xrayutilities

version 1.4

Dominik Kriegner

Eugen Wintersberger

November 28, 2017

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

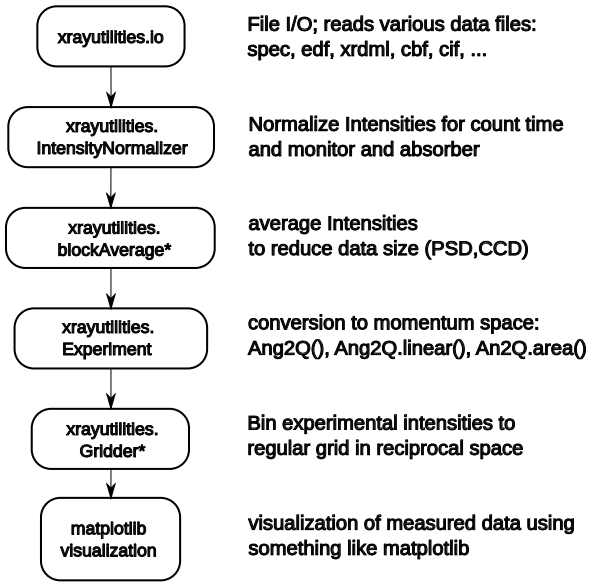
To get in touch with us or report an issue please use the mailing list or the Github issue tracker. When you want to follow announcements of major changes or new releases its recommended to sign up for the mailing list

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.

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.

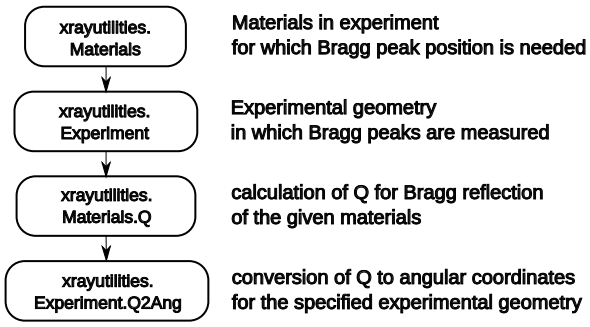
Since the amount of data taken with modern detectors often is too large to be able to work with them properly, a functions for reducing the data from linear and area detectors are provided. They use block-averaging to reduce the amount of data. Use those carefully not to loose the features you are interested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

```
1 """
2 Example script to show how to use xrayutilities to read and plot
3 reciprocal space map scans from a spec file created at the ESRF/ID10B
4
5 for details about the measurement see:
6     D Kriegner et al. Nanotechnology 22 425704 (2011)
7     http://dx.doi.org/10.1088/0957-4484/22/42/425704
8 """
9
```

```
import os

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

# global setting for the experiment
sample = "test"  # sample name used also as file name for the data file
energy = 8042.5  # x-ray energy in eV
center_ch = 715.9  # center channel of the linear detector
chpdeg = 345.28  # channels per degree of the linear detector
roi = [100, 1340]  # region of interest of the detector
nchannel = 1500  # number of channels of the detector

# intensity normalizer function responsible for count time and absorber
# correction
absfun = lambda d: d["detcorr"] / d["psd2"].astype(numpy.float)
normalizer_detcorr = xu.IntensityNormalizer(
    "MCA",
    mon="Monitor",
    time="Seconds",
    absfun=absfun)

# substrate material used for Bragg peak calculation to correct for
# experimental offsets
InP = xu.materials.InP

# initialize experimental class to specify the reference directions of your
# crystal
# 11-2: inplane reference
# 111: surface normal
hxrd = xu.HXRD(InP.Q(1, 1, -2), InP.Q(1, 1, 1), en=energy)

# configure linear detector
# detector direction + parameters need to be given
# mounted along z direction, which corresponds to twotheta
hxrd.Ang2Q.init_linear('z-', center_ch, nchannel, chpdeg=chpdeg, roi=roi)

# read spec file and save to HDF5-file
# since reading is much faster from HDF5 once the data are transformed
h5file = os.path.join("data", sample + ".h5")
try:
    s  # try if spec file object already exist ("run -i" in ipython)
except NameError:
    s = xu.io.SPECFile(sample + ".spec", path="data")
else:
    s.Update()
s.Save2HDF5(h5file)

##################################
# InP (333) reciprocal space map
omalign = 43.0529  # experimental aligned values
ttalign = 86.0733
[omnominal, dummy, dummy, ttnominal] = hxrd.Q2Ang(
    InP.Q(3, 3, 3))  # nominal values of the substrate peak

# read the data from the HDF5 file
# scan number:36, names of motors in spec file: omega= sample rocking, gamma =
# twotheta
[om, tt], MAP = xu.io.geth5_scan(h5file, 36, 'omega', 'gamma')
```

```
# normalize the intensity values (absorber and count time corrections)
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'$Q_{[11\bar2]}$ ($\AA^{-1}$)')
plt.ylabel(r'$Q_{[\bar1\bar1\bar1]}$ ($\AA^{-1}$)')
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.

xrayutilities Python package

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>

for more details see the full API documentation of **`xrayutilities`** found here: Examples and API-documentation.

Source Installation

Express instructions

- install the dependencies (Windows: pythonxy; 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:

```
> python setup.py install
```

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

Note

The python package of *xrayutilities* was formerly called "xrutils"

Detailed instructions

Installing *xrayutilities* is done using Python's distutils

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 dependencies are:

- **C-compiler** Gnu Compiler Collection or any compatible C compiler. On windows you most probably want to use the Microsoft compilers.
- **HDF5** a versatile binary data format (library is implemented in C). Although the library is not called directly, it is needed by the h5py Python module (see below).
- **Python** the scripting language in which most of *xrayutilities* code is written in. (version 2.7 or >= 3.2)
- **git** a version control system used to keep track on the *xrayutilities* development. (only needed for development)

Additionally, the following Python modules are needed in order to make *xrayutilities* work as intended:

- **Numpy** a Python module providing numerical array objects (version >= 1.8)
- **Scipy** a Python module providing standard numerical routines, which is heavily using numpy arrays (version >= 0.11.0)
- **h5py** a powerful Python interface to HDF5.
- **Matplotlib** a Python module for high quality 1D and 2D plotting (optionally)

- **lmfit** a Python module for least-squares minimization with bounds and constraints (optionally needed for fitting XRR data)
- **IPython** although not a dependency of *xrayutilities* the IPython shell is perfectly suited for the interactive use of the *xrayutilities* python package.

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

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

Building and installing the library and python package

xrayutilities uses the distutils packaging system to build and install all of its components. You can perform the installation by executing

```
>python setup.py install
```

or

```
>python setup.py 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 /usr/lib/ on Unix systems or in the Python installation directory on Windows.

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(x,y) this is 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 the Python(x,y) or WinPython Python distributions, which include already all of the needed dependencies for installing *xrayutilities*.

The setup of the environment variables is also done by the Python distributions. 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, otherwise one can follow the general installation instructions. Depending on your compiler on Microsoft Windows it might be necessary to perform the building of the Python extension separately and specify the compiler manually. This is done by

```
python setup.py build -c <compiler_name>
```

Using Python(x,y) you want to specify 'mingw32' as compiler name. With the WinPython it is recommended to use the MS Visual Studio Express 2008 (which is freely available for download) and can also build the code for 64bit Windows. In this case us 'msvc' as compiler name.

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 `examples` directory in the download found here.

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.:

```
 1 import xrayutilities as xu
 2 import os
 3
 4 # open spec file or use open SPECfile instance
 5 try: s
 6 except NameError:
 7     s = xu.io.SPECFile("sample_name.spec", path="./specdir")
 8
 9 # method (1)
10 s.scan10.ReadData()
11 scan10data = s.scan10.data
12
13 # method (2)
14 h5file = os.path.join("h5dir", "h5file.h5")
15 s.Save2HDF5(h5file) # save content of SPEC file to HDF5 file
16 # read data from HDF5 file
17 [angle1, angle2], scan10data = xu.io.geth5_scan(h5file, [10],
18                                                 "motorname1",
19                                                 "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)

```
1 s.Update() # reparse for new scans in open SPECFile instance
2
```

```
# 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

Other formats

Other formats which can be read include

- files recorded from Panalytical diffractometers in the `.xrdml` format.
- files produces by the experimental control software at Hasylab/Desy (spectra).
- 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`)
- basic support is also provided for reading of `cif` files from structure database to extract unit cell parameters. Currently all materials read from CIF files are, however, represented by the P1 space-group.

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

Angle calculation using `experiment` *and* `material` *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.

```
 1 import xrayutilities as xu
 2 Si = xu.materials.Si  # load material from materials submodule
 3
 4 # initialize experimental class with directions from experiment
 5 hxrd = xu.HXRD(Si.Q(1, 1, -2), Si.Q(1, 1, 1))
 6 # calculate angles of Bragg reflections and print them to the screen
 7 om, chi, phi, tt = hxrd.Q2Ang(Si.Q(1, 1, 1))
 8 print("Si (111)")
 9 print("om,tt: %8.3f %8.3f" % (om, tt))
10 om, chi, phi, tt = hxrd.Q2Ang(Si.Q(2, 2, 4))
11 print("Si (224)")
12 print("om,tt: %8.3f %8.3f" % (om, tt))
```

Note that on line 5 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-alpha1.

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:

```
gid = xu.GID(Si.Q(1, -1, 0), Si.Q(0, 0, 1))
# calculate angles and print them to the screen
(alphai, azimuth, tt, beta) = gid.Q2Ang(Si.Q(2, -2, 0))
print("azimuth,tt: %8.3f %8.3f" % (azimuth, tt))
```

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.

```
1 import xrayutilities as xu
2 import numpy
3
4 energy = 'CuKa12'
5
6 # creating powder experiment
7 xup = xu.PowderExperiment(en=energy)
8 theta = arange(0, 70, 0.01)
9 q = xup.Ang2Q(theta)
```

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

```
>>> print(xu.simpack.PowderDiffraction(xu.materials.In))
   Powder diffraction object
   -------------------------
   Powder-In (volume: 1, )
   Lattice:
   a1 = (3.252300 0.000000 0.000000), 3.252300
   a2 = (0.000000 3.252300 0.000000), 3.252300
   a3 = (0.000000 0.000000 4.946100), 4.946100
   alpha = 90.000000, beta = 90.000000, gamma = 90.000000
   Lattice base:
   Base point 0: In (49) (0.000000 0.000000 0.000000) occ=1.00 b=0.00
   Base point 1: In (49) (0.500000 0.500000 0.500000) occ=1.00 b=0.00
   Reflections:
   --------------
         h k l     |    tth    |    |Q|    |    Int     |   Int (%)
      ------------------------------------------------------------
        [0, 1, -1]    32.9338      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
     ....
```

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:

::

    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.

_.. note: previously you could use the Gridder's gdata object, which was always an internal buffer and should not be used anymore!

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.

Using the `material` *class*

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 should the definition of zincblende InP as well as for its hexagonal wurtzite polytype as two examples:

```
 1 import xrayutilities as xu
 2
 3 # elements (which contain their x-ray optical properties) are loaded from
 4 # xrayutilities.materials.elements
 5 In = xu.materials.elements.In
 6 P = xu.materials.elements.P
 7
 8 # define elastic parameters of the material we use a helper function which
 9 # creates the 6x6 tensor needed from the only 3 free parameters of a cubic
10 # material.
```

```
elastictensor = xu.materials.CubicElasticTensor(10.11e+10, 5.61e+10,
                                                4.56e+10)
# definition of zincblende InP:
InP = xu.materials.Crystal(
    "InP", xu.materials.SGLattice(216, 5.8687, atoms=[In, P],
                                  pos=['4a', '4c']),
    elastictensor)

# a hexagonal equivalent which shows how parameters change for material
# definition with a different space group. Since the elasticity tensor is
# optional its not specified here.
InPWZ = xu.materials.Crystal(
    "InP(WZ)", xu.materials.SGLattice(186, 4.1423, 6.8013,
                                      atoms=[In, P], pos=[('2b', 0),
                                                          ('2b', 3/8.)]))
```

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.

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

```
import xrayutilities as xu
import numpy

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

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

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
import numpy

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

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

```
print("f1: %8.4g" % Fe.f1(en))
print("f2: %8.4g" % Fe.f2(en))
```

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).

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

```
import xrayutilities as xu
import numpy as np

def Pnma(a, b, c):
    #create orthorhombic unit cell
    l = xu.materials.Lattice([a, 0, 0], [0, b, 0], [0, 0, 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.HXRD(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('SmFeO3: \thkl ', hkl, '\tqvec ', np.round(q_material, 5))
print('Lattice plane distance: %.4f' % SmFeO3.planeDistance(hkl))

#### 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 incidence angle fixed to 5 degree
# om, phi, tt, delta
bounds = (5, (-180, 180), (-1, 90), (-1, 90))
ang,qerror,errcode = xu.Q2AngFit(q_laboratory, hxrd, bounds)
print('err %d (%.3g) angles %s' % (errcode, qerror, str(np.round(ang, 5))))
# check that qerror is small!!
print('sanity check with back-transformation (hkl): ',
      np.round(hxrd.Ang2HKL(*ang,mat=SmFeO3),5))
```

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"
```

```
21
22 qconv = xu.QConversion(['z+', 'y-'], ['z+', 'y-'], [1, 0, 0])
23 hxrd = xu.HXRD(Si.Q(1, 1, -2), Si.Q(1, 1, 1), wl=wl, qconv=qconv)
24
25 # manually selected images
26
27 s = xu.io.SPECFile(specfile, path=datadir)
28 for num in [61, 62, 63, 20, 21, 26, 27, 28]:
29     s[num].ReadData()
30     try:
31         imagenrs = numpy.append(imagenrs, s[num].data['ccd_n'])
32     except:
33         imagenrs = s[num].data['ccd_n']
34
35 # avoid images which do not have to full beam on the detector as well as
36 # other which show signal due to cosmic radiation
37 avoid_images = [37, 57, 62, 63, 65, 87, 99, 106, 110, 111, 126, 130, 175,
38                 181, 183, 185, 204, 206, 207, 208, 211, 212, 233, 237, 261,
39                 275, 290]
40
41 images = []
42 ang1 = []  # outer detector angle
43 ang2 = []  # inner detector angle
44 sang = []  # sample rocking angle
45 hkls = []  # Miller indices of the reference reflections
46
47
48 def hotpixelkill(ccd):
49     """
50     function to remove hot pixels from CCD frames
51     ADD REMOVE VALUES IF NEEDED!
52     """
53     ccd[304, 97] = 0
54     ccd[303, 96] = 0
55     return ccd
56
57 # read images and angular positions from the data file
58 # this might differ for data taken at different beamlines since
59 # they way how motor positions are stored is not always consistent
60 for imgnr in numpy.sort(list(set(imagenrs) - set(avoid_images))[::4]):
61     filename = os.path.join(imgdir, filetmp % imgnr)
62     edf = xu.io.EDFFile(filename)
63     ccd = hotpixelkill(edf.data)
64     images.append(ccd)
65     ang1.append(float(edf.header['motor_pos'].split()[4]))
66     ang2.append(float(edf.header['motor_pos'].split()[3]))
67     sang.append(float(edf.header['motor_pos'].split()[1]))
68     if imgnr > 1293.:
69         hkls.append((0, 0, 0))
70     elif imgnr < 139:
71         hkls.append((0, 0, numpy.sqrt(27)))  # (3,3,3))
72     else:
73         hkls.append((0, 0, numpy.sqrt(75)))  # (5,5,5))
74
75 # call the fit for the detector parameters.
76 # Detector arm rotations and primary beam direction need to be given
77 # in total 8 detector parameters + 2 additional parameters for the reference
78 # crystal orientation and the wavelength are fitted, however the 4 misalignment
79 # parameters of the detector and the 3 other parameters can be fixed.
80 # 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 found here.

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 by:

```
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(mat_cf, 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.:

```
1 sub = xu.simpack.Layer(xu.materials.Si, inf)
2 lay1 = xu.simpack.Layer(xu.materials.Ge, 200)
3 lay2 = xu.simpack.Layer(xu.materials.SiO2, 30)
4 ls = xu.simpack.LayerStack('Si/Ge', sub, lay1, lay2)
5 # or equivalently
6 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)
ls = 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.:

```
1 sub = xu.simpack.Layer(xu.materials.Si, inf)
2 buf1 = xu.simpack.Layer(xu.materials.SiGe(0.5), 5000, relaxation=1.0)
3 buf2 = xu.simpack.Layer(xu.materials.SiGe(0.8), 5000, relaxation=1.0)
4 lay1 = xu.simpack.Layer(xu.materials.SiGe(0.6), 50, relaxation=0.0)
5 lay2 = xu.simpack.Layer(xu.materials.SiGe(1.0), 50, relaxation=0.0)
6 # create pseudomorphic superlattice stack
7 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 1µm.:

```
1 buf = xu.simpack.GradedLayerStack(xu.materials.SiGe,
2                                   0.2,  # xfrom Si0.8Ge0.2
3                                   0.7,  # xto Si0.3Ge0.7
4                                   100,  # number of sublayers
```

```
5                                   10000,  # total thickness
6                                   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

Currently only the Parrat formalism including non-correlated roughnesses is included for specular x-ray reflectivity calculations. A minimal working example for a reflectivity calculation follows.:

```
 1 # building a stack of layers
 2 sub = xu.simpack.Layer(xu.materials.GaAs, inf, roughness=2.0)
 3 lay1 = xu.simpack.Layer(xu.materials.AlGaAs(0.25), 75, roughness=2.5)
 4 lay2 = xu.simpack.Layer(xu.materials.AlGaAs(0.75), 25, roughness=3.0)
 5 pls = xu.simpack.PseudomorphicStack001('pseudo', sub+5*(lay1+lay2))
 6
 7 # reflectivity calculation
 8 m = xu.simpack.SpecularReflectivityModel(pls, sample_width=5, beam_width=0.3)
 9 ai = linspace(0, 5, 10000)
10 Ixrr = m.simulate(ai)
```

In addition to the layer thickness also the roughness and relative density 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 **`fit_xrr()`** function 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.

```
 1 from matplotlib.pylab import *
 2 import xrayutilities as xu
 3 import lmfit
 4 import numpy
 5
 6 # load experimental data
 7 ai, edata, eps = numpy.loadtxt('data/xrr_data.txt'), unpack=True)
 8 ai /= 2.0
 9
10 # define layers
11 # SiO2 / Ru(5) / CoFe(3) / IrMn(3) / AlOx(10)
```

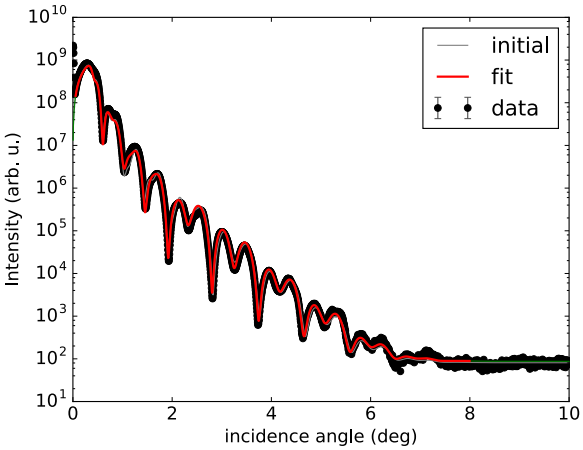
```
lSiO2 = xu.simpack.Layer(xu.materials.SiO2, inf)
lRu = xu.simpack.Layer(xu.materials.Ru, 50)
rho_cf = 0.5*8900 + 0.5*7874
mat_cf = xu.materials.Amorphous('CoFe', rho_cf)
lCoFe = xu.simpack.Layer(mat_cf, 30)
lIrMn = xu.simpack.Layer(xu.materials.Ir20Mn80, 30)
lAl2O3 = xu.simpack.Layer(xu.materials.Al2O3, 100)

m = xu.simpack.SpecularReflectivityModel(lSiO2, lRu, lCoFe, lIrMn, lAl2O3,
                                         energy='CuKa1')

p = lmfit.Parameters()
#          (Name                ,     Value,  Vary,    Min,   Max,  Expr)
p.add_many(('SiO2_thickness'    , numpy.inf, False,  None, None,  None),
           ('SiO2_roughness'    ,       2.5,  True,     0,    8,  None),
           ('Ru_thickness'      ,      47.0,  True,    25,   70,  None),
           ('Ru_roughness'      ,       2.8,  True,     0,    8,  None),
           ('Ru_density'        ,       1.0,  True,   0.8,  1.0,  None),
           ('CoFe_thickness'    ,      27.0,  True,    15,   50,  None),
           ('CoFe_roughness'    ,       4.6,  True,     0,    8,  None),
           ('CoFe_density'      ,       1.0,  True,   0.8,  1.2,  None),
           ('Ir20Mn80_thickness',      21.0,  True,    15,   40,  None),
           ('Ir20Mn80_roughness',       3.0,  True,     0,    8,  None),
           ('Ir20Mn80_density'  ,       1.1,  True,   0.8,  1.2,  None),
           ('Al2O3_thickness'   ,     100.0,  True,    70,  130,  None),
           ('Al2O3_roughness'   ,       5.5,  True,     0,    8,  None),
           ('Al2O3_density'     ,       1.0,  True,   0.8,  1.2,  None),
           ('I0'                ,    6.75e9,  True,   3e9,  8e9,  None),
           ('background'        ,        81,  True,    40,  100,  None),
           ('sample_width'      ,       6.0, False,     2,    8,  None),
           ('beam_width'        ,      0.25, False,   0.2,  0.4,  None),
           ('resolution_width'  ,      0.02, False,  0.01, 0.05,  None))

res = xu.simpack.fit_xrr(m, p, ai, data=edata, eps=eps, xmin=0.05, xmax=8.0,
                         plot=True, verbose=True)
lmfit.report_fit(res, min_correl=0.5)
```

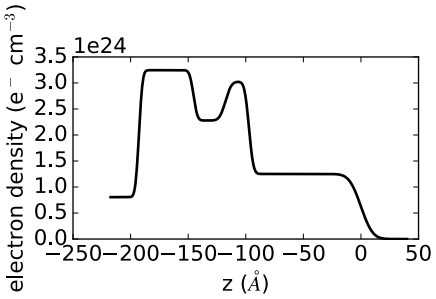
This script can interactively show the fitting progress and after the fitting shows the final plot including the x-ray reflectivity trace of the initial and final parameters.



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
```



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:

```
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.:

```
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.

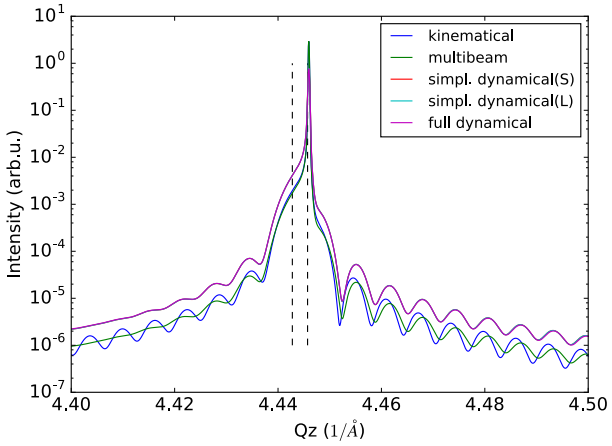
The **`DynamicalModel`** supports the calculation of diffracted signal for 'S' and 'P' polarization geometry. To simulate diffraction data of laboratory sources with Ge(220) monochromator crystal one should use:

```
1 qGe220 = linalg.norm(xu.materials.Ge.Q(2, 2, 0))
2 thMono = arcsin(qGe220 * lam / (4*pi))
3 md = xu.simpack.DynamicalModel(pls, energy='CuKa1',
4                                 Cmono=cos(2 * thMono),
5                                 polarization='both')
6 Idyn = md.simulate(ai, hkl=(0, 0, 4))
```

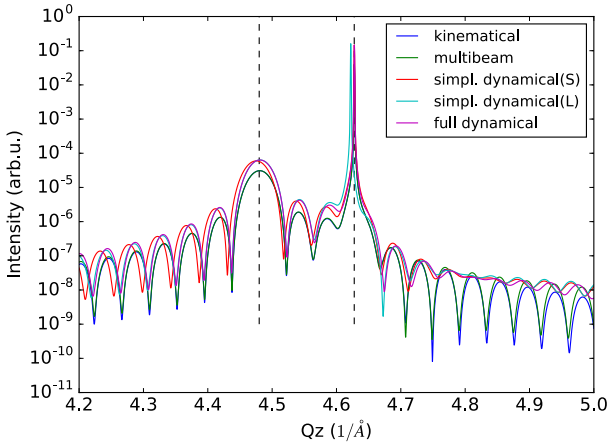
Comparison of diffraction models

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.

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.

The **`PowderModel`** internally uses **`PowderDiffraction`** for its calculations which is based on the fundamental parameters approach as implemented and documented here and here.

Several setup specific parameters should be adjusted by a user-specific configuration file are 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:

```
import numpy
import xrayutilities as xu

tt = numpy.arange(5, 120, 0.01)
Fe_powder = xu.simpack.Powder(xu.materials.Fe, 1,
                              crystallite_size_gauss=100e-9)
Co_powder = xu.simpack.Powder(xu.materials.Co, 5,  # 5 times more Co
                              crystallite_size_gauss=200e-9)
pm = xu.simpack.PowderModel(Fe_powder, Co_powder, I0=100)
inte = pm.simulate(tt)
```

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

```
import numpy
import xrayutilities as xu
from multiprocessing import freeze_support

def main():
    tt = numpy.arange(5, 120, 0.01)
    Fe_powder = xu.simpack.Powder(xu.materials.Fe, 1,
                                  crystallite_size_gauss=100e-9)
    Co_powder = xu.simpack.Powder(xu.materials.Co, 5,  # 5 times more Co
                                  crystallite_size_gauss=200e-9)
    pm = xu.simpack.PowderModel(Fe_powder, Co_powder, I0=100)
    inte = pm.simulate(tt)

if __name__ == '__main__':
    freeze_support()
    main()
```

xrayutilities package

Subpackages

xrayutilities.analysis package

Submodules

xrayutilities.analysis.line_cuts module

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_ang`**(`qx`, `qz`, `intensity`, `omcenter`, `ttcenter`, `omrange`, `npoints`, `**kwargs`)

extracts an omega scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: omega-position at which the omega scan should be extracted

ttcenter: 2theta-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative omega positions are returned :(default: True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,omint: omega scan coordinates and intensities (bounds=False)

om,omint,(qxb, qzb): omega scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omcut, intcut = get_omega_scan(qx,qz,intensity,0.0,5.0,2.0,200)
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_bounds_ang`**(`omcenter`, `ttcenter`, `omrange`, `npoints`, `**kwargs`)

return reciprocal space boundaries of omega scan

Parameters: **omcenter: omega-position at which the omega scan should be extracted**

ttcenter: 2theta-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qx,qz: reciprocal space coordinates of the omega scan boundaries

Examples

```
>>> qxb,qzb = get_omega_scan_bounds_ang(1.0,4.0,2.4,240,qrange=0.1)
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_q`**(qx, qz, `intensity`, `qxcenter`, `qzcenter`, `omrange`, `npoints`, `**kwargs`)

extracts an omega scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the omega scan should be extracted

qzcenter: qz-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative omega positions are returned :(default: True)

bounds: flag to specify if the scan bounds should be returned; :(default: False)

Returns:

om,omint: omega scan coordinates and intensities (bounds=False)

om,omint,(qxb, qzb): omega scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omcut, intcut = get_omega_scan(qx,qz,intensity,0.0,5.0,2.0,200)
```

`xrayutilities.analysis.line_cuts.`**`get_qx_scan`**(qx, qz, `intensity`, `qzpos`, `**kwargs`)

extract qx line scan at position qzpos from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given range along qz

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

bounds: flag to specify if the scan bounds of the extracted scan should be returned (default:False)

Returns:

qx,qxint: qx scan coordinates and intensities (bounds=False)

qx,qxint,(qxb,qyb): qx scan coordinates and intensities + scan bounds for plotting

Examples

```
>>> qxcut,qxcut_int = get_qx_scan(qx,qz,inten,5.0,qrange=0.03)
```

xrayutilities.analysis.line_cuts.**get_qz_scan**(qx, qz, intensity, qxpos, **kwargs)
extract qz line scan at position qxpos from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given range along qx

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qz,qzint: qz scan coordinates and intensities

Examples

```
>>> qzcut,qzcut_int = get_qz_scan(qx,qz,inten,1.5,qrange=0.03)
```

xrayutilities.analysis.line_cuts.**get_qz_scan_int**(qx, qz, intensity, qxpos, **kwargs)
extracts a qz scan from a gridded reciprocal space map with integration along omega (sample rocking angle) or 2theta direction

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

angrange: integration range in angular direction

qmin,qmax: minimum and maximum value of extracted scan axis

bounds: flag to specify if the scan bounds of the extracted scan should be returned (default:False)

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

wl: wavelength used to determine angular integration positions

Returns:

qz,qzint: qz scan coordinates and intensities (bounds=False)

qz,qzint,(qzb,qzb): qz scan coordinates and intensities + scan bounds for plotting

Examples

```
>>> qzcut,qzcut_int = get_qz_scan_int(qx,qz,inten,5.0,omrange=0.3)
```

xrayutilities.analysis.line_cuts.**get_radial_scan_ang**(qx, qz, intensity, omcenter, ttcenter, ttrange, npoints, **kwargs)
extracts a radial scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: om-position at which the radial scan should be extracted

ttcenter: tt-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,tt,radint: omega,two theta scan coordinates and intensities (bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omc, ttc, cut_int = get_radial_scan_ang(qx, qz, intensity, 32.0, 64.0,
                                            30.0, 800, omrange = 0.2)
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan_bounds_ang`**(`omcenter`, `ttcenter`, `ttrange`, `npoints`, `**kwargs`)

return reciprocal space boundaries of radial scan

Parameters: **omcenter: om-position at which the radial scan should be extracted**

ttcenter: tt-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qxrad,qzrad: reciprocal space boundaries of radial scan

Examples

```
>>>
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan_q`**(qx, qz, `intensity`, `qxcenter`, `qzcenter`, `ttrange`, `npoints`, `**kwargs`)

extracts a radial scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the radial scan should be extracted

qzcenter: qz-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,tt,radint: omega,two theta scan coordinates and intensities (bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omc, ttc, cut_int = get_radial_scan_q(qx, qz, intensity, 0.0, 5.0,
                                          1.0, 100, omrange = 0.01)
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_ang`**(`qx`, `qz`, `intensity`, `omcenter`, `ttcenter`, `ttrange`, `npoints`, `**kwargs`)

extracts a twotheta scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: om-position at which the 2theta scan should be extracted

ttcenter: tt-position at which the 2theta scan should be extracted

ttrange: two theta range of the scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

tt,ttint: two theta scan coordinates and intensities (bounds=False)

tt,ttint,(qxb,qzb): 2theta scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> ttc,cut_int = get_ttheta_scan_ang(qx,qz,intensity,32.0,64.0,4.0,400)
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_bounds_ang`**(`omcenter`, `ttcenter`, `ttrange`, `npoints`, `**kwargs`)

return reciprocal space boundaries of 2theta scan

Parameters: **omcenter: om-position at which the 2theta scan should be extracted**

ttcenter: tt-position at which the 2theta scan should be extracted

ttrange: two theta range of the 2theta scan to extract

npoints: number of points of the 2theta scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qxtt,qztt: reciprocal space boundaries of 2theta scan (bounds=False)

tt,ttint,(qxb,qzb): 2theta scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>>
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_q`**(`qx`, `qz`, `intensity`, `qxcenter`, `qzcenter`, `ttrange`, `npoints`, `**kwargs`)

extracts a twotheta scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the 2theta scan should be extracted

qzcenter: qz-position at which the 2theta scan should be extracted

ttrange: two theta range of the scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

tt,ttint: two theta scan coordinates and intensities (bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> ttc,cut_int = get_ttheta_scan_q(qx,qz,intensity,0.0,4.0,4.4,440)
```

xrayutilities.analysis.line_cuts.**getindex** (x, y, xgrid, ygrid)

gives the indices of the point x,y in the grid given by xgrid ygrid xgrid,ygrid must be arrays containing equidistant points

