XRUTILS Users Manual

Eugen Wintersberger   Dominik Kriegner

August 29, 2012

Contents

Chapter 1

Introduction

pass :-)

1.1 About this document

The purpose of this file is mainly to provide a starting point for new users and a repository of usage examples for xrutils. Feel free to add new things, change errors and expand the examples as you think it is useful.

Chapter 2

Installation

Installing `xrutils` is a two steps process

1. install required C libraries and Python modules
2. build and install the `xrutils` C library and Python module

All steps are described in detail in this chapter. The package can be installed on Linux, Mac OS X and Microsoft Windows, however it is only tested on Linux/Unix platforms. Due to the lack of an package manager the installation on MS Windows platforms is cumbersome (see notes below). Please inform one of the authors in case the installation fails!

2.1 Required third party software

To keep the coding effort as small as possible `xrutils` depends on a large number of third party libraries and Python modules.

The needed dependencies are:

GCC Gnu Compiler Collection or an compatible C compiler. On windows you most probably should use MinGW or CygWin.

HDF5 a versatile binary data format (library is implemented in C). Although the library is not called directly, it is needed by the pytables Python module (see below).

Python the scripting language in which most of `xrutils` code is written in.

Latex a typesetting system used to write the documentation of the package (optionally)

git a version control system used to keep track on the `xrayutilities` development. (only needed for development)

Additionally, the following Python modules are needed in order to make `xrutils` work as intended

Scons a pythonic autotools/make replacement used for building the C library.

Numpy a Python module providing numerical array objects

Scipy a Python module providing standard numerical routines, which is heavily using numpy arrays

Python-Tables a powerful Python interface to HDF5. On windows you need to install numexpr as PyTables depends on this package.

Matplotlib a Python module for high quality 1D and 2D plotting (optionally)

IPython although not a dependency of `xrutils` the IPython shell is perfectly suited for the interactive use of the `xrutils` python package.

After installing all required packages you can continue with installing and building the C library.

2.2 Building and installing the library and python package

`xrutils` uses the SCons build system to compile the C components of the system. You can build the library simply by typing

```
>scons
```

in the root directory of the source distribution. To build using debug flags (`-g -O0`) type

```
>scons debug=1
```

instead. After building, the library and python package are installed by

```
>scons install --prefix=<install path>
```

The library is installed in `<install path>/lib`. Installation of the Python module is done via the `distutils` package (called by scons automatically). The –prefix option sets the root directory for the installation. If it is omitted the libary is installed under /usr/lib/ on Unix systems.

The documentation can be built with

```
>scons doc
```

which creates this file under `doc/manual/`.

2.3 Setup of the Python package

You need to make your Python installation aware of where to look for the module. This is usually only needed when installing in non-standard `<install path>` locations. For this case append the installation directory to your `PYTHONPATH` environment variable by

```
>export PYTHONPATH=$PYTHONPATH:<local install path>/lib64/python2.7/site-packages
```

on a Unix/Linux terminal. Or, to make this configuration persistent append this line to your local `.bashrc` file in your home directory. On MS Windows you would like to create a environment variable in the system preferences under system in the advanced tab. Be sure to use the correct directory which might be similar to

```
<local install path>/Lib/site-packages
```

on Windows systems.

2.4 Notes for installing on Windows

Since there is no packages manager on Windows the packages need to be installed manual (including all the dependecies) or a pre-packed solution needs to be used. We strongly suggest to use the pyhton(x,y) python distribution, which includes already most of the needed dependencies for installing xrayutilities.

When using python(x,y) you only have to install scons in addition, all other dependencies are available as plugins to python(x,y) and are installed by default anyhow. The setup of the environment variables is also done by the python(x,y) installation. One can proceed with the installation of xrutils directly!

