xrayutilities Documentation

version 1.7

2024, Dominik Kriegner, Eugen Wintersberger

April 19, 2024

Contents

Welcome to xrayutilities's documentation!

If you look for downloading the package go to Sourceforge or GitHub (source distribution) or the Python package index (MS Windows binary).

Read more about *xrayutilities* below or in Journal of Applied Crystallography 2013, Volume 46, 1162-1170

Installation

The easiest way to install *xrayutilities* is using the *Python package index version <https://pypi.python.org/pypi/xrayutilities>* and execute

```
> pip install xrayutilities
```

If you prefer the installation from sources see the Source Installation below.

Introduction

Mailing list and issue tracker

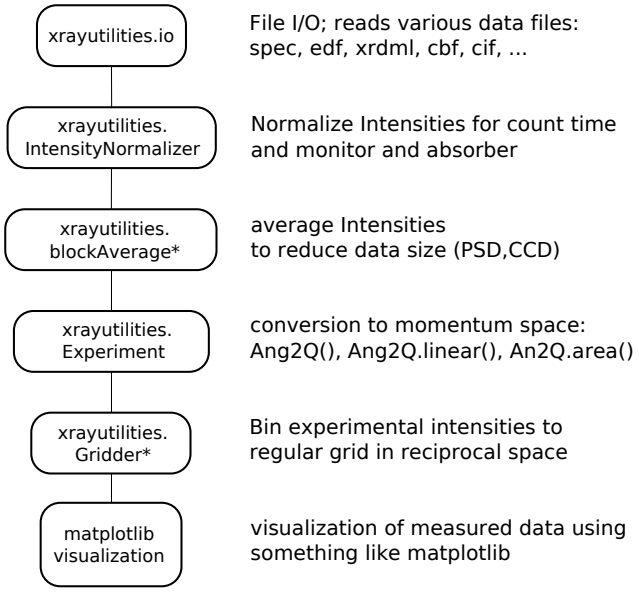
To get in touch with us or report an issue please use the mailing list or the Github issue tracker. When you want to follow announcements of major changes or new releases its recommended to sign up for the mailing list

Overview

xrayutilities is a collection of scripts used to analyze and simulate x-ray diffraction data. It consists of a python package and several routines coded in C. It especially useful for the reciprocal space conversion of diffraction data taken with linear and area detectors. Several models for the simulation of thin film reflectivity and diffraction curves are included. For details see the full API documentation of **`xrayutilities`** found here: Examples and API-documentation.

In the following few concepts of usage for the *xrayutilities* package will be described. First one should get a brief idea of how to analyze x-ray diffraction data with *xrayutilities*. Following that the concept of how angular coordinates of Bragg reflections are calculated is presented. Before describing in detail the installation a minimal example for thin film simulations is shown.

Concept of usage



xrayutilities provides a set of functions to read experimental data from various data file formats. All of them are gathered in the **`io`**-subpackage. After reading data with a function from the io-submodule the data might be corrected for monitor

counts and/or absorption factor of a beam attenuator. A special set of functions is provided to perform this for point, linear and area detectors.

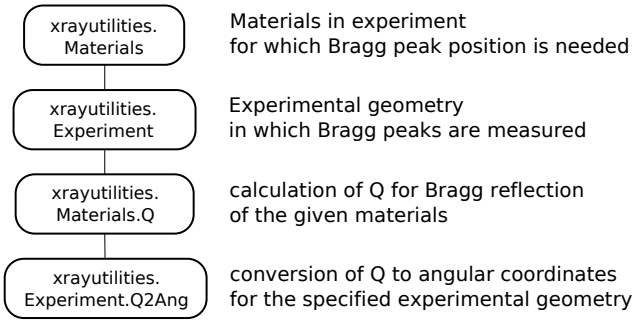
Since the amount of data taken with modern detectors often is too large to be able to work with them properly, a functions for reducing the data from linear and area detectors are provided. They use block-averaging to reduce the amount of data. Use those carefully not to loose the features you are interested in in your measurements.

After the pre-treatment of the data, the core part of the package is the transformation of the angular data to reciprocal space. This is done as described in more detail below using the **`experiment`**-module`. The classes provided within the experiment module provide routines to help performing X-ray diffraction experiments. This includes methods to calculate the diffraction angles (described below) needed to align crystalline samples and to convert data between angular and reciprocal space. The conversion from angular to reciprocal space is implemented very general for various goniometer geometries. It is especially useful in combination with linear and area detectors as described in this article. In standard cases, Users will only need the initialized routines, which predefine a certain goniometer geometry like the popular four-cirlce and six-circle geometries.

After the conversion to reciprocal space, it is convenient to transform the data to a regular grid for visualization. For this purpose the **`gridder`**-module has been included into *xrayutilities*. For the visualization of the data in reciprocal space the usage of matplotlib is recommended.

A practical example showing the usage is given below.

Angle calculation using the material classes



Calculation of angles needed to align Bragg reflections in various diffraction geometries is done using the Materials defined in the **`materials`**-package. This package provides a set of classes to describe crystal lattices and materials. Once such a material is properly defined one can calculate its properties, which includes the reciprocal lattice points, lattice plane distances, optical properties like the refractive index, the structure factor (including the atomic scattering factor) and the complex polarizability. These atomic properties are extracted from a database included in *xrayutilities*.

Using such a material and an experimental class from the **`experiment`**-module, describing the experimental setup, the needed diffraction angles can be calculated for certain coplanar diffraction (high, low incidence), grazing incidence diffraction and also special non-coplanar diffraction geometries. In the predefined experimental classes fixed geometries are used. For angle calculation of custom geometries using arbitrary geometries (max. of three free angles) the **`q2ang_fit`**-module can be used as described in one of the included example files.

hello world

A first example with step by step explanation is shown in the following. It showcases the use of *xrayutilities* to calculate angles and read a scan recorded with a linear detector from spec-file and plots the result as reciprocal space map using matplotlib.

```
"""
Example script to show how to use xrayutilities to read and plot
reciprocal space map scans from a spec file created at the ESRF/ID10B

for details about the measurement see:
    D Kriegner et al. Nanotechnology 22 425704 (2011)
    http://dx.doi.org/10.1088/0957-4484/22/42/425704
"""

import os
```

```
11
12 import matplotlib.pyplot as plt
13 import xrayutilities as xu
14
15 # global setting for the experiment
16 sample = "test"  # sample name used also as file name for the data file
17 energy = 8042.5  # x-ray energy in eV
18 center_ch = 715.9  # center channel of the linear detector
19 chpdeg = 345.28  # channels per degree of the linear detector
20 roi = [100, 1340]  # region of interest of the detector
21 nchannel = 1500  # number of channels of the detector
22 datapath = os.path.join("examples", "data")
23
24 # intensity normalizer function responsible for count time and absorber
25 # correction
26 normalizer_detcorr = xu.IntensityNormalizer(
27     "MCA",
28     mon="Monitor",
29     time="Seconds",
30     absfun=lambda d: d["detcorr"] / d["psd2"].astype(float))
31
32 # substrate material used for Bragg peak calculation to correct for
33 # experimental offsets
34 InP = xu.materials.InP
35
36 # initialize experimental class to specify the reference directions of your
37 # crystal
38 # 11-2: inplane reference
39 # 111: surface normal
40 hxrd = xu.HXRD(InP.Q(1, 1, -2), InP.Q(1, 1, 1), en=energy)
41
42 # configure linear detector
43 # detector direction + parameters need to be given
44 # mounted along z direction, which corresponds to twotheta
45 hxrd.Ang2Q.init_linear('z-', center_ch, nchannel, chpdeg=chpdeg, roi=roi)
46
47 # read spec file and save to HDF5-file
48 # since reading is much faster from HDF5 once the data are transformed
49 h5file = os.path.join(datapath, sample + ".h5")
50 try:
51     s  # try if spec file object already exist ("run -i" in ipython)
52 except NameError:
53     s = xu.io.SPECFile(sample + ".spec.bz2", path=datapath)
54 else:
55     s.Update()
56 s.Save2HDF5(h5file)
57
58 ###################################
59 # InP (333) reciprocal space map
60 omalign = 43.0529  # experimental aligned values
61 ttalign = 86.0733
62 [omnominal, _, _, ttnominal] = hxrd.Q2Ang(
63     InP.Q(3, 3, 3))  # nominal values of the substrate peak
64
65 # read the data from the HDF5 file
66 # scan number:36, names of motors in spec file: omega= sample rocking, gamma =
67 # twotheta
68 [om, tt], MAP = xu.io.geth5_scan(h5file, 36, 'omega', 'gamma')
69 # normalize the intensity values (absorber and count time corrections)
70 psdraw = normalizer_detcorr(MAP)
```

```
# remove unusable detector channels/regions (no averaging of detector channels)
psd = xu.blockAveragePSD(psdraw, 1, roi=roi)

# convert angular coordinates to reciprocal space + correct for offsets
[qx, qy, qz] = hxrd.Ang2Q.linear(
    om, tt,
    delta=[omalign - omnominal, ttalign - ttnominal])

# calculate data on a regular grid of 200x201 points
gridder = xu.Gridder2D(200, 201)
gridder(qy, qz, psd)
# maplog function limits the shown dynamic range to 8 orders of magnitude
# from the maxium
INT = xu.maplog(gridder.data.T, 8., 0)

# plot the intensity as contour plot using matplotlib
plt.figure()
cf = plt.contourf(gridder.xaxis, gridder.yaxis, INT, 100, extend='min')
plt.xlabel(r'$Q_{[11\bar2]}$ ($\mathrm{\AA}^{-1}$)')
plt.ylabel(r'$Q_{[\bar1\bar1\bar1]}$ ($\mathrm{\AA}^{-1}$)')
cb = plt.colorbar(cf)
cb.set_label(r"$\log($Int$)$ (cps)")
```

More such examples can be found on the Examples page.

X-ray diffraction and reflectivity simulations

xrayutilties includes a database with optical properties of materials and therefore simulation of reflectivity and diffraction data can be accomplished with relatively litte additional input. When the stack of layers is defined along with the layer thickness and material several models for calculation of X-ray reflectivity and dynamical/kinematical X-ray diffraction are provided.

A minimal example for an AlGaAs superlattice structure is shown below. It shows how a basic stack of a superlattice is built from its ingredients and how the reflectivity and dynamical diffraction model are initialized in the most basic form:

```
import xrayutilities as xu
# Build the pseudomorphic sample stack using the elastic parameters
sub = xu.simpack.Layer(xu.materials.GaAs, inf)
lay1 = xu.simpack.Layer(xu.materials.AlGaAs(0.25), 75, relaxation=0.0)
lay2 = xu.simpack.Layer(xu.materials.AlGaAs(0.75), 25, relaxation=0.0)
pls = xu.simpack.PseudomorphicStack001('pseudo', sub+10*(lay1+lay2))
# simulate reflectivity
m = xu.simpack.SpecularReflectivityModel(pls, sample_width=5, beam_width=0.3)
alphai = linspace(0, 10, 1000)
Ixrr = m.simulate(alphai)
# simulate dynamical diffraction curve
alphai = linspace(29, 38, 1000)
md = xu.simpack.DynamicalModel(pls)
Idyn = md.simulate(alphai, hkl=(0, 0, 4))
```

More detailed examples and description of model parameters can be found on the Simulation examples page or in the `examples` directory.

Source Installation

Express instructions

- install the dependencies (Windows: Python(x,y) or WinPython; Linux/Unix: see below for dependencies).

- download *xrayutilities* from here or use git to check out the latest version.
- open a command line and navigate to the downloaded sources and execute:

```
> pip install .
```

which will install *xrayutilities* to the default directory. It should be possible to use it (*import xrayutilities*) from now on in python scripts.

Detailed instructions

Installing *xrayutilities* is done using Python's setuptools

The package can be installed on Linux, Mac OS X and Microsoft Windows, however, it is mostly tested on Linux/Unix platforms. Please inform one of the authors in case the installation fails!

Required third party software

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

The needed runtime dependencies are:

- **Python** the scripting language in which most of *xrayutilities* code is written in. (>= 3.6, for Python 2.7 use *xrayutilities* 1.5.X or older)
- **Numpy** a Python module providing numerical array objects (version >= 1.9)
- **Scipy** a Python module providing standard numerical routines, which is heavily using numpy arrays (version >= 0.13.0)
- **h5py** a powerful Python interface to HDF5.
- **lmfit** a Python module for least-squares minimization with bounds and constraints (needed for fitting XRR/XRD data)

For several features optional dependencies are needed:

- **Matplotlib** a Python module for high quality 1D and 2D plotting (optional, version >= 3.1.0)
- **IPython** although not a dependency of *xrayutilities* the IPython shell is perfectly suited for the interactive use of the *xrayutilities* python package.
- **mayavi** only used optionally in Crystal.show_unitcell where it produces a superior visualization to otherwise used Matplotlib 3D plots

Additionally, the following Python modules are needed when building *xrayutilities* from source or wanting to test your installation:

- **C-compiler** Gnu Compiler Collection or any compatible C compiler. On windows you most probably want to use the Microsoft compilers.
- **Python development headers**
- **setuptools** build system
- **pytest** needed for running the pre-configured unittest environment, which in principal can also be achieved only by the unittest package (optional)

For building the documention (which you do not need to do) the requirements are:

- **sphinx** the Python documentation generator
- **numpydoc** sphinx-extension needed to parse the API-documention
- **rst2pdf** pdf-generation using sphinx
- **sphinx_rtd_theme** sphinx theme used
- **svglib** library needed by rst2pdf to include svg images into the pdf documentation

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

Building and installing the library and python package

Although the *setup.py* script can be called manually its recommended to always use pip to install *xrayutilities*, which can be done by executing

```
>pip install .
```

or

```
>pip install --prefix=INSTALLPATH .
```

in the root directory of the source distribution.

The *–prefix* option sets the root directory for the installation. If it is omitted the libary is installed under the systems default directories (recommended).

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 (Using Python package managers this should be done automatically). Be sure to use the correct directory which might be similar to

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

on Windows systems.

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 two be used. We strongly suggest to use either Anaconda, Python(x,y) or WinPython Python distributions, which include already all of the needed dependencies for installing *xrayutilities*.

One can proceed with the installation of *xrayutilities* directly! The easiest way to do this on windows is to use the binaries distributed on the Python package index or using *pip*, otherwise one can follow the general installation instructions. On Anaconda it can also be done using the conda-forge *xrayutilities* package.

Examples and API-documentation

Examples

In the following a few code-snippets are shown which should help you getting started with *xrayutilities*. Not all of the codes shown in the following will be run-able as stand-alone script. For fully running scripts look in the GitHub examples or in the download.

Reading data from data files

The **`io`** submodule provides classes for reading x-ray diffraction data in various formats. In the following few examples are given.

Reading SPEC files

Working with spec files in *xrayutilities* 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 reread it, but start from the last position they were reading and work with files including data from linear detectors.

An working example for both methods is given in the following.

```
import xrayutilities as xu
import os

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

# method (1)
s.scan10.ReadData()
scan10data = s.scan10.data

# method (2)
h5file = os.path.join("h5dir", "h5file.h5")
s.Save2HDF5(h5file) # save content of SPEC file to HDF5 file
# read data from HDF5 file
[angle1, angle2], scan10data = xu.io.geth5_scan(h5file, [10],
                                                "motorname1",
                                                "motorname2")
```

> Seealso
>
> the fully working example hello world

In the following it is shown how to re-parsing the SPEC file for new scans and reread the scans (1) or update the HDF5 file(2)

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

# reread data method (1)
s.scan10.ReadData()
scan10data = s.scan10.data

# reread data method (2)
s.Save2HDF5(h5) # save content of SPEC file to HDF5 file
# read data from HDF5 file
[angle1, angle2], scan10data = xu.io.geth5_scan(h5file, [10],
                                                "motorname1",
                                                "motorname2")
```

Reading EDF files

EDF files are mostly used to store CCD frames at ESRF recorded from various different detectors. 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 xrayutilities as xu
import numpy

```

```
specfile = "specfile.spec"
h5file = "h5file.h5"

s = xu.io.SPECFile(specfile)
s.Save2HDF5(h5file) # save to hdf5 file

# read ccd frames from EDF files
for i in range(1, 1001, 1):
    efile = "edfdir/sample_%04d.edf" % i
    e = xu.io.edf.EDFFile(efile)
    e.ReadData()
    e.Save2HDF5(h5file, group="/frelon_%04d" % i)
```

Seealso

the fully working example provided in the examples directory perfectly suited for reading data from beamline ID01

Reading XRDML files

Files recorded by Panalytical diffractometers in the `.xrdml` format can be parsed. All supported file formats can also be parsed transparently when they are saved as compressed files using common compression formats. The parsing of such compressed `.xrdml` files conversion to reciprocal space and visualization by gridding is shown below:

```
import xrayutilities as xu
om, tt, psd = xu.io.getxrdml_map('rsm_%d.xrdml.bz2', [1, 2, 3, 4, 5],
                                 path='data')
# or using the more flexible function
tt, om, psd = xu.io.getxrdml_scan('rsm_%d.xrdml.bz2', 'Omega',
                                  scannrs=[1, 2, 3, 4, 5], path='data')
# single files can also be opened directly using the low level approach
xf = xu.io.XRDMLFile('data/rsm_1.xrdml.bz2')
# then see xf.scan and its attributes
```

Seealso

the fully working example provided in the examples directory

Other formats

Other formats which can be read include

- Rigaku `.ras` files.
- files produces by the experimental control software at Hasylab/Desy (spectra).
- numor files from the ILL neutron facility
- ccd images in the tiff file format produced by RoperScientific CCD cameras and Perkin Elmer detectors.
- files from recorded by Seifert diffractometer control software (`.nja`)
- support is also provided for reading of `cif` files from structure databases to extract unit cell parameters as well es read data from those files (pdCIF, ESG files)

See the examples directory for more information and working example scripts.

Angle calculation using `Experiment` and `materials` classes

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 xrayutilities as xu
>>> Si = xu.materials.Si  # load material from materials submodule
>>>
>>> # initialize experimental class with directions from experiment
>>> hxrd = xu.HXRD(Si.Q(1, 1, -2), Si.Q(1, 1, 1))
>>> # calculate angles of Bragg reflections and print them to the screen
>>> om, chi, phi, tt = hxrd.Q2Ang(Si.Q(1, 1, 1))
>>> print("Si (111)")
Si (111)
>>> print(f"om, tt: {om:8.3f} {tt:8.3f}")
om, tt:   14.221   28.442
>>> om, chi, phi, tt = hxrd.Q2Ang(Si.Q(2, 2, 4))
>>> print("Si (224)")
Si (224)
>>> print(f"om, tt: {om:8.3f} {tt:8.3f}")
om, tt:   63.485   88.028
```

Note that above the **HXRD** class is initialized without specifying the energy used in the experiment. It will use the default energy stored in the configuration file, which defaults to CuK-α_1.

One could also call:

```
hxrd = xu.HXRD(Si.Q(1, 1, -2), Si.Q(1, 1, 1), en=10000)  # energy in eV
```

to specify the energy explicitly. The **HXRD** class by default describes a four-circle goniometer as described in more detail here.

Similar functions exist for other experimental geometries. For grazing incidence diffraction one might use

```
>>> import xrayutilities as xu
>>> gid = xu.GID(xu.materials.Si.Q(1, -1, 0), xu.materials.Si.Q(0, 0, 1))
>>> # calculate angles and print them to the screen
>>> (alphai, azimuth, tt, beta) = gid.Q2Ang(xu.materials.Si.Q(2, -2, 0))
>>> print(f"azimuth, tt: {azimuth:8.3f} {tt:8.3f}")
azimuth, tt:  113.651   47.302
```

There is on implementation of a GID 2S+2D diffractometer. Be sure to check if the order of the detector circles fits your goniometer, otherwise define one yourself!

There exists also a powder diffraction class, which is able to convert powder scans from angular to reciprocal space.

```
import xrayutilities as xu
import numpy
energy = 'CuKa12'
# creating powder experiment
xup = xu.PowderExperiment(en=energy)
theta = numpy.arange(0, 70, 0.01)
q = xup.Ang2Q(theta)
```

More information about powdered materials can be obtained from the **PowderDiffraction** class. It contains information about peak positions and intensities

```
>>> import xrayutilities as xu
>>> print(xu.simpack.PowderDiffraction(xu.materials.In))
Powder diffraction object
-------------------------
Powder-In (a: 3.2523, c: 4.9461, at0_In_2a_occupation: 1, at0_In_2a_biso: 0, volume: 1, )
Lattice:
139 tetragonal I4/mmm: a = 3.2523, b = 3.2523 c= 4.9461
alpha = 90.000, beta = 90.000, gamma = 90.000
Lattice base:
```

```
0: In (49) 2a  occ=1.000 b=0.000
Reflection conditions:
 general: hkl: h+k+l=2n, hk0: h+k=2n, 0kl: k+l=2n, hhl: l=2n, 00l: l=2n, h00: h=2n
2a      : None

Reflections:
------------
      h k l     |    tth    |    |Q|    |Int     |   Int (%)
   ------------------------------------------------------------------
      (1, 0, 1)    32.9339      2.312       217.24      100.00
      (0, 0, 2)    36.2964      2.541        41.69       19.19
      (1, 1, 0)    39.1392      2.732        67.54       31.09
     (1, 1, -2)    54.4383      3.731        50.58       23.28
      (2, 0, 0)    56.5486      3.864        22.47       10.34
     (1, 0, -3)    63.1775      4.273        31.82       14.65
      (2, 1, 1)    67.0127      4.503        53.09       24.44
      (2, 0, 2)    69.0720      4.624        24.22       11.15
      (0, 0, 4)    77.0641      5.081         4.43        2.04
      (2, 2, 0)    84.1193      5.464         7.13        3.28
     (2, 1, -3)    89.8592      5.761        24.97       11.49
      (1, 1, 4)    90.0301      5.769        12.44        5.73
      (3, 0, 1)    93.3390      5.933        11.75        5.41
     (2, -2, 2)    95.2543      6.026        11.43        5.26
      (3, 1, 0)    97.0033      6.109        11.19        5.15
     (2, 0, -4)   102.9976      6.384        10.69        4.92
      (3, 1, 2)   108.4189      6.617        21.19        9.75
      (1, 0, 5)   108.9602      6.639        10.60        4.88
      (3, 0, 3)   116.5074      6.936        11.01        5.07
     (2, -3, 1)   120.4651      7.081        22.87       10.53
     (2, 2, -4)   132.3519      7.462        13.70        6.31
      (0, 0, 6)   138.2722      7.622         3.88        1.79
     (2, 1, -5)   140.6857      7.681        32.94       15.16
      (4, 0, 0)   142.6631      7.728         8.67        3.99
     (3, 2, -3)   153.5192      7.940        48.97       22.54
      (3, 1, 4)   153.9051      7.946        49.71       22.88
      (4, 1, 1)   162.8984      8.066        76.17       35.07
      (1, 1, 6)   166.0961      8.097        46.91       21.60
      (4, 0, 2)   171.5398      8.135        77.25       35.56
```

If you are interested in simulations of powder diffraction patterns look at section Powder diffraction simulations

Using the `Gridder` classes

xrayutilities provides Gridder classes for 1D, 2D, and 3D data sets. These Gridders map irregular spaced data onto a regular grid. This is often needed after transforming data measured at equally spaced angular positions to reciprocal space were their spacing is irregular.

In 1D this process actually equals the calculation of a histogram. Below you find the most basic way of using the Gridder in 2D. Other dimensions work very similar.

The most easiest use (what most user might need) is

```python
import xrayutilities as xu # import Python package
g = xu.Gridder2D(100, 101) # initialize the Gridder object, which will
# perform Gridding to a regular grid with 100x101 points
#====== load some data here =====
g(x, y, data) # call the gridder with the data
griddata = g.data # the data attribute contains the gridded data.
```

A more complicated example showing also sequential gridding is shown below. You need sequential gridding when you can not load all data at the same time, which is often problematic with 3D data sets. In such cases you need to specify the data range before the first call to the gridder.

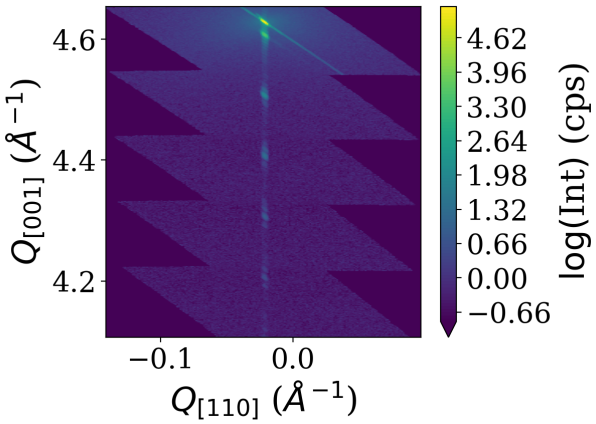
```
import xrayutilities as xu # import Python package
g = xu.Gridder2D(100, 101) # initialize the Gridder object
g.KeepData(True)
g.dataRange(1, 2, 3, 4)  # (xgrd_min, xgrd_max, ygrd_min, ygrd_max)
#====== load some data here =====
g(x, y, data) # call the gridder with the data
griddata = g.data # the data attribute contains the so far gridded data.

#====== load some more data here =====
g(x, y, data) # call the gridder with the new data
griddata = g.data # the data attribute contains the combined gridded data.
```

Gridder2D for visualization

Based on the example of parsed data from XRDML files shown above (Reading XRDML files) we show here how to use the **`Gridder2D`** class together with matplotlibs contourf.

```
Si = xu.materials.Si
hxrd = xu.HXRD(Si.Q(1, 1, 0), Si.Q(0, 0, 1))
qx, qy, qz = hxrd.Ang2Q(om, tt)
gridder = xu.Gridder2D(200, 600)
gridder(qy, qz, psd)
INT = xu.maplog(gridder.data.transpose(), 6, 0)
# plot the intensity as contour plot
plt.figure()
cf = plt.contourf(gridder.xaxis, gridder.yaxis, INT, 100, extend='min')
plt.xlabel(r'$Q_{[110]}$ ($\mathrm{\AA^{-1}}$)')
plt.ylabel(r'$Q_{[001]}$ ($\mathrm{\AA^{-1}}$)')
cb = plt.colorbar(cf)
cb.set_label(r"$\log($Int$)$ (cps)")
plt.tight_layout()
```

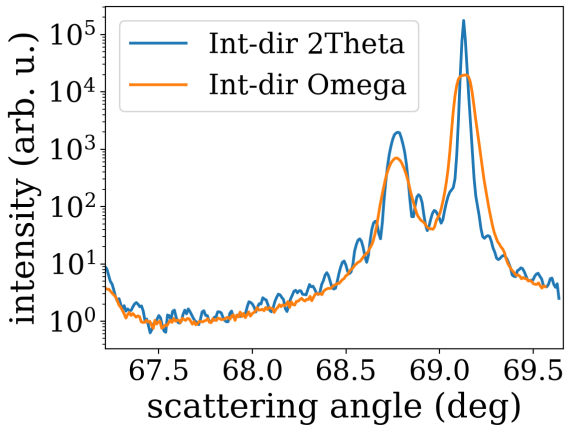
The shown script results in the plot of the reciprocal space map shown below.



Line cuts from reciprocal space maps

Using the **`analysis`** subpackage one can produce line cuts. Starting from the reciprocal space data produced by the reciprocal space conversion as in the last example code we extract radial scan along the crystal truncation rod. For the extraction of line scans the respective functions offer to integrate the data along certain directions. In the present case integration along ‘2Theta’ gives the best result since a broadening in that direction was caused by the beam footprint in the particular experiment. For different line cut functions various integration directions are possible. They are visualized in the figure below.

```
# line cut with integration along 2theta to remove beam footprint broadening
qzc, qzint, cmask = xu.analysis.get_radial_scan([qy, qz], psd, [0, 4.5],
                                                1001, 0.155, intdir='2theta')

# line cut with integration along omega
qzc_om, qzint_om, cmask_om = xu.analysis.get_radial_scan([qy, qz], psd, [0, 4.5],
                                                1001, 0.155, intdir='omega')
plt.figure()
plt.semilogy(qzc, qzint, label='Int-dir 2Theta')
plt.semilogy(qzc_om, qzint_om, label='Int-dir Omega')
plt.xlabel(r'scattering angle (deg)')
plt.ylabel(r'intensity (arb. u.)')
plt.legend()
plt.tight_layout()
```



Seealso

the fully working example provided in the examples directory and the other line cut functions in **`line_cuts`**

Using the `materials` subpackage

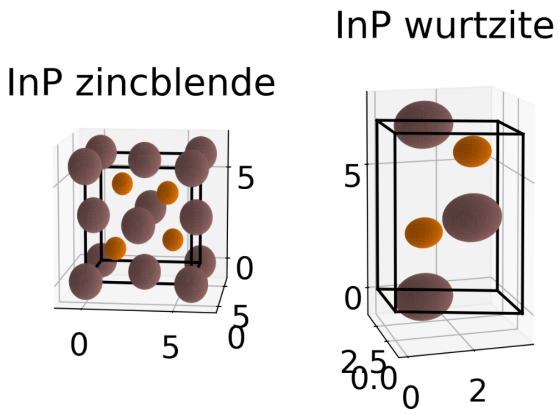
xrayutilities provides a set of Python classes to describe crystal lattices and materials.

Examples show how to define a new material by defining 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. Data for this are taken from a database which is included in the download.

First defining a new material from scratch is shown. This is done from the space group and Wyckhoff positions of the atoms inside the unit cell. Depending on the space group number the initialization of a new **`SGLattice`** object expects a different amount of parameters. For a cubic materials only the lattice parameter *a* should be given while for a triclinic materials *a*, *b*, *c*, *alpha*, *beta*, and *gamma* have to be specified. Its similar for the Wyckoff positions. While some Wyckoff positions require only the type of atom others have some free paramters which can be specified. Below we show the definition of zincblende InP as well as for its hexagonal wurtzite polytype together with a quick visualization of the unit cells. A more accurate visualization of the unit cell can be performed when using **`show_unitcell()`** with the Mayavi mode or by using the CIF-exporter and an external tool.

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

# elements (which contain their x-ray optical properties) are loaded from
# xrayutilities.materials.elements
In = xu.materials.elements.In
P = xu.materials.elements.P
```

```
 8
 9 # define elastic parameters of the material we use a helper function which
10 # creates the 6x6 tensor needed from the only 3 free parameters of a cubic
11 # material.
12 elastictensor = xu.materials.CubicElasticTensor(10.11e+10, 5.61e+10,
13                                                  4.56e+10)
14 # definition of zincblende InP:
15 InP = xu.materials.Crystal(
16     "InP", xu.materials.SGLattice(216, 5.8687, atoms=[In, P],
17                                   pos=['4a', '4c']),
18     elastictensor)
19
20 # a hexagonal equivalent which shows how parameters change for material
21 # definition with a different space group. Since the elasticity tensor is
22 # optional its not specified here.
23 InPWZ = xu.materials.Crystal(
24     "InP(WZ)", xu.materials.SGLattice(186, 4.1423, 6.8013,
25                                       atoms=[In, P], pos=[('2b', 0),
26                                                           ('2b', 3/8.)]))
27 f = plt.figure()
28 InP.show_unitcell(fig=f, subplot=121)
29 plt.title('InP zincblende')
30 InPWZ.show_unitcell(fig=f, subplot=122)
31 plt.title('InP wurtzite')
```



InP (in both variants) is already included in the xu.materials module and can be loaded by

```
InP = xu.materials.InP
InPWZ = xu.materials.InPWZ
```

Similar definitions exist for many other materials. Alternatively to giving the Wyckoff labels and parameters one can also specify the position of one atom for every unique site within the unit cell. *xrayutilities* will then search the corresponding Wyckoff position of this atom and populate therefore populate all equivalent sites as well. For the example of InP in zincblende form the material definition could also look as shown below. Note that instead of the elements also the elemental symbol as string can be used:

```
InP = xu.materials.Crystal(
    "InP", xu.materials.SGLattice(216, 5.8687, atoms=["In", "P"],
                                  pos=[(0,0,0), (1/4, 1/4, 1/4)]))
```

Using the material properties the calculation of the reflection strength of a Bragg reflection can be done as follows

```
1 import xrayutilities as xu
2
3 # defining material and experimental setup
4 InAs = xu.materials.InAs
5 energy= 8048 # eV
6
```

```
# 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.StructureFactor(qvec, energy)
    print(f"|F| = {abs(F):8.3f}")
```

Similar also the energy dependence of the structure factor can be determined

```
import matplotlib.pyplot as plt

energy= numpy.linspace(500, 20000, 5000) # 500 - 20000 eV
F = InAs.StructureFactorForEnergy(InAs.Q(1, 1, 1), energy)

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()
```

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

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

Fe = xu.materials.elements.Fe # iron atom
Q = [0, 0, 1.9]
en = 10000 # energy in eV

print(f"Iron (Fe): E: {en:9.1f} eV")
print(f"f0: {Fe.f0(xu.math.VecNorm(Q)):8.4g}")
print(f"f1: {Fe.f1(en):8.4g}")
print(f"f2: {Fe.f2(en):8.4g}")
```

```
Iron (Fe): E:   10000.0 eV
f0:    21.78
f1:  -0.0178
f2:    2.239
```

Transformation of `SGLattice`

`SGLattice`-objects can be transformed to use a different unit cell setting. This can be used to for example change the origin choice after the material definition or to convert into a totally different setting, e.g. for simulation purposes.

The code below shows the example of the Diamond structure converted between the two different origin choices

```
import numpy
import xrayutilities as xu
C1 = xu.materials.SGLattice("227:1", 3.5668, atoms=["C"],
                            pos=[(0,0,0)])
C2 = xu.materials.SGLattice("227:2", 3.5668, atoms=["C"],
                            pos=[(1/8, 1/8, 1/8)])
C1 == C2  # False
C3 = C2.transform(numpy.identity(3), (1/8, 1/8, 1/8))
C3 == C1  # True
```

For dynamical diffraction simulations of cubic crystals with (111) surface it might be required to convert the unit cell in a way that a principle axis is pointing along the surface normal. Using an apropriate conversion matrix this is shown for the example of InP

```
import xrayutilities as xu
InP111_lattice = xu.materials.InP.lattice.transform(((-1/2, 0, 1),
                                                     (1/2, -1/2, 1),
```

```
4                                                    (0, 1/2, 1)),
5                                                   (0, 0, 0))
```

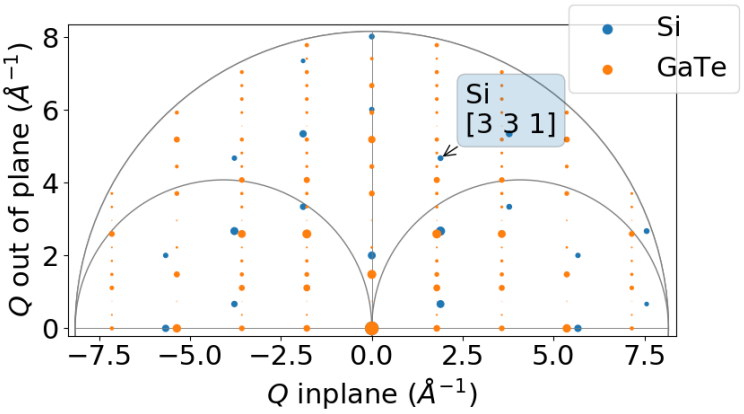
While the built in InP uses the cubic setting with space group F-43m(#216) the converted lattice has rhombohedral space group (in this case R3m(#160)) and converted atomic positions.

Visualization of the Bragg peaks in a reciprocal space plane

If you want to explore which peaks are available and reachable in coplanar diffraction geometry and what their relationship between different materials is *xrayutilities* provides a function which generates a slightly interactive plot which helps you with this task.

```
 1 import xrayutilities as xu
 2 mat = xu.materials.Crystal('GaTe',
 3                            xu.materials.SGLattice(194, 4.06, 16.96,
 4                                                   atoms=['Ga', 'Te'],
 5                                                   pos=[('4f', 0.17),
 6                                                        ('4f', 0.602)]))
 7 ttmax = 160
 8 sub = xu.materials.Si
 9 hsub = xu.HXRD(sub.Q(1, 1, -2), sub.Q(1, 1, 1))
10 ax, h = xu.materials.show_reciprocal_space_plane(sub, hsub, ttmax=160)
11 hxrd = xu.HXRD(mat.Q(1, 0, 0), mat.Q(0, 0, 1))
12 ax, h2 = xu.materials.show_reciprocal_space_plane(mat, hxrd, ax=ax)
```

The generated plot shows all the existing Bragg spots, their *(hkl)* label is shown when the mouse is over a certain spot and the diffraction angles calculated by the given **HXRD** object is printed when you click on a certain spot. Not that the primary beam is assumed to come from the left, meaning that high incidence geometry occurs for all peaks with positive inplane momentum transfer.



Calculation of diffraction angles for a general geometry

Often the restricted predefined geometries are not corresponding to the experimental setup, nevertheless *xrayutilities* is able to calculate the goniometer angles needed to reach a certain reciprocal space position.

For this purpose the goniometer together with the geometric restrictions need to be defined and the q-vector in laboratory reference frame needs to be specified. This works for arbitrary goniometer, however, the user is expected to set up bounds to put restrictions to the number of free angles to obtain reproducible results. In general only three angles are needed to fit an arbitrary q-vector (2 sample + 1 detector angles or 1 sample + 2 detector). More goniometer angles can be kept free if some pseudo-angle constraints are used instead.

The example below shows the necessary code to perform such an angle calculation for a custom defined material with orthorhombic unit cell.

```
1 import xrayutilities as xu
2 import numpy as np
3
4 def Pnma(a, b, c):
```

```
    # create orthorhombic unit cell with space-group 62,
    # here for simplicity without base
    l = xu.materials.SGLattice(62, a, b, c)
    return l

latticeConstants=[5.600, 7.706, 5.3995]
SmFeO3 = xu.materials.Crystal("SmFeO3", Pnma(*latticeConstants))
# 2S+2D goniometer
qconv=xu.QConversion(('x+', 'z+'), ('z+', 'x+'), (0, 1, 0))
# [1,1,0] surface normal
hxrd = xu.Experiment(SmFeO3.Q(0, 0, 1), SmFeO3.Q(1, 1, 0), qconv=qconv)

hkl=(2, 0, 0)
q_material = SmFeO3.Q(hkl)
q_laboratory = hxrd.Transform(q_material) # transform

print(f"SmFeO3: hkl {hkl}, qvec {np.round(q_material, 5)}")
print(f"Lattice plane distance: {SmFeO3.planeDistance(hkl):.4f}")

#### determine the goniometer angles with the correct geometry restrictions
# tell bounds of angles / (min,max) pair or fixed value for all motors.
# maximum of three free motors! here the first goniometer angle is fixed.
# om, phi, tt, delta
bounds = (5, (-180, 180), (-1, 90), (-1, 90))
ang, qerror, errcode = xu.Q2AngFit(q_laboratory, hxrd, bounds)
print(f"err {errcode} ({qerror:.3g}) angles {np.round(ang, 5)}")
# check that qerror is small!!
print("sanity check with back-transformation (hkl): ",
      np.round(hxrd.Ang2HKL(*ang,mat=SmFeO3),5))
```

The output of the code above would be similar to:

```
SmFeO3: hkl (2, 0, 0), qvec [2.24399 0.      0.     ]
Lattice plane distance: 2.8000
err 0 (9.61e-09) angles [ 5.      20.44854 19.65315 25.69328]
sanity check with back-transformation (hkl):  [ 2.  0. -0.]
```

In the example above all angles can be kept free if a pseudo-angle constraint is used in addition. This is shown below for the incidence angle, which when fixed to 5 degree results in the same goniometer angles as shown above. Currently two helper functions for incidence and exit angles (**`incidenceAngleConst()`** and **`exitAngleConst()`**) are implemented, but user-defined functions can be supplied.

```
aiconstraint = xu.q2ang_fit.incidenceAngleConst
bounds = ((0, 90), (-180, 180), (-1, 90), (-1, 90))
ang, qerror, errcode = xu.Q2AngFit(
    q_laboratory, hxrd, bounds,
    constraints=[{'type':'eq', 'fun': lambda a: aiconstraint(a, 5, hxrd)}, ])
```

User-specific config file

Several options of *xrayutilities* can be changed by options in a config file. This includes the default x-ray energy as well as parameters to set the number of threads used by the parallel code and the verbosity of the output.

The default options are stored inside the installad Python module and should not be changed. Instead it is suggested to use a user-specific config file '~/.xrayutilities.conf' or a 'xrayutilities.conf' file in the working directory.

An example of such a user config file is shown below:

```
# begin of xrayutilities configuration
[xrayutilities]

# verbosity level of information and debugging outputs
#   0: no output
```

```
#   1: very import notes for users
#   2: less import notes for users (e.g. intermediate results)
#   3: debuging output (e.g. print everything, which could be interesing)
#   levels can be changed in the config file as well
verbosity = 1

# default wavelength in angstrom,
wavelength = MoKa1 # Molybdenum K alpha1 radiation (17479.374eV)

# default energy in eV
# if energy is given wavelength settings will be ignored
#energy = 10000 #eV

# number of threads to use in parallel sections of the code
nthreads = 1
#   0: the maximum number of available threads will be used (as returned by
#      omp_get_max_threads())
#   n: n-threads will be used
```

Determining detector parameters

In the following three examples of how to determine the detector parameters for linear and area detectors is given. The procedure we use is in more detail described in this article.

Linear detectors

To determine the detector parameters of a linear detector one needs to perform a scan with the detector angle through the primary beam and aquire a detector spectrum at any point.

Using the following script determines the parameters necessary for the detector initialization, which are:

- pixelwidth of one channel
- the center channel
- and the detector tilt (optional)

```
"""
example script to show how the detector parameters
such as pixel width, center channel and detector tilt
can be determined for a linear detector.
"""

import os

import xrayutilities as xu

# load any data file with with the detector spectra of a reference scan
# in the primary beam, here I use spectra measured with a Seifert XRD
# diffractometer
dfile = os.path.join("data", "primarybeam_alignment20130403_2_dis350.nja")
s = xu.io.SeifertScan(dfile)

ang = s.axispos["T"]  # detector angles during the scan
spectra = s.data[:, :, 1]  # detector spectra aquired

# determine detector parameters
# this function accepts some optional arguments to describe the goniometer
# see the API documentation
pwidth, cch, tilt = xu.analysis.linear_detector_calib(ang, spectra,
                                                      usetilt=True)
```

Area detector (Variant 1)

To determine the detector parameters of a area detector one needs to perform scans with the detector angles through the primary beam and aquire a detector images at any position. For the area detector at least two scans (one with the outer detector and and one with the inner detector angle) are required.

Using the following script determines the parameters necessary for the detector initialization from such scans in the primary beam only. Further down we discuss an other variant which is also able to use additionally detector images recorded at the Bragg reflection of a known reference crystal.

The determined detector parameters are:

- center channels: position of the primary beam at the true zero position of the goniometer (considering the outer angle offset) (2 parameters)
- pixelwidth of the channels in both directions (2 parameters), these two parameters can be replaced by the detector distance (1 parameter) if the pixel size is given as an input
- detector tilt azimuth in degree from 0 to 360
- detector tilt angle in degree (>0deg)
- detector rotation around the primary beam in degree
- outer angle offset, which describes a offset of the outer detector angle from its true zero position

The misalignment parameters as well as the pixel size can be fixed during the fitting.

