# SCS Toolbox Documentation

**SCS**

**Nov 26, 2024**

# CONTENTS:

CHAPTER

ONE

# INSTALLATION

## 1.1 Recommended: Proposal Environment setup

The proposal specific environment is installed by launching in a shell terminal on Maxwell:

```
module load exfel exfel-python
scs-activate-toolbox-kernel --proposal 2780
```

where in this example 2780 is the proposal number. After this and refreshing the browser, a new kernel named `SCS Toolbox (p002780)` is available and should be used to run jupyter notebooks on the Maxwell Jupyter hub.

## 1.2 Figures setup: enabling matplotlib constrained layout

To get the best looking figures generated by the SCS Toolbox, you need to enable the experimental constrained_layout solver in matplotlib. This is done in jupyter notebook with adding at the start the following lines:

```
import matplotlib.pyplot as plt
plt.rcParams['figure.constrained_layout.use'] = True
```

## 1.3 Alternative: Manual ToolBox Installation

The ToolBox may be installed in any environment. However, it depends on the extra_data and the euxfel_bunch_pattern package, which are no official third party python modules. Within environments where the latter are not present, they need to be installed by hand.

In most cases, you will want to install it in the *exfel_python* environment which is the one corresponding to the *xfel* kernel on max-jhub.

To do so, first activate that environment:

```
module load exfel exfel-python
```

Then, check that the scs_toolbox is not already installed:

```
pip show toolbox_scs
```

If the toolbox has been installed in your home directory previously, everything is set up. If you need to upgrade the toolbox to a more recent version, you have to either uninstall the current version:

```
pip uninstall toolbox_scs
```

or if it was installed in development mode, go in the toolbox directory and pull the last commits from git:

```
cd #toolbox top folder
git pull
```

Otherwise it needs to be installed (only once). In that you first need to clone the toolbox somewhere (in your home directory for example) and install the package. Here the -e flag install the package in development mode, which means no files are copied but only linked such that any changes made to the toolbox files will have effect:

```
cd ~
git clone https://git.xfel.eu/SCS/ToolBox.git
cd ~/ToolBox
pip install --user -e ".[maxwell]"
```

CHAPTER

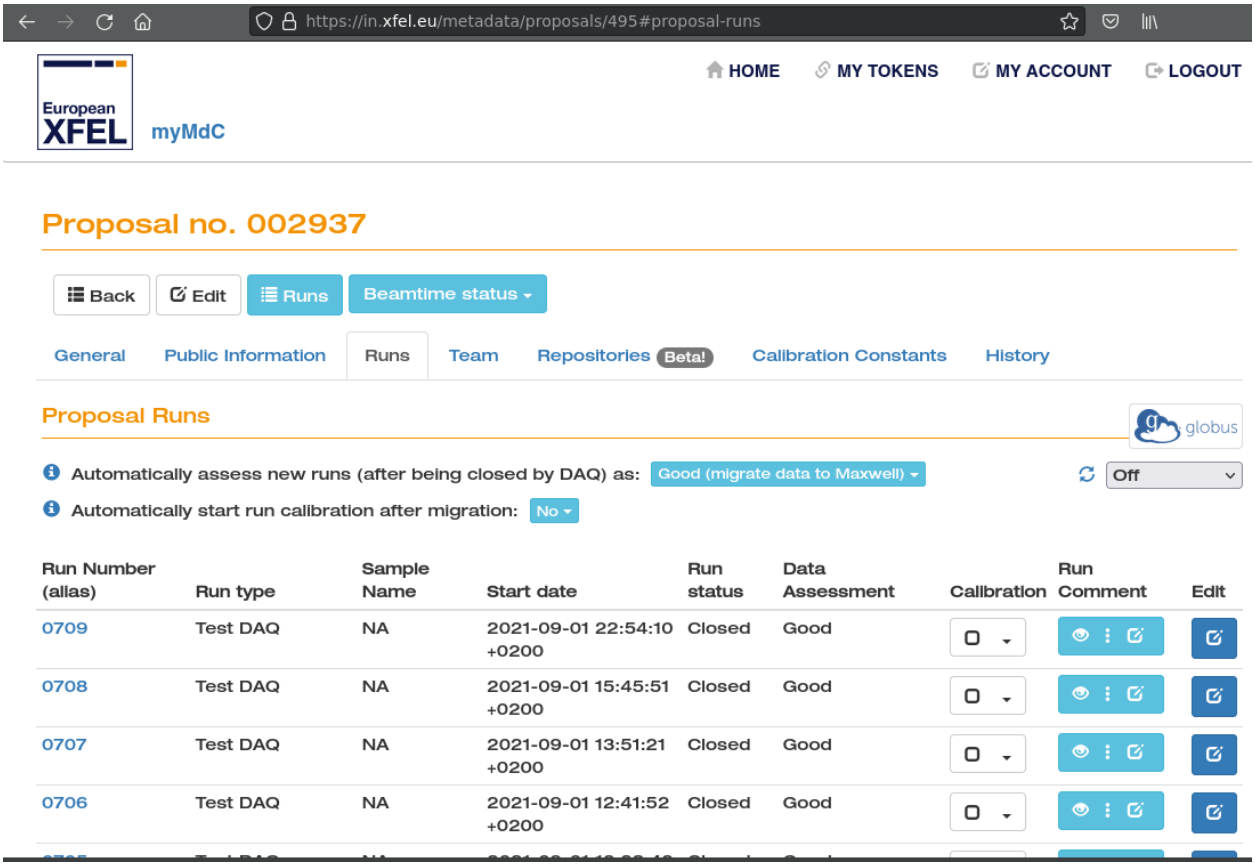
TWO

# TRANSFERRING DATA

The DAQ system save data on the online cluster. To analyze data on the Maxwell offline cluster, they need to be transferred there. This is achieved by login at:

https://in.xfel.eu/metadata

and then navigating to the proposal, then to the `Runs` tab from where runs can be transferred to the offline cluster by marking them as `Good` in the `Data Assessment`. Depending on the amount of data in the run, this can take a while.



CHAPTER

# THREE

# PROCESSING DATA

On the Maxwell Jupyter hub:

https://max-jhub.desy.de

notebooks can be executed using the corresponding Proposal Environment kernel or the `xfel` kernel if the toolbox was manually installed. For quick startup, example notebooks (.ipynb files) can be directly downloaded from the *How to's* section by clicking on the `View page source`.

## 3.1 How to's

### 3.1.1 Loading run data

- load run and data.
- load data in memory.

### 3.1.2 Reading the bunch pattern

- bunch pattern decoding.

### 3.1.3 Extracting peaks from digitizers

- How to extract peaks from digitizer traces.

### 3.1.4 Determining the FEL or OL beam size and the fluence

- Knife-edge scan and fluence calculation.

### 3.1.5 Finding time overlap by transient reflectivity

Transient reflectivity of the optical laser measured on a large bandgap material pumped by the FEL is often used at SCS to find the time overlap between the two beams. The example notebook

- Transient reflectivity measurement

shows how to analyze such data, including correcting the delay by the bunch arrival monitor (BAM).

### 3.1.6 DSSC

#### DSSC data binning

In scattering experiment one typically wants to bin DSSC image data versus time delay between pump and probe or versus photon energy. After this first data reduction steps, one can do azimuthal integration on a much smaller amount of data.

The DSSC data binning procedure is based on the notebook Dask DSSC module binning. It performs DSSC data binning against a coordinate specified by *xaxis* which can be *nrj* for the photon energy, *delay* in which case the delay stage position will be converted in picoseconds and corrected but the BAM, or another slow data channel. Specific pulse pattern can be defined, such as:

```
['pumped', 'unpumped']
```

which will be repeated. XGM data will also be binned similarly to the DSSC data.

Since this data reduction step can be quite time consuming for large datasets, it is recommended to launch the notebook via a SLURM script. The script can be downloaded from `scripts/bin_dssc_module_job.sh` and reads as:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --partition=exfel
#SBATCH --time=12:00:00
#SBATCH --mail-type=END,FAIL
#SBATCH --output=logs/%j-%x.out

while getopts ":p:d:r:k:m:x:b:" option
do
    case $option in
        p) PROPOSAL="$OPTARG";;
        d) DARK="$OPTARG";;
        r) RUN="$OPTARG";;
        k) KERNEL="$OPTARG";;
        m) MODULE_GROUP="$OPTARG";;
        x) XAXIS="$OPTARG";;
        b) BINWIDTH="$OPTARG";;
        \?) echo "Unknown option"
            exit 1;;
        :) echo "Missing option for input flag"
           exit 1;;
    esac
done

# Load xfel environment
```

(continues on next page)

(continued from previous page)

```
source /etc/profile.d/modules.sh
module load exfel exfel-python

echo processing run $RUN
PDIR=$(findxfel $PROPOSAL)
PPROPOSAL="p$(printf '%06d' $PROPOSAL)"
RDIR="$PDIR/usr/processed_runs/r$(printf '%04d' $RUN)"
mkdir $RDIR

NB='Dask DSSC module binning.ipynb'

# kernel list can be seen from 'jupyter kernelspec list'
if [ -z "${KERNEL}" ]; then
          KERNEL="toolbox_$PPROPOSAL"
fi

python -c "import papermill as pm; pm.execute_notebook(\
  '$NB', \
  '$RDIR/output$MODULE_GROUP.ipynb', \
  kernel_name='$KERNEL', \
  parameters=dict(proposalNB=int('$PROPOSAL'), \
                  dark_runNB=int('$DARK'), \
                  runNB=int('$RUN'), \
                  module_group=int('$MODULE_GROUP'), \
                  path='$RDIR/', \
                  xaxis='$XAXIS', \
                  bin_width=float('$BINWIDTH')))"
```

It is launched with the following:

```
sbatch ./bin_dssc_module_job.sh -p 2719 -d 180 -r 179 -m 0 -x delay -b 0.1
sbatch ./bin_dssc_module_job.sh -p 2719 -d 180 -r 179 -m 1 -x delay -b 0.1
sbatch ./bin_dssc_module_job.sh -p 2719 -d 180 -r 179 -m 2 -x delay -b 0.1
sbatch ./bin_dssc_module_job.sh -p 2719 -d 180 -r 179 -m 3 -x delay -b 0.1
```

where 2719 is the proposal number, 180 is the dark run number, 179 is the run nummber and 0, 1, 2 and 3 are the 4 module group, each job processing a set of 4 DSSC module, delay is the bin axis and 0.1 is the bin width.

The result will be 16 *.h5 files, one per module, saved in the folder specified in the script, a copy of which can be found in the *scripts* folder in the toolbox source. This files can then be loaded and combined with:

```
import xarray as xr
data = xr.open_mfdataset(path + '/*.h5', parallel=True, join='inner')
```

#### DSSC azimuthal integration

Azimuthal integration can be performed with pyFAI which can utilize the hexagonal pixel shape information from the DSSC geometry to split the intensity in a pixel in the bins covered by it. An example notebook Azimuthal integration of DSSC with pyFAI.ipynb is available.

A second example notebook DSSC scattering time-delay.ipynb demonstrates how to:

- refine the geometry such that the scattering pattern is centered before azimuthal integration
- perform azimuthal integration on a time delay dataset with `xr.apply_ufunc` for multiprocessing.
- plot a two-dimensional map of the scattering change as function of scattering vector and time delay
- integrate certain scattering vector range and plot a time trace

#### DSSC fine timing

When DSSC is reused after a period of inactivity or when the DSSC gain setting use a different operation frequency the DSSC fine trigger delay needs to be checked. To analysis runs recorded with different fine delay, one can use the notebook DSSC fine delay with SCS toolbox.ipynb.

#### DSSC quadrant geometry

To check or refined the DSSC geometry or quadrants position, the following notebook can be used DSSC create geometry.ipynb.

#### Legacy DSSC binning procedure

Most of the functions within toolbox_scs.detectors can be accessed directly. This is useful during development, or when working in a non-standardized way, which is often neccessary during data evaluation. For frequent routines there is the possibility to use dssc objects that guarantee consistent data structure, and reduce the amount of recurring code within the notebook.

- bin data using toolbox_scs.tbdet -> *to be documented*.
- bin data using the DSSCBinner.
- post processing, data analysis -> *to be documented*

### 3.1.7 Photo-Electron Spectrometer (PES)

- Basic analysis of PES spectra.

### 3.1.8 BOZ: Beam-Splitting Off-axis Zone plate analysis

The BOZ analysis consists of 4 notebooks and a script. The first notebook BOZ analysis part I.a Correction determination is used to determine all the necessary correction, that is the flat field correction from the zone plate optics and the non-linearity correction from the DSSC gain. The inputs are a dark run and a run with X-rays on three broken or empty membranes. For the latter, an alternative is to use pre-edge data on an actual sample. The result is a JSON file that contains the flat field and non-linearity correction as well as the parameters used for their determination such that this can be reproduced and investigated in case of issues. The determination of the flat field correction is rather quick, few minutes and is the most important correction for the change in XAS computed from the -1st and +1st order. For quick correction of the online preview one can bypass the non-linearity calculation by taking the JSON file as soon as

it appears. The determination of the non-linearity correction is a lot longer and can take some 2 to 8 hours depending on the number of pulses in the train. For this reason, the computation can also be done on GPUs in 30min instead. A GPU notebook adapted for CHEM experiment with liquid jet and normalization implement for S K-edge is available at OnlineGPU BOZ analysis part I.a Correction determination S K-egde.

The other option is to use a script that can be downloaded from `scripts/boz_parameters_job.sh` and reads as:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --partition=allgpu
#SBATCH --constraint=V100
#SBATCH --time=2:00:00
#SBATCH --mail-type=END,FAIL
#SBATCH --output=logs/%j-%x.out

ROISTH='1'
SATLEVEL='500'
MODULE='15'

while getopts ":p:d:r:k:g:t:s:m:" option
do
    case $option in
        p) PROPOSAL="$OPTARG";;
        d) DARK="$OPTARG";;
        r) RUN="$OPTARG";;
        k) KERNEL="$OPTARG";;
        g) GAIN="$OPTARG";;
        t) ROISTH="$OPTARG";;
        s) SATLEVEL="$OPTARG";;
        m) MODULE="$OPTARG";;
        \?) echo "Unknown option"
            exit 1;;
        :) echo "Missing option for input flag"
           exit 1;;
    esac
done

# Load xfel environment
source /etc/profile.d/modules.sh
module load exfel exfel-python

echo processing run $RUN
PDIR=$(findxfel $PROPOSAL)
PPROPOSAL="p$(printf '%06d' $PROPOSAL)"
RDIR="$PDIR/usr/processed_runs/r$(printf '%04d' $RUN)"
mkdir $RDIR

NB='BOZ analysis part I.a Correction determination.ipynb'

# kernel list can be seen from 'jupyter kernelspec list'
if [ -z "${KERNEL}" ]; then
        KERNEL="toolbox_$PPROPOSAL"
fi

```

(continues on next page)

(continued from previous page)

```
python -c "import papermill as pm; pm.execute_notebook(\
  '$NB', \
  '$RDIR/output.ipynb', \
  kernel_name='$KERNEL', \
  parameters=dict(proposal=int('$PROPOSAL'), \
                  darkrun=int('$DARK'), \
                  run=int('$RUN'), \
                  module=int('$MODULE'), \
                  gain=float('$GAIN'), \
                  rois_th=float('$ROISTH'), \
                  sat_level=int('$SATLEVEL')))"
```

It uses the first notebook and is launched via slurm:

```
sbatch ./boz_parameters_job.sh -p 2937 -d 615 -r 614 -g 3
```

where 2937 is the proposal run number, where 615 is the dark run number, 614 is the run on 3 broken membranes and 3 is the DSSC gain in photon per bin. The proposal run number is defined inside the script file.

The second notebook BOZ analysis part I.b Correction validation can be used to check how well the calculated correction still work on a characterization run recorded later, i.e. on 3 broken membrane or empty membranes.

The third notebook BOZ analysis part II.1 Small data then use the JSON correction file to load all needed corrections and process an run, saving the rois extracted DSSC as well as aligning them to photon energy and delay stage in a small data h5 file.

That small data h5 file can then be loaded and the data binned to compute a spectrum or a time resolved XAS scan using the fourth and final notebook BOZ analysis part II.2 Binning

### 3.1.9 Point detectors

Detectors that produce one point per pulse, or 0D detectors, are all handled in a similar way. Such detectors are, for instance, the X-ray Gas Monitor (XGM), the Transmitted Intensity Monitor (TIM), the electron Bunch Arrival Monitor (BAM) or the photo diodes monitoring the PP laser.

### 3.1.10 HRIXS

- Analyzing HRIXS data

### 3.1.11 Viking spectrometer

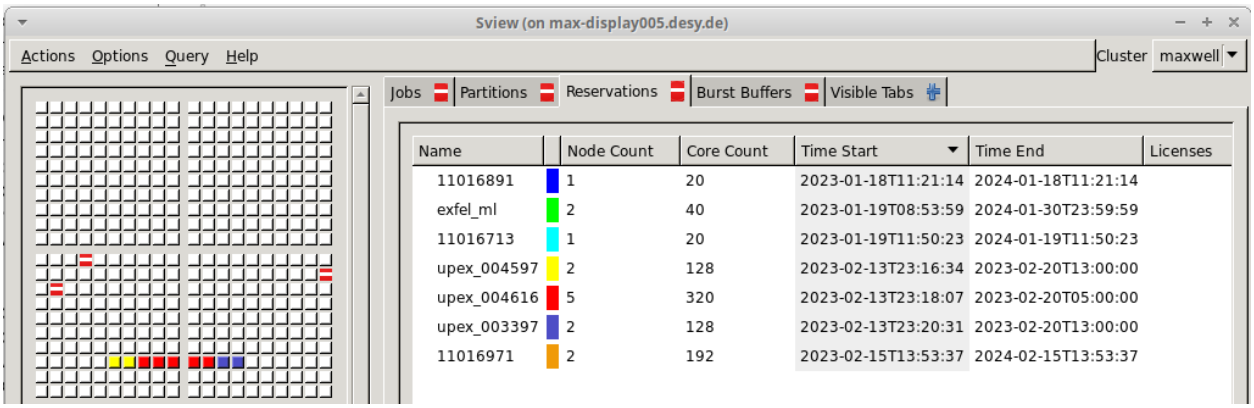
- Analysis of Viking spectrometer data

### 3.1.12 SLURM, sbatch, partition, reservation

Scripts launched by `sbatch` command can employ magic cookie with `#SBATCH` to pass options SLURM, such as which partition to run on. To work, the magic cookie has to be at the beginning of the line. This means that:

- to comment out a magic cookie, adding another "#" before it is sufficient
- to comment a line to detail what the option does, it is best practice to put the comment on the line before

Reserved partition are of the form "upex_003333" where 3333 is the proposal number. To check what reserved partition are existing, their start and end date, one can `ssh` to `max-display` and use the command `sview`.




To use a reserved partition with `sbatch`, one can use the magic cookie

```
#SBATCH --reservation=upex_003333
```

instead of the usual

```
#SBATCH --partition=upex
```

## 3.2 Maintainers

### 3.2.1 Creating a Toolbox environment in the proposal folder

A Toolbox environment can be created by running the following commands in Maxwell:

```
module load exfel exfel-python
scs-create-toolbox-env --proposal <PROPOSAL>
```

where <PROPOSAL> is the desired proposal number. This will create a Python environment and will download and install the Toolbox source code. It will result to the creation of the following folders in the path <PROPOSAL_PATH>/scratch.

```
<PROPOSAL_PATH>/scratch/
├─ checkouts/
│   ├─ toolbox_<PROPOSAL>/
│   │   ├─ <source code>
├─ envs/
│   ├─ toolbox_<PROPOSAL>/
│   │   ├─ <Python files>
```

The `checkouts` folder contains the Toolbox source codes, correspondingly labeled according to the environment identifier, which is the proposal number by default. The downloaded code defaults to the **master** version at the time when the environment is created.

The `envs` folder contains the Python environment with the packages necessary to run the Toolbox. It is also correspondingly labeled according to the environment identifier, which is the proposal number by default.

---

**Note:** One can find the proposal path by running `findxfel <PROPOSAL>`.

---

It is a good practice to tag the Toolbox version at a given milestone. This version can be then supplied to the script as:

```
scs-create-toolbox-env --proposal <PROPOSAL> --version <VERSION>
```

It might also be helpful to supply an identifier to distinguish environments from each other. This can be done by running:

```
scs-create-toolbox-env --proposal <PROPOSAL> --identifier <IDENTIFIER>
```

The environment would then be identified as `toolbox_<IDENTIFIER>` instead of `toolbox_<PROPOSAL>`.

### 3.2.2 Installing additional packages

In order to install additional packages in a Toolbox environment, one should run the following commands:

```
cd <PROPOSAL_PATH>/scratch/
source envs/toolbox_<IDENTIFIER>/bin/activate
pip install ...
```

There's no need to load the `exfel` module.

### 3.2.3 Updating the source codes

Should there be desired changes in the Toolbox codes, may it be bug fixes or additional features during beamtime, one can freely modify the source codes in the following path: `<PROPOSAL_PATH>/scratch/checkouts/toolbox_<IDENTIFIER>`. The contents of this folder should be a normal git repository. Any changes can be easily done (e.g., editing a line of code, checking out a different branch, etc.) and such changes are immediately reflected on the environment.

## 3.3 Release Notes

### 3.3.1 unreleased

- **Bug fixes**
  - fix issue:75 regarding pulse Id assignment when reading BAM data MR:272
  - fix issue:80 regarding new bunch pattern table size when reading BAM data MR:279
  - fix issue:84 regarding loading only a given amount of BOZ DSSC data in memory MR:288
  - fix issue:86 regarding loading XGM MR:291
  - fix issue:88 regarding loading data from multiple folders MR:303

- **Improvements**
  - update documentation on knife-edge scan and fluence calcuation MR:276, MR:278
  - update scannerY mnemonics MR:281
  - move loading logic to mnemonics MR:283, MR:286
  - add MaranaX mnemonics MR:285
  - add Chem diagnostics and Gotthard II mnemonics MR:292
  - make hRIXS centroid threshold configurable MR:298
  - drop MaranaX non-existing mnemonics and add attributes on dataset MR:300
  - streamline digitizer functions MR:304
  - add BAM average and BAM feedbacks mnemonics, Viking mnemonics MR:305, MR:306
  - improved function to load PES spectra MR:309
  - Gotthard-II mnemonics and pulse alignment MR:310, MR:311
  - Adds `AppleX` polarization mnemonics MR:313
- **New Features**
  - fix issue:75 add frame counts when aggregating RIXS data MR:274
  - BOZ normalization using 2D flat field polylines for S K-edge MR:293
  - BOZ GPU notebook MR:294
  - update hRIXS class for MaranaX MR:287
  - introduce MaranaX class for parallelized centroiding MR:297
  - New PES functions to save / load average traces MR:309

### 3.3.2 1.7.0

- **Bug fixes**
  - fix issue:61 regarding sign of XAS in some cases MR:207
  - Use xarray.values instead of .to_numpy() for backward-compatibility MR:214
  - fix issue:23 regarding API documentation generation MR:221
  - fix issue:64 regarding loading a subset of trains MR:226, MR:230
  - fix issue:67 regarding bunch pattern MR:245
  - fix issue:44 regarding notebook for DSSC geometry quadrants alignment MR:247
  - fix issue:52 use extra-geom detector helper function for azimuthal integration MR:248
  - fix issue:18 regarding using reserved partition with sbatch MR:250
  - fix issue:68 now using findxfel command MR:256
  - fix issue:69 to speed up BOZ Small data and deal with missing intra-darks MR:256
  - fix issue:71 regarding rounding error in BOZ analysis related to timing MR:256
  - fix issue:40 update documentation regarding toolbox deployment folder in scratch MR:268
- **Improvements**

  - remove calls to matplotlib tight_layout MR:206
  - Improved hRIXS class and utilities MR:182
  - Documentation on extracting digitizer peaks, clean up of digitizer functions MR:215
  - Improved peak-finding algorithm for digitizer traces MR:216, MR:227
  - Only load bunch pattern table when necessary MR:234, MR:245
  - Document the HRIXS class MR:238
  - notebook example of DSSC azimuthal integration for time delay scans MR:249
  - provide drop-intra-darks option in BOZ analysis MR:256
  - improve automatic ROIs finding for BOZ analysis MR:256
  - prevent flat field correction from turning negative in BOZ analysis MR:259
  - update documentation to the new exfel-python environment MR:266

- **New Features**
  - Read signal description from Fast ADC and ADQ412 digitizers MR:209, MR:212
  - Mnemonics for XRD devices MR:208
  - Add function to align OL to FEL pulse Id MR:218
  - Add reflectivity routine MR:218
  - Possibility to extract run values of mnemonics MR:220, MR:232
  - Add get_undulator_config function MR:225
  - Document the HRIXS class MR:238
  - Include double counts for hRIXS SPC algorithms MR:239
  - Add Viking spectrometer analysis class MR:240
  - Add GPU accelaration for BOZ correction determination MR:254
  - Issues warning when loading data with > 5% missing trains MR:263

### 3.3.3 1.6.0

- **Bug fixes**
  - fix issue:45 SLURM scripts embedded in and download link available from documentation MR:171
  - fix issue:8 regarding azimuthal integration with pyFAI and hexagonal DSSC pixel splitting by providing an example notebook MR:174
  - fix issue:46 with a change in dask groupby mean behavior MR:174
  - fix issue:47 SLURM script not using the correct kernel MR:176
  - fix issue:51 make sure that BAM units are in ps MR:183
  - fix issue:50 and issue:54 relating to package dependencies
  - fix issue:57 adds target mono energy mnemonic
  - fix issue:55 implementingd dask auto rechunking in notebooks
  - fix issue:53 wrong flat field correction sometimes being calculated

  - fix issue:56 future warning on xarray.ufuncs MR:189

- **Improvements**
  - update version of BAM mnemonics MR:175
  - update version GATT-related mnemonics, add *transmission_col2* MR:172
  - reorganize the Howto section MR:169
  - improve SLURM scripts with named arguments MR:176
  - adds notebook for DSSC fine timing analysis MR:184 and MR:185
  - numerous improvements for the flat field correction calculation in the BOZ analysis, including fitting domain functions, hexagonal pixel lattice, possibility to switch off flat field symmetry constraints and a refine fit function with regularization term MR:186
  - simplifies flat field calculation by using directly the refined fit procedure which works with far fewer input parameters MR:202
- **New Features**
  - add routine for fluence calibration MR:180
  - add Fast ADC 2 mnemonics MR:200

### 3.3.4 1.5.0

- **Bug fixes**
  - fix issue:39 providing a changelog in the documentation MR:164
  - fix issue:37 BOZ analysis MR:158
  - fix issue:36 mnemonics MR:159
  - fix issue:35 BOZ notebook dependencies MR:157
  - fix issue:34 BOZ time delay calculations and plotting MR:154
  - fix issue:32 significantly speeding up in XAS binning calculation MR:151
  - fix issue:27 improving BOZ analysis MR:146
  - fix issue:28 pp pattern in DSSC dask binning MR:144
  - fix issue:26 several BOZ analysis improvements MR:135
- **Improvements**
  - checks if single-version mnemonics is in all_sources MR:163
  - add `get_bam_params()` MR:160
  - only check keys if menmonic has more than one version commit:ae724d3c
  - add FFT focus lens mnemonics MR:156
  - add dask as dependency MR:155
  - renamed FFT sample Z mnemonics MR:153
  - add virtual sample camera LLC_webcam1 into mnemonics MR:152
  - fix digitizer check params MR:149
  - improve installation instruction MR:145

  - add Newton camera MR:142
  - simplified `mnemonics_for_run()` commit:3cc98c16
  - adds Horizontal FDM to mnemonics MR:141
  - add setup documentation MR:140
  - numerous PES fixes MR:143, MR:130, MR:129, MR:138, MR:137
  - change in FFT sample Z mnemonics MR:125 and MR:124
  - add MTE3 camera MR:123
  - add KB benders averager MR:120 and MR:119
- **New Features**
  - implement the Beam-splitting Off-axis Zone-plate analysis: MR:150, MR:139, MR:136, MR:134, MR:133, MR:132, MR:131, MR:128, MR:127, MR:126, MR:115
  - introduce dask assisted DSSC binning, fixing issue:24 and issue:17

### 3.3.5 1.4.0

- **Bug fixes**
  - fix issue:22 using extra-data read machinery MR:105
  - fix issue:21 and issue:12 introducing mnemonics version MR:104
- **Improvements**
  - fix `get_array()`, add wrappers to some of *extra_data* basic functions MR:116
  - new FastADC mnemonics MR:112
  - refactor packaging MR:106
  - add `load_bpt()` function commit:9e2c1107
  - add XTD10 MCP mnemonics commit:8b550c9b
  - add `digitizer_type()` function commit:75eb0bca
  - separate FastADC and ADQ412 code commit:939d32b9
  - documentation centralized on rtd.xfel.eu MR:103
  - simplify digitizer functions and pulseId coordinates assignment for XGM and digitizers MR:100
- **New Features**
  - base knife-edge scan analysis implementation MR:107
  - add PES MR:108
  - integrate documentation to rtd.xfel.eu MR:103 and MR:99

### 3.3.6 1.1.2rc1

- **Bug Fixes**
- **Improvements**
- **New Feature**
  - introduce change in package structure, sphinx documentation and DSSC binning class MR:87

## 3.4 API Reference

This page contains auto-generated API reference documentation[1].

### 3.4.1 toolbox_scs

#### Subpackages

##### toolbox_scs.base

#### Subpackages

##### toolbox_scs.base.tests

#### Submodules

##### toolbox_scs.base.tests.test_knife_edge

#### Module Contents

#### Functions

| |
|---|
| *test_range_mask*() |
| *test_prepare_arrays_nans*() |
| *test_prepare_arrays_size*() |
| *test_prepare_arrays_range*() |
| *test_knife_edge_base*() |
| *_with_values*(array, value[, num]) |

toolbox_scs.base.tests.test_knife_edge.**test_range_mask**()

[1] Created with sphinx-autoapi

toolbox_scs.base.tests.test_knife_edge.**test_prepare_arrays_nans**()

toolbox_scs.base.tests.test_knife_edge.**test_prepare_arrays_size**()

toolbox_scs.base.tests.test_knife_edge.**test_prepare_arrays_range**()

toolbox_scs.base.tests.test_knife_edge.**test_knife_edge_base**()

toolbox_scs.base.tests.test_knife_edge.**_with_values**(*array*, *value*, *num=5*)

### Submodules

toolbox_scs.base.knife_edge

### Module Contents

### Functions

| | |
|---|---|
| *knife_edge*(positions, intensities[, axisRange, p0]) | Calculates the beam radius at 1/e^2 from a knife-edge scan by |
| *knife_edge_base*(positions, intensities[, axisRange, p0]) | The base implementation of the knife-edge scan analysis. |

toolbox_scs.base.knife_edge.**knife_edge**(*positions*, *intensities*, *axisRange=None*, *p0=None*)

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(a,b,u) = a*erfc(u) + b or where u = sqrt(2)*(x-x0)/w0 with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **positions** (*np.ndarray*) – Motor position values, typically 1D
- **intensities** (*np.ndarray*) – Intensity values, could be either 1D or 2D, with the number or rows equivalent with the position size
- **axisRange** (*sequence of two floats or None*) – Edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – Initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 um is assumed.

**Returns**

- **width** (*float*) – The beam radius at 1/e^2
- **std** (*float*) – The standard deviation of the width

toolbox_scs.base.knife_edge.**knife_edge_base**(*positions*, *intensities*, *axisRange=None*, *p0=None*)

The base implementation of the knife-edge scan analysis.

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(a,b,u) = a*erfc(u) + b or where u = sqrt(2)*(x-x0)/w0 with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **positions** (*np.ndarray*) – Motor position values, typically 1D
- **intensities** (*np.ndarray*) – Intensity values, could be either 1D or 2D, with the number or rows equivalent with the position size

- **axisRange** (*sequence of two floats or None*) – Edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – Initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 um is assumed.

**Returns**

- **popt** (*sequence of floats or None*) – The parameters of the resulting fit.
- **pcov** (*sequence of floats*) – The covariance matrix of the resulting fit.

### Package Contents

### Functions

| | |
|---|---|
| *knife_edge* | |
| *knife_edge_base*(positions, intensities[, axisRange, p0]) | The base implementation of the knife-edge scan analysis. |

### Attributes

| |
|---|
| *__all__* |

toolbox_scs.base.**knife_edge**(*positions*, *intensities*, *axisRange=None*, *p0=None*)

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(a,b,u) = a*erfc(u) + b or where u = sqrt(2)*(x-x0)/w0 with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **positions** (*np.ndarray*) – Motor position values, typically 1D
- **intensities** (*np.ndarray*) – Intensity values, could be either 1D or 2D, with the number or rows equivalent with the position size
- **axisRange** (*sequence of two floats or None*) – Edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – Initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 um is assumed.

**Returns**

- **width** (*float*) – The beam radius at 1/e^2
- **std** (*float*) – The standard deviation of the width

toolbox_scs.base.**knife_edge_base**(*positions*, *intensities*, *axisRange=None*, *p0=None*)

The base implementation of the knife-edge scan analysis.

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(a,b,u) = a*erfc(u) + b or where u = sqrt(2)*(x-x0)/w0 with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **positions** (*np.ndarray*) – Motor position values, typically 1D
- **intensities** (*np.ndarray*) – Intensity values, could be either 1D or 2D, with the number or rows equivalent with the position size
- **axisRange** (*sequence of two floats or None*) – Edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – Initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 um is assumed.

**Returns**

- **popt** (*sequence of floats or None*) – The parameters of the resulting fit.
- **pcov** (*sequence of floats*) – The covariance matrix of the resulting fit.

toolbox_scs.base.**__all__**

## toolbox_scs.detectors

### Submodules

## toolbox_scs.detectors.azimuthal_integrator

### Module Contents

### Classes

| |
|---|
| *AzimuthalIntegrator* |
| *AzimuthalIntegratorDSSC* |

**class** toolbox_scs.detectors.azimuthal_integrator.**AzimuthalIntegrator**(*imageshape*, *center*, *polar_range*, *aspect=204 / 236*, ***kwargs*)

Bases: object

**_calc_dist_array**(*shape*, *center*, *aspect*)

Calculate pixel coordinates for the given shape.

**_calc_indices**(***kwargs*)

Calculates the list of indices for the flattened image array.

**_calc_polar_mask**(*polar_range*)

**calc_q**(*distance*, *wavelength*)

Calculate momentum transfer coordinate.

**Parameters**

- **distance** (*float*) – Sample - detector distance in meter
- **wavelength** (*float*) – wavelength of scattered light in meter

**Returns**
**deltaq** – Momentum transfer coordinate in 1/m

**Return type**
np.ndarray

**__call__**(*image*)

**class** toolbox_scs.detectors.azimuthal_integrator.**AzimuthalIntegratorDSSC**(*geom*, *polar_range*, *dxdy=(0, 0)*, ***kwargs*)

Bases: *AzimuthalIntegrator*

**_calc_dist_array**(*geom*, *dxdy*)

Calculate pixel coordinates for the given shape.

### toolbox_scs.detectors.bam_detectors

Beam Arrival Monitor related sub-routines

Copyright (2021) SCS Team.

(contributions preferrably comply with pep8 code structure guidelines.)

## Module Contents

## Functions

| | |
|---|---|
| *get_bam*(run[, mnemonics, merge_with, bunchPattern, ...]) | Load beam arrival monitor (BAM) data and align their pulse ID |
| *get_bam_params*(run[, mnemo_or_source]) | Extract the run values of bamStatus[1-3] and bamError. |

toolbox_scs.detectors.bam_detectors.**get_bam**(*run*, *mnemonics=None*, *merge_with=None*, *bunchPattern='sase3'*, *pulseIds=None*)

Load beam arrival monitor (BAM) data and align their pulse ID according to the bunch pattern. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing data set (merge_with).

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the bam data.
- **mnemonics** (*str or list of str*) – mnemonics for BAM, e.g. "BAM1932M" or ["BAM414", "BAM1932M"]. the arrays are either taken from merge_with or loaded from the DataCollection run.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. If merge_with contains variables in mnemonics, they will be selected, aligned and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks. The pulse ID dimension will be named 'sa1_pId', 'sa3_pId' or 'ol_pId', respectively.
- **pulseIds** (*list, 1D array*) – Pulse Ids. If None, they are automatically loaded.

