XRSTools user manual

Ch.J. Sahle*
ESRF - The European Synchrotron Radiation Facility, Grenoble, France.
(Dated: January 16, 2020)

This is a general and more technical introduction to the Python package XRSTools, a collection of Python modules for non-resonant inelastic x-ray scattering. XRSTools, in its current state, is an experimental code but has now been used during and after a number of experiments to extract publication grade data.

Contents

INTRODUCTION

Welcome to the XRSTools program package for non-resonant inelastic x-ray scattering. XRSTools is a collection of Python moldules (classes and functions) written to aid the planning and performing of synchrotron based NIXS experiments and analyze data from such experiments. Part of the functions used in the package are based on Matlab cade written at the Helixs-Group of the University of Helsinki, part of it is based on Matlab code written at the ESRF, and some of it is new. The aim of the code is to simplify and standardize experimental procedures, especially when it comes to the use of the new multi-analyzer beamlines equipped with 2D detectors.

As this is written, the code is adapted solely to the ESRF beamline ID20, but it should be straightforward to extend it to support also data from other beamlines. In fact, there is an experimental version for the PetraIII beamline P01.

The ideas and physics behind this code are summarized in [1]. If you used the package or functions from it, please cite this paper. Besides this paper on the package, there are a few other published items out there, to which I would refer anyone to who would like to dig deeper into inelastic x-ray scattering: 1. The book of W. Schülke is probably the most complete reference guide on this subject [2]. 2. A few review papers [3, 4]. 3. And the papers on which most of the 'background' subtraction by XRSTools is based on [6, 7]. I would also like to point out the existing experimental endstations by citing their beamline papers [8–12].

If you are not familiar with the Python programming language at all, maybe it is a good idea to have a look at some of the tutorials there are online (e.g. at `https://docs.python.org/2/tutorial/`). A book that has helped me a lot is this this one [5].

To finish this introductory section, let's remind ourselves that this code is to be used at your own risk, it is, no doubt, not without bugs. If you find one, please let me know about it so we can try to fix it. So let's dig into the code.

INSTALLATION

XRSTools comes as a Python package, i.e. as a *.tar.gz* file called *xrstools-¡version¿.tar.gz*. Extract this file on your disk and change into the created directory:

```
tar -xzf xrstools-<version>.tar.gz
cd xrstools-<version>
```

For a global installation on your system type:

```
sudo python setup.py install
```

a local installation into a directory `<dir>` can be made by

```
python setup.py install --home=<dir>
```

You can find more information about how to install Python packages at `https://docs.python.org/2/install/`. If you do a local installation, do not forget to set your PYTHONPATH variable

```
export PYTHONPATH=$PYTHONPATH:/path/to/XRSTools
```

or add it to your bashrc.

PLANNING

using an input file

When planning XRS experiments, there are usually two things to consider: first, which momentum transfer(s) to measure at and, the second question is, if there will be enough counts from the sample such that it you can take statistically meaningful data in a reasonable amount of beam time.

XRSTools has a module, called `xrs_prediction`, to do that. Since we need quite a few variables for this prediction of intensities, `xrs_prediction` can read an input file and returns a graph of the estimated cross section. An example of an input file is given in appendix ().

The expected intensity is given by

$$I = I_0 \frac{\mathrm{d}^2\sigma}{\mathrm{d}\Omega \mathrm{d}\omega_2} \Delta\Omega \cdot \Delta\omega_2 \cdot \rho \cdot d \cdot a_{\mathrm{r/t}} \cdot R \cdot D. \tag{1}$$

Here, I_0 is the incident photon flux, $\frac{\mathrm{d}^2\sigma}{\mathrm{d}\Omega \mathrm{d}\omega_2}$ is the DDSCS, $\Delta\Omega$ is the solid angle of detection, $\Delta\omega_2$ is the energy resolution, ρ the number density of scatterers in the interaction volume, d is the sample thickness, $a_{\mathrm{r/t}}$ is the sample absorption factor, and R and D are the finite reflectivity of the analyzer crystals and the detector efficiency, respectively.

In XRSTools, each of the factors in equation (1) is a separate class in the xrs_prediction module. Thus, there are six classes

- detector,
- analyzer,
- sample,
- polarization,
- x-ray beam,
- some output parameters,

each of which has a separate block in it's input file. Here is a summary of the variables to be set for each of these blocks.