Parameters: **x,y: coordinates of the point of interest (float)**

xgrid,ygrid: grid coordinates in x and y direction (array)

Returns:

ix,iy: index of the closest gridpoint (lower left) of the point (x,y)

xrayutilities.analysis.line_cuts3d module

xrayutilities.analysis.line_cuts3d.**get_qx_scan3d** (gridder, qypos, qzpos, **kwargs)

extract qx line scan at position y,z from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qypos,qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qx,qxint: qx scan coordinates and intensities

Examples

```
>>> qxcut,qxcut_int = get_qx_scan3d(gridder,0,0,qrange=0.03)
```

xrayutilities.analysis.line_cuts3d.**get_qy_scan3d** (gridder, qxpos, qzpos, **kwargs)

extract qy line scan at position x,z from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qxpos,qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qy,qyint: qy scan coordinates and intensities

Examples

```
>>> qycut,qycut_int = get_qy_scan3d(gridder,0,0,qrange=0.03)
```

xrayutilities.analysis.line_cuts3d.**get_qz_scan3d** (gridder, qxpos, qypos, **kwargs)

extract qz line scan at position x,y from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qxpos,qypos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qz,qzint: qz scan coordinates and intensities

Examples

```
>>> qzcut,qzcut_int = get_qz_scan3d(gridder,0,0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts3d.`**`getindex3d`** (`x`, `y`, `z`, `xgrid`, `ygrid`, `zgrid`)

gives the indices of the point x,y,z in the grid given by xgrid ygrid zgrid xgrid,ygrid,zgrid must be arrays containing equidistant points

Parameters: **x, y, z: coordinates of the point of interest (float)**

xgrid, ygrid, zgrid: grid coordinates in x, y, z direction (array)

Returns: ix, iy, iz: index of the closest gridpoint (lower left) of the point

(x, y, z)

xrayutilities.analysis.misc module

miscellaneous functions helpful in the analysis and experiment

`xrayutilities.analysis.misc.`**`getangles`** (`peak`, `sur`, `inp`)

calculates the chi and phi angles for a given peak

Parameters: **peak: array which gives hkl for the peak of interest**

sur: hkl of the surface

inp: inplane reference peak or direction

Returns: [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.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 ..... outer detector arm angle**

angle2: inner detector arm angle

ccdimages: images of the ccd taken at the angles given above

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

r_i: primary beam direction [xyz][+-] default 'x+'

keyword_arguments:

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

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

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

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

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

nwindow: 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: 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, 'config'), fix=(False, False, True, 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 .. 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: outer detector arm angle

angle2: inner detector arm angle

ccdimages: images of the ccd taken at the angles given above

hkls: array/list of hkl values for every image

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

material: material used as reference crystal

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

r_i: primary beam direction [xyz][+-] default 'x+'

keyword_arguments:

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

cut_off: cut off intensity to decide if image is used for the determination or not. default: 0.1 = 10%

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

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

nwindow: 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: flag to tell if you want to see debug output of the script (switch this to true only if you can handle it :))

`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: angular coordinates of the measurement (numpy.ndarray)**

either with size of psd or of psd.shape[0]

psd: intensity values needed for fitting

omalign: aligned omega value, used as first guess in the fit

ttalign: 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 class used for the conversion between angular and reciprocal space.

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

peaktype: can be 'Gauss' or 'Lorentz' to fit either of the two peak shapes

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

Returns:

omfit,ttfit,params,covariance: fitted angular values, and the fit parameters (of the Gaussian/Lorentzian) as well as their errors

`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 of angles in degree of measured detector spectra**

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

keyword arguments:

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

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

other options are passed to psd_chdeg function, options include:

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

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

Note: see help of psd_chdeg for more options

Returns: pixelwidth (at one meter distance), center_channel[, detector_tilt]

Note

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

distance L
pixelwidth is negative in case the hit channel number decreases upon an increase of the detector angle The function also prints out how a linear detector can be initialized using the results obtained from this calibration. Carefully check the results

`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: azimuths in which the reflection was aligned (deg)**

aomega: aligned omega values (deg)

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

omega0: 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: flag to specify if a visualization of the fit is wanted. :default: True

Returns: [omega0,phi0,miscut]

list with fitted values for

omega0: the omega value of the reflection should be close to the nominal one

phi0: the azimuth in which the primary beam looks upstairs

miscut: amplitude of the sinusoidal variation == miscut angle

`xrayutilities.analysis.sample_align.`**psd_chdeg** (`angles`, `channels`, `stdev=None`, `usetilt=True`, `plot=True`, `datap='kx'`, `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: detector angles for which the position of the beam was**

measured

channels: detector channels where the beam was found

keyword arguments:

stdev: standard deviation of the beam position

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

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

datap: plot format of data points

modelline: plot format of modelline

modeltilt: plot format of modeltilt

fignum: figure number to use for the plot

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

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

dlabel: label of the data line to be used in the plot

figtitle: boolean to tell if the figure title should show the fit parameters

Returns: (pixelwidth,centerch,tilt)

pixelwidth: 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: center channel of the detector

tilt: tilt of the detector from perpendicular to the beam

(will be zero in case of usetilt=False)

Note

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

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** : primary beam channel number

angles: list or numpy.array with angles

channels: list or numpy.array with corresponding detector channels

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

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

Examples

```
>>> psd_refl_align(500,[0,0.1,0.2,0.3],[550,600,640,700])
```

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: **xrayutilities.io.filedir.FileDirectory**

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

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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

xrayutilities.io.cbf.**makeNaturalName**(name)

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.`**`gettty08_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: name of the scans, for multiple scans this needs to be a**

template string

scannumbers: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent)

**keyargs: keyword arguments are passed on to tty08File

Returns: MAP

or

[ang1,ang2,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with 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.
Required constructor arguments:

filename: a string with the name of the tty08-file

Optional keyword arguments:

mcadir: directory name of MCA files

`Read``()`

Read the data from the file

`ReadMCA``()`

xrayutilities.io.edf module

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

Bases: **`xrayutilities.io.filedir.FileDirectory`**

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

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: 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 ..... a HDF5 file object or name**

optional keyword arguments:

group: group where to store the data (default to the root of the file)
comp: activate compression - true by default

data

`xrayutilities.io.edf.`**`makeNaturalName`** (`name`)

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.

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

Parameters: **nx,ny: number of bins in x,y direction**

optional keyword arguments:

counter: name of the counter to use for gridding (default: 'mpx4int' (ID01))
gridrange: range for the gridder: format: ((xmin,xmax),(ymin,ymax))

Returns: 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: name of the motor for which the position is wanted**

Returns:

val: 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** (filename, scannr, xmotor='adcX', ymotor='adcY', path='')

Bases: **xrayutilities.io.fastscan.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

getccdFileTemplate (specscan**,** datadir=None**,** keepdir=0**,** numfmt='%04d'**)**

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

Parameters: **specscan: spec-scan object from which header the CCD directory should**

be extracted

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

numfmt: format string for the CCD file number (optional)

Returns:

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

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

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

Parameters: **nx,ny: number of bins in x,y direction**

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

optional:

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

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

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

filterfunc: 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: regular x,y-grid as well as 4-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.

`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: list of shifts in x-direction for every FastScan in the**

data structure

deltay: same for the y-direction

`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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

optional:

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

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

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: number of bins in x,y direction**

optional keyword arguments:

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

gridrange: 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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

qnx: number of points in the Qx direction of the gridded reciprocal space map

qny: same for y direction

qnz: same for z directino

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

optional:

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

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

filterfunc: 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: 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, 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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

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

optional:

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

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

filterfunc: 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: sample orientation matrix

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

Returns:

qx,qy,qz,ccddata,valuelist: raw data of the reciprocal space map and

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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

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

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.

`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: filename of the file to open (full including path)**

mode: mode in which the file should be opened

Returns: file handle of the opened file

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.

Required constructor arguments:

filename: a string with the name of the data file

`Read``()`

Read the data from the 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: number of the numors, or list of numbers. This will be**

transformed to a string and used as a filename (int, str, or iterable (list, tuple))

***args: names of the motors (optional) (strings)**

e.g.: 'omega', 'gamma'

****kwargs: keyword arguments are passed on to numorFile. e.g. 'path' for**

the files directory

Returns: data

or

[ang1,ang2,...],data:

angular positions position together with 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=<type '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.

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: 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

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

Bases: **xrayutilities.io.imagereader.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.

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

Bases: **xrayutilities.io.imagereader.ImageReader**

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

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

Bases: **xrayutilities.io.imagereader.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.

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

Bases: **xrayutilities.io.imagereader.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.

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

Parameters: **filename: filename of the image to be read. so far only single**

filenames are supported. The data might be compressed.

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

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

`xrayutilities.io.panalytical_xml.`**`getOmPixcel`**(`omraw`, `ttraw`)
function to reshape the Omega values into a form needed for further treatment with xrayutilities

`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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames

scannrs: int or list of scan numbers

path: common path to the filenames

roi: region of interest for the PIXCel detector, for other measurements this is not usefull!

Returns:

om,tt,psd: 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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames given by the scannrs keyword argument

***motors: motor names to return: e.g.: 'Omega','2Theta',...**

one can also use abbreviations 'Omega' = 'om' = 'o' '2Theta' = 'tt' = 't' 'Chi' = 'c' 'Phi' = 'p'

****kwargs:**

scannrs: int or list of scan numbers

path: common path to the filenames

Returns: scanmot,mot1,mot2,...,detectorint: as flattened numpy arrays

Examples

```
>>> scanmot,om,tt,inte = xrayutilities.io.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

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

Bases: **xrayutilities.io.pdcif.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

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

Required constructor arguments:

filename: a string with the name of the ras-file

keyword argument (optional):

path: path to the data file

Read ()
Read the data from the file

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
Required constructor arguments:

filename: file name of the data file

pos: seek position of the RAS_HEADER_START line

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: name of the scans, for multiple scans this needs to be a**

template string

scannumbers: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent) **kwargs: keyword arguments forwarded to RASFile function

Returns: rasdata

or

[ang1,ang2,...],rasdata:

angular positions are extracted from the respective scan header together with all the data values as stored in the data file (includes the intensities e.g. rasdata['int']).

Examples

```
>>> [om,tt],MAP = xu.io.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.

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: 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 use 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

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

parse ()

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

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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames

scannrs: int or list of scan numbers

path: common path to the filenames

scantype: 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: number of channels of the MCA (needed for "tsk" measurements)

Returns:

om,tt,psd: as flattened numpy arrays

Examples

```
>>> om,tt,psd = xrayutilities.io.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**

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={})
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.

required arguments:

h5f: a HDF5 file object or its filename

optional keyword arguments:

comp: 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.

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

Parse()

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={}, 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.

input arguments:

h5f: a HDF5 file object or its filename

optional keyword arguments:

group: name or group object of the HDF5 group where to store the data
title: a string with the title for the data, defaults to the name of scan if empty
optattrs: a dictionary with optional attributes to store for the data
comp: activate compression - true by default

SetMCAParams (`mca_column_format`, `mca_channels`, `mca_start`, `mca_stop`)
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

required input aguments:

mca_column_format: number of columns used to save the data
mca_channels: number of MCA channels stored
mca_start: first channel that is stored
mca_stop: last channel that is stored

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: 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:**

newfig: if True a new figure instance will be created otherwise an existing one will be used
logy: 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 object of a HDF5 file opened using h5py or its filename**

scans: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent)

****kwargs (optional):**

samplename: string with the hdf5-group containing the scan data if ommited the first child node of h5f.root will be used
rettype: how to return motor positions. by default a list of arrays is returned. when rettype == 'numpy' a record array will be returned.

Returns: MAP

or

[ang1,ang2,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with all the data values as stored in the data file (includes the intensities e.g. MAP['MCA']).

Examples

```
>>> [om, tt], MAP = xu.io.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 object**

scans: number of the scans of the reciprocal space map (int,tuple or list)

args: names of the motors and counters (strings)

keyword arguments:

rettype: 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,...]:

coordinates and counters from the SPEC file

Examples

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

`xrayutilities.io.spec.`**`makeNaturalName`**(`name`)

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.
Required constructor arguments:

filename: a string with the name of the SPECTRA file

Optional keyword arguments:

mcatmp: template for the MCA files

mcastart,mcastop: 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.

required input arguments:

h5file: string or HDF5 file object
name: name of the group where to store the data

optional keyword arguments:

group: root group where to store the data
mcaname: Name of the MCA in the HDF5 file

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

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.

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

append (`col`**)**

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

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

`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 object of a HDF5 file opened using h5py**

scans: number of the scans of the reciprocal space map (int,tuple or list)

***args: arbitrary number of motor names (strings)**

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

****kwargs (optional):**

mca: name of the mca data (if available) otherwise None :(default: "MCA")
samplename: 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,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with all 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**

f(q, en='config')
function to calculate the atomic structure factor F

Parameters: **q: momentum transfer**

en: energy for which F should be calculated, if omitted the value from the xrayutilities configuration is used

Returns: f (float)

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: flag, result: if the flag is True then result contains the cached result, otherwise result is None

max_cache_length = *10*

set_cache(prop, key, result)
set result to be cached to speed up future calls

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.**CIFFile**(filename, digits=3)
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

Lattice()
returns a lattice object with the structure from the CIF file

Parse()
function to parse a CIF file. 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.

xrayutilities.materials.cif.**testwp** (parint, wp, cifpos, digits)
test if a Wyckoff position can describe the given position from a CIF file

Parameters: **parint: integer telling which Parameters the given Wyckoff position has**

wp: expression of the Wyckoff position (string of tuple)
cifpos: (x,y,z) position of the atom in the CIF file
digits: number of digits for which for a comparison of floating point numbers will be rounded to

Returns: foundflag, pars: flag to tell if the positions match and if necessary any parameters associated with the position

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.

required input arguments:

dbname: name of the database
dbdesc: 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.

required input arguments:

name: a string with the name of the material
description: a string with a description of the material

GetF0 (q**,** dset='default'**)**
Obtain the f0 scattering factor component for a particular momentum transfer q.

required input argument:

q: single float value or numpy array
dset: 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.

required input arguments:

en: float or numpy array with the energy

GetF2 (en**)**
Return the imaginary part of the scattering factor for a certain energy en.

required input arguments:

en: float or numpy array with the energy

Open (`mode='r'`**)**
Open an existing database file.

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

required input argument:

parameters: list or numpy array with the fit parameters
subset: specifies under which name the f0 values should be saved

SetF1F2 (`en`**,** `f1`**,** `f2`**)**
Set f1, f2 values for the active material.

required input arguments:

en: list or numpy array with energy in (eV)
f1: list or numpy array with f1 values
f2: list or numpy array with 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.

required input arguments:

name: string with the name of the material

SetWeight (`weight`**)**
Save weight of the element as float

required input argument:

weight: atomic standard weight of the element (float)

`xrayutilities.materials.database.`**add_f0_from_intertab** (`db`, `itf`)
Read f0 data from International Tables of Crystallography and add it to the database.

`xrayutilities.materials.database.`**add_f0_from_xop** (`db`, `xop`)
Read f0 data from f0_xop.dat and add it to the database.

`xrayutilities.materials.database.`**add_f1f2_from_ascii_file** (`db`, `asciifile`, `element`)
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`)
Read f1 and f2 data from Henke database and add it to the database.

`xrayutilities.materials.database.`**add_f1f2_from_kissel** (`db`, `kf`)
Read f1 and f2 data from Henke database and add it to the database.

`xrayutilities.materials.database.`**add_mass_from_NIST** (`db`, `nistfile`)
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.`**init_material_db** (`db`)

xrayutilities.materials.elements module

xrayutilities.materials.lattice module

module handling crystal lattice structures. A Lattice consists of unit cell parameters and a LatticeBase. It offers methods to calculate the reciprocal space position of Bragg peaks and their structure factor.

xrayutilities.materials.lattice.**AlGaAsLattice** (aal, aga, aas, a, x)

xrayutilities.materials.lattice.**BCCLattice** (aa, a)

xrayutilities.materials.lattice.**BCTLattice** (aa, a, c)

xrayutilities.materials.lattice.**BaddeleyiteLattice** (aa, ab, a, b, c, beta)

xrayutilities.materials.lattice.**CsClLattice** (aa, ab, a)

xrayutilities.materials.lattice.**CubicFm3mBaF2** (aa, ab, a)

xrayutilities.materials.lattice.**CubicLattice** (a, base=None)
Returns a Lattice object representing a cubic lattice.

Parameters: **a: lattice parameter**

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**DiamondLattice** (aa, a)

xrayutilities.materials.lattice.**FCCLattice** (aa, a)

xrayutilities.materials.lattice.**FCCSharedLattice** (aa, ab, occa, occb, a)

xrayutilities.materials.lattice.**GeTeRhombohedral** (aa, ab, a, ang, x=0.237)

xrayutilities.materials.lattice.**HCPLattice** (aa, a, c)

xrayutilities.materials.lattice.**Hexagonal3CLattice** (aa, ab, a, c)

xrayutilities.materials.lattice.**Hexagonal4HLattice** (aa, ab, a, c, u=0.1875, v1=0.25, v2=0.4375)

xrayutilities.materials.lattice.**Hexagonal6HLattice** (aa, ab, a, c)

xrayutilities.materials.lattice.**HexagonalLattice** (a, c, base=None)
Returns a Lattice object representing a hexagonal lattice.

Parameters: **a: lattice parameter a**

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**LaB6Lattice** (aa, ab, a, oa=1, ob=1, ba=0, bb=0)

class xrayutilities.materials.lattice.**Lattice** (a1, a2, a3, base=None)
Bases: **object**
class Lattice: This object represents a Bravais lattice. A lattice consists of a base and unit cell defined by three vectors.

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:: Note: NO elastic response of the material will be considered!

requiered input arguments:

eps: a 3x3 matrix independent strain components

GetPoint (*args)
determine lattice points with indices given in the argument

Examples

```
>>> 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])
```

ReciprocalLattice ()

UnitCellVolume ()
function to calculate the unit cell volume of a lattice (angstrom^3)

a

a1

a2

a3

alpha

b

beta

c

gamma

class xrayutilities.materials.lattice.**LatticeBase** (*args, **keyargs)
Bases: **list**
The LatticeBase class implements a container for a set of points that form the base of a crystal lattice. An instance of this class can be treated as a simple container object.

append (atom**,** pos**,** occ=1.0**,** b=0.0**)**
add new Atom to the lattice base

Parameters: **atom: atom object to be added**

pos: position of the atom

occ: occupancy (default=1.0)

b: 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)

xrayutilities.materials.lattice.**MagnetiteLattice** (aa, ab, ac, a, x=0.255)

xrayutilities.materials.lattice.**MonoclinicLattice** (a, b, c, beta, base=None)
Returns a Lattice object representing a hexagonal lattice.

Parameters: **a: lattice parameter a**

b: lattice parameter b

c: lattice parameter c

beta: monoclinic unit cell angle beta (deg)

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**NaumanniteLattice** (aa, ab, a, b, c)

xrayutilities.materials.lattice.**NiAsLattice** (aa, ab, a, c, biso=0.0)

xrayutilities.materials.lattice.**OrthorhombicLattice** (a, b, c, base=None)
Returns a Lattice object representing a tetragonal lattice.

Parameters: **a: lattice parameter a**

b: lattice parameter b

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**PerovskiteTypeRhombohedral** (aa, ab, ac, a, ang)

xrayutilities.materials.lattice.**QuartzLattice** (aa, ab, a, b, c)

xrayutilities.materials.lattice.**RockSaltLattice** (aa, ab, a)
creates the primitive unit cell of a RockSalt structure. For the more commonly used cubic reprentation see RockSalt_Cubic_Lattice

xrayutilities.materials.lattice.**RockSalt_Cubic_Lattice** (aa, ab, a)

xrayutilities.materials.lattice.**RutileLattice** (aa, ab, a, c, u)

xrayutilities.materials.lattice.**SiGeLattice** (asi, age, a, xge)

xrayutilities.materials.lattice.**TetragonalIndiumLattice** (aa, a, c)

xrayutilities.materials.lattice.**TetragonalLattice** (a, c, base=None)
Returns a Lattice object representing a tetragonal lattice.

Parameters: **a: lattice parameter a**

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**TetragonalTinLattice** (aa, a, c)

xrayutilities.materials.lattice.**TriclinicLattice** (a, b, c, alpha, beta, gamma, base=None)

xrayutilities.materials.lattice.**TrigonalR3mh** (aa, a, c)

xrayutilities.materials.lattice.**WurtziteLattice** (aa, ab, a, c, u=0.375, biso=0.0)

xrayutilities.materials.lattice.**ZincBlendeLattice** (aa, ab, a)

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: **xrayutilities.materials.material.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** : Miller Indices

sub: substrate material or lattice constant (Instance of Crystal class or float)

exp: Experiment class from which the Transformation object and ndir are needed

Returns: **qy,qz** : reciprocal space coordinates of the corners of the relaxation triangle

static **check_compatibility**(matA, matB)

static **lattice_const_AB**(latA, latB, x, name='')

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: property (lattice parameter/angle) of material A and B.**

A property can be a scalar or vector.

x: fraction of material B in the alloy.

name: label of the property which is interpolated. Can be 'a', 'b', 'c', 'alpha', 'beta', or 'gamma'.

x

class xrayutilities.materials.material.**Amorphous**(name, density, atoms=None, cij=None)

Bases: **xrayutilities.materials.material.Material**

amorphous materials are described by this class

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: x-ray energy eV, if omitted the value from the xrayutilities** configuration is used

Returns: delta (float)

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: x-ray energy eV, if omitted the value from the xrayutilities** configuration is used

Returns: beta (float)

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: string containing the chemical fomula**

Returns: list of tuples with 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.

required input arguments:

cij: (6,6) cij matrix as a numpy array

return value:

cijkl: (3,3,3,3) cijkl 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.

required input arguments:

cijkl: (3,3,3,3) cijkl tensor as numpy array

return value:

cij: (6,6) cij matrix as a numpy array

class `xrayutilities.materials.material.`**`Crystal`**`(name, lat, cij=None, thetaDebye=None)`
Bases: **`xrayutilities.materials.material.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. Note: NO elastic response of the material will be considered!

`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 numpy array with the 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 numpy array with the 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: vectorial momentum transfer (vectors as list,tuple**

or numpy array are valid)

en: energy in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

Returns: 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: vectorial momentum transfer (vectors as list,tuple**

or numpy array are valid)

en: list, tuple or array of energy values in eV

temp: temperature used for Debye-Waller-factor calculation

Returns: 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: 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: energy value in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

Returns: complex valued structure factor array

a

a1

a2

a3

alpha

b

beta

c

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: momentum transfer in (1/A)**

en: xray energy in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

polarization: either 'S' (default) sigma or 'P' pi polarization

Returns: (abs(chih_real),abs(chih_imag)) complex polarizability

dTheta (Q, en='config')
function to calculate the refractive peak shift

Parameters: **Q: momentum transfer (1/A)**

en: x-ray energy (eV), if omitted the value from the xrayutilities configuration is used

Returns: **deltaTheta:** 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: x-ray energy eV, if omitted the value from the xrayutilities**
configuration is used

Returns: delta (float)

density
calculates the mass density of an material from the mass of the atoms in the unit cell.

Returns: 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

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 the unit cell.

Parameters: **pos: list or numpy array with the fractional coordinated in the**
unit cell

keyword arguments:

maxdist: maximum distance wanted in the list of neighbors :(default: 7)

Returns: list of tuples with (distance,atomType,multiplicity) giving distance
(sorted) and type of neighboring atoms together with the amount of atoms at the given distance

classmethod **fromCIF**(ciffilename)
Create a Crystal from a CIF file. The CIF-filename will be used as name of the created Crystal. Note: since the CIF file parser is currently not able to detect the correct space group of the material all materials created by this method will be represented by the P1 space-group!

Parameters: **ciffilename: filename of the CIF file**

Returns: Crystal instance

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: x-ray energy eV, if omitted the value from the xrayutilities**
configuration is used

Returns: beta (float)

planeDistance(*hkl)
determines the lattice plane spacing for the planes specified by (hkl)

Parameters: **h,k,l: Miller indices of the lattice planes given either as**
list,tuple or seperate arguments

Returns: **d:** the lattice plane spacing as float

Examples

```
>>> xu.materials.Si.planeDistance(0,0,4)
1.3577600000000001
```

or

```
>>> xu.materials.Si.planeDistance((1,1,1))
3.1356124059796255
```

class xrayutilities.materials.material.**CubicAlloy** (matA, matB, x)
Bases: **xrayutilities.materials.material.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** : inplane peak position of reflection hkl of the alloy in reciprocal space

q_perp: perpendicular peak position of the reflection hkl of the alloy in reciprocal space

hkl: Miller indices of the measured asymmetric reflection

sur: Miller indices of the surface (determines the perpendicular direction)

Returns: content, [a_inplane, a_perp, a_bulk_perp(x), eps_inplane, eps_perp]:

the content of B in the alloy determined from the input variables and the lattice constants 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** : perpendicular peak position of the reflection hkl of the alloy in reciprocal space

hkl: Miller indices of the measured symmetric reflection (also defines the surface normal

inpr: Miller indices of a Bragg peak defining the inplane reference direction

asub: substrate lattice constant

relax: degree of relaxation (needed to obtain the content from symmetric reciprocal space position)

Returns: **content** : the content of B in the alloy determined from the input variables

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: independent components of the elastic tensor of cubic** materials

Returns: 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: independent components of the elastic tensor of**

a hexagonal material

Returns: 6x6 matrix with elastic constants

class xrayutilities.materials.material.**Material**(name, cij=None)

Bases: **abc.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.

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: energy of the x-rays (in eV, optional)**

Returns: 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: energy of the x-rays, if omitted the value from the**

xrayutilities configuration is used

deg: return angle in degree if True otherwise radians (default:True)

Returns: Angle of total external reflection

delta(en='config')

abstract method which every implementation of a Material has to override

density

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)

lam

mu

nu

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.

Parameters: **sub: substrate material**

layer: bulk material of the layer

relaxation: degree of relaxation 0: pseudomorphic, 1: relaxed :(default: 0)

trans: 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.

`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: independent components of the elastic tensor of cubic**

materials

Returns: 6x6 matrix with elastic constants

Implementation according to a patch submitted by Julian Stangl

`xrayutilities.materials.material.`**`index_map_ij2ijkl`**(`ij`)

`xrayutilities.materials.material.`**`index_map_ijkl2ij`**(`i`, `j`)

xrayutilities.materials.predefined_materials module

class `xrayutilities.materials.predefined_materials.`**`AlGaAs`**(`x`)

Bases: **`xrayutilities.materials.material.CubicAlloy`**

class `xrayutilities.materials.predefined_materials.`**`SiGe`**(`x`)

Bases: **`xrayutilities.materials.material.CubicAlloy`**

static **`lattice_const_AB`**(`latA`, `latB`, `x`, `**kwargs`)

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.`**`RangeDict`**

Bases: **`dict`**

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.
this replaces the deprecated Lattice class

`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:: Note: Here you specify the strain and not the stress -> NO elastic

response of the material will be considered!

Parameters: **eps: a 3x3 matrix with all strain components**

`GetHKL`(`*args`)

determine the Miller indices of the given reciprocal lattice points

GetPoint (*args)
determine lattice points with indices given in the argument

Examples

```
>>> 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)

a

alpha

b

base ()
generator of atomic position within the unit cell.

beta

c

classmethod **convert_to_P1** (sglat)
create a P1 equivalent of the given SGLattice instance.

Parameters: **sglat: space group lattice instance to be converted to P1.**

Returns: SGLattice instance with the same properties as sglat, however in the P1 setting.

classmethod **fromLattice** (lat, verbose=True)
create a SGLattice from an old Lattice instance. Since the space-group is not known it will always be 1 (triclinic). This is helper routine to make the transition period for users easier. It will be removed in the next major release!

Parameters: **lat: deprecated Lattice instance**

Returns: SGLattice instance with the same properties as lat

gamma

isequivalent (hkl1, hkl2, equalq=False)
primitive way of determining if hkl1 and hkl2 are two crystallographical equivalent pairs of Miller indices

Parameters: **hkl1,2: Miller indices to be checked for equivalence**

equalq: If False the length of the two q-vactors will be compared. If True it is assumed that the length of the q-vectors of hkl1 and hkl2 is equal!

Returns: True or False

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.

append (`atom`**,** `pos`**,** `occ=1.0`**,** `b=0.0`**)**
add new Atom to the lattice base

Parameters: **atom: atom object to be added**

pos: Wyckoff position of the atom, along with its parameters. :Examples: ('2i', (0.1, 0.2, 0.3)), or '1a'

occ: occupancy (default=1.0)

b: 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)

`xrayutilities.materials.spacegrouplattice.`**`get_default_sgrp_suf`** (`sgrp_nr`)
determine default space group suffix

xrayutilities.materials.wyckpos module

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:
a4*z**4 + a3*z**3 + a2*z**2 + a1*z + a0

Returns: tuple of the four (complex) solutions of aboves equation.

