py4dstmt

Release 0.14.0

Ben Savitsky & Alex Rakowski

Apr 26, 2023
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 ............................................................ 301
  1.6 Graphical User Interface ........................................... 301
  1.7 Support & Contributions ........................................... 301
  1.8 License ............................................................ 302
  1.9 Acknowledgements .................................................. 315

2 Indices and tables 317

Python Module Index 319

Index 321
py4DSTEM is an open source set of python tools for processing and analysis of *four-dimensional scanning transmission electron microscopy (4D-STEM)* data.
1.1 What is 4D-STEM?

Scanning Transmission Electron Microscopy (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
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

```bash
1 conda create -n py4dstem python=3.9
2 conda activate py4dstem
3 conda install -c conda-forge py4dstem
4 conda install -c conda-forge pywin32
5 # optional but recommended
6 conda install jupyterlab pymatgen
```

Linux

Listing 2: Linux base install

```bash
1 conda create -n py4dstem python=3.9
2 conda activate py4dstem
3 conda install -c conda-forge py4dstem
4 # optional but recommended
5 conda install jupyterlab pymatgen
```

Mac (Intel)

Listing 3: Intel Mac base install

```bash
1 conda create -n py4dstem python=3.9
2 conda activate py4dstem
3 conda install -c conda-forge py4dstem
4 # optional but recommended
5 conda install jupyterlab pymatgen
```

Mac (Apple Silicon M1/M2)
### 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

```bash
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

```bash
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

```bash
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

```bash
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

```bash
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

```bash
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

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

```bash
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

```bash
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

```bash
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

```bash
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

```bash
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

```bash
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

```bash
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](https://github.com).

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

```bash
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 contribute your own features you’ll need to install py4DSTEM from source. Luckily this is easy and can be done by simply running:

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

Alternatively, you can try single step method:

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

1.2. Installation
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 available or simply run the below to get the latest version. While Docker is extremely powerful and aims to greatly simplify deploying 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.

```bash
1   docker pull arakowsk/py4dstem:latest
2   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 depending on os, terminal etc.
- GPU drivers - tricky to explain

1.2.5 Virtual Environments

Attention: Virtual environments

A Python virtual environment is its own isolated 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.
1.3 Examples

1.3.1 First Steps

Once py4DSTEM has been successfully 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

```python
# 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 demonstrating 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 analyses you’ll learn running the tutorials.
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 built on top of, see https://github.com/py4dstem/emdfile.

To read file written by older versions of py4DSTEM, different keyword arguments should be passed. See the docstring for py4DSTEM.io.native.legacy.read_py4DSTEM_legacy for further details.

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 and 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', binfo: 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.
- **binfo** *(int)* – Diffraction space binning factor for bin-on-load.
- **filetype** *(str)* – Used to override automatic filetype detection.
- ****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 a .h5 file at filepath. Specific behavior depends on the `data`, `mode`, `tree`, and `emdpath` arguments.

Calling

```python
>>> 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’)

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 determine 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.

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

Prints the contents of an h5 file from a filepath.
**Plotting**

```python
py4DSTEM.show(ar, figsize=(8, 8), 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, polar_elliptical_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, **kwargs)
```

General visualization function for 2D arrays.

The simplest use of this function is:

```python
>>> 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:

```python
>>> 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:

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

will overlay a single red square, and

```python
>>> 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:

```python
>>> 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

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

```python
>>> 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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 colormap; 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 minimum 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 function’s 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 masks=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 calibration or pixelsize/pixelunits for scalebar. If False, no scalebar is added.

- ****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 requirement for py4DSTEM, and perform any additional checks. Defaults to False.

- **gratuitously_verbose** *(bool, optional)* – Provides more in-depth 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:

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

is equivalent to

```python
>>> 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**

```python
class py4DSTEM.Array(data: ndarray, name: str | None = None, units: str | None = 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:

```python
>>> 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:

```python
>>> ar = Array(
>>>     np.ones((20,20,256,256)),
>>>     name = 'test_array',
>>>     units = 'intensity',
>>>     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,

```python
>>> 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.