In case you want to do it the hard way install all of the following (versions in brackets indicate the tested set of versions by the author (2.3.2012)):

- MinGW (0.4α)
- Python (2.7.2)
- scons (2.1.0)
- numpy (1.6.1)
- scipy (0.10.1)
- numexpr (1.4.2) needed for pytables
- pytables (2.3.1)
- matplotlib (1.1.0)
- ipython (0.12)

It is suggested to add the MinGW binary directory, as well as the Python and Python-scripts directory to the Path environment variable as described above! Installation is done as described above. In the simplest case (installation to default directories) this is done by:

```
> scons install
```

Chapter 3

Basic concepts

This chapter describes two concepts of usage for the `xrutils` package, which help with planning and analyzing experiments.

First the flowchart of how angular coordinates of Bragg reflections are calculated is shown (Fig. 3.1). After that a sketch on how to analyze x-ray diffraction data with `xrutils` is shown in Fig. 3.2. More detailed descriptions of the steps described here given in the subsequent chapters.

3.1 Usage examples

3.1.1 spec-file from rotating anode

The example shows the use of various parts of the xrutils package. It reads a spec-file; saves the content into a HDF5 file; reads a particular scan and plots it as reciprocal space map.

```
import xrutils as xu
import numpy
import tables
import matplotlib.pyplot as plt

sample = "ko222"

cch = 501
chpdeg = 250.333
roi=[80,920]

#define substrate material + experimental class
InAs = xu.materials.InAs
expcub = xu.HXRD(InAs.Q(1,1,-2),InAs.Q(1,1,1))

#config psd and initialize a intensity normalizer
expcub.Ang2Q.init_linear('z+',cch,1024.,chpdeg=chpdeg,roi=roi)
RA_normalizer_psd =
    xu.IntensityNormalizer("MCA",time="Seconds",absfun=lambda d:
    d["PSDCORR"]/d["PSD"].astype(numpy.float))

# print theoretic angular positions
ang111 = expcub.Q2Ang(InAs.Q(1,1,1))
ang331 = expcub.Q2Ang(InAs.Q(3,3,1))
ang224 = expcub.Q2Ang(InAs.Q(2,2,4))
```

```
print(" hkl\t%10s\t%10s\t%10s" %("om","tt","phi"))
print("------------------------------------------------------------")
print(" 111\t %10.3f\t%10.3f\t%10.3f"
    %(ang111[0],ang111[3],ang111[2]))
print(" 331\t %10.3f\t%10.3f\t%10.3f"
    %(ang331[0],ang331[3],ang331[2]))
print(" 224\t %10.3f\t%10.3f\t%10.3f"
    %(ang224[0],ang224[3],ang224[2]))

#read spec file and save to HDF5
try: s
except NameError: s = xu.io.SPECFile(sample+".spec")
else: s.Update()

h5file = "./data/"+sample+".h5"
try: h5.isopen
except NameError:
    h5 = tables.openFile(h5file,mode='a')
else:
    if not h5.isopen:
        h5 = tables.openFile(h5file,mode='a')
s.Save2HDF5(h5)

# read reciprocal space map from HDF5 file and plot it

uom = 12.7279 # aligned angular positions
utt = 25.4447
[thom,dummy,dummy,thtt] = expcub.Q2Ang(InAs.Q(1,1,1))

# parse map from file (omega scan around InAs(111) with PSD)
[mu,nu],sdata = xu.io.geth5_map(h5,[17],"Mu","Nu") # 17 is the
    scannumber

# normalize intensity and apply region of interests
PSDRAW = RA_normalizer_psd(MAP)
PSD = xu.blockAveragePSD(PSDRAW, 1, roi=roi)
# conversion to reciprocal space
[qx,qy,qz] = expcub.Ang2Q.linear(mu,nu,delta=[ uom-thom, utt-thtt])

# grid intensity for visualization
gridder = xu.Gridder2D(200,200)
gridder(qy,qz,PSD)

# restrict dynamic range of measurement and make log10
INT = xu.maplog(gridder.gdata.transpose(),6,0)

# plot the map
plt.figure(); plt.clf()
cf = plt.contourf(gridder.xaxis, gridder.yaxis, INT,50)
plt.xlabel(r'$Q_y$ ($\AA^{-1}$)'); plt.ylabel(r'$Q_z$
    ($\AA^{-1}$)')
#plt.subplots_adjust(bottom=0.12, top=0.92)
plt.colorbar(cf,ticks=numpy.arange(INT.min(),INT.max()+.1,
             round((INT.max()-INT.min())/10,1)),format="%.1f")
plt.figtext(0.76,0.93,r"$\log($Int$)$ (cps)")
plt.savefig(sample+"_111rsm.png",dpi=200)

#close HDF5-datafile
h5.close()
```

