6.0221415e23,   # Avogadro constants
    kB =   1.3806505e-23,  # (J/K) Boltzmann constant
    h  =   6.6260693e-34,  # (Js) Planck constant
    G  =   6.6742e-11,     # (m^3/kg/s^2) gravitational constant
    e  =   1.60217653e-19, # (J) electron volt
    me = 510.998918        # electron mass in keV
)

Constants.add(
    ep = 1/(Constants.mu*Constants.c**2),          # (F/m) electric constant eps_0
    hc = 1e7*Constants.h*Constants.c/Constants.e,          # (keV A) hc
    re = Constants.c**2*Constants.e/Constants.me,          # classical radius of e
    ia = 2*Constants.h/(Constants.e**2*Constants.mu*Constants.c)
# 1/fine-structure
)

def strtoz(string):
    ''' Converts a string of stoichiometry into a dictionary
        of corresponding atomic numbers and element count.
        E.G H2O -> { ('H':1):2, ('O':8):1 }
'''
    string = string.replace('Air', 'N4O')
    return _strtoz(string)


def _strtoz(string):
    try:
        return { int(string) : 1 }
    except ValueError:
        # Not an integer
        (leftBracket,rightBracket) = _getmatchedparentheses(string)
        if leftBracket != -1:
            # string is of form a(b)c
            a_z = _strtoz(string[0:leftBracket])
            b_z = _strtoz(string[leftBracket+1:rightBracket])

            c = string[rightBracket+1:]
            factor = 1
            if len(c):
                try:
                    factor = int(c[0])
                    c = c[1:]
                except ValueError:
                    # No integer followed parenthesis.
                    pass
            c_z = _strtoz(c)
            for element,count in b_z.items():
                a_z[element] = a_z.get(element,0) + factor*count
            for element,count in c_z.items():
                a_z[element] = a_z.get(element,0) + count
            return a_z
        else:
            # string with only elements and integers
            result = dict()
            while string:
                count = 1
                if len(string) == 0:
                    continue
                if len(string) > 1 and string[1].islower():
                    el = (string[0:2],Elements[string[0:2]])
                    string = string[2:]
                else:
                    if not string[0] in Elements.keys():
                        raise Exception('Invalid element %s' % string[0])
                    el = (string[0],Elements[string[0]])
                    string = string[1:]
                try:
                    count = int(string[0])
                    string = string[1:]
                except (IndexError,ValueError):
                    # No count given
                    pass

                result[el] = result.get(el,0) + count
            return result
    raise Exception('FIXME')

def _getmatchedparentheses(string):
    balance = 0
    left = string.find('(')
    index = left
    for char in string[left:]:
        if char == '(':
            balance += 1
        elif char == ')':
            balance -= 1
        if balance == 0:
            return (left,index)
        index += 1
    if balance > 0:
        raise ValueError('Unmatched ( parenthesis')
    elif balance < 0:
        raise ValueError('Unmatched ) parenthesis')
    return (left,index)
```

Listing C.7: xrdtoolkit/utils.py

```
import numpy as np
import optparse,os, time, inspect

import xrdtoolkit, fabio
```

```
from xrdtoolkit import files

def debug_print(**kwargs):
    for name,var in kwargs.iteritems():
        print('%s = %s' % (name,var,))

def strip_none_values(dictionary):
    return dict([ (o,v) for o,v in dictionary.items() if not (v == None or
[None])])

def flatten(val):
    if type(val) == dict:
        return dict([ (o,flatten(v)) for o,v in val.items()])

    if hasattr(val, '__iter__') and len(val) == 1:
        return val[0]

    if hasattr(val, '__iter__'):
        flat_list = []
        for item in val:
            if hasattr(item, '__iter__'):
                flat_list.extend(flatten(item))
            else:
                flat_list.append(item)
        return flat_list
    else:
        return [val]

def convert(val,val_type):
    if type(val) == dict:
        return dict([ (o,convert(v,val_type),) for o,v in val.items() ])
    if hasattr(val, '__iter__'):
        return [ convert(o,val_type) for o in val ]
    try:
        return val_type(str(val).strip())
    except ValueError:
        return val

class Script(object):
    def __init__(self):
        self.usage = "Usage: %prog <options>"
        self.description=""
        self.timings = []

    def parser_setup(self):
        parser = optparse.OptionParser(usage=self.usage, description=self.descript

        parser.add_option("-V", "--version", dest="version", action="store_true",
                          help="print version of the program and quit", metavar="
        parser.add_option("-v", "--verbose",
                          action="store_true", dest="verbose", default=False,
                          help="switch to debug/verbose mode")
        parser.add_option("-s", "--silent",
                          action="store_true", dest="silent", default=False,
                          help="supress output to terminal.")
        parser.add_option("-t", "--timings",
                          action="store_true", dest="timings", default=False,
                          help="Report execution times.")
        self.parser = parser

    def parse(self):
        (self.options, self.args) = self.parser.parse_args()

    @classmethod
    def timed(cls, fun):
        def wrapper(self, *args, **kwargs):
            start = time.time()
            fun(self, *args, **kwargs)
            self.timings.append((fun.__name__,time.time()-start),)
        return wrapper

    def print_timings(self):
        if not self.options.timings:
            return
        print("=== Execution time ===")
        for (i,j) in self.timings:
            print('%s: %.2fs' % (i.replace('_',' ').capitalize(),j))


    def print_verbose(self, *args, **kwargs):
        if self.options.verbose or not self.options.silent:
            if 'indent' in kwargs:
                for i in xrange(0, kwargs['indent']):
                    print "    ",
            for arg in args:
               print arg,
            print
```

Listing C.8: xrdtoolkit/files.py

```
import os
import numpy as np

import fabio

import xrdtoolkit

EDF = '.edf'
HDF5 = '.h5'
IMAGE_EXTENSIONS = [ EDF, HDF5 ]

def matchImageFiles(path):
    """ Return list of images starting with given path """