```python
>>> 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

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

```python
>>> 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 \textit{dims} parameter is not passed, all dim vectors are populated this way.

**dim\_units** (list): \textbf{the units for the calibration dim vectors. If nothing is passed, all dim 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): \textbf{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): \textbf{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 \textit{slicelabels} list, with the syntax array\['label'] or array.get_slice('label'). If \textit{slicelabels} is True or is a list with length less than the final dimension length, unassigned dimensions are autopopulated with labels array\textit{ij}. The flag array.is\_stack is set to True and the array.rank attribute is set to N-1.}

Returns
A new Array instance

\textit{get\_dim}(\textit{n})
Return the \textit{n}’th dim vector

\textit{dim}(\textit{n})
Return the \textit{n}’th dim vector

\textit{set\_dim}(\textit{n}: int, \textit{dim}: list | ndarray, \textit{units}: str | None = None, \textit{name}: str | None = None)
Sets the \textit{n}’th dim vector, using \textit{dim} as described in the Array documentation. If \textit{units} and/or \textit{name} are passed, sets these values for the \textit{n}’th dim vector.

Accepts:
\textit{n} (int): specifies which dim vector \textit{dim} (list or array): length must be either 2, or equal to the length of the \textit{n}’th axis of the data array
\textit{units} (Optional, str): \textit{name} (Optional, str):

\textit{get\_dim\_units}(\textit{n})
Return the \textit{n}’th dim vector units

\textit{set\_dim\_units}(\textit{n}: int, \textit{units}: str)
Sets the \textit{n}’th dim vector units to \textit{units}.

Accepts:
\textit{n} (int): specifies which dim vector units (str): new units

\textit{get\_dim\_name}(\textit{n})
Get the \textit{n}’th dim vector name

\textit{set\_dim\_name}(\textit{n}: int, \textit{name}: str)
Sets the \textit{n}’th dim vector name to \textit{name}. 

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

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

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 a branch from one tree, starting at this node, onto another tree at target node.
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) the new tree’s root node

static log_new_node(method)
Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside new_node.metadata called _fn_call_* , where * is the name of the decorated method, which stores the args/kwags/params passed to the generating 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 -

```python
>>> .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

```python
>>> .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 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

### BraggVectors

**class** py4DSTEM.BraggVectors*(Rshape, Qshape, name='braggvectors')*

Stores bragg scattering information for a 4D datacube.

```python
>>> braggvectors = BraggVectors( datacube.Rshape, datacube.Qshape )
```

initializes an instance of the appropriate shape for a DataCube `datacube`.

```python
>>> braggvectors.vectors[rx,ry]
>>> braggvectors.vectors_uncal[rx,ry]
```

or their aliases

```python
>>> braggvectors.v[rx,ry]
>>> braggvectors.v_uncal[rx,ry]
```

retrieve the calibrated and uncalibrated bragg vectors at scan position [rx,ry], and

```python
>>> braggvectors.v[rx,ry]['qx']
>>> braggvectors.v[rx,ry]['qy']
>>> braggvectors.v[rx,ry]['intensity']
```
retrieve the position and intensity of the scattering.

```python
__init__(Rshape, Qshape, name='braggvectors')
```

```python
to_h5(group)
```
Constructs the group, adds the bragg vector pointlists, and adds metadata describing the shape

```python
add_indices_to_braggpeaks(maxPeakSpacing, mask=None, returncalc=False)
```
Using the peak positions (qx,qy) and indices (h,k) in the PointList lattice, identify the indices for each peak in the PointListArray braggpeaks. Return a new braggpeaks_indexed PointListArray, containing a copy of the braggpeaks plus three additional data columns – ‘h’, ‘k’, and ‘index_mask’ – specifying the peak indices with the ints (h,k) and indicating whether the peak was successfully indexed or not with the bool index_mask. If mask is specified, only the locations where mask is True are indexed.

Parameters

- **maxPeakSpacing** *(float)* – Maximum distance from the ideal lattice points to include a peak for indexing
- **qx_shift** *(number)* – the shift of the origin in the lattice PointList relative to the braggpeaks PointListArray
- **qy_shift** *(number)* – the shift of the origin in the lattice PointList relative to the braggpeaks PointListArray
- **mask** *(bool)* – Boolean mask, same shape as the pointlistarray, indicating which locations should be indexed. This can be used to index different regions of the scan with different lattices

```python
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().

```python
calibrate(use_fitted_origin=True, returncalc=False)
```
Determines which calibrations are present in set.calibrations (of origin, elliptical, pixel, rotational), and applies any it finds to self.v_uncal, storing the output in self.v.