Listing 3.1: reading a spec-file, saving it to HDF5, reading a particular scan, convert it to reciprocal space, plot it using matplotlib

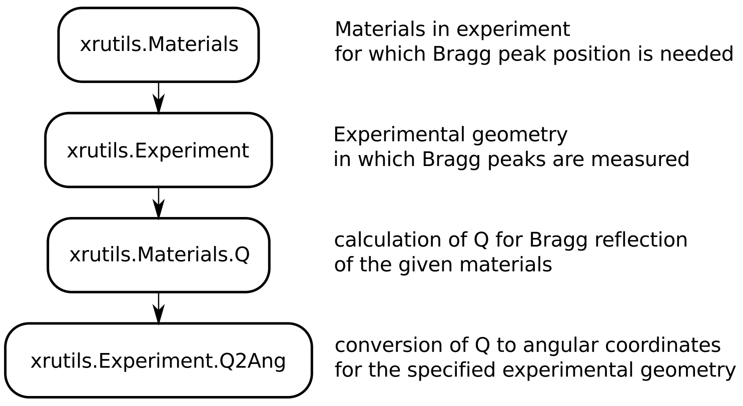
xrutils.Materials
Materials in experiment for which Bragg peak position is needed
xrutils.Experiment
Experimental geometry in which Bragg peaks are measured
xrutils.Materials.Q
calculation of Q for Bragg reflection of the given materials
xrutils.Experiment.Q2Ang
conversion of Q to angular coordinates for the specified experimental geometry

Figure 3.1: Flow diagram showing how to calculate angular positions of Bragg reflection using `xrutils`.

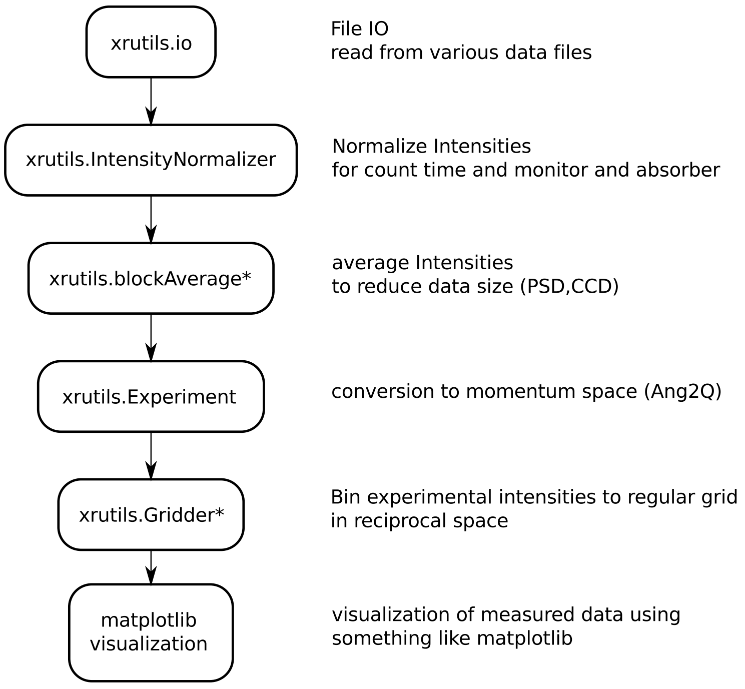
xrutils.io
File IO read from various data files
xrutils.IntensityNormalizer
Normalize Intensities for count time and monitor and absorber
xrutils.blockAverage*
average Intensities to reduce data size (PSD,CCD)
xrutils.Experiment
conversion to momentum space (Ang2Q)
xrutils.Gridder*
Bin experimental intensities to regular grid in reciprocal space
matplotlib visualization
visualization of measured data using something like matplotlib

Figure 3.2: Flow diagram showing how to analyze x-ray diffraction data using `xrutils`.