    (directory,file_prefix) = os.path.split(os.path.expanduser(path))
    if directory == '':
        directory = '.'
    file_names = [ o for o in os.listdir(directory) if o.startswith(file_p
    return (file_prefix, directory, file_names)

def saveDataset(file_handle, data, data_set='/entry/image'):
    group = file_handle.require_group(os.path.dirname(data_set))
    dataset = group.require_dataset(
                    name=os.path.basename(data_set),
                    shape=data.shape,
                    dtype=data.dtype
                    )
    dataset[:] = data

class ImageFile:
    def __init__(self,file_path):
        self.extension = os.path.splitext(file_path)[1]
        self.file_path = file_path

    def getNFrames(self):
        if self.extension == HDF5:
            return 1
        elif self.extension == EDF:
            return fabio.open(self.file_path).nframes

    def getImage(self, data_set='/entry/image'):
        if self.extension == HDF5:
            import h5py
            with h5py.File(self.file_path) as f:
                try:
                    self.image = f[data_set].value
                except KeyError:
                    if xrdtoolkit.IMAGE_PATH not in f:
                        raise KeyError('Data set %s does not exist' % (data_set,)
                    else:
                        self.image = f[xrdtoolkit.IMAGE_PATH]
        else:
            import fabio
            self.image = fabio.open(self.file_path).data
        return self.image

    def saveImage(self, image, data_set='/xrdtoolkit/image'):
        if self.extension == '.h5':
            import h5py
            with h5py.File(self.file_path) as f:
                saveDataset(f, image, data_set)
        elif self.extension == '.edf':
            import fabio
            edf_image = fabio.edfimage.edfimage()
            if image.ndim == 3:
                edf_image.setData(image[0])
                for i in xrange(1,image.shape[0]):
                    edf_image.appendFrame(data=image[i])
            elif image.ndim > 3:
                raise RuntimeError("Number of dimensions greater than 3.")
            else:
                edf_image.setData(image)
            edf_image.write(self.file_path)

def ImageSequence(file_paths, data_set=xrdtoolkit.IMAGE_PATH, group_frames=False)
    if all([os.path.splitext(file_name)[1] == '.edf' for file_name in file_paths]
        edf_image = fabio.edfimage.edfimage().read(file_paths[0])
        nframes = edf_image.nframes
        if nframes > 1:
            for f in file_paths:
                edf_image = fabio.open(f)
                if group_frames:
                    ret = np.zeros([edf_image.nframes] + edf_image.dims)
                    for i in xrange(0,edf_image.nframes):
                        ret[i] = edf_image.getframe(i).data
                    yield ret
                else:
                    for i in xrange(0,edf_image.nframes):
                        yield edf_image.getframe(i).data
        else:
            for f in file_paths:
                yield edf_image.fastReadData(f)

    else:
        for f in file_paths:
            yield ImageFile(f).getImage(data_set)


def averageImages(file_paths, method='median', flatten=False):
    """ Load and average a list of images.
        By default multi-frame files maintain their shape,
        that is, frames are averaged across files and not
        over internal frames"""
    if not hasattr(file_paths, '__iter__'):
        file_paths = [ file_paths ]
    file_paths = [ f for f in file_paths if os.path.isfile(f)]
    if len(file_paths) == 0:
        raise Exception("No valid files to average")

    img = ImageFile(file_paths[0]).getImage()
    dtype = img.dtype
    nframes = ImageFile(file_paths[0]).getNFrames()
    image_dims = tuple(img.shape)
    image_count = len(file_paths)

    edf_files = [ fabio.open(path) for path in file_paths ]

    if not flatten:
        res = np.zeros((nframes,) + image_dims, dtype=dtype)
        image_stack = np.zeros((image_count,) + image_dims, dtype=dtype)
        for i in xrange(0,nframes):
            print 'Averaging frame %s' % i
            for j in xrange(0,image_count):
                if nframes == 1:
                    image_stack[j] = edf_files[j].data
                else:
                    image_stack[j] = edf_files[j].getframe(i).data
            if method == 'median':
                res[i] = np.median(image_stack,axis=0).astype(dtype)
            elif method == 'mean':
                res[i] = np.mean(image_stack,axis=0).astype(dtype)
            else:
                raise Exception('METHOD NOT IMPLEMENTED')
    else:
        image_stack = np.array([o for o in ImageSequence(file_paths)])
        if method == 'median':
            res = np.median(image_stack,axis=0).astype(dtype)
        elif method == 'mean':
            res = np.mean(image_stack,axis=0).astype(dtype)
        else:
            raise Exception('METHOD NOT IMPLEMENTED')

    return res.squeeze()
```

Listing C.9: xrdtoolkit/sample.py

```
import common, crossection
import pickle
import numpy as np

from common import Constants

class Sample:
    fields = ['thickness','density','compound','chi']
    units  = ('cm'        ,'g/cm^3' ,''          ,u'\u00B0')
    def __init__(self, thickness, density, compound, chi):
        """ Sample with thickness in cm, density in g/cm^3
            compound as a string and sample angle chi in degrees """
        self.thickness = thickness
        self.density = density
        if isinstance(compound,basestring):
            self.compound = (compound,common.strtoz(compound))
        else:
            self.compound = compound
        self.chi = chi
    def __unicode__(self):
        format_str = u": %s%%s\n".join([ o.capitalize() for o in Sample.fields ])
        format_str +=u": %s%%s\n"
        return format_str % tuple(self._attribute_list()) % Sample.units
    def __str__(self):
        return unicode(self).encode('utf-8')
    def _attribute_list(self):
        return [getattr(self,o) for o in Sample.fields]

    # Computed quantities of compund

    def mass_attenuation(self,E):
        """ Mass attenuation coefficient cm^2 g^-1. """
        total_cross_section, cumm_density, total_mass = _weighted(crossect
        return total_cross_section*(Constants.Na*1e-24)/total_mass

    def form_factor(self,q):
        """ Calculates the form factor of sample given
            scattering vector q = 4*pi*sin(Theta)/lambda """
        def interpolate():
            """ FIXME """
            return 1
        form_factor = _weighted(interpolate)(q)
        return form_factor

def _weighted(fun):
    """ Decorator for calculations which are to be weighted
        according to sample stoichiometry. Returns a function
        which calls fun for each element in the sample. """
    def wrapped_fun(sample,*_args,**_kwargs):
        Z = [ o[1] for o in sample.compound[1].keys() ]
        weights = sample.compound[1].values()
        res = 0
        for i in xrange(0,len(weights)):
            res += weights[i]*fun(z=Z[i],*_args,**_kwargs)
        return res
    return wrapped_fun
```

Listing C.10: xrdtoolkit/bxtal.py

