py4dstem

Release 0.14.14

Ben Savitsky & Alex Rakowski

Apr 04, 2024

CONTENTS

1 Contents **3**

1.1 What is 4D-STEM? . . . 3
1.2 Installation . . . 3
1.3 Examples . . . 13
1.4 API . . . 16
1.5 API Index . . . 253
1.6 Graphical User Interface . . . 253
1.7 Support & Contributions . . . 253
1.8 License . . . 254
1.9 Acknowledgements . . . 267

2 Indices and tables **269**

Python Module Index **271**

Index **273**

py4DSTEM is an open source set of python tools for processing and analysis of *four-dimensional scanning transmission electron microscopy (4D-STEM)* data.

CHAPTER

ONE

CONTENTS

1.1 What is 4D-STEM?

Scanning **T**ransmission **E**lectron **M**icropscopy (STEM) is a powerful tool for materials characterization. In a traditional STEM experiment, a beam of high energy electrons is focused to a very fine probe - on the order of, or even smaller than, the spacing between atoms - and rastered across the surface of the sample. A conventional two-dimensional STEM image is formed by populating the value of each pixel with the electron flux through a detector at the corresponding beam position. In a high resolution tool, this enables imaging at the level of atoms.

Four-dimensional scanning transmission electron microscopy (4D-STEM) uses a fast, pixelated electron detector to collect far more information than a traditional STEM experiment. In 4D-STEM, a pixelated detector is used to record a 2D diffraction image at every raster position of the beam. A 4D-STEM scan thus results in a 4D data array: two dimensions in diffraction space (i.e. the detector pixels), and two dimensions in real space (i.e. the rastering of the beam).

4D-STEM data is information rich. A 4D datacube can be collapsed in real space to yield information comparable to nanobeam electron diffraction experiment, or in diffraction space to yield a variety of virtual images, corresponding to both traditional STEM imaging modes as well as more exotic virtual imaging modalities. The structure, symmetries, and spacings of Bragg disks can be used to extract spatially resolved maps of crystallinity, grain orientations, and lattice strain. Redundant information in overlapping Bragg disks can be leveraged to calculate the sample potential. Structure in the diffracted halos of amorphous systems can be used to describe the short and medium range order.

py4DSTEM supports many different modes of 4D-STEM analysis.

1.2 Installation

Table of Contents

- *Installation*
  - *Setting up Python*
  - *Recommended Installation*
    - *Anaconda*
      - *Windows*
      - *Linux*
      - *Mac (Intel)*
      - *Mac (Apple Silicon M1/M2)*

- *Advanced Installation*
  - *Installing optional dependencies:*
  - *Anaconda*
    - *Windows*
    - *Linux*
    - *Mac (Intel)*
    - *Mac (Apple Silicon M1/M2)*
  - *Pip*
    - *Windows*
    - *Linux*
    - *Mac (Intel)*
    - *Mac (Apple Silicon M1/M2)*
  - *Installing from Source*
  - *Docker*
    - *Overview*
    - *Installation*
- *Troubleshooting*
- *Virtual Environments*

1.2.1 Setting up Python

The recommended installation for py4DSTEM uses the Anaconda Python distribution. Alternatives such as Miniconda, Mamba, pip virtualenv, and poetry will work, but here we assume the use of Anaconda. See *Virtual Environments*, for more details. The instructions to download and install Anaconda can be found here.

1.2.2 Recommended Installation

There are three ways to install py4DSTEM:

1. Anaconda (miniconda / mamba)
2. Pip
3. Installing from Source

The easiest way to install py4DSTEM is to use the pre packaged anaconda version. This is an overview of what the installation process looks like, for OS specific instructions see below.

Anaconda

Windows

Listing 1: Windows base install

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
conda install -c conda-forge pywin32
# optional but recomended
conda install jupyterlab pymatgen
```

Linux

Listing 2: Linux base install

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
# optional but recomended
conda install jupyterlab pymatgen
```

Mac (Intel)

Listing 3: Intel Mac base install

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
# optional but recomended
conda install jupyterlab pymatgen
```

Mac (Apple Silicon M1/M2)

Listing 4: Apple Silicon Mac base install

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install pyqt hdf5
conda install -c conda-forge py4dstem
# optional but recomended
conda install jupyterlab pymatgen
```

1.2.3 Advanced Installation

Installing optional dependencies:

Some of the features and modules require extra dependencies which can easily be installed using either Anaconda or Pip.

Anaconda

Windows

Listing 5: Windows Anaconda install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem pymatgen
conda install -c conda-forge pywin32
```

Running py4DSTEM code with GPU acceleration requires an NVIDIA GPU (AMD has beta support but hasn't been tested) and Nvidia Drivers installed on the system.

Listing 6: Windows Anaconda install GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem cupy cudatoolkit
conda install -c conda-forge pywin32
```

If you are looking to run the ML-AI features you are required to install tensorflow, this can be done with CPU only and GPU support.

Listing 7: Windows Anaconda install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
pip install tensorflow==2.4.1 tensorflow-addons<=0.14 crystal4D
conda install -c conda-forge pywin32
```

Listing 8: Windows Anaconda install ML-AI GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
conda install -c conda-forge cupy cudatoolkit=11.0
pip install tensorflow==2.4.1 tensorflow-addons<=0.14 crystal4D
conda install -c conda-forge pywin32
```

Linux

Listing 9: Linux Anaconda install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem pymatgen
```

Running py4DSTEM code with GPU acceleration requires an NVIDIA GPU (AMD has beta support but hasn't been tested) and Nvidia Drivers installed on the system.

Listing 10: Linux Anaconda install GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem cupy cudatoolkit
```

If you are looking to run the ML-AI features you are required to install tensorflow, this can be done with CPU only and GPU support.

Listing 11: Linux Anaconda install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
pip install tensorflow==2.4.1 tensorflow-addons<=0.14 crystal4D
```

Listing 12: Linux Anaconda install ML-AI GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
conda install -c conda-forge cupy cudatoolkit=11.0
pip install tensorflow==2.4.1 tensorflow-addons<=0.14 crystal4D
```

Mac (Intel)

Listing 13: Intel Mac Anaconda install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem pymatgen
```

Tensorflow does not support AMD GPUs so while ML-AI features can be run on an Intel Mac they are not GPU accelerated

Listing 14: Intel Mac Anaconda install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
pip install tensorflow==2.4.1 tensorflow-addons<=0.14 crystal4D
```

Mac (Apple Silicon M1/M2)

Listing 15: Apple Silicon Mac Anaconda install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem pymatgen
```

Tensorflow's support of Apple silicon GPUs is limited, and while there are steps that should enable GPU acceleration they have not been tested, but CPU only has been tested.

Listing 16: Apple Silicon Mac Anaconda install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge py4dstem
pip install tensorflow==2.4.1 tensorflow-addons<=0.14 crystal4D
```

Attention: GPU Accelerated Tensorflow on Apple Silicon

This is an untested install method and it may not work. If you try and face issues please post an issue on github.

Listing 17: Apple Silicon Mac Anaconda install ML-AI GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c apple tensorflow-deps
pip install tensorflow-macos==2.5.0 tensorflow-addons<=0.14 crystal4D tensorflow-metal
conda install -c conda-forge py4dstem
```

Pip

Windows

Listing 18: Windows pip install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[acom]
conda install -c conda-forge pywin32
```

Running py4DSTEM code with GPU acceleration requires an NVIDIA GPU (AMD has beta support but hasn't been tested) and Nvidia Drivers installed on the system.

Listing 19: Windows pip install GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[cuda]
conda install -c conda-forge pywin32
```

If you are looking to run the ML-AI features you are required to install tensorflow, this can be done with CPU only and GPU support.

Listing 20: Windows pip install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[aiml]
conda install -c conda-forge pywin32
```

Listing 21: Windows pip install ML-AI GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge cudatoolkit=11.0
pip install py4dstem[aiml-cuda]
conda install -c conda-forge pywin32
```

Linux

Listing 22: Linux pip install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[acom]
```

Running py4DSTEM code with GPU acceleration requires an NVIDIA GPU (AMD has beta support but hasn't been tested) and Nvidia Drivers installed on the system.

Listing 23: Linux pip install GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[cuda]
```

If you are looking to run the ML-AI features you are required to install tensorflow, this can be done with CPU only and GPU support.

Listing 24: Linux pip install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[aiml]
```

Listing 25: Linux pip install ML-AI GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c conda-forge cudatoolkit=11.0
pip install py4dstem[aiml-cuda]
```

Mac (Intel)

Listing 26: Intel Mac pip install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[acom]
```

Tensorflow does not support AMD GPUs so while ML-AI features can be run on an Intel Mac they are not GPU accelerated

Listing 27: Intel Mac pip install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[aiml]
```

Mac (Apple Silicon M1/M2)

Listing 28: Apple Silicon Mac pip install ACOM

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[acom]
conda install -c conda-forge py4dstem pymatgen
```

Tensorflow's support of Apple silicon GPUs is limited, and while there are steps that should enable GPU acceleration they have not been tested, but CPU only has been tested.

Listing 29: Apple Silicon Mac Anaconda install ML-AI CPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
pip install py4dstem[aiml]
```

Attention: GPU Accelerated Tensorflow on Apple Silicon

This is an untested install method and it may not work. If you try and face issues please post an issue on github.

Listing 30: Apple Silicon Mac Anaconda install ML-AI GPU

```
conda create -n py4dstem python=3.9
conda activate py4dstem
conda install -c apple tensorflow-deps
pip install tensorflow-macos==2.5.0 tensorflow-addons<=0.14 crystal4D tensorflow-metal␣
↪py4dstem
```

Installing from Source

To checkout the latest bleeding edge features, or contriubte your own features you'll need to install py4DSTEM from source. Luckily this is easy and can be done by simply running:

```
git clone
git checkout <branch> # e.g. git checkout dev
pip install -e .
```

Alternatively, you can try single step method:

```
pip install git+https://github.com/py4DSTEM/py4DSTEM.git@dev # install the dev branch
```

Docker

Overview

> "Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production." c.f. Docker website

Installation

There are py4DSTEM Docker images available on dockerhub, which can be pulled and run or built upon. Checkout the dockerhub repository to see all the versions aviale or simply run the below to get the latest version. While Docker is extremely powerful and aims to greatly simplify depolying software, it is also a complex and nuanced topic. If you are interested in using it, and are having troubles getting it to work please file an issue on the github. To use Docker you'll first need to install Docker. After which you can run the images with the following commands.

```
docker pull arakowsk/py4dstem:latest
docker run <Docker options> py4dstem:latest <commands> <args>
```

Alternatively, you can use Docker Desktop which is a GUI interface for Docker and may be an easier method for running the images for less experienced users.

1.2.4 Troubleshooting

If you face any issues, see the common errors below, and if there's no solution please file an issue on the git repository.

Some common errors:

- make sure you've activated the right environment
- when installing subsections sometimes the quotation marks can be tricky dpeending on os, terminal etc.
- GPU drivers - tricky to explain

1.2.5 Virtual Environments

Attention: Virtual environments

A Python virtual environment is its own siloed version of Python, with its own set of packages and modules, kept separate from any other Python installations on your system. In the instructions above, we created a virtual environment to make sure packages that have different dependencies don't conflict with one another. For instance, as of this writing, some of the scientific Python packages don't work well with Python 3.9 - but you might have some other applications on your computer that *need* Python 3.9. Using virtual environments solves this problem. In this example, we're creating and navigating virtual environments using Anaconda.

Because these directions install py4DSTEM to its own virtual environment, each time you want to use py4DSTEM, you'll need to activate this environment.

- In the command line, you can do this with `conda activate py4dstem`.

- In the Anaconda Navigator, you can do this by clicking on the Environments tab and then clicking on `py4dstem`.

1.3 Examples

1.3.1 First Steps

Once py4DSTEM has been succsesfully installed, you can start using it in Python the usual way. The most popular way is using Jupyter Notebooks, but py4DSTEM can be run in python scripts, iPython, spyder, etc.

Listing 31: Your first py4DSTEM script

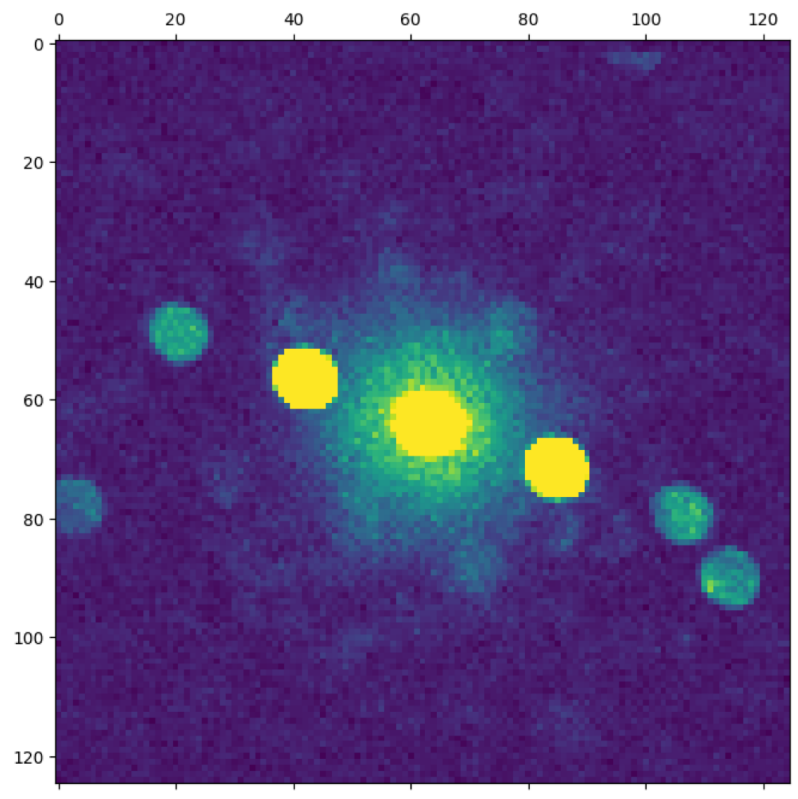
```
# Import the needed packages
import py4DSTEM

# This line displays the current version of py4DSTEM:
py4DSTEM.__version__

# download the dataset
py4DSTEM.io.download_file_from_google_drive(
    '1PmbCYosA1eYydWmmZebvf6uon9k_5g_S',
    'simulatedAuNanoplatelet_binned.h5'
)
file_data = "simulatedAuNanoplatelet_binned.h5"

# Load the data
datacube = py4DSTEM.io.read(
    file_data,
    data_id = 'polyAu_4DSTEM'     # The file above has several blocks of data inside
)

# plot a diffraction pattern
py4DSTEM.show(
    datacube[10,30],
    intensity_range='absolute',
    vmin=20,
    vmax=200,
    cmap='viridis',
)
```

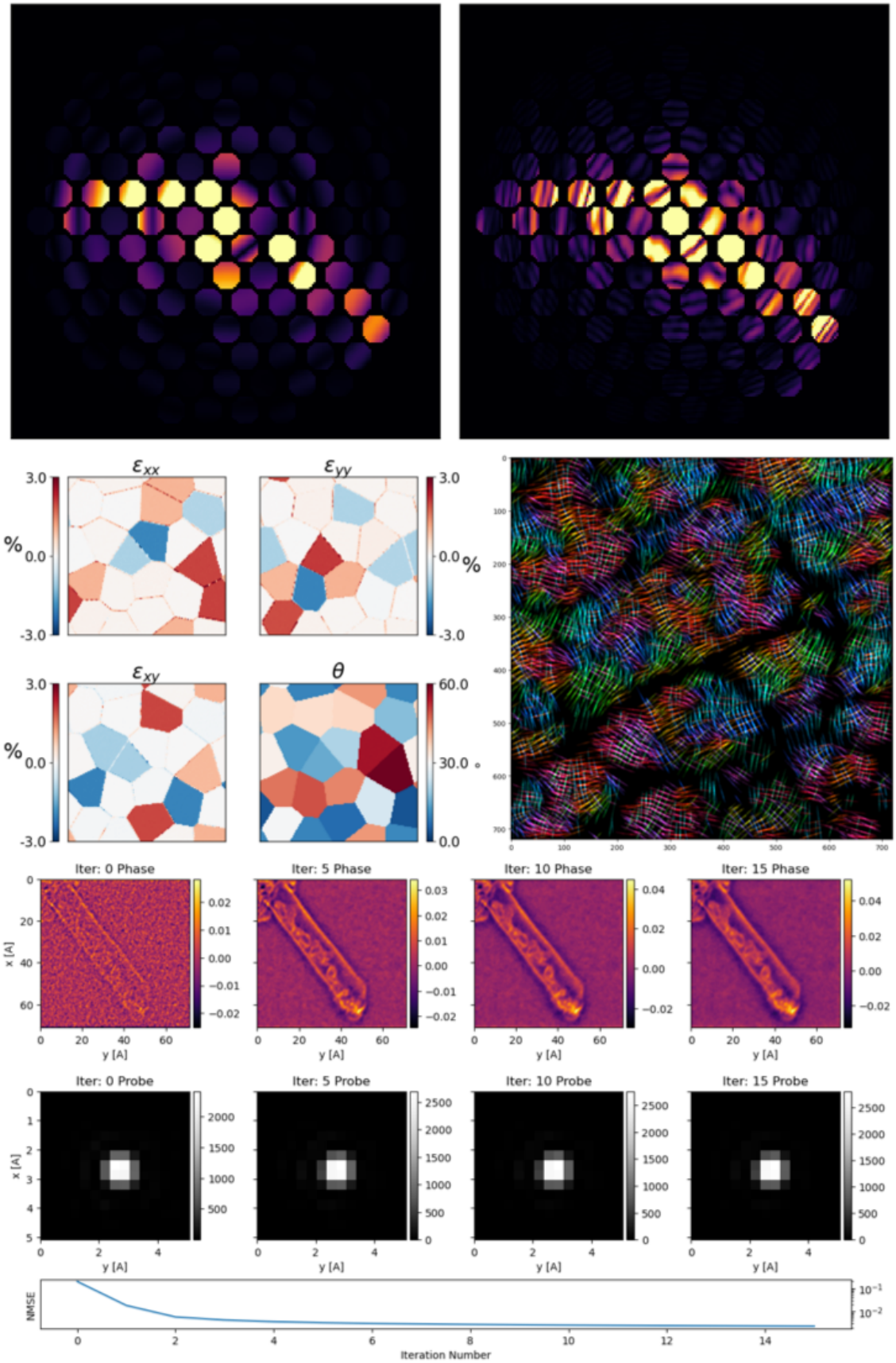


Congratulations you've just plotted your first diffraction pattern.

If you run into trouble, refer back to the installation instructions *Installation*. Remember to make sure you've activated the right *Python environment*.

1.3.2 Next Steps

For a more extensive overview checkout the tutorial github repository to see example notebooks demonstraing the features of py4DSTEM. These can be downloaded and run locally or run through the browser using binder. Here are some example plots from different anaylses you'll learn running the tutorials.


ε_{xx}
ε_{yy}
ε_{xy}
θ
%
°
Iter: 0 Phase
Iter: 5 Phase
Iter: 10 Phase
Iter: 15 Phase
Iter: 0 Probe
Iter: 5 Probe
Iter: 10 Probe
Iter: 15 Probe
x [A]
y [A]
NMSE
Iteration Number

1.4 API

For a full index of py4DSTEM functions and classes check out *API Index*

1.4.1 py4DSTEM

There are some shortcuts available for regularly used functions and utilities

Table of Contents

- *py4DSTEM*
  - *IO*
  - *Plotting*
  - *Utilities*

IO

py4DSTEM.**read**(*filepath: str | Path*, *datapath: str | None = None*, *tree: bool | str | None = True*, *verbose: bool | None = False*, ***kwargs*)

A file reader for native py4DSTEM / EMD files. To read non-native formats, use *py4DSTEM.import_file*.

For files written by py4DSTEM version 0.14+, the function arguments are those listed here - filepath, datapath, and tree. See below for descriptions.

Files written by py4DSTEM v0.14+ are EMD 1.0 files, an HDF5 based format. For a description and complete file specification, see https://emdatasets.com/format/. For the Python implementation of EMD 1.0 read-write routines which py4DSTEM is build on top of, see https://github.com/py4dstem/emdfile.

To read file written by older verions of py4DSTEM, different keyword arguments should be passed. See the docstring for *py4DSTEM.io.native.legacy.read_py4DSTEM_legacy* for a complete list. For example, *data_id* may need to be specified to select dataset.

Parameters

- **filepath** (*str or Path*) – the file path
- **datapath** (*str or None*) – the path within the H5 file to the data group to read from. If there is a single EMD data tree in the file, *datapath* may be left as None, and the path will be set to the root node of that tree. If *datapath* is None and there are multiple EMD trees, this function will issue a warning a return a list of paths to the root nodes of all EMD trees it finds. Otherwise, should be a '/' delimited path to the data node of interest, for example passing 'rootnode/somedata/someotherdata' will set the node called 'someotherdata' as the point to read from. To print the tree of data nodes present in a file to the screen, use *py4DSTEM.print_h5_tree(filepath)*.
- **tree** (*True or False or 'noroot'*) – indicates what data should be loaded, relative to the target data group specified with *datapath*. Enables reading the target data node only if *tree* is False, reading the target node as well as recursively reading the tree of data underneath it if

tree is True, or recursively reading the tree of data underneath the target node but excluding the target node itself if *tree* is to 'noroot'.

Returns
(the data)

py4DSTEM.**import_file**(*filepath: str | Path*, *mem: str | None = 'RAM'*, *binfactor: int | None = 1*, *filetype: str | None = None*, ***kwargs*)

Reader for non-native file formats. Parses the filetype, and calls the appropriate reader. Supports Gatan DM3/4, some EMPAD file versions, Gatan K2 bin/gtg, and mib formats.

Parameters

- **filepath** (*str or Path*) – Path to the file.
- **mem** (*str*) – Must be "RAM" or "MEMMAP". Specifies how the data is loaded; "RAM" transfer the data from storage to RAM, while "MEMMAP" leaves the data in storage and creates a memory map which points to the diffraction patterns, allowing them to be retrieved individually from storage.
- **binfactor** (*int*) – Diffraction space binning factor for bin-on-load.
- **filetype** (*str*) – Used to override automatic filetype detection. options include "dm", "empad", "gatan_K2_bin", "mib", "arina", "abTEM"
- ****kwargs** – any additional kwargs are passed to the downstream reader - refer to the individual filetype reader function call signatures and docstrings for more details.

Returns
(DataCube or Array) returns a DataCube if 4D data is found, otherwise returns an Array

py4DSTEM.**save**(*filepath*, *data*, *mode='w'*, *emdpath=None*, *tree=True*)

Saves data to an EMD 1.0 formatted HDF5 file at filepath.

For the full docstring, see py4DSTEM.emdfile.save.

py4DSTEM.**print_h5_tree**(*filepath*, *show_metadata=False*)

Prints the contents of an h5 file from a filepath.

Plotting

py4DSTEM.**show**(*ar*, *figsize=(5, 5)*, *cmap='gray'*, *scaling='none'*, *intensity_range='ordered'*, *clipvals=None*, *vmin=None*, *vmax=None*, *min=None*, *max=None*, *power=None*, *power_offset=True*, *combine_images=False*, *ticks=True*, *bordercolor=None*, *borderwidth=5*, *show_image=True*, *return_ar_scaled=False*, *return_intensity_range=False*, *returncax=False*, *returnfig=False*, *figax=None*, *hist=False*, *n_bins=256*, *mask=None*, *mask_color='k'*, *mask_alpha=0.0*, *masked_intensity_range=False*, *rectangle=None*, *circle=None*, *annulus=None*, *ellipse=None*, *points=None*, *grid_overlay=None*, *cartesian_grid=None*, *polarelliptical_grid=None*, *rtheta_grid=None*, *scalebar=None*, *calibration=None*, *rx=None*, *ry=None*, *space='Q'*, *pixelsize=None*, *pixelunits=None*, *x0=None*, *y0=None*, *a=None*, *e=None*, *theta=None*, *title=None*, *show_fft=False*, *apply_hanning_window=True*, *show_cbar=False*, ***kwargs*)

General visualization function for 2D arrays.

The simplest use of this function is:

```
>>> show(ar)
```

which will generate and display a matplotlib figure showing the 2D array `ar`. Additional functionality includes:

- scaling the image (log scaling, power law scaling)
- displaying the image histogram
- altering the histogram clip values
- masking some subset of the image
- setting the colormap
- adding geometric overlays (e.g. points, circles, rectangles, annuli)
- adding informational overlays (scalebars, coordinate grids, oriented axes or vectors)
- further customization tools

These are each discussed in turn below.

Scaling:
Setting the parameter `scaling` will scale the display image. Options are 'none', 'auto', 'power', or 'log'. If 'power' is specified, the parameter `power` must also be passed. The underlying data is not altered. Values less than or equal to zero are set to zero. If the image histogram is displayed using `hist=True`, the scaled image histogram is shown.

Examples:

```
>>> show(ar,scaling='log')
>>> show(ar,power=0.5)
>>> show(ar,scaling='power',power=0.5,hist=True)
```

Histogram:
Setting the argument `hist=True` will display the image histogram, instead of the image. The displayed histogram will reflect any scaling requested. The number of bins can be set with `n_bins`. The upper and lower clip values, indicating where the image display will be saturated, are shown with dashed lines.

Intensity range:
Controlling the lower and upper values at which the display image will be saturated is accomplished with the `intensity_range` parameter, or its (soon deprecated) alias `clipvals`, in combination with `vmin`, and `vmax`. The method by which the upper and lower clip values are determined is controlled by `intensity_range`, and must be a string in ('None','ordered','minmax','absolute','std','centered'). See the argument description for `intensity_range` for a description of the behavior for each. The clip values can be returned with the `return_intensity_range` parameter.

Masking:
If a numpy masked array is passed to show, the function will automatically mask the appropriate pixels. Alternatively, a boolean array of the same shape as the data array may be passed to the `mask` argument, and these pixels will be masked. Masked pixels are displayed as a single uniform color, black by default, and which can be specified with the `mask_color` argument. Masked pixels are excluded when displaying the histogram or computing clip values. The mask can also be blended with the hidden data by setting the `mask_alpha` argument.

Overlays (geometric):
The function natively supports overlaying points, circles, rectangles, annuli, and ellipses. Each is invoked by passing a dictionary to the appropriate input variable specifying the geometry and features of the requested overlay. For example:

```
>>> show(ar, rectangle={'lims':(10,20,10,20),'color':'r'})
```

will overlay a single red square, and

```
>>> show(ar, annulus={'center':[(28,68),(92,160)],
                      'radii':[(16,24),(12,36)],
                      'fill':True,
                      'alpha':[0.9,0.3],
                      'color':['r',(0,1,1,1)]})
```

will overlay two annuli with two different centers, radii, colors, and transparencies. For a description of the accepted dictionary parameters for each type of overlay, see the visualize functions add_*, where * = ('rectangle','circle','annulus','ellipse','points'). (These docstrings are under construction!)

Overlays (informational):

Informational overlays supported by this function include coordinate axes (cartesian, polar-elliptical, or r-theta) and scalebars. These are added by passing the appropriate input argument a dictionary of the desired parameters, as with geometric overlays. However, there are two key differences between these overlays and the geometric overlays. First, informational overlays (coordinate systems and scalebars) require information about the plot - e.g. the position of the origin, the pixel sizes, the pixel units, any elliptical distortions, etc. The easiest way to pass this information is by pass a Calibration object containing this info to `show` as the keyword `calibration`. Second, once the coordinate information has been passed, informational overlays can autoselect their own parameters, thus simply passing an empty dict to one of these parameters will add that overlay.

For example:

```
>>> show(dp, scalebar={}, calibration=calibration)
```

will display the diffraction pattern `dp` with a scalebar overlaid in the bottom left corner given the pixel size and units described in `calibration`, and

```
>>> show(dp, calibration=calibration, scalebar={'length':0.5,'width':2,
                                                'position':'ul','label':True'})
```

will display a more customized scalebar.

When overlaying coordinate grids, it is important to note that some relevant parameters, e.g. the position of the origin, may change by scan position. In these cases, the parameters `rx`,``ry`` must also be passed to `show`, to tell the `Calibration` object where to look for the relevant parameters. For example:

```
>>> show(dp, cartesian_grid={}, calibration=calibration, rx=2,ry=5)
```

will overlay a cartesian coordinate grid on the diffraction pattern at scan position (2,5). Adding

```
>>> show(dp, calibration=calibration, rx=2, ry=5, cartesian_grid={'label':True,
            'alpha':0.7,'color':'r'})
```

will customize the appearance of the grid further. And

```
>>> show(im, calibration=calibration, cartesian_grid={}, space='R')
```

displays a cartesian grid over a real space image. For more details, see the documentation for the visualize functions add_*, where * = ('scalebar', 'cartesian_grid', 'polarelliptical_grid', 'rtheta_grid'). (Under construction!)

Further customization:

Most parameters accepted by a matplotlib axis will be accepted by `show`. Pass a valid matplotlib colormap or a known string indicating a colormap as the argument `cmap` to specify the colormap. Pass `figsize` to specify the figure size. Etc.

Further customization can be accomplished by either (1) returning the figure generated by show and then manipulating it using the normal matplotlib functions, or (2) generating a matplotlib Figure with Axes any way you like (e.g. with `plt.subplots`) and then using this function to plot inside a single one of the Axes of your choice.

Option (1) is accomplished by simply passing this function `returnfig=True`. Thus:

```
>>> fig,ax = show(ar, returnfig=True)
```

will now give you direct access to the figure and axes to continue to alter. Option (2) is accomplished by passing an existing figure and axis to `show` as a 2-tuple to the `figax` argument. Thus:

```
>>> fig,(ax1,ax2) = plt.subplots(1,2)
>>> show(ar, figax=(fig,ax1))
>>> show(ar, figax=(fig,ax2), hist=True)
```

will generate a 2-axis figure, and then plot the array `ar` as an image on the left, while plotting its histogram on the right.

Parameters

- **ar** (*2D array or a list of 2D arrays*) – the data to plot. Normally this is a 2D array of the data. If a list of 2D arrays is passed, plots a corresponding grid of images.
- **figsize** (*2-tuple*) – size of the plot
- **cmap** (*colormap*) – any matplotlib cmap; default is gray
- **scaling** (*str*) – selects a scaling scheme for the intensity values. Default is none. Accepted values:
  - 'none': do not scale intensity values
  - 'full': fill entire color range with sorted intensity values
  - 'power': power law scaling
  - 'log': values where ar<=0 are set to 0
- **intensity_range** (*str*) –

  method for setting clipvalues (min and max intensities).
  The original name "clipvals" is now deprecated. Default is 'ordered'. Accepted values:

  - 'ordered': vmin/vmax are set to fractions of the distribution of pixel values in the array, e.g. vmin=0.02 will set the minumum display value to saturate the lower 2% of pixels
  - 'minmax': The vmin/vmax values are np.min(ar)/np.max(r)
  - 'absolute': The vmin/vmax values are set to the values of the vmin,vmax arguments received by this function
  - **'std': The vmin/vmax values are `np.median(ar) -/+ N*np.std(ar)`, and**
    N is this functions min,max vals.
  - 'centered': The vmin/vmax values are set to `c -/+ m`, where by default 'c' is zero and m is the max(abs(ar-c), or the two params can be user specified using the kwargs vmin/vmax -> c/m.
- **vmin** (*number*) – min intensity, behavior depends on clipvals
- **vmax** (*number*) – max intensity, behavior depends on clipvals

- **min** – alias' for vmin,vmax, throws deprecation warning
- **max** – alias' for vmin,vmax, throws deprecation warning
- **power** (*number*) – specifies the scaling power
- **power_offset** (*bool*) – If true, image has min value subtracted before power scaling
- **ticks** (*bool*) – Turn outer tick marks on or off
- **bordercolor** (*color or None*) – if not None, add a border of this color. The color can be anything matplotlib recognizes as a color.
- **borderwidth** (*number*) –
- **returnfig** (*bool*) – if True, the function returns the tuple (figure,axis)
- **figax** (*None or 2-tuple*) – controls which matplotlib Axes object draws the image. If None, generates a new figure with a single Axes instance. Otherwise, ax must be a 2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.
- **hist** (*bool*) – if True, instead of plotting a 2D image in ax, plots a histogram of the intensity values of ar, after any scaling this function has performed. Plots the clipvals as dashed vertical lines
- **n_bins** (*int*) – number of hist bins
- **mask** (*None or boolean array*) – if not None, must have the same shape as 'ar'. Wherever mask==True, plot the pixel normally, and where `mask==False`, pixel values are set to mask_color. If hist==True, ignore these values in the histogram. If `mask_alpha` is specified, the mask is blended with the array underneath, with 0 yielding an opaque mask and 1 yielding a fully transparent mask. If `mask_color` is set to `'empty'` instead of a matplotlib.color, nothing is done to pixels where `mask==False`, allowing overlaying multiple arrays in different regions of an image by invoking the ``figax` kwarg over multiple calls to show
- **mask_color** (*color*) – see 'mask'
- **mask_alpha** (*float*) – see 'mask'
- **masked_intensity_range** (*bool*) – controls if masked pixel values are included when determining the display value range; False indicates that all pixel values will be used to determine the intensity range, True indicates only unmasked pixels will be used
- **scalebar** (*None or dict or Bool*) – if None, and a DiffractionSlice or RealSlice with calibrations is passed, adds a scalebar. If scalebar is not displaying the proper calibration, check .calibration pixel_size and pixel_units. If None and an array is passed, does not add a scalebar. If a dict is passed, it is propagated to the add_scalebar function which will attempt to use it to overlay a scalebar. If True, uses calibraiton or pixelsize/pixelunits for scalebar. If False, no scalebar is added.
- **show_fft** (*bool*) – if True, plots 2D-fft of array
- **apply_hanning_window** (*bool*) – If True, a 2D Hann window is applied to the array before applying the FFT
- **show_cbar** (*bool*) – if True, adds cbar
- ****kwargs** – any keywords accepted by matplotlib's ax.matshow()

Returns

if returnfig==False (default), the figure is plotted and nothing is returned. if returnfig==True, return the figure and the axis.

Utilities

py4DSTEM.**check_config**(*verbose: bool = False*, *gratuitously_verbose: bool = False*) → None

This function checks the state of required imports to run py4DSTEM.

Default behaviour will provide a summary of install dependencies for each module e.g. Base, ACOM etc.

Parameters

- **verbose** (*bool, optional*) – Will provide the status of all possible requriements for py4DSTEM, and perform any additonal checks. Defaults to False.
- **gratuitously_verbose** (*bool, optional*) – Provides more indepth analysis. Defaults to False.

Returns

None

py4DSTEM.**join**(*a*, **p*)

Join two or more pathname components, inserting '/' as needed. If any component is an absolute path, all previous path components will be discarded. An empty last part will result in a path that ends with a separator.

py4DSTEM.**tqdmnd**(**args*, ***kwargs*)

An N-dimensional extension of tqdm providing an iterator and progress bar over the product of multiple iterators.

Example Usage:

```
>>> for x,y in tqdmnd(5,6):
>>>     <expression>
```

is equivalent to

```
>>> for x in range(5):
>>>     for y in range(6):
>>>         <expression>
```

with a tqdmnd-style progress bar printed to standard output.

Accepts:

***args: Any number of integers or iterators. Each integer N**
is converted to a *range(N)* iterator. Then a loop is constructed from the Cartesian product of all iterables.

****kwargs: keyword arguments passed through directly to tqdm.**
Full details are available at https://tqdm.github.io A few useful ones:

disable (bool): if True, hide the progress bar keep (bool): if True, delete the progress bar after completion unit (str): unit name for the display of iteration speed unit_scale (bool): whether to scale the displayed units and add

SI prefixes

desc (str): message displayed in front of the progress bar

Returns

At each iteration, a tuple of indices is returned, corresponding to the values of each input iterator (in the same order as the inputs).

1.4.2 Classes

Table of Contents

- *Classes*
  - *Array*
  - *BraggVectors*
  - *Calibration*
  - *Custom*
  - *Data*
  - *DataCube*
  - *DiffractionSlice*
  - *Metadata*
  - *Node*
  - *PointList*
  - *PointListArray*
  - *Probe*
  - *QPoints*
  - *RealSlice*
  - *VirtualDiffraction*
  - *VirtualImage*

Array

class py4DSTEM.**Array**(*data: ndarray*, *name: str | None = 'array'*, *units: str | None = ''*, *dims: list | None = None*, *dim_names: list | None = None*, *dim_units: list | None = None*, *slicelabels=None*)

A class which stores any N-dimensional array-like data, plus basic metadata: a name and units, as well as calibrations for each axis of the array, and names and units for those axis calibrations.

In the simplest usage, only a data array is passed:

```
>>> ar = Array(np.ones((20,20,256,256)))
```

will create an array instance whose data is the numpy array passed, and with automatically populated dimension calibrations in units of pixels.

Additional arguments may be passed to populate the object metadata:

```
>>> ar = Array(
>>>     np.ones((20,20,256,256)),
>>>     name = 'test_array',
>>>     units = 'intensity',
>>>     dims = [
>>>         [0,5],
```

(continues on next page)

(continued from previous page)

```
>>>         [0,5],
>>>         [0,0.01],
>>>         [0,0.01]
>>>     ],
>>>     dim_units = [
>>>         'nm',
>>>         'nm',
>>>         'A^-1',
>>>         'A^-1'
>>>     ],
>>>     dim_names = [
>>>         'rx',
>>>         'ry',
>>>         'qx',
>>>         'qy'
>>>     ],
>>> )
```

will create an array with a name and units for its data, where its first two dimensions are in units of nanometers, have pixel sizes of 5nm, and are described by the handles 'rx' and 'ry', and where its last two dimensions are in units of inverse Angstroms, have pixels sizes of 0.01A^-1, and are described by the handles 'qx' and 'qy'.

Arrays in which the length of each pixel is non-constant are also supported. For instance,

```
>>> x = np.logspace(0,1,100)
>>> y = np.sin(x)
>>> ar = Array(
>>>     y,
>>>     dims = [
>>>         x
>>>     ]
>>> )
```

generates an array representing the values of the sine function sampled 100 times along a logarithmic interval from 1 to 10. In this example, this data could then be plotted with, e.g.

```
>>> plt.scatter(ar.dims[0], ar.data)
```

If the *slicelabels* keyword is passed, the first N-1 dimensions of the array are treated normally, while the final dimension is used to represent distinct arrays which share a common shape and set of dim vectors. Thus

```
>>> ar = Array(
>>>     np.ones((50,50,4)),
>>>     name = 'test_array_stack',
>>>     units = 'intensity',
>>>     dims = [
>>>         [0,2],
>>>         [0,2]
>>>     ],
>>>     dim_units = [
>>>         'nm',
>>>         'nm'
>>>     ],
```

(continues on next page)

(continued from previous page)

```
>>>     dim_names = [
>>>         'rx',
>>>         'ry'
>>>     ],
>>>     slicelabels = [
>>>         'a',
>>>         'b',
>>>         'c',
>>>         'd'
>>>     ]
>>> )
```

will generate a single Array instance containing 4 arrays which each have a shape (50,50) and a common set of dim vectors ['rx','ry'], and which can be indexed into with the names assigned in *slicelabels* using

```
>>> ar.get_slice('a')
```

which will return a 2D (non-stack-like) Array instance with shape (50,50) and the dims assigned above. The Array attribute .rank is equal to the number of dimensions for a non-stack-like Array, and is equal to N-1 for stack-like arrays.

__init__(*data: ndarray*, *name: str | None = 'array'*, *units: str | None = ''*, *dims: list | None = None*, *dim_names: list | None = None*, *dim_units: list | None = None*, *slicelabels=None*)

Accepts:

data (np.ndarray): the data name (str): the name of the Array units (str): units for the pixel values dims (variable): calibration vectors for each of the axes of the data

array. Valid values for each element of the list are None, a number, a 2-element list/array, or an M-element list/array where M is the data array. If None is passed, the dim will be populated with integer values starting at 0 and its units will be set to pixels. If a number is passed, the dim is populated with a vector beginning at zero and increasing linearly by this step size. If a 2-element list/array is passed, the dim is populated with a linear vector with these two numbers as the first two elements. If a list/array of length M is passed, this is used as the dim vector, (and must therefore match this dimension's length). If dims recieves a list of fewer than N arguments for an N-dimensional data array, the extra dimensions are populated as if None were passed, using integer pixel values. If the *dims* parameter is not passed, all dim vectors are populated this way.

dim_units (list): the units for the calibration dim vectors. If
nothing is passed, dims vectors which have been populated automatically with integers corresponding to pixel numbers will be assigned units of 'pixels', and any other dim vectors will be assigned units of 'unknown'. If a list with length < the array dimensions, the passed values are assumed to apply to the first N dimensions, and the remaining values are populated with 'pixels' or 'unknown' as above.

dim_names (list): labels for each axis of the data array. Values
which are not passed, following the same logic as described above, will be autopopulated with the name "dim#" where # is the axis number.

slicelabels (None or True or list): if not None, must be True or a
list of strings, indicating a "stack-like" array. In this case, the first N-1 dimensions of the array are treated normally, in the sense of populating dims, dim_names, and dim_units, while the final dimension is treated distinctly: it indexes into distinct arrays which share a set of dimension attributes, and can be sliced into using the string labels from the *slicelabels* list, with the syntax

array['label'] or array.get_slice('label'). If *slicelabels* is *True* or is a list with length less than the final dimension length, unassigned dimensions are autopopulated with labels *array{i}*. The flag array.is_stack is set to True and the array.rank attribute is set to N-1.

Returns
A new Array instance

`get_dim`(*n*)

Return the n'th dim vector

`dim`(*n*)

Return the n'th dim vector

`set_dim`(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:
n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

`get_dim_units`(*n*)

Return the n'th dim vector units

`set_dim_units`(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:
n (int): specifies which dim vector units (str): new units

`get_dim_name`(*n*)

Get the n'th dim vector name

`set_dim_name`(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:
n (int): specifies which dim vector name (str): new name

`to_h5`(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new array's Group

`add_to_tree`(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

`cut_from_tree`(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

`force_add_to_tree`(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

`classmethod from_h5`(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

`get_from_tree`(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

`graft`(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

`static newnode`(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

BraggVectors

class py4DSTEM.**BraggVectors**(*Rshape*, *Qshape*, *name='braggvectors'*, *verbose=False*, *calibration=None*)

Stores localized bragg scattering positions and intensities for a 4D-STEM datacube.

Raw (detector coordinate) vectors are accessible as

```
>>> braggvectors.raw[ scan_x, scan_y ]
```

and calibrated vectors as

```
>>> braggvectors.cal[ scan_x, scan_y ]
```

To set which calibrations are being applied, call

```
>>> braggvectors.setcal(
>>>     center = bool,
>>>     ellipse = bool,
>>>     pixel = bool,
>>>     rotate = bool
>>> )
```

If .setcal is not called, calibrations will be automatically selected based based on the contents of the instance's *calibrations* property. The calibrations performed in the last call to *braggvectors.cal* are exposed as

```
>>> braggvectors.calstate
```

After grabbing some vectors

```
>>> vects = braggvectors.raw[ scan_x,scan_y ]
```

the values themselves are accessible as

```
>>> vects.qx,vects.qy,vects.I
>>> vects['qx'],vects['qy'],vects['intensity']
```

Alternatively, you can access the centered vectors in pixel units with

```
>>> vects.get_vectors(
>>>     scan_x,
>>>     scan_y,
>>>     center = bool,
>>>     ellipse = bool,
>>>     pixel = bool,
>>>     rotate = bool
>>> )
```

which will return the vectors at scan position (scan_x,scan_y) with the requested calibrations applied.

__init__(*Rshape*, *Qshape*, *name='braggvectors'*, *verbose=False*, *calibration=None*)

set_raw_vectors(*x*)

Given some PointListArray x of the correct shape, sets this to the raw vectors

property raw

Calling

```
>>> raw[ scan_x, scan_y ]
```

returns those bragg vectors.

property cal

Calling

```
>>> cal[ scan_x, scan_y ]
```

retrieves data. Use *.setcal* to set the calibrations to be applied, or *.calstate* to see which calibrations are currently set. Calibrations are initially all set to False. Call *.setcal()* (with no arguments) to automatically detect which calibrations are present and apply those.

setcal(*center=None*, *ellipse=None*, *pixel=None*, *rotate=None*)

Calling

```
>>> braggvectors.setcal(
>>>     center = bool,
>>>     ellipse = bool,
>>>     pixel = bool,
>>>     rotate = bool,
>>> )
```

sets the calibrations that will be applied to vectors subsequently retrieved with

```
>>> braggvectors.cal[ scan_x, scan_y ]
```

Any arguments left as *None* will be automatically set based on the calibration measurements available.

calibrate()

Autoupdate the calstate when relevant calibrations are set

get_vectors(*scan_x*, *scan_y*, *center*, *ellipse*, *pixel*, *rotate*)

Returns the bragg vectors at the specified scan position with the specified calibration state.

Parameters

- **scan_x** (*int*) –
- **scan_y** (*int*) –
- **center** (*bool*) –
- **ellipse** (*bool*) –
- **pixel** (*bool*) –
- **rotate** (*bool*) –

Returns
vectors

Return type
BVects

to_h5(*group*)

Constructs the group, adds the bragg vector pointlists, and adds metadata describing the shape

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

fit_origin(*mask=None*, *fitfunction='plane'*, *robust=False*, *robust_steps=3*, *robust_thresh=2*, *mask_check_data=True*, *plot=True*, *plot_range=None*, *cmap='RdBu_r'*, *returncalc=True*, ***kwargs*)

Fit origin of bragg vectors.

Parameters

- **mask** (*2b boolean array, optional*) – ignore points where mask=True
- **fitfunction** (*str, optional*) – must be 'plane' or 'parabola' or 'bezier_two'
- **robust** (*bool, optional*) – If set to True, fit will be repeated with outliers removed.
- **robust_steps** (*int, optional*) – Optional parameter. Number of robust iterations performed after initial fit.
- **robust_thresh** (*int, optional*) – Threshold for including points, in units of root-mean-square (standard deviations) error of the predicted values after fitting.
- **mask_check_data** (*bool*) – Get mask from origin measurements equal to zero. (TODO - replace)
- **plot** (*bool, optional*) – plot results
- **plot_range** (*float*) – min and max color range for plot (pixels)
- **cmap** (*colormap*) – plotting colormap

Returns

Return value depends on returnfitp. If `returnfitp==False` (default), returns a 4-tuple containing:

- **qx0_fit**: *(ndarray)* the fit origin x-position
- **qy0_fit**: *(ndarray)* the fit origin y-position
- **qx0_residuals**: *(ndarray)* the x-position fit residuals
- **qy0_residuals**: *(ndarray)* the y-position fit residuals

Return type
(variable)

`fit_p_ellipse`(*bvm*, *center*, *fitradii*, *mask=None*, *returncalc=False*, ***kwargs*)

Parameters

- **bvm** (*BraggVectorMap*) – a 2D array used for ellipse fitting
- **center** (*2-tuple of floats*) – the center (x0,y0) of the annular fitting region
- **fitradii** (*2-tuple of floats*) – inner and outer radii (ri,ro) of the fit region
- **mask** (*ar-shaped ndarray of bools*) – ignore data wherever mask==True

Returns
p_ellipse if returncal is True

`force_add_to_tree`(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

`classmethod from_h5`(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_bragg_vector_map(*mode='cal'*, *sampling=1*, *weights=None*, *weights_thresh=0.005*)

Returns a 2D histogram of Bragg vector intensities in diffraction space, aka a Bragg vector map.

Parameters

- **mode** (*str*) – Must be 'cal' or 'raw'. Use the calibrated or raw vector positions.
- **sampling** (*number*) – The sampling rate of the histogram, in units of the camera's sampling. *sampling = 2* upsamples and *sampling = 0.5* downsamples, each by a factor of 2.
- **weights** (*None or array*) – If None, use all real space scan positions. Otherwise must be a real space shaped array representing a weighting factor applied to vector intensities from each scan position. If weights is boolean uses beam positions where weights is True. If weights is number-like, scales by the values, and skips positions where wieghts<weights_thresh.
- **weights_thresh** (*number*) – If weights is an array of numbers, pixels where weights>weight_thresh are skipped.

Returns

An Array with .data representing the data, and .dim[0] and .dim[1] representing the sampling grid.

Return type

BraggVectorHistogram

get_bvm(*mode='cal'*, *sampling=1*, *weights=None*, *weights_thresh=0.005*)

Returns a 2D histogram of Bragg vector intensities in diffraction space, aka a Bragg vector map.

Parameters

- **mode** (*str*) – Must be 'cal' or 'raw'. Use the calibrated or raw vector positions.
- **sampling** (*number*) – The sampling rate of the histogram, in units of the camera's sampling. *sampling = 2* upsamples and *sampling = 0.5* downsamples, each by a factor of 2.
- **weights** (*None or array*) – If None, use all real space scan positions. Otherwise must be a real space shaped array representing a weighting factor applied to vector intensities from each scan position. If weights is boolean uses beam positions where weights is True. If weights is number-like, scales by the values, and skips positions where wieghts<weights_thresh.
- **weights_thresh** (*number*) – If weights is an array of numbers, pixels where weights>weight_thresh are skipped.

Returns

An Array with .data representing the data, and .dim[0] and .dim[1] representing the sampling grid.

Return type

BraggVectorHistogram

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

get_masked_peaks(*mask*, *update_inplace=False*, *returncalc=True*)

Alias for *mask_in_Q*.

`get_virtual_image`(*mode=None*, *geometry=None*, *name='bragg_virtual_image'*, *returncalc=True*, *center=True*, *ellipse=True*, *pixel=True*, *rotate=True*)

Calculates a virtual image based on the values of the Braggvectors integrated over some detector function determined by *mode* and *geometry*.

Parameters

- **mode** (*str*) –

  defines the type of detector used. Options:

  - 'circular', 'circle': uses round detector, like bright field
  - 'annular', 'annulus': uses annular detector, like dark field

- **geometry** (*variable*) –

  expected value depends on the value of *mode*, as follows:

  - 'circle', 'circular': nested 2-tuple, ((qx,qy),radius)
  - 'annular' or 'annulus': nested 2-tuple, ((qx,qy),(radius_i,radius_o))

  Values can be in pixels or calibrated units. Note that (qx,qy) can be skipped, which assumes peaks centered at (0,0).

- **center** (*bool*) – Apply calibration - center coordinate.
- **ellipse** (*bool*) – Apply calibration - elliptical correction.
- **pixel** (*bool*) – Apply calibration - pixel size.
- **rotate** (*bool*) – Apply calibration - QR rotation.

Returns
virtual_im

Return type
VirtualImage

`graft`(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

`histogram`(*mode='cal'*, *sampling=1*, *weights=None*, *weights_thresh=0.005*)

Returns a 2D histogram of Bragg vector intensities in diffraction space, aka a Bragg vector map.

Parameters

- **mode** (*str*) – Must be 'cal' or 'raw'. Use the calibrated or raw vector positions.
- **sampling** (*number*) – The sampling rate of the histogram, in units of the camera's sampling. *sampling = 2* upsamples and *sampling = 0.5* downsamples, each by a factor of 2.

- **weights** (*None or array*) – If None, use all real space scan positions. Otherwise must be a real space shaped array representing a weighting factor applied to vector intensities from each scan position. If weights is boolean uses beam positions where weights is True. If weights is number-like, scales by the values, and skips positions where wieghts<weights_thresh.
- **weights_thresh** (*number*) – If weights is an array of numbers, pixels where weights>weight_thresh are skipped.

Returns
An Array with .data representing the data, and .dim[0] and .dim[1] representing the sampling grid.

Return type
BraggVectorHistogram

mask_in_Q(*mask*, *update_inplace=False*, *returncalc=True*)

Remove peaks which fall inside the diffraction shaped boolean array *mask*, in raw (uncalibrated) peak positions.

Parameters

- **mask** (*2d boolean array*) – The mask. Must be diffraction space shaped
- **update_inplace** (*bool*) – If False (default) copies this BraggVectors instance and removes peaks from the copied instance. If True, removes peaks from this instance.
- **returncalc** (*bool*) – Toggles returning the answer

Returns
bvects

Return type
BraggVectors

mask_in_R(*mask*, *update_inplace=False*, *returncalc=True*)

Remove peaks which fall inside the real space shaped boolean array *mask*.

Parameters

- **mask** (*2d boolean array*) – The mask. Must be real space shaped
- **update_inplace** (*bool*) – If False (default) copies this BraggVectors instance and removes peaks from the copied instance. If True, removes peaks from this instance.
- **returncalc** (*bool*) – Toggles returning the answer

Returns
bvects

Return type
BraggVectors

measure_origin(*center_guess=None*, *score_method=None*, *findcenter='max'*)

Finds the diffraction shifts of the center beam using the raw Bragg vector measurements.

If a center guess is not specified, first, a guess at the unscattered beam position is determined, either by taking the CoM of the Bragg vector map, or by taking its maximal pixel. Once a unscattered beam position is determined, the Bragg peak closest to this position is identified. The shifts in these peaks positions from their average are returned as the diffraction shifts.

Parameters

- **center_guess** (*2-tuple*) – initial guess for the center

- **score_method** (*str*) –

  Method used to find center peak

  - 'intensity': finds the most intense Bragg peak near the center
  - 'distance': finds the closest Bragg peak to the center
  - 'intensity weighted distance': determines center through a combination of weighting distance and intensity

- **(str)** (*findcenter*) – position options: 'CoM', or 'max.' Only used if center_guess is None. CoM finds the center of mass of bragg ector map, 'max' uses its brightest pixel.

- **Returns** – (3-tuple): A 3-tuple comprised of:

  - **qx0** *((R_Nx,R_Ny)-shaped array)*: the origin x-coord
  - **qy0** *((R_Nx,R_Ny)-shaped array)*: the origin y-coord
  - **braggvectormap** *((R_Nx,R_Ny)-shaped array)*: the Bragg vector map of only the Bragg peaks identified with the unscattered beam. Useful for diagnostic purposes.

measure_origin_beamstop(*center_guess*, *radii*, *max_dist=None*, *max_iter=1*, ***kwargs*)

Find the origin from a set of braggpeaks assuming there is a beamstop, by identifying pairs of conjugate peaks inside an annular region and finding their centers of mass.

Parameters

- **center_guess** (*2-tuple*) – qx0,qy0
- **radii** (*2-tuple*) – the inner and outer radii of the specified annular region
- **max_dist** (*number*) – the maximum allowed distance between the reflection of two peaks to consider them conjugate pairs
- **max_iter** (*integer*) – for values >1, repeats the algorithm after updating center_guess

Returns
the origins

Return type
(2d masked array)

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

plot(*index: tuple[int, int] | list[int]*, *cal: str = 'cal'*, *returnfig: bool = False*, ***kwargs*)

Plot Bragg vector, from a specified index. Calls py4DSTEM.process.diffraction.plot_diffraction_pattern(braggvectors.<cal/ra
**kwargs). Optionally can return the figure.

Parameters

- **index** (*tuple[int,int] | list[int]*) – scan position for which Bragg vectors to plot

- **cal** (*str, optional*) – Choice to plot calibrated or raw Bragg vectors must be 'raw' or 'cal', by default 'cal'
- **returnfig** (*bool, optional*) – Boolean to return figure or not, by default False

Returns
matplotlib figure, axes returned if *returnfig* is True

Return type
tuple (figure, axes)

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_strainmap(*name: str | None = None*)

Generate a StrainMap object from the BraggVectors equivalent to py4DSTEM.StrainMap(braggvectors=braggvectors)

Parameters
name (*str, optional*) – The name of the strainmap. Defaults to None which reverts to default name 'strainmap'.

Returns
A py4DSTEM StrainMap object generated from the BraggVectors

Return type
py4DSTEM.StrainMap

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))     # as above
>>> .tree(graft=(node,False))    # as above, discard root metadata
>>> .tree(graft=(node,'copy'))   # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

Calibration

class py4DSTEM.**Calibration**(*name: str | None = 'calibration'*, *root:* Root *| None = None*)

Stores calibration measurements.

Usage

For some calibration instance *c*

```
>>> c['x'] = y
```

will set the value of some calibration item called 'x' to y, and

```
>>> _y = c['x']
```

will return the value currently stored as 'x' and assign it to _y. Additionally, for calibration items in the list *l* given below, the syntax

```
>>> c.set_p(p)
>>> p = c.get_p()
```

is equivalent to

```
>>> c.p = p
>>> p = c.p
```

is equivalent to

```
>>> c['p'] = p
>>> p = c['p']
```

where in the first line of each couplet the parameter *p* is set and in the second it's retrieved, for parameters p in the list

l = [

Q_pixel_size, * R_pixel_size, * Q_pixel_units, * R_pixel_units, * qx0, qy0, qx0_mean, qy0_mean, qx0shift, qy0shift, origin, * origin_meas, origin_meas_mask, origin_shift, a, * b, * theta, * p_ellipse, * ellipse, * QR_rotation_degrees, * QR_flip, * QR_rotflip, * probe_semiangle, probe_param, probe_center, probe_convergence_semiangle_pixels, probe_convergence_semiangle_mrad,

]

There are two advantages to using the getter/setter syntax for parameters in *l* (e.g. either c.set_p or c.p) instead of the normal dictionary-like getter/setter syntax (i.e. c['p']). These are (1) enabling retrieving parameters by beam scan position, and (2) enabling propagation of any calibration changes to downstream data objects which are affected by the altered calibrations. See below.

Get a parameter by beam scan position

Some parameters support retrieval by beam scan position. In these cases, calling

```
>>> c.get_p(rx,ry)
```

will return the value of parameter p at beam position (rx,ry). This works only for the above syntax. Using either of

```
>>> c.p
>>> c['p']
```

will return an R-space shaped array.

Trigger downstream calibrations

Some objects store their own internal calibration state, which depends on the calibrations stored here. For example, a DataCube stores dimension vectors which calibrate its 4 dimensions, and which depend on the pixel sizes and the origin position.

Modifying certain parameters therefore can trigger other objects which depend on these parameters to re-calibrate themselves by calling their .calibrate() method, if the object has one. Methods marked with a * in the list *l* above have this property. Only objects registered with the Calibration instance will have their .calibrate method triggered by changing these parameters. An object *data* can be registered by calling

```
>>> c.register_target( data )
```

and deregistered with

```
>>> c.deregister_target( data )
```

If an object without a .calibrate method is registerd when a * method is called, nothing happens.

The .calibrate methods are triggered by setting some parameter *p* using either

```
>>> c.set_p( val )
```

or

```
>>> c.p = val
```

syntax. Setting the parameter with

```
>>> c['p'] = val
```

will not trigger re-calibrations.

Calibration + Data

Data in py4DSTEM is stored in filetree like representations, and Calibration instances are the top-level objects in these trees, in that they live here:

Root
|–metadata | |– *—> calibration <—* | |–some_object(e.g.datacube) | |–another_object(e.g.max_dp) | |–etc.
|–etc. :

Every py4DSTEM Data object has a tree with a calibration, and calling

```
>>> data.calibration
```

will return the that Calibration instance. See also the docstring for the *Data* class.

Attaching an object to a different Calibration

To modify the calibration associated with some object *data*, use

```
>>> c.attach( data )
```

where *c* is the new calibration instance. This (1) moves *data* into the top level of *c*'s data tree, which means the new calibration will now be accessible normally at

```
>>> data.calibration
```

and (2) if and only if *data* was registered with its old calibration, de-registers it there and registers it with the new calibration. If *data* was not registered with the old calibration and it should be registered with the new one, *c.register_target(data)* should be called.

To attach *data* to a different location in the calibration instance's tree, use *node.attach(data)*. See the Data.attach docstring.

__init__(*name: str | None = 'calibration'*, *root:* Root | *None = None*)

Parameters
name (*optional, str*) –

attach(*data*)

Attach *data* to this calibration instance, placing it in the top level of the Calibration instance's tree. If *data* was in a different data tree, remove it. If *data* was registered with a different calibration instance, de-register it there and register it here. If *data* was not previously registerd and it should be, after attaching it run *self.register_target(data)*.

register_target(*new_target*)

Register an object to recieve calls to it *calibrate* method when certain calibrations get updated

unregister_target(*target*)

Unlink an object from recieving calls to *calibrate* when certain calibration values are changed

set_origin_meas(*x*)

Parameters
x (*2-tuple or 3 uple of 2D R-shaped arrays*) – qx0,qy0,[mask]

`set_probe_param`(*x*)

Parameters
x (*3-tuple*) – (probe size, x0, y0)

`to_h5`(*group*)

Saves the metadata dictionary _params to group, then adds the calibration's target's list

`classmethod from_h5`(*group*)

Takes a valid group for an HDF5 file object which is open in read mode. Determines if it's a valid Metadata representation, and if so loads and returns it as a Calibration instance. Otherwise, raises an exception.

Accepts:
group (HDF5 group)

Returns
A Calibration instance

Custom

`class` py4DSTEM.`Custom`(*name='custom'*)

`__init__`(*name='custom'*)

`to_h5`(*group*)

Constructs an h5 group, adds metadata, and adds all attributes which point to EMD nodes.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new node's Group

`add_to_tree`(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

`cut_from_tree`(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

`force_add_to_tree`(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()             # show tree from current node
>>> .tree(show=True)    # show from root
>>> .tree(show=False)   # show from current node
>>> .tree(add=node)     # add a child node
>>> .tree(get='path')   # return a '/' delimited child node
>>> .tree(get='/path')  # as above, starting at root
>>> .tree(cut=True)     # remove/return a branch, keep root metadata
>>> .tree(cut=False)    # remove/return a branch, discard root md
>>> .tree(cut='copy')   # remove/return a branch, copy root metadata
```

(continues on next page)

(continued from previous page)

```
>>> .tree(graft=node)   # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

Data

class py4DSTEM.**Data**(*calibration=None*)

The purpose of the *Data* class is to ensure calibrations are linked to data containing class instances, while allowing multiple objects to share a single Calibration. The calibrations of a Data instance *data* is accessible as

```
>>> data.calibration
```

In py4DSTEM, Data containing objects are stored internally in filetree like representations, defined by the EMD1.0 and *emdfile* specifications, e.g.

Root

|–metadata | |–calibration | |–some_object(e.g.datacube) | |–another_object(e.g.max_dp) | |–etc. | |–one_more_object(e.g.crystal) | |–etc. :

Calibrations are metadata which always live in the root of such a tree. Running *data.calibration* returns the calibrations from the tree root, and therefore the same calibration instance is referred to be all objects in the same tree. The root itself is accessible from any Data instance as

```
>>> data.root
```

To examine the tree of a Data instance, in a Python interpreter do

```
>>> data.tree(True)
```

to display the whole data tree, and

```
>>> data.tree()
```

to display the tree of from the current node on, i.e. the branch downstream of *data*.

Calling

```
>>> data.calibration
```

will raise a warning and return None if no root calibrations are found.

Some objects should be modified when the calibrations change - these objects must have .calibrate() method, which is called any time relevant calibration parameters change if the object has been registered with the calibrations.

To transfer *data* from it's current tree to another existing tree, use