[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,y: data coordinates (do NOT need to be regularly spaced)**

data: data set used for fitting (e.g. intensity at the data coords)

start: set of starting parameters for the fit used as first parameter of function fit_function

drange: 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: [xmin,xmax,ymin,ymax]

fit_function: function which should be fitted, must accept the parameters (x,y,*params)

Returns:

(fitparam,cov): the set of fitted parameters and covariance matrix

`xrayutilities.math.fit.`**`gauss_fit`** (`xdata`, `ydata`, `iparams=[]`, `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: xcoordinates of the data to be fitted**

ydata: ycoordinates of the data which should be fit

keyword parameters:

iparams: initial paramters for the fit, determined automatically if not given

maxit: maximal iteration number of the fit

Returns: params,sd_params,itlim

the Gauss parameters as defined in function Gauss1d(x, *param) and their

errors of the fit, as well as a boolean flag which is false in the case of

a successful fit

`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: data coordinates and values**

Returns: p, rsq: parameters of the linear fit (slope, offest) and the R^2 value

Examples

```
>>> (k, d), R2 = xu.math.linregress(x, y)
```

`xrayutilities.math.fit.`**`multGaussFit`**`(*args, **kwargs)`

convenience function to keep API stable see multPeakFit for documentation

`xrayutilities.math.fit.`**`multGaussPlot`**`(*args, **kwargs)`

convenience function to keep API stable see multPeakPlot for documentation

`xrayutilities.math.fit.`**`multPeakFit`**`(x, data, peakpos, peakwidth, dranges=None, peaktype='Gaussian')`

function to fit multiple Gaussian/Lorentzian peaks with linear background to a set of data

Parameters: **x: x-coordinate of the data**

data: data array with same length as x

peakpos: initial parameters for the peak positions

peakwidth: initial values for the peak width

dranges: 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: type of peaks to be used: can be either 'Gaussian' or 'Lorentzian'

Returns: pos,sigma,amp,background

pos: list of peak positions derived by the fit

sigma: list of peak width derived by the fit

amp: list of amplitudes of the peaks derived by the fit

background: array of background values at positions x

`xrayutilities.math.fit.`**`multPeakPlot`**`(x, fpos, fwidth, famp, background, dranges=None, peaktype='Gaussian', fig='xu_plot', fact=1.0)`

function to plot multiple Gaussian/Lorentz peaks with background values given by an array

Parameters: **x: x-coordinate of the data**

fpos: list of positions of the peaks
fwidth: list of width of the peaks
famp: list of amplitudes of the peaks
background: array with background values
dranges: list of tuples with (min,max) value of the data ranges to use. does not need to have the same number of entries as fpos
peaktype: type of peaks to be used: can be either 'Gaussian' or 'Lorentzian'
fig: matplotlib figure number or name
fact: factor to use as multiplicator in the plot

`xrayutilities.math.fit.`**`peak_fit`** (`xdata`, `ydata`, `iparams=[]`, `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: xcoordinates of the data to be fitted**

ydata: ycoordinates of the data which should be fit

keyword parameters:

iparams: initial paramters for the fit, determined automatically if not specified
peaktype: type of peak to fit: 'Gauss', 'Lorentz', 'PseudoVoigt', 'PseudoVoigtAsym', 'PseudoVoigtAsym2'
maxit: maximal iteration number of the fit
background: type of background, either 'constant' or 'linear'
plot: 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: returns the fitted function, which takes the independent variables as only argument (f(x))

Returns: params,sd_params,itlim[,fitfunc]

the parameters as defined in function Gauss1d/Lorentz1d/PseudoVoigt1d/

PseudoVoigt1dasym(x, *param). In the case of linear background one more

parameter is included! For every parameter the corresponding errors of the

fit 'sd_params' are returned. A boolean flag 'itlim', which is False in

the case of a successful fit is added by default. Further 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 as needed for the calculation of the thermal Debye-Waller-factor by numerical integration
for definition see: :http: //en.wikipedia.org/wiki/Debye_function
D1(x) = (1/x) int_0^x t/(exp(t)-1) dt

Parameters: **x ... argument of the Debye function (float)**

Returns:
D1(x): float value of the Debye function

`xrayutilities.math.functions.`**`Gauss1d`** (x, *p)

function to calculate a general one dimensional Gaussian

Parameters: **p: list of parameters of the Gaussian**

[XCEN,SIGMA,AMP,BACKGROUND] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

x: coordinate(s) where the function should be evaluated

Returns: 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) >>> Gauss1d(numpy.linspace(0,10,100), 5, 1, 1e3, 0)

`xrayutilities.math.functions.`**`Gauss1dArea`** (*p)
function to calculate the area of a Gauss function with neglected background

Parameters: **p: list of parameters of the Gauss-function**

[XCEN,SIGMA,AMP,BACKGROUND]

Returns: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Gauss-function**

[XCEN,YCEN,SIGMAX,SIGMAY,AMP,ANGLE,BACKGROUND]

Returns: 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: **p: list of parameters of the Gauss-function**

[XCEN,YCEN,ZCEN,SIGMAX,SIGMAY,SIGMAZ,AMP,BACKGROUND] SIGMA = FWHM / (2*sqrt(2*log(2)))

x,y,z: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND]

x: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND]

Returns: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Gaussian**

[XCEN,SIGMA] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Lorentzian**

[XCEN,FWHM]

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND,ETA] :ETA: 0 ...1 0 means pure Gauss and 1 means pure Lorentz

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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: coordinate(s) where the function should be evaluated**

p: 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: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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: any scalar of ndarray object**

Returns:

s: Heaviside step function evaluated for all values of x

`xrayutilities.math.functions.`**`kill_spike`**(data, threshold=2.0)
function to smooth **single** data points which differ from the average of the neighboring data points by more than the threshold factor. 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: 1d numpy array with experimental data**

threshold: threshold factor to identify strange data points

Returns: 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: coordinate where the function should be evaluated**

args: 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: 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: coordinates where the function should be evaluated**

args: 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: 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: 1D data array**

n: number of data points to average

Returns:

xsmooth: 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: position of the data points**

data: data values

background: type of background, either 'none', 'constant' or 'linear'

full_output: return background cleaned data and background-parameters

Returns: center of mass position (single float)

`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: position of the data points**

data: data values

Returns: fwhm value (single float)

xrayutilities.math.transforms module

class `xrayutilities.math.transforms.`**`AxisToZ`** (newzaxis)
Bases: **`xrayutilities.math.transforms.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.

class `xrayutilities.math.transforms.`**`AxisToZ_keepXY`** (newzaxis)
Bases: **`xrayutilities.math.transforms.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

class `xrayutilities.math.transforms.`**`CoordinateTransform`** (v1, v2, v3)

Bases: **xrayutilities.math.transforms.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!

class xrayutilities.math.transforms.**Transform**(matrix)
Bases: **object**

inverse(args, rank=1)
performs inverse transformation a vector, matrix or tensor of rank 4

Parameters: **args: 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: rank of the supplied object. allowed values are 1, 2, and 4

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.**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: numpy.array or list of length 3**

axis: numpy.array or list of length 3
ang: rotation angle in degree (deg=True) or in rad (deg=False)
deg: boolean which 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([  6.12323400e-17,   1.00000000e+00,   0.00000000e+00])
```

xrayutilities.math.transforms.**tensorprod**(vec1, vec2)
function implements an elementwise multiplication of two vectors

xrayutilities.math.vector module

module with vector operations for vectors of size 3, since for so short vectors numpy does not give the best performance explicit implementation of the equations is performed together with error checking to ensure vectors of length 3.

xrayutilities.math.vector.**VecAngle**((v1.v2)/(norm(v1)*norm(v2)))

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

optional keyword arguments:

deg: (default: false) return result in degree otherwise in radiants

Returns: float value with the angle inclined by the two vectors

`xrayutilities.math.vector.`**`VecCross`** (`v1`, `v2`, `out=None`)
Calculate the vector cross product.

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

out: optional output vector

Returns: float value

`xrayutilities.math.vector.`**`VecDot`** (`v1`, `v2`)
Calculate the vector dot product.

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

Returns: float value

`xrayutilities.math.vector.`**`VecNorm`** (`v`)
Calculate the norm of a vector.

Parameters: **v ..... vector as list or numpy array**

Returns: float holding the vector norm

`xrayutilities.math.vector.`**`VecUnit`** (`v`)
Calculate the unit vector of v.

Parameters: **v ..... vector as list or numpy array**

Returns: numpy array with the unit vector

`xrayutilities.math.vector.`**`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: vector of length 3**

Returns: [xyz][+-]

`xrayutilities.math.vector.`**`getVector`** (`string`)
returns unit vector along a rotation axis given in the syntax 'x+' 'z-' or equivalents

Parameters: **string [xyz][+-]**

Returns: vector along the given direction as numpy array

Module contents

xrayutilities.simpack package

Submodules

xrayutilities.simpack.darwin_theory module

class `xrayutilities.simpack.darwin_theory.`**`DarwinModel`** (`qz`, `qx=0`, `qy=0`, `**kwargs`)

Bases: **xrayutilities.simpack.models.LayerModel**
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_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: 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: **xrayutilities.simpack.darwin_theory.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 = *5.6532499999999999*

classmethod **abulk** (x)
calculate the bulk (relaxed) lattice parameter of the Al_{x}Ga_{1-x}As alloy

asub = *5.6532499999999999*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors (temp=300)
calculates the needed atomic structure factors

static **poisson_ratio** (x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelAlloy** (qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.DarwinModel**, **abc.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.

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

make_monolayers (s)
create monolayer sequence from layer list

Parameters: **s: 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: inplane lattice parameter of the substrate

Returns: 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: monolayer list created by make_monolayer()**

prop: name of the property which should be evaluated. Use 'x' for the chemical composition and 'ai' for the inplane lattice parameter.

Returns: zm, propx: z-position, value of the property prop for every

monolayer. z=0 is the surface

class xrayutilities.simpack.darwin_theory.**DarwinModelGaInAs001** (qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.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 = *5.6532499999999999*

classmethod **abulk** (x)
calculate the bulk (relaxed) lattice parameter of the Ga_{1-x}In_{x}As alloy

asub = *5.6532499999999999*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors(temp=300**)**
calculates the needed atomic structure factors

static **poisson_ratio** (x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelSiGe001** (qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.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 = *5.4310400000000003*

classmethod **abulk** (x)
calculate the bulk (relaxed) lattice parameter of the alloy

asub = *5.4310400000000003*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors(temp=300)
calculates the needed atomic structure factors

static **poisson_ratio**(x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

xrayutilities.simpack.darwin_theory.**GradedBuffer**(xfrom, xto, nsteps, thickness, relaxation=1)
create a multistep graded composition buffer.

Parameters: **xfrom: begin of the composition gradient**

xto: end of the composition gradient
nsteps: number of steps of the gradient
thickness: total thickness of the Buffer in A
relaxation: relaxation of the buffer

Returns: layer object 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

xrayutilities.simpack.fit.**fit_xrr**(reflmod, params, ai, data=None, eps=None, xmin=-inf, xmax=inf, plot=False, verbose=False, elog=True, maxfev=500)
optimize function for a Reflectivity Model using lmfit. The fitting parameters must be specified as instance of lmfits Parameters class.

Parameters: **reflmod: preconfigured SpecularReflectivityModel**

params: instance of lmfits Parameters class. For every layer the parameters '{}_thickness', '{}_roughness', '{}_density', with '{}' representing the layer name are supported. In addition the setup parameters: - 'I0' primary beam intensity - 'background' background added to the simulation - 'sample_width' size of the sample along the beam - 'beam_width' width of the beam in the same units - 'resolution_width' width of the resolution function in deg - 'shift' experimental shift of the incidence angle array
ai: array of incidence angles for the calculation
data: experimental data which should be fitted
eps: (optional) error bar of the data
xmin: minimum value of ai which should be used. a mask is generated to cut away other data
xmax: maximum value of ai which should be used. a mask is generated to cut away other data
plot: 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.
verbose: flag to tell if the variation of the fitting error should be output during the fit.
elog: logarithmic error during the fit
maxfev: maximum number of function evaluations during the leastsq optimization

Returns:

res: MinimizerResult object from lmfit, which contains the fitted parameters in res.params (see res.params.pretty_print) or try lmfit.report_fit(res)

xrayutilities.simpack.helpers module

`xrayutilities.simpack.helpers.`**`coplanar_alphai`**(qx, qz, en='config')
calculate coplanar incidence angle from knowledge of the qx and qz coordinates

Parameters: **qx: inplane momentum transfer component**

qz: out of plane momentum transfer component

en: x-ray energy (eV). By default the value from the config is used.

Returns: the incidence angle in degree. points in the Laue zone are set to 'nan'.

`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: inplane momentum transfer component**

alphai: incidence angle (deg)

en: x-ray energy (eV). By default the value from the config is used.

Returns: the qz position for the given incidence angle

xrayutilities.simpack.models module

class `xrayutilities.simpack.models.`**`DynamicalModel`**(*args, **kwargs)
Bases: **`xrayutilities.simpack.models.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: vector of incidence angles (deg)**

hkl: Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry: 'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

rettype: 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: vector of intensities of the diffracted signal

class `xrayutilities.simpack.models.`**`KinematicalModel`**(*args, **kwargs)
Bases: **`xrayutilities.simpack.models.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_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: simulation positions along qz**

hkl: Miller indices of the Bragg peak whos truncation rod should be calculated

absorption: flag to tell if absorption correction should be used

refraction: 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: 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: vector of the ratios of the diffracted and primary fluxes

class xrayutilities.simpack.models.**KinematicalMultiBeamModel**(*args, **kwargs)
Bases: **xrayutilities.simpack.models.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.

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: simulation positions along qz**

hkl: Miller indices of the Bragg peak whos truncation rod should be calculated

absorption: flag to tell if absorption correction should be used

refraction: 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: 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: vector of the ratios of the diffracted and primary fluxes

class xrayutilities.simpack.models.**LayerModel**(*args, **kwargs)
Bases: **xrayutilities.simpack.models.Model**, **abc.ABC**
generic model class from which further thin film models can be derived from

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.

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

convolute_resolution (x, y**)**
convolve simulation result with a resolution function

Parameters: **x: x-values of the simulation, units of x also decide about the**

unit of the resolution_width parameter

y: y-values of the simulation

Returns: convoluted y-data with same shape as y

scale_simulation (y**)**
scale simulation result with primary beam flux/intensity and add a background.

Parameters: **y: y-values of the simulation**

Returns: scaled y values

class xrayutilities.simpack.models.**SimpleDynamicalCoplanarModel** (*args, **kwargs)
Bases: **xrayutilities.simpack.models.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

Note: This model should not be used in real life scenarios since the made approximations severely fail for distances far from the reference position.

set_hkl (*hkl**)**
To speed up future calculations of the same Bragg peak optical parameters can be pre-calculated using this function.

Parameters: **hkl: 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: vector of incidence angles (deg)**

hkl: Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry: 'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

idxref: 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: vector of intensities of the diffracted signal

class xrayutilities.simpack.models.**SpecularReflectivityModel** (*args, **kwargs)
Bases: **xrayutilities.simpack.models.LayerModel**
model for specular reflectivity calculations

densityprofile (nz, plot=False)
calculates the electron density of the layerstack from the thickness and roughness of the individual layers

Parameters: **nz: number of values on which the profile should be calculated**

plot: flag to tell if a plot of the profile should be created

Returns: z, eprof: coordinates and electron profile. z = 0 corresponds to the surface

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: vector of incidence angles**

Returns: vector of intensities of the reflectivity signal

xrayutilities.simpack.models.**startdelta** (start, delta, num)

xrayutilities.simpack.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.simpack.mpl_helper.**SqrtAllowNegScale** (axis, **kwargs)
Bases: **matplotlib.scale.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: **matplotlib.transforms.Transform**

input_dims = *1*

inverted ()

is_separable = *True*

output_dims = *1*

transform_non_affine (a)

class SqrtAllowNegScale.**SqrtTransform** (shorthand_name=None)
Bases: **matplotlib.transforms.Transform**

input_dims = *1*

inverted ()
return the inverse transform for this transform.

is_separable = *True*

output_dims = *1*

transform_non_affine (a)
This transform takes an Nx1 numpy array and returns a transformed copy.

SqrtAllowNegScale.**get_transform** ()

SqrtAllowNegScale.**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.

SqrtAllowNegScale.**name** = *'sqrt'*

SqrtAllowNegScale.**set_default_locators_and_formatters** (axis)

class xrayutilities.simpack.mpl_helper.**SqrtTickLocator** (nbins=7, symmetric=True)
Bases: **matplotlib.ticker.Locator**

set_params (nbins, symmetric)
Set parameters within this locator.

tick_values (vmin, vmax)

view_limits (dmin, dmax)
Set the view limits to the nearest multiples of base that contain the data

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)
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.

Noteable class parameters:

max_history_length:	the number of histories to cache (default=5); can be overridden if memory is an issue.
length_scale_m:	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.

add_buffer (b**)**
add a numpy array to the list of objects that can be thrown away on pickling.

Parameters: **b: the buffer to add to the list**

Returns: 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: the edge of the function farthest from the singularity,**

referenced to epsvals

innerbound: the edge closest to the singularity, referenced to epsvals

epsvals: the array of two-theta values or offsets

destination: an array into which final results are summed. modified in place!

peakpos: the position of the singularity, referenced to epsvals.

y0: the constant offset

k: the scale factor

Returns: (*lower_index*, *upper_index*) 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: a list of convolvers to select. If *None*, use all**

found convolvers.

compute_derivative: if *True*, also return d/dx(function) for peak position fitting

Returns: a profile_data object with much information about the peak

conv_absorption ()
compute the sample transparency correction, including the finite-thickness version

Returns: the convolver

conv_axial ()
compute the Fourier transform of the axial divergence component

Returns: 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: the convolver

conv_emission ()
compute the emission spectrum and (for convenience) the particle size widths

Returns: the convolver for the emission and particle sizes

Note

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: 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

conv_receiver_slit ()
compute the rectangular convolution for the receiver slit or SiPSD pixel size

Returns: the convolver

conv_si_psd ()
compute the convolver for the integral of defocusing of the face of an Si PSD

Returns: the convolver

conv_smoother ()
compute the convolver to smooth the final result with a Gaussian before downsampling.

Returns: the convolver

conv_tube_tails ()
compute the Fourier transform of the rectangular tube tails function

Returns: 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: length of the xray filament**

Ls: length of the sample
Lr: length of the receiver slit
R: diffractometer length, assumed symmetrical
twotheta: angle, in radians, of the center of the computation
beta: offset angle
epsvals: array of offsets from center of computation, in radians

Returns: (*epsvals*, *idxmin*, *idxmax*, *I2p*, *I2m*).

idxmin and *idxmax* are the full python-style bounds of the non-zero region of *I2p* and *I2m*. *I2p* and *I2m* are 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: length of the xray filament**

Ls: length of the sample
Lr: length of the receiver slit
R: the (assumed symmetrical) diffractometer radius
twotheta: angle, in radians, of the center of the computation
epsvals: array of offsets from center of computation, in radians
sollerIdeg: the full-width (both sides) cutoff angle of the incident Soller slit
sollerDdeg: the full-width (both sides) cutoff angle of the detector Soller slit
nsteps: the number of subdivisions for the integral
axDiv: not used

Returns: 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: the name of the convolver cache buffer to update**

width: the width in 2-theta space of the tophat

Returns: the updated convolver buffer, or *None* if the width was *None*

get_conv (`name`**,** `key`**,** `format=<type '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: the name of the convolver to seek**

key: any hashable object which identifies the parameters for the computation
format: the type of the array to create, if one is not found.

Returns: 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 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: 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: a number of bins.**

Returns: a bin count somewhat larger than *count* which is efficient for FFT

info_emission = *{'param_info': {'emiss_lor_widths': ('Lorenztian emission fwhm (m)', (1e-13,)), 'crystallite_size_lor': ('Lorentzian crystallite size fwhm (m)', 1e-06), 'emiss_wavelengths': ('wavelengths (m)', (1.58e-10,)), 'emiss_intensities': ('relative intensities', (1.0,)), 'emiss_gauss_widths': ('Gaussian emissions fwhm*

(m)', (1e-13,)), 'crystallite_size_gauss': ('Gaussian crystallite size fwhm (m)', 1e-06)}, 'help': 'this should be help information', 'group_name': 'Incident beam and crystal size'}

info_global = *{'param_info': {'twotheta0_deg': ('Bragg center of peak (degrees)', 30.0), 'd': ('d spacing (m)', 4e-10), 'dominant_wavelength': ('wavelength of most intense line (m)', 1.5e-10)}, 'help': 'this should be help information', 'group_name': 'Global parameters'}*

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 calles when the class attribute 'isotropic' is False.

Parameters: **hkl1,2: Miller indices to be checked for equivalence**

crystalsystem: symmetry class of the material which is considered

Returns: True or False

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: exact position of center bin, in degrees**

twotheta_approx_window_fullwidth_deg: approximate desired width

twotheta_exact_bin_spacing_deg: the exact bin spacing to use

set_parameters **(**`convolver='global'`**,** `**kwargs`**)**
update the dictionary of parameters associated with the given convolver

Parameters: **convolver: the name of the convolver. name 'global', e.g., attaches**

to function 'conv_global'

kwargs: 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: the center position of the middle bin of**

the window, in degrees

twotheta_window_fullwidth_deg: the full width of the window, in degrees

twotheta_output_points: the number of bins in the final output

str_emission ()
format the emission spectrum and crystal size information

Returns: the formatted information

str_global ()
returns a string representation for the global context.

Returns: report on global parameters.

class xrayutilities.simpack.powder.**PowderDiffraction** (mat, **kwargs)
Bases: **xrayutilities.experiment.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 used 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: two theta values at which the powder pattern should be**

calculated.

Note

Note: Bragg peaks are only included up to tt_cutoff set in
the class constructor!

****kwargs: additional keyword arguments are passed to the Convolve**
function

Returns: output intensity values for the twotheta values given in the input

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: two theta values at which the powder pattern should be**

calculated.

window_width: width of the calculation window of a single peak

mode: 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

correction_factor (ang)
calculate the correction factor for the diffracted intensities. This contains the polarization effects and the Lorentz factor

Parameters: **ang: theta diffraction angles for which the correction should be**
calculated

Returns:
f: array of the same shape as ang containing the correction factors

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 if isotropic lines at the same q-position can be merged to one line to reduce the calculational effort

Parameters: **data: numpy field array with values of 'hkl' (Miller indices of the**
peaks), 'q' (q-position), and 'r' (reflection strength) as produced by the structure_factors method

Returns: hkl, q, ang, r: Miller indices, q-position, diffraction angle (Theta),
and reflection strength of the material

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

structure_factors (tt_cutoff)
determine structure factors/reflection strength of all Bragg peaks up to tt_cutoff

Parameters: **tt_cutoff: upper cutoff value of 2theta until which the reflection**
strength are calculated

Returns: numpy array with field for 'hkl' (Miller indices of the peaks),
'q' (q-position), and 'r' (reflection strength) of the Bragg peaks

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={})
update settings of all instances of FP_profile

Parameters: **newsettings: dictionary with new settings. It has to include one**

subdictionary for every convolver which should have its settings changed.

wavelength

window_width

xrayutilities.simpack.powder.**chunkify**(lst, n)

class xrayutilities.simpack.powder.**convolver_handler**
Bases: **object**
manage the convolvers of on process

add_convolver (convolver)

calc (run, ttpeaks)
calculate profile function for selected convolvers

Parameters: **run: list of flags of length of convolvers to tell which convolver**

needs to be run

ttpeaks: peak positions for the convolvers

Returns: 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')
Bases: **multiprocessing.managers.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

add_symbol (**kwargs)
add new symbols to both the attributes and dictionary for the class

Parameters: **kwargs 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.

create_fitparameters ()
function to create a fit model with all instrument and sample parameters.

Parameters: **pass**

Returns: lmfit Parameters instance

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: angular values for the fit

data: experimental intensities for the fit

std: standard deviation of the experimental data. if 'None" the sqrt of the data will be used

maxfev: maximal number of simulations during the least squares refinement

Returns: lmfit MinimizerResult

set_background (`btype`**,** `**kwargs`**)**
define background as spline or polynomial function

Parameters: **btype: background type: either 'polynomial' or 'spline'. Depending on**

this value the expected keyword arguments differ.

kwargs: **'spline':**

x: x-values (twotheta) of the background points

y: intensity values of the background

'polynomial':

p: polynomial coefficients from the highest degree to the constant term. len of p decides about the degree of the polynomial

set_lmfit_parameters (`lmparams`**)**
function to update the settings of this class during an least squares fit

Parameters: **lmparams: lmfit Parameters list of sample and instrument parameters**

set_parameters (`params`**)**
set simulation parameters of all subobjects

Parameters: **params: 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: positions at which the powder spectrum should be evaluated**

****kwargs:**

background: 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: summed powder diffraction intensity of all materials present in the model

Known issue: possibility to add a background is currently missing!

`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: experimental datapoints**

sim: simulated data

weight: weight factor in the least squares sum. If it is None the weight is estimated from the counting statistics of 'exp'

std: standard deviation of the experimental data. alternative way of specifying the weight factor. when both are given weight overwrites std!

Nvar: number of variables in the refinement

disp: flag to tell if a line with the calculated values should be printed.

Returns: M, Rp, Rwp, Rwpexp, chi2

`xrayutilities.simpack.powdermodel.`**`plot_powder`**(`twotheta`, `exp`, `sim`, `mask=None`, `scale='sqrt'`, `fig='XU:powder'`, `show_diff=True`, `show_legend=True`)

Convenience function to plot the comparison between experimental and simulated powder diffraction data

Parameters: **twotheta: angle values used for the x-axis of the plot (deg)**

exp: experimental data (same shape as twotheta). If None only the simulation and no difference will be plotted

sim: simulated data

mask: mask to reduce the twotheta values to the be used as x-coordinates of sim

scale: string specifying the scale of the y-axis. Valid are: 'linear, 'sqrt', and 'log'.

fig: matplotlib figure name (figure will be cleared!)

show_diff: flag to specify if a difference curve should be shown

show_legend: flag to specify if a legend should be shown

xrayutilities.simpack.smaterials module

class `xrayutilities.simpack.smaterials.`**`CrystalStack`**(`name`, `*args`)

Bases: **`xrayutilities.simpack.smaterials.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: **`xrayutilities.simpack.smaterials.CrystalStack`**

generates a sequence of layers with a gradient in chemical composition

class `xrayutilities.simpack.smaterials.`**`Layer`**(`material`, `thickness`, `**kwargs`)

Bases: **`xrayutilities.simpack.smaterials.SMaterial`**

Object describing part of a thin film sample. The properties of a layer :are:

Material: an xrayutilties material describing optical and crystal properties of the thin film

Thickness: film thickness in Angstrom

Roughness: root mean square roughness of the top interface in Angstrom

class xrayutilities.simpack.smaterials.**LayerStack**(name, *args)
Bases: **xrayutilities.simpack.smaterials.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: **_abcoll.MutableSequence**
class representing the basics of a list of materials for simulations within xrayutilities. It extends the built in list type.

check(v)

insert(i, v)

class xrayutilities.simpack.smaterials.**Powder**(material, volume, **kwargs)
Bases: **xrayutilities.simpack.smaterials.SMaterial**
Object describing part of a powder sample. The properties of a powder :are:

Material: an xrayutilties material (Crystal) describing optical and crystal properties of the thin film
Volume: powder's volume (in pseudo units, since only the relative volume enters the calculation)

Optionally also the following can be set:

crystallite_size_lor: Lorentzian crystallite size fwhm (m)
crystallite_size_gauss: Gaussian crystallite size fwhm (m)
strain_lor: extra peak width proportional to tan(theta)
strain_gauss: extra peak width proportional to tan(theta)

class xrayutilities.simpack.smaterials.**PowderList**(name, *args)
Bases: **xrayutilities.simpack.smaterials.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: **xrayutilities.simpack.smaterials.CrystalStack**
generate a sequence of pseudomorphic crystalline Layers. Surface orientation is assumed to be 001 and materials must be cubic/tetragonal.

insert(i, v)

make_epitaxial(i)

trans = <*xrayutilities.math.transforms.Transform object*>

class xrayutilities.simpack.smaterials.**PseudomorphicStack111**(name, *args)
Bases: **xrayutilities.simpack.smaterials.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, **kwargs)
Bases: **object**
Simulation Material. Extends the xrayutilities Materials by properties needed for simulations

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`**(`msg`)

Bases: **`exceptions.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)

Attibutes

expr -- input expression in which the error occurred :msg: -- explanation of the error

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, Powder

`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: ****kwargs: optional keyword arguments**

B: reciprocal space conversion matrix of a Crystal. You can specify the matrix B (default identiy matrix) shape needs to be (3,3)

mat: 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: orientation matrix U can be given. If none is given the orientation defined in the Experiment class is used.

dettype: detector type: one of ('point', 'linear', 'area') decides which routine of Ang2Q to call. default 'point'

delta: 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: x-ray wavelength in angstroem (default: self._wl)

en: x-ray energy in eV (default: converted self._wl)

deg: flag to tell if angles are passed as degree (default: True)

sampledis: sample displacement vector in relative units of the detector distance (default: (0, 0, 0))

Returns: H K L coordinates as numpy.ndarray with shape (* , 3)

where * 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

energy

wavelength

class xrayutilities.experiment.**GID** (idir, ndir, **keyargs)

Bases: **xrayutilities.experiment.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 self.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: 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: optional keyword arguments**

delta: 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 ai,phi,tt,beta - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * corresponds to the number of points given in the input

Q2Ang (`Q`**,** `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)

Parameters: **Q: a list or numpy array of shape (3) with**

q-space vector components

optional keyword arguments:

trans: True/False apply coordinate transformation on Q

deg: True/Flase (default True) determines if the angles are returned in radians or degrees

Returns: a numpy array of shape (4) with the 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)

class `xrayutilities.experiment.`**`GISAXS`** (`idir`, `ndir`, `**keyargs`)

Bases: **`xrayutilities.experiment.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: 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: optional keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of length 3. Used angles are than ai,tt,beta - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * corresponds to the number of points given in the input

Q2Ang (`Q`**,** `trans=True`**,** `deg=True`**,** `**kwargs`**)**

class `xrayutilities.experiment.`**`HXRD`**(`idir`, `ndir`, `geometry='hi_lo'`, `**keyargs`)

Bases: **`xrayutilities.experiment.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 self.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: 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: optional keyword arguments**

delta: 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: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * 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.

Parameters: **Q: a list, tuple or numpy array of shape (3) with**

q-space vector components or 3 separate lists with qx,qy,qz

optional keyword arguments:

trans: True/False apply coordinate transformation on Q (default True)

deg: True/Flase (default True) determines if the angles are returned in radians or degrees

geometry: determines the scattering 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]

default: self.geometry

refrac: boolean to determine if refraction is taken into account :default: False if True then also a material must be given

mat: Crystal object; needed to obtain its optical properties for refraction correction, otherwise not used

full_output: boolean to determine if additional output is given to determine scattering angles more accurately in case refraction is set to True. default: False

fi,fd: 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: 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

if full_output:

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

class `xrayutilities.experiment.`**`NonCOP`**(`idir`, `ndir`, `**keyargs`)

Bases: **`xrayutilities.experiment.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,twotheta) goniometer to help with x-ray diffraction experiments. Linear and area detectors can be treated as described in "help self.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: 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: optional keyword arguments**

delta: 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: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * 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.

Parameters: **Q: a list, tuple or numpy array of shape (3) with**

q-space vector components or 3 separate lists with qx,qy,qz

optional keyword arguments:

trans: True/False apply coordinate transformation on Q (default True)

deg: True/Flase (default True) determines if the angles are returned in radians or degree

Returns: a numpy array of shape (4) with four scattering angles which are

[omega, chi, phi, twotheta]

omega: sample rocking angle

chi: sample tilt

phi: sample azimuth

twotheta: scattering angle (detector)

class xrayutilities.experiment.**PowderExperiment** (**kwargs)
Bases: **xrayutilities.experiment.Experiment**
Experimental class for powder diffraction which helps to convert theta angles to momentum transfer space

Q2Ang (qpos**,** deg=True**)**
Converts reciprocal space values to theta angles

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

UB

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: sample and detector angles as numpy array, lists or**

Scalars 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: possible keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of len(*args). Used angles are than *args - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample. Used to determine not Q but (hkl) (default: self.UB)

roi: region of interest for the detector pixels; e.g. [100, 900, 200, 800] (default: self._area_roi)

Nav: number of channels to average to reduce data size e.g. [2, 2] (default: self._area_nav)

wl: x-ray wavelength in angstroem (default: self._wl)

en: 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: flag to tell if angles are passed as degree (default: True)

sampledis: sample displacement vector in same units as the detector distance (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

detectorAxis
property handler for _detectorAxis

Returns: list of detector axis following the syntax /[xyz][+-]/

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: detector angles. Only detector arm angles as described by the**

detectorAxis attribute must be given.

****kwargs: optional keyword arguments**

- **dim:** dimension of the detector for which the position should be determined
- **roi:** region of interest for the detector pixels; :(default: self._area_roi/self._linear_roi)
- **Nav:** number of channels to average to reduce data size; :(default: self._area_nav/self._linear_nav)
- **deg:** flag to tell if angles are passed as degree (default: True)

Returns: numpy array with the detector distance

getDetectorPos(*args*, ***kwargs*)
obtains the detector position vector by applying the detector arm rotations.

Parameters: ***args: detector angles. Only detector arm angles as described by the**

detectorAxis attribute must be given.

****kwargs: optional keyword arguments**

- **dim:** dimension of the detector for which the position should be determined
- **roi:** region of interest for the detector pixels; :(default: self._area_roi/self._linear_roi)
- **Nav:** number of channels to average to reduce data size; :(default: self._area_nav/self._linear_nav)
- **deg:** flag to tell if angles are passed as degree (default: True)

Returns: 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: direction of the detector (along the pixel**

direction 1); e.g. 'z+' means higher pixel numbers at larger z positions

- **detectorDir2:** direction of the detector (along the pixel direction 2); e.g. 'x+'
- **cch1,2:** center pixel, in direction of self.r_i at zero detectorAngles
- **Nch1:** number of detector pixels along direction 1
- **Nch2:** number of detector pixels along direction 2
- **distance:** distance of center pixel from center of rotation
- **pwidth1,2:** width of one pixel (same unit as distance)
- **chpdeg1,2:** channels per degree (only absolute value is relevant) sign determined through detectorDir1,2
- **detrot:** angle of the detector rotation around primary beam direction (used to correct misalignments)
- **tiltazimuth:** direction of the tilt vector in the detector plane (in degree)
- **tilt:** tilt of the detector plane around an axis normal to the direction given by the tiltazimuth

Note

Note: Either distance and pwidth1,2 or chpdeg1,2 must be given !!

Note

Note: the channel numbers run from 0 .. NchX-1
****kwargs: optional keyword arguments**

Nav: number of channels to average to reduce data size :(default: [1, 1])
roi: region of interest for the detector pixels; e.g. [100, 900, 200, 800]

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: direction of the detector (along the pixel array);**

e.g. 'z+'

cch: center channel, in direction of self.r_i at zero detectorAngles
Nchannel: total number of detector channels
distance: distance of center channel from center of rotation
pixelwidth: width of one pixel (same unit as distance)
chpdeg: channels per degree (only absolute value is relevant) sign determined through detectorDir

!! Either distance and pixelwidth or chpdeg must be given !!

tilt: tilt of the detector axis from the detectorDir (in degree)

Note

Note: the channel numbers run from 0 .. Nchannel-1
****kwargs: optional keyword arguments**

Nav: number of channels to average to reduce data size :(default: 1)
roi: region of interest for the detector pixels; e.g. [100,900]

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: sample and detector angles as numpy array, lists or**

Scalars 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: possible keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of len(*args) used angles are than *args - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

Nav: number of channels to average to reduce data size :(default: self._linear_nav)

roi: region of interest for the detector pixels; e.g. [100,900] (default: self._linear_roi)

wl: x-ray wavelength in angstroem (default: self._wl)

en: 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: flag to tell if angles are passed as degree (default: True)

sampledis: sample displacement vector in same units as the detector distance (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: sample and detector angles as numpy array, lists**

or Scalars 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: optional keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of len(*args) used angles are than *args - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

wl: x-ray wavelength in angstroem (default: self._wl)

en: 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: flag to tell if angles are passed as degree :(default: True)

sampledis: sample displacement vector in relative units of the detector distance (default: (0,0,0))

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * corresponds to the number of points given in the input

sampleAxis

property handler for _sampleAxis

Returns: list of sample axis 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: vector to transform (sequence, list, numpy array)**

args: 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: rotated vector as numpy.array

wavelength

xrayutilities.gridder module

class xrayutilities.gridder.**FuzzyGridder1D**(nx)

Bases: **xrayutilities.gridder.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).

class xrayutilities.gridder.**Gridder**
Bases: **abc.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

data
return gridded data (performs normalization if switched on)

class xrayutilities.gridder.**Gridder1D** (nx)
Bases: **xrayutilities.gridder.Gridder**

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: minimum value of the gridding range**

max: maximum value of the gridding range
fixed: 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: output filename**

header: optional header for the data file.

xaxis
Returns the xaxis of the gridder the returned values correspond to the center of the data bins used by the gridding algorithm

xrayutilities.gridder.**axis** (min_value, max_value, n)
Compute the a grid axis.

Parameters: **min_value ..... axis minimum value**

max_value ..... axis maximum value

n ..... 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: **xrayutilities.gridder.Gridder1D**

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: **xrayutilities.gridder2d.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.

class xrayutilities.gridder2d.**Gridder2D**(nx, ny)
Bases: **xrayutilities.gridder.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 ..... number of points in x-direction**

ny ..... number of points in y-direction

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: minimum value of the gridding range in x,y**

xmax,ymax: maximum value of the gridding range in x,y

fixed: 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: output filename**

header: optional header for the data file.

xaxis

xmatrix

yaxis

ymatrix

class xrayutilities.gridder2d.**Gridder2DList**(nx, ny)
Bases: **xrayutilities.gridder2d.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()

data
return gridded data, in this special version no normalization is defined!

xrayutilities.gridder3d module

class xrayutilities.gridder3d.**FuzzyGridder3D**(nx, ny, nz)
Bases: **xrayutilities.gridder3d.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.

class xrayutilities.gridder3d.**Gridder3D**(nx, ny, nz)
Bases: **xrayutilities.gridder.Gridder**

SetResolution(nx, ny, nz)

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: minimum value of the gridding range in x,y,z**

xmax,ymax,zmax: maximum value of the gridding range in x,y,z

fixed: flag to turn fixed range gridding on (True (default)) or off (False)

xaxis

xmatrix

yaxis

ymatrix

zaxis

zmatrix

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

absfun
absfun property handler
returns the costum correction function or None

avmon
av_mon property handler
returns the value of the average monitor or None if average is calculated from the monitor field

darkfield
flatfield property handler
returns the current set darkfield of the detector or None if not set

det
det property handler
returns the detector field name

flatfield
flatfield property handler
returns the current set flatfield of the detector or None if not set

mon
mon property handler
returns the monitor field name or None if not set

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: data which should be contracted (length N)**

Nav: number of values which should be averaged

Returns: 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: array of 2D data shape (N,M)**

Nav1,2: a field of (Nav1 x Nav2) values is contracted

****kwargs: optional keyword argument**

roi: region of interest for the 2D array. e.g. [20,980,40,960] N = 980-20; M = 960-40

Returns: block averaged numpy array with type numpy.double with shape

(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: array of 2D data shape (Nspectra,Nchannels)**

Nav: number of channels which should be averaged

****kwargs: optional keyword argument**

roi: region of interest for the 2D array. e.g. [20,980] Nchannels = 980-20

Returns: 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=())`

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: q-vector for which the angular positions should be calculated**

expclass: experimental class used to define the goniometer for which the angles should be calculated.

keyword arguments(optional):

bounds: list of 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: orientation matrix of the sample to be used in the conversion

startvalues: 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: sequence of constraint dictionaries. This allows applying arbitrary (e.g. pseudo-angle) contraints by supplying according constraint functions. (see scipy.optimize.minimize). The supplied function will be called with the arguments (angles, qvec, Experiment, U).

Returns: fittedangles, qerror, errcode:

list of fitted goniometer angles, the error in reciprocal space and the errcode of the scipy minimize function. for a successful fit the error code should be <=2

`xrayutilities.q2ang_fit.`**`exitAngleConst`**`(angles, alphaf, hxrd)`

helper function for an pseudo-angle constraint for the Q2AngFit-routine.

Parameters: **angles: fit parameters of Q2AngFit**

alphaf: the exit angle which should be fixed
hxrd: 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.`**`import_lmfit`**`(funcname='XU')`
lazy import function for lmfit

`xrayutilities.utilities.`**`import_matplotlib_pyplot`**`(funcname='XU')`
lazy import function of matplotlib.pyplot

Parameters: **funcname: identification string of the calling function**
Returns: flag, pyplot: the flag is True if the loading was successful and False otherwise. On success pyplot is the matplotlib.pyplot package.

`xrayutilities.utilities.`**`maplog`**`(inte, dynlow='config', dynhigh='config', **keyargs)`
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** : numpy.array, values to be cut in range

dynlow: 10^(-dynlow) will be the minimum cut off
dynhigh: 10^(-dynhigh) will be the maximum cut off

optional keyword arguments (NOT IMPLEMENTED):

abslow: 10^(abslow) will be taken as lower boundary
abshigh: 10^(abshigh) will be taken as higher boundary

Returns: numpy.array of the same shape as inte, where values smaller/larger then 10^(-dynlow,dynhigh) were replaced by 10^(-dynlow,dynhigh)

Examples