```
from numpy import arcsin, array, cos, cross, exp, nonzero, ones, \
                 pi, sin, sqrt

def elcom(hkl,ver,chi):
    ''' FIXME: docstring '''
    y = cross(ver,hkl); y /= sqrt(sum(y**2));
    z = array(hkl); z = z/sqrt(sum(z**2));
    c = cos(pi*chi/180); s = sin(pi*chi/180);
    hz = z[0]*c-y[0]*s; kz = z[1]*c-y[1]*s; lz = z[2]*c-y[2]*s;
    hy = z[0]*s+y[0]*c; ky = z[1]*s+y[1]*c; ly = z[2]*s+y[2]*c;

    s11 = 7.68; s12 = -2.14; s44 = 12.6; sa = s11-s12-s44/2;

```

```python
    s33 = s12+s44/2+(hz**4+kz**4+lz**4)*sa;
    s23 = s12+(hz**2*hy**2+kz**2*ky**2+lz**2*ly**2)*sa;
    s34 = 2*(hz**3*hy+kz**3*ky+lz**3*ly)*sa;
    return(s33,s23,s34);

def bw(hkl,ver,chi,E,T,D):
    ''' FIXME: docstring '''
    s33,s23,s34 = elcom(hkl,ver,chi); d = 5.43/sqrt(sum(array(hkl)**2));
    th = arcsin(6.1993/(E*d)); x = pi*chi/180; sx = sin(x); cx = cos(x);
    a = sx-(s23*sx+s34*cx)/s33; g = cos(x+th)*cos(x-th);
    return(-E*T*(sx+g*a)/(D*sin(th)),th*180/pi);

def teff(mu,T,th,chi):
    ''' FIXME: docstring '''
    cos_plus = abs(cos(pi*(th+chi)/180)); cos_minus = abs(cos(pi*(th-chi)/180));
    s = ones(chi.shape); p = ones(chi.shape);
    j = nonzero(abs(chi) < 90-abs(th)); s[j] = -1; p[j] = exp(-mu*T/cos_minus[j])
    te = p*(1-exp(-mu*T*(1/cos_plus+s/cos_minus)))/(mu*(1+cos_plus*s/cos_minus));
    i = nonzero(abs(cos_plus+s*cos_minus)<1.0e-10); te[i] = T*p[i]/cos_plus[i];
    return(te*cos_plus)

def rint(hkl,ver,chi,E,T,D,mu):
    ''' FIXME: docstring '''
    w,th = bw(hkl,ver,chi,E,T,D);
    t = teff(mu,T,th,chi);
    return(abs(w)*t/T);
```

Listing C.11: xrdtoolkit/crossection.py

```python
from common import Constants, XrayTable
import numpy as np

def get_cross_section(E,z):
    ''' Calculate cross section of element with atomic number z,
        Energy is in units of keV'''
    if z < 1 or z > 92:
        raise Exception('Z is out of range')
    if not isinstance(E,np.ndarray):
        E = np.array(E)
    logE = np.log(E)
    B = XrayTable[z]['Edge']
    A = np.array([ logE**i for i in xrange(0,4)])
    cross_section = np.exp(np.dot(XrayTable[z]['Absorption'],A))
    Q = np.array([ B[i] <= E for i in xrange(0,5) ] + [ np.ones(E.shape,dtype=boo
    Q[1:6] *= 1-Q[0:5]
    cross_section[0:6] = XrayTable[z]['JumpMatrix'].dot(cross_section[0:6]
    return np.array([sum(cross_section), XrayTable[z]['Density']*np.ones(E

def klein_nishina(e0, two_theta, polarisation=0):
    ''' Returns the klein-nishina crossection in barns and final photon en
    given energy in keV, 2theta in degrees and the linear stokes polarisat
    '''
    if not isinstance(e0,np.ndarray):
        e0 = np.array(e0)
    ct = np.cos(np.deg2rad(two_theta))
    r2 = (Constants.re*1e-8)**2 * 1e24
    k  = 1/(1+e0*(1-ct)/Constants.me)
    s  = r2*k**2 * (1/k+k-(1-polarisation)*(1-ct**2))/2
    return (s,e0*k)

def thomson(two_theta, polarisation=0):
    ''' Returns thomson crossection in barns given
        2theta in degrees and the linear stokes polarisation. '''
    ct = np.cos(np.deg2rad(two_theta))
    r2 = (Constants.re*1e-8)**2 * 1e24
    return (r2/2)*(2-(1-ct**2)*(1-polarisation))
```

Appendix D

The xrdtoolkit script files

Listing D.1: scripts/xrdtoolkit-assemble

```
#!/usr/bin/env python

import numpy as np
import os, optparse, itertools, locale
import h5py, fabio

import xrdtoolkit
from xrdtoolkit import f2w, files, utils, fit

class Assembler(utils.Script):
    def __init__(self, **kwargs):
        super(Assembler, self).__init__()
        self.usage = "xrdtoolkit-assemble <options> diffractogram.h5"
        self.description = """
        """
        self.peak = {}
        self.peak.update(kwargs)

    def parser_setup(self):
        super(Assembler, self).parser_setup()

        input_group = optparse.OptionGroup(self.parser, "Input options")
        input_group.add_option("--position", dest="peak_position",
                               help="Peak channel position")
        input_group.add_option("--fwhm", dest="peak_fwhm",
                               help="Peak FWHM full width at half maximum")
        input_group.add_option("--fit-width", dest="fit_width",
                          help="Number of FWHM to include in peak fitting.
        input_group.add_option("--shape", dest="peak_shape",
                          help="Peak shape. [gaussian|delta]", default='ga
        input_group.add_option("--peak-file", dest="peak_file",
                          help="Exported peaks from DAWN.", metavar="FILE"
        input_group.add_option("--input-set", dest="input_set",
                          help='Override data set [default %s].' % (xrdtoo
                          metavar="STRING", default=xrdtoolkit.DIFFRACTOGR
        self.parser.add_option("--flip", dest="flip",
                          action="store_true", default=False,
                          help="Flip every other scan line.")


        output_group = optparse.OptionGroup(self.parser, "Output options")
        output_group.add_option("-o", "--out", dest="outfile",
                          help="Output file.", metavar="FILE", default="si

        self.parser.add_option_group(input_group)
        self.parser.add_option_group(output_group)

    @utils.Script.timed
    def parse(self):
        super(Assembler, self).parse()

        self.do_flip = self.options.flip

        stacks = len(self.args)
        if stacks == 0 or not os.path.exists(self.args[0]):
            self.parser.error("Please specify diffractograms")

        if not all(os.path.exists(o) for o in self.args):
            self.parser.error("Could not read all files")

        self.print_verbose('Loading slice %s' % 0)
        diffractogram = files.ImageFile(self.args[0]).getImage(self.option
        darkcurrent_profile = files.ImageFile(self.args[0]).getImage(xrdtc
        self.input_data = np.zeros((stacks,)+diffractogram.shape)
        self.darkcurrent_profile = np.zeros((stacks,)+darkcurrent_profile.
        self.input_data[0] = diffractogram
        self.darkcurrent_profile[0] = darkcurrent_profile

        diff_seq = files.ImageSequence(self.args,data_set=self.options.inp
        dark_seq = files.ImageSequence(self.args,data_set=xrdtoolkit.DARKC
        for i in xrange(1,stacks):
            self.print_verbose('Loading slice %s' % i)
            self.input_data[i] = next(diff_seq)
            self.darkcurrent_profile[i] = next(dark_seq)