Chapter 4

The `materials` submodule

`xrutils` provides a set of classes to describe crystal lattices and materials.

Examples show how to define a new material by definings its lattice and deriving a new material, furthermore materials can be used to calculate the structure factor of a Bragg reflection for an specific energy or the energy dependency of its structure factor for anomalous scattering.

```
import xrutils as xu

# defining a ZincBlendeLattice with two types of atoms and lattice
    constant a
def ZincBlendeLattice(aa,ab,a):
    #create lattice base
    lb = xu.materials.LatticeBase()
    lb.append(aa,[0,0,0])
    lb.append(aa,[0.5,0.5,0])
    lb.append(aa,[0.5,0,0.5])
    lb.append(aa,[0,0.5,0.5])
    lb.append(ab,[0.25,0.25,0.25])
    lb.append(ab,[0.75,0.75,0.25])
    lb.append(ab,[0.75,0.25,0.75])
    lb.append(ab,[0.25,0.75,0.75])

    #create lattice vectors
    a1 = [a,0,0]
    a2 = [0,a,0]
    a3 = [0,0,a]

    l = xu.materials.Lattice(a1,a2,a3,base=lb)
    return l

# defining InP, no elastic properties are given,
# helper functions exist to create the (6,6) elastic tensor for
    cubic materials
InP  =
    xu.materials.Material("InP",ZincBlendeLattice(xu.elements.In,
    xu.elements.P,5.8687), numpy.zeros((6,6),dtype=numpy.double))
# InP is of course already included in the xu.materials module
```

Listing 4.1: defining a new material from scratch. This consists of an lattice with base and the type of atoms with elastic constantsof the material.

```
import xrutils as xu
import numpy

# defining material and experimental setup
InAs = xu.materials.InAs
energy= 8048 # eV

# calculate the structure factor for InAs (111) (222) (333)
hkllist = [[1,1,1],[2,2,2],[3,3,3]]
for hkl in hkllist:
    qvec = InAs.Q(hkl)
    F = InAs.lattice.StructureFactor(energy,qvec)
    print(" |F| = %8.3f" %numpy.abs(F))
```

Listing 4.2: calculation of the reflection strength of a Bragg reflection

```
import xrutils as xu
import numpy
import matplotlib.pyplot as plt

# defining material and experimental setup
InAs = xu.materials.InAs
energy= numpy.linspace(500,20000,5000) # 500 - 20000 eV

F = InAs.lattice.StructureFactorForEnergy(energy,InAs.Q(1,1,1))

plt.figure(); plt.clf()
plt.plot(energy,F.real,'k-',label='Re(F)')
plt.plot(energy,F.imag,'r-',label='Imag(F)')
plt.xlabel("Energy (eV)"); plt.ylabel("F"); plt.legend()
```

Listing 4.3: energy dependency of the structure factor

It is also possible to calculate the structure factor of atoms which may be needed for input into XRD simulations.

```
# f = f0(|Q|) + f1(en) + j * f2(en)
import xrutils as xu
import numpy

Fe = xu.materials.elements.Fe # iron atom
Q = numpy.array([0,0,1.9],dtype=numpy.double)
en = 10000 # energy in eV

print "Iron (Fe): E: %9.1f eV" % en
print "f0: %8.4g" % Fe.f0(numpy.linalg.norm(Q))
print "f1: %8.4g" % Fe.f1(en)
print "f2: %8.4g" % Fe.f2(en)
```

Listing 4.4: components of the structure factor for simulations

Chapter 5

The `Experiment` classes

The `Experiment` class and derived classes provide routines to help performing X-ray diffraction experiments. This includes methods to calculate the diffraction angles needed to align samples and to convert data between angular and reciprocal space. The conversion from angular to reciprocal space is implemented very general. Users should in normal cases only need the initialized routines in the Experiment classes.