```
>>> lint = maplog(int,5,2)
```

xrayutilities.utilities_noconf module

xrayutilities utilities contains a conglomeration of useful functions this part of utilities does not need the config class

`xrayutilities.utilities_noconf.`**`clear_bit`**`(f, offset)`
clears the bet at an offset

`xrayutilities.utilities_noconf.`**`en2lam`**`(inp)`
converts the input energy in eV to a wavelength in Angstrom

Parameters: **inp** : 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: energy either as scalar or array with value in eV, which**

will be returned unchanged; or string with name of emission line

Returns: energy in eV as float

`xrayutilities.utilities_noconf.`**`exchange_filepath`**(`orig`, `new`, `keep=0`)

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)

Parameters: **orig: original filename which should have its data root replaced**

new: new path which should be used instead

keep: (optional) number of inner most directory names which should be kept the same in the output (default = 0). Note that the filename is always return unchanged also with keep=0.

Returns: filename string

Examples

```
>>> exchange_filepath('/dir_a/subdir/sam/file.txt', '/data', 1)
'/data/sam/file.txt'
```

`xrayutilities.utilities_noconf.`**`exchange_path`**(`orig`, `new`, `keep=0`)

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)

Parameters: **orig: original path which should be replaced by the new path**

new: new path which should be used instead

keep: (optional) number of inner most directory names which should be kept the same in the output (default = 0)

Returns: directory path string

Examples

```
>>> exchange_path('/dir_a/subdir/img/sam', '/home/user/data', 2)
'/home/user/data/img/sam'
```

`xrayutilities.utilities_noconf.`**`lam2en`**(`inp`)

converts the input wavelength in Angstrom to an energy in eV

Parameters: **inp** : wavelength in Angstrom

Returns: float, energy in eV

Examples

```
>>> energy = lam2en(1.5406)
```

`xrayutilities.utilities_noconf.`**`set_bit`**(`f`, `offset`)

sets the bit at an offset

`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: wavelength; If scalar or array the wavelength in Angstrom will be**

returned unchanged, string with emission name is converted to wavelength

Returns: wavelength in Angstrom as float

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>

xrayutilities.analysis package

Submodules

xrayutilities.analysis.line_cuts module

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_ang`** (`qx`, `qz`, `intensity`, `omcenter`, `ttcenter`, `omrange`, `npoints`, `**kwargs`)

extracts an omega scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: omega-position at which the omega scan should be extracted

ttcenter: 2theta-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative omega positions are returned :(default: True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,omint: omega scan coordinates and intensities (bounds=False)

om,omint,(qxb, qzb): omega scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omcut, intcut = get_omega_scan(qx,qz,intensity,0.0,5.0,2.0,200)
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_bounds_ang`** (`omcenter`, `ttcenter`, `omrange`, `npoints`, `**kwargs`)

return reciprocal space boundaries of omega scan

Parameters: **omcenter: omega-position at which the omega scan should be extracted**

ttcenter: 2theta-position at which the omega scan should be extracted
omrange: range of the omega scan to extract
npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction
lam: wavelength for use in the conversion to angular coordinates

Returns:

qx,qz: reciprocal space coordinates of the omega scan boundaries

Examples

```
>>> qxb,qzb = get_omega_scan_bounds_ang(1.0,4.0,2.4,240,qrange=0.1)
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_q`**(qx, qz, intensity, qxcenter, qzcenter, omrange, npoints, **kwargs)

extracts an omega scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer
intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)
qxcenter: qx-position at which the omega scan should be extracted
qzcenter: qz-position at which the omega scan should be extracted
omrange: range of the omega scan to extract
npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction
Nint: number of subscans used for the integration (optionally)
lam: wavelength for use in the conversion to angular coordinates
relative: determines if absolute or relative omega positions are returned :(default: True)
bounds: flag to specify if the scan bounds should be returned; :(default: False)

Returns:

om,omint: omega scan coordinates and intensities (bounds=False)

om,omint,(qxb, qzb): omega scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omcut, intcut = get_omega_scan(qx,qz,intensity,0.0,5.0,2.0,200)
```

`xrayutilities.analysis.line_cuts.`**`get_qx_scan`**(qx, qz, intensity, qzpos, **kwargs)

extract qx line scan at position qzpos from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given range along qz

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

bounds: flag to specify if the scan bounds of the extracted scan should be returned (default:False)

Returns:

qx,qxint: qx scan coordinates and intensities (bounds=False)

qx,qxint,(qxb,qyb): qx scan coordinates and intensities + scan bounds for plotting

Examples

```
>>> qxcut,qxcut_int = get_qx_scan(qx,qz,inten,5.0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts.`**`get_qz_scan`**(`qx`, `qz`, `intensity`, `qxpos`, `**kwargs`)

extract qz line scan at position qxpos from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given range along qx

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qz,qzint: qz scan coordinates and intensities

Examples

```
>>> qzcut,qzcut_int = get_qz_scan(qx,qz,inten,1.5,qrange=0.03)
```

`xrayutilities.analysis.line_cuts.`**`get_qz_scan_int`**(`qx`, `qz`, `intensity`, `qxpos`, `**kwargs`)

extracts a qz scan from a gridded reciprocal space map with integration along omega (sample rocking angle) or 2theta direction

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

angrange: integration range in angular direction

qmin,qmax: minimum and maximum value of extracted scan axis

bounds: flag to specify if the scan bounds of the extracted scan should be returned (default:False)

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

wl: wavelength used to determine angular integration positions

Returns:

qz,qzint: qz scan coordinates and intensities (bounds=False)

qz,qzint,(qzb,qzb): qz scan coordinates and intensities + scan bounds for plotting

Examples

```
>>> qzcut,qzcut_int = get_qz_scan_int(qx,qz,inten,5.0,omrange=0.3)
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan_ang`**(qx, qz, intensity, omcenter, ttcenter, ttrange, npoints, **kwargs)

extracts a radial scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: om-position at which the radial scan should be extracted

ttcenter: tt-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,tt,radint: omega,two theta scan coordinates and intensities

(bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omc, ttc, cut_int = get_radial_scan_ang(qx, qz, intensity, 32.0, 64.0,
                                            30.0, 800, omrange = 0.2)
```

xrayutilities.analysis.line_cuts.**get_radial_scan_bounds_ang**(omcenter, ttcenter, ttrange, npoints, **kwargs)

return reciprocal space boundaries of radial scan

Parameters: **omcenter: om-position at which the radial scan should be extracted**

ttcenter: tt-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qxrad,qzrad: reciprocal space boundaries of radial scan

Examples

```
>>>
```

xrayutilities.analysis.line_cuts.**get_radial_scan_q**(qx, qz, intensity, qxcenter, qzcenter, ttrange, npoints, **kwargs)

extracts a radial scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the radial scan should be extracted

qzcenter: qz-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,tt,radint: omega,two theta scan coordinates and intensities

(bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omc, ttc, cut_int = get_radial_scan_q(qx, qz, intensity, 0.0, 5.0,
                                          1.0, 100, omrange = 0.01)
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_ang`**(qx, qz, intensity, omcenter, ttcenter, ttrange, npoints, **kwargs)

extracts a twotheta scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: om-position at which the 2theta scan should be extracted

ttcenter: tt-position at which the 2theta scan should be extracted

ttrange: two theta range of the scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

tt,ttint: two theta scan coordinates and intensities (bounds=False)

tt,ttint,(qxb,qzb): 2theta scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> ttc,cut_int = get_ttheta_scan_ang(qx,qz,intensity,32.0,64.0,4.0,400)
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_bounds_ang`**(omcenter, ttcenter, ttrange, npoints, **kwargs)

return reciprocal space boundaries of 2theta scan

Parameters: **omcenter: om-position at which the 2theta scan should be extracted**

ttcenter: tt-position at which the 2theta scan should be extracted

ttrange: two theta range of the 2theta scan to extract

npoints: number of points of the 2theta scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qxtt,qztt: reciprocal space boundaries of 2theta scan (bounds=False)

tt,ttint,(qxb,qzb): 2theta scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>>
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_q`** (qx, qz, intensity, qxcenter, qzcenter, ttrange, npoints, **kwargs)
extracts a twotheta scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer
intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)
qxcenter: qx-position at which the 2theta scan should be extracted
qzcenter: qz-position at which the 2theta scan should be extracted
ttrange: two theta range of the scan to extract
npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction
Nint: number of subscans used for the integration (optionally)
lam: wavelength for use in the conversion to angular coordinates
relative: determines if absolute or relative two theta positions are returned (default=True)
bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

tt,ttint: two theta scan coordinates and intensities (bounds=False)
om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> ttc,cut_int = get_ttheta_scan_q(qx,qz,intensity,0.0,4.0,4.4,440)
```

`xrayutilities.analysis.line_cuts.`**`getindex`** (x, y, xgrid, ygrid)
gives the indices of the point x,y in the grid given by xgrid ygrid xgrid,ygrid must be arrays containing equidistant points

Parameters: **x,y: coordinates of the point of interest (float)**

xgrid,ygrid: grid coordinates in x and y direction (array)

Returns:

ix,iy: index of the closest gridpoint (lower left) of the point (x,y)

xrayutilities.analysis.line_cuts3d module

`xrayutilities.analysis.line_cuts3d.`**`get_qx_scan3d`** (gridder, qypos, qzpos, **kwargs)
extract qx line scan at position y,z from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qypos,qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction
qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qx,qxint: qx scan coordinates and intensities

Examples

```
>>> qxcut,qxcut_int = get_qx_scan3d(gridder,0,0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts3d.`**`get_qy_scan3d`** (`gridder`, `qxpos`, `qzpos`, `**kwargs`)
extract qy line scan at position x,z from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qxpos,qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qy,qyint: qy scan coordinates and intensities

Examples

```
>>> qycut,qycut_int = get_qy_scan3d(gridder,0,0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts3d.`**`get_qz_scan3d`** (`gridder`, `qxpos`, `qypos`, `**kwargs`)
extract qz line scan at position x,y from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qxpos,qypos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qz,qzint: qz scan coordinates and intensities

Examples

```
>>> qzcut,qzcut_int = get_qz_scan3d(gridder,0,0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts3d.`**`getindex3d`** (`x`, `y`, `z`, `xgrid`, `ygrid`, `zgrid`)
gives the indices of the point x,y,z in the grid given by xgrid ygrid zgrid xgrid,ygrid,zgrid must be arrays containing equidistant points

Parameters: **x, y, z: coordinates of the point of interest (float)**

xgrid, ygrid, zgrid: grid coordinates in x, y, z direction (array)

Returns: ix, iy, iz: index of the closest gridpoint (lower left) of the point

(x, y, z)

xrayutilities.analysis.misc module

miscellaneous functions helpful in the analysis and experiment

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

Parameters: **peak: array which gives hkl for the peak of interest**

sur: hkl of the surface

inp: inplane reference peak or direction

Returns: [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.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 ..... outer detector arm angle**

angle2: inner detector arm angle

ccdimages: images of the ccd taken at the angles given above

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

r_i: primary beam direction [xyz][+-] default 'x+'

keyword_arguments:

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

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

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

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

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

nwindow: 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: 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, '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 .. 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: outer detector arm angle

angle2: inner detector arm angle

ccdimages: images of the ccd taken at the angles given above

hkls: array/list of hkl values for every image

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

material: material used as reference crystal

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

r_i: primary beam direction [xyz][+-] default 'x+'

keyword_arguments:

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

cut_off: cut off intensity to decide if image is used for the determination or not. default: 0.1 = 10%

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

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

nwindow: 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: flag to tell if you want to see debug output of the script (switch this to true only if you can handle it :))

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: angular coordinates of the measurement (numpy.ndarray)**

either with size of psd or of psd.shape[0]

psd: intensity values needed for fitting

omalign: aligned omega value, used as first guess in the fit

ttalign: 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 class used for the conversion between angular and reciprocal space.

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

peaktype: can be 'Gauss' or 'Lorentz' to fit either of the two peak shapes

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

Returns:

omfit,ttfit,params,covariance: fitted angular values, and the fit parameters (of the Gaussian/Lorentzian) as well as their errors

`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 of angles in degree of measured detector spectra**

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

keyword arguments:

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

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

other options are passed to psd_chdeg function, options include:

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

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

Note: see help of psd_chdeg for more options

Returns: pixelwidth (at one meter distance), center_channel[, detector_tilt]

Note

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

distance L
pixelwidth is negative in case the hit channel number decreases upon an increase of the detector angle The function also prints out how a linear detector can be initialized using the results obtained from this calibration. Carefully check the results

`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: azimuths in which the reflection was aligned (deg)**

aomega: aligned omega values (deg)

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

omega0: 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: flag to specify if a visualization of the fit is wanted. :default: True

Returns: [omega0,phi0,miscut]

list with fitted values for

omega0: the omega value of the reflection should be close to the nominal one

phi0: the azimuth in which the primary beam looks upstairs

miscut: amplitude of the sinusoidal variation == miscut angle

`xrayutilities.analysis.sample_align.`**psd_chdeg** (`angles`, `channels`, `stdev=None`, `usetilt=True`, `plot=True`, `datap='kx'`, `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: detector angles for which the position of the beam was**

measured

channels: detector channels where the beam was found

keyword arguments:

stdev: standard deviation of the beam position

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

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

datap: plot format of data points

modelline: plot format of modelline

modeltilt: plot format of modeltilt

fignum: figure number to use for the plot

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

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

dlabel: label of the data line to be used in the plot

figtitle: boolean to tell if the figure title should show the fit parameters

Returns: (pixelwidth,centerch,tilt)

pixelwidth: 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: center channel of the detector

tilt: tilt of the detector from perpendicular to the beam

(will be zero in case of usetilt=False)

Note

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

`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** : primary beam channel number

angles: list or numpy.array with angles

channels: list or numpy.array with corresponding detector channels

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

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

Examples

```
>>> psd_refl_align(500,[0,0.1,0.2,0.3],[550,600,640,700])
```

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: **`xrayutilities.io.filedir.FileDirectory`**

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

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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

xrayutilities.io.cbf.**makeNaturalName**(name)

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.**gettty08_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: name of the scans, for multiple scans this needs to be a**

template string

scannumbers: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent)

**keyargs: keyword arguments are passed on to tty08File

Returns: MAP

or

[ang1,ang2,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with 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.
Required constructor arguments:

filename: a string with the name of the tty08-file

Optional keyword arguments:

mcadir: directory name of MCA files

Read ()

Read the data from the file

ReadMCA ()

xrayutilities.io.edf module

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

Bases: **xrayutilities.io.filedir.FileDirectory**

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

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: 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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

data

xrayutilities.io.edf.**makeNaturalName** (name)

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.

grid2D (nx**,** ny**,** **kwargs**)**

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

Parameters: **nx,ny: number of bins in x,y direction**

optional keyword arguments:

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

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

Returns: 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: name of the motor for which the position is wanted**

Returns:
val: 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** (filename, scannr, xmotor='adcX', ymotor='adcY', path='')
Bases: **xrayutilities.io.fastscan.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

getccdFileTemplate (specscan, datadir=None, keepdir=0, numfmt='%04d')
function to extract the CCD file template string from the comment in the SPEC-file scan-header

Parameters: **specscan: spec-scan object from which header the CCD directory should**

be extracted

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

numfmt: format string for the CCD file number (optional)

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

gridCCD (nx, ny, ccdnr, roi=None, datadir=None, keepdir=0, nav=[1, 1], gridrange=None, filterfunc=None, imgoffset=0)
function to grid the internal data and ccd files and return the gridded X,Y and DATA values. DATA represents a 4D with first two dimensions representing X,Y and the remaining two dimensions representing detector channels

Parameters: **nx,ny: number of bins in x,y direction**

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

optional:

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

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

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

filterfunc: 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: regular x,y-grid as well as 4-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.

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: list of shifts in x-direction for every FastScan in the**

data structure

deltay: same for the y-direction

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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

optional:

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

Returns: [[motorpos1, ccdnrs1], [motorpos2, ccdnrs2], ...] where motorposN is

from the N-ths FastScan in the series and ccdnrsN is the list of according CCD-frames

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: number of bins in x,y direction**

optional keyword arguments:

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

gridrange: 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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

qnx: number of points in the Qx direction of the gridded reciprocal space map

qny: same for y direction

qnz: same for z directino

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

optional:

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

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

filterfunc: 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: 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, 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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

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

optional:

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

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

filterfunc: 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: sample orientation matrix

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

Returns:

qx,qy,qz,ccddata,valuelist: raw data of the reciprocal space map and 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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

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

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.

`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: filename of the file to open (full including path)**

mode: mode in which the file should be opened

Returns: file handle of the opened file

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.

Required constructor arguments:

filename: a string with the name of the data file

`Read``()`

Read the data from the 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: number of the numors, or list of numbers. This will be**

transformed to a string and used as a filename (int, str, or iterable (list, tuple))

***args: names of the motors (optional) (strings)**

e.g.: 'omega', 'gamma'

****kwargs: keyword arguments are passed on to numorFile. e.g. 'path' for**

the files directory

Returns: data

or

[ang1,ang2,...],data:

angular positions position together with 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=<type '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.

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: 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

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

Bases: **xrayutilities.io.imagereader.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.

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

Bases: **xrayutilities.io.imagereader.ImageReader**

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

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

Bases: **xrayutilities.io.imagereader.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.

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

Bases: **xrayutilities.io.imagereader.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.

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

Parameters: **filename: filename of the image to be read. so far only single**

filenames are supported. The data might be compressed.

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

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

`xrayutilities.io.panalytical_xml.`**`getOmPixcel`**(`omraw`, `ttraw`)
function to reshape the Omega values into a form needed for further treatment with xrayutilities

`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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames

scannrs: int or list of scan numbers

path: common path to the filenames

roi: region of interest for the PIXCel detector, for other measurements this is not usefull!

Returns:

om,tt,psd: 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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames given by the scannrs keyword argument

***motors: motor names to return: e.g.: 'Omega','2Theta',...**

one can also use abbreviations 'Omega' = 'om' = 'o' '2Theta' = 'tt' = 't' 'Chi' = 'c' 'Phi' = 'p'

****kwargs:**

scannrs: int or list of scan numbers

path: common path to the filenames

Returns: scanmot,mot1,mot2,...,detectorint: as flattened numpy arrays

Examples

```
>>> scanmot,om,tt,inte = xrayutilities.io.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

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

Bases: **`xrayutilities.io.pdcif.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

`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

Required constructor arguments:

filename: a string with the name of the ras-file

keyword argument (optional):

path: path to the data file

Read ()
Read the data from the file

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
Required constructor arguments:

filename: file name of the data file

pos: seek position of the RAS_HEADER_START line

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: name of the scans, for multiple scans this needs to be a**

template string

scannumbers: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent) **kwargs: keyword arguments forwarded to RASFile function

Returns: rasdata

or

[ang1,ang2,...],rasdata:

angular positions are extracted from the respective scan header together with all the data values as stored in the data file (includes the intensities e.g. rasdata['int']).

Examples

```
>>> [om,tt],MAP = xu.io.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.

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: 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 use 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

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

parse ()

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

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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames

scannrs: int or list of scan numbers

path: common path to the filenames

scantype: 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: number of channels of the MCA (needed for "tsk" measurements)

Returns:

om,tt,psd: as flattened numpy arrays

Examples

```
>>> om,tt,psd = xrayutilities.io.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**

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={}**)**
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.

required arguments:

h5f: a HDF5 file object or its filename

optional keyword arguments:

comp: 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.

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

Parse ()

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={}**,** 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.

input arguments:

h5f: a HDF5 file object or its filename

optional keyword arguments:

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

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

optattrs: a dictionary with optional attributes to store for the data

comp: activate compression - true by default

SetMCAParams (`mca_column_format`, `mca_channels`, `mca_start`, `mca_stop`)

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

required input aguments:

mca_column_format: number of columns used to save the data

mca_channels: number of MCA channels stored

mca_start: first channel that is stored

mca_stop: last channel that is stored

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: 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:**

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

logy: 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 object of a HDF5 file opened using h5py or its filename**

scans: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent)

****kwargs (optional):**

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

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

Returns: MAP

or

[ang1,ang2,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with all the data values as stored in the data file (includes the intensities e.g. MAP['MCA']).

Examples

```
>>> [om, tt], MAP = xu.io.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 object**

scans: number of the scans of the reciprocal space map (int,tuple or list)

args: names of the motors and counters (strings)

keyword arguments:

rettype: 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,...]:

coordinates and counters from the SPEC file

Examples

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

`xrayutilities.io.spec.`**`makeNaturalName`** (`name`)

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.
Required constructor arguments:

filename: a string with the name of the SPECTRA file

Optional keyword arguments:

mcatmp: template for the MCA files

mcastart,mcastop: 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.

required input arguments:

h5file: string or HDF5 file object
name: name of the group where to store the data

optional keyword arguments:

group: root group where to store the data
mcaname: Name of the MCA in the HDF5 file

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

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.

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

append (`col`**)**

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

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

`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 object of a HDF5 file opened using h5py**

scans: number of the scans of the reciprocal space map (int,tuple or list)

***args: arbitrary number of motor names (strings)**

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

****kwargs (optional):**

mca: name of the mca data (if available) otherwise None :(default: "MCA")
samplename: 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,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with all 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**

f (q, en='config')
function to calculate the atomic structure factor F

Parameters: **q: momentum transfer**

en: energy for which F should be calculated, if omitted the value from the xrayutilities configuration is used

Returns: f (float)

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: flag, result: if the flag is True then result contains the cached result, otherwise result is None

max_cache_length = *10*

set_cache (prop, key, result)
set result to be cached to speed up future calls

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.**CIFFile** (filename, digits=3)
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

Lattice ()
returns a lattice object with the structure from the CIF file

Parse ()
function to parse a CIF file. 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.

xrayutilities.materials.cif.**testwp** (parint, wp, cifpos, digits)
test if a Wyckoff position can describe the given position from a CIF file

Parameters: **parint: integer telling which Parameters the given Wyckoff position has**

wp: expression of the Wyckoff position (string of tuple)
cifpos: (x,y,z) position of the atom in the CIF file
digits: number of digits for which for a comparison of floating point numbers will be rounded to

Returns: foundflag, pars: flag to tell if the positions match and if necessary any parameters associated with the position

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.

required input arguments:

dbname: name of the database
dbdesc: 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.

required input arguments:

name: a string with the name of the material
description: a string with a description of the material

GetF0 (q, dset='default')
Obtain the f0 scattering factor component for a particular momentum transfer q.

required input argument:

q: single float value or numpy array
dset: 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.

required input arguments:

en: float or numpy array with the energy

GetF2 (en)
Return the imaginary part of the scattering factor for a certain energy en.

required input arguments:

en: float or numpy array with the energy

Open (mode='r')
Open an existing database file.