        self.output_shape = self.input_data.shape[:-1]

        self.peaks = [{
                'fwhm'       : self.options.peak_fwhm,
                'position'   : self.options.peak_position,
                'shape'      : self.options.peak_shape,
                'fit_width'  : self.options.fit_width
        }]

        self.peaks[0] = utils.strip_none_values(self.peak)
        self.peaks[0] = utils.convert(self.peak, float)

        if self.options.peak_file:
            # Replace peak list with fits
            # from dat file exported in DAWN
            self.peaks = []
            with open(self.options.peak_file, 'rb') as f:
                try:
                    data = f.readlines()
                    assert(len(data)>1)
                    header = data[0]
                    assert('#' in header)
                    data = data[1:]
                    header = header[1:].strip().lower().split()
                    for peak_line in data:
                        peak_line = peak_line.split('#')[0]
                        # Convert floats with locale-aware atof
                        peak_line = map(locale.atof, peak_line.split())
                        peak = dict(zip(header, peak_line))
                        if 'shape' not in peak:
                            peak['shape'] = fit.GAUSSIAN
                        peak = utils.strip_none_values(peak)
                        if 'fit_width' in peak:
                            print peak['fit_width']
                        if 'position' in peak and 'fwhm' in peak:
                            self.peaks.append(peak)
                except:
                    self.parser.error("Could not parse peak file %s" % (self.opti


        self.out_file = h5py.File(self.options.outfile)
        self.sinogram_group = self.out_file.require_group(xrdtoolkit.SINOG
        self.peak_group = self.out_file.require_group(xrdtoolkit.SINOGRAM_


    @utils.Script.timed
    def assemble_sinograms(self):
        for peak in self.peaks:
            self.peak_name = '%s (%s,%s)' % (peak['shape'],peak['position
            if self.peak_name in self.sinogram_group:
                self.print_verbose(self.peak_name, " already assembled")
                continue

            self.sinogram = self.sinogram_group.require_dataset(
                                 name=self.peak_name,
                                 shape=tuple(self.output_shape),
                                 dtype="float32"
            )
            self.print_verbose("Assembling ", self.peak_name)
            self.fit_peak(peak)

    def fit_peak(self, peak):
        """ Do weighted least squares fitting of peak shape
        to linear background using data in interval of four FWHM. """
        if 'fit_width' in peak:
            fit_width = peak['fit_width'] / 2.0
        else:
            fit_width = 2
        r_min = int(np.round(peak['position'] - fit_width*peak['fwhm']))
        r_max = int(np.round(peak['position'] + fit_width*peak['fwhm']))
        r = np.arange(r_min,r_max)

        peak_fun = fit.get_peak_function(**peak)

        sinogram_peaks = self.peak_group.require_dataset(
                             name=self.peak_name,
                             shape=tuple(self.output_shape) + (2,r.size,),
                             dtype="float32"
        )

        A = np.array([np.ones(r.size), r, peak_fun(r)])
        darkcurrent = self.darkcurrent_profile[..., r_min:r_max]
        if len(self.darkcurrent_profile.shape) > 2:
            if darkcurrent.shape[1] != self.input_data.shape[-2]:
                self.parser.error("Darkcurrent frames does not match the number o
            nframes = darkcurrent.shape[1]
        else:
            darkcurrent = self.darkcurrent_profile[...,r_min:r_max]
            darkcurrent.shape = (1,1,) + darkcurrent.shape
            nframes = 1

        ''' Fit peaks for sinogram. Last dimension is assumed
            to be the radial profile '''
        # Use temporary numpy array for assembly since we need reverse ::-1
        # indexing for flipping, which is not available with h5py
        # There would also be a lot of overhead since h5py writes on update.
        sinogram = np.zeros(self.sinogram.shape)
        for key in itertools.product(*map(xrange, self.input_data.shape[:-1])):
            # Key iterates over all indices in stack of sinograms
            signal = self.input_data[key][r_min:r_max]
            # Index dark profile. Stack number and frame number.
            dark = darkcurrent[key[0],key[-1] % nframes,...]
            # Use signal+dark as squared error. dy ~ sqrt(y) in poisson statistic
            w = 1.0 / (signal+dark)
            B = A * np.array([w,w,w])
            covC = np.linalg.inv(np.dot(B,A.T))
            c = np.dot(np.dot(covC, B), signal)
            sinogram[key] = c[2]

            sinogram_peaks[key+(0,)] = signal
            sinogram_peaks[key+(1,)] = np.dot(A.T,c)

        if self.do_flip:
            rev = sinogram[...,1::2,::-1].copy()
            sinogram[...,1::2,:] = rev
        self.sinogram[:] = sinogram


    @utils.Script.timed
    def output(self):
        self.out_file.close()

if __name__ == '__main__':
    locale.setlocale(locale.LC_NUMERIC, '')

    assmbl = Assembler()

    assmbl.parser_setup()
    assmbl.parse()
    assmbl.assemble_sinograms()
    assmbl.output()
else:
    pass
    # Initialize assmbl with values from DAWN
    # assmbl = Assembler(position=.., fwhm = ..., shape="gaussian")
```

Listing D.2: scripts/xrdtoolkit-average

```
#!/usr/bin/env python

import os, optparse
import numpy as np

import xrdtoolkit
from xrdtoolkit import f2w, files, utils

class Averager(utils.Script):
    def __init__(self):
        super(Averager, self).__init__()
        description = """
        Merge diffraction images and write average to file.
        """
    def parser_setup(self):
        super(Averager, self).parser_setup()

        output_group = optparse.OptionGroup(self.parser, "Output options")
        output_group.add_option("-o", "--out", dest="outfile",
                            help="File to save averaged dataset", metavar="F
        output_group.add_option("--data-set", dest="data_set",
                            help="Location to save data set (hdf5 etc).", m
        self.parser.add_option_group(output_group)
        self.parser.add_option("--method",
                            dest="method", default="median",
                            metavar="[median][mean]", help="Choose between a
        self.parser.add_option("--flatten", dest="flatten",
                            action="store_true", default=False,
                            help="Average over frames in multiframe files.")

    def parse(self):
        super(Averager, self).parse()

        if len(self.args) == 0:
            self.parser.error("Please provide some images to average.")

        if len(self.options.data_set.split('/')) < 2:
            self.parser.error("Dataset should be on the form '/group_name/data_se

    @utils.Script.timed
    def average(self):
        self.image = files.averageImages(self.args,method=self.options.method, fl