Detector

The variables in the 'Detector' section are used to calculate the efficiency of the Detector at the energy used via:

$$D = 1.0 - \exp(-d * \mu), \tag{2}$$

where D is the efficiency of the detector, d is the thickness of the active detector chip, and μ is the photoelectric absorption of the detector active material.

variable	description	type	default	unit
energy	analyzer energy	float	9.7	keV
thickness	detector chip thickness	float	500.0	microns
material	detector active material	string	'Si'	-
pixel_size	number of pixels (obsolete)	list	[256,768]	int

Analyzer

The analyzer class is meant to calculate the analyzer reflectivity for the used analyzer material and reflection. For an estimate of the analyzer reflectivity, the Takagi-Taupin equations[**? ? ?**] are solved for a spherically bent crystal analyzers. We approximate R in (1) by averaging over the full width at half maximum of the resulting reflectivity curve. Also $\Delta\Omega$ comes from the parameters in this section of the input file.

variable	description	type	default	unit
material	analyzer material	string	'Si'	-
hkl	hkl of reflection used	list of ints	[6,6,0]	-
mask_d	analyzer mask diameter	float	60.0	mm
bend_r	analyzer curvature radius	float	1.0	meters
diced	keyword analyzer is diced (obsolete)	boolean	false	-
thickness	analyzer thickness	float	500.0	microns
energy_resolution	approx. analyzer resolution	float	0.5	eV
database_dir	path to reflectivity database	string	-	-

Sample

Whereas most parameters in the input file have default values, the sample part is the one section that has to be filled with parameters on the details of the sample in question. Here is a list for the necessary ones.

Right now, the program assumes that the sample is spherical. A spherical sample is the ideal shape for the multi analyzer spectrometer at ID20. Future versions will also have options for flat samples (either in transmission or reflection geometry).

Most of the input parameters used in this section of the input file are Python lists. The composition of the sample is passed by a list of chemical sum formulas (e.g. ['SiO2','H2O']), the relative concentration and densities of its constituents as lists in the same order as the sum formulas (e.g. [0.4, 0.6] and [... , ...] for a sample made up of 40 % SiO2 and 60 % water). This, however, means that the program assumes the sample to also be a sort of emulsion of its constituents, each of which contribute separately to the total DDSCS.

variable	description	type	default	unit
chem_formulas	chemical sum formulas	list of strings	-	-
concentrations	concentrations for each chem formula	list of ints	-	rel. units
densities	densities of constituents	list of ints	-	g/cm^3
molar_masses	molar masses of each constituent	list of ints	-	g/mol
angle_tth	scattering angle [2Th]	float	-	degrees
sample_thickness	sample thickness	float	-	cm

Polarization

Here, you can put some information about the Thomson part of the DDSCS, in the future (for simplicity) this could also move to the beamline specific paramters.

variable	description	type	default	unit
scattering_plane	scattering plane used ('vertical' or 'horizontal')	keyword	'vertical'	-
polarization	degree of polarization in your beam	float	0.99	%

X-ray beam

Here is still some info on the incident beam.

variable	description	type	default	unit
i0_intensity	number of incident photons	int	1e13	1/second
beam_heigh	beam size in the vertical direction	int	10.0	microns
beam_width	beam size in the horizontal direction	int	20.0	microns

Output

Finally, there are some parameters to determine the output.

variable	description	type	default	unit
eloss_range	energy loss range for the output	np.array	np.arange(0.0,1000.0,0.1)	eV
E0	analyzer energy	int	9.7	keV

Outlook

As mentioned earlier, the samples are assumed to be spherical and inmissible. Rearrangement of some input/output paramters and internal variables should enable also different geometries and maybe also an easier-to-read input file.

A GUI should also be straightforward to implement.

using the command line

All functions and classes used when calling the prediction routine via an input file are, of course, also accessible from the Python command line.

GETTING STARTED WITH THE COMMAND LINE

Start by importing the XRSTools routines and some useful other modules like Pylab for plotting and Numpy for some other Matlab like behavior.

```
from xrstools import xrs_read, ...
from pylab import *
import numpy as np
```

I usually like to set the Pylab interactive mode to 'on'. This way, Pylab acts somewhat similar to Matlab plotting.

```
ion()
```

PERFORMING

During experiments, there are mainly two things that are important for the XRS user, the sample alignment and some fast online data analysis. XRSTools has some functions to make this easy.