5.1 Qconversion - general angular to momentum space conversion

Conversion of angular coordinates to reciprocal space can be tedious if one needs specialized code for several machines. This is an attempt in which it is tried to provide a more general solution to the problem of the conversion. Therefore the code is not only feeded with the angular coordinates but also with a description of the used goniometer geometry. The conversion for scans with an point detector and also routines for PSD (linear) and CCD (area) detectors are implemented.

5.2 mathematical background

For the routine to work for general goniometer setups a description of the goniometer type is needed. This is done in the following coordinate system:

description of the rotation axis

In general always look from arrow head of the axis towards the tail, then

- clockwise rotation means left-handed (negative),
- and anti-clockwise meand right-handed (positive)

i. e.: clockwise/left-handed rotation around the x-axis is decribed by the following rotation matrix:

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix} \tag{5.1}$$

The routines need to be supplied with the goniometer geometry. Therefore the goniometer axes are specified by their axis [xyz] and their sense of rotation [+-]. This description needs to be supplied for the sample and detector circles for the case where all axis are at zero position starting with the outermost circle.

For the Seifert XRD system this would mean: Sample circles: Omega + Chi + Phi -¿ ["x+","y+","z+"] Detector circles: Theta -¿ ["x+",]

5.2.1 calculation of the momentum transfer

The calculation the reciprocal space coordinates from angular positions is done as described in J. Appl. Cryst. (1999) **32** 614. Therefore we introduce the following nomenclatur:

$\boldsymbol{k}_{i,f}$	incidence and exiting wave vector of the x-ray radiation
$\boldsymbol{h}_c$	reciprocal space vector in carthesian coordinate system of the crystal
$\boldsymbol{h}_u$	reciprocal space vector in the coordinate system of the inner most circle of the goniometer
$\boldsymbol{h}$	reciprocal space vector in laboratory coordinate system
$\boldsymbol{Q}_L$	momentum transfer in laboratory coordinate system
$\underline{\underline{U}}$	orientation matrix of the crystal
$\underline{\underline{S}}$	rotation matrix for the sample goniometer
$\underline{\underline{D}}$	rotation matrix for the detector circles

To observe diffraction at position of $\boldsymbol{h}_c$ in reciprocal space the condition

$$\boldsymbol{h} = \boldsymbol{Q}_L \tag{5.2}$$

must be met. Where

$$\boldsymbol{h} = \underline{\underline{R}}\boldsymbol{h}_u = \underline{\underline{S}}\,\underline{\underline{U}}\boldsymbol{h}_c \tag{5.3}$$

and

$$\boldsymbol{Q}_L = \boldsymbol{k}_f - \boldsymbol{k}_i = \left(\underline{\underline{D}} - \underline{\underline{1}}\right)\boldsymbol{k}_i \tag{5.4}$$

The reciprocal space position $\boldsymbol{h}_c$ can than be calculated from Eq. 5.2 as

$$\boldsymbol{h}_c = \left(\underline{\underline{S}}\,\underline{\underline{U}}\right)^{-1}\left(\underline{\underline{D}} - \underline{\underline{1}}\right)\boldsymbol{k}_i \tag{5.5}$$

The rotation matrices $\underline{\underline{S}}$ and $\underline{\underline{D}}$ can be deduces from the description of the goniometer by multipliing the rotation matrices from each circle starting with the outer most.

$$\underline{\underline{D}}, \underline{\underline{S}} = (\text{outer most}) \cdot \ldots \cdot (\text{inner most}) \tag{5.6}$$

Note that all matrices are orthonormal, because they correspond to some basis transformation form one to another orthonormal coordinate system. Only the matrix $\underline{\underline{B}}$ which describes the transformation from the Miller indices $\boldsymbol{h}$ to the reciprocal space vector $\boldsymbol{h}_c$ must not be orthonormal (e. g. hexagonal crystals).

$$\boldsymbol{h}_c = \underline{\underline{B}}\boldsymbol{h} \tag{5.7}$$

5.3 implementation

The numerical expensive routines are coded in C because they seem to be a performance critical step in the analysis of XRD data. Because of the general form of the code some overhead must be excepted. Some wrapper functions are written in Python because of the ability of the fast coding.