```
"""
example script to show the detector parameter determination for area detectors
from images recorded in the primary beam
"""

import os

import xrayutilities as xu

en = 10300.0  # eV
datadir = os.path.join("data", "wire_")  # data path for CCD files
# template for the CCD file names
filetmp = os.path.join(datadir, "wire_12_%05d.edf.gz")

# manually selected images
# select images which have the primary beam fully on the CCD
imagenrs = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
            20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]

images = []
ang1 = []
ang2 = []

# read images and angular positions from the data file
# this might differ for data taken at different beamlines since
# they way how motor positions are stored is not always consistent
for imgnr in imagenrs:
    filename = filetmp % imgnr
    edf = xu.io.EDFFile(filename)
    images.append(edf.data)
    ang1.append(float(edf.header['ESRF_ID01_PSIC_NANO_NU']))
    ang2.append(float(edf.header['ESRF_ID01_PSIC_NANO_DEL']))


# call the fit for the detector parameters
# detector arm rotations and primary beam direction need to be given.
# in total 9 parameters are fitted, however the severl of them can
# be fixed. These are the detector tilt azimuth, the detector tilt angle, the
```

```
# detector rotation around the primary beam and the outer angle offset
# The detector pixel size or the detector distance should be kept unfixed to
# be optimized by the fit.
param, eps = xu.analysis.sample_align.area_detector_calib(
    ang1, ang2, images, ['z+', 'y-'], 'x+',
    start=(None, None, 1.0, 45, 0, -0.7, 0),
    fix=(False, False, True, False, False, False, False),
    wl=xu.en2lam(en))
```

A possible output of this script could be

> fitted parameters: epsilon: 8.0712e-08 (2,['Parameter convergence']) param: (cch1,cch2,pwidth1,pwidth2,tiltazimuth,tilt,detrot,outerangle_offset) param: 140.07 998.34 4.4545e-05 4.4996e-05 72.0 1.97 -0.792 -1.543 please check the resulting data (consider setting plot=True) detector rotation axis / primary beam direction (given by user): ['z+', 'y-'] / x+ detector pixel directions / distance: z- y+ / 1 detector initialization with: init_area('z-', 'y+', cch1=140.07, cch2=998.34, Nch1=516, Nch2=516, pwidth1=4.4545e-05, pwidth2=4.4996e-05, distance=1., detrot=-0.792, tiltazimuth=72.0, tilt=1.543) AND ALWAYS USE an (additional) OFFSET of -1.9741deg in the OUTER DETECTOR ANGLE!

The output gives the fitted detector parameters and compiles the Python code line one needs to use to initialize the detector. Important to note is that the outer angle offset which was determined by the fit (-1.9741 degree in the aboves example) is not included in the initialization of the detector parameters *but* needs to be used in every call to the q-conversion function as offset. This step needs to be performed manually by the user!

Area detector (Variant 2)

In addition to scans in the primary beam this variant enables also the use of detector images recorded in scans at Bragg reflections of a known reference materials. However this also required that the sample orientation and x-ray wavelength need to be fit. To keep the additional parameters as small as possible we only implemented this for symmetric coplanar diffractions.

The advantage of this method is that it is more sensitive to the outer angle offset also at large detector distances. The additional parameters are:

- sample tilt angle in degree
- sample tilt azimuth in degree
- and the x-ray wavelength in angstrom

```
"""
example script to show the detector parameter determination for area detectors
from images recorded in the primary beam and at known symmetric coplanar Bragg
reflections of a reference crystal
"""

import os

import numpy
import xrayutilities as xu

Si = xu.materials.Si

datadir = 'data'
specfile = "si_align.spec"

en = 15000  # eV
wl = xu.en2lam(en)
imgdir = os.path.join(datadir, "si_align_")  # data path for CCD files
filetmp = "si_align_12_%04d.edf.gz"

qconv = xu.QConversion(['z+', 'y-'], ['z+', 'y-'], [1, 0, 0])
hxrd = xu.HXRD(Si.Q(1, 1, -2), Si.Q(1, 1, 1), wl=wl, qconv=qconv)

```

```
# manually selected images

s = xu.io.SPECFile(specfile, path=datadir)
imagenrs = []
for num in [61, 62, 63, 20, 21, 26, 27, 28]:
    s[num].ReadData()
    imagenrs = numpy.append(imagenrs, s[num].data['ccd_n'])

# avoid images which do not have to full beam on the detector as well as
# other which show signal due to cosmic radiation
avoid_images = [37, 57, 62, 63, 65, 87, 99, 106, 110, 111, 126, 130, 175,
                181, 183, 185, 204, 206, 207, 208, 211, 212, 233, 237, 261,
                275, 290]

images = []
ang1 = []  # outer detector angle
ang2 = []  # inner detector angle
sang = []  # sample rocking angle
hkls = []  # Miller indices of the reference reflections


def hotpixelkill(ccd):
    """
    function to remove hot pixels from CCD frames
    ADD REMOVE VALUES IF NEEDED!
    """
    ccd[304, 97] = 0
    ccd[303, 96] = 0
    return ccd


# read images and angular positions from the data file
# this might differ for data taken at different beamlines since
# they way how motor positions are stored is not always consistent
for imgnr in numpy.sort(list(set(imagenrs) - set(avoid_images))[::4]):
    filename = os.path.join(imgdir, filetmp % imgnr)
    edf = xu.io.EDFFile(filename)
    ccd = hotpixelkill(edf.data)
    images.append(ccd)
    ang1.append(float(edf.header['motor_pos'].split()[4]))
    ang2.append(float(edf.header['motor_pos'].split()[3]))
    sang.append(float(edf.header['motor_pos'].split()[1]))
    if imgnr > 1293.:
        hkls.append((0, 0, 0))
    elif imgnr < 139:
        hkls.append((0, 0, numpy.sqrt(27)))  # (3,3,3))
    else:
        hkls.append((0, 0, numpy.sqrt(75)))  # (5,5,5))

# call the fit for the detector parameters.
# Detector arm rotations and primary beam direction need to be given
# in total 8 detector parameters + 2 additional parameters for the reference
# crystal orientation and the wavelength are fitted, however the 4 misalignment
# parameters of the detector and the 3 other parameters can be fixed.
# The fixable parameters are detector tilt azimuth, the detector tilt angle,
# the detector rotation around the primary beam, the outer angle offset, sample
# tilt, sample tilt azimuth and the x-ray wavelength
# Additionally if accurately known the detector pixel size can be given and
# fixed and instead the detector distance can be fitted.
param, eps = xu.analysis.area_detector_calib_hkl(
```

```
    sang, ang1, ang2, images, hkls, hxrd, Si, ['z+', 'y-'], 'x+',
    start=(None, None, 1.0, 45, 1.69, -0.55, -1.0, 1.3, 60., wl),
    fix=(False, False, True, False, False, False, False, False, False, False),
    plot=True)

# Following is an example of the output of the summary of the
# area_detector_calib_hkl function
# total time needed for fit: 624.51sec
# fitted parameters: epsilon: 9.9159e-08 (2,['Parameter convergence'])
# param:
# (cch1,cch2,pwidth1,pwidth2,tiltazimuth,tilt,detrot,outerangle_offset,
# sampletilt,stazimuth,wavelength)
# param: 367.12 349.27 6.8187e-05 6.8405e-05 131.4 2.87 -0.390 -0.061 1.201
# 318.44 0.8254
# please check the resulting data (consider setting plot=True)
# detector rotation axis / primary beam direction (given by user): ['z+', 'y-']
# / x+
# detector pixel directions / distance: z- y+ / 1
#   detector initialization with:
#   init_area('z-', 'y+', cch1=367.12, cch2=349.27, Nch1=516, Nch2=516,
#   pwidth1=6.8187e-05, pwidth2=6.8405e-05, distance=1., detrot=-0.390,
#   tiltazimuth=131.4, tilt=2.867)
# AND ALWAYS USE an (additional) OFFSET of -0.0611deg in the OUTER
# DETECTOR ANGLE!
```

Simulation examples

In the following a few code-snippets are shown which should help you getting started with reflectivity and diffraction simulations using *xrayutilities*. All simulations in *xrayutilities* are for layers systems and currently there are no plans to extend this to other geometries. Note that not all of the codes shown in the following will be run-able as stand-alone scripts. For fully running scripts look in the examples directory in the download.

Building Layer stacks for simulations

The basis of all simulations in *xrayutilities* are stacks of layers. Therefore several functions exist to build up such layered systems. The basic building block of all of them is a **`Layer`** object which takes a material and its thickness in ångström as initializing parameter.

```
import xrayutilities as xu
lay = xu.simpack.Layer(xu.materials.Si, 200)
```

In the shown example a silicon layer with 20 nm thickness is created. The first argument is the material of the layer. For diffraction simulations this needs to be derived from the **`Crystal`**-class. This means all predefined materials in *xrayutitities* can be used for this purpose. For x-ray reflectivity simulations, however, also knowing the chemical composition and density of the material is sufficient.

A 5 nm thick metallic CoFe compound layer can therefore be defined sublayers

```
rho_cf = 0.5*8900 + 0.5*7874  # mass density in kg/m^3
mCoFe = xu.materials.Amorphous('CoFe', rho_cf)
lCoFe = xu.simpack.Layer(mCoFe, 50)
```

Note

The **`Layer`** object can have several more model dependent properties discussed in detail below.

When several layers are defined they can be combined to a **`LayerStack`** which is used for the simulations below.

```
sub = xu.simpack.Layer(xu.materials.Si, float('inf'))
lay1 = xu.simpack.Layer(xu.materials.Ge, 200)
lay2 = xu.simpack.Layer(xu.materials.SiO2, 30)
ls = xu.simpack.LayerStack('Si/Ge', sub, lay1, lay2)
# or equivalently
ls = xu.simpack.LayerStack('Si/Ge', sub + lay1 + lay2)
```

The last two lines show two different options of creating a stack of layers. As is shown in the last example the substrate thickness can be infinite (see below) and layers can be also stacked by summation. For creation of more complicated superlattice stacks one can further use multiplication

```
lay1 = xu.simpack.Layer(xu.materials.SiGe(0.3), 50)
lay2 = xu.simpack.Layer(xu.materials.SiGe(0.6), 40)
layerstack = xu.simpack.LayerStack('Si/SiGe SL', sub + 5*(lay1 + lay2))
```

Pseudomorphic Layers

All stacks of layers described above use the materials in the layer as they are supplied. However, epitaxial systems often adopt the inplane lattice parameter of the layers beneath. To mimic this behavior you can either supply the **`Layer`** objects which custom **`Crystal`** objects which have the appropriate lattice parameters or use the **`PseudomorphicStack*`** classes which to the adaption of the lattice parameters automatically. In this respect the 'relaxation' parameter of the **`Layer`** class is important since it allows to create partially/fully relaxed layers.

```
sub = xu.simpack.Layer(xu.materials.Si, float('inf'))
buf1 = xu.simpack.Layer(xu.materials.SiGe(0.5), 5000, relaxation=1.0)
buf2 = xu.simpack.Layer(xu.materials.SiGe(0.8), 5000, relaxation=1.0)
lay1 = xu.simpack.Layer(xu.materials.SiGe(0.6), 50, relaxation=0.0)
lay2 = xu.simpack.Layer(xu.materials.SiGe(1.0), 50, relaxation=0.0)
# create pseudomorphic superlattice stack
pls = xu.simpack.PseudomorphicStack001('SL 5/5', sub+buf1+buf2+5*(lay1+lay2))
```

Note

As indicated by the function name the PseudomorphicStack currently only works for (001) surfaces and cubic materials. Implementations for other surface orientations are planned.

If you would like to check the resulting lattice objects of the different layers you could use:

```
for l in pls:
    print(l.material.lattice)
```

Special layer types

So far one special layer mimicking a layer with gradually changing chemical composition is implemented. It consists of several thin sublayers of constant composition. So in order to obtain a smooth grading one has to select enough sublayers. This however has a negativ impact on the performance of all simulation models. A tradeoff needs to found! Below a graded SiGe buffer is shown which consists of 100 sublayers and has total thickness of 1 µm.

```
buf = xu.simpack.GradedLayerStack(xu.materials.SiGe,
                                  0.2,  # xfrom Si0.8Ge0.2
                                  0.7,  # xto Si0.3Ge0.7
                                  100,  # number of sublayers
                                  10000,  # total thickness
                                  relaxation=1.0)
```

Setting up a model

This sectiondescribes the parameters which are common for all diffraction models in *xrayutilties*.`simpack`. All models need a list of Layers for which the reflected/diffracted signal will be calculated. Further all models have some common parameters which allow scaling and background addition in the model output and contain general information about the calculation which are model-independent. These are

- 'experiment': an **`Experiment`**/**`HXRD`** object which defines the surface geometry of the model. If none is given a default class with (001) surface is generated.
- 'resolution_width': width of the Gaussian resolution function used to convolute with the data. The unit of this parameters depends on the model and can be either in degree or 1/AA.
- 'I0': is the primary beam flux/intensity
- 'background': is the background added to the simulation after it was scaled by I0
- 'energy': energy in eV used to obtain the optical parameters for the simulation. The energy can alternatively also be supplied via the 'experiment' parameter, however, the 'energy' value overrules this setting. If no energy is given the default energy from the configuration is used.

The mentioned parameters can be supplied to the constructor method of all model classes derived from **`LayerModel`**, which applies to all examples mentioned below.

```
m = xu.simpack.SpecularReflectivityModel(layerstack, I0=1e6, background=1,
                                         resolution_width=0.001)
```

Reflectivity calculation and fitting

This section shows the calculation and fitting of specular x-ray reflectivity curves as well as the calculation of diffuse x-ray reflectivity curves/maps.

Specular x-ray reflectivity

For the specular reflectivity models currently only the Parrat formalism including non-correlated roughnesses is implemented. A minimal working example for a reflectivity calculation follows.

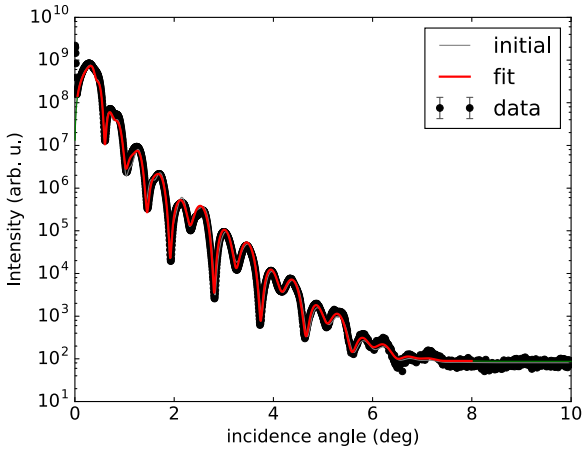
```
import numpy
# building a stack of layers
sub = xu.simpack.Layer(xu.materials.GaAs, float('inf'), roughness=2.0)
lay1 = xu.simpack.Layer(xu.materials.AlGaAs(0.25), 75, roughness=2.5)
lay2 = xu.simpack.Layer(xu.materials.AlGaAs(0.75), 25, roughness=3.0)
pls = xu.simpack.PseudomorphicStack001('pseudo', sub+5*(lay1+lay2))

# reflectivity calculation
m = xu.simpack.SpecularReflectivityModel(pls, sample_width=5, beam_width=0.3)
ai = numpy.linspace(0, 5, 10000)
Ixrr = m.simulate(ai)
```

In addition to the layer thickness also the roughness and density (in kg/m^3) of a Layer can be set since they are important for the reflectivity calculation. This can be done upon definition of the **`Layer`** or also manipulated at any later stage. Such x-ray reflectivity calculations can also be fitted to experimental data using the **`FitModel`** class which is shown in detail in the example below (which is also included in the example directory). The fitting is performed using the lmfit Python package which needs to be installed when you want to use this fitting function. This package allows to build complicated models including bounds and correlations between parameters.

```
import lmfit
import numpy

import xrayutilities as xu

# load experimental data
ai, edata, eps = numpy.loadtxt('data/xrr_data.txt'), unpack=True)
ai /= 2.0
```

```
 9
10 # define layers
11 # SiO2 / Ru(5) / CoFe(3) / IrMn(3) / AlOx(10)
12 lSiO2 = xu.simpack.Layer(xu.materials.SiO2, numpy.inf, roughness=2.5)
13 lRu = xu.simpack.Layer(xu.materials.Ru, 47, roughness=2.8)
14 rho_cf = 0.5*8900 + 0.5*7874
15 mat_cf = xu.materials.Amorphous('CoFe', rho_cf)
16 lCoFe = xu.simpack.Layer(mat_cf, 27, roughness=4.6)
17 lIrMn = xu.simpack.Layer(xu.materials.Ir20Mn80, 21, roughness=3.0)
18 lAl2O3 = xu.simpack.Layer(xu.materials.Al2O3, 100, roughness=5.5)
19
20 # create model
21 m = xu.simpack.SpecularReflectivityModel(lSiO2, lRu, lCoFe, lIrMn, lAl2O3,
22                                          energy='CuKa1', resolution_width=0.02,
23                                          sample_width=6, beam_width=0.25,
24                                          background=81, I0=6.35e9)
25
26 # embed model in fit code
27 fitm = xu.simpack.FitModel(m, plot=True, verbose=True)
28
29 # set some parameter limitations
30 fitm.set_param_hint('SiO2_density', vary=False)
31 fitm.set_param_hint('Al2O3_density', min=0.8*xu.materials.Al2O3.density,
32                     max=1.2*xu.materials.Al2O3.density)
33 p = fitm.make_params()
34 fitm.set_fit_limits(xmin=0.05, xmax=8.0)
35
36 # perform the fit
37 res = fitm.fit(edata, p, ai, weights=1/eps)
38 lmfit.report_fit(res, min_correl=0.5)
39 # export the fit result for the full data range (Note that only data between
40 # xmin and xmax were actually optimized)
41 numpy.savetxt(
42     "xrrfit.dat",
43     numpy.vstack((ai, res.eval(res.params, x=ai))).T,
44     header="incidence angle (deg), fitted intensity (arb. u.)",
45 )
```

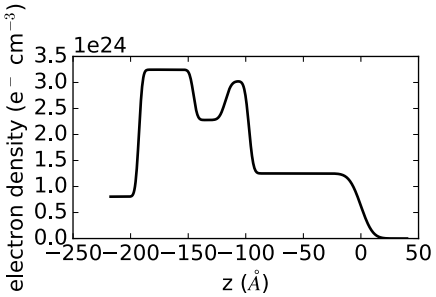
This script can interactively show the fitting progress and after the fitting shows the final plot including the x-ray reflectivity trace of the initial and final parameters.



The picture shows the final plot of the fitting example shown in one of the example scripts.

After building a **SpecularReflectivityModel** is built or fitted the density profile resulting from the thickness and roughness of layers can be plotted easily by

```
m.densityprofile(500, plot=True)  # 500 number of points
```

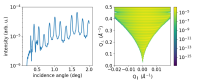


Diffuse reflectivity calculations

For the calculation of diffuse x-ray reflectivity the **`LayerStack`** is built equally as shown before. The only difference is that an additional parameter for the lateral correlation length of the roughness can be included: *lat_correl*. The **`DiffuseReflectivityModel`** also takes special parameters which change the vertical correlection length and the way how the diffuse reflectivity is calculated (to be document in more detail). For a Si/Ge superlattice with 5 periods the calculation of the diffuse reflectivity signal at the specular rod is calculated using the **`simulate()`** method. A map of the diffuse reflectivity which can be obtained in the coplanar reflection plane can be calculated with the **`simulate_map()`** method.

```
from matplotlib.pylab import *
import xrayutilities as xu
sub = xu.simpack.Layer(xu.materials.Si, inf, roughness=1, lat_correl=100)
lay1 = xu.simpack.Layer(xu.materials.Si, 200, roughness=1, lat_correl=200)
lay2 = xu.simpack.Layer(xu.materials.Ge, 70, roughness=3, lat_correl=50)
ls = xu.simpack.LayerStack('SL 5', sub+5*(lay2+lay1))

alphai = arange(0.17, 2, 0.001)  # for the calculation on the specular rod
qz = arange(0, 0.5, 0.0005)  # for the map calculation
qL = arange(-0.02, 0.02, 0.0003)

m = xu.simpack.DiffuseReflectivityModel(ls, sample_width=10, beam_width=1,
                                        energy='CuKa1', vert_correl=1000,
                                        vert_nu=0, H=1, method=2, vert_int=0)
d = m.simulate(alphai)
imap = m.simulate_map(qL, qz)

figure()
subplot(121)
semilogy(alphai, d, label='diffuse XRR')
xlabel('incidence angle (deg)')
ylabel('intensity (arb. u.)')
ylim(1e-6, 1e-4)

subplot(122)
pcolor(qL, qz, imap.T, norm=mpl.colors.LogNorm())
xlabel(r'Q$_\parallel$ ($\mathrm{\AA^{-1}}$)')
ylabel(r'Q$_\perp$ ($\mathrm{\AA^{-1}}$)')
colorbar()
tight_layout()
```

The resulting figure shows the simulation result. Currently you have to refer to the docstrings and implementation for further details.



Diffraction calculation

From the very same models as used for XRR calculation one can also perform crystal truncation rod simulations around certain Bragg peaks using various different diffraction models. Depending on the system to model you will have to choose the most apropriate model. Below a short description of the implemented models is given followed by two examples.

Kinematical diffraction models

The most basic models consider only the kinematic diffraction of layers and substrate. Especially the semiinfinite substrate is not well described using the kinematical approximation which results in considerable deviations in close vicinity to substrate Bragg peak with respect to the more acurate dynamical diffraction models.

Such a basic model is employed by

```
en = 9000  # eV
mk = xu.simpack.KinematicalModel(pls, energy=en, resolution_width=0.0001)
Ikin = mk.simulate(qz, hkl=(0, 0, 4))
```

A more appealing kinematical model is represented by the **`KinematicalMultiBeamModel`** class which implements a true multibeam theory is, however, restricted to the use of (001) surfaces and layer thicknesses will be changed to be a multiple of the out of plane lattice spacing. This is necessary since otherwise the structure factor of the unit cell can not be used for the calculation.

It can be employed by

```
mk = xu.simpack.KinematicalMultiBeamModel(pls, energy=en,
                                          surface_hkl=(0, 0, 1),
                                          resolution_width=0.0001)
Imult = mk.simulate(qz, hkl=(0, 0, 4))
```

This model is expected to provide good results especially far away from the substrate peak where the influence of other Bragg peaks on the truncation rod and the variation of the structure factor can not be neglected.

Both kinematical model's **`simulate()`** method offers two keyword arguments with which basic absorption and refraction correction can be added to the basic models.

Note

The kinematical models can also handle a semi-infinitely thick substrate which results in a diverging intensity at the Bragg peak but provides a basic description of the substrates truncation rod.

Dynamical diffraction models

Acurate description of the diffraction from thin films in close vicinity to the diffraction signal from a bulk substrate is only possible using the dynamical diffraction theory. In **xrayutilities** the dynamical two-beam theory with 4 tiepoints for the calculation of the dispersion surface is implemented. To use this theory you have to supply the **`simulate()`** method with the incidence angle in degree. Accordingly the 'resolution_width' parameter is also in degree for this model.

```
resol = 0.001  # resolution in incidence angle
md = xu.simpack.DynamicalModel(pls, energy=en, resolution_width=resol)
Idyn = md.simulate(ai, hkl=(0, 0, 4))
```

A second simplified dynamical model (**`SimpleDynamicalCoplanarModel`**) is also implemented should, however, not be used since its approximations cause mistakes in almost all relevant cases.

The **`DynamicalModel`** supports the calculation of diffracted signal for 'S' and 'P' polarization geometry. To simulate diffraction data of laboratory sources with Ge(220) monochromator crystal one should use:

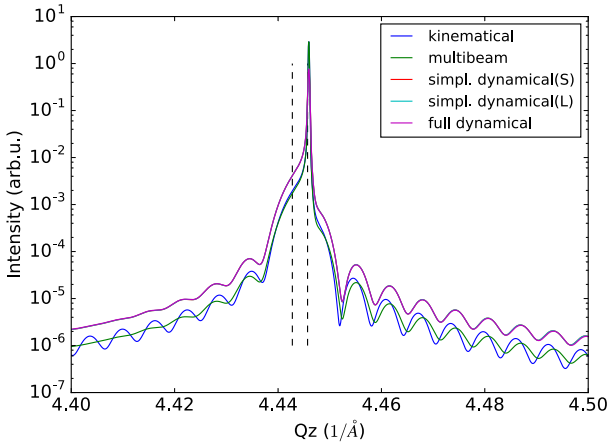
```
.. testcode::
```

```
import math qGe220 = linalg.norm(xu.materials.Ge.Q(2, 2, 0)) thMono = math.asin(qGe220 * xu.config.WAVELENGTH / (4*math.pi)) md = xu.simpack.DynamicalModel(pls,
    Cmono=math.cos(2 * thMono), polarization='both')
Idyn = md.simulate(ai, hkl=(0, 0, 4))
```

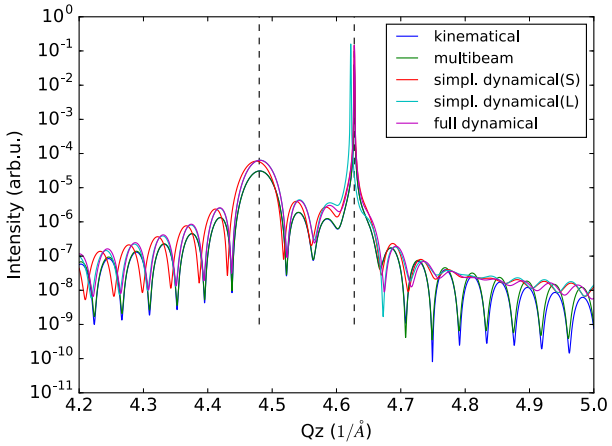
Comparison of diffraction models

Below we show the different implemented models for the case of epitaxial GaAs/AlGaAs and Si/SiGe bilayers. These two cases have very different separation of the layer Bragg peak from the substrate and therefore provide good model system for our models.

We will compare the (004) Bragg peak calculated with different models and but otherwise equal parameters. For scripts used to perform the shown calculation you are referred to the `examples` directory.



XRD simulations of the (004) Bragg peak of ~100 nm AlGaAs on GaAs(001) using various diffraction models



XRD simulations of the (004) Bragg peak of 15 nm $Si_{0.4}$ $Ge_{0.6}$ on Si(001) using various diffraction models

As can be seen in the images we find that for the AlGaAs system all models except the very basic kinematical model yield an very similar diffraction signal. The second kinematic diffraction model considering the contribution of multiple Bragg peaks on the same truncation rod fails to describe only the ratio of substrate and layer signal, but otherwise results in a very similar line shape as the traces obtained by the dynamic theory.

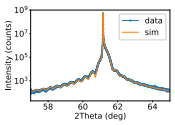
For the SiGe/Si bilayer system bigger differences between the kinematic and dynamic models are found. Further also the difference between the simpler and more sophisticated dynamic model gets obvious further away from the reference position. Interestingly also the multibeam kinematic theory differs considerable from the best dynamic model. As is evident from this second comparison the correct choice of model for the particular system under condideration is crucial for comparison with experimental data.

Fitting of diffraction data

All diffraction models can be embedded into the **`FitModel`** class, which is suitable to refine the model parameters. Below (and in the examples directory) a runnable script is shown which shows the fitting for a pseudomorphic InMnAs epilayer on InAs(001). The fitting is performed using the lmfit Python package which needs to be installed when you want to use this fitting function. As one can see below the **`set_param_hint()`** function can be used to set up the respective fit parameters including their boundaries and possible correlation with other parameters of the model. It should be equally possible to fit more complex layer structures, however, I expect that one needs to adjust manually the starting parameters to yield something very reasonable. Since this capabilities are rather new please report back any success/problems you have with this via the mailing list.

```
from copy import deepcopy
import xrayutilities as xu
from matplotlib.pylab import *

# global parameters
wavelength = xu.wavelength('CuKa1')
offset = -0.035  # angular offset of the zero position of the data

# set up LayerStack for simulation: InAs(001)/(In,Mn)As(~25 nm)
InAs = deepcopy(xu.materials.InAs)  # do not modify internal database
InAs.a = 6.057
lInAs = xu.simpack.Layer(InAs, inf)
InMnAs = xu.materials.Crystal('InMnAs', xu.materials.SGLattice(
    216, 6.050, atoms=('In', 'Mn', 'As'), pos=('4a', '4a', '4c'),
    occ=(0.99, 0.01, 1)), cij=InAs.cij)
lInMnAs = xu.simpack.Layer(InMnAs, 254, relaxation=0)
pstack = xu.simpack.PseudomorphicStack001('list', lInAs, lInMnAs)

# set up simulation object
thetaMono = arcsin(wavelength/(2 * xu.materials.Ge.planeDistance(2, 2, 0)))
Cmono = cos(2 * thetaMono)
dyn = xu.simpack.DynamicalModel(pstack, I0=1.5e9, background=0,
                                resolution_width=2e-3, polarization='both',
                                Cmono=Cmono)
fitmdyn = xu.simpack.FitModel(dyn)
fitmdyn.set_param_hint('InMnAs_c', vary=True, min=6.02, max= 6.06)
fitmdyn.set_param_hint('InAs_a', vary=True)
fitmdyn.set_param_hint('InMnAs_a', expr='InAs_a')
fitmdyn.set_param_hint('resolution_width', vary=True)
params = fitmdyn.make_params()

# plot experimental data
f = figure(figsize=(7,5))
d = xu.io.RASFile('inas_layer_radial_002_004.ras.bz2', path='examples/data')
scan = d.scans[-1]
tt = scan.data[scan.scan_axis] - offset
semilogy(tt, scan.data['int'], 'o-', ms=3, label='data')

# perform fit and plot the result
fitmdyn.lmodel.set_hkl((0, 0, 4))
ai = (d.scans[-1].data[d.scan.scan_axis] - offset)/2
fitr = fitmdyn.fit(d.scans[-1].data['int'], params, ai)
# print(fitr.fit_report())  # to get a summary of the fitted parameters
```

The resulting figure shows reasonable agreement between the dynamic diffraction simulation and the experimental data.



Powder diffraction simulations

Powder diffraction patterns can be calculated using **PowderModel**. A specialized class for the definition of powdered materials named **Powder** exists. The class constructor takes the materials volume and several material parameters specific for the powder material. Among them are *crystallite_size_gauss* and *strain_gauss* which can be used to include the effect of finite crystallite size and microstrain. Texture modelled by the March-Dollase pole density offers the *preferred_orientation* direction parameter as well as a *preferred_orientation_factor* variable.

The **PowderModel** internally uses **PowderDiffraction** for its calculations which is based on the fundamental parameters approach. The fundamental parameters approach code used here can was originally implemented and documented here: http://dx.doi.org/10.6028/jres. 120.014.c and http://dx.doi.org/10.6028/jres.120.014.

Several setup specific parameters should be adjusted by a user-specific configuration file or by supplying the appropriate parameters using the *fpsettings* argument of **PowderModel**.

If the correct settings are included in the config file the powder diffraction signal of a mixed sample of Co and Fe can be calculated with

```
 1 import numpy
 2 import xrayutilities as xu
 3
 4 tt = numpy.arange(5, 120, 0.01)
 5 Fe_powder = xu.simpack.Powder(xu.materials.Fe, 1,
 6                               crystallite_size_gauss=100e-9)
 7 Co_powder = xu.simpack.Powder(xu.materials.Co, 5,  # 5 times more Co
 8                               crystallite_size_gauss=200e-9)
 9 pm = xu.simpack.PowderModel(Fe_powder, Co_powder, I0=100)
10 inte = pm.simulate(tt)
11 pm.close()  # after end-of-use
```

Note that in MS windows and macOS you need to encapsulate this code into a dummy function to allow for the multiprocessing module to work correctly. See the Python documentation for details. The code then must look like:

```
 1 import numpy
 2 import xrayutilities as xu
 3 from multiprocessing import freeze_support
 4
 5 def main():
 6     tt = numpy.arange(5, 120, 0.01)
 7     Fe_powder = xu.simpack.Powder(xu.materials.Fe, 1,
 8                                   crystallite_size_gauss=100e-9)
 9     Co_powder = xu.simpack.Powder(xu.materials.Co, 5,  # 5 times more Co
10                                   crystallite_size_gauss=200e-9)
11     pm = xu.simpack.PowderModel(Fe_powder, Co_powder, I0=100)
12     inte = pm.simulate(tt)
13     pm.close()
14
15 if __name__ == '__main__':
16     freeze_support()  # only required on MS Windows
17     main()
```

xrayutilities

xrayutilities package

Subpackages

xrayutilities.analysis package

Submodules

xrayutilities.analysis.line_cuts module

`xrayutilities.analysis.line_cuts.`**`get_arbitrary_line`**(qpos, intensity, point, vec, npoints, intrange)

extracts a line scan from reciprocal space map data along an arbitrary line defined by the point 'point' and propergation vector 'vec'. Integration of the data is performed in a cylindrical volume along the line. This function works for 2D and 3D input data!

Parameters:

qpos : *list of array-like objects*

arrays of x, y (list with two components) or x, y, z (list with three components) momentum transfers

intensity : *array-like*

2D or 3D array of reciprocal space intensity with shape equal to the qpos entries

point : *tuple, list or array-like*

point on the extraction line (2 or 3 coordinates)

vec : *tuple, list or array-like*

propergation vector of the extraction line (2 or 3 coordinates)

npoints : *int*

number of points in the output data

intrange : *float*

radius of the cylindrical integration volume around the extraction line

Returns:

qpos, qint : *ndarray*

line scan coordinates and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> qcut, qint, mask = get_arbitrary_line([qx, qy, qz], inten,     ... (1.1, 2.2, 0.0),
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan`**(qpos, intensity, cutpos, npoints, intrange, **kwargs)

extracts an omega scan from reciprocal space map data with integration along either the 2theta, or radial (omega-2theta) direction. The coplanar diffraction geometry with qy and qz as diffraction plane is assumed. This is consistent with the coplanar geometry implemented in the HXRD-experiment class.
This function works for 2D and 3D input data in the same way!

Parameters: **qpos :** *list of array-like objects*

arrays of y, z (list with two components) or x, y, z (list with three components) momentum transfers

intensity : *array-like*

2D or 3D array of reciprocal space intensity with shape equal to the qpos entries

cutpos : *tuple or list*

y/z-position or x/y/z-position at which the line scan should be extracted. this must be have two entries for 2D data (z-position) and a three for 3D data

npoints : *int*

number of points in the output data

intrange : *float*

integration range in along *intdir* in degree. data will be integrated from *-intrange .. +intrange*

intdir : *{'2theta', 'radial'}, optional*

integration direction: '2theta': scattering angle (default), or 'radial': omega-2theta direction.

wl : *float or str, optional*

wavelength used to determine angular integration positions

Note:

Although applicable for any set of data, the extraction only makes sense when the data are aligned into the y/z-plane.

Returns: **om, omint :** *ndarray*

omega scan coordinates and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> omcut, omcut_int, mask = get_omega_scan([qy, qz], inten, [2.0, 5.0],     ... 250, intra
```

xrayutilities.analysis.line_cuts.**get_qx_scan**(qpos, intensity, cutpos, npoints, intrange, **kwargs)

extracts a qx scan from 3D reciprocal space map data with integration along either, the perpendicular plane in q-space, omega (sample rocking angle) or 2theta direction. For the integration in angular space (omega, or 2theta) the coplanar diffraction geometry with qy and qz as diffraction plane is assumed. This is consistent with the coplanar geometry implemented in the HXRD-experiment class.

Parameters: **qpos :** *list of array-like objects*

arrays of x, y, z (list with three components) momentum transfers

intensity : *array-like*

3D array of reciprocal space intensity with shape equal to the qpos entries

cutpos : *tuple/list*

y/z-position at which the line scan should be extracted. this must be and a tuple/list with the qy, qz cut position

npoints : *int*

number of points in the output data

intrange : *float*

integration range in along *intdir*, either in 1/AA (*q*) or degree ('omega', or '2theta'). data will be integrated from *-intrange .. +intrange*

intdir : *{'q', 'omega', '2theta'}, optional*

integration direction: 'q': perpendicular Q-plane (default), 'omega': sample rocking angle, or '2theta': scattering angle.

wl : *float or str, optional*

wavelength used to determine angular integration positions

Note:

The angular integration directions although applicable for any set of data only makes sense when the data are aligned into the y/z-plane.

Returns: **qx, qxint :** *ndarray*

qx scan coordinates and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> qxcut, qxcut_int, mask = get_qx_scan([qx, qy, qz], inten, [0, 2.0], \
... 250, intrange=0.01)
```

`xrayutilities.analysis.line_cuts.`**`get_qy_scan`**`(qpos, intensity, cutpos, npoints, intrange, **kwargs)`

extracts a qy scan from reciprocal space map data with integration along either, the perpendicular plane in q-space, omega (sample rocking angle) or 2theta direction. For the integration in angular space (omega, or 2theta) the coplanar diffraction geometry with qy and qz as diffraction plane is assumed. This is consistent with the coplanar geometry implemented in the HXRD-experiment class.
This function works for 2D and 3D input data in the same way!

Parameters: **qpos :** *list of array-like objects*

arrays of y, z (list with two components) or x, y, z (list with three components) momentum transfers

intensity : *array-like*

2D or 3D array of reciprocal space intensity with shape equal to the qpos entries

cutpos : *float or tuple/list*

x/z-position at which the line scan should be extracted. this must be a float for 2D data (z-position) and a tuple with two values for 3D data

npoints : *int*

number of points in the output data

intrange : *float*

integration range in along *intdir*, either in 1/AA (*q*) or degree ('omega', or '2theta'). data will be integrated from *-intrange .. +intrange*

intdir : *{'q', 'omega', '2theta'}, optional*

integration direction: 'q': perpendicular Q-plane (default), 'omega': sample rocking angle, or '2theta': scattering angle.

wl : *float or str, optional*

wavelength used to determine angular integration positions

Note:

For 3D data the angular integration directions although applicable for any set of data only makes sense when the data are aligned into the y/z-plane.

Returns: **qy, qyint :** *ndarray*

qy scan coordinates and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> qycut, qycut_int, mask = get_qy_scan([qy, qz], inten, 5.0, 250, \
... intrange=0.02, intdir='2theta')
```

xrayutilities.analysis.line_cuts.**get_qz_scan**(qpos, intensity, cutpos, npoints, intrange, **kwargs)

extracts a qz scan from reciprocal space map data with integration along either, the perpendicular plane in q-space, omega (sample rocking angle) or 2theta direction. For the integration in angular space (omega, or 2theta) the coplanar diffraction geometry with qy and qz as diffraction plane is assumed. This is consistent with the coplanar geometry implemented in the HXRD-experiment class.
This function works for 2D and 3D input data in the same way!

Parameters: **qpos :** *list of array-like objects*

arrays of y, z (list with two components) or x, y, z (list with three components) momentum transfers

intensity : *array-like*

2D or 3D array of reciprocal space intensity with shape equal to the qpos entries

cutpos : *float or tuple/list*

x/y-position at which the line scan should be extracted. this must be a float for 2D data and a tuple with two values for 3D data

npoints : *int*

number of points in the output data

intrange : *float*

integration range in along *intdir*, either in 1/AA (*q*) or degree ('omega', or '2theta'). data will be integrated from *-intrange/2 .. +intrange/2*

intdir : *{'q', 'omega', '2theta'}, optional*

integration direction: 'q': perpendicular Q-plane (default), 'omega': sample rocking angle, or '2theta': scattering angle.

wl : *float or str, optional*

wavelength used to determine angular integration positions

Note:

For 3D data the angular integration directions although applicable for any set of data only makes sense when the data are aligned into the y/z-plane.

Returns: **qz, qzint :** *ndarray*

qz scan coordinates and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> qzcut, qzcut_int, mask = get_qz_scan([qy, qz], inten, 3.0, 200,\
... intrange=0.3)
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan`**(`qpos`, `intensity`, `cutpos`, `npoints`, `intrange`, `**kwargs`)

extracts a radial scan from reciprocal space map data with integration along either the omega or 2theta direction. The coplanar diffraction geometry with qy and qz as diffraction plane is assumed. This is consistent with the coplanar geometry implemented in the HXRD-experiment class.
This function works for 2D and 3D input data in the same way!

Parameters: **qpos :** *list of array-like objects*

arrays of y, z (list with two components) or x, y, z (list with three components) momentum transfers

intensity : *array-like*

2D or 3D array of reciprocal space intensity with shape equal to the qpos entries

cutpos : *tuple or list*

y/z-position or x/y/z-position at which the line scan should be extracted. this must be have two entries for 2D data (z-position) and a three for 3D data

npoints : *int*

number of points in the output data

intrange : *float*

integration range in along *intdir* in degree. data will be integrated from *-intrange .. +intrange*

intdir : *{'omega', '2theta'}, optional*

integration direction: 'omega': sample rocking angle (default), '2theta': scattering angle

wl : *float or str, optional*

wavelength used to determine angular integration positions

Note:

Although applicable for any set of data, the extraction only makes sense when the data are aligned into the y/z-plane.