Parameters

- **use_fitted_origin** *(bool)* – determine if using fitted origin or measured origin

Returns

(PointListArray)

```python
choose_lattice_vectors(index_g0, index_g1, index_g2, mode='centered', plot=True,
subpixel='multicorr', upsample_factor=16, sigma=0, minAbsoluteIntensity=0,
minRelativeIntensity=0, relativeToPeak=0, minSpacing=0, edgeBoundary=1,
maxNumPeaks=10, bvm_vis_params={}, returncalc=False)
```
Choose which lattice vectors to use for strain mapping.

Parameters

- **index_g0** *(int)* – origin
- **index_g1** *(int)* – second point of vector 1
- **index_g2** *(int)* – second point of vector 2
- **mode** *(str)* – centered or raw bragg map
- **plot** *(bool)* – plot bragg vector maps and vectors
• **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 resolution 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

**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_lattice_vectors_all_DPs** *(returncalc=False)*

Fits lattice vectors g1, g2 to each diffraction pattern in braggpeaks, given some known (h,k) indexing.

**fit_origin** *(mask=None, fitfunction=’plane’, robust=False, robust_steps=3, robust_thresh=2, mask_check_data=True, plot=True, plot_range=None, fit_vis_params=None, 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)

**Returns**

Return value depends on `returnfit`. If `returnfit=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)

**class method 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_bvm** *(mode='centered')*

Gets a Bragg vector map, a 2D histogram of Bragg scattering vectors.

**Parameters**

• **Qshape** *(2 tuple)* – diffraction space shape

• **mode** *(str)* – must be ‘raw’ or ‘centered’. TODO, sampling selection

• **array** *(str)* –

**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)*

Removes all bragg peaks which fall inside `mask` in the raw (uncalibrated) positions.

**Parameters**

• **mask** *(bool)* – binary image where peaks will be deleted

• **update_inplace** *(bool)* – if True, removes peaks from this BraggVectors instance. If False, returns a new BraggVectors instance with the requested peaks removed

• **returncalc** *(bool)* – if True, return the BraggVectors

**Returns**

• **(BraggVectors or None)**
get_rotated_strain_map(\texttt{mode}, \texttt{g\_reference=None}, \texttt{returncalc=True}, \texttt{flip\_theta=False})

Starting from a strain map defined with respect to the \textit{xy} coordinate system of diffraction space, i.e. where \textit{exx} and \textit{eyy} are the compression/tension along the \textit{Qx} and \textit{Qy} directions, respectively, get a strain map defined with respect to some other right-handed coordinate system, in which the \textit{x-axis} is oriented along \textit{(xaxis\_x, xaxis\_y)}.

Parameters

\texttt{g\_reference} (\texttt{tuple}) – reference coordinate system for \textit{xaxis\_x} and \textit{xaxis\_y}

get_strain_from_reference_g1g2(\texttt{mask}, \texttt{returncalc=False})

Gets a strain map from the reference lattice vectors \textit{g1,g2} and lattice vector map \textit{g1g2\_map}.

Parameters

\texttt{mask} (\texttt{ndarray of bools}) – use lattice vectors from \textit{g1g2\_map} scan positions wherever \texttt{mask==True}

get_strain_from_reference_region(\texttt{mask}, \texttt{returncalc=False})

Gets a strain map from the reference region of real space specified by \texttt{mask} and the lattice vector map \textit{g1g2\_map}.

Parameters

\texttt{mask} (\texttt{ndarray of bools}) – use lattice vectors from \textit{g1g2\_map} scan positions wherever \texttt{mask==True}

graft(\texttt{node}, \texttt{merge\_metadata=True})

Moves a branch from one tree, starting at this node, onto another tree at target \texttt{node}.