general things

- all matrices are stored row-major (C standard) in linear array
- angles are passed as radians

rotation matrices

The used rotation matrices follow the rotation sense definition given earlier. In the following the rotation matrices for the positive rotation around the coordinate axes are given.

$$\underline{\underline{R_x}}(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \tag{5.8}$$

$$\underline{\underline{R_y}}(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix} \tag{5.9}$$

$$\underline{\underline{R_z}}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

For the

matrix inversion

The needed matrix inversion of a 3×3 matrix $\underline{\underline{A}}$ is done using the adjugate matrix formula

$$\underline{\underline{A}}^{-1} = \frac{\operatorname{adj} \underline{\underline{A}}}{\det \underline{\underline{A}}}\ , \tag{5.11}$$

which yields the formula

$$\underline{\underline{A}}^{-1} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} = \frac{1}{\det \underline{\underline{A}}} \begin{pmatrix} a_{11}a_{22} - a_{12}a_{21} & a_{02}a_{21} - a_{01}a_{22} & a_{01}a_{12} - a_{02}a_{11} \\ a_{12}a_{20} - a_{10}a_{22} & a_{00}a_{22} - a_{02}a_{20} & a_{02}a_{10} - a_{00}a_{12} \\ a_{10}a_{21} - a_{11}a_{20} & a_{01}a_{20} - a_{00}a_{21} & a_{00}a_{11} - a_{01}a_{10} \end{pmatrix} \tag{5.12}$$

with

$$\det \underline{\underline{A}} = a_{00}a_{11}a_{22} + a_{01}a_{12}a_{20} + a_{02}a_{10}a_{21} - a_{02}a_{11}a_{20} - a_{01}a_{10}a_{22} - a_{00}a_{12}a_{21} \; . \tag{5.13}$$

5.4 1D and 2D detectors

The C-code for linear and area detectors performs an exact conversion from real to momentum space by assuming the knowledge of the position and orientation of the detector in real space as well as its pixel distance. If the position and pixel size is not known it can be calculated from the channels per degree (N) which are mostly determined while recording measurements.

Therefore a distance of $d = 1$ (units are irrelevant here) is assumed and the corresponding pixel size is calclulated via

$$w_{\text{Pixel}} = \frac{2d}{N} \tan 0.5^\circ \tag{5.14}$$

The channels per degree of the detector are mostly determined symmetric with respect to the center channel. Therefore a factor 2 and angle 0.5° was introduced (see Fig. 5.2).

For the conversion to momentum space not only the distance d, pixel width w but also the detector direction is needed (Fig. 5.2). The coordinate system is assumed to be the one defined in Fig. 5.1.

The conversion is then done similar to Eq. 5.5. For the calculation of $\boldsymbol{k}_f$ the direction vector of each detector pixel $\hat{\boldsymbol{r}}_d$ must be used in the conversion routine, which leads to the following formulation:

$$\boldsymbol{h}_c = \left(\underline{\underline{SU}}\right)^{-1} \left(|\boldsymbol{k}_i| \underline{\underline{D}} \hat{\boldsymbol{r}}_d - \boldsymbol{k}_i\right) \tag{5.15}$$

5.5 HXRD experiments

Methods for high angle x-ray diffraction experiments. Mostly for experiments performed in coplanar scattering geometry. An example will be given for the calculation of the position of Bragg reflections.

```
import xrutils as xu

Si = xu.materials.Si  # load material from materials submodule

# initialize experimental class with directions from experiment
exp = xu.HXRD(Si.Q(1,1,-2),Si.Q(1,1,1))
```

```
# calculate angles and print them to the screen
angs = exp.Q2Ang(Si.Q(1,1,1))
print("Si (111)")
print("om: %8.3f" %angs[0])
print("tt: %8.3f" %angs[3])

angs = exp.Q2Ang(Si.Q(2,2,4))
print("Si (224)")
print("om: %8.3f" %angs[0])
print("tt: %8.3f" %angs[3])
```