**Returns**
merged with Dataset *merge_with* if provided.

**Return type**
xarray Dataset with pulse-resolved BAM variables aligned,

### Example

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2711, 303)
>>> bam = tb.get_bam(run, 'BAM1932S')
```

toolbox_scs.detectors.bam_detectors.**get_bam_params**(*run*, *mnemo_or_source='BAM1932S'*)

Extract the run values of bamStatus[1-3] and bamError.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the bam data.
- **mnemo_or_source** (*str*) – mnemonic of the BAM, e.g. 'BAM414', or source name, e.g. 'SCS_ILH_LAS/DOOCS/BAM_414_B2.

**Returns**
**params** – dictionnary containing the extracted parameters.

**Return type**
dict

**Note:** The extracted parameters are run values, they do not reflect any possible change during the run.

## toolbox_scs.detectors.digitizers

Digitizers related sub-routines

Copyright (2021) SCS Team.

(contributions preferrably comply with pep8 code structure guidelines.)

### Module Contents

### Functions

| | |
|---|---|
| get_peaks(run, data, mnemonic[, useRaw, autoFind, ...]) | Extract peaks from one source (channel) of a digitizer. |
| get_dig_avg_trace(run, mnemonic[, ntrains]) | Compute the average over ntrains evenly spaced accross all trains |
| check_peak_params(run, mnemonic[, raw_trace, ntrains, ...]) | Checks and plots the peak parameters (pulse window and baseline window |
| get_tim_peaks(run[, mnemonic, merge_with, ...]) | Automatically computes TIM peaks. Sources can be loaded on the |
| get_laser_peaks(run[, mnemonic, merge_with, ...]) | Extracts laser photodiode signal (peak intensity) from Fast ADC |
| get_digitizer_peaks(run, mnemonic[, merge_with, ...]) | Automatically computes digitizer peaks. A source can be loaded on the |
| digitizer_signal_description(run[, digitizer]) | Check for the existence of signal description and return all corresponding |

toolbox_scs.detectors.digitizers.**get_peaks**(*run*, *data*, *mnemonic*, *useRaw=True*, *autoFind=True*, *integParams=None*, *bunchPattern='sase3'*, *bpt=None*, *extra_dim=None*, *indices=None*)

Extract peaks from one source (channel) of a digitizer.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data
- **data** (*xarray DataArray or str*) – array containing the raw traces or peak-integrated values from the digitizer. If str, must be one of the ToolBox mnemonics. If None, the data is loaded via the source and key arguments.
- **mnemonic** (*str or dict*) – ToolBox mnemonic or dict with source and key as in {'source': 'SCS_UTC1_ADQ/ADC/1:network', 'key': 'digitizers.channel_1_A.raw.samples'}
- **useRaw** (*bool*) – If True, extract peaks from raw traces. If False, uses the APD (or peaks) data from the digitizer.
- **autoFind** (*bool*) – If True, finds integration parameters by inspecting the average raw trace. Only valid if useRaw is True.
- **integParams** (*dict*) – dictionnary containing the integration parameters for raw trace integration: 'pulseStart', 'pulseStop', 'baseStart', 'baseStop', 'period', 'npulses'. Not used if autoFind is True. All keys are required when bunch pattern is missing.
- **bunchPattern** (*str*) – match the peaks to the bunch pattern: 'sase1', 'sase3', 'scs_ppl'. This will dictate the name of the pulse ID coordinates: 'sa1_pId', 'sa3_pId' or 'scs_ppl'.
- **bpt** (*xarray DataArray*) – bunch pattern table
- **extra_dim** (*str*) – Name given to the dimension along the peaks. If None, the name is given according to the bunchPattern.
- **indices** (*array, slice*) – indices from the peak-integrated data to retrieve. Only required when bunch pattern is missing and useRaw is False.

**Return type**

xarray.DataArray containing digitizer peaks with pulse coordinates

toolbox_scs.detectors.digitizers.**get_dig_avg_trace**(*run*, *mnemonic*, *ntrains=None*)

Compute the average over ntrains evenly spaced accross all trains of a digitizer trace.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – ToolBox mnemonic of the digitizer data, e.g. 'MCP2apd'.
- **ntrains** (*int*) – Number of trains used to calculate the average raw trace. If None, all trains are used.

**Returns**

**trace** – The average digitizer trace

**Return type**

DataArray

toolbox_scs.detectors.digitizers.**check_peak_params**(*run*, *mnemonic*, *raw_trace=None*, *ntrains=200*, *params=None*, *plot=True*, *show_all=False*, *bunchPattern='sase3'*)

Checks and plots the peak parameters (pulse window and baseline window of a raw digitizer trace) used to compute the peak integration. These parameters are either set by the digitizer peak-integration settings, or are determined by a peak finding algorithm (used in get_tim_peaks or get_fast_adc_peaks) when the inputs are raw traces. The parameters can also be provided manually for visual inspection. The plot either shows the first and last pulse of the trace or the entire trace.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – ToolBox mnemonic of the digitizer data, e.g. 'MCP2apd'.
- **raw_trace** (*optional, 1D numpy array or xarray DataArray*) – Raw trace to display. If None, the average raw trace over ntrains of the corresponding channel is loaded (this can be time-consuming).
- **ntrains** (*optional, int*) – Only used if raw_trace is None. Number of trains used to calculate the average raw trace of the corresponding channel.
- **plot** (*bool*) – If True, displays the raw trace and peak integration regions.
- **show_all** (*bool*) – If True, displays the entire raw trace and all peak integration regions (this can be time-consuming). If False, shows the first and last pulse according to the bunchPattern.
- **bunchPattern** (*optional, str*) – Only used if plot is True. Checks the bunch pattern against the digitizer peak parameters and shows potential mismatch.

**Return type**

dictionnary of peak integration parameters

toolbox_scs.detectors.digitizers.**get_tim_peaks**(*run*, *mnemonic=None*, *merge_with=None*, *bunchPattern='sase3'*, *integParams=None*, *keepAllSase=False*)

Automatically computes TIM peaks. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing data set (merge_with). The bunch pattern table is used to assign the pulse id coordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonics for TIM, e.g. "MCP2apd".

- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The TIM variables of merge_with (if any) will also be computed and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks. The pulse ID dimension will be named 'sa1_pId', 'sa3_pId' or 'ol_pId', respectively.
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.
- **keepAllSase** (*bool*) – Only relevant in case of sase-dedicated trains. If True, all trains are kept, else only those of the bunchPattern are kept.

**Returns**

- *xarray Dataset with TIM variables substituted by*
- *the peak caclulated values (e.g. "MCP2raw" becomes*
- *"MCP2peaks"), merged with Dataset *merge_with* if provided.*

toolbox_scs.detectors.digitizers.**get_laser_peaks**(*run*, *mnemonic=None*, *merge_with=None*, *bunchPattern='scs_ppl'*, *integParams=None*)

Extracts laser photodiode signal (peak intensity) from Fast ADC digitizer. Sources can be loaded on the fly via the mnemonics argument, and/or processed from an existing data set (merge_with). The PP laser bunch pattern is used to assign the pulse idcoordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonic for FastADC corresponding to laser signal, e.g. "FastADC2peaks" or 'I0_ILHraw'.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The FastADC variables of merge_with (if any) will also be computed and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks.
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.

**Returns**

- *xarray Dataset with all Fast ADC variables substituted by*
- *the peak caclulated values (e.g. "FastADC2raw" becomes*
- *"FastADC2peaks").*

toolbox_scs.detectors.digitizers.**get_digitizer_peaks**(*run*, *mnemonic*, *merge_with=None*, *bunchPattern='sase3'*, *integParams=None*, *keepAllSase=False*)

Automatically computes digitizer peaks. A source can be loaded on the fly via the mnemonic argument, or processed from an existing data set (merge_with). The bunch pattern table is used to assign the pulse id coordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonic for FastADC or ADQ412, e.g. "I0_ILHraw" or "MCP3apd". The data is either loaded from the DataCollection or taken from merge_with.

- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one.
- **bunchPattern** (*str or dict*) – 'sase1' or 'sase3' or 'scs_ppl', 'None': bunch pattern
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.
- **keepAllSase** (*bool*) – Only relevant in case of sase-dedicated trains. If True, all trains are kept, else only those of the bunchPattern are kept.

**Returns**

- *xarray Dataset with digitizer peak variables. Raw variables are*
- *substituted by the peak caclulated values (e.g. "FastADC2raw" becomes*
- *"FastADC2peaks").*

toolbox_scs.detectors.digitizers.**digitizer_signal_description**(*run*, *digitizer=None*)

Check for the existence of signal description and return all corresponding channels in a dictionnary.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **digitizer** (*str or list of str (default=None)*) – Name of the digitizer: one in ['FastADC', FastADC2', 'ADQ412', 'ADQ412_2] If None, all digitizers are used

**Returns**
**signal_description** – the digitizer channels.

**Return type**
dictionnary containing the signal description of

### Example

import toolbox_scs as tb run = tb.open_run(3481, 100) signals = tb.digitizer_signal_description(run) signals_fadc2 = tb.digitizer_signal_description(run, 'FastADC2')

## toolbox_scs.detectors.dssc

DSSC-detector class module

The dssc detector class. It represents a namespace for frequent evaluation while implicitly applying/requiring certain structure/naming conventions to its objects.

**comments:**

- contributions should comply with pep8 code structure guidelines.
- Plot routines don't fit into objects since they are rather fluent. They have been outsourced to dssc_plot.py. They can now be accessed as toolbox_scs member functions.

### Module Contents

### Classes

| |
|---|
| *DSSCBinner* |
| *DSSCFormatter* |

**class** toolbox_scs.detectors.dssc.**DSSCBinner**(*proposal_nr*, *run_nr*, *binners={}*, *xgm_name='SCS_SA3'*, *tim_names=['MCP1apd', 'MCP2apd', 'MCP3apd']*, *dssc_coords_stride=2*)

**__del__**()

**add_binner**(*name*, *binner*)

Add additional binner to internal dictionary

**Parameters**

- **name** (*str*) – name of binner to be created
- **binner** (*xarray.DataArray*) – An array that represents a map how the respective coordinate should be binned.

**Raises**

*ToolBoxValueError* – Exception: Raises exception in case the name does not correspond to a valid binner name. To be generalized.

**load_xgm**()

load xgm data and construct coordinate array according to corresponding dssc frame number.

**load_tim**()

load tim data and construct coordinate array according to corresponding dssc frame number.

**create_pulsemask**(*use_data='xgm'*, *threshold=(0, np.inf)*)

creates a mask for dssc frames according to measured xgm intensity. Once such a mask has been constructed, it will be used in the data reduction process to drop out-of-bounds pulses.

**get_info**()

Returns the expected shape of the binned dataset, in case binners have been defined.

**_bin_metadata**(*data*)

**get_xgm_binned**()

Bin the xgm data according to the binners of the dssc data. The result can eventually be merged into the final dataset by the DSSCFormatter.

**Returns**

**xgm_data** – xarray dataset containing the binned xgm data

**Return type**

xarray.DataSet

**get_tim_binned**()

Bin the tim data according to the binners of the dssc data. The result can eventually be merged into the final dataset by the DSSCFormatter.

**Returns**
**tim_data** – xarray dataset containing the binned tim data

**Return type**
xarray.DataSet

**process_data**(*modules=[]*, *filepath='./'*, *chunksize=512*, *backend='loky'*, *n_jobs=None*, *dark_image=None*, *xgm_normalization=False*, *normevery=1*)

Load and bin dssc data according to self.bins. No data is returned by this method. The condensed data is written to file by the worker processes directly.

**Parameters**

- **modules** (*list of ints*) – a list containing the module numbers that should be processed. If empty, all modules are processed.
- **filepath** (*str*) – the path where the files containing the reduced data should be stored.
- **chunksize** (*int*) – The number of trains that should be read in one iterative step.
- **backend** (*str*) – joblib multiprocessing backend to be used. At the moment it can be any of joblibs standard backends: 'loky' (default), 'multiprocessing', 'threading'. Anything else than the default is experimental and not appropriately implemented in the dbdet member function 'bin_data'.
- **n_jobs** (*int*) – inversely proportional of the number of cpu's available for one job. Tasks within one job can grab a maximum of n_CPU_tot/n_jobs of cpu's. Note that when using the default backend there is no need to adjust this parameter with the current implementation.
- **dark_image** (*xarray.DataArray*) – DataArray with dimensions compatible with the loaded dssc data. If given, it will be subtracted from the dssc data before the binning. The dark image needs to be of dimension module, trainId, pulse, x and y.
- **xgm_normalization** (*boolean*) – if true, the dssc data is normalized by the xgm data before the binning.
- **normevery** (*int*) – integer indicating which out of normevery frame will be normalized.

**class** toolbox_scs.detectors.dssc.**DSSCFormatter**(*filepath*)

**combine_files**(*filenames=[]*)

Read the files given in filenames, and store the data in the class variable 'data'. If no filenames are given, it tries to read the files stored in the class-internal variable '_filenames'.

**Parameters**
**filenames** (*list*) – list of strings containing the names of the files to be combined.

**add_dataArray**(*groups=[]*)

Reads addional xarray-data from the first file given in the list of filenames. This assumes that all the files in the folder contain the same additional data. To be generalized.

**Parameters**
**groups** (*list*) – list of strings with the names of the groups in the h5 file, containing additional xarray data.

**add_attributes**(*attributes={}*)

Add additional information, such as run-type, as attributes to the formatted .h5 file.

**Parameters**
**attributes** (*dictionary*) – a dictionary, containing information or data of any kind, that will be added to the formatted .h5 file as attributes.

**save_formatted_data**(*filename*)

Create a .h5 file containing the main dataset in the group called 'data'. Additional groups will be created for the content of the variable 'data_array'. Metadata about the file is added in the form of attributes.

**Parameters**

**filename** (*str*) – the name of the file to be created

## toolbox_scs.detectors.dssc_data

### Module Contents

### Functions

| | |
|---|---|
| *save_xarray*(fname, data[, group, mode]) | Store xarray Dataset in the specified location |
| *save_attributes_h5*(fname[, data]) | Adding attributes to a hdf5 file. This function is intended to be used to |
| *load_xarray*(fname[, group, form]) | Load stored xarray Dataset. |
| *get_data_formatted*([filenames, data_list]) | Combines the given data into one dataset. For any of extra_data's data |

toolbox_scs.detectors.dssc_data.**save_xarray**(*fname*, *data*, *group='data'*, *mode='a'*)

Store xarray Dataset in the specified location

**Parameters**

- **data** (*xarray.DataSet*) – The data to be stored
- **fname** (*str, int*) – filename
- **overwrite** (*bool*) – overwrite existing data

**Raises**

*ToolBoxFileError* – Exception: File existed, but overwrite was set to False.

toolbox_scs.detectors.dssc_data.**save_attributes_h5**(*fname*, *data={}*)

Adding attributes to a hdf5 file. This function is intended to be used to attach metadata to a processed run.

**Parameters**

- **fname** (*str*) – filename as string
- **data** (*dictionary*) – the data that should be added to the file in form of a dictionary.

toolbox_scs.detectors.dssc_data.**load_xarray**(*fname*, *group='data'*, *form='dataset'*)

Load stored xarray Dataset. Comment: This function exists because of a problem with the standard netcdf engine that is malfunctioning due to related software installed in the exfel-python environment. May be dropped at some point.

**Parameters**

- **fname** (*str*) – filename as string
- **group** (*str*) – the name of the xarray dataset (group in h5 file).
- **form** (*str*) – specify whether the data to be loaded is a 'dataset' or a 'array'.

toolbox_scs.detectors.dssc_data.**get_data_formatted**(*filenames=[]*, *data_list=[]*)

Combines the given data into one dataset. For any of extra_data's data types, an xarray.Dataset is returned. The data is sorted along the 'module' dimension. The array dimension have the order 'trainId', 'pulse', 'module', 'x', 'y'. This order is required by the extra_geometry package.

**Parameters**

- **filenames** (*list of str*) – files to be combined as a list of names. Calls '_data_from_list' to actually load the data.
- **data_list** (*list*) – list containing the already loaded data

**Returns**
**data** – A xarray.Dataset containing the combined data.

**Return type**
xarray.Dataset

## toolbox_scs.detectors.dssc_misc

DSSC-related sub-routines.

comment: contributions should comply with pep8 code structure guidelines.

### Module Contents

### Functions

| | |
|---|---|
| *load_dssc_info*(proposal, run_nr) | Loads the first data file for DSSC module 0 (this is hard-coded) |
| *create_dssc_bins*(name, coordinates, bins) | Creates a single entry for the dssc binner dictionary. The produced xarray |
| *get_xgm_formatted*(run_obj, xgm_name, dssc_frame_coords) | Load the xgm data and define coordinates along the pulse dimension. |
| *quickmask_DSSC_ASIC*(poslist) | Returns a mask for the given DSSC geometry with ASICs given in poslist |
| *load_mask*(fname, dssc_mask) | Load a DSSC mask file. |

toolbox_scs.detectors.dssc_misc.**load_dssc_info**(*proposal*, *run_nr*)

Loads the first data file for DSSC module 0 (this is hardcoded) and returns the detector_info dictionary

**Parameters**

- **proposal** (*str, int*) – number of proposal
- **run_nr** (*str, int*) – number of run

**Returns**
**info** – {'dims': tuple, 'frames_per_train': int, 'total_frames': int}

**Return type**
dictionary

toolbox_scs.detectors.dssc_misc.**create_dssc_bins**(*name*, *coordinates*, *bins*)

Creates a single entry for the dssc binner dictionary. The produced xarray data-array will later be used to perform grouping operations according to the given bins.

**Parameters**

- **name** (*str*) – name of the coordinate to be binned.
- **coordinates** (*numpy.ndarray*) – the original coordinate values (1D)
- **bins** (*numpy.ndarray*) – the bins according to which the corresponding dimension should be grouped.

**Returns**

**da** – A pre-formatted xarray.DataArray relating the specified dimension with its bins.

**Return type**

xarray.DataArray

### Examples

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2212, 235, include='*DA*')
```

1.) binner along 'pulse' dimension. Group data into two bins. >>> bins_pulse = ['pumped', 'unpumped'] * 10 >>> binner_pulse = tb.create_dssc_bins("pulse",

np.linspace(0,19,20, dtype=int), bins_pulse)

**2.) binner along 'train' dimension. Group data into bins corresponding**
to the positions of a delay stage for instance.

```
>>> bins_trainId = tb.get_array(run, 'PP800_PhaseShifter', 0.04)
>>> binner_train = tb.create_dssc_bins("trainId",
                           run.trainIds,
                           bins_trainId.values)
```

toolbox_scs.detectors.dssc_misc.**get_xgm_formatted**(*run_obj*, *xgm_name*, *dssc_frame_coords*)

Load the xgm data and define coordinates along the pulse dimension.

**Parameters**

- **run_obj** (*extra_data.DataCollection*) – DataCollection object providing access to the xgm data to be loaded
- **xgm_name** (*str*) – valid mnemonic of a xgm source
- **dssc_frame_coords** (*int, list*) – defines which dssc frames should be normalized using data from the xgm.

**Returns**

**xgm** – xgm data with coordinate 'pulse'.

**Return type**

xarray.DataArray

toolbox_scs.detectors.dssc_misc.**quickmask_DSSC_ASIC**(*poslist*)

Returns a mask for the given DSSC geometry with ASICs given in poslist blanked. poslist is a list of (module, row, column) tuples. Each module consists of 2 rows and 8 columns of individual ASICS.

Copyright (c) 2019, Michael Schneider Copyright (c) 2020, SCS-team license: BSD 3-Clause License (see LICENSE_BSD for more info)

toolbox_scs.detectors.dssc_misc.**load_mask**(*fname*, *dssc_mask*)

Load a DSSC mask file.

Copyright (c) 2019, Michael Schneider Copyright (c) 2020, SCS-team license: BSD 3-Clause License (see LICENSE_BSD for more info)

**Parameters**
**fname** (*str*) – string of the filename of the mask file

**Return type**
dssc_mask

## toolbox_scs.detectors.dssc_plot

DSSC visualization routines

Plotting sub-routines frequently done in combination with dssc analysis. The initial code is based on: https://github.com/dscran/dssc_process /blob/master/example_image_process_pulsemask.ipynb

Todo: For visualization of statistical information we could eventually switch to seaborn: https://seaborn.pydata.org/

### Module Contents

### Functions

| | |
|---|---|
| *plot_xgm_threshold*(xgm[, xgm_min, xgm_max, run_nr, ...]) | |
| *plot_binner*(binner[, yname, xname, run_nr]) | |
| *plot_binner_hist*(binner[, dname, run_nr]) | |
| *plot_hist_processed*(hist_data) | |

toolbox_scs.detectors.dssc_plot.**plot_xgm_threshold**(*xgm*, *xgm_min=None*, *xgm_max=None*, *run_nr=''*, *safe_fig=False*)

toolbox_scs.detectors.dssc_plot.**plot_binner**(*binner*, *yname='data'*, *xname='data'*, *run_nr=''*)

toolbox_scs.detectors.dssc_plot.**plot_binner_hist**(*binner*, *dname='data'*, *run_nr=''*)

toolbox_scs.detectors.dssc_plot.**plot_hist_processed**(*hist_data*)

### toolbox_scs.detectors.dssc_processing

DSSC-related sub-routines.

comment: contributions should comply with pep8 code structure guidelines.

## Module Contents

## Functions

| | |
|---|---|
| *process_dssc_data*(proposal, run_nr, module, chunksize, ...) | Collects and reduces DSSC data for a single module. |

toolbox_scs.detectors.dssc_processing.**process_dssc_data**(*proposal*, *run_nr*, *module*, *chunksize*, *info*, *dssc_binners*, *path='./'*, *pulsemask=None*, *dark_image=None*, *xgm_mnemonic='SCS_SA3'*, *xgm_normalization=False*, *normevery=1*)

Collects and reduces DSSC data for a single module.

Copyright (c) 2020, SCS-team

**Parameters**

- **proposal** (*int*) – proposal number
- **run_nr** (*int*) – run number
- **module** (*int*) – DSSC module to process
- **chunksize** (*int*) – number of trains to load simultaneously
- **info** (*dictionary*) – dictionary containing keys 'dims', 'frames_per_train', 'total_frames', 'trainIds', 'number_of_trains'.
- **dssc_binners** (*dictionary*) – a dictionary containing binner objects created by the ToolBox member function "create_binner()"
- **path** (*str*) – location in which the .h5 files, containing the binned data, should be stored.
- **pulsemask** (*numpy.ndarray*) – array of booleans to be used to mask dssc data according to xgm data.
- **dark_image** (*xarray.DataArray*) – an xarray dataarray with matching coordinates with the loaded data. If dark_image is not None it will be subtracted from each individual dssc frame.
- **xgm_normalization** (*bool*) – true if the data should be divided by the corresponding xgm value.
- **xgm_mnemonic** (*str*) – Mnemonic of the xgm data to be used for normalization.
- **normevery** (*int*) – One out of normevery dssc frames will be normalized.

**Returns**

**module_data** – xarray datastructure containing data binned according to bins.

**Return type**

xarray.Dataset

toolbox_scs.detectors.fccd

## Module Contents

## Classes

| |
|---|
| *FastCCD* |

## Functions

| |
|---|
| *process_one_module*(job) |

class toolbox_scs.detectors.fccd.**FastCCD**(*proposal*, *distance=1*, *raw=False*)

**__del__**()

**open_run**(*run_nr*, *isDark=False*, *t0=0.0*)

Open a run with extra-data and prepare the virtual dataset for multiprocessing

**inputs:**

run_nr: the run number isDark: True if the run is a dark run t0: optional t0 in mm

**load_gain**(*fname*)

Load a gain map by giving the filename

**collect_fastccd_file**()

Collect the raw fastCCD h5 files.

**define_scan**(*vname*, *bins*)

Prepare the binning of the FastCCD data.

**inputs:**

**vname: variable name for the scan, can be a mnemonic string from ToolBox**
or a dictionnary with ['source', 'key'] fields

bins: step size (or bins_edge but not yet implemented)

**plot_scan**()

Plot a previously defined scan to see the scan range and the statistics.

**plot_xgm_hist**(*nbins=100*)

Plots an histogram of the SCS XGM dedicated SAS3 data.

**inputs:**

nbins: number of the bins for the histogram.

**xgm_filter**(*xgm_low=-np.inf*, *xgm_high=np.inf*)

Filters the data by train. If one pulse within a train has an SASE3 SCS XGM value below xgm_low or above xgm_high, that train will be dropped from the dataset.

**inputs:**

xgm_low: low threshold value xgm_high: high threshold value

**load_mask**(*fname*, *plot=True*)

Load a FastCCD mask file.

**input:**

fname: string of the filename of the mask file plot: if True, the loaded mask is plotted

**binning**()

Bin the FastCCD data by the predifined scan type (FastCCD.define()) using multiprocessing

**save**(*save_folder=None*, *overwrite=False*)

Save the crunched data.

**inputs:**

save_folder: string of the fodler where to save the data. overwrite: boolean whether or not to overwrite existing files.

**load_binned**(*runNB*, *dark_runNB=None*, *xgm_norm=True*, *save_folder=None*)

load previously binned (crunched) FastCCD data by FastCCD.crunch() and FastCCD.save()

**inputs:**

runNB: run number to load dark_runNB: run number of the corresponding dark xgm_norm: normlize by XGM data if True save_folder: path string where the crunched data are saved

**plot_FastCCD**(*use_mask=True*, *p_low=1*, *p_high=98*, *vmin=None*, *vmax=None*)

Plot pumped and unpumped FastCCD images.

**inputs:**

use_mask: if True, a mask is applied on the FastCCD. p_low: low percentile value to adjust the contrast scale on the unpumped and pumped image p_high: high percentile value to adjust the contrast scale on the unpumped and pumped image vmin: low value of the image scale vmax: high value of the image scale

**azimuthal_int**(*wl*, *center=None*, *angle_range=[0, 180 - 1e-06]*, *dr=1*, *use_mask=True*)

Perform azimuthal integration of 1D binned FastCCD run.

**inputs:**

wl: photon wavelength center: center of integration angle_range: angles of integration dr: dr use_mask: if True, use the loaded mask

**plot_azimuthal_int**(*kind='difference'*, *lim=None*)

Plot a computed azimuthal integration.

**inputs:**

kind: (str) either 'difference' or 'relative' to change the type of plot.

**plot_azimuthal_line_cut**(*data*, *qranges*, *qwidths*)

Plot line scans on top of the data.

**inputs:**

data: an azimuthal integrated xarray DataArray with 'delta_q (1/nm)' as one of its dimension. qranges: a list of q-range qwidth: a list of q-width, same length as qranges

toolbox_scs.detectors.fccd.**process_one_module**(*job*)

### toolbox_scs.detectors.gotthard2

Gotthard-II detector related sub-routines



(contributions preferrably comply with pep8 code structure guidelines.)

#### Module Contents

#### Functions

| | |
|---|---|
| *extract_GH2*(ds, run[, firstFrame, bunchPattern, gh2_dim]) | Select and align the frames of the Gotthard-II that have been exposed |

toolbox_scs.detectors.gotthard2.**extract_GH2**(*ds*, *run*, *firstFrame=0*, *bunchPattern='scs_ppl'*, *gh2_dim='gh2_pId'*)

Select and align the frames of the Gotthard-II that have been exposed to light.

**Parameters**

- **ds** (*xarray.Dataset*) – The dataset containing GH2 data
- **run** (*extra_data.DataCollection*) – The run containing the bunch pattern source
- **firstFrame** (*int*) – The GH2 frame number corresponding to the first pulse of the train.
- **bunchPattern** (*str in ['scs_ppl', 'sase3']*) – the bunch pattern used to align data. For 'scs_ppl', the gh2_pId dimension in renamed 'ol_pId', and for 'sase3' gh2_pId is renamed 'sa3_pId'.
- **gh2_dim** (*str*) – The name of the dimension that corresponds to the Gotthard-II frames.

**Returns**

**nds** – The aligned and reduced dataset with only-data-containing GH2 variables.

**Return type**

xarray Dataset

### toolbox_scs.detectors.hrixs

#### Module Contents

#### Classes

| | |
|---|---|
| *hRIXS* | The hRIXS analysis, especially curvature correction |
| *MaranaX* | A spin-off of the hRIXS class: with parallelized centroiding |

**class** toolbox_scs.detectors.hrixs.**hRIXS**(*proposalNB*, *detector='MaranaX'*)

The hRIXS analysis, especially curvature correction

The objects of this class contain the meta-information about the settings of the spectrometer, not the actual data, except possibly a dark image for background subtraction.

The actual data is loaded into `xarray`s, and stays there.

**PROPOSAL**

the number of the proposal

**Type**
int

**DETECTOR**

the detector to be used. Can be ['hRIXS_det', 'MaranaX'] defaults to 'hRIXS_det' for backward-compatibility.

**Type**
str

**X_RANGE**

the slice to take in the dispersive direction, in pixels. Defaults to the entire width.

**Type**
slice

**Y_RANGE**

the slice to take in the energy direction

**Type**
slice

**THRESHOLD**

pixel counts above which a hit candidate is assumed, for centroiding. use None if you want to give it in standard deviations instead.

**Type**
float

**STD_THRESHOLD**

same as THRESHOLD, in standard deviations.

**DBL_THRESHOLD**

threshold controling whether a detected hit is considered to be a double hit.

**BINS**

the number of bins used in centroiding

**Type**
int

**CURVE_A, CURVE_B**

the coefficients of the parabola for the curvature correction

**Type**
float

**USE_DARK**

whether to do dark subtraction. Is initially *False*, magically switches to *True* if a dark has been loaded, but may be reset.

**Type**
bool

**ENERGY_INTERCEPT, ENERGY_SLOPE**

The calibration from pixel to energy

**FIELDS**

the fields to be loaded from the data. Add additional fields if so desired.

### Example

proposal = 3145 h = hRIXS(proposal) h.Y_RANGE = slice(700, 900) h.CURVE_B = -3.695346575286939e-07 h.CURVE_A = 0.024084479232443695 h.ENERGY_SLOPE = 0.018387 h.ENERGY_INTERCEPT = 498.27 h.STD_THRESHOLD = 3.5

**DETECTOR_FIELDS**

**aggregators**

**set_params**(***params*)

**get_params**(**params*)

**from_run**(*runNB*, *proposal=None*, *extra_fields=()*, *drop_first=False*, *subset=None*)

load a run

Load the run *runNB*. A thin wrapper around *toolbox.load*. :param drop_first: if True, the first image in the run is removed from the dataset. :type drop_first: bool

### Example

data = h.from_run(145) # load run 145

data1 = h.from_run(145) # load run 145 data2 = h.from_run(155) # load run 155 data = xarray.concat([data1, data2], 'trainId') # combine both

**load_dark**(*runNB*, *proposal=None*)

load a dark run

Load the dark run *runNB* from *proposal*. The latter defaults to the current proposal. The dark is stored in this *hRIXS* object, and subsequent analyses use it for background subtraction.

### Example

h.load_dark(166) # load dark run 166

**find_curvature**(*runNB*, *proposal=None*, *plot=True*, *args=None*, ***kwargs*)

find the curvature correction coefficients

The hRIXS has some abberations which leads to the spectroscopic lines being curved on the detector. We approximate these abberations with a parabola for later correction.

Load a run and determine the curvature. The curvature is set in *self*, and returned as a pair of floats.

**Parameters**

- **runNB** (*int*) – the run number to use
- **proposal** (*int*) – the proposal to use, default to the current proposal
- **plot** (*bool*) – whether to plot the found curvature onto the data

- **args** (*pair of float, optional*) – a starting value to prime the fitting routine

### Example

h.find_curvature(155) # use run 155 to fit the curvature

**centroid_one**(*image*)

find the position of photons with sub-pixel precision

A photon is supposed to have hit the detector if the intensity within a 2-by-2 square exceeds a threshold. In this case the position of the photon is calculated as the center-of-mass in a 4-by-4 square.

Return the list of x, y coordinate pairs, corrected by the curvature.

**centroid_two**(*image*, *energy*)

determine position of photon hits on detector

The algrothm is taken from the ESRF RIXS toolbox. The thresholds for determining photon hits are given by the incident photon energy

The function returns arrays containing the single and double hits as x and y coordinates

**centroid**(*data*, *bins=None*, *method='auto'*)

calculate a spectrum by finding the centroid of individual photons

This takes the *xarray.Dataset data* and returns a copy of it, with a new *xarray.DataArray* named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

Added a key for switching between algorithims choices are "auto" and "manual" which selects for method for determining whether thresholds there is a photon hit. It changes whether centroid_one or centroid_two is used.

### Example

h.centroid(data) # find photons in all images of the run data.spectrum[0, :].plot() # plot the spectrum of the first image

**parabola**(*x*)

**integrate**(*data*)

calculate a spectrum by integration

This takes the *xarray data* and returns a copy of it, with a new dataarray named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

First the energy that corresponds to each pixel is calculated. Then all pixels within an energy range are summed, where the intensity of one pixel is distributed among the two energy ranges the pixel spans, proportionally to the overlap between the pixel and bin energy ranges.

The resulting data is normalized to one pixel, so the average intensity that arrived on one pixel.

### Example

h.integrate(data) # create spectrum by summing pixels data.spectrum[0, :].plot() # plot the spectrum of the first image

**aggregator**(*da*, *dim*)

**aggregate**(*ds*, *var=None*, *dim='trainId'*)

aggregate (i.e. mostly sum) all data within one dataset

take all images in a dataset and aggregate them and their metadata. For images, spectra and normalizations that means adding them, for others (e.g. delays) adding would not make sense, so we treat them properly. The aggregation functions of each variable are defined in the aggregators attribute of the class. If var is specified, group the dataset by var prior to aggregation. A new variable "counts" gives the number of frames aggregated in each group.

**Parameters**

- **ds** (*xarray Dataset*) – the dataset containing RIXS data
- **var** (*string*) – One of the variables in the dataset. If var is specified, the dataset is grouped by var prior to aggregation. This is useful for sorting e.g. a dataset that contains multiple delays.
- **dim** (*string*) – the dimension over which to aggregate the data

### Example

h.centroid(data) # create spectra from finding photons agg = h.aggregate(data) # sum all spectra agg.spectrum.plot() # plot the resulting spectrum

agg2 = h.aggregate(data, 'hRIXS_delay') # group data by delay agg2.spectrum[0, :].plot() # plot the spectrum for first value

**aggregate_ds**(*ds*, *dim='trainId'*)

**normalize**(*data*, *which='hRIXS_norm'*)

Adds a 'normalized' variable to the dataset defined as the ration between 'spectrum' and 'which'

**Parameters**

- **data** (*xarray Dataset*) – the dataset containing hRIXS data
- **which** (*string, default="hRIXS_norm"*) – one of the variables of the dataset, usually "hRIXS_norm" or "counts"

**class** toolbox_scs.detectors.hrixs.**MaranaX**(**args*, ***kwargs*)

Bases: *hRIXS*

A spin-off of the hRIXS class: with parallelized centroiding

**NUM_MAX_HITS = 30**

**centroid**(*data*, *bins=None*, ***kwargs*)

calculate a spectrum by finding the centroid of individual photons

This takes the *xarray.Dataset data* and returns a copy of it, with a new *xarray.DataArray* named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

Added a key for switching between algorithims choices are "auto" and "manual" which selects for method for determining whether thresholds there is a photon hit. It changes whether centroid_one or centroid_two is used.

#### Example

h.centroid(data) # find photons in all images of the run data.spectrum[0, :].plot() # plot the spectrum of the first image

**_centroid_tb_map**(_, *index*, *data*)

**_centroid_map**(*index*, *, *image*, *energy*)

**_centroid_task**(*index*, *image*, *energy*)

**_histogram_task**(*index*, *total*, *double*, *default_range*)

**centroid_from_run**(*runNB*, *proposal=None*, *extra_fields=()*, *drop_first=False*, *subset=None*, *bins=None*, *return_hits=False*)

A combined function of *from_run()* and *centroid()*, which uses *extra_data* and *pasha* to avoid bulk loading of files.

**_centroid_ed_map**(_, *index*, *trainId*, *data*)

**static _mnemo_to_prop**(*mnemo*)

**_is_mnemo_in_run**(*mnemo*, *run*)

### toolbox_scs.detectors.pes

Beam Arrival Monitor related sub-routines



(contributions preferrably comply with pep8 code structure guidelines.)

#### Module Contents

#### Functions

| | |
|---|---|
| *get_pes_tof*(proposal, runNB, mnemonic[, start, ...]) | Extracts time-of-flight spectra from raw digitizer traces. The spectra |
| *get_pes_params*(run[, channel]) | Extract PES parameters for a given extra_data DataCollection. |
| *save_pes_avg_traces*(proposal, runNB[, channels, subdir]) | Save average traces of PES into an h5 file. |
| *load_pes_avg_traces*(proposal, runNB[, channels, subdir]) | Load existing PES average traces. |

toolbox_scs.detectors.pes.**get_pes_tof**(*proposal*, *runNB*, *mnemonic*, *start=0*, *origin=None*, *width=None*, *subtract_baseline=False*, *baseStart=None*, *baseWidth=40*, *merge_with=None*)

Extracts time-of-flight spectra from raw digitizer traces. The spectra are aligned by pulse Id using the SASE 3 bunch pattern. If origin is not None, a time coordinate in nanoseconds 'time_ns' is computed and added to the DataArray.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **mnemonic** (*str*) – mnemonic for PES, e.g. “PES_2Araw”.
- **start** (*int*) – starting sample of the first spectrum in the raw trace.
- **origin** (*int*) – sample of the trace that corresponds to time-of-flight origin, also called prompt. Used to compute the ‘time_ns’ coordinates. If None, computation of ‘time_ns’ is skipped.
- **width** (*int*) – number of samples per spectra. If None, the number of samples for 4.5 MHz repetition rate is used.
- **subtract_baseline** (*bool*) – If True, subtract baseline defined by baseStart and baseWidth to each spectrum.
- **baseStart** (*int*) – starting sample of the baseline.
- **baseWidth** (*int*) – number of samples to average (starting from baseStart) for baseline calculation.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one.

**Returns**
**pes** – DataArray containing the PES time-of-flight spectra.

**Return type**
xarray DataArray

### Example

```
>>> import toolbox_scs as tb
>>> import toolbox_scs.detectors as tbdet
>>> proposal, runNB = 900447, 12
>>> pes = tbdet.get_pes_tof(proposal, runNB, 'PES_2Araw',
>>>                         start=2557, origin=76)
```

toolbox_scs.detectors.pes.**get_pes_params**(*run*, *channel=None*)

Extract PES parameters for a given extra_data DataCollection. Parameters are gas, binding energy, retardation voltages or all voltages of the MPOD.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data
- **channel** (*str*) – Channel name or PES mnemonic, e.g. ‘2A’ or ‘PES_1Craw’. If None, or if the channel is not found in the data, the retardation voltage for all channels is retrieved.