Returns: **tt, omttint :** *ndarray*

omega-2theta scan coordinates (2theta values) and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> ttcut, omtt_int, mask = get_radial_scan([qy, qz], inten, [2.0, 5.0],     ... 250, intra
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan`**`(qpos, intensity, cutpos, npoints, intrange, **kwargs)`

extracts a 2theta scan from reciprocal space map data with integration along either the omega or radial direction. The coplanar diffraction geometry with qy and qz as diffraction plane is assumed. This is consistent with the coplanar geometry implemented in the HXRD-experiment class.
This function works for 2D and 3D input data in the same way!

Parameters: **qpos :** *list of array-like objects*

arrays of y, z (list with two components) or x, y, z (list with three components) momentum transfers

intensity : *array-like*

2D or 3D array of reciprocal space intensity with shape equal to the qpos entries

cutpos : *tuple or list*

y/z-position or x/y/z-position at which the line scan should be extracted. this must be have two entries for 2D data (z-position) and a three for 3D data

npoints : *int*

number of points in the output data

intrange : *float*

integration range in along *intdir* in degree. data will be integrated from *-intrange .. +intrange*

intdir : *{'omega', 'radial'}, optional*

integration direction: 'omega': sample rocking angle (default), 'radial': omega-2theta direction

wl : *float or str, optional*

wavelength used to determine angular integration positions

Note:

Although applicable for any set of data, the extraction only makes sense when the data are aligned into the y/z-plane.

Returns: **tt, ttint :** *ndarray*

2theta scan coordinates and intensities

used_mask : *ndarray*

mask of used data, shape is the same as the input intensity: True for points which contributed, False for all others

Examples

```
>>> ttcut, tt_int, mask = get_ttheta_scan([qy, qz], inten, [2.0, 5.0],     ... 250, intrang
```

xrayutilities.analysis.misc module

miscellaneous functions helpful in the analysis and experiment

`xrayutilities.analysis.misc.`**`coplanar_intensity`**(`mat`, `exp`, `hkl`, `thickness`, `thMono`, `sample_width=10`, `beam_width=1`)

Calculates the expected intensity of a Bragg peak from an epitaxial thin film measured in coplanar geometry (integration over omega and 2theta in angular space!)

Parameters: **mat :** *Crystal*
Crystal instance for structure factor calculation
exp : *Experiment*
Experimental(HXRD) class for the angle calculation
hkl : *list, tuple or array-like*
Miller indices of the peak to calculate
thickness : *float*
film thickness in nm
thMono : *float*
Bragg angle of the monochromator (deg)
sample_width : *float, optional*
width of the sample along the beam
beam_width : *float, optional*
width of the beam in the same units as the sample size

Returns: **float**
intensity of the peak

`xrayutilities.analysis.misc.`**`getangles`** (`peak`, `sur`, `inp`)
calculates the chi and phi angles for a given peak

Parameters: **peak :** *list or array-like*
hkl for the peak of interest
sur : *list or array-like*
hkl of the surface
inp : *list or array-like*
inplane reference peak or direction

Returns: **list**
[chi, phi] for the given peak on surface sur with inplane direction inp as reference

Examples

To get the angles for the -224 peak on a 111 surface type

```
>>> [chi, phi] = getangles([-2, 2, 4], [1, 1, 1], [2, 2, 4])
```

`xrayutilities.analysis.misc.`**`getunitvector`** (`chi`, `phi`, `ndir`=(0, 0, 1), `idir`=(1, 0, 0))
return unit vector determined by spherical angles and definition of the polar axis and inplane reference direction (phi=0)

Parameters: **chi, phi :** *float*
spherical angles (polar and azimuthal) in degree
ndir : *tuple, list or array-like*
polar/z-axis (determines chi=0)
idir : *tuple, list or array-like*
azimuthal axis (determines phi=0)

xrayutilities.analysis.sample_align module

functions to help with experimental alignment during experiments, especially for experiments with linear and area detectors

`xrayutilities.analysis.sample_align.`**`area_detector_calib`** (`angle1`, `angle2`, `ccdimages`, `detaxis`, `r_i`, `plot`=True, `cut_off`=0.7, `start`=(None, None, 1, 0, 0, 0, 0), `fix`=(False, False, True, False, False, False, False), `fig`=None, `wl`=None, `plotlog`=False, `nwindow`=50, `debug`=False)
function to calibrate the detector parameters of an area detector it determines the detector tilt possible rotations and offsets in the detector arm angles

Parameters: **angle1 :** *array-like*

outer detector arm angle

angle2 : *array-like*

inner detector arm angle

ccdimages : *array-like*

images of the ccd taken at the angles given above

detaxis : *list of str*

detector arm rotation axis; default: [‘z+’, ‘y-‘]

r_i : *str*

primary beam direction [xyz][+-]; default ‘x+’

plot : *bool, optional*

flag to determine if results and intermediate results should be plotted; default: True

cut_off : *float, optional*

cut off intensity to decide if image is used for the determination or not; default: 0.7 = 70%

start : *tuple, optional*

sequence of start values of the fit for parameters, which can not be estimated automatically or might want to be fixed. These are: pwidth1, pwidth2, distance, tiltazimuth, tilt, detector_rotation, outerangle_offset. By default (None, None, 1, 0, 0, 0, 0) is used.

fix : *tuple of bool*

fix parameters of start (default: (False, False, True, False, False, False, False)) It is strongly recommended to either fix the distance or the pwidth1, 2 values.

fig : *Figure, optional*

matplotlib figure used for plotting the error default: None (creates own figure)

wl : *float or str*

wavelength of the experiment in angstrom (default: config.WAVELENGTH) value does not really matter here but does affect the scaling of the error

plotlog : *bool*

flag to specify if the created error plot should be on log-scale

nwindow : *int*

window size for determination of the center of mass position after the center of mass of every full image is determined, the center of mass is determined again using a window of size nwindow in order to reduce the effect of hot pixels.

debug : *bool*

flag to specify that you want to see verbose output and saving of images to show if the CEN determination works

`xrayutilities.analysis.sample_align.`**`area_detector_calib_hkl`**(`sampleang`, `angle1`, `angle2`, `ccdimages`, `hkls`, `experiment`, `material`, `detaxis`, `r_i`, `plot`=True, `cut_off`=0.7, `start`=(None, None, 1, 0, 0, 0, 0, 0, 0, 0, 'config'), `fix`=(False, False, True, False, False, False, False, False, False, False), `fig`=None, `plotlog`=False, `nwindow`=50, `debug`=False)

function to calibrate the detector parameters of an area detector it determines the detector tilt possible rotations and offsets in the detector arm angles

in this variant not only scans through the primary beam but also scans at a set of symmetric reflections can be used for the detector parameter determination. for this not only the detector parameters but in addition the sample orientation and wavelength need to be fit. Both images from the primary beam hkl = (0, 0, 0) and from a symmetric reflection hkl = (h, k, l) need to be given for a successful run.

Parameters: **sampleang :** *array-like*

sample rocking angle (needed to align the reflections (same rotation direction as inner detector rotation)) other sample angle are not allowed to be changed during the scans

angle1 : *array-like*

outer detector arm angle

angle2 : *array-like*

inner detector arm angle

ccdimages : *array-like*

images of the ccd taken at the angles given above

hkls : *list or array-like*

hkl values for every image

experiment : *Experiment*

Experiment class object needed to get the UB matrix for the hkl peak treatment

material : *Crystal*

material used as reference crystal

detaxis : *list of str*

detector arm rotation axis; default: ['z+', 'y-']

r_i : *str*

primary beam direction [xyz][+-]; default 'x+'

plot : *bool, optional*

flag to determine if results and intermediate results should be plotted; default: True

cut_off : *float, optional*

cut off intensity to decide if image is used for the determination or not; default: 0.7 = 70%

start : *tuple, optional*

sequence of start values of the fit for parameters, which can not be estimated automatically or might want to be fixed. These are: pwidth1, pwidth2, distance, tiltazimuth, tilt, detector_rotation, outerangle_offset, sampletilt, sampletiltazimuth, wavelength. By default (None, None, 1, 0, 0, 0, 0, 0, 0, 'config').

fix : *tuple of bool*

fix parameters of start (default: (False, False, True, False, False, False, False, False, False, False)) It is strongly recommended to either fix the distance or the pwidth1, 2 values.

fig : *Figure, optional*

matplotlib figure used for plotting the error default: None (creates own figure)

plotlog : *bool*

flag to specify if the created error plot should be on log-scale

nwindow : *int*

window size for determination of the center of mass position after the center of mass of every full image is determined, the center of mass is determined again using a window of size nwindow in order to reduce the effect of hot pixels.

debug : *bool*

flag to specify that you want to see verbose output and saving of images to show if the CEN determination works

`xrayutilities.analysis.sample_align.`**`fit_bragg_peak`** (`om`, `tt`, `psd`, `omalign`, `ttalign`, `exphxrd`, `frange`=(0.03, 0.03), `peaktype`='Gauss', `plot`=True)

helper function to determine the Bragg peak position in a reciprocal space map used to obtain the position needed for correction of the data. the determination is done by fitting a two dimensional Gaussian (xrayutilities.math.Gauss2d) or Lorentzian (xrayutilities.math.Lorentz2d)

PLEASE ALWAYS CHECK THE RESULT CAREFULLY!

Parameters: **om, tt :** *array-like*

angular coordinates of the measurement either with size of psd or of psd.shape[0]

psd : *array-like*

intensity values needed for fitting

omalign : *float*

aligned omega value, used as first guess in the fit

ttalign : *float*

aligned two theta values used as first guess in the fit these values are also used to set the range for the fit: the peak should be within +/-frangeAA^{-1} of those values

exphxrd : *Experiment*

experiment class used for the conversion between angular and reciprocal space.

frange : *tuple of float, optional*

data range used for the fit in both directions (see above for details default:(0.03, 0.03) unit: AA^{-1})

peaktype : *{'Gauss', 'Lorentz'}*

peak type to fit

plot : *bool, optional*

if True (default) function will plot the result of the fit in comparison with the measurement.

Returns: **omfit, ttfit :** *float*

fitted angular values

params : *list*

fit parameters (of the Gaussian/Lorentzian)

covariance : *ndarray*

covariance matrix of the fit parameters

`xrayutilities.analysis.sample_align.`**`linear_detector_calib`** (`angle`, `mca_spectra`, `**keyargs`)

function to calibrate the detector distance/channel per degrees for a straight linear detector mounted on a detector arm

Parameters: **angle :** *array-like*

array of angles in degree of measured detector spectra

mca_spectra : *array-like*

corresponding detector spectra (shape: (len(angle), Nchannels)

r_i : *str, optional*

primary beam direction as vector [xyz][+-]; default: 'y+'

detaxis : *str, optional*

detector arm rotation axis [xyz][+-]; default: 'x+'

Returns: **pixelwidth :** *float*

width of the pixel at one meter distance, pixelwidth is negative in case the hit channel number decreases upon an increase of the detector angle

center_channel : *float*

central channel of the detector

detector_tilt : *float, optional*

if usetilt=True the fitted tilt of the detector is also returned

Note

L/pixelwidth*pi/180 ~= channel/degree, with the sample detector distance L

The function also prints out how a linear detector can be initialized using

the results obtained from this calibration. Carefully check the results

Other Parameters:

plot : *bool*

flag to specify if a visualization of the fit should be done

usetilt : *bool*

whether to use model considering a detector tilt, i.e. deviation angle of the pixel direction from orthogonal to the primary beam (default: True)

Seealso

psd_chdeg

low level function with more configurable options

`xrayutilities.analysis.sample_align.`**`miscut_calc`**(`phi`, `aomega`, `zeros`=None, `omega0`=None, `plot`=True)

function to calculate the miscut direction and miscut angle of a sample by fitting a sinusoidal function to the variation of the aligned omega values of more than two reflections. The function can also be used to fit reflectivity alignment values in various azimuths.

Parameters:

phi : *list, tuple or array-like*

azimuths in which the reflection was aligned (deg)

aomega : *list, tuple or array-like*

aligned omega values (deg)

zeros : *list, tuple or array-like, optional*

angles at which surface is parallel to the beam (deg). For the analysis the angles (aomega - zeros) are used.

omega0 : *float, optional*

if specified the nominal value of the reflection is not included as fit parameter, but is fixed to the specified value. This value is MANDATORY if ONLY TWO AZIMUTHs are given.

plot : *bool, optional*

flag to specify if a visualization of the fit is wanted. default: True

Returns:

omega0 : *float*

the omega value of the reflection should be close to the nominal one

phi0 : *float*

the azimuth in which the primary beam looks upstairs

miscut : *float*

amplitude of the sinusoidal variation == miscut angle

`xrayutilities.analysis.sample_align.`**`psd_chdeg`**(`angles`, `channels`, `stdev`=None, `usetilt`=True, `plot`=True, `datap`='xk', `modelline`='--r', `modeltilt`='-b', `fignum`=None, `mlabel`='fit', `mtiltlabel`='fit w/tilt', `dlabel`='data', `figtitle`=True)

function to determine the channels per degree using a linear fit of the function nchannel = center_ch+chdeg*tan(angles) or the equivalent including a detector tilt

Parameters:

angles : *array-like*

detector angles for which the position of the beam was measured

channels : *array-like*

detector channels where the beam was found

stdev : *array-like, optional*

standard deviation of the beam position

plot : *bool, optional*

flag to specify if a visualization of the fit should be done

usetilt : *bool, optional*

whether to use model considering a detector tilt, i.e. deviation angle of the pixel direction from orthogonal to the primary beam (default: True)

Returns:

pixelwidth : *float*

the width of one detector channel @ 1m distance, which is negative in case the hit channel number decreases upon an increase of the detector angle.

centerch : *float*

center channel of the detector

tilt : *float*

tilt of the detector from perpendicular to the beam (will be zero in case of usetilt=False)

Note

L/pixelwidth*pi/180 = channel/degree for large detector distance with the sample detector disctance L

Other Parameters:

datap : *str, optional*

plot format of data points

modelline : *str, optional*

plot format of modelline

modeltilt : *str, optional*

plot format of modeltilt

fignum : *int or str, optional*

figure number to use for the plot

mlabel : *str*

label of the model w/o tilt to be used in the plot

mtiltlabel : *str*

label of the model with tilt to be used in the plot

dlabel : *str*

label of the data line to be used in the plot

figtitle : *bool*

flag to tell if the figure title should show the fit parameters

`xrayutilities.analysis.sample_align.`**`psd_refl_align`**(`primarybeam`, `angles`, `channels`, `plot`=True)

function which calculates the angle at which the sample is parallel to the beam from various angles and detector channels from the reflected beam. The function can be used during the half beam alignment with a linear detector.

Parameters: **primarybeam :** *int*
primary beam channel number
angles : *list or array-like*
incidence angles
channels : *list or array-like*
corresponding detector channels
plot : *bool, optional*
flag to specify if a visualization of the fit is wanted default : True

Returns: **float**
angle at which the sample is parallel to the beam

Examples

```
>>> zeroangle = psd_refl_align(500, [0, 0.1, 0.2, 0.3],
... [550, 600, 640, 700])
XU.analysis.psd_refl_align: sample is parallel to beam at goniometer angle -0.0986 (R^2=0.9
```

Module contents

xrayutilities.analysis is a package for assisting with the analysis of x-ray diffraction data, mainly reciprocal space maps

Routines for obtaining line cuts from gridded reciprocal space maps are offered, with the ability to integrate the intensity perpendicular to the line cut direction.

xrayutilities.io package

Submodules

xrayutilities.io.cbf module

class xrayutilities.io.cbf.**CBFDirectory**(datapath, ext='cbf', **keyargs)
Bases: **FileDirectory**
Parses a directory for CBF files, which can be stored to a HDF5 file for further usage

__init__ (datapath**,** ext='cbf'**,** **keyargs**)**

Parameters: **datapath :** *str*
directory of the CBF files
ext : *str, optional*
extension of the ccd files in the datapath (default: “cbf”)
keyargs : *dict, optional*
further keyword arguments are passed to CBFFile

class xrayutilities.io.cbf.**CBFFile**(fname, nxkey='X-Binary-Size-Fastest-Dimension', nykey='X-Binary-Size-Second-Dimension', dtkey='DataType', path=None)
Bases: **object**

ReadData ()
Read the CCD data into the .data object this function is called by the initialization

Save2HDF5 (h5f**,** group='/'**,** comp=True**)**
Saves the data stored in the EDF file in a HDF5 file as a HDF5 array. By default the data is stored in the root group of the HDF5 file - this can be changed by passing the name of a target group or a path to the target group via the “group” keyword argument.

Parameters: **h5f :** *file-handle or str*

a HDF5 file object or name

group : *str, optional*

group where to store the data (default to the root of the file)

comp : *bool, optional*

activate compression - true by default

`__init__` **(**`fname`**,** `nxkey='X-Binary-Size-Fastest-Dimension'`**,** `nykey='X-Binary-Size-Second-Dimension'`**,** `dtkey='DataType'`**,** `path=None`**)**

CBF detector image parser

Parameters: **fname :** *str*

name of the CBF file of type .cbf or .cbf.gz

nxkey : *str, optional*

name of the header key that holds the number of points in x-direction

nykey : *str, optional*

name of the header key that holds the number of points in y-direction

dtkey : *str, optional*

name of the header key that holds the datatype for the binary data

path : *str, optional*

path to the CBF file

xrayutilities.io.desy_tty08 module

class for reading data + header information from tty08 data files

tty08 is a system used at beamline P08 at Hasylab Hamburg and creates simple ASCII files to save the data. Information is easily read from the multicolumn data file. the functions below enable also to parse the information of the header

`xrayutilities.io.desy_tty08.`**`getty08_scan`** (`scanname`, `scannumbers`, `*args`, `**keyargs`)

function to obtain the angular cooridinates as well as intensity values saved in TTY08 datafiles. Especially usefull for reciprocal space map measurements, and to combine date from several scans
further more it is possible to obtain even more positions from the data file if more than two string arguments with its names are given

Parameters: **scanname :** *str*

name of the scans, for multiple scans this needs to be a template string

scannumbers : *int, tuple or list*

number of the scans of the reciprocal space map

args : *str, optional*

names of the motors. to read reciprocal space maps measured in coplanar diffraction give:

- *omname*: the name of the omega motor (or its equivalent)
- *ttname*: the name of the two theta motor (or its equivalent)

keyargs : *dict, optional*

keyword arguments are passed on to tty08File

Returns: **[ang1, ang2, …] :** *list, optional*

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D), omitted if no *args* are given

MAP : *ndarray*

All the data values as stored in the data file (includes the intensities e.g. MAP[‘MCA’]).

Examples

```
>>> [om, tt], MAP = xu.io.gettty08_scan('text%05d.dat', 36, 'omega',
... 'gamma')
```

class xrayutilities.io.desy_tty08.**tty08File**(filename, path=None, mcadir=None)

Bases: **object**

Represents a tty08 data file. The file is read during the Constructor call. This class should work for data stored at beamline P08 using the tty08 acquisition system.

Parameters: **filename :** *str*

tty08-filename

mcadir : *str, optional*

directory name of MCA files

Read ()

Read the data from the file

ReadMCA ()

__init__ (filename**,** path=None**,** mcadir=None**)**

xrayutilities.io.edf module

class xrayutilities.io.edf.**EDFDirectory**(datapath, ext='edf', **keyargs)

Bases: **FileDirectory**

Parses a directory for EDF files, which can be stored to a HDF5 file for further usage

__init__ (datapath**,** ext='edf'**,** **keyargs**)**

Parameters: **datapath :** *str*

directory of the EDF file

ext : *str, optional*

extension of the ccd files in the datapath (default: “edf”)

keyargs : *dict, optional*

further keyword arguments are passed to EDFFile

class xrayutilities.io.edf.**EDFFile**(fname, nxkey='Dim_1', nykey='Dim_2', dtkey='DataType', path='', header=True, keep_open=False)

Bases: **object**

Parse ()

Parse file to find the number of entries and read the respective header information

ReadData (nimg=0**)**

Read the CCD data of the specified image and return the data this function is called automatically when the ‘data’ property is accessed, but can also be called manually when only a certain image from the file is needed.

Parameters: **nimg :** *int, optional*

number of the image which should be read (starts with 0)

Save2HDF5 (h5f**,** group='/'**,** comp=True**)**

Saves the data stored in the EDF file in a HDF5 file as a HDF5 array. By default the data is stored in the root group of the HDF5 file - this can be changed by passing the name of a target group or a path to the target group via the “group” keyword argument.

Parameters: **h5f :** *file-handle or str*
a HDF5 file object or name
group : *str, optional*
group where to store the data (default to the root of the file)
comp : *bool, optional*
activate compression - true by default

__init__ **(**fname**,** nxkey='Dim_1'**,** nykey='Dim_2'**,** dtkey='DataType'**,** path=''**,** header=True**,** keep_open=False**)**

Parameters: **fname :** *str*
name of the EDF file of type .edf or .edf.gz
nxkey : *str, optional*
name of the header key that holds the number of points in x-direction
nykey : *str, optional*
name of the header key that holds the number of points in y-direction
dtkey : *str, optional*
name of the header key that holds the datatype for the binary data
path : *str, optional*
path to the EDF file
header : *bool, optional*
has header (default true)
keep_open : *bool, optional*
if True the file handle is kept open between multiple calls which can cause significant speed-ups

property **data**

xrayutilities.io.fastscan module

modules to help with the analysis of FastScan data acquired at the ESRF. FastScan data are X-ray data (various detectors possible) acquired during scanning the sample in real space with a Piezo Scanner. The same functions might be used to analze traditional SPEC mesh scans.

The module provides three core classes:

- FastScan
- FastScanCCD
- FastScanSeries

where the first two are able to parse single mesh/FastScans when one is interested in data of a single channel detector or are detector and the last one is able to parse full series of such mesh scans with either type of detector

see examples/xrayutilities_kmap_ESRF.py for an example script

class xrayutilities.io.fastscan.**FastScan** (filename, scannr, xmotor='adcX', ymotor='adcY', path='')
Bases: **object**
class to help parsing and treating fast scan data. FastScan is the aquisition of X-ray data while scanning the sample with piezo stages in real space. It's is available at several beamlines at the ESRF synchrotron light-source.

__init__ **(**filename**,** scannr**,** xmotor='adcX'**,** ymotor='adcY'**,** path=''**)**
Constructor routine for the FastScan object. It initializes the object and parses the spec-scan for the needed data which are saved in properties of the FastScan object.

Parameters: **filename :** *str*
file name of the fast scan spec file
scannr : *int*
scannr of the to be parsed fast scan
xmotor : *str, optional*
motor name of the x-motor (default: ‘adcX’ (ID01))
ymotor : *str, optional*
motor name of the y-motor (default: ‘adcY’ (ID01))
path : *str, optional*
optional path of the FastScan spec file

grid2D (nx**,** ny**,** ****kwargs**)**
function to grid the counter data and return the gridded X, Y and Intensity values.

Parameters: **nx, ny :** *int*
number of bins in x, and y direction
counter : *str, optional*
name of the counter to use for gridding (default: ‘mpx4int’ (ID01))
gridrange : *tuple, optional*
range for the gridder: format: ((xmin, xmax), (ymin, ymax))
Returns: **Gridder2D**
Gridder2D object with X, Y, data on regular x, y-grid

motorposition (motorname**)**
read the position of motor with name given by motorname from the data file. In case the motor is included in the data columns the returned object is an array with all the values from the file (although retrace clean is respected if already performed). In the case the motor is not moved during the scan only one value is returned.

Parameters: **motorname :** *str*
name of the motor for which the position is wanted
Returns: **ndarray**
motor position(s) of motor with name motorname during the scan

parse ()
parse the specfile for the scan number specified in the constructor and store the needed informations in the object properties

retrace_clean ()
function to clean the data of the scan from retrace artifacts created by the zig-zag scanning motion of the piezo actuators the function cleans the xvalues, yvalues and data attribute of the FastScan object.

class xrayutilities.io.fastscan.**FastScanCCD** (*args, **kwargs)
Bases: **FastScan**
class to help parsing and treating fast scan data including CCD frames. FastScan is the aquisition of X-ray data while scanning the sample with piezo stages in real space. It’s is available at several beamlines at the ESRF synchrotron light-source. During such fast scan at every grid point CCD frames are recorded and need to be analyzed

__init__ (*args**,** **kwargs**)**

Parameters: **imagefiletype :** *str, optional*
image file extension, either ‘edf’ / ‘edf.gz’ (default) or ‘h5’
other parameters are passed on to FastScanCCD

getCCD (ccdnr**,** roi=None**,** datadir=None**,** keepdir=0**,** replacedir=None**,** nav=(1, 1)**,** filterfunc=None**)**

function to read the ccd files and return the raw X, Y and DATA values. DATA represents a 3D object with first dimension representing the data point index and the remaining two dimensions representing detector channels

Parameters:

ccdnr : *array-like or str*

array with ccd file numbers of length length(FastScanCCD.data) OR a string with the data column name for the file ccd-numbers

roi : *tuple, optional*

region of interest on the 2D detector. should be a list of lower and upper bounds of detector channels for the two pixel directions (default: None)

datadir : *str, optional*

the CCD filenames are usually parsed from the SPEC file. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept using the keepdir option.

keepdir : *int, optional*

number of directories which should be taken from the SPEC file. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

nav : *tuple or list, optional*

number of detector pixel which will be averaged together (reduces the date size)

filterfunc : *callable*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

Returns:

X, Y : *ndarray*

x, y-array (1D)

DATA : *ndarray*

3-dimensional data object

`getccdFileTemplate` **(**`specscan`**,** `datadir`=None**,** `keepdir`=0**,** `replacedir`=None**)**

function to extract the CCD file template string from the comment in the SPEC-file scan-header.

Parameters:

specscan : *SpecScan*

spec-scan object from which header the CCD directory should be extracted

datadir : *str, optional*

the CCD filenames are usually parsed from the scan object. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept using the keepdir option.

keepdir : *int, optional*

number of directories which should be taken from the specscan. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

Returns:

fmtstr : *str*

format string for the CCD file name using one number to build the real file name

filenr : *int*

starting file number

gridCCD (nx**,** ny**,** ccdnr**,** roi=None**,** datadir=None**,** keepdir=0**,** replacedir=None**,** nav=(1, 1)**,** gridrange=None**,** filterfunc=None**)**

function to grid the internal data and ccd files and return the gridded X, Y and DATA values. DATA represents a 4D object with first two dimensions representing X, Y and the remaining two dimensions representing detector channels

Parameters:

nx, ny : *int*

number of bins in x, and y direction

ccdnr : *array-like or str*

array with ccd file numbers of length length(FastScanCCD.data) OR a string with the data column name for the file ccd-numbers

roi : *tuple, optional*

region of interest on the 2D detector. should be a list of lower and upper bounds of detector channels for the two pixel directions (default: None)

datadir : *str, optional*

the CCD filenames are usually parsed from the SPEC file. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept using the keepdir option.

keepdir : *int, optional*

number of directories which should be taken from the SPEC file. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

nav : *tuple or list, optional*

number of detector pixel which will be averaged together (reduces the date size)

gridrange : *tuple*

range for the gridder: format: ((xmin, xmax), (ymin, ymax))

filterfunc : *callable*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

Returns:

X, Y: ndarray

regular x, y-grid

DATA : *ndarray*

4-dimensional data object

processCCD (ccdnr**,** roi**,** datadir=None**,** keepdir=0**,** replacedir=None**,** filterfunc=None**)**

function to read a region of interest (ROI) from the ccd files and return the raw X, Y and intensity from ROI.

Parameters: **ccdnr :** *array-like or str*

array with ccd file numbers of length length(FastScanCCD.data) OR a string with the data column name for the file ccd-numbers

roi : *tuple or list*

region of interest on the 2D detector. Either a list of lower and upper bounds of detector channels for the two pixel directions as tuple or a list of mask arrays

datadir : *str, optional*

the CCD filenames are usually parsed from the SPEC file. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept using the keepdir option.

keepdir : *int, optional*

number of directories which should be taken from the SPEC file. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

filterfunc : *callable, optional*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

Returns: **X, Y, DATA :** *ndarray*

x, y-array (1D) as well as 1-dimensional data object

class `xrayutilities.io.fastscan.`**`FastScanSeries`**`(filenames, scannrs, nx, ny, *args, **kwargs)`

Bases: **`object`**

class to help parsing and treating a series of fast scan data including CCD frames. FastScan is the aquisition of X-ray data while scanning the sample with piezo stages in real space. It's is available at several beamlines at the ESRF synchrotron light-source. During such fast scan at every grid point CCD frames are recorded and need to be analyzed.

For the series of FastScans we assume that they are measured at different goniometer angles and therefore transform the data to reciprocal space.

`__init__``(filenames, scannrs, nx, ny, *args, **kwargs)`

Constructor routine for the FastScanSeries object. It initializes the object and creates a list of FastScanCCD objects. Importantly it also expects the motor names of the angles needed for reciprocal space conversion.

Parameters: **filenames :** *list or str*

file names of the fast scan spec files, in case of more than one filename supply a list of names and also a list of scan numbers for the different files in the 'scannrs' argument

scannrs : *list*

scannrs of the to be parsed fast scans. in case of one specfile this is a list of numbers (e.g. [1, 2, 3]). when multiple filenames are given supply a separate list for every file (e.g. [[1, 2, 3],[2, 4]])

nx, ny : *int*

grid-points for the real space grid

args : *str*

motor names for the Reciprocal space conversion. The order needs be as required by the `QConversion.area()` function.

xmotor : *str, optional*

motor name of the x-motor (default: 'adcX' (ID01))

ymotor : *str, optional*

motor name of the y-motor (default: 'adcY' (ID01))

ccdnr : *str, optional*

name of the ccd-number data column (default: 'imgnr' (ID01))

counter : *str, optional*

name of a defined counter (roi) in the spec file (default: 'mpx4int' (ID01))

path : *str, optional*

path of the FastScan spec file (default: '')

`align` (`deltax`, `deltay`)

Since a sample drift or shift due to rotation often occurs between different FastScans it should be corrected before combining them. Since determining such a shift is not straight-forward in general the user needs to supply the routine with the shifts in order correct the x, y-values for the different FastScans. Such a routine could for example use the integrated CCD intensities and determine the shift using a cross-convolution.

Parameters: **deltax, deltay :** *list*

list of shifts in x/y-direction for every FastScan in the data structure

`getCCDFrames` (`posx`, `posy`, `typ`='real')

function to determine the list of ccd-frame numbers for a specific real space position. The real space position must be within the data limits of the FastScanSeries otherwise an ValueError is thrown

Parameters: **posx :** *float*

real space x-position or index in x direction

posy : *float*

real space y-position or index in y direction

typ : *{'real', 'index'}, optional*

type of coordinates. specifies if the position is specified as real space coordinate or as index. (default: 'real')

Returns: **list**

`[[motorpos1, ccdnrs1], [motorpos2, ccdnrs2], ...]` where motorposN is from the N-ths FastScan in the series and ccdnrsN is the list of according CCD-frames

`get_average_RSM` (`qnx`, `qny`, `qnz`, `qconv`, `datadir`=None, `keepdir`=0, `replacedir`=None, `roi`=None, `nav`=(1, 1), `filterfunc`=None)

function to return the reciprocal space map data averaged over all x, y positions from a series of FastScan measurements. It necessary to give the QConversion-object to be used for the reciprocal space conversion. The QConversion-object is expected to have the 'area' conversion routines configured properly. This function needs to read all detector images, so be prepared to lean back for a moment!

Parameters: **qnx, qny, qnz :** *int*

number of points used for the 3D Gridder

qconv : *QConversion*

QConversion-object to be used for the conversion of the CCD-data to reciprocal space

roi : *tuple, optional*

region of interest on the 2D detector. should be a list of lower and upper bounds of detector channels for the two pixel directions (default: None)

nav : *tuple or list, optional*

number of detector pixel which will be averaged together (reduces the date size)

filterfunc : *callable, optional*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

datadir : *str, optional*

the CCD filenames are usually parsed from the SPEC file. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept/replaced using the keepdir/replacedir option.

keepdir : *int, optional*

number of directories which should be taken from the SPEC file. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

Returns: **Gridder3D**

gridded reciprocal space map

get_sxrd_for_qrange (qrange**,** qconv**,** datadir=None**,** keepdir=0**,** replacedir=None**,** roi=None**,** nav=(1, 1)**,** filterfunc=None**)**

function to return the real space data averaged over a certain q-range from a series of FastScan measurements. It necessary to give the QConversion-object to be used for the reciprocal space conversion. The QConversion-object is expected to have the ‘area’ conversion routines configured properly.

Note

This function assumes that all FastScans were performed in the same real space positions, no gridding or aligning is performed!

Parameters: **qrange :** *list or tuple*

q-limits defining a box in reciprocal space. six values are needed: [minx, maxx, miny, …, maxz]

qconv : *QConversion*

QConversion object to be used for the conversion of the CCD-data to reciprocal space

roi : *tuple, optional*

region of interest on the 2D detector. should be a list of lower and upper bounds of detector channels for the two pixel directions (default: None)

nav : *tuple or list, optional*

number of detector pixel which will be averaged together (reduces the date size)

filterfunc : *callable, optional*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

datadir : *str, optional*

the CCD filenames are usually parsed from the SPEC file. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept/replaced using the keepdir/replacedir option.

keepdir : *int, optional*

number of directories which should be taken from the SPEC file. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

Returns: **xvalues, yvalues, data :** *ndarray*

x, y, and data values

`grid2Dall`(`nx`**,** `ny`**,** `**kwargs`**)**

function to grid the counter data and return the gridded X, Y and Intensity values from all the FastScanSeries.

Parameters: **nx, ny :** *int*

number of bins in x, and y direction

counter : *str, optional*

name of the counter to use for gridding (default: ‘mpx4int’ (ID01))

gridrange : *tuple, optional*

range for the gridder: format: ((xmin, xmax), (ymin, ymax))

Returns: **Gridder2D**

object with X, Y, data on regular x, y-grid

`gridRSM`(`posx`**,** `posy`**,** `qnx`**,** `qny`**,** `qnz`**,** `qconv`**,** `roi`=None**,** `nav`=(1, 1)**,** `typ`='real'**,** `filterfunc`=None**,** `**kwargs`**)**

function to calculate the reciprocal space map at a certain x, y-position from a series of FastScan measurements it is necessary to specify the number of grid-oints for the reciprocal space map and the QConversion-object to be used for the reciprocal space conversion. The QConversion-object is expected to have the ‘area’ conversion routines configured properly.

Parameters: **posx :** *float*

real space x-position or index in x direction

posy : *float*

real space y-position or index in y direction

qnx, qny, qnz : *int*

number of points in the Qx, Qy, Qz direction of the gridded reciprocal space map

qconv : *QConversion*

QConversion-object to be used for the conversion of the CCD-data to reciprocal space

roi : *tuple, optional*

region of interest on the 2D detector. should be a list of lower and upper bounds of detector channels for the two pixel directions (default: None)

nav : *tuple or list, optional*

number of detector pixel which will be averaged together (reduces the date size)

typ : *{'real', 'index'}, optional*

type of coordinates. specifies if the position is specified as real space coordinate or as index. (default: 'real')

filterfunc : *callable, optional*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

UB : *ndarray*

sample orientation matrix

Returns: **Gridder3D**

object with gridded reciprocal space map

rawRSM (posx, posy, qconv, roi=None, nav=(1, 1), typ='real', datadir=None, keepdir=0, replacedir=None, filterfunc=None, **kwargs)

function to return the reciprocal space map data at a certain x, y-position from a series of FastScan measurements. It necessary to give the QConversion-object to be used for the reciprocal space conversion. The QConversion-object is expected to have the 'area' conversion routines configured properly.

Parameters:

posx : *float*

real space x-position or index in x direction

posy : *float*

real space y-position or index in y direction

qconv : *QConversion*

QConversion-object to be used for the conversion of the CCD-data to reciprocal space

roi : *tuple, optional*

region of interest on the 2D detector. should be a list of lower and upper bounds of detector channels for the two pixel directions (default: None)

nav : *tuple or list, optional*

number of detector pixel which will be averaged together (reduces the date size)

typ : *{'real', 'index'}, optional*

type of coordinates. specifies if the position is specified as real space coordinate or as index. (default: 'real')

filterfunc : *callable, optional*

function applied to the CCD-frames before any processing. this function should take a single argument which is the ccddata which need to be returned with the same shape! e.g. remove hot pixels, flat/darkfield correction

UB : *array-like, optional*

sample orientation matrix

datadir : *str, optional*

the CCD filenames are usually parsed from the SPEC file. With this option the directory used for the data can be overwritten. Specify the datadir as simple string. Alternatively the innermost directory structure can be automatically taken from the specfile. If this is needed specify the number of directories which should be kept using the keepdir option.

keepdir : *int, optional*

number of directories which should be taken from the SPEC file. (default: 0)

replacedir : *int, optional*

number of outer most directory names which should be replaced in the output (default = None). One can either give keepdir, or replacedir, with replace taking preference if both are given.

Returns:

qx, qy, qz : *ndarray*

reciprocal space positions of the reciprocal space map

ccddata : *ndarray*

raw data of the reciprocal space map

valuelist : *ndarray*

valuelist containing the ccdframe numbers and corresponding motor positions

`read_motors` ()

read motor values from the series of fast scans

`retrace_clean` ()

perform retrace clean for every FastScan in the series

xrayutilities.io.filedir module

class `xrayutilities.io.filedir.`**`FileDirectory`**`(datapath, ext, parser, **keyargs)`

Bases: **`object`**

Parses a directory for files, which can be stored to a HDF5 file for further usage. The file parser is given to the constructor and must provide a Save2HDF5 method.

`Save2HDF5` **(**`h5f`**,** `group=''`**,** `comp=True`**)**

Saves the data stored in the found files in the specified directory in a HDF5 file as a HDF5 arrays in a subgroup. By default the data is stored in a group given by the foldername - this can be changed by passing the name of a target group or a path to the target group via the “group” keyword argument.

Parameters:

h5f : *file-handle or str*

a HDF5 file object or name

group : *str, optional*

group where to store the data (defaults to pathname if group is empty string)

comp : *bool, optional*

activate compression - true by default

__init__ (`datapath`, `ext`, `parser`, `**keyargs`)

Parameters:

datapath : *str*

directory of the files

ext : *str*

extension of the files in the datapath

parser : *class*

Parser class for the data files.

keyargs : *dict*

further keyword arguments are passed to the constructor of the parser

xrayutilities.io.helper module

convenience functions to open files for various data file reader

these functions should be used in new parsers since they transparently allow to open gzipped and bzipped files

`xrayutilities.io.helper.`**`generate_filenames`** (`filetemplate`, `scannrs`=None)

generate a list of filenames from a template and replacement values.

Parameters:

filetemplate: str or list

template string which should contain placeholders if scannrs is not None

scannrs: iterable, optional

list of scan numbers. If None then the filetemplate will be returned.

Returns:

list of filenames. If only a single filename is returned it will still be

encapsulated in a list

Examples

```
>>> generate_filenames("filename_%d.ras", [1, 2, 3])
['filename_1.ras', 'filename_2.ras', 'filename_3.ras']

>>> generate_filenames("filename_{}.ras", [1, 2, 3])
['filename_1.ras', 'filename_2.ras', 'filename_3.ras']

>>> generate_filenames("filename_{}_{}.ras", [(11, 1), (21, 2), (31, 3)])
['filename_11_1.ras', 'filename_21_2.ras', 'filename_31_3.ras']

>>> generate_filenames("filename_%d.ras", 1)
['filename_1.ras']

>>> generate_filenames("filename.ras")
['filename.ras']
```

```
>>> generate_filenames(["filename.ras", "othername.ras"])
['filename.ras', 'othername.ras']
```

class xrayutilities.io.helper.**xu_h5open** (f, mode='r')
Bases: **object**
helper object to decide if a HDF5 file has to be opened/closed when using with a ‘with’ statement.

__init__ (f**,** mode='r'**)**

Parameters: **f :** *str*
filename or h5py.File instance
mode : *str, optional*
mode in which the file should be opened. ignored in case a file handle is passed as f

xrayutilities.io.helper.**xu_open** (filename, mode='rb')
function to open a file no matter if zipped or not. Files with extension ‘.gz’, ‘.bz2’, and ‘.xz’ are assumed to be compressed and transparently opened to read like usual files.

Parameters: **filename :** *str or bytes*
filename of the file to open or a bytes-stream with the file contents
mode : *str, optional*
mode in which the file should be opened
Returns: **file-handle**
handle of the opened file
Raises: **IOError**
If the file does not exist an IOError is raised by the open routine, which is not caught within the function

xrayutilities.io.ill_numor module

module for reading ILL data files (station D23): numor files

class xrayutilities.io.ill_numor.**numorFile** (filename, path=None)
Bases: **object**
Represents a ILL data file (numor). The file is read during the Constructor call. This class should work for created at station D23 using the mad acquisition system.

Parameters: **filename :** *str*
a string with the name of the data file

Read ()
Read the data from the file

__init__ (filename**,** path=None**)**
constructor for the data file parser

Parameters: **filename :** *str*
a string with the name of the data file
path : *str, optional*
directory of the data file

columns = *{0: ('detector', 'monitor', 'time', 'gamma', 'omega', 'psi'), 1: ('detector', 'monitor', 'time', 'gamma'), 2: ('detector', 'monitor', 'time', 'omega'), 5: ('detector', 'monitor', 'time', 'psi')}*

getline (fid**)**

ssplit (string**)**

multispace split. splits string at two or more spaces after stripping it.

`xrayutilities.io.ill_numor.`**`numor_scan`**(`scannumbers`, `*args`, `**kwargs`)

function to obtain the angular cooridinates as well as intensity values saved in numor datafiles. Especially useful for combining several scans into one data object.

Parameters: **scannumbers :** *int or str or iterable*

number of the numors, or list of numbers. This will be transformed to a string and used as a filename

args : *str, optional*

names of the motors e.g.: ‘omega’, ‘gamma’

kwargs : *dict*

keyword arguments are passed on to numorFile. e.g. ‘path’ for the files directory

Returns: **[ang1, ang2, …] :** *list*

angular positions list, omitted if no args are given

data : *ndarray*

all the data values.