Listing 5.1: calculation of angles for Si Bragg reflections

5.6 GID experiments

There are two implementations for GID experiments. Both describe 2S+2D diffractometers. One of them describes a setup as available in Linz at the rotating anode, the other one describes a setup as available at ID10B/ESRF. The difference is the mounting order of the detector circles.

```
import xrutils as xu

Si = xu.materials.Si  # load material from materials submodule

# initialize experimental class with directions from experiment
exp = xu.GID(Si.Q(1,-1,0),Si.Q(0,0,1))

# calculate angles and print them to the screen
(alphai,azimuth,tt,beta) = exp.Q2Ang(Si.Q(2,-2,0))
print("Si (2-20)")
print("azi: %8.3f" %azimuth)
print("tt : %8.3f" %tt)
```

Listing 5.2: calculation of angles for Si Bragg reflections in GID

5.7 Powder diffraction

The powder diffraction class is able to convert powder scans from angular to reciprocal space and furthermore powder scans of materials can be simulated in a very easy way.

```
import xrutils as xu

energy = 10000 # eV
powder = xu.Powder(xu.materials.Si,en=energy) # just give some
    material

# convert absolute q-space value to theta = 2theta/2
theta = powder.Q2Ang(2.0) # 2.0 A^{-1}
# and back
qpos = powder.Ang2Q(theta) # qpos = 2.0
```

Listing 5.3: conversion between angular and reciprocal space using the powder diffraction class

```
import xrutils as xu
import matplotlib.pyplot as plt

energy = (2*8048 + 8028)/3. # copper k alpha 1,2

# creating Indium powder
In_powder = xu.Powder(xu.materials.Indium,en=energy)
# calculating the reflection strength for the powder
In_powder.PowderIntensity()

# convoluting the peaks with a gaussian in q-space
peak_width = 0.01 # in q-space
resolution = 0.0005 # resolution in q-space
In_th,In_int = In_powder.Convolute(resolution,peak_width)

plt.figure(); plt.clf()
ax1 = plt.subplot(111)
plt.xlabel(r"2Theta (deg)"); plt.ylabel(r"Intensity")
# plot the convoluted signal
plt.plot(In_th*2,In_int/In_int.max(),'k-',label="Indium powder
    convolution")
# plot each peak in a bar plot
plt.bar(In_powder.ang*2, In_powder.data/In_powder.data.max(),
    width=0.3, bottom=0, linewidth=0, color='r',align='center',
    orientation='vertical',label="Indium powder bar plot")

plt.legend(); ax1.set_xlim(15,100); plt.grid()
```

Listing 5.4: simulation of an powder diffraction scan and two distinct ways of plotting the results

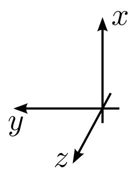

Figure 5.1: Used right handed coordinate system: x .. up; y .. left; z .. towards the viewer

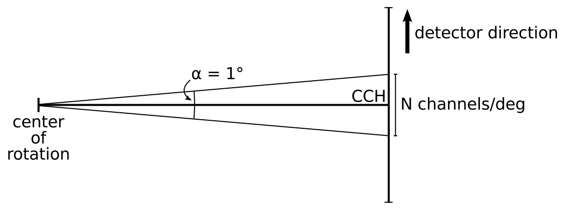

Figure 5.2: Illustration of a linear detector and channels per degree. (CCH: center channel)

Chapter 6

The io submodule

The `io` submodule provides classes for reading x-ray diffraction data in various formats.

6.1 Reading SPEC files

Working with spec files in xrutils can be done in two distinct ways.

1. parsing the spec file for scan headers; and parsing the data only when needed
2. parsing the spec file for scan headers; parsing all data and dump them to an HDF5 file; reading the data from the HDF5 file.

Both methods have their pros and cons. For example when you parse the spec-files over a network connection you need to re-read the data again over the network if using method 1) whereas you can dump them to a local file with method 2). But you will parse data of the complete file while dumping it to the HDF5 file.

Both methods work incremental, so they do not start at the beginning of the file when you reparse it, but start from the last position they were reading.