```
>>> data.attach(some_other_data)
```

which will move the data to the new tree. If the data was registered with it's old calibrations, this will also de-register it there and register it with the new calibrations such that .calibrate() is called when it should be.

See also the Calibration docstring.

__init__(*calibration=None*)

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

DataCube

class py4DSTEM.**DataCube**(*data: ndarray*, *name: str | None = 'datacube'*, *slicelabels: bool | list | None = None*, *calibration:* Calibration *| None = None*)

Storage and processing methods for 4D-STEM datasets.

__init__(*data: ndarray*, *name: str | None = 'datacube'*, *slicelabels: bool | list | None = None*, *calibration:* Calibration *| None = None*)

Accepts:

data (np.ndarray): the data name (str): the name of the datacube calibration (None or Calibration or 'pass'): default (None)

creates and attaches a new Calibration instance to root metadata, or, passing a Calibration instance uses this instead.

slicelabels (None or list): names for slices if this is a
stack of datacubes

Returns
A new DataCube instance.

calibrate()

Calibrate the coordinate axes of the datacube. Using the calibrations at self.calibration, sets the 4 dim vectors (Qx,Qy,Rx,Ry) according to the pixel size, units and origin positions, then updates the meshgrids representing Q and R space.

copy()

Copys datacube

add(*data*, *name=''*)

Adds a block of data to the DataCube's tree. If *data* is an instance of an EMD/py4DSTEM class, add it to the tree. If it's a numpy array, turn it into an Array instance, then save to the tree.

set_scan_shape(*Rshape*)

Reshape the data given the real space scan shape.

Accepts:
Rshape (2-tuple)

swap_RQ()

Swaps the first and last two dimensions of the 4D datacube.

swap_Rxy()

Swaps the real space x and y coordinates.

swap_Qxy()

Swaps the diffraction space x and y coordinates.

crop_Q(*ROI*)

Crops the data in diffraction space about the region specified by ROI.

Accepts:
ROI (4-tuple): Specifies (Qx_min,Qx_max,Qy_min,Qy_max)

crop_R(*ROI*)

Crops the data in real space about the region specified by ROI.

Accepts:
ROI (4-tuple): Specifies (Rx_min,Rx_max,Ry_min,Ry_max)

bin_Q(*N*, *dtype=None*)

Bins the data in diffraction space by bin factor N

Parameters

- **N** (*int*) – The binning factor
- **dtype** (*a datatype (optional)*) – Specify the datatype for the output. If not passed, the datatype is left unchanged

Returns
datacube

Return type
DataCube

pad_Q(*N=None*, *output_size=None*)

Pads the data in diffraction space by pad factor N, or to match output_size.

Accepts:
N (float, or Sequence[float]): the padding factor output_size ((int,int)): the padded output size

resample_Q(*N=None*, *output_size=None*, *method='bilinear'*, *conserve_array_sums=False*)

Resamples the data in diffraction space by resampling factor N, or to match output_size, using either 'fourier' or 'bilinear' interpolation.

Accepts:
N (float, or Sequence[float]): the resampling factor output_size ((int,int)): the resampled output size method (str): 'fourier' or 'bilinear' (default)

bin_Q_mmap(*N*, *dtype=<class 'numpy.float32'>*)

Bins the data in diffraction space by bin factor N for memory mapped data

Accepts:
N (int): the binning factor dtype: the data type

bin_R(*N*)

Bins the data in real space by bin factor N

Accepts:
N (int): the binning factor

thin_R(*N*)

Reduces the data in real space by skipping every N patterns in the x and y directions.

Accepts:
N (int): the thinning factor

filter_hot_pixels(*thresh*, *ind_compare=1*, *return_mask=False*)

This function performs pixel filtering to remove hot / bright pixels. We first compute a moving local ordering filter, applied to the mean diffraction image. This ordering filter will return a single value from the local sorted intensity values, given by ind_compare. ind_compare=0 would be the highest intensity, =1 would be the second hightest, etc. Next, a mask is generated for all pixels which are least a value thresh higher than the local ordering filter output. Finally, we loop through all diffraction images, and any pixels defined by mask are replaced by their 3x3 local median.

Parameters

- **datacube** (DataCube) –
- **thresh** (*float*) – threshold for replacing hot pixels, if pixel value minus local ordering filter exceeds it.
- **ind_compare** (*int*) – which median filter value to compare against. 0 = brightest pixel, 1 = next brightest, etc.
- **return_mask** (*bool*) – if True, returns the filter mask

Returns

datacube (DataCube) mask (optional, boolean Array) the bad pixel mask

median_filter_masked_pixels(*mask*, *kernel_width: int = 3*)

This function fixes a datacube where the same pixels are consistently bad. It requires a mask that identifies all the bad pixels in the dataset. Then for each diffraction pattern, a median kernel is applied around each bad pixel with the specified width.

get_vacuum_probe(*ROI=None*, *align=True*, *mask=None*, *threshold=0.0*, *expansion=12*, *opening=3*, *verbose=False*, *returncalc=True*)

Computes a vacuum probe.

Which diffraction patterns are included in the calculation is specified by the *ROI* parameter. Diffraction patterns are aligned before averaging if *align* is True (default). A global mask is applied to each diffraction pattern before aligning/averaging if *mask* is specified. After averaging, a final masking step is applied according to the parameters *threshold*, *expansion*, and *opening*.

Parameters

- **ROI** (*optional, boolean array or len 4 list/tuple*) – If unspecified, uses the whole datacube. If a boolean array is passed must be real-space shaped, and True pixels are used. If a 4-tuple is passed, uses the region inside the limits (rx_min,rx_max,ry_min,ry_max)
- **align** (*optional, bool*) – if True, aligns the probes before averaging
- **mask** (*optional, array*) – mask applied to each diffraction pattern before alignment and averaging
- **threshold** (*float*) – in the final masking step, values less than max(probe)*threshold are considered outside the probe
- **expansion** (*int*) – number of pixels by which the final mask is expanded after thresholding
- **opening** (*int*) – size of binary opening applied to the final mask to eliminate stray bright pixels
- **verbose** (*bool*) – toggles verbose output
- **returncalc** (*bool*) – if True, returns the answer

Returns
probe – the vacuum probe

Return type
Probe, optional

`get_probe_size`(*dp=None*, *thresh_lower=0.01*, *thresh_upper=0.99*, *N=100*, *plot=False*, *returncal=True*, *write_to_cal=True*, ***kwargs*)

Gets the center and radius of the probe in the diffraction plane.

The algorithm is as follows: First, create a series of N binary masks, by thresholding the diffraction pattern DP with a linspace of N thresholds from thresh_lower to thresh_upper, measured relative to the maximum intensity in DP. Using the area of each binary mask, calculate the radius r of a circular probe. Because the central disk is typically very intense relative to the rest of the DP, r should change very little over a wide range of intermediate values of the threshold. The range in which r is trustworthy is found by taking the derivative of r(thresh) and finding identifying where it is small. The radius is taken to be the mean of these r values. Using the threshold corresponding to this r, a mask is created and the CoM of the DP times this mask it taken. This is taken to be the origin x0,y0.

Parameters

- **dp** (*str or array*) – specifies the diffraction pattern in which to find the central disk. A position averaged, or shift-corrected and averaged, DP works best. If mode is None, the diffraction pattern stored in the tree from 'get_dp_mean' is used. If mode is a string it specifies the name of another virtual diffraction pattern in the tree. If mode is an array, the array is used to calculate probe size.
- **thresh_lower** (*float, 0 to 1*) – the lower limit of threshold values
- **thresh_upper** (*float, 0 to 1*) – the upper limit of threshold values
- **N** (*int*) – the number of thresholds / masks to use
- **plot** (*bool*) – if True plots results
- **plot_params** (*dict*) – dictionary to modify defaults in plot
- **return_calc** (*bool*) – if True returns 3-tuple described below
- **write_to_cal** (*bool*) – if True, looks for a Calibration instance and writes the measured probe radius there

Returns

A 3-tuple containing:

- **r**: *(float)* the central disk radius, in pixels
- **x0**: *(float)* the x position of the central disk center
- **y0**: *(float)* the y position of the central disk center

Return type
(3-tuple)

`find_Bragg_disks`(*template*, *data=None*, *radial_bksb=False*, *filter_function=None*, *corrPower=1*, *sigma=None*, *sigma_dp=0*, *sigma_cc=2*, *subpixel='multicorr'*, *upsample_factor=16*, *minAbsoluteIntensity=0*, *minRelativeIntensity=0.005*, *relativeToPeak=0*, *minPeakSpacing=60*, *edgeBoundary=20*, *maxNumPeaks=70*, *CUDA=False*, *CUDA_batched=True*, *distributed=None*, *ML=False*, *ml_model_path=None*, *ml_num_attempts=1*, *ml_batch_size=8*, *name='braggvectors'*, *returncalc=True*)

Finds the Bragg disks in the diffraction patterns represented by *data* by cross/phase correlatin with *template*.

Behavior depends on *data*. If it is None (default), runs on the whole DataCube, and stores the output in its tree. Otherwise, nothing is stored in tree, but some value is returned. Valid entries are:

- **a 2-tuple of numbers (rx,ry): run on this diffraction image,**
  and return a QPoints instance
- **a 2-tuple of arrays (rx,ry): run on these diffraction images,**
  and return a list of QPoints instances
- **an Rspace shapped 2D boolean array: run on the diffraction images**
  specified by the True counts and return a list of QPoints instances

For disk detection on a full DataCube, the calculation can be performed on the CPU, GPU or a cluster. By default the CPU is used. If *CUDA* is set to True, tries to use the GPU. If *CUDA_batched* is also set to True, batches the FFT/IFFT computations on the GPU. For distribution to a cluster, distributed must be set to a dictionary, with contents describing how distributed processing should be performed - see below for details.

For each diffraction pattern, the algorithm works in 4 steps:

(1) any pre-processing is performed to the diffraction image. This is accomplished by passing a callable function to the argument *filter_function*, a bool to the argument *radial_bksb*, or a value >0 to *sigma_dp*. If none of these are passed, this step is skipped.

(2) the diffraction image is cross correlated with the template. Phase/hybrid correlations can be used instead by setting the *corrPower* argument. Cross correlation can be skipped entirely, and the subsequent steps performed directly on the diffraction image instead of the cross correlation, by passing None to *template*.

(3) the maxima of the cross correlation are located and their positions and intensities stored. The cross correlation may be passed through a gaussian filter first by passing the *sigma_cc* argument. The method for maximum detection can be set with the *subpixel* parameter. Options, from something like fastest/least precise to slowest/most precise are 'pixel', 'poly', and 'multicorr'.

(4) filtering is applied to remove untrusted or undesired positive counts, based on their intensity (*minRelativeIntensity*,`relativeToPeak`, *minAbsoluteIntensity*) their proximity to one another or the image edge (*minPeakSpacing*, *edgeBoundary*), and the total number of peaks per pattern (*maxNumPeaks*).

Parameters

- **template** (*2D array*) – the vacuum probe template, in real space. For Probe instances, this is *probe.kernel*. If None, does not perform a cross correlation.
- **data** (*variable*) – see above
- **radial_bksb** (*bool*) – if True, computes a radial background given by the median of the (circular) polar transform of each each diffraction pattern, and subtracts this background from the pattern before applying any filter function and computing the cross correlation. The origin position must be set in the datacube's calibrations. Currently only supported for full datacubes on the CPU.
- **filter_function** (*callable*) – filtering function to apply to each diffraction pattern before peak finding. Must be a function of only one argument (the diffraction pattern) and return the filtered diffraction pattern. The shape of the returned DP must match the shape of the probe kernel (but does not need to match the shape of the input diffraction pattern, e.g. the filter can be used to bin the diffraction pattern). If using distributed disk detection, the function must be able to be pickled with by dill.
- **corrPower** (*float between 0 and 1, inclusive*) – the cross correlation power. A value of 1 corresponds to a cross correlation, 0 corresponds to a phase correlation, and intermediate values correspond to hybrid correlations.

- **sigma** (*float*) – alias for *sigma_cc*
- **sigma_dp** (*float*) – if >0, a gaussian smoothing filter with this standard deviation is applied to the diffraction pattern before maxima are detected
- **sigma_cc** (*float*) – if >0, a gaussian smoothing filter with this standard deviation is applied to the cross correlation before maxima are detected
- **subpixel** (*str*) – Whether to use subpixel fitting, and which algorithm to use. Must be in ('none','poly','multicorr').
  - 'none': performs no subpixel fitting
  - 'poly': polynomial interpolation of correlogram peaks (default)
  - 'multicorr': uses the multicorr algorithm with DFT upsampling
- **upsample_factor** (*int*) – upsampling factor for subpixel fitting (only used when subpixel='multicorr')
- **minAbsoluteIntensity** (*float*) – the minimum acceptable correlation peak intensity, on an absolute scale
- **minRelativeIntensity** (*float*) – the minimum acceptable correlation peak intensity, relative to the intensity of the brightest peak
- **relativeToPeak** (*int*) – specifies the peak against which the minimum relative intensity is measured – 0=brightest maximum. 1=next brightest, etc.
- **minPeakSpacing** (*float*) – the minimum acceptable spacing between detected peaks
- **(int)** (*edgeBoundary*) – the diffraction image edge, in pixels.
- **maxNumPeaks** (*int*) – the maximum number of peaks to return
- **CUDA** (*bool*) – If True, import cupy and use an NVIDIA GPU to perform disk detection
- **CUDA_batched** (*bool*) – If True, and CUDA is selected, the FFT and IFFT steps of disk detection are performed in batches to better utilize GPU resources.
- **distributed** (*dict*) – contains information for parallel processing using an IPyParallel or Dask distributed cluster. Valid keys are:
  - ipyparallel (dict):
  - **client_file (str): path to client json for connecting to your** existing IPyParallel cluster
  - **dask (dict): client (object): a dask client that connects to** your existing Dask cluster
  - **data_file (str): the absolute path to your original data** file containing the datacube
  - **cluster_path (str): defaults to the working directory during** processing

  if distributed is None, which is the default, processing will be in serial
- **name** (*str*) – name for the output BraggVectors
- **returncalc** (*bool*) – if True, returns the answer

Returns

See above.

Return type
variable

get_beamstop_mask(*threshold=0.25*, *distance_edge=2.0*, *include_edges=True*, *sigma=0*, *use_max_dp=False*, *scale_radial=None*, *name='mask_beamstop'*, *returncalc=True*)

This function uses the mean diffraction pattern plus a threshold to create a beamstop mask.

Parameters

- **threshold** (*float*) – Value from 0 to 1 defining initial threshold for beamstop mask, taken from the sorted intensity values - 0 is the dimmest pixel, while 1 uses the brighted pixels.
- **distance_edge** (*float*) – How many pixels to expand the mask.
- **include_edges** (*bool*) – If set to True, edge pixels will be included in the mask.
- **sigma** (*float*) – Gaussain blur std to apply to image before thresholding.
- **use_max_dp** (*bool*) – Use the max DP instead of the mean DP.
- **scale_radial** (*float*) – Scale from center of image by this factor (can help with edge)
- **name** (*string*) – Name of the output array.
- **returncalc** (*bool*) – Set to true to return the result.

Returns
if returncalc is True, returns the beamstop mask

Return type
(Optional)

get_radial_bkgrnd(*rx*, *ry*, *sigma=2*)

Computes and returns a background image for the diffraction pattern at (rx,ry), populated by radial rings of constant intensity about the origin, with the value of each ring given by the median value of the diffraction pattern at that radial distance.

Parameters

- **rx** (*int*) – The x-coord of the beam position
- **ry** (*int*) – The y-coord of the beam position
- **sigma** (*number*) – If >0, applying a gaussian smoothing in the radial direction before returning

Returns
background – The radial background

Return type
ndarray

get_radial_bksb_dp(*rx*, *ry*, *sigma=2*)

Computes and returns the diffraction pattern at beam position (rx,ry) with a radial background subtracted. See the docstring for datacube.get_radial_background for more info.

Parameters

- **rx** (*int*) – The x-coord of the beam position
- **ry** (*int*) – The y-coord of the beam position

- **sigma** (*number*) – If >0, applying a gaussian smoothing in the radial direction before returning

Returns
data – The radial background subtracted diffraction image

Return type
ndarray

get_local_ave_dp(*rx*, *ry*, *radial_bksb=False*, *sigma=2*, *braggmask=False*, *braggvectors=None*, *braggmask_radius=None*)

Computes and returns the diffraction pattern at beam position (rx,ry) after weighted local averaging with its nearest-neighbor patterns, using a 3x3 gaussian kernel for the weightings.

Parameters

- **rx** (*int*) – The x-coord of the beam position
- **ry** (*int*) – The y-coord of the beam position
- **radial_bksb** (*bool*) – It True, apply a radial background subtraction to each pattern before averaging
- **sigma** (*number*) – If radial_bksb is True, use this sigma for radial smoothing of the background
- **braggmask** (*bool*) – If True, masks bragg scattering at each scan position before averaging. *braggvectors* and *braggmask_radius* must be specified.
- **braggvectors** (BraggVectors) – The Bragg vectors to use for masking
- **braggmask_radius** (*number*) – The radius about each Bragg point to mask

Returns
data – The radial background subtracted diffraction image

Return type
ndarray

get_braggmask(*braggvectors*, *rx*, *ry*, *radius*)

Returns a boolean mask which is False in a radius of *radius* around each bragg scattering vector at scan position (rx,ry).

Parameters

- **braggvectors** (BraggVectors) – The bragg vectors
- **rx** (*int*) – The x-coord of the beam position
- **ry** (*int*) – The y-coord of the beam position
- **radius** (*number*) – mask pixels about each bragg vector to this radial distance

Returns
mask

Return type
boolean ndarray

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

dim(*n*)

Return the n'th dim vector

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

static get_calibrated_detector_geometry(*calibration*, *mode*, *geometry*, *centered*, *calibrated*)

Determine the detector geometry in pixels, given some mode and geometry in calibrated units, where the calibration state is specified by { centered, calibrated}

Parameters

- **calibration** (Calibration) – Used to retrieve the center positions. If *None*, confirms that centered and calibrated are False then passes, otherwise raises an exception
- **mode** (*str*) – see the DataCube.get_virtual_image docstring
- **geometry** (*variable*) – see the DataCube.get_virtual_image docstring
- **centered** (*bool*) – see the DataCube.get_virtual_image docstring
- **calibrated** (*bool*) – see the DataCube.get_virtual_image docstring

Returns
geo – the geometry in detector pixels

Return type
tuple

get_dim(*n*)

Return the n'th dim vector

get_dim_name(*n*)

Get the n'th dim vector name

get_dim_units(*n*)

Return the n'th dim vector units

get_dp_max(*returncalc=True*)

Calculates the max diffraction pattern.

Calls *DataCube.get_virtual_diffraction* - see that method's docstring for more custimizable virtual diffraction.

Parameters
returncalc (*bool*) – toggles returning the answer

Returns
max_dp

Return type
VirtualDiffraction

get_dp_mean(*returncalc=True*)

Calculates the mean diffraction pattern.

Calls *DataCube.get_virtual_diffraction* - see that method's docstring for more custimizable virtual diffraction.

Parameters
returncalc (*bool*) – toggles returning the answer

Returns
mean_dp

Return type
VirtualDiffraction

get_dp_median(*returncalc=True*)

Calculates the max diffraction pattern.

Calls *DataCube.get_virtual_diffraction* - see that method's docstring for more custimizable virtual diffraction.

Parameters
returncalc (*bool*) – toggles returning the answer

Returns
max_dp

Return type
VirtualDiffraction

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

get_virtual_diffraction(*method*, *mask=None*, *shift_center=False*, *subpixel=False*, *verbose=True*, *name='virtual_diffraction'*, *returncalc=True*)

Function to calculate virtual diffraction images.

Parameters

- **method** (*str*) – defines method used for averaging/combining diffraction patterns. Options are ('mean', 'median', 'max')
- **mask** (*None or 2D array*) – if None (default), all pixels are used. Otherwise, must be a boolean or floating point or complex array with the same shape as real space. For bool arrays, only True pixels are used in the computation. Otherwise a weighted average is performed.
- **shift_center** (*bool*) – toggles shifting the diffraction patterns to account for beam shift. Currently only supported for 'max' and 'mean' modes. Default is False.
- **subpixel** (*bool*) – if shift_center is True, toggles subpixel shifts via Fourier interpolation. Ignored if shift_center is False.
- **verbose** (*bool*) – toggles progress bar
- **name** (*string*) – name for the output DiffractionImage instance
- **returncalc** (*bool*) – toggles returning the output

Returns
diff_im

Return type
DiffractionImage

get_virtual_image(*mode*, *geometry*, *centered=False*, *calibrated=False*, *shift_center=False*, *subpixel=False*, *verbose=True*, *dask=False*, *return_mask=False*, *name='virtual_image'*, *returncalc=True*, *test_config=False*)

Calculate a virtual image.

The detector is determined by the combination of the *mode* and *geometry* arguments, supporting point, circular, rectangular, annular, and custom mask detectors. The values passed to geometry may be given with respect to an origin at the corner of the detector array or with respect to the calibrated center position, and in units of pixels or real calibrated units, depending on the values of the *centered* and *calibrated* arguments, respectively. The mask may be shifted pattern-by-pattern to account for diffraction scan shifts using the *shift_center* argument.

The computed virtual image is stored in the datacube's tree, and is also returned by default.

Parameters

- **mode** (*str*) – defines geometry mode for calculating virtual image, and the expected input for the *geometry* argument. options:
  - 'point': uses a single pixel detector
  - 'circle', 'circular': uses a round detector, like bright field
  - 'annular', 'annulus': uses an annular detector, like dark field
  - 'rectangle', 'square', 'rectangular': uses rectangular detector
  - 'mask': any diffraction-space shaped 2D array, representing a flexible detector
- **geometry** (*variable*) – the expected value of this argument is determined by *mode* as follows:
  - 'point': 2-tuple, (qx,qy), ints
  - 'circle', 'circular': nested 2-tuple, ((qx,qy),radius),
  - 'annular', 'annulus': nested 2-tuple, ((qx,qy),(radius_i,radius_o)),
  - 'rectangle', 'square', 'rectangular': 4-tuple, (xmin,xmax,ymin,ymax)

- ***mask*: any boolean or floating point 2D array with the same size as datacube.Qshape**

- **centered** (*bool*) – if False, the origin is in the upper left corner. If True, the origin is set to the mean origin in the datacube calibrations, so that a bright-field image could be specified with, e.g., geometry=((0,0),R). The origin can set with datacube.calibration.set_origin(). For *mode="mask"*, has no effect. Default is False.
- **calibrated** (*bool*) – if True, geometry is specified in units of 'A^-1' instead of pixels. The datacube's calibrations must have its *"Q_pixel_units"* parameter set to "A^-1". For *mode="mask"*, has no effect. Default is False.
- **shift_center** (*bool*) – if True, the mask is shifted at each real space position to account for any shifting of the origin of the diffraction images. The datacube's calibration['origin'] parameter must be set. The shift applied to each pattern is the difference between the local origin position and the mean origin position over all patterns, rounded to the nearest integer for speed. Default is False. If *shift_center* is True, *centered* is automatically set to True.
- **subpixel** (*bool*) – if True, applies subpixel shifts to virtual image
- **verbose** (*bool*) – toggles a progress bar
- **dask** (*bool*) – if True, use dask to distribute the calculation
- **return_mask** (*bool*) – if False (default) returns a virtual image as usual. Otherwise does *not* compute or return a virtual image, instead finding and returning the mask that will be used in subsequent calls to this function using these same parameters. In this case, must be either *True* or a 2-tuple of integers corresponding to *(rx,ry)*. If True is passed, returns the mask used if *shift_center* is set to False. If a 2-tuple is passed, returns the mask used at scan position (rx,ry) if *shift_center* is set to True. Nothing is added to the datacube's tree.
- **name** (*str*) – the output object's name
- **returncalc** (*bool*) – if True, returns the output
- **test_config** (*bool*) – if True, prints the Boolean values of (*centered*,`calibrated`,`shift_center`). Does not compute the virtual image.

Returns
virt_im

Return type
VirtualImage (optional, if returncalc is True)

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

make_bragg_mask(*Qshape*, *g1*, *g2*, *radius*, *origin*, *max_q*, *return_sum=True*, *include_origin=True*, *rotation_deg=0*, ***kwargs*)

Creates and returns a mask consisting of circular disks about the points of a 2D lattice.

Parameters

- **Qshape** (*2 tuple*) – the shape of diffraction space
- **g1** (*len 2 array or tuple*) – the lattice vectors
- **g2** (*len 2 array or tuple*) – the lattice vectors
- **radius** (*number*) – the disk radius
- **origin** (*len 2 array or tuple*) – the origin
- **max_q** (*nuumber*) – the maxima distance to tile to
- **return_sum** (*bool*) – if False, return a 3D array, where each slice contains a single disk; if False, return a single 2D masks of all disks
- **include_origin** (*bool*) – if False, removes origin disk
- **rotation_deg** (*float*) – rotate g1 and g2 vectors

Returns
(2 or 3D array) the mask

static make_detector(*shape*, *mode*, *geometry*)

Generate a 2D mask representing a detector function.

Parameters

- **shape** (*2-tuple*) – defines shape of mask. Should be the shape of diffraction space.
- **mode** (*str*) – defines geometry mode for calculating virtual image. See the docstring for DataCube.get_virtual_image
- **geometry** (*variable*) – defines geometry for calculating virtual image. See the docstring for DataCube.get_virtual_image

Returns
detector_mask

Return type
2d array

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

position_detector(*mode*, *geometry*, *data=None*, *centered=None*, *calibrated=None*, *shift_center=False*, *subpixel=True*, *scan_position=None*, *invert=False*, *color='r'*, *alpha=0.7*, ***kwargs*)

Position a virtual detector by displaying a mask over a diffraction space image. Calling *.get_virtual_image()* using the same *mode* and *geometry* parameters will compute a virtual image using this detector.

Parameters

- **mode** (*str*) – see the DataCube.get_virtual_image docstring
- **geometry** (*variable*) – see the DataCube.get_virtual_image docstring
- **data** (*None or 2d-array or 2-tuple of ints*) – The diffraction image to overlay the mask on. If *None* (default), looks for a max or mean or median diffraction image in this order and if found, uses it, otherwise, uses the diffraction pattern at scan position (0,0). If a 2d array is passed, must be diffraction space shaped array. If a 2-tuple is passed, uses the diffraction pattern at scan position (rx,ry).
- **centered** (*bool*) – see the DataCube.get_virtual_image docstring
- **calibrated** (*bool*) – see the DataCube.get_virtual_image docstring
- **shift_center** (*None or bool or 2-tuple of ints*) – If *None* (default) and *data* is either None or an array, the mask is not shifted. If *None* and *data* is a 2-tuple, shifts the mask according to the origin at the scan position (rx,ry) specified in *data*. If False, does not shift the mask. If True and *data* is a 2-tuple, shifts the mask accordingly, and if True and *data* is any other value, raises an error. If *shift_center* is a 2-tuple, shifts the mask according to the origin value at this 2-tuple regardless of the value of *data* (enabling e.g. overlaying the mask for a specific scan position on a max or mean diffraction image.)
- **subpixel** (*bool*) – if True, applies subpixel shifts to virtual image
- **invert** (*bool*) – if True, invert the masked pixel (i.e. pixels *outside* the detector are overlaid with a mask)
- **color** (*any matplotlib color specification*) – the mask color
- **alpha** (*number*) – the mask transparency
- **kwargs** (*dict*) – Any additional arguments are passed on to the show() function

set_dim(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:

n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

set_dim_name(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:

n (int): specifies which dim vector name (str): new name

set_dim_units(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:

n (int): specifies which dim vector units (str): new units

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new array's Group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

DiffractionSlice

class py4DSTEM.**DiffractionSlice**(*data: ndarray*, *name: str | None = 'diffractionslice'*, *units: str | None = 'intensity'*, *slicelabels: bool | list | None = None*, *calibration=None*)

Stores a diffraction-space shaped 2D data array.

__init__(*data: ndarray*, *name: str | None = 'diffractionslice'*, *units: str | None = 'intensity'*, *slicelabels: bool | list | None = None*, *calibration=None*)

Accepts:

data (np.ndarray): the data name (str): the name of the diffslice units (str): units of the pixel values slicelabels(None or list): names for slices if this is a 3D stack

Returns

(DiffractionSlice instance)

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

dim(*n*)

Return the n'th dim vector

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_dim(*n*)

Return the n'th dim vector

get_dim_name(*n*)

Get the n'th dim vector name

get_dim_units(*n*)

Return the n'th dim vector units

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

set_dim(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:
n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

set_dim_name(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:
n (int): specifies which dim vector name (str): new name

set_dim_units(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:
n (int): specifies which dim vector units (str): new units

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new array's Group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

Metadata

class py4DSTEM.**Metadata**(*name: str | None = 'metadata'*, *data: dict | None = None*)

Stores metadata in the form of a flat (non-nested) dictionary. Keys are arbitrary strings. Values may be strings, numbers, arrays, or lists of the above types.

Usage:

```
>>> meta = Metadata()
>>> meta['param'] = value
>>> val = meta['param']
```

If the parameter has not been set, the getter methods return None.

__init__(*name: str | None = 'metadata'*, *data: dict | None = None*)

Parameters
name (*Optional, string*) –

copy(*name=None*)

to_h5(*group*)

Accepts an h5py Group which is open in write or append mode. Writes a new group with this object's name and saves its metadata in it.

Accepts:
group (h5py Group)

classmethod from_h5(*group*)

Accepts an h5py Group which is open in read mode, confirms that it represents an EMD MetadataDict group, then loads and returns it as a Metadata instance.

Accepts:
group (HDF5 group)

Returns
(Metadata)

Node

class py4DSTEM.**Node**(*name: str | None = 'node'*)

Nodes contain attributes and methods paralleling the EMD 1.0 file specification in Python runtime objects.

EMD 1.0 is a singly-rooted file format. That is to say: An EMD data object can and must exist in one and only one EMD tree. An EMD file can contain any number of EMD trees, each containing data and metadata which is, within the limits of the EMD group specifications, of some arbitrary complexity. An EMD 1.0 file thus represents, stores, and enables access to some arbitrary data in long term storage on a file system in the form of an HDF5 file. The Node class provides machinery for building trees of data and metadata which mirror the EMD tree format but which exist in a live Python instance, rather than on the file system. This facilitates ease of transfer between Python and the file system.

Nodes are intended to be used a base class on which other, more complex classes can be biult. Nodes themselves contain the machinery for managing a tree heirarchy of other Nodes and Metadata instances, and for reading and writing those trees. They do not contain any particular data. Classes storing data and analysis methods which inherit from Node will inherit its tree management and EMD i/o functionality.

Below, the 4 elements of the node class are each described in turn: roots, trees, metadata, and i/o.

ROOTS

EMD data objects can and must exist in one and only one EMD tree, each of which must have a single, named root node. To parallel this in our runtime objects, each Node has a root property, which can be found by calling *self.root*.

By default new nodes have their root set to None. If a node with *.root == None* is saved to file, it is placed inside a new root with the same name as the object itself, and this is then saved to the file as a new (minimal) EMD tree.

A new root node can be instantiated by calling

```
>>> rootnode = Root(name=some_name).
```

Objects added to an existing rooted tree (including a new root node) automatically have their root assigned to the root of that tree. Adding objects to trees is discussed below.

TREES

The tree associated with a node can be manipulated with the .tree method. If we have some rooted node *node1* and some unrooted node *node2*, the unrooted node can be added to the existing tree as a child of the rooted node with

```
>>> node1.tree(node2)
```

If we have a rooted node *node1* and another rooted node *node2*, we can't simply add node2 with the code above, as this would create a conflict between the two roots. In this case, we can move node2 from its current tree to the new tree using

```
>>> node1.tree(graft=node2)
```

The .tree method has various additional functionalities, including printing the tree, retrieving objects from the tree, and cutting branches from the tree. These are summarized below:

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keep root metadata
>>> .tree(graft=(node,True))     # as above
>>> .tree(graft=(node,False))    # as above, discard root metadata
>>> .tree(graft=(node,'copy'))   # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string), i.e. in most cases, the keyword can be dropped. So

```
>>> .tree()
>>> .tree(node)
>>> .tree(True)
>>> .tree('some/node')
```

will, respectively, print the tree from the current node to screen, add the node *node* to the tree, pring the tree from the root node to screen, and return the node at the emdpath 'some/node'.

If a node needs to be added to a tree and it may or may not already have its own root, calling

```
>>> .tree(add=node, force=True)
```

or

```
>>> .tree(node, force=True)
```

will add the node to the tree, using a simple add if node has no root, and grafting it if it does have a root.

METADATA

Nodes can contain any number of Metadata instances, each of which wraps a Python dictionary of some arbitrary complexity (to within the limits of the Metadata group EMD specification, which limits permissible values somewhat).

The code:

```
>>> md1 = Metadata(name='md1')
>>> md2 = Metadata(name='md2')
>>> <<<  some code populating md1 + md2 >>>
>>> node.metadata = md1
>>> node.metadata = md2
```

will create two Metadata objects, populate them with data, then add them to the node. Note that Node.metadata is *not* a Python attribute, it is specially defined property, such that the last line of code does not overwrite the line before it - rather, assigning to the .metadata property adds the new metadata object to a running dictionary of arbitrarily many metadata objects. Both of these two metadata instances can therefore still be retrieved, using:

```
>>> x = node.metadata['md1']
>>> y = node.metadata['md2']
```

Note, however, that if the second metadata instance has an identical name to the first instance, then in *will* overwrite the old instance.

I/O

TODO

__init__(*name: str | None = 'node'*)

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:

node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) this tree's root node

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) the new root node

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this node, tags indicating the groups EMD type and Python class, and any metadata in this node.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new node's Group

PointList

class py4DSTEM.**PointList**(*data: ndarray*, *name: str | None = 'pointlist'*)

A wrapper around structured numpy arrays, with read/write functionality in/out of EMD formatted HDF5 files.

__init__(*data: ndarray*, *name: str | None = 'pointlist'*)

Instantiate a PointList.

Parameters

- **data** (*structured numpy ndarray*) – the data; the dtype of this array will specify the fields of the PointList.
- **name** (*str*) – name for the PointList

Returns
a PointList instance

add(*data*)

Appends a numpy structured array. Its dtypes must agree with the existing data.

remove(*mask*)

Removes points wherever mask==True

sort(*field*, *order='ascending'*)

Sorts the point list according to field, which must be a field in self.dtype. order should be 'descending' or 'ascending'.

copy(*name=None*)

Returns a copy of the PointList. If name=None, sets to *{name}_copy*

add_fields(*new_fields*, *name=''*)

Creates a copy of the PointList, but with additional fields given by new_fields.

Parameters

- **new_fields** – a list of 2-tuples, ('name', dtype)
- **name** – a name for the new pointlist

add_data_by_field(*data*, *fields=None*)

Add a list of data arrays to the PointList, in the fields given by *fields*. If *fields* is not specified, assumes the data arrays are in the same order as self.fields

Parameters
data (*list*) – arrays of data to add to each field

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

`force_add_to_tree`(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

`classmethod from_h5`(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

`get_from_tree`(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

`graft`(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

`static newnode`(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

`show_tree`(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this PointList, tags indicating its EMD type and Python class, and the pointlist's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new pointlist's group

PointListArray

class py4DSTEM.**PointListArray**(*dtype*, *shape*, *name: str | None = 'pointlistarray'*)

An 2D array of PointLists which share common coordinates.

__init__(*dtype*, *shape*, *name: str | None = 'pointlistarray'*)

Creates an empty PointListArray.

Parameters

- **dtype** – the dtype of the numpy structured arrays which will comprise the data of each PointList
- **shape** (*2-tuple of ints*) – the shape of the array of PointLists
- **name** (*str*) – a name for the PointListArray

Returns

a PointListArray instance

get_pointlist(*i*, *j*, *name=None*)

Returns the pointlist at i,j

copy(*name=''*)

Returns a copy of itself.

add_fields(*new_fields*, *name=''*)

Creates a copy of the PointListArray, but with additional fields given by new_fields.

Parameters

- **new_fields** – a list of 2-tuples, ('name', dtype)
- **name** – a name for the new pointlist

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this PointListArray, tags indicating its EMD type and Python class, and the pointlistarray's data and metadata.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new pointlistarray's group

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:

node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))     # as above
>>> .tree(graft=(node,False))    # as above, discard root metadata
>>> .tree(graft=(node,'copy'))   # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

Probe

class py4DSTEM.**Probe**(*data: ndarray*, *name: str | None = 'probe'*)

Stores a vacuum probe.

Both a vacuum probe and a kernel for cross-correlative template matching derived from that probe are stored and can be accessed at

```
>>> p.probe
>>> p.kernel
```

respectively, for some Probe instance *p*. If a kernel has not been computed the latter expression returns None.

__init__(*data: ndarray*, *name: str | None = 'probe'*)

Accepts:

data (2D or 3D np.ndarray): the vacuum probe, or
the vacuum probe + kernel

name (str): a name

Returns
(Probe)

classmethod from_vacuum_data(*data*, *mask=None*, *threshold=0.2*, *expansion=12*, *opening=3*)

Generates and returns a vacuum probe Probe instance from either a 2D vacuum image or a 3D stack of vacuum diffraction patterns.

The probe is multiplied by *mask*, if it's passed. An additional masking step zeros values outside of a mask determined by *threshold*, *expansion*, and *opening*, generated by first computing the binary image probe < max(probe)*threshold, then applying a binary expansion and then opening to this image. No alignment is performed - i.e. it is assumed that the beam was stationary during acquisition of the stack. To align the images, use the DataCube .get_vacuum_probe method.

Parameters

- **data** (*2D or 3D array*) – the vacuum diffraction data. For 3D stacks, use shape (N,Q_Nx,Q_Ny)
- **mask** (*boolean array, optional*) – mask applied to the probe
- **threshold** (*float*) – threshold determining mask which zeros values outside of probe
- **expansion** (*int*) – number of pixels by which the zeroing mask is expanded to capture the full probe
- **opening** (*int*) – size of binary opening used to eliminate stray bright pixels

Returns
probe – the vacuum probe

Return type
Probe

classmethod generate_synthetic_probe(*radius*, *width*, *Qshape*)

Makes a synthetic probe, with the functional form of a disk blurred by a sigmoid (a logistic function).

Parameters

- **radius** (*float*) – the probe radius

- **width** (*float*) – the blurring of the probe edge. width represents the full width of the blur, with x=-w/2 to x=+w/2 about the edge spanning values of ~0.12 to 0.88
- **Qshape** (*2 tuple*) – the diffraction plane dimensions

Returns
probe – the probe

Return type
Probe

measure_disk(*thresh_lower=0.01*, *thresh_upper=0.99*, *N=100*, *returncalc=True*, *data=None*)

Finds the center and radius of an average probe image.

A naive algorithm. Creates a series of N binary masks by thresholding the probe image a linspace of N thresholds from thresh_lower to thresh_upper, relative to the image max/min. For each mask, we find the square root of the number of True valued pixels divided by pi to estimate a radius. Because the central disk is intense relative to the remainder of the image, the computed radii are expected to vary very little over a wider range threshold values. A range of r values considered trustworthy is estimated by taking the derivative r(thresh)/dthresh identifying where it is small, and the mean of this range is returned as the radius. A center is estimated using a binary thresholded image in combination with the center of mass operator.

Parameters

- **thresh_lower** (*float, 0 to 1*) – the lower limit of threshold values
- **thresh_upper** (*float, 0 to 1)*) – the upper limit of threshold values
- **N** (*int*) – the number of thresholds / masks to use
- **returncalc** (*True*) – toggles returning the answer
- **data** (*2d array, optional*) – if passed, uses this 2D array in place of the probe image when performing the computation. This also supresses storing the results in the Probe's calibration metadata

Returns
r, x0, y0 – the radius and origin

Return type
(3-tuple)

get_kernel(*mode='flat'*, *origin=None*, *data=None*, *returncalc=True*, ***kwargs*)

Creates a cross-correlation kernel from the vacuum probe.

Specific behavior and valid keyword arguments depend on the *mode* specified. In each case, the center of the probe is shifted to the origin and the kernel normalized such that it sums to 1. This is the only processing performed if mode is 'flat'. Otherwise, a centrosymmetric region of negative intensity is added around the probe intended to promote edge-filtering-like behavior during cross correlation, with the functional form of the subtracted region defined by *mode* and the relevant **kwargs. For normalization, flat probes integrate to 1, and the remaining probes integrate to 1 before subtraction and 0 after. Required keyword arguments are:

- 'flat': No required arguments. This mode is recommended for bullseye or other structured probes
- 'gaussian': Required arg *sigma* (number), the width (standard deviation) of a centered gaussian to be subtracted.
- 'sigmoid': Required arg *radii* (2-tuple), the inner and outer radii (ri,ro) of an annular region with a sine-squared sigmoidal radial profile to be subtracted.

- 'sigmoid_log': Required arg *radii* (2-tuple), the inner and outer radii (ri,ro) of an annular region with a logistic sigmoidal radial profile to be subtracted.

Parameters

- **mode** (*str*) – must be in 'flat','gaussian','sigmoid','sigmoid_log'
- **origin** (*2-tuple, optional*) – specify the origin. If not passed, looks for a value for the probe origin in metadata. If not found there, calls .measure_disk.
- **data** (*2d array, optional*) – if specified, uses this array instead of the probe image to compute the kernel
- ****kwargs** – see descriptions above

Returns
kernel

Return type
2D array

static get_probe_kernel_flat(*probe*, *origin=None*, *bilinear=False*)

Creates a cross-correlation kernel from the vacuum probe by normalizing and shifting the center.

Parameters

- **probe** (*2d array*) – the vacuum probe
- **origin** (*2-tuple (optional)*) – the origin of diffraction space. If not specified, finds the origin using get_probe_radius.
- **bilinear** (*bool (optional)*) – By default probe is shifted via a Fourier transform. Setting this to True overrides it and uses bilinear shifting. Not recommended!

Returns
kernel – the cross-correlation kernel corresponding to the probe, in real space

Return type
ndarray

static get_probe_kernel_edge_gaussian(*probe*, *sigma*, *origin=None*, *bilinear=True*)

Creates a cross-correlation kernel from the probe, subtracting a gaussian from the normalized probe such that the kernel integrates to zero, then shifting the center of the probe to the array corners.

Parameters

- **probe** (*ndarray*) – the diffraction pattern corresponding to the probe over vacuum
- **sigma** (*float*) – the width of the gaussian to subtract, relative to the standard deviation of the probe
- **origin** (*2-tuple (optional)*) – the origin of diffraction space. If not specified, finds the origin using get_probe_radius.
- **bilinear** (*bool*) – By default probe is shifted via a Fourier transform. Setting this to True overrides it and uses bilinear shifting. Not recommended!

Returns
kernel – the cross-correlation kernel

Return type
ndarray

static get_probe_kernel_edge_sigmoid(*probe*, *radii*, *origin=None*, *type='sine_squared'*, *bilinear=True*)

Creates a convolution kernel from an average probe, subtracting an annular trench about the probe such that the kernel integrates to zero, then shifting the center of the probe to the array corners.

Parameters

- **probe** (*ndarray*) – the diffraction pattern corresponding to the probe over vacuum
- **radii** (*2-tuple*) – the sigmoid inner and outer radii
- **origin** (*2-tuple (optional)*) – the origin of diffraction space. If not specified, finds the origin using get_probe_radius.
- **type** (*string*) – must be 'logistic' or 'sine_squared'
- **bilinear** (*bool*) – By default probe is shifted via a Fourier transform. Setting this to True overrides it and uses bilinear shifting. Not recommended!

Returns

kernel – the cross-correlation kernel

Return type

2d array

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) the new root node

dim(*n*)

Return the n'th dim vector

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:

group (h5py Group)

Returns
(Node)

get_dim(*n*)

Return the n'th dim vector

get_dim_name(*n*)

Get the n'th dim vector name

get_dim_units(*n*)

Return the n'th dim vector units

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

set_dim(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:
n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

set_dim_name(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:
n (int): specifies which dim vector name (str): new name

set_dim_units(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:

n (int): specifies which dim vector units (str): new units

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new array's Group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

QPoints

class py4DSTEM.**QPoints**(*data: ndarray*, *name: str | None = 'qpoints'*)

Stores a set of diffraction space points, with fields 'qx', 'qy' and 'intensity'

__init__(*data: ndarray*, *name: str | None = 'qpoints'*)

Accepts:

data (structured numpy ndarray): should have three fields, which will be renamed 'qx','qy','intensity'

name (str): the name of the QPoints instance

Returns
A new QPoints instance

`add`(*data*)

Appends a numpy structured array. Its dtypes must agree with the existing data.

`add_data_by_field`(*data*, *fields=None*)

Add a list of data arrays to the PointList, in the fields given by *fields*. If *fields* is not specified, assumes the data arrays are in the same order as self.fields

Parameters
data (*list*) – arrays of data to add to each field

`add_fields`(*new_fields*, *name=''*)

Creates a copy of the PointList, but with additional fields given by new_fields.

Parameters

- **new_fields** – a list of 2-tuples, ('name', dtype)
- **name** – a name for the new pointlist

`add_to_tree`(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

`attach`(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

`copy`(*name=None*)

Returns a copy of the PointList. If name=None, sets to *{name}_copy*

`cut_from_tree`(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

`force_add_to_tree`(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

`classmethod from_h5`(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:

node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

remove(*mask*)

Removes points wherever mask==True

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

sort(*field*, *order='ascending'*)

Sorts the point list according to field, which must be a field in self.dtype. order should be 'descending' or 'ascending'.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this PointList, tags indicating its EMD type and Python class, and the pointlist's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new pointlist's group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))     # as above
>>> .tree(graft=(node,False))    # as above, discard root metadata
>>> .tree(graft=(node,'copy'))   # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

RealSlice

class py4DSTEM.**RealSlice**(*data: ndarray*, *name: str | None = 'realslice'*, *units: str | None = 'intensity'*, *slicelabels: bool | list | None = None*, *calibration=None*)

Stores a real-space shaped 2D data array.

__init__(*data: ndarray*, *name: str | None = 'realslice'*, *units: str | None = 'intensity'*, *slicelabels: bool | list | None = None*, *calibration=None*)

Accepts:

data (np.ndarray): the data name (str): the name of the realslice slicelabels(None or list): names for slices if this is a stack of

realslices

Returns

A new RealSlice instance

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

dim(*n*)

Return the n'th dim vector

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_dim(*n*)

Return the n'th dim vector

get_dim_name(*n*)

Get the n'th dim vector name

get_dim_units(*n*)

Return the n'th dim vector units

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

set_dim(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:

n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

set_dim_name(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:

n (int): specifies which dim vector name (str): new name

set_dim_units(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:

n (int): specifies which dim vector units (str): new units

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new array's Group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()             # show tree from current node
>>> .tree(show=True)    # show from root
>>> .tree(show=False)   # show from current node
>>> .tree(add=node)     # add a child node
>>> .tree(get='path')   # return a '/' delimited child node
>>> .tree(get='/path')  # as above, starting at root
>>> .tree(cut=True)     # remove/return a branch, keep root metadata
>>> .tree(cut=False)    # remove/return a branch, discard root md
>>> .tree(cut='copy')   # remove/return a branch, copy root metadata
>>> .tree(graft=node)   # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

VirtualDiffraction

class py4DSTEM.**VirtualDiffraction**(*data: ndarray*, *name: str | None = 'virtualdiffraction'*)

Stores a diffraction-space shaped 2D image with metadata indicating how this image was generated from a self.

__init__(*data: ndarray*, *name: str | None = 'virtualdiffraction'*)

Parameters

- **data** (*np.ndarray*) – the 2D data
- **name** (*str*) – the name

Returns
A new VirtualDiffraction instance

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

dim(*n*)

Return the n'th dim vector

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_dim(*n*)

Return the n'th dim vector

get_dim_name(*n*)

Get the n'th dim vector name

get_dim_units(*n*)

Return the n'th dim vector units

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

set_dim(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:
n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

set_dim_name(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:
n (int): specifies which dim vector name (str): new name

set_dim_units(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:

n (int): specifies which dim vector units (str): new units

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new array's Group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

VirtualImage

class py4DSTEM.**VirtualImage**(*data: ndarray*, *name: str | None = 'virtualimage'*)

A container for storing virtual image data and metadata, including the real-space shaped 2D image and metadata indicating how this image was generated from a datacube.

__init__(*data: ndarray*, *name: str | None = 'virtualimage'*)

Parameters

- **data** (*np.ndarray*) – the 2D data
- **name** (*str*) – the name

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

attach(*node*)

Attach *node* to the current object's tree, attaching calibration and detaching calibrations as needed.

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's
metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) the new root node

dim(*n*)

Return the n'th dim vector

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

get_dim(*n*)

Return the n'th dim vector

get_dim_name(*n*)

Get the n'th dim vector name

get_dim_units(*n*)

Return the n'th dim vector units

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:
node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns
(Node) this tree's root node

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

set_dim(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:
n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

set_dim_name(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:
n (int): specifies which dim vector name (str): new name

set_dim_units(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:
n (int): specifies which dim vector units (str): new units

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new array's Group

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

1.4.3 io

Table of Contents

- *io*
  - *filereaders*
  - *google_drive_downloader*
  - *importfile*
  - *legacy*
  - *parsefiletype*

filereaders

py4DSTEM.io.filereaders.empad.**read_empad**(*filename*, *mem='RAM'*, *binfactor=1*, *metadata=False*, ***kwargs*)

Reads the EMPAD file at filename, returning a DataCube.

EMPAD files are shaped as 130x128 arrays, consisting of 128x128 arrays of data followed by two rows of metadata. For each frame, its position in the scan is embedded in the metadata. By extracting the scan position of the first and last frames, the function determines the scan size. Then, the full dataset is loaded and cropped to the 128x128 valid region.

Accepts:

filename (str) path to the EMPAD file EMPAD_shape (kwarg, tuple) Manually specify the shape of the data for files that do not

contain metadata in the .raw file. This will typically be:
(# scan pixels x, # scan pixels y, 130, 128)

Returns
data (DataCube) the 4D datacube, excluding the metadata rows.

py4DSTEM.io.filereaders.read_K2.**read_gatan_K2_bin**(*fp*, *mem='MEMMAP'*, *binfactor=1*, *metadata=False*, ***kwargs*)

Read a K2 binary 4D-STEM file.

Parameters

- **fp** – str Path to the file
- **mem** (*str, optional*) – Specifies how the data should be stored; must be "RAM" or "MEMMAP". See docstring for py4DSTEM.file.io.read. Default is "MEMMAP".
- **binfactor** – (int, optional): Bin the data, in diffraction space, as it's loaded. See docstring for py4DSTEM.file.io.read. Must be 1, retained only for compatibility.
- **metadata** (*bool, optional*) – if True, returns the file metadata as a Metadata instance.

Returns

The return value depends on usage:

- if metadata==False, returns the 4D-STEM dataset as a DataCube
- if metadata==True, returns the metadata as a Metadata instance

Note that metadata is read either way - in the latter case ONLY metadata is read and returned, in the former case a DataCube is returned with the metadata attached at datacube.metadata

Return type
(variable)

class py4DSTEM.io.filereaders.read_K2.**K2DataArray**(*filepath*, *sync_block_IDs=True*, *hidden_stripe_noise_reduction=True*)

K2DataArray provides an interface to a set of Gatan K2IS binary output files. This object behaves *similar* to a numpy memmap into the data, and supports 4-D indexing and slicing. Slices into this object return np.ndarray objects.

The object is created by passing the path to any of: (i) the folder containing the raw data, (ii) the *.gtg metadata file, or (iii) one of the raw data *.bin files. In any case, there should be only one dataset (8 *.bin's and a *.gtg) in the folder.

===== Filtering and Noise Reduction ===== This object is read-only—you cannot edit the data on disk, which means that some DataCube functions like swap_RQ() will not work.

The K2IS has a "resolution" of 1920x1792, but actually saves hidden stripes in the raw data. By setting the hidden_stripe_noise_reduction flag to True, the electronic noise in these stripes is used to reduce the readout noise. (This is on by default.)

If you want to take a separate background to subtract, set *dark_reference* to specify this background. This is then subtracted from the frames as they are called out (no matter where the object is referenced! So, for instance, Bragg disk detection will operate on the background- subtracted diffraction patterns!). However, mixing the auto-background and specified background is potentially dangerous and (currently!) not allowed. To switch back from user-background to auto-background, just delete the user background, i.e. *del(dc.data4D.dark_reference)*

Note: If you call dc.data4D[:,:,:,:] on a DataCube with a K2DataArray this will read the entire stack into memory. To reduce RAM pressure, only call small slices or loop over each diffraction pattern.

__init__(*filepath*, *sync_block_IDs=True*, *hidden_stripe_noise_reduction=True*)

py4DSTEM.io.filereaders.read_mib.**load_mib**(*file_path*, *mem='MEMMAP'*, *binfactor=1*, *reshape=True*, *flip=True*, *scan=(256, 256)*, ***kwargs*)

Read a MIB file and return as py4DSTEM DataCube.

The scan size is not encoded in the MIB metadata - by default it is set to (256,256), and can be modified by passing the keyword *scan*.

py4DSTEM.io.filereaders.read_mib.**manageHeader**(*fname*)

Get necessary information from the header of the .mib file. :param fname: Filename for header file. :type fname: str

Returns

hdr – (DataOffset,NChips,PixelDepthInFile,sensorLayout,Timestamp,shuttertime,bitdepth)

Return type

tuple

Examples

#Output for 6bit 256*256 data: #(768, 4, 'R64', '2x2', '2019-06-14 11:46:12.607836', 0.0002, 6) #Output for 12bit single frame nor RAW: #(768, 4, 'U16', '2x2', '2019-06-06 11:12:42.001309', 0.001, 12)

py4DSTEM.io.filereaders.read_mib.**parse_hdr**(*fp*)

Parse information from mib file header info from _manageHeader function. :param fp: Filepath to .mib file. :type fp: str

Returns

hdr_info – Dictionary containing header info extracted from .mib file. The entries of the dictionary are as follows: 'width': int

pixels, detector number of pixels in x direction,

'height': int
pixels detector number of pixels in y direction,

'Assembly Size': str
configuration of the detector chips, e.g. '2x2' for quad,

'offset': int
number of characters in the header before the first frame starts,

'data-type': str
always 'unsigned',

'data-length': str
identifying dtype,

'Counter Depth (number)': int
counter bit depth,

'raw': str
regular binary 'MIB' or raw binary 'R64',

'byte-order': str
always 'dont-care',

'record-by': str
'image' or 'vector' - only 'image' encountered,

'title': str
path of the mib file without extension, e.g. '/dls/e02/data/2020/cm26481-1/Merlin/testing/20200204 115306/test',

'date': str
date created, e.g. '20200204',

'time': str
time created, e.g. '11:53:32.295336',

'data offset': int
number of characters at the header.

Return type
dict

py4DSTEM.io.filereaders.read_mib.**get_mib_memmap**(*fp*, *mmap_mode='r'*)

Reads the binary mib file into a numpy memmap object and returns as dask array object. :param fp: MIB file name / path :type fp: str :param mmap_mode: memmpap read mode - default is 'r' :type mmap_mode: str

Returns
data_da – data as a dask array object

Return type
dask array

py4DSTEM.io.filereaders.read_mib.**get_mib_depth**(*hdr_info*, *fp*)

Determine the total number of frames based on .mib file size. :param hdr_info: Dictionary containing header info extracted from .mib file. :type hdr_info: dict :param fp: Path to .mib file. :type fp: filepath

Returns
depth – Number of frames in the stack

Return type
int

py4DSTEM.io.filereaders.read_mib.**get_hdr_bits**(*hdr_info*)

Gets the number of character bits for the header for each frame given the data type. :param hdr_info: output of the parse_hdr function :type hdr_info: dict

Returns
hdr_bits – number of characters in the header

Return type
int

google_drive_downloader

py4DSTEM.io.google_drive_downloader.**gdrive_download**(*id_*, *destination=None*, *overwrite=False*, *filename=None*, *verbose=True*)

Downloads a file or collection of files from google drive.

Parameters

- **id** (*str*) – File ID for the desired file. May be either a key from the list of files and collections of files accessible at get_sample_file_ids(), or a complete url, or the portions of a google drive link specifying it's google file ID, i.e. for the address https://drive.google.com/file/d/1bHv3u61Cr-y_GkdWHrJGh1lw2VKmt3UM/, the id string '1bHv3u61Cr-y_GkdWHrJGh1lw2VKmt3UM'.
- **destination** (*None or str*) – The location files are downloaded to. If a collection of files has been specified, creates a new directory at the specified destination and downloads the collection there. If None, downloads to the current working directory. Otherwise must be a string or Path pointint to a valid location on the filesystem.
- **overwrite** (*bool*) – Turns overwrite protection on/off.
- **filename** (*None or str*) – Used only if *id_* is a url or gdrive id. In these cases, specifies the name of the output file. If left as None, saves to 'gdrivedownload.file'. If *id_* is a key from the sample file id list, this parameter is ignored.
- **verbose** (*bool*) – Toggles verbose output

importfile

py4DSTEM.io.importfile.**import_file**(*filepath: str | Path*, *mem: str | None = 'RAM'*, *binfactor: int | None = 1*, *filetype: str | None = None*, ***kwargs*)

Reader for non-native file formats. Parses the filetype, and calls the appropriate reader. Supports Gatan DM3/4, some EMPAD file versions, Gatan K2 bin/gtg, and mib formats.

Parameters

- **filepath** (*str or Path*) – Path to the file.
- **mem** (*str*) – Must be "RAM" or "MEMMAP". Specifies how the data is loaded; "RAM" transfer the data from storage to RAM, while "MEMMAP" leaves the data in storage and creates a memory map which points to the diffraction patterns, allowing them to be retrieved individually from storage.
- **binfactor** (*int*) – Diffraction space binning factor for bin-on-load.
- **filetype** (*str*) – Used to override automatic filetype detection. options include "dm", "empad", "gatan_K2_bin", "mib", "arina", "abTEM"
- ****kwargs** – any additional kwargs are passed to the downstream reader - refer to the individual filetype reader function call signatures and docstrings for more details.

Returns

(DataCube or Array) returns a DataCube if 4D data is found, otherwise returns an Array

legacy

This is the h5py package, a Python interface to the HDF5 scientific data format.

py4DSTEM.io.legacy.read_legacy_12.**read_legacy12**(*filepath*, ***kwargs*)

File reader for older legacy py4DSTEM (v<0.13) formated HDF5 files.

Different file versions Precise behavior is detemined by which arguments are passed – see below.

Parameters

- **filepath** (*str or pathlib.Path*) – When passed a filepath only, this function checks if the path points to a valid py4DSTEM file, then prints its contents to screen.
- **data_id** (*int/str/list, optional*) – Specifies which data to load. Use integers to specify the data index, or strings to specify data names. A list or tuple returns a list of DataObjects. Returns the specified data.
- **topgroup** (*str, optional*) – Stricty, a py4DSTEM file is considered to be everything inside a toplevel subdirectory within the HDF5 file, so that if desired one can place many py4DSTEM files inside a single H5. In this case, when loading data, the topgroup argument is passed to indicate which py4DSTEM file to load. If an H5 containing multiple py4DSTEM files is passed without a topgroup specified, the topgroup names are printed to screen.
- **mem** (*str, optional*) – Only used if a single DataCube is loaded. In this case, mem specifies how the data should be stored; must be "RAM" or "MEMMAP". See docstring for py4DSTEM.file.io.read. Default is "RAM".
- **binfactor** (*int, optional*) – Only used if a single DataCube is loaded. In this case, a binfactor of > 1 causes the data to be binned by this amount as it's loaded.
- **dtype** (*dtype, optional*) – Used when binning data, ignored otherwise. Defaults to whatever the type of the raw data is, to avoid enlarging data size. May be useful to avoid 'wraparound' errors.

Returns

The output depends on usage:

- If no input arguments with return values (i.e. data_id or metadata) are passed, nothing is returned.
- Otherwise, a single DataObject or list of DataObjects are returned, based on the value of the argument data_id.

Return type
(variable)

py4DSTEM.io.legacy.read_legacy_13.**read_legacy13**(*filepath*, *root: str | None = None*, *tree: bool | str | None = True*)

File reader for legacy py4DSTEM (v=0.13.x) formated HDF5 files.

Parameters

- **filepath** (*str or Path*) – the file path
- **root** (*str*) – the path to the data group in the HDF5 file to read from. To examine an HDF5 file written by py4DSTEM in order to determine this path, call *py4DSTEM.print_h5_tree(filepath)*. If left unspecified, looks in the file and if it finds a single top-level object, loads it. If it finds multiple top-level objects, prints a warning and returns a list of root paths to the top-level object found.

- **tree** (*bool or str*) – indicates what data should be loaded, relative to the root group specified above. Must be in (*True* or *False* or *noroot*). If set to *False*, the only the data in the root group is loaded, plus any associated calibrations. If set to *True*, loads the root group, and all other data groups nested underneath it in the file tree. If set to *'noroot'*, loads all other data groups nested under the root group in the file tree, but does *not* load the data inside the root group (allowing, e.g., loading all the data nested under a DataCube13 without loading the whole datacube).

Returns
(the data)

py4DSTEM.io.legacy.read_legacy_13.**print_v13h5_tree**(*filepath*, *show_metadata=False*)

Prints the contents of an h5 file from a filepath.

py4DSTEM.io.legacy.read_legacy_13.**print_v13h5pyFile_tree**(*f*, *tablevel=0*, *linelevels=[]*, *show_metadata=False*)

Prints the contents of an h5 file from an open h5py File instance.

py4DSTEM.io.legacy.read_utils.**get_py4DSTEM_topgroups**(*filepath*)

Returns a list of toplevel groups in an HDF5 file which are valid py4DSTEM file trees.

py4DSTEM.io.legacy.read_utils.**is_py4DSTEM_version13**(*filepath*)

Returns True for data written by a py4DSTEM v0.13.x release.

py4DSTEM.io.legacy.read_utils.**is_py4DSTEM_file**(*filepath*)

Returns True iff filepath points to a py4DSTEM formatted (EMD type 2) file.

py4DSTEM.io.legacy.read_utils.**get_py4DSTEM_version**(*filepath*, *topgroup='4DSTEM_experiment'*)

Returns the version (major,minor,release) of a py4DSTEM file.

py4DSTEM.io.legacy.read_utils.**get_UUID**(*filepath*, *topgroup='4DSTEM_experiment'*)

Returns the UUID of a py4DSTEM file, or if unavailable returns -1.

py4DSTEM.io.legacy.read_utils.**version_is_geq**(*current*, *minimum*)

Returns True iff current version (major,minor,release) is greater than or equal to minimum."

py4DSTEM.io.legacy.read_utils.**get_N_dataobjects**(*filepath*, *topgroup='4DSTEM_experiment'*)

Returns a 7-tuple of ints with the numbers of: DataCubes, CountedDataCubes, DiffractionSlices, RealSlices, PointLists, PointListArrays, total DataObjects.

parsefiletype

1.4.4 preprocess

Table of Contents

- *preprocess*
  - *darkreference*
  - *electroncount*
  - *preprocess*
  - *radialbkgrd*
  - *utils*

darkreference

py4DSTEM.preprocess.darkreference.**get_bksbtr_DP**(*datacube*, *darkref*, *Rx*, *Ry*)

Returns a background subtracted diffraction pattern.

Parameters

- **datacube** (DataCube) – data to background subtract
- **darkref** (*ndarray*) – dark reference. must have shape (datacube.Q_Nx, datacube.Q_Ny)
- **Rx** (*int*) – the scan position of the diffraction pattern of interest
- **Ry** (*int*) – the scan position of the diffraction pattern of interest

Returns

(ndarray) the background subtracted diffraction pattern

py4DSTEM.preprocess.darkreference.**get_darkreference**(*datacube*, *N_frames*, *width_x=0*, *width_y=0*, *side_x='end'*, *side_y='end'*)

Gets a dark reference image.

Select N_frames random frames (DPs) from datacube. Find streaking noise in the horizontal and vertical directions, by finding the average values along a thin strip of width_x/width_y pixels along the detector edges. Which edges are used is controlled by side_x/side_y, which must be 'start' or 'end'. Streaks along only one direction can be used by setting width_x or width_y to 0, which disables correcting streaks in this direction.

Note that the data is cast to float before computing the background, and should similarly be cast to float before performing a subtraction. This avoids integer clipping and wraparound errors.

Parameters

- **datacube** (DataCube) – data to background subtract
- **N_frames** (*int*) – number of random diffraction patterns to use
- **width_x** (*int*) – width of the ROI strip for finding streaking in x
- **width_y** (*int*) – see above
- **side_x** (*str*) – use a strip from the start or end of the array. Must be 'start' or 'end', defaults to 'end'
- **side_y** (*str*) – see above

Returns

a 2D ndarray of shape (datacube.Q_Nx, datacube.Ny) giving the background.

Return type

(ndarray)

py4DSTEM.preprocess.darkreference.**get_background_streaks**(*datacube*, *N_frames*, *width*, *side='end'*, *direction='x'*)

Gets background streaking in either the x- or y-direction, by finding the average of a strip of pixels along the edge of the detector over a random selection of diffraction patterns, and returns a dark reference array.

Note that the data is cast to float before computing the background, and should similarly be cast to float before performing a subtraction. This avoids integer clipping and wraparound errors.

Parameters

- **datacube** (DataCube) – data to background subtract
- **N_frames** (*int*) – number of random frames to use

- **width** (*int*) – width of the ROI strip for background identification
- **side** (*str, optional*) – use a strip from the start or end of the array. Must be 'start' or 'end', defaults to 'end'
- **directions** (*str*) – the direction of background streaks to find. Must be either 'x' or 'y' defaults to 'x'

Returns
a 2D ndarray of shape (datacube.Q_Nx,datacube.Q_Ny), giving the the x- or y-direction background streaking.

Return type
(ndarray)

py4DSTEM.preprocess.darkreference.**get_background_streaks_x**(*datacube*, *width*, *N_frames*, *side='start'*)

Gets background streaking, by finding the average of a strip of pixels along the y-edge of the detector over a random selection of diffraction patterns.

See docstring for get_background_streaks() for more info.

py4DSTEM.preprocess.darkreference.**get_background_streaks_y**(*datacube*, *N_frames*, *width*, *side='start'*)

Gets background streaking, by finding the average of a strip of pixels along the x-edge of the detector over a random selection of diffraction patterns.

See docstring for get_background_streaks_1D() for more info.

electroncount

py4DSTEM.preprocess.electroncount.**electron_count**(*datacube*, *darkreference*, *Nsamples=40*, *thresh_bkgrnd_Nsigma=4*, *thresh_xray_Nsigma=10*, *binfactor=1*, *sub_pixel=True*, *output='pointlist'*)

Performs electron counting.

The algorithm is as follows: From a random sampling of frames, calculate an x-ray and background threshold value. In each frame, subtract the dark reference, then apply the two thresholds. Find all local maxima with respect to the nearest neighbor pixels. These are considered electron strike events.

Thresholds are specified in units of standard deviations, either of a gaussian fit to the histogram background noise (for thresh_bkgrnd) or of the histogram itself (for thresh_xray). The background (lower) threshold is more important; we will always be missing some real electron counts and incorrectly counting some noise as electron strikes - this threshold controls their relative balance. The x-ray threshold may be set fairly high.

Parameters

- **datacube** – a 4D numpy.ndarray pointing to the datacube. Note: the R/Q axes are flipped with respect to py4DSTEM DataCube objects
- **darkreference** – a 2D numpy.ndarray with the dark reference
- **Nsamples** – the number of frames to use in dark reference and threshold calculation.
- **thresh_bkgrnd_Nsigma** – the background threshold is `mean(guassian fit) + (this #)*std(gaussian fit)` where the gaussian fit is to the background noise.
- **thresh_xray_Nsigma** – the X-ray threshold is `mean(hist) +/- (this #)*std(hist)` where hist is the histogram of all pixel values in the Nsamples random frames

- **binfactor** – the binnning factor
- **sub_pixel** (*bool*) – controls whether subpixel refinement is performed
- **output** (*str*) – controls output format; must be 'datacube' or 'pointlist'

Returns

(variable) if output=='pointlist', returns a PointListArray of all electron counts in each frame. If output=='datacube', returns a 4D array of bools, with True indicating electron strikes

py4DSTEM.preprocess.electroncount.**electron_count_GPU**(*datacube*, *darkreference*, *Nsamples=40*, *thresh_bkgrnd_Nsigma=4*, *thresh_xray_Nsigma=10*, *binfactor=1*, *sub_pixel=True*, *output='pointlist'*)

Performs electron counting on the GPU.

Uses pytorch to interface between numpy and cuda. Requires cuda and pytorch. This function expects datacube to be a np.memmap object. See electron_count() for additional documentation.

py4DSTEM.preprocess.electroncount.**calculate_thresholds**(*datacube*, *darkreference*, *Nsamples=20*, *thresh_bkgrnd_Nsigma=4*, *thresh_xray_Nsigma=10*, *return_params=False*)

Calculate the upper and lower thresholds for thresholding what to register as an electron count.

Both thresholds are determined from the histogram of detector pixel values summed over Nsamples frames. The thresholds are set to:

```
thresh_xray_Nsigma =    mean(histogram)    + thresh_upper * std(histogram)
thresh_bkgrnd_N_sigma = mean(guassian fit) + thresh_lower * std(gaussian fit)
```

For more info, see the electron_count docstring.

Parameters

- **datacube** – a 4D numpy.ndarrau pointing to the datacube
- **darkreference** – a 2D numpy.ndarray with the dark reference
- **Nsamples** – the number of frames to use in dark reference and threshold calculation.
- **thresh_bkgrnd_Nsigma** – the background threshold is `mean(guassian fit) + (this #)*std(gaussian fit)` where the gaussian fit is to the background noise.
- **thresh_xray_Nsigma** – the X-ray threshold is `mean(hist) + (this #)*std(hist)` where hist is the histogram of all pixel values in the Nsamples random frames
- **return_params** – bool, if True return n,hist of the histogram and popt of the gaussian fit

Returns

A 5-tuple containing:

- **thresh_bkgrnd**: the background threshold
- **thresh_xray**: the X-ray threshold
- **n**: returned iff return_params==True. The histogram values
- **hist**: returned iff return_params==True. The histogram bin edges
- **popt**: returned iff return_params==True. The fit gaussian parameters, (A, mu, sigma).

Return type

(5-tuple)

py4DSTEM.preprocess.electroncount.**torch_bin**(*array*, *device*, *factor=2*)

Bin data on the GPU using torch.

Parameters

- **array** – a 2D numpy array
- **device** – a torch device class instance
- **factor** (*int*) – the binning factor

Returns
the binned array

Return type
(array)

py4DSTEM.preprocess.electroncount.**counted_datacube_to_pointlistarray**(*counted_datacube*, *subpixel=False*)

Converts an electron counted datacube to PointListArray.

Parameters

- **counted_datacube** – a 4D array of bools, with true indicating an electron strike.
- **subpixel** (*bool*) – controls if subpixel electron strike positions are expected

Returns
a PointListArray of electron strike events

Return type
(*PointListArray*)

py4DSTEM.preprocess.electroncount.**counted_pointlistarray_to_datacube**(*counted_pointlistarray*, *shape*, *subpixel=False*)

Converts an electron counted PointListArray to a datacube.

Parameters

- **counted_pointlistarray** (PointListArray) – a PointListArray of electron strike events
- **shape** (*4-tuple*) – a length 4 tuple of ints containing (R_Nx,R_Ny,Q_Nx,Q_Ny)
- **subpixel** (*bool*) – controls if subpixel electron strike positions are expected

Returns
a 4D array of bools, with true indicating an electron strike.

Return type
(4D array of bools)

preprocess

py4DSTEM.preprocess.preprocess.**set_scan_shape**(*datacube*, *R_Nx*, *R_Ny*)

Reshape the data given the real space scan shape.

py4DSTEM.preprocess.preprocess.**swap_RQ**(*datacube*)

Swaps real and reciprocal space coordinates, so that if

```
>>> datacube.data.shape
(Rx,Ry,Qx,Qy)
```

Then

```
>>> swap_RQ(datacube).data.shape
(Qx,Qy,Rx,Ry)
```

py4DSTEM.preprocess.preprocess.**swap_Rxy**(*datacube*)

Swaps real space x and y coordinates, so that if

```
>>> datacube.data.shape
(Ry,Rx,Qx,Qy)
```

Then

```
>>> swap_Rxy(datacube).data.shape
(Rx,Ry,Qx,Qy)
```

py4DSTEM.preprocess.preprocess.**swap_Qxy**(*datacube*)

Swaps reciprocal space x and y coordinates, so that if