    @utils.Script.timed
    def output(self):
        out = files.ImageFile(self.options.outfile)
        out.saveImage(self.image, self.options.data_set)


if __name__ == '__main__':
    avg = Averager()

    avg.parser_setup()
    avg.parse()
    avg.average()
    avg.output()
    avg.print_timings()
```

Listing D.3: scripts/xrdtoolkit-calibrate

```
#!/usr/bin/env python

import numpy as np
import os, optparse

import xrdtoolkit
from xrdtoolkit import f2w, files, utils


try:
    import pyFAI
except ImportError:
    pyFAI = None

class Calibration(utils.Script):
    def __init__(self):
        super(Calibration, self).__init__()
        self.usage = 'Usage: %prog <options> CALIBRATION_IMAGE [DARK_CURRENT1,DAR
        self.description = """
        XRD calibration routine based on ring shape.
        """

    def parser_setup(self):
        super(Calibration, self).parser_setup()

        self.parser.add_option("--pixels", dest="pixels",
                          action="store_true", default=False,
                          help="Origin and calibration limits in pixels in
        file_group = optparse.OptionGroup(self.parser, "File options")
        file_group.add_option("-o", "--out", dest="outfile",
                      help="Save calibration image after subtracting darkc
        file_group.add_option("--data-set", dest="data_set",
                     help="Location to save data set.", metavar="STRING",
        file_group.add_option("-p", "--poni", dest="poni_file",
                      help="File to save detector geometry.", metavar="FIL

        detector_group = optparse.OptionGroup(self.parser, "Detector optio
        detector_group.add_option("-D", "--detector", dest="detector_name"
                      help="Detector name", default=None)
        detector_group.add_option("--distance", dest="detector_distance",
                      help="Detector distance from sample", metavar="dista
        detector_group.add_option("--binning", dest="detector_binning", na
                      help="Number of pixels that detector is set to group
                      metavar="x y",default=None)
        detector_group.add_option ('--tilt', dest="detector_tilt", nargs=
                      help="",
                      metavar="a b [degrees]", default=None)
        detector_group.add_option ('--origin', dest="detector_origin", nar
                      help="Initial detector origin wrt beam",
                      metavar="x y [mm]", default=None)

        calibration_group = optparse.OptionGroup(self.parser, "Calibration
        calibration_group.add_option ('--limits', dest="limits", nargs=2,
                      metavar="lower upper", help="Radial distance [mm] to

        self.parser.add_option_group(file_group)
        self.parser.add_option_group(detector_group)
        self.parser.add_option_group(calibration_group)

    def parse(self):
        super(Calibration, self).parse()

        if len(self.args) == 0:
            self.parser.print_help()
            self.parser.exit()

        # Set up self.detector
        DETECTOR_KWARGS = {
                'distance': self.options.detector_distance,
                'binning' : self.options.detector_binning,
                'origin'  : self.options.detector_origin,
                'tilt'    : self.options.detector_tilt,
        }

        if self.options.detector_name is not None:
            DETECTOR_KWARGS = utils.strip_none_values(DETECTOR_KWARGS)
            DETECTOR_KWARGS = utils.flatten(DETECTOR_KWARGS)
            DETECTOR_KWARGS = utils.convert(DETECTOR_KWARGS, float)
            self.detector = f2w.get_detector(self.options.detector_name, **DETECT

            if self.options.pixels and 'origin' in DETECTOR_KWARGS:
                self.detector.setorigin(list(np.multiply(self.detector._origin,se

        else:
            self.parser.error("self.detector missing")


        if len(self.args) > 1:
            self.dark_current = files.ImageFile(self.args[0]).getImage()
            self.calibration_image = files.ImageFile(self.args[1]).getImage()
        else:
            self.calibration_image = files.ImageFile(self.args[0]).getImage()
            self.dark_current = None

        if self.options.outfile:
            self.print_verbose('---> Saving %s' % (self.options.outfile,))
            f = files.ImageFile(self.options.outfile)
            cal = self.calibration_image.astype('int')-self.dark_current
            cal[cal<0] = 0  # Usually calibration image is uin16, so is prone to
            f.saveImage(cal.astype(self.calibration_image.dtype), self.options.d

        self.calibrator = f2w.Calibrator(self.calibration_image, self.dark_curren

        self.CALIBRATION_KWARGS = {
                'limits' : self.options.limits
        }
        self.CALIBRATION_KWARGS = utils.strip_none_values(self.CALIBRATION
        self.CALIBRATION_KWARGS = utils.flatten(self.CALIBRATION_KWARGS)
        self.CALIBRATION_KWARGS = utils.convert(self.CALIBRATION_KWARGS, fl
        if self.options.pixels and 'limits' in self.CALIBRATION_KWARGS:
            self.CALIBRATION_KWARGS['limits'] = list(np.multiply(self.CALI

    @utils.Script.timed
    def calibrate(self):
        self.print_verbose("---> Calibrating")
        self.calibrator.calibrate(**self.CALIBRATION_KWARGS)
        self.print_verbose(self.calibrator)

        if pyFAI is None and self.options.poni_file:
            self.parser.error("pyFAI needed for PONI file")
        elif pyFAI is not None and self.options.poni_file:
            from pyFAI import geometry
            g = geometry.Geometry()

            # converto to fit2D tilt plane notation from projected tilt an
            alpha = self.detector._tilt[0] * np.pi / 180
            beta  = self.detector._tilt[1] * np.pi / 180
            tiltPlanRotation = np.arctan(alpha/beta)
            tilt = alpha * np.sin(tiltPlanRotation) + beta * np.cos(tiltPla

            # Convert to degrees used by fit2D
            tiltPlanRotation = tiltPlanRotation * 180.0 / np.pi
            tilt = tilt * 180.0 / np.pi

            self.print_verbose("tilt ", tilt)
            self.print_verbose("tiltPlanRotation ", tiltPlanRotation)
            g.setFit2D(self.detector._distance,
                       self.detector._origin[1]/self.detector._pixelsize[1
                       self.detector._origin[0]/self.detector._pixelsize[0
                       tiltPlanRotation=tiltPlanRotation,
  # deg -> rad
                       tilt=tilt,    # deg -> rad
                       pixelX=self.detector._pixelsize[0]*1000.0,
  # mm -> um
                       pixelY=self.detector._pixelsize[1]*1000.0)
  # mm -> um
            self.print_verbose("---> Writing geometry to", self.options.po
            g.save(self.options.poni_file)

    def output(self):
        if self.options.outfile is None:
            return
        if self.dark_current is not None:
            calibration_profile = self.detector.integrate(self.calibration_image
        else:
            calibration_profile = self.detector.integrate(self.calibration_image)
        files.ImageFile(self.options.outfile).saveImage(calibration_profile[1], x

if __name__ == "__main__":
    cal = Calibration()
    cal.parser_setup()
    cal.parse()
    cal.calibrate()
    cal.output()
    cal.print_timings()
```

Listing D.4: scripts/xrdtoolkit-integrate