An working example for methode two is given in the following.

```
import tables
import xrutils as xu

# open spec file or use open SPECfile instance
try: s
except NameError:
    s = xu.io.SPECFile("sample_name.spec",path="./specdir")

# method (1)
scan10 = s[9] # Returns a SPECScan class
scan10.ReadData()
scan10data = scan10.data

# method (2)
h5file = "./h5dir/h5file.h5"
try: h5.isopen # open HDF5 file or use open one
except NameError:
    h5 = tables.openFile(h5file,mode='a')
```

```
else:
    if not h5.isopen: h5 = tables.openFile(h5file,mode='a')
s.Save2HDF5(h5) # save content of SPEC file to HDF5 file
# read data from HDF5 file
scan10data = h5.root.sample_name.scan_10.data.read()
```

Listing 6.1: parsing a SPEC file and read data (1) or dumping the data to an HDF5 file (2)

```
...

s.Update() # reparse for new scans in open SPECFile instance

# reread data method (1)
scan10 = s[9] # Returns a SPECScan class
scan10.ReadData()
scan10data = scan10.data

# reread data method (2)
s.Save2HDF5(h5) # save content of SPEC file to HDF5 file
# read data from HDF5 file
scan10data = h5.root.sample_name.scan_10.data.read()
```

Listing 6.2: reparse the SPEC file for new scans and reread the scans (1) or update the HDF5 file(2)

6.2 Reading EDF files

EDF files are mostly used to store CCD frames at ESRF. This format is therefore used in combination with SPEC files. In an example the EDFFile class is used to parse the data from EDF files and store them to an HDF5 file. HDF5 if perfectly suited because it can handle large amount of data and compression.

```
import tables
import xrutils as xu
import numpy

specfile = "specdir/mo3211b.dat"
h5file = "h5dir/mo3211b.h5"
h5 = tables.openFile(h5file,mode='a')

s = xu.io.SPECFile(specfile,path=specdir)
s.Save2HDF5(h5) # save to hdf5 file

# read ccd frames from EDF files
for i in range(1,1000,1):
    efile = "edfdir/sample_%04d.edf" %i
    e = xu.io.edf.EDFFile(efile,path=specdir)
    e.ReadData()
    g5 = h5.createGroup(h5.root,"frelon_%04d" %i)
    e.Save2HDF5(h5,group=g5)

h5.close()
```

Listing 6.3: script to parse and plot a reciprocal space map recorded with Seifert's XRD control software

6.3 Reading Bruker CCD data

6.4 Reading Radicon files

Usage of Radicon files is deprecated. They were used by an old control software of Radicon. There are functions to read this format!

6.5 Reading Seifert ASCII files

Two functions to read ASCII data from Seifert's XRD control software are available. One which reads single scans most times recorded with the point detector and the other one for parsing files which contain multiple scans (e.g.: RSMs recorded with the PSD).

```python
import xrutils as xu
import matplotlib.pyplot as plt
import numpy

exp111 = xu.HXRD([1,1,-2],[1,1,1])
#
data = xu.io.SeifertMultiScan('seifertmultiscan.nja','T','O')

om   = data.m2_pos[:,numpy.newaxis]*numpy.ones(data.int.shape)
tt   = data.sm_pos[numpy.newaxis,:]*numpy.ones(data.int.shape)

[qdummy,qy,qz] = exp111.Ang2Q(om,tt,0.)

gridder = xu.Gridder2D(300,300)
gridder(qy,qz,data.int)
INT = numpy.log10(gridder.gdata.transpose().clip(0.01,1e9))

plt.figure(); plt.clf()
plt.contourf(gridder.xaxis,gridder.yaxis,INT,50)
plt.xlabel(r'Qx'); plt.ylabel(r'Qz'); plt.colorbar()
```

Listing 6.4: script to parse and plot a reciprocal space map recorded with Seifert's XRD control software

```python
import xrutils as xu
import matplotlib.pyplot as plt

data = xu.io.SeifertScan('seifert.nja')

plt.figure(); plt.clf()
plt.plot(d.data[:,0],d.data[:,1],'k-')
plt.xlabel(r'scan axis (deg)'); plt.ylabel(r'Int (cps)');
    plt.grid()
```

Listing 6.5: script to parse and plot a single scan recorded with Seifert's XRD control software