**Returns**
**params** – dictionnary of PES parameters

**Return type**
dict

toolbox_scs.detectors.pes.**save_pes_avg_traces**(*proposal*, *runNB*, *channels=None*, *subdir='usr/processed_runs'*)

Save average traces of PES into an h5 file.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **channels** (*str or list*) – The PES channels or mnemonics, e.g. '2A', ['2A', '3C'], ['PES_1Araw', 'PES_4Draw', '3B']
- **subdir** (*str*) – subdirectory. The data is stored in <proposal path>/<subdir>/r{runNB:04d}/f'r{runNB:04d}-pes-data.h5'
- **Output** –
- **------** –
- **traces.** (*xarray Dataset saved in a h5 file containing the PES average*) –

toolbox_scs.detectors.pes.**load_pes_avg_traces**(*proposal*, *runNB*, *channels=None*, *subdir='usr/processed_runs'*)

Load existing PES average traces.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **channels** (*str or list*) – The PES channels or mnemonics, e.g. '2A', ['2A', '3C'], ['PES_1Araw', 'PES_4Draw', '3B']
- **subdir** (*str*) – subdirectory. The data is stored in <proposal path>/<subdir>/r{runNB:04d}/f'r{runNB:04d}-pes-data.h5'
- **Output** –
- **------** –
- **ds** (*xarray Dataset*) – dataset containing the PES average traces.

## toolbox_scs.detectors.viking

### Module Contents

### Classes

| | |
|---|---|
| *Viking* | The Viking analysis (spectrometer used in combination with Andor Newton |

class toolbox_scs.detectors.viking.**Viking**(*proposalNB*)

The Viking analysis (spectrometer used in combination with Andor Newton camera)

The objects of this class contain the meta-information about the settings of the spectrometer, not the actual data, except possibly a dark image for background subtraction.

The actual data is loaded into *xarray`s via the method `from_run()*, and stays there.

**PROPOSAL**

the number of the proposal

**Type**
int

**X_RANGE**

the slice to take in the non-dispersive direction, in pixels. Defaults to the entire width.

**Type**
slice

**Y_RANGE**

the slice to take in the energy dispersive direction

**Type**
slice

**USE_DARK**

whether to do dark subtraction. Is initially *False*, magically switches to *True* if a dark has been loaded, but may be reset.

**Type**
bool

**ENERGY_CALIB**

The 2nd degree polynomial coefficients for calibration from pixel to energy. Defaults to [0, 1, 0] (no calibration applied).

**Type**
1D array (len=3)

**BL_POLY_DEG**

the dgree of the polynomial used for baseline subtraction. Defaults to 1.

**Type**
int

**BL_SIGNAL_RANGE**

the dispersive-axis range, defined by an interval [min, max], to avoid when fitting a polynomial for baseline subtraction. Multiple ranges can be provided in the form [[min1, max1], [min2, max2], . . . ].

**Type**
list

**FIELDS**

the fields to be loaded from the data. Add additional fields if so desired.

**Type**
list of str

### Example

proposal = 2953 v = Viking(proposal) v.X_RANGE = slice(0, 1900) v.Y_RANGE = slice(38, 80) v.ENERGY_CALIB = [1.47802667e-06, 2.30600328e-02, 5.15884589e+02] v.BL_SIGNAL_RANGE = [500, 545]

**set_params**(***params*)

**get_params**(**params*)

**from_run**(*runNB*, *add_attrs=True*)

load a run

Load the run *runNB*. A thin wrapper around *toolbox_scs.load*.

**Parameters**

- **runNB** (*int*) – the run number
- **add_attrs** (*bool*) – if True, adds the camera parameters as attributes to the dataset (see get_camera_params())
- **Output** –
- **------** –
- **ds** (*xarray Dataset*) – the dataset containing the camera images

### Example

data = v.from_run(145) # load run 145

data1 = v.from_run(145) # load run 145 data2 = v.from_run(155) # load run 155 data = xarray.concat([data1, data2], 'trainId') # combine both

**load_dark**(*runNB=None*)

**integrate**(*data*)

This function calculates the mean over the non-dispersive dimension to create a spectrum. If the camera parameters are known, the spectrum is multiplied by the number of photoelectrons per ADC count. A new variable "spectrum" is added to the data.

**get_camera_gain**(*run*)

Get the preamp gain of the camera in the Viking spectrometer for a specified run.

**Parameters**

- **run** (*extra_data DataCollection*) – information on the run
- **Output** –
- **------** –
- **gain** (*int*) –

**e_per_counts**(*run*, *gain=None*)

Conversion factor from camera digital counts to photoelectrons per count. The values can be found in the camera datasheet (Andor Newton) but they have been slightly corrected for High Sensitivity mode after analysis of runs 1204, 1207 and 1208, proposal 2937.

**Parameters**

- **run** (*extra_data DataCollection*) – information on the run

- **gain** (*int*) – the camera preamp gain
- **Output** –
- **------** –
- **ret** (*float*) – photoelectrons per count

**get_camera_params**(*run*)

**removePolyBaseline**(*data*)

Removes a polynomial baseline to a spectrum, assuming a fixed position for the signal.

**Parameters**

- **data** (*xarray Dataset*) – The Viking data containing the variable "spectrum"
- **Output** –
- **------** –
- **data** – the original dataset with the added variable "spectrum_nobl" containing the baseline subtracted spectra.

**xas**(*data*, *data_ref*, *thickness=1*, *plot=False*, *plot_errors=True*, *xas_ylim=(-1, 3)*)

Given two independent datasets (one with sample and one reference), this calculates the average XAS spectrum (absorption coefficient), associated standard deviation and standard error. The absorption coefficient is defined as -log(It/I0)/thickness.

**Parameters**

- **data** (*xarray Dataset*) – the dataset containing the spectra with sample
- **data_ref** (*xarray Dataset*) – the dataset containing the spectra without sample
- **thickness** (*float*) – the thickness used for the calculation of the absorption coefficient
- **plot** (*bool*) – If True, plot the resulting average spectra.
- **plot_errors** (*bool*) – If True, adds the 95% confidence interval on the spectra.
- **xas_ylim** (*tuple or list of float*) – the y limits for the XAS plot.
- **Output** –
- **------** –
- **xas** (*xarray Dataset*) – the dataset containing the computed XAS quantities: I0, It, absorptionCoef and their associated errors.

**calibrate**(*runList*, *plot=True*)

This routine determines the calibration coefficients to translate the camera pixels into energy in eV. The Viking spectrometer is calibrated using the beamline monochromator: runs with various monochromatized photon energy are recorded and their peak position on the detector are determined by Gaussian fitting. The energy vs. position data is then fitted to a second degree polynomial.

**Parameters**

- **runList** (*list of int*) – the list of runs containing the monochromatized spectra
- **plot** (*bool*) – if True, the spectra, their Gaussian fits and the calibration curve are plotted.
- **Output** –
- **------** –
- **energy_calib** (*np.array*) – the calibration coefficients (2nd degree polynomial)

### toolbox_scs.detectors.xgm

XGM related sub-routines



(contributions preferrably comply with pep8 code structure guidelines.)

## Module Contents

## Functions

| | |
|---|---|
| *get_xgm*(run[, mnemonics, merge_with, indices]) | Load and/or computes XGM data. Sources can be loaded on the |
| *calibrate_xgm*(run, data[, xgm, plot]) | Calculates the calibration factor F between the photon flux (slow signal) |

toolbox_scs.detectors.xgm.**get_xgm**(*run*, *mnemonics=None*, *merge_with=None*, *indices=slice(0, None)*)

Load and/or computes XGM data. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing dataset (merge_with). The bunch pattern table is used to assign the pulse id coordinates if the number of pulses has changed during the run.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the xgm data.
- **mnemonics** (*str or list of str*) – mnemonics for XGM, e.g. "SCS_SA3" or ["XTD10_XGM", "SCS_XGM"]. If None, defaults to "SCS_SA3" in case no merge_with dataset is provided.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The XGM variables of merge_with (if any) will also be computed and merged.
- **indices** (*slice, list, 1D array*) – Pulse indices of the XGM array in case bunch pattern is missing.

**Returns**

merged with Dataset *merge_with* if provided.

**Return type**

xarray Dataset with pulse-resolved XGM variables aligned,

### Example

```
>>> import toolbox_scs as tb
>>> run, ds = tb.load(2212, 213, 'SCS_SA3')
>>> ds['SCS_SA3']
```

toolbox_scs.detectors.xgm.**calibrate_xgm**(*run*, *data*, *xgm='SCS'*, *plot=False*)

Calculates the calibration factor F between the photon flux (slow signal) and the fast signal (pulse-resolved) of the sase 3 pulses. The calibrated fast signal is equal to the uncalibrated one multiplied by F.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.

- **data** (*xarray Dataset*) – dataset containing the pulse-resolved sase 3 signal, e.g. 'SCS_SA3'
- **xgm** (*str*) – one in {'XTD10', 'SCS'}
- **plot** (*bool*) – If True, shows a plot of the photon flux, averaged fast signal and calibrated fast signal.

**Returns**

**F** – calibration factor F defined as: calibrated XGM [microJ] = F * fast XGM array ('SCS_SA3' or 'XTD10_SA3')

**Return type**

float

### Example

```
>>> import toolbox_scs as tb
>>> import toolbox_scs.detectors as tbdet
>>> run, data = tb.load(900074, 69, ['SCS_XGM'])
>>> ds = tbdet.get_xgm(run, merge_with=data)
>>> F = tbdet.calibrate_xgm(run, ds, plot=True)
>>> # Add calibrated XGM to the dataset:
>>> ds['SCS_SA3_uJ'] = F * ds['SCS_SA3']
```

## Package Contents

## Classes

| | |
|---|---|
| *AzimuthalIntegrator* | |
| *AzimuthalIntegratorDSSC* | |
| *DSSCBinner* | |
| *DSSCFormatter* | |
| *hRIXS* | The hRIXS analysis, especially curvature correction |
| *MaranaX* | A spin-off of the hRIXS class: with parallelized centroiding |
| *Viking* | The Viking analysis (spectrometer used in combination with Andor Newton |

### Functions

| | |
|---|---|
| get_bam(run[, mnemonics, merge_with, bunchPattern, ...]) | Load beam arrival monitor (BAM) data and align their pulse ID |
| get_bam_params(run[, mnemo_or_source]) | Extract the run values of bamStatus[1-3] and bamError. |
| check_peak_params(run, mnemonic[, raw_trace, ntrains, ...]) | Checks and plots the peak parameters (pulse window and baseline window |
| get_digitizer_peaks(run, mnemonic[, merge_with, ...]) | Automatically computes digitizer peaks. A source can be loaded on the |
| get_laser_peaks(run[, mnemonic, merge_with, ...]) | Extracts laser photodiode signal (peak intensity) from Fast ADC |
| get_peaks(run, data, mnemonic[, useRaw, autoFind, ...]) | Extract peaks from one source (channel) of a digitizer. |
| get_tim_peaks(run[, mnemonic, merge_with, ...]) | Automatically computes TIM peaks. Sources can be loaded on the |
| digitizer_signal_description(run[, digitizer]) | Check for the existence of signal description and return all corresponding |
| get_dig_avg_trace(run, mnemonic[, ntrains]) | Compute the average over ntrains evenly spaced accross all trains |
| get_data_formatted([filenames, data_list]) | Combines the given data into one dataset. For any of extra_data's data |
| load_xarray(fname[, group, form]) | Load stored xarray Dataset. |
| save_attributes_h5(fname[, data]) | Adding attributes to a hdf5 file. This function is intended to be used to |
| save_xarray(fname, data[, group, mode]) | Store xarray Dataset in the specified location |
| create_dssc_bins(name, coordinates, bins) | Creates a single entry for the dssc binner dictionary. The produced xarray |
| get_xgm_formatted(run_obj, xgm_name, dssc_frame_coords) | Load the xgm data and define coordinates along the pulse dimension. |
| load_dssc_info(proposal, run_nr) | Loads the first data file for DSSC module 0 (this is hard-coded) |
| load_mask(fname, dssc_mask) | Load a DSSC mask file. |
| quickmask_DSSC_ASIC(poslist) | Returns a mask for the given DSSC geometry with ASICs given in poslist |
| process_dssc_data(proposal, run_nr, module, chunksize, ...) | Collects and reduces DSSC data for a single module. |
| extract_GH2(ds, run[, firstFrame, bunchPattern, gh2_dim]) | Select and align the frames of the Gotthard-II that have been exposed |
| get_pes_params(run[, channel]) | Extract PES parameters for a given extra_data DataCollection. |
| get_pes_tof(proposal, runNB, mnemonic[, start, ...]) | Extracts time-of-flight spectra from raw digitizer traces. The spectra |
| save_pes_avg_traces(proposal, runNB[, channels, subdir]) | Save average traces of PES into an h5 file. |
| load_pes_avg_traces(proposal, runNB[, channels, subdir]) | Load existing PES average traces. |
| calibrate_xgm(run, data[, xgm, plot]) | Calculates the calibration factor F between the photon flux (slow signal) |
| get_xgm(run[, mnemonics, merge_with, indices]) | Load and/or computes XGM data. Sources can be loaded on the |

### Attributes

__all__

class toolbox_scs.detectors.**AzimuthalIntegrator**(*imageshape*, *center*, *polar_range*, *aspect=204 / 236*, ***kwargs*)

Bases: object

**_calc_dist_array**(*shape*, *center*, *aspect*)

Calculate pixel coordinates for the given shape.

**_calc_indices**(***kwargs*)

Calculates the list of indices for the flattened image array.

**_calc_polar_mask**(*polar_range*)

**calc_q**(*distance*, *wavelength*)

Calculate momentum transfer coordinate.

**Parameters**

- **distance** (*float*) – Sample - detector distance in meter
- **wavelength** (*float*) – wavelength of scattered light in meter

**Returns**
**deltaq** – Momentum transfer coordinate in 1/m

**Return type**
np.ndarray

**__call__**(*image*)

class toolbox_scs.detectors.**AzimuthalIntegratorDSSC**(*geom*, *polar_range*, *dxdy=(0, 0)*, ***kwargs*)

Bases: *AzimuthalIntegrator*

**_calc_dist_array**(*geom*, *dxdy*)

Calculate pixel coordinates for the given shape.

toolbox_scs.detectors.**get_bam**(*run*, *mnemonics=None*, *merge_with=None*, *bunchPattern='sase3'*, *pulseIds=None*)

Load beam arrival monitor (BAM) data and align their pulse ID according to the bunch pattern. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing data set (merge_with).

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the bam data.
- **mnemonics** (*str or list of str*) – mnemonics for BAM, e.g. "BAM1932M" or ["BAM414", "BAM1932M"]. the arrays are either taken from merge_with or loaded from the DataCollection run.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. If merge_with contains variables in mnemonics, they will be selected, aligned and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks. The pulse ID dimension will be named 'sa1_pId', 'sa3_pId' or 'ol_pId', respectively.

- **pulseIds** (*list, 1D array*) – Pulse Ids. If None, they are automatically loaded.

**Returns**
merged with Dataset *merge_with* if provided.

**Return type**
xarray Dataset with pulse-resolved BAM variables aligned,

### Example

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2711, 303)
>>> bam = tb.get_bam(run, 'BAM1932S')
```

toolbox_scs.detectors.**get_bam_params**(*run*, *mnemo_or_source='BAM1932S'*)

Extract the run values of bamStatus[1-3] and bamError.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the bam data.
- **mnemo_or_source** (*str*) – mnemonic of the BAM, e.g. 'BAM414', or source name, e.g. 'SCS_ILH_LAS/DOOCS/BAM_414_B2.

**Returns**
**params** – dictionnary containing the extracted parameters.

**Return type**
dict

---

**Note:** The extracted parameters are run values, they do not reflect any possible change during the run.

---

toolbox_scs.detectors.**check_peak_params**(*run*, *mnemonic*, *raw_trace=None*, *ntrains=200*, *params=None*, *plot=True*, *show_all=False*, *bunchPattern='sase3'*)

Checks and plots the peak parameters (pulse window and baseline window of a raw digitizer trace) used to compute the peak integration. These parameters are either set by the digitizer peak-integration settings, or are determined by a peak finding algorithm (used in get_tim_peaks or get_fast_adc_peaks) when the inputs are raw traces. The parameters can also be provided manually for visual inspection. The plot either shows the first and last pulse of the trace or the entire trace.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – ToolBox mnemonic of the digitizer data, e.g. 'MCP2apd'.
- **raw_trace** (*optional, 1D numpy array or xarray DataArray*) – Raw trace to display. If None, the average raw trace over ntrains of the corresponding channel is loaded (this can be time-consuming).
- **ntrains** (*optional, int*) – Only used if raw_trace is None. Number of trains used to calculate the average raw trace of the corresponding channel.
- **plot** (*bool*) – If True, displays the raw trace and peak integration regions.
- **show_all** (*bool*) – If True, displays the entire raw trace and all peak integration regions (this can be time-consuming). If False, shows the first and last pulse according to the bunchPattern.

- **bunchPattern** (*optional, str*) – Only used if plot is True. Checks the bunch pattern against the digitizer peak parameters and shows potential mismatch.

**Return type**
dictionnary of peak integration parameters

toolbox_scs.detectors.**get_digitizer_peaks**(*run*, *mnemonic*, *merge_with=None*, *bunchPattern='sase3'*, *integParams=None*, *keepAllSase=False*)

Automatically computes digitizer peaks. A source can be loaded on the fly via the mnemonic argument, or processed from an existing data set (merge_with). The bunch pattern table is used to assign the pulse id coordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonic for FastADC or ADQ412, e.g. "I0_ILHraw" or "MCP3apd". The data is either loaded from the DataCollection or taken from merge_with.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one.
- **bunchPattern** (*str or dict*) – 'sase1' or 'sase3' or 'scs_ppl', 'None': bunch pattern
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.
- **keepAllSase** (*bool*) – Only relevant in case of sase-dedicated trains. If True, all trains are kept, else only those of the bunchPattern are kept.

**Returns**

- *xarray Dataset with digitizer peak variables. Raw variables are*
- *substituted by the peak caclulated values (e.g. "FastADC2raw" becomes*
- *"FastADC2peaks").*

toolbox_scs.detectors.**get_laser_peaks**(*run*, *mnemonic=None*, *merge_with=None*, *bunchPattern='scs_ppl'*, *integParams=None*)

Extracts laser photodiode signal (peak intensity) from Fast ADC digitizer. Sources can be loaded on the fly via the mnemonics argument, and/or processed from an existing data set (merge_with). The PP laser bunch pattern is used to assign the pulse idcoordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonic for FastADC corresponding to laser signal, e.g. "FastADC2peaks" or 'I0_ILHraw'.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The FastADC variables of merge_with (if any) will also be computed and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks.
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.

**Returns**

- *xarray Dataset with all Fast ADC variables substituted by*
- *the peak caclulated values (e.g. "FastADC2raw" becomes*

- *"FastADC2peaks").*

toolbox_scs.detectors.**get_peaks**(*run*, *data*, *mnemonic*, *useRaw=True*, *autoFind=True*, *integParams=None*, *bunchPattern='sase3'*, *bpt=None*, *extra_dim=None*, *indices=None*)

Extract peaks from one source (channel) of a digitizer.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data
- **data** (*xarray DataArray or str*) – array containing the raw traces or peak-integrated values from the digitizer. If str, must be one of the ToolBox mnemonics. If None, the data is loaded via the source and key arguments.
- **mnemonic** (*str or dict*) – ToolBox mnemonic or dict with source and key as in {'source': 'SCS_UTC1_ADQ/ADC/1:network', 'key': 'digitizers.channel_1_A.raw.samples'}
- **useRaw** (*bool*) – If True, extract peaks from raw traces. If False, uses the APD (or peaks) data from the digitizer.
- **autoFind** (*bool*) – If True, finds integration parameters by inspecting the average raw trace. Only valid if useRaw is True.
- **integParams** (*dict*) – dictionnary containing the integration parameters for raw trace integration: 'pulseStart', 'pulseStop', 'baseStart', 'baseStop', 'period', 'npulses'. Not used if autoFind is True. All keys are required when bunch pattern is missing.
- **bunchPattern** (*str*) – match the peaks to the bunch pattern: 'sase1', 'sase3', 'scs_ppl'. This will dictate the name of the pulse ID coordinates: 'sa1_pId', 'sa3_pId' or 'scs_ppl'.
- **bpt** (*xarray DataArray*) – bunch pattern table
- **extra_dim** (*str*) – Name given to the dimension along the peaks. If None, the name is given according to the bunchPattern.
- **indices** (*array, slice*) – indices from the peak-integrated data to retrieve. Only required when bunch pattern is missing and useRaw is False.

**Return type**
xarray.DataArray containing digitizer peaks with pulse coordinates

toolbox_scs.detectors.**get_tim_peaks**(*run*, *mnemonic=None*, *merge_with=None*, *bunchPattern='sase3'*, *integParams=None*, *keepAllSase=False*)

Automatically computes TIM peaks. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing data set (merge_with). The bunch pattern table is used to assign the pulse id coordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonics for TIM, e.g. "MCP2apd".
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The TIM variables of merge_with (if any) will also be computed and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks. The pulse ID dimension will be named 'sa1_pId', 'sa3_pId' or 'ol_pId', respectively.
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.
- **keepAllSase** (*bool*) – Only relevant in case of sase-dedicated trains. If True, all trains are kept, else only those of the bunchPattern are kept.

**Returns**

- *xarray Dataset with TIM variables substituted by*
- *the peak caclulated values (e.g. "MCP2raw" becomes*
- *"MCP2peaks"), merged with Dataset *merge_with* if provided.*

toolbox_scs.detectors.**digitizer_signal_description**(*run*, *digitizer=None*)

Check for the existence of signal description and return all corresponding channels in a dictionnary.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **digitizer** (*str or list of str (default=None)*) – Name of the digitizer: one in ['FastADC', FastADC2', 'ADQ412', 'ADQ412_2] If None, all digitizers are used

**Returns**
**signal_description** – the digitizer channels.

**Return type**
dictionnary containing the signal description of

### Example

import toolbox_scs as tb run = tb.open_run(3481, 100) signals = tb.digitizer_signal_description(run) signals_fadc2 = tb.digitizer_signal_description(run, 'FastADC2')

toolbox_scs.detectors.**get_dig_avg_trace**(*run*, *mnemonic*, *ntrains=None*)

Compute the average over ntrains evenly spaced accross all trains of a digitizer trace.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – ToolBox mnemonic of the digitizer data, e.g. 'MCP2apd'.
- **ntrains** (*int*) – Number of trains used to calculate the average raw trace. If None, all trains are used.

**Returns**
**trace** – The average digitizer trace

**Return type**
DataArray

**class** toolbox_scs.detectors.**DSSCBinner**(*proposal_nr*, *run_nr*, *binners={}*, *xgm_name='SCS_SA3'*, *tim_names=['MCP1apd', 'MCP2apd', 'MCP3apd']*, *dssc_coords_stride=2*)

**__del__**()

**add_binner**(*name*, *binner*)

Add additional binner to internal dictionary

**Parameters**

- **name** (*str*) – name of binner to be created
- **binner** (*xarray.DataArray*) – An array that represents a map how the respective coordinate should be binned.

**Raises**
*ToolBoxValueError* – Exception: Raises exception in case the name does not correspond to a valid binner name. To be generalized.

`load_xgm()`

load xgm data and construct coordinate array according to corresponding dssc frame number.

`load_tim()`

load tim data and construct coordinate array according to corresponding dssc frame number.

`create_pulsemask(use_data='xgm', threshold=(0, np.inf))`

creates a mask for dssc frames according to measured xgm intensity. Once such a mask has been constructed, it will be used in the data reduction process to drop out-of-bounds pulses.

`get_info()`

Returns the expected shape of the binned dataset, in case binners have been defined.

`_bin_metadata(data)`

`get_xgm_binned()`

Bin the xgm data according to the binners of the dssc data. The result can eventually be merged into the final dataset by the DSSCFormatter.

**Returns**
**xgm_data** – xarray dataset containing the binned xgm data

**Return type**
xarray.DataSet

`get_tim_binned()`

Bin the tim data according to the binners of the dssc data. The result can eventually be merged into the final dataset by the DSSCFormatter.

**Returns**
**tim_data** – xarray dataset containing the binned tim data

**Return type**
xarray.DataSet

`process_data(modules=[], filepath='./', chunksize=512, backend='loky', n_jobs=None, dark_image=None, xgm_normalization=False, normevery=1)`

Load and bin dssc data according to self.bins. No data is returned by this method. The condensed data is written to file by the worker processes directly.

**Parameters**

- **modules** (*list of ints*) – a list containing the module numbers that should be processed. If empty, all modules are processed.
- **filepath** (*str*) – the path where the files containing the reduced data should be stored.
- **chunksize** (*int*) – The number of trains that should be read in one iterative step.
- **backend** (*str*) – joblib multiprocessing backend to be used. At the moment it can be any of joblibs standard backends: 'loky' (default), 'multiprocessing', 'threading'. Anything else than the default is experimental and not appropriately implemented in the dbdet member function 'bin_data'.
- **n_jobs** (*int*) – inversely proportional of the number of cpu's available for one job. Tasks within one job can grab a maximum of n_CPU_tot/n_jobs of cpu's. Note that when using

the default backend there is no need to adjust this parameter with the current implementation.

- **dark_image** (*xarray.DataArray*) – DataArray with dimensions compatible with the loaded dssc data. If given, it will be subtracted from the dssc data before the binning. The dark image needs to be of dimension module, trainId, pulse, x and y.
- **xgm_normalization** (*boolean*) – if true, the dssc data is normalized by the xgm data before the binning.
- **normevery** (*int*) – integer indicating which out of normevery frame will be normalized.

class toolbox_scs.detectors.**DSSCFormatter**(*filepath*)

**combine_files**(*filenames=[]*)

Read the files given in filenames, and store the data in the class variable 'data'. If no filenames are given, it tries to read the files stored in the class-internal variable '_filenames'.

**Parameters**
**filenames** (*list*) – list of strings containing the names of the files to be combined.

**add_dataArray**(*groups=[]*)

Reads addional xarray-data from the first file given in the list of filenames. This assumes that all the files in the folder contain the same additional data. To be generalized.

**Parameters**
**groups** (*list*) – list of strings with the names of the groups in the h5 file, containing additional xarray data.

**add_attributes**(*attributes={}*)

Add additional information, such as run-type, as attributes to the formatted .h5 file.

**Parameters**
**attributes** (*dictionary*) – a dictionary, containing information or data of any kind, that will be added to the formatted .h5 file as attributes.

**save_formatted_data**(*filename*)

Create a .h5 file containing the main dataset in the group called 'data'. Additional groups will be created for the content of the variable 'data_array'. Metadata about the file is added in the form of attributes.

**Parameters**
**filename** (*str*) – the name of the file to be created

toolbox_scs.detectors.**get_data_formatted**(*filenames=[]*, *data_list=[]*)

Combines the given data into one dataset. For any of extra_data's data types, an xarray.Dataset is returned. The data is sorted along the 'module' dimension. The array dimension have the order 'trainId', 'pulse', 'module', 'x', 'y'. This order is required by the extra_geometry package.

**Parameters**

- **filenames** (*list of str*) – files to be combined as a list of names. Calls '_data_from_list' to actually load the data.
- **data_list** (*list*) – list containing the already loaded data

**Returns**
**data** – A xarray.Dataset containing the combined data.

**Return type**
xarray.Dataset

toolbox_scs.detectors.**load_xarray**(*fname*, *group='data'*, *form='dataset'*)

Load stored xarray Dataset. Comment: This function exists because of a problem with the standard netcdf engine that is malfunctioning due to related software installed in the exfel-python environment. May be dropped at some point.

**Parameters**

- **fname** (*str*) – filename as string
- **group** (*str*) – the name of the xarray dataset (group in h5 file).
- **form** (*str*) – specify whether the data to be loaded is a 'dataset' or a 'array'.

toolbox_scs.detectors.**save_attributes_h5**(*fname*, *data={}*)

Adding attributes to a hdf5 file. This function is intended to be used to attach metadata to a processed run.

**Parameters**

- **fname** (*str*) – filename as string
- **data** (*dictionary*) – the data that should be added to the file in form of a dictionary.

toolbox_scs.detectors.**save_xarray**(*fname*, *data*, *group='data'*, *mode='a'*)

Store xarray Dataset in the specified location

**Parameters**

- **data** (*xarray.DataSet*) – The data to be stored
- **fname** (*str, int*) – filename
- **overwrite** (*bool*) – overwrite existing data

**Raises**

*ToolBoxFileError* – Exception: File existed, but overwrite was set to False.

toolbox_scs.detectors.**create_dssc_bins**(*name*, *coordinates*, *bins*)

Creates a single entry for the dssc binner dictionary. The produced xarray data-array will later be used to perform grouping operations according to the given bins.

**Parameters**

- **name** (*str*) – name of the coordinate to be binned.
- **coordinates** (*numpy.ndarray*) – the original coordinate values (1D)
- **bins** (*numpy.ndarray*) – the bins according to which the corresponding dimension should be grouped.

**Returns**

**da** – A pre-formatted xarray.DataArray relating the specified dimension with its bins.

**Return type**

xarray.DataArray

### Examples

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2212, 235, include='*DA*')
```

1.) binner along 'pulse' dimension. Group data into two bins. >>> bins_pulse = ['pumped', 'unpumped'] * 10
>>> binner_pulse = tb.create_dssc_bins("pulse",

np.linspace(0,19,20, dtype=int), bins_pulse)

**2.) binner along 'train' dimension. Group data into bins corresponding**
to the positions of a delay stage for instance.

```
>>> bins_trainId = tb.get_array(run, 'PP800_PhaseShifter', 0.04)
>>> binner_train = tb.create_dssc_bins("trainId",
                              run.trainIds,
                              bins_trainId.values)
```

toolbox_scs.detectors.**get_xgm_formatted**(*run_obj*, *xgm_name*, *dssc_frame_coords*)

Load the xgm data and define coordinates along the pulse dimension.

**Parameters**

- **run_obj** (*extra_data.DataCollection*) – DataCollection object providing access to the xgm data to be loaded
- **xgm_name** (*str*) – valid mnemonic of a xgm source
- **dssc_frame_coords** (*int, list*) – defines which dssc frames should be normalized using data from the xgm.

**Returns**
**xgm** – xgm data with coordinate 'pulse'.

**Return type**
xarray.DataArray

toolbox_scs.detectors.**load_dssc_info**(*proposal*, *run_nr*)

Loads the first data file for DSSC module 0 (this is hardcoded) and returns the detector_info dictionary

**Parameters**

- **proposal** (*str, int*) – number of proposal
- **run_nr** (*str, int*) – number of run

**Returns**
**info** – {'dims': tuple, 'frames_per_train': int, 'total_frames': int}

**Return type**
dictionary

toolbox_scs.detectors.**load_mask**(*fname*, *dssc_mask*)

Load a DSSC mask file.

Copyright (c) 2019, Michael Schneider Copyright (c) 2020, SCS-team license: BSD 3-Clause License (see LICENSE_BSD for more info)

**Parameters**
**fname** (*str*) – string of the filename of the mask file

**Return type**
dssc_mask

toolbox_scs.detectors.**quickmask_DSSC_ASIC**(*poslist*)

Returns a mask for the given DSSC geometry with ASICs given in poslist blanked. poslist is a list of (module, row, column) tuples. Each module consists of 2 rows and 8 columns of individual ASICS.

Copyright (c) 2019, Michael Schneider Copyright (c) 2020, SCS-team license: BSD 3-Clause License (see LICENSE_BSD for more info)

toolbox_scs.detectors.**process_dssc_data**(*proposal*, *run_nr*, *module*, *chunksize*, *info*, *dssc_binners*, *path='./'*, *pulsemask=None*, *dark_image=None*, *xgm_mnemonic='SCS_SA3'*, *xgm_normalization=False*, *normevery=1*)

Collects and reduces DSSC data for a single module.

Copyright (c) 2020, SCS-team

**Parameters**

- **proposal** (*int*) – proposal number
- **run_nr** (*int*) – run number
- **module** (*int*) – DSSC module to process
- **chunksize** (*int*) – number of trains to load simultaneously
- **info** (*dictionary*) – dictionary containing keys 'dims', 'frames_per_train', 'total_frames', 'trainIds', 'number_of_trains'.
- **dssc_binners** (*dictionary*) – a dictionary containing binner objects created by the ToolBox member function "create_binner()"
- **path** (*str*) – location in which the .h5 files, containing the binned data, should be stored.
- **pulsemask** (*numpy.ndarray*) – array of booleans to be used to mask dssc data according to xgm data.
- **dark_image** (*xarray.DataArray*) – an xarray dataarray with matching coordinates with the loaded data. If dark_image is not None it will be subtracted from each individual dssc frame.
- **xgm_normalization** (*bool*) – true if the data should be divided by the corresponding xgm value.
- **xgm_mnemonic** (*str*) – Mnemonic of the xgm data to be used for normalization.
- **normevery** (*int*) – One out of normevery dssc frames will be normalized.

**Returns**
**module_data** – xarray datastructure containing data binned according to bins.

**Return type**
xarray.Dataset

toolbox_scs.detectors.**extract_GH2**(*ds*, *run*, *firstFrame=0*, *bunchPattern='scs_ppl'*, *gh2_dim='gh2_pId'*)

Select and align the frames of the Gotthard-II that have been exposed to light.

**Parameters**

- **ds** (*xarray.Dataset*) – The dataset containing GH2 data
- **run** (*extra_data.DataCollection*) – The run containing the bunch pattern source

- **firstFrame** (*int*) – The GH2 frame number corresponding to the first pulse of the train.
- **bunchPattern** (*str in ['scs_ppl', 'sase3']*) – the bunch pattern used to align data. For 'scs_ppl', the gh2_pId dimension in renamed 'ol_pId', and for 'sase3' gh2_pId is renamed 'sa3_pId'.
- **gh2_dim** (*str*) – The name of the dimension that corresponds to the Gotthard-II frames.

**Returns**
**nds** – The aligned and reduced dataset with only-data-containing GH2 variables.

**Return type**
xarray Dataset

`class toolbox_scs.detectors.hRIXS(proposalNB, detector='MaranaX')`

The hRIXS analysis, especially curvature correction

The objects of this class contain the meta-information about the settings of the spectrometer, not the actual data, except possibly a dark image for background subtraction.

The actual data is loaded into `xarray`s, and stays there.

**PROPOSAL**

the number of the proposal

**Type**
int

**DETECTOR**

the detector to be used. Can be ['hRIXS_det', 'MaranaX'] defaults to 'hRIXS_det' for backward-compatibility.

**Type**
str

**X_RANGE**

the slice to take in the dispersive direction, in pixels. Defaults to the entire width.

**Type**
slice

**Y_RANGE**

the slice to take in the energy direction

**Type**
slice

**THRESHOLD**

pixel counts above which a hit candidate is assumed, for centroiding. use None if you want to give it in standard deviations instead.

**Type**
float

**STD_THRESHOLD**

same as THRESHOLD, in standard deviations.

**DBL_THRESHOLD**

threshold controling whether a detected hit is considered to be a double hit.

**BINS**

the number of bins used in centroiding

**Type**
int

**CURVE_A, CURVE_B**

the coefficients of the parabola for the curvature correction

**Type**
float

**USE_DARK**

whether to do dark subtraction. Is initially *False*, magically switches to *True* if a dark has been loaded, but may be reset.

**Type**
bool

**ENERGY_INTERCEPT, ENERGY_SLOPE**

The calibration from pixel to energy

**FIELDS**

the fields to be loaded from the data. Add additional fields if so desired.

### Example

proposal = 3145 h = hRIXS(proposal) h.Y_RANGE = slice(700, 900) h.CURVE_B = -3.695346575286939e-07 h.CURVE_A = 0.024084479232443695 h.ENERGY_SLOPE = 0.018387 h.ENERGY_INTERCEPT = 498.27 h.STD_THRESHOLD = 3.5

**DETECTOR_FIELDS**

**aggregators**

**set_params**(***params*)

**get_params**(**params*)

**from_run**(*runNB*, *proposal=None*, *extra_fields=()*, *drop_first=False*, *subset=None*)

load a run

Load the run *runNB*. A thin wrapper around *toolbox.load*. :param drop_first: if True, the first image in the run is removed from the dataset. :type drop_first: bool

#### Example

data = h.from_run(145) # load run 145

data1 = h.from_run(145) # load run 145 data2 = h.from_run(155) # load run 155 data = xarray.concat([data1, data2], 'trainId') # combine both

**load_dark**(*runNB*, *proposal=None*)

load a dark run

Load the dark run *runNB* from *proposal*. The latter defaults to the current proposal. The dark is stored in this *hRIXS* object, and subsequent analyses use it for background subtraction.

### Example

h.load_dark(166) # load dark run 166

**find_curvature**(*runNB*, *proposal=None*, *plot=True*, *args=None*, ***kwargs*)

find the curvature correction coefficients

The hRIXS has some abberations which leads to the spectroscopic lines being curved on the detector. We approximate these abberations with a parabola for later correction.