```
#!/usr/bin/env python

import numpy as np
import os, sys, optparse
import itertools, threading, Queue
import h5py, fabio

import xrdtoolkit
from xrdtoolkit import f2w, files, utils

try:
    import pyFAI
except ImportError:
    pyFAI = None

class Integrator(utils.Script):
    def __init__(self):
        super(Integrator, self).__init__()
        self.description = """
        Integrate diffraction images and assemble into dataset.
        Experiment parameters are assumed to be separated by underscore

        E.g.  NAME_xxx_yyy_zzz.edf

        will produce a data set with powder profiles in dimensions x,y,z.
        """
        self.usage="Usage: <options> --data-prefix=/mnt/data/../EXPERIMENT

        self.disable_threads    = False
        self.diable_fast_edf    = False
        self.disable_gpu        = False

        self.integration_points = 1500

    def parser_setup(self):
        super(Integrator, self).parser_setup()

        input_group = optparse.OptionGroup(self.parser, "Input options")
        input_group.add_option("--data-path", dest="data_path",
                               metavar="/path/IMAGE_xyz_",
                               help="Integrate files starting with this path.")
        input_group.add_option("--dark-path", dest="dark_path",
                               metavar="/path/DARK_xyz_", default=None,
                               help="Use darkcurrent files starting with this p
        input_group.add_option("--dark", dest="dark", default=None,
                               help="Darkcurrent image.")
        input_group.add_option("-p", "--ponit", dest="poni_file",
                               help="Name of poni file with detector geometry."

        output_group = optparse.OptionGroup(self.parser, "Output options")
        output_group.add_option("-o", "--out", dest="outfile",
                               help="File to save integrated dataset.", metavar
        output_group.add_option("--data-set", dest="data_set",
                               help="Location to save data set.", metavar="STRI

        self.parser.add_option_group(input_group)
        self.parser.add_option_group(output_group)

        # TODO make sure reshaping the output file is done correctly
        self.parser.add_option("--disable-gpu",
                               action="store_true", dest="disable_gpu",
                               help="Disable GPU for integration.")
        self.parser.add_option("--disable-fast-edf",
                               action="store_true", dest="disable_fast_edf",
                               help="Disable fast reading of EDF data.")
        self.parser.add_option("--disable-threads",
                               action="store_true", dest="disable_threads",
                               help="Disable threaded loading of files.")
        self.parser.add_option("--nbuffer", dest="nbuffers", default=2,
                               help="Number of files to buffer when loading fil
        self.parser.add_option("--points", metavar="POINTS", dest="integration_po
                               help="Number of points to keep radially.")

    @utils.Script.timed
    def parse(self):
        super(Integrator, self).parse()

        if not self.options.data_path:
            self.parser.error("Please specify --data-path")

        (data_prefix, data_directory, self.data_names) = files.matchImageFiles(se
        self.files = [ os.path.join(data_directory, file_name) for file_name in s

        if len(self.data_names) == 0:
            self.parser.error('No data files found starting with %s at %s' % (dat

        if self.options.dark_path is not None:
            (dark_prefix, dark_directory, dark_names) = files.matchImageFiles(sel
            if len(dark_names) == 0:
                self.parser.error('No dark files found starting with %s at %s' %
            self.print_verbose("--> Averaging dark images")
            self.dark = files.averageImages([ os.path.join(dark_directory, o) for

        if self.options.dark and os.path.exists(self.options.dark):
            self.dark = files.ImageFile(self.options.dark).getImage(xrdtoolkit.AV


        if self.options.disable_gpu is not None:
            self.disable_gpu = self.options.disable_gpu
        if self.options.disable_threads is not None:
            self.disable_threads = self.options.disable_threads
        if self.options.disable_fast_edf is not None:
            self.disable_fast_edf = self.options.disable_fast_edf

        self.nbuffers = self.options.nbuffers

        if self.options.integration_points is not None:
            self.integration_points = self.options.integration_points

        if self.options.poni_file and os.path.exists(self.options.poni_file):
            self.integrator = pyFAI.load(self.options.poni_file)
        else:
            self.parser.error("Need poni file to set up Azimuthal integrator.")

        if not 'dark' in self.__dict__:
            self.parser.error("No darkcurrent provided")

        if len(self.options.data_set.split('/')) < 2:
            self.parser.error("Dataset should be on the form '/group_name/

        indices = [[int(parm) for parm in os.path.splitext(o)[0].split(da
                        if parm is not '']
                    for o in self.data_names]
        indicesT = np.array(indices).T

        min_indices = indicesT.argmin(axis=1)
        max_indices = indicesT.argmax(axis=1)

        parameter_interval = np.array([ [ indicesT[i][min_indices[i]], ind
        dimensions = [(i,j-i+1) for i,j in parameter_interval]

        self.nframes = files.ImageFile(self.files[0]).getNFrames()
        dimensions = dimensions + [(0,self.nframes)]
        dimensions = [(minVal,count) for minVal,count in dimensions
if count > 1]
        self.parameter_count = [count for minVal,count in dimensions ]

        if self.dark.ndim > 2:
            if self.dark.shape[0] != self.nframes:
                self.parser.error("Number of darkcurrent frames does not 
            self.darkprofile_shape = (self.dark.shape[0], self.integration
        else:
            self.darkprofile_shape = (self.integration_points,)

        self.dimensions = len(dimensions)
        self.print_verbose('Number of dimensions: ', self.dimensions)



    @utils.Script.timed
    def create_dataset(self):
        self.print_verbose("--> Creating dataset")

        self.print_verbose("Data set size:", tuple(self.parameter_count) +

        self.hd5 = h5py.File(self.options.outfile)
        group = self.hd5.require_group(os.path.dirname(self.options.data_s
        if self.options.data_set in group:
                group[self.options.data_set].resize(tuple(self.parameter_count) + (se
        self.diffractogram_hd5 = group.require_dataset(
                            name=os.path.basename(self.options.data_set),
                            shape=tuple(self.parameter_count) + (self.integration_
                            chunks=tuple(self.parameter_count) + (1,),
                            dtype="float32"
        )
        self.diffractogram = np.zeros(tuple(self.parameter_count) + (self.integra
        self.darkcurrent = group.require_dataset(
                            name=xrdtoolkit.DARKCURRENT_DATA_SET,
                            shape=self.darkprofile_shape,
                            dtype="float32"
        )
        self.two_theta = group.require_dataset(
                            name=xrdtoolkit.TWO_THETA_DATA_SET,
                            shape=(self.integration_points,),
                            dtype="float32"
        )