Sample alignment

An elegant way to align samples contained in some complicated sample environment is to make use of the imaging properties of bent crystal analyzers in combination with a 2D detector (check out [13] and [1] to learn more about the details).

Start by importing the main XRSTools data reading routine and some useful other modules like Pylab for plotting and Numpy for some other Matlab like behavior.

```
from xrstools import xrs_read
from pylab import *
import numpy as np
```

Now create an instance of the read_id20 class and give it some sensible name (name of the compound/sample, for simplicity, we will just call the variable 'image').

```
image = xrs_read.read_id20(absfilename,energycolumn='energy',monitorcolumn='kap4dio'
    ,edfName=None,single_image=True)
```

The input parameters for this class are explained in the following table.

variable	description	type	default	unit
absfilename	absolute path to the SPEC file	string	-	-
energycolumn	name of the energy counter in the SPEC file	string	'energy'	-
monitorcolumn	name of the monitor counter in the SPEC file	string	'kap4dio'	-
edfName	prefix of the EDF files	string	None	-
single_image	keyword, if there is a single detector image	boolean	True	-

To construct an image from an alignment scan (i.e. a sample translation scan), use the *make_posscan_image* (for 'make position-scan image') function from the 'image' object you have just created.

```
image.make_posscan_image(scannumber,motorname,filename=None)
```

variable	description	type	default	unit
scannumber	scannumber from the alignment scan you want to image	string	-	-
motorname	name of the motor scanned ('stx', 'sty', 'stz')	string	-	-
filename	(optional) absolute path if image should be saved	string	None	-

This will open a plot window showing a sum of all detector images taken during the translation scan. Using the zoom tool of the plotting window ('little magnifying glass' at the lower left corner of the plotting window) zoom into the spot you like (**screenshots of this procedure and link to 'setting the ROIs' section.**), go back to your command line and press enter. This will close the ROI window and open a new one with your reconstructed image. On the y-axis you will find the motor and range you scanned, on the x-axis of your plot you will have the pixels along the beam (remember that one pixel is 55 micron, which will give you the conversion into real distances).

And here is how it looks like. **take examples from the acetic acid data also used in the paper...**

raw data, quick and dirty

Here is a minimal set of functions you will need for a 'quick and dirty' online data analysis.

```
from xrstools import xrs_read
from pylab import *
import numpy as np
ion()
sample = xrs_read.read_id20(absfilename,energycolumn='energy',monitorcolumn='kap4dio
    ',edfName=None,single_image=True)
sample.loadelastic(scannumbers)
sample.get_zoom_rois(scannumbers)
sample.getrawdata()
sample.loadscandirect(scannumbers,name)
sample.getspectrum()
sample.geteloss()
```

Let us go through these (and similar functions) in more detail and explain each of them and (most of) the parameters needed. The idea is that you 1. import modules from the XRSTools package (and others from the standard library such as Numpy and Pylab), 2. create an instance of the *xrs_read.read_id20* class, and 3. use the class's features to load SPEC- and EDF-files, set regions of interest (ROIs), integrate the detector images, and make simple plots of the data.

starting up

To start up, import some elastic line and create some regions of interest. If you have not imported the XRSTools module (and some other useful build in modules) now is a good time to do it.

```
from xrstools import xrs_read
from pylab import *
import numpy as np
```

Let's also use Pylab's interactive mode (this makes Pylab behave a bit more like Matlab/Octave).

```
ion()
```

As within the sample alignment, first create an object of the 'xrs_read.read_id20' class.

```
sample = xrs_read.read_id20(absfilename,energycolumn='energy',monitorcolumn='kap4dio
    ',edfName=None,single_image=True)
```

I chose the variable name 'sample', since the idea is to have one object of the *read_id20* class for each sample. Next, load some SPEC- and according EDF-files.

```
sample.loadelastic(scannumbers, fromtofile=False)
```

Here is a description of the parameters this function takes.

variable	description	type	default	unit
scannumbers	number or list of number of scans to load	int or list of ints	-	-
fromtofile	keyword, if scan instance should be saved in a file (experimental)	boolean	False	-

Now, we can use this elastic line to define some ROIs.

selecting the ROIs

XRSTool's xrs_rois module provides a number of ways to define ROIs, so let's go through them one by one. In principle, you will have to follow the instructions printed out in the shell and on the plotting windows.