Load a run and determine the curvature. The curvature is set in *self*, and returned as a pair of floats.

**Parameters**

- **runNB** (*int*) – the run number to use
- **proposal** (*int*) – the proposal to use, default to the current proposal
- **plot** (*bool*) – whether to plot the found curvature onto the data
- **args** (*pair of float, optional*) – a starting value to prime the fitting routine

### Example

h.find_curvature(155) # use run 155 to fit the curvature

**centroid_one**(*image*)

find the position of photons with sub-pixel precision

A photon is supposed to have hit the detector if the intensity within a 2-by-2 square exceeds a threshold. In this case the position of the photon is calculated as the center-of-mass in a 4-by-4 square.

Return the list of x, y coordinate pairs, corrected by the curvature.

**centroid_two**(*image*, *energy*)

determine position of photon hits on detector

The algrothm is taken from the ESRF RIXS toolbox. The thresholds for determining photon hits are given by the incident photon energy

The function returns arrays containing the single and double hits as x and y coordinates

**centroid**(*data*, *bins=None*, *method='auto'*)

calculate a spectrum by finding the centroid of individual photons

This takes the *xarray.Dataset data* and returns a copy of it, with a new *xarray.DataArray* named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

Added a key for switching between algorithims choices are “auto” and “manual” which selects for method for determining whether thresholds there is a photon hit. It changes whether centroid_one or centroid_two is used.

### Example

h.centroid(data) # find photons in all images of the run data.spectrum[0, :].plot() # plot the spectrum of the first image

**parabola**(*x*)

**integrate**(*data*)

calculate a spectrum by integration

This takes the *xarray data* and returns a copy of it, with a new dataarray named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

First the energy that corresponds to each pixel is calculated. Then all pixels within an energy range are summed, where the intensity of one pixel is distributed among the two energy ranges the pixel spans, proportionally to the overlap between the pixel and bin energy ranges.

The resulting data is normalized to one pixel, so the average intensity that arrived on one pixel.

### Example

h.integrate(data) # create spectrum by summing pixels data.spectrum[0, :].plot() # plot the spectrum of the first image

**aggregator**(*da*, *dim*)

**aggregate**(*ds*, *var=None*, *dim='trainId'*)

aggregate (i.e. mostly sum) all data within one dataset

take all images in a dataset and aggregate them and their metadata. For images, spectra and normalizations that means adding them, for others (e.g. delays) adding would not make sense, so we treat them properly. The aggregation functions of each variable are defined in the aggregators attribute of the class. If var is specified, group the dataset by var prior to aggregation. A new variable "counts" gives the number of frames aggregated in each group.

**Parameters**

- **ds** (*xarray Dataset*) – the dataset containing RIXS data
- **var** (*string*) – One of the variables in the dataset. If var is specified, the dataset is grouped by var prior to aggregation. This is useful for sorting e.g. a dataset that contains multiple delays.
- **dim** (*string*) – the dimension over which to aggregate the data

### Example

h.centroid(data) # create spectra from finding photons agg = h.aggregate(data) # sum all spectra agg.spectrum.plot() # plot the resulting spectrum

agg2 = h.aggregate(data, 'hRIXS_delay') # group data by delay agg2.spectrum[0, :].plot() # plot the spectrum for first value

**aggregate_ds**(*ds*, *dim='trainId'*)

**normalize**(*data*, *which='hRIXS_norm'*)

Adds a 'normalized' variable to the dataset defined as the ration between 'spectrum' and 'which'

**Parameters**

- **data** (*xarray Dataset*) – the dataset containing hRIXS data
- **which** (*string, default="hRIXS_norm"*) – one of the variables of the dataset, usually "hRIXS_norm" or "counts"

class toolbox_scs.detectors.**MaranaX**(**args*, ***kwargs*)

Bases: *hRIXS*

A spin-off of the hRIXS class: with parallelized centroiding

**NUM_MAX_HITS = 30**

**centroid**(*data*, *bins=None*, ***kwargs*)

calculate a spectrum by finding the centroid of individual photons

This takes the *xarray.Dataset data* and returns a copy of it, with a new *xarray.DataArray* named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

Added a key for switching between algorithims choices are "auto" and "manual" which selects for method for determining whether thresholds there is a photon hit. It changes whether centroid_one or centroid_two is used.

### Example

h.centroid(data) # find photons in all images of the run data.spectrum[0, :].plot() # plot the spectrum of the first image

**_centroid_tb_map**(*_*, *index*, *data*)

**_centroid_map**(*index*, ***, *image*, *energy*)

**_centroid_task**(*index*, *image*, *energy*)

**_histogram_task**(*index*, *total*, *double*, *default_range*)

**centroid_from_run**(*runNB*, *proposal=None*, *extra_fields=()*, *drop_first=False*, *subset=None*, *bins=None*, *return_hits=False*)

A combined function of *from_run()* and *centroid()*, which uses *extra_data* and *pasha* to avoid bulk loading of files.

**_centroid_ed_map**(*_*, *index*, *trainId*, *data*)

**static _mnemo_to_prop**(*mnemo*)

**_is_mnemo_in_run**(*mnemo*, *run*)

toolbox_scs.detectors.**get_pes_params**(*run*, *channel=None*)

Extract PES parameters for a given extra_data DataCollection. Parameters are gas, binding energy, retardation voltages or all voltages of the MPOD.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data
- **channel** (*str*) – Channel name or PES mnemonic, e.g. '2A' or 'PES_1Craw'. If None, or if the channel is not found in the data, the retardation voltage for all channels is retrieved.

**Returns**

**params** – dictionnary of PES parameters

**Return type**
dict

toolbox_scs.detectors.**get_pes_tof**(*proposal*, *runNB*, *mnemonic*, *start=0*, *origin=None*, *width=None*, *subtract_baseline=False*, *baseStart=None*, *baseWidth=40*, *merge_with=None*)

Extracts time-of-flight spectra from raw digitizer traces. The spectra are aligned by pulse Id using the SASE 3 bunch pattern. If origin is not None, a time coordinate in nanoseconds 'time_ns' is computed and added to the DataArray.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **mnemonic** (*str*) – mnemonic for PES, e.g. "PES_2Araw".
- **start** (*int*) – starting sample of the first spectrum in the raw trace.
- **origin** (*int*) – sample of the trace that corresponds to time-of-flight origin, also called prompt. Used to compute the 'time_ns' coordinates. If None, computation of 'time_ns' is skipped.
- **width** (*int*) – number of samples per spectra. If None, the number of samples for 4.5 MHz repetition rate is used.
- **subtract_baseline** (*bool*) – If True, subtract baseline defined by baseStart and baseWidth to each spectrum.
- **baseStart** (*int*) – starting sample of the baseline.
- **baseWidth** (*int*) – number of samples to average (starting from baseStart) for baseline calculation.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one.

**Returns**
**pes** – DataArray containing the PES time-of-flight spectra.

**Return type**
xarray DataArray

### Example

```
>>> import toolbox_scs as tb
>>> import toolbox_scs.detectors as tbdet
>>> proposal, runNB = 900447, 12
>>> pes = tbdet.get_pes_tof(proposal, runNB, 'PES_2Araw',
>>>                         start=2557, origin=76)
```

toolbox_scs.detectors.**save_pes_avg_traces**(*proposal*, *runNB*, *channels=None*, *subdir='usr/processed_runs'*)

Save average traces of PES into an h5 file.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.

- **channels** (*str or list*) – The PES channels or mnemonics, e.g. '2A', ['2A', '3C'], ['PES_1Araw', 'PES_4Draw', '3B']
- **subdir** (*str*) – subdirectory. The data is stored in <proposal path>/<subdir>/r{runNB:04d}/f'r{runNB:04d}-pes-data.h5'
- **Output** –
- **------** –
- **traces.** (*xarray Dataset saved in a h5 file containing the PES average*) –

toolbox_scs.detectors.**load_pes_avg_traces**(*proposal*, *runNB*, *channels=None*, *subdir='usr/processed_runs'*)

Load existing PES average traces.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **channels** (*str or list*) – The PES channels or mnemonics, e.g. '2A', ['2A', '3C'], ['PES_1Araw', 'PES_4Draw', '3B']
- **subdir** (*str*) – subdirectory. The data is stored in <proposal path>/<subdir>/r{runNB:04d}/f'r{runNB:04d}-pes-data.h5'
- **Output** –
- **------** –
- **ds** (*xarray Dataset*) – dataset containing the PES average traces.

class toolbox_scs.detectors.**Viking**(*proposalNB*)

The Viking analysis (spectrometer used in combination with Andor Newton camera)

The objects of this class contain the meta-information about the settings of the spectrometer, not the actual data, except possibly a dark image for background subtraction.

The actual data is loaded into *xarray`s via the method `from_run()*, and stays there.

**PROPOSAL**

the number of the proposal

**Type**
int

**X_RANGE**

the slice to take in the non-dispersive direction, in pixels. Defaults to the entire width.

**Type**
slice

**Y_RANGE**

the slice to take in the energy dispersive direction

**Type**
slice

**USE_DARK**

whether to do dark subtraction. Is initially *False*, magically switches to *True* if a dark has been loaded, but may be reset.

**Type**
bool

**ENERGY_CALIB**

The 2nd degree polynomial coefficients for calibration from pixel to energy. Defaults to [0, 1, 0] (no calibration applied).

**Type**
1D array (len=3)

**BL_POLY_DEG**

the dgree of the polynomial used for baseline subtraction. Defaults to 1.

**Type**
int

**BL_SIGNAL_RANGE**

the dispersive-axis range, defined by an interval [min, max], to avoid when fitting a polynomial for baseline subtraction. Multiple ranges can be provided in the form [[min1, max1], [min2, max2], ...].

**Type**
list

**FIELDS**

the fields to be loaded from the data. Add additional fields if so desired.

**Type**
list of str

### Example

proposal = 2953 v = Viking(proposal) v.X_RANGE = slice(0, 1900) v.Y_RANGE = slice(38, 80) v.ENERGY_CALIB = [1.47802667e-06, 2.30600328e-02, 5.15884589e+02] v.BL_SIGNAL_RANGE = [500, 545]

**set_params**(***params*)

**get_params**(**params*)

**from_run**(*runNB*, *add_attrs=True*)

load a run

Load the run *runNB*. A thin wrapper around *toolbox_scs.load*.

**Parameters**

- **runNB** (*int*) – the run number
- **add_attrs** (*bool*) – if True, adds the camera parameters as attributes to the dataset (see get_camera_params())
- **Output** –
- **------** –
- **ds** (*xarray Dataset*) – the dataset containing the camera images

### Example

data = v.from_run(145) # load run 145

data1 = v.from_run(145) # load run 145 data2 = v.from_run(155) # load run 155 data = xarray.concat([data1, data2], 'trainId') # combine both

**load_dark**(*runNB=None*)

**integrate**(*data*)

This function calculates the mean over the non-dispersive dimension to create a spectrum. If the camera parameters are known, the spectrum is multiplied by the number of photoelectrons per ADC count. A new variable "spectrum" is added to the data.

**get_camera_gain**(*run*)

Get the preamp gain of the camera in the Viking spectrometer for a specified run.

**Parameters**

- **run** (*extra_data DataCollection*) – information on the run
- **Output** –
- **------** –
- **gain** (*int*) –

**e_per_counts**(*run*, *gain=None*)

Conversion factor from camera digital counts to photoelectrons per count. The values can be found in the camera datasheet (Andor Newton) but they have been slightly corrected for High Sensitivity mode after analysis of runs 1204, 1207 and 1208, proposal 2937.

**Parameters**

- **run** (*extra_data DataCollection*) – information on the run
- **gain** (*int*) – the camera preamp gain
- **Output** –
- **------** –
- **ret** (*float*) – photoelectrons per count

**get_camera_params**(*run*)

**removePolyBaseline**(*data*)

Removes a polynomial baseline to a spectrum, assuming a fixed position for the signal.

**Parameters**

- **data** (*xarray Dataset*) – The Viking data containing the variable "spectrum"
- **Output** –
- **------** –
- **data** – the original dataset with the added variable "spectrum_nobl" containing the baseline subtracted spectra.

**xas**(*data*, *data_ref*, *thickness=1*, *plot=False*, *plot_errors=True*, *xas_ylim=(-1, 3)*)

Given two independent datasets (one with sample and one reference), this calculates the average XAS spectrum (absorption coefficient), associated standard deviation and standard error. The absorption coefficient is defined as -log(It/I0)/thickness.

**Parameters**

- **data** (*xarray Dataset*) – the dataset containing the spectra with sample
- **data_ref** (*xarray Dataset*) – the dataset containing the spectra without sample
- **thickness** (*float*) – the thickness used for the calculation of the absorption coefficient
- **plot** (*bool*) – If True, plot the resulting average spectra.
- **plot_errors** (*bool*) – If True, adds the 95% confidence interval on the spectra.
- **xas_ylim** (*tuple or list of float*) – the y limits for the XAS plot.
- **Output** –
- **------** –
- **xas** (*xarray Dataset*) – the dataset containing the computed XAS quantities: I0, It, absorptionCoef and their associated errors.

**calibrate**(*runList*, *plot=True*)

This routine determines the calibration coefficients to translate the camera pixels into energy in eV. The Viking spectrometer is calibrated using the beamline monochromator: runs with various monochromatized photon energy are recorded and their peak position on the detector are determined by Gaussian fitting. The energy vs. position data is then fitted to a second degree polynomial.

**Parameters**

- **runList** (*list of int*) – the list of runs containing the monochromatized spectra
- **plot** (*bool*) – if True, the spectra, their Gaussian fits and the calibration curve are plotted.
- **Output** –
- **------** –
- **energy_calib** (*np.array*) – the calibration coefficients (2nd degree polynomial)

toolbox_scs.detectors.**calibrate_xgm**(*run*, *data*, *xgm='SCS'*, *plot=False*)

Calculates the calibration factor F between the photon flux (slow signal) and the fast signal (pulse-resolved) of the sase 3 pulses. The calibrated fast signal is equal to the uncalibrated one multiplied by F.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **data** (*xarray Dataset*) – dataset containing the pulse-resolved sase 3 signal, e.g. 'SCS_SA3'
- **xgm** (*str*) – one in {'XTD10', 'SCS'}
- **plot** (*bool*) – If True, shows a plot of the photon flux, averaged fast signal and calibrated fast signal.

**Returns**

**F** – calibration factor F defined as: calibrated XGM [microJ] = F * fast XGM array ('SCS_SA3' or 'XTD10_SA3')

**Return type**

float

### Example

```
>>> import toolbox_scs as tb
>>> import toolbox_scs.detectors as tbdet
>>> run, data = tb.load(900074, 69, ['SCS_XGM'])
>>> ds = tbdet.get_xgm(run, merge_with=data)
>>> F = tbdet.calibrate_xgm(run, ds, plot=True)
>>> # Add calibrated XGM to the dataset:
>>> ds['SCS_SA3_uJ'] = F * ds['SCS_SA3']
```

toolbox_scs.detectors.**get_xgm**(*run*, *mnemonics=None*, *merge_with=None*, *indices=slice(0, None)*)

Load and/or computes XGM data. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing dataset (merge_with). The bunch pattern table is used to assign the pulse id coordinates if the number of pulses has changed during the run.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the xgm data.
- **mnemonics** (*str or list of str*) – mnemonics for XGM, e.g. "SCS_SA3" or ["XTD10_XGM", "SCS_XGM"]. If None, defaults to "SCS_SA3" in case no merge_with dataset is provided.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The XGM variables of merge_with (if any) will also be computed and merged.
- **indices** (*slice, list, 1D array*) – Pulse indices of the XGM array in case bunch pattern is missing.

**Returns**

merged with Dataset *merge_with* if provided.

**Return type**

xarray Dataset with pulse-resolved XGM variables aligned,

### Example

```
>>> import toolbox_scs as tb
>>> run, ds = tb.load(2212, 213, 'SCS_SA3')
>>> ds['SCS_SA3']
```

toolbox_scs.detectors.**__all__**

## toolbox_scs.misc

### Submodules

### toolbox_scs.misc.bunch_pattern

Toolbox for SCS.

Various utilities function to quickly process data measured at the SCS instruments.

### Module Contents

### Functions

| | |
|---|---|
| *npulses_has_changed*(run[, loc, run_mnemonics]) | Checks if the number of pulses has changed during the run for |
| *get_sase_pId*(run[, loc, run_mnemonics, bpt, merge_with]) | Returns the pulse Ids of the specified *loc* during a run. |
| *extractBunchPattern*([bp_table, key, runDir]) | generate the bunch pattern and number of pulses of a source directly from the |
| *pulsePatternInfo*(data[, plot]) | display general information on the pulse patterns operated by SASE1 and SASE3. |
| *repRate*([data, runNB, proposalNB, key]) | Calculates the pulse repetition rate (in kHz) in sase |

toolbox_scs.misc.bunch_pattern.**npulses_has_changed**(*run*, *loc='sase3'*, *run_mnemonics=None*)

Checks if the number of pulses has changed during the run for a specific location *loc* (='sase1', 'sase3', 'scs_ppl' or 'laser') If the source is not found in the run, returns True.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}
- **run_mnemonics** (*dict*) – the mnemonics for the run (see *menonics_for_run*)

**Returns**

**ret** – True if the number of pulses has changed or the source was not found, False if the number of pulses did not change.

**Return type**

bool

toolbox_scs.misc.bunch_pattern.**get_sase_pId**(*run*, *loc='sase3'*, *run_mnemonics=None*, *bpt=None*, *merge_with=None*)

Returns the pulse Ids of the specified *loc* during a run. If the number of pulses has changed during the run, it loads the bunch pattern table and extract all pulse Ids used.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}
- **run_mnemonics** (*dict*) – the mnemonics for the run (see *menonics_for_run*)
- **bpt** (*2D-array*) – The bunch pattern table. Used only if the number of pulses has changed. If None, it is loaded on the fly.
- **merge_with** (*xarray.Dataset*) – dataset that may contain the bunch pattern table to use in case the number of pulses has changed. If merge_with does not contain the bunch pattern table, it is loaded and added as a variable 'bunchPatternTable' to merge_with.

**Returns**

**pulseIds** – the pulse ids at the specified location. Returns None if the mnemonic is not in the run.

**Return type**

np.array

toolbox_scs.misc.bunch_pattern.**extractBunchPattern**(*bp_table=None*, *key='sase3'*, *runDir=None*)

generate the bunch pattern and number of pulses of a source directly from the bunch pattern table and not using the MDL device BUNCH_DECODER. This is inspired by the euxfel_bunch_pattern package, https://git.xfel.eu/gitlab/karaboDevices/euxfel_bunch_pattern Inputs:

**bp_table: DataArray corresponding to the mnemonics "bunchPatternTable".**
If None, the bunch pattern table is loaded using runDir.

key: str, ['sase1', 'sase2', 'sase3', 'scs_ppl'] runDir: extra-data DataCollection. Required only if bp_table is None.

**Outputs:**

bunchPattern: DataArray containing indices of the sase/laser pulses for each train npulses: DataArray containing the number of pulses for each train matched: 2-D DataArray mask (trainId x 2700), True where 'key' has pulses

toolbox_scs.misc.bunch_pattern.**pulsePatternInfo**(*data*, *plot=False*)

display general information on the pulse patterns operated by SASE1 and SASE3. This is useful to track changes of number of pulses or mode of operation of SASE1 and SASE3. It also determines which SASE comes first in the train and the minimum separation between the two SASE sub-trains.

**Inputs:**

data: xarray Dataset containing pulse pattern info from the bunch decoder MDL: {'sase1, sase3', 'npulses_sase1', 'npulses_sase3'} plot: bool enabling/disabling the plotting of the pulse patterns

**Outputs:**

print of pulse pattern info. If plot==True, plot of the pulse pattern.

toolbox_scs.misc.bunch_pattern.**repRate**(*data=None*, *runNB=None*, *proposalNB=None*, *key='sase3'*)

Calculates the pulse repetition rate (in kHz) in sase according to the bunch pattern and assuming a grid of 4.5 MHz.

### Inputs:

data: xarray Dataset containing pulse pattern, needed if runNB is none runNB: int or str, run number. Needed if data is None proposal: int or str, proposal where to find the run. Needed if data is None key: str in [sase1, sase2, sase3, scs_ppl], source for which the

repetition rate is calculated

### Output:

f: repetition rate in kHz

### toolbox_scs.misc.bunch_pattern_external

A collection of wrappers around the the euxfel_bunch_pattern pkg

The euxfel_bunch_pattern package provides generic methods to extract information from the bunch pattern tables. To ease its use from within the toolbox some of its methods are wrapped. Like this they show up in the users namespace in a self-explanatory way.

## Module Contents

## Functions

| | |
|---|---|
| *is_pulse_at*(bpt, loc) | Check for prescence of a pulse at the location provided. |
| *is_sase_3*(bpt) | Check for prescence of a SASE3 pulse. |
| *is_sase_1*(bpt) | Check for prescence of a SASE1 pulse. |
| *is_ppl*(bpt) | Check for prescence of pp-laser pulse. |

toolbox_scs.misc.bunch_pattern_external.**is_pulse_at**(*bpt*, *loc*)

Check for prescence of a pulse at the location provided.

**Parameters**

- **bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}

**Returns**
**boolean** – true if a pulse is present at *loc*.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.bunch_pattern_external.**is_sase_3**(*bpt*)

Check for prescence of a SASE3 pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if SASE3 pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.bunch_pattern_external.**is_sase_1**(*bpt*)

Check for prescence of a SASE1 pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if SASE1 pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.bunch_pattern_external.**is_ppl**(*bpt*)

Check for prescence of pp-laser pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if pp-laser pulse is present.

**Return type**
numpy array, xarray DataArray

## toolbox_scs.misc.laser_utils

### Module Contents

### Functions

| | |
|---|---|
| *positionToDelay*(pos[, origin, invert, reflections]) | converts a motor position in mm into optical delay in picosecond |
| *delayToPosition*(delay[, origin, invert, reflections]) | converts an optical delay in picosecond into a motor position in mm |
| *degToRelPower*(x[, theta0]) | converts a half-wave plate position in degrees into relative power |
| *fluenceCalibration*(hwp, power_mW, npulses, w0x[, w0y, ...]) | Given a measurement of relative powers or half wave plate angles |
| *align_ol_to_fel_pId*(ds[, ol_dim, fel_dim, offset, ...]) | Aligns the optical laser (OL) pulse Ids to the FEL pulse Ids. |

toolbox_scs.misc.laser_utils.**positionToDelay**(*pos*, *origin=0*, *invert=True*, *reflections=1*)

converts a motor position in mm into optical delay in picosecond Inputs:

pos: array-like delay stage motor position origin: motor position of time zero in mm invert: bool, inverts the sign of delay if True reflections: number of bounces in the delay stage

**Output:**
delay in picosecond

toolbox_scs.misc.laser_utils.**delayToPosition**(*delay*, *origin=0*, *invert=True*, *reflections=1*)

converts an optical delay in picosecond into a motor position in mm Inputs:

delay: array-like delay in ps origin: motor position of time zero in mm invert: bool, inverts the sign of delay if True reflections: number of bounces in the delay stage

**Output:**
delay in picosecond

toolbox_scs.misc.laser_utils.**degToRelPower**(*x*, *theta0=0*)

converts a half-wave plate position in degrees into relative power between 0 and 1. Inputs:

x: array-like positions of half-wave plate, in degrees theta0: position for which relative power is zero

**Output:**
array-like relative power

toolbox_scs.misc.laser_utils.**fluenceCalibration**(*hwp*, *power_mW*, *npulses*, *w0x*, *w0y=None*, *train_rep_rate=10*, *fit_order=1*, *plot=True*, *xlabel='HWP [%]'*)

Given a measurement of relative powers or half wave plate angles and averaged powers in mW, this routine calculates the corresponding fluence and fits a polynomial to the data.

**Parameters**

- **hwp** (*array-like (N)*) – angle or relative power from the half wave plate
- **power_mW** (*array-like (N)*) – measured power in mW by powermeter
- **npulses** (*int*) – number of pulses per train during power measurement
- **w0x** (*float*) – radius at 1/e^2 in x-axis in meter
- **w0y** (*float, optional*) – radius at 1/e^2 in y-axis in meter. If None, w0y=w0x is assumed.
- **train_rep_rate** (*float*) – repetition rate of the FEL, by default equals to 10 Hz.
- **fit_order** (*int*) – order of the polynomial fit
- **plot** (*bool*) – Plot the results if True
- **xlabel** (*str*) – xlabel for the plot
- **Output** –
- **------** –
- **F** (*ndarray (N)*) – fluence in mJ/cm^2
- **fit_F** (*ndarray*) – coefficients of the fluence polynomial fit
- **E** (*ndarray (N)*) – pulse energy in microJ
- **fit_E** (*ndarray*) – coefficients of the fluence polynomial fit

toolbox_scs.misc.laser_utils.**align_ol_to_fel_pId**(*ds*, *ol_dim='ol_pId'*, *fel_dim='sa3_pId'*, *offset=0*, *fill_value=np.nan*)

Aligns the optical laser (OL) pulse Ids to the FEL pulse Ids. The new OL coordinates are calculated as ds[ol_dim] + ds[fel_dim][0] + offset. The ol_dim is then removed, and if the number of OL and FEL pulses are different, the missing values are replaced by fill_value (NaN by default).

**Parameters**

- **ds** (*xarray.Dataset*) – Dataset containing both OL and FEL dimensions
- **ol_dim** (*str*) – name of the OL dimension
- **fel_dim** (*str*) – name of the FEL dimension
- **offset** (*int*) – offset added to the OL pulse Ids.
- **fill_value** (*(scalar or dict-like, optional)*) – Value to use for newly missing values. If a dict-like, maps variable names to fill values. Use a data array's name to refer to its values.
- **Output** –
- **------** –
- **ds** – The newly aligned dataset

toolbox_scs.misc.undulator

## Module Contents

## Functions

| | |
|---|---|
| *get_undulator_config*(run[, park_pos, plot]) | Extract the undulator cells configuration from a given run. |

toolbox_scs.misc.undulator.**get_undulator_config**(*run*, *park_pos=62.0*, *plot=True*)

Extract the undulator cells configuration from a given run. The gap size and K factor as well as the magnetic chicane delay and photon energy of colors 1, 2 and 3 are compiled into an xarray Dataset.

Note: This function looks at run control values, it does not reflect any change of values during the run. Do not use to extract configuration when scanning the undulator.

**Parameters**

- **run** (*EXtra-Data DataCollection*) – The run containing the undulator information
- **park_pos** (*float, optional*) – The parked position of a cell (i.e. when fully opened)
- **plot** (*bool, optional*) – If True, plot the undulator cells configuration

**Returns**

**cells** – The resulting dataset of the undulator configuration

**Return type**

xarray Dataset

## Package Contents

### Functions

| | |
|---|---|
| extractBunchPattern([bp_table, key, runDir]) | generate the bunch pattern and number of pulses of a source directly from the |
| get_sase_pId(run[, loc, run_mnemonics, bpt, merge_with]) | Returns the pulse Ids of the specified *loc* during a run. |
| npulses_has_changed(run[, loc, run_mnemonics]) | Checks if the number of pulses has changed during the run for |
| pulsePatternInfo(data[, plot]) | display general information on the pulse patterns operated by SASE1 and SASE3. |
| repRate([data, runNB, proposalNB, key]) | Calculates the pulse repetition rate (in kHz) in sase |
| is_sase_3(bpt) | Check for prescence of a SASE3 pulse. |
| is_sase_1(bpt) | Check for prescence of a SASE1 pulse. |
| is_ppl(bpt) | Check for prescence of pp-laser pulse. |
| is_pulse_at(bpt, loc) | Check for prescence of a pulse at the location provided. |
| degToRelPower(x[, theta0]) | converts a half-wave plate position in degrees into relative power |
| positionToDelay(pos[, origin, invert, reflections]) | converts a motor position in mm into optical delay in picosecond |
| delayToPosition(delay[, origin, invert, reflections]) | converts an optical delay in picosecond into a motor position in mm |
| fluenceCalibration(hwp, power_mW, npulses, w0x[, w0y, ...]) | Given a measurement of relative powers or half wave plate angles |
| align_ol_to_fel_pId(ds[, ol_dim, fel_dim, offset, ...]) | Aligns the optical laser (OL) pulse Ids to the FEL pulse Ids. |
| get_undulator_config(run[, park_pos, plot]) | Extract the undulator cells configuration from a given run. |

### Attributes

| |
|---|
| __all__ |

toolbox_scs.misc.**extractBunchPattern**(*bp_table=None*, *key='sase3'*, *runDir=None*)

generate the bunch pattern and number of pulses of a source directly from the bunch pattern table and not using the MDL device BUNCH_DECODER. This is inspired by the euxfel_bunch_pattern package, https://git.xfel.eu/gitlab/karaboDevices/euxfel_bunch_pattern Inputs:

**bp_table: DataArray corresponding to the mnemonics "bunchPatternTable".**
If None, the bunch pattern table is loaded using runDir.

key: str, ['sase1', 'sase2', 'sase3', 'scs_ppl'] runDir: extra-data DataCollection. Required only if bp_table is None.

**Outputs:**

bunchPattern: DataArray containing indices of the sase/laser pulses for each train npulses: DataArray containing the number of pulses for each train matched: 2-D DataArray mask (trainId x 2700), True where 'key' has pulses

toolbox_scs.misc.**get_sase_pId**(*run*, *loc='sase3'*, *run_mnemonics=None*, *bpt=None*, *merge_with=None*)

Returns the pulse Ids of the specified *loc* during a run. If the number of pulses has changed during the run, it loads the bunch pattern table and extract all pulse Ids used.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}
- **run_mnemonics** (*dict*) – the mnemonics for the run (see *menonics_for_run*)
- **bpt** (*2D-array*) – The bunch pattern table. Used only if the number of pulses has changed. If None, it is loaded on the fly.
- **merge_with** (*xarray.Dataset*) – dataset that may contain the bunch pattern table to use in case the number of pulses has changed. If merge_with does not contain the bunch pattern table, it is loaded and added as a variable 'bunchPatternTable' to merge_with.

**Returns**

**pulseIds** – the pulse ids at the specified location. Returns None if the mnemonic is not in the run.

**Return type**

np.array

toolbox_scs.misc.**npulses_has_changed**(*run*, *loc='sase3'*, *run_mnemonics=None*)

Checks if the number of pulses has changed during the run for a specific location *loc* (='sase1', 'sase3', 'scs_ppl' or 'laser') If the source is not found in the run, returns True.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}
- **run_mnemonics** (*dict*) – the mnemonics for the run (see *menonics_for_run*)

**Returns**

**ret** – True if the number of pulses has changed or the source was not found, False if the number of pulses did not change.

**Return type**

bool

toolbox_scs.misc.**pulsePatternInfo**(*data*, *plot=False*)

display general information on the pulse patterns operated by SASE1 and SASE3. This is useful to track changes of number of pulses or mode of operation of SASE1 and SASE3. It also determines which SASE comes first in the train and the minimum separation between the two SASE sub-trains.

**Inputs:**

data: xarray Dataset containing pulse pattern info from the bunch decoder MDL: {'sase1, sase3', 'npulses_sase1', 'npulses_sase3'} plot: bool enabling/disabling the plotting of the pulse patterns

**Outputs:**

print of pulse pattern info. If plot==True, plot of the pulse pattern.

toolbox_scs.misc.**repRate**(*data=None*, *runNB=None*, *proposalNB=None*, *key='sase3'*)

Calculates the pulse repetition rate (in kHz) in sase according to the bunch pattern and assuming a grid of 4.5 MHz.

#### Inputs:

data: xarray Dataset containing pulse pattern, needed if runNB is none runNB: int or str, run number. Needed if data is None proposal: int or str, proposal where to find the run. Needed if data is None key: str in [sase1, sase2, sase3, scs_ppl], source for which the

repetition rate is calculated

#### Output:

f: repetition rate in kHz

toolbox_scs.misc.**is_sase_3**(*bpt*)

Check for prescence of a SASE3 pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if SASE3 pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.**is_sase_1**(*bpt*)

Check for prescence of a SASE1 pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if SASE1 pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.**is_ppl**(*bpt*)

Check for prescence of pp-laser pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if pp-laser pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.**is_pulse_at**(*bpt*, *loc*)

Check for prescence of a pulse at the location provided.

**Parameters**

- **bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}

**Returns**
**boolean** – true if a pulse is present at *loc*.

**Return type**
numpy array, xarray DataArray

toolbox_scs.misc.**degToRelPower**(*x*, *theta0=0*)

converts a half-wave plate position in degrees into relative power between 0 and 1. Inputs:

x: array-like positions of half-wave plate, in degrees theta0: position for which relative power is zero

**Output:**
array-like relative power

toolbox_scs.misc.**positionToDelay**(*pos*, *origin=0*, *invert=True*, *reflections=1*)

converts a motor position in mm into optical delay in picosecond Inputs:

pos: array-like delay stage motor position origin: motor position of time zero in mm invert: bool, inverts the sign of delay if True reflections: number of bounces in the delay stage

**Output:**
delay in picosecond

toolbox_scs.misc.**delayToPosition**(*delay*, *origin=0*, *invert=True*, *reflections=1*)

converts an optical delay in picosecond into a motor position in mm Inputs:

delay: array-like delay in ps origin: motor position of time zero in mm invert: bool, inverts the sign of delay if True reflections: number of bounces in the delay stage

**Output:**
delay in picosecond

toolbox_scs.misc.**fluenceCalibration**(*hwp*, *power_mW*, *npulses*, *w0x*, *w0y=None*, *train_rep_rate=10*, *fit_order=1*, *plot=True*, *xlabel='HWP [%]'*)

Given a measurement of relative powers or half wave plate angles and averaged powers in mW, this routine calculates the corresponding fluence and fits a polynomial to the data.

**Parameters**

- **hwp** (*array-like (N)*) – angle or relative power from the half wave plate
- **power_mW** (*array-like (N)*) – measured power in mW by powermeter
- **npulses** (*int*) – number of pulses per train during power measurement
- **w0x** (*float*) – radius at 1/e^2 in x-axis in meter
- **w0y** (*float, optional*) – radius at 1/e^2 in y-axis in meter. If None, w0y=w0x is assumed.
- **train_rep_rate** (*float*) – repetition rate of the FEL, by default equals to 10 Hz.
- **fit_order** (*int*) – order of the polynomial fit
- **plot** (*bool*) – Plot the results if True
- **xlabel** (*str*) – xlabel for the plot
- **Output** –
- **------** –
- **F** (*ndarray (N)*) – fluence in mJ/cm^2
- **fit_F** (*ndarray*) – coefficients of the fluence polynomial fit

- **E** (*ndarray (N)*) – pulse energy in microJ
- **fit_E** (*ndarray*) – coefficients of the fluence polynomial fit

toolbox_scs.misc.**align_ol_to_fel_pId**(*ds*, *ol_dim='ol_pId'*, *fel_dim='sa3_pId'*, *offset=0*, *fill_value=np.nan*)

Aligns the optical laser (OL) pulse Ids to the FEL pulse Ids. The new OL coordinates are calculated as ds[ol_dim] + ds[fel_dim][0] + offset. The ol_dim is then removed, and if the number of OL and FEL pulses are different, the missing values are replaced by fill_value (NaN by default).

**Parameters**

- **ds** (*xarray.Dataset*) – Dataset containing both OL and FEL dimensions
- **ol_dim** (*str*) – name of the OL dimension
- **fel_dim** (*str*) – name of the FEL dimension
- **offset** (*int*) – offset added to the OL pulse Ids.
- **fill_value** (*(scalar or dict-like, optional)*) – Value to use for newly missing values. If a dict-like, maps variable names to fill values. Use a data array's name to refer to its values.
- **Output** –
- **------** –
- **ds** – The newly aligned dataset

toolbox_scs.misc.**get_undulator_config**(*run*, *park_pos=62.0*, *plot=True*)

Extract the undulator cells configuration from a given run. The gap size and K factor as well as the magnetic chicane delay and photon energy of colors 1, 2 and 3 are compiled into an xarray Dataset.

Note: This function looks at run control values, it does not reflect any change of values during the run. Do not use to extract configuration when scanning the undulator.

**Parameters**

- **run** (*EXtra-Data DataCollection*) – The run containing the undulator information
- **park_pos** (*float, optional*) – The parked position of a cell (i.e. when fully opened)
- **plot** (*bool, optional*) – If True, plot the undulator cells configuration

**Returns**
**cells** – The resulting dataset of the undulator configuration

**Return type**
xarray Dataset

toolbox_scs.misc.**__all__**

## toolbox_scs.routines

### Submodules

#### toolbox_scs.routines.Reflectivity

Toolbox for SCS.

Various utilities function to quickly process data measured at the SCS instrument.

### Module Contents

### Functions

| | |
|---|---|
| *reflectivity*(data[, Iokey, Irkey, delaykey, binWidth, ...]) | Computes the reflectivity R = 100*(Ir/Io[pumped] / Ir/Io[unpumped] - 1) |

toolbox_scs.routines.Reflectivity.**reflectivity**(*data*, *Iokey='FastADC5peaks'*, *Irkey='FastADC3peaks'*, *delaykey='PP800_DelayLine'*, *binWidth=0.05*, *positionToDelay=True*, *origin=None*, *invert=False*, *pumpedOnly=False*, *alternateTrains=False*, *pumpOnEven=True*, *Ioweights=False*, *plot=True*, *plotErrors=True*, *units='mm'*)

Computes the reflectivity R = 100*(Ir/Io[pumped] / Ir/Io[unpumped] - 1) as a function of delay. Delay can be a motor position in mm or an optical delay in ps, with possibility to convert from position to delay. The default scheme is alternating pulses pumped/unpumped/... in each train, also possible are alternating trains and pumped only. If fitting is enabled, attempts a double exponential (default) or step function fit.