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

required input argument:

parameters: list or numpy array with the fit parameters
subset: specifies under which name the f0 values should be saved

SetF1F2 (en, f1, f2)
Set f1, f2 values for the active material.

required input arguments:

en: list or numpy array with energy in (eV)
f1: list or numpy array with f1 values
f2: list or numpy array with 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.

required input arguments:

name: string with the name of the material

SetWeight (weight)
Save weight of the element as float

required input argument:

weight: atomic standard weight of the element (float)

xrayutilities.materials.database.**add_f0_from_intertab** (db, itf)
Read f0 data from International Tables of Crystallography and add it to the database.

xrayutilities.materials.database.**add_f0_from_xop** (db, xop)
Read f0 data from f0_xop.dat and add it to the database.

xrayutilities.materials.database.**add_f1f2_from_ascii_file** (db, asciifile, element)
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)
Read f1 and f2 data from Henke database and add it to the database.

xrayutilities.materials.database.**add_f1f2_from_kissel** (db, kf)
Read f1 and f2 data from Henke database and add it to the database.

xrayutilities.materials.database.**add_mass_from_NIST** (db, nistfile)
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.**init_material_db** (db)

xrayutilities.materials.elements module

xrayutilities.materials.lattice module

module handling crystal lattice structures. A Lattice consists of unit cell parameters and a LatticeBase. It offers methods to calculate the reciprocal space position of Bragg peaks and their structure factor.

xrayutilities.materials.lattice.**AlGaAsLattice** (aal, aga, aas, a, x)

xrayutilities.materials.lattice.**BCCLattice** (aa, a)

xrayutilities.materials.lattice.**BCTLattice** (aa, a, c)

xrayutilities.materials.lattice.**BaddeleyiteLattice** (aa, ab, a, b, c, beta)

xrayutilities.materials.lattice.**CsClLattice** (aa, ab, a)

xrayutilities.materials.lattice.**CubicFm3mBaF2** (aa, ab, a)

xrayutilities.materials.lattice.**CubicLattice** (a, base=None)

Returns a Lattice object representing a cubic lattice.

Parameters: **a: lattice parameter**

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**DiamondLattice** (aa, a)

xrayutilities.materials.lattice.**FCCLattice** (aa, a)

xrayutilities.materials.lattice.**FCCSharedLattice** (aa, ab, occa, occb, a)

xrayutilities.materials.lattice.**GeTeRhombohedral** (aa, ab, a, ang, x=0.237)

xrayutilities.materials.lattice.**HCPLattice** (aa, a, c)

xrayutilities.materials.lattice.**Hexagonal3CLattice** (aa, ab, a, c)

xrayutilities.materials.lattice.**Hexagonal4HLattice** (aa, ab, a, c, u=0.1875, v1=0.25, v2=0.4375)

xrayutilities.materials.lattice.**Hexagonal6HLattice** (aa, ab, a, c)

xrayutilities.materials.lattice.**HexagonalLattice** (a, c, base=None)

Returns a Lattice object representing a hexagonal lattice.

Parameters: **a: lattice parameter a**

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**LaB6Lattice** (aa, ab, a, oa=1, ob=1, ba=0, bb=0)

class xrayutilities.materials.lattice.**Lattice** (a1, a2, a3, base=None)

Bases: **object**

class Lattice: This object represents a Bravais lattice. A lattice consists of a base and unit cell defined by three vectors.

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:: Note: NO elastic response of the material will be considered!

requiered input arguments:

eps: a 3x3 matrix independent strain components

GetPoint (*args)

determine lattice points with indices given in the argument

Examples

```
>>> 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])
```

ReciprocalLattice ()

UnitCellVolume ()
function to calculate the unit cell volume of a lattice (angstrom^3)

a

a1

a2

a3

alpha

b

beta

c

gamma

class xrayutilities.materials.lattice.**LatticeBase** (*args, **keyargs)
Bases: **list**
The LatticeBase class implements a container for a set of points that form the base of a crystal lattice. An instance of this class can be treated as a simple container object.

append (atom**,** pos**,** occ=1.0**,** b=0.0**)**
add new Atom to the lattice base

Parameters: **atom: atom object to be added**

pos: position of the atom

occ: occupancy (default=1.0)

b: 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)

xrayutilities.materials.lattice.**MagnetiteLattice** (aa, ab, ac, a, x=0.255)

xrayutilities.materials.lattice.**MonoclinicLattice** (a, b, c, beta, base=None)
Returns a Lattice object representing a hexagonal lattice.

Parameters: **a: lattice parameter a**

b: lattice parameter b

c: lattice parameter c

beta: monoclinic unit cell angle beta (deg)

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**NaumanniteLattice** (aa, ab, a, b, c)

xrayutilities.materials.lattice.**NiAsLattice** (aa, ab, a, c, biso=0.0)

xrayutilities.materials.lattice.**OrthorhombicLattice** (a, b, c, base=None)
Returns a Lattice object representing a tetragonal lattice.

Parameters: **a: lattice parameter a**

b: lattice parameter b

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**PerovskiteTypeRhombohedral** (aa, ab, ac, a, ang)

xrayutilities.materials.lattice.**QuartzLattice** (aa, ab, a, b, c)

xrayutilities.materials.lattice.**RockSaltLattice** (aa, ab, a)
creates the primitive unit cell of a RockSalt structure. For the more commonly used cubic reprentation see RockSalt_Cubic_Lattice

xrayutilities.materials.lattice.**RockSalt_Cubic_Lattice** (aa, ab, a)

xrayutilities.materials.lattice.**RutileLattice** (aa, ab, a, c, u)

xrayutilities.materials.lattice.**SiGeLattice** (asi, age, a, xge)

xrayutilities.materials.lattice.**TetragonalIndiumLattice** (aa, a, c)

xrayutilities.materials.lattice.**TetragonalLattice** (a, c, base=None)
Returns a Lattice object representing a tetragonal lattice.

Parameters: **a: lattice parameter a**

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**TetragonalTinLattice** (aa, a, c)

xrayutilities.materials.lattice.**TriclinicLattice** (a, b, c, alpha, beta, gamma, base=None)

xrayutilities.materials.lattice.**TrigonalR3mh** (aa, a, c)

xrayutilities.materials.lattice.**WurtziteLattice** (aa, ab, a, c, u=0.375, biso=0.0)

xrayutilities.materials.lattice.**ZincBlendeLattice** (aa, ab, a)

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: **xrayutilities.materials.material.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** : Miller Indices

sub: substrate material or lattice constant (Instance of Crystal class or float)

exp: Experiment class from which the Transformation object and ndir are needed

Returns: **qy,qz** : reciprocal space coordinates of the corners of the relaxation triangle

static **check_compatibility**(matA, matB)

static **lattice_const_AB**(latA, latB, x, name='')
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: property (lattice parameter/angle) of material A and B.**

A property can be a scalar or vector.

x: fraction of material B in the alloy.

name: label of the property which is interpolated. Can be 'a', 'b', 'c', 'alpha', 'beta', or 'gamma'.

x

class xrayutilities.materials.material.**Amorphous**(name, density, atoms=None, cij=None)
Bases: **xrayutilities.materials.material.Material**
amorphous materials are described by this class

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: x-ray energy eV, if omitted the value from the xrayutilities** configuration is used

Returns: delta (float)

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: x-ray energy eV, if omitted the value from the xrayutilities** configuration is used

Returns: beta (float)

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: string containing the chemical fomula**

Returns: list of tuples with 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.

required input arguments:

cij: (6,6) cij matrix as a numpy array

return value:

cijkl: (3,3,3,3) cijkl 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.

required input arguments:

cijkl: (3,3,3,3) cijkl tensor as numpy array

return value:

cij: (6,6) cij matrix as a numpy array

class `xrayutilities.materials.material.`**`Crystal`**`(name, lat, cij=None, thetaDebye=None)`

Bases: **`xrayutilities.materials.material.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. Note: NO elastic response of the material will be considered!

`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 numpy array with the 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 numpy array with the 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: vectorial momentum transfer (vectors as list,tuple**

or numpy array are valid)

en: energy in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

Returns: 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: vectorial momentum transfer (vectors as list,tuple**

or numpy array are valid)

en: list, tuple or array of energy values in eV

temp: temperature used for Debye-Waller-factor calculation

Returns: 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: 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: energy value in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

Returns: complex valued structure factor array

a

a1

a2

a3

alpha

b

beta

c

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: momentum transfer in (1/A)**

en: xray energy in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

polarization: either 'S' (default) sigma or 'P' pi polarization

Returns: (abs(chih_real),abs(chih_imag)) complex polarizability

dTheta (Q, en='config')
function to calculate the refractive peak shift

Parameters: **Q: momentum transfer (1/A)**

en: x-ray energy (eV), if omitted the value from the xrayutilities configuration is used

Returns:

deltaTheta: 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: x-ray energy eV, if omitted the value from the xrayutilities**

configuration is used

Returns: delta (float)

density
calculates the mass density of an material from the mass of the atoms in the unit cell.

Returns: 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

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 the unit cell.

Parameters: **pos: list or numpy array with the fractional coordinated in the**

unit cell

keyword arguments:

maxdist: maximum distance wanted in the list of neighbors :(default: 7)

Returns: list of tuples with (distance,atomType,multiplicity) giving distance

(sorted) and type of neighboring atoms together with the amount of atoms at the given distance

classmethod **fromCIF** (ciffilename)
Create a Crystal from a CIF file. The CIF-filename will be used as name of the created Crystal. Note: since the CIF file parser is currently not able to detect the correct space group of the material all materials created by this method will be represented by the P1 space-group!

Parameters: **ciffilename: filename of the CIF file**

Returns: Crystal instance

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: x-ray energy eV, if omitted the value from the xrayutilities**

configuration is used

Returns: beta (float)

planeDistance (*hkl)
determines the lattice plane spacing for the planes specified by (hkl)

Parameters: **h,k,l: Miller indices of the lattice planes given either as**

list,tuple or seperate arguments

Returns:

d: the lattice plane spacing as float

Examples

```
>>> xu.materials.Si.planeDistance(0,0,4)
1.3577600000000001
```

or

```
>>> xu.materials.Si.planeDistance((1,1,1))
3.1356124059796255
```

class xrayutilities.materials.material.**CubicAlloy** (matA, matB, x)
Bases: **xrayutilities.materials.material.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** : inplane peak position of reflection hkl of

the alloy in reciprocal space

q_perp: perpendicular peak position of the reflection hkl of the alloy in reciprocal space

hkl: Miller indices of the measured asymmetric reflection

sur: Miller indices of the surface (determines the perpendicular direction)

Returns: content, [a_inplane, a_perp, a_bulk_perp(x), eps_inplane, eps_perp]:

the content of B in the alloy determined from the input variables and the lattice constants 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** : perpendicular peak position of the reflection

hkl of the alloy in reciprocal space

hkl: Miller indices of the measured symmetric reflection (also defines the surface normal

inpr: Miller indices of a Bragg peak defining the inplane reference direction

asub: substrate lattice constant

relax: degree of relaxation (needed to obtain the content from symmetric reciprocal space position)

Returns: **content** : the content of B in the alloy determined from the input

variables

`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: independent components of the elastic tensor of cubic**

materials

Returns: 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: independent components of the elastic tensor of**

a hexagonal material

Returns: 6x6 matrix with elastic constants

class `xrayutilities.materials.material.`**`Material`** (`name`, `cij=None`)

Bases: **`abc.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.

`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: energy of the x-rays (in eV, optional)**

Returns: 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: energy of the x-rays, if omitted the value from the**

xrayutilities configuration is used

deg: return angle in degree if True otherwise radians (default:True)

Returns: Angle of total external reflection

`delta` **(**`en='config'`**)**

abstract method which every implementation of a Material has to override

density

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)

lam

mu

nu

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.

Parameters: **sub: substrate material**

layer: bulk material of the layer

relaxation: degree of relaxation 0: pseudomorphic, 1: relaxed :(default: 0)

trans: 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.

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: independent components of the elastic tensor of cubic**
materials

Returns: 6x6 matrix with elastic constants
Implementation according to a patch submitted by Julian Stangl

xrayutilities.materials.material.**index_map_ij2ijkl**(ij)

xrayutilities.materials.material.**index_map_ijkl2ij**(i, j)

xrayutilities.materials.predefined_materials module

class xrayutilities.materials.predefined_materials.**AlGaAs**(x)
Bases: **xrayutilities.materials.material.CubicAlloy**

class xrayutilities.materials.predefined_materials.**SiGe**(x)
Bases: **xrayutilities.materials.material.CubicAlloy**

static **lattice_const_AB**(latA, latB, x, **kwargs)
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.**RangeDict**
Bases: **dict**

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.
this replaces the deprecated Lattice class

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:: Note: Here you specify the strain and not the stress -> NO elastic

response of the material will be considered!

Parameters: **eps: a 3x3 matrix with all strain components**

GetHKL(*args)
determine the Miller indices of the given reciprocal lattice points

GetPoint(*args)
determine lattice points with indices given in the argument

Examples

```
>>> 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)

a

alpha

b

base()
generator of atomic position within the unit cell.

beta

c

classmethod **convert_to_P1**(sglat)
create a P1 equivalent of the given SGLattice instance.

Parameters: **sglat: space group lattice instance to be converted to P1.**

Returns: SGLattice instance with the same properties as sglat, however in the P1 setting.

classmethod **fromLattice** (lat, verbose=True)
create a SGLattice from an old Lattice instance. Since the space-group is not known it will always be 1 (triclinic). This is helper routine to make the transition period for users easier. It will be removed in the next major release!

Parameters: **lat: deprecated Lattice instance**

Returns: SGLattice instance with the same properties as lat

gamma

isequivalent (hkl1**,** hkl2**,** equalq=False**)**
primitive way of determining if hkl1 and hkl2 are two crystallographical equivalent pairs of Miller indices

Parameters: **hkl1,2: Miller indices to be checked for equivalence**

equalq: If False the length of the two q-vactors will be compared. If True it is assumed that the length of the q-vectors of hkl1 and hkl2 is equal!

Returns: True or False

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.

append (atom**,** pos**,** occ=1.0**,** b=0.0**)**
add new Atom to the lattice base

Parameters: **atom: atom object to be added**

pos: Wyckoff position of the atom, along with its parameters. :Examples: ('2i', (0.1, 0.2, 0.3)), or '1a'

occ: occupancy (default=1.0)

b: 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)

xrayutilities.materials.spacegrouplattice.**get_default_sgrp_suf** (sgrp_nr)
determine default space group suffix

xrayutilities.materials.wyckpos module

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:
a4*z**4 + a3*z**3 + a2*z**2 + a1*z + a0

Returns: tuple of the four (complex) solutions of aboves equation.

[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,y: data coordinates (do NOT need to be regularly spaced)**

data: data set used for fitting (e.g. intensity at the data coords)

start: set of starting parameters for the fit used as first parameter of function fit_function

drange: 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: [xmin,xmax,ymin,ymax]

fit_function: function which should be fitted, must accept the parameters (x,y,*params)

Returns: **(fitparam,cov):** the set of fitted parameters and covariance matrix

`xrayutilities.math.fit.`**`gauss_fit`** (xdata, ydata, iparams=[], 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: xcoordinates of the data to be fitted**

ydata: ycoordinates of the data which should be fit

keyword parameters:

iparams: initial paramters for the fit, determined automatically if not given

maxit: maximal iteration number of the fit

Returns: params,sd_params,itlim

the Gauss parameters as defined in function Gauss1d(x, *param) and their

errors of the fit, as well as a boolean flag which is false in the case of

a successful fit

`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: data coordinates and values**

Returns: p, rsq: parameters of the linear fit (slope, offest) and the R^2 value

Examples

```
>>> (k, d), R2 = xu.math.linregress(x, y)
```

`xrayutilities.math.fit.`**`multGaussFit`** (*args, **kwargs)
convenience function to keep API stable see multPeakFit for documentation

`xrayutilities.math.fit.`**`multGaussPlot`** (*args, **kwargs)
convenience function to keep API stable see multPeakPlot for documentation

`xrayutilities.math.fit.`**`multPeakFit`**(x, data, peakpos, peakwidth, dranges=None, peaktype='Gaussian')

function to fit multiple Gaussian/Lorentzian peaks with linear background to a set of data

Parameters: **x: x-coordinate of the data**

data: data array with same length as x

peakpos: initial parameters for the peak positions

peakwidth: initial values for the peak width

dranges: 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: type of peaks to be used: can be either 'Gaussian' or 'Lorentzian'

Returns: pos,sigma,amp,background

pos: list of peak positions derived by the fit

sigma: list of peak width derived by the fit

amp: list of amplitudes of the peaks derived by the fit

background: array of background values at positions x

`xrayutilities.math.fit.`**`multPeakPlot`**(x, fpos, fwidth, famp, background, dranges=None, peaktype='Gaussian', fig='xu_plot', fact=1.0)

function to plot multiple Gaussian/Lorentz peaks with background values given by an array

Parameters: **x: x-coordinate of the data**

fpos: list of positions of the peaks

fwidth: list of width of the peaks

famp: list of amplitudes of the peaks

background: array with background values

dranges: list of tuples with (min,max) value of the data ranges to use. does not need to have the same number of entries as fpos

peaktype: type of peaks to be used: can be either 'Gaussian' or 'Lorentzian'

fig: matplotlib figure number or name

fact: factor to use as multiplicator in the plot

`xrayutilities.math.fit.`**`peak_fit`**(xdata, ydata, iparams=[], 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: xcoordinates of the data to be fitted**

ydata: ycoordinates of the data which should be fit

keyword parameters:

iparams: initial paramters for the fit, determined automatically if not specified

peaktype: type of peak to fit: 'Gauss', 'Lorentz', 'PseudoVoigt', 'PseudoVoigtAsym', 'PseudoVoigtAsym2'

maxit: maximal iteration number of the fit

background: type of background, either 'constant' or 'linear'

plot: 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: returns the fitted function, which takes the independent variables as only argument (f(x))

Returns: params,sd_params,itlim[,fitfunc]

the parameters as defined in function Gauss1d/Lorentz1d/PseudoVoigt1d/ PseudoVoigt1dasym(x, *param). In the case of linear background one more parameter is included! For every parameter the corresponding errors of the fit 'sd_params' are returned. A boolean flag 'itlim', which is False in the case of a successful fit is added by default. Further 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 as needed for the calculation of the thermal Debye-Waller-factor by numerical integration
for definition see: :http: //en.wikipedia.org/wiki/Debye_function
D1(x) = (1/x) int_0^x t/(exp(t)-1) dt

Parameters: **x ... argument of the Debye function (float)**

Returns:

D1(x): float value of the Debye function

`xrayutilities.math.functions.`**`Gauss1d`** (x, *p)
function to calculate a general one dimensional Gaussian

Parameters: **p: list of parameters of the Gaussian**

[XCEN,SIGMA,AMP,BACKGROUND] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

x: coordinate(s) where the function should be evaluated

Returns: 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) >>> Gauss1d(numpy.linspace(0,10,100), 5, 1, 1e3, 0)

`xrayutilities.math.functions.`**`Gauss1dArea`** (*p)
function to calculate the area of a Gauss function with neglected background

Parameters: **p: list of parameters of the Gauss-function**

[XCEN,SIGMA,AMP,BACKGROUND]

Returns: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Gauss-function**

[XCEN,YCEN,SIGMAX,SIGMAY,AMP,ANGLE,BACKGROUND]

Returns: 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: **p: list of parameters of the Gauss-function**

[XCEN,YCEN,ZCEN,SIGMAX,SIGMAY,SIGMAZ,AMP,BACKGROUND] SIGMA = FWHM / (2*sqrt(2*log(2)))

x,y,z: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND]

x: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND]

Returns: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Gaussian**

[XCEN,SIGMA] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Lorentzian**

[XCEN,FWHM]

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND,ETA] :ETA: 0 ...1 0 means pure Gauss and 1 means pure Lorentz

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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: coordinate(s) where the function should be evaluated**

p: 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: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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: any scalar of ndarray object**

Returns:

s: Heaviside step function evaluated for all values of x

`xrayutilities.math.functions.`**`kill_spike`**(data, threshold=2.0)

function to smooth **single** data points which differ from the average of the neighboring data points by more than the threshold factor. 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: 1d numpy array with experimental data**

threshold: threshold factor to identify strange data points

Returns: 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: coordinate where the function should be evaluated**

args: 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: 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: coordinates where the function should be evaluated**

args: 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: 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: 1D data array**

n: number of data points to average

Returns:

xsmooth: 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: position of the data points**

data: data values
background: type of background, either 'none', 'constant' or 'linear'
full_output: return background cleaned data and background-parameters

Returns: center of mass position (single float)

`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: position of the data points**

data: data values

Returns: fwhm value (single float)

xrayutilities.math.transforms module

class `xrayutilities.math.transforms.`**`AxisToZ`** (`newzaxis`)
Bases: **`xrayutilities.math.transforms.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.

class `xrayutilities.math.transforms.`**`AxisToZ_keepXY`** (`newzaxis`)
Bases: **`xrayutilities.math.transforms.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

class `xrayutilities.math.transforms.`**`CoordinateTransform`** (`v1`, `v2`, `v3`)
Bases: **`xrayutilities.math.transforms.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!

class `xrayutilities.math.transforms.`**`Transform`** (`matrix`)
Bases: **`object`**

`inverse` **(**`args`**,** `rank=1`**)**
performs inverse transformation a vector, matrix or tensor of rank 4

Parameters: **args: 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: rank of the supplied object. allowed values are 1, 2, and 4

`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.`**`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: numpy.array or list of length 3**

axis: numpy.array or list of length 3
ang: rotation angle in degree (deg=True) or in rad (deg=False)
deg: boolean which 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([  6.12323400e-17,   1.00000000e+00,   0.00000000e+00])
```

`xrayutilities.math.transforms.`**`tensorprod`** (`vec1`, `vec2`)
function implements an elementwise multiplication of two vectors

xrayutilities.math.vector module

module with vector operations for vectors of size 3, since for so short vectors numpy does not give the best performance explicit implementation of the equations is performed together with error checking to ensure vectors of length 3.

`xrayutilities.math.vector.`**`VecAngle`** (`(v1.v2)/(norm(v1)*norm(v2))`)

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

optional keyword arguments:

deg: (default: false) return result in degree otherwise in radiants

Returns: float value with the angle inclined by the two vectors

`xrayutilities.math.vector.`**`VecCross`** (`v1`, `v2`, `out=None`)
Calculate the vector cross product.

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list
out: optional output vector

Returns: float value

`xrayutilities.math.vector.`**`VecDot`** (`v1`, `v2`)
Calculate the vector dot product.

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

Returns: float value

`xrayutilities.math.vector.`**`VecNorm`** (`v`)
Calculate the norm of a vector.

Parameters: **v ..... vector as list or numpy array**

Returns: float holding the vector norm

`xrayutilities.math.vector.`**`VecUnit`** (`v`)
Calculate the unit vector of v.

Parameters: **v ..... vector as list or numpy array**

Returns: numpy array with the unit vector

`xrayutilities.math.vector.`**`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: vector of length 3**

Returns: [xyz][+-]

`xrayutilities.math.vector.`**`getVector`** (`string`)
returns unit vector along a rotation axis given in the syntax 'x+' 'z-' or equivalents

Parameters: **string [xyz][+-]**

Returns: vector along the given direction as numpy array

Module contents

xrayutilities.simpack package

Submodules

xrayutilities.simpack.darwin_theory module

class `xrayutilities.simpack.darwin_theory.`**`DarwinModel`** (`qz`, `qx=0`, `qy=0`, `**kwargs`)
Bases: **`xrayutilities.simpack.models.LayerModel`**
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_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: 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: **`xrayutilities.simpack.darwin_theory.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` = *5.6532499999999999*

classmethod **`abulk`** (x)
calculate the bulk (relaxed) lattice parameter of the Al_{x}Ga_{1-x}As alloy

`asub` = *5.6532499999999999*

`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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors (temp=300**)**
calculates the needed atomic structure factors

static **poisson_ratio** (x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelAlloy** (qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.DarwinModel**, **abc.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.

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

make_monolayers (s**)**
create monolayer sequence from layer list

Parameters: **s: 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: inplane lattice parameter of the substrate

Returns: 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: monolayer list created by make_monolayer()**

prop: name of the property which should be evaluated. Use 'x' for the chemical composition and 'ai' for the inplane lattice parameter.

Returns: zm, propx: z-position, value of the property prop for every monolayer. z=0 is the surface

class xrayutilities.simpack.darwin_theory.**DarwinModelGaInAs001**(qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.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 = *5.6532499999999999*

classmethod **abulk**(x)
calculate the bulk (relaxed) lattice parameter of the Ga_{1-x}In_{x}As alloy

asub = *5.6532499999999999*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors(temp=300**)**
calculates the needed atomic structure factors

static **poisson_ratio**(x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelSiGe001**(qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.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 = *5.4310400000000003*

classmethod **abulk** (x)
calculate the bulk (relaxed) lattice parameter of the alloy

asub = *5.4310400000000003*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors (temp=300**)**
calculates the needed atomic structure factors

static **poisson_ratio** (x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

xrayutilities.simpack.darwin_theory.**GradedBuffer** (xfrom, xto, nsteps, thickness, relaxation=1)
create a multistep graded composition buffer.

Parameters: **xfrom: begin of the composition gradient**

xto: end of the composition gradient

nsteps: number of steps of the gradient

thickness: total thickness of the Buffer in A

relaxation: relaxation of the buffer

Returns: layer object 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

xrayutilities.simpack.fit.**fit_xrr** (reflmod, params, ai, data=None, eps=None, xmin=-inf, xmax=inf, plot=False, verbose=False, elog=True, maxfev=500)
optimize function for a Reflectivity Model using lmfit. The fitting parameters must be specified as instance of lmfits Parameters class.

Parameters: **reflmod: preconfigured SpecularReflectivityModel**

params: instance of lmfits Parameters class. For every layer the parameters '{}_thickness', '{}_roughness', '{}_density', with '{}' representing the layer name are supported. In addition the setup parameters: - 'I0' primary beam intensity - 'background' background added to the simulation - 'sample_width' size of the sample along the beam - 'beam_width' width of the beam in the same units - 'resolution_width' width of the resolution function in deg - 'shift' experimental shift of the incidence angle array

ai: array of incidence angles for the calculation

data: experimental data which should be fitted

eps: (optional) error bar of the data

xmin: minimum value of ai which should be used. a mask is generated to cut away other data

xmax: maximum value of ai which should be used. a mask is generated to cut away other data

plot: 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.

verbose: flag to tell if the variation of the fitting error should be output during the fit.

elog: logarithmic error during the fit

maxfev: maximum number of function evaluations during the leastsq optimization

Returns:

res: MinimizerResult object from lmfit, which contains the fitted parameters in res.params (see res.params.pretty_print) or try lmfit.report_fit(res)

xrayutilities.simpack.helpers module

`xrayutilities.simpack.helpers.`**`coplanar_alphai`**(qx, qz, en='config')
calculate coplanar incidence angle from knowledge of the qx and qz coordinates

Parameters: **qx: inplane momentum transfer component**

qz: out of plane momentum transfer component

en: x-ray energy (eV). By default the value from the config is used.

Returns: the incidence angle in degree. points in the Laue zone are set to 'nan'.

`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: inplane momentum transfer component**

alphai: incidence angle (deg)

en: x-ray energy (eV). By default the value from the config is used.

Returns: the qz position for the given incidence angle

xrayutilities.simpack.models module

class `xrayutilities.simpack.models.`**`DynamicalModel`**(*args, **kwargs)
Bases: **`xrayutilities.simpack.models.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: vector of incidence angles (deg)**

hkl: Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry: 'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

rettype: 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: vector of intensities of the diffracted signal

class `xrayutilities.simpack.models.`**KinematicalModel**(`*args`, `**kwargs`)
Bases: **xrayutilities.simpack.models.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_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: simulation positions along qz**

hkl: Miller indices of the Bragg peak whos truncation rod should be calculated

absorption: flag to tell if absorption correction should be used

refraction: 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: 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: vector of the ratios of the diffracted and primary fluxes

class `xrayutilities.simpack.models.`**KinematicalMultiBeamModel**(`*args`, `**kwargs`)
Bases: **xrayutilities.simpack.models.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.

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: simulation positions along qz**

hkl: Miller indices of the Bragg peak whos truncation rod should be calculated

absorption: flag to tell if absorption correction should be used

refraction: 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: 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: vector of the ratios of the diffracted and primary fluxes

class xrayutilities.simpack.models.**LayerModel**(*args, **kwargs)

Bases: **xrayutilities.simpack.models.Model**, **abc.ABC**

generic model class from which further thin film models can be derived from

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.

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

convolute_resolution(x, y)

convolve simulation result with a resolution function

Parameters: **x: x-values of the simulation, units of x also decide about the**

unit of the resolution_width parameter

y: y-values of the simulation

Returns: convoluted y-data with same shape as y

scale_simulation(y)

scale simulation result with primary beam flux/intensity and add a background.

Parameters: **y: y-values of the simulation**

Returns: scaled y values

class xrayutilities.simpack.models.**SimpleDynamicalCoplanarModel**(*args, **kwargs)

Bases: **xrayutilities.simpack.models.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

Note: This model should not be used in real life scenarios since the made approximations severely fail for distances far from the reference position.

set_hkl(*hkl)
To speed up future calculations of the same Bragg peak optical parameters can be pre-calculated using this function.

Parameters: **hkl: 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: vector of incidence angles (deg)**

hkl: Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry: 'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

idxref: 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: vector of intensities of the diffracted signal

class xrayutilities.simpack.models.**SpecularReflectivityModel**(*args, **kwargs)
Bases: **xrayutilities.simpack.models.LayerModel**
model for specular reflectivity calculations

densityprofile(nz, plot=False)
calculates the electron density of the layerstack from the thickness and roughness of the individual layers

Parameters: **nz: number of values on which the profile should be calculated**

plot: flag to tell if a plot of the profile should be created

Returns: z, eprof: coordinates and electron profile. z = 0 corresponds to the surface

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: vector of incidence angles**

Returns: vector of intensities of the reflectivity signal

xrayutilities.simpack.models.**startdelta**(start, delta, num)

xrayutilities.simpack.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.simpack.mpl_helper.**SqrtAllowNegScale**(axis, **kwargs)
Bases: **matplotlib.scale.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: **matplotlib.transforms.Transform**

input_dims = *1*

inverted ()

is_separable = *True*

output_dims = *1*

transform_non_affine (a)

class SqrtAllowNegScale.**SqrtTransform** (shorthand_name=None)

Bases: **matplotlib.transforms.Transform**

input_dims = *1*

inverted ()

return the inverse transform for this transform.

is_separable = *True*

output_dims = *1*

transform_non_affine (a)

This transform takes an Nx1 numpy array and returns a transformed copy.

SqrtAllowNegScale.**get_transform ()**

SqrtAllowNegScale.**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.

SqrtAllowNegScale.**name** = *'sqrt'*

SqrtAllowNegScale.**set_default_locators_and_formatters** (axis)

class xrayutilities.simpack.mpl_helper.**SqrtTickLocator** (nbins=7, symmetric=True)

Bases: **matplotlib.ticker.Locator**

set_params (nbins, symmetric)

Set parameters within this locator.

tick_values (vmin, vmax)

view_limits (dmin, dmax)

Set the view limits to the nearest multiples of base that contain the data

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)`

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.

Noteable class parameters:

max_history_length: the number of histories to cache (default=5); can be overridden if memory is an issue.

length_scale_m: 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.

`add_buffer` **(**b**)**

add a numpy array to the list of objects that can be thrown away on pickling.