Examples

```
>>> [om, gam], data = xu.io.numor_scan(414363, 'omega', 'gamma')
```

xrayutilities.io.imagereader module

class `xrayutilities.io.imagereader.`**`ImageReader`**(`nop1`, `nop2`, `hdrlen=0`, `flatfield=None`, `darkfield=None`, `dtype=<class 'numpy.int16'>`, `byte_swap=False`)

Bases: **`object`**

parse CCD frames in the form of tiffs or binary data (`*.bin`) to numpy arrays. ignore the header since it seems to contain no useful data

The routine was tested so far with

1. RoperScientific files with 4096x4096 pixels created at Hasylab Hamburg, which save an 16bit integer per point.
2. Perkin Elmer images created at Hasylab Hamburg with 2048x2048 pixels.

`__init__` **(**`nop1`**,** `nop2`**,** `hdrlen=0`**,** `flatfield=None`**,** `darkfield=None`**,** `dtype=<class 'numpy.int16'>`**,** `byte_swap=False`**)**

initialize the ImageReader reader, which includes setting the dimension of the images as well as defining the data used for flat- and darkfield correction!

Parameters: **nop1, nop2 :** *int*

number of pixels in the first and second dimension of the image

hdrlen : *int, optional*

length of the file header which should be ignored

flatfield : *str or ndarray, optional*

filename or data for flatfield correction. supported file types include (.bin/.tif (also compressed .xz or .gz) and .npy files). otherwise a 2D numpy array should be given

darkfield : *str or ndarray, optional*

filename or data for darkfield correction. same types as for flat field are supported.

dtype : *numpy.dtype, optional*

datatype of the stored values (default: numpy.int16)

byte_swap : *bool, optional*

flag which determines bytes are swapped after reading

`readImage` **(**`filename`**,** `path=None`**)**

read image file and correct for dark- and flatfield in case the necessary data are available.

returned data = ((image data)-(darkfield))/flatfield*average(flatfield)

Parameters: **filename :** *str*

filename of the image to be read. so far only single filenames are supported. The data might be compressed. supported extensions: .tif, .bin and .bin.xz

path : *str, optional*

path of the data files

class xrayutilities.io.imagereader.**PerkinElmer**(**keyargs)

Bases: **ImageReader**

parse PerkinElmer CCD frames (*.tif) to numpy arrays Ignore the header since it seems to contain no useful data The routine was tested only for files with 2048x2048 pixel images created at Hasylab Hamburg which save an 32bit float per point.

__init__(**keyargs**)**

initialize the PerkinElmer reader, which includes setting the dimension of the images as well as defining the data used for flat- and darkfield correction!

Parameters: **flatfield :** *str or ndarray, optional*

filename or data for flatfield correction. supported file types include (.bin .bin.xz and .npy files). otherwise a 2D numpy array should be given

darkfield : *str or ndarray, optional*

filename or data for darkfield correction. same types as for flat field are supported.

class xrayutilities.io.imagereader.**Pilatus100K**(**keyargs)

Bases: **ImageReader**

parse Dectris Pilatus 100k frames (*.tiff) to numpy arrays Ignore the header since it seems to contain no useful data

__init__(**keyargs**)**

initialize the Piulatus100k reader, which includes setting the dimension of the images as well as defining the data used for flat- and darkfield correction!

Parameters: **flatfield :** *str or ndarray, optional*

filename or data for flatfield correction. supported file types include (.bin .bin.xz and .npy files). otherwise a 2D numpy array should be given

darkfield : *str or ndarray, optional*

filename or data for darkfield correction. same types as for flat field are supported.

class xrayutilities.io.imagereader.**RoperCCD**(**keyargs)

Bases: **ImageReader**

parse RoperScientific CCD frames (*.bin) to numpy arrays Ignore the header since it seems to contain no useful data

The routine was tested only for files with 4096x4096 pixel images created at Hasylab Hamburg which save an 16bit integer per point.

__init__(**keyargs**)**

initialize the RoperCCD reader, which includes setting the dimension of the images as well as defining the data used for flat- and darkfield correction!

Parameters: **flatfield :** *str or ndarray, optional*

filename or data for flatfield correction. supported file types include (.bin .bin.xz and .npy files). otherwise a 2D numpy array should be given

darkfield : *str or ndarray, optional*

filename or data for darkfield correction. same types as for flat field are supported.

class xrayutilities.io.imagereader.**TIFFRead**(filename, path=None)

Bases: **ImageReader**

class to Parse a TIFF file including extraction of information from the file header in order to determine the image size and data type

The data stored in the image are available in the ‘data’ property.

__init__ (filename, path=None)
initialization of the class which will prepare the parser and parse the files content into class properties

Parameters: **filename :** *str*
file name of the TIFF-like image file

path : *str, optional*
path of the data file

xrayutilities.io.imagereader.**get_tiff** (filename, path=None)
read tiff image file and return the data

Parameters: **filename :** *str*
filename of the image to be read. so far only single filenames are supported. The data might be compressed.

path : *str, optional*
path of the data file

xrayutilities.io.panalytical_xml module

Panalytical XML (www.XRDML.com) data file parser

based on the native python xml.dom.minidom module. want to keep the number of dependancies as small as possible

class xrayutilities.io.panalytical_xml.**XRDMLFile** (fname, path='')
Bases: **object**
class to handle XRDML data files. The class is supplied with a file name and uses the XRDMLScan class to parse the xrdMeasurement in the file

__init__ (fname, path='')
initialization routine supplied with a filename the file is automatically parsed and the data are available in the “scan” object. If more <xrdMeasurement> tags are present, which should not be the case, their data is present in the “scans” object.

Parameters: **fname :** *str*
filename of the XRDML file

path : *str, optional*
path to the XRDML file

class xrayutilities.io.panalytical_xml.**XRDMLMeasurement** (measurement, namespace='')
Bases: **object**
class to handle scans in a XRDML datafile

__init__ (measurement, namespace='')
initialization routine for a XRDML measurement which parses are all scans within this measurement.

xrayutilities.io.panalytical_xml.**getxrdml_map** (filetemplate, scannrs=None, path='.', roi=None)
parses multiple XRDML file and concatenates the results for parsing the xrayutilities.io.XRDMLFile class is used. The function can be used for parsing maps measured with the PIXCel 1D detector (and in limited way also for data acquired with a point detector -> see getxrdml_scan instead).

Parameters: **filetemplate :** *str*

template string for the file names, can contain a %d or other replacement variables which are understood be **`generate_filenames()`**. also see the scannrs argument which is used to specify the replacement variables.

scannrs : *int or list, optional*

scan number(s)

path : *str, optional*

common path to the filenames

roi : *tuple, optional*

region of interest for the PIXCel detector, for other measurements this is not useful!

Returns: **om, tt, psd :** *ndarray*

motor positions and data as flattened numpy arrays

Examples

```
>>> om, tt, psd = xrayutilities.io.getxrdml_map("samplename_%d.xrdml",
... [1, 2], path="data")
```

`xrayutilities.io.panalytical_xml.`**`getxrdml_scan`**(`filetemplate`, `*motors`, `**kwargs`)

parses multiple XRDML file and concatenates the results for parsing the xrayutilities.io.XRDMLFile class is used. The function can be used for parsing arbitrary scans and will return the the motor values of the scan motor and additionally the positions of the motors given by in the `*motors` argument

Parameters: **filetemplate :** *str*

template string for the file names, can contain a %d or other replacement variables which are understood be **`generate_filenames()`**. also see the scannrs keyword argument which is used to specify the replacement variables.

motors : *str*

motor names to return: e.g.: ‘Omega’, ‘2Theta’, … one can also use abbreviations:

- ‘Omega’ = ‘om’ = ‘o’
- ‘2Theta’ = ‘tt’ = ‘t’
- ‘Chi’ = ‘c’
- ‘Phi’ = ‘p’

scannrs : *int or list, optional*

scan number(s)

path : *str, optional*

common path to the filenames

Returns: **scanmot, mot1, mot2,…, detectorint :** *ndarray*

motor positions and data as flattened numpy arrays

Examples

```
>>> scanmot, om, tt, inte = getxrdml_scan(
... "samplename_1.xrdml", 'om', 'tt', path="data")
```

xrayutilities.io.pdcif module

class `xrayutilities.io.pdcif.`**`pdCIF`**(`filename`, `datacolumn=None`)

Bases: **`object`**

the class implements a primitive parser for pdCIF-like files. It reads every entry and collects the information in the header attribute. The first loop containing one of the intensity fields is assumed to be the data the user is interested in and is transfered to the data array which is stored as numpy record array the columns can be accessed by name intensity fields:

- *_pd_meas_counts_total*
- *_pd_meas_intensity_total*
- *_pd_proc_intensity_total*
- *_pd_proc_intensity_net*
- *_pd_calc_intensity_total*
- *_pd_calc_intensity_net*

alternatively the data column name can be given as argument to the constructor

Parse ()
parser of the pdCIF file. the method reads the data from the file and fills the data and header attributes with content

__init__ (filename**,** datacolumn=None**)**
contructor of the pdCIF class

Parameters: **filename :** *str*
filename of the file to be parsed

datacolumn : *str, optional*
name of data column to identify the data loop (default =None; means that a list of default names is used)

class xrayutilities.io.pdcif.**pdESG** (filename, datacolumn=None)
Bases: **pdCIF**
class for parsing multiple pdCIF loops in one file. This includes especially `*.esg` files which are supposed to consist of multiple loops of pdCIF data with equal length.
Upon parsing the class tries to combine the data of these different scans into a single data matrix -> same shape of subscan data is assumed

Parse ()
parser of the pdCIF file. the method reads the data from the file and fills the data and header attributes with content

__init__ (filename**,** datacolumn=None**)**
contructor of the pdCIF class

Parameters: **filename :** *str*
filename of the file to be parsed

datacolumn : *str, optional*
name of data column to identify the data loop (default =None; means that a list of default names is used)

xrayutilities.io.pdcif.**remove_comments** (line, sep='#')

xrayutilities.io.rigaku_ras module

class for reading data + header information from Rigaku RAS (3-column ASCII) files

Such datafiles are generated by the Smartlab Guidance software from Rigaku.

class xrayutilities.io.rigaku_ras.**RASFile** (filename, path=None)
Bases: **object**
Represents a RAS data file. The file is read during the constructor call

Parameters: **filename :** *str*
name of the ras-file

path : *str, optional*
path to the data file

Read ()

Read the data from the file

__init__ (filename**,** path=None**)**

class xrayutilities.io.rigaku_ras.**RASScan** (filename, pos)
Bases: **object**
Represents a single Scan portion of a RAS data file. The scan is parsed during the constructor call

Parameters: **filename :** *str*
file name of the data file
pos : *int*
seek position of the 'RAS_HEADER_START' line

__init__ (filename**,** pos**)**

xrayutilities.io.rigaku_ras.**getras_scan** (scanname, scannumbers, *args, **kwargs)
function to obtain the angular cooridinates as well as intensity values saved in RAS datafiles. Especially useful for reciprocal space map measurements, and to combine date from several scans
further more it is possible to obtain even more positions from the data file if more than two string arguments with its names are given

Parameters: **scanname :** *str or list*
name of the scans, for multiple scans this can be a template string or a list of filenames. See **generate_filenames()** for details and examples.
scannumbers : *int, tuple or list or None*
List of scan numbers or generally replacement values for the template string given as scanname. Set to None if not needed.
args : *str, optional*
names of the motors. to read reciprocal space maps measured in coplanar diffraction give:

- omname: name of the omega motor (or its equivalent)
- ttname: name of the two theta motor (or its equivalent)

kwargs : *dict*
keyword arguments forwarded to RASFile function

Returns: **[ang1, ang2, ...] :** *list*
angular positions are extracted from the respective scan header, or motor positions during the scan. this is omitted if no *args* are given
rasdata : *ndarray*
the data values (includes the intensities e.g. rasdata['int']).

Examples

```
>>> [om, tt], MAP = getras_scan('text%05d.ras', 36, 'Omega',
... 'TwoTheta')
```

xrayutilities.io.rotanode_alignment module

parser for the alignment log file of the rotating anode

class xrayutilities.io.rotanode_alignment.**RA_Alignment** (filename)
Bases: **object**
class to parse the data file created by the alignment routine (tpalign) at the rotating anode spec installation
this routine does an iterative alignment procedure and saves the center of mass values were it moves after each scan. It iterates between two different peaks and iteratively aligns at each peak between two different motors (om/chi at symmetric peaks, om/phi at asymmetric peaks)

Parse ()
parser to read the alignment log and obtain the aligned values at every iteration.

__init__ (filename**)**
initialization function to initialize the objects variables and opens the file

Parameters: **filename :** *str*
filename of the alignment log file

get (key**)**

keys ()
returns a list of keys for which aligned values were parsed

plot (pname**)**
function to plot the alignment history for a given peak

Parameters: **pname :** *str*
peakname for which the alignment should be plotted

xrayutilities.io.seifert module

a set of routines to convert Seifert ASCII files to HDF5 in fact there exist two posibilities how the data is stored (depending on the used detector):

1. as a simple line scan (using the point detector)
2. as a map using the PSD

In the first case the data ist stored

class xrayutilities.io.seifert.**SeifertHeader**
Bases: **object**
helper class to represent a Seifert (NJA) scan file header

__init__ ()

class xrayutilities.io.seifert.**SeifertMultiScan** (filename, m_scan, m2, path='')
Bases: **object**
Class to parse a Seifert (NJA) multiscan file

__init__ (filename**,** m_scan**,** m2**,** path='**)**
Parse data from a multiscan Seifert file.

Parameters: **filename :** *str*
name of the NJA file
m_scan : *str*
name of the scan axis
m2 : *str*
name of the second moving motor
path : *str, optional*
path to the datafile

parse ()

class xrayutilities.io.seifert.**SeifertScan** (filename, path='')
Bases: **object**
Class to parse a single Seifert (NJA) scan file

__init__ (filename**,** path='**)**
Constructor for a SeifertScan object.

Parameters: **filename :** *str*
a string with the name of the file to read
path : *str, optional*
path to the datafile

parse ()

xrayutilities.io.seifert.**getSeifert_map** (filetemplate, scannrs=None, path='.', scantype='map', Nchannels=1280)
parses multiple Seifert *.nja files and concatenates the results. for parsing the xrayutilities.io.SeifertMultiScan class is used. The function can be used for parsing maps measured with the Meteor1D and point detector.

Parameters: **filetemplate :** *str or list*
template string for the file names, or list of filenames. See **generate_filenames()** for details.
scannrs : *int or list, optional*
scan number(s), or other values needed to generate filenames from the filetemplate.
path : *str, optional*
common path to the filenames
scantype : *{'map', 'O2T', 'tsk'}, optional*
type of datafile: can be either 'map' (reciprocal space map measured with a regular Seifert job (default)) or 'tsk' (MCA spectra measured using the TaskInterpreter)
Nchannels : *int, optional*
number of channels of the MCA (needed for 'tsk' measurements)
Returns: **om, tt, psd :** *ndarray*
positions and data as flattened numpy arrays

Examples

```
>>> om, tt, psd = getSeifert_map("samplename_%d.xrdml", [1, 2],
... path="data")
```

xrayutilities.io.seifert.**repair_key** (key)
Repair a key string in the sense that the string is changed in a way that it can be used as a valid Python identifier. For that purpose all blanks within the string will be replaced by _ and leading numbers get an preceeding _.

xrayutilities.io.spec module

a class for observing a SPEC data file

Motivation:

SPEC files can become quite large. Therefore, subsequently reading the entire file to extract a single scan is a quite cumbersome procedure. This module is a proof of concept code to write a file observer starting a reread of the file starting from a stored offset (last known scan position)

class xrayutilities.io.spec.**SPECCmdLine** (n, prompt, cmdl, out='')
Bases: **object**

__init__ (n**,** prompt**,** cmdl**,** out='**)**

class xrayutilities.io.spec.**SPECFile** (filename, path='')
Bases: **object**
This class represents a single SPEC file. The class provides methodes for updateing an already opened file which makes it particular interesting for interactive use.

Parse ()
Parses the file from the starting at last_offset and adding found scans to the scan list.

Save2HDF5 (h5f**,** comp=True**,** optattrs=None**)**
Save the entire file in an HDF5 file. For that purpose a group is set up in the root group of the file with the name of the file without extension and leading path. If the method is called after an previous update only the scans not written to the file meanwhile are saved.

Parameters: **h5f :** *file-handle or str*
a HDF5 file object or its filename

comp : *bool, optional*
activate compression - true by default

Update ()
reread the file and add newly added files. The parsing starts at the data offset of the last scan gathered during the last parsing run.

__init__ (filename**,** path=''**)**
SPECFile init routine

Parameters: **filename :** *str*
filename of the spec file

path : *str, optional*
path to the specfile

class xrayutilities.io.spec.**SPECLog** (filename, prompt, path='')
Bases: **object**
class to parse a SPEC log file to find the command history

Parse ()

__init__ (filename**,** prompt**,** path=''**)**
init routine for a class to read a SPEC log file

Parameters: **filename :** *str*
SPEC log file name

prompt : *str*
SPEC command prompt (e.g. 'PSIC' or 'SPEC')

path : *str, optional*
directory where the SPEC log can be found

class xrayutilities.io.spec.**SPECScan** (name, scannr, command, date, time, itime, colnames, hoffset, doffset, fname, imopnames, imopvalues, scan_status)
Bases: **object**
Represents a single SPEC scan. This class is usually not called by the user directly but used via the SPECFile class.

ClearData ()
Delete the data stored in a scan after it is no longer used.

ReadData ()
Set the data attribute of the scan class.

Save2HDF5 (h5f**,** group='/'**,** title=''**,** optattrs=None**,** comp=True**)**
Save a SPEC scan to an HDF5 file. The method creates a group with the name of the scan and stores the data there as a table object with name "data". By default the scan group is created under the root group of the HDF5 file. The title of the scan group is ususally the scan command. Metadata of the scan are stored as attributes to the scan group. Additional custom attributes to the scan group can be passed as a dictionary via the optattrs keyword argument.

Parameters: **h5f :** *file-handle or str*

a HDF5 file object or its filename

group : *str, optional*

name or group object of the HDF5 group where to store the data

title : *str, optional*

a string with the title for the data, defaults to the name of scan if empty

optattrs : *dict, optional*

a dictionary with optional attributes to store for the data

comp : *bool, optional*

activate compression - true by default

SetMCAParams (mca_column_format**,** mca_channels**,** mca_start**,** mca_stop**,** mca_channel_names**,** has_sardana_mca**)**

Set the parameters used to save the MCA data to the file. This method calculates the number of lines used to store the MCA data from the number of columns and the

Parameters: **mca_column_format :** *int*

number of columns used to save the data

mca_channels : *int*

number of MCA channels stored

mca_start : *int*

first channel that is stored

mca_stop : *int*

last channel that is stored

mca_channel_names : *list(str)*

names of mca channels

has_sardana_mca : *bool*

True for Sardana MCA, False for SPEC MCA

__init__ (name**,** scannr**,** command**,** date**,** time**,** itime**,** colnames**,** hoffset**,** doffset**,** fname**,** imopnames**,** imopvalues**,** scan_status**)**

Constructor for the SPECScan class.

Parameters: **name :** *str*
name of the scan
scannr : *int*
Number of the scan in the specfile
command : *str*
command used to write the scan
date : *str*
starting date of the scan
time : *str*
starting time of the scan
itime : *int*
integration time
colnames : *list*
list of names of the data columns
hoffset : *int*
file byte offset to the header of the scan
doffset : *int*
file byte offset to the data section of the scan
fname : *str*
file name of the SPEC file the scan belongs to
imopnames : *list of str*
motor names for the initial motor positions array
imopvalues : *list*
intial motor positions array
scan_status : *{'OK', 'NODATA', 'CORRUPTED', 'ABORTED'}*
scan status as string

getheader_element (key**,** firstonly=True**)**
return the value-string of the first appearance of this SPECScan's header element, or a list of all values if firstonly=False

Parameters: **specscan :** *SPECScan*

key : *str*
name of the key to return; e.g. 'UMONO' or 'D'
firstonly : *bool, optional*
flag to specify if all instances or only the first one should be returned

Returns: **valuestring :** *str*
header value (if firstonly=True)
[str1, str2, ...] : *list*
header values (if firstonly=False)

plot (*args**, ****keyargs**)**
Plot scan data to a matplotlib figure. If newfig=True a new figure instance will be created. If logy=True (default is False) the y-axis will be plotted with a logarithmic scale.

Parameters: **args :** *list*

arguments for the plot: first argument is the name of x-value column the following pairs of arguments are the y-value names and plot styles allowed are 3, 5, 7,… number of arguments

keyargs : *dict, optional*

newfig : *bool, optional*

if True a new figure instance will be created otherwise an existing one will be used

logy : *bool, optional*

if True a semilogy plot will be done

`xrayutilities.io.spec.`**`geth5_scan`**(`h5f`, `scans`, `*args`, `**kwargs`)

function to obtain the angular cooridinates as well as intensity values saved in an HDF5 file, which was created from a spec file by the Save2HDF5 method. Especially useful for reciprocal space map measurements.
further more it is possible to obtain even more positions from the data file if more than two string arguments with its names are given

Parameters: **h5f :** *file-handle or str*

file object of a HDF5 file opened using h5py or its filename

scans : *int, tuple or list*

number of the scans of the reciprocal space map

args : *str, optional*

names of the motors. to read reciprocal space maps measured in coplanar diffraction give:

- omname: name of the omega motor (or its equivalent)
- ttname: name of the two theta motor (or its equivalent)

kwargs : *dict, optional*

samplename: str, optional

string with the hdf5-group containing the scan data if ommited the first child node of h5f.root will be used

rettype: {'list', 'numpy'}, optional

how to return motor positions. by default a list of arrays is returned. when rettype == 'numpy' a record array will be returned.

Returns: **[ang1, ang2, …] :** *list*

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D), this list is omitted if no *args* are given

MAP : *ndarray*

the data values as stored in the data file (includes the intensities e.g. MAP['MCA']).

Examples

```
>>> [om, tt], MAP = geth5_scan("h5file", 36,
... 'omega', 'gamma')
```

`xrayutilities.io.spec.`**`getspec_scan`**(`specf`, `scans`, `*args`, `**kwargs`)

function to obtain the angular cooridinates as well as intensity values saved in a SPECFile. Especially useful to combine the data from multiple scans.
further more it is possible to obtain even more positions from the data file if more than two string arguments with its names are given

Parameters: **specf :** *SPECFile*

file object

scans : *int, tuple or list*

number of the scans

args : *str*

names of the motors and counters

rettype : *{'list', 'numpy'}, optional*

how to return motor positions. by default a list of arrays is returned. when rettype == 'numpy' a record array will be returned.

Returns: **[ang1, ang2, ...] :** *list*

coordinates and counters from the SPEC file

Examples

```
>>> [om, tt, cnt2] = getspec_scan(s, 36, 'omega', 'gamma',
... 'Counter2')
```

xrayutilities.io.spectra module

module to handle spectra data

class xrayutilities.io.spectra.**SPECTRAFile**(filename, mcatmp=None, mcastart=None, mcastop=None)

Bases: **object**

Represents a SPECTRA data file. The file is read during the Constructor call. This class should work for data stored at beamlines P08 and BW2 at HASYLAB.

Parameters: **filename :** *str*

a string with the name of the SPECTRA file

mcatmp : *str, optional*

template for the MCA files

mcastart, mcastop : *int, optional*

start and stop index for the MCA files, if not given, the class tries to determine the start and stop index automatically.

Read ()

Read the data from the file.

ReadMCA ()

Save2HDF5 (h5file**,** name**,** group='/'**,** mcaname='MCA'**)**

Saves the scan to an HDF5 file. The scan is saved to a seperate group of name "name". h5file is either a string for the file name or a HDF5 file object. If the mca attribute is not None mca data will be stored to an chunked array of with name mcaname.

Parameters: **h5file :** *file-handle or str*

HDF5 file object or name

name : *str*

name of the group where to store the data

group : *str, optional*

root group where to store the data

mcaname : *str, optional*

Name of the MCA in the HDF5 file

Returns: **bool or None**

The method returns None in the case of everything went fine, True otherwise.

__init__ (filename**,** mcatmp=None**,** mcastart=None**,** mcastop=None**)**

class xrayutilities.io.spectra.**SPECTRAFileComments**
Bases: **dict**
Class that describes the comments in the header of a SPECTRA file. The different comments are accessible via the comment keys.

__init__ ()

class xrayutilities.io.spectra.**SPECTRAFileData**
Bases: **object**

__init__ ()

append (col**)**

class xrayutilities.io.spectra.**SPECTRAFileDataColumn** (index, name, unit, type)
Bases: **object**

__init__ (index**,** name**,** unit**,** type**)**

class xrayutilities.io.spectra.**SPECTRAFileParameters**
Bases: **dict**

__init__ ()

xrayutilities.io.spectra.**geth5_spectra_map** (h5file, scans, *args, **kwargs)
function to obtain the omega and twotheta as well as intensity values for a reciprocal space map saved in an HDF5 file, which was created from a spectra file by the Save2HDF5 method.
further more it is possible to obtain even more positions from the data file if more than two string arguments with its names are given

Parameters:

h5f : *file-handle or str*
file object of a HDF5 file opened using h5py

scans : *int, tuple or list*
number of the scans of the reciprocal space map

args: str, optional
arbitrary number of motor names

- omname: name of the omega motor (or its equivalent)
- ttname: name of the two theta motor (or its equivalent)

kwargs : *dict, optional*

mca : *str, optional*
name of the mca data (if available) otherwise None (default: “MCA”)

samplename : *str, optional*
string with the hdf5-group containing the scan data if omitted the first child node of h5f.root will be used to determine the sample name

Returns:

[ang1, ang2, …] : *list*
angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D). one entry for every *args*-argument given to the function

MAP : *ndarray*
the data values as stored in the data file (includes the intensities e.g. MAP[‘MCA’]).

Module contents

xrayutilities.materials package

Submodules

xrayutilities.materials.atom module

module containing the Atom class which handles the database access for atomic scattering factors and the atomic mass.

class xrayutilities.materials.atom.**Atom**(name, num)
Bases: **object**

__init__(name, num)

property **color**

f(q, en='config')
function to calculate the atomic structure factor F

Parameters: **q :** *float, array-like*
momentum transfer
en : *float or str, optional*
energy for which F should be calculated, if omitted the value from the xrayutilities configuration is used
Returns: **float or array-like**
value(s) of the atomic structure factor

f0(q)

f1(en='config')

f2(en='config')

get_cache(prop, key)
check if a cached value exists to speed up repeated database requests

Returns: **bool**
True then result contains the cached otherwise False and result is None
result : *database value*

max_cache_length = *1000*

property **radius**

set_cache(prop, key, result)
set result to be cached to speed up future calls

property **weight**

xrayutilities.materials.atom.**get_key**(*args)
generate a hash key for several possible types of arguments

xrayutilities.materials.cif module

class xrayutilities.materials.cif.**CIFDataset**(fid, name, digits)
Bases: **object**

class for parsing CIF (Crystallographic Information File) files. The class aims to provide an additional way of creating material classes instead of manual entering of the information the lattice constants and unit cell structure are parsed from the CIF file

Parse (fid)
function to parse a CIF data set. The function reads the space group symmetry operations and the basic atom positions as well as the lattice constants and unit cell angles

SGLattice (use_p1=False)
create a SGLattice object with the structure from the CIF file

SymStruct ()
function to obtain the list of different atom positions in the unit cell for the different types of atoms and determine the space group number and origin choice if available. The data are obtained from the data parsed from the CIF file.

__init__ (fid, name, digits)
initialization of the CIFDataset class. This class parses one data block.

Parameters:
fid : *filehandle*
file handle set to the beginning of the data block to be parsed
name : *str*
identifier string of the dataset
digits : *int*
number of digits to check if position is unique

class xrayutilities.materials.cif.**CIFFile** (filestr, digits=4)
Bases: **object**
class for parsing CIF (Crystallographic Information File) files. The class aims to provide an additional way of creating material classes instead of manual entering of the information the lattice constants and unit cell structure are parsed from the CIF file.
If multiple datasets are present in the CIF file this class will attempt to parse all of them into the the data dictionary. By default all methods access the first data set found in the file.

Parse (fid)
function to parse a CIF file. The function reads all the included data sets and adds them to the data dictionary.

SGLattice (dataset=None, use_p1=False)
create a SGLattice object with the structure from the CIF dataset

Parameters:
dataset : *str, optional*
name of the dataset to use. if None the default one will be used.
use_p1 : *bool, optional*
force the use of P1 symmetry, default False

__init__ (filestr, digits=4)
initialization of the CIFFile class

Parameters:
filestr : *str, bytes*
CIF filename or string representation of the CIF file
digits : *int, optional*
number of digits to check if position is unique

xrayutilities.materials.cif.**cifexport** (filename, mat)
function to export a Crystal instance to CIF file. This in particular includes the atomic coordinates, however, ignores for example the elastic parameters.

xrayutilities.materials.database module

module to handle the access to the optical parameters database

class xrayutilities.materials.database.**DataBase**(fname)
Bases: **object**

Close()
Close an opend database file.

Create(dbname, dbdesc)
Creates a new database. If the database file already exists its content is delete.

Parameters: **dbname :** *str*
name of the database
dbdesc : *str*
a short description of the database

CreateMaterial(name, description)
This method creates a new material. If the material group already exists the procedure is aborted.

Parameters: **name :** *str*
name of the material
description : *str*
description of the material

GetF0(q, dset='default')
Obtain the f0 scattering factor component for a particular momentum transfer q.

Parameters: **q :** *float or array-like*
momentum transfer
dset : *str, optional*
specifies which dataset (different oxidation states) should be used

GetF1(en)
Return the second, energy dependent, real part of the scattering factor for a certain energy en.

Parameters: **en :** *float or array-like*
energy

GetF2(en)
Return the imaginary part of the scattering factor for a certain energy en.

Parameters: **en :** *float or array-like*
energy

Open(mode='r')
Open an existing database file.

SetColor(color)
Save color of the element for visualization

Parameters: **color :** *tuple, str*
matplotlib color for the element

SetF0(parameters, subset='default')
Save f0 fit parameters for the set material. The fit parameters are stored in the following order: c, a1, b1,……., a4, b4

Parameters: **parameters :** *list or array-like*
fit parameters
subset : *str, optional*
name the f0 dataset

SetF1F2 (en**,** f1**,** f2**)**
Set f1, f2 values for the active material.

Parameters: **en :** *list or array-like*
energy in (eV)
f1 : *list or array-like*
f1 values
f2 : *list or array-like*
f2 values

SetMaterial (name**)**
Set a particular material in the database as the actual material. All operations like setting and getting optical constants are done for this particular material.

Parameters: **name :** *str*
name of the material

SetRadius (radius**)**
Save atomic radius for visualization

Parameters: **radius: float**
atomic radius in angstrom

SetWeight (weight**)**
Save weight of the element as float

Parameters: **weight :** *float*
atomic standard weight of the element

__init__ (fname**)**

xrayutilities.materials.database.**add_color_from_JMOL** (db, cfile, verbose=False)
Read color from JMOL color table and save it to the database.

xrayutilities.materials.database.**add_f0_from_intertab** (db, itf, verbose=False)
Read f0 data from International Tables of Crystallography and add it to the database.

xrayutilities.materials.database.**add_f0_from_xop** (db, xop, verbose=False)
Read f0 data from f0_xop.dat and add it to the database.

xrayutilities.materials.database.**add_f1f2_from_ascii_file** (db, asciifile, element, verbose=False)
Read f1 and f2 data for specific element from ASCII file (3 columns) and save it to the database.

xrayutilities.materials.database.**add_f1f2_from_henkedb** (db, hf, verbose=False)
Read f1 and f2 data from Henke database and add it to the database.

xrayutilities.materials.database.**add_f1f2_from_kissel** (db, kf, verbose=False)
Read f1 and f2 data from Henke database and add it to the database.

xrayutilities.materials.database.**add_mass_from_NIST** (db, nistfile, verbose=False)
Read atoms standard mass and save it to the database. The mass of the natural isotope mixture is taken from the NIST data!

xrayutilities.materials.database.**add_radius_from_WIKI** (db, dfile, verbose=False)
Read radius from Wikipedia radius table and save it to the database.

xrayutilities.materials.database.**createAndFillDatabase** (fname, dpath=None, verbose=False)

function to create the database and fill it with values from the various source files.

Parameters: **fname :** *str*

Filename of the database to be created (including the path)

dpath : *str, optional*

directory where all the source data files are stored

verbose : *bool, optional*

flag to determine the verbosity of the script (default: False)

`xrayutilities.materials.database.`**`init_material_db`** (db)

xrayutilities.materials.elements module

xrayutilities.materials.heuslerlib module

implement convenience functions to define Heusler materials.

`xrayutilities.materials.heuslerlib.`**`FullHeuslerCubic225`** (X, Y, Z, a, `biso`=(0, 0, 0), `occ`=(1, 1, 1))

Full Heusler structure with formula X2YZ. Strukturberichte symbol L2_1; space group Fm-3m (225)

Parameters: **X, Y, Z :** *str or Element*

elements

a : *float*

cubic lattice parameter in angstrom

biso : *list of floats, optional*

Debye Waller factors for X, Y, Z elements

occ : *list of floats, optional*

occupation numbers for the elements X, Y, Z

Returns: **Crystal**

Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`FullHeuslerCubic225_A2`** (X, Y, Z, a, `a2dis`, `biso`=(0, 0, 0), `occ`=(1, 1, 1))

Full Heusler structure with formula X2YZ. Strukturberichte symbol L2_1; space group Fm-3m (225) with A2-type (W) disorder

Parameters: **X, Y, Z :** *str or Element*

elements

a : *float*

cubic lattice parameter in angstrom

a2dis : *float*

amount of A2-type disorder (0: fully ordered, 1: fully disordered)

biso : *list of floats, optional*

Debye Waller factors for X, Y, Z elements

occ : *list of floats, optional*

occupation numbers for the elements X, Y, Z

Returns: **Crystal**

Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`FullHeuslerCubic225_B2`** (X, Y, Z, a, `b2dis`, `biso`=(0, 0, 0), `occ`=(1, 1, 1))

Full Heusler structure with formula X2YZ. Strukturberichte symbol L2_1; space group Fm-3m (225) with B2-type (CsCl) disorder

Parameters: **X, Y, Z :** *str or Element*

elements

a : *float*

cubic lattice parameter in angstrom

b2dis : *float*

amount of B2-type disorder (0: fully ordered, 1: fully disordered)

biso : *list of floats, optional*

Debye Waller factors for X, Y, Z elements

occ : *list of floats, optional*

occupation numbers for the elements X, Y, Z

Returns: **Crystal**

Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`FullHeuslerCubic225_DO3`** (X, Y, Z, a, `do3disxy`, `do3disxz`, `biso`=(0, 0, 0), `occ`=(1, 1, 1))

Full Heusler structure with formula X2YZ. Strukturberichte symbol L2_1; space group Fm-3m (225) with DO_3-type (BiF3) disorder, either between atoms X <-> Y or X <-> Z.

Parameters: **X, Y, Z :** *str or Element*

elements

a : *float*

cubic lattice parameter in angstrom

do3disxy : *float*

amount of DO_3-type disorder between X and Y atoms (0: fully ordered, 1: fully disordered)

do3disxz : *float*

amount of DO_3-type disorder between X and Z atoms (0: fully ordered, 1: fully disordered)

biso : *list of floats, optional*

Debye Waller factors for X, Y, Z elements

occ : *list of floats, optional*

occupation numbers for the elements X, Y, Z

Returns: **Crystal**

Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`HeuslerHexagonal194`** (X, Y, Z, a, c, `biso`=(0, 0, 0), `occ`=(1, 1, 1))

Hexagonal Heusler structure with formula XYZ space group P63/mmc (194)

Parameters: **X, Y, Z :** *str or Element*

elements

a, c : *float*

hexagonal lattice parameters in angstrom

Returns: **Crystal**

Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`HeuslerTetragonal119`** (X, Y, Z, a, c, `biso`=(0, 0, 0), `occ`=(1, 1, 1))

Tetragonal Heusler structure with formula X2YZ space group I-4m2 (119)

Parameters: **X, Y, Z :** *str or Element*

elements

a, c : *float*

tetragonal lattice parameters in angstrom

Returns: **Crystal**
Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`HeuslerTetragonal139`** (X, Y, Z, a, c, biso=(0, 0, 0), occ=(1, 1, 1))
Tetragonal Heusler structure with formula X2YZ space group I4/mmm (139)

Parameters: **X, Y, Z :** *str or Element*
elements
a, c : *float*
tetragonal lattice parameters in angstrom
Returns: **Crystal**
Crystal describing the Heusler material

`xrayutilities.materials.heuslerlib.`**`InverseHeuslerCubic216`** (X, Y, Z, a, biso=(0, 0, 0), occ=(1, 1, 1))
Full Heusler structure with formula (XY)X'Z structure; space group F-43m (216)

Parameters: **X, Y, Z :** *str or Element*
elements
a : *float*
cubic lattice parameter in angstrom
Returns: **Crystal**
Crystal describing the Heusler material

xrayutilities.materials.material module

Classes decribing materials. Materials are devided with respect to their crystalline state in either Amorphous or Crystal types. While for most materials their crystalline state is defined few materials are also included as amorphous which can be useful for calculation of their optical properties.

class `xrayutilities.materials.material.`**`Alloy`** (matA, matB, x)
Bases: **`Crystal`**
alloys two materials from the same crystal system. If the materials have the same space group the Wyckoff positions within the unit cell will also reflect the alloying.

`RelaxationTriangle` **(**`hkl`**,** `sub`**,** `exp`**)**
function which returns the relaxation triangle for a Alloy of given composition. Reciprocal space coordinates are calculated using the user-supplied experimental class

Parameters: **hkl :** *list or array-like*
Miller Indices
sub : *Crystal, or float*
substrate material or lattice constant
exp : *Experiment*
object from which the Transformation object and ndir are needed
Returns: **qy, qz :** *float*
reciprocal space coordinates of the corners of the relaxation triangle

`__init__` **(**`matA`**,** `matB`**,** `x`**)**

static **`check_compatibility`** **(**`matA`**,** `matB`**)**

static **`lattice_const_AB`** **(**`latA`**,** `latB`**,** `x`**)**
method to calculated the interpolation of lattice parameters and unit cell angles of the Alloy. By default linear interpolation between the value of material A and B is performed.

Parameters: **latA, latB :** *float or vector*

property (lattice parameter/angle) of material A and B. A property can be a scalar or vector.

x : *float*

fraction of material B in the alloy.

property **x**

class xrayutilities.materials.material.**Amorphous** (name, density, atoms=None, cij=None)

Bases: **Material**

amorphous materials are described by this class

__init__ (name**,** density**,** atoms=None**,** cij=None**)**

constructor of an amorphous material. The amorphous material is described by its density and atom composition.

Parameters: **name :** *str*

name of the material. To allow automatic parsing of the chemical elements use the abbreviation of the chemical element from the periodic table. To specify alloys, use e.g. ‘Ir0.2Mn0.8’ or ‘H2O’.

density : *float*

mass density in kg/m^3

atoms : *list, optional*

list of atoms together with their fractional content. When the name is a simply chemical formula then this can be None. To specify more complicated materials use [(‘Ir’, 0.2), (‘Mn’, 0.8), …]. Instead of the elements as string you can also use an Atom object. If the contents to not add up to 1 they will be normalized without notice.

cij : *array-like, optional*

elasticity matrix

chi0 (en='config'**)**

calculates the complex chi_0 values often needed in simulations. They are closely related to delta and beta (n = 1 + chi_r0/2 + i*chi_i0/2 vs. n = 1 - delta + i*beta)

delta (en='config'**)**

function to calculate the real part of the deviation of the refractive index from 1 (n=1-delta+i*beta)

Parameters: **en :** *float, array-like or str, optional*

energy of the x-rays in eV

Returns: **float or array-like**

ibeta (en='config'**)**

function to calculate the imaginary part of the deviation of the refractive index from 1 (n=1-delta+i*beta)

Parameters: **en :** *float, array-like or str, optional*

energy of the x-rays in eV

Returns: **float or array-like**

***static* parseChemForm (**cstring**)**

Parse a string containing a simple chemical formula and transform it to a list of elements together with their relative atomic fraction. e.g. ‘H2O’ -> [(H, 2/3), (O, 1/3)], where H and O are the Element objects of Hydrogen and Oxygen. Note that every chemical element needs to start with a capital letter! Complicated formulas containing bracket are not supported!

Parameters: **cstring :** *str*

string containing the chemical fomula

Returns: **list of tuples**
chemical element and atomic fraction

`xrayutilities.materials.material.`**`Cij2Cijkl`**`(cij)`
Converts the elastic constants matrix (tensor of rank 2) to the full rank 4 cijkl tensor.

Parameters: **cij :** *array-like*
(6, 6) cij matrix
Returns: **cijkl ndarray**
(3, 3, 3, 3) cijkl tensor as numpy array

`xrayutilities.materials.material.`**`Cij2Sijkl`**`(cij)`
Converts the elastic constants matrix (tensor of rank 2) to the full rank 4 sijkl compliance tensor.

Parameters: **cij :** *array-like*
(6, 6) cij matrix
Returns: **sijkl ndarray**
(3, 3, 3, 3) sijkl tensor as numpy array

`xrayutilities.materials.material.`**`Cijkl2Cij`**`(cijkl)`
Converts the full rank 4 tensor of the elastic constants to the (6, 6) matrix of elastic constants.

Parameters: **cijkl ndarray**
(3, 3, 3, 3) cijkl tensor as numpy array
Returns: **cij :** *array-like*
(6, 6) cij matrix

class `xrayutilities.materials.material.`**`Crystal`**`(name, lat, cij=None, thetaDebye=None)`
Bases: **`Material`**
Crystalline materials are described by this class

`ApplyStrain`**`(strain)`**
Applies a certain strain on the lattice of the material. The result is a change in the base vectors of the real space as well as reciprocal space lattice. The full strain matrix (3x3) needs to be given, which can be GetStrain's output.

Note

NO elastic response of the material will be considered!

property **`B`**

`GetMismatch`**`(mat)`**
Calculate the mismatch strain between the material and a second material

`HKL`**`(*q)`**
Return the HKL-coordinates for a certain Q-space position.