**Parameters**

- **data** (*xarray Dataset*) – Dataset containing the Io, Ir and delay data
- **Iokey** (*str*) – Name of the Io variable
- **Irkey** (*str*) – Name of the Ir variable
- **delaykey** (*str*) – Name of the delay variable (motor position in mm or optical delay in ps)
- **binWidth** (*float*) – width of bin in units of delay variable
- **positionToDelay** (*bool*) – If True, adds a time axis converted from position axis according to origin and invert parameters. Ignored if origin is None.
- **origin** (*float*) – Position of time overlap, shown as a vertical line. Used if positionToDelay is True to convert position to time axis.
- **invert** (*bool*) – Used if positionToDelay is True to convert position to time axis.
- **pumpedOnly** (*bool*) – Assumes that all trains and pulses are pumped. In this case, Delta R is defined as Ir/Io.
- **alternateTrains** (*bool*) – If True, assumes that trains alternate between pumped and unpumped data.
- **pumpOnEven** (*bool*) – Only used if alternateTrains=True. If True, even trains are pumped, if False, odd trains are pumped.
- **Ioweights** (*bool*) – If True, computes the ratio of the means instead of the mean of the ratios Irkey/Iokey. Useful when dealing with large intensity variations.
- **plot** (*bool*) – If True, plots the results.
- **plotErrors** (*bool*) – If True, plots the 95% confidence interval.
- **Output** –
- **------** – xarray Dataset containing the binned Delta R, standard deviation, standard error, counts and delays, and the fitting results if full is True.

### toolbox_scs.routines.XAS

Toolbox for XAS experiments.

Based on the LCLS LO59 experiment libraries.

Time-resolved XAS and XMCD with uncertainties.



## Module Contents

## Functions

| | |
|---|---|
| *xas*(nrun[, bins, Iokey, Itkey, nrjkey, Iooffset, ...]) | Compute the XAS spectra from a xarray nrun. |
| *xasxmcd*(dataP, dataN) | Compute XAS and XMCD from data with both magnetic field direction |

toolbox_scs.routines.XAS.**xas**(*nrun*, *bins=None*, *Iokey='SCS_SA3'*, *Itkey='MCP3peaks'*, *nrjkey='nrj'*, *Iooffset=0*, *plot=False*, *fluorescence=False*)

Compute the XAS spectra from a xarray nrun.

**Inputs:**

nrun: xarray of SCS data bins: an array of bin-edges or an integer number of

desired bins or a float for the desired bin width.

Iokey: string for the Io fields, typically 'SCS_XGM' Itkey: string for the It fields, typically 'MCP3apd' nrjkey: string for the nrj fields, typically 'nrj' Iooffset: offset to apply on Io plot: boolean, displays a XAS spectrum if True fluorescence: boolean, if True, absorption is the ratio,

if False, absorption is negative log

**Outputs:**

**a dictionnary containing:**

nrj: the bin centers muA: the absorption sigmaA: standard deviation on the absorption sterrA: standard error on the absorption muIo: the mean of the Io counts: the number of events in each bin

toolbox_scs.routines.XAS.**xasxmcd**(*dataP*, *dataN*)

Compute XAS and XMCD from data with both magnetic field direction Inputs:

dataP: structured array for positive field dataN: structured array for negative field

**Outputs:**

xas: structured array for the sum xmcd: structured array for the difference

### toolbox_scs.routines.boz

Beam splitting Off-axis Zone plate analysis routines.



## Module Contents

## Classes

| | |
|---|---|
| *parameters* | Parameters contains all input parameters for the BOZ corrections. |

### Functions

| | |
|---|---|
| *get_roi_pixel_pos*(roi, params) | Compute fake or real pixel position of an roi from roi center. |
| *bad_pixel_map*(params) | Compute the bad pixels map. |
| *inspect_dark*(arr[, mean_th, std_th]) | Inspect dark run data and plot diagnostic. |
| *histogram_module*(arr[, mask]) | Compute a histogram of the 9 bits raw pixel values over a module. |
| *inspect_histogram*(arr[, arr_dark, mask, extra_lines]) | Compute and plot a histogram of the 9 bits raw pixel values. |
| *find_rois*(data_mean, threshold[, extended]) | Find rois from 3 beams configuration. |
| *find_rois_from_params*(params) | Find rois from 3 beams configuration. |
| *inspect_rois*(data_mean, rois[, threshold, allrois]) | Find rois from 3 beams configuration from mean module image. |
| *compute_flat_field_correction*(rois, params[, plot]) | |
| *inspect_flat_field_domain*(avg, rois, prod_th, ratio_th) | Extract beams roi from average image and compute the ratio. |
| *inspect_plane_fitting*(avg, rois[, domain, vmin, vmax]) | |
| *plane_fitting_domain*(avg, rois, prod_th, ratio_th) | Extract beams roi, compute their ratio and the domain. |
| *plane_fitting*(params) | Fit the plane flat-field normalization. |
| *ff_refine_crit*(p, alpha, params, arr_dark, arr, tid, ...) | Criteria for the ff_refine_fit. |
| *ff_refine_fit*(params[, crit]) | Refine the flat-field fit by minimizing data spread. |
| *nl_domain*(N, low, high) | Create the input domain where the non-linear correction defined. |
| *nl_lut*(domain, dy) | Compute the non-linear correction. |
| *nl_crit*(p, domain, alpha, arr_dark, arr, tid, rois, ...) | Criteria for the non linear correction. |
| *nl_crit_sk*(p, domain, alpha, arr_dark, arr, tid, rois, ...) | Non linear correction criteria, combining 'n' and 'p' as reference. |
| *nl_fit*(params, domain[, ff, crit]) | Fit non linearities correction function. |
| *inspect_nl_fit*(res_fit) | Plot the progress of the fit. |
| *snr*(sig, ref[, methods, verbose]) | Compute mean, std and SNR from transmitted and I0 signals. |
| *inspect_Fnl*(Fnl) | Plot the correction function Fnl. |
| *inspect_correction*(params[, gain]) | Comparison plot of the different corrections. |
| *inspect_correction_sk*(params, ff[, gain]) | Comparison plot of the different corrections, combining 'n' and 'p'. |
| *load_dssc_module*(proposalNB, runNB[, moduleNB, ...]) | Load single module dssc data as dask array. |
| *average_module*(arr[, dark, ret, mask, sat_roi, ...]) | Compute the average or std over a module. |
| *process_module*(arr, tid, dark, rois[, mask, ...]) | Process one module and extract roi intensity. |
| *process*(Fmodel, arr_dark, arr, tid, rois, mask, flat_field) | Process dark and run data with corrections. |
| *inspect_saturation*(data, gain[, Nbins]) | Plot roi integrated histogram of the data with saturation |

**class** toolbox_scs.routines.boz.**parameters**(*proposal*, *darkrun*, *run*, *module*, *gain*, *drop_intra_darks=True*)

Parameters contains all input parameters for the BOZ corrections.

This is used in beam splitting off-axis zone plate spectrocopy analysis as well as the during the determination of

correction parameters themselves to ensure they can be reproduced.

#### Inputs

proposal: int, proposal number darkrun: int, run number for the dark run run: int, run number for the data run module: int, DSSC module number gain: float, number of ph per bin drop_intra_darks: drop every second DSSC frame

**dask_load_persistently**(*dark_data_size_Gb=None*, *data_size_Gb=None*)

Load dask data array in memory.

#### Inputs

**dark_data_size_Gb: float, optional size of dark to load in memory,**
in Gb

data_size_Gb: float, optional size of data to load in memory, in Gb

**use_gpu**()

**set_mask**(*arr*)

Set mask of bad pixels.

#### Inputs

**arr: either a boolean array of a DSSC module image or a list of bad**
pixel indices

**get_mask**()

Get the boolean array bad pixel of a DSSC module.

**get_mask_idx**()

Get the list of bad pixel indices.

**flat_field_guess**(*guess=None*)

Set the flat-field guess parameter for the fit and returns it.

#### Inputs

**guess: a list of 8 floats, the 4 first to define the plane**
ax+by+cz+d=0 for 'n' beam and the 4 last for the 'p' beam in case mirror symmetry is disbaled

**set_flat_field**(*ff_params*, *ff_type='plane'*, *prod_th=None*, *ratio_th=None*)

Set the flat-field plane definition.

#### Inputs

ff_params: list of parameters ff_type: string identifying the type of flat field normalization, default is 'plane'.

**get_flat_field**()
Get the flat-field plane definition.

**set_Fnl**(*Fnl*)
Set the non-linear correction function.

**get_Fnl**()
Get the non-linear correction function.

**save**(*path='./'*)
Save the parameters as a JSON file.

#### Inputs

path: str, where to save the file, default to './'

**classmethod load**(*fname*)
Load parameters from a JSON file.

#### Inputs

fname: string, name a the JSON file to load

**__str__**()
Return str(self).

toolbox_scs.routines.boz.**get_roi_pixel_pos**(*roi*, *params*)
Compute fake or real pixel position of an roi from roi center.

#### Inputs:

roi: dictionnary params: parameters

#### Returns:

X, Y: 1-d array of pixel position.

toolbox_scs.routines.boz.**bad_pixel_map**(*params*)
Compute the bad pixels map.

#### Inputs

params: parameters

**rtype**
bad pixel map

toolbox_scs.routines.boz.**inspect_dark**(*arr*, *mean_th=(None, None)*, *std_th=(None, None)*)

Inspect dark run data and plot diagnostic.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) mean_th: tuple of threshold (low, high), default (None, None), to compute

a mask of good pixels for which the mean dark value lie inside this range

**std_th: tuple of threshold (low, high), default (None, None), to compute a**
mask of bad pixels for which the dark std value lie inside this range

**returns**
**fig**

**rtype**
matplotlib figure

toolbox_scs.routines.boz.**histogram_module**(*arr*, *mask=None*)

Compute a histogram of the 9 bits raw pixel values over a module.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) mask: optional bad pixel mask

**rtype**
histogram

toolbox_scs.routines.boz.**inspect_histogram**(*arr*, *arr_dark=None*, *mask=None*, *extra_lines=False*)

Compute and plot a histogram of the 9 bits raw pixel values.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) arr: dask array of reshaped dssc dark data (trainId, pulseId, x, y) mask: optional bad pixel mask extra_lines: boolean, default False, plot extra lines at period values

**returns**

- **(h, hd)** (*histogram of arr, arr_dark*)
- *figure*

toolbox_scs.routines.boz.**find_rois**(*data_mean*, *threshold*, *extended=False*)

Find rois from 3 beams configuration.

#### Inputs

data_mean: dark corrected average image threshold: threshold value to find beams extended: boolean, True to define additional ASICS based rois

**returns**
**rois**

**rtype**
dictionnary of rois

toolbox_scs.routines.boz.**find_rois_from_params**(*params*)

Find rois from 3 beams configuration.

#### Inputs

params: parameters

**returns**
**rois**

**rtype**
dictionnary of rois

toolbox_scs.routines.boz.**inspect_rois**(*data_mean*, *rois*, *threshold=None*, *allrois=False*)

Find rois from 3 beams configuration from mean module image.

#### Inputs

data_mean: mean module image threshold: float, default None, threshold value used to detect beams

boundaries

**allrois: boolean, default False, plot all rois defined in rois or only the**
main ones (['n', '0', 'p'])

**rtype**
matplotlib figure

toolbox_scs.routines.boz.**compute_flat_field_correction**(*rois*, *params*, *plot=False*)

toolbox_scs.routines.boz.**inspect_flat_field_domain**(*avg*, *rois*, *prod_th*, *ratio_th*, *vmin=None*, *vmax=None*)

Extract beams roi from average image and compute the ratio.

### Inputs

**avg: module average image with no saturated shots for the flat-field**
determination

rois: dictionnary or ROIs prod_th, ratio_th: tuple of floats for low and high threshold on

product and ratio

vmin: imshow vmin level, default None will use 5 percentile value vmax: imshow vmax level, default None will use 99.8 percentile value

**returns**

- **fig** (*matplotlib figure plotted*)
- **domain** (*a tuple (n_m, p_m) of domain for the 'n' and 'p' order*)

toolbox_scs.routines.boz.**inspect_plane_fitting**(*avg*, *rois*, *domain=None*, *vmin=None*, *vmax=None*)

toolbox_scs.routines.boz.**plane_fitting_domain**(*avg*, *rois*, *prod_th*, *ratio_th*)

Extract beams roi, compute their ratio and the domain.

### Inputs

**avg: module average image with no saturated shots for the flat-field**
determination

**rois: dictionnary or rois containing the 3 beams ['n', '0', 'p'] with '0'**
as the reference beam in the middle

**prod_th: float tuple, low and hight threshold level to determine the plane**
fitting domain on the product image of the orders

**ratio_th: float tuple, low and high threshold level to determine the plane**
fitting domain on the ratio image of the orders

**returns**

- **n** (*img ratio 'n'/'0'*)
- **n_m** (*mask where the the product 'n'*'0' is higher than 5 indicting that the*) – img ratio 'n'/'0' is defined
- **p** (*img ratio 'p'/'0'*)
- **p_m** (*mask where the the product 'p'*'0' is higher than 5 indicting that the*) – img ratio 'p'/'0' is defined

toolbox_scs.routines.boz.**plane_fitting**(*params*)

Fit the plane flat-field normalization.

### Inputs

params: parameters

**returns**
**res** – defines the plane as a*x + b*y + c*z + d = 0

**rtype**
the minimization result. The fitted vector res.x = [a, b, c, d]

toolbox_scs.routines.boz.**ff_refine_crit**(*p*, *alpha*, *params*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *sat_level=511*)

Criteria for the ff_refine_fit.

### Inputs

p: ff plane params: parameters arr_dark: dark data arr: data tid: train id of arr data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask fo good pixels sat_level: integer, default 511, at which level pixel begin to saturate

**rtype**
sum of standard deviation on binned 0th order intensity

toolbox_scs.routines.boz.**ff_refine_fit**(*params*, *crit=ff_refine_crit*)

Refine the flat-field fit by minimizing data spread.

### Inputs

params: parameters

**returns**

- **res** (*scipy minimize result. res.x is the optimized parameters*)
- **fitrres** (*iteration index arrays of criteria results for*) – [alpha=0, alpha, alpha=1]

toolbox_scs.routines.boz.**nl_domain**(*N*, *low*, *high*)

Create the input domain where the non-linear correction defined.

### Inputs

N: integer, number of control points or intervals low: input values below or equal to low will not be corrected high: input values higher or equal to high will not be corrected

**rtype**
array of 2**9 integer values with N segments

toolbox_scs.routines.boz.**nl_lut**(*domain*, *dy*)

Compute the non-linear correction.

### Inputs

**domain: input domain where dy is defined. For zero no correction is**
defined. For non-zero value x, dy[x] is applied.

**dy: a vector of deviation from linearity on control point homogeneously**
dispersed over 9 bits.

**returns**
**F_INL** – lookup table with 9 bits integer input

**rtype**
default None, non linear correction function given as a

toolbox_scs.routines.boz.**nl_crit**(*p*, *domain*, *alpha*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Criteria for the non linear correction.

### Inputs

p: vector of dy non linear correction domain: domain over which the non linear correction is defined alpha: float, coefficient scaling the cost of the correction function

in the criterion

arr_dark: dark data arr: data tid: train id of arr data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask fo good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**returns**

- *(1.0 - alpha)*err1 + alpha*err2, where err1 is the 1e8 times the mean of*
- *error squared from a transmission of 1.0 and err2 is the sum of the square*
- *of the deviation from the ideal detector response.*

toolbox_scs.routines.boz.**nl_crit_sk**(*p*, *domain*, *alpha*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Non linear correction criteria, combining ‘n’ and ‘p’ as reference.

### Inputs

p: vector of dy non linear correction domain: domain over which the non linear correction is defined alpha: float, coefficient scaling the cost of the correction function

in the criterion

arr_dark: dark data arr: data tid: train id of arr data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask fo good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**returns**

- *(1.0 - alpha)*err1 + alpha*err2, where err1 is the 1e8 times the mean of*
- *error squared from a transmission of 1.0 and err2 is the sum of the square*
- *of the deviation from the ideal detector response.*

toolbox_scs.routines.boz.**nl_fit**(*params*, *domain*, *ff=None*, *crit=None*)

Fit non linearities correction function.

### Inputs

params: parameters domain: array of index ff: array, flat field correction crit: function, criteria function

**returns**

- **res** (*scipy minimize result. res.x is the optimized parameters*)
- **fitrres** (*iteration index arrays of criteria results for*) – [alpha=0, alpha, alpha=1]

toolbox_scs.routines.boz.**inspect_nl_fit**(*res_fit*)

Plot the progress of the fit.

### Inputs

res_fit:

**rtype**
matplotlib figure

toolbox_scs.routines.boz.**snr**(*sig*, *ref*, *methods=None*, *verbose=False*)

Compute mean, std and SNR from transmitted and I0 signals.

### Inputs

sig: 1D signal samples ref: 1D reference samples methods: None by default or list of strings to select which methods to use.

Possible values are 'direct', 'weighted', 'diff'. In case of None, all methods will be calculated.

verbose: booleand, if True prints calculated values

**returns**

- *dictionnary of [methods][value] where value is 'mu' for mean and 's' for*
- *standard deviation.*

toolbox_scs.routines.boz.**inspect_Fnl**(*Fnl*)

Plot the correction function Fnl.

### Inputs

Fnl: non linear correction function lookup table

**rtype**
matplotlib figure

toolbox_scs.routines.boz.**inspect_correction**(*params*, *gain=None*)

Comparison plot of the different corrections.

### Inputs

params: parameters gain: float, default None, DSSC gain in ph/bin

**rtype**
matplotlib figure

toolbox_scs.routines.boz.**inspect_correction_sk**(*params*, *ff*, *gain=None*)

Comparison plot of the different corrections, combining 'n' and 'p'.

### Inputs

params: parameters gain: float, default None, DSSC gain in ph/bin

**rtype**
matplotlib figure

toolbox_scs.routines.boz.**load_dssc_module**(*proposalNB*, *runNB*, *moduleNB=15*, *subset=slice(None)*, *drop_intra_darks=True*, *persist=False*, *data_size_Gb=None*)

Load single module dssc data as dask array.

### Inputs

proposalNB: proposal number runNB: run number moduleNB: default 15, module number subset: default slice(None), subset of trains to load drop_intra_darks: boolean, default True, remove intra darks from the data persist: default False, load all data persistently in memory data_size_Gb: float, if persist is True, can optionaly restrict

the amount of data loaded for dark data and run data in Gb

**returns**

- **arr** (*dask array of reshaped dssc data (trainId, pulseId, x, y)*)
- **tid** (*array of train id number*)

toolbox_scs.routines.boz.**average_module**(*arr*, *dark=None*, *ret='mean'*, *mask=None*, *sat_roi=None*, *sat_level=300*, *F_INL=None*)

Compute the average or std over a module.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) dark: default None, dark to be substracted ret: string, either 'mean' to compute the mean or 'std' to compute the

standard deviation

mask: default None, mask of bad pixels to ignore sat_roi: roi over which to check for pixel with values larger than

sat_level to drop the image from the average or std

sat_level: int, minimum pixel value for a pixel to be considered saturated F_INL: default None, non linear correction function given as a

lookup table with 9 bits integer input

**rtype**
average or standard deviation image

toolbox_scs.routines.boz.**process_module**(*arr*, *tid*, *dark*, *rois*, *mask=None*, *sat_level=511*, *flat_field=None*, *F_INL=None*, *use_gpu=False*)

Process one module and extract roi intensity.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) tid: array of train id number dark: pulse resolved dark image to remove rois: dictionnary of rois mask: default None, mask of ignored pixels sat_level: integer, default 511, at which level pixel begin to saturate flat_field: default None, flat-field correction F_INL: default None, non-linear correction function given as a

lookup table with 9 bits integer input

**rtype**
dataset of extracted pulse and train resolved roi intensities.

toolbox_scs.routines.boz.**process**(*Fmodel*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Process dark and run data with corrections.

### Inputs

Fmodel: correction lookup table arr_dark: dark data arr: data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask of good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**rtype**
roi extracted intensities

toolbox_scs.routines.boz.**inspect_saturation**(*data*, *gain*, *Nbins=200*)

Plot roi integrated histogram of the data with saturation

### Inputs

data: xarray of roi integrated DSSC data gain: nominal DSSC gain in ph/bin Nbins: number of bins for the histogram, by default 200

**returns**

- **f** (*handle to the matplotlib figure*)
- **h** (*xarray of the histogram data*)

### toolbox_scs.routines.knife_edge

Toolbox for SCS.

Various utilities function to quickly process data measured at the SCS instrument.



## Module Contents

## Functions

| | |
|---|---|
| knife_edge(ds[, axisKey, signalKey, axisRange, p0, ...]) | Calculates the beam radius at 1/e^2 from a knife-edge scan by |

toolbox_scs.routines.knife_edge.**knife_edge**(*ds*, *axisKey='scannerX'*, *signalKey='FastADC4peaks'*, *axisRange=None*, *p0=None*, *full=False*, *plot=False*, *display=False*)

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(x, x0, w0, a, b) = a*erfc(np.sqrt(2)*(x-x0)/w0) + b with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **ds** (*xarray Dataset*) – dataset containing the detector signal and the motor position.
- **axisKey** (*str*) – key of the axis against which the knife-edge is performed.
- **signalKey** (*str*) – key of the detector signal.
- **axisRange** (*list of floats*) – edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 micrometers is assumed.
- **full** (*bool*) – If False, returns the beam radius and standard error. If True, returns the popt, pcov list of parameters and covariance matrix from scipy.optimize.curve_fit.
- **plot** (*bool*) – If True, plots the data and the result of the fit. Default is False.
- **display** (*bool*) – If True, displays info on the fit. True when plot is True, default is False.

**Returns**

error from the fit in mm. If full is True, returns parameters and covariance matrix from scipy.optimize.curve_fit function.

**Return type**

If full is False, tuple with beam radius at 1/e^2 in mm and standard

### Package Contents

### Classes

| | |
|---|---|
| *parameters* | Parameters contains all input parameters for the BOZ corrections. |

### Functions

| | |
|---|---|
| *xas*(nrun[, bins, Iokey, Itkey, nrjkey, Iooffset, ...]) | Compute the XAS spectra from a xarray nrun. |
| *xasxmcd*(dataP, dataN) | Compute XAS and XMCD from data with both magnetic field direction |
| *get_roi_pixel_pos*(roi, params) | Compute fake or real pixel position of an roi from roi center. |
| *bad_pixel_map*(params) | Compute the bad pixels map. |
| *inspect_dark*(arr[, mean_th, std_th]) | Inspect dark run data and plot diagnostic. |
| *histogram_module*(arr[, mask]) | Compute a histogram of the 9 bits raw pixel values over a module. |
| *inspect_histogram*(arr[, arr_dark, mask, extra_lines]) | Compute and plot a histogram of the 9 bits raw pixel values. |
| *find_rois*(data_mean, threshold[, extended]) | Find rois from 3 beams configuration. |
| *find_rois_from_params*(params) | Find rois from 3 beams configuration. |
| *inspect_rois*(data_mean, rois[, threshold, allrois]) | Find rois from 3 beams configuration from mean module image. |
| *compute_flat_field_correction*(rois, params[, plot]) | |
| *inspect_flat_field_domain*(avg, rois, prod_th, ratio_th) | Extract beams roi from average image and compute the ratio. |
| *inspect_plane_fitting*(avg, rois[, domain, vmin, vmax]) | |
| *plane_fitting_domain*(avg, rois, prod_th, ratio_th) | Extract beams roi, compute their ratio and the domain. |
| *plane_fitting*(params) | Fit the plane flat-field normalization. |
| *ff_refine_crit*(p, alpha, params, arr_dark, arr, tid, ...) | Criteria for the ff_refine_fit. |
| *ff_refine_fit*(params[, crit]) | Refine the flat-field fit by minimizing data spread. |
| *nl_domain*(N, low, high) | Create the input domain where the non-linear correction defined. |
| *nl_lut*(domain, dy) | Compute the non-linear correction. |
| *nl_crit*(p, domain, alpha, arr_dark, arr, tid, rois, ...) | Criteria for the non linear correction. |
| *nl_crit_sk*(p, domain, alpha, arr_dark, arr, tid, rois, ...) | Non linear correction criteria, combining 'n' and 'p' as reference. |
| *nl_fit*(params, domain[, ff, crit]) | Fit non linearities correction function. |
| *inspect_nl_fit*(res_fit) | Plot the progress of the fit. |
| *snr*(sig, ref[, methods, verbose]) | Compute mean, std and SNR from transmitted and I0 signals. |
| *inspect_Fnl*(Fnl) | Plot the correction function Fnl. |
| *inspect_correction*(params[, gain]) | Comparison plot of the different corrections. |
| *inspect_correction_sk*(params, ff[, gain]) | Comparison plot of the different corrections, combining 'n' and 'p'. |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| load_dssc_module(proposalNB, runNB[, moduleNB, ...]) | Load single module dssc data as dask array. |
| average_module(arr[, dark, ret, mask, sat_roi, ...]) | Compute the average or std over a module. |
| process_module(arr, tid, dark, rois[, mask, ...]) | Process one module and extract roi intensity. |
| process(Fmodel, arr_dark, arr, tid, rois, mask, flat_field) | Process dark and run data with corrections. |
| inspect_saturation(data, gain[, Nbins]) | Plot roi integrated histogram of the data with saturation |
| reflectivity(data[, Iokey, Irkey, delaykey, binWidth, ...]) | Computes the reflectivity R = 100*(Ir/Io[pumped] / Ir/Io[unpumped] - 1) |
| knife_edge | Toolbox for SCS. |

### Attributes

| |
|---|
| __all__ |

toolbox_scs.routines.**xas**(*nrun*, *bins=None*, *Iokey='SCS_SA3'*, *Itkey='MCP3peaks'*, *nrjkey='nrj'*, *Iooffset=0*, *plot=False*, *fluorescence=False*)

Compute the XAS spectra from a xarray nrun.

**Inputs:**

nrun: xarray of SCS data bins: an array of bin-edges or an integer number of

desired bins or a float for the desired bin width.

Iokey: string for the Io fields, typically 'SCS_XGM' Itkey: string for the It fields, typically 'MCP3apd' nrjkey: string for the nrj fields, typically 'nrj' Iooffset: offset to apply on Io plot: boolean, displays a XAS spectrum if True fluorescence: boolean, if True, absorption is the ratio,

if False, absorption is negative log

**Outputs:**

**a dictionnary containing:**

nrj: the bin centers muA: the absorption sigmaA: standard deviation on the absorption sterrA: standard error on the absorption muIo: the mean of the Io counts: the number of events in each bin

toolbox_scs.routines.**xasxmcd**(*dataP*, *dataN*)

Compute XAS and XMCD from data with both magnetic field direction Inputs:

dataP: structured array for positive field dataN: structured array for negative field

**Outputs:**

xas: structured array for the sum xmcd: structured array for the difference

**class** toolbox_scs.routines.**parameters**(*proposal*, *darkrun*, *run*, *module*, *gain*, *drop_intra_darks=True*)

Parameters contains all input parameters for the BOZ corrections.

This is used in beam splitting off-axis zone plate spectrocopy analysis as well as the during the determination of correction parameters themselves to ensure they can be reproduced.

#### Inputs

proposal: int, proposal number darkrun: int, run number for the dark run run: int, run number for the data run module: int, DSSC module number gain: float, number of ph per bin drop_intra_darks: drop every second DSSC frame

**dask_load_persistently**(*dark_data_size_Gb=None*, *data_size_Gb=None*)

Load dask data array in memory.

#### Inputs

**dark_data_size_Gb: float, optional size of dark to load in memory,**
in Gb

data_size_Gb: float, optional size of data to load in memory, in Gb

**use_gpu**()

**set_mask**(*arr*)

Set mask of bad pixels.

#### Inputs

**arr: either a boolean array of a DSSC module image or a list of bad**
pixel indices

**get_mask**()

Get the boolean array bad pixel of a DSSC module.

**get_mask_idx**()

Get the list of bad pixel indices.

**flat_field_guess**(*guess=None*)

Set the flat-field guess parameter for the fit and returns it.

#### Inputs

**guess: a list of 8 floats, the 4 first to define the plane**
ax+by+cz+d=0 for 'n' beam and the 4 last for the 'p' beam in case mirror symmetry is disbaled

**set_flat_field**(*ff_params*, *ff_type='plane'*, *prod_th=None*, *ratio_th=None*)

Set the flat-field plane definition.

#### Inputs

ff_params: list of parameters ff_type: string identifying the type of flat field normalization, default is 'plane'.

**get_flat_field**()
Get the flat-field plane definition.

**set_Fnl**(*Fnl*)
Set the non-linear correction function.

**get_Fnl**()
Get the non-linear correction function.

**save**(*path='./'*)
Save the parameters as a JSON file.

#### Inputs

path: str, where to save the file, default to './'

**classmethod load**(*fname*)
Load parameters from a JSON file.

#### Inputs

fname: string, name a the JSON file to load

**__str__**()
Return str(self).

toolbox_scs.routines.**get_roi_pixel_pos**(*roi*, *params*)
Compute fake or real pixel position of an roi from roi center.

#### Inputs:

roi: dictionnary params: parameters

#### Returns:

X, Y: 1-d array of pixel position.

toolbox_scs.routines.**bad_pixel_map**(*params*)
Compute the bad pixels map.

#### Inputs

params: parameters

**rtype**
bad pixel map

toolbox_scs.routines.**inspect_dark**(*arr*, *mean_th=(None, None)*, *std_th=(None, None)*)

Inspect dark run data and plot diagnostic.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) mean_th: tuple of threshold (low, high), default (None, None), to compute

a mask of good pixels for which the mean dark value lie inside this range

**std_th: tuple of threshold (low, high), default (None, None), to compute a**
mask of bad pixels for which the dark std value lie inside this range

**returns**
**fig**

**rtype**
matplotlib figure

toolbox_scs.routines.**histogram_module**(*arr*, *mask=None*)

Compute a histogram of the 9 bits raw pixel values over a module.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) mask: optional bad pixel mask

**rtype**
histogram

toolbox_scs.routines.**inspect_histogram**(*arr*, *arr_dark=None*, *mask=None*, *extra_lines=False*)

Compute and plot a histogram of the 9 bits raw pixel values.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) arr: dask array of reshaped dssc dark data (trainId, pulseId, x, y) mask: optional bad pixel mask extra_lines: boolean, default False, plot extra lines at period values

**returns**

- **(h, hd)** (*histogram of arr, arr_dark*)
- *figure*

toolbox_scs.routines.**find_rois**(*data_mean*, *threshold*, *extended=False*)

Find rois from 3 beams configuration.

#### Inputs

data_mean: dark corrected average image threshold: threshold value to find beams extended: boolean, True to define additional ASICS based rois

**returns**
**rois**

**rtype**
dictionnary of rois

toolbox_scs.routines.**find_rois_from_params**(*params*)

Find rois from 3 beams configuration.

#### Inputs

params: parameters

**returns**
**rois**

**rtype**
dictionnary of rois

toolbox_scs.routines.**inspect_rois**(*data_mean*, *rois*, *threshold=None*, *allrois=False*)

Find rois from 3 beams configuration from mean module image.

#### Inputs

data_mean: mean module image threshold: float, default None, threshold value used to detect beams

boundaries

**allrois: boolean, default False, plot all rois defined in rois or only the**
main ones (['n', '0', 'p'])

**rtype**
matplotlib figure

toolbox_scs.routines.**compute_flat_field_correction**(*rois*, *params*, *plot=False*)

toolbox_scs.routines.**inspect_flat_field_domain**(*avg*, *rois*, *prod_th*, *ratio_th*, *vmin=None*, *vmax=None*)

Extract beams roi from average image and compute the ratio.

#### Inputs

**avg: module average image with no saturated shots for the flat-field**
determination

rois: dictionnary or ROIs prod_th, ratio_th: tuple of floats for low and high threshold on

product and ratio

vmin: imshow vmin level, default None will use 5 percentile value vmax: imshow vmax level, default None will use 99.8 percentile value

**returns**

- **fig** (*matplotlib figure plotted*)
- **domain** (*a tuple (n_m, p_m) of domain for the 'n' and 'p' order*)

toolbox_scs.routines.**inspect_plane_fitting**(*avg*, *rois*, *domain=None*, *vmin=None*, *vmax=None*)

toolbox_scs.routines.**plane_fitting_domain**(*avg*, *rois*, *prod_th*, *ratio_th*)

Extract beams roi, compute their ratio and the domain.

### Inputs

**avg: module average image with no saturated shots for the flat-field**
determination

**rois: dictionnary or rois containing the 3 beams ['n', '0', 'p'] with '0'**
as the reference beam in the middle

**prod_th: float tuple, low and hight threshold level to determine the plane**
fitting domain on the product image of the orders

**ratio_th: float tuple, low and high threshold level to determine the plane**
fitting domain on the ratio image of the orders

**returns**

- **n** (*img ratio 'n'/'0'*)
- **n_m** (*mask where the the product 'n'*'0' is higher than 5 indicting that the*) – img ratio 'n'/'0' is defined
- **p** (*img ratio 'p'/'0'*)
- **p_m** (*mask where the the product 'p'*'0' is higher than 5 indicting that the*) – img ratio 'p'/'0' is defined

toolbox_scs.routines.**plane_fitting**(*params*)

Fit the plane flat-field normalization.

### Inputs

params: parameters

**returns**
**res** – defines the plane as a*x + b*y + c*z + d = 0

**rtype**
the minimization result. The fitted vector res.x = [a, b, c, d]

toolbox_scs.routines.**ff_refine_crit**(*p*, *alpha*, *params*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *sat_level=511*)

Criteria for the ff_refine_fit.

### Inputs

p: ff plane params: parameters arr_dark: dark data arr: data tid: train id of arr data rois: ['n', '0', 'p', 'sat'] rois mask: mask fo good pixels sat_level: integer, default 511, at which level pixel begin to saturate

**rtype**
sum of standard deviation on binned 0th order intensity

toolbox_scs.routines.**ff_refine_fit**(*params*, *crit=ff_refine_crit*)

Refine the flat-field fit by minimizing data spread.

### Inputs

params: parameters

**returns**

- **res** (*scipy minimize result. res.x is the optimized parameters*)
- **fitrres** (*iteration index arrays of criteria results for*) – [alpha=0, alpha, alpha=1]

toolbox_scs.routines.**nl_domain**(*N*, *low*, *high*)

Create the input domain where the non-linear correction defined.

### Inputs

N: integer, number of control points or intervals low: input values below or equal to low will not be corrected high: input values higher or equal to high will not be corrected

**rtype**
array of 2**9 integer values with N segments

toolbox_scs.routines.**nl_lut**(*domain*, *dy*)

Compute the non-linear correction.

### Inputs

**domain: input domain where dy is defined. For zero no correction is**
defined. For non-zero value x, dy[x] is applied.

**dy: a vector of deviation from linearity on control point homogeneously**
dispersed over 9 bits.

**returns**
**F_INL** – lookup table with 9 bits integer input

**rtype**
default None, non linear correction function given as a

toolbox_scs.routines.**nl_crit**(*p*, *domain*, *alpha*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Criteria for the non linear correction.

### Inputs

p: vector of dy non linear correction domain: domain over which the non linear correction is defined alpha: float, coefficient scaling the cost of the correction function

in the criterion

arr_dark: dark data arr: data tid: train id of arr data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask fo good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**returns**

- *(1.0 - alpha)*err1 + alpha*err2, where err1 is the 1e8 times the mean of*
- *error squared from a transmission of 1.0 and err2 is the sum of the square*
- *of the deviation from the ideal detector response.*

toolbox_scs.routines.**nl_crit_sk**(*p*, *domain*, *alpha*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Non linear correction criteria, combining ‘n’ and ‘p’ as reference.

### Inputs

p: vector of dy non linear correction domain: domain over which the non linear correction is defined alpha: float, coefficient scaling the cost of the correction function

in the criterion

arr_dark: dark data arr: data tid: train id of arr data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask fo good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**returns**

- *(1.0 - alpha)*err1 + alpha*err2, where err1 is the 1e8 times the mean of*
- *error squared from a transmission of 1.0 and err2 is the sum of the square*
- *of the deviation from the ideal detector response.*

toolbox_scs.routines.**nl_fit**(*params*, *domain*, *ff=None*, *crit=None*)

Fit non linearities correction function.

### Inputs

params: parameters domain: array of index ff: array, flat field correction crit: function, criteria function

**returns**

- **res** (*scipy minimize result. res.x is the optimized parameters*)
- **fitrres** (*iteration index arrays of criteria results for*) – [alpha=0, alpha, alpha=1]

toolbox_scs.routines.**inspect_nl_fit**(*res_fit*)

Plot the progress of the fit.

#### Inputs

res_fit:

**rtype**
matplotlib figure

toolbox_scs.routines.**snr**(*sig*, *ref*, *methods=None*, *verbose=False*)

Compute mean, std and SNR from transmitted and I0 signals.

#### Inputs

sig: 1D signal samples ref: 1D reference samples methods: None by default or list of strings to select which methods to use.