Parameters: **b: the buffer to add to the list**

Returns: 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: the edge of the function farthest from the singularity,**

referenced to epsvals

innerbound: the edge closest to the singularity, referenced to epsvals

epsvals: the array of two-theta values or offsets

destination: an array into which final results are summed. modified in place!

peakpos: the position of the singularity, referenced to epsvals.

y0: the constant offset

k: the scale factor

Returns: (*lower_index*, *upper_index*) 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: a list of convolvers to select. If *None*, use all**

found convolvers.

compute_derivative: if *True*, also return d/dx(function) for peak position fitting

Returns: a profile_data object with much information about the peak

conv_absorption ()
compute the sample transparency correction, including the finite-thickness version

Returns: the convolver

conv_axial ()
compute the Fourier transform of the axial divergence component

Returns: 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: the convolver

conv_emission ()
compute the emission spectrum and (for convenience) the particle size widths

Returns: the convolver for the emission and particle sizes

Note

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: 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

conv_receiver_slit ()
compute the rectangular convolution for the receiver slit or SiPSD pixel size

Returns: the convolver

conv_si_psd ()
compute the convolver for the integral of defocusing of the face of an Si PSD

Returns: the convolver

conv_smoother ()

compute the convolver to smooth the final result with a Gaussian before downsampling.

Returns: the convolver

conv_tube_tails ()
compute the Fourier transform of the rectangular tube tails function

Returns: 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: length of the xray filament**

Ls: length of the sample
Lr: length of the receiver slit
R: diffractometer length, assumed symmetrical
twotheta: angle, in radians, of the center of the computation
beta: offset angle
epsvals: array of offsets from center of computation, in radians

Returns: (*epsvals*, *idxmin*, *idxmax*, *I2p*, *I2m*).

idxmin and *idxmax* are the full python-style bounds of the non-zero region of *I2p* and *I2m*. *I2p* and *I2m* are 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: length of the xray filament**

Ls: length of the sample
Lr: length of the receiver slit
R: the (assumed symmetrical) diffractometer radius
twotheta: angle, in radians, of the center of the computation
epsvals: array of offsets from center of computation, in radians
sollerIdeg: the full-width (both sides) cutoff angle of the incident Soller slit
sollerDdeg: the full-width (both sides) cutoff angle of the detector Soller slit
nsteps: the number of subdivisions for the integral
axDiv: not used

Returns: 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: the name of the convolver cache buffer to update**

width: the width in 2-theta space of the tophat

Returns: the updated convolver buffer, or *None* if the width was *None*

get_conv (name**,** key**,** format=<type '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: the name of the convolver to seek**

key: any hashable object which identifies the parameters for the computation

format: the type of the array to create, if one is not found.

Returns: 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 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: 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: a number of bins.**

Returns: a bin count somewhat larger than *count* which is efficient for FFT

info_emission = *{'param_info': {'emiss_lor_widths': ('Lorenztian emission fwhm (m)', (1e-13,)), 'crystallite_size_lor': ('Lorentzian crystallite size fwhm (m)', 1e-06), 'emiss_wavelengths': ('wavelengths (m)', (1.58e-10,)), 'emiss_intensities': ('relative intensities', (1.0,)), 'emiss_gauss_widths': ('Gaussian emissions fwhm (m)', (1e-13,)), 'crystallite_size_gauss': ('Gaussian crystallite size fwhm (m)', 1e-06)}, 'help': 'this should be help information', 'group_name': 'Incident beam and crystal size'}*

info_global = *{'param_info': {'twotheta0_deg': ('Bragg center of peak (degrees)', 30.0), 'd': ('d spacing (m)', 4e-10), 'dominant_wavelength': ('wavelength of most intense line (m)', 1.5e-10)}, 'help': 'this should be help information', 'group_name': 'Global parameters'}*

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 calles when the class attribute 'isotropic' is False.

Parameters: **hkl1,2: Miller indices to be checked for equivalence**

crystalsystem: symmetry class of the material which is considered

Returns: True or False

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: exact position of center bin, in degrees**

twotheta_approx_window_fullwidth_deg: approximate desired width

twotheta_exact_bin_spacing_deg: the exact bin spacing to use

set_parameters(convolver='global', **kwargs)
update the dictionary of parameters associated with the given convolver

Parameters: **convolver: the name of the convolver. name 'global', e.g., attaches**

to function 'conv_global'

kwargs: 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: the center position of the middle bin of**

the window, in degrees

twotheta_window_fullwidth_deg: the full width of the window, in degrees

twotheta_output_points: the number of bins in the final output

str_emission()
format the emission spectrum and crystal size information

Returns: the formatted information

str_global()
returns a string representation for the global context.

Returns: report on global parameters.

class xrayutilities.simpack.powder.**PowderDiffraction**(mat, **kwargs)
Bases: **xrayutilities.experiment.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 used 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: two theta values at which the powder pattern should be**

calculated.

Note

Note: Bragg peaks are only included up to tt_cutoff set in
the class constructor!

****kwargs: additional keyword arguments are passed to the Convolve**
function

Returns: output intensity values for the twotheta values given in the input

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: two theta values at which the powder pattern should be**

calculated.

window_width: width of the calculation window of a single peak

mode: 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

correction_factor (`ang`**)**
calculate the correction factor for the diffracted intensities. This contains the polarization effects and the Lorentz factor

Parameters: **ang: theta diffraction angles for which the correction should be**

calculated

Returns:

f: array of the same shape as ang containing the correction factors

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 if isotropic lines at the same q-position can be merged to one line to reduce the calculational effort

Parameters: **data: numpy field array with values of 'hkl' (Miller indices of the**

peaks), 'q' (q-position), and 'r' (reflection strength) as produced by the structure_factors method

Returns: hkl, q, ang, r: Miller indices, q-position, diffraction angle (Theta),
and reflection strength of the material

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

structure_factors (`tt_cutoff`**)**
determine structure factors/reflection strength of all Bragg peaks up to tt_cutoff

Parameters: **tt_cutoff: upper cutoff value of 2theta until which the reflection**
strength are calculated

Returns: numpy array with field for 'hkl' (Miller indices of the peaks),
'q' (q-position), and 'r' (reflection strength) of the Bragg peaks

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={}`**)**
update settings of all instances of FP_profile

Parameters: **newsettings: dictionary with new settings. It has to include one**
subdictionary for every convolver which should have its settings changed.

wavelength

window_width

`xrayutilities.simpack.powder.`**chunkify** (`lst`, `n`)

class `xrayutilities.simpack.powder.`**convolver_handler**
Bases: **object**
manage the convolvers of on process

add_convolver (`convolver`**)**

calc (`run`**,** `ttpeaks`**)**
calculate profile function for selected convolvers

Parameters: **run: list of flags of length of convolvers to tell which convolver**
needs to be run

ttpeaks: peak positions for the convolvers

Returns: 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')
Bases: **multiprocessing.managers.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

add_symbol (**kwargs)
add new symbols to both the attributes and dictionary for the class

Parameters: **kwargs 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.

create_fitparameters ()
function to create a fit model with all instrument and sample parameters.

Parameters: **pass**

Returns: lmfit Parameters instance

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: angular values for the fit

data: experimental intensities for the fit

std: standard deviation of the experimental data. if 'None" the sqrt of the data will be used

maxfev: maximal number of simulations during the least squares refinement

Returns: lmfit MinimizerResult

set_background (btype, **kwargs)
define background as spline or polynomial function

Parameters: **btype: background type: either 'polynomial' or 'spline'. Depending on**

this value the expected keyword arguments differ.

kwargs: **'spline':**

x: x-values (twotheta) of the background points

y: intensity values of the background

'polynomial':

p: polynomial coefficients from the highest degree to the constant term. len of p decides about the degree of the polynomial

set_lmfit_parameters(lmparams**)**
function to update the settings of this class during an least squares fit

Parameters: **lmparams: lmfit Parameters list of sample and instrument parameters**

set_parameters(params**)**
set simulation parameters of all subobjects

Parameters: **params: 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: positions at which the powder spectrum should be evaluated**

****kwargs:**

background: 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: summed powder diffraction intensity of all materials present in the

model

Known issue: possibility to add a background is currently missing!

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: experimental datapoints**

sim: simulated data

weight: weight factor in the least squares sum. If it is None the weight is estimated from the counting statistics of 'exp'

std: standard deviation of the experimental data. alternative way of specifying the weight factor. when both are given weight overwrites std!

Nvar: number of variables in the refinement

disp: flag to tell if a line with the calculated values should be printed.

Returns: M, Rp, Rwp, Rwpexp, chi2

xrayutilities.simpack.powdermodel.**plot_powder** (twotheta, exp, sim, mask=None, scale='sqrt', fig='XU:powder', show_diff=True, show_legend=True)
Convenience function to plot the comparison between experimental and simulated powder diffraction data

Parameters: **twotheta: angle values used for the x-axis of the plot (deg)**

exp: experimental data (same shape as twotheta). If None only the simulation and no difference will be plotted

sim: simulated data

mask: mask to reduce the twotheta values to the be used as x-coordinates of sim

scale: string specifying the scale of the y-axis. Valid are: 'linear, 'sqrt', and 'log'.

fig: matplotlib figure name (figure will be cleared!)

show_diff: flag to specify if a difference curve should be shown

show_legend: flag to specify if a legend should be shown

xrayutilities.simpack.smaterials module

class xrayutilities.simpack.smaterials.**CrystalStack** (name, *args)
Bases: **xrayutilities.simpack.smaterials.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: **xrayutilities.simpack.smaterials.CrystalStack**
generates a sequence of layers with a gradient in chemical composition

class xrayutilities.simpack.smaterials.**Layer** (material, thickness, **kwargs)
Bases: **xrayutilities.simpack.smaterials.SMaterial**
Object describing part of a thin film sample. The properties of a layer :are:

Material: an xrayutilties material describing optical and crystal properties of the thin film

Thickness: film thickness in Angstrom

Roughness: root mean square roughness of the top interface in Angstrom

class xrayutilities.simpack.smaterials.**LayerStack** (name, *args)
Bases: **xrayutilities.simpack.smaterials.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: **_abcoll.MutableSequence**
class representing the basics of a list of materials for simulations within xrayutilities. It extends the built in list type.

check (v)

insert (i, v)

class xrayutilities.simpack.smaterials.**Powder** (material, volume, **kwargs)
Bases: **xrayutilities.simpack.smaterials.SMaterial**
Object describing part of a powder sample. The properties of a powder :are:

Material: an xrayutilties material (Crystal) describing optical and crystal properties of the thin film

Volume: powder's volume (in pseudo units, since only the relative volume enters the calculation)

Optionally also the following can be set:

crystallite_size_lor:	Lorentzian crystallite size fwhm (m)
crystallite_size_gauss:	Gaussian crystallite size fwhm (m)
strain_lor:	extra peak width proportional to tan(theta)
strain_gauss:	extra peak width proportional to tan(theta)

class `xrayutilities.simpack.smaterials.`**`PowderList`**(`name`, `*args`)
Bases: **`xrayutilities.simpack.smaterials.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: **`xrayutilities.simpack.smaterials.CrystalStack`**
generate a sequence of pseudomorphic crystalline Layers. Surface orientation is assumed to be 001 and materials must be cubic/tetragonal.

`insert`(`i`, `v`)

`make_epitaxial`(`i`)

`trans` = *<xrayutilities.math.transforms.Transform object>*

class `xrayutilities.simpack.smaterials.`**`PseudomorphicStack111`**(`name`, `*args`)
Bases: **`xrayutilities.simpack.smaterials.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`, `**kwargs`)
Bases: **`object`**
Simulation Material. Extends the xrayutilities Materials by properties needed for simulations

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.

xrayutilities

xrayutilities package

Subpackages

xrayutilities.analysis package

Submodules

xrayutilities.analysis.line_cuts module

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_ang`** (`qx`, `qz`, `intensity`, `omcenter`, `ttcenter`, `omrange`, `npoints`, `**kwargs`)

extracts an omega scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: omega-position at which the omega scan should be extracted

ttcenter: 2theta-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative omega positions are returned :(default: True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns: **om,omint:** omega scan coordinates and intensities (bounds=False)

om,omint,(qxb, qzb): omega scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omcut, intcut = get_omega_scan(qx,qz,intensity,0.0,5.0,2.0,200)
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_bounds_ang`** (`omcenter`, `ttcenter`, `omrange`, `npoints`, `**kwargs`)

return reciprocal space boundaries of omega scan

Parameters: **omcenter: omega-position at which the omega scan should be extracted**

ttcenter: 2theta-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

lam: wavelength for use in the conversion to angular coordinates

Returns: **qx,qz:** reciprocal space coordinates of the omega scan boundaries

Examples

```
>>> qxb,qzb = get_omega_scan_bounds_ang(1.0,4.0,2.4,240,qrange=0.1)
```

`xrayutilities.analysis.line_cuts.`**`get_omega_scan_q`** (`qx`, `qz`, `intensity`, `qxcenter`, `qzcenter`, `omrange`, `npoints`, `**kwargs`)

extracts an omega scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the omega scan should be extracted

qzcenter: qz-position at which the omega scan should be extracted

omrange: range of the omega scan to extract

npoints: number of points of the omega scan

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative omega positions are returned :(default: True)

bounds: flag to specify if the scan bounds should be returned; :(default: False)

Returns:

om,omint: omega scan coordinates and intensities (bounds=False)

om,omint,(qxb, qzb): omega scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omcut, intcut = get_omega_scan(qx,qz,intensity,0.0,5.0,2.0,200)
```

`xrayutilities.analysis.line_cuts.`**`get_qx_scan`**(qx, qz, intensity, qzpos, **kwargs)

extract qx line scan at position qzpos from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given range along qz

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

bounds: flag to specify if the scan bounds of the extracted scan should be returned (default:False)

Returns:

qx,qxint: qx scan coordinates and intensities (bounds=False)

qx,qxint,(qxb,qyb): qx scan coordinates and intensities + scan bounds for plotting

Examples

```
>>> qxcut,qxcut_int = get_qx_scan(qx,qz,inten,5.0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts.`**`get_qz_scan`**(qx, qz, intensity, qxpos, **kwargs)

extract qz line scan at position qxpos from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given range along qx

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qz,qzint: qz scan coordinates and intensities

Examples

```
>>> qzcut,qzcut_int = get_qz_scan(qx,qz,inten,1.5,qrange=0.03)
```

`xrayutilities.analysis.line_cuts.`**`get_qz_scan_int`** (`qx`, `qz`, `intensity`, `qxpos`, `**kwargs`)

extracts a qz scan from a gridded reciprocal space map with integration along omega (sample rocking angle) or 2theta direction

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

angrange: integration range in angular direction

qmin,qmax: minimum and maximum value of extracted scan axis

bounds: flag to specify if the scan bounds of the extracted scan should be returned (default:False)

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

wl: wavelength used to determine angular integration positions

Returns:

qz,qzint: qz scan coordinates and intensities (bounds=False)

qz,qzint,(qzb,qzb): qz scan coordinates and intensities + scan bounds for plotting

Examples

```
>>> qzcut,qzcut_int = get_qz_scan_int(qx,qz,inten,5.0,omrange=0.3)
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan_ang`** (`qx`, `qz`, `intensity`, `omcenter`, `ttcenter`, `ttrange`, `npoints`, `**kwargs`)

extracts a radial scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: om-position at which the radial scan should be extracted

ttcenter: tt-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,tt,radint: omega,two theta scan coordinates and intensities (bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omc, ttc, cut_int = get_radial_scan_ang(qx, qz, intensity, 32.0, 64.0,
                                            30.0, 800, omrange = 0.2)
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan_bounds_ang`**(`omcenter`, `ttcenter`, `ttrange`, `npoints`, `**kwargs`)

return reciprocal space boundaries of radial scan

Parameters: **omcenter: om-position at which the radial scan should be extracted**

ttcenter: tt-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qxrad,qzrad: reciprocal space boundaries of radial scan

Examples

```
>>>
```

`xrayutilities.analysis.line_cuts.`**`get_radial_scan_q`**(qx, qz, `intensity`, `qxcenter`, `qzcenter`, `ttrange`, `npoints`, `**kwargs`)

extracts a radial scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the radial scan should be extracted

qzcenter: qz-position at which the radial scan should be extracted

ttrange: two theta range of the radial scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range perpendicular to scan direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

om,tt,radint: omega,two theta scan coordinates and intensities (bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> omc, ttc, cut_int = get_radial_scan_q(qx, qz, intensity, 0.0, 5.0,
                                          1.0, 100, omrange = 0.01)
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_ang`**(`qx`, `qz`, `intensity`, `omcenter`, `ttcenter`, `ttrange`, `npoints`, `**kwargs`)

extracts a twotheta scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

omcenter: om-position at which the 2theta scan should be extracted

ttcenter: tt-position at which the 2theta scan should be extracted

ttrange: two theta range of the scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

tt,ttint: two theta scan coordinates and intensities (bounds=False)

tt,ttint,(qxb,qzb): 2theta scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> ttc,cut_int = get_ttheta_scan_ang(qx,qz,intensity,32.0,64.0,4.0,400)
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_bounds_ang`**`(omcenter, ttcenter, ttrange, npoints, **kwargs)`

return reciprocal space boundaries of 2theta scan

Parameters: **omcenter: om-position at which the 2theta scan should be extracted**

ttcenter: tt-position at which the 2theta scan should be extracted

ttrange: two theta range of the 2theta scan to extract

npoints: number of points of the 2theta scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

lam: wavelength for use in the conversion to angular coordinates

Returns:

qxtt,qztt: reciprocal space boundaries of 2theta scan (bounds=False)

tt,ttint,(qxb,qzb): 2theta scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>>
```

`xrayutilities.analysis.line_cuts.`**`get_ttheta_scan_q`**`(qx, qz, intensity, qxcenter, qzcenter, ttrange, npoints, **kwargs)`

extracts a twotheta scan from a gridded reciprocal space map

Parameters: **qx: equidistant array of qx momentum transfer**

qz: equidistant array of qz momentum transfer

intensity: 2D array of gridded reciprocal space intensity with shape (qx.size,qz.size)

qxcenter: qx-position at which the 2theta scan should be extracted

qzcenter: qz-position at which the 2theta scan should be extracted

ttrange: two theta range of the scan to extract

npoints: number of points of the radial scan

****kwargs: possible keyword arguments:**

omrange: integration range in omega direction

Nint: number of subscans used for the integration (optionally)

lam: wavelength for use in the conversion to angular coordinates

relative: determines if absolute or relative two theta positions are returned (default=True)

bounds: flag to specify if the scan bounds should be returned :(default: False)

Returns:

tt,ttint: two theta scan coordinates and intensities (bounds=False)

om,tt,radint,(qxb,qzb): radial scan coordinates and intensities + reciprocal space bounds of the extraced scan (bounds=True)

Examples

```
>>> ttc,cut_int = get_ttheta_scan_q(qx,qz,intensity,0.0,4.0,4.4,440)
```

xrayutilities.analysis.line_cuts.**getindex** (x, y, xgrid, ygrid)

gives the indices of the point x,y in the grid given by xgrid ygrid xgrid,ygrid must be arrays containing equidistant points

Parameters: **x,y: coordinates of the point of interest (float)**

xgrid,ygrid: grid coordinates in x and y direction (array)

Returns:

ix,iy: index of the closest gridpoint (lower left) of the point (x,y)

xrayutilities.analysis.line_cuts3d module

xrayutilities.analysis.line_cuts3d.**get_qx_scan3d** (gridder, qypos, qzpos, **kwargs)

extract qx line scan at position y,z from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qypos,qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qx,qxint: qx scan coordinates and intensities

Examples

```
>>> qxcut,qxcut_int = get_qx_scan3d(gridder,0,0,qrange=0.03)
```

xrayutilities.analysis.line_cuts3d.**get_qy_scan3d** (gridder, qxpos, qzpos, **kwargs)

extract qy line scan at position x,z from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qxpos,qzpos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qy,qyint: qy scan coordinates and intensities

Examples

```
>>> qycut,qycut_int = get_qy_scan3d(gridder,0,0,qrange=0.03)
```

xrayutilities.analysis.line_cuts3d.**get_qz_scan3d** (gridder, qxpos, qypos, **kwargs)

extract qz line scan at position x,y from a gridded reciprocal space map by taking the closest line of the intensity matrix, or summing up a given area around this position

Parameters: **gridder: 3d xrayutilities.Gridder3D object containing the data**

qxpos,qypos: position at which the line scan should be extracted

****kwargs: possible keyword arguments:**

qrange: integration range perpendicular to scan direction

qmin,qmax: minimum and maximum value of extracted scan axis

Returns:

qz,qzint: qz scan coordinates and intensities

Examples

```
>>> qzcut,qzcut_int = get_qz_scan3d(gridder,0,0,qrange=0.03)
```

`xrayutilities.analysis.line_cuts3d.`**`getindex3d`** (`x`, `y`, `z`, `xgrid`, `ygrid`, `zgrid`)

gives the indices of the point x,y,z in the grid given by xgrid ygrid zgrid xgrid,ygrid,zgrid must be arrays containing equidistant points

Parameters: **x, y, z: coordinates of the point of interest (float)**

xgrid, ygrid, zgrid: grid coordinates in x, y, z direction (array)

Returns: ix, iy, iz: index of the closest gridpoint (lower left) of the point

(x, y, z)

xrayutilities.analysis.misc module

miscellaneous functions helpful in the analysis and experiment

`xrayutilities.analysis.misc.`**`getangles`** (`peak`, `sur`, `inp`)

calculates the chi and phi angles for a given peak

Parameters: **peak: array which gives hkl for the peak of interest**

sur: hkl of the surface

inp: inplane reference peak or direction

Returns: [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.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 ..... outer detector arm angle**

angle2: inner detector arm angle

ccdimages: images of the ccd taken at the angles given above

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

r_i: primary beam direction [xyz][+-] default 'x+'

keyword_arguments:

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

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

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

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

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

nwindow: 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: 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, 'config'), fix=(False, False, True, 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 .. 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: outer detector arm angle

angle2: inner detector arm angle

ccdimages: images of the ccd taken at the angles given above

hkls: array/list of hkl values for every image

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

material: material used as reference crystal

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

r_i: primary beam direction [xyz][+-] default 'x+'

keyword_arguments:

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

cut_off: cut off intensity to decide if image is used for the determination or not. default: 0.1 = 10%

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

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

nwindow: 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: flag to tell if you want to see debug output of the script (switch this to true only if you can handle it :))

`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: angular coordinates of the measurement (numpy.ndarray)**

either with size of psd or of psd.shape[0]

psd: intensity values needed for fitting

omalign: aligned omega value, used as first guess in the fit

ttalign: 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 class used for the conversion between angular and reciprocal space.

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

peaktype: can be 'Gauss' or 'Lorentz' to fit either of the two peak shapes

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

Returns:

omfit,ttfit,params,covariance: fitted angular values, and the fit parameters (of the Gaussian/Lorentzian) as well as their errors

`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 of angles in degree of measured detector spectra**

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

keyword arguments:

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

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

other options are passed to psd_chdeg function, options include:

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

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

Note: see help of psd_chdeg for more options

Returns: pixelwidth (at one meter distance), center_channel[, detector_tilt]

Note

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

distance L
pixelwidth is negative in case the hit channel number decreases upon an increase of the detector angle The function also prints out how a linear detector can be initialized using the results obtained from this calibration. Carefully check the results

`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: azimuths in which the reflection was aligned (deg)**

aomega: aligned omega values (deg)

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

omega0: 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: flag to specify if a visualization of the fit is wanted. :default: True

Returns: [omega0,phi0,miscut]

list with fitted values for

omega0: the omega value of the reflection should be close to the nominal one

phi0: the azimuth in which the primary beam looks upstairs

miscut: amplitude of the sinusoidal variation == miscut angle

`xrayutilities.analysis.sample_align.`**psd_chdeg** (`angles`, `channels`, `stdev=None`, `usetilt=True`, `plot=True`, `datap='kx'`, `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: detector angles for which the position of the beam was**

measured

channels: detector channels where the beam was found

keyword arguments:

stdev: standard deviation of the beam position

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

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

datap: plot format of data points

modelline: plot format of modelline

modeltilt: plot format of modeltilt

fignum: figure number to use for the plot

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

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

dlabel: label of the data line to be used in the plot

figtitle: boolean to tell if the figure title should show the fit parameters

Returns: (pixelwidth,centerch,tilt)

pixelwidth: 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: center channel of the detector

tilt: tilt of the detector from perpendicular to the beam

(will be zero in case of usetilt=False)

Note

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

`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** : primary beam channel number

angles: list or numpy.array with angles

channels: list or numpy.array with corresponding detector channels

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

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

Examples

```
>>> psd_refl_align(500,[0,0.1,0.2,0.3],[550,600,640,700])
```

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: **`xrayutilities.io.filedir.FileDirectory`**

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

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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

`xrayutilities.io.cbf.`**`makeNaturalName`** (`name`)

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.`**`gettty08_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: name of the scans, for multiple scans this needs to be a**

template string

scannumbers: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent)

**keyargs: keyword arguments are passed on to tty08File

Returns: MAP

or

[ang1,ang2,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with 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.
Required constructor arguments:

filename: a string with the name of the tty08-file

Optional keyword arguments:

mcadir: directory name of MCA files

`Read` **()**

Read the data from the file

`ReadMCA` **()**

xrayutilities.io.edf module

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

Bases: **`xrayutilities.io.filedir.FileDirectory`**

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

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: 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 ..... a HDF5 file object or name**

optional keyword arguments:

group: group where to store the data (default to the root of the file)
comp: activate compression - true by default

data

`xrayutilities.io.edf.`**`makeNaturalName`** (`name`)

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.

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

Parameters: **nx,ny: number of bins in x,y direction**

optional keyword arguments:

counter: name of the counter to use for gridding (default: 'mpx4int' (ID01))
gridrange: range for the gridder: format: ((xmin,xmax),(ymin,ymax))

Returns: 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: name of the motor for which the position is wanted**

Returns:

val: 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** (filename, scannr, xmotor='adcX', ymotor='adcY', path='')

Bases: **xrayutilities.io.fastscan.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

getccdFileTemplate (specscan**,** datadir=None**,** keepdir=0**,** numfmt='%04d'**)**

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

Parameters: **specscan: spec-scan object from which header the CCD directory should**

be extracted

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

numfmt: format string for the CCD file number (optional)

Returns:

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

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

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

Parameters: **nx,ny: number of bins in x,y direction**

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

optional:

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

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

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

filterfunc: 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: regular x,y-grid as well as 4-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.

`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: list of shifts in x-direction for every FastScan in the**

data structure

deltay: same for the y-direction

`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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

optional:

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

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

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: number of bins in x,y direction**

optional keyword arguments:

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

gridrange: 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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

qnx: number of points in the Qx direction of the gridded reciprocal space map

qny: same for y direction

qnz: same for z directino

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

optional:

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

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

filterfunc: 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: 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, 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: real space x-position or index in x direction**

posy: real space y-position or index in y direction

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

optional:

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

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

filterfunc: 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: sample orientation matrix

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

Returns:

qx,qy,qz,ccddata,valuelist: raw data of the reciprocal space map and

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 ..... a HDF5 file object or name**

optional keyword arguments:

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

comp: activate compression - true by default

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

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.

`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: filename of the file to open (full including path)**

mode: mode in which the file should be opened

Returns: file handle of the opened file

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.
Required constructor arguments:

filename: a string with the name of the data file

`Read``()`
Read the data from the 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: number of the numors, or list of numbers. This will be**

transformed to a string and used as a filename (int, str, or iterable (list, tuple))

***args: names of the motors (optional) (strings)**

e.g.: 'omega', 'gamma'

****kwargs: keyword arguments are passed on to numorFile. e.g. 'path' for**

the files directory

Returns: data

or

[ang1,ang2,...],data:

angular positions position together with 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=<type '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.

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: 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

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

Bases: **xrayutilities.io.imagereader.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.

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

Bases: **xrayutilities.io.imagereader.ImageReader**

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

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

Bases: **xrayutilities.io.imagereader.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.

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

Bases: **xrayutilities.io.imagereader.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.

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

Parameters: **filename: filename of the image to be read. so far only single**

filenames are supported. The data might be compressed.

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

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

`xrayutilities.io.panalytical_xml.`**`getOmPixcel`**(`omraw`, `ttraw`)
function to reshape the Omega values into a form needed for further treatment with xrayutilities

`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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames

scannrs: int or list of scan numbers

path: common path to the filenames

roi: region of interest for the PIXCel detector, for other measurements this is not usefull!

Returns:

om,tt,psd: 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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames given by the scannrs keyword argument

***motors: motor names to return: e.g.: 'Omega','2Theta',...**

one can also use abbreviations 'Omega' = 'om' = 'o' '2Theta' = 'tt' = 't' 'Chi' = 'c' 'Phi' = 'p'

****kwargs:**

scannrs: int or list of scan numbers

path: common path to the filenames

Returns: scanmot,mot1,mot2,...,detectorint: as flattened numpy arrays

Examples

```
>>> scanmot,om,tt,inte = xrayutilities.io.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

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

Bases: **xrayutilities.io.pdcif.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

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
Required constructor arguments:

filename: a string with the name of the ras-file

keyword argument (optional):

path: path to the data file

Read ()
Read the data from the file

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
Required constructor arguments:

filename: file name of the data file

pos: seek position of the RAS_HEADER_START line

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: name of the scans, for multiple scans this needs to be a**

template string

scannumbers: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent) **kwargs: keyword arguments forwarded to RASFile function

Returns: rasdata

or

[ang1,ang2,...],rasdata:

angular positions are extracted from the respective scan header together with all the data values as stored in the data file (includes the intensities e.g. rasdata['int']).

Examples

```
>>> [om,tt],MAP = xu.io.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.

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: 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 use 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

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

parse ()

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

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: template string for the file names, can contain**

a %d which is replaced by the scan number or be a list of filenames

scannrs: int or list of scan numbers

path: common path to the filenames

scantype: 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: number of channels of the MCA (needed for "tsk" measurements)

Returns:

om,tt,psd: as flattened numpy arrays

Examples

```
>>> om,tt,psd = xrayutilities.io.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`**

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={}`**)**
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.

required arguments:

h5f: a HDF5 file object or its filename

optional keyword arguments:

comp: 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.

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

`Parse` **()**

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={}`**,** `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.

input arguments:

h5f: a HDF5 file object or its filename

optional keyword arguments:

group: name or group object of the HDF5 group where to store the data
title: a string with the title for the data, defaults to the name of scan if empty
optattrs: a dictionary with optional attributes to store for the data
comp: activate compression - true by default

SetMCAParams (`mca_column_format`**,** `mca_channels`**,** `mca_start`**,** `mca_stop`**)**
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

required input aguments:

mca_column_format: number of columns used to save the data
mca_channels: number of MCA channels stored
mca_start: first channel that is stored
mca_stop: last channel that is stored

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: 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:**

newfig: if True a new figure instance will be created otherwise an existing one will be used
logy: 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 object of a HDF5 file opened using h5py or its filename**

scans: number of the scans of the reciprocal space map (int,tuple or list)

*args: names of the motors (optional) (strings) to read reciprocal space maps measured in coplanar diffraction give: :omname: e.g. name of the omega motor (or its equivalent) :ttname: e.g. name of the two theta motor (or its equivalent)

****kwargs (optional):**

samplename: string with the hdf5-group containing the scan data if ommited the first child node of h5f.root will be used
rettype: how to return motor positions. by default a list of arrays is returned. when rettype == 'numpy' a record array will be returned.

Returns: MAP

or

[ang1,ang2,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with all the data values as stored in the data file (includes the intensities e.g. MAP['MCA']).

Examples

```
>>> [om, tt], MAP = xu.io.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 object**

scans: number of the scans of the reciprocal space map (int,tuple or list)

args: names of the motors and counters (strings)

keyword arguments:

rettype: 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,...]:

coordinates and counters from the SPEC file

Examples

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

`xrayutilities.io.spec.`**`makeNaturalName`** (`name`)

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.
Required constructor arguments:

filename: a string with the name of the SPECTRA file

Optional keyword arguments:

mcatmp: template for the MCA files

mcastart,mcastop: 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.

required input arguments:

h5file: string or HDF5 file object
name: name of the group where to store the data

optional keyword arguments:

group: root group where to store the data
mcaname: Name of the MCA in the HDF5 file

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

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.