Parameters: **q :** *list or array-like*
Q-position. its also possible to use HKL(qx, qy, qz).

`Q`**`(*hkl)`**
Return the Q-space position for a certain material.

Parameters: **hkl :** *list or array-like*
Miller indices (or Q(h, k, l) is also possible)

`StructureFactor`**`(q, en='config', temp=0)`**
calculates the structure factor of a material for a certain momentum transfer and energy at a certain temperature of the material

Parameters: **q :** *list, tuple or array-like*
vectorial momentum transfer
en : *float or str, optional*
x-ray energy eV, if omitted the value from the xrayutilities configuration is used
temp : *float*
temperature used for Debye-Waller-factor calculation
Returns: **complex**
the complex structure factor

StructureFactorForEnergy (q0**,** en**,** temp=0**)**
calculates the structure factor of a material for a certain momentum transfer and a bunch of energies

Parameters: **q0 :** *list, tuple or array-like*
vectorial momentum transfer
en : *list, tuple or array-like*
energy values in eV
temp : *float*
temperature used for Debye-Waller-factor calculation
Returns: **array-like**
complex valued structure factor array

StructureFactorForQ (q**,** en0='config'**,** temp=0**)**
calculates the structure factor of a material for a bunch of momentum transfers and a certain energy

Parameters: **q :** *list of vectors or array-like*
vectorial momentum transfers; list of vectores (list, tuple or array) of length 3 e.g.: (Si.Q(0, 0, 4), Si.Q(0, 0, 4.1),…) or numpy.array([Si.Q(0, 0, 4), Si.Q(0, 0, 4.1)])
en0 : *float or str, optional*
x-ray energy eV, if omitted the value from the xrayutilities configuration is used
temp : *float*
temperature used for Debye-Waller-factor calculation
Returns: **array-like**
complex valued structure factor array

__init__ (name**,** lat**,** cij=None**,** thetaDebye=None**)**

property **a**

property **a1**

property **a2**

property **a3**

property **alpha**

property **b**

property **beta**

property **c**

chemical_composition (natoms=None**,** with_spaces=False**,** ndigits=2**)**
determine chemical composition from occupancy of atomic positions.

Parameters: **mat :** *Crystal*

instance of Crystal

natoms : *int, optional*

number of atoms to normalize the formula, if None some automatic normalization is attempted using the greatest common divisor of the number of atoms per unit cell. If the number of atoms of any element is fractional natoms=1 is used.

with_spaces : *bool, optional*

add spaces between the different entries in the output string for CIF combatibility

ndigits : *int, optional*

number of digits to which floating point numbers are rounded to

Returns: **str**

representation of the chemical composition

chi0 (en='config'**)**

calculates the complex chi_0 values often needed in simulations. They are closely related to delta and beta (n = 1 + chi_r0/2 + i*chi_i0/2 vs. n = 1 - delta + i*beta)

chih (q**,** en='config'**,** temp=0**,** polarization='S'**)**

calculates the complex polarizability of a material for a certain momentum transfer and energy

Parameters: **q :** *list, tuple or array-like*

momentum transfer vector in (1/A)

en : *float or str, optional*

x-ray energy eV, if omitted the value from the xrayutilities configuration is used

temp : *float, optional*

temperature used for Debye-Waller-factor calculation

polarization : *{'S', 'P'}, optional*

sigma or pi polarization

Returns: **tuple**

(abs(chih_real), abs(chih_imag)) complex polarizability

dTheta (Q**,** en='config'**)**

function to calculate the refractive peak shift

Parameters: **Q :** *list, tuple or array-like*

momentum transfer vector (1/A)

en : *float or str, optional*

x-ray energy eV, if omitted the value from the xrayutilities configuration is used

Returns: **float**

peak shift in degree

delta (en='config'**)**

function to calculate the real part of the deviation of the refractive index from 1 (n=1-delta+i*beta)

Parameters: **en :** *float or str, optional*

x-ray energy eV, if omitted the value from the xrayutilities configuration is used

Returns: **float**

property **density**

calculates the mass density of an material from the mass of the atoms in the unit cell.

Returns: **float**

mass density in kg/m^3

distances ()
function to obtain distances of atoms in the crystal up to the unit cell size (largest value of a, b, c is the cut-off) returns a list of tuples with distance d and number of occurence n [(d1, n1), (d2, n2),…]

> **Note**
> if the base of the material is empty the list will be empty

environment (`*pos, **kwargs`**)**
Returns a list of neighboring atoms for a given position within the unit cell. If the material does not contain any atoms a dummy atom will be placed on the unit cell corners.

Parameters: **pos :** *list or array-like*
fractional coordinate in the unit cell
maxdist : *float*
maximum distance wanted in the list of neighbors (default: 7)
Returns: **list of tuples**
(distance, atomType, multiplicity) giving distance sorted list of atoms

classmethod **fromCIF (**`ciffilestr, **kwargs`**)**
Create a Crystal from a CIF file. The default data-set from the cif file will be used to create the Crystal.

Parameters: **ciffilestr :** *str, bytes*
filename of the CIF file or string representation of the CIF file
kwargs : *dict*
keyword arguments are passed to the init-method of CIFFile
Returns: **Crystal**

property **gamma**

ibeta (en='config'**)**
function to calculate the imaginary part of the deviation of the refractive index from 1 (n=1-delta+i*beta)

Parameters: **en :** *float or str, optional*
x-ray energy eV, if omitted the value from the xrayutilities configuration is used
Returns: **float**

loadLatticefromCIF (`ciffilestr`**)**
load the unit cell data (lattice) from the CIF file. Other material properties stay unchanged.

Parameters: **ciffilestr :** *str, bytes*
filename of the CIF file or string representation of the CIF file

planeDistance (`*hkl`**)**
determines the lattice plane spacing for the planes specified by (hkl)

Parameters: **h, k, l :** *list, tuple or floats*
Miller indices of the lattice planes given either as list, tuple or seperate arguments
Returns: **float**
the lattice plane spacing

Examples

```
>>> import xrayutilities as xu
>>> xu.materials.Si.planeDistance(0, 0, 4)
1.3577600000000003
```

or

```
>>> xu.materials.Si.planeDistance((1, 1, 1))
3.1356124059796264
```

show_unitcell (fig=None**,** subplot=111**,** scale=0.6**,** complexity=11**,** linewidth=1.5**,** mode='matplotlib'**)**
visualization of the unit cell using either matplotlibs basic 3D functionality (expect rendering inaccuracies!) or the mayavi mlab package (accurate rendering -> recommended!)

> **Note**
>
> For more flexible visualization consider using the CIF-export feature and use a proper crystal structure viewer.

Parameters:

fig : *matplotlib Figure, Mayavi Scene, or None, optional*

subplot : *int or list, optional*

subplot to use for the visualization when using matplotlib. This argument of fowarded to the first argument of matplotlibs *add_subplot* function

scale : *float, optional*

scale the size of the atoms by this additional factor. By default the size of the atoms corresponds to 60% of their atomic radius.

complexity : *int, optional*

number of steps to approximate the atoms as spheres. Higher values make spheres more accurate, but cause slower plotting.

linewidth : *float, optional*

line thickness of the unit cell outline

mode : *str, optional*

defines the plot backend used, can be 'matplotlib' (default) or 'mayavi'.

Returns: **figure object of either matplotlib or Mayavi**

toCIF (ciffilename**)**
Export the Crystal to a CIF file.

Parameters:

ciffilename : *str*

filename of the CIF file

class xrayutilities.materials.material.**CubicAlloy** (matA, matB, x)
Bases: **Alloy**

ContentBasym (q_inp**,** q_perp**,** hkl**,** sur**)**
function that determines the content of B in the alloy from the reciprocal space position of an asymmetric peak.

Parameters: **q_inp :** *float*

inplane peak position of reflection hkl of the alloy in reciprocal space

q_perp : *float*

perpendicular peak position of the reflection hkl of the alloy in reciprocal space

hkl : *list*

Miller indices of the measured asymmetric reflection

sur : *list*

Miller indices of the surface (determines the perpendicular direction)

Returns: **content :** *float*

content of B in the alloy determined from the input variables

list

[a_inplane a_perp, a_bulk_perp(x), eps_inplane, eps_perp]; lattice parameters calculated from the reciprocal space positions as well as the strain (eps) of the layer

ContentBsym (q_perp**,** hkl**,** inpr**,** asub**,** relax**)**

function that determines the content of B in the alloy from the reciprocal space position of a symetric peak. As an additional input the substrates lattice parameter and the degree of relaxation must be given

Parameters: **q_perp :** *float*

perpendicular peak position of the reflection hkl of the alloy in reciprocal space

hkl : *list*

Miller indices of the measured symmetric reflection (also defines the surface normal

inpr : *list*

Miller indices of a Bragg peak defining the inplane reference direction

asub : *float*

substrate lattice parameter

relax : *float*

degree of relaxation (needed to obtain the content from symmetric reciprocal space position)

Returns: **content :** *float*

the content of B in the alloy determined from the input variables

__init__ (matA**,** matB**,** x**)**

xrayutilities.materials.material.**CubicElasticTensor** (c11, c12, c44)

Assemble the 6x6 matrix of elastic constants for a cubic material from the three independent components of a cubic crystal

Parameters: **c11, c12, c44 :** *float*

independent components of the elastic tensor of cubic materials

Returns: **cij :** *ndarray*

6x6 matrix with elastic constants

xrayutilities.materials.material.**HexagonalElasticTensor** (c11, c12, c13, c33, c44)

Assemble the 6x6 matrix of elastic constants for a hexagonal material from the five independent components of a hexagonal crystal

Parameters: **c11, c12, c13, c33, c44 :** *float*

independent components of the elastic tensor of a hexagonal material

Returns: **cij :** *ndarray*

6x6 matrix with elastic constants

class xrayutilities.materials.material.**Material** (name, cij=None)

Bases: **ABC**
base class for all Materials. common properties of amorphous and crystalline materials are described by this class from which Amorphous and Crystal are derived from.

GetStrain (sig**)**
Obtains the strain matrix (3x3) from an applied stress matrix (3x3) using a material's full rank elastic tensor (3x3x3x3). The full stress matrix (3x3) needs to be given. The results can then be used as an input in ApplyStrain. Inverse operation of GetStress.

Parameters: **sig :** *list, tuple or array-like*
stress matrix (3x3) in N/m^2

GetStress (eps**)**
Obtains the strain matrix (3x3) from an applied stress matrix (3x3) using a material's full rank elastic tensor (3x3x3x3). The full stress matrix (3x3) needs to be given. Inverse operation of GetStrain.

Parameters: **eps :** *list, tuple or array-like*
strain matrix (3x3)

__init__ (name**,** cij=None**)**

absorption_length (en='config'**)**
wavelength dependent x-ray absorption length defined as mu = lambda/(2*pi*2*beta) with lambda and beta as the x-ray wavelength and complex part of the refractive index respectively.

Parameters: **en :** *float or str, optional*
energy of the x-rays in eV

Returns: **float**
the absorption length in um

chi0 (en='config'**)**
calculates the complex chi_0 values often needed in simulations. They are closely related to delta and beta (n = 1 + chi_r0/2 + i*chi_i0/2 vs. n = 1 - delta + i*beta)

critical_angle (en='config'**,** deg=True**)**
calculate critical angle for total external reflection

Parameters: **en :** *float or str, optional*
energy of the x-rays in eV, if omitted the value from the xrayutilities configuration is used

deg : *bool, optional*
return angle in degree if True otherwise radians (default:True)

Returns: **float**
Angle of total external reflection

***abstract* delta (**en='config'**)**
abstract method which every implementation of a Material has to override

property **density**

***abstract* ibeta (**en='config'**)**
abstract method which every implementation of a Material has to override

idx_refraction (en='config'**)**
function to calculate the complex index of refraction of a material in the x-ray range

Parameters: **en :** *energy of the x-rays, if omitted the value from the*
xrayutilities configuration is used

Returns: **n (complex)**

property **lam**

property **mu**

property **nu**

poisson_ratio (direction**,** perpendicular**)**
Obtain the Poisson ratio for a certain extenstion direction and one perpendicular direction.

Parameters: **direction: vector (array of length 3)**
Axial extension direction.
perpendicular: vector
Lateral contraction direction.
Returns: **Poisson ratio**

youngs_modulus (direction**,** sijkl=None**)**
Obtain Youngs Modulus for a certain direction

Parameters: **direction: vector (array of length 3)**
Vectorial direction for this the Youngs modulus should be obtained. This does not need to be normalized.
Returns: **Youngs modulus in Pa**

xrayutilities.materials.material.**MonoclinicElasticTensor** (c11, c12, c13, c16, c22, c23, c26, c33, c36, c44, c45, c55, c66)

Assemble the 6x6 matrix of elastic constants for a monoclinic material from the thirteen independent components of a monoclinic crystal

Parameters: **c11, c12, c13, c16, c22, c23, c26, c33, c36, c44, c45, c55, c66 :** *float*
independent components of the elastic tensor of monoclinic materials
Returns: **cij :** *ndarray*
6x6 matrix with elastic constants

xrayutilities.materials.material.**PseudomorphicMaterial** (sub, layer, relaxation=0, trans=None)
This function returns a material whos lattice is pseudomorphic on a particular substrate material. The two materials must have similar unit cell definitions for the algorithm to work correctly, i.e. it does not work for combiniations of materials with different lattice symmetry. It is also crucial that the layer object includes values for the elastic tensor.

Parameters: **sub :** *Crystal*
substrate material
layer : *Crystal*
bulk material of the layer, including its elasticity tensor
relaxation : *float, optional*
degree of relaxation 0: pseudomorphic, 1: relaxed (default: 0)
trans : *Tranform*
Transformation which transforms lattice directions into a surface orientated coordinate frame (x, y inplane, z out of plane). If None a (001) surface geometry of a cubic material is assumed.
Returns: **An instance of Crystal holding the new pseudomorphically**

strained material.

Raises: **InputError**
If the layer material has no elastic parameters

xrayutilities.materials.material.**TrigonalElasticTensor** (c11, c12, c13, c14, c15, c33, c44)

Assemble the 6x6 matrix of elastic constants for a trigonal material from the seven independent components of a trigonal crystal

Parameters: **c11, c12, c13, c14, c15, c33, c44 :** *float*

independent components of the elastic tensor of trigonal materials

Returns: **cij :** *ndarray*

6x6 matrix with elastic constants

xrayutilities.materials.material.**WZTensorFromCub** (c11ZB, c12ZB, c44ZB)

Determines the hexagonal elastic tensor from the values of the cubic elastic tensor under the assumptions presented in Phys. Rev. B 6, 4546 (1972), which are valid for the WZ <-> ZB polymorphs.

Parameters: **c11, c12, c44 :** *float*

independent components of the elastic tensor of cubic materials

Returns: **cij :** *ndarray*

6x6 matrix with elastic constants

Implementation according to a patch submitted by Julian Stangl

xrayutilities.materials.material.**check_symmetric** (matrix)

xrayutilities.materials.material.**index_map_ij2ijkl** (ij)

xrayutilities.materials.material.**index_map_ijkl2ij** (i, j)

xrayutilities.materials.plot module

xrayutilities.materials.plot.**show_reciprocal_space_plane** (mat, exp, ttmax=None, maxqout=0.01, scalef=100, ax=None, color=None, show_Laue=True, show_legend=True, projection='perpendicular', label=None, **kwargs)

show a plot of the coplanar diffraction plane with peak positions for the respective material. the size of the spots is scaled with the strength of the structure factor

Parameters: **mat: Crystal**

instance of Crystal for structure factor calculations

exp: Experiment

instance of Experiment (needs to be HXRD, or FourC for onclick action to work correctly). defines the inplane and out of plane direction as well as the sample azimuth

ttmax: float, optional

maximal 2Theta angle to consider, by default 180deg

maxqout: float, optional

maximal out of plane q for plotted Bragg peaks as fraction of exp.k0

scalef: float, or callable, optional

scale factor or function for the marker size. If this is a function it should take only one float argument and return another float which is used as ‘s’ parameter in matplotlib.pyplot.scatter

ax: matplotlib.Axes, optional

matplotlib Axes to use for the plot, useful if multiple materials should be plotted in one plot

color: matplotlib color, optional

show_Laue: bool, optional

flag to indicate if the Laue zones should be indicated

show_legend: bool, optional

flag to indiate if a legend should be shown

projection: ‘perpendicular’, ‘polar’, optional

type of projection for Bragg peaks which do not fall into the diffraction plane. ‘perpendicular’ (default) uses only the inplane component in the scattering plane, whereas ‘polar’ uses the vectorial absolute value of the two inplane components. See also the ‘maxqout’ option.

label: None or str, optional

label to be used for the legend. If ‘None’ the name of the material will be used.

kwargs: optional

kwargs are forwarded to matplotlib.pyplot.scatter and allow to change the appearance of the points.

Returns: **Axes, plot_handle**

xrayutilities.materials.predefined_materials module

class xrayutilities.materials.predefined_materials.**AlGaAs**(x)
Bases: **CubicAlloy**

__init__(x**)**
Al_{1-x} Ga_x As cubic compound

class xrayutilities.materials.predefined_materials.**SiGe**(x)
Bases: **CubicAlloy**

__init__(x**)**
Si_{1-x} Ge_x cubic compound

static **lattice_const_AB(**latA, latB, x**)**
method to calculate the lattice parameter of the SiGe alloy with composition Si_{1-x}Ge_x

xrayutilities.materials.spacegrouplattice module

module handling crystal lattice structures. A SGLattice consists of a space group number and the position of atoms specified as Wyckoff positions along with their parameters. Depending on the space group symmetry only certain parameters of the resulting instance will be settable! A cubic lattice for example allows only to set its ‘a’ lattice parameter but none of the other unit cell shape parameters.

class `xrayutilities.materials.spacegrouplattice.`**`SGLattice`**(`sgrp`, `*args`, `**kwargs`)

Bases: **`object`**

lattice object created from the space group number and corresponding unit cell parameters. atoms in the unit cell are specified by their Wyckoff position and their free parameters.

`ApplyStrain`(`eps`)

Applies a certain strain on a lattice. The result is a change in the base vectors. The full strain matrix (3x3) needs to be given.

Note

Here you specify the strain and not the stress -> NO elastic response of the material will be considered!

Note

Although the symmetry of the crystal can be lowered by this operation the spacegroup remains unchanged! The ‘free_parameters’ attribute is, however, updated to mimic the possible reduction of the symmetry.

Parameters: **eps :** *array-like*

a 3x3 matrix with all strain components

property **B**

`GetHKL`(`*args`)

determine the Miller indices of the given reciprocal lattice points

`GetPoint`(`*args`)

determine lattice points with indices given in the argument

Examples

```
>>> import xrayutilities as xu
>>> xu.materials.Si.lattice.GetPoint(0, 0, 4)
array([ 0.     ,  0.     , 21.72416])
```

or

```
>>> xu.materials.Si.lattice.GetPoint((1, 1, 1))
array([5.43104, 5.43104, 5.43104])
```

`GetQ`(`*args`)

determine the reciprocal lattice points with indices given in the argument

`UnitCellVolume`()

function to calculate the unit cell volume of a lattice (angstrom^3)

`__init__`(`sgrp`, `*args`, `**kwargs`)

initialize class with space group number and atom list

Parameters: **sgrp :** *int or str*

Space group number

***args :** *float*

space group parameters. depending on the space group number this are 1 (cubic) to 6 (triclinic) parameters. cubic : a (lattice parameter). hexagonal : a, c. trigonal : a, c. tetragonal : a, c. orthorhombic : a, b, c. monoclinic : a, b, c, beta (in degree). triclinic : a, b, c, alpha, beta, gamma (in degree).

atoms : *list, optional*

list of elements either as Element object or string with the element name. If you specify atoms you have to also give the same number of Wyckoff positions

pos : *list, optional*

list of the atomic positions within the unit cell. This can be given as Wyckoff position along with its parameters or any position of an atom which will be used to identify the Wyckoff position. If a position has no free parameter the parameters can be omitted. Example: [('2i', (0.1, 0.2, 0.3)), '1a', (0, 0.5, 0)]

occ : *list, optional*

site occupation for the atoms. This is optional and defaults to 1 if not given.

b : *list, optional*

b-factor of the atom used as exp(-b*q**2/(4*pi)**2) to reduce the intensity of this atom (only used in case of temp=0 in StructureFactor and chi calculation)

property **a**

property **ai**

property **alpha**

property **b**

base ()
generator of atomic position within the unit cell.

property **beta**

property **c**

convert_to_P1 ()
create a P1 equivalent of this SGLattice instance.

Returns: **SGLattice**

instance with the same properties as the present lattice, however, in the P1 setting.

equivalent_hkls (hkl**)**
returns a list of equivalent hkl peaks depending on the crystal system

findsym ()
method to return the highest symmetry description of the current material. This method does not consider to change the unit cell dimensions but only searches the highest symmetry spacegroup which with the current unit cell setting can be described. It is therefore not an implementation of FINDSYM [1].

Returns: **new SGLattice-instance**

a new SGLattice instance is returned with the highest available symmetry description. (see restrictions above)

[1] https://stokes.byu.edu/iso/findsym.php

property **gamma**

get_allowed_hkl (qmax**)**
return a set of all allowed reflections up to a maximal specified momentum transfer.

Parameters: **qmax :** *float*

maximal momentum transfer

Returns: **hklset :** *set*

set of allowed hkl reflections

hkl_allowed (hkl**,** returnequivalents=False**)**

check if Bragg reflection with Miller indices hkl can exist according to the reflection conditions. If no reflection conditions are available this function returns True for all hkl values!

Parameters: **hkl :** *tuple or list*

Miller indices of the reflection to check

returnequivalents : *bool, optional*

If True all the equivalent Miller indices of hkl are returned in a set as second return argument.

Returns: **allowed :** *bool*

True if reflection can have non-zero structure factor, false otherwise

equivalents : *set, optional*

set of equivalent Miller indices if returnequivalents is True

property **iscentrosymmetric**

returns a boolean to determine if the lattice has centrosymmetry.

isequivalent (hkl1**,** hkl2**)**

determining if hkl1 and hkl2 are two crystallographical equivalent pairs of Miller indices. Note that this function considers the effect of non-centrosymmetry!

Parameters: **hkl1, hkl2 :** *list*

Miller indices to be checked for equivalence

Returns: **bool**

reflection_conditions ()

return string of reflection conditions, both general (from space group) and of Wyckoff positions

property **symops**

return the set of symmetry operations from the general Wyckoff position of the space group.

transform (mat**,** origin**)**

Transform the unit cell with the matrix and origin shift given in the parameters. This function returns a new instance of SGLattice which contains the highest possible symmetry description of the transformed unit cell. After the transformation (see [1]) the findsym method is used to create the new SGLattice instance.

Parameters: **mat :** *(3, 3) list, or ndarray, optional*

transformation matrix of the unit cell. The matrix definition aims to be consistent with what is used on the Bilbao Crystallographic Server [1]. This only defines the linear part, while the origin shift is given by origin.

origin : *(3,) list, or ndarray*

origin shift of the unit cell [1].

[1] https://www.cryst.ehu.es/cgi-bin/cryst/programs/nph-doc-trmat

class xrayutilities.materials.spacegrouplattice.**SymOp** (D, t, m=1)

Bases: **object**

Class descriping a symmetry operation in a crystal. The symmetry operation is characterized by a 3x3 transformation matrix as well as a 3-vector describing a translation. For magnetic symmetry operations also the time reversal symmetry can be specified (not used in xrayutilities)

property **D**

transformation matrix of the symmetry operation

__init__ (D**,** t**,** m=1**)**
Initialize the symmetry operation

Parameters: **D :** *array-like*
transformation matrix (3x3)

t : *array-like*
translation vector (3)

m : *int, optional*
indicates time reversal in magnetic groups. +1 (default, no time reveral) or -1

apply (vec**,** foldback=True**)**

apply_axial (vec**)**

apply_rotation (vec**)**

combine (other**)**

static **foldback (**v**)**

classmethod **from_xyz (**xyz**)**
create a SymOp from the xyz notation typically used in CIF files.

Parameters: **xyz :** *str*
string describing the symmetry operation (e.g. '-y, -x, z')

property **t**
translation vector of the symmetry operation

xyz (showtimerev=False**)**
return the symmetry operation in xyz notation

class xrayutilities.materials.spacegrouplattice.**WyckoffBase** (*args, **kwargs)
Bases: **list**
The WyckoffBase class implements a container for a set of Wyckoff positions that form the base of a crystal lattice. An instance of this class can be treated as a simple container object.

__init__ (*args**,** **kwargs**)**

append (atom**,** pos**,** occ=1.0**,** b=0.0**)**
add new Atom to the lattice base

Parameters: **atom :** *Atom*
object to be added

pos : *tuple or str*
Wyckoff position of the atom, along with its parameters. Examples: ('2i', (0.1, 0.2, 0.3)), or '1a'

occ : *float, optional*
occupancy (default=1.0)

b : *float, optional*
b-factor of the atom used as exp(-b*q**2/(4*pi)**2) to reduce the intensity of this atom (only used in case of temp=0 in StructureFactor and chi calculation)

static **entry_eq (**e1**,** e2**)**
compare two entries including all its properties to be equal

Parameters: **e1, e2: tuple**

tuples with length 4 containing the entries of WyckoffBase which should be compared

index (`item`**)**

return the index of the atom (same element, position, and Debye Waller factor). The occupancy is not checked intentionally. If the item is not present a ValueError is raised.

Parameters: **item :** *tuple or list*

WyckoffBase entry

Returns: **int**

static **pos_eq (**`pos1`**,** `pos2`**)**

compare Wyckoff positions

Parameters: **pos1, pos2: tuple**

tuples with Wyckoff label and optional parameters

`xrayutilities.materials.spacegrouplattice.`**`get_default_sgrp_suf`** (`sgrp_nr`)

determine default space group suffix

`xrayutilities.materials.spacegrouplattice.`**`get_possible_sgrp_suf`** (`sgrp_nr`)

determine possible space group suffix. Multiple suffixes might be possible for one space group due to different origin choice, unique axis, or choice of the unit cell shape.

Parameters: **sgrp_nr :** *int*

space group number

Returns: **str or list**

either an empty string or a list of possible valid suffix strings

`xrayutilities.materials.spacegrouplattice.`**`get_wyckpos`** (`sgrp`, `atompos`)

test all Wyckoff positions on every atomic position

Parameters: **sgrp :** *str*

space group name

atompos : *list*

list of atomic positions to identify. All atomic positions are expected to belong to one and the same Wyckoff position!

Returns: **position argument for WyckoffBase.append**

`xrayutilities.materials.spacegrouplattice.`**`testwp`** (`parint`, `wyckpos`, `cifpos`, `digits=8`)

test if a Wyckoff position can describe the given position from a CIF file

Parameters: **parint :** *int*

telling which Parameters the given Wyckoff position has

wyckpos : *str or tuple*

expression of the Wyckoff position

cifpos : *list, or tuple or array-like*

(x, y, z) position of the atom in the CIF file

digits : *int*

number of digits for which for a comparison of floating point numbers will be rounded to. By default xu.config.DIGITS is used.

Returns: **foundflag :** *bool*

flag to tell if the positions match

pars : *array-like or None*

parameters associated with the position or None if no parameters are needed

xrayutilities.materials.wyckpos module

class xrayutilities.materials.wyckpos.**RangeDict**
Bases: **dict**
Dictionary type which uses range as keys

Module contents

xrayutilities.math package

Submodules

xrayutilities.math.algebra module

module providing analytic algebraic functions not implemented in scipy or any other dependency of xrayutilities. In particular the analytic solution of a quartic equation which is needed for the solution of the dynamic scattering equations.

xrayutilities.math.algebra.**solve_quartic**(a4, a3, a2, a1, a0)
analytic solution [1] of the general quartic equation. The solved equation takes the form $a_4z^4 + a_3z^3 + a_2z^2 + a_1z + a_0$

Returns: **tuple**
tuple of the four (complex) solutions of aboves equation.

References

1 http://mathworld.wolfram.com/QuarticEquation.html

xrayutilities.math.fit module

module with a function wrapper to scipy.optimize.leastsq for fitting of a 2D function to a peak or a 1D Gauss fit with the odr package

xrayutilities.math.fit.**fit_peak2d**(x, y, data, start, drange, fit_function, maxfev=2000)
fit a two dimensional function to a two dimensional data set e.g. a reciprocal space map.

Parameters: **x :** *array-like*
first data coordinate (does not need to be regularly spaced)
y : *array-like*
second data coordinate (does not need to be regularly spaced)
data : *array-like*
data set used for fitting (e.g. intensity at the data coordinates)
start : *list*
set of starting parameters for the fit used as first parameter of function fit_function
drange : *list*
limits for the data ranges used in the fitting algorithm, e.g. it is clever to use only a small region around the peak which should be fitted, i.e. [xmin, xmax, ymin, ymax]
fit_function : *callable*
function which should be fitted. Call signature must be **fit_function(x, y, *params) -> ndarray()**

Returns: **fitparam :** *list*
fitted parameters
cov : *array-like*
covariance matrix

xrayutilities.math.fit.**gauss_fit**(xdata, ydata, iparams=None, maxit=300)

Gauss fit function using odr-pack wrapper in scipy similar to https://github.com/tiagopereira/python_tips/wiki/Scipy%3A-curve-fitting

Parameters: **xdata :** *array-like*

x-coordinates of the data to be fitted

ydata : *array-like*

y-coordinates of the data which should be fit

iparams: list, optional

initial paramters for the fit, determined automatically if not given

maxit : *int, optional*

maximal iteration number of the fit

Returns: **params :** *list*

the parameters as defined in function `Gauss1d(x, *param)`

sd_params : *list*

For every parameter the corresponding errors are returned.

itlim : *bool*

flag to tell if the iteration limit was reached, should be False

`xrayutilities.math.fit.`**`linregress`** (x, y)

fast linregress to avoid usage of scipy.stats which is slow! NaN values in y are ignored by this function.

Parameters: **x, y :** *array-like*

data coordinates and values

Returns: **p :** *tuple*

parameters of the linear fit (slope, offset)

rsq: float

R^2 value

Examples

```
>>> (k, d), R2 = linregress([1, 2, 3], [3.3, 4.1, 5.05])
```

`xrayutilities.math.fit.`**`multPeakFit`** (x, `data`, `peakpos`, `peakwidth`, `dranges`=None, `peaktype`='Gaussian', `returnerror`=False)

function to fit multiple Gaussian/Lorentzian peaks with linear background to a set of data

Parameters: **x :** *array-like*

x-coordinate of the data

data : *array-like*

data array with same length as *x*

peakpos : *list*

initial parameters for the peak positions

peakwidth : *list*

initial values for the peak width

dranges : *list of tuples*

list of tuples with (min, max) value of the data ranges to use. does not need to have the same number of entries as peakpos

peaktype : *{'Gaussian', 'Lorentzian'}*

type of peaks to be used

returnerror : *bool*

decides if the fit errors of pos, sigma, and amp are returned (default: False)

Returns: **pos :** *list*
peak positions derived by the fit
sigma : *list*
peak width derived by the fit
amp : *list*
amplitudes of the peaks derived by the fit
background : *array-like*
background values at positions *x*
if returnerror == True:
sd_pos : *list*
standard error of peak positions as returned by scipy.odr.Output
sd_sigma : *list*
standard error of the peak width
sd_amp : *list*
standard error of the peak amplitude

`xrayutilities.math.fit.`**`multPeakPlot`** (`x`, `fpos`, `fwidth`, `famp`, `background`, `dranges`=None, `peaktype`='Gaussian', `fig`='xu_plot', `ax`=None, `fact`=1.0)
function to plot multiple Gaussian/Lorentz peaks with background values given by an array

Parameters: **x :** *array-like*
x-coordinate of the data
fpos : *list*
positions of the peaks
fwidth : *list*
width of the peaks
famp : *list*
amplitudes of the peaks
background : *array-like*
background values, same shape as *x*
dranges : *list of tuples*
list of (min, max) values of the data ranges to use. does not need to have the same number of entries as fpos
peaktype : *{'Gaussian', 'Lorentzian'}*
type of peaks to be used
fig : *int, str, or None*
matplotlib figure number or name
ax : *matplotlib.Axes*
matplotlib axes as alternative to the figure name
fact : *float*
factor to use as multiplicator in the plot

`xrayutilities.math.fit.`**`peak_fit`** (`xdata`, `ydata`, `iparams`=None, `peaktype`='Gauss', `maxit`=300, `background`='constant', `plot`=False, `func_out`=False, `debug`=False)
fit function using odr-pack wrapper in scipy similar to https://github.com/tiagopereira/python_tips/wiki/Scipy%3A-curve-fitting for Gauss, Lorentz or Pseudovoigt-functions

Parameters: **xdata :** *array_like*

x-coordinates of the data to be fitted

ydata : *array_like*

y-coordinates of the data which should be fit

iparams : *list, optional*

initial paramters, determined automatically if not specified

peaktype : *{'Gauss', 'Lorentz', 'PseudoVoigt',*

'PseudoVoigtAsym', 'PseudoVoigtAsym2'}, optional

type of peak to fit

maxit : *int, optional*

maximal iteration number of the fit

background : *{'constant', 'linear'}, optional*

type of background function

plot : *bool or str, optional*

flag to ask for a plot to visually judge the fit. If plot is a string it will be used as figure name, which makes reusing the figures easier.

func_out : *bool, optional*

returns the fitted function, which takes the independent variables as only argument (f(x))

Returns: **params :** *list*

the parameters as defined in function *Gauss1d/Lorentz1d/PseudoVoigt1d/PseudoVoigt1dasym*. In the case of linear background one more parameter is included!

sd_params : *list*

For every parameter the corresponding errors are returned.

itlim : *bool*

flag to tell if the iteration limit was reached, should be False

fitfunc : *function, optional*

the function used in the fit can be returned (see func_out).

xrayutilities.math.functions module

module with several common function needed in xray data analysis

`xrayutilities.math.functions.`**`Debye1`** (x)

function to calculate the first Debye function [1] as needed for the calculation of the thermal Debye-Waller-factor by numerical integration

$$D_1(x) = (1/x)\int_0^x t/(\exp(t) - 1)dt$$

Parameters: **x :** *float*

argument of the Debye function

Returns: **float**

D1(x) float value of the Debye function

References

1 http://en.wikipedia.org/wiki/Debye_function

`xrayutilities.math.functions.`**`Gauss1d`** (x, *p)

function to calculate a general one dimensional Gaussian

Parameters: **x :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Gaussian [XCEN, SIGMA, AMP, BACKGROUND] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

Returns: **array-like**

the value of the Gaussian described by the parameters p at position x

Examples

Calling with a list of parameters needs a call looking as shown below (note the '*') or explicit listing of the parameters

```
>>> Gauss1d(x, *p)

>>> import numpy
>>> Gauss1d(numpy.linspace(0, 10, 10), 5, 1, 1e3, 0)
array([3.72665317e-03, 5.19975743e-01, 2.11096565e+01, 2.49352209e+02,
       8.56996891e+02, 8.56996891e+02, 2.49352209e+02, 2.11096565e+01,
       5.19975743e-01, 3.72665317e-03])
```

`xrayutilities.math.functions.`**`Gauss1dArea`** (*p)

function to calculate the area of a Gauss function with neglected background

Parameters: **p :** *list*

list of parameters of the Gauss-function [XCEN, SIGMA, AMP, BACKGROUND]

Returns: **float**

the area of the Gaussian described by the parameters *p*

`xrayutilities.math.functions.`**`Gauss1d_der_p`** (x, *p)

function to calculate the derivative of a Gaussian with respect the parameters p
for parameter description see Gauss1d

`xrayutilities.math.functions.`**`Gauss1d_der_x`** (x, *p)

function to calculate the derivative of a Gaussian with respect to x
for parameter description see Gauss1d

`xrayutilities.math.functions.`**`Gauss2d`** (x, y, *p)

function to calculate a general two dimensional Gaussian

Parameters: **x, y :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Gauss-function [XCEN, YCEN, SIGMAX, SIGMAY, AMP, BACKGROUND, ANGLE]; SIGMA = FWHM / (2*sqrt(2*log(2))); ANGLE = rotation of the X, Y direction of the Gaussian in radians

Returns: **array-like**

the value of the Gaussian described by the parameters p at position (x, y)

`xrayutilities.math.functions.`**`Gauss2dArea`** (*p)

function to calculate the area of a 2D Gauss function with neglected background

Parameters: **p :** *list*

list of parameters of the Gauss-function [XCEN, YCEN, SIGMAX, SIGMAY, AMP, ANGLE, BACKGROUND]

Returns: **float**

the area of the Gaussian described by the parameters *p*

`xrayutilities.math.functions.`**`Gauss3d`** (x, y, z, *p)

function to calculate a general three dimensional Gaussian

Parameters: **x, y, z :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Gauss-function [XCEN, YCEN, ZCEN, SIGMAX, SIGMAY, SIGMAZ, AMP, BACKGROUND];

SIGMA = FWHM / (2*sqrt(2*log(2)))

Returns: **array-like**

the value of the Gaussian described by the parameters p at positions (x, y, z)

`xrayutilities.math.functions.`**`Lorentz1d`** (x, *p)
function to calculate a general one dimensional Lorentzian

Parameters: **x :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Lorentz-function [XCEN, FWHM, AMP, BACKGROUND]

Returns: **array-like**

the value of the Lorentian described by the parameters p at position (x, y)

`xrayutilities.math.functions.`**`Lorentz1dArea`** (*p)
function to calculate the area of a Lorentz function with neglected background

Parameters: **p :** *list*

list of parameters of the Lorentz-function [XCEN, FWHM, AMP, BACKGROUND]

Returns: **float**

the area of the Lorentzian described by the parameters *p*

`xrayutilities.math.functions.`**`Lorentz1d_der_p`** (x, *p)
function to calculate the derivative of a Gaussian with respect the parameters p
for parameter description see Lorentz1d

`xrayutilities.math.functions.`**`Lorentz1d_der_x`** (x, *p)
function to calculate the derivative of a Gaussian with respect to x
for parameter description see Lorentz1d

`xrayutilities.math.functions.`**`Lorentz2d`** (x, y, *p)
function to calculate a general two dimensional Lorentzian

Parameters: **x, y :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Lorentz-function [XCEN, YCEN, FWHMX, FWHMY, AMP, BACKGROUND, ANGLE]; ANGLE = rotation of the X, Y direction of the Lorentzian in radians

Returns: **array-like**

the value of the Lorentian described by the parameters p at position (x, y)

`xrayutilities.math.functions.`**`NormGauss1d`** (x, *p)
function to calculate a normalized one dimensional Gaussian

Parameters: **x :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Gaussian [XCEN, SIGMA]; for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

Returns: **array-like**

the value of the normalized Gaussian described by the parameters p at position x

xrayutilities.math.functions.**NormLorentz1d**(x, *p)
function to calculate a normalized one dimensional Lorentzian

Parameters: **x :** *array-like*
coordinate(s) where the function should be evaluated
p : *list*
list of parameters of the Lorentzian [XCEN, FWHM]
Returns: **array-like**
the value of the normalized Lorentzian described by the parameters p at position x

xrayutilities.math.functions.**PseudoVoigt1d**(x, *p)
function to calculate a pseudo Voigt function as linear combination of a Gauss and Lorentz peak

Parameters: **x :** *array-like*
coordinate(s) where the function should be evaluated
p : *list*
list of parameters of the pseudo Voigt-function [XCEN, FWHM, AMP, BACKGROUND, ETA]; ETA: 0 …1 0 means pure Gauss and 1 means pure Lorentz
Returns: **array-like**
the value of the PseudoVoigt described by the parameters p at position *x*

xrayutilities.math.functions.**PseudoVoigt1dArea**(*p)
function to calculate the area of a pseudo Voigt function with neglected background

Parameters: **p :** *list*
list of parameters of the Lorentz-function [XCEN, FWHM, AMP, BACKGROUND, ETA]; ETA: 0 …1 0 means pure Gauss and 1 means pure Lorentz
Returns: **float**
the area of the PseudoVoigt described by the parameters *p*

xrayutilities.math.functions.**PseudoVoigt1d_der_p**(x, *p)
function to calculate the derivative of a PseudoVoigt with respect the parameters *p*
for parameter description see PseudoVoigt1d

xrayutilities.math.functions.**PseudoVoigt1d_der_x**(x, *p)
function to calculate the derivative of a PseudoVoigt with respect to *x*
for parameter description see PseudoVoigt1d

xrayutilities.math.functions.**PseudoVoigt1dasym**(x, *p)
function to calculate an asymmetric pseudo Voigt function as linear combination of asymmetric Gauss and Lorentz peak

Parameters: **x :** *array-like*
coordinate(s) where the function should be evaluated
p : *list*
list of parameters of the pseudo Voigt-function [XCEN, FWHMLEFT, FWHMRIGHT, AMP, BACKGROUND, ETA]; ETA: 0 …1 0 means pure Gauss and 1 means pure Lorentz
Returns: **array-like**
the value of the PseudoVoigt described by the parameters p at position *x*

xrayutilities.math.functions.**PseudoVoigt1dasym2**(x, *p)
function to calculate an asymmetric pseudo Voigt function as linear combination of asymmetric Gauss and Lorentz peak

Parameters: **x :** *naddray*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the pseudo Voigt-function [XCEN, FWHMLEFT, FWHMRIGHT, AMP, BACKGROUND, ETALEFT, ETARIGHT]; ETA: 0 …1 0 means pure Gauss and 1 means pure Lorentz

Returns: **array-like**

the value of the PseudoVoigt described by the parameters p at position *x*

`xrayutilities.math.functions.`**`PseudoVoigt2d`** (x, y, *p)

function to calculate a pseudo Voigt function as linear combination of a Gauss and Lorentz peak in two dimensions

Parameters: **x, y :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the pseudo Voigt-function [XCEN, YCEN, FWHMX, FWHMY, AMP, BACKGROUND, ANGLE, ETA]; ETA: 0 …1 0 means pure Gauss and 1 means pure Lorentz

Returns: **array-like**

the value of the PseudoVoigt described by the parameters *p* at position *(x, y)*

`xrayutilities.math.functions.`**`TwoGauss2d`** (x, y, *p)

function to calculate two general two dimensional Gaussians

Parameters: **x, y :** *array-like*

coordinate(s) where the function should be evaluated

p : *list*

list of parameters of the Gauss-function [XCEN1, YCEN1, SIGMAX1, SIGMAY1, AMP1, ANGLE1, XCEN2, YCEN2, SIGMAX2, SIGMAY2, AMP2, ANGLE2, BACKGROUND]; SIGMA = FWHM / (2*sqrt(2*log(2))) ANGLE = rotation of the X, Y direction of the Gaussian in radians

Returns: **array-like**

the value of the Gaussian described by the parameters p at position (x, y)

`xrayutilities.math.functions.`**`heaviside`** (x)

Heaviside step function for numpy arrays

Parameters: **x: scalar or array-like**

argument of the step function

Returns: **int or array-like**

Heaviside step function evaluated for all values of *x* with datatype integer

`xrayutilities.math.functions.`**`kill_spike`** (`data`, `threshold`=2.0, `offset`=None)

function to smooth **single** data points which differ from the average of the neighboring data points by more than the threshold factor or more than the offset value. Such spikes will be replaced by the mean value of the next neighbors.