```
>>> datacube.data.shape
(Rx,Ry,Qy,Qx)
```

Then

```
>>> swap_Qxy(datacube).data.shape
(Rx,Ry,Qx,Qy)
```

py4DSTEM.preprocess.preprocess.**bin_data_diffraction**(*datacube*, *bin_factor*, *dtype=None*)

Performs diffraction space binning of data by bin_factor.

Parameters

- **N** (*int*) – The binning factor
- **dtype** (*a datatype (optional)*) – Specify the datatype for the output. If not passed, the datatype is left unchanged

py4DSTEM.preprocess.preprocess.**bin_data_mmap**(*datacube*, *bin_factor*, *dtype=<class 'numpy.float32'>*)

Performs diffraction space binning of data by bin_factor.

py4DSTEM.preprocess.preprocess.**bin_data_real**(*datacube*, *bin_factor*)

Performs diffraction space binning of data by bin_factor.

py4DSTEM.preprocess.preprocess.**thin_data_real**(*datacube*, *thinning_factor*)

Reduces data size by a factor of *thinning_factor^2 by skipping every `thinning_factor* beam positions in both x and y.

py4DSTEM.preprocess.preprocess.**filter_hot_pixels**(*datacube*, *thresh*, *ind_compare=1*, *return_mask=False*)

This function performs pixel filtering to remove hot / bright pixels. A mean diffraction pattern is calculated, then a moving local ordering filter is applied to it, finding and sorting the intensities of the 21 pixels nearest each pixel (where 21 = (the pixel itself) + (nearest neighbors) + (next nearest neighbors) = (1) + (8) + (12) = 21; the next nearest neighbors exclude the corners of the NNN square of pixels). This filter then returns a single value at each pixel given by the N'th highest value of these 21 sorted values, where N is specified by *ind_compare*. ind_compare=0 specifies the highest intensity, =1 is the second hightest, etc. Next, a mask is generated which is True for all pixels which are least a value *thresh* higher than the local ordering filter output. Thus for the default *ind_compare* value of 1, the mask will be True wherever the mean diffraction pattern is higher than the second

brightest pixel in it's local window by at least a value of *thresh*. Finally, we loop through all diffraction images, and any pixels defined by mask are replaced by their 3x3 local median.

Parameters

- **datacube** (DataCube) – The 4D atacube
- **thresh** (*float*) – Threshold for replacing hot pixels, if pixel value minus local ordering filter exceeds it.
- **ind_compare** (*int*) – Which median filter value to compare against. 0 = brightest pixel, 1 = next brightest, etc.
- **return_mask** (*bool*) – If True, returns the filter mask

Returns

- **datacube** (*Datacube*)
- **mask** (*bool*) – (optional) the bad pixel mask

py4DSTEM.preprocess.preprocess.**median_filter_masked_pixels**(*datacube*, *mask*, *kernel_width: int = 3*)

This function fixes a datacube where the same pixels are consistently bad. It requires a mask that identifies all the bad pixels in the dataset. Then for each diffraction pattern, a median kernel is applied around each bad pixel with the specified width.

Parameters

- **datacube** – Datacube to be filtered
- **mask** – a boolean mask that specifies the bad pixels in the datacube
- **(optional)** (*kernel_width*) – specifies the width of the median kernel

Return type
filtered datacube

py4DSTEM.preprocess.preprocess.**datacube_diffraction_shift**(*datacube*, *xshifts*, *yshifts*, *periodic=True*, *bilinear=False*)

This function shifts each 2D diffraction image by the values defined by (xshifts,yshifts). The shift values can be scalars (same shift for all images) or arrays with the same dimensions as the probe positions in datacube.

Parameters

- **datacube** (DataCube) – py4DSTEM DataCube
- **xshifts** (*float*) – Array or scalar value for the x dim shifts
- **yshifts** (*float*) – Array or scalar value for the y dim shifts
- **periodic** (*bool*) – Flag for periodic boundary conditions. If set to false, boundaries are assumed to be periodic.
- **bilinear** – Flag for bilinear image shifts. If set to False, Fourier shifting is used.

py4DSTEM.preprocess.preprocess.**resample_data_diffraction**(*datacube*, *resampling_factor=None*, *output_size=None*, *method='bilinear'*, *conserve_array_sums=False*)

Performs diffraction space resampling of data by resampling_factor or to match output_size.

py4DSTEM.preprocess.preprocess.**pad_data_diffraction**(*datacube*, *pad_factor=None*, *output_size=None*)

Performs diffraction space padding of data by pad_factor or to match output_size.

radialbkgrd

Functions for generating radially averaged backgrounds

py4DSTEM.preprocess.radialbkgrd.**get_1D_polar_background**(*data*, *p_ellipse*, *center=None*, *maskUpdateIter=3*, *min_relative_threshold=4*, *smoothing=False*, *smoothingWindowSize=3*, *smoothingPolyOrder=4*, *smoothing_log=True*, *min_background_value=0.001*, *return_polararr=False*)

Gets the median polar background for a diffraction pattern

Parameters

- **data** (*ndarray*) – the data for which to find the polar eliptical background, usually a diffraction pattern
- **p_ellipse** (*5-tuple*) – the ellipse parameters (qx0,qy0,a,b,theta)
- **center** (*2-tuple or None*) – if None, the center point from *p_ellipse* is used. Otherwise, the center point in *p_ellipse* is ignored, and this argument is used as (qx0,qy0) instead.
- **maskUpdate_iter** (*integer*) –
- **min_relative_threshold** (*float*) –
- **smoothing** (*bool*) – if true, applies a Savitzky-Golay smoothing filter
- **smoothingWindowSize** (*integer*) – size of the smoothing window, must be odd number
- **smoothingPolyOrder** (*number*) – order of the polynomial smoothing to be applied
- **smoothing_log** (*bool*) – if true log smoothing is performed
- **min_background_value** (*float*) – if log smoothing is true, a zero value will be replaced with a small nonzero float
- **return_polar_arr** (*bool*) – if True the polar transform with the masked high intensity peaks will be returned

Returns

- **background1D**: 1D polar elliptical background
- **r_bins**: the elliptically transformed radius associated with background1D
- **polarData** (optional): the masked polar transform from which the background is computed, returned iff *return_polar_arr==True*

Return type
2- or 3-tuple of ndarrays

py4DSTEM.preprocess.radialbkgrd.**get_2D_polar_background**(*data*, *background1D*, *r_bins*, *p_ellipse*, *center=None*)

Gets 2D polar elliptical background from linear 1D background

Parameters

- **data** (*ndarray*) – the data for which to find the polar eliptical background, usually a diffraction pattern

- **background1D** (*ndarray*) – a vector representing the radial elliptical background
- **r_bins** (*ndarray*) – a vector of the elliptically transformed radius associated with background1D
- **p_ellipse** (*5-tuple*) – the ellipse parameters (qx0,qy0,a,b,theta)
- **center** (*2-tuple or None*) – if None, the center point from *p_ellipse* is used. Otherwise, the center point in *p_ellipse* is ignored, and this argument is used as (qx0,qy0) instead.

Returns
2D polar elliptical median background image

Return type
ndarray

utils

py4DSTEM.preprocess.utils.**bin2D**(*array*, *factor*, *dtype=<class 'numpy.float64'>*)

Bin a 2D ndarray by binfactor.

Parameters

- **array** (*2D numpy array*) –
- **factor** (*int*) – the binning factor
- **dtype** (*numpy dtype*) – datatype for binned array. default is numpy default for np.zeros()

Returns
the binned array

py4DSTEM.preprocess.utils.**make_Fourier_coords2D**(*Nx*, *Ny*, *pixelSize=1*)

Generates Fourier coordinates for a (Nx,Ny)-shaped 2D array.
Specifying the pixelSize argument sets a unit size.

py4DSTEM.preprocess.utils.**get_shifted_ar**(*ar*, *xshift*, *yshift*, *periodic=True*, *bilinear=False*, *device='cpu'*)

Shifts array ar by the shift vector (xshift,yshift), using the either

the Fourier shift theorem (i.e. with sinc interpolation), or bilinear resampling. Boundary conditions can be periodic or not.

Parameters

- **ar** (*float*) – input array
- **xshift** (*float*) – shift along axis 0 (x) in pixels
- **yshift** (*float*) – shift along axis 1 (y) in pixels
- **periodic** (*bool*) – flag for periodic boundary conditions
- **bilinear** (*bool*) – flag for bilinear image shifts
- **device** – calculation device will be perfomed on. Must be 'cpu' or 'gpu'

py4DSTEM.preprocess.utils.**get_maxima_2D**(*ar*, *subpixel='poly'*, *upsample_factor=16*, *sigma=0*, *minAbsoluteIntensity=0*, *minRelativeIntensity=0*, *relativeToPeak=0*, *minSpacing=0*, *edgeBoundary=1*, *maxNumPeaks=1*, *_ar_FT=None*)

Finds the maximal points of a 2D array.

Parameters

- **ar** (*array*) –
- **subpixel** (*str*) – specifies the subpixel resolution algorithm to use. must be in ('pixel','poly','multicorr'), which correspond to pixel resolution, subpixel resolution by fitting a parabola, and subpixel resultion by Fourier upsampling.
- **upsample_factor** – the upsampling factor for the 'multicorr' algorithm
- **sigma** – if >0, applies a gaussian filter
- **maxNumPeaks** – the maximum number of maxima to return
- **minAbsoluteIntensity** – minSpacing, edgeBoundary, maxNumPeaks: filtering applied after maximum detection and before subpixel refinement
- **minRelativeIntensity** – minSpacing, edgeBoundary, maxNumPeaks: filtering applied after maximum detection and before subpixel refinement
- **relativeToPeak** – minSpacing, edgeBoundary, maxNumPeaks: filtering applied after maximum detection and before subpixel refinement

:param
[minSpacing, edgeBoundary, maxNumPeaks: filtering applied] after maximum detection and before subpixel refinement

Parameters
_ar_FT (*complex array*) – None, uses this argument as the Fourier transform of *ar*, instead of recomputing it

Returns
a structured array with fields 'x','y','intensity'

py4DSTEM.preprocess.utils.**filter_2D_maxima**(*maxima*, *minAbsoluteIntensity=0*, *minRelativeIntensity=0*, *relativeToPeak=0*, *minSpacing=0*, *edgeBoundary=1*, *maxNumPeaks=1*)

Parameters

- **maxima** – a numpy structured array with fields 'x', 'y', 'intensity'
- **minAbsoluteIntensity** – delete counts with intensity below this value
- **minRelativeIntensity** – delete counts with intensity below this value times the intensity of the i'th peak, where i is given by *relativeToPeak*
- **relativeToPeak** – see above
- **minSpacing** – if two peaks are within this euclidean distance from one another, delete the less intense of the two
- **edgeBoundary** – delete peaks within this distance of the image edge
- **maxNumPeaks** – an integer. defaults to 1

Returns
a numpy structured array with fields 'x', 'y', 'intensity'

py4DSTEM.preprocess.utils.**linear_interpolation_2D**(*ar*, *x*, *y*)

Calculates the 2D linear interpolation of array ar at position x,y using the four nearest array elements.

1.4.5 process

Table of Contents

- *process*
  - *calibration*
  - *classification*
  - *diffraction*
  - *diskdetection*
  - *fit*
  - *latticevectors*
  - *phase*
  - *probe*
  - *rdf*
  - *utils*
  - *virtualdiffraction*
  - *virtualimage*
  - *wholepatternfit*

calibration

Functions related to elliptical calibration, such as fitting elliptical distortions.

The user-facing representation of ellipses is in terms of the following 5 :param x0: :param y0 the center of the ellipse: :param a the semimajor axis length: :param b the semiminor axis length: :param theta the: to the x-axis, in radians :type theta the: positive, right handed

More details about the elliptical parameterization used can be found in the module docstring for process/utils/elliptical_coords.py.

py4DSTEM.process.calibration.ellipse.**fit_ellipse_1D**(*ar*, *center=None*, *fitradii=None*, *mask=None*)

For a 2d array ar, fits a 1d elliptical curve to the data inside an annulus centered at *center* with inner and outer radii at *fitradii*. The data to fit make optionally be additionally masked with the boolean array mask. See module docstring for more info.

Parameters

- **ar** (*ndarray*) – array containing the data to fit
- **center** (*2-tuple of floats*) – the center (x0,y0) of the annular fitting region
- **fitradii** (*2-tuple of floats*) – inner and outer radii (ri,ro) of the fit region
- **mask** (*ar-shaped ndarray of bools*) – ignore data wherever mask==True

Returns

A 5-tuple containing the ellipse parameters:

- **x0**: the center x-position

- **y0**: the center y-position
- **a**: the semimajor axis length
- **b**: the semiminor axis length
- **theta**: the tilt of the ellipse semimajor axis with respect to the x-axis, in radians

Return type
(5-tuple of floats)

py4DSTEM.process.calibration.ellipse.**ellipse_err**(*p*, *x*, *y*, *val*)

For a point (x,y) in a 2d cartesian space, and a function taking the value val at point (x,y), and some 1d ellipse in this space given by

```
A(x-x0)^2 + B(x-x0)(y-y0) + C(y-y0)^2 = 1
```

this function computes the error associated with the function's value at (x,y) given by its deviation from the ellipse times val.

Note that this function is for internal use, and uses ellipse parameters *p* given in canonical form (x0,y0,A,B,C), which is different from the ellipse parameterization used in all the user-facing functions, for reasons of numerical stability.

py4DSTEM.process.calibration.ellipse.**fit_ellipse_amorphous_ring**(*data*, *center*, *fitradii*, *p0=None*, *mask=None*)

Fit the amorphous halo of a diffraction pattern, including any elliptical distortion.

The fit function is:

```
f(x,y; I0,I1,sigma0,sigma1,sigma2,c_bkgd,x0,y0,A,B,C) =
    Norm(r; I0,sigma0,0) +
    Norm(r; I1,sigma1,R)*Theta(r-R)
    Norm(r; I1,sigma2,R)*Theta(R-r) + c_bkgd
```

where

- (x,y) are cartesian coordinates,
- r is the radial coordinate,
- (I0,I1,sigma0,sigma1,sigma2,c_bkgd,x0,y0,R,B,C) are parameters,
- Norm(x;I,s,u) is a gaussian in the variable x with maximum amplitude I, standard deviation s, and mean u
- Theta(x) is a Heavyside step function
- R is the radial center of the double sided gaussian, derived from (A,B,C) and set to the mean of the semiaxis lengths

The function thus contains a pair of gaussian-shaped peaks along the radial direction of a polar-elliptical parametrization of a 2D plane. The first gaussian is centered at the origin. The second gaussian is centered about some finite R, and is 'two-faced': it's comprised of two half-gaussians of different standard deviations, stitched together at their mean value of R. This Janus (two-faced ;p) gaussian thus comprises an elliptical ring with different inner and outer widths.

The parameters of the fit function are

- I0: the intensity of the first gaussian function
- I1: the intensity of the Janus gaussian
- sigma0: std of first gaussian
- sigma1: inner std of Janus gaussian

- sigma2: outer std of Janus gaussian
- c_bkgd: a constant offset
- x0,y0: the origin
- A,B,C: The ellipse parameters, in the form Ax^2 + Bxy + Cy^2 = 1

Parameters

- **data** (*2d array*) – the data
- **center** (*2-tuple of numbers*) – the center (x0,y0)
- **fitradii** (*2-tuple of numbers*) – the inner and outer radii of the fitting annulus
- **p0** (*11-tuple*) – initial guess parameters. If p0 is None, the function will compute a guess at all parameters. If p0 is a 11-tuple it must be populated by some mix of numbers and None; any parameters which are set to None will be guessed by the function. The parameters are the 11 parameters of the fit function described above, p0 = (I0,I1,sigma0,sigma1,sigma2,c_bkgd,x0,y0,A,B,C). Note that x0,y0 are redundant; their guess values are the x0,y0 values passed to the main function, but if they are passed as elements of p0 these will take precendence.
- **mask** (*2d array of bools*) – only fit to datapoints where mask is True

Returns

Returns a 2-tuple.

The first element is the ellipse parameters need to elliptically parametrize diffraction space, and is itself a 5-tuple:

- **x0**: x center
- **y0**: y center,
- **a**: the semimajor axis length
- **b**: the semiminor axis length
- **theta**: tilt of a-axis w.r.t x-axis, in radians

The second element is the full set of fit parameters to the double sided gaussian function, described above, and is an 11-tuple

Return type
(2-tuple comprised of a 5-tuple and an 11-tuple)

py4DSTEM.process.calibration.ellipse.**double_sided_gaussian_fiterr**(*p*, *x*, *y*, *val*)

Returns the fit error associated with a point (x,y) with value val, given parameters p.

py4DSTEM.process.calibration.ellipse.**double_sided_gaussian**(*p*, *x*, *y*)

Return the value of the double-sided gaussian function at point (x,y) given parameters p, described in detail in the fit_ellipse_amorphous_ring docstring.

py4DSTEM.process.calibration.ellipse.**constrain_degenerate_ellipse**(*data*, *p_ellipse*, *r_inner*, *r_outer*, *phi_known*, *fitrad=6*)

When fitting an ellipse to data containing 4 diffraction spots in a narrow annulus about the central beam, the answer is degenerate: an infinite number of ellipses correctly fit this data. Starting from one ellipse in the degenerate family of ellipses, this function selects the ellipse which will yield a final angle of phi_known between a pair of the diffraction peaks after performing elliptical distortion correction.

Note that there are two possible angles which phi_known might refer to, because the angle of interest is well defined up to a complementary angle. This function is written such that phi_known should be the smaller of these two angles.

Parameters

- **data** (*ndarray*) –
- **p_ellipse** (*5-tuple*) – the ellipse parameters (x0,y0,a,b,theta)
- **r_inner** (*float*) – the fitting annulus inner radius
- **r_outer** (*float*) – the fitting annulus outer radius
- **phi_known** (*float*) – the known angle between a pair of diffraction peaks, in radians
- **fitrad** (*float*) – the region about the fixed data point used to refine its position

Returns

A 2-tuple containing:

- **a_constrained**: *(float)* the first semiaxis of the selected ellipse
- **b_constrained**: *(float)* the second semiaxis of the selected ellipse

Return type
(2-tuple)

py4DSTEM.process.calibration.origin.**fit_origin**(*data*, *mask=None*, *fitfunction='plane'*, *returnfitp=False*, *robust=False*, *robust_steps=3*, *robust_thresh=2*)

Fits the position of the origin of diffraction space to a plane or parabola, given some 2D arrays (qx0_meas,qy0_meas) of measured center positions, optionally masked by the Boolean array *mask*. The 2D data arrays may be passed directly as a 2-tuple to the arg *data*, or, if *data* is either a DataCube or Calibration instance, they will be retreived automatically. If a DataCube or Calibration are passed, fitted origin and residuals are stored there directly.

Parameters

- **data** (*2-tuple of 2d arrays*) – the measured origin position (qx0,qy0)
- **mask** (*2b boolean array, optional*) – ignore points where mask=False
- **fitfunction** (*str, optional*) – must be 'plane' or 'parabola' or 'bezier_two' or 'constant'
- **returnfitp** (*bool, optional*) – if True, returns the fit parameters
- **robust** (*bool, optional*) – If set to True, fit will be repeated with outliers removed.
- **robust_steps** (*int, optional*) – Optional parameter. Number of robust iterations performed after initial fit.
- **robust_thresh** (*int, optional*) – Threshold for including points, in units of root-mean-square (standard deviations) error of the predicted values after fitting.

Returns

Return value depends on returnfitp. If `returnfitp==False` (default), returns a 4-tuple containing:

- **qx0_fit**: *(ndarray)* the fit origin x-position
- **qy0_fit**: *(ndarray)* the fit origin y-position
- **qx0_residuals**: *(ndarray)* the x-position fit residuals

- **qy0_residuals**: *(ndarray)* the y-position fit residuals

If `returnfitp==True`, returns a 2-tuple. The first element is the 4-tuple described above. The second element is a 4-tuple (popt_x,popt_y,pcov_x,pcov_y) giving fit parameters and covariance matrices with respect to the chosen fitting function.

Return type
(variable)

py4DSTEM.process.calibration.origin.**get_origin_single_dp**(*dp*, *r*, *rscale=1.2*)

Find the origin for a single diffraction pattern, assuming (a) there is no beam stop, and (b) the center beam contains the highest intensity.

Parameters

- **dp** (*ndarray*) – the diffraction pattern
- **r** (*number*) – the approximate disk radius
- **rscale** (*number*) – factor by which *r* is scaled to generate a mask

Returns
The origin

Return type
(2-tuple)

py4DSTEM.process.calibration.origin.**get_origin**(*datacube*, *r=None*, *rscale=1.2*, *dp_max=None*, *mask=None*, *fast_center=False*)

Find the origin for all diffraction patterns in a datacube, assuming (a) there is no beam stop, and (b) the center beam contains the highest intensity. Stores the origin positions in the Calibration associated with datacube, and optionally also returns them.

Parameters

- **datacube** (DataCube) – the data
- **r** (*number or None*) – the approximate radius of the center disk. If None (default), tries to compute r using the get_probe_size method. The data used for this is controlled by dp_max.
- **rscale** (*number*) – expand 'r' by this amount to form a mask about the center disk when taking its center of mass
- **dp_max** (*ndarray or None*) – the diffraction pattern or dp-shaped array used to compute the center disk radius, if r is left unspecified. Behavior depends on type:
  - if `dp_max==None` (default), computes and uses the maximal diffraction pattern. Note that for a large datacube, this may be a slow operation.
  - otherwise, this should be a (Q_Nx,Q_Ny) shaped array
- **mask** (*ndarray or None*) – if not None, should be an (R_Nx,R_Ny) shaped boolean array. Origin is found only where mask==True, and masked arrays are returned for qx0,qy0
- **fast_center** – (bool) Skip the center of mass refinement step.

Returns
the origin, (x,y) at each scan position

Return type
(2-tuple of (R_Nx,R_Ny)-shaped ndarrays)

py4DSTEM.process.calibration.origin.**get_origin_friedel**(*datacube:* DataCube, *mask=None*, *upsample_factor=1*, *device='cpu'*, *return_cpu=True*)

Fit the origin for each diffraction pattern, with or without a beam stop. The method we have developed here is a heavily modified version of masked cross correlation, where we use Friedel symmetry of the diffraction pattern to find the common center.

More details about how the correlation step can be found in: https://doi.org/10.1109/TIP.2011.2181402

Parameters

- **datacube** (*(DataCube)*) – The 4D dataset.
- **mask** (*(np array, optional)*) – Boolean mask which is False under the beamstop and True in the diffraction pattern. One approach to generating this mask is to apply a suitable threshold on the average diffraction pattern and use binary opening/closing to remove any holes. If no mask is provided, this method will likely not work with a beamstop.
- **upsample_factor** (*(int)*) – Upsample factor for subpixel fitting of the image shifts.
- **device** (*string*) – 'cpu' or 'gpu' to select device
- **return_cpu** (*bool*) – Return arrays on cpu.

Returns

(tuple of np arrays) measured center position of each diffraction pattern

Return type

qx0, qy0

py4DSTEM.process.calibration.probe.**get_probe_size**(*DP*, *thresh_lower=0.01*, *thresh_upper=0.99*, *N=100*)

Gets the center and radius of the probe in the diffraction plane.

The algorithm is as follows: First, create a series of N binary masks, by thresholding the diffraction pattern DP with a linspace of N thresholds from thresh_lower to thresh_upper, measured relative to the maximum intensity in DP. Using the area of each binary mask, calculate the radius r of a circular probe. Because the central disk is typically very intense relative to the rest of the DP, r should change very little over a wide range of intermediate values of the threshold. The range in which r is trustworthy is found by taking the derivative of r(thresh) and finding identifying where it is small. The radius is taken to be the mean of these r values. Using the threshold corresponding to this r, a mask is created and the CoM of the DP times this mask it taken. This is taken to be the origin x0,y0.

Parameters

- **DP** (*2D array*) – the diffraction pattern in which to find the central disk. A position averaged, or shift-corrected and averaged, DP works best.
- **thresh_lower** (*float, 0 to 1*) – the lower limit of threshold values
- **thresh_upper** (*float, 0 to 1*) – the upper limit of threshold values
- **N** (*int*) – the number of thresholds / masks to use

Returns

A 3-tuple containing:

- **r**: *(float)* the central disk radius, in pixels
- **x0**: *(float)* the x position of the central disk center
- **y0**: *(float)* the y position of the central disk center

Return type
(3-tuple)

py4DSTEM.process.calibration.qpixelsize.**get_Q_pixel_size**(*q_meas*, *q_known*, *units='A'*)

Computes the size of the Q-space pixels.

Parameters

- **q_meas** (*number*) – a measured distance in q-space in pixels
- **q_known** (*number*) – the corresponding known *real space* distance
- **unit** (*str*) – the units of the real space value of *q_known*

Returns
the detector pixel size, the associated units

Return type
(number,str)

py4DSTEM.process.calibration.qpixelsize.**get_dq_from_indexed_peaks**(*qs*, *hkl*, *a*)

Get dq, the size of the detector pixels in the diffraction plane, in inverse length units, using a set of measured peak distances from the optic axis, their Miller indices, and the known unit cell size.

Parameters

- **qs** (*array*) – the measured peak positions
- **hkl** (*list/tuple of length-3 tuples*) – the Miller indices of the peak positions qs. The length of qs and hkl must be the same. To ignore any peaks, for this peak set (h,k,l)=(0,0,0).
- **a** (*number*) – the unit cell size

Returns

A 4-tuple containing:

- **dq**: *(number)* the detector pixel size
- **qs_fit**: *(array)* the fit positions of the peaks
- **hkl_fit**: *(list/tuple of length-3 tuples)* the Miller indices of the fit peaks
- **mask**: *(array of bools)* False wherever hkl[i]==(0,0,0)

Return type
(4-tuple)

py4DSTEM.process.calibration.rotation.**compare_QR_rotation**(*im_R*, *im_Q*, *QR_rotation*, *R_rotation=0*, *R_position=None*, *Q_position=None*, *R_pos_anchor='center'*, *Q_pos_anchor='center'*, *R_length=0.33*, *Q_length=0.33*, *R_width=0.001*, *Q_width=0.001*, *R_head_length_adjust=1*, *Q_head_length_adjust=1*, *R_head_width_adjust=1*, *Q_head_width_adjust=1*, *R_color='r'*, *Q_color='r'*, *figsize=(10, 5)*, *returnfig=False*)

Visualize a rotational offset between an image in real space, e.g. a STEM virtual image, and an image in diffraction space, e.g. a defocused CBED shadow image of the same region, by displaying an arrow overlaid over each

of these two images with the specified QR rotation applied. The QR rotation is defined as the counter-clockwise rotation from real space to diffraction space, in degrees.

Parameters

- **im_R** (*numpy array or other 2D image-like object (e.g. a VirtualImage)*) – A real space image, e.g. a STEM virtual image
- **im_Q** (*numpy array or other 2D image-like object*) – A diffraction space image, e.g. a defocused CBED image
- **QR_rotation** (*number*) – The counterclockwise rotation from real space to diffraction space, in degrees
- **R_rotation** (*number*) – The orientation of the arrow drawn in real space, in degrees
- **R_position** (*None or 2-tuple*) – The position of the anchor point for the R-space arrow. If None, defaults to the center of the image
- **Q_position** (*None or 2-tuple*) – The position of the anchor point for the Q-space arrow. If None, defaults to the center of the image
- **R_pos_anchor** (*'center' or 'tail' or 'head'*) – The anchor point for the R-space arrow, i.e. the point being specified by the *R_position* parameter
- **Q_pos_anchor** (*'center' or 'tail' or 'head'*) – The anchor point for the Q-space arrow, i.e. the point being specified by the *Q_position* parameter
- **R_length** (*number or None*) – The length of the R-space arrow, as a fraction of the mean size of the image
- **Q_length** (*number or None*) – The length of the Q-space arrow, as a fraction of the mean size of the image
- **R_width** (*number*) – The width of the R-space arrow
- **Q_width** (*number*) – The width of the R-space arrow
- **R_head_length_adjust** (*number*) – Scaling factor for the R-space arrow head length
- **Q_head_length_adjust** (*number*) – Scaling factor for the Q-space arrow head length
- **R_head_width_adjust** (*number*) – Scaling factor for the R-space arrow head width
- **Q_head_width_adjust** (*number*) – Scaling factor for the Q-space arrow head width
- **R_color** (*color*) – Color of the R-space arrow
- **Q_color** (*color*) – Color of the Q-space arrow
- **figsize** (*2-tuple*) – The figure size
- **returnfig** (*bool*) – Toggles returning the figure and axes

py4DSTEM.process.calibration.rotation.**get_Qvector_from_Rvector**(*vx*, *vy*, *QR_rotation*)

For some vector (vx,vy) in real space, and some rotation QR between real and reciprocal space, determine the corresponding orientation in diffraction space. Returns both R and Q vectors, normalized.

Parameters

- **vx** (*numbers*) – the (x,y) components of a real space vector
- **vy** (*numbers*) – the (x,y) components of a real space vector
- **QR_rotation** (*number*) – the offset angle between real and reciprocal space.
- **Specifically** –

- **to** (*the counterclockwise rotation of real space with respect*) –
- **degrees.** (*diffraction space. In*) –

Returns

4-tuple consisting of:

- **vx_R**: the x component of the normalized real space vector
- **vy_R**: the y component of the normalized real space vector
- **vx_Q**: the x component of the normalized reciprocal space vector
- **vy_Q**: the y component of the normalized reciprocal space vector

Return type
(4-tuple)

py4DSTEM.process.calibration.rotation.**get_Rvector_from_Qvector**(*vx*, *vy*, *QR_rotation*)

For some vector (vx,vy) in diffraction space, and some rotation QR between real and reciprocal space, determine the corresponding orientation in diffraction space. Returns both R and Q vectors, normalized.

Parameters

- **vx** (*numbers*) – the (x,y) components of a reciprocal space vector
- **vy** (*numbers*) – the (x,y) components of a reciprocal space vector
- **QR_rotation** (*number*) – the offset angle between real and reciprocal space. Specifically, the counterclockwise rotation of real space with respect to diffraction space. In degrees.

Returns

4-tuple consisting of:

- **vx_R**: the x component of the normalized real space vector
- **vy_R**: the y component of the normalized real space vector
- **vx_Q**: the x component of the normalized reciprocal space vector
- **vy_Q**: the y component of the normalized reciprocal space vector

Return type
(4-tuple)

classification

class py4DSTEM.process.classification.braggvectorclassification.**BraggVectorClassification**(*braggpeaks*, *Qx*, *Qy*, *X_is_boolean=Tr*, *max_dist=None*)

A class for classifying 4D-STEM data based on which Bragg peaks are found at each diffraction pattern.

A BraggVectorClassification instance enables classification using several methods; a brief overview is provided here, with more details in each individual method's documentation.

Initialization methods:

__init__:
Determine the initial classes. The approach here involves first segmenting diffraction space, using maxima of a Bragg vector map.

get_initial_classes_by_cooccurrence:

Class refinement methods: Each of these methods creates a new set of candidate classes, *but does not yet overwrite the old classes*. This enables the new classes to be viewed and compared to the old classes before deciding whether to accept or reject them. Thus running two of these methods in succession, without accepting changes in between, simply discards the first set of candidate classes.

nmf:
Nonnegative matrix factorization (X = WH) to refine the classes. Briefly, after constructing a matrix X which describes which Bragg peaks were observed in each diffraction pattern, we factor X into two smaller matrices, W and H. Physically, W and H describe a small set of classes, each of which corresponds to some subset of (or, more strictly, weights for) the Bragg peaks and the scan positions. We additionally impose the contraint that, on physical grounds, all the elements of X, W, and H must be nonnegative.

split:
If any classes contain multiple non-contiguous segments in real space, divide these into distinct classes.

merge:
If any classes contain sufficient overlap in both scan positions and BPs, merge them into a single class.

Accepting/rejecting changes:

accept:
Updates classes (the W and H matrices) with the current candidate classes.

reject:
Discard the current candidate classes.

Class examination methods:

get_class:
get a single class, returning both its BP weights and scan position weights

get_class_BPs:
get the BP weights for a single class

get_class_image:
get the image, i.e. scan position weights, associated with a single class

get_candidate_class:
as above, for the current candidate class

get_candidate_class_BPs:
as above, for the current candidate class

get_candidate_class_image:
as above, for the current candidate class

Parameters

- **braggpeaks** (PointListArray) – Bragg peaks; must have coords 'qx' and 'qy'
- **Qx** (*ndarray of floats*) – x-coords of the voronoi points
- **Qy** (*ndarray of floats*) – y-coords of the voronoi points
- **X_is_boolean** (*bool*) – if True, populate X with bools (BP is or is not present). if False, populate X with floats (BP c.c. intensities)

- **max_dist** (*None or number*) – maximum distance from a given voronoi point a peak can be and still be associated with this label

__init__(*braggpeaks*, *Qx*, *Qy*, *X_is_boolean=True*, *max_dist=None*)

Initializes a BraggVectorClassification instance.

This method: 1. Gets integer labels for all of the detected Bragg peaks, according to which

(Qx,Qy) is closest, then generating a corresponding set of integers for each scan position. See get_braggpeak_labels_by_scan_position() docstring for more info.

2. Generates the data matrix X. See the nmf() method docstring for more info.

This method should be followed by one of the methods which populates the initial classes - currently, either get_initial_classes_by_cooccurrence() or get_initial_classes_from_images. These methods generate the W and H matrices – i.e. the decompositions of the X matrix in terms of scan positions and Bragg peaks – which are necessary for any subsequent processing.

Parameters

- **braggpeaks** (PointListArray) – Bragg peaks; must have coords 'qx' and 'qy'
- **Qx** (*ndarray of floats*) – x-coords of the voronoi points
- **Qy** (*ndarray of floats*) – y-coords of the voronoi points
- **X_is_boolean** (*bool*) – if True, populate X with bools (BP is or is not present). if False, populate X with floats (BP c.c. intensities)
- **max_dist** (*None or number*) – maximum distance from a given voronoi point a peak can be and still be associated with this label

R_Nx

shape of real space (x)

R_Ny

shape of real space (y)

Qx

x-coordinates of the voronoi points

Qy

y-coordinates of the voronoi points

braggpeak_labels

the sets of Bragg peaks present at each scan position

N_feat

first dimension of the data matrix; the number of bragg peaks

N_meas

second dimension of the data matrix; the number of scan positions

X

the data matrix

get_initial_classes_by_cooccurrence(*thresh=0.3*, *BP_fraction_thresh=0.1*, *max_iterations=200*, *X_is_boolean=True*, *n_corr_init=2*)

Populate the initial classes by finding sets of Bragg peaks that tend to co-occur in the same diffraction patterns.

Beginning from the sets of Bragg peaks labels for each scan position (determined in __init__), this method gets initial classes by determining which labels are most likely to co-occur with each other – see get_initial_classes() docstring for more info. Then the matrices W and H are generated – see nmf() doscstring for discussion.

Parameters

- **thresh** (*float in [0,1]*) – threshold for adding new BPs to a class
- **BP_fraction_thresh** (*float in [0,1]*) – algorithm terminates if fewer than this fraction of the BPs have not been assigned to a class
- **max_iterations** (*int*) – algorithm terminates after this many iterations
- **n_corr_init** (*int*) – seed new classes by finding maxima of the n-point joint probability function. Must be 2 or 3.

get_initial_classes_from_images(*class_images*)

Populate the initial classes using a set of user-defined class images.

Parameters

class_images (*ndarray*) – must have shape (R_Nx,R_Ny,N_c), where N_c is the number of classes, and class_images[:,:,i] is the image of class i.

nmf(*max_iterations=1*)

Nonnegative matrix factorization to refine the classes.

The data matrix `X` is factored into two smaller matrices, `W` and `H`:

```
X = WH
```

Here,

- ``X``is the data matrix. It has shape (N_feat,N_meas), where N_feat is the number of Bragg peak integer labels (i.e. len(Qx)) and N_meas is the number of diffraction patterns (i.e. R_Nx*R_Ny). Element X[i,j] represents the value of the i'th BP in the j'th DP. The values depend on the flag datamatrix_is_boolean: if True, X[i,j] is 1 if this BP was present in this DP, or 0 if not; if False, X[i,j] is the cross correlation intensity of this BP in this DP.
- `W` is the class matrix. It has shape (N_feat,N_c), where N_c is the number of classes. The i'th column vector, w_i = W[:,i], describes the weight of each Bragg peak in the i'th class. w_i has length N_feat, and w_i[j] describes how strongly the j'th BP is associated with the i'th class.
- `H` is the coefficient matrix. It has shape (N_c,N_meas). The i'th column vector H[:,i] describes the contribution of each class to scan position i.

Alternatively, we can completely equivalently think of H as a class matrix, and W as a coeffient matrix. In this picture, the i'th row vector of H, h_i = H[i,:], describes the weight of each scan position in the i'th class. h_i has length N_meas, and h_i[j] describes how strongly the j'th scan position is associated with the i'th class. The row vector W[i,:] is then a coefficient vector, which gives the contributions each of the (H) classes to the measured values of the i'th BP. These pictures are related by a transpose: X = WH is equivalent to X.T = (H.T)(W.T).

In nonnegative matrix factorization we impose the constrain, here on physical grounds, that all elements of X, W, and H should be nonnegative.

The computation itself is performed using the sklearn nmf class. When this method is called, the three relevant matrices should already be defined. This method refines W and H, with up to max_iterations NMF steps.

Parameters

max_iterations (*int*) – the maximum number of NMF steps to take

split(*sigma=2*, *threshold_split=0.25*, *expand_mask=1*, *minimum_pixels=1*)

If any classes contain multiple non-contiguous segments in real space, divide these regions into distinct classes.

Algorithm is as follows: First, an image of each class is obtained from its scan position weights. Then, the image is convolved with a gaussian of std sigma. This is then turned into a binary mask, by thresholding with threshold_split. Stray pixels are eliminated by performing a one pixel binary closing, then binary opening. The mask is then expanded by expand_mask pixels. Finally, the contiguous regions of the resulting mask are found. These become the new class components by scan position.

The splitting itself involves creating two classes - i.e. adding a column to W and a row to H. The new BP classes (W columns) have exactly the same values as the old BP class. The two new scan position classes (H rows) divide up the non-zero entries of the old scan position class into two or more non-intersecting subsets, each of which becomes its own new class.

Parameters

- **sigma** (*float*) – std of gaussian kernel used to smooth the class images before thresholding and splitting.
- **threshold_split** (*float*) – used to threshold the class image to create a binary mask.
- **expand_mask** (*int*) – number of pixels by which to expand the mask before separating into contiguous regions.
- **minimum_pixels** (*int*) – if, after splitting, a potential new class contains fewer than this number of pixels, ignore it

merge(*threshBPs=0.1*, *threshScanPosition=0.1*, *return_params=True*)

If any classes contain sufficient overlap in both scan positions and BPs, merge them into a single class.

The algorithm is as follows: First, the Pearson correlation coefficient matrix is calculated for the classes according to both their diffraction space, Bragg peak representations (i.e. the correlations of the columns of W) and according to their real space, scan position representations (i.e. the correlations of the rows of H). Class pairs whose BP correlation coefficient exceeds threshBPs and whose scan position correlation coefficient exceed threshScanPosition are deemed 'sufficiently overlapped', and are marked as merge candidates. To account for intransitivity issues (e.g. class pairs 1/2 and 2/3 are merge candidates, but class pair 1/3 is not), merging is then performed beginning with candidate pairs with the greatest product of the two correlation coefficients, skipping later merge candidate pairs if one of the two classes has already been merged.

The algorithm can be looped until no more merge candidates satisfying the specified thresholds remain with the merge_iterative method.

The merging itself involves turning two classes into one by combining a pair of W columns (i.e. the BP representations of the classes) and the corresponding pair of H rows (i.e. the scan position representation of the class) into a single W column / H row. In terms of scan positions, the new row of H is generated by simply adding the two old H rows. In terms of Bragg peaks, the new column of W is generated by adding the two old columns of W, while weighting each by its total intensity in real space (i.e. the sum of its H row).

Parameters

- **threshBPs** (*float*) – the threshold for the bragg peaks correlation coefficient, above which the two classes are considered candidates for merging
- **threshScanPosition** (*float*) – the threshold for the scan position correlation coefficient, above which two classes are considered candidates for merging
- **return_params** (*bool*) – if True, returns W_corr, H_corr, and merge_candidates. Otherwise, returns nothing. Incompatible with iterative=True.

merge_by_class_index(*i*, *j*)

Merge classes i and j into a single class.

Columns i and j of W pair of W (i.e. the BP representations of the classes) and the corresponding pair of H rows (i.e. the scan position representation of the class) are mergedinto a single W column / H row. In terms of scan positions, the new row of H is generated by simply adding the two old H rows. In terms of Bragg peaks, the new column of W is generated by adding the two old columns of W, while weighting each by its total intensity in real space (i.e. the sum of its H row).

Parameters

- **i** (*int*) – index of the first class to merge
- **j** (*int*) – index of the second class to merge

split_by_class_index(*i*, *sigma=2*, *threshold_split=0.25*, *expand_mask=1*, *minimum_pixels=1*)

If class i contains multiple non-contiguous segments in real space, divide these regions into distinct classes.

Algorithm is as described in the docstring for self.split.

Parameters

- **i** (*int*) – index of the class to split
- **sigma** (*float*) – std of gaussian kernel used to smooth the class images before thresholding and splitting.
- **threshold_split** (*float*) – used to threshold the class image to create a binary mask.
- **expand_mask** (*int*) – number of pixels by which to expand the mask before separating into contiguous regions.
- **minimum_pixels** (*int*) – if, after splitting, a potential new class contains fewer than this number of pixels, ignore it

remove_class(*i*)

Remove class i.

Parameters

i (*int*) – index of the class to remove

merge_iterative(*threshBPs=0.1*, *threshScanPosition=0.1*)

If any classes contain sufficient overlap in both scan positions and BPs, merge them into a single class.

Identical to the merge method, with the addition of iterating until no new merge pairs are found.

Parameters

- **threshBPs** (*float*) – the threshold for the bragg peaks correlation coefficient, above which the two classes are considered candidates for merging
- **threshScanPosition** (*float*) – the threshold for the scan position correlation coefficient, above which two classes are considered candidates for merging

`accept()`

Updates classes (the W and H matrices) with the current candidate classes.

`reject()`

Discard the current candidate classes.

`get_class(i)`

Get a single class, returning both its BP weights and scan position weights.

Parameters
i (*int*) – the class index

Returns

A 2-tuple containing:

- **class_BPs**: *(length N_feat array of floats)* the weights of the N_feat Bragg peaks for this class
- **class_image**: *(shape (R_Nx,R_Ny) array of floats)* the weights of each scan position in this class

Return type
(2-tuple)

`get_class_BPs(i)`

Get a single class, returning its BP weights.

Parameters
i (*int*) – the class index

Returns
the weights of the N_feat Bragg peaks for this class

Return type
(length N_feat array of floats)

`get_class_image(i)`

Get a single class, returning its scan position weights.

Parameters
i (*int*) – the class index

Returns
the weights of each scan position in this class

Return type
(shape (R_Nx,R_Ny) array of floats)

`get_candidate_class(i)`

Get a single candidate class, returning both its BP weights and scan position weights.

Parameters
i (*int*) –

Returns

A 2-tuple containing:

- **class_BPs**: *(length N_feat array of floats)* the weights of the N_feat Bragg peaks for this class
- **class_image**: *(shape (R_Nx,R_Ny) array of floats)* the weights of each scan position in this class

Return type
(2-tuple)

`get_candidate_class_BPs(i)`

Get a single candidate class, returning its BP weights.

Accepts:
i (int) the class index

Returns

class_BPs (length N_feat array of floats) the weights of the N_feat Bragg peaks for this class

`get_candidate_class_image(i)`

Get a single candidate class, returning its scan position weights.

Parameters
i (*int*) – the class index

Returns
the weights of each scan position in this class

Return type
(shape (R_Nx,R_Ny) array of floats)

py4DSTEM.process.classification.braggvectorclassification.**get_braggpeak_labels_by_scan_position**(*braggpea*
Qx,
Qy,
max_dist

For each scan position, gets a set of integers, specifying the bragg peaks at this scan position.

From a set of positions in diffraction space (Qx,Qy), assign each detected bragg peak in the PointListArray braggpeaks a label corresponding to the index of the closest position; thus for a bragg peak at (qx,qy), if the closest position in (Qx,Qy) is (Qx[i],Qy[i]), assign this peak the label i. This is equivalent to assigning each bragg peak (qx,qy) a label according to the Voronoi region it lives in, given a voronoi tesselation seeded from the points (Qx,Qy).

For each scan position, get the set of all indices i for all bragg peaks found at this scan position.

Parameters

- **braggpeaks** (PointListArray) – Bragg peaks; must have coords 'qx' and 'qy'
- **Qx** (*ndarray of floats*) – x-coords of the voronoi points
- **Qy** (*ndarray of floats*) – y-coords of the voronoi points
- **max_dist** (*None or number*) – maximum distance from a given voronoi point a peak can be and still be associated with this label

Returns
(list of lists of sets) the labels found at each scan position. Scan position (Rx,Ry) is accessed via braggpeak_labels[Rx][Ry]

py4DSTEM.process.classification.braggvectorclassification.**get_initial_classes**(*braggpeak_labels*,
N, *thresh=0.3*,
BP_fraction_thresh=0.1,
max_iterations=200,
n_corr_init=2)

From the sets of Bragg peaks present at each scan position, get an initial guess classes at which Bragg peaks should be grouped together into classes.

The algorithm is as follows: 1. Calculate an n-point correlation function, i.e. the joint probability of any given n BPs coexisting in a diffraction pattern. n is controlled by n_corr_init, and must be 2 or 3. peaks i, j, and k are all in the same DP. 2. Find the BP triplet maximizing the 3-point function; include these three BPs in a class. 3. Get all DPs containing the class BPs. From these, find the next most likely BP to also be present. If its probability of coexisting with the known class BPs is greater than thresh, add it to the class and repeat this step. Otherwise, proceed to the next step. 4. Check: if the new class is the same as a class that has already been found, OR if the fraction of BPs which have not yet been placed in a class is less than BP_fraction_thresh, or more than max_iterations have been attempted, finish, returning all classes. Otherwise, set all slices of the 3-point function containing the BPs in the new class to zero, and begin a new iteration, starting at step 2 using the new, altered 3-point function.

Parameters

- **N** (*int*) – the total number of indexed Bragg peaks in the 4D-STEM dataset
- **braggpeak_labels** (*list of lists of sets*) – the Bragg peak labels found at each scan position; see get_braggpeak_labels_by_scan_position().
- **thresh** (*float in [0,1]*) – threshold for adding new BPs to a class
- **BP_fraction_thresh** (*float in [0,1]*) – algorithm terminates if fewer than this fraction of the BPs have not been assigned to a class
- **max_iterations** (*int*) – algorithm terminates after this many iterations
- **n_corr_init** (*int*) – seed new classes by finding maxima of the n-point joint probability function. Must be 2 or 3.

Returns
the sets of Bragg peaks constituting the classes

Return type
(list of sets)

py4DSTEM.process.classification.classutils.**get_class_DP**(*datacube*, *class_image*, *thresh=0.01*, *xshifts=None*, *yshifts=None*, *darkref=None*, *intshifts=True*)

Get the average diffraction pattern for the class described in real space by class_image.

Parameters

- **datacube** (DataCube) – a datacube
- **class_image** (*2D array*) – the weight of the class at each position in real space
- **thresh** (*float*) – only include diffraction patterns for scan positions with a value greater than or equal to thresh in class_image
- **xshifts** (*2D array, or None*) – the x diffraction shifts at each real space pixel. If None, no shifting is performed.
- **yshifts** (*2D array, or None*) – the y diffraction shifts at each real space pixel. If None, no shifting is performed.
- **darkref** (*2D array, or None*) – background to remove from each diffraction pattern
- **intshifts** (*bool*) – if True, round shifts to the nearest integer to speed up computation

Returns
the average diffraction pattern for the class

Return type
(2D array)

py4DSTEM.process.classification.classutils.**get_class_DP_without_Bragg_scattering**(*datacube*, *class_image*, *braggpeaks*, *radius*, *x0*, *y0*, *thresh=0.01*, *xshifts=None*, *yshifts=None*, *darkref=None*, *intshifts=True*)

Get the average diffraction pattern, removing any Bragg scattering, for the class described in real space by class_image.

Bragg scattering is eliminated by masking circles of size radius about each of the detected peaks in braggpeaks in each diffraction pattern before adding to the average image. Importantly, braggpeaks refers to the peak positions in the raw data - i.e. BEFORE any shift correction is applied. Passing shifted Bragg peaks will yield incorrect results. For speed, the Bragg peaks are removed with a binary mask, rather than a continuous sigmoid, so selecting a radius that is slightly (~1 pix) larger than the disk size is recommended.

Parameters

- **datacube** (DataCube) – a datacube
- **class_image** (*2D array*) – the weight of the class at each position in real space
- **braggpeaks** (PointListArray) – the detected Bragg peak positions, with respect to the raw data (i.e. not diffraction shift or ellipse corrected)
- **radius** (*number*) – the radius to mask about each detected Bragg peak - should be slightly larger than the disk radius
- **x0** (*number*) – x-position of the optic axis
- **y0** (*number*) – y-position of the optic axis
- **thresh** (*float*) – only include diffraction patterns for scan positions with a value greater than or equal to thresh in class_image
- **xshifts** (*2D array, or None*) – the x diffraction shifts at each real space pixel. If None, no shifting is performed.
- **yshifts** (*2D array, or None*) – the y diffraction shifts at each real space pixel. If None, no shifting is performed.
- **darkref** (*2D array, or None*) – background to remove from each diffraction pattern
- **intshifts** (*bool*) – if True, round shifts to the nearest integer to speed up computation

Returns
class_DP (2D array) the average diffraction pattern for the class

class py4DSTEM.process.classification.featurization.**Featurization**(*features*, *R_Nx*, *R_Ny*, *name*)

A class for feature selection, modification, and classification of 4D-STEM data based on a user defined array of input features for each pattern. Features are stored under Featurization. Features and can be used for a variety of unsupervised classification tasks.

Initialization methods:

__init__:
Creates instance of featurization

concatenate_features:
Creates instance of featurization from a list of featurization instances

from_braggvectors:
Creates instance of featurization from a BraggVectors instance

Feature Dictionary Modification Methods

add_feature:
Adds features to the features array

remove_feature:
Removes features to the features array

Feature Preprocessing Methods

MinMaxScaler:
Performs sklearn MinMaxScaler operation on features stored at a key

RobustScaler:
Performs sklearn RobustScaler operation on features stored at a key

mean_feature:
Takes the rowwise average of a matrix stored at a key, such that only one column is left, reducing a set of n features down to 1 feature per pattern.

median_feature:
Takes the rowwise median of a matrix stored at a key, such that only one column is left, reducing a set of n features down to 1 feature per pattern.

max_feature:
Takes the rowwise max of a matrix stored at a key, such that only one column is left, reducing a set of n features down to 1 feature per pattern.

Classification Methods

PCA:
Principal Component Analysis to refine features.

ICA:
Independent Component Analysis to refine features.

NMF:
Performs either traditional or iterative Nonnegative Matrix Factorization (NMF) to refine features.

GMM:
Gaussian mixture model to predict class labels. Fits a gaussian based on covariance of features.

Class Examination Methods

get_class_DPs:
Gets weighted class diffraction patterns (DPs) for an NMF or GMM operation

get_class_ims:
Gets weighted class images (ims) for an NMF or GMM operation

__init__(*features*, *R_Nx*, *R_Ny*, *name*)

Initializes classification instance.

This method: 1. Generates key:value pair to access input features 2. Initializes the empty dictionaries for feature modification and classification

Parameters

- **features** (*list*) – A list of ndarrays which will each be associated with value stored at the key in the same index within the list
- **R_Nx** (*int*) – The real space x dimension of the dataset
- **R_Ny** (*int*) – The real space y dimension of the dataset
- **name** (*str*) – The name of the featurization object

Returns
New Featurization instance

Return type
new_instance

from_braggvectors(*bins_x*, *bins_y*, *intensity_scale*, *name*, *mask=None*)

Initialize a featurization instance from a BraggVectors instance

Parameters

- **braggvectors** (BraggVectors) – BraggVectors instance containing calibrations
- **bins_x** (*int*) – Number of pixels per bin in x direction
- **bins_y** (*int*) – Number of pixels per bin in y direction
- **intensity_scale** (*float*) – Value to scale intensity of detected disks by
- **name** (*str*) – Name of featurization instance
- **mask** (*bool*) – Mask to remove disks in unwanted positions in diffraction space

Returns
Featurization instance

Return type
new_instance

Details:
Transforms the calibrated pointlistarray in BraggVectors instance into a numpy array that can be clustered using the methods in featurization.

concatenate_features(*name*)

Concatenates featurization instances (features) and outputs a new Featurization instance containing the concatenated features from each featurization instance. R_Nx, R_Ny will be inherited from the featurization instances and must be consistent across objects.

Parameters

- **features** (*list*) – A list of keys to be concatenated into one array
- **name** (*str*) – The name of the featurization instance

Returns
Featurization instance

Return type
new_instance

add_features(*feature*)

Add a feature to the end of the features array

Parameters

- **key** (*int, float, str*) – A key in which a feature can be accessed from
- **feature** (*ndarray*) – The feature associated with the key

delete_features(*index*)

Deletes feature columns from the feature array

Parameters
index (*int, list*) – A key which will be removed

mean_feature(*index*)

Takes columnwise mean and replaces features in 'index'.

Parameters
index (*list of int*) – Indices of features to take the mean of. New feature array is placed in self.features.

median_feature(*index*)

Takes columnwise median and replaces features in 'index'. New feature array is placed in self.features.

Parameters
index (*list of int*) – Indices of features to take the median of.

max_feature(*index*)

Takes columnwise max and replaces features in 'index'. New feature array is placed in self.features.

Parameters
index (*list of int*) – Indices of features to take the max of.

MinMaxScaler(*return_scaled=True*)

Uses sklearn MinMaxScaler to scale a subset of the input features. Replaces a feature with the positive shifted array.

Parameters
return_scaled (*bool*) – returns the scaled array

RobustScaler(*return_scaled=True*)

Uses sklearn RobustScaler to scale a subset of the input features. Replaces a feature with the positive shifted array.

Parameters
return_scaled (*bool*) – returns the scaled array

shift_positive(*return_scaled=True*)

Replaces a feature with the positive shifted array.

Parameters
return_scaled (*bool*) – returns the scaled array

PCA(*components*, *return_results=False*)

Performs PCA on features

Parameters
components (*list*) – A list of ints for each key. This will be the output number of features

ICA(*components*, *return_results=True*)

Performs ICA on features

Parameters
components (*list*) – A list of ints for each key. This will be the output number of features

NMF(*max_components*, *num_models*, *merge_thresh=1*, *max_iterations=1*, *random_seed=None*, *save_all_models=True*, *return_results=False*)

Performs either traditional Nonnegative Matrix Factoriation (NMF) or iteratively on input features. For Traditional NMF:

set either merge_threshold = 1, max_iterations = 1, or both. Default is to set

Parameters

- **max_components** (*int*) – Number of initial components to start the first NMF iteration
- **merge_thresh** (*float*) – Correlation threshold to merge features
- **num_models** (*int*) – Number of independent models to run (number of learners that will be combined in consensus).
- **max_iterations** (*int*) – Number of iterations. Default 1, which runs traditional NMF
- **random_seed** (*int*) – Random seed.
- **save_all_models** (*bool*) – Whether or not to return all of the models - default is to return all outputs for consensus clustering. if False, will only return the model with the lowest NMF reconstruction error.
- **return_results** (*bool*) – Whether or not to return the final class weights

Details:

This method may require trial and error for proper selection of parameters. To perform traditional NMF, the defaults should be used:

merge_thresh = 1 max_iterations = 1

Note that the max_components in this case will be equivalent to the number of classes the NMF model identifies.

Iterative NMF calculates the correlation between all of the output columns from an NMF iteration, merges the features correlated above the merge_thresh, and performs NMF until either max_iterations is reached or until no more columns are correlated above merge_thresh.

GMM(*cv*, *components*, *num_models*, *random_seed=None*, *return_results=False*)

Performs gaussian mixture model on input features

Parameters

- **cv** (*str*) – Covariance type - must be 'spherical', 'tied', 'diag', or 'full'
- **components** (*int*) – Number of components
- **num_models** (*int*) – Number of models to run
- **random_seed** (*int*) – Random seed

get_class_DPs(*datacube*, *method*, *thresh*)

Returns weighted class patterns based on classification instance datacube must be vectorized in real space (shape = (R_Nx * R_Ny, 1, Q_Nx, Q_Ny)

Parameters

- **classification_method** (*str*) – Either 'nmf' or 'gmm' - finds location of clusters

- **datacube** (*py4DSTEM datacube*) – Vectorized in real space, with shape (R_Nx * R_Ny, Q_Nx, Q_Ny)

get_class_ims(*classification_method*)

Returns weighted class maps based on classification instance

Parameters

classification_method (*str*) – Location to retrieve class images from - NMF, GMM, PCA, or ICA

spatial_separation(*size*, *threshold=0*, *method=None*, *clean=True*)

Identify spatially distinct regions from class images and separate based on a threshold and size.

Parameters

- **size** (*int*) – Number of pixels which is the minimum to keep a class - all spatially distinct regions with less than 'size' pixels will be removed
- **threshold** (*float*) – Intensity weight of a component to keep
- **method** (*str*) – (Optional) Filter method, default None. Accepts options 'yen' and 'otsu'.
- **clean** (*bool*) – Whether or not to 'clean' cluster sets based on overlap, i.e. remove clusters that do not have any unique components

consensus(*threshold=0*, *location='spatially_separated_ims'*, *split=0*, *method='mean'*, *drop_bins=0*)

Consensus Clustering takes the outcome of a prepared set of 2D images from each cluster and averages the outcomes.

Parameters

- **threshold** (*float*) – Threshold weights, default 0
- **location** (*str*) – Where to get the consensus from - after spatial separation = 'spatially_separated_ims'
- **split_value** (*float*) – Threshold in which to separate classes during label correspondence (Default 0). This should be proportional to the expected class weights- the sum of the weights in the current class image that match nonzero values in each bin are calculated and then checked for splitting.
- **method** (*str*) – Method in which to combine the consensus clusters - either mean or median.
- **drop_bins** (*int*) – Number of clusters needed in each class to keep cluster set in the consensus. Default 0, meaning

Details:

This method involves 2 steps: Label correspondence and consensus clustering.

Label correspondence sorts the classes found by the independent models into bins based on class overlap in real space. Arguments related to label correspondence are the threshold and split_value. The threshold is related to the weights of the independent classes. If the weight of the observation in the class is less than the threshold, it will be set to 0. The split_value indicates the extent of similarity the independent classes must have before intializing a new bin. The default is 0 - this means if the class of interest has 0 overlap with the identified bins, a new bin will be created. The value is based on the sum of the weights in the current class image that match the nonzero values in the current bins.

Consensus clustering combines these sorted bin into 1 class based on the selected method (either 'mean' which takes the average of the bin, or 'median' which takes the median of the bin). Bins with less than the drop_bins value will not be included in the final results.

diffraction

py4DSTEM.process.diffraction.WK_scattering_factors.**compute_WK_factor**(*g: ndarray*, *Z: int*, *accelerating_voltage: float*, *thermal_sigma: float | None = None*, *include_core: bool = True*, *include_phonon: bool = True*, *verbose=False*) → complex128

Compute the Weickenmeier-Kohl atomic scattering factors, using the parameterization of the elastic part and computation of the inelastic part found in EMsoftLib/others.f90. Return value should be in Å.

This implementation always returns the absorptive, relativistically corrected factors.

Currently this is mostly a direct translation of the Fortran code, along with the accompanying comments from the original in quotation marks. Colin Ophus vectorized it around v0.13.17. Currently it is only vectorized over *g* (i.e. *Z* and all other args must be a single value.)

This method uses an 8-parameter fit to the elastic form factors, and then computes the absorptive form factors using an analytic solution based on that fitting function.

Args: (note that these values cannot be arrays: the code is not vectorized)

g (float/ndarray): Scattering vector magnitude in the crystallographic/py4DSTEM
convention, 1/d_hkl in units of 1/Å

Z (int): Atomic number. Data are available for H thru Cf (1 thru 98) accelerating_voltage (float): Accelerating voltage in eV. thermal_sigma (float): RMS atomic displacement for TDS, in Å

(This is often written as ⟨u⟩in papers)

include_core (bool): If True, include the core loss contribution to the absorptive
form factors.

include_phonon (bool): If True, include the phonon/TDS contribution to the
absorptive form factors.

Returns
The computed atomic form factor

Return type
Fscatt (np.complex128)

py4DSTEM.process.diffraction.WK_scattering_factors.**RIH2**(*X*)

WERTET X*EXP(-X)*EI(X) AUS FUER GROSSE X DURCH INTERPOLATION DER TABELLE … AUS ABRAMOWITZ

class py4DSTEM.process.diffraction.crystal.**Crystal**(*positions*, *numbers*, *cell*, *occupancy=None*)

A class storing a single crystal structure, and associated diffraction data.

orientation_plan(*zone_axis_range: ndarray = array([[0, 1, 1], [1, 1, 1]])*, *angle_step_zone_axis: float = 2.0*, *angle_coarse_zone_axis: float | None = None*, *angle_refine_range: float | None = None*, *angle_step_in_plane: float = 2.0*, *accel_voltage: float = 300000.0*, *corr_kernel_size: float = 0.08*, *radial_power: float = 1.0*, *intensity_power: float = 0.25*, *calculate_correlation_array=True*, *tol_peak_delete=None*, *tol_distance: float = 0.01*, *fiber_axis=None*, *fiber_angles=None*, *figsize: list | tuple | ndarray = (6, 6)*, *CUDA: bool = False*, *progress_bar: bool = True*)

Calculate the rotation basis arrays for an SO(3) rotation correlogram.

Parameters

- **zone_axis_range** (*float*) – Row vectors give the range for zone axis orientations. If user specifies 2 vectors (2x3 array), we start at [0,0,1]

  to make z-x-z rotation work.

  If user specifies 3 vectors (3x3 array), plan will span these vectors. Setting to 'full' as a string will use a hemispherical range. Setting to 'half' as a string will use a quarter sphere range. Setting to 'fiber' as a string will make a spherical cap around a given vector. Setting to 'auto' will use pymatgen to determine the point group symmetry

  of the structure and choose an appropriate zone_axis_range

- **angle_step_zone_axis** (*float*) – Approximate angular step size for zone axis search [degrees]
- **angle_coarse_zone_axis** (*float*) – Coarse step size for zone axis search [degrees]. Setting to None uses the same value as angle_step_zone_axis.
- **angle_refine_range** (*float*) – Range of angles to use for zone axis refinement. Setting to None uses same value as angle_coarse_zone_axis.
- **angle_step_in_plane** (*float*) – Approximate angular step size for in-plane rotation [degrees]
- **accel_voltage** (*float*) – Accelerating voltage for electrons [Volts]
- **corr_kernel_size** (*float*) – Correlation kernel size length. The size of the overlap kernel between the measured Bragg peaks and diffraction library Bragg peaks. [1/Angstroms]
- **radial_power** (*float*) – Power for scaling the correlation intensity as a function of the peak radius
- **intensity_power** (*float*) – Power for scaling the correlation intensity as a function of the peak intensity
- **calculate_correlation_array** (*bool*) – Set to false to skip calculating the correlation array. This is useful when we only want the angular range / rotation matrices.
- **tol_peak_delete** (*float*) – Distance to delete peaks for multiple matches. Default is kernel_size * 0.5
- **tol_distance** (*float*) – Distance tolerance for radial shell assignment [1/Angstroms]
- **fiber_axis** (*float*) – (3,) vector specifying the fiber axis
- **fiber_angles** (*float*) – (2,) vector specifying angle range from fiber axis, and in-plane angular range [degrees]
- **cartesian_directions** (*bool*) – When set to true, all zone axes and projection directions are specified in Cartesian directions.
- **figsize** (*float*) – (2,) vector giving the figure size
- **CUDA** (*bool*) – Use CUDA for the Fourier operations.
- **progress_bar** (*bool*) – If false no progress bar is displayed

match_orientations(*bragg_peaks_array: PointListArray*, *num_matches_return: int = 1*, *min_angle_between_matches_deg=None*, *min_number_peaks: int = 3*, *inversion_symmetry: bool = True*, *multiple_corr_reset: bool = True*, *return_orientation: bool = True*, *progress_bar: bool = True*)

Parameters

- **bragg_peaks_array** (PointListArray) – PointListArray containing the Bragg peaks and intensities, with calibrations applied
- **num_matches_return** (*int*) – return these many matches as 3th dim of orient (matrix)
- **min_angle_between_matches_deg** (*int*) – Minimum angle between zone axis of multiple matches, in degrees. Note that I haven't thought how to handle in-plane rotations, since multiple matches are possible.
- **min_number_peaks** (*int*) – Minimum number of peaks required to perform ACOM matching
- **inversion_symmetry** (*bool*) – check for inversion symmetry in the matches
- **multiple_corr_reset** (*bool*) – keep original correlation score for multiple matches
- **return_orientation** (*bool*) – Return orientation map from function for inspection. The map is always stored in the Crystal object.
- **progress_bar** (*bool*) – Show or hide the progress bar

match_single_pattern(*bragg_peaks: PointList*, *num_matches_return: int = 1*, *min_angle_between_matches_deg=None*, *min_number_peaks=3*, *inversion_symmetry=True*, *multiple_corr_reset=True*, *plot_polar: bool = False*, *plot_corr: bool = False*, *returnfig: bool = False*, *figsize: list | tuple | ndarray = (12, 4)*, *verbose: bool = False*)

Solve for the best fit orientation of a single diffraction pattern.

Parameters

- **bragg_peaks** (PointList) – numpy array containing the Bragg positions and intensities ('qx', 'qy', 'intensity')
- **num_matches_return** (*int*) – return these many matches as 3th dim of orient (matrix)
- **min_angle_between_matches_deg** (*int*) – Minimum angle between zone axis of multiple matches, in degrees. Note that I haven't thought how to handle in-plane rotations, since multiple matches are possible.
- **min_number_peaks** (*int*) – Minimum number of peaks required to perform ACOM matching
- **bool** (*multiple_corr_reset*) – check for inversion symmetry in the matches
- **bool** – keep original correlation score for multiple matches
- **subpixel_tilt** (*bool*) – set to false for faster matching, returning the nearest corr point
- **plot_polar** (*bool*) – set to true to plot the polar transform of the diffraction pattern
- **plot_corr** (*bool*) – set to true to plot the resulting correlogram
- **returnfig** (*bool*) – return figure handles

- **figsize** (*list*) – size of figure
- **verbose** (*bool*) – Print the fitted zone axes, correlation scores
- **CUDA** (*bool*) – Enable CUDA for the FFT steps

Returns

- **orientation** (*Orientation*) – Orientation class containing all outputs
- **fig, ax** (*handles*) – Figure handles for the plotting output

cluster_grains(*threshold_add=1.0*, *threshold_grow=0.1*, *angle_tolerance_deg=5.0*, *progress_bar=True*)

Cluster grains using rotation criterion, and correlation values.

Parameters

- **threshold_add** (*float*) – Minimum signal required for a probe position to initialize a cluster.
- **threshold_grow** (*float*) – Minimum signal required for a probe position to be added to a cluster.
- **angle_tolerance_deg** (*float*) – Rotation rolerance for clustering grains.
- **progress_bar** (*bool*) – Turns on the progress bar for the polar transformation

cluster_orientation_map(*stripe_width=(2, 2)*, *area_min=2*)

Produce a new orientation map from the clustered grains. Use a stripe pattern for the overlapping grains.

Parameters

- **stripe_width** (*(int,int)*) – Width of stripes for plotting maps with overlapping grains
- **area_min** (*(int)*) – Minimum size of grains to include

Returns
The clustered orientation map

Return type
orientation_map

calculate_strain(*bragg_peaks_array: PointListArray*, *orientation_map: OrientationMap*, *corr_kernel_size=None*, *sigma_excitation_error=0.02*, *tol_excitation_error_mult: float = 3*, *tol_intensity: float = 0.0001*, *k_max: float | None = None*, *min_num_peaks=5*, *intensity_weighting=False*, *robust=True*, *robust_thresh=3.0*, *rotation_range=None*, *mask_from_corr=True*, *corr_range=(0, 2)*, *corr_normalize=True*, *progress_bar=True*)

This function takes in both a PointListArray containing Bragg peaks, and a corresponding OrientationMap, and uses least squares to compute the deformation tensor which transforms the simulated diffraction pattern into the experimental pattern, for all probe positons.

TODO: add robust fitting?

Parameters

- **(PointListArray)** (*bragg_peaks_array*) – All Bragg peaks
- **(OrientationMap)** (*orientation_map*) – Orientation map generated from ACOM
- **(float)** (*rotation_range*) – Correlation kernel size - if user does not specify, uses self.corr_kernel_size.
- **(float)** – sigma value for envelope applied to s_g (excitation errors) in units of inverse Angstroms

- **(float)** – tolerance in units of sigma for s_g inclusion
- **float)** (*tol_intensity (np*) – tolerance in intensity units for inclusion of diffraction spots
- **(float)** – Maximum scattering vector
- **(int)** (*min_num_peaks*) – Minimum number of peaks required.
- **intensity_weighting** (*bool*) – Set to True to weight least squares by experimental peak intensity.
- **robust_fitting** (*bool*) – Set to True to use robust fitting, which performs outlier rejection.
- **robust_thresh** (*float*) – Threshold for robust fitting weights.
- **(float)** – Maximum rotation range in radians (for symmetry reduction).
- **(bool)** (*corr_normalize*) – Show progress bar
- **(bool)** – Use ACOM correlation signal for mask
- **(np.ndarray)** (*corr_range*) – Range of correlation signals for mask
- **(bool)** – Normalize correlation signal before masking

Returns
strain tensor

Return type
strain_map (*RealSlice*)

symmetry_reduce_directions(*orientation*, *match_ind=0*, *plot_output=False*, *figsize=(15, 6)*, *el_shift=0.0*, *az_shift=-30.0*)

This function calculates the symmetry-reduced cartesian directions from and orientation matrix stored in orientation.matrix, and outputs them into orientation.family. It optionally plots the 3D output.

save_ang_file(*file_name*, *orientation_map*, *ind_orientation=0*, *pixel_size=1.0*, *pixel_units='px'*, *transpose_xy=True*, *flip_x=False*, *flip_y=False*)

This function outputs an ascii text file in the .ang format, containing the Euler angles of an orientation map.

Parameters

- **file_name** (*str*) – Path to save .ang file.
- **orientation_map** (OrientationMap) – Class containing orientation matrices, correlation values, etc.
- **ind_orientation** (*int*) – Which orientation match to plot if num_matches > 1
- **pixel_size** (*float*) – Pixel size, if known.
- **pixel_units** (*str*) – Units of the pixel size
- **transpose_xy** (*bool*) – Transpose x and y pixel coordinates.
- **flip_x** (*bool*) – Swap x direction pixels (after transpose).

Returns
nothing

plot_structure(*orientation_matrix: ndarray | None = None*, *zone_axis_lattice: ndarray | None = None*, *proj_x_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *proj_x_cartesian: ndarray | None = None*, *size_marker: float = 400*, *tol_distance: float = 0.001*, *plot_limit: ndarray | None = None*, *camera_dist: float | None = None*, *show_axes: bool = False*, *perspective_axes: bool = True*, *figsize: tuple | list | ndarray = (8, 8)*, *returnfig: bool = False*)

Quick 3D plot of the untit cell /atomic structure.

Parameters

- **orientation_matrix** (*array*) – (3,3) orientation matrix, where columns represent projection directions.
- **zone_axis_lattice** (*array*) – (3,) projection direction in lattice indices
- **proj_x_lattice** (*array*) – (3,) x-axis direction in lattice indices
- **zone_axis_cartesian** (*array*) – (3,) cartesian projection direction
- **proj_x_cartesian** (*array*) – (3,) cartesian projection direction
- **scale_markers** (*float*) – Size scaling for markers
- **tol_distance** (*float*) – Tolerance for repeating atoms on edges on cell boundaries.
- **plot_limit** (*float*) – (2,3) numpy array containing x y z plot min and max in columns. Default is 1.1* unit cell dimensions.
- **camera_dist** (*float*) – Move camera closer to the plot (relative to matplotlib default of 10)
- **show_axes** (*bool*) – Whether to plot axes or not.
- **perspective_axes** (*bool*) – Select either perspective (true) or orthogonal (false) axes
- **figsize** (*2 element float*) – Size scaling of figure axes.
- **returnfig** (*bool*) – Return figure and axes handles.

Returns

fig, ax (optional) figure and axes handles

plot_structure_factors(*orientation_matrix: ndarray | None = None*, *zone_axis_lattice: ndarray | None = None*, *proj_x_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *proj_x_cartesian: ndarray | None = None*, *scale_markers: float = 1000.0*, *plot_limit: list | tuple | ndarray | None = None*, *camera_dist: float | None = None*, *show_axes: bool = True*, *perspective_axes: bool = True*, *figsize: list | tuple | ndarray = (8, 8)*, *returnfig: bool = False*)

3D scatter plot of the structure factors using magnitude^2, i.e. intensity.

Parameters

- **orientation_matrix** (*array*) – (3,3) orientation matrix, where columns represent projection directions.
- **zone_axis_lattice** (*array*) – (3,) projection direction in lattice indices
- **proj_x_lattice** (*array*) – (3,) x-axis direction in lattice indices
- **zone_axis_cartesian** (*array*) – (3,) cartesian projection direction
- **proj_x_cartesian** (*array*) – (3,) cartesian projection direction
- **scale_markers** (*float*) – size scaling for markers

- **plot_limit** (*float*) – x y z plot limits, default is [-1 1]*self.k_max
- **camera_dist** (*float*) – Move camera closer to the plot (relative to matplotlib default of 10)
- **show_axes** (*bool*) – Whether to plot axes or not.
- **perspective_axes** (*bool*) – Select either perspective (true) or orthogonal (false) axes
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns
fig, ax (optional) figure and axes handles

plot_scattering_intensity(*k_min=0.0*, *k_max=None*, *k_step=0.001*, *k_broadening=0.0*, *k_power_scale=0.0*, *int_power_scale=0.5*, *int_scale=1.0*, *remove_origin=True*, *bragg_peaks=None*, *bragg_k_power=0.0*, *bragg_intensity_power=1.0*, *bragg_k_broadening=0.005*, *figsize: list | tuple | ndarray = (10, 4)*, *returnfig: bool = False*)

1D plot of the structure factors

Parameters

- **k_min** (*float*) – min k value for profile range.
- **k_max** (*float*) – max k value for profile range.
- **k_step** (*float*) – Step size of k in profile range.
- **k_broadening** (*float*) – Broadening of simulated pattern.
- **k_power_scale** (*float*) – Scale SF intensities by k**k_power_scale.
- **int_power_scale** (*float*) – Scale SF intensities**int_power_scale.
- **int_scale** (*float*) – Scale output profile by this value.
- **remove_origin** (*bool*) – Remove origin from plot.
- **bragg_peaks** (BraggVectors) – Passed in bragg_peaks for comparison with simulated pattern.
- **bragg_k_power** (*float*) – bragg_peaks scaled by k**bragg_k_power.
- **bragg_intensity_power** (*float*) – bragg_peaks scaled by intensities**bragg_intensity_power.
- **bragg_k_broadening** (*float*) – Broadening applied to bragg_peaks.
- **figsize** (*list, tuple, np.ndarray*) – Figure size for plot.
- **(bool)** (*returnfig*) – Return figure and axes handles if this is True.

Returns
figure and axes handles

Return type
fig, ax (optional)

plot_orientation_zones(*azim_elev: list | tuple | ndarray | None = None*, *proj_dir_lattice: list | tuple | ndarray | None = None*, *proj_dir_cartesian: list | tuple | ndarray | None = None*, *tol_den=10*, *marker_size: float = 20*, *plot_limit: list | tuple | ndarray = array([-1.1, 1.1])*, *figsize: list | tuple | ndarray = (8, 8)*, *returnfig: bool = False*)

3D scatter plot of the structure factors using magnitude^2, i.e. intensity.

Parameters

- **azim_elev** (*array*) – az and el angles for plot
- **proj_dir_lattice** (*array*) – (3,) projection direction in lattice
- **proj_dir_cartesian** – (array): (3,) projection direction in cartesian
- **tol_den** (*int*) – tolerance for rational index denominator
- **dir_proj** (*float*) – projection direction, either [elev azim] or normal vector Default is mean vector of self.orientation_zone_axis_range rows
- **marker_size** (*float*) – size of markers
- **plot_limit** (*float*) – x y z plot limits, default is [0, 1.05]
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns

fig, ax (optional) figure and axes handles

plot_orientation_plan(*index_plot: int = 0*, *zone_axis_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *figsize: list | tuple | ndarray = (14, 6)*, *returnfig: bool = False*)

3D scatter plot of the structure factors using magnitude^2, i.e. intensity.

Parameters

- **index_plot** (*int*) – which index slice to plot
- **zone_axis_plot** (*3 element float*) – which zone axis slice to plot
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns

fig, ax (optional) figure and axes handles

plot_orientation_maps(*orientation_map=None*, *orientation_ind: int = 0*, *dir_in_plane_degrees: float = 0.0*, *corr_range: ndarray = array([0, 5])*, *corr_normalize: bool = True*, *scale_legend: bool | None = None*, *figsize: list | tuple | ndarray = (16, 5)*, *figbound: list | tuple | ndarray = (0.01, 0.005)*, *show_axes: bool = True*, *camera_dist=None*, *plot_limit=None*, *plot_layout=0*, *swap_axes_xy_limits=False*, *returnfig: bool = False*, *progress_bar=False*)

Plot the orientation maps.

Parameters

- **orientation_map** (OrientationMap) – Class containing orientation matrices, correlation values, etc. Optional - can reference internally stored OrientationMap.
- **orientation_ind** (*int*) – Which orientation match to plot if num_matches > 1
- **dir_in_plane_degrees** (*float*) – In-plane angle to plot in degrees. Default is 0 / x-axis / vertical down.
- **corr_range** (*np.ndarray*) – Correlation intensity range for the plot
- **corr_normalize** (*bool*) – If true, set mean correlation to 1.

- **scale_legend** (*float*) – 2 elements, x and y scaling of legend panel
- **figsize** (*array*) – 2 elements defining figure size
- **figbound** (*array*) – 2 elements defining figure boundary
- **show_axes** (*bool*) – Flag setting whether orienation map axes are visible.
- **camera_dist** (*float*) – distance of camera from legend
- **plot_limit** (*array*) – 2x3 array defining plot boundaries of legend
- **plot_layout** (*int*) – subplot layout: 0 - 1 row, 3 col 1 - 3 row, 1 col
- **swap_axes_xy_limits** (*bool*) – swap x and y boundaries for legend (not sure why we need this in some cases)
- **returnfig** (*bool*) – set to True to return figure and axes handles
- **progress_bar** (*bool*) – Enable progressbar when calculating orientation images.

Returns
RGB images fig, axs (handles): Figure and axes handes for the

Return type
images_orientation (int)

Note: Currently, no symmetry reduction. Therefore the x and y orientations are going to be correct only for [001][011][111] orientation triangle.

---

plot_fiber_orientation_maps(*orientation_map*, *orientation_ind: int = 0*, *symmetry_order: int | None = None*, *symmetry_mirror: bool = False*, *dir_in_plane_degrees: float = 0.0*, *corr_range: ndarray = array([0, 2])*, *corr_normalize: bool = True*, *show_axes: bool = True*, *medfilt_size: int | None = None*, *cmap_out_of_plane: str = 'plasma'*, *leg_size: int = 200*, *figsize: list | tuple | ndarray = (12, 8)*, *figbound: list | tuple | ndarray = (0.005, 0.04)*, *returnfig: bool = False*)

Generate and plot the orientation maps from fiber texture plots.

Parameters

- **orientation_map** (OrientationMap) – Class containing orientation matrices, correlation values, etc.
- **orientation_ind** (*int*) – Which orientation match to plot if num_matches > 1
- **dir_in_plane_degrees** (*float*) – Reference in-plane angle (degrees). Default is 0 / x-axis / vertical down.
- **corr_range** (*np.ndarray*) – Correlation intensity range for the plot
- **corr_normalize** (*bool*) – If true, set mean correlation to 1.
- **show_axes** (*bool*) – Flag setting whether orienation map axes are visible.
- **figsize** (*array*) – 2 elements defining figure size
- **figbound** (*array*) – 2 elements defining figure boundary
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns
RGB images fig, axs (handles): Figure and axes handes for the

Return type
images_orientation (int)

Note: Currently, no symmetry reduction. Therefore the x and y orientations are going to be correct only for [001][011][111] orientation triangle.

---

plot_clusters(*area_min=2*, *outline_grains=True*, *outline_thickness=1*, *fill_grains=0.25*, *smooth_grains=1.0*, *cmap='viridis'*, *figsize=(8, 8)*, *returnfig=False*)

Plot the clusters as an image.

Parameters

- **area_min** (*int (optional)*) – Min cluster size to include, in units of probe positions.
- **outline_grains** (*bool (optional)*) – Set to True to draw grains with outlines
- **outline_thickness** (*int (optional)*) – Thickenss of the grain outline
- **fill_grains** (*float (optional)*) – Outlined grains are filled with this value in pixels.
- **smooth_grains** (*float (optional)*) – Grain boundaries are smoothed by this value in pixels.
- **figsize** (*tuple*) – Size of the figure panel
- **returnfig** (*bool*) – Setting this to true returns the figure and axis handles

Returns
Figure and axes handles

Return type
fig, ax (optional)

plot_cluster_size(*area_min=None*, *area_max=None*, *area_step=1*, *weight_intensity=False*, *pixel_area=1.0*, *pixel_area_units='px^2'*, *figsize=(8, 6)*, *returnfig=False*)

Plot the cluster sizes

Parameters

- **area_min** (*int (optional)*) – Min area to include in pixels^2
- **area_max** (*int (optional)*) – Max area bin in pixels^2
- **area_step** (*int (optional)*) – Step size of the histogram bin in pixels^2
- **weight_intensity** (*bool*) – Weight histogram by the peak intensity.
- **pixel_area** (*float*) – Size of pixel area unit square
- **pixel_area_units** (*string*) – Units of the pixel area
- **figsize** (*tuple*) – Size of the figure panel
- **returnfig** (*bool*) – Setting this to true returns the figure and axis handles

Returns
Figure and axes handles

Return type
fig, ax (optional)

calibrate_pixel_size(*bragg_peaks*, *scale_pixel_size=1.0*, *bragg_k_power=1.0*, *bragg_intensity_power=1.0*, *k_min=0.0*, *k_max=None*, *k_step=0.002*, *k_broadening=0.002*, *fit_all_intensities=False*, *set_calibration_in_place=False*, *verbose=True*, *plot_result=False*, *figsize: list | tuple | ndarray = (12, 6)*, *returnfig=False*)

Use the calculated structure factor scattering lengths to compute 1D diffraction patterns, and solve the best-fit relative scaling between them. Returns the fit pixel size in Å^-1.

Parameters

- **bragg_peaks** (BraggVectors) – Input Bragg vectors.
- **scale_pixel_size** (*float*) – Initial guess for scaling of the existing pixel size If the pixel size is currently uncalibrated, this is a guess of the pixel size in Å^-1. If the pixel size is already (approximately) calibrated, this is the scaling factor to correct that existing calibration.
- **bragg_k_power** (*float*) – Input Bragg peak intensities are multiplied by k**bragg_k_power to change the weighting of longer scattering vectors
- **bragg_intensity_power** (*float*) – Input Bragg peak intensities are raised power **bragg_intensity_power.
- **k_min** (*float*) – min k value for fitting range (Å^-1)
- **k_max** (*float*) – max k value for fitting range (Å^-1)
- **k_step** (*float) step size of k in fitting range (Å^-1*) –
- **k_broadening** (*float*) – Initial guess for Gaussian broadening of simulated pattern (Å^-1)
- **fit_all_intensities** (*bool*) – Set to true to allow all peak intensities to change independently. False forces a single intensity scaling for all peaks.
- **set_calibration** (*bool*) – if True, set the fit pixel size to the calibration metadata, and calibrate bragg_peaks
- **verbose** (*bool*) – Output the calibrated pixel size.
- **plot_result** (*bool*) – Plot the resulting fit.
- **figsize** (*list, tuple, np.ndarray*) – Figure size of the plot.
- **returnfig** (*bool*) – Return handles figure and axis

Returns

fig, ax – Figure and axis handles, if returnfig=True.

Return type

handles, optional

calibrate_unit_cell(*bragg_peaks*, *coef_index=None*, *coef_update=None*, *bragg_k_power=1.0*, *bragg_intensity_power=1.0*, *k_min=0.0*, *k_max=None*, *k_step=0.005*, *k_broadening=0.02*, *fit_all_intensities=True*, *verbose=True*, *plot_result=False*, *figsize: list | tuple | ndarray = (12, 6)*, *returnfig=False*)

Solve for the best fit scaling between the computed structure factors and bragg_peaks.

Parameters

- **bragg_peaks** (BraggVectors) – Input Bragg vectors.
- **coef_index** (*list of ints*) – List of ints that act as pointers to unit cell parameters and angles to update.

- **coef_update** (*list of bool*) – List of booleans to indicate whether or not to update the cell at that position
- **bragg_k_power** (*float*) – Input Bragg peak intensities are multiplied by k**bragg_k_power to change the weighting of longer scattering vectors
- **bragg_intensity_power** (*float*) – Input Bragg peak intensities are raised power **bragg_intensity_power.
- **k_min** (*float*) – min k value for fitting range (Å^-1)
- **k_max** (*float*) – max k value for fitting range (Å^-1)
- **k_step** (*float*) – step size of k in fitting range (Å^-1)
- **k_broadening** (*float*) – Initial guess for Gaussian broadening of simulated pattern (Å^-1)
- **fit_all_intensities** (*bool*) – Set to true to allow all peak intensities to change independently False forces a single intensity scaling.
- **verbose** (*bool*) – Output the calibrated pixel size.
- **plot_result** (*bool*) – Plot the resulting fit.
- **figsize** (*list, tuple, np.ndarray*) –
- **returnfig** (*bool*) – Return handles figure and axis

Returns
Optional figure and axis handles, if returnfig=True.

Return type
fig, ax (handles)

Details: User has the option to define what is allowed to update in the unit cell using the arguments coef_index and coef_update. Each has 6 entries, corresponding to the a, b, c, alpha, beta, gamma parameters of the unit cell, in this order. The coef_update argument is a list of bools specifying whether or not the unit cell value will be allowed to change (True) or must maintain the original value (False) upon fitting. The coef_index argument provides a pointer to the index in which the code will update to.

For example, to update a, b, c, alpha, beta, gamma all independently of eachother, the following arguments should be used:

coef_index = [0, 1, 2, 3, 4, 5] coef_update = [True, True, True, True, True, True,]

The default is set to automatically define what can update in a unit cell based on the point group constraints. When either 'coef_index' or 'coef_update' are None, these constraints will be automatically pulled from the pointgroup.

For example, the default for cubic unit cells is:
coef_index = [0, 0, 0, 3, 3, 3] coef_update = [True, True, True, False, False, False]

Which allows a, b, and c to update (True in first 3 indices of coef_update) but b and c update based on the value of a (0 in the 1 and 2 list entries in coef_index) such that a = b = c. While coef_update is False for alpha, beta, and gamma (entries 3, 4, 5), no updates will be made to the angles.

The user has the option to predefine coef_index or coef_update to override defaults. In the coef_update list, there must be 6 entries and each are boolean. In the coef_index list, there must be 6 entries, with the first 3 entries being between 0 - 2 and the last 3 entries between 3 - 5. These act as pointers to pull the updated parameter from.

generate_dynamical_diffraction_pattern(*beams: PointList*, *thickness: float | list | tuple | ndarray*, *zone_axis_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *foil_normal_lattice: ndarray | None = None*, *foil_normal_cartesian: ndarray | None = None*, *verbose: bool = False*, *always_return_list: bool = False*, *dynamical_matrix_cache: DynamicalMatrixCache | None = None*, *return_complex: bool = False*, *return_eigenvectors: bool = False*, *return_Smatrix: bool = False*) → *PointList* | List[*PointList*]

Generate a dynamical diffraction pattern (or thickness series of patterns) using the Bloch wave method.

The beams to be included in the Bloch calculation must be pre-calculated and passed as a PointList containing at least (qx, qy, h, k, l) fields.

If `thickness` is a single value, one new PointList will be returned. If `thickness` is a sequence of values, a list of PointLists will be returned,

corresponding to each thickness value in the input.

Frequent reference will be made to "Introduction to conventional transmission electron microscopy"
by DeGraef, whose overall approach we follow here.

Parameters

- **beams** (*PointList*) – PointList from the kinematical diffraction generator which will define the beams included in the Bloch calculation
- **thickness** (*float or list/array*) – The main Bloch calculation can be reused for multiple thicknesses without much overhead.
- **direction.** (*zone_axis & foil_normal Incident beam orientation and foil normal*) – Each can be specified in the Cartesian or crystallographic basis, using e.g. zone_axis_lattice or zone_axis_cartesian. These are internally parsed by Crystal.parse_orientation

Less commonly used args:

always_return_list (bool): When True, the return is always a list of PointLists,
even for a single thickness

dynamical_matrix_cache: (DyanmicalMatrixCache) Dataclass used for caching of the
dynamical matrix. If the cached matrix does not exist, it is computed and stored. Subsequent calls will use the cached matrix for the off-diagonal components of the A matrix and overwrite the diagonal elements. This is used for CBED calculations.

return_complex (bool): When True, returns both the complex amplitude and intensity. Defaults to (False)

Returns

Bragg peaks with fields [qx, qy, intensity, h, k, l]
or

[bragg_peaks,...] (PointList): If thickness is a list/array, or always_return_list is True,
a list of PointLists is returned.

if return_complex = True:

bragg_peaks (PointList): Bragg peaks with fields [qx, qy, intensity, amplitude, h, k, l]
or

[bragg_peaks,...] (PointList): If thickness is a list/array, or always_return_list is True,
a list of PointLists is returned.

if return_Smatrix = True:

[S_matrix, ...], psi_0: Returns a list of S-matrices for each thickness (this is always a list),
and the vector representing the incident plane wave. The beams of the S-matrix have the same order as in the input *beams*.

Return type
bragg_peaks (*PointList*)

generate_CBED(*beams: ~emdfile.classes.pointlist.PointList*, *thickness: float | list | tuple | ~numpy.ndarray*, *alpha_mrad: float*, *pixel_size_inv_A: float*, *DP_size_inv_A: float | None = None*, *zone_axis_lattice: ~numpy.ndarray | None = None*, *zone_axis_cartesian: ~numpy.ndarray | None = None*, *foil_normal_lattice: ~numpy.ndarray | None = None*, *foil_normal_cartesian: ~numpy.ndarray | None = None*, *LACBED: bool = False*, *dtype: ~numpy.dtype = <class 'numpy.float32'>*, *verbose: bool = False*, *progress_bar: bool = True*, *return_mask: bool = False*, *two_beam_zone_axis_lattice: ~numpy.ndarray | None = None*, *return_probe: bool = False*) → ndarray | List[ndarray] | Dict[Tuple[int], ndarray]

Generate a dynamical CBED pattern using the Bloch wave method.

Parameters

- **beams** (PointList) – PointList from the kinematical diffraction generator which will define the beams included in the Bloch calculation
- **thickness** (*float or list/array*) – The main Bloch calculation can be reused for multiple thicknesses without much overhead.
- **alpha_mrad** (*float*) – Convergence angle for CBED pattern. Note that if disks in the calculation overlap, they will be added incoherently, and the resulting CBED will thus represent the average over the unit cell (i.e. a PACBED pattern, as described in LeBeau et al., Ultramicroscopy 110(2): 2010.)
- **pixel_size_inv_A** (*float*) – CBED pixel size in 1/Å.
- **DP_size_inv_A** (*optional float*) – If specified, defines the extents of the diffraction pattern. If left unspecified, the DP will be automatically scaled to fit all of the beams present in the input plus some small buffer.
- **zone_axis** (*np float vector*) – 3 element projection direction for sim pattern Can also be a 3x3 orientation matrix (zone axis 3rd column)
- **foil_normal** – 3 element foil normal - set to None to use zone_axis
- **LACBED** (*bool*) – keyed by tuples of (h,k,l).
- **proj_x_axis** (*np float vector*) – 3 element vector defining image x axis (vertical)
- **PointList** (*two_beam_zone_axis_lattice When only two beams are present in the "beams"*) – the computation of the projected crystallographic directions becomes ambiguous. In this case, you must specify the indices of the zone axis used to generate the beams.

:param
[the computation of the projected crystallographic directions] becomes ambiguous. In this case, you must specify the indices of the zone axis used to generate the beams.

Parameters
return_probe (*bool*) – If True, the probe (np.ndarray) will be returned in additon to the CBED

Returns
CBED pattern as np.ndarray If thickness is a sequence: CBED patterns for each thickness value as a list of np.ndarrays If LACBED is True and thickness is scalar: Dictionary with tuples of ints (h,k,l) as keys, mapping to np.ndarray. If LACBED is True and thickness is a sequence: List of dictionaries, structured as above. If return_probe is True: will return a tuple (<CBED/LACBED object>, Probe)

Return type
If thickness is a scalar

calculate_dynamical_structure_factors(*accelerating_voltage: float*, *method: str = 'WK-CP'*, *k_max: float = 2.0*, *thermal_sigma: float | dict | None = None*, *tol_structure_factor: float = 0.0*, *recompute_kinematic_structure_factors=True*, *g_vec_precision=None*)

Calculate and store the relativistic corrected structure factors used for Bloch computations in a dictionary for faster lookup.

Parameters

- **accelerating_voltage** (*float*) – accelerating voltage in eV
- **method** (*str*) – Choose which parameterization of the structure factors to use: "Lobato": Uses the kinematic structure factors from crystal.py, using the parameterization from

  Lobato & Van Dyck, Acta Cryst A 70:6 (2014)

  "Lobato-absorptive": Lobato factors plus an imaginary part
  equal to 0.1•f, as a simple but inaccurate way to include absorption, per Hashimoto, Howie, & Whelan, Proc R Soc Lond A 269:80-103 (1962)

  "WK": Uses the Weickenmeier-Kohl parameterization for
  the elastic form factors, including Debye-Waller factor, with no absorption, as described in Weickenmeier & Kohl, Acta Cryst A 47:5 (1991)

  "WK-C": WK form factors plus the "core" contribution to absorption
  following H. Rose, Optik 45:2 (1976)

  "WK-P": WK form factors plus the phonon/TDS absorptive contribution "WK-CP": WK form factors plus core and phonon absorption (default)
- **k_max** (*float*) – max scattering length to compute structure factors to. Setting this to 2x the k_max used in generating the beamsn included in a simulation will retain all possible couplings
- **thermal_sigma** (*float or dict{int->float}*) – RMS atomic diplacement for attenuating form factors to account for thermal broadening of the potential, only used when a "WK" method is selected. Required when WK-P or WK-CP are selected. Units are Å. (This is often written as ⟨u⟩in papers) To specify different ⟨u⟩ for each element, pass a dictionary with Z as the key, mapping to the appropriate float value

- **tol_structure_factor** (*float*) – tolerance for removing low-valued structure factors. Reflections with structure factor below the tolerance will have zero coupling in the dynamical calculations (i.e. they are the ignored weak beams)
- **recompute_kinematic_structure_factors** (*bool*) – When True, recomputes the kinematic structure factors using the same tol_structure_factor, and with k_max set to *half* the k_max for the dynamical factors. The factor of half ensures that every beam in a simulation can couple to every other beam (no high-angle couplings in the Bloch matrix are set to zero.)
- **g_vec_precision** (*optional int*) – If specified, rounds |g| to this many decimal places so that automatic caching of the atomic form factors is not slowed down due to floating point errors. Setting this to 3 can give substantial speedup at the cost of some reduced accuracy
- **factors.** (*See WK_scattering_factors.py for details on the Weickenmeier-Kohl form*) –

__init__(*positions*, *numbers*, *cell*, *occupancy=None*)

Parameters

- **positions** (*np.array*) – fractional coordinates of each atom in the cell
- **numbers** (*np.array*) – Z number for each atom in the cell, if one number passed it is used for all atom positions
- **cell** (*np.array*) – specify the unit cell, using a variable number of parameters 1 number: the lattice parameter for a cubic cell 3 numbers: the three lattice parameters for an orthorhombic cell 6 numbers: the a,b,c lattice parameters and ,, angles for any cell 3x3 array: row vectors containing the (u,v,w) lattice vectors.
- **occupancy** (*np.array*) – Partial occupancy values for each atomic site. Must match the length of positions

positions

fractional atomic coordinates

get_strained_crystal(*exx=0.0*, *eyy=0.0*, *ezz=0.0*, *exy=0.0*, *exz=0.0*, *eyz=0.0*, *deformation_matrix=None*, *return_deformation_matrix=False*)

This method returns new Crystal class with strain applied. The directions of (x,y,z) are with respect to the default Crystal orientation, which can be checked with print(Crystal.lat_real) applied to the original Crystal.

Strains are given in fractional values, so exx = 0.01 is 1% strain along the x direction. Deformation matrix should be of the form:

deformation_matrix = np.array([
[1.0+exx, 1.0*exy, 1.0*exz], [1.0*exy, 1.0+eyy, 1.0*eyz], [1.0*exz, 1.0*eyz, 1.0+ezz],

])

Parameters

- **(float)** (*eyz*) – fractional strain along the xx direction
- **(float)** – fractional strain along the yy direction
- **(float)** – fractional strain along the zz direction
- **(float)** – fractional strain along the xy direction
- **(float)** – fractional strain along the xz direction

- **(float)** – fractional strain along the yz direction
- **(np.ndarray)** (*deformation_matrix*) – 3x3 array describing deformation matrix
- **(bool)** (*return_deformation_matrix*) – boolean switch to return deformation matrix

Returns

- *return_deformation_matrix == False* – strained_crystal (py4DSTEM.Crystal)
- *return_deformation_matrix == True* – (strained_crystal, deformation_matrix)

static from_ase(*atoms*)

Create a py4DSTEM Crystal object from an ASE atoms object

Parameters

atoms (*ase.Atoms*) – an ASE atoms object

static from_prismatic(*filepath*)

Create a py4DSTEM Crystal object from an prismatic style xyz co-ordinate file

Parameters

filepath (*str|Pathlib.Path*) – path to the prismatic format xyz file

static from_CIF(*CIF*, *primitive: bool = True*, *conventional_standard_structure: bool = True*)

Create a Crystal object from a CIF file, using pymatgen to import the CIF

Note that pymatgen typically prefers to return primitive unit cells, which can be overridden by setting conventional_standard_structure=True.

Parameters

- **CIF** – (str or Path) path to the CIF File
- **conventional_standard_structure** – (bool) if True, conventional standard unit cell will be returned instead of the primitive unit cell pymatgen typically returns

static from_pymatgen_structure(*structure=None*, *formula=None*, *space_grp=None*, *MP_key=None*, *conventional_standard_structure=True*)

Create a Crystal object from a pymatgen Structure object. If a Materials Project API key is installed, you may pass the Materials Project ID of a structure, which will be fetched through the MP API. For setup information see: https://pymatgen.org/usage.html#setting-the-pmg-mapi-key-in-the-config-file. Alternatively, Materials Porject API key can be pass as an argument through the function (MP_key). To get your API key, please visit Materials Project website and login/sign up using your email id. Once logged in, go to the dashboard to generate your own API key (https://materialsproject.org/dashboard).

Note that pymatgen typically prefers to return primitive unit cells, which can be overridden by setting conventional_standard_structure=True.

Parameters

- **structure** – (pymatgen Structure or str), if specified as a string, it will be considered as a Materials Project ID of a structure, otherwise it will accept only pymatgen Structure object. if None, MP database will be queried using the specified formula and/or space groups for the available structure
- **formula** – (str), pretty formula to search in the MP database, (note that the forumlas in MP database are not always formatted in the conventional order. Please visit Materials Project website for information (https://materialsproject.org/) if None, structure argument must not be None

- **space_grp** – (int) space group number of the forumula provided to query MP database. If None, MP will search for all the available space groups for the formula provided and will consider the one with lowest unit cell volume, only specify when using formula to search MP database
- **MP_key** – (str) Materials Project API key
- **conventional_standard_structure** – (bool) if True, conventional standard unit cell will be returned instead of the primitive unit cell pymatgen returns

static from_unitcell_parameters(*latt_params*, *elements*, *positions*, *space_group=None*, *lattice_type='cubic'*, *from_cartesian=False*, *conventional_standard_structure=True*)

Create a Crystal using pymatgen to generate unit cell manually from user inputs

Parameters

- **latt_params** – (list of floats) list of lattice parameters. For example, for cubic: latt_params = [a], for hexagonal: latt_params = [a, c], for monoclinic: latt_params = [a,b,c,beta], and in general: latt_params = [a,b,c,alpha,beta,gamma]
- **elements** – (list of strings) list of elements, for example for SnS: elements = [“Sn”, “S”]
- **positions** – (list) list of (x,y,z) positions for each element present in the elements, default: fractional coord
- **space_group** – (optional) (string or int) space group of the crystal system, if specified, unit cell will be created using pymatgen Structure.from_spacegroup function
- **lattice_type** – (string) type of crystal family: cubic, hexagonal, triclinic etc; default: ‘cubic’
- **from_cartesian** – (bool) if True, positions will be considered as cartesian, default: False
- **conventional_standard_structure** – (bool) if True, conventional standard unit cell will be returned instead of the primitive unit cell pymatgen returns

Returns

Crystal object

setup_diffraction(*accelerating_voltage: float*)

Set up attributes used for diffraction calculations without going through the full ACOM pipeline.

calculate_structure_factors(*k_max: float = 2.0*, *tol_structure_factor: float = 0.0001*, *return_intensities: bool = False*)

Calculate structure factors for all hkl indices up to max scattering vector k_max

Parameters

- **k_max** (*float*) – max scattering vector to include (1/Angstroms)
- **tol_structure_factor** (*float*) – tolerance for removing low-valued structure factors
- **return_intensities** (*bool*) – return the intensities and positions of all structure factor peaks.

Returns

Tuple of the q vectors and intensities of each structure factor.

Return type
(q_SF, I_SF)

generate_diffraction_pattern(*orientation: Orientation | None = None*, *ind_orientation: int | None = 0*, *orientation_matrix: ndarray | None = None*, *zone_axis_lattice: ndarray | None = None*, *proj_x_lattice: ndarray | None = None*, *foil_normal_lattice: list | tuple | ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *proj_x_cartesian: ndarray | None = None*, *foil_normal_cartesian: list | tuple | ndarray | None = None*, *sigma_excitation_error: float = 0.02*, *tol_excitation_error_mult: float = 3*, *tol_intensity: float = 0.0001*, *k_max: float | None = None*, *keep_qz=False*, *return_orientation_matrix=False*)

Generate a single diffraction pattern, return all peaks as a pointlist.

Parameters

- **orientation** (*Orientation*) – an Orientation class object
- **orientations** (*ind_orientation If input is an Orientation class object with multiple*) – this input can be used to select a specific orientation.

:param : this input can be used to select a specific orientation. :param orientation_matrix: (3,3) orientation matrix, where columns represent projection directions. :type orientation_matrix: array :param zone_axis_lattice: (3,) projection direction in lattice indices :type zone_axis_lattice: array :param proj_x_lattice: (3,) x-axis direction in lattice indices :type proj_x_lattice: array :param zone_axis_cartesian: (3,) cartesian projection direction :type zone_axis_cartesian: array :param proj_x_cartesian: (3,) cartesian projection direction :type proj_x_cartesian: array :param foil_normal: 3 element foil normal - set to None to use zone_axis :param proj_x_axis: 3 element vector defining image x axis (vertical) :type proj_x_axis: np float vector :param accel_voltage: Accelerating voltage in Volts. If not specified,

we check to see if crystal already has voltage specified.

Parameters

- **sigma_excitation_error** (*float*) – sigma value for envelope applied to s_g (excitation errors) in units of inverse Angstroms
- **tol_excitation_error_mult** (*float*) – tolerance in units of sigma for s_g inclusion
- **tol_intensity** (*np float*) – tolerance in intensity units for inclusion of diffraction spots
- **k_max** (*float*) – Maximum scattering vector
- **keep_qz** (*bool*) – Flag to return out-of-plane diffraction vectors
- **return_orientation_matrix** (*bool*) – Return the orientation matrix

Returns
list of all Bragg peaks with fields [qx, qy, intensity, h, k, l] orientation_matrix (array): 3x3 orientation matrix (optional)

Return type
bragg_peaks (*PointList*)

generate_ring_pattern(*k_max=2.0*, *use_bloch=False*, *thickness=None*, *bloch_params=None*, *orientation_plan_params=None*, *sigma_excitation_error=0.02*, *tol_intensity=0.001*, *plot_rings=True*, *plot_params={}*, *return_calc=True*)

Calculate polycrystalline diffraction pattern from structure

Parameters

- **k_max** (*float*) – Maximum scattering vector
- **use_bloch** (*bool*) – if true, use dynamic instead of kinematic approach
- **thickness** (*float*) – thickness in Ångström to evaluate diffraction patterns, only needed for dynamical calculations
- **bloch_params** (*dict*) – optional, parameters to calculate dynamical structure factor, see calculate_dynamical_structure_factors doc strings
- **orientation_plan_params** (*dict*) – optional, parameters to calculate orientation plan, see orientation_plan doc strings
- **sigma_excitation_error** (*float*) – sigma value for envelope applied to s_g (excitation errors) in units of inverse Angstroms
- **tol_intensity** (*np float*) – tolerance in intensity units for inclusion of diffraction spots
- **plot_rings** (*bool*) – if true, plot diffraction rings with plot_ring_pattern
- **return_calc** (*bool*) – return radii and intensities

Returns

radii of ring pattern in units of scattering vector k intensity_unique (np array): intensity of rings weighted by frequency of diffraciton spots

Return type

radii_unique (np array)

generate_projected_potential(*im_size=(256, 256)*, *pixel_size_angstroms=0.1*, *potential_radius_angstroms=3.0*, *sigma_image_blur_angstroms=0.1*, *thickness_angstroms=100*, *power_scale=1.0*, *plot_result=False*, *figsize=(6, 6)*, *orientation:* Orientation | *None = None*, *ind_orientation: int | None = 0*, *orientation_matrix: ndarray | None = None*, *zone_axis_lattice: ndarray | None = None*, *proj_x_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *proj_x_cartesian: ndarray | None = None*)

Generate an image of the projected potential of crystal in real space, using cell tiling, and a lookup table of the atomic potentials. Note that we round atomic positions to the nearest pixel for speed.

TODO - fix scattering prefactor so that output units are sensible.

Parameters

- **im_size** (*tuple, list, np.array*) – (2,) vector specifying the output size in pixels.
- **pixel_size_angstroms** (*float*) – Pixel size in Angstroms.
- **potential_radius_angstroms** (*float*) – Radius in Angstroms for how far to integrate the atomic potentials
- **sigma_image_blur_angstroms** (*float*) – Image blurring in Angstroms.
- **thickness_angstroms** (*float*) – Thickness of the sample in Angstroms. Set thickness_thickness_angstroms = 0 to skip thickness projection.

- **power_scale** (*float*) – Power law scaling of potentials. Set to 2.0 to approximate Z^2 images.
- **plot_result** (*bool*) – Plot the projected potential image.
- **figsize** – (2,) vector giving the size of the output.
- **orientation** (Orientation) – An Orientation class object
- **ind_orientation** (*int*) – If input is an Orientation class object with multiple orientations, this input can be used to select a specific orientation.
- **orientation_matrix** (*array*) – (3,3) orientation matrix, where columns represent projection directions.
- **zone_axis_lattice** (*array*) – (3,) projection direction in lattice indices
- **proj_x_lattice** (*array*)) – (3,) x-axis direction in lattice indices
- **zone_axis_cartesian** (*array*) – (3,) cartesian projection direction
- **proj_x_cartesian** (*array*) – (3,) cartesian projection direction

Returns
im_potential – Output image of the projected potential.

Return type
(np.array)

excitation_errors(*g*, *foil_normal=None*)

Calculate the excitation errors, assuming k0 = [0, 0, -1/lambda]. If foil normal is not specified, we assume it is [0,0,-1].

calculate_bragg_peak_histogram(*bragg_peaks*, *bragg_k_power=1.0*, *bragg_intensity_power=1.0*, *k_min=0.0*, *k_max=None*, *k_step=0.005*)

Prepare experimental bragg peaks for lattice parameter or unit cell fitting.

Parameters

- **bragg_peaks** (BraggVectors) – Input Bragg vectors.
- **bragg_k_power** (*float*) – Input Bragg peak intensities are multiplied by k**bragg_k_power to change the weighting of longer scattering vectors
- **bragg_intensity_power** (*float*) – Input Bragg peak intensities are raised power **bragg_intensity_power.
- **k_min** (*float*) – min k value for fitting range (Å^-1)
- **k_max** (*float*) – max k value for fitting range (Å^-1)
- **k_step** (*float*) – step size of k in fitting range (Å^-1)

Returns
Bragg vectors after calibration fig, ax (handles): Optional figure and axis handles, if returnfig=True.

Return type
bragg_peaks_cali (*BraggVectors*)

py4DSTEM.process.diffraction.crystal.**generate_moire_diffraction_pattern**(*bragg_peaks_0*, *bragg_peaks_1*, *thresh_0=0.0002*, *thresh_1=0.0002*, *exx_1=0.0*, *eyy_1=0.0*, *exy_1=0.0*, *phi_1=0.0*, *power=2.0*)

Calculate a Moire lattice from 2 parent diffraction patterns. The second lattice can be rotated and strained with respect to the original lattice. Note that this strain is applied in real space, and so the inverse of the calculated infinitestimal strain tensor is applied.

Parameters

- **bragg_peaks_0** (*BraggVector*) – Bragg vectors for parent lattice 0.
- **bragg_peaks_1** (*BraggVector*) – Bragg vectors for parent lattice 1.
- **thresh_0** (*float*) – Intensity threshold for structure factors from lattice 0.
- **thresh_1** (*float*) – Intensity threshold for structure factors from lattice 1.
- **exx_1** (*float*) – Strain of lattice 1 in x direction (vertical) in real space.
- **eyy_1** (*float*) – Strain of lattice 1 in y direction (horizontal) in real space.
- **exy_1** (*float*) – Shear strain of lattice 1 in (x,y) direction (diagonal) in real space.
- **phi_1** (*float*) – Rotation of lattice 1 in real space.
- **power** (*float*) – Plotting power law (default is amplitude**2.0, i.e. intensity).

Returns

parent_peaks_0, parent_peaks_1, moire_peaks – Bragg vectors for the rotated & strained parent lattices and the moire lattice

Return type

BraggVectors

py4DSTEM.process.diffraction.crystal.**plot_moire_diffraction_pattern**(*bragg_parent_0*, *bragg_parent_1*, *bragg_moire*, *int_range=(0, 0.005)*, *k_max=1.0*, *plot_subpixel=True*, *labels=None*, *marker_size_parent=16*, *marker_size_moire=4*, *text_size_parent=10*, *text_size_moire=6*, *add_labels_parent=False*, *add_labels_moire=False*, *dist_labels=0.03*, *dist_check=0.06*, *sep_labels=0.03*, *figsize=(8, 6)*, *returnfig=False*)

Plot Moire lattice and parent lattices.

Parameters

- **bragg_peaks_0** (*BraggVector*) – Bragg vectors for parent lattice 0.
- **bragg_peaks_1** (*BraggVector*) – Bragg vectors for parent lattice 1.
- **bragg_moire** (*BraggVector*) – Bragg vectors for moire lattice.
- **int_range** (*(float, float)*) – Plotting intensity range for the Moire peaks.
- **k_max** (*float*) – Max k value of the plotted Moire lattice.
- **plot_subpixel** (*bool*) – Apply subpixel corrections to the Bragg spot positions. Matplotlib default scatter plot rounds to the nearest pixel.
- **labels** (*list*) – List of text labels for parent lattices
- **marker_size_parent** (*float*) – Size of plot markers for the two parent lattices.
- **marker_size_moire** (*float*) – Size of plot markers for the Moire lattice.
- **text_size_parent** (*float*) – Label text size for parent lattice.
- **text_size_moire** (*float*) – Label text size for Moire lattice.
- **add_labels_parent** (*bool*) – Plot the parent lattice index labels.
- **add_labels_moire** (*bool*) – Plot the parent lattice index labels for the Moire spots.
- **dist_labels** (*float*) – Distance to move the labels off the spots.
- **dist_check** (*float*) – Set to some distance to "push" the labels away from each other if they are within this distance.
- **sep_labels** (*float*) – Separation distance for labels which are "pushed" apart.
- **figsize** (*(float,float)*) – Size of output figure.
- **returnfig** (*bool*) – Return the (fix,ax) handles of the plot.

Returns

fig, ax – Figure and axes handles for the moire plot.

Return type

matplotlib handles (optional)

py4DSTEM.process.diffraction.crystal_ACOM.**orientation_plan**(*self*, *zone_axis_range: ndarray = array([[0, 1, 1], [1, 1, 1]])*, *angle_step_zone_axis: float = 2.0*, *angle_coarse_zone_axis: float | None = None*, *angle_refine_range: float | None = None*, *angle_step_in_plane: float = 2.0*, *accel_voltage: float = 300000.0*, *corr_kernel_size: float = 0.08*, *radial_power: float = 1.0*, *intensity_power: float = 0.25*, *calculate_correlation_array=True*, *tol_peak_delete=None*, *tol_distance: float = 0.01*, *fiber_axis=None*, *fiber_angles=None*, *figsize: list | tuple | ndarray = (6, 6)*, *CUDA: bool = False*, *progress_bar: bool = True*)

Calculate the rotation basis arrays for an SO(3) rotation correlogram.

Parameters

- **zone_axis_range** (*float*) – Row vectors give the range for zone axis orientations. If user specifies 2 vectors (2x3 array), we start at [0,0,1]

  to make z-x-z rotation work.

  If user specifies 3 vectors (3x3 array), plan will span these vectors. Setting to 'full' as a string will use a hemispherical range. Setting to 'half' as a string will use a quarter sphere range. Setting to 'fiber' as a string will make a spherical cap around a given vector. Setting to 'auto' will use pymatgen to determine the point group symmetry

  of the structure and choose an appropriate zone_axis_range

- **angle_step_zone_axis** (*float*) – Approximate angular step size for zone axis search [degrees]
- **angle_coarse_zone_axis** (*float*) – Coarse step size for zone axis search [degrees]. Setting to None uses the same value as angle_step_zone_axis.
- **angle_refine_range** (*float*) – Range of angles to use for zone axis refinement. Setting to None uses same value as angle_coarse_zone_axis.
- **angle_step_in_plane** (*float*) – Approximate angular step size for in-plane rotation [degrees]
- **accel_voltage** (*float*) – Accelerating voltage for electrons [Volts]
- **corr_kernel_size** (*float*) – Correlation kernel size length. The size of the overlap kernel between the measured Bragg peaks and diffraction library Bragg peaks. [1/Angstroms]
- **radial_power** (*float*) – Power for scaling the correlation intensity as a function of the peak radius
- **intensity_power** (*float*) – Power for scaling the correlation intensity as a function of the peak intensity
- **calculate_correlation_array** (*bool*) – Set to false to skip calculating the correlation array. This is useful when we only want the angular range / rotation matrices.
- **tol_peak_delete** (*float*) – Distance to delete peaks for multiple matches. Default is kernel_size * 0.5
- **tol_distance** (*float*) – Distance tolerance for radial shell assignment [1/Angstroms]
- **fiber_axis** (*float*) – (3,) vector specifying the fiber axis
- **fiber_angles** (*float*) – (2,) vector specifying angle range from fiber axis, and in-plane angular range [degrees]
- **cartesian_directions** (*bool*) – When set to true, all zone axes and projection directions are specified in Cartesian directions.
- **figsize** (*float*) – (2,) vector giving the figure size
- **CUDA** (*bool*) – Use CUDA for the Fourier operations.
- **progress_bar** (*bool*) – If false no progress bar is displayed

py4DSTEM.process.diffraction.crystal_ACOM.**match_orientations**(*self*, *bragg_peaks_array: PointListArray*, *num_matches_return: int = 1*, *min_angle_between_matches_deg=None*, *min_number_peaks: int = 3*, *inversion_symmetry: bool = True*, *multiple_corr_reset: bool = True*, *return_orientation: bool = True*, *progress_bar: bool = True*)

Parameters

- **bragg_peaks_array** (PointListArray) – PointListArray containing the Bragg peaks and intensities, with calibrations applied
- **num_matches_return** (*int*) – return these many matches as 3th dim of orient (matrix)
- **min_angle_between_matches_deg** (*int*) – Minimum angle between zone axis of multiple matches, in degrees. Note that I haven't thought how to handle in-plane rotations, since multiple matches are possible.
- **min_number_peaks** (*int*) – Minimum number of peaks required to perform ACOM matching
- **inversion_symmetry** (*bool*) – check for inversion symmetry in the matches
- **multiple_corr_reset** (*bool*) – keep original correlation score for multiple matches
- **return_orientation** (*bool*) – Return orientation map from function for inspection. The map is always stored in the Crystal object.
- **progress_bar** (*bool*) – Show or hide the progress bar

py4DSTEM.process.diffraction.crystal_ACOM.**match_single_pattern**(*self*, *bragg_peaks:* PointList, *num_matches_return: int = 1*, *min_angle_between_matches_deg=None*, *min_number_peaks=3*, *inversion_symmetry=True*, *multiple_corr_reset=True*, *plot_polar: bool = False*, *plot_corr: bool = False*, *returnfig: bool = False*, *figsize: list | tuple | ndarray = (12, 4)*, *verbose: bool = False*)

Solve for the best fit orientation of a single diffraction pattern.

Parameters

- **bragg_peaks** (PointList) – numpy array containing the Bragg positions and intensities ('qx', 'qy', 'intensity')
- **num_matches_return** (*int*) – return these many matches as 3th dim of orient (matrix)
- **min_angle_between_matches_deg** (*int*) – Minimum angle between zone axis of multiple matches, in degrees. Note that I haven't thought how to handle in-plane rotations, since multiple matches are possible.
- **min_number_peaks** (*int*) – Minimum number of peaks required to perform ACOM matching
- **bool** (*multiple_corr_reset*) – check for inversion symmetry in the matches
- **bool** – keep original correlation score for multiple matches
- **subpixel_tilt** (*bool*) – set to false for faster matching, returning the nearest corr point
- **plot_polar** (*bool*) – set to true to plot the polar transform of the diffraction pattern
- **plot_corr** (*bool*) – set to true to plot the resulting correlogram
- **returnfig** (*bool*) – return figure handles
- **figsize** (*list*) – size of figure

- **verbose** (*bool*) – Print the fitted zone axes, correlation scores
- **CUDA** (*bool*) – Enable CUDA for the FFT steps

Returns

- **orientation** (*Orientation*) – Orientation class containing all outputs
- **fig, ax** (*handles*) – Figure handles for the plotting output

py4DSTEM.process.diffraction.crystal_ACOM.**cluster_grains**(*self*, *threshold_add=1.0*, *threshold_grow=0.1*, *angle_tolerance_deg=5.0*, *progress_bar=True*)

Cluster grains using rotation criterion, and correlation values.

Parameters

- **threshold_add** (*float*) – Minimum signal required for a probe position to initialize a cluster.
- **threshold_grow** (*float*) – Minimum signal required for a probe position to be added to a cluster.
- **angle_tolerance_deg** (*float*) – Rotation rolerance for clustering grains.
- **progress_bar** (*bool*) – Turns on the progress bar for the polar transformation

py4DSTEM.process.diffraction.crystal_ACOM.**cluster_orientation_map**(*self*, *stripe_width=(2, 2)*, *area_min=2*)

Produce a new orientation map from the clustered grains. Use a stripe pattern for the overlapping grains.

Parameters

- **stripe_width** (*(int,int)*) – Width of stripes for plotting maps with overlapping grains
- **area_min** (*(int)*) – Minimum size of grains to include

Returns
The clustered orientation map

Return type
orientation_map

py4DSTEM.process.diffraction.crystal_ACOM.**calculate_strain**(*self*, *bragg_peaks_array: PointListArray*, *orientation_map: OrientationMap*, *corr_kernel_size=None*, *sigma_excitation_error=0.02*, *tol_excitation_error_mult: float = 3*, *tol_intensity: float = 0.0001*, *k_max: float | None = None*, *min_num_peaks=5*, *intensity_weighting=False*, *robust=True*, *robust_thresh=3.0*, *rotation_range=None*, *mask_from_corr=True*, *corr_range=(0, 2)*, *corr_normalize=True*, *progress_bar=True*)

This function takes in both a PointListArray containing Bragg peaks, and a corresponding OrientationMap, and uses least squares to compute the deformation tensor which transforms the simulated diffraction pattern into the experimental pattern, for all probe positons.

TODO: add robust fitting?

Parameters

- **(PointListArray)** (*bragg_peaks_array*) – All Bragg peaks
- **(OrientationMap)** (*orientation_map*) – Orientation map generated from ACOM
- **(float)** (*rotation_range*) – Correlation kernel size - if user does not specify, uses self.corr_kernel_size.
- **(float)** – sigma value for envelope applied to s_g (excitation errors) in units of inverse Angstroms
- **(float)** – tolerance in units of sigma for s_g inclusion
- **float)** (*tol_intensity (np*) – tolerance in intensity units for inclusion of diffraction spots
- **(float)** – Maximum scattering vector
- **(int)** (*min_num_peaks*) – Minimum number of peaks required.
- **intensity_weighting** (*bool*) – Set to True to weight least squares by experimental peak intensity.
- **robust_fitting** (*bool*) – Set to True to use robust fitting, which performs outlier rejection.
- **robust_thresh** (*float*) – Threshold for robust fitting weights.
- **(float)** – Maximum rotation range in radians (for symmetry reduction).
- **(bool)** (*corr_normalize*) – Show progress bar
- **(bool)** – Use ACOM correlation signal for mask
- **(np.ndarray)** (*corr_range*) – Range of correlation signals for mask
- **(bool)** – Normalize correlation signal before masking

Returns

strain tensor

Return type

strain_map (*RealSlice*)

py4DSTEM.process.diffraction.crystal_ACOM.**save_ang_file**(*self*, *file_name*, *orientation_map*, *ind_orientation=0*, *pixel_size=1.0*, *pixel_units='px'*, *transpose_xy=True*, *flip_x=False*, *flip_y=False*)

This function outputs an ascii text file in the .ang format, containing the Euler angles of an orientation map.

Parameters

- **file_name** (*str*) – Path to save .ang file.
- **orientation_map** (OrientationMap) – Class containing orientation matrices, correlation values, etc.
- **ind_orientation** (*int*) – Which orientation match to plot if num_matches > 1
- **pixel_size** (*float*) – Pixel size, if known.
- **pixel_units** (*str*) – Units of the pixel size
- **transpose_xy** (*bool*) – Transpose x and y pixel coordinates.

- **flip_x** (*bool*) – Swap x direction pixels (after transpose).

Returns
nothing

py4DSTEM.process.diffraction.crystal_ACOM.**symmetry_reduce_directions**(*self*, *orientation*, *match_ind=0*, *plot_output=False*, *figsize=(15, 6)*, *el_shift=0.0*, *az_shift=-30.0*)

This function calculates the symmetry-reduced cartesian directions from and orientation matrix stored in orientation.matrix, and outputs them into orientation.family. It optionally plots the 3D output.

class py4DSTEM.process.diffraction.crystal_bloch.**DynamicalMatrixCache**(*has_valid_cache: bool = False*, *cached_U_gmh: <built-in function array> = None*)

__init__(*has_valid_cache: bool = False*, *cached_U_gmh: array | None = None*) → None

py4DSTEM.process.diffraction.crystal_bloch.**calculate_dynamical_structure_factors**(*self*, *accelerating_voltage: float*, *method: str = 'WK-CP'*, *k_max: float = 2.0*, *thermal_sigma: float | dict | None = None*, *tol_structure_factor: float = 0.0*, *recompute_kinematic_structure_fac g_vec_precision=None*)

Calculate and store the relativistic corrected structure factors used for Bloch computations in a dictionary for faster lookup.

Parameters

- **accelerating_voltage** (*float*) – accelerating voltage in eV
- **method** (*str*) – Choose which parameterization of the structure factors to use: "Lobato": Uses the kinematic structure factors from crystal.py, using the parameterization from

  Lobato & Van Dyck, Acta Cryst A 70:6 (2014)

  "Lobato-absorptive": Lobato factors plus an imaginary part
  equal to 0.1•f, as a simple but inaccurate way to include absorption, per Hashimoto, Howie, & Whelan, Proc R Soc Lond A 269:80-103 (1962)

"WK": Uses the Weickenmeier-Kohl parameterization for
the elastic form factors, including Debye-Waller factor, with no absorption, as described in Weickenmeier & Kohl, Acta Cryst A 47:5 (1991)

"WK-C": WK form factors plus the "core" contribution to absorption
following H. Rose, Optik 45:2 (1976)

"WK-P": WK form factors plus the phonon/TDS absorptive contribution "WK-CP": WK form factors plus core and phonon absorption (default)

- **k_max** (*float*) – max scattering length to compute structure factors to. Setting this to 2x the k_max used in generating the beamsn included in a simulation will retain all possible couplings
- **thermal_sigma** (*float or dict{int->float}*) – RMS atomic diplacement for attenuating form factors to account for thermal broadening of the potential, only used when a "WK" method is selected. Required when WK-P or WK-CP are selected. Units are Å. (This is often written as ⟨u⟩in papers) To specify different ⟨u⟩ for each element, pass a dictionary with Z as the key, mapping to the appropriate float value
- **tol_structure_factor** (*float*) – tolerance for removing low-valued structure factors. Reflections with structure factor below the tolerance will have zero coupling in the dynamical calculations (i.e. they are the ignored weak beams)
- **recompute_kinematic_structure_factors** (*bool*) – When True, recomputes the kinematic structure factors using the same tol_structure_factor, and with k_max set to *half* the k_max for the dynamical factors. The factor of half ensures that every beam in a simulation can couple to every other beam (no high-angle couplings in the Bloch matrix are set to zero.)
- **g_vec_precision** (*optional int*) – If specified, rounds **|g|** to this many decimal places so that automatic caching of the atomic form factors is not slowed down due to floating point errors. Setting this to 3 can give substantial speedup at the cost of some reduced accuracy
- **factors.** (*See WK_scattering_factors.py for details on the Weickenmeier-Kohl form*) –

py4DSTEM.process.diffraction.crystal_bloch.**generate_dynamical_diffraction_pattern**(*self*, *beams: PointList*, *thickness: float | list | tuple | ndarray*, *zone_axis_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *foil_normal_lattice: ndarray | None = None*, *foil_normal_cartesian: ndarray | None = None*, *verbose: bool = False*, *always_return_list: bool = False*, *dynamical_matrix_cache: DynamicalMatrixCache | None = None*, *return_complex: bool = False*, *return_eigenvectors: bool = False*, *return_Smatrix: bool = False*) → *PointList* | List[*PointList*]

Generate a dynamical diffraction pattern (or thickness series of patterns) using the Bloch wave method.

The beams to be included in the Bloch calculation must be pre-calculated and passed as a PointList containing at least (qx, qy, h, k, l) fields.

If `thickness` is a single value, one new PointList will be returned. If `thickness` is a sequence of values, a list

of PointLists will be returned,

corresponding to each thickness value in the input.

Frequent reference will be made to "Introduction to conventional transmission electron microscopy"
by DeGraef, whose overall approach we follow here.

Parameters

- **beams** (PointList) – PointList from the kinematical diffraction generator which will define the beams included in the Bloch calculation
- **thickness** (*float or list/array*) – The main Bloch calculation can be reused for multiple thicknesses without much overhead.
- **direction.** (*zone_axis & foil_normal Incident beam orientation and foil normal*) – Each can be specified in the Cartesian or crystallographic basis, using e.g. zone_axis_lattice or zone_axis_cartesian. These are internally parsed by Crystal.parse_orientation

Less commonly used args:

always_return_list (bool): When True, the return is always a list of PointLists,
even for a single thickness

dynamical_matrix_cache: (DyanmicalMatrixCache) Dataclass used for caching of the
dynamical matrix. If the cached matrix does not exist, it is computed and stored. Subsequent calls will use the cached matrix for the off-diagonal components of the A matrix and overwrite the diagonal elements. This is used for CBED calculations.

return_complex (bool): When True, returns both the complex amplitude and intensity. Defaults to (False)

Returns

Bragg peaks with fields [qx, qy, intensity, h, k, l]
or

[bragg_peaks,...] (PointList): If thickness is a list/array, or always_return_list is True,
a list of PointLists is returned.

if return_complex = True:

bragg_peaks (PointList): Bragg peaks with fields [qx, qy, intensity, amplitude, h, k, l]
or

[bragg_peaks,...] (PointList): If thickness is a list/array, or always_return_list is True,
a list of PointLists is returned.

if return_Smatrix = True:

[S_matrix, ...], psi_0: Returns a list of S-matrices for each thickness (this is always a list),
and the vector representing the incident plane wave. The beams of the S-matrix have the same order as in the input *beams*.

Return type
bragg_peaks (*PointList*)

py4DSTEM.process.diffraction.crystal_bloch.**generate_CBED**(*self*, *beams: ~emdfile.classes.pointlist.PointList*, *thickness: float | list | tuple | ~numpy.ndarray*, *alpha_mrad: float*, *pixel_size_inv_A: float*, *DP_size_inv_A: float | None = None*, *zone_axis_lattice: ~numpy.ndarray | None = None*, *zone_axis_cartesian: ~numpy.ndarray | None = None*, *foil_normal_lattice: ~numpy.ndarray | None = None*, *foil_normal_cartesian: ~numpy.ndarray | None = None*, *LACBED: bool = False*, *dtype: ~numpy.dtype = <class 'numpy.float32'>*, *verbose: bool = False*, *progress_bar: bool = True*, *return_mask: bool = False*, *two_beam_zone_axis_lattice: ~numpy.ndarray | None = None*, *return_probe: bool = False*) → ndarray | List[ndarray] | Dict[Tuple[int], ndarray]

Generate a dynamical CBED pattern using the Bloch wave method.

Parameters

- **beams** (PointList) – PointList from the kinematical diffraction generator which will define the beams included in the Bloch calculation
- **thickness** (*float or list/array*) – The main Bloch calculation can be reused for multiple thicknesses without much overhead.
- **alpha_mrad** (*float*) – Convergence angle for CBED pattern. Note that if disks in the calculation overlap, they will be added incoherently, and the resulting CBED will thus represent the average over the unit cell (i.e. a PACBED pattern, as described in LeBeau et al., Ultramicroscopy 110(2): 2010.)
- **pixel_size_inv_A** (*float*) – CBED pixel size in 1/Å.
- **DP_size_inv_A** (*optional float*) – If specified, defines the extents of the diffraction pattern. If left unspecified, the DP will be automatically scaled to fit all of the beams present in the input plus some small buffer.
- **zone_axis** (*np float vector*) – 3 element projection direction for sim pattern Can also be a 3x3 orientation matrix (zone axis 3rd column)
- **foil_normal** – 3 element foil normal - set to None to use zone_axis
- **LACBED** (*bool*) – keyed by tuples of (h,k,l).
- **proj_x_axis** (*np float vector*) – 3 element vector defining image x axis (vertical)
- **PointList** (*two_beam_zone_axis_lattice When only two beams are present in the "beams"*) – the computation of the projected crystallographic directions becomes ambiguous. In this case, you must specify the indices of the zone axis used to generate the beams.

:param

[the computation of the projected crystallographic directions] becomes ambiguous. In this case, you must specify the indices of the zone axis used to generate the beams.

Parameters

return_probe (*bool*) – If True, the probe (np.ndarray) will be returned in additon to the CBED

Returns

CBED pattern as np.ndarray If thickness is a sequence: CBED patterns for each thickness value as a list of np.ndarrays If LACBED is True and thickness is scalar: Dictionary with tuples of ints (h,k,l) as keys, mapping to np.ndarray. If LACBED is True and thickness is a sequence: List of dictionaries, structured as above. If return_probe is True: will return a tuple (<CBED/LACBED object>, Probe)

Return type

If thickness is a scalar

py4DSTEM.process.diffraction.crystal_calibrate.**calibrate_pixel_size**(*self*, *bragg_peaks*, *scale_pixel_size=1.0*, *bragg_k_power=1.0*, *bragg_intensity_power=1.0*, *k_min=0.0*, *k_max=None*, *k_step=0.002*, *k_broadening=0.002*, *fit_all_intensities=False*, *set_calibration_in_place=False*, *verbose=True*, *plot_result=False*, *figsize: list | tuple | ndarray = (12, 6)*, *returnfig=False*)

Use the calculated structure factor scattering lengths to compute 1D diffraction patterns, and solve the best-fit relative scaling between them. Returns the fit pixel size in Å^-1.

Parameters

- **bragg_peaks** (BraggVectors) – Input Bragg vectors.
- **scale_pixel_size** (*float*) – Initial guess for scaling of the existing pixel size If the pixel size is currently uncalibrated, this is a guess of the pixel size in Å^-1. If the pixel size is already (approximately) calibrated, this is the scaling factor to correct that existing calibration.
- **bragg_k_power** (*float*) – Input Bragg peak intensities are multiplied by k**bragg_k_power to change the weighting of longer scattering vectors
- **bragg_intensity_power** (*float*) – Input Bragg peak intensities are raised power **bragg_intensity_power.
- **k_min** (*float*) – min k value for fitting range (Å^-1)
- **k_max** (*float*) – max k value for fitting range (Å^-1)
- **k_step** (*float) step size of k in fitting range (Å^-1*) –
- **k_broadening** (*float*) – Initial guess for Gaussian broadening of simulated pattern (Å^-1)
- **fit_all_intensities** (*bool*) – Set to true to allow all peak intensities to change independently. False forces a single intensity scaling for all peaks.
- **set_calibration** (*bool*) – if True, set the fit pixel size to the calibration metadata, and calibrate bragg_peaks
- **verbose** (*bool*) – Output the calibrated pixel size.

- **plot_result** (*bool*) – Plot the resulting fit.
- **figsize** (*list, tuple, np.ndarray*) – Figure size of the plot.
- **returnfig** (*bool*) – Return handles figure and axis

Returns
fig, ax – Figure and axis handles, if returnfig=True.

Return type
handles, optional

py4DSTEM.process.diffraction.crystal_calibrate.**calibrate_unit_cell**(*self*, *bragg_peaks*, *coef_index=None*, *coef_update=None*, *bragg_k_power=1.0*, *bragg_intensity_power=1.0*, *k_min=0.0*, *k_max=None*, *k_step=0.005*, *k_broadening=0.02*, *fit_all_intensities=True*, *verbose=True*, *plot_result=False*, *figsize: list | tuple | ndarray = (12, 6)*, *returnfig=False*)

Solve for the best fit scaling between the computed structure factors and bragg_peaks.

Parameters

- **bragg_peaks** (BraggVectors) – Input Bragg vectors.
- **coef_index** (*list of ints*) – List of ints that act as pointers to unit cell parameters and angles to update.
- **coef_update** (*list of bool*) – List of booleans to indicate whether or not to update the cell at that position
- **bragg_k_power** (*float*) – Input Bragg peak intensities are multiplied by k**bragg_k_power to change the weighting of longer scattering vectors
- **bragg_intensity_power** (*float*) – Input Bragg peak intensities are raised power **bragg_intensity_power.
- **k_min** (*float*) – min k value for fitting range (Å^-1)
- **k_max** (*float*) – max k value for fitting range (Å^-1)
- **k_step** (*float*) – step size of k in fitting range (Å^-1)
- **k_broadening** (*float*) – Initial guess for Gaussian broadening of simulated pattern (Å^-1)
- **fit_all_intensities** (*bool*) – Set to true to allow all peak intensities to change independently False forces a single intensity scaling.
- **verbose** (*bool*) – Output the calibrated pixel size.
- **plot_result** (*bool*) – Plot the resulting fit.
- **figsize** (*list, tuple, np.ndarray*) –
- **returnfig** (*bool*) – Return handles figure and axis

Returns
Optional figure and axis handles, if returnfig=True.

Return type
fig, ax (handles)

Details: User has the option to define what is allowed to update in the unit cell using the arguments coef_index and coef_update. Each has 6 entries, corresponding to the a, b, c, alpha, beta, gamma parameters of the unit cell, in this order. The coef_update argument is a list of bools specifying whether or not the unit cell value will be allowed to change (True) or must maintain the original value (False) upon fitting. The coef_index argument provides a pointer to the index in which the code will update to.

For example, to update a, b, c, alpha, beta, gamma all independently of eachother, the following arguments should be used:

coef_index = [0, 1, 2, 3, 4, 5] coef_update = [True, True, True, True, True, True,]

The default is set to automatically define what can update in a unit cell based on the point group constraints. When either 'coef_index' or 'coef_update' are None, these constraints will be automatically pulled from the pointgroup.

For example, the default for cubic unit cells is:
coef_index = [0, 0, 0, 3, 3, 3] coef_update = [True, True, True, False, False, False]

Which allows a, b, and c to update (True in first 3 indices of coef_update) but b and c update based on the value of a (0 in the 1 and 2 list entries in coef_index) such that a = b = c. While coef_update is False for alpha, beta, and gamma (entries 3, 4, 5), no updates will be made to the angles.

The user has the option to predefine coef_index or coef_update to override defaults. In the coef_update list, there must be 6 entries and each are boolean. In the coef_index list, there must be 6 entries, with the first 3 entries being between 0 - 2 and the last 3 entries between 3 - 5. These act as pointers to pull the updated parameter from.

class py4DSTEM.process.diffraction.crystal_phase.**Crystal_Phase**(*crystals*, *orientation_maps*, *name*)

A class storing multiple crystal structures, and associated diffraction data. Must be initialized after matching orientations to a pointlistarray???

__init__(*crystals*, *orientation_maps*, *name*)

Parameters

- **crystals** (*list*) – List of crystal instances
- **orientation_maps** (*list*) – List of orientation maps
- **name** (*str*) – Name of Crystal_Phase instance

plot_all_phase_maps(*map_scale_values=None*, *index=0*)

Visualize phase maps of dataset.

Parameters
map_scale_values (*float*) – Value to scale correlations by

quantify_phase(*pointlistarray*, *tolerance_distance=0.08*, *method='nnls'*, *intensity_power=0*, *mask_peaks=None*)

Quantification of the phase of a crystal based on the crystal instances and the pointlistarray.

Parameters

- **pointlisarray** (*pointlistarray*) – Pointlistarray to quantify phase of
- **tolerance_distance** (*float*) – Distance allowed between a peak and match
- **method** (*str*) – Numerical method used to quantify phase

- **intensity_power** (*float*) – ...
- **mask_peaks** (*list, optional*) – A pointer of which positions to mask peaks from

Details:

quantify_phase_pointlist(*pointlistarray*, *position*, *method='nnls'*, *tolerance_distance=0.08*, *intensity_power=0*, *mask_peaks=None*)

Parameters

- **pointlisarray** (*pointlistarray*) – Pointlistarray to quantify phase of
- **position** (*tuple/list*) – Position of pointlist in pointlistarray
- **tolerance_distance** (*float*) – Distance allowed between a peak and match
- **method** (*str*) – Numerical method used to quantify phase
- **intensity_power** (*float*) – ...
- **mask_peaks** (*list, optional*) – A pointer of which positions to mask peaks from

Returns

Peak matches in the rows of array and the crystals in the columns phase_weights (np.ndarray): Weights of each phase phase_residuals (np.ndarray): Residuals crystal_identity (list): List of lists, where the each entry represents the position in the

crystal and orientation match that is associated with the phase weights. for example, if the output was [[0,0], [0,1], [1,0], [0,1]], the first entry [0,0] in phase weights is associated with the first crystal the first match within that crystal. [0,1] is the first crystal and the second match within that crystal.

Return type

pointlist_peak_intensity_matches (np.ndarray)

py4DSTEM.process.diffraction.crystal_viz.**plot_structure**(*self*, *orientation_matrix: ndarray | None = None*, *zone_axis_lattice: ndarray | None = None*, *proj_x_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *proj_x_cartesian: ndarray | None = None*, *size_marker: float = 400*, *tol_distance: float = 0.001*, *plot_limit: ndarray | None = None*, *camera_dist: float | None = None*, *show_axes: bool = False*, *perspective_axes: bool = True*, *figsize: tuple | list | ndarray = (8, 8)*, *returnfig: bool = False*)

Quick 3D plot of the untit cell /atomic structure.

Parameters

- **orientation_matrix** (*array*) – (3,3) orientation matrix, where columns represent projection directions.
- **zone_axis_lattice** (*array*) – (3,) projection direction in lattice indices
- **proj_x_lattice** (*array*) – (3,) x-axis direction in lattice indices
- **zone_axis_cartesian** (*array*) – (3,) cartesian projection direction
- **proj_x_cartesian** (*array*) – (3,) cartesian projection direction

- **scale_markers** (*float*) – Size scaling for markers
- **tol_distance** (*float*) – Tolerance for repeating atoms on edges on cell boundaries.
- **plot_limit** (*float*) – (2,3) numpy array containing x y z plot min and max in columns. Default is 1.1* unit cell dimensions.
- **camera_dist** (*float*) – Move camera closer to the plot (relative to matplotlib default of 10)
- **show_axes** (*bool*) – Whether to plot axes or not.
- **perspective_axes** (*bool*) – Select either perspective (true) or orthogonal (false) axes
- **figsize** (*2 element float*) – Size scaling of figure axes.
- **returnfig** (*bool*) – Return figure and axes handles.

Returns

fig, ax (optional) figure and axes handles

py4DSTEM.process.diffraction.crystal_viz.**plot_structure_factors**(*self*, *orientation_matrix: ndarray | None = None*, *zone_axis_lattice: ndarray | None = None*, *proj_x_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *proj_x_cartesian: ndarray | None = None*, *scale_markers: float = 1000.0*, *plot_limit: list | tuple | ndarray | None = None*, *camera_dist: float | None = None*, *show_axes: bool = True*, *perspective_axes: bool = True*, *figsize: list | tuple | ndarray = (8, 8)*, *returnfig: bool = False*)

3D scatter plot of the structure factors using magnitude^2, i.e. intensity.

Parameters

- **orientation_matrix** (*array*) – (3,3) orientation matrix, where columns represent projection directions.
- **zone_axis_lattice** (*array*) – (3,) projection direction in lattice indices
- **proj_x_lattice** (*array*) – (3,) x-axis direction in lattice indices
- **zone_axis_cartesian** (*array*) – (3,) cartesian projection direction
- **proj_x_cartesian** (*array*) – (3,) cartesian projection direction
- **scale_markers** (*float*) – size scaling for markers
- **plot_limit** (*float*) – x y z plot limits, default is [-1 1]*self.k_max
- **camera_dist** (*float*) – Move camera closer to the plot (relative to matplotlib default of 10)
- **show_axes** (*bool*) – Whether to plot axes or not.
- **perspective_axes** (*bool*) – Select either perspective (true) or orthogonal (false) axes
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns
fig, ax (optional) figure and axes handles

py4DSTEM.process.diffraction.crystal_viz.**plot_scattering_intensity**(*self*, *k_min=0.0*, *k_max=None*, *k_step=0.001*, *k_broadening=0.0*, *k_power_scale=0.0*, *int_power_scale=0.5*, *int_scale=1.0*, *remove_origin=True*, *bragg_peaks=None*, *bragg_k_power=0.0*, *bragg_intensity_power=1.0*, *bragg_k_broadening=0.005*, *figsize: list | tuple | ndarray = (10, 4)*, *returnfig: bool = False*)

1D plot of the structure factors

Parameters

- **k_min** (*float*) – min k value for profile range.
- **k_max** (*float*) – max k value for profile range.
- **k_step** (*float*) – Step size of k in profile range.
- **k_broadening** (*float*) – Broadening of simulated pattern.
- **k_power_scale** (*float*) – Scale SF intensities by k**k_power_scale.
- **int_power_scale** (*float*) – Scale SF intensities**int_power_scale.
- **int_scale** (*float*) – Scale output profile by this value.
- **remove_origin** (*bool*) – Remove origin from plot.
- **bragg_peaks** (BraggVectors) – Passed in bragg_peaks for comparison with simulated pattern.
- **bragg_k_power** (*float*) – bragg_peaks scaled by k**bragg_k_power.
- **bragg_intensity_power** (*float*) – bragg_peaks scaled by intensities**bragg_intensity_power.
- **bragg_k_broadening** (*float*) – Broadening applied to bragg_peaks.
- **figsize** (*list, tuple, np.ndarray*) – Figure size for plot.
- **(bool)** (*returnfig*) – Return figure and axes handles if this is True.

Returns
figure and axes handles

Return type
fig, ax (optional)

py4DSTEM.process.diffraction.crystal_viz.**plot_orientation_zones**(*self*, *azim_elev: list | tuple | ndarray | None = None*, *proj_dir_lattice: list | tuple | ndarray | None = None*, *proj_dir_cartesian: list | tuple | ndarray | None = None*, *tol_den=10*, *marker_size: float = 20*, *plot_limit: list | tuple | ndarray = array([-1.1, 1.1])*, *figsize: list | tuple | ndarray = (8, 8)*, *returnfig: bool = False*)

3D scatter plot of the structure factors using magnitude^2, i.e. intensity.

Parameters

- **azim_elev** (*array*) – az and el angles for plot
- **proj_dir_lattice** (*array*) – (3,) projection direction in lattice
- **proj_dir_cartesian** – (array): (3,) projection direction in cartesian
- **tol_den** (*int*) – tolerance for rational index denominator
- **dir_proj** (*float*) – projection direction, either [elev azim] or normal vector Default is mean vector of self.orientation_zone_axis_range rows
- **marker_size** (*float*) – size of markers
- **plot_limit** (*float*) – x y z plot limits, default is [0, 1.05]
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns

fig, ax (optional) figure and axes handles

py4DSTEM.process.diffraction.crystal_viz.**plot_orientation_plan**(*self*, *index_plot: int = 0*, *zone_axis_lattice: ndarray | None = None*, *zone_axis_cartesian: ndarray | None = None*, *figsize: list | tuple | ndarray = (14, 6)*, *returnfig: bool = False*)

3D scatter plot of the structure factors using magnitude^2, i.e. intensity.

Parameters

- **index_plot** (*int*) – which index slice to plot
- **zone_axis_plot** (*3 element float*) – which zone axis slice to plot
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns

fig, ax (optional) figure and axes handles

py4DSTEM.process.diffraction.crystal_viz.**plot_diffraction_pattern**(*bragg_peaks: PointList*, *bragg_peaks_compare: PointList | None = None*, *scale_markers: float = 500*, *scale_markers_compare: float | None = None*, *power_markers: float = 1*, *plot_range_kx_ky: list | tuple | ndarray | None = None*, *add_labels: bool = True*, *shift_labels: float = 0.08*, *shift_marker: float = 0.005*, *min_marker_size: float = 1e-06*, *max_marker_size: float = 1000*, *figsize: list | tuple | ndarray = (12, 6)*, *returnfig: bool = False*, *input_fig_handle=None*)

2D scatter plot of the Bragg peaks

Parameters

- **bragg_peaks** (*PointList*) – numpy array containing (‘qx’, ‘qy’, ‘intensity’, ‘h’, ‘k’, ‘l’)
- **bragg_peaks_compare** (*PointList*) – numpy array containing (‘qx’, ‘qy’, ‘intensity’)
- **scale_markers** (*float*) – size scaling for markers
- **scale_markers_compare** (*float*) – size scaling for markers of comparison
- **power_markers** (*float*) – power law scaling for marks (default is 1, i.e. amplitude)
- **plot_range_kx_ky** (*float*) – 2 element numpy vector giving the plot range
- **add_labels** (*bool*) – flag to add hkl labels to peaks
- **min_marker_size** (*float*) – minimum marker size for the comparison peaks
- **max_marker_size** (*float*) – maximum marker size for the comparison peaks
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles
- **input_fig_handle** (*fig,ax*) –

py4DSTEM.process.diffraction.crystal_viz.**plot_orientation_maps**(*self*, *orientation_map=None*, *orientation_ind: int = 0*, *dir_in_plane_degrees: float = 0.0*, *corr_range: ndarray = array([0, 5])*, *corr_normalize: bool = True*, *scale_legend: bool | None = None*, *figsize: list | tuple | ndarray = (16, 5)*, *figbound: list | tuple | ndarray = (0.01, 0.005)*, *show_axes: bool = True*, *camera_dist=None*, *plot_limit=None*, *plot_layout=0*, *swap_axes_xy_limits=False*, *returnfig: bool = False*, *progress_bar=False*)

Plot the orientation maps.

Parameters

- **orientation_map** (OrientationMap) – Class containing orientation matrices, correlation values, etc. Optional - can reference internally stored OrientationMap.
- **orientation_ind** (*int*) – Which orientation match to plot if num_matches > 1
- **dir_in_plane_degrees** (*float*) – In-plane angle to plot in degrees. Default is 0 / x-axis / vertical down.
- **corr_range** (*np.ndarray*) – Correlation intensity range for the plot
- **corr_normalize** (*bool*) – If true, set mean correlation to 1.
- **scale_legend** (*float*) – 2 elements, x and y scaling of legend panel
- **figsize** (*array*) – 2 elements defining figure size
- **figbound** (*array*) – 2 elements defining figure boundary
- **show_axes** (*bool*) – Flag setting whether orienation map axes are visible.
- **camera_dist** (*float*) – distance of camera from legend
- **plot_limit** (*array*) – 2x3 array defining plot boundaries of legend
- **plot_layout** (*int*) – subplot layout: 0 - 1 row, 3 col 1 - 3 row, 1 col
- **swap_axes_xy_limits** (*bool*) – swap x and y boundaries for legend (not sure why we need this in some cases)
- **returnfig** (*bool*) – set to True to return figure and axes handles
- **progress_bar** (*bool*) – Enable progressbar when calculating orientation images.

Returns

RGB images fig, axs (handles): Figure and axes handes for the

Return type

images_orientation (int)

Note: Currently, no symmetry reduction. Therefore the x and y orientations are going to be correct only for [001][011][111] orientation triangle.

---

py4DSTEM.process.diffraction.crystal_viz.**plot_fiber_orientation_maps**(*self*, *orientation_map*, *orientation_ind: int = 0*, *symmetry_order: int | None = None*, *symmetry_mirror: bool = False*, *dir_in_plane_degrees: float = 0.0*, *corr_range: ndarray = array([0, 2])*, *corr_normalize: bool = True*, *show_axes: bool = True*, *medfilt_size: int | None = None*, *cmap_out_of_plane: str = 'plasma'*, *leg_size: int = 200*, *figsize: list | tuple | ndarray = (12, 8)*, *figbound: list | tuple | ndarray = (0.005, 0.04)*, *returnfig: bool = False*)

Generate and plot the orientation maps from fiber texture plots.

Parameters

- **orientation_map** (OrientationMap) – Class containing orientation matrices, correlation values, etc.
- **orientation_ind** (*int*) – Which orientation match to plot if num_matches > 1
- **dir_in_plane_degrees** (*float*) – Reference in-plane angle (degrees). Default is 0 / x-axis / vertical down.
- **corr_range** (*np.ndarray*) – Correlation intensity range for the plot
- **corr_normalize** (*bool*) – If true, set mean correlation to 1.
- **show_axes** (*bool*) – Flag setting whether orienation map axes are visible.
- **figsize** (*array*) – 2 elements defining figure size
- **figbound** (*array*) – 2 elements defining figure boundary
- **returnfig** (*bool*) – set to True to return figure and axes handles

Returns

RGB images fig, axs (handles): Figure and axes handes for the

Return type

images_orientation (int)

Note: Currently, no symmetry reduction. Therefore the x and y orientations are going to be correct only for [001][011][111] orientation triangle.

---

py4DSTEM.process.diffraction.crystal_viz.**plot_clusters**(*self*, *area_min=2*, *outline_grains=True*, *outline_thickness=1*, *fill_grains=0.25*, *smooth_grains=1.0*, *cmap='viridis'*, *figsize=(8, 8)*, *returnfig=False*)

Plot the clusters as an image.

Parameters

- **area_min** (*int (optional)*) – Min cluster size to include, in units of probe positions.
- **outline_grains** (*bool (optional)*) – Set to True to draw grains with outlines
- **outline_thickness** (*int (optional)*) – Thickenss of the grain outline
- **fill_grains** (*float (optional)*) – Outlined grains are filled with this value in pixels.
- **smooth_grains** (*float (optional)*) – Grain boundaries are smoothed by this value in pixels.
- **figsize** (*tuple*) – Size of the figure panel
- **returnfig** (*bool*) – Setting this to true returns the figure and axis handles

Returns
Figure and axes handles

Return type
fig, ax (optional)

py4DSTEM.process.diffraction.crystal_viz.**plot_cluster_size**(*self*, *area_min=None*, *area_max=None*, *area_step=1*, *weight_intensity=False*, *pixel_area=1.0*, *pixel_area_units='px^2'*, *figsize=(8, 6)*, *returnfig=False*)

Plot the cluster sizes

Parameters

- **area_min** (*int (optional)*) – Min area to include in pixels^2
- **area_max** (*int (optional)*) – Max area bin in pixels^2
- **area_step** (*int (optional)*) – Step size of the histogram bin in pixels^2
- **weight_intensity** (*bool*) – Weight histogram by the peak intensity.
- **pixel_area** (*float*) – Size of pixel area unit square
- **pixel_area_units** (*string*) – Units of the pixel area
- **figsize** (*tuple*) – Size of the figure panel
- **returnfig** (*bool*) – Setting this to true returns the figure and axis handles

Returns
Figure and axes handles

Return type
fig, ax (optional)

py4DSTEM.process.diffraction.crystal_viz.**atomic_colors**(*Z*, *scheme='jmol'*)

Return atomic colors for Z.

Modes are “colin” and “jmol”. “colin” uses the handmade but incomplete scheme of Colin Ophus “jmol” uses the JMOL scheme, from http://jmol.sourceforge.net/jscolors

which includes all elements up to 109

py4DSTEM.process.diffraction.crystal_viz.**plot_ring_pattern**(*radii*, *intensity*, *theta=[-3.141592653589793, 3.141592653589793, 200]*, *intensity_scale=1*, *intensity_constant=False*, *color='k'*, *figsize=(10, 10)*, *returnfig=False*, *input_fig_handle=None*, ***kwargs*)

2D plot of diffraction rings

Parameters

- **radii** (PointList) – 1D numpy array containing radii for diffraction rings
- **intensity** (PointList) – 1D numpy array containing intensities for diffraciton rings
- **theta** (*3-tuple*) – first two values specify angle range, and the last specifies the number of points used for plotting
- **intensity_scale** (*float*) – size scaling for ring thickness
- **intensity_constant** (*bool*) – if true, all rings are plotted with same line width
- **color** (*matplotlib color*) – color of ring, any format recognized by matplotlib
- **figsize** (*2 element float*) – size scaling of figure axes
- **returnfig** (*bool*) – set to True to return figure and axes handles
- **input_fig_handle** (*fig,ax*) –

py4DSTEM.process.diffraction.flowlines.**make_orientation_histogram**(*bragg_peaks: PointListArray | None = None*, *radial_ranges: ndarray | None = None*, *orientation_map=None*, *orientation_ind: int = 0*, *orientation_growth_angles: array = 0.0*, *orientation_separate_bins: bool = False*, *orientation_flip_sign: bool = False*, *upsample_factor=4.0*, *theta_step_deg=1.0*, *sigma_x=1.0*, *sigma_y=1.0*, *sigma_theta=3.0*, *normalize_intensity_image: bool = False*, *normalize_intensity_stack: bool = True*, *progress_bar: bool = True*)

Create an 3D or 4D orientation histogram from a braggpeaks PointListArray from user-specified radial ranges, or from the Euler angles from a fiber texture OrientationMap generated by the ACOM module of py4DSTEM.

Parameters

- **bragg_peaks** (PointListArray) – 2D of pointlists containing centered peak locations.
- **radial_ranges** (*np array*) – Size (N x 2) array for N radial bins, or (2,) for a single bin.
- **orientation_map** (OrientationMap) – Class containing the Euler angles to generate a flowline map.

- **orientation_ind** (*int*) – Index of the orientation map (default 0)
- **orientation_growth_angles** (*array*) – Angles to place into histogram, relative to orientation.
- **orientation_separate_bins** (*bool*) – whether to place multiple angles into multiple radial bins.
- **upsample_factor** (*float*) – Upsample factor
- **theta_step_deg** (*float*) – Step size along annular direction in degrees
- **sigma_x** (*float*) – Smoothing in x direction before upsample
- **sigma_y** (*float*) – Smoothing in x direction before upsample
- **sigma_theta** (*float*) – Smoothing in annular direction (units of bins, periodic)
- **normalize_intensity_image** (*bool*) – Normalize to max peak intensity = 1, per image
- **normalize_intensity_stack** (*bool*) – Normalize to max peak intensity = 1, all images
- **progress_bar** (*bool*) – Enable progress bar

Returns

4D array containing Bragg peak intensity histogram
[radial_bin x_probe y_probe theta]

Return type
orient_hist (array)

py4DSTEM.process.diffraction.flowlines.**make_flowline_map**(*orient_hist*, *thresh_seed=0.2*, *thresh_grow=0.05*, *thresh_collision=0.001*, *sep_seeds=None*, *sep_xy=6.0*, *sep_theta=5.0*, *sort_seeds='intensity'*, *linewidth=2.0*, *step_size=0.5*, *min_steps=4*, *max_steps=1000*, *sigma_x=1.0*, *sigma_y=1.0*, *sigma_theta=2.0*, *progress_bar: bool = True*)

Create an 3D or 4D orientation flowline map - essentially a pixelated "stream map" which represents diffraction data.

Parameters

- **orient_hist** (*array*) – Histogram of all orientations with coordinates [radial_bin x_probe y_probe theta] We assume theta bin ranges from 0 to 180 degrees and is periodic.
- **thresh_seed** (*float*) – Threshold for seed generation in histogram.
- **thresh_grow** (*float*) – Threshold for flowline growth in histogram.
- **thresh_collision** (*float*) – Threshold for termination of flowline growth in histogram.
- **sep_seeds** (*float*) – Initial seed separation in bins - set to None to use default value, which is equal to 0.5*sep_xy.
- **sep_xy** (*float*) – Search radius for flowline direction in x and y.
- **=** (*sep_theta*) – Search radius for flowline direction in theta.

- **sort_seeds** (*str*) – How to sort the initial seeds for growth: None - no sorting 'intensity' - sort by histogram intensity 'random' - random order
- **linewidth** (*float*) – Thickness of the flowlines in pixels.
- **step_size** (*float*) – Step size for flowline growth in pixels.
- **min_steps** (*int*) – Minimum number of steps for a flowline to be drawn.
- **max_steps** (*int*) – Maximum number of steps for a flowline to be drawn.
- **sigma_x** (*float*) – Weighted sigma in x direction for direction update.
- **sigma_y** (*float*) – Weighted sigma in y direction for direction update.
- **sigma_theta** (*float*) – Weighted sigma in theta for direction update.
- **progress_bar** (*bool*) – Enable progress bar

Returns

4D array containing flowlines
[radial_bin x_probe y_probe theta]

Return type
orient_flowlines (array)

py4DSTEM.process.diffraction.flowlines.**make_flowline_rainbow_image**(*orient_flowlines*, *int_range=[0, 0.2]*, *sym_rotation_order=2*, *theta_offset=0.0*, *greyscale=False*, *greyscale_max=True*, *white_background=False*, *power_scaling=1.0*, *sum_radial_bins=False*, *plot_images=True*, *figsize=None*)

Generate RGB output images from the flowline arrays.

Parameters

- **orient_flowline** (*array*) – Histogram of all orientations with coordinates [x y radial_bin theta] We assume theta bin ranges from 0 to 180 degrees and is periodic.
- **int_range** (*float*) –
- **sym_rotation_order** (*int*) – rotational symmety for colouring
- **theta_offset** (*float*) – Offset the anglular coloring by this value in radians.
- **greyscale** (*bool*) – Set to False for color output, True for greyscale output.
- **greyscale_max** (*bool*) – If output is greyscale, use max instead of mean for overlapping flowlines.
- **white_background** (*bool*) – For either color or greyscale output, switch to white background (from black).
- **power_scaling** (*float*) – Power law scaling for flowline intensity output.
- **sum_radial_bins** (*bool*) – Sum all radial bins (alternative is to output separate images).
- **plot_images** (*bool*) – Plot the outputs for quick visualization.
- **figsize** (*2-tuple*) – Size of output figure.

Returns
3D or 4D array containing flowline images

Return type
im_flowline (array)

py4DSTEM.process.diffraction.flowlines.**make_flowline_rainbow_legend**(*im_size=array([256, 256])*, *sym_rotation_order=2*, *theta_offset=0.0*, *white_background=False*, *return_image=False*, *radial_range=array([0.45, 0.9])*, *plot_legend=True*, *figsize=(4, 4)*)

This function generates a legend for a the rainbow colored flowline maps, and returns it as an RGB image.

Parameters

- **im_size** (*np.array*) – Size of legend image in pixels.
- **sym_rotation_order** (*int*) – rotational symmety for colouring
- **theta_offset** (*float*) – Offset the anglular coloring by this value in radians.
- **white_background** (*bool*) – For either color or greyscale output, switch to white background (from black).
- **return_image** (*bool*) – Return the image array.
- **radial_range** (*np.array*) – Inner and outer radius for the legend ring.
- **plot_legend** (*bool*) – Plot the generated legend.
- **figsize** (*tuple or list*) – Size of the plotted legend.

Returns
Image array for the legend.

Return type
im_legend (array)

py4DSTEM.process.diffraction.flowlines.**make_flowline_combined_image**(*orient_flowlines*, *int_range=[0, 0.2]*, *cvals=array([[0., 0.7, 0.], [1., 0., 0.], [0., 0.7, 1.]])*, *white_background=False*, *power_scaling=1.0*, *sum_radial_bins=True*, *plot_images=True*, *figsize=None*)

Generate RGB output images from the flowline arrays.

Parameters

- **orient_flowline** (*array*) – Histogram of all orientations with coordinates [x y radial_bin theta] We assume theta bin ranges from 0 to 180 degrees and is periodic.
- **int_range** (*float*) –
- **cvals** (*array*) – Nx3 size array containing RGB colors for different radial ibns.
- **white_background** (*bool*) – For either color or greyscale output, switch to white background (from black).

- **power_scaling** (*float*) – Power law scaling for flowline intensities.
- **sum_radial_bins** (*bool*) – Sum outputs over radial bins.
- **plot_images** (*bool*) – Plot the output images for quick visualization.
- **figsize** (*2-tuple*) – Size of output figure.

Returns
flowline images

Return type
im_flowline (array)

py4DSTEM.process.diffraction.flowlines.**orientation_correlation**(*orient_hist*, *radius_max=None*, *progress_bar=True*)

Take in the 4D orientation histogram, and compute the distance-angle (auto)correlations

Parameters

- **orient_hist** (*array*) – 3D or 4D histogram of all orientations with coordinates [x y radial_bin theta]
- **radius_max** (*float*) – Maximum radial distance for correlogram calculation. If set to None, the maximum radius will be set to min(orient_hist.shape[0],orient_hist.shape[1])/2.

Returns
3D or 4D array containing correlation images as function of (dr,dtheta)

Return type
orient_corr (array)

py4DSTEM.process.diffraction.flowlines.**plot_orientation_correlation**(*orient_corr*, *prob_range=[0.1, 10.0]*, *calculate_coefs=False*, *fraction_coefs=0.5*, *length_fit_slope=10*, *plot_overlaid_coefs=True*, *inds_plot=None*, *pixel_size=None*, *pixel_units=None*, *fontsize=10*, *figsize=(8, 6)*, *returnfig=False*)

Plot the distance-angle (auto)correlations in orient_corr.

Parameters

- **(array)** (*figsize*) – 3D or 4D array containing correlation images as function of (dr,dtheta) 1st index represents each pair of rings.
- **(array)** – Plotting range in units of "multiples of random distribution".
- **(bool)** (*returnfig*) – If this value is True, the 0.5 and 0.1 distribution fraction of the radial and annular correlations will be calculated and printed.
- **(float)** (*fontsize*) – What fraction to calculate the correlation distribution coefficients for.
- **(int)** (*length_fit_slope*) – Number of pixels to fit the slope of angular vs radial intercept.
- **(bool)** – If this value is True, the 0.5 and 0.1 distribution fraction of the radial and annular correlations will be overlaid onto the plots.

- **(float)** – Which indices to plot for orient_corr. Set to "None" to plot all pairs.
- **(float)** – Pixel size for x axis.
- **(str)** (*pixel_units*) – units of pixels.
- **(float)** – Font size. Title will be slightly larger, axis slightly smaller.
- **(array)** – Size of the figure panels.
- **(bool)** – Set to True to return figure axes.

Returns
Figure and axes handles (optional).

Return type
fig, ax (handles)

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv – command line arguments; argv[0] is the script pathname if known path – module search path; path[0] is the script directory, else '' modules – dictionary of loaded modules

displayhook – called to show results in an interactive session excepthook – called to handle any uncaught exception other than SystemExit

> To customize printing in an interactive session or to install a custom top-level exception handler, assign other functions to replace these.

stdin – standard input file object; used by input() stdout – standard output file object; used by print() stderr – standard error object; used for error messages

> By assigning other file objects (or objects that behave like files) to these, it is possible to redirect all of the interpreter's I/O.

last_type – type of last uncaught exception last_value – value of last uncaught exception last_traceback – traceback of last uncaught exception

> These three are only available in an interactive session after a traceback has been printed.

Static objects:

builtin_module_names – tuple of module names built into this interpreter copyright – copyright notice pertaining to this interpreter exec_prefix – prefix used to find the machine-specific Python library executable – absolute path of the executable binary of the Python interpreter float_info – a named tuple with information about the float implementation. float_repr_style – string indicating the style of repr() output for floats hash_info – a named tuple with information about the hash algorithm. hexversion – version information encoded as a single integer implementation – Python implementation information. int_info – a named tuple with information about the int implementation. maxsize – the largest supported length of containers. maxunicode – the value of the largest Unicode code point platform – platform identifier prefix – prefix used to find the Python library thread_info – a named tuple with information about the thread implementation. version – the version of this interpreter as a string version_info – version information as a named tuple __stdin__ – the original stdin; don't touch! __stdout__ – the original stdout; don't touch! __stderr__ – the original stderr; don't touch! __displayhook__ – the original displayhook; don't touch! __excepthook__ – the original excepthook; don't touch!

Functions:

displayhook() – print an object to the screen, and save it in builtins._ excepthook() – print an exception and its traceback to sys.stderr exc_info() – return thread-safe information about the current exception exit() – exit the interpreter by raising SystemExit getdlopenflags() – returns flags to be used for dlopen() calls getprofile() – get the global profiling function getrefcount() – return the reference count for an object (plus one :-) getrecursionlimit() – return the max recursion

depth for the interpreter getsizeof() – return the size of an object in bytes gettrace() – get the global debug tracing function setdlopenflags() – set the flags to be used for dlopen() calls setprofile() – set the global profiling function setrecursionlimit() – set the max recursion depth for the interpreter settrace() – set the global debug tracing function

class py4DSTEM.process.diffraction.utils.**Orientation**(*num_matches: int*)

A class for storing output orientations, generated by fitting a Crystal class orientation plan or Bloch wave pattern matching to a PointList.

__init__(*num_matches: int*) → None

class py4DSTEM.process.diffraction.utils.**OrientationMap**(*num_x: int*, *num_y: int*, *num_matches: int*)

A class for storing output orientations, generated by fitting a Crystal class orientation plan or Bloch wave pattern matching to a PointListArray.

__init__(*num_x: int*, *num_y: int*, *num_matches: int*) → None

py4DSTEM.process.diffraction.utils.**sort_orientation_maps**(*orientation_map*, *sort='intensity'*, *cluster_thresh=0.1*)

Sort the orientation maps along the ind_match direction, either by intensity or by clustering similar angles (greedily, in order of intensity).

Parameters

- **OrientationMap** (*orientation_map Initial*) –
- **sort** (*string*) – "intensity" or "cluster" for sorting method.
- **cluster_thresh** (*float*) – similarity threshold for clustering method

Returns

orientation_sort Sorted OrientationMap

py4DSTEM.process.diffraction.utils.**calc_1D_profile**(*k*, *g_coords*, *g_int*, *remove_origin=True*, *k_broadening=0.0*, *int_scale=None*, *normalize_intensity=True*)

Utility function to calculate a 1D histogram from the diffraction vector lengths stored in a Crystal class.

Parameters

- **k** (*np.array*) – k coordinates.
- **g_coords** (*np.array*) – Scattering vector lengths g.
- **bragg_intensity_power** (*np.array*) – Scattering vector intensities.
- **remove_origin** (*bool*) – Remove the origin peak from the profile.
- **k_broadening** (*float*) – Broadening applied to full profile.
- **int_scale** (*np.array*) – Either a scalar value mulitiplied into all peak intensities, or a vector with 1 value per peak to scale peaks individually.
- **normalize_intensity** – Normalize maximum output value to 1.

diskdetection

fit

py4DSTEM.process.fit.fit.**fit_1D_gaussian**(*xdata*, *ydata*, *xmin*, *xmax*)

Fits a 1D gaussian to the subset of the 1D curve f(xdata)=ydata within the window (xmin,xmax). Returns A,mu,sigma. Retrieve the full curve with