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

append (`col`**)**

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

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

`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 object of a HDF5 file opened using h5py**

scans: number of the scans of the reciprocal space map (int,tuple or list)

***args: arbitrary number of motor names (strings)**

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

****kwargs (optional):**

mca: name of the mca data (if available) otherwise None :(default: "MCA")
samplename: 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,...],MAP:

angular positions of the center channel of the position sensitive detector (numpy.ndarray 1D) together with all 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**

f(q, en='config')
function to calculate the atomic structure factor F

Parameters: **q: momentum transfer**

en: energy for which F should be calculated, if omitted the value from the xrayutilities configuration is used

Returns: f (float)

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: flag, result: if the flag is True then result contains the cached result, otherwise result is None

max_cache_length = *10*

set_cache(prop, key, result)
set result to be cached to speed up future calls

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.**CIFFile**(filename, digits=3)
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

Lattice()
returns a lattice object with the structure from the CIF file

Parse()
function to parse a CIF file. 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.

xrayutilities.materials.cif.**testwp** (parint, wp, cifpos, digits)
test if a Wyckoff position can describe the given position from a CIF file

Parameters: **parint: integer telling which Parameters the given Wyckoff position has**

wp: expression of the Wyckoff position (string of tuple)
cifpos: (x,y,z) position of the atom in the CIF file
digits: number of digits for which for a comparison of floating point numbers will be rounded to

Returns: foundflag, pars: flag to tell if the positions match and if necessary any parameters associated with the position

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.

required input arguments:

dbname: name of the database
dbdesc: 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.

required input arguments:

name: a string with the name of the material
description: a string with a description of the material

GetF0 (q**,** dset='default'**)**
Obtain the f0 scattering factor component for a particular momentum transfer q.

required input argument:

q: single float value or numpy array
dset: 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.

required input arguments:

en: float or numpy array with the energy

GetF2 (en**)**
Return the imaginary part of the scattering factor for a certain energy en.

required input arguments:

en: float or numpy array with the energy

Open (`mode='r'`**)**
Open an existing database file.

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

required input argument:

parameters: list or numpy array with the fit parameters
subset: specifies under which name the f0 values should be saved

SetF1F2 (`en`**,** `f1`**,** `f2`**)**
Set f1, f2 values for the active material.

required input arguments:

en: list or numpy array with energy in (eV)
f1: list or numpy array with f1 values
f2: list or numpy array with 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.

required input arguments:

name: string with the name of the material

SetWeight (`weight`**)**
Save weight of the element as float

required input argument:

weight: atomic standard weight of the element (float)

`xrayutilities.materials.database.`**add_f0_from_intertab** (`db`, `itf`)
Read f0 data from International Tables of Crystallography and add it to the database.

`xrayutilities.materials.database.`**add_f0_from_xop** (`db`, `xop`)
Read f0 data from f0_xop.dat and add it to the database.

`xrayutilities.materials.database.`**add_f1f2_from_ascii_file** (`db`, `asciifile`, `element`)
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`)
Read f1 and f2 data from Henke database and add it to the database.

`xrayutilities.materials.database.`**add_f1f2_from_kissel** (`db`, `kf`)
Read f1 and f2 data from Henke database and add it to the database.

`xrayutilities.materials.database.`**add_mass_from_NIST** (`db`, `nistfile`)
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.`**init_material_db** (`db`)

xrayutilities.materials.elements module

xrayutilities.materials.lattice module

module handling crystal lattice structures. A Lattice consists of unit cell parameters and a LatticeBase. It offers methods to calculate the reciprocal space position of Bragg peaks and their structure factor.

xrayutilities.materials.lattice.**AlGaAsLattice** (aal, aga, aas, a, x)

xrayutilities.materials.lattice.**BCCLattice** (aa, a)

xrayutilities.materials.lattice.**BCTLattice** (aa, a, c)

xrayutilities.materials.lattice.**BaddeleyiteLattice** (aa, ab, a, b, c, beta)

xrayutilities.materials.lattice.**CsClLattice** (aa, ab, a)

xrayutilities.materials.lattice.**CubicFm3mBaF2** (aa, ab, a)

xrayutilities.materials.lattice.**CubicLattice** (a, base=None)
Returns a Lattice object representing a cubic lattice.

Parameters: **a: lattice parameter**

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**DiamondLattice** (aa, a)

xrayutilities.materials.lattice.**FCCLattice** (aa, a)

xrayutilities.materials.lattice.**FCCSharedLattice** (aa, ab, occa, occb, a)

xrayutilities.materials.lattice.**GeTeRhombohedral** (aa, ab, a, ang, x=0.237)

xrayutilities.materials.lattice.**HCPLattice** (aa, a, c)

xrayutilities.materials.lattice.**Hexagonal3CLattice** (aa, ab, a, c)

xrayutilities.materials.lattice.**Hexagonal4HLattice** (aa, ab, a, c, u=0.1875, v1=0.25, v2=0.4375)

xrayutilities.materials.lattice.**Hexagonal6HLattice** (aa, ab, a, c)

xrayutilities.materials.lattice.**HexagonalLattice** (a, c, base=None)
Returns a Lattice object representing a hexagonal lattice.

Parameters: **a: lattice parameter a**

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

xrayutilities.materials.lattice.**LaB6Lattice** (aa, ab, a, oa=1, ob=1, ba=0, bb=0)

class xrayutilities.materials.lattice.**Lattice** (a1, a2, a3, base=None)
Bases: **object**
class Lattice: This object represents a Bravais lattice. A lattice consists of a base and unit cell defined by three vectors.

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:: Note: NO elastic response of the material will be considered!

requiered input arguments:

eps: a 3x3 matrix independent strain components

GetPoint (*args**)**
determine lattice points with indices given in the argument

Examples

```
>>> 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])
```

ReciprocalLattice ()

UnitCellVolume ()
function to calculate the unit cell volume of a lattice (angstrom^3)

a

a1

a2

a3

alpha

b

beta

c

gamma

class xrayutilities.materials.lattice.**LatticeBase** (*args, **keyargs)
Bases: **list**
The LatticeBase class implements a container for a set of points that form the base of a crystal lattice. An instance of this class can be treated as a simple container object.

append (atom**,** pos**,** occ=1.0**,** b=0.0**)**
add new Atom to the lattice base

Parameters: **atom: atom object to be added**

pos: position of the atom
occ: occupancy (default=1.0)
b: 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)

xrayutilities.materials.lattice.**MagnetiteLattice** (aa, ab, ac, a, x=0.255)

xrayutilities.materials.lattice.**MonoclinicLattice** (a, b, c, beta, base=None)
Returns a Lattice object representing a hexagonal lattice.

Parameters: **a: lattice parameter a**

b: lattice parameter b
c: lattice parameter c
beta: monoclinic unit cell angle beta (deg)
base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

`xrayutilities.materials.lattice.`**`NaumanniteLattice`** (aa, ab, a, b, c)

`xrayutilities.materials.lattice.`**`NiAsLattice`** (aa, ab, a, c, biso=0.0)

`xrayutilities.materials.lattice.`**`OrthorhombicLattice`** (a, b, c, base=None)
Returns a Lattice object representing a tetragonal lattice.

Parameters: **a: lattice parameter a**

b: lattice parameter b

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

`xrayutilities.materials.lattice.`**`PerovskiteTypeRhombohedral`** (aa, ab, ac, a, ang)

`xrayutilities.materials.lattice.`**`QuartzLattice`** (aa, ab, a, b, c)

`xrayutilities.materials.lattice.`**`RockSaltLattice`** (aa, ab, a)
creates the primitive unit cell of a RockSalt structure. For the more commonly used cubic reprentation see RockSalt_Cubic_Lattice

`xrayutilities.materials.lattice.`**`RockSalt_Cubic_Lattice`** (aa, ab, a)

`xrayutilities.materials.lattice.`**`RutileLattice`** (aa, ab, a, c, u)

`xrayutilities.materials.lattice.`**`SiGeLattice`** (asi, age, a, xge)

`xrayutilities.materials.lattice.`**`TetragonalIndiumLattice`** (aa, a, c)

`xrayutilities.materials.lattice.`**`TetragonalLattice`** (a, c, base=None)
Returns a Lattice object representing a tetragonal lattice.

Parameters: **a: lattice parameter a**

c: lattice parameter c

base: instance of LatticeBase, representing the internal structure of the unit cell

Returns: an instance of Lattice class

`xrayutilities.materials.lattice.`**`TetragonalTinLattice`** (aa, a, c)

`xrayutilities.materials.lattice.`**`TriclinicLattice`** (a, b, c, alpha, beta, gamma, base=None)

`xrayutilities.materials.lattice.`**`TrigonalR3mh`** (aa, a, c)

`xrayutilities.materials.lattice.`**`WurtziteLattice`** (aa, ab, a, c, u=0.375, biso=0.0)

`xrayutilities.materials.lattice.`**`ZincBlendeLattice`** (aa, ab, a)

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: **`xrayutilities.materials.material.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** : Miller Indices

sub: substrate material or lattice constant (Instance of Crystal class or float)

exp: Experiment class from which the Transformation object and ndir are needed

Returns: **qy,qz** : reciprocal space coordinates of the corners of the relaxation triangle

static **check_compatibility** (matA, matB)

static **lattice_const_AB** (latA, latB, x, name='')

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: property (lattice parameter/angle) of material A and B.**

A property can be a scalar or vector.

x: fraction of material B in the alloy.

name: label of the property which is interpolated. Can be 'a', 'b', 'c', 'alpha', 'beta', or 'gamma'.

x

class xrayutilities.materials.material.**Amorphous** (name, density, atoms=None, cij=None)

Bases: **xrayutilities.materials.material.Material**

amorphous materials are described by this class

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: x-ray energy eV, if omitted the value from the xrayutilities** configuration is used

Returns: delta (float)

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: x-ray energy eV, if omitted the value from the xrayutilities** configuration is used

Returns: beta (float)

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: string containing the chemical fomula**

Returns: list of tuples with 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.

required input arguments:

cij: (6,6) cij matrix as a numpy array

return value:

cijkl: (3,3,3,3) cijkl 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.

required input arguments:

cijkl: (3,3,3,3) cijkl tensor as numpy array

return value:

cij: (6,6) cij matrix as a numpy array

class `xrayutilities.materials.material.`**`Crystal`** (`name`, `lat`, `cij=None`, `thetaDebye=None`)
Bases: **`xrayutilities.materials.material.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. Note: NO elastic response of the material will be considered!

`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 numpy array with the 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 numpy array with the 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: vectorial momentum transfer (vectors as list,tuple**

or numpy array are valid)

en: energy in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

Returns: 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: vectorial momentum transfer (vectors as list,tuple**

or numpy array are valid)

en: list, tuple or array of energy values in eV

temp: temperature used for Debye-Waller-factor calculation

Returns: 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: 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: energy value in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

Returns: complex valued structure factor array

a

a1

a2

a3

alpha

b

beta

c

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: momentum transfer in (1/A)**

en: xray energy in eV, if omitted the value from the xrayutilities configuration is used

temp: temperature used for Debye-Waller-factor calculation

polarization: either 'S' (default) sigma or 'P' pi polarization

Returns: (abs(chih_real),abs(chih_imag)) complex polarizability

dTheta (Q**,** en='config'**)**
function to calculate the refractive peak shift

Parameters: **Q: momentum transfer (1/A)**

en: x-ray energy (eV), if omitted the value from the xrayutilities configuration is used

Returns: **deltaTheta:** 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: x-ray energy eV, if omitted the value from the xrayutilities**

configuration is used

Returns: delta (float)

density
calculates the mass density of an material from the mass of the atoms in the unit cell.

Returns: 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

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 the unit cell.

Parameters: **pos: list or numpy array with the fractional coordinated in the**

unit cell

keyword arguments:

maxdist: maximum distance wanted in the list of neighbors :(default: 7)

Returns: list of tuples with (distance,atomType,multiplicity) giving distance

(sorted) and type of neighboring atoms together with the amount of atoms at the given distance

classmethod **fromCIF**(ciffilename)
Create a Crystal from a CIF file. The CIF-filename will be used as name of the created Crystal. Note: since the CIF file parser is currently not able to detect the correct space group of the material all materials created by this method will be represented by the P1 space-group!

Parameters: **ciffilename: filename of the CIF file**

Returns: Crystal instance

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: x-ray energy eV, if omitted the value from the xrayutilities**

configuration is used

Returns: beta (float)

planeDistance(*hkl)
determines the lattice plane spacing for the planes specified by (hkl)

Parameters: **h,k,l: Miller indices of the lattice planes given either as**

list,tuple or seperate arguments

Returns: **d:** the lattice plane spacing as float

Examples

```
>>> xu.materials.Si.planeDistance(0,0,4)
1.3577600000000001
```

or

```
>>> xu.materials.Si.planeDistance((1,1,1))
3.1356124059796255
```

class xrayutilities.materials.material.**CubicAlloy** (matA, matB, x)
Bases: **xrayutilities.materials.material.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** : inplane peak position of reflection hkl of

the alloy in reciprocal space

q_perp: perpendicular peak position of the reflection hkl of the alloy in reciprocal space

hkl: Miller indices of the measured asymmetric reflection

sur: Miller indices of the surface (determines the perpendicular direction)

Returns: content, [a_inplane, a_perp, a_bulk_perp(x), eps_inplane, eps_perp]:

the content of B in the alloy determined from the input variables and the lattice constants 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** : perpendicular peak position of the reflection

hkl of the alloy in reciprocal space

hkl: Miller indices of the measured symmetric reflection (also defines the surface normal

inpr: Miller indices of a Bragg peak defining the inplane reference direction

asub: substrate lattice constant

relax: degree of relaxation (needed to obtain the content from symmetric reciprocal space position)

Returns: **content** : the content of B in the alloy determined from the input

variables

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: independent components of the elastic tensor of cubic**

materials

Returns: 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: independent components of the elastic tensor of**

a hexagonal material

Returns: 6x6 matrix with elastic constants

class xrayutilities.materials.material.**Material**(name, cij=None)

Bases: **abc.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.

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: energy of the x-rays (in eV, optional)**

Returns: 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: energy of the x-rays, if omitted the value from the**

xrayutilities configuration is used

deg: return angle in degree if True otherwise radians (default:True)

Returns: Angle of total external reflection

delta(en='config')

abstract method which every implementation of a Material has to override

density

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)

lam

mu

nu

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.

Parameters: **sub: substrate material**

layer: bulk material of the layer

relaxation: degree of relaxation 0: pseudomorphic, 1: relaxed :(default: 0)

trans: 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.

`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: independent components of the elastic tensor of cubic**

materials

Returns: 6x6 matrix with elastic constants

Implementation according to a patch submitted by Julian Stangl

`xrayutilities.materials.material.`**`index_map_ij2ijkl`**`(ij)`

`xrayutilities.materials.material.`**`index_map_ijkl2ij`**`(i, j)`

xrayutilities.materials.predefined_materials module

class `xrayutilities.materials.predefined_materials.`**`AlGaAs`**`(x)`

Bases: **`xrayutilities.materials.material.CubicAlloy`**

class `xrayutilities.materials.predefined_materials.`**`SiGe`**`(x)`

Bases: **`xrayutilities.materials.material.CubicAlloy`**

static **`lattice_const_AB`**`(latA, latB, x, **kwargs)`

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.`**`RangeDict`**

Bases: **`dict`**

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.
this replaces the deprecated Lattice class

`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:: Note: Here you specify the strain and not the stress -> NO elastic

response of the material will be considered!

Parameters: **eps: a 3x3 matrix with all strain components**

`GetHKL`**`(`**`*args`**`)`**

determine the Miller indices of the given reciprocal lattice points

GetPoint (*args)
determine lattice points with indices given in the argument

Examples

```
>>> 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)

a

alpha

b

base ()
generator of atomic position within the unit cell.

beta

c

classmethod **convert_to_P1** (sglat)
create a P1 equivalent of the given SGLattice instance.

Parameters: **sglat: space group lattice instance to be converted to P1.**

Returns: SGLattice instance with the same properties as sglat, however in the P1 setting.

classmethod **fromLattice** (lat, verbose=True)
create a SGLattice from an old Lattice instance. Since the space-group is not known it will always be 1 (triclinic). This is helper routine to make the transition period for users easier. It will be removed in the next major release!

Parameters: **lat: deprecated Lattice instance**

Returns: SGLattice instance with the same properties as lat

gamma

isequivalent (hkl1, hkl2, equalq=False)
primitive way of determining if hkl1 and hkl2 are two crystallographical equivalent pairs of Miller indices

Parameters: **hkl1,2: Miller indices to be checked for equivalence**

equalq: If False the length of the two q-vactors will be compared. If True it is assumed that the length of the q-vectors of hkl1 and hkl2 is equal!

Returns: True or False

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.

append (`atom`**,** `pos`**,** `occ=1.0`**,** `b=0.0`**)**
add new Atom to the lattice base

Parameters: **atom: atom object to be added**

pos: Wyckoff position of the atom, along with its parameters. :Examples: ('2i', (0.1, 0.2, 0.3)), or '1a'

occ: occupancy (default=1.0)

b: 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)

`xrayutilities.materials.spacegrouplattice.`**`get_default_sgrp_suf`** (`sgrp_nr`)
determine default space group suffix

xrayutilities.materials.wyckpos module

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:
a4*z**4 + a3*z**3 + a2*z**2 + a1*z + a0

Returns: tuple of the four (complex) solutions of aboves equation.

[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,y: data coordinates (do NOT need to be regularly spaced)**

data: data set used for fitting (e.g. intensity at the data coords)

start: set of starting parameters for the fit used as first parameter of function fit_function

drange: 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: [xmin,xmax,ymin,ymax]

fit_function: function which should be fitted, must accept the parameters (x,y,*params)

Returns: **(fitparam,cov):** the set of fitted parameters and covariance matrix

`xrayutilities.math.fit.`**`gauss_fit`** (`xdata`, `ydata`, `iparams=[]`, `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: xcoordinates of the data to be fitted**

ydata: ycoordinates of the data which should be fit

keyword parameters:

iparams: initial paramters for the fit, determined automatically if not given

maxit: maximal iteration number of the fit

Returns: params,sd_params,itlim

the Gauss parameters as defined in function Gauss1d(x, *param) and their

errors of the fit, as well as a boolean flag which is false in the case of

a successful fit

`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: data coordinates and values**

Returns: p, rsq: parameters of the linear fit (slope, offest) and the R^2 value

Examples

```
>>> (k, d), R2 = xu.math.linregress(x, y)
```

`xrayutilities.math.fit.`**`multGaussFit`** (*args, **kwargs)

convenience function to keep API stable see multPeakFit for documentation

`xrayutilities.math.fit.`**`multGaussPlot`** (*args, **kwargs)

convenience function to keep API stable see multPeakPlot for documentation

`xrayutilities.math.fit.`**`multPeakFit`** (x, data, peakpos, peakwidth, dranges=None, peaktype='Gaussian')

function to fit multiple Gaussian/Lorentzian peaks with linear background to a set of data

Parameters: **x: x-coordinate of the data**

data: data array with same length as x

peakpos: initial parameters for the peak positions

peakwidth: initial values for the peak width

dranges: 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: type of peaks to be used: can be either 'Gaussian' or 'Lorentzian'

Returns: pos,sigma,amp,background

pos: list of peak positions derived by the fit

sigma: list of peak width derived by the fit

amp: list of amplitudes of the peaks derived by the fit

background: array of background values at positions x

`xrayutilities.math.fit.`**`multPeakPlot`** (x, fpos, fwidth, famp, background, dranges=None, peaktype='Gaussian', fig='xu_plot', fact=1.0)

function to plot multiple Gaussian/Lorentz peaks with background values given by an array

Parameters: **x: x-coordinate of the data**

fpos: list of positions of the peaks
fwidth: list of width of the peaks
famp: list of amplitudes of the peaks
background: array with background values
dranges: list of tuples with (min,max) value of the data ranges to use. does not need to have the same number of entries as fpos
peaktype: type of peaks to be used: can be either 'Gaussian' or 'Lorentzian'
fig: matplotlib figure number or name
fact: factor to use as multiplicator in the plot

`xrayutilities.math.fit.`**`peak_fit`** (`xdata`, `ydata`, `iparams=[]`, `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: xcoordinates of the data to be fitted**

ydata: ycoordinates of the data which should be fit

keyword parameters:

iparams: initial paramters for the fit, determined automatically if not specified
peaktype: type of peak to fit: 'Gauss', 'Lorentz', 'PseudoVoigt', 'PseudoVoigtAsym', 'PseudoVoigtAsym2'
maxit: maximal iteration number of the fit
background: type of background, either 'constant' or 'linear'
plot: 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: returns the fitted function, which takes the independent variables as only argument (f(x))

Returns: params,sd_params,itlim[,fitfunc]

the parameters as defined in function Gauss1d/Lorentz1d/PseudoVoigt1d/

PseudoVoigt1dasym(x, *param). In the case of linear background one more

parameter is included! For every parameter the corresponding errors of the

fit 'sd_params' are returned. A boolean flag 'itlim', which is False in

the case of a successful fit is added by default. Further 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 as needed for the calculation of the thermal Debye-Waller-factor by numerical integration
for definition see: :http: //en.wikipedia.org/wiki/Debye_function
D1(x) = (1/x) int_0^x t/(exp(t)-1) dt

Parameters: **x ... argument of the Debye function (float)**

Returns:

D1(x): float value of the Debye function

`xrayutilities.math.functions.`**`Gauss1d`** (x, *p)

function to calculate a general one dimensional Gaussian

Parameters: **p: list of parameters of the Gaussian**

[XCEN,SIGMA,AMP,BACKGROUND] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

x: coordinate(s) where the function should be evaluated

Returns: 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) >>> Gauss1d(numpy.linspace(0,10,100), 5, 1, 1e3, 0)

`xrayutilities.math.functions.`**`Gauss1dArea`** (*p)
function to calculate the area of a Gauss function with neglected background

Parameters: **p: list of parameters of the Gauss-function**

[XCEN,SIGMA,AMP,BACKGROUND]

Returns: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Gauss-function**

[XCEN,YCEN,SIGMAX,SIGMAY,AMP,ANGLE,BACKGROUND]

Returns: 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: **p: list of parameters of the Gauss-function**

[XCEN,YCEN,ZCEN,SIGMAX,SIGMAY,SIGMAZ,AMP,BACKGROUND] SIGMA = FWHM / (2*sqrt(2*log(2)))

x,y,z: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND]

x: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND]

Returns: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Gaussian**

[XCEN,SIGMA] for information: SIGMA = FWHM / (2*sqrt(2*log(2)))

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: list of parameters of the Lorentzian**

[XCEN,FWHM]

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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 of parameters of the Lorentz-function**

[XCEN,FWHM,AMP,BACKGROUND,ETA] :ETA: 0 ...1 0 means pure Gauss and 1 means pure Lorentz

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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: **p: 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

x: coordinate(s) where the function should be evaluated

Returns: 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: coordinate(s) where the function should be evaluated**

p: 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: 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: **p: 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

x,y: coordinate(s) where the function should be evaluated

Returns: 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: any scalar of ndarray object**

Returns:

s: Heaviside step function evaluated for all values of x

`xrayutilities.math.functions.`**`kill_spike`** (data, threshold=2.0)
function to smooth **single** data points which differ from the average of the neighboring data points by more than the threshold factor. 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: 1d numpy array with experimental data**

threshold: threshold factor to identify strange data points

Returns: 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: coordinate where the function should be evaluated**

args: 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: 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: coordinates where the function should be evaluated**

args: 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: 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: 1D data array**

n: number of data points to average

Returns:

xsmooth: 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: position of the data points**

data: data values

background: type of background, either 'none', 'constant' or 'linear'

full_output: return background cleaned data and background-parameters

Returns: center of mass position (single float)

`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: position of the data points**

data: data values

Returns: fwhm value (single float)

xrayutilities.math.transforms module

class `xrayutilities.math.transforms.`**`AxisToZ`** (newzaxis)
Bases: **`xrayutilities.math.transforms.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.

class `xrayutilities.math.transforms.`**`AxisToZ_keepXY`** (newzaxis)
Bases: **`xrayutilities.math.transforms.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

class `xrayutilities.math.transforms.`**`CoordinateTransform`** (v1, v2, v3)
Bases: **`xrayutilities.math.transforms.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!

class xrayutilities.math.transforms.**Transform**(matrix)
Bases: **object**

inverse(args, rank=1)
performs inverse transformation a vector, matrix or tensor of rank 4

Parameters: **args: 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: rank of the supplied object. allowed values are 1, 2, and 4

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.**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: numpy.array or list of length 3**

axis: numpy.array or list of length 3
ang: rotation angle in degree (deg=True) or in rad (deg=False)
deg: boolean which 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([  6.12323400e-17,   1.00000000e+00,   0.00000000e+00])
```

xrayutilities.math.transforms.**tensorprod**(vec1, vec2)
function implements an elementwise multiplication of two vectors

xrayutilities.math.vector module

module with vector operations for vectors of size 3, since for so short vectors numpy does not give the best performance explicit implementation of the equations is performed together with error checking to ensure vectors of length 3.

xrayutilities.math.vector.**VecAngle**((v1.v2)/(norm(v1)*norm(v2)))

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

optional keyword arguments:

deg: (default: false) return result in degree otherwise in radiants

Returns: float value with the angle inclined by the two vectors

`xrayutilities.math.vector.`**`VecCross`**(`v1`, `v2`, `out=None`)
Calculate the vector cross product.

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

out: optional output vector

Returns: float value

`xrayutilities.math.vector.`**`VecDot`**(`v1`, `v2`)
Calculate the vector dot product.

Parameters: **v1 ..... vector as numpy array or list**

v2: vector as numpy array or list

Returns: float value

`xrayutilities.math.vector.`**`VecNorm`**(`v`)
Calculate the norm of a vector.

Parameters: **v ..... vector as list or numpy array**

Returns: float holding the vector norm

`xrayutilities.math.vector.`**`VecUnit`**(`v`)
Calculate the unit vector of v.

Parameters: **v ..... vector as list or numpy array**

Returns: numpy array with the unit vector

`xrayutilities.math.vector.`**`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: vector of length 3**

Returns: [xyz][+-]

`xrayutilities.math.vector.`**`getVector`**(`string`)
returns unit vector along a rotation axis given in the syntax 'x+' 'z-' or equivalents

Parameters: **string [xyz][+-]**

Returns: vector along the given direction as numpy array

Module contents

xrayutilities.simpack package

Submodules

xrayutilities.simpack.darwin_theory module

class `xrayutilities.simpack.darwin_theory.`**`DarwinModel`**(`qz`, `qx=0`, `qy=0`, `**kwargs`)
Bases: **`xrayutilities.simpack.models.LayerModel`**
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_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: 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: **xrayutilities.simpack.darwin_theory.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 = *5.6532499999999999*

classmethod **abulk**(x)
calculate the bulk (relaxed) lattice parameter of the Al_{x}Ga_{1-x}As alloy

asub = *5.6532499999999999*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors(temp=300)
calculates the needed atomic structure factors

static **poisson_ratio**(x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelAlloy**(qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.DarwinModel**, **abc.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.

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

make_monolayers (s**)**

create monolayer sequence from layer list

Parameters: **s: 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: inplane lattice parameter of the substrate

Returns: 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: monolayer list created by make_monolayer()**

prop: name of the property which should be evaluated. Use 'x' for the chemical composition and 'ai' for the inplane lattice parameter.

Returns: zm, propx: z-position, value of the property prop for every

monolayer. z=0 is the surface

class xrayutilities.simpack.darwin_theory.**DarwinModelGaInAs001** (qz, qx=0, qy=0, **kwargs)

Bases: **xrayutilities.simpack.darwin_theory.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 = *5.6532499999999999*

classmethod **abulk** (x)

calculate the bulk (relaxed) lattice parameter of the Ga_{1-x}In_{x}As alloy

asub = *5.6532499999999999*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors(temp=300**)**
calculates the needed atomic structure factors

static **poisson_ratio** (x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

class xrayutilities.simpack.darwin_theory.**DarwinModelSiGe001** (qz, qx=0, qy=0, **kwargs)
Bases: **xrayutilities.simpack.darwin_theory.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 = *5.4310400000000003*

classmethod **abulk** (x)
calculate the bulk (relaxed) lattice parameter of the alloy

asub = *5.4310400000000003*

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: chemical composition parameter**

apar: 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: relaxation parameter. 1=relaxed, 0=pseudomorphic

Returns: dperp, apar

init_structurefactors(temp=300**)**
calculates the needed atomic structure factors

static **poisson_ratio** (x)
calculate the Poisson ratio of the alloy

re = *2.8179403227e-05*

xrayutilities.simpack.darwin_theory.**GradedBuffer**(xfrom, xto, nsteps, thickness, relaxation=1)
create a multistep graded composition buffer.

Parameters: **xfrom: begin of the composition gradient**

xto: end of the composition gradient
nsteps: number of steps of the gradient
thickness: total thickness of the Buffer in A
relaxation: relaxation of the buffer

Returns: layer object 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

xrayutilities.simpack.fit.**fit_xrr**(reflmod, params, ai, data=None, eps=None, xmin=-inf, xmax=inf, plot=False, verbose=False, elog=True, maxfev=500)
optimize function for a Reflectivity Model using lmfit. The fitting parameters must be specified as instance of lmfits Parameters class.

Parameters: **reflmod: preconfigured SpecularReflectivityModel**

params: instance of lmfits Parameters class. For every layer the parameters '{}_thickness', '{}_roughness', '{}_density', with '{}' representing the layer name are supported. In addition the setup parameters: - 'I0' primary beam intensity - 'background' background added to the simulation - 'sample_width' size of the sample along the beam - 'beam_width' width of the beam in the same units - 'resolution_width' width of the resolution function in deg - 'shift' experimental shift of the incidence angle array
ai: array of incidence angles for the calculation
data: experimental data which should be fitted
eps: (optional) error bar of the data
xmin: minimum value of ai which should be used. a mask is generated to cut away other data
xmax: maximum value of ai which should be used. a mask is generated to cut away other data
plot: 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.
verbose: flag to tell if the variation of the fitting error should be output during the fit.
elog: logarithmic error during the fit
maxfev: maximum number of function evaluations during the leastsq optimization

Returns:

res: MinimizerResult object from lmfit, which contains the fitted parameters in res.params (see res.params.pretty_print) or try lmfit.report_fit(res)

xrayutilities.simpack.helpers module

`xrayutilities.simpack.helpers.`**`coplanar_alphai`** (qx, qz, en='config')
calculate coplanar incidence angle from knowledge of the qx and qz coordinates

Parameters: **qx: inplane momentum transfer component**

qz: out of plane momentum transfer component

en: x-ray energy (eV). By default the value from the config is used.

Returns: the incidence angle in degree. points in the Laue zone are set to 'nan'.

`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: inplane momentum transfer component**

alphai: incidence angle (deg)

en: x-ray energy (eV). By default the value from the config is used.