Warning

Use this function carefully not to manipulate your data!

Parameters: **data :** *array-like*

1d numpy array with experimental data

threshold : *float or None*

threshold factor to identify outlier data points. If None it will be ignored.

offset : *None or float*

offset value to identify outlier data points. If None it will be ignored.

Returns: **array-like**

1d data-array with spikes removed

`xrayutilities.math.functions.`**`multPeak1d`**(x, `*args`)

function to calculate the sum of multiple peaks in 1D. the peaks can be of different type and a background function (polynom) can also be included.

Parameters: **x :** *array-like*

coordinate where the function should be evaluated

args : *list*

list of peak/function types and parameters for every function type two arguments need to be given first the type of function as string with possible values 'g': Gaussian, 'l': Lorentzian, 'v': PseudoVoigt, 'a': asym. PseudoVoigt, 'p': polynom the second type of arguments is the tuple/list of parameters of the respective function. See documentation of math.Gauss1d, math.Lorentz1d, math.PseudoVoigt1d, math.PseudoVoigt1dasym, and numpy.polyval for details of the different function types.

Returns: **array-like**

value of the sum of functions at position *x*

`xrayutilities.math.functions.`**`multPeak2d`**(x, y, `*args`)

function to calculate the sum of multiple peaks in 2D. the peaks can be of different type and a background function (polynom) can also be included.

Parameters: **x, y :** *array-like*

coordinates where the function should be evaluated

args : *list*

list of peak/function types and parameters for every function type two arguments need to be given first the type of function as string with possible values 'g': Gaussian, 'l': Lorentzian, 'v': PseudoVoigt, 'c': constant the second type of arguments is the tuple/list of parameters of the respective function. See documentation of math.Gauss2d, math.Lorentz2d, math.PseudoVoigt2d for details of the different function types. The constant accepts a single float which will be added to the data

Returns: **array-like**

value of the sum of functions at position *(x, y)*

`xrayutilities.math.functions.`**`smooth`**(x, n)

function to smooth an array of data by averaging N adjacent data points

Parameters: **x :** *array-like*

1D data array

n : *int*

number of data points to average

Returns: **xsmooth: array-like**

smoothed array with same length as x

xrayutilities.math.misc module

`xrayutilities.math.misc.`**`center_of_mass`**(pos, data, background='none', full_output=False)

function to determine the center of mass of an array

Parameters: **pos :** *array-like*
position of the data points
data : *array-like*
data values
background : *{'none', 'constant', 'linear'}*
type of background, either 'none', 'constant' or 'linear'
full_output : *bool*
return background cleaned data and background-parameters
Returns: **float**
center of mass position

`xrayutilities.math.misc.`**`fwhm_exp`** (`pos`, `data`)
function to determine the full width at half maximum value of experimental data. Please check the obtained value visually (noise influences the result)

Parameters: **pos :** *array-like*
position of the data points
data : *array-like*
data values
Returns: **float**
fwhm value

`xrayutilities.math.misc.`**`gcd`** (`lst`)
greatest common divisor function using library functions

Parameters: **lst: array-like**
array of integer values for which the greatest common divisor should be determined
Returns: **gcd: int**

xrayutilities.math.transforms module

`xrayutilities.math.transforms.`**`ArbRotation`** (`axis`, `alpha`, `deg=True`)
Returns a transform that represents a rotation around an arbitrary axis by the angle alpha. positive rotation is anti-clockwise when looking from positive end of axis vector

Parameters: **axis :** *list or array-like*
rotation axis
alpha : *float*
rotation angle in degree (deg=True) or in rad (deg=False)
deg : *bool*
determines the input format of ang (default: True)
Returns: **Transform**

class `xrayutilities.math.transforms.`**`AxisToZ`** (`newzaxis`)
Bases: **`CoordinateTransform`**
Creates a coordinate transformation to move a certain axis to the z-axis. The rotation is done along the great circle. The x-axis of the new coordinate frame is created to be normal to the new and original z-axis. The new y-axis is create in order to obtain a right handed coordinate system.

`__init__` **(**`newzaxis`**)**
initialize the CoordinateTransformation to move a certain axis to the z-axis

Parameters: **newzaxis :** *list or array-like*
new z-axis

class xrayutilities.math.transforms.**AxisToZ_keepXY**(newzaxis)
Bases: **CoordinateTransform**
Creates a coordinate transformation to move a certain axis to the z-axis. The rotation is done along the great circle. The x-axis/y-axis of the new coordinate frame is created to be similar to the old x and y directions. This variant of AxisToZ assumes that the new Z-axis has its main component along the Z-direction

__init__(newzaxis**)**
initialize the CoordinateTransformation to move a certain axis to the z-axis

Parameters: **newzaxis :** *list or array-like*
new z-axis

class xrayutilities.math.transforms.**CoordinateTransform**(v1, v2, v3)
Bases: **Transform**
Create a Transformation object which transforms a point into a new coordinate frame. The new frame is determined by the three vectors v1/norm(v1), v2/norm(v2) and v3/norm(v3), which need to be orthogonal!

__init__(v1**,** v2**,** v3**)**
initialization routine for Coordinate transformation

Parameters: **v1, v2, v3 :** *list, tuple or array-like*
new base vectors

Returns: **Transform**
An instance of a Transform class

class xrayutilities.math.transforms.**Transform**(matrix)
Bases: **object**

__call__(args**,** rank=1**)**
transforms a vector, matrix or tensor of rank 4 (e.g. elasticity tensor)

Parameters: **args :** *list or array-like*
object to transform, list or numpy array of shape (…, n) (…, n, n), (…, n, n, n, n) where n is the size of the transformation matrix.

rank : *int*
rank of the supplied object. allowed values are 1, 2, and 4

__init__(matrix**)**

property **imatrix**

inverse(args**,** rank=1**)**
performs inverse transformation a vector, matrix or tensor of rank 4

Parameters: **args :** *list or array-like*
object to transform, list or numpy array of shape (…, n) (…, n, n), (…, n, n, n, n) where n is the size of the transformation matrix.

rank : *int*
rank of the supplied object. allowed values are 1, 2, and 4

xrayutilities.math.transforms.**VecAngle**((v1.v2)/(norm(v1)*norm(v2)))
alpha = acos((v1.v2)/(norm(v1)*norm(v2)))

Parameters: **v1, v2 :** *list or array-like*

input vector(s), either one vector or an array of vectors with shape (n, 3)

deg: bool, optional

True: return result in degree, False: in radiants (default: False)

Returns: **float or ndarray**

the angle included by the two vectors *v1* and *v2*, either a single float or an array with shape (n,)

`xrayutilities.math.transforms.`**`VecCross`** (`v1`, `v2`, `out`=None)

Calculate the vector cross product.

Parameters: **v1, v2 :** *list or array-like*

input vector(s), either one vector or an array of vectors with shape (n, 3)

out : *list or array-like, optional*

output vector

Returns: **ndarray**

cross product either of shape (3,) or (n, 3)

`xrayutilities.math.transforms.`**`VecDot`** (`v1`, `v2`)

Calculate the vector dot product.

Parameters: **v1, v2 :** *list or array-like*

input vector(s), either one vector or an array of vectors with shape (n, 3)

Returns: **float or ndarray**

innter product of the vectors, either a single float or (n,)

`xrayutilities.math.transforms.`**`VecNorm`** (`v`)

Calculate the norm of a vector.

Parameters: **v :** *list or array-like*

input vector(s), either one vector or an array of vectors with shape (n, 3)

Returns: **float or ndarray**

vector norm, either a single float or shape (n,)

`xrayutilities.math.transforms.`**`VecUnit`** (`v`)

Calculate the unit vector of v.

Parameters: **v :** *list or array-like*

input vector(s), either one vector or an array of vectors with shape (n, 3)

Returns: **ndarray**

unit vector of *v*, either shape (3,) or (n, 3)

`xrayutilities.math.transforms.`**`XRotation`** (`alpha`, `deg`=True)

Returns a transform that represents a rotation about the x-axis by an angle alpha. If deg=True the angle is assumed to be in degree, otherwise the function expects radiants.

`xrayutilities.math.transforms.`**`YRotation`** (`alpha`, `deg`=True)

Returns a transform that represents a rotation about the y-axis by an angle alpha. If deg=True the angle is assumed to be in degree, otherwise the function expects radiants.

`xrayutilities.math.transforms.`**`ZRotation`** (`alpha`, `deg`=True)

Returns a transform that represents a rotation about the z-axis by an angle alpha. If deg=True the angle is assumed to be in degree, otherwise the function expects radiants.

`xrayutilities.math.transforms.`**`distance`** (`x`, `y`, `z`, `point`, `vec`)

calculate the distance between the point (x, y, z) and the line defined by the point and vector vec

Parameters: **x :** *float or ndarray*
x coordinate(s) of the point(s)
y : *float or ndarray*
y coordinate(s) of the point(s)
z : *float or ndarray*
z coordinate(s) of the point(s)
point : *tuple, list or ndarray*
3D point on the line to which the distance should be calculated
vec : *tuple, list or ndarray*
3D vector defining the propergation direction of the line

`xrayutilities.math.transforms.`**`getSyntax`** (`vec`)
returns vector direction in the syntax 'x+' 'z-' or equivalents therefore works only for principle vectors of the coordinate system like e.g. [1, 0, 0] or [0, 2, 0]

Parameters: **vec :** *list or array-like*
vector of length 3
Returns: **str**
vector string following the synthax [xyz][+-]

`xrayutilities.math.transforms.`**`getVector`** (`string`)
returns unit vector along a rotation axis given in the syntax 'x+' 'z-' or equivalents

Parameters: **string: str**
vector string following the synthax [xyz][+-]
Returns: **ndarray**
vector along the given direction

`xrayutilities.math.transforms.`**`mycross`** (`vec`, `mat`)
function implements the cross-product of a vector with each column of a matrix

`xrayutilities.math.transforms.`**`rotarb`** (`vec`, `axis`, `ang`, `deg`=True)
function implements the rotation around an arbitrary axis by an angle ang positive rotation is anti-clockwise when looking from positive end of axis vector

Parameters: **vec :** *list or array-like*
vector to rotate
axis : *list or array-like*
rotation axis
ang : *float*
rotation angle in degree (deg=True) or in rad (deg=False)
deg : *bool*
determines the input format of ang (default: True)
Returns: **rotvec :** *rotated vector as numpy.array*

Examples

```
>>> rotarb([1, 0, 0], [0, 0, 1], 90)
array([6.123234e-17, 1.000000e+00, 0.000000e+00])
```

`xrayutilities.math.transforms.`**`tensorprod`** (`vec1`, `vec2`)
function implements an elementwise multiplication of two vectors

Module contents

xrayutilities.simpack package

Submodules

xrayutilities.simpack.darwin_theory module

class xrayutilities.simpack.darwin_theory.**DarwinModel**(qz, qx=0, qy=0, **kwargs)

Bases: **LayerModel**, **ABC**

model class inmplementing the basics of the Darwin theory for layers materials. This class is not fully functional and should be used to derive working models for particular material systems.
To make the class functional the user needs to implement the init_structurefactors() and _calc_mono() methods

__init__(qz, qx=0, qy=0, **kwargs)

constructor of the model class. The arguments consist of basic parameters which are needed to prepare the calculation of the model.

Parameters:

qz : *array-like*

momentum transfer values for the calculation

qx, qy : *float, optional*

inplane momentum transfer (not implemented!)

I0 : *float, optional*

the primary beam intensity

background : *float, optional*

the background added to the simulation

resolution_width : *float, optional*

width of the resolution function (deg)

polarization : *{'S', 'P', 'both'}*

polarization of the x-ray beam. If set to 'both' also Cmono, the polarization factor of the monochromator should be set

experiment : *Experiment, optional*

experiment class containing geometry and energy of the experiment.

Cmono : *float, optional*

polarization factor of the monochromator

energy : *float or str, optional*

x-ray energy in eV

init_structurefactors()

calculates the needed atomic structure factors

ncalls = *0*

simulate(ml)

main simulation function for the Darwin model. will calculate the reflected intensity

Parameters:

ml : *iterable*

monolayer sequence of the sample. This should be created with the function make_monolayer(). see its documentation for details

class xrayutilities.simpack.darwin_theory.**DarwinModelAlGaAs001**(qz, qx=0, qy=0, **kwargs)

Bases: **DarwinModelAlloy**

Darwin theory of diffraction for Al_x Ga_{1-x} As layers. The model is based on separation of the sample structure into building blocks of atomic planes from which a multibeam dynamical model is calculated.

AlAs = *<xrayutilities.materials.material.Crystal object>*

GaAs = *<xrayutilities.materials.material.Crystal object>*

aGaAs = *5.65325*

classmethod **abulk(**x**)**
calculate the bulk (relaxed) lattice parameter of the Al_{x}Ga_{1-x}As alloy

asub = *5.65325*

eAl = *Al (13)*

eAs = *As (33)*

eGa = *Ga (31)*

classmethod **get_dperp_apar(**x**,** apar**,** r=1**)**
calculate inplane lattice parameter and the out of plane lattice plane spacing (of the atomic planes!) from composition and relaxation

Parameters:
x : *float*
chemical composition parameter
apar : *float*
inplane lattice parameter of the material below the current layer (onto which the present layer is strained to). This value also served as a reference for the relaxation parameter.
r : *float*
relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns:
dperp : *float*
perpendicular d-spacing
apar : *float*
inplane lattice parameter

init_structurefactors(temp=300**)**
calculates the needed atomic structure factors

Parameters:
temp : *float, optional*
temperature used for the Debye model

re = *2.8179403262e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelAlloy**(qz, qx=0, qy=0, **kwargs)
Bases: **DarwinModel**, **ABC**
extension of the DarwinModel for an binary alloy system were one parameter is used to determine the chemical composition
To make the class functional the user needs to implement the get_dperp_apar() method and define the substrate lattice parameter (asub). See the DarwinModelSiGe001 class for an implementation example.

asub = *None*

abstract **get_dperp_apar(**x**,** apar**,** r=1**)**
calculate inplane lattice parameter and the out of plane lattice plane spacing (of the atomic planes!) from composition and relaxation.

Parameters: **x :** *float*

chemical composition parameter

apar : *float*

inplane lattice parameter of the material below the current layer (onto which the present layer is strained to). This value also served as a reference for the relaxation parameter.

r : *float*

relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: **dperp :** *float*

apar : *float*

make_monolayers (s**)**

create monolayer sequence from layer list

Parameters: **s :** *list*

layer model. list of layer dictionaries including possibility to form superlattices. As an example 5 repetitions of a Si(10nm)/Ge(15nm) superlattice on Si would like like:

```
>>> s = [(5, [{'t': 100, 'x': 0, 'r': 0},
...           {'t': 150, 'x': 1, 'r': 0}]),
...      {'t': 3500000, 'x': 0, 'r': 0}]
```

the dictionaries must contain 't': thickness in A, 'x': chemical composition, and either 'r': relaxation or 'ai': inplane lattice parameter. Future implementations for asymmetric peaks might include layer type 'l' (not yet implemented). Already now any additional property in the dictionary will be handed on to the returned monolayer list.

asub : *float*

inplane lattice parameter of the substrate

Returns: **list**

monolayer list in a format understood by the simulate and xGe_profile methods

prop_profile (ml, prop**)**

calculate the profile of chemical composition or inplane lattice spacing from a monolayer list. One value for each monolayer in the sample is returned.

Parameters: **ml :** *list*

monolayer list created by make_monolayer()

prop : *str*

name of the property which should be evaluated. Use 'x' for the chemical composition and 'ai' for the inplane lattice parameter.

Returns: **zm :** *ndarray*

z-position, z-0 is the surface

propx : *ndarray*

value of the property prop for every monolayer

class xrayutilities.simpack.darwin_theory.**DarwinModelGaInAs001** (qz, qx=0, qy=0, **kwargs)

Bases: **DarwinModelAlloy**

Darwin theory of diffraction for Ga_{1-x} In_x As layers. The model is based on separation of the sample structure into building blocks of atomic planes from which a multibeam dynamical model is calculated.

GaAs = *<xrayutilities.materials.material.Crystal object>*

InAs = *<xrayutilities.materials.material.Crystal object>*

aGaAs = *5.65325*

classmethod **abulk(**x**)**
calculate the bulk (relaxed) lattice parameter of the Ga_{1-x}In_{x}As alloy

asub = *5.65325*

eAs = *As (33)*

eGa = *Ga (31)*

eIn = *In (49)*

classmethod **get_dperp_apar(**x**,** apar**,** r=1**)**
calculate inplane lattice parameter and the out of plane lattice plane spacing (of the atomic planes!) from composition and relaxation

Parameters: **x :** *float*
chemical composition parameter

apar : *float*
inplane lattice parameter of the material below the current layer (onto which the present layer is strained to). This value also served as a reference for the relaxation parameter.

r : *float*
relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: **dperp :** *float*
perpendicular d-spacing

apar : *float*
inplane lattice parameter

init_structurefactors(temp=300**)**
calculates the needed atomic structure factors

Parameters: **temp :** *float, optional*
temperature used for the Debye model

re = *2.8179403262e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelSiGe001**(qz, qx=0, qy=0, **kwargs)
Bases: **DarwinModelAlloy**
model class implementing the Darwin theory of diffraction for SiGe layers. The model is based on separation of the sample structure into building blocks of atomic planes from which a multibeam dynamical model is calculated.

Ge = *<xrayutilities.materials.material.Crystal object>*

Si = *<xrayutilities.materials.material.Crystal object>*

aSi = *5.43104*

classmethod **abulk(**x**)**
calculate the bulk (relaxed) lattice parameter of the alloy

asub = *5.43104*

eGe = *Ge (32)*

eSi = *Si (14)*

classmethod **get_dperp_apar(**x**,** apar**,** r=1**)**

calculate inplane lattice parameter and the out of plane lattice plane spacing (of the atomic planes!) from composition and relaxation

Parameters: **x :** *float*
chemical composition parameter
apar : *float*
inplane lattice parameter of the material below the current layer (onto which the present layer is strained to). This value also served as a reference for the relaxation parameter.
r : *float, optional*
relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: **dperp :** *float*
perpendicular d-spacing
apar : *float*
inplane lattice parameter

init_structurefactors (temp=300)
calculates the needed atomic structure factors

Parameters: **temp :** *float, optional*
temperature used for the Debye model

re = *2.8179403262e-05*

xrayutilities.simpack.darwin_theory.**GradedBuffer** (xfrom, xto, nsteps, thickness, relaxation=1)
create a multistep graded composition buffer.

Parameters: **xfrom :** *float*
begin of the composition gradient
xto : *float*
end of the composition gradient
nsteps : *int*
number of steps of the gradient
thickness : *float*
total thickness of the Buffer in A
relaxation : *float*
relaxation of the buffer

Returns: **list**
layer list needed for the Darwin model simulation

xrayutilities.simpack.darwin_theory.**getfirst** (iterable, key)
helper function to obtain the first item in a nested iterable

xrayutilities.simpack.darwin_theory.**getit** (it, key)
generator to obtain items from nested iterable

xrayutilities.simpack.fit module

class xrayutilities.simpack.fit.**FitModel** (lmodel, verbose=False, plot=False, elog=True, **kwargs)
Bases: **Model**
Wrapper for the lmfit Model class working for instances of LayerModel
Typically this means that after initialization of *FitModel* you want to use make_params to get a *lmfit.Parameters* list which one customizes for fitting.
Later on you can call *fit* and *eval* methods with those parameter list.

__init__ (lmodel, verbose=False, plot=False, elog=True, **kwargs)

initialization of a FitModel which uses lmfit for the actual fitting, and generates an according lmfit.Model internally for the given pre-configured LayerModel, or subclasses thereof which includes models for reflectivity, kinematic and dynamic diffraction.

Parameters:

lmodel : *LayerModel*

pre-configured instance of LayerModel or any subclass

verbose : *bool, optional*

flag to enable verbose printing during fitting

plot : *bool or str, optional*

flag to decide wheter an plot should be created showing the fit's progress. If plot is a string it will be used as figure name, which makes reusing the figures easier.

elog : *bool, optional*

flag to enable a logarithmic error metric between model and data. Since the dynamic range of data is often large its often benefitial to keep this enabled.

kwargs : *dict, optional*

additional keyword arguments are forwarded to the *simulate* method of *lmodel*

fit (`data`**,** `params`**,** `x`**,** `weights`=None**,** `fit_kws`=None**,** `lmfit_kws`=None**,** `**kwargs`**)**

wrapper around lmfit.Model.fit which enables plotting during the fitting

Parameters:

data : *ndarray*

experimental values

params : *lmfit.Parameters*

list of parameters for the fit, use make_params for generation

x : *ndarray*

independent variable (incidence angle or q-position depending on the model)

weights : *ndarray, optional*

values of weights for the fit, same size as data

fit_kws : *dict, optional*

Options to pass to the minimizer being used

lmfit_kws : *dict, optional*

keyword arguments which are passed to lmfit.Model.fit

kwargs : *dict, optional*

keyword arguments passed to lmfit.Model.eval

Returns: **lmfit.ModelResult**

set_fit_limits (`xmin`=-inf**,** `xmax`=inf**,** `mask`=None**)**

set fit limits. If mask is given it must have the same size as the *data* and *x* variables given to fit. If mask is None it will be generated from xmin and xmax.

Parameters:

xmin : *float, optional*

minimum value of x-values to include in the fit

xmax : *float, optional*

maximum value of x-values to include in the fit

mask : *boolean array, optional*

mask to be used for the data given to the fit

xrayutilities.simpack.helpers module

`xrayutilities.simpack.helpers.`**coplanar_alphaf** (qx, qz, en='config', nan_Laue=False)

calculate coplanar exit angle from knowledge of the qx (inplane) and qz (out of plane) coordinates

Parameters: **qx :** *array-like*
inplane momentum transfer component
qz : *array-like*
out of plane momentum transfer component
en : *float or str, optional*
x-ray energy (eV). By default the value from the config is used.
nan_Laue : *bool, optional*
set output points inside the Laue zone to ‘nan’. default: False
Returns: **alphaf :** *array-like*
the exit angle in degree. points in the Laue zone are set to ‘nan’.

`xrayutilities.simpack.helpers.`**`coplanar_alphai`**(qx, qz, en='config', nan_Laue=False)
calculate coplanar incidence angle from knowledge of the qx (inplane) and qz (out of plane) coordinates

Parameters: **qx :** *array-like*
inplane momentum transfer component
qz : *array-like*
out of plane momentum transfer component
en : *float or str, optional*
x-ray energy (eV). By default the value from the config is used.
nan_Laue : *bool, optional*
set output points inside the Laue zone to ‘nan’. default: False
Returns: **alphai :** *array-like*
the incidence angle in degree

`xrayutilities.simpack.helpers.`**`get_qz`**(qx, alphai, en='config')
calculate the qz position from the qx position and the incidence angle for a coplanar diffraction geometry

Parameters: **qx :** *array-like*
inplane momentum transfer component
alphai : *array-like*
incidence angle (deg)
en : *float or str, optional*
x-ray energy (eV). By default the value from the config is used.
Returns: **array-like**
the qz position for the given incidence angle. The output is nan in case the inplane momentum transfer can not be reached.

xrayutilities.simpack.models module

class `xrayutilities.simpack.models.`**`DiffuseReflectivityModel`**(*args, **kwargs)
Bases: **`SpecularReflectivityModel`**
model for diffuse reflectivity calculations
The ‘simulate’ method calculates the diffuse reflectivity on the specular rod in coplanar geometry in analogy to the SpecularReflectivityModel.
The ‘simulate_map’ method calculates the diffuse reflectivity for a 2D set of Q-positions. This method can also calculate the intensity for other geometries, like GISAXS with constant incidence angle or a quasi omega/2theta scan in GISAXS geometry.

`__init__`(*args, **kwargs)
constructor for a reflectivity model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: **args :** *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

kwargs : *dict*

optional parameters for the simulation; supported are:

I0 : *float, optional*

the primary beam intensity

background : *float, optional*

the background added to the simulation

sample_width : *float, optional*

width of the sample along the beam

beam_width : *float, optional*

beam width in the same units as the sample width

resolution_width : *float, optional*

defines the width of the resolution (deg)

energy : *float, optional*

sets the experimental energy (eV)

H : *float, optional*

Hurst factor defining the fractal dimension of the roughness (0..1, very slow for H != 1 or H != 0.5), default: 1

vert_correl : *float, optional*

vertical correlation length in angstrom, 0 means full replication

vert_nu : *float, optional*

exponent in the vertical correlation function

method : *int, optional*

1..simple DWBA (default), 2..full DWBA (slower)

vert_int : *int, optional*

0..no integration over the vertical divergence, 1..with integration over the vertical divergence

qL_zero : *float, optional*

value of inplane q-coordinate which can be considered 0, using method 2 it is important to avoid exact 0 and this value will be used instead

simulate (alphai**)**

performs the actual diffuse reflectivity calculation for the specified incidence angles. This method always uses the coplanar geometry independent of the one set during the initialization.

Parameters: **alphai :** *array-like*

vector of incidence angles

Returns: **array-like**

vector of intensities of the reflectivity signal

simulate_map (qL, qz**)**

performs diffuse reflectivity calculation for the rectangular grid of reciprocal space positions define by qL and qz. This method uses the method and geometry set during the initialization of the class.

Parameters: **qL :** *array-like*

lateral coordinate in reciprocal space (vector with NqL components)

qz : *array-like*

vertical coordinate in reciprocal space (vector with Nqz components)

Returns: **array-like**

matrix of intensities of the reflectivity signal, with shape (len(qL), len(qz))

class xrayutilities.simpack.models.**DynamicalModel**(*args, **kwargs)

Bases: **SimpleDynamicalCoplanarModel**

Dynamical diffraction model for specular and off-specular qz-scans. Calculation of the flux of reflected and diffracted waves for general asymmetric coplanar diffraction from an arbitrary pseudomorphic multilayer is performed by a generalized 2-beam theory (4 tiepoints, S and P polarizations)

The first layer in the model is always assumed to be the semiinfinite substrate indepentent of its given thickness

simulate(alphai**,** hkl=None**,** geometry='hi_lo'**,** rettype='intensity'**)**

performs the actual diffraction calculation for the specified incidence angles and uses an analytic solution for the quartic dispersion equation

Parameters:

alphai : *array-like*

vector of incidence angles (deg)

hkl : *list or tuple, optional*

Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry : *{'hi_lo', 'lo_hi'}, optional*

'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

rettype : *{'intensity', 'field', 'all'}, optional*

type of the return value. 'intensity' (default): returns the diffracted beam flux convoluted with the resolution function; 'field': returns the electric field (complex) without convolution with the resolution function, 'all': returns the electric field, ai, af (both in degree), and the reflected intensity.

Returns:

array-like

vector of intensities of the diffracted signal, possibly changed return value due the rettype setting!

class xrayutilities.simpack.models.**DynamicalReflectivityModel**(*args, **kwargs)

Bases: **SpecularReflectivityModel**

model for Dynamical Specular Reflectivity Simulations. It uses the transfer Matrix methods as given in chapter 3 "Daillant, J., & Gibaud, A. (2008). X-ray and Neutron Reflectivity"

__init__(*args**,** **kwargs**)**

constructor for a reflectivity model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters:

args : *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

kwargs: dict

optional parameters for the simulation; supported are:

I0 : *float, optional*

the primary beam intensity

background : *float, optional*

the background added to the simulation

sample_width : *float, optional*

width of the sample along the beam

beam_width : *float, optional*

beam width in the same units as the sample width

resolution_width : *float, optional*

width of the resolution (deg)

energy : *float or str*

x-ray energy in eV

polarization: ['P', 'S']

x-ray polarization

scanEnergy (energies**,** angle**)**
Simulates the Dynamical Reflectivity as a function of photon energy at fixed angle.

Parameters: **energies: numpy.ndarray or list**
photon energies (in eV).
angle : *float*
fixed incidence angle
Returns: **reflectivity: array-like**
vector of intensities of the reflectivity signal
transmitivity: array-like
vector of intensities of the transmitted signal

simulate (alphai**)**
Simulates the Dynamical Reflectivity as a function of angle of incidence

Parameters: **alphai :** *array-like*
vector of incidence angles
Returns: **reflectivity: array-like**
vector of intensities of the reflectivity signal
transmitivity: array-like
vector of intensities of the transmitted signal

class xrayutilities.simpack.models.**KinematicalModel**(*args, **kwargs)
Bases: **LayerModel**
Kinematical diffraction model for specular and off-specular qz-scans. The model calculates the kinematical contribution of one (hkl) Bragg peak, however considers the variation of the structure factor for different ‘q’. The surface geometry is specified using the Experiment-object given to the constructor.

__init__ (*args**,** **kwargs**)**
constructor for a kinematic thin film model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: ***args :** *LayerStack or Layers*
either one LayerStack or several Layer objects can be given
****kwargs :** *dict*
optional parameters; also see LayerModel/Model.
experiment : *Experiment*
Experiment class containing geometry and energy of the experiment.

init_chi0 ()
calculates the needed optical parameters for the simulation. If any of the materials/layers is changing its properties this function needs to be called again before another correct simulation is made. (Changes of thickness does NOT require this!)

simulate (qz**,** hkl**,** absorption=False**,** refraction=False**,** rettype='intensity'**)**
performs the actual kinematical diffraction calculation on the Qz positions specified considering the contribution from a single Bragg peak.

Parameters: **qz :** *array-like*

simulation positions along qz

hkl : *list or tuple*

Miller indices of the Bragg peak whos truncation rod should be calculated

absorption : *bool, optional*

flag to tell if absorption correction should be used

refraction : *bool, optional*

flag to tell if basic refraction correction should be performed. If refraction is True absorption correction is also included independent of the absorption flag.

rettype : *{'intensity', 'field', 'all'}*

type of the return value. 'intensity' (default): returns the diffracted beam flux convoluted with the resolution function; 'field': returns the electric field (complex) without convolution with the resolution function, 'all': returns the electric field, ai, af (both in degree), and the reflected intensity.

Returns: **array-like**

return value depends on the setting of *rettype*, by default only the calculate intensity is returned

class xrayutilities.simpack.models.**KinematicalMultiBeamModel**(*args, **kwargs)

Bases: **KinematicalModel**

Kinematical diffraction model for specular and off-specular qz-scans. The model calculates the kinematical contribution of several Bragg peaks on the truncation rod and considers the variation of the structure factor. In order to use a analytical description for the kinematic diffraction signal all layer thicknesses are changed to a multiple of the respective lattice parameter along qz. Therefore this description only works for (001) surfaces.

__init__(*args, **kwargs**)**

constructor for a kinematic thin film model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: ***args :** *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

****kwargs :** *dict*

optional parameters. see also LayerModel/Model.

experiment : *Experiment*

Experiment class containing geometry and energy of the experiment.

surface_hkl : *list or tuple*

Miller indices of the surface (default: (0, 0, 1))

simulate(qz**,** hkl**,** absorption=False**,** refraction=True**,** rettype='intensity'**)**

performs the actual kinematical diffraction calculation on the Qz positions specified considering the contribution from a full truncation rod

Parameters: **qz :** *array-like*

simulation positions along qz

hkl : *list or tuple*

Miller indices of the Bragg peak whos truncation rod should be calculated

absorption : *bool, optional*

flag to tell if absorption correction should be used

refraction : *bool, optional,*

flag to tell if basic refraction correction should be performed. If refraction is True absorption correction is also included independent of the absorption flag.

rettype : *{'intensity', 'field', 'all'}*

type of the return value. 'intensity' (default): returns the diffracted beam flux convoluted with the resolution function; 'field': returns the electric field (complex) without convolution with the resolution function, 'all': returns the electric field, ai, af (both in degree), and the reflected intensity.

Returns: **array-like**

return value depends on the setting of *rettype*, by default only the calculate intensity is returned

class xrayutilities.simpack.models.**LayerModel**(*args, **kwargs)

Bases: **Model**, **ABC**

generic model class from which further thin film models can be derived from

__init__ (*args, **kwargs**)**

constructor for a thin film model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: ***args :** *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

****kwargs :** *dict*

optional parameters for the simulation. ones not listed below are forwarded to the superclass.

experiment : *Experiment, optional*

class containing geometry and energy of the experiment.

polarization: {'S', 'P', 'both'}, optional

polarization of the x-ray beam. If set to 'both' also Cmono, the polarization factor of the monochromator should be set

Cmono : *float, optional*

polarization factor of the monochromator

get_polarizations ()

return list of polarizations which should be calculated

join_polarizations (Is, Ip**)**

method to calculate the total diffracted intensity from the intensities of S and P-polarization.

***abstract* simulate ()**

abstract method that every implementation of a LayerModel has to override.

class xrayutilities.simpack.models.**Model**(experiment, **kwargs)

Bases: **object**

generic model class from which further models can be derived from

__init__ (experiment**,** **kwargs**)**

constructor for a generical simulation model. currently only the experiment class describing the diffraction geometry is stored in the base class

Parameters: **experiment :** *Experiment*
class describing the diffraction geometry, energy and wavelength of the model
resolution_width : *float, optional*
defines the width of the resolution
I0 : *float, optional*
the primary beam flux/intensity
background : *float, optional*
the background added to the simulation
energy : *float or str*
the experimental energy in eV
resolution_type : *{'Gauss', 'Lorentz'}, optional*
type of resolution function, default: Gauss

convolute_resolution(x, y**)**
convolve simulation result with a resolution function

Parameters: **x :** *array-like*
x-values of the simulation, units of x also decide about the unit of the resolution_width parameter
y : *array-like*
y-values of the simulation
Returns: **array-like**
convoluted y-data with same shape as y

property **energy**

scale_simulation(y**)**
scale simulation result with primary beam flux/intensity and add a background.

Parameters: **y :** *array-like*
y-values of the simulation
Returns: **array-like**
scaled y-values

class xrayutilities.simpack.models.**ResonantReflectivityModel**(*args, **kwargs)
Bases: **SpecularReflectivityModel**
model for specular reflectivity calculations CURRENTLY UNDER DEVELOPEMENT! DO NOT USE!

__init__(*args, **kwargs**)**
constructor for a reflectivity model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: **args :** *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

kwargs: dict

optional parameters for the simulation; supported are:

I0 : *float, optional*

the primary beam intensity

background : *float, optional*

the background added to the simulation

sample_width : *float, optional*

width of the sample along the beam

beam_width : *float, optional*

beam width in the same units as the sample width

resolution_width : *float, optional*

width of the resolution (deg)

energy : *float or str*

x-ray energy in eV

polarization: ['P', 'S']

x-ray polarization

simulate(alphai**)**

performs the actual reflectivity calculation for the specified incidence angles

Parameters: **alphai :** *array-like*

vector of incidence angles

Returns: **array-like**

vector of intensities of the reflectivity signal

class xrayutilities.simpack.models.**SimpleDynamicalCoplanarModel**(*args, **kwargs)

Bases: **KinematicalModel**

Dynamical diffraction model for specular and off-specular qz-scans. Calculation of the flux of reflected and diffracted waves for general asymmetric coplanar diffraction from an arbitrary pseudomorphic multilayer is performed by a simplified 2-beam theory (2 tiepoints, S and P polarizations)

No restrictions are made for the surface orientation.

The first layer in the model is always assumed to be the semiinfinite substrate indepentent of its given thickness

Note

This model should not be used in real life scenarios since the made approximations severely fail for distances far from the reference position.

__init__(*args, **kwargs**)**

constructor for a diffraction model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: ***args :** *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

****kwargs: dict**

optional parameters for the simulation

I0 : *float, optional*

the primary beam intensity

background : *float, optional*

the background added to the simulation

resolution_width : *float, optional*

the width of the resolution (deg)

polarization: {'S', 'P', 'both'}

polarization of the x-ray beam. If set to 'both' also Cmono, the polarization factor of the monochromator should be set

Cmono : *float, optional*

polarization factor of the monochromator

energy : *float or str*

the experimental energy in eV

experiment : *Experiment*

Experiment class containing geometry of the sample; surface orientation!

set_hkl(*hkl)

To speed up future calculations of the same Bragg peak optical parameters can be pre-calculated using this function.

Parameters: **hkl :** *list or tuple*

Miller indices of the Bragg peak for the calculation

simulate(alphai, hkl=None, geometry='hi_lo', idxref=1)

performs the actual diffraction calculation for the specified incidence angles.

Parameters: **alphai :** *array-like*

vector of incidence angles (deg)

hkl : *list or tuple, optional*

Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry : *{'hi_lo', 'lo_hi'}, optional*

'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

idxref : *int, optional*

index of the reference layer. In order to get accurate peak position of the film peak you want this to be the index of the film peak (default: 1). For the substrate use 0.

Returns: **array-like**

vector of intensities of the diffracted signal

class xrayutilities.simpack.models.**SpecularReflectivityModel**(*args, **kwargs)

Bases: **LayerModel**

model for specular reflectivity calculations

__init__(*args, **kwargs)

constructor for a reflectivity model. The arguments consist of a LayerStack or individual Layer(s). Optional parameters are specified in the keyword arguments.

Parameters: **args :** *LayerStack or Layers*

either one LayerStack or several Layer objects can be given

kwargs: dict

optional parameters for the simulation; supported are:

I0 : *float, optional*

the primary beam intensity

background : *float, optional*

the background added to the simulation

sample_width : *float, optional*

width of the sample along the beam

beam_width : *float, optional*

beam width in the same units as the sample width

beam_shape : *str, optional*

beam_shape can be either 'hat' (default) or 'gaussian'. beam_width will be accordingly interpreted as width of the hat function or sigma of the Gaussian function.

offset : *float, optional*

angular offset of the incidence angle (deg)

resolution_width : *float, optional*

width of the resolution (deg)

energy : *float or str*

x-ray energy in eV

`densityprofile` **(**`nz`**,** `plot=False`**,** `individual_layers=False`**)**

calculates the electron density of the layerstack from the thickness and roughness of the individual layers

Parameters: **nz :** *int*

number of values on which the profile should be calculated

plot : *bool, optional*

flag to tell if a plot of the profile should be created

individual_layers : *bool, optional*

return the density contributions of all layers as additional return value.

Returns: **z :** *array-like*

z-coordinates, z = 0 corresponds to the surface

eprof : *array-like*

electron profile

layereprof : *2D array, optional*

electron profile of every sublayer with shape (nlayer, nz). This is only returned when individual_layers=True

`init_cd` **()**

calculates the needed optical parameters for the simulation. If any of the materials/layers is changing its properties this function needs to be called again before another correct simulation is made. (Changes of thickness and roughness do NOT require this!)

`simulate` **(**`alphai`**)**

performs the actual reflectivity calculation for the specified incidence angles

Parameters: **alphai :** *array-like*

vector of incidence angles

Returns: **array-like**

vector of intensities of the reflectivity signal

xrayutilities.simpack.models.**effectiveDensitySlicing**(layerstack, step, roughness=0, cutoff=1e-05)

Function to slice a LayerStack into many amorphous sublayers for effective density modelling of X-ray reflectivity of thin and rough multilayers. The resulting LayerStack will consist of perfectly smooth layers with average density/composition resulting from an error-function like transition between the rough layers of the initial stack. At the surface an vacuum layer is automatically added to the initial stack.

Parameters:
layerstack : *initial LayerStack, can contain only Amorhous layers!*

step : *thickness (in angstrom) of the slices in the returned* LayerStack

roughness : *roughness of the created sublayers (in angstrom)*

cutoff : *layers with relative weights below this value will be ignored*

Returns:
LayerStack

xrayutilities.simpack.models.**startdelta**(start, delta, num)

xrayutilities.simpack.mosaicity module

xrayutilities.simpack.mosaicity.**mosaic_analytic**(qx, qz, RL, RV, Delta, hx, hz, shape)

simulation of the coplanar reciprocal space map of a single mosaic layer using a simple analytic approximation

Parameters:
qx : *array-like*
vector of the qx values (offset from the Bragg peak)

qz : *array-like*
vector of the qz values (offset from the Bragg peak)

RL : *float*
lateral block radius in angstrom

RV : *float*
vertical block radius in angstrom

Delta : *float*
root mean square misorientation of the grains in degree

hx : *float*
lateral component of the diffraction vector

hz : *float*
vertical component of the diffraction vector

shape : *float*
shape factor (1..Gaussian)

Returns:
array-like

2D array with calculated intensities

xrayutilities.simpack.powder module

This module contains the core definitions for the XRD Fundamental Parameneters Model (FPA) computation in Python. The main computational class is FP_profile, which stores cached information to allow it to efficiently recompute profiles when parameters have been modified. For the user an Powder class is available which can calculate a complete powder pattern of a crystalline material.

The diffractometer line profile functions are calculated by methods from Cheary & Coelho 1998 and Mullen & Cline paper and 'R' package. Accumulate all convolutions in Fourier space, for efficiency, except for axial divergence, which needs to be weighted in real space for I3 integral.