Accepts:

\texttt{node} (\texttt{Node}): \texttt{merge\_metadata} (\texttt{True}, \texttt{False}, or \texttt{‘copy’}): if \texttt{True} adds the old root’s metadata to the new root; if \texttt{False} adds no metadata to the new root; if \texttt{‘copy’} adds copies of all metadata from the old root to the new root.

Returns

(\texttt{Node}) the new tree’s root node

index_bragg_directions(\texttt{x0=None}, \texttt{y0=None}, \texttt{plot=True}, \texttt{bvm\_vis\_params={}}, \texttt{returncalc=False})

From an origin \texttt{(x0,y0)}, a set of reciprocal lattice vectors \textit{gx,gy}, and an pair of lattice vectors \textit{g1=(g1x,g1y)}, \textit{g2=(g2x,g2y)}, find the indices \texttt{(h,k)} of all the reciprocal lattice directions.

Parameters

\begin{itemize}
  \item \texttt{x0} (\texttt{float}) – \textit{x}-coord of origin
  \item \texttt{y0} (\texttt{float}) – \textit{y}-coord of origin
  \item \texttt{Plot} (\texttt{bool}) – plot results
\end{itemize}

static log_new_node(\texttt{method})

Node subclass methods which generate and return a new node may be decorated with \texttt{@log\_new\_node}. This method creates a new Metadata dict stored inside \texttt{new\_node\_metadata} called \texttt{\_fn\_call\_*}, where \texttt{*} is the name of the decorated method, which stores the args/kwargs/params passed to the generating method.

measure_origin(\texttt{mode}, \texttt{returncalc=True}, \texttt{**kwargs})

Valid \texttt{mode} arguments are “beamstop” and “no\_beamstop”. Use-cases and input arguments:

“\texttt{no\_beamstop}” - A set of \textbf{bragg peaks for data with no beamstop, and in which} the center beam is brightest throughout. No required kwargs, optional kwargs are any accepted by \texttt{process.calibration.origin, get\_origin\_from\_braggpeaks}. 

1.4. API
“beamstop” - A set of bragg peaks for data with a beamstop. Req’d kwargs are center_guess (2-tuple) and radii (2-tuple) specifying an annular region in which to search for conjugate pairs of Bragg peaks to use for calibrating. Optional kwargs are those accepted by process.calibration.origin.get_origin_beamstop_braggpeaks.

Args:
center_guess (2-tuple) radii (2-tuple)

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)

Usage -

```
>>> .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).

Calibration

class py4DSTEM.Calibration(name: str | None = 'calibration', datacube=None)
Stores calibration measurements.

Usage:

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

If the parameter has not been set, the getter methods return None. For parameters with multiple values, they’re returned as a tuple. If any of the multiple values can’t be found, a single None is returned instead. Some parameters may have distinct values for each scan position; these are stored as 2D arrays, and

```
>>> c.get_p()
```

will return the entire 2D array, while
>>> c.get_p(rx,ry)

will return the value of $p$ at position $rx,ry$.

The Calibration object is capable of automatically calling the `calibrate` method of any other py4DSTEM objects when certain calibrations are updated. The methods that trigger propagation of calibration information are tagged with the `@propagating_calibration` decorator. Use the `register_target` method to set up an object to recieve calls to `calibrate`

```python
__init__(name: str | None = 'calibration', datacube=None)
```

Parameters

- `name` (optional, str)

```python
set_Q_pixel_size(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_R_pixel_size(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_Q_pixel_units(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_R_pixel_units(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_origin(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_origin_meas(x)
```

Parameters

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

```python
set_probe_param(x)
```

Parameters

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

```python
set_ellipse(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_p_ellipse(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
set_QR_rotflip(**kwargs)
```

Update the parameters the caller wanted by calling the wrapped method, then loop through the list of targets and call their `calibrate` methods.