Returns: the qz position for the given incidence angle

xrayutilities.simpack.models module

class `xrayutilities.simpack.models.`**`DynamicalModel`**(*args, **kwargs)
Bases: **`xrayutilities.simpack.models.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: vector of incidence angles (deg)**

hkl: Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry: 'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

rettype: 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: vector of intensities of the diffracted signal

class `xrayutilities.simpack.models.`**`KinematicalModel`**(*args, **kwargs)
Bases: **`xrayutilities.simpack.models.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_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: simulation positions along qz**

hkl: Miller indices of the Bragg peak whos truncation rod should be calculated

absorption: flag to tell if absorption correction should be used

refraction: 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: 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: vector of the ratios of the diffracted and primary fluxes

class `xrayutilities.simpack.models.`**`KinematicalMultiBeamModel`**`(*args, **kwargs)`

Bases: **`xrayutilities.simpack.models.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.

`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: simulation positions along qz**

hkl: Miller indices of the Bragg peak whos truncation rod should be calculated

absorption: flag to tell if absorption correction should be used

refraction: 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: 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: vector of the ratios of the diffracted and primary fluxes

class `xrayutilities.simpack.models.`**`LayerModel`**`(*args, **kwargs)`

Bases: **`xrayutilities.simpack.models.Model`**, **`abc.ABC`**

generic model class from which further thin film models can be derived from

`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.

`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

convolute_resolution(x, y**)**
convolve simulation result with a resolution function

Parameters: **x: x-values of the simulation, units of x also decide about the**

unit of the resolution_width parameter

y: y-values of the simulation

Returns: convoluted y-data with same shape as y

scale_simulation(y**)**
scale simulation result with primary beam flux/intensity and add a background.

Parameters: **y: y-values of the simulation**

Returns: scaled y values

class xrayutilities.simpack.models.**SimpleDynamicalCoplanarModel**(*args, **kwargs)
Bases: **xrayutilities.simpack.models.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

Note: This model should not be used in real life scenarios since the made approximations severely fail for distances far from the reference position.

set_hkl(*hkl**)**
To speed up future calculations of the same Bragg peak optical parameters can be pre-calculated using this function.

Parameters: **hkl: 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: vector of incidence angles (deg)**

hkl: Miller indices of the diffraction vector (preferable use set_hkl method to speed up repeated calculations of the same peak!)

geometry: 'hi_lo' for grazing exit (default) and 'lo_hi' for grazing incidence

idxref: 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: vector of intensities of the diffracted signal

class xrayutilities.simpack.models.**SpecularReflectivityModel**(*args, **kwargs)
Bases: **xrayutilities.simpack.models.LayerModel**
model for specular reflectivity calculations

densityprofile(nz, plot=False**)**
calculates the electron density of the layerstack from the thickness and roughness of the individual layers

Parameters: **nz: number of values on which the profile should be calculated**

plot: flag to tell if a plot of the profile should be created

Returns: z, eprof: coordinates and electron profile. z = 0 corresponds to the surface

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: vector of incidence angles**
Returns: vector of intensities of the reflectivity signal

xrayutilities.simpack.models.**startdelta** (start, delta, num)

xrayutilities.simpack.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.simpack.mpl_helper.**SqrtAllowNegScale** (axis, **kwargs)
Bases: **matplotlib.scale.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: **matplotlib.transforms.Transform**

input_dims = *1*

inverted ()

is_separable = *True*

output_dims = *1*

transform_non_affine (a**)**

class SqrtAllowNegScale.**SqrtTransform** (shorthand_name=None)
Bases: **matplotlib.transforms.Transform**

input_dims = *1*

inverted ()
return the inverse transform for this transform.

is_separable = *True*

output_dims = *1*

transform_non_affine (a**)**
This transform takes an Nx1 numpy array and returns a transformed copy.

SqrtAllowNegScale.**get_transform ()**

SqrtAllowNegScale.**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.

SqrtAllowNegScale.**name** = *'sqrt'*

SqrtAllowNegScale.**set_default_locators_and_formatters**(axis)

class xrayutilities.simpack.mpl_helper.**SqrtTickLocator**(nbins=7, symmetric=True)
Bases: **matplotlib.ticker.Locator**

set_params(nbins, symmetric)
Set parameters within this locator.

tick_values(vmin, vmax)

view_limits(dmin, dmax)
Set the view limits to the nearest multiples of base that contain the data

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)
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.

Noteable class parameters:

max_history_length:	the number of histories to cache (default=5); can be overridden if memory is an issue.
length_scale_m:	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.

add_buffer(b)
add a numpy array to the list of objects that can be thrown away on pickling.

Parameters: **b: the buffer to add to the list**

Returns: 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: the edge of the function farthest from the singularity,**

referenced to epsvals

innerbound: the edge closest to the singularity, referenced to epsvals

epsvals: the array of two-theta values or offsets

destination: an array into which final results are summed. modified in place!

peakpos: the position of the singularity, referenced to epsvals.

y0: the constant offset

k: the scale factor

Returns: (*lower_index*, *upper_index*) 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: a list of convolvers to select. If *None*, use all**

found convolvers.

compute_derivative: if *True*, also return d/dx(function) for peak position fitting

Returns: a profile_data object with much information about the peak

conv_absorption ()
compute the sample transparency correction, including the finite-thickness version

Returns: the convolver

conv_axial ()
compute the Fourier transform of the axial divergence component

Returns: 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: the convolver

conv_emission ()
compute the emission spectrum and (for convenience) the particle size widths

Returns: the convolver for the emission and particle sizes

Note

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: 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

conv_receiver_slit ()
compute the rectangular convolution for the receiver slit or SiPSD pixel size

Returns: the convolver

conv_si_psd ()
compute the convolver for the integral of defocusing of the face of an Si PSD

Returns: the convolver

conv_smoother ()
compute the convolver to smooth the final result with a Gaussian before downsampling.

Returns: the convolver

conv_tube_tails ()
compute the Fourier transform of the rectangular tube tails function

Returns: 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: length of the xray filament**

Ls: length of the sample
Lr: length of the receiver slit
R: diffractometer length, assumed symmetrical
twotheta: angle, in radians, of the center of the computation
beta: offset angle
epsvals: array of offsets from center of computation, in radians

Returns: (*epsvals*, *idxmin*, *idxmax*, *I2p*, *I2m*).

idxmin and *idxmax* are the full python-style bounds of the non-zero region of *I2p* and *I2m*. *I2p* and *I2m* are 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: length of the xray filament**

Ls: length of the sample
Lr: length of the receiver slit
R: the (assumed symmetrical) diffractometer radius
twotheta: angle, in radians, of the center of the computation
epsvals: array of offsets from center of computation, in radians
sollerIdeg: the full-width (both sides) cutoff angle of the incident Soller slit
sollerDdeg: the full-width (both sides) cutoff angle of the detector Soller slit
nsteps: the number of subdivisions for the integral
axDiv: not used

Returns: 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: the name of the convolver cache buffer to update**

width: the width in 2-theta space of the tophat

Returns: the updated convolver buffer, or *None* if the width was *None*

get_conv (`name, key, format=<type '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: the name of the convolver to seek**

key: any hashable object which identifies the parameters for the computation
format: the type of the array to create, if one is not found.

Returns: 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 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: 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: a number of bins.**

Returns: a bin count somewhat larger than *count* which is efficient for FFT

info_emission = *{'param_info': {'emiss_lor_widths': ('Lorenztian emission fwhm (m)', (1e-13,)), 'crystallite_size_lor': ('Lorentzian crystallite size fwhm (m)', 1e-06), 'emiss_wavelengths': ('wavelengths (m)', (1.58e-10,)), 'emiss_intensities': ('relative intensities', (1.0,)), 'emiss_gauss_widths': ('Gaussian emissions fwhm*

(m)', (1e-13,)), 'crystallite_size_gauss': ('Gaussian crystallite size fwhm (m)', 1e-06)}, 'help': 'this should be help information', 'group_name': 'Incident beam and crystal size'}

info_global = *{'param_info': {'twotheta0_deg': ('Bragg center of peak (degrees)', 30.0), 'd': ('d spacing (m)', 4e-10), 'dominant_wavelength': ('wavelength of most intense line (m)', 1.5e-10)}, 'help': 'this should be help information', 'group_name': 'Global parameters'}*

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 calles when the class attribute 'isotropic' is False.

Parameters: **hkl1,2: Miller indices to be checked for equivalence**

crystalsystem: symmetry class of the material which is considered

Returns: True or False

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: exact position of center bin, in degrees**

twotheta_approx_window_fullwidth_deg: approximate desired width

twotheta_exact_bin_spacing_deg: the exact bin spacing to use

set_parameters (`convolver='global'`**,** `**kwargs`**)**
update the dictionary of parameters associated with the given convolver

Parameters: **convolver: the name of the convolver. name 'global', e.g., attaches**

to function 'conv_global'

kwargs: 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: the center position of the middle bin of**

the window, in degrees

twotheta_window_fullwidth_deg: the full width of the window, in degrees

twotheta_output_points: the number of bins in the final output

str_emission ()
format the emission spectrum and crystal size information

Returns: the formatted information

str_global ()
returns a string representation for the global context.

Returns: report on global parameters.

class xrayutilities.simpack.powder.**PowderDiffraction** (mat, **kwargs)
Bases: **xrayutilities.experiment.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 used 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: two theta values at which the powder pattern should be**

calculated.

Note

Note: Bragg peaks are only included up to tt_cutoff set in
the class constructor!

****kwargs: additional keyword arguments are passed to the Convolve**
function

Returns: output intensity values for the twotheta values given in the input

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: two theta values at which the powder pattern should be**

calculated.

window_width: width of the calculation window of a single peak

mode: 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

correction_factor (ang)
calculate the correction factor for the diffracted intensities. This contains the polarization effects and the Lorentz factor

Parameters: **ang: theta diffraction angles for which the correction should be**
calculated

Returns:
f: array of the same shape as ang containing the correction factors

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 if isotropic lines at the same q-position can be merged to one line to reduce the calculational effort

Parameters: **data: numpy field array with values of 'hkl' (Miller indices of the**
peaks), 'q' (q-position), and 'r' (reflection strength) as produced by the structure_factors method

Returns: hkl, q, ang, r: Miller indices, q-position, diffraction angle (Theta),
and reflection strength of the material

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

structure_factors (tt_cutoff)
determine structure factors/reflection strength of all Bragg peaks up to tt_cutoff

Parameters: **tt_cutoff: upper cutoff value of 2theta until which the reflection**
strength are calculated

Returns: numpy array with field for 'hkl' (Miller indices of the peaks),
'q' (q-position), and 'r' (reflection strength) of the Bragg peaks

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={})
update settings of all instances of FP_profile

Parameters: **newsettings: dictionary with new settings. It has to include one**

subdictionary for every convolver which should have its settings changed.

wavelength

window_width

xrayutilities.simpack.powder.**chunkify**(lst, n)

class xrayutilities.simpack.powder.**convolver_handler**
Bases: **object**
manage the convolvers of on process

add_convolver (convolver)

calc (run, ttpeaks)
calculate profile function for selected convolvers

Parameters: **run: list of flags of length of convolvers to tell which convolver**

needs to be run

ttpeaks: peak positions for the convolvers

Returns: 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')
Bases: **multiprocessing.managers.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

add_symbol (**kwargs)
add new symbols to both the attributes and dictionary for the class

Parameters: **kwargs 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.

create_fitparameters ()
function to create a fit model with all instrument and sample parameters.

Parameters: **pass**

Returns: lmfit Parameters instance

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: angular values for the fit

data: experimental intensities for the fit

std: standard deviation of the experimental data. if 'None" the sqrt of the data will be used

maxfev: maximal number of simulations during the least squares refinement

Returns: lmfit MinimizerResult

set_background (`btype`**,** `**kwargs`**)**
define background as spline or polynomial function

Parameters: **btype: background type: either 'polynomial' or 'spline'. Depending on**

this value the expected keyword arguments differ.

kwargs: **'spline':**

x: x-values (twotheta) of the background points

y: intensity values of the background

'polynomial':

p: polynomial coefficients from the highest degree to the constant term. len of p decides about the degree of the polynomial

set_lmfit_parameters (`lmparams`**)**
function to update the settings of this class during an least squares fit

Parameters: **lmparams: lmfit Parameters list of sample and instrument parameters**

set_parameters (`params`**)**
set simulation parameters of all subobjects

Parameters: **params: 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: positions at which the powder spectrum should be evaluated**

****kwargs:**

background: 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: summed powder diffraction intensity of all materials present in the model

Known issue: possibility to add a background is currently missing!

`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: experimental datapoints**

sim: simulated data

weight: weight factor in the least squares sum. If it is None the weight is estimated from the counting statistics of 'exp'

std: standard deviation of the experimental data. alternative way of specifying the weight factor. when both are given weight overwrites std!

Nvar: number of variables in the refinement

disp: flag to tell if a line with the calculated values should be printed.

Returns: M, Rp, Rwp, Rwpexp, chi2

`xrayutilities.simpack.powdermodel.`**`plot_powder`**(`twotheta`, `exp`, `sim`, `mask=None`, `scale='sqrt'`, `fig='XU:powder'`, `show_diff=True`, `show_legend=True`)

Convenience function to plot the comparison between experimental and simulated powder diffraction data

Parameters: **twotheta: angle values used for the x-axis of the plot (deg)**

exp: experimental data (same shape as twotheta). If None only the simulation and no difference will be plotted

sim: simulated data

mask: mask to reduce the twotheta values to the be used as x-coordinates of sim

scale: string specifying the scale of the y-axis. Valid are: 'linear, 'sqrt', and 'log'.

fig: matplotlib figure name (figure will be cleared!)

show_diff: flag to specify if a difference curve should be shown

show_legend: flag to specify if a legend should be shown

xrayutilities.simpack.smaterials module

class `xrayutilities.simpack.smaterials.`**`CrystalStack`**(`name`, `*args`)

Bases: **`xrayutilities.simpack.smaterials.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: **`xrayutilities.simpack.smaterials.CrystalStack`**

generates a sequence of layers with a gradient in chemical composition

class `xrayutilities.simpack.smaterials.`**`Layer`**(`material`, `thickness`, `**kwargs`)

Bases: **`xrayutilities.simpack.smaterials.SMaterial`**

Object describing part of a thin film sample. The properties of a layer :are:

Material: an xrayutilties material describing optical and crystal properties of the thin film

Thickness: film thickness in Angstrom

Roughness: root mean square roughness of the top interface in Angstrom

class xrayutilities.simpack.smaterials.**LayerStack**(name, *args)
Bases: **xrayutilities.simpack.smaterials.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: **_abcoll.MutableSequence**
class representing the basics of a list of materials for simulations within xrayutilities. It extends the built in list type.

check(v)

insert(i, v)

class xrayutilities.simpack.smaterials.**Powder**(material, volume, **kwargs)
Bases: **xrayutilities.simpack.smaterials.SMaterial**
Object describing part of a powder sample. The properties of a powder :are:

Material: an xrayutilties material (Crystal) describing optical and crystal properties of the thin film
Volume: powder's volume (in pseudo units, since only the relative volume enters the calculation)

Optionally also the following can be set:

crystallite_size_lor: Lorentzian crystallite size fwhm (m)
crystallite_size_gauss: Gaussian crystallite size fwhm (m)
strain_lor: extra peak width proportional to tan(theta)
strain_gauss: extra peak width proportional to tan(theta)

class xrayutilities.simpack.smaterials.**PowderList**(name, *args)
Bases: **xrayutilities.simpack.smaterials.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: **xrayutilities.simpack.smaterials.CrystalStack**
generate a sequence of pseudomorphic crystalline Layers. Surface orientation is assumed to be 001 and materials must be cubic/tetragonal.

insert(i, v)

make_epitaxial(i)

trans = <*xrayutilities.math.transforms.Transform object*>

class xrayutilities.simpack.smaterials.**PseudomorphicStack111**(name, *args)
Bases: **xrayutilities.simpack.smaterials.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, **kwargs)
Bases: **object**
Simulation Material. Extends the xrayutilities Materials by properties needed for simulations

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`** (`msg`)

Bases: **`exceptions.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)

Attibutes

expr -- input expression in which the error occurred :msg: -- explanation of the error

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, Powder

`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: ****kwargs: optional keyword arguments**

B: reciprocal space conversion matrix of a Crystal. You can specify the matrix B (default identiy matrix) shape needs to be (3,3)

mat: 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: orientation matrix U can be given. If none is given the orientation defined in the Experiment class is used.

dettype: detector type: one of ('point', 'linear', 'area') decides which routine of Ang2Q to call. default 'point'

delta: 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: x-ray wavelength in angstroem (default: self._wl)

en: x-ray energy in eV (default: converted self._wl)

deg: flag to tell if angles are passed as degree (default: True)

sampledis: sample displacement vector in relative units of the detector distance (default: (0, 0, 0))

Returns: H K L coordinates as numpy.ndarray with shape (* , 3)

where * 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

energy

wavelength

class xrayutilities.experiment.**GID** (idir, ndir, **keyargs)

Bases: **xrayutilities.experiment.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 self.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: 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: optional keyword arguments**

delta: 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 ai,phi,tt,beta - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * corresponds to the number of points given in the input

Q2Ang (`Q`**,** `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)

Parameters: **Q: a list or numpy array of shape (3) with**

q-space vector components

optional keyword arguments:

trans: True/False apply coordinate transformation on Q

deg: True/Flase (default True) determines if the angles are returned in radians or degrees

Returns: a numpy array of shape (4) with the 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)

class `xrayutilities.experiment.`**`GISAXS`** (`idir`, `ndir`, `**keyargs`)

Bases: **`xrayutilities.experiment.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: 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: optional keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of length 3. Used angles are than ai,tt,beta - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * corresponds to the number of points given in the input

Q2Ang (`Q`**,** `trans=True`**,** `deg=True`**,** `**kwargs`**)**

class `xrayutilities.experiment.`**`HXRD`**(`idir`, `ndir`, `geometry='hi_lo'`, `**keyargs`)

Bases: **`xrayutilities.experiment.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 self.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: 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: optional keyword arguments**

delta: 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: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * 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.

Parameters: **Q: a list, tuple or numpy array of shape (3) with**

q-space vector components or 3 separate lists with qx,qy,qz

optional keyword arguments:

trans: True/False apply coordinate transformation on Q (default True)

deg: True/Flase (default True) determines if the angles are returned in radians or degrees

geometry: determines the scattering 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]

default: self.geometry

refrac: boolean to determine if refraction is taken into account :default: False if True then also a material must be given

mat: Crystal object; needed to obtain its optical properties for refraction correction, otherwise not used

full_output: boolean to determine if additional output is given to determine scattering angles more accurately in case refraction is set to True. default: False

fi,fd: 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: 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

if full_output:

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

class `xrayutilities.experiment.`**`NonCOP`**(`idir`, `ndir`, `**keyargs`)

Bases: **`xrayutilities.experiment.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,twotheta) goniometer to help with x-ray diffraction experiments. Linear and area detectors can be treated as described in "help self.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: 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: optional keyword arguments**

delta: 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: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) :(default: identity matrix)

wl: x-ray wavelength in angstroem (default: self._wl)

deg: flag to tell if angles are passed as degree (default: True)

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * 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.

Parameters: **Q: a list, tuple or numpy array of shape (3) with**

q-space vector components or 3 separate lists with qx,qy,qz

optional keyword arguments:

trans: True/False apply coordinate transformation on Q (default True)

deg: True/Flase (default True) determines if the angles are returned in radians or degree

Returns: a numpy array of shape (4) with four scattering angles which are

[omega, chi, phi, twotheta]

omega: sample rocking angle

chi: sample tilt

phi: sample azimuth

twotheta: scattering angle (detector)

class xrayutilities.experiment.**PowderExperiment** (**kwargs)
Bases: **xrayutilities.experiment.Experiment**
Experimental class for powder diffraction which helps to convert theta angles to momentum transfer space

Q2Ang (qpos**,** deg=True**)**
Converts reciprocal space values to theta angles

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

UB

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: sample and detector angles as numpy array, lists or**

Scalars 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: possible keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of len(*args). Used angles are than *args - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample. Used to determine not Q but (hkl) (default: self.UB)

roi: region of interest for the detector pixels; e.g. [100, 900, 200, 800] (default: self._area_roi)

Nav: number of channels to average to reduce data size e.g. [2, 2] (default: self._area_nav)

wl: x-ray wavelength in angstroem (default: self._wl)

en: 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: flag to tell if angles are passed as degree (default: True)

sampledis: sample displacement vector in same units as the detector distance (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

detectorAxis

property handler for _detectorAxis

Returns: list of detector axis following the syntax /[xyz][+-]/

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: detector angles. Only detector arm angles as described by the**

detectorAxis attribute must be given.

****kwargs: optional keyword arguments**

dim: dimension of the detector for which the position should be determined

roi: region of interest for the detector pixels; :(default: self._area_roi/self._linear_roi)

Nav: number of channels to average to reduce data size; :(default: self._area_nav/self._linear_nav)

deg: flag to tell if angles are passed as degree (default: True)

Returns: numpy array with the detector distance

getDetectorPos(*args*, ***kwargs*)
obtains the detector position vector by applying the detector arm rotations.

Parameters: ***args: detector angles. Only detector arm angles as described by the**

detectorAxis attribute must be given.

****kwargs: optional keyword arguments**

dim: dimension of the detector for which the position should be determined

roi: region of interest for the detector pixels; :(default: self._area_roi/self._linear_roi)

Nav: number of channels to average to reduce data size; :(default: self._area_nav/self._linear_nav)

deg: flag to tell if angles are passed as degree (default: True)

Returns: 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: direction of the detector (along the pixel**

direction 1); e.g. 'z+' means higher pixel numbers at larger z positions

detectorDir2: direction of the detector (along the pixel direction 2); e.g. 'x+'

cch1,2: center pixel, in direction of self.r_i at zero detectorAngles

Nch1: number of detector pixels along direction 1

Nch2: number of detector pixels along direction 2

distance: distance of center pixel from center of rotation

pwidth1,2: width of one pixel (same unit as distance)

chpdeg1,2: channels per degree (only absolute value is relevant) sign determined through detectorDir1,2

detrot: angle of the detector rotation around primary beam direction (used to correct misalignments)

tiltazimuth: direction of the tilt vector in the detector plane (in degree)

tilt: tilt of the detector plane around an axis normal to the direction given by the tiltazimuth

Note

Note: Either distance and pwidth1,2 or chpdeg1,2 must be given !!

Note

Note: the channel numbers run from 0 .. NchX-1
****kwargs: optional keyword arguments**

Nav: number of channels to average to reduce data size :(default: [1, 1])
roi: region of interest for the detector pixels; e.g. [100, 900, 200, 800]

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: direction of the detector (along the pixel array);**

e.g. 'z+'

cch: center channel, in direction of self.r_i at zero detectorAngles
Nchannel: total number of detector channels
distance: distance of center channel from center of rotation
pixelwidth: width of one pixel (same unit as distance)
chpdeg: channels per degree (only absolute value is relevant) sign determined through detectorDir

!! Either distance and pixelwidth or chpdeg must be given !!

tilt: tilt of the detector axis from the detectorDir (in degree)

Note

Note: the channel numbers run from 0 .. Nchannel-1
****kwargs: optional keyword arguments**

Nav: number of channels to average to reduce data size :(default: 1)
roi: region of interest for the detector pixels; e.g. [100,900]

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: sample and detector angles as numpy array, lists or**

Scalars 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: possible keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of len(*args) used angles are than *args - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

Nav: number of channels to average to reduce data size :(default: self._linear_nav)

roi: region of interest for the detector pixels; e.g. [100,900] (default: self._linear_roi)

wl: x-ray wavelength in angstroem (default: self._wl)

en: 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: flag to tell if angles are passed as degree (default: True)

sampledis: sample displacement vector in same units as the detector distance (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: sample and detector angles as numpy array, lists**

or Scalars 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: optional keyword arguments**

delta: giving delta angles to correct the given ones for misalignment delta must be an numpy array or list of len(*args) used angles are than *args - delta

UB: matrix for conversion from (hkl) coordinates to Q of sample used to determine not Q but (hkl) (default: self.UB)

wl: x-ray wavelength in angstroem (default: self._wl)

en: 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: flag to tell if angles are passed as degree :(default: True)

sampledis: sample displacement vector in relative units of the detector distance (default: (0,0,0))

Returns: reciprocal space positions as numpy.ndarray with shape (* , 3)

where * corresponds to the number of points given in the input

sampleAxis

property handler for _sampleAxis

Returns: list of sample axis 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: vector to transform (sequence, list, numpy array)**

args: 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: rotated vector as numpy.array

wavelength

xrayutilities.gridder module

class xrayutilities.gridder.**FuzzyGridder1D**(nx)

Bases: **xrayutilities.gridder.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).

class xrayutilities.gridder.**Gridder**
Bases: **abc.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

data
return gridded data (performs normalization if switched on)

class xrayutilities.gridder.**Gridder1D** (nx)
Bases: **xrayutilities.gridder.Gridder**

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: minimum value of the gridding range**

max: maximum value of the gridding range
fixed: 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: output filename**

header: optional header for the data file.

xaxis
Returns the xaxis of the gridder the returned values correspond to the center of the data bins used by the gridding algorithm

xrayutilities.gridder.**axis** (min_value, max_value, n)
Compute the a grid axis.

Parameters: **min_value ..... axis minimum value**

max_value ..... axis maximum value

n ..... 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: **xrayutilities.gridder.Gridder1D**

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: **xrayutilities.gridder2d.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.

class xrayutilities.gridder2d.**Gridder2D**(nx, ny)
Bases: **xrayutilities.gridder.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 ..... number of points in x-direction**

ny ..... number of points in y-direction

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: minimum value of the gridding range in x,y**

xmax,ymax: maximum value of the gridding range in x,y

fixed: 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: output filename**

header: optional header for the data file.

xaxis

xmatrix

yaxis

ymatrix

class xrayutilities.gridder2d.**Gridder2DList** (nx, ny)
Bases: **xrayutilities.gridder2d.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 ()

data
return gridded data, in this special version no normalization is defined!

xrayutilities.gridder3d module

class xrayutilities.gridder3d.**FuzzyGridder3D** (nx, ny, nz)
Bases: **xrayutilities.gridder3d.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.

class xrayutilities.gridder3d.**Gridder3D** (nx, ny, nz)
Bases: **xrayutilities.gridder.Gridder**

SetResolution (nx**,** ny**,** nz**)**

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: minimum value of the gridding range in x,y,z**

xmax,ymax,zmax: maximum value of the gridding range in x,y,z

fixed: flag to turn fixed range gridding on (True (default)) or off (False)

xaxis

xmatrix

yaxis

ymatrix

zaxis

zmatrix

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

absfun
absfun property handler
returns the costum correction function or None

avmon
av_mon property handler
returns the value of the average monitor or None if average is calculated from the monitor field

darkfield
flatfield property handler
returns the current set darkfield of the detector or None if not set

det
det property handler
returns the detector field name

flatfield
flatfield property handler
returns the current set flatfield of the detector or None if not set

mon
mon property handler
returns the monitor field name or None if not set

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: data which should be contracted (length N)**

Nav: number of values which should be averaged

Returns: 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: array of 2D data shape (N,M)**

Nav1,2: a field of (Nav1 x Nav2) values is contracted

****kwargs: optional keyword argument**

roi: region of interest for the 2D array. e.g. [20,980,40,960] N = 980-20; M = 960-40

Returns: block averaged numpy array with type numpy.double with shape

(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: array of 2D data shape (Nspectra,Nchannels)**

Nav: number of channels which should be averaged

****kwargs: optional keyword argument**

roi: region of interest for the 2D array. e.g. [20,980] Nchannels = 980-20

Returns: 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=())`

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: q-vector for which the angular positions should be calculated**

expclass: experimental class used to define the goniometer for which the angles should be calculated.

keyword arguments(optional):

bounds: list of 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: orientation matrix of the sample to be used in the conversion

startvalues: 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: sequence of constraint dictionaries. This allows applying arbitrary (e.g. pseudo-angle) contraints by supplying according constraint functions. (see scipy.optimize.minimize). The supplied function will be called with the arguments (angles, qvec, Experiment, U).

Returns: fittedangles, qerror, errcode:

list of fitted goniometer angles, the error in reciprocal space and the errcode of the scipy minimize function. for a successful fit the error code should be <=2

`xrayutilities.q2ang_fit.`**`exitAngleConst`**`(angles, alphaf, hxrd)`

helper function for an pseudo-angle constraint for the Q2AngFit-routine.

Parameters: **angles: fit parameters of Q2AngFit**

alphaf: the exit angle which should be fixed
hxrd: 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.`**`import_lmfit`** (`funcname='XU'`)
lazy import function for lmfit

`xrayutilities.utilities.`**`import_matplotlib_pyplot`** (`funcname='XU'`)
lazy import function of matplotlib.pyplot

Parameters: **funcname: identification string of the calling function**
Returns: flag, pyplot: the flag is True if the loading was successful and False otherwise. On success pyplot is the matplotlib.pyplot package.

`xrayutilities.utilities.`**`maplog`** (`inte`, `dynlow='config'`, `dynhigh='config'`, `**keyargs`)
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** : numpy.array, values to be cut in range

dynlow: 10^(-dynlow) will be the minimum cut off
dynhigh: 10^(-dynhigh) will be the maximum cut off

optional keyword arguments (NOT IMPLEMENTED):

abslow: 10^(abslow) will be taken as lower boundary
abshigh: 10^(abshigh) will be taken as higher boundary

Returns: numpy.array of the same shape as inte, where values smaller/larger then 10^(-dynlow,dynhigh) were replaced by 10^(-dynlow,dynhigh)

Examples

```
>>> lint = maplog(int,5,2)
```

xrayutilities.utilities_noconf module

xrayutilities utilities contains a conglomeration of useful functions this part of utilities does not need the config class

`xrayutilities.utilities_noconf.`**`clear_bit`** (`f`, `offset`)
clears the bet at an offset

`xrayutilities.utilities_noconf.`**`en2lam`** (`inp`)
converts the input energy in eV to a wavelength in Angstrom

Parameters: **inp** : 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: energy either as scalar or array with value in eV, which**

will be returned unchanged; or string with name of emission line

Returns: energy in eV as float

`xrayutilities.utilities_noconf.`**`exchange_filepath`**`(orig, new, keep=0)`
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)

Parameters: **orig: original filename which should have its data root replaced**

new: new path which should be used instead

keep: (optional) number of inner most directory names which should be kept the same in the output (default = 0). Note that the filename is always return unchanged also with keep=0.

Returns: filename string

Examples

```
>>> exchange_filepath('/dir_a/subdir/sam/file.txt', '/data', 1)
'/data/sam/file.txt'
```

`xrayutilities.utilities_noconf.`**`exchange_path`**`(orig, new, keep=0)`
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)

Parameters: **orig: original path which should be replaced by the new path**

new: new path which should be used instead

keep: (optional) number of inner most directory names which should be kept the same in the output (default = 0)

Returns: directory path string

Examples

```
>>> exchange_path('/dir_a/subdir/img/sam', '/home/user/data', 2)
'/home/user/data/img/sam'
```

`xrayutilities.utilities_noconf.`**`lam2en`**`(inp)`
converts the input wavelength in Angstrom to an energy in eV

Parameters: **inp** : wavelength in Angstrom

Returns: float, energy in eV

Examples

```
>>> energy = lam2en(1.5406)
```

`xrayutilities.utilities_noconf.`**`set_bit`**`(f, offset)`
sets the bit at an offset

`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: wavelength; If scalar or array the wavelength in Angstrom will be**

returned unchanged, string with emission name is converted to wavelength

Returns: wavelength in Angstrom as float

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

F

G

L

M

N

O

P

Q

R

S

Y

Z

Python Module Index

x