    @utils.Script.timed
    def integrate_darkcurrent(self):
        self.print_verbose("--> Integrating darkcurrent")
        if self.dark.ndim > 2:
            self.print_verbose("     multiple frames.")
            if self.dark.shape[0] != self.nframes:
                self.parser.error("Number of darkcurrent frames does not match th
            darkcurrent_profile = np.zeros((self.nframes,self.integration_points)
            (self.two_theta[:],darkcurrent_profile[0]) = self.integrate(self.dark
            for i in xrange(1,self.nframes):
                darkcurrent_profile[i] = self.integrate(self.dark[i,...])[1]
        else:
            (self.two_theta[:],darkcurrent_profile) = self.integrate(self.dark)
        # Save integrated darkcurrent to file for reference only
        self.darkcurrent[:] = darkcurrent_profile




    @utils.Script.timed
    def integrate_and_assemble(self):
        if self.disable_threads:
            i=0
            images = files.ImageSequence(self.files)
            for index in itertools.product(*map(xrange,self.parameter_count)):
                    self.integrate_single_file(i, index, next(images))
                    i=i+1
                return

        self.image_queue = Queue.Queue(self.nframes*self.nbuffers+1)
        self.finished = False
        self.abort    = False

        intg_thread = threading.Thread(target=intg.integrate_files)
        load_thread = threading.Thread(target=intg.load_files)

        load_thread.setDaemon(True)

        intg_thread.start()
        load_thread.start()

        while(not self.finished and not self.abort):
            try:
                intg_thread.join(2)
                self.print_verbose(self.image_queue.qsize(), " images in q
            except KeyboardInterrupt:
                self.abort = True

        # Wait for integration thread to finish in case of aborting.
        intg_thread.join()

    def load_files(self):
        for image in files.ImageSequence(self.files):
            self.image_queue.put(image,block=True)


    @utils.Script.timed
    def integrate_files(self):

        # load all dark-images in list

        i=0
        for index in itertools.product(*map(xrange,self.parameter_count)):
            if self.abort:
                return
            self.integrate_single_file(i, index, self.image_queue.get())
            self.image_queue.task_done()
            i += 1
        self.finished = True
```

```python

    def integrate(self, image, dark=None):
        if not self.disable_gpu:
            tth, I = self.integrator.integrate1d(
                        image,
                        self.integration_points,
                        dark=dark,
                        unit='2th_deg',
                        method="lut_ocl", # GPU method with Look-up-table
                        safe=False, # Faster. Disables some LUT validation checks
                        )
        else:
            tth, I = self.integrator.integrate1d(
                        image,
                        self.integration_points,
                        dark=dark,
                        unit='2th_deg',
                        method="lut", # CPU method with Look-up-table
                        safe=False, # Faster. Disables some LUT validation checks
                        )
        return (tth,I)

    def integrate_single_file(self, iteration, index, image):
        if iteration % self.nframes == 0:
            self.print_verbose(" ---> " ,self.data_names[iteration/self.nframes])
        self.print_verbose("Integrating ", index, indent=1)

        if self.nframes > 1:
            (tth,I) = self.integrate(image, self.dark[iteration % self.nframes])
        else:
            (tth,I) = self.integrate(image, self.dark)
        self.diffractogram[tuple(index)] = I

    @utils.Script.timed
    def output(self):
        self.diffractogram_hd5[:] = self.diffractogram
        self.hd5.close()

if __name__ == '__main__':
    intg = Integrator()

    intg.parser_setup()
    intg.parse()
    intg.create_dataset()
    intg.integrate_darkcurrent()
    intg.integrate_and_assemble()

    intg.output()
    intg.print_timings()
```

Listing D.5: scripts/xrdtoolkit-reconstruct

```
#!/usr/bin/env python

import numpy as np
import os, optparse
import h5py, fabio

from skimage import transform

import xrdtoolkit
from xrdtoolkit import files, utils, tomo

class Reconstructor(utils.Script):
    def __init__(self, **kwargs):
        super(Reconstructor, self).__init__()
        self.description = """
        Correct sinograms and perform reconstruction.
        By default the sinogram is split into odd and even rows
        which are correlated to correct for offset in interleaved
        scans, followed by a search for the center of rotation.
        """

    def parser_setup(self):
        super(Reconstructor, self).parser_setup()

        input_group = optparse.OptionGroup(self.parser, "Input options")
        input_group.add_option("--sinogram-group", dest="sinogram_group",
                            help="Group containing sinograms.", default=xrdt
        input_group.add_option("--iterations", dest="sart_iterations", def
                            help="Number of iterations to perform SART recon
        self.parser.add_option("--no-center-correction", dest="no_correct_
                            action="store_true", default=False,
                            help="Attempt to find center of rotation [defaul
        self.parser.add_option("--no-denterlacing", dest="no_correct_inter
                            action="store_true", default=False,
                            help="Perform deinterlacing [default]")
        self.parser.add_option("--no-bragg-spots", dest="no_bragg_spots",
                            action="store_true", default=False,
                            help="Attempt to find and remove bragg spots [default]"
        self.parser.add_option("--disable-corrections", dest="disable_corrections
                            action="store_true", default=False,
                            help="Do not perform any corrections prior to reconstru

        output_group = optparse.OptionGroup(self.parser, "Output options")
        #output_group.add_option("--sinogram-group", dest="sinogram_group",
        #                           help="Group containing sinograms.", default=xrdtoolkit

        self.parser.add_option_group(input_group)

    @utils.Script.timed
    def parse(self):
        super(Reconstructor, self).parse()

        self.sart_iterations = utils.convert(self.options.sart_iterations,int)

        self.correct_interlacing = not self.options.no_correct_interlacing
        self.correct_center      = not self.options.no_correct_center
        self.remove_bragg_spots  = not self.options.no_bragg_spots

        if self.options.disable_corrections:
            self.correct_interlacing = False
            self.correct_center = False
            self.remove_bragg_spots = False

        if len(self.args) > 1 or len(self.args) == 0:
            self.parser.error("Please specify one and only one sinogram file")

        try:
            self.sino_file = h5py.File(self.args[0])
        except:
            self.parser.error("Could not open sinogram")

        if not self.options.sinogram_group in self.sino_file:
            self.parser.error('File does not have group %s' % (self.options.sinogr

        self.sino_group = self.sino_file[self.options.sinogram_group]
        self.corrected_group = self.sino_file.require_group(xrdtoolkit.CORRECTED_
        self.reconstruction_group = self.sino_file.require_group(xrdtoolkit.RECON