Zoom ROIs To define ROIs by using Matplotlib's built-in zoom function, use this.

```
sample.get_zoom_rois(scannumbers)
```

- activate the zoom function in the plot-window
- click the next button
- zoom into your first ROI
- click the next button
- repeat until you zoomed into the last ROI
- click the finish button

Look at the function's docstr for more help:

```
help(sample.get_zoom_rois()) .
```

Line ROIs To define ROIs by clicking two points, use this.

```
sample.get_linear_rois(scannumbers)
```

- activate the zoom function in the plot-window
- click the next button
- zoom into your first ROI
- click the next button
- repeat until you zoomed into the last ROI
- click the finish button

Look at the function's docstr for more help:

```
help(sample.get_linear_rois()) .
```

Polygon ROIs To define ROIs by clicking multiple points, use this.

```
sample.get_plygon_rois(scannumbers)
```

- blabla

Look at the function's docstr for more help:

```
help(sample.get_polygon_rois()) .
```

Automatic ROIs There are two ways of selecting automatic ROIs. One which uses the entire 6-detector image and one that goes through each detector.

```
sample.get_auto_rois(scannumbers)
```

-

Look at the function's docstr for more help:

```
help(sample.get_auto_rois()) .
```

Saving and loading ROIs from files Once you have defined some ROIs, it may be a good idea to save the ROIs into a text file so that you will not have to redefine them in a possible later session.

To save ROIs into a file use

```
sample.save_rois(filename) ,
```

where 'filename' is a path and filename for your ROIs.

loading the data

Now, it is time to load some data.

The general idea of the package is, that you measure different parts of your spectra in different scans. These could e.g. be frequent scans of the quasielastic peak, a rough overview over the whole range to estimate the valence background, and then your spectral region of interest. Also the absorption edge may be devided into several parts, not all of which may be desired with the same statistical accuracy. In the end, all of these scans should be added up, stitched together and shifted with respect to the elastic line energy.

The most general function for loading data is:

```
sample.loadscan(scannumbers, scantype='generic', fromtofile=False)
```

variable	description	type	default	unit
scannumbers	number or list of number of scans to load	int or list of ints	-	-
fromtofile	keyword, if scan instance should be saved in a file (experimental)	boolean	False	-

The idea is, that all scans that you load into your variable 'sample' will have an attribute 'scantype' by which the program knows which scans belong together (to add them up).

There are a few 'scantypes' which are special in XRSTools. The first one is 'elastic' to define a scan as an elastic line scan. There is a shorthand version of the loadscan function to load elastic lines:

```
sample.loadelastic(scannumbers, fromtofile=False) .
```

The second special scantype is 'long' which is used for overview scans. Also here, there is a shorthand version

```
sample.loadlong(scannumbers, fromtofile=False) .
```

Since most XRS spectra are stiched together from several single scans (e.g. an elastic line scan, an overview scan, and several regions across some edge, or even different edges).

plotting

A CLOSER LOOK

Right now, the most efficient way to measure XRS spectra is by splitting up the spectral regions around an absorption edge and weigh the different regions by counting time and energy increment. On top of these regions, we usually take elastic line scans frequently for energy calibration. Consider the oxygen K-edge of water as an example. For a decent background subtraction, we usually take a long overview scan on a rough energy grid that covers the whole range of the Compton/Valence excitation. The near edge, we usually devide in 3-5 regions, one (10-15 eV) before the edge, one that covers the pre- and main-edge region, one which covers the post-edge region, and one that covers the first big EXAFS-like oscillation around 555 eV energy loss. To easily add up data from different scans and analyzer crystals, we usually keep the same energy grid in all near-edge regions and weigh by chosing different counting times.