Possible values are 'direct', 'weighted', 'diff'. In case of None, all methods will be calculated.

verbose: booleand, if True prints calculated values

**returns**

- *dictionnary of [methods][value] where value is 'mu' for mean and 's' for*
- *standard deviation.*

toolbox_scs.routines.**inspect_Fnl**(*Fnl*)

Plot the correction function Fnl.

#### Inputs

Fnl: non linear correction function lookup table

**rtype**
matplotlib figure

toolbox_scs.routines.**inspect_correction**(*params*, *gain=None*)

Comparison plot of the different corrections.

#### Inputs

params: parameters gain: float, default None, DSSC gain in ph/bin

**rtype**
matplotlib figure

toolbox_scs.routines.**inspect_correction_sk**(*params*, *ff*, *gain=None*)

Comparison plot of the different corrections, combining 'n' and 'p'.

### Inputs

params: parameters gain: float, default None, DSSC gain in ph/bin

**rtype**
matplotlib figure

toolbox_scs.routines.**load_dssc_module**(*proposalNB*, *runNB*, *moduleNB=15*, *subset=slice(None)*, *drop_intra_darks=True*, *persist=False*, *data_size_Gb=None*)

Load single module dssc data as dask array.

### Inputs

proposalNB: proposal number runNB: run number moduleNB: default 15, module number subset: default slice(None), subset of trains to load drop_intra_darks: boolean, default True, remove intra darks from the data persist: default False, load all data persistently in memory data_size_Gb: float, if persist is True, can optionaly restrict

the amount of data loaded for dark data and run data in Gb

**returns**

- **arr** (*dask array of reshaped dssc data (trainId, pulseId, x, y)*)
- **tid** (*array of train id number*)

toolbox_scs.routines.**average_module**(*arr*, *dark=None*, *ret='mean'*, *mask=None*, *sat_roi=None*, *sat_level=300*, *F_INL=None*)

Compute the average or std over a module.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) dark: default None, dark to be substracted ret: string, either 'mean' to compute the mean or 'std' to compute the

standard deviation

mask: default None, mask of bad pixels to ignore sat_roi: roi over which to check for pixel with values larger than

sat_level to drop the image from the average or std

sat_level: int, minimum pixel value for a pixel to be considered saturated F_INL: default None, non linear correction function given as a

lookup table with 9 bits integer input

**rtype**
average or standard deviation image

toolbox_scs.routines.**process_module**(*arr*, *tid*, *dark*, *rois*, *mask=None*, *sat_level=511*, *flat_field=None*, *F_INL=None*, *use_gpu=False*)

Process one module and extract roi intensity.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) tid: array of train id number dark: pulse resolved dark image to remove rois: dictionnary of rois mask: default None, mask of ignored pixels sat_level: integer, default 511, at which level pixel begin to saturate flat_field: default None, flat-field correction F_INL: default None, non-linear correction function given as a

lookup table with 9 bits integer input

**rtype**
dataset of extracted pulse and train resolved roi intensities.

toolbox_scs.routines.**process**(*Fmodel*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Process dark and run data with corrections.

### Inputs

Fmodel: correction lookup table arr_dark: dark data arr: data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask of good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**rtype**
roi extracted intensities

toolbox_scs.routines.**inspect_saturation**(*data*, *gain*, *Nbins=200*)

Plot roi integrated histogram of the data with saturation

### Inputs

data: xarray of roi integrated DSSC data gain: nominal DSSC gain in ph/bin Nbins: number of bins for the histogram, by default 200

**returns**

- **f** (*handle to the matplotlib figure*)
- **h** (*xarray of the histogram data*)

toolbox_scs.routines.**reflectivity**(*data*, *Iokey='FastADC5peaks'*, *Irkey='FastADC3peaks'*, *delaykey='PP800_DelayLine'*, *binWidth=0.05*, *positionToDelay=True*, *origin=None*, *invert=False*, *pumpedOnly=False*, *alternateTrains=False*, *pumpOnEven=True*, *Ioweights=False*, *plot=True*, *plotErrors=True*, *units='mm'*)

Computes the reflectivity R = 100*(Ir/Io[pumped] / Ir/Io[unpumped] - 1) as a function of delay. Delay can be a motor position in mm or an optical delay in ps, with possibility to convert from position to delay. The default scheme is alternating pulses pumped/unpumped/... in each train, also possible are alternating trains and pumped only. If fitting is enabled, attempts a double exponential (default) or step function fit.

**Parameters**

- **data** (*xarray Dataset*) – Dataset containing the Io, Ir and delay data
- **Iokey** (*str*) – Name of the Io variable

- **Irkey** (*str*) – Name of the Ir variable
- **delaykey** (*str*) – Name of the delay variable (motor position in mm or optical delay in ps)
- **binWidth** (*float*) – width of bin in units of delay variable
- **positionToDelay** (*bool*) – If True, adds a time axis converted from position axis according to origin and invert parameters. Ignored if origin is None.
- **origin** (*float*) – Position of time overlap, shown as a vertical line. Used if positionToDelay is True to convert position to time axis.
- **invert** (*bool*) – Used if positionToDelay is True to convert position to time axis.
- **pumpedOnly** (*bool*) – Assumes that all trains and pulses are pumped. In this case, Delta R is defined as Ir/Io.
- **alternateTrains** (*bool*) – If True, assumes that trains alternate between pumped and unpumped data.
- **pumpOnEven** (*bool*) – Only used if alternateTrains=True. If True, even trains are pumped, if False, odd trains are pumped.
- **Ioweights** (*bool*) – If True, computes the ratio of the means instead of the mean of the ratios Irkey/Iokey. Useful when dealing with large intensity variations.
- **plot** (*bool*) – If True, plots the results.
- **plotErrors** (*bool*) – If True, plots the 95% confidence interval.
- **Output** –
- **------** – xarray Dataset containing the binned Delta R, standard deviation, standard error, counts and delays, and the fitting results if full is True.

toolbox_scs.routines.**knife_edge**(*ds*, *axisKey='scannerX'*, *signalKey='FastADC4peaks'*, *axisRange=None*, *p0=None*, *full=False*, *plot=False*, *display=False*)

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(x, x0, w0, a, b) = a*erfc(np.sqrt(2)*(x-x0)/w0) + b with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **ds** (*xarray Dataset*) – dataset containing the detector signal and the motor position.
- **axisKey** (*str*) – key of the axis against which the knife-edge is performed.
- **signalKey** (*str*) – key of the detector signal.
- **axisRange** (*list of floats*) – edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 micrometers is assumed.
- **full** (*bool*) – If False, returns the beam radius and standard error. If True, returns the popt, pcov list of parameters and covariance matrix from scipy.optimize.curve_fit.
- **plot** (*bool*) – If True, plots the data and the result of the fit. Default is False.
- **display** (*bool*) – If True, displays info on the fit. True when plot is True, default is False.

**Returns**

error from the fit in mm. If full is True, returns parameters and covariance matrix from scipy.optimize.curve_fit function.

**Return type**
If full is False, tuple with beam radius at 1/e^2 in mm and standard

toolbox_scs.routines.**__all__**

### toolbox_scs.test

## Submodules

### toolbox_scs.test.test_dssc_cls

## Module Contents

## Classes

| | |
|---|---|
| *TestDSSC* | A class whose instances are single test cases. |

## Functions

| |
|---|
| *setup_tmp_dir*() |
| *cleanup_tmp_dir*() |
| *list_suites*() |
| *suite*(*tests) |
| *main*(*cliargs) |

## Attributes

| |
|---|
| *log_root* |
| *suites* |
| *_temp_dirs* |
| *parser* |

toolbox_scs.test.test_dssc_cls.**log_root**

toolbox_scs.test.test_dssc_cls.**suites**

toolbox_scs.test.test_dssc_cls.**_temp_dirs = ['tmp']**

toolbox_scs.test.test_dssc_cls.**setup_tmp_dir**()

toolbox_scs.test.test_dssc_cls.**cleanup_tmp_dir**()

**class** toolbox_scs.test.test_dssc_cls.**TestDSSC**(*methodName='runTest'*)

Bases: unittest.TestCase

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the __init__ method, the base class __init__ method must always be called. It is important that subclasses should not change the signature of their __init__ method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when

the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- **longMessage: determines whether long messages (including repr of**
  objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- **maxDiff: sets the maximum length of a diff in failure messages**
  by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

**classmethod setUpClass**()

Hook method for setting up class fixture before running tests in the class.

**classmethod tearDownClass**()

Hook method for deconstructing the class fixture after running all tests in the class.

**test_create**()

**test_use_xgm_tim**()

**test_processing_quick**()

**test_normalization_all**()

toolbox_scs.test.test_dssc_cls.**list_suites**()

toolbox_scs.test.test_dssc_cls.**suite**(**tests*)

toolbox_scs.test.test_dssc_cls.**main**(**cliargs*)

toolbox_scs.test.test_dssc_cls.**parser**

### toolbox_scs.test.test_hrixs

#### Module Contents

#### Classes

| | |
|---|---|
| *TestHRIXS* | A class whose instances are single test cases. |

**class** toolbox_scs.test.test_hrixs.**TestHRIXS**(*methodName='runTest'*)

Bases: unittest.TestCase

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the __init__ method, the base class __init__ method must always be called. It is important that subclasses should not change the signature of their __init__ method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when

the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- **longMessage: determines whether long messages (including repr of**
  objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- **maxDiff: sets the maximum length of a diff in failure messages**
  by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

**test_integration**()

**test_centroid**()

**test_getparam**()

### toolbox_scs.test.test_misc

#### Module Contents

#### Classes

| | |
|---|---|
| *TestDataAccess* | A class whose instances are single test cases. |

### Functions

| | |
|---|---|
| *list_suites*() | |
| *suite*(*tests) | |
| *start_tests*(*cliargs) | |

### Attributes

| | |
|---|---|
| *proposalNB* | |
| *runNB* | |
| *suites* | |
| *parser* | |

toolbox_scs.test.test_misc.**proposalNB = 2511**

toolbox_scs.test.test_misc.**runNB = 176**

toolbox_scs.test.test_misc.**suites**

**class** toolbox_scs.test.test_misc.**TestDataAccess**(*methodName='runTest'*)

Bases: unittest.TestCase

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the __init__ method, the base class __init__ method must always be called. It is important that subclasses should not change the signature of their __init__ method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when

> the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- **longMessage: determines whether long messages (including repr of**
  objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.

- **maxDiff: sets the maximum length of a diff in failure messages**
  by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

**classmethod setUpClass()**

Hook method for setting up class fixture before running tests in the class.

**classmethod tearDownClass()**

Hook method for deconstructing the class fixture after running all tests in the class.

**setUp()**

Hook method for setting up the test fixture before exercising it.

**tearDown()**

Hook method for deconstructing the test fixture after testing it.

**test_isppl()**

**test_issase1()**

**test_issase3()**

**test_extractBunchPattern()**

**test_pulsePatternInfo()**

toolbox_scs.test.test_misc.**list_suites**()

toolbox_scs.test.test_misc.**suite**(**tests*)

toolbox_scs.test.test_misc.**start_tests**(**cliargs*)

toolbox_scs.test.test_misc.**parser**

## toolbox_scs.test.test_top_level

### Module Contents

### Classes

| | |
|---|---|
| *TestToolbox* | A class whose instances are single test cases. |

### Functions

| |
|---|
| *list_suites*() |
| *suite*(*tests) |
| *main*(*cliargs) |

### Attributes

| |
|---|
| *log_root* |
| *suites* |
| *parser* |

toolbox_scs.test.test_top_level.**log_root**

toolbox_scs.test.test_top_level.**suites**

**class** toolbox_scs.test.test_top_level.**TestToolbox**(*methodName='runTest'*)

Bases: unittest.TestCase

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the __init__ method, the base class __init__ method must always be called. It is important that subclasses should not change the signature of their __init__ method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when

> the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- **longMessage: determines whether long messages (including repr of**
  objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- **maxDiff: sets the maximum length of a diff in failure messages**
  by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

**classmethod setUpClass**()

Hook method for setting up class fixture before running tests in the class.

**classmethod tearDownClass**()

Hook method for deconstructing the class fixture after running all tests in the class.

**setUp**()

Hook method for setting up the test fixture before exercising it.

**tearDown**()

Hook method for deconstructing the test fixture after testing it.

**test_constant**()

test_load()

test_openrun()

test_openrunpath()

test_loadbinnedarray()

toolbox_scs.test.test_top_level.**list_suites**()

toolbox_scs.test.test_top_level.**suite**(**tests*)

toolbox_scs.test.test_top_level.**main**(**cliargs*)

toolbox_scs.test.test_top_level.**parser**

## toolbox_scs.test.test_utils

### Module Contents

### Classes

| | |
|---|---|
| *TestDataAccess* | A class whose instances are single test cases. |

### Functions

| |
|---|
| *list_suites*() |
| *suite*(*tests) |
| *main*(*cliargs) |

### Attributes

| |
|---|
| *suites* |
| *parser* |

toolbox_scs.test.test_utils.**suites**

toolbox_scs.test.test_utils.**list_suites**()

**class** toolbox_scs.test.test_utils.**TestDataAccess**(*methodName='runTest'*)

Bases: unittest.TestCase

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the __init__ method, the base class __init__ method must always be called. It is important that subclasses should not change the signature of their __init__ method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes: * failureException: determines which exception will be raised when

the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- **longMessage: determines whether long messages (including repr of**
  objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- **maxDiff: sets the maximum length of a diff in failure messages**
  by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

**classmethod setUpClass()**

Hook method for setting up class fixture before running tests in the class.

**classmethod tearDownClass()**

Hook method for deconstructing the class fixture after running all tests in the class.

**setUp()**

Hook method for setting up the test fixture before exercising it.

**tearDown()**

Hook method for deconstructing the test fixture after testing it.

**test_rundir1()**

**test_rundir2()**

**test_rundir3()**

toolbox_scs.test.test_utils.**suite**(**tests*)

toolbox_scs.test.test_utils.**main**(**cliargs*)

toolbox_scs.test.test_utils.**parser**

### toolbox_scs.util

#### Submodules

### toolbox_scs.util.exceptions

#### Module Contents

**exception** toolbox_scs.util.exceptions.**ToolBoxError**

Bases: Exception

Parent Toolbox exception. (to be defined)

**exception** toolbox_scs.util.exceptions.**ToolBoxPathError**(*message=''*, *path=''*)

Bases: *ToolBoxError*

Parent Toolbox exception. (to be defined)

**exception** toolbox_scs.util.exceptions.**ToolBoxTypeError**(*msg=''*, *dtype=''*)

Bases: *ToolBoxError*

Parent Toolbox exception. (to be defined)

**exception** toolbox_scs.util.exceptions.**ToolBoxValueError**(*msg=''*, *val=None*)

Bases: *ToolBoxError*

Parent Toolbox exception. (to be defined)

**exception** toolbox_scs.util.exceptions.**ToolBoxFileError**(*msg=''*, *val=''*)

Bases: *ToolBoxError*

Parent Toolbox exception. (to be defined)

### toolbox_scs.util.pkg

#### Module Contents

#### Functions

*get_version*()

toolbox_scs.util.pkg.**get_version**()

#### Submodules

### toolbox_scs.constants

#### Module Contents

toolbox_scs.constants.**mnemonics**

### toolbox_scs.load

Toolbox for SCS.

Various utilities function to quickly process data measured at the SCS instruments.



## Module Contents

## Functions

| | |
|---|---|
| *load*([proposalNB, runNB, fields, data, display, ...]) | Load a run and extract the data. Output is an xarray with aligned |
| *run_by_path*(path) | Return specified run |
| *find_run_path*(proposalNB, runNB[, data]) | Return the run path given the specified proposal and run numbers. |
| *open_run*(proposalNB, runNB[, subset]) | Get extra_data.DataCollection in a given proposal. |
| *get_array*([run, mnemonic, stepsize, subset, data, ...]) | Loads one data array for the specified mnemonic and rounds its values to |
| *load_run_values*(prop_or_run[, runNB, which]) | Load the run value for each mnemonic whose source is a CONTORL |
| *concatenateRuns*(runs) | Sorts and concatenate a list of runs with identical data variables |
| *check_data_rate*(run[, fields]) | Calculates the fraction of train ids that contain data in a run. |

toolbox_scs.load.**load**(*proposalNB=None*, *runNB=None*, *fields=None*, *data='all'*, *display=False*, *validate=False*, *subset=None*, *rois={}*, *extract_digitizers=True*, *extract_xgm=True*, *extract_bam=True*, *bunchPattern='sase3'*, *parallelize=True*)

Load a run and extract the data. Output is an xarray with aligned trainIds.

**Parameters**

- **proposalNB** (*str, int*) – proposal number e.g. 'p002252' or 2252
- **runNB** (*str, int*) – run number as integer
- **fields** (*str, list of str, list of dict*) – list of mnemonics to load specific data such as "fastccd", "SCS_XGM", or dictionnaries defining a custom mnemonic such as {"extra": {'source': 'SCS_CDIFFT_MAG/SUPPLY/CURRENT',

  'key': 'actual_current.value', 'dim': None}}
- **data** (*str or Sequence of str*) – 'raw', 'proc' (processed), or any other location relative to the proposal path with data per run to access. May also be 'all' (both 'raw' and 'proc') or a sequence of strings to load data from several locations, with later locations overwriting sources present in earlier ones. The default is 'raw'.
- **display** (*bool*) – whether to show the run.info or not
- **validate** (*bool*) – whether to run extra-data-validate or not
- **subset** (slice or extra_data.by_index or **numpy.s_**) – a subset of train that can be loaded with extra_data.by_index[:5] for the first 5 trains. If None, all trains are retrieved.

- **rois** (*dict*) – a dictionnary of mnemonics with a list of rois definition and the desired names, for example: {'fastccd': {'ref': {'roi': by_index[730:890, 535:720],
  'dim': ['ref_x', 'ref_y']},
  **'sam': {'roi':by_index[1050:1210, 535:720],**
  'dim': ['sam_x', 'sam_y']}}}
- **extract_digitizers** (*bool*) – If True, extracts the peaks from digitizer variables and aligns the pulse Id according to the fadc_bp bunch pattern.
- **extract_xgm** (*bool*) – If True, extracts the values from XGM variables (e.g. 'SCS_SA3', 'XTD10_XGM') and aligns the pulse Id with the sase1 / sase3 bunch pattern.
- **extract_bam** (*bool*) – If True, extracts the values from BAM variables (e.g. 'BAM1932M') and aligns the pulse Id with the sase3 bunch pattern.
- **bunchPattern** (*str*) – bunch pattern used to extract the Fast ADC pulses. A string or a dict as in:

  ```
  {'FFT_PD2': 'sase3', 'ILH_I0': 'scs_ppl'}
  ```

  Ignored if extract_digitizers=False.
- **parallelize** (*bool*) – from EXtra-Data: enable or disable opening files in parallel. Particularly useful if creating child processes is not allowed (e.g. in a daemonized multiprocessing.Process).

**Returns**
**run, ds** – extra_data DataCollection of the proposal and run number and an xarray Dataset with aligned trainIds and pulseIds

**Return type**
DataCollection, xarray.Dataset

### Example

```
>>> import toolbox_scs as tb
>>> run, data = tb.load(2212, 208, ['SCS_SA3', 'MCP2apd', 'nrj'])
```

toolbox_scs.load.**run_by_path**(*path*)

Return specified run

Wraps the extra_data RunDirectory routine, to ease its use for the scs-toolbox user.

**Parameters**
**path** (*str*) – path to the run directory

**Returns**
**run** – DataCollection object containing information about the specified run. Data can be loaded using built-in class methods.

**Return type**
extra_data.DataCollection

toolbox_scs.load.**find_run_path**(*proposalNB*, *runNB*, *data='raw'*)

Return the run path given the specified proposal and run numbers.

**Parameters**

- **proposalNB** (*(str, int)*) – proposal number e.g. 'p002252' or 2252
- **runNB** (*(str, int)*) – run number as integer
- **data** (*str*) – 'raw', 'proc' (processed) or 'all' (both 'raw' and 'proc') to access data from either or both of those folders. If 'all' is used, sources present in 'proc' overwrite those in 'raw'. The default is 'raw'.

**Returns**
**path** – The run path.

**Return type**
str

toolbox_scs.load.**open_run**(*proposalNB*, *runNB*, *subset=None*, ***kwargs*)

Get extra_data.DataCollection in a given proposal. Wraps the extra_data open_run routine and adds subset selection, out of convenience for the toolbox user. More information can be found in the extra_data documentation.

**Parameters**

- **proposalNB** (*(str, int)*) – proposal number e.g. 'p002252' or 2252
- **runNB** (*(str, int)*) – run number e.g. 17 or 'r0017'
- **subset** (slice or extra_data.by_index or **numpy.s_**) – a subset of train that can be loaded with extra_data.by_index[:5] for the first 5 trains. If None, all trains are retrieved.
- ****kwargs** –
- **--------** –
- **data** (*str*) – default -> 'raw'
- **include** (*str*) – default -> '*'

**Returns**
**run** – DataCollection object containing information about the specified run. Data can be loaded using built-in class methods.

**Return type**
extra_data.DataCollection

toolbox_scs.load.**get_array**(*run=None*, *mnemonic=None*, *stepsize=None*, *subset=None*, *data='raw'*, *proposalNB=None*, *runNB=None*)

Loads one data array for the specified mnemonic and rounds its values to integer multiples of stepsize for consistent grouping (except for stepsize=None). Returns a 1D array of ones if mnemonic is set to None.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data. Used if proposalNB and runNB are None.
- **mnemonic** (*str*) – Identifier of a single item in the mnemonic collection. None creates a dummy 1D array of ones with length equal to the number of trains.
- **stepsize** (*float*) – nominal stepsize of the array data - values will be rounded to integer multiples of this value.
- **subset** (slice or extra_data.by_index or **numpy.s_**) – a subset of train that can be loaded with extra_data.by_index[:5] for the first 5 trains. If None, all trains are retrieved.
- **data** (*str or Sequence of str*) – 'raw', 'proc' (processed), or any other location relative to the proposal path with data per run to access. May also be 'all' (both 'raw' and 'proc')

or a sequence of strings to load data from several locations, with later locations overwriting sources present in earlier ones. The default is 'raw'.

- **proposalNB** (*(str, int)*) – proposal number e.g. 'p002252' or 2252.
- **runNB** (*(str, int)*) – run number e.g. 17 or 'r0017'.

**Returns**
**data** – xarray DataArray containing rounded array values using the trainId as coordinate.

**Return type**
xarray.DataArray

**Raises**
***ToolBoxValueError*** – Exception: Toolbox specific exception, indicating a non-valid mnemonic entry

### Example

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2212, 235)
>>> mnemonic = 'PP800_PhaseShifter'
>>> data_PhaseShifter = tb.get_array(run, mnemonic, 0.5)
```

toolbox_scs.load.**load_run_values**(*prop_or_run*, *runNB=None*, *which='mnemonics'*)

Load the run value for each mnemonic whose source is a CONTORL source (see extra-data DataCollection.get_run_value() for details)

**Parameters**

- **prop_or_run** (*extra_data DataCollection or int*) – The run (DataCollection) to check for mnemonics. Alternatively, the proposal number (int), for which the runNB is also required.
- **runNB** (*int*) – The run number. Only used if the first argument is the proposal number.
- **which** (*str*) – 'mnemonics' or 'all'. If 'mnemonics', only the run values for the ToolBox mnemonics are retrieved. If 'all', a compiled dictionnary of all control sources run values is returned.
- **Output** –
- **------** –
- **run_values** (*a dictionnary containing the mnemonic or all run values.*) –

toolbox_scs.load.**concatenateRuns**(*runs*)

Sorts and concatenate a list of runs with identical data variables along the trainId dimension.

**Input:**
runs: (list) the xarray Datasets to concatenate

**Output:**
a concatenated xarray Dataset

toolbox_scs.load.**check_data_rate**(*run*, *fields=None*)

Calculates the fraction of train ids that contain data in a run.

**Parameters**

- **run** (*extra_data DataCollection*) – the DataCollection associated to the data.

- **fields** (*str, list of str or dict*) – mnemonics to check. If None, all mnemonics in the run are checked. A custom mnemonic can be defined with a dictionnary: {'extra': {'source': 'SCS_CDIFFT_MAG/SUPPLY/CURRENT', 'key': 'actual_current.value'}}
- **Output** –
- **------** – ret: dictionnary dictionnary with mnemonic as keys and fraction of train ids that contain data as values.

### toolbox_scs.mnemonics_machinery

Handling ToolBox mnemonics sub-routines



(contributions preferrably comply with pep8 code structure guidelines.)

## Module Contents

## Functions

| | |
|---|---|
| *mnemonics_for_run*(prop_or_run[, runNB]) | Returns the availble ToolBox mnemonics for a give extra_data |

toolbox_scs.mnemonics_machinery.**mnemonics_for_run**(*prop_or_run*, *runNB=None*)

Returns the availble ToolBox mnemonics for a give extra_data DataCollection, or a given proposal + run number.

**Parameters**

- **prop_or_run** (*extra_data DataCollection or int*) – The run (DataCollection) to check for mnemonics. Alternatively, the proposal number (int), for which the runNB is also required.
- **runNB** (*int*) – The run number. Only used if the first argument is the proposal number.

**Returns**

**mnemonics** – The dictionnary of mnemonics that are available in the run.

**Return type**

dict

### Example

```
>>> import toolbox_scs as tb
>>> tb.mnemonics_for_run(2212, 213)
```

### Package Contents

### Classes

| | |
|---|---|
| *AzimuthalIntegrator* | |
| *AzimuthalIntegratorDSSC* | |
| *DSSCBinner* | |
| *DSSCFormatter* | |
| *hRIXS* | The hRIXS analysis, especially curvature correction |
| *MaranaX* | A spin-off of the hRIXS class: with parallelized centroiding |
| *Viking* | The Viking analysis (spectrometer used in combination with Andor Newton |
| *parameters* | Parameters contains all input parameters for the BOZ corrections. |

### Functions

| | |
|---|---|
| *get_bam*(run[, mnemonics, merge_with, bunchPattern, ...]) | Load beam arrival monitor (BAM) data and align their pulse ID |
| *get_bam_params*(run[, mnemo_or_source]) | Extract the run values of bamStatus[1-3] and bamError. |
| *check_peak_params*(run, mnemonic[, raw_trace, ntrains, ...]) | Checks and plots the peak parameters (pulse window and baseline window |
| *get_digitizer_peaks*(run, mnemonic[, merge_with, ...]) | Automatically computes digitizer peaks. A source can be loaded on the |
| *get_laser_peaks*(run[, mnemonic, merge_with, ...]) | Extracts laser photodiode signal (peak intensity) from Fast ADC |
| *get_peaks*(run, data, mnemonic[, useRaw, autoFind, ...]) | Extract peaks from one source (channel) of a digitizer. |
| *get_tim_peaks*(run[, mnemonic, merge_with, ...]) | Automatically computes TIM peaks. Sources can be loaded on the |
| *digitizer_signal_description*(run[, digitizer]) | Check for the existence of signal description and return all corresponding |
| *get_dig_avg_trace*(run, mnemonic[, ntrains]) | Compute the average over ntrains evenly spaced accross all trains |
| *get_data_formatted*([filenames, data_list]) | Combines the given data into one dataset. For any of extra_data's data |
| *load_xarray*(fname[, group, form]) | Load stored xarray Dataset. |
| *save_attributes_h5*(fname[, data]) | Adding attributes to a hdf5 file. This function is intended to be used to |
| *save_xarray*(fname, data[, group, mode]) | Store xarray Dataset in the specified location |
| *create_dssc_bins*(name, coordinates, bins) | Creates a single entry for the dssc binner dictionary. The produced xarray |
| *get_xgm_formatted*(run_obj, xgm_name, dssc_frame_coords) | Load the xgm data and define coordinates along the pulse dimension. |

continues on next page

Table 2 – continued from previous page

| | |
|---|---|
| load_dssc_info(proposal, run_nr) | Loads the first data file for DSSC module 0 (this is hard-coded) |
| load_mask(fname, dssc_mask) | Load a DSSC mask file. |
| quickmask_DSSC_ASIC(poslist) | Returns a mask for the given DSSC geometry with ASICs given in poslist |
| process_dssc_data(proposal, run_nr, module, chunksize, ...) | Collects and reduces DSSC data for a single module. |
| extract_GH2(ds, run[, firstFrame, bunchPattern, gh2_dim]) | Select and align the frames of the Gotthard-II that have been exposed |
| get_pes_params(run[, channel]) | Extract PES parameters for a given extra_data DataCollection. |
| get_pes_tof(proposal, runNB, mnemonic[, start, ...]) | Extracts time-of-flight spectra from raw digitizer traces. The spectra |
| save_pes_avg_traces(proposal, runNB[, channels, subdir]) | Save average traces of PES into an h5 file. |
| load_pes_avg_traces(proposal, runNB[, channels, subdir]) | Load existing PES average traces. |
| calibrate_xgm(run, data[, xgm, plot]) | Calculates the calibration factor F between the photon flux (slow signal) |
| get_xgm(run[, mnemonics, merge_with, indices]) | Load and/or computes XGM data. Sources can be loaded on the |
| concatenateRuns(runs) | Sorts and concatenate a list of runs with identical data variables |
| find_run_path(proposalNB, runNB[, data]) | Return the run path given the specified proposal and run numbers. |
| get_array([run, mnemonic, stepsize, subset, data, ...]) | Loads one data array for the specified mnemonic and rounds its values to |
| load | Toolbox for SCS. |
| open_run(proposalNB, runNB[, subset]) | Get extra_data.DataCollection in a given proposal. |
| run_by_path(path) | Return specified run |
| load_run_values(prop_or_run[, runNB, which]) | Load the run value for each mnemonic whose source is a CONTORL |
| check_data_rate(run[, fields]) | Calculates the fraction of train ids that contain data in a run. |
| extractBunchPattern([bp_table, key, runDir]) | generate the bunch pattern and number of pulses of a source directly from the |
| get_sase_pId(run[, loc, run_mnemonics, bpt, merge_with]) | Returns the pulse Ids of the specified *loc* during a run. |
| npulses_has_changed(run[, loc, run_mnemonics]) | Checks if the number of pulses has changed during the run for |
| pulsePatternInfo(data[, plot]) | display general information on the pulse patterns operated by SASE1 and SASE3. |
| repRate([data, runNB, proposalNB, key]) | Calculates the pulse repetition rate (in kHz) in sase |
| is_sase_3(bpt) | Check for prescence of a SASE3 pulse. |
| is_sase_1(bpt) | Check for prescence of a SASE1 pulse. |
| is_ppl(bpt) | Check for prescence of pp-laser pulse. |
| is_pulse_at(bpt, loc) | Check for prescence of a pulse at the location provided. |
| degToRelPower(x[, theta0]) | converts a half-wave plate position in degrees into relative power |
| positionToDelay(pos[, origin, invert, reflections]) | converts a motor position in mm into optical delay in picosecond |

continues on next page

Table 2 – continued from previous page

| | |
|---|---|
| delayToPosition(delay[, origin, invert, reflections]) | converts an optical delay in picosecond into a motor position in mm |
| fluenceCalibration(hwp, power_mW, npulses, w0x[, w0y, ...]) | Given a measurement of relative powers or half wave plate angles |
| align_ol_to_fel_pId(ds[, ol_dim, fel_dim, offset, ...]) | Aligns the optical laser (OL) pulse Ids to the FEL pulse Ids. |
| get_undulator_config(run[, park_pos, plot]) | Extract the undulator cells configuration from a given run. |
| mnemonics_for_run(prop_or_run[, runNB]) | Returns the availble ToolBox mnemonics for a give extra_data |
| xas(nrun[, bins, Iokey, Itkey, nrjkey, Iooffset, ...]) | Compute the XAS spectra from a xarray nrun. |
| xasxmcd(dataP, dataN) | Compute XAS and XMCD from data with both magnetic field direction |
| get_roi_pixel_pos(roi, params) | Compute fake or real pixel position of an roi from roi center. |
| bad_pixel_map(params) | Compute the bad pixels map. |
| inspect_dark(arr[, mean_th, std_th]) | Inspect dark run data and plot diagnostic. |
| histogram_module(arr[, mask]) | Compute a histogram of the 9 bits raw pixel values over a module. |
| inspect_histogram(arr[, arr_dark, mask, extra_lines]) | Compute and plot a histogram of the 9 bits raw pixel values. |
| find_rois(data_mean, threshold[, extended]) | Find rois from 3 beams configuration. |
| find_rois_from_params(params) | Find rois from 3 beams configuration. |
| inspect_rois(data_mean, rois[, threshold, allrois]) | Find rois from 3 beams configuration from mean module image. |
| compute_flat_field_correction(rois, params[, plot]) | |
| inspect_flat_field_domain(avg, rois, prod_th, ratio_th) | Extract beams roi from average image and compute the ratio. |
| inspect_plane_fitting(avg, rois[, domain, vmin, vmax]) | |
| plane_fitting_domain(avg, rois, prod_th, ratio_th) | Extract beams roi, compute their ratio and the domain. |
| plane_fitting(params) | Fit the plane flat-field normalization. |
| ff_refine_crit(p, alpha, params, arr_dark, arr, tid, ...) | Criteria for the ff_refine_fit. |
| ff_refine_fit(params[, crit]) | Refine the flat-field fit by minimizing data spread. |
| nl_domain(N, low, high) | Create the input domain where the non-linear correction defined. |
| nl_lut(domain, dy) | Compute the non-linear correction. |
| nl_crit(p, domain, alpha, arr_dark, arr, tid, rois, ...) | Criteria for the non linear correction. |
| nl_crit_sk(p, domain, alpha, arr_dark, arr, tid, rois, ...) | Non linear correction criteria, combining 'n' and 'p' as reference. |
| nl_fit(params, domain[, ff, crit]) | Fit non linearities correction function. |
| inspect_nl_fit(res_fit) | Plot the progress of the fit. |
| snr(sig, ref[, methods, verbose]) | Compute mean, std and SNR from transmitted and I0 signals. |
| inspect_Fnl(Fnl) | Plot the correction function Fnl. |
| inspect_correction(params[, gain]) | Comparison plot of the different corrections. |
| inspect_correction_sk(params, ff[, gain]) | Comparison plot of the different corrections, combining 'n' and 'p'. |

continues on next page

Table 2 – continued from previous page

| | |
|---|---|
| *load_dssc_module*(proposalNB, runNB[, moduleNB, ...]) | Load single module dssc data as dask array. |
| *average_module*(arr[, dark, ret, mask, sat_roi, ...]) | Compute the average or std over a module. |
| *process_module*(arr, tid, dark, rois[, mask, ...]) | Process one module and extract roi intensity. |
| *process*(Fmodel, arr_dark, arr, tid, rois, mask, flat_field) | Process dark and run data with corrections. |
| *inspect_saturation*(data, gain[, Nbins]) | Plot roi integrated histogram of the data with saturation |
| *reflectivity*(data[, Iokey, Irkey, delaykey, binWidth, ...]) | Computes the reflectivity R = 100*(Ir/Io[pumped] / Ir/Io[unpumped] - 1) |
| *knife_edge*(ds[, axisKey, signalKey, axisRange, p0, ...]) | Calculates the beam radius at 1/e^2 from a knife-edge scan by |

### Attributes

| |
|---|
| *mnemonics* |
| *__all__* |
| *__all__* |
| *__all__* |
| *__all__* |

toolbox_scs.**mnemonics**

toolbox_scs.**__all__**

**class** toolbox_scs.**AzimuthalIntegrator**(*imageshape*, *center*, *polar_range*, *aspect=204 / 236*, ***kwargs*)

Bases: object

**_calc_dist_array**(*shape*, *center*, *aspect*)

Calculate pixel coordinates for the given shape.

**_calc_indices**(***kwargs*)

Calculates the list of indices for the flattened image array.

**_calc_polar_mask**(*polar_range*)

**calc_q**(*distance*, *wavelength*)

Calculate momentum transfer coordinate.

**Parameters**

- **distance** (*float*) – Sample - detector distance in meter
- **wavelength** (*float*) – wavelength of scattered light in meter

**Returns**

**deltaq** – Momentum transfer coordinate in 1/m

**Return type**

np.ndarray

__call__(*image*)

class toolbox_scs.**AzimuthalIntegratorDSSC**(*geom*, *polar_range*, *dxdy=(0, 0)*, ***kwargs*)

Bases: *AzimuthalIntegrator*

_calc_dist_array(*geom*, *dxdy*)

Calculate pixel coordinates for the given shape.

toolbox_scs.**get_bam**(*run*, *mnemonics=None*, *merge_with=None*, *bunchPattern='sase3'*, *pulseIds=None*)

Load beam arrival monitor (BAM) data and align their pulse ID according to the bunch pattern. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing data set (merge_with).

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the bam data.
- **mnemonics** (*str or list of str*) – mnemonics for BAM, e.g. "BAM1932M" or ["BAM414", "BAM1932M"]. the arrays are either taken from merge_with or loaded from the DataCollection run.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. If merge_with contains variables in mnemonics, they will be selected, aligned and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks. The pulse ID dimension will be named 'sa1_pId', 'sa3_pId' or 'ol_pId', respectively.
- **pulseIds** (*list, 1D array*) – Pulse Ids. If None, they are automatically loaded.

**Returns**

merged with Dataset *merge_with* if provided.

**Return type**

xarray Dataset with pulse-resolved BAM variables aligned,

### Example

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2711, 303)
>>> bam = tb.get_bam(run, 'BAM1932S')
```

toolbox_scs.**get_bam_params**(*run*, *mnemo_or_source='BAM1932S'*)

Extract the run values of bamStatus[1-3] and bamError.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the bam data.
- **mnemo_or_source** (*str*) – mnemonic of the BAM, e.g. 'BAM414', or source name, e.g. 'SCS_ILH_LAS/DOOCS/BAM_414_B2.