    @utils.Script.timed
    def process_sinograms(self):
        for key,data_set in self.sino_group.items():
            self.print_verbose("——>  ", key)
            sinogram = data_set.value
            if len(data_set.shape) > 2:
                slices = data_set.shape[0]
            else:
                slices = 1
                sinogram.shape = (1,)+data_set.shape

            corrected_dataset = self.corrected_group.require_dataset(
                    name=key,
                    shape=sinogram.squeeze().shape,
                    dtype="float32"
            )
            ret = np.zeros(sinogram.shape)
            for i in xrange(0,slices):
                corrected_sinogram = sinogram[i]
                if self.correct_interlacing:
                    corrected_sinogram = tomo.sino_deinterlace(corrected_s
                if self.correct_center:
                    corrected_sinogram = tomo.sino_center(corrected_sinogr
                if self.remove_bragg_spots:
                    corrected_sinogram = tomo.sino_remove_bragg_spots(corr

                ret[i,...] = corrected_sinogram

            corrected_dataset[:] = ret.squeeze()


    @utils.Script.timed
    def reconstruct(self):
        for key,data_set in self.corrected_group.items():
            corrected_sinogram = data_set.value
            if len(data_set.shape) > 2:
                slices = data_set.shape[0]
            else:
                slices = 1
                corrected_sinogram.shape= (1,)+data_set.shape

            for i in xrange(0,slices):
                # skimage expects columns of projections so transpose
                sino = corrected_sinogram[i].astype('double').T

                if key in self.reconstruction_group:
                    del self.reconstruction_group[key]
                reconstruction_dataset = self.reconstruction_group.require_datase
                        name=key,
                        shape=(slices,)+tuple([sino.shape[0]]*2),
                        dtype="float32"
                )

                # Do a fast, filtered back projection reconstruction
                # as initial guess for the SART reconstruction procedure
                reconstruction = transform.iradon(
                        sino,
                        output_size = sino.shape[0]
                )
                try:
                    for it in xrange(0,self.sart_iterations):
                        reconstruction = transform.iradon_sart(
                                sino,
                                image=reconstruction
                        )
                except AttributeError:
                    self.print_verbose("This version of skimage does not support

                reconstruction_dataset[i] = reconstruction

    @utils.Script.timed
    def output(self):
        self.sino_file.close()

if __name__ == '__main__':
    reconst = Reconstructor()

    reconst.parser_setup()
    reconst.parse()
    reconst.process_sinograms()
    reconst.reconstruct()
    reconst.output()
else:
    pass
    # Initialize assmbl with values from DAWN
    # assmbl = Reconstructor(position=.., fwhm = ..., shape="gaussian")
```

Bibliography

[1] Data analysis workbench (dawn). an eclipse based workbench for doing scientific data analysis. `http://www.dawnsci.org/home`.

[2] Id15 - high energy scattering beamline. `http://www.esrf.eu/UsersAndScience/Experiments/StructMaterials/ID15`.

[3] Python for Scientific Computing. *Computing in Science*, 9(3):10 – 20, 2007.

[4] M. Álvarez Murga, P. Bleuet, and J.-L. Hodeau. Diffraction/scattering computed tomography for three-dimensional characterization of multi-phase crystalline and amorphous materialsThis article forms part of a special issue dedicated to advanced diffraction imaging methods of materials, which will be pu. *Journal of Applied Crystallography*, 45(6):1109–1124, November 2012.

[5] Michelle Álvarez Murga, Pierre Bleuet, Leonel Marques, Christophe Lepoittevin, Nathalie Boudet, Gaston Gabarino, Mohamed Mezouar, and Jean-Louis Hodeau. Microstructural mapping of C60 phase transformation into disordered graphite at high pressure, using X-ray diffraction microtomography. *Journal of Applied Crystallography*, 44(1):163–171, December 2011.

[6] J. Baruchel, J.Y. Buffiere, and E. Maire. *X-ray Tomography in Material Science*. Hermes Science, 2000.

[7] Pierre Bleuet, Eléonore Welcomme, Eric Dooryhée, Jean Susini, Jean-Louis Hodeau, and Philippe Walter. Probing the structure of heterogeneous diluted materials by diffraction tomography. *Nature materials*, 7(6):468–72, June 2008.

[8] Christopher K. Egan, Simon D.M. Jacques, Marco Di Michiel, Biao Cai, Mathijs W. Zandbergen, Peter D. Lee, Andrew M. Beale, and Robert J. Cernik. Non-invasive imaging of the crystalline structure within a human tooth. *Acta Biomaterialia*, 9(9):8337–8345, 2013.

[9] A P Hammersley. FIT2D: An Introduction and Overview. Internal

report, ESRF, 2008.

[10] Veijo Honkimaki. Various matlab procedures for x-ray calculations. Unpublished: Private communication.

[11] Jérôme Kieffer and Dimitrios Karkoulis. PyFAI, a versatile library for azimuthal regrouping. *Journal of Physics: Conference Series*, 425(20):202012, March 2013.

[12] U Kleuker, P Suortti, W Weyrich, and P Spanne. Feasibility study of x-ray diffraction computed tomography for medical imaging. *Physics in Medicine and Biology*, 43(10):2911–2923, October 1998.

[13] Wolfgang Ludwig, Sø eren Schmidt, Erik Mejdal Lauridsen, and Henning Friis Poulsen. X-ray diffraction contrast tomography: a novel technique for three-dimensional grain mapping of polycrystals. I. Direct beam case. *Journal of Applied Crystallography*, 41(2):302–309, March 2008.

[14] Matthew G. O'Brien, Simon D. M. Jacques, Marco Di Michiel, Paul Barnes, Bert M. Weckhuysen, and Andrew M. Beale. Active phase evolution in single Ni/Al2O3 methanation catalyst bodies studied in real time using combined μ-XRD-CT and μ-absorption-CT. *Chemical Science*, 3(2):509, January 2012.

[15] L Salvo, P Cloetens, E Maire, S Zabler, J.J Blandin, J.Y Buffière, W Ludwig, E Boller, D Bellet, and C Josserond. X-ray micro-tomography an attractive characterisation technique in materials science. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 200:273–286, 2003.

[16] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. NIH Image to ImageJ: 25 years of image analysis. *Nature Methods*, 9(7):671–675, June 2012.

[17] Weingärtner Tim and Rüdiger Dillmann. Split-and-Merge Segmentation using Octrees. pages 1–5.

[18] Marco Voltolini, Maria Chiara Dalconi, Gilberto Artioli, Matteo Parisatto, Luca Valentini, Vincenzo Russo, Anne Bonnin, and Remi Tucoulou. Understanding cement hydration at the microscale: new opportunities from 'pencil-beam' synchrotron X-ray diffraction tomography. *Journal of Applied Crystallography*, 46(1):142–152, January 2013.