```python
copy(name=None)
```
**register_target**(new_target)
Register an object to receive calls to it *calibrate* method when certain calibrations get updated

**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)

**unregister_target**(target)
Unlink an object from receiving calls to *calibrate* when certain calibration values are changed

**from_h5**( )
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
classmethod `from_h5(group)`
Takes an h5py Group which is open in read mode. Confirms that 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 a branch from one tree, starting at this node, onto another tree at target `node`.

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) the new tree’s root node

`static log_new_node(method)`
Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside `new_node.metadata` called `fin_call_*`, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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, 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

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

Data

class py4DSTEM.Data
    __init__()

DataCube

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

Storage and processing methods for 4D-STEM datasets.

__init__ (data: ndarray, name: str | None = 'datacube', calibration: Calibration | None = None, slicelabels: bool | list | 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. ‘skip’ is for internal use for the reader

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

    Returns
        A new DataCube instance.

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.

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().

bin_Q (N)
    Bins the data in diffraction space by bin factor N

    Accepts:
        N (int): the binning factor

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

**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)

**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

**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 highest, 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
find_Bragg_disks(template, data=None, filter_function=None, corrPower=1, sigma=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 by cross correlation with template.

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

1. optional pre-processing by passing the image through some filter_function, which should accept and return 2D arrays

2. the 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 steps 3 and 4 performed directly on the diffraction image itself rather than a 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 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).

Running on a subset of the data may be desirable, and is controlled by the data parameter. If 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 shaped 2D boolean array: run on the diffraction images specified by the True counts and return a list of QPoints instances

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.
- filter_function (callable) – filtering function to apply to each diffraction pattern before peakfinding. 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 giving hybrid correlations.
- sigma (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

• **edgeBoundary** *(int)* – minimum acceptable distance for detected peaks from 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
(BraggVectors or QPoints or list of QPoints)

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_beamstop_mask(threshold=0.25, distance_edge=4.0, include_edges=True, 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 brightest 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.
- **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_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(method='max', mode=None, geometry=None, calibrated=False, shift_center=False, verbose=True, name='dp_max', returncalc=True)

Function to calculate maximum virtual diffraction. Default captures pattern across entire 4D-dataset.

Parameters

- **datacube** *(Datacube)* – datacube class object which stores 4D-dataset needed for calculation
- **mode** *(str)* – defines mode for selecting area in real space to use for virtual diffraction. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. Options:
  - ‘point’ uses singular point as detector
  - ‘circle’ or ‘circular’ uses round detector, like bright field
  - ‘annular’ or ‘annulus’ uses annular detector, like dark field
  - ‘mask’ flexible detector, any 2D array
- **geometry** *(variable)* – valid entries are determined by the mode, values in pixels argument, as follows. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. If mode is None the geometry will not be applied.
  - ‘point’: 2-tuple, (rx,ry),
    rx and ry are each single float or int to define center
‘circle’ or ‘circular’: nested 2-tuple, ((rx,ry),radius),
‘annular’ or ‘annulus’: nested 2-tuple, ((rx,ry),(radius_i,radius_o)),
‘rectangle’, ‘square’, ‘rectangular’: 4-tuple, (xmin,xmax,ymin,ymax)

mask: flexible detector, any boolean or floating point 2D array with the same shape as datacube.Rshape

• calibrated (bool) – if True, geometry is specified in units of ‘A’ instead of pixels.