**Returns**

**params** – dictionnary containing the extracted parameters.

**Return type**

dict

**Note:** The extracted parameters are run values, they do not reflect any possible change during the run.

toolbox_scs.**check_peak_params**(*run*, *mnemonic*, *raw_trace=None*, *ntrains=200*, *params=None*, *plot=True*, *show_all=False*, *bunchPattern='sase3'*)

Checks and plots the peak parameters (pulse window and baseline window of a raw digitizer trace) used to compute the peak integration. These parameters are either set by the digitizer peak-integration settings, or are determined by a peak finding algorithm (used in get_tim_peaks or get_fast_adc_peaks) when the inputs are raw traces. The parameters can also be provided manually for visual inspection. The plot either shows the first and last pulse of the trace or the entire trace.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – ToolBox mnemonic of the digitizer data, e.g. 'MCP2apd'.
- **raw_trace** (*optional, 1D numpy array or xarray DataArray*) – Raw trace to display. If None, the average raw trace over ntrains of the corresponding channel is loaded (this can be time-consuming).
- **ntrains** (*optional, int*) – Only used if raw_trace is None. Number of trains used to calculate the average raw trace of the corresponding channel.
- **plot** (*bool*) – If True, displays the raw trace and peak integration regions.
- **show_all** (*bool*) – If True, displays the entire raw trace and all peak integration regions (this can be time-consuming). If False, shows the first and last pulse according to the bunchPattern.
- **bunchPattern** (*optional, str*) – Only used if plot is True. Checks the bunch pattern against the digitizer peak parameters and shows potential mismatch.

**Return type**
dictionnary of peak integration parameters

toolbox_scs.**get_digitizer_peaks**(*run*, *mnemonic*, *merge_with=None*, *bunchPattern='sase3'*, *integParams=None*, *keepAllSase=False*)

Automatically computes digitizer peaks. A source can be loaded on the fly via the mnemonic argument, or processed from an existing data set (merge_with). The bunch pattern table is used to assign the pulse id coordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonic for FastADC or ADQ412, e.g. "I0_ILHraw" or "MCP3apd". The data is either loaded from the DataCollection or taken from merge_with.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one.
- **bunchPattern** (*str or dict*) – 'sase1' or 'sase3' or 'scs_ppl', 'None': bunch pattern
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.
- **keepAllSase** (*bool*) – Only relevant in case of sase-dedicated trains. If True, all trains are kept, else only those of the bunchPattern are kept.

**Returns**

- *xarray Dataset with digitizer peak variables. Raw variables are*
- *substituted by the peak caclulated values (e.g. "FastADC2raw" becomes*
- *"FastADC2peaks").*

toolbox_scs.**get_laser_peaks**(*run*, *mnemonic=None*, *merge_with=None*, *bunchPattern='scs_ppl'*, *integParams=None*)

Extracts laser photodiode signal (peak intensity) from Fast ADC digitizer. Sources can be loaded on the fly via the mnemonics argument, and/or processed from an existing data set (merge_with). The PP laser bunch pattern is used to assign the pulse idcoordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonic for FastADC corresponding to laser signal, e.g. "FastADC2peaks" or 'I0_ILHraw'.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The FastADC variables of merge_with (if any) will also be computed and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks.
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.

**Returns**

- *xarray Dataset with all Fast ADC variables substituted by*
- *the peak caclulated values (e.g. "FastADC2raw" becomes*
- *"FastADC2peaks").*

toolbox_scs.**get_peaks**(*run*, *data*, *mnemonic*, *useRaw=True*, *autoFind=True*, *integParams=None*, *bunchPattern='sase3'*, *bpt=None*, *extra_dim=None*, *indices=None*)

Extract peaks from one source (channel) of a digitizer.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data
- **data** (*xarray DataArray or str*) – array containing the raw traces or peak-integrated values from the digitizer. If str, must be one of the ToolBox mnemonics. If None, the data is loaded via the source and key arguments.
- **mnemonic** (*str or dict*) – ToolBox mnemonic or dict with source and key as in {'source': 'SCS_UTC1_ADQ/ADC/1:network', 'key': 'digitizers.channel_1_A.raw.samples'}
- **useRaw** (*bool*) – If True, extract peaks from raw traces. If False, uses the APD (or peaks) data from the digitizer.
- **autoFind** (*bool*) – If True, finds integration parameters by inspecting the average raw trace. Only valid if useRaw is True.
- **integParams** (*dict*) – dictionnary containing the integration parameters for raw trace integration: 'pulseStart', 'pulseStop', 'baseStart', 'baseStop', 'period', 'npulses'. Not used if autoFind is True. All keys are required when bunch pattern is missing.
- **bunchPattern** (*str*) – match the peaks to the bunch pattern: 'sase1', 'sase3', 'scs_ppl'. This will dictate the name of the pulse ID coordinates: 'sa1_pId', 'sa3_pId' or 'scs_ppl'.
- **bpt** (*xarray DataArray*) – bunch pattern table
- **extra_dim** (*str*) – Name given to the dimension along the peaks. If None, the name is given according to the bunchPattern.

- **indices** (*array, slice*) – indices from the peak-integrated data to retrieve. Only required when bunch pattern is missing and useRaw is False.

**Return type**
xarray.DataArray containing digitizer peaks with pulse coordinates

toolbox_scs.**get_tim_peaks**(*run*, *mnemonic=None*, *merge_with=None*, *bunchPattern='sase3'*, *integParams=None*, *keepAllSase=False*)

Automatically computes TIM peaks. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing data set (merge_with). The bunch pattern table is used to assign the pulse id coordinates.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – mnemonics for TIM, e.g. "MCP2apd".
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The TIM variables of merge_with (if any) will also be computed and merged.
- **bunchPattern** (*str*) – 'sase1' or 'sase3' or 'scs_ppl', bunch pattern used to extract peaks. The pulse ID dimension will be named 'sa1_pId', 'sa3_pId' or 'ol_pId', respectively.
- **integParams** (*dict*) – dictionnary for raw trace integration, e.g. {'pulseStart':100, 'pulsestop':200, 'baseStart':50, 'baseStop':99, 'period':24, 'npulses':500}. If None, integration parameters are computed automatically.
- **keepAllSase** (*bool*) – Only relevant in case of sase-dedicated trains. If True, all trains are kept, else only those of the bunchPattern are kept.

**Returns**

- *xarray Dataset with TIM variables substituted by*
- *the peak caclulated values (e.g. "MCP2raw" becomes*
- *"MCP2peaks"), merged with Dataset *merge_with* if provided.*

toolbox_scs.**digitizer_signal_description**(*run*, *digitizer=None*)

Check for the existence of signal description and return all corresponding channels in a dictionnary.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **digitizer** (*str or list of str (default=None)*) – Name of the digitizer: one in ['FastADC', FastADC2', 'ADQ412', 'ADQ412_2] If None, all digitizers are used

**Returns**
**signal_description** – the digitizer channels.

**Return type**
dictionnary containing the signal description of

### Example

import toolbox_scs as tb run = tb.open_run(3481, 100) signals = tb.digitizer_signal_description(run) signals_fadc2 = tb.digitizer_signal_description(run, 'FastADC2')

toolbox_scs.**get_dig_avg_trace**(*run*, *mnemonic*, *ntrains=None*)

Compute the average over ntrains evenly spaced accross all trains of a digitizer trace.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **mnemonic** (*str*) – ToolBox mnemonic of the digitizer data, e.g. 'MCP2apd'.
- **ntrains** (*int*) – Number of trains used to calculate the average raw trace. If None, all trains are used.

**Returns**

**trace** – The average digitizer trace

**Return type**

DataArray

class toolbox_scs.**DSSCBinner**(*proposal_nr*, *run_nr*, *binners={}*, *xgm_name='SCS_SA3'*, *tim_names=['MCP1apd', 'MCP2apd', 'MCP3apd']*, *dssc_coords_stride=2*)

**__del__**()

**add_binner**(*name*, *binner*)

Add additional binner to internal dictionary

**Parameters**

- **name** (*str*) – name of binner to be created
- **binner** (*xarray.DataArray*) – An array that represents a map how the respective coordinate should be binned.

**Raises**

*ToolBoxValueError* – Exception: Raises exception in case the name does not correspond to a valid binner name. To be generalized.

**load_xgm**()

load xgm data and construct coordinate array according to corresponding dssc frame number.

**load_tim**()

load tim data and construct coordinate array according to corresponding dssc frame number.

**create_pulsemask**(*use_data='xgm'*, *threshold=(0, np.inf)*)

creates a mask for dssc frames according to measured xgm intensity. Once such a mask has been constructed, it will be used in the data reduction process to drop out-of-bounds pulses.

**get_info**()

Returns the expected shape of the binned dataset, in case binners have been defined.

**_bin_metadata**(*data*)

**get_xgm_binned**()

Bin the xgm data according to the binners of the dssc data. The result can eventually be merged into the final dataset by the DSSCFormatter.

**Returns**
**xgm_data** – xarray dataset containing the binned xgm data

**Return type**
xarray.DataSet

`get_tim_binned()`

Bin the tim data according to the binners of the dssc data. The result can eventually be merged into the final dataset by the DSSCFormatter.

**Returns**
**tim_data** – xarray dataset containing the binned tim data

**Return type**
xarray.DataSet

`process_data(modules=[], filepath='./', chunksize=512, backend='loky', n_jobs=None, dark_image=None, xgm_normalization=False, normevery=1)`

Load and bin dssc data according to self.bins. No data is returned by this method. The condensed data is written to file by the worker processes directly.

**Parameters**

- **modules** (*list of ints*) – a list containing the module numbers that should be processed. If empty, all modules are processed.
- **filepath** (*str*) – the path where the files containing the reduced data should be stored.
- **chunksize** (*int*) – The number of trains that should be read in one iterative step.
- **backend** (*str*) – joblib multiprocessing backend to be used. At the moment it can be any of joblibs standard backends: 'loky' (default), 'multiprocessing', 'threading'. Anything else than the default is experimental and not appropriately implemented in the dbdet member function 'bin_data'.
- **n_jobs** (*int*) – inversely proportional of the number of cpu's available for one job. Tasks within one job can grab a maximum of n_CPU_tot/n_jobs of cpu's. Note that when using the default backend there is no need to adjust this parameter with the current implementation.
- **dark_image** (*xarray.DataArray*) – DataArray with dimensions compatible with the loaded dssc data. If given, it will be subtracted from the dssc data before the binning. The dark image needs to be of dimension module, trainId, pulse, x and y.
- **xgm_normalization** (*boolean*) – if true, the dssc data is normalized by the xgm data before the binning.
- **normevery** (*int*) – integer indicating which out of normevery frame will be normalized.

`class toolbox_scs.DSSCFormatter(filepath)`

`combine_files(filenames=[])`

Read the files given in filenames, and store the data in the class variable 'data'. If no filenames are given, it tries to read the files stored in the class-internal variable '_filenames'.

**Parameters**
**filenames** (*list*) – list of strings containing the names of the files to be combined.

`add_dataArray(groups=[])`

Reads addional xarray-data from the first file given in the list of filenames. This assumes that all the files in the folder contain the same additional data. To be generalized.

**Parameters**
**groups** (*list*) – list of strings with the names of the groups in the h5 file, containing additional xarray data.

**add_attributes**(*attributes={}*)

Add additional information, such as run-type, as attributes to the formatted .h5 file.

**Parameters**
**attributes** (*dictionary*) – a dictionary, containing information or data of any kind, that will be added to the formatted .h5 file as attributes.

**save_formatted_data**(*filename*)

Create a .h5 file containing the main dataset in the group called 'data'. Additional groups will be created for the content of the variable 'data_array'. Metadata about the file is added in the form of attributes.

**Parameters**
**filename** (*str*) – the name of the file to be created

toolbox_scs.**get_data_formatted**(*filenames=[]*, *data_list=[]*)

Combines the given data into one dataset. For any of extra_data's data types, an xarray.Dataset is returned. The data is sorted along the 'module' dimension. The array dimension have the order 'trainId', 'pulse', 'module', 'x', 'y'. This order is required by the extra_geometry package.

**Parameters**

- **filenames** (*list of str*) – files to be combined as a list of names. Calls '_data_from_list' to actually load the data.
- **data_list** (*list*) – list containing the already loaded data

**Returns**
**data** – A xarray.Dataset containing the combined data.

**Return type**
xarray.Dataset

toolbox_scs.**load_xarray**(*fname*, *group='data'*, *form='dataset'*)

Load stored xarray Dataset. Comment: This function exists because of a problem with the standard netcdf engine that is malfunctioning due to related software installed in the exfel-python environment. May be dropped at some point.

**Parameters**

- **fname** (*str*) – filename as string
- **group** (*str*) – the name of the xarray dataset (group in h5 file).
- **form** (*str*) – specify whether the data to be loaded is a 'dataset' or a 'array'.

toolbox_scs.**save_attributes_h5**(*fname*, *data={}*)

Adding attributes to a hdf5 file. This function is intended to be used to attach metadata to a processed run.

**Parameters**

- **fname** (*str*) – filename as string
- **data** (*dictionary*) – the data that should be added to the file in form of a dictionary.

toolbox_scs.**save_xarray**(*fname*, *data*, *group='data'*, *mode='a'*)

Store xarray Dataset in the specified location

**Parameters**

- **data** (*xarray.DataSet*) – The data to be stored

- **fname** (*str, int*) – filename
- **overwrite** (*bool*) – overwrite existing data

**Raises**
*ToolBoxFileError* – Exception: File existed, but overwrite was set to False.

toolbox_scs.**create_dssc_bins**(*name*, *coordinates*, *bins*)

Creates a single entry for the dssc binner dictionary. The produced xarray data-array will later be used to perform grouping operations according to the given bins.

**Parameters**

- **name** (*str*) – name of the coordinate to be binned.
- **coordinates** (*numpy.ndarray*) – the original coordinate values (1D)
- **bins** (*numpy.ndarray*) – the bins according to which the corresponding dimension should be grouped.

**Returns**
**da** – A pre-formatted xarray.DataArray relating the specified dimension with its bins.

**Return type**
xarray.DataArray

### Examples

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2212, 235, include='*DA*')
```

1.) binner along 'pulse' dimension. Group data into two bins. >>> bins_pulse = ['pumped', 'unpumped'] * 10 >>> binner_pulse = tb.create_dssc_bins("pulse",

np.linspace(0,19,20, dtype=int), bins_pulse)

**2.) binner along 'train' dimension. Group data into bins corresponding**
to the positions of a delay stage for instance.

```
>>> bins_trainId = tb.get_array(run, 'PP800_PhaseShifter', 0.04)
>>> binner_train = tb.create_dssc_bins("trainId",
                           run.trainIds,
                           bins_trainId.values)
```

toolbox_scs.**get_xgm_formatted**(*run_obj*, *xgm_name*, *dssc_frame_coords*)

Load the xgm data and define coordinates along the pulse dimension.

**Parameters**

- **run_obj** (*extra_data.DataCollection*) – DataCollection object providing access to the xgm data to be loaded
- **xgm_name** (*str*) – valid mnemonic of a xgm source
- **dssc_frame_coords** (*int, list*) – defines which dssc frames should be normalized using data from the xgm.

**Returns**
**xgm** – xgm data with coordinate 'pulse'.

**Return type**
xarray.DataArray

toolbox_scs.**load_dssc_info**(*proposal*, *run_nr*)

Loads the first data file for DSSC module 0 (this is hardcoded) and returns the detector_info dictionary

**Parameters**

- **proposal** (*str, int*) – number of proposal
- **run_nr** (*str, int*) – number of run

**Returns**
**info** – {'dims': tuple, 'frames_per_train': int, 'total_frames': int}

**Return type**
dictionary

toolbox_scs.**load_mask**(*fname*, *dssc_mask*)

Load a DSSC mask file.

Copyright (c) 2019, Michael Schneider Copyright (c) 2020, SCS-team license: BSD 3-Clause License (see LICENSE_BSD for more info)

**Parameters**
**fname** (*str*) – string of the filename of the mask file

**Return type**
dssc_mask

toolbox_scs.**quickmask_DSSC_ASIC**(*poslist*)

Returns a mask for the given DSSC geometry with ASICs given in poslist blanked. poslist is a list of (module, row, column) tuples. Each module consists of 2 rows and 8 columns of individual ASICS.

Copyright (c) 2019, Michael Schneider Copyright (c) 2020, SCS-team license: BSD 3-Clause License (see LICENSE_BSD for more info)

toolbox_scs.**process_dssc_data**(*proposal*, *run_nr*, *module*, *chunksize*, *info*, *dssc_binners*, *path='./'*, *pulsemask=None*, *dark_image=None*, *xgm_mnemonic='SCS_SA3'*, *xgm_normalization=False*, *normevery=1*)

Collects and reduces DSSC data for a single module.

Copyright (c) 2020, SCS-team

**Parameters**

- **proposal** (*int*) – proposal number
- **run_nr** (*int*) – run number
- **module** (*int*) – DSSC module to process
- **chunksize** (*int*) – number of trains to load simultaneously
- **info** (*dictionary*) – dictionary containing keys 'dims', 'frames_per_train', 'total_frames', 'trainIds', 'number_of_trains'.
- **dssc_binners** (*dictionary*) – a dictionary containing binner objects created by the ToolBox member function "create_binner()"
- **path** (*str*) – location in which the .h5 files, containing the binned data, should be stored.
- **pulsemask** (*numpy.ndarray*) – array of booleans to be used to mask dssc data according to xgm data.

- **dark_image** (*xarray.DataArray*) – an xarray dataarray with matching coordinates with the loaded data. If dark_image is not None it will be subtracted from each individual dssc frame.
- **xgm_normalization** (*bool*) – true if the data should be divided by the corresponding xgm value.
- **xgm_mnemonic** (*str*) – Mnemonic of the xgm data to be used for normalization.
- **normevery** (*int*) – One out of normevery dssc frames will be normalized.

**Returns**
**module_data** – xarray datastructure containing data binned according to bins.

**Return type**
xarray.Dataset

toolbox_scs.**extract_GH2**(*ds*, *run*, *firstFrame=0*, *bunchPattern='scs_ppl'*, *gh2_dim='gh2_pId'*)

Select and align the frames of the Gotthard-II that have been exposed to light.

**Parameters**

- **ds** (*xarray.Dataset*) – The dataset containing GH2 data
- **run** (*extra_data.DataCollection*) – The run containing the bunch pattern source
- **firstFrame** (*int*) – The GH2 frame number corresponding to the first pulse of the train.
- **bunchPattern** (*str in ['scs_ppl', 'sase3']*) – the bunch pattern used to align data. For 'scs_ppl', the gh2_pId dimension in renamed 'ol_pId', and for 'sase3' gh2_pId is renamed 'sa3_pId'.
- **gh2_dim** (*str*) – The name of the dimension that corresponds to the Gotthard-II frames.

**Returns**
**nds** – The aligned and reduced dataset with only-data-containing GH2 variables.

**Return type**
xarray Dataset

**class** toolbox_scs.**hRIXS**(*proposalNB*, *detector='MaranaX'*)

The hRIXS analysis, especially curvature correction

The objects of this class contain the meta-information about the settings of the spectrometer, not the actual data, except possibly a dark image for background subtraction.

The actual data is loaded into `xarray`s, and stays there.

**PROPOSAL**

the number of the proposal

**Type**
int

**DETECTOR**

the detector to be used. Can be ['hRIXS_det', 'MaranaX'] defaults to 'hRIXS_det' for backward-compatibility.

**Type**
str

**X_RANGE**

the slice to take in the dispersive direction, in pixels. Defaults to the entire width.

**Type**
slice

**Y_RANGE**

the slice to take in the energy direction

**Type**
slice

**THRESHOLD**

pixel counts above which a hit candidate is assumed, for centroiding. use None if you want to give it in standard deviations instead.

**Type**
float

**STD_THRESHOLD**

same as THRESHOLD, in standard deviations.

**DBL_THRESHOLD**

threshold controling whether a detected hit is considered to be a double hit.

**BINS**

the number of bins used in centroiding

**Type**
int

**CURVE_A, CURVE_B**

the coefficients of the parabola for the curvature correction

**Type**
float

**USE_DARK**

whether to do dark subtraction. Is initially *False*, magically switches to *True* if a dark has been loaded, but may be reset.

**Type**
bool

**ENERGY_INTERCEPT, ENERGY_SLOPE**

The calibration from pixel to energy

**FIELDS**

the fields to be loaded from the data. Add additional fields if so desired.

### Example

proposal = 3145 h = hRIXS(proposal) h.Y_RANGE = slice(700, 900) h.CURVE_B = -3.695346575286939e-07 h.CURVE_A = 0.024084479232443695 h.ENERGY_SLOPE = 0.018387 h.ENERGY_INTERCEPT = 498.27 h.STD_THRESHOLD = 3.5

**DETECTOR_FIELDS**

**aggregators**

**set_params**(***params*)

**get_params**(**params*)

**from_run**(*runNB*, *proposal=None*, *extra_fields=()*, *drop_first=False*, *subset=None*)

load a run

Load the run *runNB*. A thin wrapper around *toolbox.load*. :param drop_first: if True, the first image in the run is removed from the dataset. :type drop_first: bool

### Example

data = h.from_run(145) # load run 145

data1 = h.from_run(145) # load run 145 data2 = h.from_run(155) # load run 155 data = xarray.concat([data1, data2], 'trainId') # combine both

**load_dark**(*runNB*, *proposal=None*)

load a dark run

Load the dark run *runNB* from *proposal*. The latter defaults to the current proposal. The dark is stored in this *hRIXS* object, and subsequent analyses use it for background subtraction.

### Example

h.load_dark(166) # load dark run 166

**find_curvature**(*runNB*, *proposal=None*, *plot=True*, *args=None*, ***kwargs*)

find the curvature correction coefficients

The hRIXS has some abberations which leads to the spectroscopic lines being curved on the detector. We approximate these abberations with a parabola for later correction.

Load a run and determine the curvature. The curvature is set in *self*, and returned as a pair of floats.

**Parameters**

- **runNB** (*int*) – the run number to use
- **proposal** (*int*) – the proposal to use, default to the current proposal
- **plot** (*bool*) – whether to plot the found curvature onto the data
- **args** (*pair of float, optional*) – a starting value to prime the fitting routine

### Example

h.find_curvature(155) # use run 155 to fit the curvature

**centroid_one**(*image*)

find the position of photons with sub-pixel precision

A photon is supposed to have hit the detector if the intensity within a 2-by-2 square exceeds a threshold. In this case the position of the photon is calculated as the center-of-mass in a 4-by-4 square.

Return the list of x, y coordinate pairs, corrected by the curvature.

**centroid_two**(*image*, *energy*)

determine position of photon hits on detector

The algrothm is taken from the ESRF RIXS toolbox. The thresholds for determining photon hits are given by the incident photon energy

The function returns arrays containing the single and double hits as x and y coordinates

**centroid**(*data*, *bins=None*, *method='auto'*)

calculate a spectrum by finding the centroid of individual photons

This takes the *xarray.Dataset data* and returns a copy of it, with a new *xarray.DataArray* named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

Added a key for switching between algorithims choices are "auto" and "manual" which selects for method for determining whether thresholds there is a photon hit. It changes whether centroid_one or centroid_two is used.

### Example

h.centroid(data) # find photons in all images of the run data.spectrum[0, :].plot() # plot the spectrum of the first image

**parabola**(*x*)

**integrate**(*data*)

calculate a spectrum by integration

This takes the *xarray data* and returns a copy of it, with a new dataarray named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

First the energy that corresponds to each pixel is calculated. Then all pixels within an energy range are summed, where the intensity of one pixel is distributed among the two energy ranges the pixel spans, proportionally to the overlap between the pixel and bin energy ranges.

The resulting data is normalized to one pixel, so the average intensity that arrived on one pixel.

### Example

h.integrate(data) # create spectrum by summing pixels data.spectrum[0, :].plot() # plot the spectrum of the first image

**aggregator**(*da*, *dim*)

**aggregate**(*ds*, *var=None*, *dim='trainId'*)

aggregate (i.e. mostly sum) all data within one dataset

take all images in a dataset and aggregate them and their metadata. For images, spectra and normalizations that means adding them, for others (e.g. delays) adding would not make sense, so we treat them properly. The aggregation functions of each variable are defined in the aggregators attribute of the class. If var is specified, group the dataset by var prior to aggregation. A new variable "counts" gives the number of frames aggregated in each group.

**Parameters**

- **ds** (*xarray Dataset*) – the dataset containing RIXS data

- **var** (*string*) – One of the variables in the dataset. If var is specified, the dataset is grouped by var prior to aggregation. This is useful for sorting e.g. a dataset that contains multiple delays.
- **dim** (*string*) – the dimension over which to aggregate the data

### Example

h.centroid(data) # create spectra from finding photons agg = h.aggregate(data) # sum all spectra agg.spectrum.plot() # plot the resulting spectrum

agg2 = h.aggregate(data, 'hRIXS_delay') # group data by delay agg2.spectrum[0, :].plot() # plot the spectrum for first value

**aggregate_ds**(*ds*, *dim='trainId'*)

**normalize**(*data*, *which='hRIXS_norm'*)

Adds a 'normalized' variable to the dataset defined as the ration between 'spectrum' and 'which'

**Parameters**

- **data** (*xarray Dataset*) – the dataset containing hRIXS data
- **which** (*string, default="hRIXS_norm"*) – one of the variables of the dataset, usually "hRIXS_norm" or "counts"

**class** toolbox_scs.**MaranaX**(**args*, ***kwargs*)

Bases: *hRIXS*

A spin-off of the hRIXS class: with parallelized centroiding

**NUM_MAX_HITS = 30**

**centroid**(*data*, *bins=None*, ***kwargs*)

calculate a spectrum by finding the centroid of individual photons

This takes the *xarray.Dataset data* and returns a copy of it, with a new *xarray.DataArray* named *spectrum* added, which contains the energy spectrum calculated for each hRIXS image.

Added a key for switching between algorithims choices are "auto" and "manual" which selects for method for determining whether thresholds there is a photon hit. It changes whether centroid_one or centroid_two is used.

### Example

h.centroid(data) # find photons in all images of the run data.spectrum[0, :].plot() # plot the spectrum of the first image

**_centroid_tb_map**(*_*, *index*, *data*)

**_centroid_map**(*index*, ***, *image*, *energy*)

**_centroid_task**(*index*, *image*, *energy*)

**_histogram_task**(*index*, *total*, *double*, *default_range*)

**centroid_from_run**(*runNB*, *proposal=None*, *extra_fields=()*, *drop_first=False*, *subset=None*, *bins=None*, *return_hits=False*)

A combined function of *from_run()* and *centroid()*, which uses *extra_data* and *pasha* to avoid bulk loading of files.

**_centroid_ed_map**(_, *index*, *trainId*, *data*)

**static _mnemo_to_prop**(*mnemo*)

**_is_mnemo_in_run**(*mnemo*, *run*)

toolbox_scs.**get_pes_params**(*run*, *channel=None*)

Extract PES parameters for a given extra_data DataCollection. Parameters are gas, binding energy, retardation voltages or all voltages of the MPOD.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data
- **channel** (*str*) – Channel name or PES mnemonic, e.g. '2A' or 'PES_1Craw'. If None, or if the channel is not found in the data, the retardation voltage for all channels is retrieved.

**Returns**

**params** – dictionnary of PES parameters

**Return type**

dict

toolbox_scs.**get_pes_tof**(*proposal*, *runNB*, *mnemonic*, *start=0*, *origin=None*, *width=None*, *subtract_baseline=False*, *baseStart=None*, *baseWidth=40*, *merge_with=None*)

Extracts time-of-flight spectra from raw digitizer traces. The spectra are aligned by pulse Id using the SASE 3 bunch pattern. If origin is not None, a time coordinate in nanoseconds 'time_ns' is computed and added to the DataArray.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **mnemonic** (*str*) – mnemonic for PES, e.g. "PES_2Araw".
- **start** (*int*) – starting sample of the first spectrum in the raw trace.
- **origin** (*int*) – sample of the trace that corresponds to time-of-flight origin, also called prompt. Used to compute the 'time_ns' coordinates. If None, computation of 'time_ns' is skipped.
- **width** (*int*) – number of samples per spectra. If None, the number of samples for 4.5 MHz repetition rate is used.
- **subtract_baseline** (*bool*) – If True, subtract baseline defined by baseStart and baseWidth to each spectrum.
- **baseStart** (*int*) – starting sample of the baseline.
- **baseWidth** (*int*) – number of samples to average (starting from baseStart) for baseline calculation.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one.

**Returns**
**pes** – DataArray containing the PES time-of-flight spectra.

**Return type**
xarray DataArray

### Example

```
>>> import toolbox_scs as tb
>>> import toolbox_scs.detectors as tbdet
>>> proposal, runNB = 900447, 12
>>> pes = tbdet.get_pes_tof(proposal, runNB, 'PES_2Araw',
>>>                         start=2557, origin=76)
```

toolbox_scs.**save_pes_avg_traces**(*proposal*, *runNB*, *channels=None*, *subdir='usr/processed_runs'*)

Save average traces of PES into an h5 file.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **channels** (*str or list*) – The PES channels or mnemonics, e.g. '2A', ['2A', '3C'], ['PES_1Araw', 'PES_4Draw', '3B']
- **subdir** (*str*) – subdirectory. The data is stored in <proposal path>/<subdir>/r{runNB:04d}/f'r{runNB:04d}-pes-data.h5'
- **Output** –
- **------** –
- **traces.** (*xarray Dataset saved in a h5 file containing the PES average*) –

toolbox_scs.**load_pes_avg_traces**(*proposal*, *runNB*, *channels=None*, *subdir='usr/processed_runs'*)

Load existing PES average traces.

**Parameters**

- **proposal** (*int*) – The proposal number.
- **runNB** (*int*) – The run number.
- **channels** (*str or list*) – The PES channels or mnemonics, e.g. '2A', ['2A', '3C'], ['PES_1Araw', 'PES_4Draw', '3B']
- **subdir** (*str*) – subdirectory. The data is stored in <proposal path>/<subdir>/r{runNB:04d}/f'r{runNB:04d}-pes-data.h5'
- **Output** –
- **------** –
- **ds** (*xarray Dataset*) – dataset containing the PES average traces.

**class** toolbox_scs.**Viking**(*proposalNB*)

The Viking analysis (spectrometer used in combination with Andor Newton camera)

The objects of this class contain the meta-information about the settings of the spectrometer, not the actual data, except possibly a dark image for background subtraction.

The actual data is loaded into *xarray`s via the method `from_run()*, and stays there.

**PROPOSAL**

the number of the proposal

**Type**
int

**X_RANGE**

the slice to take in the non-dispersive direction, in pixels. Defaults to the entire width.

**Type**
slice

**Y_RANGE**

the slice to take in the energy dispersive direction

**Type**
slice

**USE_DARK**

whether to do dark subtraction. Is initially *False*, magically switches to *True* if a dark has been loaded, but may be reset.

**Type**
bool

**ENERGY_CALIB**

The 2nd degree polynomial coefficients for calibration from pixel to energy. Defaults to [0, 1, 0] (no calibration applied).

**Type**
1D array (len=3)

**BL_POLY_DEG**

the dgree of the polynomial used for baseline subtraction. Defaults to 1.

**Type**
int

**BL_SIGNAL_RANGE**

the dispersive-axis range, defined by an interval [min, max], to avoid when fitting a polynomial for baseline subtraction. Multiple ranges can be provided in the form [[min1, max1], [min2, max2], ...].

**Type**
list

**FIELDS**

the fields to be loaded from the data. Add additional fields if so desired.

**Type**
list of str

### Example

proposal = 2953 v = Viking(proposal) v.X_RANGE = slice(0, 1900) v.Y_RANGE = slice(38, 80) v.ENERGY_CALIB = [1.47802667e-06, 2.30600328e-02, 5.15884589e+02] v.BL_SIGNAL_RANGE = [500, 545]

**set_params**(***params*)

**get_params**(**params*)

**from_run**(*runNB*, *add_attrs=True*)

load a run

Load the run *runNB*. A thin wrapper around *toolbox_scs.load*.

**Parameters**

- **runNB** (*int*) – the run number
- **add_attrs** (*bool*) – if True, adds the camera parameters as attributes to the dataset (see get_camera_params())
- **Output** –
- **------** –
- **ds** (*xarray Dataset*) – the dataset containing the camera images

### Example

data = v.from_run(145) # load run 145

data1 = v.from_run(145) # load run 145 data2 = v.from_run(155) # load run 155 data = xarray.concat([data1, data2], 'trainId') # combine both

**load_dark**(*runNB=None*)

**integrate**(*data*)

This function calculates the mean over the non-dispersive dimension to create a spectrum. If the camera parameters are known, the spectrum is multiplied by the number of photoelectrons per ADC count. A new variable "spectrum" is added to the data.

**get_camera_gain**(*run*)

Get the preamp gain of the camera in the Viking spectrometer for a specified run.

**Parameters**

- **run** (*extra_data DataCollection*) – information on the run
- **Output** –
- **------** –
- **gain** (*int*) –

**e_per_counts**(*run*, *gain=None*)

Conversion factor from camera digital counts to photoelectrons per count. The values can be found in the camera datasheet (Andor Newton) but they have been slightly corrected for High Sensitivity mode after analysis of runs 1204, 1207 and 1208, proposal 2937.

**Parameters**

- **run** (*extra_data DataCollection*) – information on the run

- **gain** (*int*) – the camera preamp gain
- **Output** –
- **------** –
- **ret** (*float*) – photoelectrons per count

**get_camera_params**(*run*)

**removePolyBaseline**(*data*)

Removes a polynomial baseline to a spectrum, assuming a fixed position for the signal.

**Parameters**

- **data** (*xarray Dataset*) – The Viking data containing the variable "spectrum"
- **Output** –
- **------** –
- **data** – the original dataset with the added variable "spectrum_nobl" containing the baseline subtracted spectra.

**xas**(*data*, *data_ref*, *thickness=1*, *plot=False*, *plot_errors=True*, *xas_ylim=(-1, 3)*)

Given two independent datasets (one with sample and one reference), this calculates the average XAS spectrum (absorption coefficient), associated standard deviation and standard error. The absorption coefficient is defined as -log(It/I0)/thickness.

**Parameters**

- **data** (*xarray Dataset*) – the dataset containing the spectra with sample
- **data_ref** (*xarray Dataset*) – the dataset containing the spectra without sample
- **thickness** (*float*) – the thickness used for the calculation of the absorption coefficient
- **plot** (*bool*) – If True, plot the resulting average spectra.
- **plot_errors** (*bool*) – If True, adds the 95% confidence interval on the spectra.
- **xas_ylim** (*tuple or list of float*) – the y limits for the XAS plot.
- **Output** –
- **------** –
- **xas** (*xarray Dataset*) – the dataset containing the computed XAS quantities: I0, It, absorptionCoef and their associated errors.

**calibrate**(*runList*, *plot=True*)

This routine determines the calibration coefficients to translate the camera pixels into energy in eV. The Viking spectrometer is calibrated using the beamline monochromator: runs with various monochromatized photon energy are recorded and their peak position on the detector are determined by Gaussian fitting. The energy vs. position data is then fitted to a second degree polynomial.

**Parameters**

- **runList** (*list of int*) – the list of runs containing the monochromatized spectra
- **plot** (*bool*) – if True, the spectra, their Gaussian fits and the calibration curve are plotted.
- **Output** –
- **------** –
- **energy_calib** (*np.array*) – the calibration coefficients (2nd degree polynomial)

toolbox_scs.**calibrate_xgm**(*run*, *data*, *xgm='SCS'*, *plot=False*)

Calculates the calibration factor F between the photon flux (slow signal) and the fast signal (pulse-resolved) of the sase 3 pulses. The calibrated fast signal is equal to the uncalibrated one multiplied by F.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the digitizer data.
- **data** (*xarray Dataset*) – dataset containing the pulse-resolved sase 3 signal, e.g. 'SCS_SA3'
- **xgm** (*str*) – one in {'XTD10', 'SCS'}
- **plot** (*bool*) – If True, shows a plot of the photon flux, averaged fast signal and calibrated fast signal.

**Returns**

**F** – calibration factor F defined as: calibrated XGM [microJ] = F * fast XGM array ('SCS_SA3' or 'XTD10_SA3')

**Return type**

float

### Example

```
>>> import toolbox_scs as tb
>>> import toolbox_scs.detectors as tbdet
>>> run, data = tb.load(900074, 69, ['SCS_XGM'])
>>> ds = tbdet.get_xgm(run, merge_with=data)
>>> F = tbdet.calibrate_xgm(run, ds, plot=True)
>>> # Add calibrated XGM to the dataset:
>>> ds['SCS_SA3_uJ'] = F * ds['SCS_SA3']
```

toolbox_scs.**get_xgm**(*run*, *mnemonics=None*, *merge_with=None*, *indices=slice(0, None)*)

Load and/or computes XGM data. Sources can be loaded on the fly via the mnemonics argument, or processed from an existing dataset (merge_with). The bunch pattern table is used to assign the pulse id coordinates if the number of pulses has changed during the run.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the xgm data.
- **mnemonics** (*str or list of str*) – mnemonics for XGM, e.g. "SCS_SA3" or ["XTD10_XGM", "SCS_XGM"]. If None, defaults to "SCS_SA3" in case no merge_with dataset is provided.
- **merge_with** (*xarray Dataset*) – If provided, the resulting Dataset will be merged with this one. The XGM variables of merge_with (if any) will also be computed and merged.
- **indices** (*slice, list, 1D array*) – Pulse indices of the XGM array in case bunch pattern is missing.

**Returns**

merged with Dataset *merge_with* if provided.

**Return type**

xarray Dataset with pulse-resolved XGM variables aligned,

### Example