To avoid problems when stitching different scan-regions together during the data analysis, it turns out the best practise is to let the monochromators make constant steps across the edge (i.e. let one region start exactly one energy increment after the last region's end point) and then loop over the entire range of the edge. To clarify, a possible SPEC makro to do this could look like this:

```
# define the elastic line energy
E0 = 9.76

# elastic line scan
umv energy E0-0.004
ascan energy E0-0.003 E0+0.003 60 0.1

# loop over the oxygen K-edge (constant energy step width, weigh by second/point)
while (ii < 10, ii=1, ii++){
umv energy E0+0.515
umv energy E0+0.519
ascan energy E0+0.520  E0+0.532  60  5
ascan energy E0+0.5322 E0+0.542  49 10
ascan energy E0+0.5422 E0+0.5552 60  7
ascan energy E0+0.5554 E0+0.570  60  4
}

# overview scan (covering the Compton/valence spectrum)
umv energy E0-0.011
ascan energy E0-0.01 E0+0.60 610 2
```

XRSTools 'knows' how to stich scans together is because during loading of scans, each scan is given a scan-type. A scan-type is just a string. However, there are a few special strings to describe scans that are treated specially. The special types are **elastic** and **long**. The default type is **generic**.

For the above example, I would thus choose the following strings as type.

```
h2o = xrs_read.read_id20(absfilename,energycolumn='energy',monitorcolumn='kap4dio',
    edfName=None,single_image=True)
h2o.loadelastic(14)    # load elastic scan number 14
h2o.get_zoom_rois(14) # define some ROIs by zooming
h2o.getrawdata()       # integrate the elastic line scan data
h2o.loadscandirect(15,'edge1') # load 1. part of the K-edge
h2o.loadscandirect(16,'edge2') # load 2. part of the K-edge
h2o.loadscandirect(17,'edge3') # load 3. part of the K-edge
h2o.loadscandirect(17,'edge4') # load 4. part of the K-edge
h2o.loadlongdirect(54) # load an overview scan
h2o.getspectrum() # stitch all parts together
h2o.geteloss()     # find the center of mass of the elastic line
```

Note that

```
h2o.loadelastic(14)
```

and

```
h2o.loadlongdirect(54)
```

are just shorthand commands for

```
h2o.loadscan(14,'elastic')
```

and

```
h2o.loadscandirect(14,'long')
```

Instead of loading the individual parts of the spectrum separately, there is a function that can load an entire loop of spectra (such as in the example given above).

```
h2o.loadloopirect([15],4)
```

For this to work, you have to just pass the starting numbers of all loops as a Python list (here, we only had one loop starting with scan number 14) and the number of scan within the loop (here 4).

Notice that we are using two different types of functions for loading the spectra. In the case of the elastic line we used the *loadelastic()*-function in case of the other scans we used the *loadscandirect()*-function. Loading scans with functions including the word 'direct' merely tells the program to load both the SPEC- and EDF-file, then use existing ROIs to integrate the EDF-images, and subsequently delete the EDF-files from the memory. The only reason for this is that the EDF-files are somewhat large and quickly take up all of the memory and (in most cases) they are not needed after integration.

ANALYSIS

With analysis, I mostly mean data reading as we have done before (see last chapters) and the subtraction of the backgound.

raw data

There are a few things worth mentioning that I did not cover in the on-line analysis part.
scaling setting scattering angles

the theory module

here some information about the theory module, the HF compton profiles, etc.

background subtraction

The backgroung subtraction is done using the *xrs_extraction*-module. This is the most underdeveloped module of all the module in the XRSTools package. The reason is simply that it is also the most demanding to make up functions that work for most background subtraction problems that I can think of.

IMAGING

This part of the program is purely experimental so far and thus somewhat a developer version. Functionality should be added in due time.

OUTLOOK

Here, I would like to keep a list of things to be done to improve the current version and things to add in a possible new version of the code.