More details about the applied algorithms can be found in the paper by M. H. Mendelhall et al., Journal of Research of NIST 120, 223 (2015) to which you should also refer for a careful definition of all the parameters

class xrayutilities.simpack.powder.**FP_profile** (anglemode, gaussian_smoother_bins_sigma=1.0, oversampling=10)

Bases: **object**

the main fundamental parameters class, which handles a single reflection. This class is designed to be highly extensible by inheriting new convolvers. When it is initialized, it scans its namespace for specially formatted names, which can come from mixin classes. If it finds a function name of the form conv_xxx, it will call this funtion to create a convolver. If it finds a name of the form info_xxx it will associate the dictionary with that convolver, which can be used in UI generation, for example. The class, as it stands, does nothing significant with it. If it finds str_xxx, it will use that function to format a printout of the current state of the convolver conv_xxx, to allow improved report generation for convolvers.

When it is asked to generate a profile, it calls all known convolvers. Each convolver returns the Fourier transform of its convolvution. The transforms are multiplied together, inverse transformed, and after fixing the periodicity issue, subsampled, smoothed and returned.

If a convolver returns *None*, it is not multipled into the product.

Parameters: **max_history_length :** *int*

the number of histories to cache (default=5); can be overridden if memory is an issue.

length_scale_m : *float*

length_scale_m sets scaling for nice printing of parameters. if the units are in mm everywhere, set it to 0.001, e.g. convolvers which implement their own str_xxx method may use this to format their results, especially if 'natural' units are not meters. Typical is wavelengths and lattices in nm or angstroms, for example.

__init__ (anglemode**,** gaussian_smoother_bins_sigma=1.0**,** oversampling=10**)**

initialize the instance

Parameters: **anglemode :** *{'d', 'twotheta'}*

if setup will be in terms of a d-spacing, otherwise 'twotheta' if setup will be at a fixed 2theta value.

gaussian_smoother_bins_sigma : *float*

the number of bins for post-smoothing of data. 1.0 is good. *None* means no final smoothing step.

oversampling : *int*

the number of bins internally which will get computed for each bin the the final result.

add_buffer (b**)**

add a numpy array to the list of objects that can be thrown away on pickling.

Parameters: **b :** *array-like*

the buffer to add to the list

Returns: **b :** *array-like*

return the same buffer, to make nesting easy.

axial_helper (outerbound**,** innerbound**,** epsvals**,** destination**,** peakpos=0**,** y0=0**,** k=0**)**

the function F0 from the paper. compute k/sqrt(peakpos-x)+y0 nonzero between outer & inner (inner is closer to peak) or k/sqrt(x-peakpos)+y0 if reversed (i.e. if outer > peak) fully evaluated on a specified eps grid, and stuff into destination

Parameters: **outerbound :** *float*
the edge of the function farthest from the singularity, referenced to epsvals
innerbound : *float*
the edge closest to the singularity, referenced to epsvals
epsvals : *array-like*
the array of two-theta values or offsets
destination : *array-like*
an array into which final results are summed. modified in place!
peakpos : *float*
the position of the singularity, referenced to epsvals.
y0 : *float*
the constant offset
k : *float*
the scale factor
Returns: **lower_index, upper_index :** *int*
python style bounds for region of *destination* which has been modified.

compute_line_profile **(**convolver_names=None**,** compute_derivative=False**,** return_convolver=False**)**
execute all the convolutions; if convolver_names is None, use everything we have, otherwise, use named convolutions.

Parameters: **convolver_names: list**
a list of convolvers to select. If *None*, use all found convolvers.
compute_derivative: bool
if *True*, also return d/dx(function) for peak position fitting
Returns: **object**
a profile_data object with much information about the peak

conv_absorption ()
compute the sample transparency correction, including the finite-thickness version

Returns: **array-like**
the convolver

conv_axial ()
compute the Fourier transform of the axial divergence component

Returns: **array-like**
the transform buffer, or *None* if this is being ignored

conv_displacement ()
compute the peak shift due to sample displacement and the *2theta* zero offset

Returns: **array-like**
the convolver

conv_emission ()
compute the emission spectrum and (for convenience) the particle size widths

Returns: **array-like**
the convolver for the emission and particle sizes

> **Note**
>
> the particle size and strain stuff here is just to be consistent with *Topas* and to be vaguely efficient about the computation, since all of these have the same general shape.

conv_flat_specimen ()
compute the convolver for the flat-specimen correction

Returns: **array-like**
the convolver

conv_global ()
a dummy convolver to hold global variables and information. the global context isn't really a convolver, returning *None* means ignore result

Returns: ***None***
always returns None

conv_receiver_slit ()
compute the rectangular convolution for the receiver slit or SiPSD pixel size

Returns: **array-like**
the convolver

conv_si_psd ()
compute the convolver for the integral of defocusing of the face of an Si PSD

Returns: **array-like**
the convolver

conv_smoother ()
compute the convolver to smooth the final result with a Gaussian before downsampling.

Returns: **array-like**
the convolver

conv_tube_tails ()
compute the Fourier transform of the rectangular tube tails function

Returns: **array-like**
the transform buffer, or *None* if this is being ignored

full_axdiv_I2 (Lx=None**,** Ls=None**,** Lr=None**,** R=None**,** twotheta=None**,** beta=None**,** epsvals=None**)**
return the *I2* function

Parameters: **Lx :** *float*
length of the xray filament
Ls : *float*
length of the sample
Lr : *float*
length of the receiver slit
R : *float*
diffractometer length, assumed symmetrical
twotheta : *float*
angle, in radians, of the center of the computation
beta : *float*
offset angle
epsvals : *array-like*
array of offsets from center of computation, in radians

Returns: **epsvals :** *array-like*
array of offsets from center of computation, in radians
idxmin, idxmax : *int*
the full python-style bounds of the non-zero region of *I2p* and *I2m*
I2p, I2m : *array-like*
I2+ and I2- from the paper, the contributions to the intensity

full_axdiv_I3 **(**Lx=None**,** Ls=None**,** Lr=None**,** R=None**,** twotheta=None**,** epsvals=None**,** sollerIdeg=None**,** sollerDdeg=None**,** nsteps=10**,** axDiv=''**)**
carry out the integral of *I2* over *beta* and the Soller slits.

Parameters: **Lx :** *float*
length of the xray filament
Ls : *float*
length of the sample
Lr : *float*
length of the receiver slit
R : *float*
the (assumed symmetrical) diffractometer radius
twotheta : *float*
angle, in radians, of the center of the computation
epsvals : *array-like*
array of offsets from center of computation, in radians
sollerIdeg : *float*
the full-width (both sides) cutoff angle of the incident Soller slit
sollerDdeg : *float*
the full-width (both sides) cutoff angle of the detector Soller slit
nsteps : *int*
the number of subdivisions for the integral
axDiv : *str*
not used

Returns: **array-like**
the accumulated integral, a copy of a persistent buffer *_axial*

general_tophat (name=''**,** width=None**)**

a utility to compute a transformed tophat function and save it in a convolver buffer

Parameters: **name :** *str*

the name of the convolver cache buffer to update

width : *float*

the width in 2-theta space of the tophat

Returns: **array-like**

the updated convolver buffer, or *None* if the width was *None*

get_conv (name**,** key**,** format=<class 'float'>**)**

get a cached, pre-computed convolver associated with the given parameters, or a newly zeroed convolver if the cache doesn't contain it. Recycles old cache entries.

This takes advantage of the mutability of arrays. When the contents of the array are changed by the convolver, the cached copy is implicitly updated, so that the next time this is called with the same parameters, it will return the previous array.

Parameters: **name :** *str*

the name of the convolver to seek

key : *object*

any hashable object which identifies the parameters for the computation

format : *numpy.dtype, optional*

the type of the array to create, if one is not found.

Returns: **bool**

flag, which is *True* if valid data were found, or *False* if the returned array is zero, and *array*, which must be computed by the convolver if *flag* was *False*.

get_convolver_information ()

return a list of convolvers, and what we know about them. function scans for functions named conv_xxx, and associated info_xxx entries.

Returns: **list**

list of (convolver_xxx, info_xxx) pairs

get_function_name ()

return the name of the function that called this. Useful for convolvers to identify themselves

Returns: **str**

name of calling function

get_good_bin_count (count**)**

find a bin count close to what we need, which works well for Fourier transforms.

Parameters: **count :** *int*

a number of bins.

Returns: **int**

a bin count somewhat larger than *count* which is efficient for FFT

info_emission = *{'group_name': 'Incident beam and crystal size', 'help': 'this should be help information', 'param_info': {'crystallite_size_gauss': ('Gaussian crystallite size fwhm (m)', 1e-06), 'crystallite_size_lor': ('Lorentzian crystallite size fwhm (m)', 1e-06), 'emiss_gauss_widths': ('Gaussian emissions fwhm (m)', (1e-13,)), 'emiss_intensities': ('relative intensities', (1.0,)), 'emiss_lor_widths': ('Lorenztian emission fwhm (m)', (1e-13,)), 'emiss_wavelengths': ('wavelengths (m)', (1.58e-10,))}}*

info_global = *{'group_name': 'Global parameters', 'help': 'this should be help information', 'param_info': {'d': ('d spacing (m)', 4e-10), 'dominant_wavelength': ('wavelength of most intense line (m)', 1.5e-10), 'twotheta0_deg': ('Bragg center of peak (degrees)', 30.0)}}*

classmethod **isequivalent (**hkl1**,** hkl2**,** crystalsystem**)**
function to determine if according to the convolvers included in this class two sets of Miller indices are equivalent. This function is only called when the class attribute 'isotropic' is False.

Parameters: **hkl1, hkl2 :** *list or tuple*
Miller indices to be checked for equivalence
crystalsystem : *str*
symmetry class of the material which is considered

Returns: **bool**

isotropic = *True*

length_scale_m = *1.0*

max_history_length = *5*

self_clean ()
do some cleanup to make us more compact; Instance can no longer be used after doing this, but can be pickled.

set_optimized_window (twotheta_window_center_deg**,** twotheta_approx_window_fullwidth_deg**,** twotheta_exact_bin_spacing_deg**)**
pick a bin count which factors cleanly for FFT, and adjust the window width to preserve the exact center and bin spacing

Parameters: **twotheta_window_center_deg :** *float*
exact position of center bin, in degrees
twotheta_approx_window_fullwidth_deg: float
approximate desired width
twotheta_exact_bin_spacing_deg: float
the exact bin spacing to use

set_parameters (convolver='global'**,** ****kwargs**)**
update the dictionary of parameters associated with the given convolver

Parameters: **convolver :** *str*
the name of the convolver. name 'global', e.g., attaches to function 'conv_global'
kwargs : *dict*
keyword-value pairs to update the convolvers dictionary.

set_window (twotheta_window_center_deg**,** twotheta_window_fullwidth_deg**,** twotheta_output_points**)**
move the compute window to a new location and compute grids, without resetting all parameters. Clears convolution history and sets up many arrays.

Parameters: **twotheta_window_center_deg :** *float*
the center position of the middle bin of the window, in degrees
twotheta_window_fullwidth_deg : *float*
the full width of the window, in degrees
twotheta_output_points : *int*
the number of bins in the final output

str_emission ()
format the emission spectrum and crystal size information

Returns: **str**
the formatted information

str_global()
returns a string representation for the global context.

Returns: **str**
report on global parameters.

class xrayutilities.simpack.powder.**PowderDiffraction**(mat, **kwargs)
Bases: **PowderExperiment**
Experimental class for powder diffraction. This class calculates the structure factors of powder diffraction lines and uses instances of FP_profile to perform the convolution with experimental resolution function calculated by the fundamental parameters approach. This class uses multiprocessing to speed up calculation. Set config.NTHREADS=1 to restrict this to one worker process.

Calculate(twotheta**,** **kwargs**)**
calculate the powder diffraction pattern including convolution with the resolution function and map them onto the twotheta positions. This also performs the calculation of the peak intensities from the internal material object

Parameters: **twotheta :** *array-like*
two theta values at which the powder pattern should be calculated.
kwargs : *dict*
additional keyword arguments are passed to the Convolve function
Returns: **array-like**
output intensity values for the twotheta values given in the input

Notes
Bragg peaks are only included up to tt_cutoff set in the class constructor!

Convolve(twotheta**,** window_width='config'**,** mode='multi'**)**
convolute the powder lines with the resolution function and map them onto the twotheta positions. This calculates the powder pattern excluding any background contribution

Parameters: **twotheta :** *array-like*
two theta values at which the powder pattern should be calculated.
window_width : *float, optional*
width of the calculation window of a single peak
mode : *{'multi, 'local'}, optional*
multiprocessing mode, either 'multi' to use multiple processes or 'local' to restrict the calculation to a single process
Note:
Bragg peaks are only included up to tt_cutoff set in the class constructor!
Returns: **output intensity values for the twotheta values given in the input**

__init__(mat**,** **kwargs**)**
the class is initialized with a xrayutilities.materials.Crystal instance and calculates the powder intensity and peak positions of the Crystal up to an angle of tt_cutoff. Results are stored in

data …. array with intensities ang ….. Bragg angles of the peaks (Theta!) qpos …. reciprocal space position of intensities

If *enable_simulation* is set to True the *Convolve* and *Calculate* methods can be used to calculated a powder pattern. Upon such initialization it is strongly recommend to call the *close* method to clean up the multiprocessing threads when no further calculations will be performed.

Parameters: **mat :** *Crystal or Powder*

material for the powder calculation

kwargs : *dict*

optional keyword arguments same as for the Experiment base class and:

tt_cutoff : *float, optional*

Powder peaks are calculated up to an scattering angle of tt_cutoff (deg)

enable_simulation: False, optional

enables the initialization of the convolvers. Call *close()* after you are finished with your instance!

fpclass : *FP_profile*

FP_profile derived class with possible convolver mixins. (default: FP_profile)

fpsettings : *dict*

settings dictionaries for the convolvers. Default settings are loaded from the config file.

close ()

correction_factor (ang**)**

calculate the correction factor for the diffracted intensities. This contains the polarization effects and the Lorentz factor

Parameters: **ang :** *aray-like*

theta diffraction angles for which the correction should be calculated

Returns: **f :** *array-like*

array of the same shape as ang containing the correction factors

property **energy**

init_powder_lines (tt_cutoff**)**

calculates the powder intensity and positions up to an angle of tt_cutoff (deg) and stores the result in the data dictionary whose structure is as follows:

The data dictionary has one entry per line with a unique identifier as key of the entry. The entries themself are dictionaries which have the following entries:

- hkl : (h, k, l), Miller indices of the Bragg peak
- r : reflection strength of the line
- ang : Bragg angle of the peak (theta = 2theta/2!)
- qpos : reciprocal space position

load_settings_from_config (settings**)**

load parameters from the config and update these settings with the options from the settings parameter

merge_lines (data**)**

if calculation is isotropic lines at the same q-position can be merged to one line to reduce the calculational effort

Parameters: **data :** *ndarray*

numpy field array with values of ‘hkl’ (Miller indices of the peaks), ‘q’ (q-position), and ‘r’ (reflection strength) as produced by the *reflection_strength* method

Returns: **hkl, q, ang, r :** *array-like*

Miller indices, q-position, diffraction angle (Theta), and reflection strength of the material

reflection_strength (tt_cutoff**)**

determine structure factors/reflection strength of all Bragg peaks up to tt_cutoff. This function also implements the March-Dollase model for preferred orientation in the symmetric reflection mode. Note that although this means the sample has anisotropic properties the various lines can still be merged together since at the moment no anisotropic crystal shape is supported.

Parameters: **tt_cutoff :** *float*

upper cutoff value of 2theta until which the reflection strength are calculated

Returns: **ndarray**

numpy array with field for ‘hkl’ (Miller indices of the peaks), ‘q’ (q-position), and ‘r’ (reflection strength) of the Bragg peaks

set_sample_parameters ()
load sample parameters from the Powder class and use them in all FP_profile instances of this object

set_wavelength_from_params ()
sets the wavelenth in the base class from the settings dictionary of the FP_profile classes and also set it in the ‘global’ part of the parameters

set_window (force=False**)**
sets the calcultion window for all convolvers

property **twotheta**

update_powder_lines (tt_cutoff**)**
calculates the powder intensity and positions up to an angle of tt_cutoff (deg) and updates the values in:

- ids: list of unique identifiers of the powder line
- data: array with intensities
- ang: bragg angles of the peaks (theta=2theta/2!)
- qpos: reciprocal space position of intensities

update_settings (newsettings=None**)**
update settings of all instances of FP_profile

Parameters: **newsettings :** *dict, optional*

dictionary with new settings. It has to include one subdictionary for every convolver which should have its settings changed.

property **wavelength**

property **window_width**

xrayutilities.simpack.powder.**chunkify** (lst, n)

class xrayutilities.simpack.powder.**convolver_handler**
Bases: **object**
manage the convolvers of on process

__init__ ()

add_convolver (convolver**)**

calc (run, ttpeaks**)**
calculate profile function for selected convolvers

Parameters: **run :** *list*

list of flags of length of convolvers to tell which convolver needs to be run

ttpeaks : *array-like*

peak positions for the convolvers

Returns: **list**

list of profile_data result objects

set_windows (centers, npoints, flag, width**)**

update_parameters(parameters)

class xrayutilities.simpack.powder.**manager**(address=None, authkey=None, serializer='pickle', ctx=None, *, shutdown_timeout=1.0)
Bases: **BaseManager**

class xrayutilities.simpack.powder.**profile_data**(**kwargs)
Bases: **object**
a skeleton class which makes a combined dict and namespace interface for easy pickling and data passing

__init__(**kwargs)
initialize the class

Parameters: **kwargs :** *dict*
keyword=value list to pre-populate the class

add_symbol(**kwargs)
add new symbols to both the attributes and dictionary for the class

Parameters: **kwargs :** *dict*
keyword=value pairs

xrayutilities.simpack.powdermodel module

class xrayutilities.simpack.powdermodel.**PowderModel**(*args, **kwargs)
Bases: **object**
Class to help with powder calculations for multiple materials. For basic calculations the Powder class together with the Fundamental parameters approach is used.

__init__(*args, **kwargs)
constructor for a powder model. The arguments consist of a PowderList or individual Powder(s). Optional parameters are specified in the keyword arguments.

Parameters: **args :** *PowderList or Powders*
either one PowderList or several Powder objects can be given
kwargs : *dict*
optional parameters for the simulation. supported are:
fpclass : *FP_profile, optional*
derived class with possible convolver mixins. (default: FP_profile)
fpsettings : *dict*
settings dictionaries for the convolvers. Default settings are loaded from the config file.
I0 : *float, optional*
scaling factor for the simulation result

Notes

- After the end-of-use it is advisable to call the *close()* method to cleanup the multiprocessing calculation!
- In particular interesting keys in the fpsettings dictionary are listed in the following. Note that this a short excerpt of the full functionality:
  - ‘displacement’-dictionary with keys:
    - **‘specimen_displacement’: sample’s z-displacement from the** rotation center
    - ‘zero_error_deg’: zero error of the 2theta angle
  - ‘absorption’-dictionary with keys:

- ‘sample_thickness’: sample thickness (m),
- ‘absorption_coefficient’: sample’s absorption (m^-1)

- ‘axial’-dictionary with keys:
  - ‘length_sample’: sample length in the axial direction (m)

close ()

create_fitparameters ()
function to create a fit model with all instrument and sample parameters.

Returns: **lmfit.Parameters**

fit (params**,** twotheta**,** data**,** std=None**,** maxfev=200**)**
make least squares fit with parameters supplied by the user

Parameters: **params :** *lmfit.Parameters*
object with all parameters set as intended by the user

twotheta : *array-like*
angular values for the fit

data : *array-like*
experimental intensities for the fit

std : *array-like*
standard deviation of the experimental data. if ‘None’ the sqrt of the data will be used

maxfev: int
maximal number of simulations during the least squares refinement

Returns: **lmfit.MinimizerResult**

plot (twotheta**,** showlines=True**,** label='simulation'**,** color=None**,** formatspec='-'**,** lcolors=None**,** ax=None**,** **kwargs**)**
plot the powder diffraction pattern and indicate line positions for all components in the model.

Parameters: **twotheta :** *array-like*
positions at which the powder pattern should be evaluated

showlines : *bool, optional*
flag to decide if peak positions of the components will be shown on the top of the plot

label : *str*
line label in the plot

color : *matplotlib color or None*
the color used for the line plot of the simulation

formatspec : *str*
format specifier of the simulation curve

lcolors : *list of matplotlib colors*
colors for the line indicators for the various components

ax : *matplotlib.axes or None*
axes object to be used for plotting, if its given no axes decoration like labels are set.

Further keyword arguments are passed to the simulate method.

Returns: **matplotlib.axes object or None if matplotlib is not available**

set_background (btype**,** ****kwargs**)**
define background as spline or polynomial function

Parameters: **btype :** *{polynomial', 'spline'}*
background type; Depending on this value the expected keyword arguments differ.

kwargs : *dict*
optional keyword arguments

x : *array-like, optional*
x-values (twotheta) of the background points (if btype='spline')

y : *array-like, optional*
intensity values of the background (if btype='spline')

p : *array-like, optional*
polynomial coefficients from the highest degree to the constant term. len of p decides about the degree of the polynomial (if btype='polynomial')

set_lmfit_parameters (lmparams**)**
function to update the settings of this class during an least squares fit

Parameters: **lmparams :** *lmfit.Parameters*
lmfit Parameters list of sample and instrument parameters

set_parameters (params**)**
set simulation parameters of all subobjects

Parameters: **params :** *dict*
settings dictionaries for the convolvers.

simulate (twotheta**,** ****kwargs**)**
calculate the powder diffraction pattern of all materials and sum the results based on the relative volume of the materials.

Parameters: **twotheta :** *array-like*
positions at which the powder pattern should be evaluated

kwargs : *dict*
optional keyword arguments

background : *array-like*
an array of background values (same shape as twotheta) if no background is given then the background is calculated as previously set by the set_background function or is 0.

further keyword arguments are passed to the Convolve function of of the

PowderDiffraction objects

Returns: **array-like**
summed powder diffraction intensity of all materials present in the model

xrayutilities.simpack.powdermodel.**Rietveld_error_metrics** (exp, sim, weight=None, std=None, Nvar=0, disp=False)
calculates common error metrics for Rietveld refinement.

Parameters: **exp :** *array-like*

experimental datapoints

sim : *array-like*

simulated data

weight : *array-like, optional*

weight factor in the least squares sum. If it is None the weight is estimated from the counting statistics of ‘exp’

std : *array-like, optional*

standard deviation of the experimental data. alternative way of specifying the weight factor. when both are given weight overwrites std!

Nvar : *int, optional*

number of variables in the refinement

disp : *bool, optional*

flag to tell if a line with the calculated values should be printed.

Returns: **M, Rp, Rwp, Rwpexp, chi2: float**

`xrayutilities.simpack.powdermodel.`**`plot_powder`**(`twotheta`, `exp`, `sim`, `mask`=None, `scale`='sqrt', `fig`='XU:powder', `show_diff`=True, `show_legend`=True, `labelexp`='experiment', `labelsim`='simulation', `formatexp`='.-k', `formatsim`='-r')

Convenience function to plot the comparison between experimental and simulated powder diffraction data

Parameters: **twotheta :** *array-like*

angle values used for the x-axis of the plot (deg)

exp : *array-like*

experimental data (same shape as twotheta). If None only the simulation and no difference will be plotted

sim : *array-like or PowderModel*

simulated data or PowderModel instance. If a PowderModel instance is given the plot-method of PowderModel is used.

mask : *array-like, optional*

mask to reduce the twotheta values to the be used as x-coordinates of sim

scale : *{‘linear’, ‘sqrt’, ‘log’}, optional*

string specifying the scale of the y-axis.

fig : *str or int, optional*

matplotlib figure name (figure will be cleared!)

show_diff : *bool, optional*

flag to specify if a difference curve should be shown

show_legend: bool, optional

flag to specify if a legend should be shown

labelexp : *str*

plot label (legend entry) for the experimental data

labelsim : *str*

plot label for the simulation data

formatexp : *str*

format specifier for the experimental data

formatsim : *str*

format specifier for the simulation curve

Returns: **List of lines in the plot. Empty list in case matplotlib can’t be imported**

xrayutilities.simpack.smaterials module

class xrayutilities.simpack.smaterials.**CrystalStack**(name, *args)
Bases: **LayerStack**
extends the built in list type to enable building a stack of crystalline Layers by various methods.

check(v)

class xrayutilities.simpack.smaterials.**GradedLayerStack**(alloy, xfrom, xto, nsteps, thickness, **kwargs)
Bases: **CrystalStack**
generates a sequence of layers with a gradient in chemical composition

__init__(alloy, xfrom, xto, nsteps, thickness, **kwargs)
constructor for a graded buffer of the material 'alloy' with chemical composition from 'xfrom' to 'xto' with 'nsteps' number of sublayers. The total thickness of the graded buffer is 'thickness'

Parameters:
alloy : *function*
Alloy function which allows to create a material with chemical composition 'x' by alloy(x)
xfrom, xto : *float*
chemical composition from the bottom to top
nsteps : *int*
number of sublayers in the graded buffer
thickness : *float*
total thickness of the graded stack

class xrayutilities.simpack.smaterials.**Layer**(material, thickness, **kwargs)
Bases: **SMaterial**
Object describing part of a thin film sample. The properties of a layer are :

Attributes:
material : *Material (Crystal or Amorhous)*
an xrayutilties material describing optical and crystal properties of the thin film
thickness : *float*
film thickness in angstrom

__init__(material, thickness, **kwargs)
constructor for the material saving its properties

Parameters:
material : *Material (Crystal or Amorhous)*
an xrayutilties material describing optical and crystal properties of the thin film
thickness : *float*
film thickness in angstrom
kwargs : *dict*
optional keyword arguments with further layer properties.
roughness : *float, optional*
root mean square roughness of the top interface in angstrom
density : *float, optional*
density of the material in kg/m^3; If not specified the density of the material will be used.
relaxation : *float, optional*
the degree of relaxation in case of crystalline thin films
lat_correl : *float, optional*
the lateral correlation length for diffuse reflectivity calculations

class xrayutilities.simpack.smaterials.**LayerStack**(name, *args)

Bases: **MaterialList**
extends the built in list type to enable building a stack of Layer by various methods.

check (v)

class xrayutilities.simpack.smaterials.**MaterialList** (name, *args)
Bases: **MutableSequence**
class representing the basics of a list of materials for simulations within xrayutilities. It extends the built in list type.

__init__ (name, *args)

check (v)

insert (i, v)
S.insert(index, value) – insert value before index

class xrayutilities.simpack.smaterials.**Powder** (material, volume, **kwargs)
Bases: **SMaterial**
Object describing part of a powder sample. The properties of a powder are:

Attributes:

material : *Crystal*
an xrayutilties material (Crystal) describing optical and crystal properties of the powder

volume : *float*
powder's volume (in pseudo units, since only the relative volume enters the calculation)

crystallite_size_lor : *float, optional*
Lorentzian crystallite size fwhm (m)

crystallite_size_gauss : *float, optional*
Gaussian crystallite size fwhm (m)

strain_lor : *float, optional*
extra peak width proportional to tan(theta)

strain_gauss : *float, optional*
extra peak width proportional to tan(theta)

preferred_orientation : *tuple, optional*
HKL of the preferred orientation

preferred_orientation_factor : *float, optional*
March-Dollase preferred orientation factor: < 1 for platy crystallits , > 1 for rod-like crystallites, and = 1 for random orientation of crystallites.

__init__ (material, volume, **kwargs)
constructor for the material saving its properties

Parameters:

material : *Crystal*
an xrayutilties material (Crystal) describing optical and crystal properties of the powder

volume : *float*
powder's volume (in pseudo units, since only the relative volume enters the calculation)

kwargs : *dict*
optional keyword arguments with further powder properties.

crystallite_size_lor : *float, optional*
Lorentzian crystallite size fwhm (m)

crystallite_size_gauss : *float, optional*
Gaussian crystallite size fwhm (m)

strain_lor, strain_gauss : *float, optional*
extra peak width proportional to tan(theta); typically interpreted as microstrain broadening

class xrayutilities.simpack.smaterials.**PowderList**(name, *args)
Bases: **MaterialList**
extends the built in list type to enable building a list of Powder by various methods.

check(v)

class xrayutilities.simpack.smaterials.**PseudomorphicStack001**(name, *args)
Bases: **CrystalStack**
generate a sequence of pseudomorphic crystalline Layers. Surface orientation is assumed to be 001 and materials must be cubic/tetragonal.

insert(i, v)
S.insert(index, value) – insert value before index

make_epitaxial(i)
Make the i-th sublayer pseudomorphic to the layer below.

trans = *<xrayutilities.math.transforms.Transform object>*

class xrayutilities.simpack.smaterials.**PseudomorphicStack111**(name, *args)
Bases: **PseudomorphicStack001**
generate a sequence of pseudomorphic crystalline Layers. Surface orientation is assumed to be 111 and materials must be cubic.

trans = *<xrayutilities.math.transforms.CoordinateTransform object>*

class xrayutilities.simpack.smaterials.**SMaterial**(material, name=None, **kwargs)
Bases: **object**
Simulation Material. Extends the xrayutilities Materials by properties needed for simulations

__**init**__(material, name=None, **kwargs)
initialize a simulation material by specifiying its Material and optional other properties

Parameters:
material : *Material (Crystal, or Amorphous)*
Material object containing optical/crystal properties of for the simulation; a deepcopy is used internally.
name : *str, optional*
name of the material used in the simulations
kwargs : *dict*
optional properties of the material needed for the simulation

property **material**

Module contents

simulation subpackage of xrayutilities.

This package provides possibilities to simulate X-ray diffraction and reflectivity curves of thin film samples. It could be extended for more general use in future if there is demand for that.

In addition it provides a fitting routine for reflectivity data which is based on lmfit.

Submodules

xrayutilities.config module

module to parse xrayutilities user-specific config file the parsed values are provide as global constants for the use in other parts of xrayutilities. The config file with the default constants is found in the python installation path of xrayutilities. It is however not recommended to change things there, instead the user-specific config file ~/.xrayutilities.conf or the local xrayutilities.conf file should be used.

xrayutilities.config.**trytomake**(obj, key, typefunc)

xrayutilities.exception module

xrayutilities derives its own exceptions which are raised upon wrong input when calling one of xrayutilities functions. none of the pre-defined exceptions is made for that purpose.

exception xrayutilities.exception.**InputError**
Bases: **Exception**
Exception raised for errors in the input. Either wrong datatype not handled by TypeError or missing mandatory keyword argument (Note that the obligation to give keyword arguments might depend on the value of the arguments itself)

exception xrayutilities.exception.**UsageError**
Bases: **Exception**
Exception raised when a wrong use of an object is detected.

xrayutilities.experiment module

module helping with planning and analyzing experiments. various classes are provided for describing experimental geometries, calculationof angular coordinates of Bragg reflections, conversion of angular coordinates to Q-space and determination of powder diffraction peak positions.

The strength of the module is the versatile QConversion module which can be configured to describe almost any goniometer geometry.

class xrayutilities.experiment.**Experiment**(ipdir, ndir, **keyargs)
Bases: **object**
base class for describing experiments users should use the derived classes: HXRD, GID, PowderExperiment

Ang2HKL(*args, **kwargs)
angular to (h, k, l) space conversion. It will set the UB argument to Ang2Q and pass all other parameters unchanged. See Ang2Q for description of the rest of the arguments.

Parameters: **args :** *list*

arguments forwarded to Ang2Q

kwargs : *dict, optional*

optional keyword arguments

B : *array-like, optional*

reciprocal space conversion matrix of a Crystal. You can specify the matrix B (default identiy matrix) shape needs to be (3, 3)

mat : *Crystal, optional*

Crystal object to use to obtain a B matrix (e.g. xu.materials.Si) can be used as alternative to the B keyword argument B is favored in case both are given

U : *array-like, optional*

orientation matrix U can be given. If none is given the orientation defined in the Experiment class is used.

dettype : *{'point', 'linear', 'area'}, optional*

detector type: decides which routine of Ang2Q to call. default 'point'

delta : *ndarray, list or tuple, optional*

giving delta angles to correct the given ones for misalignment. delta must be an numpy array or list of length 2. used angles are than `(om, tt) - delta`

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

en : *float or str, optional*

x-ray energy in eV (default: converted self._wl)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

sampledis : *tuple or list or array-like*

sample displacement vector in relative units of the detector distance. Applies to parallel beam geometry. (default: (0, 0, 0))

Returns: **ndarray**

H K L coordinates as numpy.ndarray with shape *(N , 3)* where *N* corresponds to the number of points given in the input (args)

`Q2Ang` (`qvec`**)**

`TiltAngle` (`q`**,** `deg=True`**)**

TiltAngle(q, deg=True): Return the angle between a q-space position and the surface normal.

Parameters: **q :** *list or numpy array with the reciprocal space position*

optional keyword arguments:

deg : *True/False whether the return value should be in degree or*

radians (default: True)

`Transform` (`v`**)**

transforms a vector, matrix or tensor of rank 4 (e.g. elasticity tensor) to the coordinate frame of the Experiment class. This is for example necessary before any Q2Ang-conversion can be performed.

Parameters: **v :** *object to transform, list or numpy array of shape*

(n,) (n, n), (n, n, n, n) where n is the rank of the transformation matrix

Returns: **transformed object of the same shape as v**

__init__(ipdir, ndir, **keyargs)
initialization of an Experiment class needs the sample orientation given by the samples surface normal and an second not colinear direction specifying the inplane reference direction in the crystal coordinate system. The orientation of the surface normal in the lab coordinate system can also be given or is automatically determined by the goniometer type (see argument sampleor).

Parameters:

ipdir : *list or tuple or array-like*

inplane reference direction (ipdir points into the primary beam direction at zero angles)

ndir : *list or tuple or array-like*

surface normal of your sample. ndir points in a direction perpendicular to the primary beam, how it is orientated in real space is determined by the parameter sampleor (see below).

keyargs : *dict, optional*

optional keyword arguments

qconv : *QConversion, optional*

QConversion object to use for the Ang2Q conversion

sampleor : *{'det', 'sam', '[xyz][+-]'}, optional*

determines the sample surface orientation with respect to the coordinate system in which the goniometer rotations are given. You can use the [xyz][+-] syntax to specify the nominal surface orientation (when all goniometer angles are zero). In addition two special values 'det' and 'sam' are available, which will let the code determine the orientation from either the inner most detector or sample rotation. 'det' means the surface is in the plane spanned by the inner most detector rotation (rotation around primary beam is ignored) and perpendicular to the primary beam. 'sam' means the surface orientation is along the innermost sample circles rotation direction (in this case this should be the azimuth motor to yield the expected results). Default is 'det'. Restrictions: the given direction can not be along the primary beam. If one needs that case, let the maintainer know. Currently this case is caught and a different axis is automatically used as z-axis.

wl : *float or str*

wavelength of the x-rays in angstrom (default: 1.5406A)

en : *float or str*

energy of the x-rays in eV (default: 8048eV == 1.5406A). the en keyword overrules the wl keyword

Note

The qconv argument does not change the Q2Ang function's behavior. See Q2AngFit function in case you want to calculate for arbitrary goniometers with some restrictions.

property **energy**

property **wavelength**

class xrayutilities.experiment.**FourC**(idir, ndir, **keyargs)
Bases: **HXRD**
class describing high angle x-ray diffraction experiments the class helps with calculating the angles of Bragg reflections as well as helps with analyzing measured data
the class describes a four circle (omega, chi, phi, twotheta) goniometer to help with coplanar x-ray diffraction experiments. Nevertheless 3D data can be treated with the use of linear and area detectors. see "help(FourCInstance.Ang2Q)"

__init__(idir, ndir, **keyargs)
initialization routine for the FourC Experiment class

Parameters: **idir, ndir, keyargs**

same as for the Experiment base class -> please look at the docstring of Experiment.__init__ for more details

geometry : *{'hi_lo', 'lo_hi', 'real'}, optional*

determines the scattering geometry :

- 'hi_lo' (default) high incidence-low exit
- 'lo_hi' low incidence - high exit
- 'real' general geometry - q-coordinates determine high or low incidence

class `xrayutilities.experiment.`**`GID`**(`idir`, `ndir`, `**keyargs`)

Bases: **`Experiment`**

class describing grazing incidence x-ray diffraction experiments the class helps with calculating the angles of Bragg reflections as well as it helps with analyzing measured data

the class describes a four circle (alpha_i, azimuth, twotheta, beta) goniometer to help with GID experiments at the ROTATING ANODE. 3D data can be treated with the use of linear and area detectors. see "help(GIDInstance.Ang2Q)"

Using this class the default sample surface orientation is determined by the inner most sample rotation (which is usually the azimuth motor).

Ang2Q (`ai`**,** `phi`**,** `tt`**,** `beta`**,** `**kwargs`**)**

angular to momentum space conversion for a point detector. Also see help GID.Ang2Q for procedures which treat line and area detectors

Parameters: **ai, phi, tt, beta :** *float or array-like*

sample and detector angles as numpy array, lists or Scalars must be given. All arguments must have the same shape or length. However, if one angle is always the same its enough to give one scalar value.

kwargs : *dict, optional*

optional keyword arguments

delta : *list, tuple or array-like, optional*

giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of length 4. Used angles are then `ai, phi, tt, beta - delta`

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: identity matrix)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

Returns: **ndarray**

reciprocal space positions as numpy.ndarray with shape *(N , 3)* where *N* corresponds to the number of points given in the input

Q2Ang (`qvec`**,** `trans`=True**,** `deg`=True**,** `**kwargs`**)**

calculate the GID angles needed in the experiment the inplane reference direction defines the direction were the reference direction is parallel to the primary beam (i.e. lattice planes perpendicular to the beam)

Note

The behavior of this function is unchanged if the goniometer definition is changed!

Parameters: **qvec :** *list, tuple or array-like*

array of shape (3) with q-space vector components or 3 separate lists with qx, qy, qz

trans : *bool, optional*

apply coordinate transformation on Q (default True)

deg : *book, optional*

(default True) determines if the angles are returned in radians or degrees

Returns: **ndarray**

a numpy array of shape (4) with four GID scattering angles which are [alpha_i, azimuth, twotheta, beta];

- alpha_i : incidence angle to surface (at the moment always 0)
- **azimuth :** *sample rotation with respect to the inplane*

  reference direction
- twotheta : scattering angle
- beta : exit angle from surface (at the moment always 0)

__init__ (idir**,** ndir**,** ****keyargs**)**

initialization routine for the GID Experiment class

- `idir` defines the inplane reference direction (idir points into the PB direction at zero angles)
- `ndir` defines the surface normal of your sample (ndir points along the innermost sample rotation axis)

Parameters: **idir, ndir, keyargs**

same as for the Experiment base class

class xrayutilities.experiment.**GISAXS** (idir, ndir, ****keyargs**)

Bases: **Experiment**

class describing grazing incidence x-ray diffraction experiments the class helps with calculating the angles of Bragg reflections as well as it helps with analyzing measured data

the class describes a three circle (alpha_i, twotheta, beta) goniometer to help with GISAXS experiments at the ROTATING ANODE. 3D data can be treated with the use of linear and area detectors. see help self.Ang2Q

Ang2Q (ai**,** tt**,** beta**,** ****kwargs**)**

angular to momentum space conversion for a point detector. Also see help GISAXS.Ang2Q for procedures which treat line and area detectors

Parameters: **ai, tt, beta :** *float or array-like*

sample and detector angles as numpy array, lists or Scalars must be given. all arguments must have the same shape or length. Howevver, if one angle is always the same its enough to give one scalar value.

kwargs : *dict, optional*

optional keyword arguments

delta : *list, tuple or array-like, optional*

giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of length 3. Used angles are then `ai, tt, beta - delta`

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: identity matrix)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

Returns: **ndarray**

reciprocal space positions as numpy.ndarray with shape *(N , 3)* where *N* corresponds to the number of points given in the input

Q2Ang (Q**,** trans=True**,** deg=True**,** **kwargs**)**

__init__ (idir**,** ndir**,** **keyargs**)**

initialization routine for the GISAXS Experiment class
idir defines the inplane reference direction (idir points into the PB direction at zero angles)

Parameters: **idir, ndir, keyargs**

same as for the Experiment base class

class xrayutilities.experiment.**HXRD** (idir, ndir, geometry='hi_lo', **keyargs)

Bases: **Experiment**

class describing high angle x-ray diffraction experiments the class helps with calculating the angles of Bragg reflections as well as helps with analyzing measured data
the class describes a two circle (omega, twotheta) goniometer to help with coplanar x-ray diffraction experiments. Nevertheless 3D data can be treated with the use of linear and area detectors. see “help(HXRDInstance.Ang2Q)”

Ang2Q (om**,** tt**,** **kwargs**)**

angular to momentum space conversion for a point detector. Also see help HXRD.Ang2Q for procedures which treat line and area detectors

Parameters: **om, tt :** *float or array-like*

sample and detector angles as numpy array, lists or Scalars must be given. All arguments must have the same shape or length. However, if one angle is always the same its enough to give one scalar value.

kwargs : *dict, optional*

optional keyword arguments

delta : *list or array-like*

giving delta angles to correct the given ones for misalignment. delta must be an numpy array or list of length 2. Used angles are than om, tt - delta

UB : *array-like*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: identity matrix)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

Returns: **ndarray**

reciprocal space positions as numpy.ndarray with shape *(N , 3)* where *N* corresponds to the number of points given in the input

Q2Ang (*Q**,** **keyargs**)**

Convert a reciprocal space vector Q to COPLANAR scattering angles. The keyword argument trans determines whether Q should be transformed to the experimental coordinate frame or not. The coplanar scattering angles correspond to a goniometer with sample rotations [‘x+’, ‘y+’, ‘z-’] and detector rotation ‘x+’ and primary beam along y. This is a standard four circle diffractometer.

Note

The behavior of this function is unchanged if the goniometer definition is changed!

Parameters: **Q :** *list, tuple or array-like*

array of shape (3) with q-space vector components or 3 separate lists with qx, qy, qz

trans : *bool, optional*

apply coordinate transformation on Q (default True)

deg : *book, optional*

(default True) determines if the angles are returned in radians or degrees

geometry : *{'hi_lo', 'lo_hi', 'real', 'realTilt'}, optional*

determines the scattering geometry (default: self.geometry):

- 'hi_lo' high incidence and low exit
- 'lo_hi' low incidence and high exit
- 'real' general geometry with angles determined by q-coordinates (azimuth); this and upper geometries return [omega, 0, phi, twotheta]
- 'realTilt' general geometry with angles determined by q-coordinates (tilt); returns [omega, chi, phi, twotheta]

refrac : *bool, optional*

determines if refraction is taken into account; if True then also a material must be given (default: False)

mat : *Crystal*

Crystal object; needed to obtain its optical properties for refraction correction, otherwise not used

full_output : *bool, optional*

determines if additional output is given to determine scattering angles more accurately in case refraction is set to True. default: False

fi, fd : *tuple or list*

if refraction correction is applied one can optionally specify the facet through which the beam enters (fi) and exits (fd) fi, fd must be the surface normal vectors (not transformed & not necessarily normalized). If omitted the normal direction of the experiment is used.