```
>>> import toolbox_scs as tb
>>> run, ds = tb.load(2212, 213, 'SCS_SA3')
>>> ds['SCS_SA3']
```

toolbox_scs.**concatenateRuns**(*runs*)

Sorts and concatenate a list of runs with identical data variables along the trainId dimension.

**Input:**
runs: (list) the xarray Datasets to concatenate

**Output:**
a concatenated xarray Dataset

toolbox_scs.**find_run_path**(*proposalNB*, *runNB*, *data='raw'*)

Return the run path given the specified proposal and run numbers.

**Parameters**

- **proposalNB** (*(str, int)*) – proposal number e.g. 'p002252' or 2252
- **runNB** (*(str, int)*) – run number as integer
- **data** (*str*) – 'raw', 'proc' (processed) or 'all' (both 'raw' and 'proc') to access data from either or both of those folders. If 'all' is used, sources present in 'proc' overwrite those in 'raw'. The default is 'raw'.

**Returns**
**path** – The run path.

**Return type**
str

toolbox_scs.**get_array**(*run=None*, *mnemonic=None*, *stepsize=None*, *subset=None*, *data='raw'*, *proposalNB=None*, *runNB=None*)

Loads one data array for the specified mnemonic and rounds its values to integer multiples of stepsize for consistent grouping (except for stepsize=None). Returns a 1D array of ones if mnemonic is set to None.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data. Used if proposalNB and runNB are None.
- **mnemonic** (*str*) – Identifier of a single item in the mnemonic collection. None creates a dummy 1D array of ones with length equal to the number of trains.
- **stepsize** (*float*) – nominal stepsize of the array data - values will be rounded to integer multiples of this value.
- **subset** (slice or extra_data.by_index or **numpy.s_**) – a subset of train that can be loaded with extra_data.by_index[:5] for the first 5 trains. If None, all trains are retrieved.
- **data** (*str or Sequence of str*) – 'raw', 'proc' (processed), or any other location relative to the proposal path with data per run to access. May also be 'all' (both 'raw' and 'proc') or a sequence of strings to load data from several locations, with later locations overwriting sources present in earlier ones. The default is 'raw'.
- **proposalNB** (*(str, int)*) – proposal number e.g. 'p002252' or 2252.
- **runNB** (*(str, int)*) – run number e.g. 17 or 'r0017'.

**Returns**
**data** – xarray DataArray containing rounded array values using the trainId as coordinate.

**Return type**
xarray.DataArray

**Raises**
*ToolBoxValueError* – Exception: Toolbox specific exception, indicating a non-valid mnemonic entry

### Example

```
>>> import toolbox_scs as tb
>>> run = tb.open_run(2212, 235)
>>> mnemonic = 'PP800_PhaseShifter'
>>> data_PhaseShifter = tb.get_array(run, mnemonic, 0.5)
```

toolbox_scs.**load**(*proposalNB=None*, *runNB=None*, *fields=None*, *data='all'*, *display=False*, *validate=False*, *subset=None*, *rois={}*, *extract_digitizers=True*, *extract_xgm=True*, *extract_bam=True*, *bunchPattern='sase3'*, *parallelize=True*)

Load a run and extract the data. Output is an xarray with aligned trainIds.

**Parameters**

- **proposalNB** (*str, int*) – proposal number e.g. 'p002252' or 2252
- **runNB** (*str, int*) – run number as integer
- **fields** (*str, list of str, list of dict*) – list of mnemonics to load specific data such as "fastccd", "SCS_XGM", or dictionnaries defining a custom mnemonic such as {"extra": {'source': 'SCS_CDIFFT_MAG/SUPPLY/CURRENT',

  'key': 'actual_current.value', 'dim': None}}
- **data** (*str or Sequence of str*) – 'raw', 'proc' (processed), or any other location relative to the proposal path with data per run to access. May also be 'all' (both 'raw' and 'proc') or a sequence of strings to load data from several locations, with later locations overwriting sources present in earlier ones. The default is 'raw'.
- **display** (*bool*) – whether to show the run.info or not
- **validate** (*bool*) – whether to run extra-data-validate or not
- **subset** (slice or extra_data.by_index or **numpy.s_**) – a subset of train that can be loaded with extra_data.by_index[:5] for the first 5 trains. If None, all trains are retrieved.
- **rois** (*dict*) – a dictionnary of mnemonics with a list of rois definition and the desired names, for example: {'fastccd': {'ref': {'roi': by_index[730:890, 535:720],

  'dim': ['ref_x', 'ref_y']},

  **'sam': {'roi':by_index[1050:1210, 535:720],**
  'dim': ['sam_x', 'sam_y']}}}
- **extract_digitizers** (*bool*) – If True, extracts the peaks from digitizer variables and aligns the pulse Id according to the fadc_bp bunch pattern.
- **extract_xgm** (*bool*) – If True, extracts the values from XGM variables (e.g. 'SCS_SA3', 'XTD10_XGM') and aligns the pulse Id with the sase1 / sase3 bunch pattern.

- **extract_bam** (*bool*) – If True, extracts the values from BAM variables (e.g. 'BAM1932M') and aligns the pulse Id with the sase3 bunch pattern.
- **bunchPattern** (*str*) – bunch pattern used to extract the Fast ADC pulses. A string or a dict as in:

```
{'FFT_PD2': 'sase3', 'ILH_I0': 'scs_ppl'}
```

  Ignored if extract_digitizers=False.

- **parallelize** (*bool*) – from EXtra-Data: enable or disable opening files in parallel. Particularly useful if creating child processes is not allowed (e.g. in a daemonized multiprocessing.Process).

**Returns**

**run, ds** – extra_data DataCollection of the proposal and run number and an xarray Dataset with aligned trainIds and pulseIds

**Return type**

DataCollection, xarray.Dataset

### Example

```
>>> import toolbox_scs as tb
>>> run, data = tb.load(2212, 208, ['SCS_SA3', 'MCP2apd', 'nrj'])
```

toolbox_scs.**open_run**(*proposalNB*, *runNB*, *subset=None*, ***kwargs*)

Get extra_data.DataCollection in a given proposal. Wraps the extra_data open_run routine and adds subset selection, out of convenience for the toolbox user. More information can be found in the extra_data documentation.

**Parameters**

- **proposalNB** (*(str, int)*) – proposal number e.g. 'p002252' or 2252
- **runNB** (*(str, int)*) – run number e.g. 17 or 'r0017'
- **subset** (slice or extra_data.by_index or **numpy.s_**) – a subset of train that can be loaded with extra_data.by_index[:5] for the first 5 trains. If None, all trains are retrieved.
- ****kwargs** –
- **--------** –
- **data** (*str*) – default -> 'raw'
- **include** (*str*) – default -> '*'

**Returns**

**run** – DataCollection object containing information about the specified run. Data can be loaded using built-in class methods.

**Return type**

extra_data.DataCollection

toolbox_scs.**run_by_path**(*path*)

Return specified run

Wraps the extra_data RunDirectory routine, to ease its use for the scs-toolbox user.

**Parameters**

**path** (*str*) – path to the run directory

**Returns**
**run** – DataCollection object containing information about the specified run. Data can be loaded using built-in class methods.

**Return type**
extra_data.DataCollection

toolbox_scs.**load_run_values**(*prop_or_run*, *runNB=None*, *which='mnemonics'*)

Load the run value for each mnemonic whose source is a CONTORL source (see extra-data DataCollection.get_run_value() for details)

**Parameters**

- **prop_or_run** (*extra_data DataCollection or int*) – The run (DataCollection) to check for mnemonics. Alternatively, the proposal number (int), for which the runNB is also required.
- **runNB** (*int*) – The run number. Only used if the first argument is the proposal number.
- **which** (*str*) – 'mnemonics' or 'all'. If 'mnemonics', only the run values for the ToolBox mnemonics are retrieved. If 'all', a compiled dictionnary of all control sources run values is returned.
- **Output** –
- **------** –
- **run_values** (*a dictionnary containing the mnemonic or all run values.*) –

toolbox_scs.**check_data_rate**(*run*, *fields=None*)

Calculates the fraction of train ids that contain data in a run.

**Parameters**

- **run** (*extra_data DataCollection*) – the DataCollection associated to the data.
- **fields** (*str, list of str or dict*) – mnemonics to check. If None, all mnemonics in the run are checked. A custom mnemonic can be defined with a dictionnary: {'extra': {'source': 'SCS_CDIFFT_MAG/SUPPLY/CURRENT', 'key': 'actual_current.value'}}
- **Output** –
- **------** – ret: dictionnary dictionnary with mnemonic as keys and fraction of train ids that contain data as values.

toolbox_scs.**__all__**

toolbox_scs.**extractBunchPattern**(*bp_table=None*, *key='sase3'*, *runDir=None*)

generate the bunch pattern and number of pulses of a source directly from the bunch pattern table and not using the MDL device BUNCH_DECODER. This is inspired by the euxfel_bunch_pattern package, https://git.xfel.eu/gitlab/karaboDevices/euxfel_bunch_pattern Inputs:

**bp_table: DataArray corresponding to the mnemonics "bunchPatternTable".**
If None, the bunch pattern table is loaded using runDir.

key: str, ['sase1', 'sase2', 'sase3', 'scs_ppl'] runDir: extra-data DataCollection. Required only if bp_table is None.

**Outputs:**
bunchPattern: DataArray containing indices of the sase/laser pulses for each train npulses: DataArray containing the number of pulses for each train matched: 2-D DataArray mask (trainId x 2700), True where 'key' has pulses

toolbox_scs.**get_sase_pId**(*run*, *loc='sase3'*, *run_mnemonics=None*, *bpt=None*, *merge_with=None*)

Returns the pulse Ids of the specified *loc* during a run. If the number of pulses has changed during the run, it loads the bunch pattern table and extract all pulse Ids used.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}
- **run_mnemonics** (*dict*) – the mnemonics for the run (see *menonics_for_run*)
- **bpt** (*2D-array*) – The bunch pattern table. Used only if the number of pulses has changed. If None, it is loaded on the fly.
- **merge_with** (*xarray.Dataset*) – dataset that may contain the bunch pattern table to use in case the number of pulses has changed. If merge_with does not contain the bunch pattern table, it is loaded and added as a variable 'bunchPatternTable' to merge_with.

**Returns**

**pulseIds** – the pulse ids at the specified location. Returns None if the mnemonic is not in the run.

**Return type**

np.array

toolbox_scs.**npulses_has_changed**(*run*, *loc='sase3'*, *run_mnemonics=None*)

Checks if the number of pulses has changed during the run for a specific location *loc* (='sase1', 'sase3', 'scs_ppl' or 'laser') If the source is not found in the run, returns True.

**Parameters**

- **run** (*extra_data.DataCollection*) – DataCollection containing the data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}
- **run_mnemonics** (*dict*) – the mnemonics for the run (see *menonics_for_run*)

**Returns**

**ret** – True if the number of pulses has changed or the source was not found, False if the number of pulses did not change.

**Return type**

bool

toolbox_scs.**pulsePatternInfo**(*data*, *plot=False*)

display general information on the pulse patterns operated by SASE1 and SASE3. This is useful to track changes of number of pulses or mode of operation of SASE1 and SASE3. It also determines which SASE comes first in the train and the minimum separation between the two SASE sub-trains.

**Inputs:**

data: xarray Dataset containing pulse pattern info from the bunch decoder MDL: {'sase1, sase3', 'npulses_sase1', 'npulses_sase3'} plot: bool enabling/disabling the plotting of the pulse patterns

**Outputs:**

print of pulse pattern info. If plot==True, plot of the pulse pattern.

toolbox_scs.**repRate**(*data=None*, *runNB=None*, *proposalNB=None*, *key='sase3'*)

Calculates the pulse repetition rate (in kHz) in sase according to the bunch pattern and assuming a grid of 4.5 MHz.

#### Inputs:

data: xarray Dataset containing pulse pattern, needed if runNB is none runNB: int or str, run number. Needed if data is None proposal: int or str, proposal where to find the run. Needed if data is None key: str in [sase1, sase2, sase3, scs_ppl], source for which the

repetition rate is calculated

#### Output:

f: repetition rate in kHz

toolbox_scs.**is_sase_3**(*bpt*)

Check for prescence of a SASE3 pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if SASE3 pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.**is_sase_1**(*bpt*)

Check for prescence of a SASE1 pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if SASE1 pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.**is_ppl**(*bpt*)

Check for prescence of pp-laser pulse.

**Parameters**
**bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.

**Returns**
**boolean** – true if pp-laser pulse is present.

**Return type**
numpy array, xarray DataArray

toolbox_scs.**is_pulse_at**(*bpt*, *loc*)

Check for prescence of a pulse at the location provided.

**Parameters**

- **bpt** (*numpy array, xarray DataArray*) – The bunch pattern data.
- **loc** (*str*) – The location where to check: {'sase1', 'sase3', 'scs_ppl'}

**Returns**
**boolean** – true if a pulse is present at *loc*.

**Return type**
numpy array, xarray DataArray

toolbox_scs.**degToRelPower**(*x*, *theta0=0*)

converts a half-wave plate position in degrees into relative power between 0 and 1. Inputs:

x: array-like positions of half-wave plate, in degrees theta0: position for which relative power is zero

**Output:**
array-like relative power

toolbox_scs.**positionToDelay**(*pos*, *origin=0*, *invert=True*, *reflections=1*)

converts a motor position in mm into optical delay in picosecond Inputs:

pos: array-like delay stage motor position origin: motor position of time zero in mm invert: bool, inverts the sign of delay if True reflections: number of bounces in the delay stage

**Output:**
delay in picosecond

toolbox_scs.**delayToPosition**(*delay*, *origin=0*, *invert=True*, *reflections=1*)

converts an optical delay in picosecond into a motor position in mm Inputs:

delay: array-like delay in ps origin: motor position of time zero in mm invert: bool, inverts the sign of delay if True reflections: number of bounces in the delay stage

**Output:**
delay in picosecond

toolbox_scs.**fluenceCalibration**(*hwp*, *power_mW*, *npulses*, *w0x*, *w0y=None*, *train_rep_rate=10*, *fit_order=1*, *plot=True*, *xlabel='HWP [%]'*)

Given a measurement of relative powers or half wave plate angles and averaged powers in mW, this routine calculates the corresponding fluence and fits a polynomial to the data.

**Parameters**

- **hwp** (*array-like (N)*) – angle or relative power from the half wave plate
- **power_mW** (*array-like (N)*) – measured power in mW by powermeter
- **npulses** (*int*) – number of pulses per train during power measurement
- **w0x** (*float*) – radius at 1/e^2 in x-axis in meter
- **w0y** (*float, optional*) – radius at 1/e^2 in y-axis in meter. If None, w0y=w0x is assumed.
- **train_rep_rate** (*float*) – repetition rate of the FEL, by default equals to 10 Hz.
- **fit_order** (*int*) – order of the polynomial fit
- **plot** (*bool*) – Plot the results if True
- **xlabel** (*str*) – xlabel for the plot
- **Output** –
- **------** –
- **F** (*ndarray (N)*) – fluence in mJ/cm^2
- **fit_F** (*ndarray*) – coefficients of the fluence polynomial fit

- **E** (*ndarray (N)*) – pulse energy in microJ
- **fit_E** (*ndarray*) – coefficients of the fluence polynomial fit

toolbox_scs.**align_ol_to_fel_pId**(*ds*, *ol_dim='ol_pId'*, *fel_dim='sa3_pId'*, *offset=0*, *fill_value=np.nan*)

Aligns the optical laser (OL) pulse Ids to the FEL pulse Ids. The new OL coordinates are calculated as ds[ol_dim] + ds[fel_dim][0] + offset. The ol_dim is then removed, and if the number of OL and FEL pulses are different, the missing values are replaced by fill_value (NaN by default).

**Parameters**

- **ds** (*xarray.Dataset*) – Dataset containing both OL and FEL dimensions
- **ol_dim** (*str*) – name of the OL dimension
- **fel_dim** (*str*) – name of the FEL dimension
- **offset** (*int*) – offset added to the OL pulse Ids.
- **fill_value** (*(scalar or dict-like, optional)*) – Value to use for newly missing values. If a dict-like, maps variable names to fill values. Use a data array's name to refer to its values.
- **Output** –
- **------** –
- **ds** – The newly aligned dataset

toolbox_scs.**get_undulator_config**(*run*, *park_pos=62.0*, *plot=True*)

Extract the undulator cells configuration from a given run. The gap size and K factor as well as the magnetic chicane delay and photon energy of colors 1, 2 and 3 are compiled into an xarray Dataset.

Note: This function looks at run control values, it does not reflect any change of values during the run. Do not use to extract configuration when scanning the undulator.

**Parameters**

- **run** (*EXtra-Data DataCollection*) – The run containing the undulator information
- **park_pos** (*float, optional*) – The parked position of a cell (i.e. when fully opened)
- **plot** (*bool, optional*) – If True, plot the undulator cells configuration

**Returns**
**cells** – The resulting dataset of the undulator configuration

**Return type**
xarray Dataset

toolbox_scs.**mnemonics_for_run**(*prop_or_run*, *runNB=None*)

Returns the availble ToolBox mnemonics for a give extra_data DataCollection, or a given proposal + run number.

**Parameters**

- **prop_or_run** (*extra_data DataCollection or int*) – The run (DataCollection) to check for mnemonics. Alternatively, the proposal number (int), for which the runNB is also required.
- **runNB** (*int*) – The run number. Only used if the first argument is the proposal number.

**Returns**
**mnemonics** – The dictionnary of mnemonics that are available in the run.

**Return type**
dict

### Example

```
>>> import toolbox_scs as tb
>>> tb.mnemonics_for_run(2212, 213)
```

toolbox_scs.**__all__**

toolbox_scs.**xas**(*nrun*, *bins=None*, *Iokey='SCS_SA3'*, *Itkey='MCP3peaks'*, *nrjkey='nrj'*, *Iooffset=0*, *plot=False*, *fluorescence=False*)

Compute the XAS spectra from a xarray nrun.

**Inputs:**

nrun: xarray of SCS data bins: an array of bin-edges or an integer number of

desired bins or a float for the desired bin width.

Iokey: string for the Io fields, typically 'SCS_XGM' Itkey: string for the It fields, typically 'MCP3apd' nrjkey: string for the nrj fields, typically 'nrj' Iooffset: offset to apply on Io plot: boolean, displays a XAS spectrum if True fluorescence: boolean, if True, absorption is the ratio,

if False, absorption is negative log

**Outputs:**

**a dictionnary containing:**

nrj: the bin centers muA: the absorption sigmaA: standard deviation on the absorption sterrA: standard error on the absorption muIo: the mean of the Io counts: the number of events in each bin

toolbox_scs.**xasxmcd**(*dataP*, *dataN*)

Compute XAS and XMCD from data with both magnetic field direction Inputs:

dataP: structured array for positive field dataN: structured array for negative field

**Outputs:**

xas: structured array for the sum xmcd: structured array for the difference

**class** toolbox_scs.**parameters**(*proposal*, *darkrun*, *run*, *module*, *gain*, *drop_intra_darks=True*)

Parameters contains all input parameters for the BOZ corrections.

This is used in beam splitting off-axis zone plate spectrocopy analysis as well as the during the determination of correction parameters themselves to ensure they can be reproduced.

### Inputs

proposal: int, proposal number darkrun: int, run number for the dark run run: int, run number for the data run module: int, DSSC module number gain: float, number of ph per bin drop_intra_darks: drop every second DSSC frame

**dask_load_persistently**(*dark_data_size_Gb=None*, *data_size_Gb=None*)

Load dask data array in memory.

#### Inputs

**dark_data_size_Gb: float, optional size of dark to load in memory,**
in Gb

data_size_Gb: float, optional size of data to load in memory, in Gb

`use_gpu`()

`set_mask`(*arr*)

Set mask of bad pixels.

#### Inputs

**arr: either a boolean array of a DSSC module image or a list of bad**
pixel indices

`get_mask`()

Get the boolean array bad pixel of a DSSC module.

`get_mask_idx`()

Get the list of bad pixel indices.

`flat_field_guess`(*guess=None*)

Set the flat-field guess parameter for the fit and returns it.

#### Inputs

**guess: a list of 8 floats, the 4 first to define the plane**
ax+by+cz+d=0 for 'n' beam and the 4 last for the 'p' beam in case mirror symmetry is disbaled

`set_flat_field`(*ff_params*, *ff_type='plane'*, *prod_th=None*, *ratio_th=None*)

Set the flat-field plane definition.

#### Inputs

ff_params: list of parameters ff_type: string identifying the type of flat field normalization,

default is 'plane'.

`get_flat_field`()

Get the flat-field plane definition.

`set_Fnl`(*Fnl*)

Set the non-linear correction function.

`get_Fnl`()

Get the non-linear correction function.

`save`(*path='./'*)

Save the parameters as a JSON file.

#### Inputs

path: str, where to save the file, default to './'

**classmethod load**(*fname*)

Load parameters from a JSON file.

#### Inputs

fname: string, name a the JSON file to load

**__str__**()

Return str(self).

toolbox_scs.**get_roi_pixel_pos**(*roi*, *params*)

Compute fake or real pixel position of an roi from roi center.

#### Inputs:

roi: dictionnary params: parameters

#### Returns:

X, Y: 1-d array of pixel position.

toolbox_scs.**bad_pixel_map**(*params*)

Compute the bad pixels map.

#### Inputs

params: parameters

**rtype**
bad pixel map

toolbox_scs.**inspect_dark**(*arr*, *mean_th=(None, None)*, *std_th=(None, None)*)

Inspect dark run data and plot diagnostic.

#### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) mean_th: tuple of threshold (low, high), default (None, None), to compute

a mask of good pixels for which the mean dark value lie inside this range

**std_th: tuple of threshold (low, high), default (None, None), to compute a**
mask of bad pixels for which the dark std value lie inside this range

**returns**
**fig**

**rtype**
matplotlib figure

toolbox_scs.**histogram_module**(*arr*, *mask=None*)

Compute a histogram of the 9 bits raw pixel values over a module.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) mask: optional bad pixel mask

**rtype**
histogram

toolbox_scs.**inspect_histogram**(*arr*, *arr_dark=None*, *mask=None*, *extra_lines=False*)

Compute and plot a histogram of the 9 bits raw pixel values.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) arr: dask array of reshaped dssc dark data (trainId, pulseId, x, y) mask: optional bad pixel mask extra_lines: boolean, default False, plot extra lines at period values

**returns**

- **(h, hd)** (*histogram of arr, arr_dark*)
- *figure*

toolbox_scs.**find_rois**(*data_mean*, *threshold*, *extended=False*)

Find rois from 3 beams configuration.

### Inputs

data_mean: dark corrected average image threshold: threshold value to find beams extended: boolean, True to define additional ASICS based rois

**returns**
**rois**

**rtype**
dictionnary of rois

toolbox_scs.**find_rois_from_params**(*params*)

Find rois from 3 beams configuration.

### Inputs

params: parameters

**returns**
**rois**

**rtype**
dictionnary of rois

toolbox_scs.**inspect_rois**(*data_mean*, *rois*, *threshold=None*, *allrois=False*)

Find rois from 3 beams configuration from mean module image.

### Inputs

data_mean: mean module image threshold: float, default None, threshold value used to detect beams

boundaries

**allrois: boolean, default False, plot all rois defined in rois or only the**
main ones (['n', '0', 'p'])

**rtype**
matplotlib figure

toolbox_scs.**compute_flat_field_correction**(*rois*, *params*, *plot=False*)

toolbox_scs.**inspect_flat_field_domain**(*avg*, *rois*, *prod_th*, *ratio_th*, *vmin=None*, *vmax=None*)

Extract beams roi from average image and compute the ratio.

### Inputs

**avg: module average image with no saturated shots for the flat-field**
determination

rois: dictionnary or ROIs prod_th, ratio_th: tuple of floats for low and high threshold on

product and ratio

vmin: imshow vmin level, default None will use 5 percentile value vmax: imshow vmax level, default None will use 99.8 percentile value

**returns**

- **fig** (*matplotlib figure plotted*)
- **domain** (*a tuple (n_m, p_m) of domain for the 'n' and 'p' order*)

toolbox_scs.**inspect_plane_fitting**(*avg*, *rois*, *domain=None*, *vmin=None*, *vmax=None*)

toolbox_scs.**plane_fitting_domain**(*avg*, *rois*, *prod_th*, *ratio_th*)

Extract beams roi, compute their ratio and the domain.

### Inputs

**avg: module average image with no saturated shots for the flat-field**
determination

**rois: dictionnary or rois containing the 3 beams ['n', '0', 'p'] with '0'**
as the reference beam in the middle

**prod_th: float tuple, low and hight threshold level to determine the plane**
fitting domain on the product image of the orders

**ratio_th: float tuple, low and high threshold level to determine the plane**
fitting domain on the ratio image of the orders

**returns**

- **n** (*img ratio 'n'/'0'*)
- **n_m** (*mask where the the product 'n''*0' is higher than 5 indicting that the*) – img ratio 'n'/'0' is defined
- **p** (*img ratio 'p'/'0'*)
- **p_m** (*mask where the the product 'p''*0' is higher than 5 indicting that the*) – img ratio 'p'/'0' is defined

toolbox_scs.**plane_fitting**(*params*)

Fit the plane flat-field normalization.

### Inputs

params: parameters

**returns**
**res** – defines the plane as a*x + b*y + c*z + d = 0

**rtype**
the minimization result. The fitted vector res.x = [a, b, c, d]

toolbox_scs.**ff_refine_crit**(*p*, *alpha*, *params*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *sat_level=511*)

Criteria for the ff_refine_fit.

### Inputs

p: ff plane params: parameters arr_dark: dark data arr: data tid: train id of arr data rois: ['n', '0', 'p', 'sat'] rois mask: mask fo good pixels sat_level: integer, default 511, at which level pixel begin to saturate

**rtype**
sum of standard deviation on binned 0th order intensity

toolbox_scs.**ff_refine_fit**(*params*, *crit=ff_refine_crit*)

Refine the flat-field fit by minimizing data spread.

### Inputs

params: parameters

**returns**

- **res** (*scipy minimize result. res.x is the optimized parameters*)
- **fitrres** (*iteration index arrays of criteria results for*) – [alpha=0, alpha, alpha=1]

toolbox_scs.**nl_domain**(*N*, *low*, *high*)

Create the input domain where the non-linear correction defined.

### Inputs

N: integer, number of control points or intervals low: input values below or equal to low will not be corrected high: input values higher or equal to high will not be corrected

**rtype**
array of 2**9 integer values with N segments

toolbox_scs.**nl_lut**(*domain*, *dy*)

Compute the non-linear correction.

### Inputs

**domain: input domain where dy is defined. For zero no correction is**
defined. For non-zero value x, dy[x] is applied.

**dy: a vector of deviation from linearity on control point homogeneously**
dispersed over 9 bits.

**returns**
**F_INL** – lookup table with 9 bits integer input

**rtype**
default None, non linear correction function given as a

toolbox_scs.**nl_crit**(*p*, *domain*, *alpha*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Criteria for the non linear correction.

### Inputs

p: vector of dy non linear correction domain: domain over which the non linear correction is defined alpha: float, coefficient scaling the cost of the correction function

in the criterion

arr_dark: dark data arr: data tid: train id of arr data rois: [‘n’, ‘0’, ‘p’, ‘sat’] rois mask: mask fo good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**returns**

- *(1.0 - alpha)*err1 + alpha*err2, where err1 is the 1e8 times the mean of*
- *error squared from a transmission of 1.0 and err2 is the sum of the square*
- *of the deviation from the ideal detector response.*

toolbox_scs.**nl_crit_sk**(*p*, *domain*, *alpha*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Non linear correction criteria, combining ‘n’ and ‘p’ as reference.

### Inputs

p: vector of dy non linear correction domain: domain over which the non linear correction is defined alpha: float, coefficient scaling the cost of the correction function

in the criterion

arr_dark: dark data arr: data tid: train id of arr data rois: ['n', '0', 'p', 'sat'] rois mask: mask fo good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**returns**

- *(1.0 - alpha)*err1 + alpha*err2, where err1 is the 1e8 times the mean of*
- *error squared from a transmission of 1.0 and err2 is the sum of the square*
- *of the deviation from the ideal detector response.*

toolbox_scs.**nl_fit**(*params*, *domain*, *ff=None*, *crit=None*)

Fit non linearities correction function.

### Inputs

params: parameters domain: array of index ff: array, flat field correction crit: function, criteria function

**returns**

- **res** (*scipy minimize result. res.x is the optimized parameters*)
- **fitrres** (*iteration index arrays of criteria results for*) – [alpha=0, alpha, alpha=1]

toolbox_scs.**inspect_nl_fit**(*res_fit*)

Plot the progress of the fit.

### Inputs

res_fit:

**rtype**
matplotlib figure

toolbox_scs.**snr**(*sig*, *ref*, *methods=None*, *verbose=False*)

Compute mean, std and SNR from transmitted and I0 signals.

### Inputs

sig: 1D signal samples ref: 1D reference samples methods: None by default or list of strings to select which methods to use.

Possible values are 'direct', 'weighted', 'diff'. In case of None, all methods will be calculated.

verbose: booleand, if True prints calculated values

**returns**

- *dictionnary of [methods][value] where value is 'mu' for mean and 's' for*
- *standard deviation.*

toolbox_scs.**inspect_Fnl**(*Fnl*)

Plot the correction function Fnl.

### Inputs

Fnl: non linear correction function lookup table

**rtype**
matplotlib figure

toolbox_scs.**inspect_correction**(*params*, *gain=None*)

Comparison plot of the different corrections.

### Inputs

params: parameters gain: float, default None, DSSC gain in ph/bin

**rtype**
matplotlib figure

toolbox_scs.**inspect_correction_sk**(*params*, *ff*, *gain=None*)

Comparison plot of the different corrections, combining 'n' and 'p'.

### Inputs

params: parameters gain: float, default None, DSSC gain in ph/bin

**rtype**
matplotlib figure

toolbox_scs.**load_dssc_module**(*proposalNB*, *runNB*, *moduleNB=15*, *subset=slice(None)*, *drop_intra_darks=True*, *persist=False*, *data_size_Gb=None*)

Load single module dssc data as dask array.

### Inputs

proposalNB: proposal number runNB: run number moduleNB: default 15, module number subset: default slice(None), subset of trains to load drop_intra_darks: boolean, default True, remove intra darks from the data persist: default False, load all data persistently in memory data_size_Gb: float, if persist is True, can optionaly restrict

the amount of data loaded for dark data and run data in Gb

**returns**

- **arr** (*dask array of reshaped dssc data (trainId, pulseId, x, y)*)
- **tid** (*array of train id number*)

toolbox_scs.**average_module**(*arr*, *dark=None*, *ret='mean'*, *mask=None*, *sat_roi=None*, *sat_level=300*, *F_INL=None*)

Compute the average or std over a module.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) dark: default None, dark to be substracted ret: string, either 'mean' to compute the mean or 'std' to compute the

standard deviation

mask: default None, mask of bad pixels to ignore sat_roi: roi over which to check for pixel with values larger than

sat_level to drop the image from the average or std

sat_level: int, minimum pixel value for a pixel to be considered saturated F_INL: default None, non linear correction function given as a

lookup table with 9 bits integer input

**rtype**
average or standard deviation image

toolbox_scs.**process_module**(*arr*, *tid*, *dark*, *rois*, *mask=None*, *sat_level=511*, *flat_field=None*, *F_INL=None*, *use_gpu=False*)

Process one module and extract roi intensity.

### Inputs

arr: dask array of reshaped dssc data (trainId, pulseId, x, y) tid: array of train id number dark: pulse resolved dark image to remove rois: dictionnary of rois mask: default None, mask of ignored pixels sat_level: integer, default 511, at which level pixel begin to saturate flat_field: default None, flat-field correction F_INL: default None, non-linear correction function given as a

lookup table with 9 bits integer input

**rtype**
dataset of extracted pulse and train resolved roi intensities.

toolbox_scs.**process**(*Fmodel*, *arr_dark*, *arr*, *tid*, *rois*, *mask*, *flat_field*, *sat_level=511*, *use_gpu=False*)

Process dark and run data with corrections.

### Inputs

Fmodel: correction lookup table arr_dark: dark data arr: data rois: ['n', '0', 'p', 'sat'] rois mask: mask of good pixels flat_field: zone plate flat-field correction sat_level: integer, default 511, at which level pixel begin to saturate

**rtype**
roi extracted intensities

toolbox_scs.**inspect_saturation**(*data*, *gain*, *Nbins=200*)

Plot roi integrated histogram of the data with saturation

### Inputs

data: xarray of roi integrated DSSC data gain: nominal DSSC gain in ph/bin Nbins: number of bins for the histogram, by default 200

**returns**

- **f** (*handle to the matplotlib figure*)
- **h** (*xarray of the histogram data*)

toolbox_scs.**reflectivity**(*data*, *Iokey='FastADC5peaks'*, *Irkey='FastADC3peaks'*, *delaykey='PP800_DelayLine'*, *binWidth=0.05*, *positionToDelay=True*, *origin=None*, *invert=False*, *pumpedOnly=False*, *alternateTrains=False*, *pumpOnEven=True*, *Ioweights=False*, *plot=True*, *plotErrors=True*, *units='mm'*)

Computes the reflectivity R = 100*(Ir/Io[pumped] / Ir/Io[unpumped] - 1) as a function of delay. Delay can be a motor position in mm or an optical delay in ps, with possibility to convert from position to delay. The default scheme is alternating pulses pumped/unpumped/... in each train, also possible are alternating trains and pumped only. If fitting is enabled, attempts a double exponential (default) or step function fit.

**Parameters**

- **data** (*xarray Dataset*) – Dataset containing the Io, Ir and delay data
- **Iokey** (*str*) – Name of the Io variable
- **Irkey** (*str*) – Name of the Ir variable
- **delaykey** (*str*) – Name of the delay variable (motor position in mm or optical delay in ps)
- **binWidth** (*float*) – width of bin in units of delay variable
- **positionToDelay** (*bool*) – If True, adds a time axis converted from position axis according to origin and invert parameters. Ignored if origin is None.
- **origin** (*float*) – Position of time overlap, shown as a vertical line. Used if positionToDelay is True to convert position to time axis.
- **invert** (*bool*) – Used if positionToDelay is True to convert position to time axis.
- **pumpedOnly** (*bool*) – Assumes that all trains and pulses are pumped. In this case, Delta R is defined as Ir/Io.
- **alternateTrains** (*bool*) – If True, assumes that trains alternate between pumped and unpumped data.
- **pumpOnEven** (*bool*) – Only used if alternateTrains=True. If True, even trains are pumped, if False, odd trains are pumped.
- **Ioweights** (*bool*) – If True, computes the ratio of the means instead of the mean of the ratios Irkey/Iokey. Useful when dealing with large intensity variations.
- **plot** (*bool*) – If True, plots the results.
- **plotErrors** (*bool*) – If True, plots the 95% confidence interval.
- **Output** –
- **------** – xarray Dataset containing the binned Delta R, standard deviation, standard error, counts and delays, and the fitting results if full is True.

toolbox_scs.**knife_edge**(*ds*, *axisKey='scannerX'*, *signalKey='FastADC4peaks'*, *axisRange=None*, *p0=None*, *full=False*, *plot=False*, *display=False*)

Calculates the beam radius at 1/e^2 from a knife-edge scan by fitting with erfc function: f(x, x0, w0, a, b) = a*erfc(np.sqrt(2)*(x-x0)/w0) + b with w0 the beam radius at 1/e^2 and x0 the beam center.

**Parameters**

- **ds** (*xarray Dataset*) – dataset containing the detector signal and the motor position.
- **axisKey** (*str*) – key of the axis against which the knife-edge is performed.
- **signalKey** (*str*) – key of the detector signal.
- **axisRange** (*list of floats*) – edges of the scanning axis between which to apply the fit.
- **p0** (*list of floats, numpy 1D array*) – initial parameters used for the fit: x0, w0, a, b. If None, a beam radius of 100 micrometers is assumed.
- **full** (*bool*) – If False, returns the beam radius and standard error. If True, returns the popt, pcov list of parameters and covariance matrix from scipy.optimize.curve_fit.
- **plot** (*bool*) – If True, plots the data and the result of the fit. Default is False.
- **display** (*bool*) – If True, displays info on the fit. True when plot is True, default is False.

**Returns**

error from the fit in mm. If full is True, returns parameters and covariance matrix from scipy.optimize.curve_fit function.

**Return type**

If full is False, tuple with beam radius at 1/e^2 in mm and standard

toolbox_scs.**__all__**

For reasons of readability, contributions preferrably comply with the PEP8 code structure guidelines.

The associated code checker, called 'flake8', can be installed via PyPi.

# PYTHON MODULE INDEX

## t

# INDEX

## Symbols

## A



## B

## C



## D

## E



## F

## G

## H



## I

## K



## L



## M

## N



## O



## P

## Q



## R



## S

## T

## U



## V



## X



## Y