```
>>> fit_gaussian = py4DSTEM.process.fit.gaussian(xdata,A,mu,sigma)
```

py4DSTEM.process.fit.fit.**fit_2D**(*function*, *data*, *data_mask=None*, *popt=None*, *robust=False*, *robust_steps=3*, *robust_thresh=2*)

Performs a 2D fit.

TODO: make returning the mask optional

Parameters

- **function** (*callable*) – Some *function(xy, **p)* where *xy* is a length 2 vector (1D np array) specifying the pixel position (x,y), and *p* is the function parameters
- **data** (*ndarray*) – Some 2D array of any shape (n,m)
- **data_mask** (*None or boolean array of shape (n,m), optional*) – If specified, fits only the pixels in *data* where this array is True
- **popt** (*dict*) – Initial guess at the parameters *p* of *function*. Note that positions in pixels (i.e. the xy positions) are linearly scaled to the space [0,1]
- **robust** (*bool*) – Toggles robust fitting, which iteratively rejects outlier data points which have a root-mean-square error beyond *robust_thresh*
- **robust_steps** (*int*) – The number of robust fitting iterations to perform
- **robust_thresh** (*int*) – The robust fitting cutoff
- **Returns** –
- **(popt** (*4-tuple*) – The optimal fit parameters, the fitting covariance matrix, the the fit array with the returned *popt* params, and the mask
- **pcov** (*4-tuple*) – The optimal fit parameters, the fitting covariance matrix, the the fit array with the returned *popt* params, and the mask
- **fit_at** (*4-tuple*) – The optimal fit parameters, the fitting covariance matrix, the the fit array with the returned *popt* params, and the mask
- **mask)** (*4-tuple*) – The optimal fit parameters, the fitting covariance matrix, the the fit array with the returned *popt* params, and the mask

py4DSTEM.process.fit.fit.**fit_2D_polar_gaussian**(*data*, *mask=None*, *p0=None*, *robust=False*, *robust_steps=3*, *robust_thresh=2*, *constant_background=False*)

NOTE - this cannot work without using pixel coordinates - something is wrong in the workflow.

Fits a 2D gaussian to the pixels in *data* which are set to True in *mask*.

The gaussian is anisotropic and oriented along (t,q), centered at (mu_t,mu_q), has standard deviations (sigma_t,sigma_q), maximum of I0, and an optional constant offset of C, and is periodic in t.

f(x,y) = I0 * exp(- (x-mu_x)^2/(2sig_x^2) + (y-mu_y)^2/(2sig_y^2)) or f(x,y) = I0 * exp(- (x-mu_x)^2/(2sig_x^2) + (y-mu_y)^2/(2sig_y^2)) + C

Parameters

- **data** (*2d array*) – the data to fit
- **p0** (*6-tuple*) – initial guess at fit parameters, (I0,mu_x,mu_y,sigma_x_sigma_y,C)
- **mask** (*2d boolean array*) – ignore pixels where mask is False
- **robust** (*bool*) – toggle robust fitting
- **robust_steps** (*int*) – number of robust fit iterations
- **robust_thresh** (*number*) – the robust fitting threshold
- **constant_background** (*bool*) – whether or not to include constant background

Returns

(popt,pcov,fit_ar) – the optimal fit parameters, the covariance matrix, and the fit array

Return type

3-tuple

latticevectors

phase

py4DSTEM.process.phase.utils.**polar_symbols = ('C10', 'C12', 'phi12', 'C21', 'phi21', 'C23', 'phi23', 'C30', 'C32', 'phi32', 'C34', 'phi34', 'C41', 'phi41', 'C43', 'phi43', 'C45', 'phi45', 'C50', 'C52', 'phi52', 'C54', 'phi54', 'C56', 'phi56')**

Symbols for the polar representation of all optical aberrations up to the fifth order.

py4DSTEM.process.phase.utils.**polar_aliases = {'C5': 'C50', 'Cs': 'C30', 'astigmatism': 'C12', 'astigmatism_angle': 'phi12', 'coma': 'C21', 'coma_angle': 'phi21', 'defocus': 'C10'}**

Aliases for the most commonly used optical aberrations.

class py4DSTEM.process.phase.utils.**ComplexProbe**(*energy: float*, *gpts: Tuple[int, int]*, *sampling: Tuple[float, float]*, *semiangle_cutoff: float = inf*, *rolloff: float = 2.0*, *vacuum_probe_intensity: ndarray | None = None*, *device: str = 'cpu'*, *focal_spread: float = 0.0*, *angular_spread: float = 0.0*, *gaussian_spread: float = 0.0*, *phase_shift: float = 0.0*, *parameters: Mapping[str, float] | None = None*, ***kwargs*)

Complex Probe Class.

Simplified version of CTF and Probe from abTEM: https://github.com/abTEM/abTEM/blob/master/abtem/transfer.py https://github.com/abTEM/abTEM/blob/master/abtem/waves.py

Parameters

- **energy** (*float*) – The electron energy of the wave functions this contrast transfer function will be applied to [eV].
- **semiangle_cutoff** (*float*) – The semiangle cutoff describes the sharp Fourier space cutoff due to the objective aperture [mrad].
- **gpts** (*Tuple[int,int]*) – Number of grid points describing the wave functions.
- **sampling** (*Tuple[float,float]*) – Lateral sampling of wave functions in Å

- **device** (*str, optional*) – Device to perform calculations on. Must be either 'cpu' or 'gpu'
- **rolloff** (*float, optional*) – Tapers the cutoff edge over the given angular range [mrad].
- **vacuum_probe_intensity** (*np.ndarray, optional*) – Squared of corner-centered aperture amplitude to use, instead of semiangle_cutoff + rolloff
- **focal_spread** (*float, optional*) – The 1/e width of the focal spread due to chromatic aberration and lens current instability [Å].
- **angular_spread** (*float, optional*) – The 1/e width of the angular deviations due to source size [mrad].
- **gaussian_spread** (*float, optional*) – The 1/e width image deflections due to vibrations and thermal magnetic noise [Å].
- **phase_shift** (*float, optional*) – A constant phase shift [radians].
- **parameters** (*dict, optional*) – Mapping from aberration symbols to their corresponding values. All aberration magnitudes should be given in Å and angles should be given in radians.
- **kwargs** – Provide the aberration coefficients as keyword arguments.

__init__(*energy: float*, *gpts: Tuple[int, int]*, *sampling: Tuple[float, float]*, *semiangle_cutoff: float = inf*, *rolloff: float = 2.0*, *vacuum_probe_intensity: ndarray | None = None*, *device: str = 'cpu'*, *focal_spread: float = 0.0*, *angular_spread: float = 0.0*, *gaussian_spread: float = 0.0*, *phase_shift: float = 0.0*, *parameters: Mapping[str, float] | None = None*, ***kwargs*)

set_parameters(*parameters: dict*)

Set the phase of the phase aberration. :param parameters: Mapping from aberration symbols to their corresponding values. :type parameters: dict

polar_coordinates(*x*, *y*)

Calculate a polar grid for a given Cartesian grid.

build()

Builds corner-centered complex probe in the center of the region of interest.

visualize(***kwargs*)

Plots the probe intensity.

py4DSTEM.process.phase.utils.**spatial_frequencies**(*gpts: ~typing.Tuple[int, int], sampling: ~typing.Tuple[float, float], xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/late packages/numpy/__init__.py'>*)

Calculate spatial frequencies of a grid.

Parameters

- **gpts** (*tuple of int*) – Number of grid points.
- **sampling** (*tuple of float*) – Sampling of the potential [1 / Å].

Return type
tuple of arrays

py4DSTEM.process.phase.utils.**fourier_translation_operator**(*positions: ~numpy.ndarray*, *shape: tuple*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dste packages/numpy/__init__.py'>*) → ndarray

Create an array representing one or more phase ramp(s) for shifting another array.

Parameters

- **positions** (*array of xy-positions*) – Positions to calculate fourier translation operators for
- **shape** (*two int*) – Array dimensions to be fourier-shifted
- **xp** (*Callable*) – Array computing module

Return type
Fourier translation operators

py4DSTEM.process.phase.utils.**fft_shift**(*array*, *positions*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/latest/lib/python3 packages/numpy/__init__.py'>*)

Fourier-shift array using positions.

Parameters

- **array** (*np.ndarray*) – Array to be shifted
- **positions** (*array of xy-positions*) – Positions to fourier-shift array with
- **xp** (*Callable*) – Array computing module

Return type
Fourier-shifted array

py4DSTEM.process.phase.utils.**subdivide_into_batches**(*num_items: int*, *num_batches: int | None = None*, *max_batch: int | None = None*)

Split an n integer into m (almost) equal integers, such that the sum of smaller integers equals n.

Parameters

- **n** (*int*) – The integer to split.
- **m** (*int*) – The number integers n will be split into.

Return type
list of int

class py4DSTEM.process.phase.utils.**AffineTransform**(*scale0: float = 1.0*, *scale1: float = 1.0*, *shear1: float = 0.0*, *angle: float = 0.0*, *t0: float = 0.0*, *t1: float = 0.0*, *dilation: float = 1.0*)

Affine Transform Class.

Simplified version of AffineTransform from tike: https://github.com/AdvancedPhotonSource/tike/blob/f9004a32fda5e49fa63b987e9ffe3c8447d59950/src/tike/ptycho/position.py

AffineTransform() -> Identity

Parameters

- **scale0** (*float*) – x-scaling
- **scale1** (*float*) – y-scaling

- **shear1** (*float*) – gamma shear
- **angle** (*float*) – theta rotation angle
- **t0** (*float*) – x-translation
- **t1** (*float*) – y-translation
- **dilation** (*float*) – Isotropic expansion (multiplies scale0 and scale1)

__init__(*scale0: float = 1.0*, *scale1: float = 1.0*, *shear1: float = 0.0*, *angle: float = 0.0*, *t0: float = 0.0*, *t1: float = 0.0*, *dilation: float = 1.0*)

classmethod fromarray(*T: ndarray*)

Return an Affine Transfrom from a 2x2 matrix. Use decomposition method from Graphics Gems 2 Section 7.1

asarray()

Return an 2x2 matrix of scale, shear, rotation. This matrix is scale @ shear @ rotate from left to right.

asarray3()

Return an 3x2 matrix of scale, shear, rotation, translation. This matrix is scale @ shear @ rotate from left to right. Expects a homogenous (z) coordinate of 1.

astuple()

Return the constructor parameters in a tuple.

py4DSTEM.process.phase.utils.**estimate_global_transformation**(*positions0: ~numpy.ndarray*, *positions1: ~numpy.ndarray*, *origin: ~typing.Tuple[int, int] = (0, 0)*, *translation_allowed: bool = True*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4d packages/numpy/__init__.py'>*)

Use least squares to estimate the global affine transformation.

py4DSTEM.process.phase.utils.**estimate_global_transformation_ransac**(*positions0: ~numpy.ndarray*, *positions1: ~numpy.ndarray*, *origin: ~typing.Tuple[int, int] = (0, 0)*, *translation_allowed: bool = True*, *min_sample: int = 64*, *max_error: float = 16*, *min_consensus: float = 0.75*, *max_iter: int = 20*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_bu packages/numpy/__init__.py'>*)

Use RANSAC to estimate the global affine transformation.

py4DSTEM.process.phase.utils.**fourier_ring_correlation**(*image_1*, *image_2*, *pixel_size=None*, *bin_size=None*, *sigma=None*, *align_images=False*, *upsample_factor=8*, *device='cpu'*, *plot_frc=True*, *frc_color='red'*, *half_bit_color='blue'*)

Computes fourier ring correlation (FRC) of 2 arrays. Arrays must bet the same size.

Parameters

image1: ndarray
first image for FRC

image2: ndarray
second image for FRC

pixel_size: tuple
size of pixels in A (x,y)

bin_size: float, optional
size of bins for ring profile

sigma: float, optional
standard deviation for Gaussian kernel

align_images: bool
if True, aligns images using DFT upsampling of cross correlation.

upsample factor: int
if align_images, upsampling for correlation. Must be greater than 2.

device: str, optional
calculation device will be perfomed on. Must be 'cpu' or 'gpu'

plot_frc: bool, optional
if True, plots frc

frc_color: str, optional
color of FRC line in plot

half_bit_color: str, optional
color of half-bit line

Returns

- **q_frc** (*ndarray*) – spatial frequencies of FRC
- **frc** (*ndarray*) – fourier ring correlation
- **half_bit** (*ndarray*) – half-bit criteria

py4DSTEM.process.phase.utils.**return_1D_profile**(*intensity*, *pixel_size=None*, *bin_size=None*, *sigma=None*, *device='cpu'*)

Return 1D radial profile from corner centered array

Parameters

intensity: ndarray
Array for computing 1D profile

pixel_size: tuple
Size of pixels in A (x,y)

bin_size: float, optional
Size of bins for ring profile

sigma: float, optional
standard deviation for Gaussian kernel

device: str, optional
calculation device will be perfomed on. Must be 'cpu' or 'gpu'

Returns

- **q_bins** (*ndarray*) – spatial frequencies of bins
- **I_bins** (*ndarray*) – Intensity of bins
- **n** (*ndarray*) – Number of pixels in each bin

py4DSTEM.process.phase.utils.**fourier_rotate_real_volume**(*array*, *angle*, *axes=(0, 1)*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/packages/numpy/__init__.py'>*)

Rotates a 3D array using three Fourier-based shear operators.

Parameters

array: ndarray
3D array to rotate

angle: float
Angle in deg to rotate array by

axes: tuple, Optional
Axes defining plane in which to rotate about

xp: Callable, optional
Array computing module

Returns
output_arr – Fourier-rotated array

Return type
ndarray

py4DSTEM.process.phase.utils.**array_slice**(*axis*, *ndim*, *start*, *end*, *step=1*)

Returns array slice along dynamic axis

py4DSTEM.process.phase.utils.**periodic_centered_difference**(*array*, *spacing*, *axis*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dste packages/numpy/__init__.py'>*)

Computes second-order centered difference with periodic BCs

py4DSTEM.process.phase.utils.**compute_divergence_periodic**(*vector_field*, *spacings*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dsten packages/numpy/__init__.py'>*)

Computes divergence of vector_field

py4DSTEM.process.phase.utils.**compute_gradient_periodic**(*scalar_field*, *spacings*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/e packages/numpy/__init__.py'>*)

Computes gradient of scalar_field

py4DSTEM.process.phase.utils.**preconditioned_laplacian_periodic_3D**(*shape*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_bui packages/numpy/__init__.py'>*)

FFT eigenvalues

py4DSTEM.process.phase.utils.**preconditioned_poisson_solver_periodic_3D**(*rhs*, *gauge=None*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/us packages/numpy/__init__.py'>*)

FFT based poisson solver

py4DSTEM.process.phase.utils.**project_vector_field_divergence_periodic_3D**(*vector_field*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/ packages/numpy/__init__.py'>*)

Returns solenoidal part of vector field using projection:

f - grad{p} s.t. laplacian{p} = div{f}

py4DSTEM.process.phase.utils.**cartesian_to_polar_transform_2Ddata**(*im_cart*, *xy_center*, *num_theta_bins=90*, *radius_max=None*, *corner_centered=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_build packages/numpy/__init__.py'>*)

Quick cartesian to polar conversion.

py4DSTEM.process.phase.utils.**polar_to_cartesian_transform_2Ddata**(*im_polar*, *xy_size*, *xy_center*, *corner_centered=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_build packages/numpy/__init__.py'>*)

Quick polar to cartesian conversion.

py4DSTEM.process.phase.utils.**regularize_probe_amplitude**(*probe_init*, *width_max_pixels=2.0*, *nearest_angular_neighbor_averaging=5*, *enforce_constant_intensity=True*, *corner_centered=False*)

Fits sigmoid for each angular direction.

Parameters

- **probe_init** (*np.array*) – 2D complex image of the probe in Fourier space.
- **width_max_pixels** (*float*) – Maximum edge width of the probe in pixels.
- **nearest_angular_neighbor_averaging** (*int*) – Number of nearest angular neighbor pixels to average to make aperture less jagged.
- **enforce_constant_intensity** (*bool*) – Set to true to make intensity inside the aperture constant.
- **corner_centered** (*bool*) – If True, the probe is assumed to be corner-centered

Returns

- **probe_corr** (*np.ndarray*) – 2D complex image of the corrected probe in Fourier space.
- **coefs_all** (*np.ndarray*) – coefficients for the sigmoid fits

py4DSTEM.process.phase.utils.**interleave_ndarray_symmetrically**(*array_nd*, *axis*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py packages/numpy/__init__.py'>*)

[a,b,c,d,e,f] -> [a,c,e,f,d,b]

py4DSTEM.process.phase.utils.**dct_II_using_FFT_base**(*array_nd*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/la packages/numpy/__init__.py'>*)

FFT-based DCT-II

py4DSTEM.process.phase.utils.**interleave_ndarray_symmetrically_inverse**(*array_nd*, *axis*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user packages/numpy/__init__.py'>*)

[a,c,e,f,d,b] -> [a,b,c,d,e,f]

py4DSTEM.process.phase.utils.**idct_II_using_FFT_base**(*array_nd*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/ packages/numpy/__init__.py'>*)

FFT-based IDCT-II

py4DSTEM.process.phase.utils.**idct_II_using_FFT**(*array_nd*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/latest/l packages/numpy/__init__.py'>*)

FFT-based IDCT-II

py4DSTEM.process.phase.utils.**preconditioned_laplacian_neumann_2D**(*shape*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds packages/numpy/__init__.py'>*)

DCT eigenvalues

py4DSTEM.process.phase.utils.**preconditioned_poisson_solver_neumann_2D**(*rhs*, *gauge=None*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user packages/numpy/__init__.py'>*)

DCT based poisson solver

py4DSTEM.process.phase.utils.**unwrap_phase_2d**(*array*, *weights=None*, *gauge=None*, *corner_centered=True*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/latest/lib packages/numpy/__init__.py'>*)

Weigted phase unwrapping using DCT-based poisson solver

py4DSTEM.process.phase.utils.**rotate_point**(*origin*, *point*, *angle*)

Rotate a point (x1, y1) counterclockwise by a given angle around a given origin (x0, y0).

Parameters

- **origin** (*2-tuple of floats*) – (x0, y0)
- **point** (*2-tuple of floats*) – (x1, y1)
- **angle** (*float (radians)*) –

Return type
rotated points (2-tuple)

py4DSTEM.process.phase.utils.**bilinearly_interpolate_array**(*image*, *xa*, *ya*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dste packages/numpy/__init__.py'>*)

Bilinear sampling of intensities from an image array and pixel positions.

Parameters

- **image** (*np.ndarray*) – Image array to sample from
- **xa** (*np.ndarray*) – Vertical interpolation sampling positions of image array in pixels
- **ya** (*np.ndarray*) – Horizontal interpolation sampling positions of image array in pixels

Returns
intensities – Bilinearly-sampled intensities of array at (xa,ya) positions

Return type
np.ndarray

py4DSTEM.process.phase.utils.**lanczos_interpolate_array**(*image*, *xa*, *ya*, *alpha*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/e packages/numpy/__init__.py'>*)

Lanczos sampling of intensities from an image array and pixel positions.

Parameters

- **image** (*np.ndarray*) – Image array to sample from
- **xa** (*np.ndarray*) – Vertical Interpolation sampling positions of image array in pixels
- **ya** (*np.ndarray*) – Horizontal interpolation sampling positions of image array in pixels
- **alpha** (*int*) – Lanczos kernel order

Returns
intensities – Lanczos-sampled intensities of array at (xa,ya) positions

Return type
np.ndarray

py4DSTEM.process.phase.utils.**pixel_rolling_kernel_density_estimate**(*stack*, *shifts*, *upsampling_factor*, *kde_sigma*, *lowpass_filter=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_bu packages/numpy/__init__.py'>*, *gaussian_filter=<function gaussian_filter>*)

kernel density estimate from a set coordinates (xa,ya) and intensity weights.

Parameters

- **stack** (*np.ndarray*) – Unshifted image stack, shape (N,P,S)
- **shifts** (*np.ndarray*) – Shifts for each image in stack, shape: (N,2)
- **upsampling_factor** (*int*) – Upsampling factor

- **kde_sigma** (*float*) – KDE gaussian kernel bandwidth in upsampled pixels
- **lowpass_filter** (*bool, optional*) – If True, the resulting KDE upsampled image is lowpass-filtered using a sinc-function

Returns
pix_output – Upsampled intensity image

Return type
np.ndarray

py4DSTEM.process.phase.utils.**bilinear_kernel_density_estimate**(*xa*, *ya*, *intensities*, *output_shape*, *kde_sigma*, *lowpass_filter=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py packages/numpy/__init__.py'>*, *gaussian_filter=<function gaussian_filter>*)

kernel density estimate from a set coordinates (xa,ya) and intensity weights.

Parameters

- **xa** (*np.ndarray*) – Vertical positions of intensity array in pixels
- **ya** (*np.ndarray*) – Horizontal positions of intensity array in pixels
- **intensities** (*np.ndarray*) – Intensity array weights
- **output_shape** (*(int,int)*) – Upsampled intensities shape
- **kde_sigma** (*float*) – KDE gaussian kernel bandwidth in upsampled pixels
- **lowpass_filter** (*bool, optional*) – If True, the resulting KDE upsampled image is lowpass-filtered using a sinc-function

Returns
pix_output – Upsampled intensity image

Return type
np.ndarray

py4DSTEM.process.phase.utils.**lanczos_kernel_density_estimate**(*xa*, *ya*, *intensities*, *output_shape*, *kde_sigma*, *alpha*, *lowpass_filter=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4 packages/numpy/__init__.py'>*, *gaussian_filter=<function gaussian_filter>*)

kernel density estimate from a set coordinates (xa,ya) and intensity weights.

Parameters

- **xa** (*np.ndarray*) – Vertical positions of intensity array in pixels
- **ya** (*np.ndarray*) – Horizontal positions of intensity array in pixels
- **intensities** (*np.ndarray*) – Intensity array weights
- **output_shape** (*(int,int)*) – Upsampled intensities shape
- **kde_sigma** (*float*) – KDE gaussian kernel bandwidth in upsampled pixels
- **alpha** (*int*) – Lanczos kernel order

- **lowpass_filter** (*bool, optional*) – If True, the resulting KDE upsampled image is lowpass-filtered using a sinc-function

Returns
pix_output – Upsampled intensity image

Return type
np.ndarray

py4DSTEM.process.phase.utils.**bilinear_resample**(*array*, *scale=None*, *output_size=None*, *mode='grid-wrap'*, *grid_mode=True*, *vectorized=True*, *conserve_array_sums=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/latest/lib/python3.8/site-packages/numpy/__init__.py'>*)

Resize an array along its final two axes. Note, this is vectorized by default and thus very memory-intensive.

The scaling of the array can be specified by passing either *scale*, which sets the scaling factor along both axes to be scaled; or by passing *output_size*, which specifies the final dimensions of the scaled axes.

Parameters

- **array** (*np.ndarray*) – Input array to be resampled
- **scale** (*float*) – Scalar value giving the scaling factor for all dimensions
- **output_size** (*(int,int)*) – Tuple of two values giving the output size for the final two axes
- **xp** (*Callable*) – Array computing module

Returns
resampled_array – Resampled array

Return type
np.ndarray

py4DSTEM.process.phase.utils.**vectorized_fourier_resample**(*array*, *scale=None*, *output_size=None*, *conserve_array_sums=False*, *xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/latest/lib/python3.8/site-packages/numpy/__init__.py'>*)

Resize a 2D array along any dimension, using Fourier interpolation. For 4D input arrays, only the final two axes can be resized. Note, this is vectorized and thus very memory-intensive.

The scaling of the array can be specified by passing either *scale*, which sets the scaling factor along both axes to be scaled; or by passing *output_size*, which specifies the final dimensions of the scaled axes (and allows for different scaling along the x,y or kx,ky axes.)

Parameters

- **array** (*np.ndarray*) – Input 2D/4D array to be resampled
- **scale** (*float*) – Scalar value giving the scaling factor for all dimensions
- **output_size** (*(int,int)*) – Tuple of two values giving eith the (x,y) or (kx,ky) output size for 2D and 4D respectively.
- **xp** (*Callable*) – Array computing module

Returns
resampled_array – Resampled 2D/4D array

Return type
np.ndarray

py4DSTEM.process.phase.utils.**partition_list**(*lst*, *size*)

Partitions lst into chunks of size. Returns a generator.

py4DSTEM.process.phase.utils.**copy_to_device**(*array*, *device='cpu'*)

Copies array to device. Default allows one to use this as asnumpy()

probe

rdf

py4DSTEM.process.rdf.amorph.**fit_stack**(*datacube*, *init_coefs*, *mask=None*)

This will fit an ellipse using the polar elliptical transform code to all the diffraction patterns. It will take in a datacube and return a coefficient array which can then be used to map strain, fit the centers, etc.

Parameters

- **datacute** – a datacube of diffraction data
- **init_coefs** – an initial starting guess for the fit
- **mask** – a mask, either 2D or 4D, for either one mask for the whole stack, or one per pattern.

Returns
an array of coefficients of the fit

py4DSTEM.process.rdf.amorph.**calculate_coef_strain**(*coef_cube*, *r_ref*)

This function will calculate the strains from a 3D matrix output by fit_stack

Coefs order:

- I0 the intensity of the first gaussian function
- I1 the intensity of the Janus gaussian
- sigma0 std of first gaussian
- sigma1 inner std of Janus gaussian
- sigma2 outer std of Janus gaussian
- c_bkgd a constant offset
- R center of the Janus gaussian
- x0,y0 the origin
- B,C 1x^2 + Bxy + Cy^2 = 1

Parameters

- **coef_cube** – output from fit_stack
- **r_ref** – a reference 0 strain radius - needed because we fit r as well as B and C

Returns

- **exx**: strain in the x axis direction in image coordinates
- **eyy**: strain in the y axis direction in image coordinates
- **exy**: shear

Return type
(3-tuple) A 3-tuple containing

py4DSTEM.process.rdf.amorph.**plot_strains**(*strains*, *cmap='RdBu_r'*, *vmin=None*, *vmax=None*, *mask=None*)

This function will plot strains with a unified color scale.

Parameters

- **strains** (*3-tuple of arrays*) – (exx, eyy, exy)
- **cmap** – imshow parameters
- **vmin** – imshow parameters
- **vmax** – imshow parameters
- **mask** – real space mask of values not to show (black)

py4DSTEM.process.rdf.amorph.**convert_stack_polar**(*datacube*, *coef_cube*)

This function will take the coef_cube from fit_stack and apply it to the image stack, to return polar transformed images.

Parameters

- **datacube** – data in datacube format
- **coef_cube** – coefs from fit_stack

Returns
polar transformed datacube

py4DSTEM.process.rdf.amorph.**compute_polar_stack_symmetries**(*datacube_polar*)

This function will take in a datacube of polar-transformed diffraction patterns, and do the autocorrelation, before taking the fourier transform along the theta direction, such that symmetries can be measured. They will be plotted by a different function

Parameters
datacube_polar – diffraction pattern cube that has been polar transformed

Returns
the normalized fft along the theta direction of the autocorrelated patterns in datacube_polar

py4DSTEM.process.rdf.amorph.**plot_symmetries**(*datacube_symmetries*, *sym_order*)

This function will take in a datacube from compute_polar_stack_symmetries and plot a specific symmetry order.

Parameters

- **datacube_symmetries** – result of compute_polar_stack_symmetries, the stack of fft'd autocorrelated diffraction patterns
- **sym_order** – symmetry order desired to plot

Returns
None

py4DSTEM.process.rdf.rdf.**get_radial_intensity**(*polar_img*, *polar_mask*)

Takes in a radial transformed image and the radial mask (if any) applied to that image. Designed to be compatible with polar-elliptical transforms from utils

py4DSTEM.process.rdf.rdf.**fit_scattering_factor**(*scale*, *elements*, *composition*, *q_arr*, *units*)

Scale is linear factor Elements is an 1D array of atomic numbers. Composition is a 1D array, same length as elements, describing the average atomic composition of the sample. If the Q_coords is a 1D array of Fourier coordinates, given in inverse Angstroms. Units is a string of 'VA' or 'A', which returns the scattering factor in volt angtroms or in angstroms.

py4DSTEM.process.rdf.rdf.**get_phi**(*radialIntensity*, *scatter*, *q_arr*)

ymean scale*scatter.fe**2

py4DSTEM.process.rdf.rdf.**get_mask**(*left*, *right*, *midpoint*, *slopes*, *q_arr*)

start is float stop is float midpoint is float slopes is [float,float]

py4DSTEM.process.rdf.rdf.**get_rdf**(*phi*, *q_arr*)

phi can be masked or not masked

utils

py4DSTEM.process.utils.cross_correlate.**get_cross_correlation**(*ar*, *template*, *corrPower=1*, *_returnval='real'*)

Get the cross/phase/hybrid correlation of *ar* with *template*, where the latter is in real space.

If _returnval is 'real', returns the real-valued cross-correlation. Otherwise, returns the complex valued result.

py4DSTEM.process.utils.cross_correlate.**get_cross_correlation_FT**(*ar*, *template_FT*, *corrPower=1*, *_returnval='real'*)

Get the cross/phase/hybrid correlation of *ar* with *template_FT*, where the latter is already in Fourier space (i.e. *template_FT* is *np.conj(np.fft.fft2(template))*.

If _returnval is 'real', returns the real-valued cross-correlation. Otherwise, returns the complex valued result.

py4DSTEM.process.utils.cross_correlate.**get_shift**(*ar1*, *ar2*, *corrPower=1*)

Determine the relative shift between a pair of arrays giving the best overlap.

Shift determination uses the brightest pixel in the cross correlation, and is

thus limited to pixel resolution. corrPower specifies the cross correlation power, with 1 corresponding to a cross correlation and 0 a phase correlation.

Args:
ar1,ar2 (2D ndarrays):

corrPower (float between 0 and 1, inclusive): 1=cross correlation, 0=phase correlation

Returns
(shiftx,shifty) - the relative image shift, in pixels

Return type
(2-tuple)

py4DSTEM.process.utils.cross_correlate.**align_images_fourier**(*G1*, *G2*, *upsample_factor*, *device='cpu'*)

Alignment of two images using DFT upsampling of cross correlation.

Parameters

- **G1** (*ndarray*) – fourier transform of image 1

- **G2** (*ndarray*) – fourier transform of image 2
- **upsample_factor** (*float*) – upsampling for correlation. Must be greater than 2.
- **device** (*str, optional*) – calculation device will be perfomed on. Must be 'cpu' or 'gpu'
- **Returns** – xy_shift [pixels]

py4DSTEM.process.utils.cross_correlate.**align_and_shift_images**(*image_1*, *image_2*, *upsample_factor*, *device='cpu'*)

Alignment of two images using DFT upsampling of cross correlation.

Parameters

- **image_1** (*ndarray*) – image 1
- **image_2** (*ndarray*) – image 2
- **upsample_factor** (*float*) – upsampling for correlation. Must be greater than 2.
- **device** (*str, optional*) – calculation device will be perfomed on. Must be 'cpu' or 'gpu'.
- **Returns** – shifted image [pixels]

Contains functions relating to polar-elliptical calculations.

This includes

- transforming data from cartesian to polar-elliptical coordinates
- converting between ellipse representations
- radial and polar-elliptical radial integration

Functions for measuring/fitting elliptical distortions are found in process/calibration/ellipse.py. Functions for computing radial and polar-elliptical radial backgrounds are found in process/preprocess/ellipse.py.

py4DSTEM uses 2 ellipse representations - one user-facing representation, and one internal representation. The user-facing represenation is in terms of the following 5 parameters:

x0,y0 the center of the ellipse a the semimajor axis length b the semiminor axis length theta the (positive, right handed) tilt of the a-axis

to the x-axis, in radians

Internally, fits are performed using the canonical ellipse parameterization, in terms of the parameters (x0,y0,A,B,C):

A(x-x0)^2 + B(x-x0)(y-y0) C(y-y0)^2 = 1

It is possible to convert between (a,b,theta) <–> (A,B,C) using the convert_ellipse_params() and convert_ellipse_params_r() methods.

Transformation from cartesian to polar-elliptical space is done using

x = x0 + a*r*cos(phi)*cos(theta) + b*r*sin(phi)*sin(theta) y = y0 + a*r*cos(phi)*sin(theta) - b*r*sin(phi)*cos(theta)

where (r,phi) are the polar-elliptical coordinates. All angular quantities are in radians.

py4DSTEM.process.utils.elliptical_coords.**convert_ellipse_params**(*A*, *B*, *C*)

Converts ellipse parameters from canonical form (A,B,C) into semi-axis lengths and tilt (a,b,theta). See module docstring for more info.

Parameters

- **A** (*floats*) – parameters of an ellipse in the form: Ax^2 + Bxy + Cy^2 = 1
- **B** (*floats*) – parameters of an ellipse in the form: Ax^2 + Bxy + Cy^2 = 1
- **C** (*floats*) – parameters of an ellipse in the form: Ax^2 + Bxy + Cy^2 = 1

Returns

A 3-tuple consisting of:

- **a**: (float) the semimajor axis length
- **b**: (float) the semiminor axis length
- **theta**: (float) the tilt of the ellipse semimajor axis with respect to the x-axis, in radians

Return type
(3-tuple)

py4DSTEM.process.utils.elliptical_coords.**convert_ellipse_params_r**(*a*, *b*, *theta*)

Converts from ellipse parameters (a,b,theta) to (A,B,C). See module docstring for more info.

Parameters

- **a** (*floats*) – parameters of an ellipse, where *a*/*b* are the semimajor/semiminor axis lengths, and theta is the tilt of the semimajor axis with respect to the x-axis, in radians.
- **b** (*floats*) – parameters of an ellipse, where *a*/*b* are the semimajor/semiminor axis lengths, and theta is the tilt of the semimajor axis with respect to the x-axis, in radians.
- **theta** (*floats*) – parameters of an ellipse, where *a*/*b* are the semimajor/semiminor axis lengths, and theta is the tilt of the semimajor axis with respect to the x-axis, in radians.

Returns

A 3-tuple consisting of (A,B,C), the ellipse parameters in
canonical form.

Return type
(3-tuple)

py4DSTEM.process.utils.elliptical_coords.**cartesian_to_polarelliptical_transform**(*cartesianData*, *p_ellipse*, *dr=1*, *dphi=0.03490658503988659*, *r_range=None*, *mask=None*, *maskThresh=0.99*)

Transforms an array of data in cartesian coordinates into a data array in polar-elliptical coordinates.

Discussion of the elliptical parametrization used can be found in the docstring for the process.utils.elliptical_coords module.

Parameters

- **cartesianData** (*2D float array*) – the data in cartesian coordinates
- **p_ellipse** (*5-tuple*) – specifies (qx0,qy0,a,b,theta), the parameters for the transformation. These are the same 5 parameters which are outputs of the elliptical fitting functions in the process.calibration module, e.g. fit_ellipse_amorphous_ring and fit_ellipse_1D. For more details, see the process.utils.elliptical_coords module docstring
- **dr** (*float*) – sampling of the (r,phi) coords: the width of the bins in r

- **dphi** (*float*) – sampling of the (r,phi) coords: the width of the bins in phi, in radians
- **r_range** (*number or length 2 list/tuple or None*) – specifies the sampling of the (r,theta) coords. Precise behavior which depends on the parameter type:
  - if None, autoselects max r value
  - if r_range is a number, specifies the maximum r value
  - if r_range is a length 2 list/tuple, specifies the min/max r values
- **mask** (*2d array of bools*) – shape must match cartesianData; where mask==False, ignore these datapoints in making the polarElliptical data array
- **maskThresh** (*float*) – the final data mask is calculated by converting mask (above) from cartesian to polar elliptical coords. Due to interpolation, this results in some non-boolean values - this is converted back to a boolean array by taking polarEllipticalMask = polarTrans(mask) < maskThresh. Cells where polarTrans is less than 1 (i.e. has at least one masked NN) should generally be masked, hence the default value of 0.99.

Returns

A 3-tuple, containing:

- **polarEllipticalData**: *(2D masked array)* a masked array containing the data and the data mask, in polarElliptical coordinates
- **rr**: *(2D array)* meshgrid of the r coordinates
- **pp**: *(2D array)* meshgrid of the phi coordinates

Return type
(3-tuple)

py4DSTEM.process.utils.elliptical_coords.**elliptical_resample_datacube**(*datacube*, *p_ellipse*, *mask=None*, *maskThresh=0.99*)

Perform elliptic resamplig on each diffraction pattern in a DataCube Detailed description of the args is found in `elliptical_resample`.

NOTE: Only use this function if you need to resample the raw data. If you only need for Bragg disk positions to be corrected, use the BraggVector calibration routines, as it is much faster to perform this on the peak positions than the entire datacube.

py4DSTEM.process.utils.elliptical_coords.**elliptical_resample**(*data*, *p_ellipse*, *mask=None*, *maskThresh=0.99*)

Resamples data with elliptic distortion to correct distortion of the input pattern.

Discussion of the elliptical parametrization used can be found in the docstring for the process.utils.elliptical_coords module.

Parameters

- **data** (*2D float array*) – the data in cartesian coordinates
- **p_ellipse** (*5-tuple*) – specifies (qx0,qy0,a,b,theta), the parameters for the transformation. These are the same 5 parameters which are outputs of the elliptical fitting functions in the process.calibration module, e.g. fit_ellipse_amorphous_ring and fit_ellipse_1D. For more details, see the process.utils.elliptical_coords module docstring
- **dr** (*float*) – sampling of the (r,phi) coords: the width of the bins in r
- **dphi** (*float*) – sampling of the (r,phi) coords: the width of the bins in phi, in radians

- **r_range** (*number or length 2 list/tuple or None*) – specifies the sampling of the (r,theta) coords. Precise behavior which depends on the parameter type:
  - if None, autoselects max r value
  - if r_range is a number, specifies the maximum r value
  - if r_range is a length 2 list/tuple, specifies the min/max r values
- **mask** (*2d array of bools*) – shape must match cartesianData; where mask==False, ignore these datapoints in making the polarElliptical data array
- **maskThresh** (*float*) – the final data mask is calculated by converting mask (above) from cartesian to polar elliptical coords. Due to interpolation, this results in some non-boolean values - this is converted back to a boolean array by taking polarEllipticalMask = polarTrans(mask) < maskThresh. Cells where polarTrans is less than 1 (i.e. has at least one masked NN) should generally be masked, hence the default value of 0.99.

Returns

A 3-tuple, containing:

- **resampled_data**: *(2D masked array)* a masked array containing the data and the data mask, in polarElliptical coordinates

Return type
(3-tuple)

py4DSTEM.process.utils.elliptical_coords.**radial_elliptical_integral**(*ar*, *dr*, *p_ellipse*, *rmax=None*)

Computes the radial integral of array ar from center (x0,y0) with a step size in r of dr.

Parameters

- **ar** (*2d array*) – the data
- **dr** (*number*) – the r sampling
- **p_ellipse** (*5-tuple*) – the parameters (x0,y0,a,b,theta) for the ellipse
- **r_max** (*float*) – maximum radial value

Returns

A 2-tuple containing:

- **rbin_centers**: *(1d array)* the bins centers of the radial integral
- **radial_integral**: *(1d array)* the radial integral

radial_integral (1d array) the radial integral

Return type
(2-tuple)

py4DSTEM.process.utils.elliptical_coords.**radial_integral**(*ar*, *x0=None*, *y0=None*, *dr=0.1*, *rmax=None*)

Computes the radial integral of array ar from center (x0,y0) with a step size in r of dr.

Parameters

- **ar** (*2d array*) – the data
- **x0** (*floats*) – the origin
- **y0** (*floats*) – the origin

- **dr** (*number*) – radial step size
- **rmax** (*float*) – maximum radial dimension

Returns

A 2-tuple containing:

- **rbin_centers**: *(1d array)* the bins centers of the radial integral
- **radial_integral**: *(1d array)* the radial integral

Return type
(2-tuple)

py4DSTEM.process.utils.masks.**get_beamstop_mask**(*dp*, *qx0*, *qy0*, *theta*, *dtheta=1*, *w=10*, *r=10*)

Generates a beamstop shaped mask.

Parameters

- **dp** (*2d array*) – a diffraction pattern
- **qx0** (*numbers*) – the center position of the beamstop
- **qy0** (*numbers*) – the center position of the beamstop
- **theta** (*number*) – the orientation of the beamstop, in degrees
- **dtheta** (*number*) – angular span of the wedge representing the beamstop, in degrees
- **w** (*integer*) – half the width of the beamstop arm, in pixels
- **r** (*number*) – the radius of a circle at the end of the beamstop, in pixels

Returns
the mask

Return type
(2d boolean array)

py4DSTEM.process.utils.masks.**make_circular_mask**(*shape*, *qxy0*, *radius*)

Create a hard circular mask, for use in DPC integration or or to use as a filter in diffraction or real space.

Parameters

- **shape** (*2-tuple of ints*) –
- **qxy0** (*2-tuple of floats) center coordinates, in pixels. Must be in (row, column*) –
- **radius** (*float*) –

Returns
mask (2D boolean array) the mask

loosely based on multicorr.py found at: https://github.com/ercius/openNCEM/blob/master/ncempy/algo/multicorr.py

modified by SEZ, May 2019 to integrate with py4DSTEM utility functions

- rewrote upsampleFFT (previously did not work correctly)
- modified upsampled_correlation to accept xyShift, the point around which to

upsample the DFT * eliminated the factor-2 FFT upsample step in favor of using parabolic for first-pass subpixel (since parabolic is so fast) * rewrote the matrix multiply DFT to be more pythonic

py4DSTEM.process.utils.multicorr.**upsampled_correlation**(*imageCorr*, *upsampleFactor*, *xyShift*, *device='cpu'*)

Refine the correlation peak of imageCorr around xyShift by DFT upsampling.

There are two approaches to Fourier upsampling for subpixel refinement: (a) one can pad an (appropriately shifted) FFT with zeros and take the inverse transform, or (b) one can compute the DFT by matrix multiplication using modified transformation matrices. The former approach is straightforward but requires performing the FFT algorithm (which is fast) on very large data. The latter method trades one speedup for a slowdown elsewhere: the matrix multiply steps are expensive but we operate on smaller matrices. Since we are only interested in a very small region of the FT around a peak of interest, we use the latter method to get a substantial speedup and enormous decrease in memory requirement. This "DFT upsampling" approach computes the transformation matrices for the matrix- multiply DFT around a small 1.5px wide region in the original *imageCorr*.

Following the matrix multiply DFT we use parabolic subpixel fitting to get even more precision! (below 1/upsampleFactor pixels)

NOTE: previous versions of multiCorr operated in two steps: using the zero- padding upsample method for a first-pass factor-2 upsampling, followed by the DFT upsampling (at whatever user-specified factor). I have implemented it differently, to better support iterating over multiple peaks. **The DFT is always upsampled around xyShift, which MUST be specified to HALF-PIXEL precision (no more, no less) to replicate the behavior of the factor-2 step.** (It is possible to refactor this so that peak detection is done on a Fourier upsampled image rather than using the parabolic subpixel and rounding as now… I like keeping it this way because all of the parameters and logic will be identical to the other subpixel methods.)

Parameters

- **imageCorr** (*complex valued ndarray*) – Complex product of the FFTs of the two images to be registered i.e. m = np.fft.fft2(DP) * probe_kernel_FT; imageCorr = np.abs(m)**(corrPower) * np.exp(1j*np.angle(m))
- **upsampleFactor** (*int*) – Upsampling factor. Must be greater than 2. (To do upsampling with factor 2, use upsampleFFT, which is faster.)
- **xyShift** – Location in original image coordinates around which to upsample the FT. This should be given to exactly half-pixel precision to replicate the initial FFT step that this implementation skips

Returns

Refined location of the peak in image coordinates.

Return type

(2-element np array)

py4DSTEM.process.utils.multicorr.**upsampleFFT**(*cc*, *device='cpu'*)

Zero-padding FFT upsampling. Returns the real IFFT of the input with 2x upsampling. This may have an error for matrices with an odd size. Takes a complex np array as input.

py4DSTEM.process.utils.multicorr.**dftUpsample**(*imageCorr*, *upsampleFactor*, *xyShift*, *device='cpu'*)

This performs a matrix multiply DFT around a small neighboring region of the inital correlation peak. By using the matrix multiply DFT to do the Fourier upsampling, the efficiency is greatly improved. This is adapted from the subfuction dftups found in the dftregistration function on the Matlab File Exchange.

https://www.mathworks.com/matlabcentral/fileexchange/18401-efficient-subpixel-image-registration-by-cross-correlation

The matrix multiplication DFT is from:

Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, "Efficient subpixel image registration algorithms," Opt. Lett. 33, 156-158 (2008). http://www.sciencedirect.com/science/article/pii/S0045790612000778

Parameters

- **imageCorr** (*complex valued ndarray*) – Correlation image between two images in Fourier space.
- **upsampleFactor** (*int*) – Scalar integer of how much to upsample.
- **xyShift** (*list of 2 floats*) – Coordinates in the UPSAMPLED GRID around which to upsample. These must be single-pixel IN THE UPSAMPLED GRID

Returns
Upsampled image from region around correlation peak.

Return type
(ndarray)

py4DSTEM.process.utils.utils.**radial_reduction**(*ar*, *x0*, *y0*, *binsize=1*, *fn=<function mean>*, *coords=None*)

Evaluate a reduction function on pixels within annular rings centered on (x0,y0), with a ring width of binsize.

By default, returns the mean value of pixels within each annulus. Some other useful reductions include: np.sum, np.std, np.count, np.median, ...

When running in a loop, pre-compute the pixel coordinates and pass them in for improved performance, like so:

coords = np.mgrid[0:ar.shape[0],0:ar.shape[1]] radial_sums = radial_reduction(ar, x0,y0, coords=coords)

py4DSTEM.process.utils.utils.**sector_mask**(*shape*, *centre*, *radius*, *angle_range=(0, 360)*)

Return a boolean mask for a circular sector. The start/stop angles in *angle_range* should be given in clockwise order.

Parameters

- **shape** – 2D shape of the mask
- **centre** – 2D center of the circular sector
- **radius** – radius of the circular mask
- **angle_range** – angular range of the circular mask

py4DSTEM.process.utils.utils.**get_qx_qy_1d**(*M*, *dx=[1, 1]*, *fft_shifted=False*)

Generates 1D Fourier coordinates for a (Nx,Ny)-shaped 2D array. Specifying the dx argument sets a unit size.

Parameters

- **M** – (2,) shape of the returned array
- **dx** – (2,) tuple, pixel size
- **fft_shifted** – True if result should be fft_shifted to have the origin in the center of the array

py4DSTEM.process.utils.utils.**get_CoM**(*ar*, *device='cpu'*, *corner_centered=False*)

Finds and returns the center of mass of array ar. If corner_centered is True, uses fftfreq for indices.

py4DSTEM.process.utils.utils.**get_maxima_1D**(*ar*, *sigma=0*, *minSpacing=0*, *minRelativeIntensity=0*, *relativeToPeak=0*)

Finds the indices where 1D array ar is a local maximum. Optional parameters allow blurring the array and filtering the output; setting each to 0 (default) turns off these functions.

Parameters

- **ar** (*1D array*) –
- **sigma** (*number*) – gaussian blur std to apply to ar before finding maxima

- **minSpacing** (*number*) – if two maxima are found within minSpacing, the dimmer one is removed
- **minRelativeIntensity** (*number*) – maxima dimmer than minRelativeIntensity compared to the relativeToPeak'th brightest maximum are removed
- **relativeToPeak** (*int*) – 0=brightest maximum. 1=next brightest, etc.

Returns
An array of indices where ar is a local maximum, sorted by intensity.

Return type
(array of ints)

py4DSTEM.process.utils.utils.**linear_interpolation_1D**(*ar*, *x*)

Calculates the 1D linear interpolation of array ar at position x using the two nearest elements.

py4DSTEM.process.utils.utils.**add_to_2D_array_from_floats**(*ar*, *x*, *y*, *I*)

Adds the values I to array ar, distributing the value between the four pixels nearest (x,y) using linear interpolation. Inputs (x,y,I) may be floats or arrays of floats.

Note that if the same [x,y] coordinate appears more than once in the input array, only the *final* value of I at that coordinate will get added.

py4DSTEM.process.utils.utils.**get_voronoi_vertices**(*voronoi*, *nx*, *ny*, *dist=10*)

From a scipy.spatial.Voronoi instance, return a list of ndarrays, where each array is shape (N,2) and contains the (x,y) positions of the vertices of a voronoi region.

The problem this function solves is that in a Voronoi instance, some vertices outside the field of view of the tesselated region are left unspecified; only the existence of a point beyond the field is referenced (which may or may not be 'at infinity'). This function specifies all points, such that the vertices and edges of the tesselation may be directly laid over data.

Parameters

- **voronoi** (*scipy.spatial.Voronoi*) – the voronoi tesselation
- **nx** (*int*) – the x field-of-view of the tesselated region
- **ny** (*int*) – the y field-of-view of the tesselated region
- **dist** (*float, optional*) – place new vertices by extending new voronoi edges outside the frame by a distance of this factor times the distance of its known vertex from the frame edge

Returns
the (x,y) coords of the vertices of each voronoi region

Return type
(list of ndarrays of shape (N,2))

py4DSTEM.process.utils.utils.**get_ewpc_filter_function**(*Q_Nx*, *Q_Ny*)

Returns a function for computing the exit wave power cepstrum of a diffraction pattern using a Hanning window. This can be passed as the filter_function in the Bragg disk detection functions (with the probe an array of ones) to find the lattice vectors by the EWPC method (but be careful as the lengths are now in realspace units!) See https://arxiv.org/abs/1911.00984

py4DSTEM.process.utils.utils.**fourier_resample**(*array*, *scale=None*, *output_size=None*, *force_nonnegative=False*, *bandlimit_nyquist=None*, *bandlimit_power=2*, *dtype=<class 'numpy.float32'>*, *conserve_array_sums=False*)

Resize a 2D array along any dimension, using Fourier interpolation / extrapolation. For 4D input arrays, only the final two axes can be resized.

The scaling of the array can be specified by passing either *scale*, which sets the scaling factor along both axes to be scaled; or by passing *output_size*, which specifies the final dimensions of the scaled axes (and allows for different scaling along the x,y or kx,ky axes.)

Parameters

- **array** (*2D/4D numpy array*) – Input array, or 4D stack of arrays, to be resized.
- **scale** (*float*) – scalar value giving the scaling factor for all dimensions
- **output_size** (*2-tuple of ints*) – two values giving either the (x,y) output size for 2D, or (kx,ky) for 4D
- **force_nonnegative** (*bool*) – Force all outputs to be nonnegative, after filtering
- **bandlimit_nyquist** (*float*) – Gaussian filter information limit in Nyquist units (0.5 max in both directions)
- **bandlimit_power** (*float*) – Gaussian filter power law scaling (higher is sharper)
- **dtype** (*numpy dtype*) – datatype for binned array. default is single precision float
- **conserve_arrray_sums** (*bool*) – If True, the sums of the array are conserved

Returns

the resized array (2D/4D numpy array)

virtualdiffraction

virtualimage

wholepatternfit

class py4DSTEM.process.wholepatternfit.wp_models.**WPFModelType**(*value*)

Flags to signify capabilities and other semantics of a Model

class py4DSTEM.process.wholepatternfit.wp_models.**WPFModel**(*name: str*, *params: dict*, *model_type=WPFModelType.DUMMY*)

Prototype class for a compent of a whole-pattern model. Holds the following:

name: human-readable name of the model params: a dict of names and initial (or returned) values of the model parameters func: a function that takes as arguments:

- the diffraction pattern being built up, which the function should modify in place
- positional arguments in the same order as the params dictionary
- **keyword arguments. this is to provide some pre-computed information for convenience**

  kwargs will include:

  - xArray, yArray meshgrid of the x and y coordinates
  - global_x0 global x-coordinate of the pattern center
  - global_y0 global y-coordinate of the pattern center

jacobian: a function that takes as arguments:

- the diffraction pattern being built up, which the function should modify in place
- positional arguments in the same order as the params dictionary
- **offset: the first index (j) that values should be written into**
  (the function should ONLY write into 0,1, and offset:offset+nParams) 0 and 1 are the entries for global_x0 and global_y0, respectively **REMEMBER TO ADD TO 0 and 1 SINCE ALL MODELS CAN CONTRIBUTE TO THIS PARTIAL DERIVATIVE**
- keyword arguments. this is to provide some pre-computed information for convenience

__init__(*name: str*, *params: dict*, *model_type=WPFModelType.DUMMY*)

class py4DSTEM.process.wholepatternfit.wp_models.**DCBackground**(*background_value=0.0*, *name='DC Background'*)

Model representing constant background intensity.

Parameters

background_value – Background intensity value. Specified as initial_value, (initial_value, deviation), or

(initial_value, lower_bound, upper_bound). See Parameter documentation for details.

__init__(*background_value=0.0*, *name='DC Background'*)

class py4DSTEM.process.wholepatternfit.wp_models.**GaussianBackground**(*WPF*, *sigma*, *intensity*, *global_center=True*, *x0=0.0*, *y0=0.0*, *name='Gaussian Background'*)

Model representing a 2D Gaussian intensity distribution

Parameters

- **WPF** (*WholePatternFit*) – Parent WPF object
- **sigma** – parameter specifying width of the Gaussian Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **intensity** – parameter specifying intensity of the Gaussian Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **global_center** (*bool*) – If True, uses same center coordinate as the global model If False, uses an independent center
- **x0** – Center coordinates of model for local origin Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **y0** – Center coordinates of model for local origin Specified as initial_value, (initial_value, deviation), or

(initial_value, lower_bound, upper_bound). See Parameter documentation for details.

__init__(*WPF*, *sigma*, *intensity*, *global_center=True*, *x0=0.0*, *y0=0.0*, *name='Gaussian Background'*)

class py4DSTEM.process.wholepatternfit.wp_models.**GaussianRing**(*WPF*, *radius*, *sigma*, *intensity*, *global_center=True*, *x0=0.0*, *y0=0.0*, *name='Gaussian Ring'*)

Model representing a halo with Gaussian falloff

Parameters

- **WPF** (*WholePatternFit*) – parent fitting object
- **radius** – radius of halo Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **sigma** – width of Gaussian falloff Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **intensity** – Intensity of the halo Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **global_center** (*bool*) – If True, uses same center coordinate as the global model If False, uses an independent center
- **x0** – Center coordinates of model for local origin Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **y0** – Center coordinates of model for local origin Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.

__init__(*WPF*, *radius*, *sigma*, *intensity*, *global_center=True*, *x0=0.0*, *y0=0.0*, *name='Gaussian Ring'*)

class py4DSTEM.process.wholepatternfit.wp_models.**SyntheticDiskLattice**(*WPF*, *ux: float*, *uy: float*, *vx: float*, *vy: float*, *disk_radius: float*, *disk_width: float*, *u_max: int*, *v_max: int*, *intensity_0: float*, *refine_radius: bool = False*, *refine_width: bool = False*, *global_center: bool = True*, *x0: float = 0.0*, *y0: float = 0.0*, *exclude_indices: list = []*, *include_indices: list | None = None*, *name='Synthetic Disk Lattice'*, *verbose=False*)

Model representing a lattice of diffraction disks with a soft edge

Parameters

- **WPF** (*WholePatternFit*) – parent fitting object
- **ux** – x and y components of the lattice vectors u and v. Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **uy** – x and y components of the lattice vectors u and v. Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **vx** – x and y components of the lattice vectors u and v. Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **vy** – x and y components of the lattice vectors u and v. Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **disk_radius** – Radius of each diffraction disk. Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **disk_width** – Width of the smooth falloff at the edge of the disk Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **u_max** – Maximum lattice indices to include in the pattern. Disks outside the pattern are automatically clipped.
- **v_max** – Maximum lattice indices to include in the pattern. Disks outside the pattern are automatically clipped.
- **intensity_0** – Initial intensity for each diffraction disk. Each disk intensity is an independent fit variable in the final model Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **refine_radius** (*bool*) – Flag whether disk radius is made a fitting parameter
- **refine_width** (*bool*) – Flag whether disk edge width is made a fitting parameter
- **global_center** (*bool*) – If True, uses same center coordinate as the global model If False, uses an independent center
- **x0** – Center coordinates of model for local origin Specified as initial_value, (initial_value, deviation), or

(initial_value, lower_bound, upper_bound). See Parameter documentation for details.

- **y0** – Center coordinates of model for local origin Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.

- **exclude_indices** (*list*) – Indices to exclude from the pattern
- **include_indices** (*list*) – If specified, only the indices in the list are added to the pattern

__init__(*WPF*, *ux: float*, *uy: float*, *vx: float*, *vy: float*, *disk_radius: float*, *disk_width: float*, *u_max: int*, *v_max: int*, *intensity_0: float*, *refine_radius: bool = False*, *refine_width: bool = False*, *global_center: bool = True*, *x0: float = 0.0*, *y0: float = 0.0*, *exclude_indices: list = []*, *include_indices: list | None = None*, *name='Synthetic Disk Lattice'*, *verbose=False*)

class py4DSTEM.process.wholepatternfit.wp_models.**SyntheticDiskMoire**(*WPF*, *lattice_a: SyntheticDiskLattice*, *lattice_b: SyntheticDiskLattice*, *intensity_0: float*, *decorated_peaks: list | None = None*, *link_moire_disk_intensities: bool = False*, *link_disk_parameters: bool = True*, *refine_width: bool = True*, *edge_width: list | None = None*, *refine_radius: bool = True*, *disk_radius: list | None = None*, *name: str = 'Moire Lattice'*)

Model of diffraction disks arising from interference between two lattices.

The Moire unit cell is determined automatically using the two input lattices.

Parameters

- **WPF** (*WholePatternFit*) – parent fitting object
- **lattice_a** (SyntheticDiskLattice) – parent lattices for the Moire
- **lattice_b** (SyntheticDiskLattice) – parent lattices for the Moire
- **intensity_0** – Initial guess of Moire disk intensity Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.

- **decorated_peaks** (*list*) – When specified, only the reflections in the list are decorated with Moire spots If not specified, all peaks are decorated
- **link_moire_disk_intensities** (*bool*) – When False, each Moire disk has an independently fit intensity When True, Moire disks arising from the same order of parent reflection share the same intensity

- **link_disk_parameters** (*bool*) – When True, edge_width and disk_radius are inherited from lattice_a
- **refine_width** (*bool*) – Flag whether disk edge width is a fit variable
- **edge_width** – Width of the soft edge of the diffraction disk. Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.
- **refine_radius** (*bool*) – Flag whether disk radius is a fit variable
- **radius** (*disk*) – Radius of the diffraction disks Specified as initial_value, (initial_value, deviation), or

  (initial_value, lower_bound, upper_bound). See Parameter documentation for details.

__init__(*WPF*, *lattice_a: SyntheticDiskLattice*, *lattice_b: SyntheticDiskLattice*, *intensity_0: float*, *decorated_peaks: list | None = None*, *link_moire_disk_intensities: bool = False*, *link_disk_parameters: bool = True*, *refine_width: bool = True*, *edge_width: list | None = None*, *refine_radius: bool = True*, *disk_radius: list | None = None*, *name: str = 'Moire Lattice'*)

class py4DSTEM.process.wholepatternfit.wp_models.**ComplexOverlapKernelDiskLattice**(*WPF*, *probe_kernel: ndarray*, *ux: float*, *uy: float*, *vx: float*, *vy: float*, *u_max: int*, *v_max: int*, *intensity_0: float*, *exclude_indices: list = []*, *global_center: bool = True*, *x0=0.0*, *y0=0.0*, *name='Complex Overlapped Disk Lattice'*, *verbose=False*)

__init__(*WPF*, *probe_kernel: ndarray*, *ux: float*, *uy: float*, *vx: float*, *vy: float*, *u_max: int*, *v_max: int*, *intensity_0: float*, *exclude_indices: list = []*, *global_center: bool = True*, *x0=0.0*, *y0=0.0*, *name='Complex Overlapped Disk Lattice'*, *verbose=False*)

class py4DSTEM.process.wholepatternfit.wp_models.**KernelDiskLattice**(*WPF*, *probe_kernel: ndarray*, *ux: float*, *uy: float*, *vx: float*, *vy: float*, *u_max: int*, *v_max: int*, *intensity_0: float*, *exclude_indices: list = []*, *global_center: bool = True*, *x0=0.0*, *y0=0.0*, *name='Custom Kernel Disk Lattice'*, *verbose=False*)

__init__(*WPF*, *probe_kernel: ndarray*, *ux: float*, *uy: float*, *vx: float*, *vy: float*, *u_max: int*, *v_max: int*, *intensity_0: float*, *exclude_indices: list = []*, *global_center: bool = True*, *x0=0.0*, *y0=0.0*, *name='Custom Kernel Disk Lattice'*, *verbose=False*)

py4DSTEM.process.wholepatternfit.wpf_viz.**show_lattice_points**(*self*, *im=None*, *vmin=None*, *vmax=None*, *power=None*, *show_vectors=True*, *crop_to_pattern=False*, *returnfig=False*, *moire_origin_idx=[0, 0, 0, 0]*, **args*, ***kwargs*)

Plotting utility to show the initial lattice points.

Parameters

- **im** (*np.ndarray*) – Optional: Image to show, defaults to mean CBED
- **vmin** (*float*) – Intensity ranges for plotting im
- **vmax** (*float*) – Intensity ranges for plotting im
- **power** (*float*) – Gamma level for showing im
- **show_vectors** (*bool*) – Flag to plot the lattice vectors
- **crop_to_pattern** (*bool*) – Flag to limit the field of view to the pattern area. If False, spots outside the pattern are shown
- **returnfig** (*bool*) – If True, (fig,ax) are returned and plt.show() is not called
- **moire_origin_idx** (*list of length 4*) – Indices of peak on which to draw Moire vectors, written as [a_u, a_v, b_u, b_v]
- **args** – Passed to plt.subplots
- **kwargs** – Passed to plt.subplots

Returns

fig,ax

Return type

If returnfig=True

1.4.6 visualize

Table of Contents

- *visualize*
  - *show*
  - *overlay*
  - *virtualimage*
  - *vis_RQ*
  - *vis_grid*
  - *vis_special*

show

py4DSTEM.visualize.**show**(*ar*, *figsize=(5, 5)*, *cmap='gray'*, *scaling='none'*, *intensity_range='ordered'*, *clipvals=None*, *vmin=None*, *vmax=None*, *min=None*, *max=None*, *power=None*, *power_offset=True*, *combine_images=False*, *ticks=True*, *bordercolor=None*, *borderwidth=5*, *show_image=True*, *return_ar_scaled=False*, *return_intensity_range=False*, *returncax=False*, *returnfig=False*, *figax=None*, *hist=False*, *n_bins=256*, *mask=None*, *mask_color='k'*, *mask_alpha=0.0*, *masked_intensity_range=False*, *rectangle=None*, *circle=None*, *annulus=None*, *ellipse=None*, *points=None*, *grid_overlay=None*, *cartesian_grid=None*, *polarelliptical_grid=None*, *rtheta_grid=None*, *scalebar=None*, *calibration=None*, *rx=None*, *ry=None*, *space='Q'*, *pixelsize=None*, *pixelunits=None*, *x0=None*, *y0=None*, *a=None*, *e=None*, *theta=None*, *title=None*, *show_fft=False*, *apply_hanning_window=True*, *show_cbar=False*, ***kwargs*)

General visualization function for 2D arrays.

The simplest use of this function is:

```
>>> show(ar)
```

which will generate and display a matplotlib figure showing the 2D array `ar`. Additional functionality includes:

- scaling the image (log scaling, power law scaling)
- displaying the image histogram
- altering the histogram clip values
- masking some subset of the image
- setting the colormap
- adding geometric overlays (e.g. points, circles, rectangles, annuli)
- adding informational overlays (scalebars, coordinate grids, oriented axes or vectors)
- further customization tools

These are each discussed in turn below.

Scaling:

Setting the parameter `scaling` will scale the display image. Options are 'none', 'auto', 'power', or 'log'. If

'power' is specified, the parameter `power` must also be passed. The underlying data is not altered. Values less than or equal to zero are set to zero. If the image histogram is displayed using `hist=True`, the scaled image histogram is shown.

Examples:

```
>>> show(ar,scaling='log')
>>> show(ar,power=0.5)
>>> show(ar,scaling='power',power=0.5,hist=True)
```

Histogram:

Setting the argument `hist=True` will display the image histogram, instead of the image. The displayed histogram will reflect any scaling requested. The number of bins can be set with `n_bins`. The upper and lower clip values, indicating where the image display will be saturated, are shown with dashed lines.

Intensity range:

Controlling the lower and upper values at which the display image will be saturated is accomplished with the `intensity_range` parameter, or its (soon deprecated) alias `clipvals`, in combination with `vmin`, and `vmax`. The method by which the upper and lower clip values are determined is controlled by `intensity_range`, and must be a string in ('None','ordered','minmax','absolute','std','centered'). See the argument description for `intensity_range` for a description of the behavior for each. The clip values can be returned with the `return_intensity_range` parameter.

Masking:

If a numpy masked array is passed to show, the function will automatically mask the appropriate pixels. Alternatively, a boolean array of the same shape as the data array may be passed to the `mask` argument, and these pixels will be masked. Masked pixels are displayed as a single uniform color, black by default, and which can be specified with the `mask_color` argument. Masked pixels are excluded when displaying the histogram or computing clip values. The mask can also be blended with the hidden data by setting the `mask_alpha` argument.

Overlays (geometric):

The function natively supports overlaying points, circles, rectangles, annuli, and ellipses. Each is invoked by passing a dictionary to the appropriate input variable specifying the geometry and features of the requested overlay. For example:

```
>>> show(ar, rectangle={'lims':(10,20,10,20),'color':'r'})
```

will overlay a single red square, and

```
>>> show(ar, annulus={'center':[(28,68),(92,160)],
                      'radii':[(16,24),(12,36)],
                      'fill':True,
                      'alpha':[0.9,0.3],
                      'color':['r',(0,1,1,1)]})
```

will overlay two annuli with two different centers, radii, colors, and transparencies. For a description of the accepted dictionary parameters for each type of overlay, see the visualize functions add_*, where * = ('rectangle','circle','annulus','ellipse','points'). (These docstrings are under construction!)

Overlays (informational):

Informational overlays supported by this function include coordinate axes (cartesian, polar-elliptical, or r-theta) and scalebars. These are added by passing the appropriate input argument a dictionary of the desired parameters, as with geometric overlays. However, there are two key differences between these overlays and the geometric overlays. First, informational overlays (coordinate systems and scalebars) require information about the plot - e.g. the position of the origin, the pixel sizes, the pixel units, any elliptical distortions, etc. The easiest way to pass this information is by pass a Calibration object containing this info to `show`

as the keyword `calibration`. Second, once the coordinate information has been passed, informational overlays can autoselect their own parameters, thus simply passing an empty dict to one of these parameters will add that overlay.

For example:

```
>>> show(dp, scalebar={}, calibration=calibration)
```

will display the diffraction pattern `dp` with a scalebar overlaid in the bottom left corner given the pixel size and units described in `calibration`, and

```
>>> show(dp, calibration=calibration, scalebar={'length':0.5,'width':2,
                                                'position':'ul','label':True'})
```

will display a more customized scalebar.

When overlaying coordinate grids, it is important to note that some relevant parameters, e.g. the position of the origin, may change by scan position. In these cases, the parameters `rx`,``ry`` must also be passed to `show`, to tell the `Calibration` object where to look for the relevant parameters. For example:

```
>>> show(dp, cartesian_grid={}, calibration=calibration, rx=2,ry=5)
```

will overlay a cartesian coordinate grid on the diffraction pattern at scan position (2,5). Adding

```
>>> show(dp, calibration=calibration, rx=2, ry=5, cartesian_grid={'label':True,
            'alpha':0.7,'color':'r'})
```

will customize the appearance of the grid further. And

```
>>> show(im, calibration=calibration, cartesian_grid={}, space='R')
```

displays a cartesian grid over a real space image. For more details, see the documentation for the visualize functions add_*, where * = ('scalebar', 'cartesian_grid', 'polarelliptical_grid', 'rtheta_grid'). (Under construction!)

Further customization:

Most parameters accepted by a matplotlib axis will be accepted by `show`. Pass a valid matplotlib colormap or a known string indicating a colormap as the argument `cmap` to specify the colormap. Pass `figsize` to specify the figure size. Etc.

Further customization can be accomplished by either (1) returning the figure generated by show and then manipulating it using the normal matplotlib functions, or (2) generating a matplotlib Figure with Axes any way you like (e.g. with `plt.subplots`) and then using this function to plot inside a single one of the Axes of your choice.

Option (1) is accomplished by simply passing this function `returnfig=True`. Thus:

```
>>> fig,ax = show(ar, returnfig=True)
```

will now give you direct access to the figure and axes to continue to alter. Option (2) is accomplished by passing an existing figure and axis to `show` as a 2-tuple to the `figax` argument. Thus:

```
>>> fig,(ax1,ax2) = plt.subplots(1,2)
>>> show(ar, figax=(fig,ax1))
>>> show(ar, figax=(fig,ax2), hist=True)
```

will generate a 2-axis figure, and then plot the array `ar` as an image on the left, while plotting its histogram on the right.

Parameters

- **ar** (*2D array or a list of 2D arrays*) – the data to plot. Normally this is a 2D array of the data. If a list of 2D arrays is passed, plots a corresponding grid of images.
- **figsize** (*2-tuple*) – size of the plot
- **cmap** (*colormap*) – any matplotlib cmap; default is gray
- **scaling** (*str*) – selects a scaling scheme for the intensity values. Default is none. Accepted values:
  - 'none': do not scale intensity values
  - 'full': fill entire color range with sorted intensity values
  - 'power': power law scaling
  - 'log': values where ar<=0 are set to 0
- **intensity_range** (*str*) –

  method for setting clipvalues (min and max intensities).
  The original name "clipvals" is now deprecated. Default is 'ordered'. Accepted values:

  - 'ordered': vmin/vmax are set to fractions of the distribution of pixel values in the array, e.g. vmin=0.02 will set the minumum display value to saturate the lower 2% of pixels
  - 'minmax': The vmin/vmax values are np.min(ar)/np.max(r)
  - 'absolute': The vmin/vmax values are set to the values of the vmin,vmax arguments received by this function
  - **'std': The vmin/vmax values are `np.median(ar) -/+ N*np.std(ar)`, and** N is this functions min,max vals.
  - 'centered': The vmin/vmax values are set to `c -/+ m`, where by default 'c' is zero and m is the max(abs(ar-c), or the two params can be user specified using the kwargs vmin/vmax -> c/m.
- **vmin** (*number*) – min intensity, behavior depends on clipvals
- **vmax** (*number*) – max intensity, behavior depends on clipvals
- **min** – alias' for vmin,vmax, throws deprecation warning
- **max** – alias' for vmin,vmax, throws deprecation warning
- **power** (*number*) – specifies the scaling power
- **power_offset** (*bool*) – If true, image has min value subtracted before power scaling
- **ticks** (*bool*) – Turn outer tick marks on or off
- **bordercolor** (*color or None*) – if not None, add a border of this color. The color can be anything matplotlib recognizes as a color.
- **borderwidth** (*number*) –
- **returnfig** (*bool*) – if True, the function returns the tuple (figure,axis)
- **figax** (*None or 2-tuple*) – controls which matplotlib Axes object draws the image. If None, generates a new figure with a single Axes instance. Otherwise, ax must be a

2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.

- **hist** (*bool*) – if True, instead of plotting a 2D image in ax, plots a histogram of the intensity values of ar, after any scaling this function has performed. Plots the clipvals as dashed vertical lines
- **n_bins** (*int*) – number of hist bins
- **mask** (*None or boolean array*) – if not None, must have the same shape as 'ar'. Wherever mask==True, plot the pixel normally, and where `mask==False`, pixel values are set to mask_color. If hist==True, ignore these values in the histogram. If `mask_alpha` is specified, the mask is blended with the array underneath, with 0 yielding an opaque mask and 1 yielding a fully transparent mask. If `mask_color` is set to `'empty'` instead of a matplotlib.color, nothing is done to pixels where `mask==False`, allowing overlaying multiple arrays in different regions of an image by invoking the ``figax` kwarg over multiple calls to show
- **mask_color** (*color*) – see 'mask'
- **mask_alpha** (*float*) – see 'mask'
- **masked_intensity_range** (*bool*) – controls if masked pixel values are included when determining the display value range; False indicates that all pixel values will be used to determine the intensity range, True indicates only unmasked pixels will be used
- **scalebar** (*None or dict or Bool*) – if None, and a DiffractionSlice or RealSlice with calibrations is passed, adds a scalebar. If scalebar is not displaying the proper calibration, check .calibration pixel_size and pixel_units. If None and an array is passed, does not add a scalebar. If a dict is passed, it is propagated to the add_scalebar function which will attempt to use it to overlay a scalebar. If True, uses calibraiton or pixelsize/pixelunits for scalebar. If False, no scalebar is added.
- **show_fft** (*bool*) – if True, plots 2D-fft of array
- **apply_hanning_window** (*bool*) – If True, a 2D Hann window is applied to the array before applying the FFT
- **show_cbar** (*bool*) – if True, adds cbar
- ****kwargs** – any keywords accepted by matplotlib's ax.matshow()

Returns

if returnfig==False (default), the figure is plotted and nothing is returned. if returnfig==True, return the figure and the axis.

py4DSTEM.visualize.**show_hist**(*arr*, *bins=200*, *vlines=None*, *vlinecolor='k'*, *vlinestyle='--'*, *returnhist=False*, *returnfig=False*)

Visualization function to show histogram from any ndarray (arr).

Accepts:

arr (ndarray) any array bins (int) number of bins that the intensity values will be sorted

into for histogram

returnhist (bool) determines whether or not the histogram values are

returned (see Returns)

returnfig (bool) determines whether or not figure and its axis are

returned (see Returns)

Returns

If

returnhist==False and returnfig==False returns nothing returnhist==True and returnfig==True returns (counts,bin_edges) the histogram

values and bin edge locations

returnhist==False and returnfig==True returns (fig,ax), the Figure and Axis returnhist==True and returnfig==True returns (hist,bin_edges),(fig,ax)

py4DSTEM.visualize.**show_Q**(*ar*, *scalebar=True*, *grid=False*, *polargrid=False*, *Q_pixel_size=None*, *Q_pixel_units=None*, *calibration=None*, *rx=None*, *ry=None*, *qx0=None*, *qy0=None*, *e=None*, *theta=None*, *scalebarloc=0*, *scalebarsize=None*, *scalebarwidth=None*, *scalebartext=None*, *scalebartextloc='above'*, *scalebartextsize=12*, *gridspacing=None*, *gridcolor='w'*, *majorgridlines=True*, *majorgridlw=1*, *majorgridls=':'*, *minorgridlines=True*, *minorgridlw=0.5*, *minorgridls=':'*, *gridlabels=False*, *gridlabelsize=12*, *gridlabelcolor='k'*, *alpha=0.35*, ***kwargs*)

Shows a diffraction space image with options for several overlays to define the scale, including a scalebar, a cartesian grid, or a polar / polar-elliptical grid.

Regardless of which overlay is requested, the function must recieve either values for Q_pixel_size and Q_pixel_units, or a Calibration instance containing these values. If both are passed, the absolutely passed values take precedence. If a cartesian grid is requested, (qx0,qy0) are required, either passed absolutely or passed as a Calibration instance with the appropriate (rx,ry) value. If a polar grid is requested, (qx0,qy0,e,theta) are required, again either absolutely or via a Calibration instance.

Any arguments accepted by the show() function (e.g. image scaling, clipvalues, etc) may be passed to this function as kwargs.

py4DSTEM.visualize.**show_rectangles**(*ar*, *lims=(0, 1, 0, 1)*, *color='r'*, *fill=True*, *alpha=0.25*, *linewidth=2*, *returnfig=False*, ***kwargs*)

Visualization function which plots a 2D array with one or more overlayed rectangles. lims is specified in the order (x0,xf,y0,yf). The rectangle bounds begin at the upper left corner of (x0,y0) and end at the upper left corner of (xf,yf) – i.e inclusive in the lower bound, exclusive in the upper bound – so that the boxed region encloses the area of array ar specified by ar[x0:xf,y0:yf].

To overlay one rectangle, lims must be a single 4-tuple. To overlay N rectangles, lims must be a list of N 4-tuples. color, fill, and alpha may each be single values, which are then applied to all the rectangles, or a length N list.

See the docstring for py4DSTEM.visualize.show() for descriptions of all input parameters not listed below.

Accepts:

lims (4-tuple, or list of N 4-tuples) the rectangle bounds (x0,xf,y0,yf) color (valid matplotlib color, or list of N colors) fill (bool or list of N bools) filled in or empty rectangles alpha (number, 0 to 1) transparency linewidth (number)

Returns

If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.**show_circles**(*ar*, *center*, *R*, *color='r'*, *fill=True*, *alpha=0.3*, *linewidth=2*, *returnfig=False*, ***kwargs*)

Visualization function which plots a 2D array with one or more overlayed circles. To overlay one circle, center must be a single 2-tuple. To overlay N circles, center must be a list of N 2-tuples. color, fill, and alpha may each be single values, which are then applied to all the circles, or a length N list.

See the docstring for py4DSTEM.visualize.show() for descriptions of all input parameters not listed below.

Accepts:

ar (2D array) the data center (2-tuple, or list of N 2-tuples) the center of the circle (x0,y0) R (number of list of N numbers) the circles radius color (valid matplotlib color, or list of N colors) fill (bool or list of N bools) filled in or empty rectangles alpha (number, 0 to 1) transparency linewidth (number)

Returns

If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.**show_ellipses**(*ar*, *center*, *a*, *b*, *theta*, *color='r'*, *fill=True*, *alpha=0.3*, *linewidth=2*, *returnfig=False*, ***kwargs*)

Visualization function which plots a 2D array with one or more overlayed ellipses. To overlay one ellipse, center must be a single 2-tuple. To overlay N circles, center must be a list of N 2-tuples. Similarly, the remaining ellipse parameters - a, e, and theta - must each be a single number or a len-N list. color, fill, and alpha may each be single values, which are then applied to all the circles, or length N lists.

See the docstring for py4DSTEM.visualize.show() for descriptions of all input parameters not listed below.

Accepts:

center (2-tuple, or list of N 2-tuples) the center of the circle (x0,y0) a (number or list of N numbers) the semimajor axis length e (number or list of N numbers) ratio of semiminor/semimajor length theta (number or list of N numbers) the tilt angle in radians color (valid matplotlib color, or list of N colors) fill (bool or list of N bools) filled in or empty rectangles alpha (number, 0 to 1) transparency linewidth (number)

Returns

If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.**show_annuli**(*ar*, *center*, *radii*, *color='r'*, *fill=True*, *alpha=0.3*, *linewidth=2*, *returnfig=False*, ***kwargs*)

Visualization function which plots a 2D array with one or more overlayed annuli. To overlay one annulus, center must be a single 2-tuple. To overlay N annuli, center must be a list of N 2-tuples. color, fill, and alpha may each be single values, which are then applied to all the circles, or a length N list.

See the docstring for py4DSTEM.visualize.show() for descriptions of all input parameters not listed below.

Accepts:

center (2-tuple, or list of N 2-tuples) the center of the annulus (x0,y0) radii (2-tuple, or list of N 2-tuples) the inner and outer radii color (string of list of N strings) fill (bool or list of N bools) filled in or empty rectangles alpha (number, 0 to 1) transparency linewidth (number)

Returns

If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.**show_points**(*ar*, *x*, *y*, *s=1*, *scale=50*, *alpha=1*, *pointcolor='r'*, *open_circles=False*, *title=None*, *returnfig=False*, ***kwargs*)

Plots a 2D array with one or more points. x and y are the point centers and must have the same length, N. s is the relative point sizes, and must have length 1 or N. scale is the size of the largest point. pointcolor have length 1 or N.

Accepts:

ar (array) the image x,y (number or iterable of numbers) the point positions s (number or iterable of num-

bers) the relative point sizes scale (number) the maximum point size title (str) title for plot pointcolor alpha

Returns

If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

overlay

py4DSTEM.visualize.overlay.**add_annuli**(*ax*, *d*)

Adds one or more annuli to Axis ax using the parameters in dictionary d.

py4DSTEM.visualize.overlay.**add_bragg_index_labels**(*ax*, *d*)

Adds labels for indexed bragg directions to a plot, using the parameters in dict d.

The dictionary d has required and optional parameters as follows:

bragg_directions (req'd) (PointList) the Bragg directions. This PointList must have
the fields 'qx','qy','h', and 'k', and may optionally have 'l'

voffset (number) vertical offset for the labels hoffset (number) horizontal offset for the labels color (color) size (number) points (bool) pointsize (number) pointcolor (color)

py4DSTEM.visualize.overlay.**add_cartesian_grid**(*ax*, *d*)

Adds an overlaid cartesian coordinate grid over an image using the parameters in dictionary d.

The dictionary d has required and optional parameters as follows:

x0,y0 (req'd) the origin Nx,Ny (req'd) the image extent space (str) 'Q' or 'R' spacing (number) spacing between gridlines pixelsize (number) pixelunits (str) lw (number) ls (str) color (color) label (bool) labelsize (number) labelcolor (color) alpha (number)

py4DSTEM.visualize.overlay.**add_circles**(*ax*, *d*)

adds one or more circles to axis ax using the parameters in dictionary d.

py4DSTEM.visualize.overlay.**add_ellipses**(*ax*, *d*)

Adds one or more ellipses to axis ax using the parameters in dictionary d.

Parameters

- **center** –
- **a** –
- **b** –
- **theta** –
- **color** –
- **fill** –
- **alpha** –
- **linewidth** –
- **linestyle** –

py4DSTEM.visualize.overlay.**add_grid_overlay**(*ax*, *d*)

adds an overlaid grid over some subset of pixels in an image using the parameters in dictionary d.

The dictionary d has required and optional parameters as follows:

x0,y0 (req'd) (ints) the corner of the grid xL,xL (req'd) (ints) the extent of the grid color (color) linewidth (number) alpha (number)

py4DSTEM.visualize.overlay.**add_pointlabels**(*ax*, *d*)

adds number indices for a set of points to axis ax using the parameters in dictionary d.

py4DSTEM.visualize.overlay.**add_points**(*ax*, *d*)

adds one or more points to axis ax using the parameters in dictionary d.

py4DSTEM.visualize.overlay.**add_polarelliptical_grid**(*ax*, *d*)

adds an overlaid polar-ellitpical coordinate grid over an image using the parameters in dictionary d.

The dictionary d has required and optional parameters as follows:

x0,y0 (req'd) the origin e,theta (req'd) the ellipticity (a/b) and major axis angle (radians) Nx,Ny (req'd) the image extent space (str) 'Q' or 'R' spacing (number) spacing between radial gridlines N_thetalines (int) the number of theta gridlines pixelsize (number) pixelunits (str) lw (number) ls (str) color (color) label (bool) labelsize (number) labelcolor (color) alpha (number)

py4DSTEM.visualize.overlay.**add_rectangles**(*ax*, *d*)

Adds one or more rectangles to Axis ax using the parameters in dictionary d.

py4DSTEM.visualize.overlay.**add_rtheta_grid**(*ar*, *d*)

py4DSTEM.visualize.overlay.**add_scalebar**(*ax*, *d*)

Adds an overlaid scalebar to an image, using the parameters in dict d.

The dictionary d has required and optional parameters as follows:

Nx,Ny (req'd) the image extent space (str) 'Q' or 'R' length (number) the scalebar length width (number) the scalebar width pixelsize (number) pixelunits (str) color (color) label (bool) labelsize (number) labelcolor (color) alpha (number) position (str) 'ul','ur','bl', or 'br' for the

upperleft, upperright, bottomleft, bottomright

ticks (bool) if False, turns off image border ticks

py4DSTEM.visualize.overlay.**add_vector**(*ax*, *d*)

Adds a vector to an image, using the parameters in dict d.

The dictionary d has required and optional parameters as follows:

x0,y0 (req'd) the tail position vx,vy (req'd) the vector color (color) width (number) label (str) labelsize (number) labelcolor (color)

py4DSTEM.visualize.overlay.**get_nice_spacing**(*Nx*, *Ny*, *pixelsize*)

Get a nice distance for gridlines, scalebars, etc

Parameters

- **Nx** (*int*) – the image dimensions
- **Nx** – the image dimensions
- **pixelsize** (*float*) – the size of each pixel, in some units

Returns

A 3-tuple containing:

- **spacing_units**: the spacing in real units
- **spacing_pixels**:the spacing in pixels
- **spacing**: the leading digits of the spacing

Return type
(3-tuple)

py4DSTEM.visualize.overlay.**is_color_like**(*c*)

Return whether *c* can be interpreted as an RGB(A) color.

virtualimage

vis_RQ

py4DSTEM.visualize.vis_RQ.**ax_addaxes**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *width=1*, *color='r'*, *labelaxes=True*, *labelsize=12*, *labelcolor='r'*, *righthandedcoords=True*)

Adds a pair of x/y axes to the matplotlib subplot ax. The user supplies the x-axis direction with (vx,vy), and the y-axis is then chosen by rotating 90 degrees, in a direction set by righthandedcoords.

Accepts:
ax (matplotlib subplot) vx,vy (numbers) x,y components of the x-axis,

Only the orientation is used; the axis is normalized and rescaled by

vlength (number) the axis length x0,y0 (numbers) the origin of the axes labelaxes (bool) if True, label 'x' and 'y' righthandedcoords (bool) if True, y-axis is counterclockwise

with respect to x-axis

py4DSTEM.visualize.vis_RQ.**ax_addaxes_QtoR**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *QR_rotation*, *width=1*, *color='r'*, *labelaxes=True*, *labelsize=12*, *labelcolor='r'*)

Adds a pair of x/y axes to the matplotlib subplot ax. The user supplies the x-axis direction with (vx,vy) in reciprocal space coordinates, and the function transforms and displays the corresponding vector in real space.

Accepts:
ax (matplotlib subplot) vx,vy (numbers) x,y components of the x-axis,

in reciprocal space coordinates. Only the orientation is used; the axes are normalized and rescaled by

vlength (number) the axis length, in real space x0,y0 (numbers) the origin of the axes, in

real space

labelaxes (bool) if True, label 'x' and 'y' QR_rotation (number) the offset angle between real and

diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

py4DSTEM.visualize.vis_RQ.**ax_addaxes_RtoQ**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *QR_rotation*, *width=1*, *color='r'*, *labelaxes=True*, *labelsize=12*, *labelcolor='r'*)

Adds a pair of x/y axes to the matplotlib subplot ax. The user supplies the x-axis direction with (vx,vy) in real space coordinates, and the function transforms and displays the corresponding vector in reciprocal space.

Accepts:
ax (matplotlib subplot) vx,vy (numbers) x,y components of the x-axis,

in real space coordinates. Only the orientation is used; the axes are normalized and rescaled by

vlength (number) the axis length, in reciprocal space x0,y0 (numbers) the origin of the axes, in

reciprocal space

labelaxes (bool) if True, label 'x' and 'y' QR_rotation (number) the offset angle between real and

diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

py4DSTEM.visualize.vis_RQ.**ax_addvector**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *width=1*, *color='r'*)

Adds a vector to the subplot at ax.

Accepts:

ax (matplotlib subplot) vx,vy (numbers) x,y components of the vector

Only the orientation is used, vector is normalized and rescaled by

vlength (number) the vector length x0,y0 (numbers) the origin / vector tail position

py4DSTEM.visualize.vis_RQ.**ax_addvector_QtoR**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *QR_rotation*, *width=1*, *color='r'*)

Adds a vector to the subplot at ax, where the vector (vx,vy) passed to the function is in reciprocal space and the plotted vector is transformed into and plotted in real space.

Accepts:

ax (matplotlib subplot) vx,vy (numbers) x,y components of the vector,

in *reciprocal* space. Only the orientation is used, vector is normalized and rescaled by

vlength (number) the vector length, in *real* space x0,y0 (numbers) the origin / vector tail position,

in *real* space

QR_rotation (number) the offset angle between real and
diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

py4DSTEM.visualize.vis_RQ.**ax_addvector_RtoQ**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *QR_rotation*, *width=1*, *color='r'*)

Adds a vector to the subplot at ax, where the vector (vx,vy) passed to the function is in real space and the plotted vector is transformed into and plotted in reciprocal space.

Accepts:

ax (matplotlib subplot) vx,vy (numbers) x,y components of the vector,

in *real* space. Only the orientation is used, vector is normalized and rescaled by

vlength (number) the vector length, in *reciprocal*
space

x0,y0 (numbers) the origin / vector tail position,
in *reciprocal* space

QR_rotation (number) the offset angle between real and
diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

py4DSTEM.visualize.vis_RQ.**show**(*ar*, *figsize=(5, 5)*, *cmap='gray'*, *scaling='none'*, *intensity_range='ordered'*, *clipvals=None*, *vmin=None*, *vmax=None*, *min=None*, *max=None*, *power=None*, *power_offset=True*, *combine_images=False*, *ticks=True*, *bordercolor=None*, *borderwidth=5*, *show_image=True*, *return_ar_scaled=False*, *return_intensity_range=False*, *returncax=False*, *returnfig=False*, *figax=None*, *hist=False*, *n_bins=256*, *mask=None*, *mask_color='k'*, *mask_alpha=0.0*, *masked_intensity_range=False*, *rectangle=None*, *circle=None*, *annulus=None*, *ellipse=None*, *points=None*, *grid_overlay=None*, *cartesian_grid=None*, *polarelliptical_grid=None*, *rtheta_grid=None*, *scalebar=None*, *calibration=None*, *rx=None*, *ry=None*, *space='Q'*, *pixelsize=None*, *pixelunits=None*, *x0=None*, *y0=None*, *a=None*, *e=None*, *theta=None*, *title=None*, *show_fft=False*, *apply_hanning_window=True*, *show_cbar=False*, ***kwargs*)

General visualization function for 2D arrays.

The simplest use of this function is:

```
>>> show(ar)
```

which will generate and display a matplotlib figure showing the 2D array `ar`. Additional functionality includes:

- scaling the image (log scaling, power law scaling)
- displaying the image histogram
- altering the histogram clip values
- masking some subset of the image
- setting the colormap
- adding geometric overlays (e.g. points, circles, rectangles, annuli)
- adding informational overlays (scalebars, coordinate grids, oriented axes or vectors)
- further customization tools

These are each discussed in turn below.

Scaling:

Setting the parameter `scaling` will scale the display image. Options are 'none', 'auto', 'power', or 'log'. If 'power' is specified, the parameter `power` must also be passed. The underlying data is not altered. Values less than or equal to zero are set to zero. If the image histogram is displayed using `hist=True`, the scaled image histogram is shown.

Examples:

```
>>> show(ar,scaling='log')
>>> show(ar,power=0.5)
>>> show(ar,scaling='power',power=0.5,hist=True)
```

Histogram:

Setting the argument `hist=True` will display the image histogram, instead of the image. The displayed histogram will reflect any scaling requested. The number of bins can be set with `n_bins`. The upper and lower clip values, indicating where the image display will be saturated, are shown with dashed lines.

Intensity range:

Controlling the lower and upper values at which the display image will be saturated is accomplished with the `intensity_range` parameter, or its (soon deprecated) alias `clipvals`, in combination with `vmin`, and `vmax`. The method by which the upper and lower clip values are determined is controlled by

intensity_range, and must be a string in ('None','ordered','minmax','absolute','std','centered'). See the argument description for intensity_range for a description of the behavior for each. The clip values can be returned with the return_intensity_range parameter.

Masking:

If a numpy masked array is passed to show, the function will automatically mask the appropriate pixels. Alternatively, a boolean array of the same shape as the data array may be passed to the mask argument, and these pixels will be masked. Masked pixels are displayed as a single uniform color, black by default, and which can be specified with the mask_color argument. Masked pixels are excluded when displaying the histogram or computing clip values. The mask can also be blended with the hidden data by setting the mask_alpha argument.

Overlays (geometric):

The function natively supports overlaying points, circles, rectangles, annuli, and ellipses. Each is invoked by passing a dictionary to the appropriate input variable specifying the geometry and features of the requested overlay. For example:

```
>>> show(ar, rectangle={'lims':(10,20,10,20),'color':'r'})
```

will overlay a single red square, and

```
>>> show(ar, annulus={'center':[(28,68),(92,160)],
                      'radii':[(16,24),(12,36)],
                      'fill':True,
                      'alpha':[0.9,0.3],
                      'color':['r',(0,1,1,1)]})
```

will overlay two annuli with two different centers, radii, colors, and transparencies. For a description of the accepted dictionary parameters for each type of overlay, see the visualize functions add_*, where * = ('rectangle','circle','annulus','ellipse','points'). (These docstrings are under construction!)

Overlays (informational):

Informational overlays supported by this function include coordinate axes (cartesian, polar-elliptical, or r-theta) and scalebars. These are added by passing the appropriate input argument a dictionary of the desired parameters, as with geometric overlays. However, there are two key differences between these overlays and the geometric overlays. First, informational overlays (coordinate systems and scalebars) require information about the plot - e.g. the position of the origin, the pixel sizes, the pixel units, any elliptical distortions, etc. The easiest way to pass this information is by pass a Calibration object containing this info to show as the keyword calibration. Second, once the coordinate information has been passed, informational overlays can autoselect their own parameters, thus simply passing an empty dict to one of these parameters will add that overlay.

For example:

```
>>> show(dp, scalebar={}, calibration=calibration)
```

will display the diffraction pattern dp with a scalebar overlaid in the bottom left corner given the pixel size and units described in calibration, and

```
>>> show(dp, calibration=calibration, scalebar={'length':0.5,'width':2,
                                                'position':'ul','label':True})
```

will display a more customized scalebar.

When overlaying coordinate grids, it is important to note that some relevant parameters, e.g. the position of the origin, may change by scan position. In these cases, the parameters rx,``ry`` must also be passed to show, to tell the Calibration object where to look for the relevant parameters. For example:

```
>>> show(dp, cartesian_grid={}, calibration=calibration, rx=2,ry=5)
```

will overlay a cartesian coordinate grid on the diffraction pattern at scan position (2,5). Adding

```
>>> show(dp, calibration=calibration, rx=2, ry=5, cartesian_grid={'label':True,
            'alpha':0.7,'color':'r'})
```

will customize the appearance of the grid further. And

```
>>> show(im, calibration=calibration, cartesian_grid={}, space='R')
```

displays a cartesian grid over a real space image. For more details, see the documentation for the visualize functions add_*, where * = ('scalebar', 'cartesian_grid', 'polarelliptical_grid', 'rtheta_grid'). (Under construction!)

Further customization:

Most parameters accepted by a matplotlib axis will be accepted by `show`. Pass a valid matplotlib colormap or a known string indicating a colormap as the argument `cmap` to specify the colormap. Pass `figsize` to specify the figure size. Etc.

Further customization can be accomplished by either (1) returning the figure generated by show and then manipulating it using the normal matplotlib functions, or (2) generating a matplotlib Figure with Axes any way you like (e.g. with `plt.subplots`) and then using this function to plot inside a single one of the Axes of your choice.

Option (1) is accomplished by simply passing this function `returnfig=True`. Thus:

```
>>> fig,ax = show(ar, returnfig=True)
```

will now give you direct access to the figure and axes to continue to alter. Option (2) is accomplished by passing an existing figure and axis to `show` as a 2-tuple to the `figax` argument. Thus:

```
>>> fig,(ax1,ax2) = plt.subplots(1,2)
>>> show(ar, figax=(fig,ax1))
>>> show(ar, figax=(fig,ax2), hist=True)
```

will generate a 2-axis figure, and then plot the array `ar` as an image on the left, while plotting its histogram on the right.

Parameters

- **ar** (*2D array or a list of 2D arrays*) – the data to plot. Normally this is a 2D array of the data. If a list of 2D arrays is passed, plots a corresponding grid of images.
- **figsize** (*2-tuple*) – size of the plot
- **cmap** (*colormap*) – any matplotlib cmap; default is gray
- **scaling** (*str*) – selects a scaling scheme for the intensity values. Default is none. Accepted values:
  - 'none': do not scale intensity values
  - 'full': fill entire color range with sorted intensity values
  - 'power': power law scaling
  - 'log': values where ar<=0 are set to 0
- **intensity_range** (*str*) –

method for setting clipvalues (min and max intensities).
The original name "clipvals" is now deprecated. Default is 'ordered'. Accepted values:

- 'ordered': vmin/vmax are set to fractions of the distribution of pixel values in the array, e.g. vmin=0.02 will set the minumum display value to saturate the lower 2% of pixels
- 'minmax': The vmin/vmax values are np.min(ar)/np.max(r)
- 'absolute': The vmin/vmax values are set to the values of the vmin,vmax arguments received by this function
- **'std': The vmin/vmax values are `np.median(ar) -/+ N*np.std(ar)`, and** N is this functions min,max vals.
- 'centered': The vmin/vmax values are set to `c -/+ m`, where by default 'c' is zero and m is the max(abs(ar-c), or the two params can be user specified using the kwargs vmin/vmax -> c/m.

- **vmin** (*number*) – min intensity, behavior depends on clipvals
- **vmax** (*number*) – max intensity, behavior depends on clipvals
- **min** – alias' for vmin,vmax, throws deprecation warning
- **max** – alias' for vmin,vmax, throws deprecation warning
- **power** (*number*) – specifies the scaling power
- **power_offset** (*bool*) – If true, image has min value subtracted before power scaling
- **ticks** (*bool*) – Turn outer tick marks on or off
- **bordercolor** (*color or None*) – if not None, add a border of this color. The color can be anything matplotlib recognizes as a color.
- **borderwidth** (*number*) –
- **returnfig** (*bool*) – if True, the function returns the tuple (figure,axis)
- **figax** (*None or 2-tuple*) – controls which matplotlib Axes object draws the image. If None, generates a new figure with a single Axes instance. Otherwise, ax must be a 2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.
- **hist** (*bool*) – if True, instead of plotting a 2D image in ax, plots a histogram of the intensity values of ar, after any scaling this function has performed. Plots the clipvals as dashed vertical lines
- **n_bins** (*int*) – number of hist bins
- **mask** (*None or boolean array*) – if not None, must have the same shape as 'ar'. Wherever mask==True, plot the pixel normally, and where `mask==False`, pixel values are set to mask_color. If hist==True, ignore these values in the histogram. If `mask_alpha` is specified, the mask is blended with the array underneath, with 0 yielding an opaque mask and 1 yielding a fully transparent mask. If `mask_color` is set to `'empty'` instead of a matplotlib.color, nothing is done to pixels where `mask==False`, allowing overlaying multiple arrays in different regions of an image by invoking the ``figax` kwarg over multiple calls to show
- **mask_color** (*color*) – see 'mask'

- **mask_alpha** (*float*) – see 'mask'
- **masked_intensity_range** (*bool*) – controls if masked pixel values are included when determining the display value range; False indicates that all pixel values will be used to determine the intensity range, True indicates only unmasked pixels will be used
- **scalebar** (*None or dict or Bool*) – if None, and a DiffractionSlice or RealSlice with calibrations is passed, adds a scalebar. If scalebar is not displaying the proper calibration, check .calibration pixel_size and pixel_units. If None and an array is passed, does not add a scalebar. If a dict is passed, it is propagated to the add_scalebar function which will attempt to use it to overlay a scalebar. If True, uses calibraiton or pixelsize/pixelunits for scalebar. If False, no scalebar is added.
- **show_fft** (*bool*) – if True, plots 2D-fft of array
- **apply_hanning_window** (*bool*) – If True, a 2D Hann window is applied to the array before applying the FFT
- **show_cbar** (*bool*) – if True, adds cbar
- ****kwargs** – any keywords accepted by matplotlib's ax.matshow()

Returns

if returnfig==False (default), the figure is plotted and nothing is returned. if returnfig==True, return the figure and the axis.

py4DSTEM.visualize.vis_RQ.**show_RQ**(*realspace_image*, *diffractionspace_image*, *realspace_pdict={}*, *diffractionspace_pdict={'scaling': 'log'}*, *figsize=(12, 6)*, *returnfig=False*)

Shows side-by-side real/reciprocal space images.

Accepts:

realspace_image (2D array) diffractionspace_image (2D array) realspace_pdict (dictionary) arguments and values to pass

to the show() fn for the real space image

diffractionspace_pdict (dictionary)

py4DSTEM.visualize.vis_RQ.**show_RQ_axes**(*realspace_image*, *diffractionspace_image*, *realspace_pdict*, *diffractionspace_pdict*, *vx*, *vy*, *vlength_R*, *vlength_Q*, *x0_R*, *y0_R*, *x0_Q*, *y0_Q*, *QR_rotation*, *vector_space='R'*, *width_R=1*, *color_R='r'*, *width_Q=1*, *color_Q='r'*, *labelaxes=True*, *labelcolor_R='r'*, *labelcolor_Q='r'*, *labelsize_R=12*, *labelsize_Q=12*, *figsize=(12, 6)*, *returnfig=False*)

Shows side-by-side real/reciprocal space images with a set of corresponding coordinate axes overlaid in each. (vx,vy) specifies the x-axis, and the y-axis is rotated 90 degrees counterclockwise in reciprocal space (relevant in case of an R/Q transposition).

Accepts:

realspace_image (2D array) diffractionspace_image (2D array) realspace_pdict (dictionary) arguments and values to pass

to the show() fn for the real space image

diffractionspace_pdict (dictionary) vx,vy (numbers) x,y components of the x-axis

in either real or diffraction space, depending on the value of vector_space. Note (vx,vy) is used for the orientation only - the vectors are normalized and rescaled by

vlength_R,vlength_Q (number or 1D arrays) the vector length in each
space, in pixels

x0_R,y0_R,x0_Q,y0_Q (number) the origins / vector tail positions QR_rotation (number) the offset angle between real and

diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

vector_space (string) must be 'R' or 'Q'. Specifies
whether the (vx,vy) values passed to this function describes a real or diffracation space vector.

py4DSTEM.visualize.vis_RQ.**show_RQ_vector**(*realspace_image*, *diffractionspace_image*, *realspace_pdict*, *diffractionspace_pdict*, *vx*, *vy*, *vlength_R*, *vlength_Q*, *x0_R*, *y0_R*, *x0_Q*, *y0_Q*, *QR_rotation*, *vector_space='R'*, *width_R=1*, *color_R='r'*, *width_Q=1*, *color_Q='r'*, *figsize=(12, 6)*, *returnfig=False*)

Shows side-by-side real/reciprocal space images with a vector overlaid in each showing corresponding directions.

Accepts:

realspace_image (2D array) diffractionspace_image (2D array) realspace_pdict (dictionary) arguments and values to pass

to the show() fn for the real space image

diffractionspace_pdict (dictionary) vx,vy (numbers) x,y components of the vector

in either real or diffraction space, depending on the value of vector_space. Note (vx,vy) is used for the orientation only - the two vectors are normalized and rescaled by

vlength_R,vlength_Q (number) the vector length in each
space, in pixels

x0_R,y0_R,x0_Q,y0_Q (numbers) the origins / vector tail positions QR_rotation (number) the offset angle between real and

diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

vector_space (string) must be 'R' or 'Q'. Specifies
whether the (vx,vy) values passed to this function describes a real or diffracation space vector.

py4DSTEM.visualize.vis_RQ.**show_RQ_vectors**(*realspace_image*, *diffractionspace_image*, *realspace_pdict*, *diffractionspace_pdict*, *vx*, *vy*, *vlength_R*, *vlength_Q*, *x0_R*, *y0_R*, *x0_Q*, *y0_Q*, *QR_rotation*, *vector_space='R'*, *width_R=1*, *color_R='r'*, *width_Q=1*, *color_Q='r'*, *figsize=(12, 6)*, *returnfig=False*)

Shows side-by-side real/reciprocal space images with several vectors overlaid in each showing corresponding directions.

Accepts:

realspace_image (2D array) diffractionspace_image (2D array) realspace_pdict (dictionary) arguments and values to pass

to the show() fn for the real space image

diffractionspace_pdict (dictionary) vx,vy (1D arrays) x,y components of the vectors

in either real or diffraction space, depending on the value of vector_space. Note (vx,vy) is used for the orientation only - the two vectors are normalized and rescaled by

vlength_R,vlenght_Q (number) the vector length in each
space, in pixels

x0_R,y0_R,x0_Q,y0_Q (numbers) the origins / vector tail positions QR_rotation (number) the offset angle between real and

diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

vector_space (string) must be 'R' or 'Q'. Specifies
whether the (vx,vy) values passed to this function describes a real or diffracation space vector.

py4DSTEM.visualize.vis_RQ.**show_points**(*ar*, *x*, *y*, *s=1*, *scale=50*, *alpha=1*, *pointcolor='r'*, *open_circles=False*, *title=None*, *returnfig=False*, ***kwargs*)

Plots a 2D array with one or more points. x and y are the point centers and must have the same length, N. s is the relative point sizes, and must have length 1 or N. scale is the size of the largest point. pointcolor have length 1 or N.

Accepts:
ar (array) the image x,y (number or iterable of numbers) the point positions s (number or iterable of numbers) the relative point sizes scale (number) the maximum point size title (str) title for plot pointcolor alpha

Returns
If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.vis_RQ.**show_selected_dp**(*datacube*, *image*, *rx*, *ry*, *figsize=(12, 6)*, *returnfig=False*, *pointsize=50*, *pointcolor='r'*, *scaling='log'*, ***kwargs*)

vis_grid

py4DSTEM.visualize.vis_grid.**_show_grid_overlay**(*image*, *x0*, *y0*, *xL*, *yL*, *color='k'*, *linewidth=1*, *alpha=1*, *returnfig=False*, ***kwargs*)

Shows the image with an overlaid boxgrid outline about the pixels beginning at (x0,y0) and with extent xL,yL in the two directions.

Accepts:
image the image array x0,y0 the corner of the grid xL,xL the extent of the grid

py4DSTEM.visualize.vis_grid.**add_grid_overlay**(*ax*, *d*)

adds an overlaid grid over some subset of pixels in an image using the parameters in dictionary d.

The dictionary d has required and optional parameters as follows:
x0,y0 (req'd) (ints) the corner of the grid xL,xL (req'd) (ints) the extent of the grid color (color) linewidth (number) alpha (number)

py4DSTEM.visualize.vis_grid.**show**(*ar*, *figsize=(5, 5)*, *cmap='gray'*, *scaling='none'*, *intensity_range='ordered'*, *clipvals=None*, *vmin=None*, *vmax=None*, *min=None*, *max=None*, *power=None*, *power_offset=True*, *combine_images=False*, *ticks=True*, *bordercolor=None*, *borderwidth=5*, *show_image=True*, *return_ar_scaled=False*, *return_intensity_range=False*, *returncax=False*, *returnfig=False*, *figax=None*, *hist=False*, *n_bins=256*, *mask=None*, *mask_color='k'*, *mask_alpha=0.0*, *masked_intensity_range=False*, *rectangle=None*, *circle=None*, *annulus=None*, *ellipse=None*, *points=None*, *grid_overlay=None*, *cartesian_grid=None*, *polarelliptical_grid=None*, *rtheta_grid=None*, *scalebar=None*, *calibration=None*, *rx=None*, *ry=None*, *space='Q'*, *pixelsize=None*, *pixelunits=None*, *x0=None*, *y0=None*, *a=None*, *e=None*, *theta=None*, *title=None*, *show_fft=False*, *apply_hanning_window=True*, *show_cbar=False*, ***kwargs*)

General visualization function for 2D arrays.

The simplest use of this function is:

```
>>> show(ar)
```

which will generate and display a matplotlib figure showing the 2D array `ar`. Additional functionality includes:

- scaling the image (log scaling, power law scaling)
- displaying the image histogram
- altering the histogram clip values
- masking some subset of the image
- setting the colormap
- adding geometric overlays (e.g. points, circles, rectangles, annuli)
- adding informational overlays (scalebars, coordinate grids, oriented axes or vectors)
- further customization tools

These are each discussed in turn below.

Scaling:

Setting the parameter `scaling` will scale the display image. Options are 'none', 'auto', 'power', or 'log'. If 'power' is specified, the parameter `power` must also be passed. The underlying data is not altered. Values less than or equal to zero are set to zero. If the image histogram is displayed using `hist=True`, the scaled image histogram is shown.

Examples:

```
>>> show(ar,scaling='log')
>>> show(ar,power=0.5)
>>> show(ar,scaling='power',power=0.5,hist=True)
```

Histogram:

Setting the argument `hist=True` will display the image histogram, instead of the image. The displayed histogram will reflect any scaling requested. The number of bins can be set with `n_bins`. The upper and lower clip values, indicating where the image display will be saturated, are shown with dashed lines.

Intensity range:

Controlling the lower and upper values at which the display image will be saturated is accomplished with the `intensity_range` parameter, or its (soon deprecated) alias `clipvals`, in combination with

vmin, and vmax. The method by which the upper and lower clip values are determined is controlled by intensity_range, and must be a string in ('None','ordered','minmax','absolute','std','centered'). See the argument description for intensity_range for a description of the behavior for each. The clip values can be returned with the return_intensity_range parameter.

Masking:

If a numpy masked array is passed to show, the function will automatically mask the appropriate pixels. Alternatively, a boolean array of the same shape as the data array may be passed to the mask argument, and these pixels will be masked. Masked pixels are displayed as a single uniform color, black by default, and which can be specified with the mask_color argument. Masked pixels are excluded when displaying the histogram or computing clip values. The mask can also be blended with the hidden data by setting the mask_alpha argument.

Overlays (geometric):

The function natively supports overlaying points, circles, rectangles, annuli, and ellipses. Each is invoked by passing a dictionary to the appropriate input variable specifying the geometry and features of the requested overlay. For example:

```
>>> show(ar, rectangle={'lims':(10,20,10,20),'color':'r'})
```

will overlay a single red square, and

```
>>> show(ar, annulus={'center':[(28,68),(92,160)],
                      'radii':[(16,24),(12,36)],
                      'fill':True,
                      'alpha':[0.9,0.3],
                      'color':['r',(0,1,1,1)]})
```

will overlay two annuli with two different centers, radii, colors, and transparencies. For a description of the accepted dictionary parameters for each type of overlay, see the visualize functions add_*, where * = ('rectangle','circle','annulus','ellipse','points'). (These docstrings are under construction!)

Overlays (informational):

Informational overlays supported by this function include coordinate axes (cartesian, polar-elliptical, or r-theta) and scalebars. These are added by passing the appropriate input argument a dictionary of the desired parameters, as with geometric overlays. However, there are two key differences between these overlays and the geometric overlays. First, informational overlays (coordinate systems and scalebars) require information about the plot - e.g. the position of the origin, the pixel sizes, the pixel units, any elliptical distortions, etc. The easiest way to pass this information is by pass a Calibration object containing this info to show as the keyword calibration. Second, once the coordinate information has been passed, informational overlays can autoselect their own parameters, thus simply passing an empty dict to one of these parameters will add that overlay.

For example:

```
>>> show(dp, scalebar={}, calibration=calibration)
```

will display the diffraction pattern dp with a scalebar overlaid in the bottom left corner given the pixel size and units described in calibration, and

```
>>> show(dp, calibration=calibration, scalebar={'length':0.5,'width':2,
                                                'position':'ul','label':True})
```

will display a more customized scalebar.

When overlaying coordinate grids, it is important to note that some relevant parameters, e.g. the position of the origin, may change by scan position. In these cases, the parameters rx,``ry`` must also be passed to

show, to tell the Calibration object where to look for the relevant parameters. For example:

```
>>> show(dp, cartesian_grid={}, calibration=calibration, rx=2,ry=5)
```

will overlay a cartesian coordinate grid on the diffraction pattern at scan position (2,5). Adding

```
>>> show(dp, calibration=calibration, rx=2, ry=5, cartesian_grid={'label':True,
            'alpha':0.7,'color':'r'})
```

will customize the appearance of the grid further. And

```
>>> show(im, calibration=calibration, cartesian_grid={}, space='R')
```

displays a cartesian grid over a real space image. For more details, see the documentation for the visualize functions add_*, where * = ('scalebar', 'cartesian_grid', 'polarelliptical_grid', 'rtheta_grid'). (Under construction!)

Further customization:

Most parameters accepted by a matplotlib axis will be accepted by show. Pass a valid matplotlib colormap or a known string indicating a colormap as the argument cmap to specify the colormap. Pass figsize to specify the figure size. Etc.

Further customization can be accomplished by either (1) returning the figure generated by show and then manipulating it using the normal matplotlib functions, or (2) generating a matplotlib Figure with Axes any way you like (e.g. with plt.subplots) and then using this function to plot inside a single one of the Axes of your choice.

Option (1) is accomplished by simply passing this function returnfig=True. Thus:

```
>>> fig,ax = show(ar, returnfig=True)
```

will now give you direct access to the figure and axes to continue to alter. Option (2) is accomplished by passing an existing figure and axis to show as a 2-tuple to the figax argument. Thus:

```
>>> fig,(ax1,ax2) = plt.subplots(1,2)
>>> show(ar, figax=(fig,ax1))
>>> show(ar, figax=(fig,ax2), hist=True)
```

will generate a 2-axis figure, and then plot the array ar as an image on the left, while plotting its histogram on the right.

Parameters

- **ar** (*2D array or a list of 2D arrays*) – the data to plot. Normally this is a 2D array of the data. If a list of 2D arrays is passed, plots a corresponding grid of images.
- **figsize** (*2-tuple*) – size of the plot
- **cmap** (*colormap*) – any matplotlib cmap; default is gray
- **scaling** (*str*) – selects a scaling scheme for the intensity values. Default is none. Accepted values:
  - 'none': do not scale intensity values
  - 'full': fill entire color range with sorted intensity values
  - 'power': power law scaling
  - 'log': values where ar<=0 are set to 0

- **intensity_range** (*str*) –

  method for setting clipvalues (min and max intensities).
  The original name "clipvals" is now deprecated. Default is 'ordered'. Accepted values:

  - 'ordered': vmin/vmax are set to fractions of the distribution of pixel values in the array, e.g. vmin=0.02 will set the minumum display value to saturate the lower 2% of pixels
  - 'minmax': The vmin/vmax values are np.min(ar)/np.max(r)
  - 'absolute': The vmin/vmax values are set to the values of the vmin,vmax arguments received by this function
  - **'std': The vmin/vmax values are `np.median(ar) -/+ N*np.std(ar)`, and** N is this functions min,max vals.
  - 'centered': The vmin/vmax values are set to `c -/+ m`, where by default 'c' is zero and m is the max(abs(ar-c), or the two params can be user specified using the kwargs vmin/vmax -> c/m.

- **vmin** (*number*) – min intensity, behavior depends on clipvals
- **vmax** (*number*) – max intensity, behavior depends on clipvals
- **min** – alias' for vmin,vmax, throws deprecation warning
- **max** – alias' for vmin,vmax, throws deprecation warning
- **power** (*number*) – specifies the scaling power
- **power_offset** (*bool*) – If true, image has min value subtracted before power scaling
- **ticks** (*bool*) – Turn outer tick marks on or off
- **bordercolor** (*color or None*) – if not None, add a border of this color. The color can be anything matplotlib recognizes as a color.
- **borderwidth** (*number*) –
- **returnfig** (*bool*) – if True, the function returns the tuple (figure,axis)
- **figax** (*None or 2-tuple*) – controls which matplotlib Axes object draws the image. If None, generates a new figure with a single Axes instance. Otherwise, ax must be a 2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.
- **hist** (*bool*) – if True, instead of plotting a 2D image in ax, plots a histogram of the intensity values of ar, after any scaling this function has performed. Plots the clipvals as dashed vertical lines
- **n_bins** (*int*) – number of hist bins
- **mask** (*None or boolean array*) – if not None, must have the same shape as 'ar'. Wherever mask==True, plot the pixel normally, and where `mask==False`, pixel values are set to mask_color. If hist==True, ignore these values in the histogram. If `mask_alpha` is specified, the mask is blended with the array underneath, with 0 yielding an opaque mask and 1 yielding a fully transparent mask. If `mask_color` is set to `'empty'` instead of a matplotlib.color, nothing is done to pixels where `mask==False`, allowing overlaying multiple arrays in different regions of an image by invoking the ``figax` kwarg over multiple calls to show

- **mask_color** (*color*) – see 'mask'
- **mask_alpha** (*float*) – see 'mask'
- **masked_intensity_range** (*bool*) – controls if masked pixel values are included when determining the display value range; False indicates that all pixel values will be used to determine the intensity range, True indicates only unmasked pixels will be used
- **scalebar** (*None or dict or Bool*) – if None, and a DiffractionSlice or RealSlice with calibrations is passed, adds a scalebar. If scalebar is not displaying the proper calibration, check .calibration pixel_size and pixel_units. If None and an array is passed, does not add a scalebar. If a dict is passed, it is propagated to the add_scalebar function which will attempt to use it to overlay a scalebar. If True, uses calibraiton or pixelsize/pixelunits for scalebar. If False, no scalebar is added.
- **show_fft** (*bool*) – if True, plots 2D-fft of array
- **apply_hanning_window** (*bool*) – If True, a 2D Hann window is applied to the array before applying the FFT
- **show_cbar** (*bool*) – if True, adds cbar
- ****kwargs** – any keywords accepted by matplotlib's ax.matshow()

Returns

if returnfig==False (default), the figure is plotted and nothing is returned. if returnfig==True, return the figure and the axis.

py4DSTEM.visualize.vis_grid.**show_DP_grid**(*datacube*, *x0*, *y0*, *xL*, *yL*, *axsize=(6, 6)*, *returnfig=False*, *space=0*, ***kwargs*)

Shows a grid of diffraction patterns from DataCube datacube, starting from scan position (x0,y0) and extending xL,yL.

Accepts:

datacube (DataCube) the 4D-STEM data (x0,y0) the corner of the grid of DPs to display xL,yL the extent of the grid axsize the size of each diffraction pattern space (number) controls the space between subplots

Returns

if returnfig==false (default), the figure is plotted and nothing is returned. if returnfig==false, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.vis_grid.**show_grid_overlay**(*image*, *x0*, *y0*, *xL*, *yL*, *color='k'*, *linewidth=1*, *alpha=1*, *returnfig=False*, ***kwargs*)

Shows the image with an overlaid boxgrid outline about the pixels beginning at (x0,y0) and with extent xL,yL in the two directions.

Accepts:

image the image array x0,y0 the corner of the grid xL,xL the extent of the grid

py4DSTEM.visualize.vis_grid.**show_image_grid**(*get_ar*, *H*, *W*, *axsize=(6, 6)*, *returnfig=False*, *figax=None*, *title=None*, *title_index=False*, *suptitle=None*, *get_bordercolor=None*, *get_x=None*, *get_y=None*, *get_pointcolors=None*, *get_s=None*, *open_circles=False*, ***kwargs*)

Displays a set of images in a grid.

The images are specified by some function get_ar(i), which returns an image for values of some integer index i. The values of i passed to get_ar are 0 through HW-1.

To display the first 4 two-dimensional slices of some 3D array ar some 3D array ar, you can do

```
>>> show_image_grid(lambda i:ar[:,:,i], H=2, W=2)
```

Its also possible to add colored borders, or overlaid points, using similar functions to get_ar, i.e. functions which return the color or set of points of interest as a function of index i, which must be defined in the range [0,HW-1].

Accepts:

get_ar a function which returns a 2D array when passed
the integers 0 through HW-1

H,W integers, the dimensions of the grid axsize the size of each image figax controls which matplotlib Axes object draws the image.

If None, generates a new figure with a single Axes instance. Otherwise, ax must be a 2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.

title if title is sting, then prints title as suptitle. If a suptitle is also provided,
the suptitle is printed insead. if title is a list of strings (ex: ['title 1','title 2']), each array has corresponding title in list.

title_index if True, prints the index i passed to get_ar over each image suptitle string, suptitle on plot get_bordercolor

if not None, should be a function defined over the same i as get_ar, and which returns a valid matplotlib color for each i. Adds a colored bounding box about each image. E.g. if *colors* is an array of colors:

```
>>> show_image_grid(lambda i:ar[:,:,i],H=2,W=2,
                    get_bordercolor=lambda i:colors[i])
```

get_x,get_y functions which returns sets of x/y positions
as a function of index i

get_s function which returns a set of point sizes
as a function of index i

get_pointcolors a function which returns a color or list of colors
as a function of index i

Returns
if returnfig==false (default), the figure is plotted and nothing is returned. if returnfig==false, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.vis_grid.**show_points**(*ar*, *x*, *y*, *s=1*, *scale=50*, *alpha=1*, *pointcolor='r'*, *open_circles=False*, *title=None*, *returnfig=False*, ***kwargs*)

Plots a 2D array with one or more points. x and y are the point centers and must have the same length, N. s is the relative point sizes, and must have length 1 or N. scale is the size of the largest point. pointcolor have length 1 or N.

Accepts:
ar (array) the image x,y (number or iterable of numbers) the point positions s (number or iterable of numbers) the relative point sizes scale (number) the maximum point size title (str) title for plot pointcolor alpha

Returns

If returnfig==False (default), the figure is plotted and nothing is returned. If returnfig==False, the figure and its one axis are returned, and can be further edited.

vis_special

py4DSTEM.visualize.vis_special.**Complex2RGB**(*complex_data*, *vmin=None*, *vmax=None*, *power=None*, *chroma_boost=1*)

complex_data (array): complex array to plot vmin (float) : minimum absolute value vmax (float) : maximum absolute value power (float) : power to raise amplitude to chroma_boost (float): boosts chroma for higher-contrast (~1-2.5)

py4DSTEM.visualize.vis_special.**add_bragg_index_labels**(*ax*, *d*)

Adds labels for indexed bragg directions to a plot, using the parameters in dict d.

The dictionary d has required and optional parameters as follows:

bragg_directions (req'd) (PointList) the Bragg directions. This PointList must have
the fields 'qx','qy','h', and 'k', and may optionally have 'l'

voffset (number) vertical offset for the labels hoffset (number) horizontal offset for the labels color (color) size (number) points (bool) pointsize (number) pointcolor (color)

py4DSTEM.visualize.vis_special.**add_ellipses**(*ax*, *d*)

Adds one or more ellipses to axis ax using the parameters in dictionary d.

Parameters

- **center** –
- **a** –
- **b** –
- **theta** –
- **color** –
- **fill** –
- **alpha** –
- **linewidth** –
- **linestyle** –

py4DSTEM.visualize.vis_special.**add_pointlabels**(*ax*, *d*)

adds number indices for a set of points to axis ax using the parameters in dictionary d.

py4DSTEM.visualize.vis_special.**add_points**(*ax*, *d*)

adds one or more points to axis ax using the parameters in dictionary d.

py4DSTEM.visualize.vis_special.**add_scalebar**(*ax*, *d*)

Adds an overlaid scalebar to an image, using the parameters in dict d.

The dictionary d has required and optional parameters as follows:

Nx,Ny (req'd) the image extent space (str) 'Q' or 'R' length (number) the scalebar length width (number) the scalebar width pixelsize (number) pixelunits (str) color (color) label (bool) labelsize (number) labelcolor (color) alpha (number) position (str) 'ul','ur','bl', or 'br' for the

upperleft, upperright, bottomleft, bottomright

ticks (bool) if False, turns off image border ticks

py4DSTEM.visualize.vis_special.**add_vector**(*ax*, *d*)

Adds a vector to an image, using the parameters in dict d.

The dictionary d has required and optional parameters as follows:

x0,y0 (req'd) the tail position vx,vy (req'd) the vector color (color) width (number) label (str) labelsize (number) labelcolor (color)

py4DSTEM.visualize.vis_special.**ax_addaxes**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *width=1*, *color='r'*, *labelaxes=True*, *labelsize=12*, *labelcolor='r'*, *righthandedcoords=True*)

Adds a pair of x/y axes to the matplotlib subplot ax. The user supplies the x-axis direction with (vx,vy), and the y-axis is then chosen by rotating 90 degrees, in a direction set by righthandedcoords.

Accepts:

ax (matplotlib subplot) vx,vy (numbers) x,y components of the x-axis,

Only the orientation is used; the axis is normalized and rescaled by

vlength (number) the axis length x0,y0 (numbers) the origin of the axes labelaxes (bool) if True, label 'x' and 'y' righthandedcoords (bool) if True, y-axis is counterclockwise

with respect to x-axis

py4DSTEM.visualize.vis_special.**ax_addaxes_QtoR**(*ax*, *vx*, *vy*, *vlength*, *x0*, *y0*, *QR_rotation*, *width=1*, *color='r'*, *labelaxes=True*, *labelsize=12*, *labelcolor='r'*)

Adds a pair of x/y axes to the matplotlib subplot ax. The user supplies the x-axis direction with (vx,vy) in reciprocal space coordinates, and the function transforms and displays the corresponding vector in real space.

Accepts:

ax (matplotlib subplot) vx,vy (numbers) x,y components of the x-axis,

in reciprocal space coordinates. Only the orientation is used; the axes are normalized and rescaled by

vlength (number) the axis length, in real space x0,y0 (numbers) the origin of the axes, in

real space

labelaxes (bool) if True, label 'x' and 'y' QR_rotation (number) the offset angle between real and

diffraction space. Specifically, this is the counterclockwise rotation of real space with respect to diffraction space. In degrees.

py4DSTEM.visualize.vis_special.**make_axes_locatable**(*axes*)

py4DSTEM.visualize.vis_special.**select_point**(*ar*, *x*, *y*, *i*, *color='lightblue'*, *color_selected='r'*, *size=20*, *returnfig=False*, ***kwargs*)

Show enumerated index labels for a set of points, with one selected point highlighted

py4DSTEM.visualize.vis_special.**show**(*ar*, *figsize=(5, 5)*, *cmap='gray'*, *scaling='none'*, *intensity_range='ordered'*, *clipvals=None*, *vmin=None*, *vmax=None*, *min=None*, *max=None*, *power=None*, *power_offset=True*, *combine_images=False*, *ticks=True*, *bordercolor=None*, *borderwidth=5*, *show_image=True*, *return_ar_scaled=False*, *return_intensity_range=False*, *returncax=False*, *returnfig=False*, *figax=None*, *hist=False*, *n_bins=256*, *mask=None*, *mask_color='k'*, *mask_alpha=0.0*, *masked_intensity_range=False*, *rectangle=None*, *circle=None*, *annulus=None*, *ellipse=None*, *points=None*, *grid_overlay=None*, *cartesian_grid=None*, *polarelliptical_grid=None*, *rtheta_grid=None*, *scalebar=None*, *calibration=None*, *rx=None*, *ry=None*, *space='Q'*, *pixelsize=None*, *pixelunits=None*, *x0=None*, *y0=None*, *a=None*, *e=None*, *theta=None*, *title=None*, *show_fft=False*, *apply_hanning_window=True*, *show_cbar=False*, ***kwargs*)

General visualization function for 2D arrays.

The simplest use of this function is:

```
>>> show(ar)
```

which will generate and display a matplotlib figure showing the 2D array `ar`. Additional functionality includes:

- scaling the image (log scaling, power law scaling)
- displaying the image histogram
- altering the histogram clip values
- masking some subset of the image
- setting the colormap
- adding geometric overlays (e.g. points, circles, rectangles, annuli)
- adding informational overlays (scalebars, coordinate grids, oriented axes or vectors)
- further customization tools

These are each discussed in turn below.

Scaling:

Setting the parameter `scaling` will scale the display image. Options are 'none', 'auto', 'power', or 'log'. If 'power' is specified, the parameter `power` must also be passed. The underlying data is not altered. Values less than or equal to zero are set to zero. If the image histogram is displayed using `hist=True`, the scaled image histogram is shown.

Examples:

```
>>> show(ar,scaling='log')
>>> show(ar,power=0.5)
>>> show(ar,scaling='power',power=0.5,hist=True)
```

Histogram:

Setting the argument `hist=True` will display the image histogram, instead of the image. The displayed histogram will reflect any scaling requested. The number of bins can be set with `n_bins`. The upper and lower clip values, indicating where the image display will be saturated, are shown with dashed lines.

Intensity range:

Controlling the lower and upper values at which the display image will be saturated is accomplished

with the `intensity_range` parameter, or its (soon deprecated) alias `clipvals`, in combination with `vmin`, and `vmax`. The method by which the upper and lower clip values are determined is controlled by `intensity_range`, and must be a string in ('None','ordered','minmax','absolute','std','centered'). See the argument description for `intensity_range` for a description of the behavior for each. The clip values can be returned with the `return_intensity_range` parameter.

Masking:

If a numpy masked array is passed to show, the function will automatically mask the appropriate pixels. Alternatively, a boolean array of the same shape as the data array may be passed to the `mask` argument, and these pixels will be masked. Masked pixels are displayed as a single uniform color, black by default, and which can be specified with the `mask_color` argument. Masked pixels are excluded when displaying the histogram or computing clip values. The mask can also be blended with the hidden data by setting the `mask_alpha` argument.

Overlays (geometric):

The function natively supports overlaying points, circles, rectangles, annuli, and ellipses. Each is invoked by passing a dictionary to the appropriate input variable specifying the geometry and features of the requested overlay. For example:

```
>>> show(ar, rectangle={'lims':(10,20,10,20),'color':'r'})
```

will overlay a single red square, and

```
>>> show(ar, annulus={'center':[(28,68),(92,160)],
                      'radii':[(16,24),(12,36)],
                      'fill':True,
                      'alpha':[0.9,0.3],
                      'color':['r',(0,1,1,1)]})
```

will overlay two annuli with two different centers, radii, colors, and transparencies. For a description of the accepted dictionary parameters for each type of overlay, see the visualize functions add_*, where * = ('rectangle','circle','annulus','ellipse','points'). (These docstrings are under construction!)

Overlays (informational):

Informational overlays supported by this function include coordinate axes (cartesian, polar-elliptical, or r-theta) and scalebars. These are added by passing the appropriate input argument a dictionary of the desired parameters, as with geometric overlays. However, there are two key differences between these overlays and the geometric overlays. First, informational overlays (coordinate systems and scalebars) require information about the plot - e.g. the position of the origin, the pixel sizes, the pixel units, any elliptical distortions, etc. The easiest way to pass this information is by pass a Calibration object containing this info to `show` as the keyword `calibration`. Second, once the coordinate information has been passed, informational overlays can autoselect their own parameters, thus simply passing an empty dict to one of these parameters will add that overlay.

For example:

```
>>> show(dp, scalebar={}, calibration=calibration)
```

will display the diffraction pattern `dp` with a scalebar overlaid in the bottom left corner given the pixel size and units described in `calibration`, and

```
>>> show(dp, calibration=calibration, scalebar={'length':0.5,'width':2,
                                                'position':'ul','label':True})
```

will display a more customized scalebar.

When overlaying coordinate grids, it is important to note that some relevant parameters, e.g. the position

of the origin, may change by scan position. In these cases, the parameters `rx`,``ry`` must also be passed to `show`, to tell the `Calibration` object where to look for the relevant parameters. For example:

```
>>> show(dp, cartesian_grid={}, calibration=calibration, rx=2,ry=5)
```

will overlay a cartesian coordinate grid on the diffraction pattern at scan position (2,5). Adding

```
>>> show(dp, calibration=calibration, rx=2, ry=5, cartesian_grid={'label':True,
            'alpha':0.7,'color':'r'})
```

will customize the appearance of the grid further. And

```
>>> show(im, calibration=calibration, cartesian_grid={}, space='R')
```

displays a cartesian grid over a real space image. For more details, see the documentation for the visualize functions add_*, where * = ('scalebar', 'cartesian_grid', 'polarelliptical_grid', 'rtheta_grid'). (Under construction!)

Further customization:

Most parameters accepted by a matplotlib axis will be accepted by `show`. Pass a valid matplotlib colormap or a known string indicating a colormap as the argument `cmap` to specify the colormap. Pass `figsize` to specify the figure size. Etc.

Further customization can be accomplished by either (1) returning the figure generated by show and then manipulating it using the normal matplotlib functions, or (2) generating a matplotlib Figure with Axes any way you like (e.g. with `plt.subplots`) and then using this function to plot inside a single one of the Axes of your choice.

Option (1) is accomplished by simply passing this function `returnfig=True`. Thus:

```
>>> fig,ax = show(ar, returnfig=True)
```

will now give you direct access to the figure and axes to continue to alter. Option (2) is accomplished by passing an existing figure and axis to `show` as a 2-tuple to the `figax` argument. Thus:

```
>>> fig,(ax1,ax2) = plt.subplots(1,2)
>>> show(ar, figax=(fig,ax1))
>>> show(ar, figax=(fig,ax2), hist=True)
```

will generate a 2-axis figure, and then plot the array `ar` as an image on the left, while plotting its histogram on the right.

Parameters

- **ar** (*2D array or a list of 2D arrays*) – the data to plot. Normally this is a 2D array of the data. If a list of 2D arrays is passed, plots a corresponding grid of images.
- **figsize** (*2-tuple*) – size of the plot
- **cmap** (*colormap*) – any matplotlib cmap; default is gray
- **scaling** (*str*) – selects a scaling scheme for the intensity values. Default is none. Accepted values:
  - 'none': do not scale intensity values
  - 'full': fill entire color range with sorted intensity values
  - 'power': power law scaling

  - 'log': values where ar<=0 are set to 0

- **intensity_range** (*str*) –

  method for setting clipvalues (min and max intensities).
  The original name "clipvals" is now deprecated. Default is 'ordered'. Accepted values:

  - 'ordered': vmin/vmax are set to fractions of the distribution of pixel values in the array, e.g. vmin=0.02 will set the minumum display value to saturate the lower 2% of pixels
  - 'minmax': The vmin/vmax values are np.min(ar)/np.max(r)
  - 'absolute': The vmin/vmax values are set to the values of the vmin,vmax arguments received by this function
  - **'std': The vmin/vmax values are `np.median(ar) -/+ N*np.std(ar)`, and** N is this functions min,max vals.
  - 'centered': The vmin/vmax values are set to `c -/+ m`, where by default 'c' is zero and m is the max(abs(ar-c), or the two params can be user specified using the kwargs vmin/vmax -> c/m.

- **vmin** (*number*) – min intensity, behavior depends on clipvals
- **vmax** (*number*) – max intensity, behavior depends on clipvals
- **min** – alias' for vmin,vmax, throws deprecation warning
- **max** – alias' for vmin,vmax, throws deprecation warning
- **power** (*number*) – specifies the scaling power
- **power_offset** (*bool*) – If true, image has min value subtracted before power scaling
- **ticks** (*bool*) – Turn outer tick marks on or off
- **bordercolor** (*color or None*) – if not None, add a border of this color. The color can be anything matplotlib recognizes as a color.
- **borderwidth** (*number*) –
- **returnfig** (*bool*) – if True, the function returns the tuple (figure,axis)
- **figax** (*None or 2-tuple*) – controls which matplotlib Axes object draws the image. If None, generates a new figure with a single Axes instance. Otherwise, ax must be a 2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.
- **hist** (*bool*) – if True, instead of plotting a 2D image in ax, plots a histogram of the intensity values of ar, after any scaling this function has performed. Plots the clipvals as dashed vertical lines
- **n_bins** (*int*) – number of hist bins
- **mask** (*None or boolean array*) – if not None, must have the same shape as 'ar'. Wherever mask==True, plot the pixel normally, and where `mask==False`, pixel values are set to mask_color. If hist==True, ignore these values in the histogram. If `mask_alpha` is specified, the mask is blended with the array underneath, with 0 yielding an opaque mask and 1 yielding a fully transparent mask. If `mask_color` is set to `'empty'` instead of a

matplotlib.color, nothing is done to pixels where `mask==False`, allowing overlaying multiple arrays in different regions of an image by invoking the ``figax` kwarg over multiple calls to show

- **mask_color** (*color*) – see 'mask'
- **mask_alpha** (*float*) – see 'mask'
- **masked_intensity_range** (*bool*) – controls if masked pixel values are included when determining the display value range; False indicates that all pixel values will be used to determine the intensity range, True indicates only unmasked pixels will be used
- **scalebar** (*None or dict or Bool*) – if None, and a DiffractionSlice or RealSlice with calibrations is passed, adds a scalebar. If scalebar is not displaying the proper calibration, check .calibration pixel_size and pixel_units. If None and an array is passed, does not add a scalebar. If a dict is passed, it is propagated to the add_scalebar function which will attempt to use it to overlay a scalebar. If True, uses calibraiton or pixelsize/pixelunits for scalebar. If False, no scalebar is added.
- **show_fft** (*bool*) – if True, plots 2D-fft of array
- **apply_hanning_window** (*bool*) – If True, a 2D Hann window is applied to the array before applying the FFT
- **show_cbar** (*bool*) – if True, adds cbar
- ****kwargs** – any keywords accepted by matplotlib's ax.matshow()

Returns

if returnfig==False (default), the figure is plotted and nothing is returned. if returnfig==True, return the figure and the axis.

py4DSTEM.visualize.vis_special.**show_amorphous_ring_fit**(*dp*, *fitradii*, *p_dsg*, *N=12*, *cmap=('gray', 'gray')*, *fitborder=True*, *fitbordercolor='k'*, *fitborderlw=0.5*, *scaling='log'*, *ellipse=False*, *ellipse_color='r'*, *ellipse_alpha=0.7*, *ellipse_lw=2*, *returnfig=False*, ***kwargs*)

Display a diffraction pattern with a fit to its amorphous ring, interleaving the data and the fit in a pinwheel pattern.

Parameters

- **dp** (*array*) – the diffraction pattern
- **fitradii** (*2-tuple of numbers*) – the min/max distances of the fitting annulus
- **p_dsg** (*11-tuple*) – the fit parameters to the double-sided gaussian function returned by fit_ellipse_amorphous_ring
- **N** (*int*) – the number of pinwheel sections
- **cmap** (*colormap or 2-tuple of colormaps*) – if passed a single cmap, uses this colormap for both the data and the fit; if passed a 2-tuple of cmaps, uses the first for the data and the second for the fit
- **fitborder** (*bool*) – if True, plots a border line around the fit data
- **fitbordercolor** (*color*) – color of the fitborder
- **fitborderlw** (*number*) – linewidth of the fitborder
- **scaling** (*str*) – the normal scaling param – see docstring for visualize.show
- **ellipse** (*bool*) – if True, overlay an ellipse

- **returnfig** (*bool*) – if True, returns the figure

py4DSTEM.visualize.vis_special.**show_class_BPs**(*ar*, *x*, *y*, *s*, *s2*, *color='r'*, *color2='y'*, ***kwargs*)

words

py4DSTEM.visualize.vis_special.**show_class_BPs_grid**(*ar*, *H*, *W*, *x*, *y*, *get_s*, *s2*, *color='r'*, *color2='y'*, *returnfig=False*, *axsize=(6, 6)*, *titlesize=0*, *get_bordercolor=None*, ***kwargs*)

words

py4DSTEM.visualize.vis_special.**show_complex**(*ar_complex*, *vmin=None*, *vmax=None*, *power=None*, *chroma_boost=1*, *cbar=True*, *scalebar=False*, *pixelunits='pixels'*, *pixelsize=1*, *returnfig=False*, ***kwargs*)

Function to plot complex arrays

Parameters

- **ar_complex** (*2D array*) – complex array to be plotted. If ar_complex is list of complex arrarys such as [array1, array2], then arrays are horizonally plotted in one figure
- **vmin** (*float, optional*) – minimum absolute value
- **vmax** (*float, optional*) – maximum absolute value if None, vmin/vmax are set to fractions of the distribution of pixel values in the array, e.g. vmin=0.02 will set the minumum display value to saturate the lower 2% of pixels
- **power** (*float,optional*) – power to raise amplitude to
- **chroma_boost** (*float*) – boosts chroma for higher-contrast (~1-2.25)
- **cbar** (*bool, optional*) – if True, include color bar
- **scalebar** (*bool, optional*) – if True, adds scale bar
- **pixelunits** (*str, optional*) – units for scalebar
- **pixelsize** (*float, optional*) – size of one pixel in pixelunits for scalebar
- **returnfig** (*bool, optional*) – if True, the function returns the tuple (figure,axis)

Returns

if returnfig==False (default), the figure is plotted and nothing is returned. if returnfig==True, return the figure and the axis.

py4DSTEM.visualize.vis_special.**show_elliptical_fit**(*ar*, *fitradii*, *p_ellipse*, *fill=True*, *color_ann='y'*, *color_ell='r'*, *alpha_ann=0.2*, *alpha_ell=0.7*, *linewidth_ann=2*, *linewidth_ell=2*, *returnfig=False*, ***kwargs*)

Plots an elliptical curve over its annular fit region.

Parameters

- **center** (*2-tuple*) – the center
- **fitradii** (*2-tuple of numbers*) – the annulus inner and outer fit radii
- **p_ellipse** (*5-tuple*) – the parameters of the fit ellipse, (qx0,qy0,a,b,theta). See the module docstring for utils.elliptical_coords for more details.
- **fill** (*bool*) – if True, fills in the annular fitting region, else shows only inner/outer edges
- **color_ann** (*color*) – annulus color

- **color_ell** (*color*) – ellipse color
- **alpha_ann** – transparency for the annulus
- **alpha_ell** – transparency forn the fit ellipse
- **linewidth_ann** –
- **linewidth_ell** –

py4DSTEM.visualize.vis_special.**show_image_grid**(*get_ar*, *H*, *W*, *axsize=(6, 6)*, *returnfig=False*, *figax=None*, *title=None*, *title_index=False*, *suptitle=None*, *get_bordercolor=None*, *get_x=None*, *get_y=None*, *get_pointcolors=None*, *get_s=None*, *open_circles=False*, ***kwargs*)

Displays a set of images in a grid.

The images are specified by some function get_ar(i), which returns an image for values of some integer index i. The values of i passed to get_ar are 0 through HW-1.

To display the first 4 two-dimensional slices of some 3D array ar some 3D array ar, you can do

```
>>> show_image_grid(lambda i:ar[:,:,i], H=2, W=2)
```

Its also possible to add colored borders, or overlaid points, using similar functions to get_ar, i.e. functions which return the color or set of points of interest as a function of index i, which must be defined in the range [0,HW-1].

Accepts:

get_ar a function which returns a 2D array when passed
the integers 0 through HW-1

H,W integers, the dimensions of the grid axsize the size of each image figax controls which matplotlib Axes object draws the image.

If None, generates a new figure with a single Axes instance. Otherwise, ax must be a 2-tuple containing the matplotlib class instances (Figure,Axes), with ar then plotted in the specified Axes instance.

title if title is sting, then prints title as suptitle. If a suptitle is also provided,
the suptitle is printed insead. if title is a list of strings (ex: ['title 1','title 2']), each array has corresponding title in list.

title_index if True, prints the index i passed to get_ar over each image suptitle string, suptitle on plot get_bordercolor

if not None, should be a function defined over the same i as get_ar, and which returns a valid matplotlib color for each i. Adds a colored bounding box about each image. E.g. if *colors* is an array of colors:

```
>>> show_image_grid(lambda i:ar[:,:,i],H=2,W=2,
                    get_bordercolor=lambda i:colors[i])
```

get_x,get_y functions which returns sets of x/y positions
as a function of index i

get_s function which returns a set of point sizes
as a function of index i

get_pointcolors a function which returns a color or list of colors
as a function of index i

Returns

if returnfig==false (default), the figure is plotted and nothing is returned. if returnfig==false, the figure and its one axis are returned, and can be further edited.

py4DSTEM.visualize.vis_special.**show_kernel**(*kernel*, *R*, *L*, *W*, *figsize=(12, 6)*, *returnfig=False*, ***kwargs*)

Plots, side by side, the probe kernel and its line profile. R is the kernel plot's window size. L and W are the length and width of the lineprofile.

py4DSTEM.visualize.vis_special.**show_max_peak_spacing**(*ar*, *spacing*, *braggdirections*, *color='g'*, *lw=2*, *returnfig=False*, ***kwargs*)

Show a circle of radius *spacing* about each Bragg direction

py4DSTEM.visualize.vis_special.**show_origin_fit**(*data*, *plot_range=None*, *axsize=(3, 3)*)

Show the measured, fit, and residuals of the origin positions.

Parameters

- **data** (*(DataCube or Calibration or (3,2)-tuple of arrays)*) – ((qx0_meas,qy0_meas),(qx0_fit,qy0_fit),(qx0_residuals,qy0_residuals))
- **plot_range** (*(tuple, list, or np.array)*) – Plotting range in units of pixels
- **axsize** (*(tuple)*) – Size of each plot axis

py4DSTEM.visualize.vis_special.**show_origin_meas**(*data*)

Show the measured positions of the origin.

Parameters

data (*DataCube or Calibration or 2-tuple of arrays (qx0,qy0)*) –

py4DSTEM.visualize.vis_special.**show_pointlabels**(*ar*, *x*, *y*, *color='lightblue'*, *size=20*, *alpha=1*, *returnfig=False*, ***kwargs*)

Show enumerated index labels for a set of points

py4DSTEM.visualize.vis_special.**show_qprofile**(*q*, *intensity*, *ymax=None*, *figsize=(12, 4)*, *returnfig=False*, *color='k'*, *xlabel='q (pixels)'*, *ylabel='Intensity (A.U.)'*, *labelsize=16*, *ticklabelsize=14*, *grid=True*, *label=None*, ***kwargs*)

Plots a diffraction space radial profile. Params:

q (1D array) the diffraction coordinate / x-axis intensity (1D array) the y-axis values ymax (number) max value for the yaxis color (matplotlib color) profile color xlabel (str) ylabel labelsize size of x and y labels ticklabelsize grid True or False label a legend label for the plotted curve

py4DSTEM.visualize.vis_special.**show_selected_dps**(*datacube*, *positions*, *im*, *bragg_pos=None*, *colors=None*, *HW=None*, *figsize_im=(6, 6)*, *figsize_dp=(4, 4)*, ***kwargs*)

Shows two plots: first, a real space image overlaid with colored dots at the specified positions; second, a grid of diffraction patterns corresponding to these scan positions.

Parameters

- **datacube** (*DataCube*) –
- **positions** (*len N list or tuple of 2-tuples*) – the scan positions
- **im** (*2d array*) – a real space image
- **bragg_pos** (*len N list of pointlistarrays*) – bragg disk positions for each position. if passed, overlays the disk positions, and supresses plot of the real space image

- **colors** (*len N list of colors or None*) –
- **HW** (*2-tuple of ints*) – diffraction pattern grid shape
- **figsize_im** (*2-tuple*) – size of the image figure
- **figsize_dp** (*2-tuple*) – size of each diffraction pattern panel
- ****kwargs** (*dict*) – arguments passed to visualize.show for the *diffraction patterns*. Default is *scaling='log'*

py4DSTEM.visualize.vis_special.**show_strain**(*data*, *vrange=[-3, 3]*, *vrange_theta=[-3, 3]*, *vrange_exx=None*, *vrange_exy=None*, *vrange_eyy=None*, *show_cbars=None*, *bordercolor='k'*, *borderwidth=1*, *titlesize=18*, *ticklabelsize=10*, *ticknumber=5*, *unitlabelsize=16*, *cmap='RdBu_r'*, *cmap_theta='PRGn'*, *mask_color='k'*, *color_axes='k'*, *show_legend=False*, *rotation_deg=0*, *show_gvects=True*, *g1=None*, *g2=None*, *color_gvects='r'*, *legend_camera_length=1.6*, *scale_gvects=0.6*, *layout='square'*, *figsize=None*, *returnfig=False*, ***kwargs*)

Display a strain map, showing the 4 strain components (e_xx,e_yy,e_xy,theta), and masking each image with strainmap.get_slice('mask')

Parameters

- **data** (*strainmap*) –
- **vrange** (*length 2 list or tuple*) – The colorbar intensity range for exx,eyy, and exy.
- **vrange_theta** (*length 2 list or tuple*) – The colorbar intensity range for theta.
- **vrange_exx** (*length 2 list or tuple*) – The colorbar intensity range for exx; overrides *vrange* for exx
- **vrange_exy** (*length 2 list or tuple*) – The colorbar intensity range for exy; overrides *vrange* for exy
- **vrange_eyy** (*length 2 list or tuple*) – The colorbar intensity range for eyy; overrides *vrange* for eyy
- **show_cbars** (*None or a tuple of strings*) – Show colorbars for the specified axes. Valid strings are 'exx', 'eyy', 'exy', and 'theta'.
- **bordercolor** (*color*) – Color for the image borders
- **borderwidth** (*number*) – Width of the image borders
- **titlesize** (*number*) – Size of the image titles
- **ticklabelsize** (*number*) – Size of the colorbar ticks
- **ticknumber** (*number*) – Number of ticks on colorbars
- **unitlabelsize** (*number*) – Size of the units label on the colorbars
- **cmap** (*colormap*) – Colormap for exx, exy, and eyy
- **cmap_theta** (*colormap*) – Colormap for theta
- **mask_color** (*color*) – Color for the background mask
- **color_axes** (*color*) – Color for the legend coordinate axes
- **show_gvects** (*bool*) – Toggles displaying the g-vectors in the legend

- **rotation_deg** (*float*) – coordinate rotation for strainmap in degrees
- **g1** (*tuple*) – g1 orientation (x,y)
- **g2** (*tuple*) – g2 orientation (x,y)
- **color_gvects** (*color*) – Color for the legend g-vectors
- **legend_camera_length** (*number*) – The distance the legend is viewed from; a smaller number yields a larger legend
- **scale_gvects** (*number*) – Scaling for the legend g-vectors relative to the coordinate axes
- **layout** (*int*) – Determines the layout of the grid which the strain components will be plotted in. Options are "square", "horizontal", "vertical."
- **figsize** (*length 2 tuple of numbers*) – Size of the figure
- **returnfig** (*bool*) – Toggles returning the figure
- ****kwargs** (*keywords passed to py4DSTEM show function*) –

py4DSTEM.visualize.vis_special.**show_voronoi**(*ar*, *x*, *y*, *color_points='r'*, *color_lines='w'*, *max_dist=None*, *returnfig=False*, ***kwargs*)

words

1.4.7 emd

Table of Contents

- *emd*
  - *Classes*
  - *Functions*

Classes

class emdfile.**Array**(*data: ndarray*, *name: str | None = 'array'*, *units: str | None = ''*, *dims: list | None = None*, *dim_names: list | None = None*, *dim_units: list | None = None*, *slicelabels=None*)

A class which stores any N-dimensional array-like data, plus basic metadata: a name and units, as well as calibrations for each axis of the array, and names and units for those axis calibrations.

In the simplest usage, only a data array is passed:

```
>>> ar = Array(np.ones((20,20,256,256)))
```

will create an array instance whose data is the numpy array passed, and with automatically populated dimension calibrations in units of pixels.

Additional arguments may be passed to populate the object metadata:

```
>>> ar = Array(
>>>     np.ones((20,20,256,256)),
>>>     name = 'test_array',
>>>     units = 'intensity',
```

(continues on next page)

(continued from previous page)

```
>>>     dims = [
>>>         [0,5],
>>>         [0,5],
>>>         [0,0.01],
>>>         [0,0.01]
>>>     ],
>>>     dim_units = [
>>>         'nm',
>>>         'nm',
>>>         'A^-1',
>>>         'A^-1'
>>>     ],
>>>     dim_names = [
>>>         'rx',
>>>         'ry',
>>>         'qx',
>>>         'qy'
>>>     ],
>>> )
```

will create an array with a name and units for its data, where its first two dimensions are in units of nanometers, have pixel sizes of 5nm, and are described by the handles 'rx' and 'ry', and where its last two dimensions are in units of inverse Angstroms, have pixels sizes of 0.01A^-1, and are described by the handles 'qx' and 'qy'.

Arrays in which the length of each pixel is non-constant are also supported. For instance,

```
>>> x = np.logspace(0,1,100)
>>> y = np.sin(x)
>>> ar = Array(
>>>     y,
>>>     dims = [
>>>         x
>>>     ]
>>> )
```

generates an array representing the values of the sine function sampled 100 times along a logarithmic interval from 1 to 10. In this example, this data could then be plotted with, e.g.

```
>>> plt.scatter(ar.dims[0], ar.data)
```

If the *slicelabels* keyword is passed, the first N-1 dimensions of the array are treated normally, while the final dimension is used to represent distinct arrays which share a common shape and set of dim vectors. Thus

```
>>> ar = Array(
>>>     np.ones((50,50,4)),
>>>     name = 'test_array_stack',
>>>     units = 'intensity',
>>>     dims = [
>>>         [0,2],
>>>         [0,2]
>>>     ],
>>>     dim_units = [
>>>         'nm',
```

(continues on next page)

(continued from previous page)

```
>>>         'nm'
>>>     ],
>>>     dim_names = [
>>>         'rx',
>>>         'ry'
>>>     ],
>>>     slicelabels = [
>>>         'a',
>>>         'b',
>>>         'c',
>>>         'd'
>>>     ]
>>> )
```

will generate a single Array instance containing 4 arrays which each have a shape (50,50) and a common set of dim vectors ['rx','ry'], and which can be indexed into with the names assigned in *slicelabels* using

```
>>> ar.get_slice('a')
```

which will return a 2D (non-stack-like) Array instance with shape (50,50) and the dims assigned above. The Array attribute .rank is equal to the number of dimensions for a non-stack-like Array, and is equal to N-1 for stack-like arrays.

__init__(*data: ndarray*, *name: str | None = 'array'*, *units: str | None = ''*, *dims: list | None = None*, *dim_names: list | None = None*, *dim_units: list | None = None*, *slicelabels=None*)

Accepts:

data (np.ndarray): the data name (str): the name of the Array units (str): units for the pixel values dims (variable): calibration vectors for each of the axes of the data

array. Valid values for each element of the list are None, a number, a 2-element list/array, or an M-element list/array where M is the data array. If None is passed, the dim will be populated with integer values starting at 0 and its units will be set to pixels. If a number is passed, the dim is populated with a vector beginning at zero and increasing linearly by this step size. If a 2-element list/array is passed, the dim is populated with a linear vector with these two numbers as the first two elements. If a list/array of length M is passed, this is used as the dim vector, (and must therefore match this dimension's length). If dims recieves a list of fewer than N arguments for an N-dimensional data array, the extra dimensions are populated as if None were passed, using integer pixel values. If the *dims* parameter is not passed, all dim vectors are populated this way.

dim_units (list): the units for the calibration dim vectors. If

nothing is passed, dims vectors which have been populated automatically with integers corresponding to pixel numbers will be assigned units of 'pixels', and any other dim vectors will be assigned units of 'unknown'. If a list with length < the array dimensions, the passed values are assumed to apply to the first N dimensions, and the remaining values are populated with 'pixels' or 'unknown' as above.

dim_names (list): labels for each axis of the data array. Values

which are not passed, following the same logic as described above, will be autopopulated with the name "dim#" where # is the axis number.

slicelabels (None or True or list): if not None, must be True or a

list of strings, indicating a "stack-like" array. In this case, the first N-1 dimensions of the array are treated normally, in the sense of populating dims, dim_names, and dim_units, while the final

dimension is treated distinctly: it indexes into distinct arrays which share a set of dimension attributes, and can be sliced into using the string labels from the *slicelabels* list, with the syntax array['label'] or array.get_slice('label'). If *slicelabels* is *True* or is a list with length less than the final dimension length, unassigned dimensions are autopopulated with labels *array{i}*. The flag array.is_stack is set to True and the array.rank attribute is set to N-1.

Returns
A new Array instance

`get_dim`(*n*)

Return the n'th dim vector

`dim`(*n*)

Return the n'th dim vector

`set_dim`(*n: int*, *dim: list | ndarray*, *units: str | None = None*, *name: str | None = None*)

Sets the n'th dim vector, using *dim* as described in the Array documentation. If *units* and/or *name* are passed, sets these values for the n'th dim vector.

Accepts:
n (int): specifies which dim vector dim (list or array): length must be either 2, or equal to the

length of the n'th axis of the data array

units (Optional, str): name: (Optional, str):

`get_dim_units`(*n*)

Return the n'th dim vector units

`set_dim_units`(*n: int*, *units: str*)

Sets the n'th dim vector units to *units*.

Accepts:
n (int): specifies which dim vector units (str): new units

`get_dim_name`(*n*)

Get the n'th dim vector name

`set_dim_name`(*n: int*, *name: str*)

Sets the n'th dim vector name to *name*.

Accepts:
n (int): specifies which dim vector name (str): new name

`to_h5`(*group*)

Takes an h5py Group instance and creates a subgroup containing this Array, tags indicating its EMD type and Python class, and the array's data and metadata.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new array's Group

`class emdfile.Custom`(*name='custom'*)

`__init__`(*name='custom'*)

to_h5(*group*)

Constructs an h5 group, adds metadata, and adds all attributes which point to EMD nodes.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new node's Group

class emdfile.**Metadata**(*name: str | None = 'metadata'*, *data: dict | None = None*)

Stores metadata in the form of a flat (non-nested) dictionary. Keys are arbitrary strings. Values may be strings, numbers, arrays, or lists of the above types.

Usage:

```
>>> meta = Metadata()
>>> meta['param'] = value
>>> val = meta['param']
```

If the parameter has not been set, the getter methods return None.

__init__(*name: str | None = 'metadata'*, *data: dict | None = None*)

Parameters
name (*Optional, string*) –

copy(*name=None*)

to_h5(*group*)

Accepts an h5py Group which is open in write or append mode. Writes a new group with this object's name and saves its metadata in it.

Accepts:
group (h5py Group)

classmethod from_h5(*group*)

Accepts an h5py Group which is open in read mode, confirms that it represents an EMD MetadataDict group, then loads and returns it as a Metadata instance.

Accepts:
group (HDF5 group)

Returns
(Metadata)

class emdfile.**Node**(*name: str | None = 'node'*)

Nodes contain attributes and methods paralleling the EMD 1.0 file specification in Python runtime objects.

EMD 1.0 is a singly-rooted file format. That is to say: An EMD data object can and must exist in one and only one EMD tree. An EMD file can contain any number of EMD trees, each containing data and metadata which is, within the limits of the EMD group specifications, of some arbitrary complexity. An EMD 1.0 file thus represents, stores, and enables access to some arbitrary data in long term storage on a file system in the form of an HDF5 file. The Node class provides machinery for building trees of data and metadata which mirror the EMD tree format but which exist in a live Python instance, rather than on the file system. This facilitates ease of transfer between Python and the file system.

Nodes are intended to be used a base class on which other, more complex classes can be biult. Nodes themselves contain the machinery for managing a tree heirarchy of other Nodes and Metadata instances, and for reading and

writing those trees. They do not contain any particular data. Classes storing data and analysis methods which inherit from Node will inherit its tree management and EMD i/o functionality.

Below, the 4 elements of the node class are each described in turn: roots, trees, metadata, and i/o.

ROOTS

EMD data objects can and must exist in one and only one EMD tree, each of which must have a single, named root node. To parallel this in our runtime objects, each Node has a root property, which can be found by calling *self.root*.

By default new nodes have their root set to None. If a node with *.root == None* is saved to file, it is placed inside a new root with the same name as the object itself, and this is then saved to the file as a new (minimal) EMD tree.

A new root node can be instantiated by calling

```
>>> rootnode = Root(name=some_name).
```

Objects added to an existing rooted tree (including a new root node) automatically have their root assigned to the root of that tree. Adding objects to trees is discussed below.

TREES

The tree associated with a node can be manipulated with the .tree method. If we have some rooted node *node1* and some unrooted node *node2*, the unrooted node can be added to the existing tree as a child of the rooted node with

```
>>> node1.tree(node2)
```

If we have a rooted node *node1* and another rooted node *node2*, we can't simply add node2 with the code above, as this would create a conflict between the two roots. In this case, we can move node2 from its current tree to the new tree using

```
>>> node1.tree(graft=node2)
```

The .tree method has various additional functionalities, including printing the tree, retrieving objects from the tree, and cutting branches from the tree. These are summarized below:

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keep root metadata
>>> .tree(graft=(node,True))     # as above
>>> .tree(graft=(node,False))    # as above, discard root metadata
>>> .tree(graft=(node,'copy'))   # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string), i.e. in most cases, the keyword can be dropped. So

```
>>> .tree()
>>> .tree(node)
>>> .tree(True)
>>> .tree('some/node')
```

will, respectively, print the tree from the current node to screen, add the node *node* to the tree, pring the tree from the root node to screen, and return the node at the emdpath 'some/node'.

If a node needs to be added to a tree and it may or may not already have its own root, calling

```
>>> .tree(add=node, force=True)
```

or

```
>>> .tree(node, force=True)
```

will add the node to the tree, using a simple add if node has no root, and grafting it if it does have a root.

METADATA

Nodes can contain any number of Metadata instances, each of which wraps a Python dictionary of some arbitrary complexity (to within the limits of the Metadata group EMD specification, which limits permissible values somewhat).

The code:

```
>>> md1 = Metadata(name='md1')
>>> md2 = Metadata(name='md2')
>>> <<<  some code populating md1 + md2 >>>
>>> node.metadata = md1
>>> node.metadata = md2
```

will create two Metadata objects, populate them with data, then add them to the node. Note that Node.metadata is *not* a Python attribute, it is specially defined property, such that the last line of code does not overwrite the line before it - rather, assigning to the .metadata property adds the new metadata object to a running dictionary of arbitrarily many metadata objects. Both of these two metadata instances can therefore still be retrieved, using:

```
>>> x = node.metadata['md1']
>>> y = node.metadata['md2']
```

Note, however, that if the second metadata instance has an identical name to the first instance, then in *will* overwrite the old instance.

I/O

TODO

__init__(*name: str | None = 'node'*)

show_tree(*root=False*)

Display the object tree. If *root* is False, displays the branch of the tree downstream from this node. If *root* is True, displays the full tree from the root node.

add_to_tree(*node*)

Add an unrooted node as a child of the current, rooted node. To move an already rooted node/branch, use *.graft()*. To create a rooted node, use *Root()*.

force_add_to_tree(*node*)

Add node *node* as a child of the current node, whether or not *node* is rooted. If it's unrooted, performs a simple add. If it is rooted, performs a graft, excluding the root metadata from *node*.

get_from_tree(*name*)

Finds and returns an object from an EMD tree using the string key *name*, with '/' delimiters between 'parent/child' nodes. Search from the root node by adding a leading '/'; otherwise, searches from the current node.

graft(*node*, *merge_metadata=True*)

Moves the branch beginning node onto this tree at this node.

For the reverse (i.e. grafting from this tree onto another tree) either use that tree's .graft method, or use this tree's ._graft.

Accepts:

node (Node): merge_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) this tree's root node

cut_from_tree(*root_metadata=True*)

Removes a branch from an object tree at this node.

A new root node is created under this object with this object's name. Metadata from the current root is transferred/not transferred to the new root according to the value of *root_metadata*.

Accepts:

root_metadata (True, False, or 'copy'): if True adds the old root's

metadata to the new root; if False adds no metadata to the new root; if 'copy' adds copies of all metadata from the old root to the new root.

Returns

(Node) the new root node

tree(*arg=None*, ***kwargs*)

Usages -

```
>>> .tree()              # show tree from current node
>>> .tree(show=True)     # show from root
>>> .tree(show=False)    # show from current node
>>> .tree(add=node)      # add a child node
>>> .tree(get='path')    # return a '/' delimited child node
>>> .tree(get='/path')   # as above, starting at root
>>> .tree(cut=True)      # remove/return a branch, keep root metadata
>>> .tree(cut=False)     # remove/return a branch, discard root md
>>> .tree(cut='copy')    # remove/return a branch, copy root metadata
>>> .tree(graft=node)    # remove/graft a branch, keeping root metadata
>>> .tree(graft=(node,True))    # as above
>>> .tree(graft=(node,False))   # as above, discard root metadata
>>> .tree(graft=(node,'copy'))  # as above, copy root metadata
```

The show, add, and get methods can be accessed directly with

```
>>> .tree(arg)
```

for an arg of the appropriate type (bool, Node, and string).

static newnode(*method*)

Decorator which may be added to node methods which product and return a new node. If such a method is decorated with

```
>>> @newnode
```

then the new node is added to the parent node's tree, and a Metadata instance is added to the new node's metadata which stores information about how the node was created, namely: method's name, the parent's class and name, and all the arguments passed to method.

classmethod from_h5(*group*)

Takes an h5py Group which is open in read mode. Confirms that a a Node of this name exists in this group, and loads and returns it with it's metadata.

Accepts:
group (h5py Group)

Returns
(Node)

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this node, tags indicating the groups EMD type and Python class, and any metadata in this node.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new node's Group

class emdfile.**PointList**(*data: ndarray*, *name: str | None = 'pointlist'*)

A wrapper around structured numpy arrays, with read/write functionality in/out of EMD formatted HDF5 files.

__init__(*data: ndarray*, *name: str | None = 'pointlist'*)

Instantiate a PointList.

Parameters

- **data** (*structured numpy ndarray*) – the data; the dtype of this array will specify the fields of the PointList.
- **name** (*str*) – name for the PointList

Returns
a PointList instance

add(*data*)

Appends a numpy structured array. Its dtypes must agree with the existing data.

remove(*mask*)

Removes points wherever mask==True

sort(*field*, *order='ascending'*)

Sorts the point list according to field, which must be a field in self.dtype. order should be 'descending' or 'ascending'.

copy(*name=None*)

Returns a copy of the PointList. If name=None, sets to *{name}_copy*

add_fields(*new_fields*, *name=''*)

Creates a copy of the PointList, but with additional fields given by new_fields.

Parameters

- **new_fields** – a list of 2-tuples, ('name', dtype)
- **name** – a name for the new pointlist

add_data_by_field(*data*, *fields=None*)

Add a list of data arrays to the PointList, in the fields given by *fields*. If *fields* is not specified, assumes the data arrays are in the same order as self.fields

Parameters

data (*list*) – arrays of data to add to each field

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this PointList, tags indicating its EMD type and Python class, and the pointlist's data and metadata.

Accepts:

group (h5py Group)

Returns

(h5py Group) the new pointlist's group

class emdfile.**PointListArray**(*dtype*, *shape*, *name: str | None = 'pointlistarray'*)

An 2D array of PointLists which share common coordinates.

__init__(*dtype*, *shape*, *name: str | None = 'pointlistarray'*)

Creates an empty PointListArray.

Parameters

- **dtype** – the dtype of the numpy structured arrays which will comprise the data of each PointList
- **shape** (*2-tuple of ints*) – the shape of the array of PointLists
- **name** (*str*) – a name for the PointListArray

Returns

a PointListArray instance

get_pointlist(*i*, *j*, *name=None*)

Returns the pointlist at i,j

copy(*name=''*)

Returns a copy of itself.

add_fields(*new_fields*, *name=''*)

Creates a copy of the PointListArray, but with additional fields given by new_fields.

Parameters

- **new_fields** – a list of 2-tuples, ('name', dtype)
- **name** – a name for the new pointlist

to_h5(*group*)

Takes an h5py Group instance and creates a subgroup containing this PointListArray, tags indicating its EMD type and Python class, and the pointlistarray's data and metadata.

Accepts:
group (h5py Group)

Returns
(h5py Group) the new pointlistarray's group

class emdfile.**Root**(*name='root'*)

A Node instance with its .root property set to itself.

__init__(*name='root'*)

Functions

emdfile.**_get_EMD_version**(*filepath*, *rootgroup=None*)

Returns the version (major,minor,release) of an EMD file.

emdfile.**_is_EMD_file**(*filepath*)

Returns True iff filepath points to a valid EMD 1.0 file.

emdfile.**_version_is_geq**(*current*, *minimum*)

Returns True iff current version (major,minor,release) is greater than or equal to minimum."

emdfile.**dirname**(*p*)

Returns the directory component of a pathname

emdfile.**join**(*a*, **p*)

Join two or more pathname components, inserting '/' as needed. If any component is an absolute path, all previous path components will be discarded. An empty last part will result in a path that ends with a separator.

emdfile.**print_h5_tree**(*filepath*, *show_metadata=False*)

Prints the contents of an h5 file from a filepath.

emdfile.**read**(*filepath*, *emdpath: str | None = None*, *tree: bool | str | None = True*, ***legacy_options*)

File reader for EMD 1.0+ files.

Parameters

- **filepath** (*str or Path*) – the file path
- **emdpath** (*str*) – path to the node in an EMD object tree to read from. May be a root node or some downstream node. Use '/' delimiters between node names. If emdpath is None, checks to see how many root nodes are present. If there is one, loads this tree. If there are several, returns a list of the root names.

- **tree** (*True or False or 'branch'*) – indicates what data should be loaded, relative to the node specified by *emdpath*. If set to *False*, only data/metadata in the specified node is loaded, plus any root metadata. If set to *True*, loads that node plus the subtree of data objects it contains (and their metadata, and the root metadata). If set to *'branch'*, loads the branch under this node as above, but does not load the node itself. If *emdpath* points to a root node, setting *tree* to *'branch'* or *True* are equivalent - both return the whole data tree.

Returns

(Root) returns a Root instance containing (1) any root metadata from
the EMD tree loaded from, and (2) a tree of one or more pieces of data/metadata

emdfile.**save**(*filepath*, *data*, *mode='w'*, *emdpath=None*, *tree=True*)

Saves data to a .h5 file at filepath.

Calling

```
>>> save(path, data)
```

if *data* is a Root instance saves this root and its entire tree to a new file. If *data* is any other type of rooted node (i.e. a node inside of some runtime data tree), this code writes a new file with a single tree using this node's root (even if this node is far downstream of the root node), placing this node and the tree branch underneath it inside that root. In both cases, the root metadata is stored in the new H5 root node. If *data* is an unrooted node (i.e. a freestanding node not connected to a tree), this code creates a new root node with no metadata and this node's name, and places this node inside that root in a new file.

If *data* is a numpy array or Python dictionary, wraps data in either an emd.Array or emd.Metadata instance, assigns the name 'np.array' or 'dictionary', places the object in a root of this name and saves. If *data* is a list of objects which are all numpy arrays, Python dictionaries, or emd.Node instances, places all these objects into a single root, assigns the roots name according to the first object in the list, and saves.

To write a single node from a tree, set *tree* to False. To write the tree underneath a node but exclude the node itself set *tree* to None.

To add to an existing EMD file, use the *mode* argument to set append or appendover mode. If the *emdpath* variable is not set and *data* has a runtime root that does not exist in the EMD root groups already present, adds the new root and writes as described above. If *emdpath* is not set and the runtime root group matches a root group that's already present, this function performs a diff operation between the root metadata and data nodes from *data* and those already in the H5 file. Append mode adds any data/metadata groups with no equivalent (i.e. same name and tree location) in the H5 tree, while skipping any data/metadata already found in the tree. Appendover adds any data/metadata with no equivalent already in the H5 tree, and overwrites any data/metadata groups that are already represented in the HDF5 with the new data. Note that this function does not attempt to take a diff between the contents of the groups and the runtime data groups - it only considers the names and their locations in the tree. If append or appendover mode are used and filepath is set to a location that does not already contain a file on the filesystem, behavior is identical to write mode. When appendover mode overwrites data, it is erasing the old links and creating new links to new data; however, the HDF5 file does not release the space on the filesystem. To free up storage, set mode to 'appendover', and this function will add a final step to re-write then delete the old file.

The *emdpath* argument is used to append to a specific location in an extant EMD file downstream of some extant root. If passed, it must point to a valid location in the EMD file. This function will then perform a diff and write as described in the prior paragraph, except beginning from the H5 node specified in *emdpath*. Note that in this case the root metadata is still compared to and added or overwritten in the H5 root node, even if the remaining data is being added to some downstream branch.

Parameters

- **filepath** – path where the file will be saved

- **data** – an EMD data class instance
- **mode** (*str*) –

  supported modes and their keys are:

  - write ('w','write')
  - overwrite ('o','overwrite')
  - append ('a','+','append')
  - appendover ('ao','oa','o+','+o','appendover')

  Write mode writes a new file, and raises an exception if a file of this name already exists. Overwrite mode deletes any file of this name that already exists and writes a new file. Append and appendover mode write a new file if no file of this name exists, or if a file of this name does exist, adds new data to the file. The specific behavior of append and appendover depend on the *data*,`emdpath`, and *tree* arguments as discussed in more detail above. Broadly, both modes attempt to detemine the difference between the data passed and that present in the extent HDF5 file tree, add any data not already in the H5, and then either skips or overwrites conflicting nodes in append or appendover mode, respectively.
- **tree** – indicates how the object tree nested inside *data* should be treated. If *True* (default), the entire tree is saved. If *False*, only this object is saved, without its tree. If *None*, saves the entire tree underneath *data*, but not the node at *data* itself.
- **emdpath** (*str or None*) – optional parameter used in conjunction with append or appendover mode; if passed in write or overwrite mode, this argument is ignored. Indicates where in an existing EMD file tree to place the data. Must be a '/' delimited string pointing to an existing EMD file tree node.

emdfile.**set_author**(*author*)

Accepts a string, which will be written to the "authoring_user" field in any EMD file headers written during this Python session

emdfile.**tqdmnd**(**args*, ***kwargs*)

An N-dimensional extension of tqdm providing an iterator and progress bar over the product of multiple iterators.

Example Usage:

```
>>> for x,y in tqdmnd(5,6):
>>>     <expression>
```

is equivalent to

```
>>> for x in range(5):
>>>     for y in range(6):
>>>         <expression>
```

with a tqdmnd-style progress bar printed to standard output.

Accepts:

***args: Any number of integers or iterators. Each integer N**
is converted to a *range(N)* iterator. Then a loop is constructed from the Cartesian product of all iterables.

****kwargs: keyword arguments passed through directly to tqdm.**
Full details are available at https://tqdm.github.io A few useful ones:

disable (bool): if True, hide the progress bar keep (bool): if True, delete the progress bar after completion unit (str): unit name for the display of iteration speed unit_scale (bool): whether to scale the displayed units and add

SI prefixes

desc (str): message displayed in front of the progress bar

Returns

At each iteration, a tuple of indices is returned, corresponding to the values of each input iterator (in the same order as the inputs).

1.5 API Index

1.6 Graphical User Interface

1.6.1 Overview

There is a GUI for viewing and performing some basic analysis of your 4D-STEM dataset. This feature is currently in development and must be installed separately. For more details you can checkout the git repositoary here

1.6.2 Installation

Currently there are no pip or conda packages and it must be install in one of two ways:

```
git clone https://github.com/sezelt/py4D-browser.git
cd py4D-browser
python setupy.py
```

Alternatively,

```
pip install git+https://github.com/sezelt/py4D-browser
```

1.7 Support & Contributions

1.7.1 Support

Think you've found a bug or are facing issues using a feature? Please let us know by creating an issue on github

1.7.2 Contributions

Looking to contirbute? Awesome we love people contributing, and it's a simple process.

1. Submit feature request on github
2. Follow the developer install instructions
3. Make any change alterations and document all functions (All code should be readable, so clarity beats cleverness)
4. Submit a PR on github.

1.8 License

py4DSTEM is released under the GNU GPV version 3 license.

1.8.1 GPLv3

(continues on next page)

(continued from previous page)

```
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
```

(continues on next page)

(continued from previous page)

```
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
```

(continues on next page)

(continued from previous page)

```
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
```

(continues on next page)

(continued from previous page)

```
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
```

(continues on next page)

(continued from previous page)

```
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
```

(continues on next page)

(continued from previous page)

```
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
```

(continues on next page)

(continued from previous page)

```
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
```

(continues on next page)

(continued from previous page)

```
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
```

(continues on next page)

(continued from previous page)

```
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
```

(continues on next page)

(continued from previous page)

```
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
```

(continues on next page)

(continued from previous page)

```
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
```

(continues on next page)

(continued from previous page)

```
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.
```

(continues on next page)

(continued from previous page)

```
The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.
```

1.9 Acknowledgements

- If you use py4DSTEM for a scientific study, please cite our open access py4DSTEM publication[1] in Microscopy and Microanalysis.
  - py4DSTEM: A Software Package for Four-Dimensional Scanning Transmission Electron Microscopy Data Analysis
- Check out the Py4DSTEM Github[2]
- We'd like to thank The developers gratefully acknowledge the financial support of the Toyota Research Institute for the research and development time which made this project possible.

TOYOTA RESEARCH INSTITUTE

- Additional funding has been provided by the US Department of Energy, Office of Science, Basic Energy Sciences.

U.S. DEPARTMENT OF ENERGY | Office of Science

- You are also free to use the py4DSTEM logo in PDF format or logo in PNG format for presentations or posters.

[1] https://doi.org/10.1017/S1431927621000477
[2] http://github.com/py4DSTEM/py4DSTEM

1.9.1 References

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

py4DSTEM.io, 86
py4DSTEM.io.filereaders, 86
py4DSTEM.io.filereaders.empad, 86
py4DSTEM.io.filereaders.read_K2, 87
py4DSTEM.io.filereaders.read_mib, 88
py4DSTEM.io.google_drive_downloader, 90
py4DSTEM.io.google_drive_downloader.gdown, 90
py4DSTEM.io.importfile, 90
py4DSTEM.io.legacy, 91
py4DSTEM.io.legacy.h5py, 91
py4DSTEM.io.legacy.legacy12, 91
py4DSTEM.io.legacy.legacy13, 91
py4DSTEM.io.legacy.read_legacy_12, 91
py4DSTEM.io.legacy.read_legacy_13, 91
py4DSTEM.io.legacy.read_utils, 92
py4DSTEM.io.parsefiletype, 92
py4DSTEM.preprocess.darkreference, 93
py4DSTEM.preprocess.electroncount, 94
py4DSTEM.preprocess.preprocess, 96
py4DSTEM.preprocess.radialbkgrd, 99
py4DSTEM.preprocess.utils, 100
py4DSTEM.process, 102
py4DSTEM.process.calibration, 102
py4DSTEM.process.calibration.ellipse, 102
py4DSTEM.process.calibration.origin, 105
py4DSTEM.process.calibration.probe, 107
py4DSTEM.process.calibration.qpixelsize, 108
py4DSTEM.process.calibration.rotation, 108
py4DSTEM.process.classification, 110
py4DSTEM.process.classification.braggvectorclassification, 110
py4DSTEM.process.classification.classutils, 118
py4DSTEM.process.classification.featurization, 119
py4DSTEM.process.diffraction, 125
py4DSTEM.process.diffraction.crystal, 125
py4DSTEM.process.diffraction.crystal_ACOM, 147
py4DSTEM.process.diffraction.crystal_bloch, 152
py4DSTEM.process.diffraction.crystal_calibrate, 157
py4DSTEM.process.diffraction.crystal_phase, 159
py4DSTEM.process.diffraction.crystal_viz, 160
py4DSTEM.process.diffraction.flowlines, 168
py4DSTEM.process.diffraction.sys, 173
py4DSTEM.process.diffraction.utils, 174
py4DSTEM.process.diffraction.WK_scattering_factors, 125
py4DSTEM.process.fit, 175
py4DSTEM.process.fit.fit, 175
py4DSTEM.process.phase, 176
py4DSTEM.process.phase.utils, 176
py4DSTEM.process.rdf.amorph, 187
py4DSTEM.process.rdf.rdf, 188
py4DSTEM.process.utils, 189
py4DSTEM.process.utils.cross_correlate, 189
py4DSTEM.process.utils.elliptical_coords, 190
py4DSTEM.process.utils.masks, 194
py4DSTEM.process.utils.multicorr, 194
py4DSTEM.process.utils.utils, 196
py4DSTEM.process.wholepatternfit, 198
py4DSTEM.process.wholepatternfit.wp_models, 198
py4DSTEM.process.wholepatternfit.wpf, 204
py4DSTEM.process.wholepatternfit.wpf_viz, 204

INDEX

Symbols

__init__() (*emdfile.Array method*), 242
__init__() (*emdfile.Custom method*), 243
__init__() (*emdfile.Metadata method*), 244
__init__() (*emdfile.Node method*), 246
__init__() (*emdfile.PointList method*), 248
__init__() (*emdfile.PointListArray method*), 249
__init__() (*emdfile.Root method*), 250
__init__() (*py4DSTEM.Array method*), 25
__init__() (*py4DSTEM.BraggVectors method*), 29
__init__() (*py4DSTEM.Calibration method*), 39
__init__() (*py4DSTEM.Custom method*), 40
__init__() (*py4DSTEM.Data method*), 43
__init__() (*py4DSTEM.DataCube method*), 43
__init__() (*py4DSTEM.DiffractionSlice method*), 57
__init__() (*py4DSTEM.Metadata method*), 60
__init__() (*py4DSTEM.Node method*), 63
__init__() (*py4DSTEM.PointList method*), 65
__init__() (*py4DSTEM.PointListArray method*), 67
__init__() (*py4DSTEM.Probe method*), 70
__init__() (*py4DSTEM.QPoints method*), 75
__init__() (*py4DSTEM.RealSlice method*), 78
__init__() (*py4DSTEM.VirtualDiffraction method*), 81
__init__() (*py4DSTEM.VirtualImage method*), 83
__init__() (*py4DSTEM.io.filereaders.read_K2.K2DataArray method*), 88
__init__() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 112
__init__() (*py4DSTEM.process.classification.featurization.Featurization method*), 120
__init__() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 140
__init__() (*py4DSTEM.process.diffraction.crystal_bloch.DynamicMatrixCache method*), 152
__init__() (*py4DSTEM.process.diffraction.crystal_phase.Crystal_Phase method*), 159
__init__() (*py4DSTEM.process.diffraction.utils.Orientation method*), 174
__init__() (*py4DSTEM.process.diffraction.utils.OrientationMap method*), 174
__init__() (*py4DSTEM.process.phase.utils.AffineTransform method*), 179
__init__() (*py4DSTEM.process.phase.utils.ComplexProbe method*), 177
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.ComplexOve method*), 203
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.DCBackgrou method*), 199
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.GaussianBac method*), 200
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.GaussianRin method*), 200
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.KernelDiskL method*), 204
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.SyntheticDis method*), 202
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.SyntheticDis method*), 203
__init__() (*py4DSTEM.process.wholepatternfit.wp_models.WPFModel method*), 199
_get_EMD_version() (*in module emdfile*), 250
_is_EMD_file() (*in module emdfile*), 250
_show_grid_overlay() (*in module py4DSTEM.visualize.vis_grid*), 222
_version_is_geq() (*in module emdfile*), 250

A

accept() (*py4DSTEM.process.classification.braggvectorclassification.Bra method*), 115
add() (*emdfile.PointList method*), 248
add() (*py4DSTEM.DataCube method*), 43
add() (*py4DSTEM.PointList method*), 65
add() (*py4DSTEM.QPoints method*), 76
add_annuli() (*in module py4DSTEM.visualize.overlay*), 212
add_bragg_index_labels() (*in module py4DSTEM.visualize.overlay*), 212
add_bragg_index_labels() (*in module py4DSTEM.visualize.vis_special*), 229
add_cartesian_grid() (*in module py4DSTEM.visualize.overlay*), 212
add_circles() (*in module py4DSTEM.visualize.overlay*), 212
add_data_by_field() (*emdfile.PointList method*), 249

add_data_by_field() (*py4DSTEM.PointList method*), 65
add_data_by_field() (*py4DSTEM.QPoints method*), 76
add_ellipses() (*in module py4DSTEM.visualize.overlay*), 212
add_ellipses() (*in module py4DSTEM.visualize.vis_special*), 229
add_features() (*py4DSTEM.process.classification.featurization_funcs method*), 121
add_fields() (*emdfile.PointList method*), 249
add_fields() (*emdfile.PointListArray method*), 249
add_fields() (*py4DSTEM.PointList method*), 65
add_fields() (*py4DSTEM.PointListArray method*), 68
add_fields() (*py4DSTEM.QPoints method*), 76
add_grid_overlay() (*in module py4DSTEM.visualize.overlay*), 212
add_grid_overlay() (*in module py4DSTEM.visualize.vis_grid*), 222
add_pointlabels() (*in module py4DSTEM.visualize.overlay*), 213
add_pointlabels() (*in module py4DSTEM.visualize.vis_special*), 229
add_points() (*in module py4DSTEM.visualize.overlay*), 213
add_points() (*in module py4DSTEM.visualize.vis_special*), 229
add_polarelliptical_grid() (*in module py4DSTEM.visualize.overlay*), 213
add_rectangles() (*in module py4DSTEM.visualize.overlay*), 213
add_rtheta_grid() (*in module py4DSTEM.visualize.overlay*), 213
add_scalebar() (*in module py4DSTEM.visualize.overlay*), 213
add_scalebar() (*in module py4DSTEM.visualize.vis_special*), 229
add_to_2D_array_from_floats() (*in module py4DSTEM.process.utils.utils*), 197
add_to_tree() (*emdfile.Node method*), 246
add_to_tree() (*py4DSTEM.Array method*), 26
add_to_tree() (*py4DSTEM.BraggVectors method*), 30
add_to_tree() (*py4DSTEM.Custom method*), 40
add_to_tree() (*py4DSTEM.DataCube method*), 50
add_to_tree() (*py4DSTEM.DiffractionSlice method*), 57
add_to_tree() (*py4DSTEM.Node method*), 63
add_to_tree() (*py4DSTEM.PointList method*), 65
add_to_tree() (*py4DSTEM.PointListArray method*), 68
add_to_tree() (*py4DSTEM.Probe method*), 73
add_to_tree() (*py4DSTEM.QPoints method*), 76
add_to_tree() (*py4DSTEM.RealSlice method*), 78
add_to_tree() (*py4DSTEM.VirtualDiffraction method*), 81
add_to_tree() (*py4DSTEM.VirtualImage method*), 83
add_vector() (*in module py4DSTEM.visualize.overlay*), 213
add_vector() (*in module py4DSTEM.visualize.vis_special*), 230
AffineTransform (*class in py4DSTEM.process.phase.utils*), 178
align_and_shift_images() (*in module py4DSTEM.process.utils.cross_correlate*), 190
align_images_fourier() (*in module py4DSTEM.process.utils.cross_correlate*), 189
Array (*class in emdfile*), 240
Array (*class in py4DSTEM*), 23
array_slice() (*in module py4DSTEM.process.phase.utils*), 181
asarray() (*py4DSTEM.process.phase.utils.AffineTransform method*), 179
asarray3() (*py4DSTEM.process.phase.utils.AffineTransform method*), 179
astuple() (*py4DSTEM.process.phase.utils.AffineTransform method*), 179
atomic_colors() (*in module py4DSTEM.process.diffraction.crystal_viz*), 167
attach() (*py4DSTEM.BraggVectors method*), 30
attach() (*py4DSTEM.Calibration method*), 39
attach() (*py4DSTEM.Data method*), 43
attach() (*py4DSTEM.DataCube method*), 50
attach() (*py4DSTEM.DiffractionSlice method*), 57
attach() (*py4DSTEM.Probe method*), 73
attach() (*py4DSTEM.QPoints method*), 76
attach() (*py4DSTEM.RealSlice method*), 78
attach() (*py4DSTEM.VirtualDiffraction method*), 81
attach() (*py4DSTEM.VirtualImage method*), 84
ax_addaxes() (*in module py4DSTEM.visualize.vis_RQ*), 214
ax_addaxes() (*in module py4DSTEM.visualize.vis_special*), 230
ax_addaxes_QtoR() (*in module py4DSTEM.visualize.vis_RQ*), 214
ax_addaxes_QtoR() (*in module py4DSTEM.visualize.vis_special*), 230
ax_addaxes_RtoQ() (*in module py4DSTEM.visualize.vis_RQ*), 214
ax_addvector() (*in module py4DSTEM.visualize.vis_RQ*), 215
ax_addvector_QtoR() (*in module py4DSTEM.visualize.vis_RQ*), 215
ax_addvector_RtoQ() (*in module py4DSTEM.visualize.vis_RQ*), 215

B

bilinear_kernel_density_estimate() (*in module py4DSTEM.process.phase.utils*), 185
bilinear_resample() (*in module py4DSTEM.process.phase.utils*), 186
bilinearly_interpolate_array() (*in module py4DSTEM.process.phase.utils*), 184
bin2D() (*in module py4DSTEM.preprocess.utils*), 100
bin_data_diffraction() (*in module py4DSTEM.preprocess.preprocess*), 97
bin_data_mmap() (*in module py4DSTEM.preprocess.preprocess*), 97
bin_data_real() (*in module py4DSTEM.preprocess.preprocess*), 97
bin_Q() (*py4DSTEM.DataCube method*), 44
bin_Q_mmap() (*py4DSTEM.DataCube method*), 44
bin_R() (*py4DSTEM.DataCube method*), 44
braggpeak_labels (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification attribute*), 112
BraggVectorClassification (*class in py4DSTEM.process.classification.braggvectorclassification*), 110
BraggVectors (*class in py4DSTEM*), 28
build() (*py4DSTEM.process.phase.utils.ComplexProbe method*), 177

C

cal (*py4DSTEM.BraggVectors property*), 29
calc_1D_profile() (*in module py4DSTEM.process.diffraction.utils*), 174
calculate_bragg_peak_histogram() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 145
calculate_coef_strain() (*in module py4DSTEM.process.rdf.amorph*), 187
calculate_dynamical_structure_factors() (*in module py4DSTEM.process.diffraction.crystal_bloch*), 152
calculate_dynamical_structure_factors() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 139
calculate_strain() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 150
calculate_strain() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 128
calculate_structure_factors() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 142
calculate_thresholds() (*in module py4DSTEM.preprocess.electroncount*), 95
calibrate() (*py4DSTEM.BraggVectors method*), 30
calibrate() (*py4DSTEM.DataCube method*), 43
calibrate_pixel_size() (*in module py4DSTEM.process.diffraction.crystal_calibrate*), 157
calibrate_pixel_size() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 134
calibrate_unit_cell() (*in module py4DSTEM.process.diffraction.crystal_calibrate*), 158
calibrate_unit_cell() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 135
Calibration (*class in py4DSTEM*), 37
cartesian_to_polar_transform_2Ddata() (*in module py4DSTEM.process.phase.utils*), 182
cartesian_to_polarelliptical_transform() (*in module py4DSTEM.process.utils.elliptical_coords*), 191
check_config() (*in module py4DSTEM*), 22
cluster_grains() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 150
cluster_grains() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 128
cluster_orientation_map() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 150
cluster_orientation_map() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 128
compare_QR_rotation() (*in module py4DSTEM.process.calibration.rotation*), 108
Complex2RGB() (*in module py4DSTEM.visualize.vis_special*), 229
ComplexOverlapKernelDiskLattice (*class in py4DSTEM.process.wholepatternfit.wp_models*), 203
ComplexProbe (*class in py4DSTEM.process.phase.utils*), 176
compute_divergence_periodic() (*in module py4DSTEM.process.phase.utils*), 181
compute_gradient_periodic() (*in module py4DSTEM.process.phase.utils*), 181
compute_polar_stack_symmetries() (*in module py4DSTEM.process.rdf.amorph*), 188
compute_WK_factor() (*in module py4DSTEM.process.diffraction.WK_scattering_factors*), 125
concatenate_features() (*py4DSTEM.process.classification.featurization.Featurization method*), 121
consensus() (*py4DSTEM.process.classification.featurization.Featurizatio method*), 124

constrain_degenerate_ellipse() (*in module py4DSTEM.process.calibration.ellipse*), 104
convert_ellipse_params() (*in module py4DSTEM.process.utils.elliptical_coords*), 190
convert_ellipse_params_r() (*in module py4DSTEM.process.utils.elliptical_coords*), 191
convert_stack_polar() (*in module py4DSTEM.process.rdf.amorph*), 188
copy() (*emdfile.Metadata method*), 244
copy() (*emdfile.PointList method*), 249
copy() (*emdfile.PointListArray method*), 249
copy() (*py4DSTEM.DataCube method*), 43
copy() (*py4DSTEM.Metadata method*), 60
copy() (*py4DSTEM.PointList method*), 65
copy() (*py4DSTEM.PointListArray method*), 67
copy() (*py4DSTEM.QPoints method*), 76
copy_to_device() (*in module py4DSTEM.process.phase.utils*), 187
counted_datacube_to_pointlistarray() (*in module py4DSTEM.preprocess.electroncount*), 96
counted_pointlistarray_to_datacube() (*in module py4DSTEM.preprocess.electroncount*), 96
crop_Q() (*py4DSTEM.DataCube method*), 44
crop_R() (*py4DSTEM.DataCube method*), 44
Crystal (*class in py4DSTEM.process.diffraction.crystal*), 125
Crystal_Phase (*class in py4DSTEM.process.diffraction.crystal_phase*), 159
Custom (*class in emdfile*), 243
Custom (*class in py4DSTEM*), 40
cut_from_tree() (*emdfile.Node method*), 247
cut_from_tree() (*py4DSTEM.Array method*), 26
cut_from_tree() (*py4DSTEM.BraggVectors method*), 30
cut_from_tree() (*py4DSTEM.Custom method*), 40
cut_from_tree() (*py4DSTEM.DataCube method*), 51
cut_from_tree() (*py4DSTEM.DiffractionSlice method*), 58
cut_from_tree() (*py4DSTEM.Node method*), 63
cut_from_tree() (*py4DSTEM.PointList method*), 65
cut_from_tree() (*py4DSTEM.PointListArray method*), 68
cut_from_tree() (*py4DSTEM.Probe method*), 73
cut_from_tree() (*py4DSTEM.QPoints method*), 76
cut_from_tree() (*py4DSTEM.RealSlice method*), 78
cut_from_tree() (*py4DSTEM.VirtualDiffraction method*), 81
cut_from_tree() (*py4DSTEM.VirtualImage method*), 84

D

Data (*class in py4DSTEM*), 42
DataCube (*class in py4DSTEM*), 43
datacube_diffraction_shift() (*in module py4DSTEM.preprocess.preprocess*), 98
DCBackground (*class in py4DSTEM.process.wholepatternfit.wp_models*), 199
dct_II_using_FFT_base() (*in module py4DSTEM.process.phase.utils*), 183
delete_features() (*py4DSTEM.process.classification.featurization.Feat method*), 122
dftUpsample() (*in module py4DSTEM.process.utils.multicorr*), 195
DiffractionSlice (*class in py4DSTEM*), 57
dim() (*emdfile.Array method*), 243
dim() (*py4DSTEM.Array method*), 26
dim() (*py4DSTEM.DataCube method*), 51
dim() (*py4DSTEM.DiffractionSlice method*), 58
dim() (*py4DSTEM.Probe method*), 73
dim() (*py4DSTEM.RealSlice method*), 79
dim() (*py4DSTEM.VirtualDiffraction method*), 81
dim() (*py4DSTEM.VirtualImage method*), 84
dirname() (*in module emdfile*), 250
double_sided_gaussian() (*in module py4DSTEM.process.calibration.ellipse*), 104
double_sided_gaussian_fiterr() (*in module py4DSTEM.process.calibration.ellipse*), 104
DynamicalMatrixCache (*class in py4DSTEM.process.diffraction.crystal_bloch*), 152

E

electron_count() (*in module py4DSTEM.preprocess.electroncount*), 94
electron_count_GPU() (*in module py4DSTEM.preprocess.electroncount*), 95
ellipse_err() (*in module py4DSTEM.process.calibration.ellipse*), 103
elliptical_resample() (*in module py4DSTEM.process.utils.elliptical_coords*), 192
elliptical_resample_datacube() (*in module py4DSTEM.process.utils.elliptical_coords*), 192
estimate_global_transformation() (*in module py4DSTEM.process.phase.utils*), 179
estimate_global_transformation_ransac() (*in module py4DSTEM.process.phase.utils*), 179
excitation_errors() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 145

F

Featurization (*class in py4DSTEM.process.classification.featurization*), 119
fft_shift() (*in module py4DSTEM.process.phase.utils*), 178
filter_2D_maxima() (*in module py4DSTEM.preprocess.utils*), 101
filter_hot_pixels() (*in module py4DSTEM.preprocess.preprocess*), 97
filter_hot_pixels() (*py4DSTEM.DataCube method*), 44
find_Bragg_disks() (*py4DSTEM.DataCube method*), 46
fit_1D_gaussian() (*in module py4DSTEM.process.fit.fit*), 175
fit_2D() (*in module py4DSTEM.process.fit.fit*), 175
fit_2D_polar_gaussian() (*in module py4DSTEM.process.fit.fit*), 175
fit_ellipse_1D() (*in module py4DSTEM.process.calibration.ellipse*), 102
fit_ellipse_amorphous_ring() (*in module py4DSTEM.process.calibration.ellipse*), 103
fit_origin() (*in module py4DSTEM.process.calibration.origin*), 105
fit_origin() (*py4DSTEM.BraggVectors method*), 30
fit_p_ellipse() (*py4DSTEM.BraggVectors method*), 31
fit_scattering_factor() (*in module py4DSTEM.process.rdf.rdf*), 188
fit_stack() (*in module py4DSTEM.process.rdf.amorph*), 187
force_add_to_tree() (*emdfile.Node method*), 246
force_add_to_tree() (*py4DSTEM.Array method*), 27
force_add_to_tree() (*py4DSTEM.BraggVectors method*), 31
force_add_to_tree() (*py4DSTEM.Custom method*), 40
force_add_to_tree() (*py4DSTEM.DataCube method*), 51
force_add_to_tree() (*py4DSTEM.DiffractionSlice method*), 58
force_add_to_tree() (*py4DSTEM.Node method*), 63
force_add_to_tree() (*py4DSTEM.PointList method*), 66
force_add_to_tree() (*py4DSTEM.PointListArray method*), 68
force_add_to_tree() (*py4DSTEM.Probe method*), 73
force_add_to_tree() (*py4DSTEM.QPoints method*), 76
force_add_to_tree() (*py4DSTEM.RealSlice method*), 79
force_add_to_tree() (*py4DSTEM.VirtualDiffraction method*), 81
force_add_to_tree() (*py4DSTEM.VirtualImage method*), 84
fourier_resample() (*in module py4DSTEM.process.utils.utils*), 197
fourier_ring_correlation() (*in module py4DSTEM.process.phase.utils*), 179
fourier_rotate_real_volume() (*in module py4DSTEM.process.phase.utils*), 181
fourier_translation_operator() (*in module py4DSTEM.process.phase.utils*), 177
from_ase() (*py4DSTEM.process.diffraction.crystal.Crystal static method*), 141
from_braggvectors() (*py4DSTEM.process.classification.featurization.Featurization method*), 121
from_CIF() (*py4DSTEM.process.diffraction.crystal.Crystal static method*), 141
from_h5() (*emdfile.Metadata class method*), 244
from_h5() (*emdfile.Node class method*), 248
from_h5() (*py4DSTEM.Array class method*), 27
from_h5() (*py4DSTEM.BraggVectors class method*), 31
from_h5() (*py4DSTEM.Calibration class method*), 40
from_h5() (*py4DSTEM.Custom class method*), 40
from_h5() (*py4DSTEM.DataCube class method*), 51
from_h5() (*py4DSTEM.DiffractionSlice class method*), 58
from_h5() (*py4DSTEM.Metadata class method*), 60
from_h5() (*py4DSTEM.Node class method*), 64
from_h5() (*py4DSTEM.PointList class method*), 66
from_h5() (*py4DSTEM.PointListArray class method*), 68
from_h5() (*py4DSTEM.Probe class method*), 73
from_h5() (*py4DSTEM.QPoints class method*), 76
from_h5() (*py4DSTEM.RealSlice class method*), 79
from_h5() (*py4DSTEM.VirtualDiffraction class method*), 81
from_h5() (*py4DSTEM.VirtualImage class method*), 84
from_prismatic() (*py4DSTEM.process.diffraction.crystal.Crystal static method*), 141
from_pymatgen_structure() (*py4DSTEM.process.diffraction.crystal.Crystal static method*), 141
from_unitcell_parameters() (*py4DSTEM.process.diffraction.crystal.Crystal static method*), 142
from_vacuum_data() (*py4DSTEM.Probe class method*), 70
fromarray() (*py4DSTEM.process.phase.utils.AffineTransform class method*), 179

G

GaussianBackground (*class in py4DSTEM.process.wholepatternfit.wp_models*), 199

GaussianRing (*class in py4DSTEM.process.wholepatternfit.wp_models*), 200
gdrive_download() (*in module py4DSTEM.io.google_drive_downloader*), 90
generate_CBED() (*in module py4DSTEM.process.diffraction.crystal_bloch*), 155
generate_CBED() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 138
generate_diffraction_pattern() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 143
generate_dynamical_diffraction_pattern() (*in module py4DSTEM.process.diffraction.crystal_bloch*), 153
generate_dynamical_diffraction_pattern() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 136
generate_moire_diffraction_pattern() (*in module py4DSTEM.process.diffraction.crystal*), 145
generate_projected_potential() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 144
generate_ring_pattern() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 143
generate_synthetic_probe() (*py4DSTEM.Probe class method*), 70
get_1D_polar_background() (*in module py4DSTEM.preprocess.radialbkgrd*), 99
get_2D_polar_background() (*in module py4DSTEM.preprocess.radialbkgrd*), 99
get_background_streaks() (*in module py4DSTEM.preprocess.darkreference*), 93
get_background_streaks_x() (*in module py4DSTEM.preprocess.darkreference*), 94
get_background_streaks_y() (*in module py4DSTEM.preprocess.darkreference*), 94
get_beamstop_mask() (*in module py4DSTEM.process.utils.masks*), 194
get_beamstop_mask() (*py4DSTEM.DataCube method*), 49
get_bksbtr_DP() (*in module py4DSTEM.preprocess.darkreference*), 93
get_bragg_vector_map() (*py4DSTEM.BraggVectors method*), 31
get_braggmask() (*py4DSTEM.DataCube method*), 50
get_braggpeak_labels_by_scan_position() (*in module py4DSTEM.process.classification.braggvectorclassification*), 117
get_bvm() (*py4DSTEM.BraggVectors method*), 32
get_calibrated_detector_geometry() (*py4DSTEM.DataCube static method*), 51
get_candidate_class() (*py4DSTEM.process.classification.braggvectorclassification.Brag method*), 116
get_candidate_class_BPs() (*py4DSTEM.process.classification.braggvectorclassification.Brag method*), 117
get_candidate_class_image() (*py4DSTEM.process.classification.braggvectorclassification.Brag method*), 117
get_class() (*py4DSTEM.process.classification.braggvectorclassification. method*), 116
get_class_BPs() (*py4DSTEM.process.classification.braggvectorclassific method*), 116
get_class_DP() (*in module py4DSTEM.process.classification.classutils*), 118
get_class_DP_without_Bragg_scattering() (*in module py4DSTEM.process.classification.classutils*), 119
get_class_DPs() (*py4DSTEM.process.classification.featurization.Featur method*), 123
get_class_image() (*py4DSTEM.process.classification.braggvectorclassi method*), 116
get_class_ims() (*py4DSTEM.process.classification.featurization.Featur method*), 124
get_CoM() (*in module py4DSTEM.process.utils.utils*), 196
get_cross_correlation() (*in module py4DSTEM.process.utils.cross_correlate*), 189
get_cross_correlation_FT() (*in module py4DSTEM.process.utils.cross_correlate*), 189
get_darkreference() (*in module py4DSTEM.preprocess.darkreference*), 93
get_dim() (*emdfile.Array method*), 243
get_dim() (*py4DSTEM.Array method*), 26
get_dim() (*py4DSTEM.DataCube method*), 51
get_dim() (*py4DSTEM.DiffractionSlice method*), 58
get_dim() (*py4DSTEM.Probe method*), 74
get_dim() (*py4DSTEM.RealSlice method*), 79
get_dim() (*py4DSTEM.VirtualDiffraction method*), 82
get_dim() (*py4DSTEM.VirtualImage method*), 84
get_dim_name() (*emdfile.Array method*), 243
get_dim_name() (*py4DSTEM.Array method*), 26
get_dim_name() (*py4DSTEM.DataCube method*), 51
get_dim_name() (*py4DSTEM.DiffractionSlice method*), 58
get_dim_name() (*py4DSTEM.Probe method*), 74
get_dim_name() (*py4DSTEM.RealSlice method*), 79
get_dim_name() (*py4DSTEM.VirtualDiffraction method*), 82

get_dim_name() (*py4DSTEM.VirtualImage method*), 84
get_dim_units() (*emdfile.Array method*), 243
get_dim_units() (*py4DSTEM.Array method*), 26
get_dim_units() (*py4DSTEM.DataCube method*), 52
get_dim_units() (*py4DSTEM.DiffractionSlice method*), 58
get_dim_units() (*py4DSTEM.Probe method*), 74
get_dim_units() (*py4DSTEM.RealSlice method*), 79
get_dim_units() (*py4DSTEM.VirtualDiffraction method*), 82
get_dim_units() (*py4DSTEM.VirtualImage method*), 84
get_dp_max() (*py4DSTEM.DataCube method*), 52
get_dp_mean() (*py4DSTEM.DataCube method*), 52
get_dp_median() (*py4DSTEM.DataCube method*), 52
get_dq_from_indexed_peaks() (*in module py4DSTEM.process.calibration.qpixelsize*), 108
get_ewpc_filter_function() (*in module py4DSTEM.process.utils.utils*), 197
get_from_tree() (*emdfile.Node method*), 247
get_from_tree() (*py4DSTEM.Array method*), 27
get_from_tree() (*py4DSTEM.BraggVectors method*), 32
get_from_tree() (*py4DSTEM.Custom method*), 41
get_from_tree() (*py4DSTEM.DataCube method*), 52
get_from_tree() (*py4DSTEM.DiffractionSlice method*), 58
get_from_tree() (*py4DSTEM.Node method*), 63
get_from_tree() (*py4DSTEM.PointList method*), 66
get_from_tree() (*py4DSTEM.PointListArray method*), 68
get_from_tree() (*py4DSTEM.Probe method*), 74
get_from_tree() (*py4DSTEM.QPoints method*), 77
get_from_tree() (*py4DSTEM.RealSlice method*), 79
get_from_tree() (*py4DSTEM.VirtualDiffraction method*), 82
get_from_tree() (*py4DSTEM.VirtualImage method*), 84
get_hdr_bits() (*in module py4DSTEM.io.filereaders.read_mib*), 89
get_initial_classes() (*in module py4DSTEM.process.classification.braggvectorclassification*), 117
get_initial_classes_by_cooccurrence() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 112
get_initial_classes_from_images() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 113
get_kernel() (*py4DSTEM.Probe method*), 71
get_local_ave_dp() (*py4DSTEM.DataCube method*), 50
get_mask() (*in module py4DSTEM.process.rdf.rdf*), 189
get_masked_peaks() (*py4DSTEM.BraggVectors method*), 32
get_maxima_1D() (*in module py4DSTEM.process.utils.utils*), 196
get_maxima_2D() (*in module py4DSTEM.preprocess.utils*), 100
get_mib_depth() (*in module py4DSTEM.io.filereaders.read_mib*), 89
get_mib_memmap() (*in module py4DSTEM.io.filereaders.read_mib*), 89
get_N_dataobjects() (*in module py4DSTEM.io.legacy.read_utils*), 92
get_nice_spacing() (*in module py4DSTEM.visualize.overlay*), 213
get_origin() (*in module py4DSTEM.process.calibration.origin*), 106
get_origin_friedel() (*in module py4DSTEM.process.calibration.origin*), 106
get_origin_single_dp() (*in module py4DSTEM.process.calibration.origin*), 106
get_phi() (*in module py4DSTEM.process.rdf.rdf*), 189
get_pointlist() (*emdfile.PointListArray method*), 249
get_pointlist() (*py4DSTEM.PointListArray method*), 67
get_probe_kernel_edge_gaussian() (*py4DSTEM.Probe static method*), 72
get_probe_kernel_edge_sigmoid() (*py4DSTEM.Probe static method*), 72
get_probe_kernel_flat() (*py4DSTEM.Probe static method*), 72
get_probe_size() (*in module py4DSTEM.process.calibration.probe*), 107
get_probe_size() (*py4DSTEM.DataCube method*), 46
get_py4DSTEM_topgroups() (*in module py4DSTEM.io.legacy.read_utils*), 92
get_py4DSTEM_version() (*in module py4DSTEM.io.legacy.read_utils*), 92
get_Q_pixel_size() (*in module py4DSTEM.process.calibration.qpixelsize*), 108
get_Qvector_from_Rvector() (*in module py4DSTEM.process.calibration.rotation*), 109
get_qx_qy_1d() (*in module py4DSTEM.process.utils.utils*), 196
get_radial_bkgrnd() (*py4DSTEM.DataCube method*), 49
get_radial_bksb_dp() (*py4DSTEM.DataCube method*), 49
get_radial_intensity() (*in module py4DSTEM.process.rdf.rdf*), 188
get_rdf() (*in module py4DSTEM.process.rdf.rdf*), 189
get_Rvector_from_Qvector() (*in module py4DSTEM.process.calibration.rotation*),

110
get_shift() (*in module py4DSTEM.process.utils.cross_correlate*), 189
get_shifted_ar() (*in module py4DSTEM.preprocess.utils*), 100
get_strained_crystal() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 140
get_UUID() (*in module py4DSTEM.io.legacy.read_utils*), 92
get_vacuum_probe() (*py4DSTEM.DataCube method*), 45
get_vectors() (*py4DSTEM.BraggVectors method*), 30
get_virtual_diffraction() (*py4DSTEM.DataCube method*), 52
get_virtual_image() (*py4DSTEM.BraggVectors method*), 32
get_virtual_image() (*py4DSTEM.DataCube method*), 53
get_voronoi_vertices() (*in module py4DSTEM.process.utils.utils*), 197
GMM() (*py4DSTEM.process.classification.featurization.Featurization method*), 123
graft() (*emdfile.Node method*), 247
graft() (*py4DSTEM.Array method*), 27
graft() (*py4DSTEM.BraggVectors method*), 33
graft() (*py4DSTEM.Custom method*), 41
graft() (*py4DSTEM.DataCube method*), 54
graft() (*py4DSTEM.DiffractionSlice method*), 58
graft() (*py4DSTEM.Node method*), 63
graft() (*py4DSTEM.PointList method*), 66
graft() (*py4DSTEM.PointListArray method*), 69
graft() (*py4DSTEM.Probe method*), 74
graft() (*py4DSTEM.QPoints method*), 77
graft() (*py4DSTEM.RealSlice method*), 79
graft() (*py4DSTEM.VirtualDiffraction method*), 82
graft() (*py4DSTEM.VirtualImage method*), 84

H

histogram() (*py4DSTEM.BraggVectors method*), 33

I

ICA() (*py4DSTEM.process.classification.featurization.Featurization method*), 122
idct_II_using_FFT() (*in module py4DSTEM.process.phase.utils*), 183
idct_II_using_FFT_base() (*in module py4DSTEM.process.phase.utils*), 183
import_file() (*in module py4DSTEM*), 17
import_file() (*in module py4DSTEM.io.importfile*), 90
interleave_ndarray_symmetrically() (*in module py4DSTEM.process.phase.utils*), 182
interleave_ndarray_symmetrically_inverse() (*in module py4DSTEM.process.phase.utils*), 183
is_color_like() (*in module py4DSTEM.visualize.overlay*), 214
is_py4DSTEM_file() (*in module py4DSTEM.io.legacy.read_utils*), 92
is_py4DSTEM_version13() (*in module py4DSTEM.io.legacy.read_utils*), 92

J

join() (*in module emdfile*), 250
join() (*in module py4DSTEM*), 22

K

K2DataArray (*class in py4DSTEM.io.filereaders.read_K2*), 87
KernelDiskLattice (*class in py4DSTEM.process.wholepatternfit.wp_models*), 203

L

lanczos_interpolate_array() (*in module py4DSTEM.process.phase.utils*), 184
lanczos_kernel_density_estimate() (*in module py4DSTEM.process.phase.utils*), 185
linear_interpolation_1D() (*in module py4DSTEM.process.utils.utils*), 197
linear_interpolation_2D() (*in module py4DSTEM.preprocess.utils*), 101
load_mib() (*in module py4DSTEM.io.filereaders.read_mib*), 88

M

make_axes_locatable() (*in module py4DSTEM.visualize.vis_special*), 230
make_bragg_mask() (*py4DSTEM.DataCube method*), 54
make_circular_mask() (*in module py4DSTEM.process.utils.masks*), 194
make_detector() (*py4DSTEM.DataCube static method*), 55
make_flowline_combined_image() (*in module py4DSTEM.process.diffraction.flowlines*), 171
make_flowline_map() (*in module py4DSTEM.process.diffraction.flowlines*), 169
make_flowline_rainbow_image() (*in module py4DSTEM.process.diffraction.flowlines*), 170
make_flowline_rainbow_legend() (*in module py4DSTEM.process.diffraction.flowlines*), 171
make_Fourier_coords2D() (*in module py4DSTEM.preprocess.utils*), 100

make_orientation_histogram() (*in module py4DSTEM.process.diffraction.flowlines*), 168
manageHeader() (*in module py4DSTEM.io.filereaders.read_mib*), 88
mask_in_Q() (*py4DSTEM.BraggVectors method*), 34
mask_in_R() (*py4DSTEM.BraggVectors method*), 34
match_orientations() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 148
match_orientations() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 126
match_single_pattern() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 149
match_single_pattern() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 127
max_feature() (*py4DSTEM.process.classification.featurization.FeatureMaker method*), 122
mean_feature() (*py4DSTEM.process.classification.featurization.FeatureMaker method*), 122
measure_disk() (*py4DSTEM.Probe method*), 71
measure_origin() (*py4DSTEM.BraggVectors method*), 34
measure_origin_beamstop() (*py4DSTEM.BraggVectors method*), 35
median_feature() (*py4DSTEM.process.classification.featurization.FeatureMaker method*), 122
median_filter_masked_pixels() (*in module py4DSTEM.preprocess.preprocess*), 98
median_filter_masked_pixels() (*py4DSTEM.DataCube method*), 45
merge() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 114
merge_by_class_index() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 115
merge_iterative() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 115
Metadata (*class in emdfile*), 244
Metadata (*class in py4DSTEM*), 60
MinMaxScaler() (*py4DSTEM.process.classification.featurization.FeatureMaker method*), 122
module
    py4DSTEM.io, 86
    py4DSTEM.io.filereaders, 86
    py4DSTEM.io.filereaders.empad, 86
    py4DSTEM.io.filereaders.read_K2, 87
    py4DSTEM.io.filereaders.read_mib, 88
    py4DSTEM.io.google_drive_downloader, 90
    py4DSTEM.io.google_drive_downloader.gdown, 90
    py4DSTEM.io.importfile, 90
    py4DSTEM.io.legacy, 91
    py4DSTEM.io.legacy.h5py, 91
    py4DSTEM.io.legacy.legacy12, 91
    py4DSTEM.io.legacy.legacy13, 91
    py4DSTEM.io.legacy.read_legacy_12, 91
    py4DSTEM.io.legacy.read_legacy_13, 91
    py4DSTEM.io.legacy.read_utils, 92
    py4DSTEM.io.parsefiletype, 92
    py4DSTEM.preprocess.darkreference, 93
    py4DSTEM.preprocess.electroncount, 94
    py4DSTEM.preprocess.preprocess, 96
    py4DSTEM.preprocess.radialbkgrd, 99
    py4DSTEM.preprocess.utils, 100
    py4DSTEM.process, 102
    py4DSTEM.process.calibration, 102
    py4DSTEM.process.calibration.ellipse, 102
    py4DSTEM.process.calibration.origin, 105
    py4DSTEM.process.calibration.probe, 107
    py4DSTEM.process.calibration.qpixelsize, 108
    py4DSTEM.process.calibration.rotation, 108
    py4DSTEM.process.classification, 110
    py4DSTEM.process.classification.braggvectorclassificat 110
    py4DSTEM.process.classification.classutils, 118
    py4DSTEM.process.classification.featurization, 119
    py4DSTEM.process.diffraction, 125
    py4DSTEM.process.diffraction.crystal, 125
    py4DSTEM.process.diffraction.crystal_ACOM, 147
    py4DSTEM.process.diffraction.crystal_bloch, 152
    py4DSTEM.process.diffraction.crystal_calibrate, 157
    py4DSTEM.process.diffraction.crystal_phase, 159
    py4DSTEM.process.diffraction.crystal_viz, 160
    py4DSTEM.process.diffraction.flowlines, 168
    py4DSTEM.process.diffraction.sys, 173
    py4DSTEM.process.diffraction.utils, 174
    py4DSTEM.process.diffraction.WK_scattering_factors, 125
    py4DSTEM.process.fit, 175
    py4DSTEM.process.fit.fit, 175
    py4DSTEM.process.phase, 176
    py4DSTEM.process.phase.utils, 176
    py4DSTEM.process.rdf.amorph, 187
    py4DSTEM.process.rdf.rdf, 188

py4DSTEM.process.utils, 189
py4DSTEM.process.utils.cross_correlate, 189
py4DSTEM.process.utils.elliptical_coords, 190
py4DSTEM.process.utils.masks, 194
py4DSTEM.process.utils.multicorr, 194
py4DSTEM.process.utils.utils, 196
py4DSTEM.process.wholepatternfit, 198
py4DSTEM.process.wholepatternfit.wp_models, 198
py4DSTEM.process.wholepatternfit.wpf, 204
py4DSTEM.process.wholepatternfit.wpf_viz, 204

N

N_feat (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification attribute*), 112
N_meas (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification attribute*), 112
newnode() (*emdfile.Node static method*), 248
newnode() (*py4DSTEM.Array static method*), 27
newnode() (*py4DSTEM.BraggVectors static method*), 35
newnode() (*py4DSTEM.Custom static method*), 41
newnode() (*py4DSTEM.DataCube static method*), 55
newnode() (*py4DSTEM.DiffractionSlice static method*), 59
newnode() (*py4DSTEM.Node static method*), 64
newnode() (*py4DSTEM.PointList static method*), 66
newnode() (*py4DSTEM.PointListArray static method*), 69
newnode() (*py4DSTEM.Probe static method*), 74
newnode() (*py4DSTEM.QPoints static method*), 77
newnode() (*py4DSTEM.RealSlice static method*), 79
newnode() (*py4DSTEM.VirtualDiffraction static method*), 82
newnode() (*py4DSTEM.VirtualImage static method*), 85
nmf() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 113
NMF() (*py4DSTEM.process.classification.featurization.Featurization method*), 122
Node (*class in emdfile*), 244
Node (*class in py4DSTEM*), 61

O

Orientation (*class in py4DSTEM.process.diffraction.utils*), 174
orientation_correlation() (*in module py4DSTEM.process.diffraction.flowlines*), 172
orientation_plan() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 147
orientation_plan() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 125
OrientationMap (*class in py4DSTEM.process.diffraction.utils*), 174

P

pad_data_diffraction() (*in module py4DSTEM.preprocess.preprocess*), 98
pad_Q() (*py4DSTEM.DataCube method*), 44
parse_hdr() (*in module py4DSTEM.io.filereaders.read_mib*), 88
partition_list() (*in module py4DSTEM.process.phase.utils*), 187
PCA() (*py4DSTEM.process.classification.featurization.Featurization method*), 122
periodic_centered_difference() (*in module py4DSTEM.process.phase.utils*), 181
pixel_rolling_kernel_density_estimate() (*in module py4DSTEM.process.phase.utils*), 184
plot() (*py4DSTEM.BraggVectors method*), 35
plot_all_phase_maps() (*py4DSTEM.process.diffraction.crystal_phase.Crystal_Phase method*), 159
plot_cluster_size() (*in module py4DSTEM.process.diffraction.crystal_viz*), 167
plot_cluster_size() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 134
plot_clusters() (*in module py4DSTEM.process.diffraction.crystal_viz*), 166
plot_clusters() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 134
plot_diffraction_pattern() (*in module py4DSTEM.process.diffraction.crystal_viz*), 163
plot_fiber_orientation_maps() (*in module py4DSTEM.process.diffraction.crystal_viz*), 165
plot_fiber_orientation_maps() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 133
plot_moire_diffraction_pattern() (*in module py4DSTEM.process.diffraction.crystal*), 146
plot_orientation_correlation() (*in module py4DSTEM.process.diffraction.flowlines*), 172
plot_orientation_maps() (*in module py4DSTEM.process.diffraction.crystal_viz*), 164
plot_orientation_maps() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 132
plot_orientation_plan() (*in module*

py4DSTEM.process.diffraction.crystal_viz), 163
plot_orientation_plan() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 132
plot_orientation_zones() (*in module py4DSTEM.process.diffraction.crystal_viz*), 162
plot_orientation_zones() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 131
plot_ring_pattern() (*in module py4DSTEM.process.diffraction.crystal_viz*), 167
plot_scattering_intensity() (*in module py4DSTEM.process.diffraction.crystal_viz*), 162
plot_scattering_intensity() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 131
plot_strains() (*in module py4DSTEM.process.rdf.amorph*), 188
plot_structure() (*in module py4DSTEM.process.diffraction.crystal_viz*), 160
plot_structure() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 129
plot_structure_factors() (*in module py4DSTEM.process.diffraction.crystal_viz*), 161
plot_structure_factors() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 130
plot_symmetries() (*in module py4DSTEM.process.rdf.amorph*), 188
PointList (*class in emdfile*), 248
PointList (*class in py4DSTEM*), 65
PointListArray (*class in emdfile*), 249
PointListArray (*class in py4DSTEM*), 67
polar_aliases (*in module py4DSTEM.process.phase.utils*), 176
polar_coordinates() (*py4DSTEM.process.phase.utils.ComplexProbe method*), 177
polar_symbols (*in module py4DSTEM.process.phase.utils*), 176
polar_to_cartesian_transform_2Ddata() (*in module py4DSTEM.process.phase.utils*), 182
position_detector() (*py4DSTEM.DataCube method*), 55
positions (*py4DSTEM.process.diffraction.crystal.Crystal attribute*), 140
preconditioned_laplacian_neumann_2D() (*in module py4DSTEM.process.phase.utils*), 183
preconditioned_laplacian_periodic_3D() (*in module py4DSTEM.process.phase.utils*), 181
preconditioned_poisson_solver_neumann_2D() (*in module py4DSTEM.process.phase.utils*), 183
preconditioned_poisson_solver_periodic_3D() (*in module py4DSTEM.process.phase.utils*), 181
print_h5_tree() (*in module emdfile*), 250
print_h5_tree() (*in module py4DSTEM*), 17
print_v13h5_tree() (*in module py4DSTEM.io.legacy.read_legacy_13*), 92
print_v13h5pyFile_tree() (*in module py4DSTEM.io.legacy.read_legacy_13*), 92
Probe (*class in py4DSTEM*), 70
project_vector_field_divergence_periodic_3D() (*in module py4DSTEM.process.phase.utils*), 182
py4DSTEM.io
    module, 86
py4DSTEM.io.filereaders
    module, 86
py4DSTEM.io.filereaders.empad
    module, 86
py4DSTEM.io.filereaders.read_K2
    module, 87
py4DSTEM.io.filereaders.read_mib
    module, 88
py4DSTEM.io.google_drive_downloader
    module, 90
py4DSTEM.io.google_drive_downloader.gdown
    module, 90
py4DSTEM.io.importfile
    module, 90
py4DSTEM.io.legacy
    module, 91
py4DSTEM.io.legacy.h5py
    module, 91
py4DSTEM.io.legacy.legacy12
    module, 91
py4DSTEM.io.legacy.legacy13
    module, 91
py4DSTEM.io.legacy.read_legacy_12
    module, 91
py4DSTEM.io.legacy.read_legacy_13
    module, 91
py4DSTEM.io.legacy.read_utils
    module, 92
py4DSTEM.io.parsefiletype
    module, 92
py4DSTEM.preprocess.darkreference
    module, 93
py4DSTEM.preprocess.electroncount
    module, 94

py4DSTEM.preprocess.preprocess
module, 96
py4DSTEM.preprocess.radialbkgrd
module, 99
py4DSTEM.preprocess.utils
module, 100
py4DSTEM.process
module, 102
py4DSTEM.process.calibration
module, 102
py4DSTEM.process.calibration.ellipse
module, 102
py4DSTEM.process.calibration.origin
module, 105
py4DSTEM.process.calibration.probe
module, 107
py4DSTEM.process.calibration.qpixelsize
module, 108
py4DSTEM.process.calibration.rotation
module, 108
py4DSTEM.process.classification
module, 110
py4DSTEM.process.classification.braggvectorclassification
module, 110
py4DSTEM.process.classification.classutils
module, 118
py4DSTEM.process.classification.featurization
module, 119
py4DSTEM.process.diffraction
module, 125
py4DSTEM.process.diffraction.crystal
module, 125
py4DSTEM.process.diffraction.crystal_ACOM
module, 147
py4DSTEM.process.diffraction.crystal_bloch
module, 152
py4DSTEM.process.diffraction.crystal_calibrate
module, 157
py4DSTEM.process.diffraction.crystal_phase
module, 159
py4DSTEM.process.diffraction.crystal_viz
module, 160
py4DSTEM.process.diffraction.flowlines
module, 168
py4DSTEM.process.diffraction.sys
module, 173
py4DSTEM.process.diffraction.utils
module, 174
py4DSTEM.process.diffraction.WK_scattering_factors
module, 125
py4DSTEM.process.fit
module, 175
py4DSTEM.process.fit.fit
module, 175
py4DSTEM.process.phase
module, 176
py4DSTEM.process.phase.utils
module, 176
py4DSTEM.process.rdf.amorph
module, 187
py4DSTEM.process.rdf.rdf
module, 188
py4DSTEM.process.utils
module, 189
py4DSTEM.process.utils.cross_correlate
module, 189
py4DSTEM.process.utils.elliptical_coords
module, 190
py4DSTEM.process.utils.masks
module, 194
py4DSTEM.process.utils.multicorr
module, 194
py4DSTEM.process.utils.utils
module, 196
py4DSTEM.process.wholepatternfit
module, 198
py4DSTEM.process.wholepatternfit.wp_models
module, 198
py4DSTEM.process.wholepatternfit.wpf
module, 204
py4DSTEM.process.wholepatternfit.wpf_viz
module, 204

Q

QPoints (*class in py4DSTEM*), 75
quantify_phase() (*py4DSTEM.process.diffraction.crystal_phase.Crystal method*), 159
quantify_phase_pointlist() (*py4DSTEM.process.diffraction.crystal_phase.Crystal_Phase method*), 160
Qx (*py4DSTEM.process.classification.braggvectorclassification.BraggVecto attribute*), 112
Qy (*py4DSTEM.process.classification.braggvectorclassification.BraggVecto attribute*), 112

R

R_Nx (*py4DSTEM.process.classification.braggvectorclassification.BraggVe attribute*), 112
R_Ny (*py4DSTEM.process.classification.braggvectorclassification.BraggVe attribute*), 112
radial_elliptical_integral() (*in module py4DSTEM.process.utils.elliptical_coords*), 193
radial_integral() (*in module py4DSTEM.process.utils.elliptical_coords*), 193
radial_reduction() (*in module py4DSTEM.process.utils.utils*), 196

raw (*py4DSTEM.BraggVectors property*), 29
read() (*in module emdfile*), 250
read() (*in module py4DSTEM*), 16
read_empad() (*in module py4DSTEM.io.filereaders.empad*), 86
read_gatan_K2_bin() (*in module py4DSTEM.io.filereaders.read_K2*), 87
read_legacy12() (*in module py4DSTEM.io.legacy.read_legacy_12*), 91
read_legacy13() (*in module py4DSTEM.io.legacy.read_legacy_13*), 91
RealSlice (*class in py4DSTEM*), 78
register_target() (*py4DSTEM.Calibration method*), 39
regularize_probe_amplitude() (*in module py4DSTEM.process.phase.utils*), 182
reject() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 116
remove() (*emdfile.PointList method*), 248
remove() (*py4DSTEM.PointList method*), 65
remove() (*py4DSTEM.QPoints method*), 77
remove_class() (*py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method*), 115
resample_data_diffraction() (*in module py4DSTEM.preprocess.preprocess*), 98
resample_Q() (*py4DSTEM.DataCube method*), 44
return_1D_profile() (*in module py4DSTEM.process.phase.utils*), 180
RIH2() (*in module py4DSTEM.process.diffraction.WK_scattering*), 125
RobustScaler() (*py4DSTEM.process.classification.featurization.Featurization method*), 122
Root (*class in emdfile*), 250
rotate_point() (*in module py4DSTEM.process.phase.utils*), 183

S

save() (*in module emdfile*), 251
save() (*in module py4DSTEM*), 17
save_ang_file() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 151
save_ang_file() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 129
sector_mask() (*in module py4DSTEM.process.utils.utils*), 196
select_point() (*in module py4DSTEM.visualize.vis_special*), 230
set_author() (*in module emdfile*), 252
set_dim() (*emdfile.Array method*), 243
set_dim() (*py4DSTEM.Array method*), 26
set_dim() (*py4DSTEM.DataCube method*), 56
set_dim() (*py4DSTEM.DiffractionSlice method*), 59
set_dim() (*py4DSTEM.Probe method*), 74
set_dim() (*py4DSTEM.RealSlice method*), 80
set_dim() (*py4DSTEM.VirtualDiffraction method*), 82
set_dim() (*py4DSTEM.VirtualImage method*), 85
set_dim_name() (*emdfile.Array method*), 243
set_dim_name() (*py4DSTEM.Array method*), 26
set_dim_name() (*py4DSTEM.DataCube method*), 56
set_dim_name() (*py4DSTEM.DiffractionSlice method*), 59
set_dim_name() (*py4DSTEM.Probe method*), 74
set_dim_name() (*py4DSTEM.RealSlice method*), 80
set_dim_name() (*py4DSTEM.VirtualDiffraction method*), 82
set_dim_name() (*py4DSTEM.VirtualImage method*), 85
set_dim_units() (*emdfile.Array method*), 243
set_dim_units() (*py4DSTEM.Array method*), 26
set_dim_units() (*py4DSTEM.DataCube method*), 56
set_dim_units() (*py4DSTEM.DiffractionSlice method*), 59
set_dim_units() (*py4DSTEM.Probe method*), 74
set_dim_units() (*py4DSTEM.RealSlice method*), 80
set_dim_units() (*py4DSTEM.VirtualDiffraction method*), 82
set_dim_units() (*py4DSTEM.VirtualImage method*), 85
set_origin_meas() (*py4DSTEM.Calibration method*), 39
set_parameters() (*py4DSTEM.process.phase.utils.ComplexProbe method*), 177
set_probe_param() (*py4DSTEM.Calibration method*), 39
set_raw_vectors() (*py4DSTEM.BraggVectors method*), 29
set_scan_shape() (*in module py4DSTEM.preprocess.preprocess*), 96
set_scan_shape() (*py4DSTEM.DataCube method*), 43
setcal() (*py4DSTEM.BraggVectors method*), 29
setup_diffraction() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 142
shift_positive() (*py4DSTEM.process.classification.featurization.Featu method*), 122
show() (*in module py4DSTEM*), 17
show() (*in module py4DSTEM.visualize*), 205
show() (*in module py4DSTEM.visualize.vis_grid*), 222
show() (*in module py4DSTEM.visualize.vis_RQ*), 215
show() (*in module py4DSTEM.visualize.vis_special*), 230
show_amorphous_ring_fit() (*in module py4DSTEM.visualize.vis_special*), 235
show_annuli() (*in module py4DSTEM.visualize*), 211
show_circles() (*in module py4DSTEM.visualize*), 210
show_class_BPs() (*in module py4DSTEM.visualize.vis_special*), 236
show_class_BPs_grid() (*in module*

py4DSTEM.visualize.vis_special), 236
show_complex() (*in module py4DSTEM.visualize.vis_special*), 236
show_DP_grid() (*in module py4DSTEM.visualize.vis_grid*), 227
show_ellipses() (*in module py4DSTEM.visualize*), 211
show_elliptical_fit() (*in module py4DSTEM.visualize.vis_special*), 236
show_grid_overlay() (*in module py4DSTEM.visualize.vis_grid*), 227
show_hist() (*in module py4DSTEM.visualize*), 209
show_image_grid() (*in module py4DSTEM.visualize.vis_grid*), 227
show_image_grid() (*in module py4DSTEM.visualize.vis_special*), 237
show_kernel() (*in module py4DSTEM.visualize.vis_special*), 238
show_lattice_points() (*in module py4DSTEM.process.wholepatternfit.wpf_viz*), 204
show_max_peak_spacing() (*in module py4DSTEM.visualize.vis_special*), 238
show_origin_fit() (*in module py4DSTEM.visualize.vis_special*), 238
show_origin_meas() (*in module py4DSTEM.visualize.vis_special*), 238
show_pointlabels() (*in module py4DSTEM.visualize.vis_special*), 238
show_points() (*in module py4DSTEM.visualize*), 211
show_points() (*in module py4DSTEM.visualize.vis_grid*), 228
show_points() (*in module py4DSTEM.visualize.vis_RQ*), 222
show_Q() (*in module py4DSTEM.visualize*), 210
show_qprofile() (*in module py4DSTEM.visualize.vis_special*), 238
show_rectangles() (*in module py4DSTEM.visualize*), 210
show_RQ() (*in module py4DSTEM.visualize.vis_RQ*), 220
show_RQ_axes() (*in module py4DSTEM.visualize.vis_RQ*), 220
show_RQ_vector() (*in module py4DSTEM.visualize.vis_RQ*), 221
show_RQ_vectors() (*in module py4DSTEM.visualize.vis_RQ*), 221
show_selected_dp() (*in module py4DSTEM.visualize.vis_RQ*), 222
show_selected_dps() (*in module py4DSTEM.visualize.vis_special*), 238
show_strain() (*in module py4DSTEM.visualize.vis_special*), 239
show_tree() (*emdfile.Node method*), 246
show_tree() (*py4DSTEM.Array method*), 28
show_tree() (*py4DSTEM.BraggVectors method*), 36
show_tree() (*py4DSTEM.Custom method*), 41
show_tree() (*py4DSTEM.DataCube method*), 56
show_tree() (*py4DSTEM.DiffractionSlice method*), 59
show_tree() (*py4DSTEM.Node method*), 63
show_tree() (*py4DSTEM.PointList method*), 66
show_tree() (*py4DSTEM.PointListArray method*), 69
show_tree() (*py4DSTEM.Probe method*), 75
show_tree() (*py4DSTEM.QPoints method*), 77
show_tree() (*py4DSTEM.RealSlice method*), 80
show_tree() (*py4DSTEM.VirtualDiffraction method*), 83
show_tree() (*py4DSTEM.VirtualImage method*), 85
show_voronoi() (*in module py4DSTEM.visualize.vis_special*), 240
sort() (*emdfile.PointList method*), 248
sort() (*py4DSTEM.PointList method*), 65
sort() (*py4DSTEM.QPoints method*), 77
sort_orientation_maps() (*in module py4DSTEM.process.diffraction.utils*), 174
spatial_frequencies() (*in module py4DSTEM.process.phase.utils*), 177
spatial_separation() (*py4DSTEM.process.classification.featurization.Featurization method*), 124
split() (*py4DSTEM.process.classification.braggvectorclassification.Brag method*), 114
split_by_class_index() (*py4DSTEM.process.classification.braggvectorclassification.Brag method*), 115
subdivide_into_batches() (*in module py4DSTEM.process.phase.utils*), 178
swap_Qxy() (*in module py4DSTEM.preprocess.preprocess*), 97
swap_Qxy() (*py4DSTEM.DataCube method*), 43
swap_RQ() (*in module py4DSTEM.preprocess.preprocess*), 96
swap_RQ() (*py4DSTEM.DataCube method*), 43
swap_Rxy() (*in module py4DSTEM.preprocess.preprocess*), 97
swap_Rxy() (*py4DSTEM.DataCube method*), 43
symmetry_reduce_directions() (*in module py4DSTEM.process.diffraction.crystal_ACOM*), 152
symmetry_reduce_directions() (*py4DSTEM.process.diffraction.crystal.Crystal method*), 129
SyntheticDiskLattice (*class in py4DSTEM.process.wholepatternfit.wp_models*), 200
SyntheticDiskMoire (*class in py4DSTEM.process.wholepatternfit.wp_models*), 202

T

thin_data_real() (*in module py4DSTEM.preprocess.preprocess*), 97
thin_R() (*py4DSTEM.DataCube method*), 44
to_h5() (*emdfile.Array method*), 243
to_h5() (*emdfile.Custom method*), 243
to_h5() (*emdfile.Metadata method*), 244
to_h5() (*emdfile.Node method*), 248
to_h5() (*emdfile.PointList method*), 249
to_h5() (*emdfile.PointListArray method*), 250
to_h5() (*py4DSTEM.Array method*), 26
to_h5() (*py4DSTEM.BraggVectors method*), 30
to_h5() (*py4DSTEM.Calibration method*), 40
to_h5() (*py4DSTEM.Custom method*), 40
to_h5() (*py4DSTEM.DataCube method*), 56
to_h5() (*py4DSTEM.DiffractionSlice method*), 59
to_h5() (*py4DSTEM.Metadata method*), 60
to_h5() (*py4DSTEM.Node method*), 64
to_h5() (*py4DSTEM.PointList method*), 67
to_h5() (*py4DSTEM.PointListArray method*), 68
to_h5() (*py4DSTEM.Probe method*), 75
to_h5() (*py4DSTEM.QPoints method*), 77
to_h5() (*py4DSTEM.RealSlice method*), 80
to_h5() (*py4DSTEM.VirtualDiffraction method*), 83
to_h5() (*py4DSTEM.VirtualImage method*), 85
to_strainmap() (*py4DSTEM.BraggVectors method*), 36
torch_bin() (*in module py4DSTEM.preprocess.electroncount*), 96
tqdmnd() (*in module emdfile*), 252
tqdmnd() (*in module py4DSTEM*), 22
tree() (*emdfile.Node method*), 247
tree() (*py4DSTEM.Array method*), 28
tree() (*py4DSTEM.BraggVectors method*), 36
tree() (*py4DSTEM.Custom method*), 41
tree() (*py4DSTEM.DataCube method*), 57
tree() (*py4DSTEM.DiffractionSlice method*), 59
tree() (*py4DSTEM.Node method*), 63
tree() (*py4DSTEM.PointList method*), 66
tree() (*py4DSTEM.PointListArray method*), 69
tree() (*py4DSTEM.Probe method*), 75
tree() (*py4DSTEM.QPoints method*), 77
tree() (*py4DSTEM.RealSlice method*), 80
tree() (*py4DSTEM.VirtualDiffraction method*), 83
tree() (*py4DSTEM.VirtualImage method*), 85

U

unregister_target() (*py4DSTEM.Calibration method*), 39
unwrap_phase_2d() (*in module py4DSTEM.process.phase.utils*), 183
upsampled_correlation() (*in module py4DSTEM.process.utils.multicorr*), 194
upsampleFFT() (*in module py4DSTEM.process.utils.multicorr*), 195

V

vectorized_fourier_resample() (*in module py4DSTEM.process.phase.utils*), 186
version_is_geq() (*in module py4DSTEM.io.legacy.read_utils*), 92
VirtualDiffraction (*class in py4DSTEM*), 81
VirtualImage (*class in py4DSTEM*), 83
visualize() (*py4DSTEM.process.phase.utils.ComplexProbe method*), 177

W

WPFModel (*class in py4DSTEM.process.wholepatternfit.wp_models*), 198
WPFModelType (*class in py4DSTEM.process.wholepatternfit.wp_models*), 198

X

X (*py4DSTEM.process.classification.braggvectorclassification.BraggVector attribute*), 112