xrs_prediction template

```
#### detector
energy     = 9.7                    # working energy, same as analyzer energy (E0)
thickness  = 500.0                  # thickness of analyzer active material (micron)
material   = 'Si'                   # analyzer active material
pixel_size = [256,768]              # number of pixels (height, width)

#### analyzer
material  = 'Si'          # analyzer material
hkl       = [6,6,0]       # hkl-indices of reflection used
mask_d    = 60.0          # mask diameter (in mm)
bend_r    = 1.0           # analyzer crystal bending radius
diced     = False         # boolean: False for bent crystals, True for diced crystals
thickness = 500.0         # analyzer crystal thickness (micron)
energy_resolution = 0.5 # expected overall resolution (eV)
database_dir = '/dir/to/refl_database/' # directory to where already calculated
                                        # reflectivities are stored

#### sample
chem_formulas = ['FeC6N6','H2O']        # chemical formulae (python list)
concentrations = [0.03,0.97]            # concentrations (python list)
densities = [1.9,1.0]                   # densities (python list)
molar_masses = [211.9494,18.02]         # molar masses of the sample's constituents,
                                        # needed for calculation of the number of
                                        # scatterers (python list)
angle_tth = 122.0                       # 2 theta scattering angle (degress)
sample_thickness = 0.030                # sample thickness/diameter (cm)

#### thomson
scattering_plane = 'vertical'           # keyword, either 'vertical' (no
    polarization
                                        # dependence) or 'horizontal' (polarization
                                        # dependence)
polarization     = 0.99                 # degree of polarization of incident
    radiation

#### beam
i0_intensity = 1e13                     # number of incident photons
beam_height  = 10.0                     # beam dimension in the vertical (microns)
beam_width   = 20.0                     # beam dimension in the horizontal (microns)

#### compton_profiles
eloss_range = np.arange(0.0,1000.0,0.1) # energy loss range for the calculation (eV)
E0          = 9.7                       # analyzer energy (keV)
```

* christoph.sahle@esrf.fr

[1] Ch.J. Sahle et al. submitted to J. Synchrotron Rad. (2014).
[2] Schülke, W. (2007). Electron dynamics by inelastic X-ray scattering. Oxford Univ. Press.
[3] Hämäläinen, K., & Manninen, S. (2001). Resonant and non-resonant inelastic x-ray scattering. Journal of Physics: Condensed Matter, 13(34), 7539.
[4] Sinha, S. K. (2001). Theory of inelastic X-ray scattering from condensed matter. Journal of Physics: Condensed Matter, 13(34), 7511.
[5] Hetland, Magnus Lie. Beginning Python. Apress, 2005.
[6] Sternemann, H., Sternemann, C., Seidler, G. T., Fister, T. T., Sakko, A., & Tolan, M. (2008). An extraction algorithm for core-level excitations in non-resonant inelastic X-ray scattering spectra. Journal of synchrotron radiation, 15(2), 162-169.
[7] Huotari, S., Pylkkänen, T., Soininen, J. A., Kas, J. J., Hämäläinen, K., & Monaco, G. (2011). X-ray-Raman-scattering-based EXAFS beyond the dipole limit. Journal of synchrotron radiation, 19(1), 106-113.
[8] Sakurai, Y., Yamaoka, H., Kimura, H., Marechal, X. M., Ohtomo, K., Mochizuki, T., et al. (1995). Design of an elliptic multipole wiggler beamline for high-energy inelastic scattering at the SPring-8. Review of scientific instruments, 66(2), 1774-1776.
[9] Fister, T. T., Seidler, G. T., Wharton, L., Battle, A. R., Ellis, T. B., Cross, J. O., et al. (2006). Multielement spectrometer for efficient measurement of the momentum transfer dependence of inelastic x-ray scattering. Review of scientific instruments, 77(6), 063901-063901.
[10] Verbeni, R., Pylkkänen, T., Huotari, S., Simonelli, L., Vanko, G., Martel, K., et al. (2009). Multiple-element spectrometer for non-resonant inelastic X-ray spectroscopy of electronic excitations. Journal of synchrotron radiation, 16(4), 469-476.
[11] Sokaras, D., Nordlund, D., Weng, T. C., Mori, R. A., Velikov, P., Wenger, D., et al. (2012). A high resolution and large solid angle x-ray Raman spectroscopy end-station at the Stanford Synchrotron Radiation Lightsource. Review of Scientific Instruments, 83(4), 043112.
[12] Hiraoka, N., H. Fukui, H. Tanida, H. Toyokawa, Y. Q. Cai, and K. D. Tsuei. "An X-ray Raman spectrometer for EXAFS studies on minerals: bent Laue spectrometer with 20 keV X-rays." Journal of synchrotron radiation 20, no. 2 (2013): 266-271.
[13] Huotari, S., Pylkkänen, T., Verbeni, R., Monaco, G., & Hämäläinen, K. (2011). Direct tomography with chemical-bond contrast. Nature materials, 10(7), 489-493.