Returns: **ndarray**

full_output=False: a numpy array of shape (4) with four scattering angles which are [omega, chi, phi, twotheta];

- omega : incidence angle with respect to surface
- chi : sample tilt for the case of non-coplanar geometry
- **phi :** *sample azimuth with respect to inplane reference*
  direction
- twotheta : scattering angle/detector angle

full_output=True: a numpy array of shape (6) with five angles which are [omega, chi, phi, twotheta, psi_i, psi_d]

- **psi_i :** *offset of the incidence beam from the scattering plane*
  due to refraction
- **pdi_d :** *offset ot the diffracted beam from the scattering plane*
  due to refraction

__init__(idir, ndir, geometry='hi_lo', **keyargs)

initialization routine for the HXRD Experiment class

Parameters: **idir, ndir, keyargs**

same as for the Experiment base class -> please look at the docstring of Experiment.__init__ for more details

geometry : *{'hi_lo', 'lo_hi', 'real'}, optional*

determines the scattering geometry :

- 'hi_lo' (default) high incidence-low exit
- 'lo_hi' low incidence - high exit
- 'real' general geometry - q-coordinates determine high or low incidence

class xrayutilities.experiment.**NonCOP**(idir, ndir, **keyargs)

Bases: **Experiment**

class describing high angle x-ray diffraction experiments. The class helps with calculating the angles of Bragg reflections as well as helps with analyzing measured data for NON-COPLANAR measurements, where the tilt is used to align asymmetric peaks, like in the case of a polefigure measurement.

The class describes a four circle (omega, chi, phi, twotheta) goniometer to help with x-ray diffraction experiments. Linear and area detectors can be treated as described in "help(NonCOPInstance.Ang2Q)"

Ang2Q (om**,** chi**,** phi**,** tt**,** **kwargs**)**

angular to momentum space conversion for a point detector. Also see help NonCOP.Ang2Q for procedures which treat line and area detectors

Parameters: **om, chi, phi, tt :** *float or array-like*

sample and detector angles as numpy array, lists or Scalars must be given. All arguments must have the same shape or length. However, if one angle is always the same its enough to give one scalar value.

kwargs : *dict, optional*

optional keyword arguments

delta : *list, tuple or array-like, optional*

giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of length 4. Used angles are than om, chi, phi, tt - delta

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: identity matrix)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

Returns: **ndarray**

reciprocal space positions as numpy.ndarray with shape *(N , 3)* where *N* corresponds to the number of points given in the input

Q2Ang (*Q**,** **keyargs**)**

Convert a reciprocal space vector Q to NON-COPLANAR scattering angles. The keyword argument trans determines whether Q should be transformed to the experimental coordinate frame or not.

Note

The behavior of this function is unchanged if the goniometer definition is changed!

Parameters: **Q :** *list, tuple or array-like*

array of shape (3) with q-space vector components or 3 separate lists with qx, qy, qz

trans : *bool, optional*

apply coordinate transformation on Q (default True)

deg : *book, optional*

(default True) determines if the angles are returned in radians or degrees

Returns: **ndarray**

a numpy array of shape (4) with four scattering angles which are [omega, chi, phi, twotheta];

- omega : incidence angle with respect to surface
- chi : sample tilt for the case of non-coplanar geometry
- **phi :** *sample azimuth with respect to inplane reference* direction
- twotheta : scattering angle/detector angle

__init__ (idir**,** ndir**,** **keyargs**)**

initialization routine for the NonCOP Experiment class

Parameters: **idir, ndir, keyargs**

same as for the Experiment base class

class xrayutilities.experiment.**PowderExperiment** (**kwargs)

Bases: **Experiment**

Experimental class for powder diffraction which helps to convert theta angles to momentum transfer space

Q2Ang (qpos**,** wl=None**,** deg=True**)**

Converts reciprocal space values to theta angles

__init__ (kwargs**)**

class constructor which takes the same keyword arguments as the Experiment class

Parameters: **kwargs :** *dict, optional*

keyword arguments same as for the Experiment base class

class xrayutilities.experiment.**QConvFlags** (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

Bases: **IntFlag**

HAS_SAMPLEDIS = *4*

HAS_TRANSLATIONS = *1*

NONE = *0*

VERBOSE = *16*

class xrayutilities.experiment.**QConversion** (sampleAxis, detectorAxis, r_i, **kwargs)

Bases: **object**

Class for the conversion of angular coordinates to momentum space for arbitrary goniometer geometries and X-ray energy. Both angular scans (where some goniometer angles change during data acquisition) and energy scans (where the energy is varied during acquisition) as well as mixed cases can be treated.

the class is configured with the initialization and does provide three distinct routines for conversion to momentum space for

- point detector: point(…) or __call__()
- linear detector: linear(…)

- area detector: area(…)

linear() and area() can only be used after the init_linear() or init_area() routines were called

property **UB**

__call__(*args*, ***kwargs*)
wrapper function for point(…)

__init__(sampleAxis, detectorAxis, r_i, ***kwargs**)**
initialize QConversion object. This means the rotation axis of the sample and detector circles need to be given: starting with the outer most circle.

Parameters:

sampleAxis : *list or tuple*

sample circles, e.g. [‘x+’, ‘z+’] would mean two sample circles whereas the outer one turns righthanded around the x axis and the inner one turns righthanded around z.

detectorAxis : *list or tuple*

detector circles e.g. [‘x-’] would mean a detector arm with a single motor turning lefthanded around the x-axis.

r_i : *array-like*

vector giving the direction of the primary beam (length is relevant only if translations are involved)

kwargs : *dict, optional*

optional keyword arguments

wl : *float or str, optional*

wavelength of the x-rays in angstrom

en : *float or str, optional*

energy of the x-rays in electronvolt

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: identity matrix)

area(*args*, ***kwargs**)**
angular to momentum space conversion for a area detector the center pixel defined by the init_area routine must be in direction of self.r_i when detector angles are zero
the detector geometry must be initialized by the init_area(…) routine

Parameters: **args :** *ndarray, list or Scalars*

sample and detector angles; in total *len(self.sampleAxis)* + *len(detectorAxis)* must be given, always starting with the outer most circle. all arguments must have the same shape or length but can be mixed with Scalars (i.e. if an angle is always the same it can be given only once instead of an array)

- **sAngles :**

  sample circle angles, number of arguments must correspond to len(self.sampleAxis)

- **dAngles :**

  detector circle angles, number of arguments must correspond to len(self.detectorAxis)

kwargs : *dict, optional*

optional keyword arguments

delta : *list or array-like, optional*

delta angles to correct the given ones for misalignment. delta must be an numpy array or list of `len(*args)`. used angles are then `*args - delta`

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

Nav : *tuple or list, optional*

number of channels to average to reduce data size e.g. [2, 2] (default: self._area_nav)

roi : *list or tuple, optional*

region of interest for the detector pixels; e.g. [100, 900, 200, 800] (default: self._area_roi)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

en : *float, optional*

x-ray energy in eV (default is converted self._wl). both wavelength and energy can also be an array which enables the QConversion for energy scans. Note that the *en* keyword overrules the *wl* keyword!

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

sampledis : *tuple or list or array-like*

sample displacement vector in relative units of the detector distance. Applies to parallel beam geometry. (default: (0, 0, 0))

Returns: **reciprocal space position of all detector pixels in a numpy.ndarray of**

shape ((*)*(self._area_roi[1] - self._area_roi[0]+1) *

(self._area_roi[3] - self._area_roi[2] + 1) , 3) were detectorDir1 is

the fastest varing

property **detectorAxis**

property handler for _detectorAxis

Returns: **list of detector axis following the syntax /[xyz][+-]/**

property **energy**

getDetectorDistance (*args, **kwargs)

obtains the detector distance by applying the detector arm movements. This is especially interesting for the case of 1 or 2D detectors to perform certain geometric corrections.

Parameters: **args :** *list*

detector angles. Only detector arm angles as described by the detectorAxis attribute must be given.

kwargs : *dict, optional*

optional keyword arguments

dim : *int, optional*

dimension of the detector for which the position should be determined

roi : *tuple or list, optional*

region of interest for the detector pixels; (default: self._area_roi/self._linear_roi)

Nav : *tuple or list, optional*

number of channels to average to reduce data size; (default: self._area_nav/self._linear_nav)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

Returns: **ndarray**

numpy array with the detector distance

getDetectorPos (*args, **kwargs)

obtains the detector position vector by applying the detector arm rotations.

Parameters: **args :** *list*

detector angles. Only detector arm angles as described by the detectorAxis attribute must be given.

kwargs : *dict, optional*

optional keyword arguments

dim : *int, optional*

dimension of the detector for which the position should be determined

roi : *tuple or list, optional*

region of interest for the detector pixels; (default: self._area_roi/self._linear_roi)

Nav : *tuple or list, optional*

number of channels to average to reduce data size; (default: self._area_nav/self._linear_nav)

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

Returns: **ndarray**

numpy array of length 3 with vector components of the detector direction. The length of the vector is k.

init_area (detectorDir1, detectorDir2, cch1, cch2, Nch1, Nch2, distance=None, pwidth1=None, pwidth2=None, chpdeg1=None, chpdeg2=None, detrot=0, tiltazimuth=0, tilt=0, **kwargs)

initialization routine for area detectors detector direction as well as distance and pixel size or channels per degree must be given. Two separate pixel sizes and channels per degree for the two orthogonal directions can be given

Parameters: **detectorDir1 :** *str*

direction of the detector (along the pixel direction 1); e.g. ‘z+’ means higher pixel numbers at larger z positions

detectorDir2 : *str*

direction of the detector (along the pixel direction 2); e.g. ‘x+’

cch1, cch2 : *float*

center pixel, in direction of self.r_i at zero detectorAngles

Nch1, Nch2 : *int*

number of detector pixels along direction 1, 2

distance : *float, optional*

distance of center pixel from center of rotation

pwidth1, pwidth2 : *float, optional*

width of one pixel (same unit as distance)

chpdeg1, chpdeg2 : *float, optional*

channels per degree (only absolute value is relevant) sign determined through *detectorDir1, detectorDir2*

detrot : *float, optional*

angle of the detector rotation around primary beam direction (used to correct misalignments)

tiltazimuth : *float, optional*

direction of the tilt vector in the detector plane (in degree)

tilt : *float, optional*

tilt of the detector plane around an axis normal to the direction given by the tiltazimuth

kwargs : *dict, optional*

optional keyword arguments

Nav : *tuple or list, optional*

number of channels to average to reduce data size (default: [1, 1])

roi : *tuple or list, optional*

region of interest for the detector pixels; e.g. [100, 900, 200, 800]

Note

Either distance and pwidth1, pwidth2 or chpdeg1, chpdeg2 must be given !!

Note

the channel numbers run from 0 .. NchX-1

init_linear (`detectorDir`**,** `cch`**,** `Nchannel`**,** `distance`=None**,** `pixelwidth`=None**,** `chpdeg`=None**,** `tilt`=0**,** `**kwargs`**)**

initialization routine for linear detectors detector direction as well as distance and pixel size or channels per degree must be given.

Parameters: **detectorDir :** *str*
direction of the detector (along the pixel array); e.g. ‘z+’

cch : *float*
center channel, in direction of self.r_i at zero detectorAngles

Nchannel : *int*
total number of detector channels

distance : *float, optional*
distance of center channel from center of rotation

pixelwidth : *float, optional*
width of one pixel (same unit as distance)

chpdeg : *float, optional*
channels per degree (only absolute value is relevant) sign determined through detectorDir

tilt : *float, optional*
tilt of the detector axis from the detectorDir (in degree)

kwargs: dict, optional
optional keyword arguments

Nav : *int, optional*
number of channels to average to reduce data size (default: 1)

roi : *tuple or list*
region of interest for the detector pixels; e.g. [100, 900]

Note

Either distance and pixelwidth or chpdeg must be given !!

Note

the channel numbers run from 0 .. Nchannel-1

linear (*args, **kwargs)
angular to momentum space conversion for a linear detector the cch of the detector must be in direction of self.r_i when detector angles are zero
the detector geometry must be initialized by the init_linear(…) routine

Parameters: **args :** *ndarray, list or Scalars*

sample and detector angles; in total *len(self.sampleAxis)* + *len(detectorAxis)* must be given, always starting with the outer most circle. all arguments must have the same shape or length but can be mixed with Scalars (i.e. if an angle is always the same it can be given only once instead of an array)

- **sAngles :**

  sample circle angles, number of arguments must correspond to len(self.sampleAxis)

- **dAngles :**

  detector circle angles, number of arguments must correspond to len(self.detectorAxis)

kwargs : *dict, optional*

optional keyword arguments

delta : *list or array-like, optional*

delta angles to correct the given ones for misalignment. delta must be an numpy array or list of `len(*args)`. used angles are then `*args - delta`

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

Nav : *int, optional*

number of channels to average to reduce data size (default: self._linear_nav)

roi : *list or tuple, optional*

region of interest for the detector pixels; e.g. [100, 900] (default: self._linear_roi)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

en : *float, optional*

x-ray energy in eV (default is converted self._wl). both wavelength and energy can also be an array which enables the QConversion for energy scans. Note that the *en* keyword overrules the *wl* keyword!

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

sampledis : *tuple or list or array-like*

sample displacement vector in relative units of the detector distance. Applies to parallel beam geometry. (default: (0, 0, 0))

Returns: **reciprocal space position of all detector pixels in a numpy.ndarray of**

shape ((*)*(self._linear_roi[1]-self._linear_roi[0]+1) , 3)

point (**args*, ***kwargs***)**

angular to momentum space conversion for a point detector located in direction of self.r_i when detector angles are zero

Parameters: **args :** *ndarray, list or Scalars*

sample and detector angles; in total *len(self.sampleAxis)* + *len(detectorAxis)* must be given, always starting with the outer most circle. all arguments must have the same shape or length but can be mixed with Scalars (i.e. if an angle is always the same it can be given only once instead of an array)

- **sAngles :**

  sample circle angles, number of arguments must correspond to len(self.sampleAxis)

- **dAngles :**

  detector circle angles, number of arguments must correspond to len(self.detectorAxis)

kwargs : *dict, optional*

optional keyword arguments

delta : *list or array-like, optional*

delta angles to correct the given ones for misalignment. delta must be an numpy array or list of `len(*args)`. used angles are then `*args - delta`

UB : *array-like, optional*

matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

wl : *float or str, optional*

x-ray wavelength in angstrom (default: self._wl)

en : *float, optional*

x-ray energy in eV (default is converted self._wl). both wavelength and energy can also be an array which enables the QConversion for energy scans. Note that the *en* keyword overrules the *wl* keyword!

deg : *bool, optional*

flag to tell if angles are passed as degree (default: True)

sampledis : *tuple or list or array-like*

sample displacement vector in relative units of the detector distance. Applies to parallel beam geometry. (default: (0, 0, 0))

Returns: **ndarray**

reciprocal space positions as numpy.ndarray with shape `(N , 3)` where *N* corresponds to the number of points given in the input

property **sampleAxis**

property handler for _sampleAxis

Returns: **list**

sample axes following the syntax /[xyzk][+-]/

transformSample2Lab (`vector`**,** `*args`**)**

transforms a vector from the sample coordinate frame to the laboratory coordinate system by applying the sample rotations from inner to outer circle.

Parameters: **vector :** *sequence, list or numpy array*

vector to transform

args : *list*

goniometer angles (sample angles or full goniometer angles can be given. If more angles than the sample circles are given they will be ignored)

Returns: **ndarray**

rotated vector as numpy.array

property **wavelength**

xrayutilities.gridder module

class xrayutilities.gridder.**FuzzyGridder1D**(nx)
Bases: **Gridder1D**
An 1D binning class considering every data point to have a finite width. If necessary one data point will be split fractionally over different data bins. This is numerically more effort but represents better the typical case of a experimental data, which do not represent a mathematical point but have a finite width (e.g. X-ray data from a 1D detector).

__call__(x, data, width=None)
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions
data : *ndarray*
numpy array of arbitrary shape with data values
width : *float, optional*
width of one data point. If not given half the bin size will be used.

class xrayutilities.gridder.**Gridder**
Bases: **ABC**
Basis class for gridders in xrayutilities. A gridder is a function mapping irregular spaced data onto a regular grid by binning the data into equally sized elements.
There are different ways of defining the regular grid of a Gridder. In xrayutilities the data bins extend beyond the data range in the input data, but the given position being the center of these bins, extends from the minimum to the maximum of the data! The main motivation for this was to create a Gridder, which when feeded with N equidistant data points and gridded with N bins would not change the data position (not the case with numpy.histogramm functions!). Of course this leads to the fact that for homogeneous point density the first and last bin in any direction are not filled as the other bins.
A different definition is used by numpy histogram functions where the bins extend only to the end of the data range. (see numpy histogram, histrogram2d, …)

Clear()
Clear so far gridded data to reuse this instance of the Gridder

KeepData(bool)

Normalize(bool)
set or unset the normalization flag. Normalization needs to be done to obtain proper gridding but may want to be disabled in certain cases when sequential gridding is performed

abstract **__call__**()
abstract call method which every implementation of a Gridder has to override

__init__()
Constructor defining default properties of any Gridder class

property **data**
return gridded data (performs normalization if switched on)

class xrayutilities.gridder.**Gridder1D**(nx)
Bases: **Gridder**

__call__(x, data)
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions
data : *ndarray*
numpy array of arbitrary shape with data values

__init__ (nx**)**
Constructor defining default properties of any Gridder class

dataRange (min**,** max**,** fixed=True**)**
define minimum and maximum data range, usually this is deduced from the given data automatically, however, for sequential gridding it is useful to set this before the first call of the gridder. data outside the range are simply ignored

Parameters: **min :** *float*
minimum value of the gridding range
max : *float*
maximum value of the gridding range
fixed : *bool, optional*
flag to turn fixed range gridding on (True (default)) or off (False)

savetxt (filename**,** header="**)**
save gridded data to a txt file with two columns. The first column is the data coordinate and the second the corresponding data value

Parameters: **filename :** *str*
output filename
header : *str, optional*
optional header for the data file.

property **xaxis**
Returns the xaxis of the gridder the returned values correspond to the center of the data bins used by the gridding algorithm

class xrayutilities.gridder.**GridderFlags** (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
Bases: **IntFlag**

NO_DATA_INIT = *1*

NO_NORMALIZATION = *4*

VERBOSE = *16*

xrayutilities.gridder.**axis** (min_value, max_value, n)
Compute the a grid axis.

Parameters: **min_value :** *float*
axis minimum value
max_value : *float*
axis maximum value
n : *int*
number of steps

xrayutilities.gridder.**delta** (min_value, max_value, n)
Compute the stepsize along an axis of a grid.

Parameters: **min_value :** *axis minimum value*

max_value : *axis maximum value*

n : *number of steps*

class xrayutilities.gridder.**npyGridder1D**(nx)
Bases: **Gridder1D**

__call__(x, data)
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions

data : *ndarray*
numpy array of arbitrary shape with data values

property **xaxis**
Returns the xaxis of the gridder the returned values correspond to the center of the data bins used by the numpy.histogram function

xrayutilities.gridder.**ones**(*args)
Compute ones for matrix generation. The shape is determined by the number of input arguments.

xrayutilities.gridder2d module

class xrayutilities.gridder2d.**FuzzyGridder2D**(nx, ny)
Bases: **Gridder2D**
An 2D binning class considering every data point to have a finite area. If necessary one data point will be split fractionally over different data bins. This is numerically more effort but represents better the typical case of a experimental data, which do not represent a mathematical point but have a finite size (e.g. X-ray data from a 2D detector or reciprocal space maps measured with point/linear detector).
Currently only a rectangular area can be considered during the gridding.

__call__(x, y, data, **kwargs)
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions

y : *ndarray*
numpy array of arbitrary shape with y positions

data : *ndarray*
numpy array of arbitrary shape with data values

width : *float or tuple or list, optional*
width of one data point. If not given half the bin size will be used. The width can be given as scalar if it is equal for both data dimensions, or as sequence of length 2.

class xrayutilities.gridder2d.**Gridder2D**(nx, ny)
Bases: **Gridder**

SetResolution(nx, ny)
Reset the resolution of the gridder. In this case the original data stored in the object will be deleted.

Parameters: **nx :** *int*
number of points in x-direction
ny : *int*
number of points in y-direction

__call__ (x**,** y**,** data**)**
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions
y : *ndarray*
numpy array of arbitrary shape with y positions
data : *ndarray*
numpy array of arbitrary shape with data values

__init__ (nx**,** ny**)**
Constructor defining default properties of any Gridder class

dataRange (xmin**,** xmax**,** ymin**,** ymax**,** fixed=True**)**
define minimum and maximum data range, usually this is deduced from the given data automatically, however, for sequential gridding it is useful to set this before the first call of the gridder. data outside the range are simply ignored

Parameters: **xmin, ymin :** *float*
minimum value of the gridding range in x, y
xmax, ymax : *float*
maximum value of the gridding range in x, y
fixed : *bool, optional*
flag to turn fixed range gridding on (True (default)) or off (False)

savetxt (filename**,** header=''**)**
save gridded data to a txt file with two columns. The first two columns are the data coordinates and the last one the corresponding data value.

Parameters: **filename :** *str*
output filename
header : *str, optional*
optional header for the data file.

property **xaxis**

property **xmatrix**

property **yaxis**

property **ymatrix**

class xrayutilities.gridder2d.**Gridder2DList** (nx, ny)
Bases: **Gridder2D**
special version of a 2D gridder which performs no actual averaging of the data in one grid/bin but just collects the data-objects belonging to one bin for further treatment by the user

Clear ()
Clear so far gridded data to reuse this instance of the Gridder

__call__ (x**,** y**,** data**)**
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the lists of data-objects belonging to one bin/grid-point.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions
y : *ndarray*
numpy array of arbitrary shape with y positions
data : *ndarray, list or tuple*
data of same length as x, y but of arbitrary type

property **data**
return gridded data, in this special version no normalization is defined!

xrayutilities.gridder3d module

class xrayutilities.gridder3d.**FuzzyGridder3D**(nx, ny, nz)
Bases: **Gridder3D**
An 3D binning class considering every data point to have a finite volume. If necessary one data point will be split fractionally over different data bins. This is numerically more effort but represents better the typical case of a experimental data, which do not represent a mathematical point but have a finite size.
Currently only a quader can be considered as volume during the gridding.

__call__(x, y, z, data, **kwargs)
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions
y : *ndarray*
numpy array of arbitrary shape with y positions
z : *ndarray*
numpy array fo arbitrary shape with z positions
data : *ndarray*
numpy array of arbitrary shape with data values
width : *float, tuple or list, optional*
width of one data point. If not given half the bin size will be used. The width can be given as scalar if it is equal for all three dimensions, or as sequence of length 3.

class xrayutilities.gridder3d.**Gridder3D**(nx, ny, nz)
Bases: **Gridder**

SetResolution(nx, ny, nz)

__call__(x, y, z, data)
Perform gridding on a set of data. After running the gridder the 'data' object in the class is holding the gridded data.

Parameters: **x :** *ndarray*
numpy array of arbitrary shape with x positions
y : *ndarray*
numpy array of arbitrary shape with y positions
z : *ndarray*
numpy array fo arbitrary shape with z positions
data : *ndarray*
numpy array of arbitrary shape with data values

__init__(nx, ny, nz)
Constructor defining default properties of any Gridder class

dataRange (xmin**,** xmax**,** ymin**,** ymax**,** zmin**,** zmax**,** fixed=True**)**
define minimum and maximum data range, usually this is deduced from the given data automatically, however, for sequential gridding it is useful to set this before the first call of the gridder. data outside the range are simply ignored

Parameters: **xmin, ymin, zmin :** *float*
minimum value of the gridding range in x, y, z

xmax, ymax, zmax : *float*
maximum value of the gridding range in x, y, z

fixed : *bool, optional*
flag to turn fixed range gridding on (True (default)) or off (False)

property **xaxis**

property **xmatrix**

property **yaxis**

property **ymatrix**

property **zaxis**

property **zmatrix**

xrayutilities.mpl_helper module

Defines new matplotlib Sqrt scale which further allows for negative values by using the sign of the original value as sign of the plotted value.

class xrayutilities.mpl_helper.**SqrtAllowNegScale** (axis, **kwargs)
Bases: **ScaleBase**
Scales data using a sqrt-function, however, allowing also negative values.

The scale function:
sign(y) * sqrt(abs(y))

The inverse scale function:
sign(y) * y**2

class **InvertedSqrtTransform** (shorthand_name=None)
Bases: **Transform**

has_inverse = *True*
True if this transform has a corresponding inverse transform.

input_dims = *1*
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

inverted ()
Return the corresponding inverse transformation.
It holds `x == self.inverted().transform(self.transform(x))`.
The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

is_separable = *True*
True if this transform is separable in the x- and y- dimensions.

output_dims = *1*
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

transform_non_affine (a**)**
Apply only the non-affine part of this transformation.
`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Parameters: **values :** *array*

The input values as an array of length **input_dims** or shape (N, **input_dims**).

Returns: **array**

The output values as an array of length **output_dims** or shape (N, **output_dims**), depending on the input.

class **SqrtTransform** (shorthand_name=None)

Bases: **Transform**

has_inverse = *True*

True if this transform has a corresponding inverse transform.

input_dims = *1*

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

inverted ()

return the inverse transform for this transform.

is_separable = *True*

True if this transform is separable in the x- and y- dimensions.

output_dims = *1*

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

transform_non_affine (a**)**

This transform takes an Nx1 `numpy` array and returns a transformed copy.

__init__ (axis**, ****kwargs**)**

Any keyword arguments passed to `set_xscale` and `set_yscale` will be passed along to the scale's constructor.

get_transform ()

Return the *.Transform* object associated with this scale.

limit_range_for_scale (vmin**,** vmax**,** minpos**)**

Override to limit the bounds of the axis to the domain of the transform. In the case of Mercator, the bounds should be limited to the threshold that was passed in. Unlike the autoscaling provided by the tick locators, this range limiting will always be adhered to, whether the axis range is set manually, determined automatically or changed through panning and zooming.

name = *'sqrt'*

set_default_locators_and_formatters (axis**)**

Set the locators and formatters of *axis* to instances suitable for this scale.

class xrayutilities.mpl_helper.**SqrtTickLocator** (nbins=7, symmetric=True)

Bases: **Locator**

__call__ ()

Return the locations of the ticks

__init__ (nbins=7**,** symmetric=True**)**

set_params (nbins**,** symmetric**)**

Set parameters within this locator.

tick_values (vmin**,** vmax**)**

Return the values of the located ticks given **vmin** and **vmax**.

Note

To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

view_limits (`dmin`, `dmax`)
Set the view limits to the nearest multiples of base that contain the data

xrayutilities.normalize module

module to provide functions that perform block averaging of intensity arrays to reduce the amount of data (mainly for PSD and CCD measurements

and

provide functions for normalizing intensities for

- count time
- absorber (user-defined function)
- monitor
- flatfield correction

class `xrayutilities.normalize.`**IntensityNormalizer** (`det=''`, `**keyargs`)
Bases: **object**
generic class for correction of intensity (point detector, or MCA, single CCD frames) for count time and absorber factors the class must be supplied with a absorber correction function and works with data structures provided by xrayutilities.io classes or the corresponding objects from hdf5 files

__call__ (`data`, `ccd=None`)
apply the correction method which was initialized to the measured data

Parameters:
data : *numpy.recarray*
data object from xrayutilities.io classes
ccd : *ndarray, optional*
optionally CCD data can be given as separate ndarray of shape (len(data), n1, n2), where n1, n2 is the shape of the CCD image.

Returns:
corrint : *ndarray*
corrected intensity as numpy.ndarray of the same shape as data[det] (or ccd.shape)

__init__ (`det=''`, `**keyargs`)
initialization of the corrector class

Parameters: **det :** *str*

detector field name of the data structure

mon : *str, optional*

monitor field name

time: float or str, optional

count time field name or count time as float

av_mon : *float, optional*

average monitor value (default: data[mon].mean())

smoothmon : *int*

number of monitor values used to get a smooth monitor signal

absfun : *callable, optional*

absorber correction function to be used as in `absorber_corrected_intensity = data[det]*absfun(data)`

flatfield : *ndarray*

flatfield of the detector; shape must be the same as data[det], and is only applied for MCA detectors

darkfield : *ndarray*

darkfield of the detector; shape must be the same as data[det], and is only applied for MCA detectors

Examples

```
>>> detcorr = IntensityNormalizer("MCA", time="Seconds",
... absfun=lambda d: d["PSDCORR"]/d["PSD"].astype(float))
```

property **absfun**
absfun property handler
returns the costum correction function or None

property **avmon**
av_mon property handler
returns the value of the average monitor or None if average is calculated from the monitor field

property **darkfield**
flatfield property handler
returns the current set darkfield of the detector or None if not set

property **det**
det property handler
returns the detector field name

property **flatfield**
flatfield property handler
returns the current set flatfield of the detector or None if not set

property **mon**
mon property handler
returns the monitor field name or None if not set

property **time**
time property handler
returns the count time or the field name of the count time or None if time is not set

`xrayutilities.normalize.`**`blockAverage1D`**(`data`, `Nav`)
perform block average for 1D array/list of Scalar values all data are used. at the end of the array a smaller cell may be used by the averaging algorithm

Parameters: **data :** *array-like*

data which should be contracted (length N)

Nav : *int*

number of values which should be averaged

Returns: **ndarray**

block averaged numpy array of data type numpy.double (length ceil(N/Nav))

`xrayutilities.normalize.`**`blockAverage2D`**(`data2d`, `Nav1`, `Nav2`, `**kwargs`)

perform a block average for 2D array of Scalar values all data are used therefore the margin cells may differ in size

Parameters: **data2d :** *ndarray*

array of 2D data shape (N, M)

Nav1, Nav2 : *int*

a field of (Nav1 x Nav2) values is contracted

kwargs : *dict, optional*

optional keyword argument

roi : *tuple or list, optional*

region of interest for the 2D array. e.g. [20, 980, 40, 960], reduces M, and M!

Returns: **ndarray**

block averaged numpy array with type numpy.double with shape (ceil(N/Nav1), ceil(M/Nav2))

`xrayutilities.normalize.`**`blockAverageCCD`**(`data3d`, `Nav1`, `Nav2`, `**kwargs`)

perform a block average for 2D frames inside a 3D array. all data are used therefore the margin cells may differ in size

Parameters: **data3d :** *ndarray*

array of 3D data shape (Nframes, N, M)

Nav1, Nav2 : *int*

a field of (Nav1 x Nav2) values is contracted

kwargs : *dict, optional*

optional keyword argument

roi : *tuple or list, optional*

region of interest for the 2D array. e.g. [20, 980, 40, 960], reduces M, and M!

Returns: **ndarray**

block averaged numpy array with type numpy.double with shape (Nframes, ceil(N/Nav1), ceil(M/Nav2))

`xrayutilities.normalize.`**`blockAveragePSD`**(`psddata`, `Nav`, `**kwargs`)

perform a block average for serveral PSD spectra all data are used therefore the last cell used for averaging may differ in size

Parameters: **psddata :** *ndarray*

array of 2D data shape (Nspectra, Nchannels)

Nav : *int*

number of channels which should be averaged

kwargs : *dict, optional*

optional keyword argument

roi : *tuple or list*

region of interest for the 2D array. e.g. [20, 980] Nchannels = 980-20

Returns: **ndarray**

block averaged psd spectra as numpy array with type numpy.double of shape (Nspectra , ceil(Nchannels/Nav))

xrayutilities.q2ang_fit module

Module provides functions to convert a q-vector from reciprocal space to angular space. a simple implementation uses scipy optimize routines to perform a fit for a arbitrary goniometer.

The user is, however, expected to use the bounds variable to put restrictions to the number of free angles to obtain reproducible results. In general only 3 angles are needed to fit an arbitrary q-vector (2 sample + 1 detector angles or 1 sample + 2 detector). More complicated restrictions can be implemented using the lmfit package. (done upon request!)

The function is based on a fitting routine. For a specific goniometer also analytic expressions from literature can be used as they are implemented in the predefined experimental classes HXRD, NonCOP, and GID.

`xrayutilities.q2ang_fit.`**`Q2AngFit`** (`qvec`, `expclass`, `bounds`=None, `ormat`=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), `startvalues`=None, `constraints`=None)

Functions to convert a q-vector from reciprocal space to angular space. This implementation uses scipy optimize routines to perform a fit for a goniometer with arbitrary number of goniometer angles.
The user *must* use the bounds variable to put restrictions to the number of free angles to obtain reproducible results. In general only 3 angles are needed to fit an arbitrary q-vector (2 sample + 1 detector angles or 1 sample + 2 detector).

Parameters:

qvec : *tuple or list or array-like*

q-vector for which the angular positions should be calculated

expclass : *Experiment*

experimental class used to define the goniometer for which the angles should be calculated.

bounds : *tuple or list*

bounds of the goniometer angles. The number of bounds must correspond to the number of goniometer angles in the expclass. Angles can also be fixed by supplying only one value for a particular angle. e.g.: ((low, up), fix, (low2, up2), (low3, up3))

ormat : *array-like*

orientation matrix of the sample to be used in the conversion

startvalues : *array-like*

start values for the fit, which can significantly speed up the conversion. The number of values must correspond to the number of angles in the goniometer of the expclass

constraints : *list, optional*

sequence of constraint dictionaries. This allows applying arbitrary (e.g. pseudo-angle) contraints by supplying according constraint functions. An entry of the constraints argument must be a dictionary with at least the ‘type’ and ‘fun’ set. ‘type’ can be either ‘eq’ or ‘ineq’ for equality or inequality constraints. ‘fun’ must be a callable function which for ‘eq’-constraints returns 0 when the equality condition is fulfilled (see constraints documentation in scipy.optimize.minimize for details). The supplied function will be called with the arguments gonimeter angle list as argument. Typically this means you will have to use a lambda function.

Returns:

fittedangles : *list*

list of fitted goniometer angles

qerror : *float*

error in reciprocal space

errcode : *int*

error-code of the scipy minimize function. for a successful fit the error code should be <=2

`xrayutilities.q2ang_fit.`**`exitAngleConst`** (`angles`, `alphaf`, `xrd`)

helper function for an pseudo-angle constraint of the exit angle. Can be used together with the Q2AngFit-routine in the ‘constraints’ argument. An example use case scenario to fix the exit angle to 1 degree would be: constraints={‘type’: ‘eq’, ‘fun’: lambda a: exitAngleConst(a, 1, xrd)}

Parameters: **angles :** *iterable*
fit parameters of Q2AngFit
alphaf : *float*
the exit angle which should be fixed
xrd : *Experiment*
the Experiment object to use for qconversion

`xrayutilities.q2ang_fit.`**`incidenceAngleConst`**(`angles`, `alphai`, `xrd`)
helper function for an pseudo-angle constraint of the incidence angle. Can be used together with the Q2AngFit-routine in the 'constraints' argument. An example use case scenario to fix the incidence angle to 1 degree would be: constraints={'type': 'eq', 'fun': lambda a: incidenceAngleConst(a, 1, xrd)}

Parameters: **angles :** *iterable*
fit parameters of Q2AngFit
alphai : *float*
the incidence angle which should be fixed
xrd : *Experiment*
the Experiment object to use for qconversion

xrayutilities.utilities module

xrayutilities utilities contains a conglomeration of useful functions which do not fit into one of the other files

`xrayutilities.utilities.`**`frac2str`**(`fnumber`, `denominator_limit=25`, `fmt='%7.4f'`)
convert a float to a string attempting to represent it as a fraction

Parameters: **fnumber :** *float*
floating point number to be represented as string
denominator_limit : *int*
maximal integer used as denominator. If f can't be expressed (within xu.config.EPSILON) by a fraction with a denominator up to this number a floating point string will be returned
fmt : *str*
format string used in case a floating point representation is needed
Returns: **str**

`xrayutilities.utilities.`**`import_matplotlib_pyplot`**(`funcname='XU'`)
lazy import function of matplotlib.pyplot

Parameters: **funcname :** *str*
identification string of the calling function
Returns: **flag :** *bool*
the flag is True if the loading was successful and False otherwise.
pyplot
On success pyplot is the matplotlib.pyplot package.

`xrayutilities.utilities.`**`import_mayavi_mlab`**(`funcname='XU'`)
lazy import function of mayavi.mlab

Parameters: **funcname :** *str*
identification string of the calling function
Returns: **flag :** *bool*
the flag is True if the loading was successful and False otherwise.
mlab
On success mlab is the mayavi.mlab package.

xrayutilities.utilities.**maplog**(inte, dynlow='config', dynhigh='config')
clips values smaller and larger as the given bounds and returns the log10 of the input array. The bounds are given as exponent with base 10 with respect to the maximum in the input array. The function is implemented in analogy to J. Stangl's matlab implementation.

Parameters: **inte :** *ndarray*
numpy.array, values to be cut in range
dynlow : *float, optional*
10^(-dynlow) will be the minimum cut off
dynhigh : *float, optional*
10^(-dynhigh) will be the maximum cut off

Returns: **ndarray**
numpy.array of the same shape as inte, where values smaller/larger than 10^(-dynlow, dynhigh) were replaced by 10^(-dynlow, dynhigh)

Examples

```
>>> maplog(((0.1 , 1e5, 1e6), (10, 100, 1000)), 5, 2)
array([[1., 4., 4.],
       [1., 2., 3.]])
```

xrayutilities.utilities_noconf module

xrayutilities utilities contains a conglomeration of useful functions this part of utilities does not need the config class

class xrayutilities.utilities_noconf.**ABC**
Bases: **object**
Helper class that provides a standard way to create an ABC using inheritance.

xrayutilities.utilities_noconf.**check_kwargs**(kwargs, valid_kwargs, identifier)
Raises an TypeError if kwargs included a key which is not in valid_kwargs.

Parameters: **kwargs :** *dict*
keyword arguments dictionary
valid_kwargs : *dict*
dictionary with valid keyword arguments and their description
identifier : *str*
string to identifier the caller of this function

xrayutilities.utilities_noconf.**en2lam**(inp)
converts the input energy in eV to a wavelength in angstrom

Parameters: **inp :** *float or str*
energy in eV

Returns: **float**
wavlength in angstrom

Examples

```
>>> wavelength = en2lam(8048)
```

xrayutilities.utilities_noconf.**energy**(en)
convert common energy names to energies in eV
so far this works with CuKa1, CuKa2, CuKa12, CuKb, MoKa1

Parameters: **en :** *float, array-like or str*
energy either as scalar or array with value in eV, which will be returned unchanged; or string with name of emission line

Returns: **float or array-like**

energy in eV

`xrayutilities.utilities_noconf.`**`exchange_filepath`** (`orig`, `new`, `keep=0`, `replace=None`)

function to exchange the root of a filename with the option of keeping the inner directory structure. This for example includes such a conversion /dir_a/subdir/sample/file.txt -> /home/user/data/sample/file.txt where the innermost directory name is kept (keep=1), or equally the three outer most are replaced (replace=3). One can either give keep, or replace, with replace taking preference if both are given. Note that replace=1 on Linux/Unix replaces only the root for absolute paths.

Parameters: **orig :** *str*

original filename which should have its data root replaced

new : *str*

new path which should be used instead

keep : *int, optional*

number of inner most directory names which should be kept the same in the output (default = 0)

replace : *int, optional*

number of outer most directory names which should be replaced in the output (default = None)

Returns: **str**

filename string

Examples

```
>>> newfile = exchange_filepath('/dir_a/subdir/sam/file.txt', '/data', 1)    # you get '/da
```

`xrayutilities.utilities_noconf.`**`exchange_path`** (`orig`, `new`, `keep=0`, `replace=None`)

function to exchange the root of a path with the option of keeping the inner directory structure. This for example includes such a conversion /dir_a/subdir/images/sample -> /home/user/data/images/sample where the two innermost directory names are kept (keep=2), or equally the three outer most are replaced (replace=3). One can either give keep, or replace, with replace taking preference if both are given. Note that replace=1 on Linux/Unix replaces only the root for absolute paths.

Parameters: **orig :** *str*

original path which should be replaced by the new path

new : *str*

new path which should be used instead

keep : *int, optional*

number of inner most directory names which should be kept the same in the output (default = 0)

replace : *int, optional*

number of outer most directory names which should be replaced in the output (default = None)

Returns: **str**

directory path string

Examples

```
>>> p = exchange_path('/dir_a/subdir/img/sam', '/home/user/data', keep=2)    # you get '/ho
```

`xrayutilities.utilities_noconf.`**`is_valid_variable_name`** (`name`)

`xrayutilities.utilities_noconf.`**`lam2en`** (`inp`)

converts the input wavelength in angstrom to an energy in eV

Parameters: **inp :** *float or str*

wavelength in angstrom

Returns: **float**

energy in eV

Examples

```
>>> energy = lam2en(1.5406)
```

xrayutilities.utilities_noconf.**makeNaturalName**(name, check=False)

xrayutilities.utilities_noconf.**wavelength**(wl)

convert common energy names to energies in eV
so far this works with CuKa1, CuKa2, CuKa12, CuKb, MoKa1

Parameters: **wl :** *float, array-like or str*

wavelength; If scalar or array the wavelength in angstrom will be returned unchanged, string with emission name is converted to wavelength

Returns: **float or array-like**

wavelength in angstrom

Module contents

xrayutilities is a Python package for assisting with x-ray diffraction experiments. Its the Python package included in *xrayutilities*.

It helps with planning experiments as well as analyzing the data.

Authors:

Dominik Kriegner <dominik.kriegner@gmail.com> and Eugen Wintersberger <eugen.wintersberger@desy.de>

Indices and tables

- **genindex**
- **modindex**
- **search**

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

X

Y

Z

Python Module Index

x