The datacube’s calibrations must have its “R_pixel_units” parameter set to “A”. If mode is None the geometry and calibration will not be applied.

• shift_center (bool) – if True, the diffraction patterns are shifted to account for beam shift or the changing of the origin through the scan. The datacube’s calibration[‘origin’] parameter must be set. Only ‘max’ and ‘mean’ supported for this option.

• verbose (bool) – if True, show progress bar

Returns
the diffraction image

Return type
(VirtualDiffraction)

get_dp_mean(method=’mean’, mode=None, geometry=None, calibrated=False, shift_center=False, verbose=True, name=’dp_mean’, returncalc=True)

Function to calculate mean virtual diffraction. Default captures pattern across entire 4D-dataset.

Parameters

• datacube (Datacube) – datacube class object which stores 4D-dataset needed for calculation

• mode (str) – defines mode for selecting area in real space to use for virtual diffraction. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. Options:
  – ‘point’ uses singular point as detector
  – ‘circle’ or ‘circular’ uses round detector, like bright field
  – ‘annular’ or ‘annulus’ uses annular detector, like dark field
  – ‘mask’ flexible detector, any 2D array

• geometry (variable) – valid entries are determined by the mode, values in pixels argument, as follows. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. If mode is None the geometry will not be applied.
  – ‘point’: 2-tuple, (rx,ry),
    qx and qy are each single float or int to define center
  – ‘circle’ or ‘circular’: nested 2-tuple, ((rx,ry),radius),
    qx, qy and radius, are each single float or int
  – ‘annular’ or ‘annulus’: nested 2-tuple, ((rx,ry),(radius_i,radius_o)),
  – ‘rectangle’, ‘square’, ‘rectangular’: 4-tuple, (xmin,xmax,ymin,ymax)
  – mask: flexible detector, any boolean or floating point 2D array with the same shape as datacube.Rshape
**calibrated** *(bool)* – if True, geometry is specified in units of ‘A’ instead of pixels. The datacube’s calibrations must have its “\(R_{\text{pixel\_units}}\)" parameter set to “A”. If mode is None the geometry and calibration will not be applied.

**shift_center** *(bool)* – if True, the diffraction patterns are shifted to account for beam shift or the changing of the origin through the scan. The datacube’s calibration[‘origin’] parameter must be set. Only ‘max’ and ‘mean’ supported for this option.

**verbose** *(bool)* – if True, show progress bar

Returns

the diffraction image

Return type

(*VirtualDiffraction*)

get_dp_median(*method=’median’, mode=None, geometry=None, calibrated=False, shift_center=False, verbose=True, name=’dp_median’, returncalc=True*)

Function to calculate median virtual diffraction. Default captures pattern across entire 4D-dataset.

Parameters

- **datacube** (*Datacube*) – datacube class object which stores 4D-dataset needed for calculation
- **mode** (*str*) – defines mode for selecting area in real space to use for virtual diffraction. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. Options:
  - ‘point’ uses singular point as detector
  - ‘circle’ or ‘circular’ uses round detector, like bright field
  - ‘annular’ or ‘annulus’ uses annular detector, like dark field
  - ‘mask’ flexible detector, any 2D array
- **geometry** (*variable*) – valid entries are determined by the mode, values in pixels argument, as follows. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. If mode is None the geometry will not be applied.
  - ‘point’: 2-tuple, (rx,ry),
  - ‘circle’ or ‘circular’: nested 2-tuple, ((rx,ry),radius),
  - ‘annular’ or ‘annulus’: nested 2-tuple, ((rx,ry),(radius_i,radius_o)),
  - ‘rectangle’, ‘square’, ‘rectangular’: 4-tuple, (xmin,xmax,ymin,ymax)
  - mask: flexible detector, any boolean or floating point 2D array with the same shape as datacube.Rshape
- **calibrated** *(bool)* – if True, geometry is specified in units of ‘A’ instead of pixels. The datacube’s calibrations must have its “\(R_{\text{pixel\_units}}\)" parameter set to “A”. If mode is None the geometry and calibration will not be applied.
- **shift_center** *(bool)* – if True, the diffraction patterns are shifted to account for beam shift or the changing of the origin through the scan. The datacube’s calibration[‘origin’] parameter must be set. Only ‘max’ and ‘mean’ supported for this option.
- **verbose** *(bool)* – if True, show progress bar
Returns
the diffraction image

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_probe_size(dp=None, thresh_lower=0.01, thresh_upper=0.99, N=100, plot=True, 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)

get_vacuum_probe(ROI=None, name='probe', returncalc=True)

Computes a vacuum probe from the DataCube by aligning and averaging either all or some subset of the diffraction patterns.
Parameters

ROI (None or boolean array or tuple) – if None, uses the whole datacube. Otherwise, uses a subset of diffraction patterns. If ROI is a boolean array, it should be Rspace shaped, and diffraction patterns where True are used. Else should be a 4-tuple representing (Rxmin,Rxmax,Rymin,Rymax) of a rectangular region to use.

Returns

(Probe) a Probe instance

get_virtual_diffraction(method='max', mode=None, geometry=None, calibrated=False, shift_center=False, verbose=True, name='virtual_diffracton', returncalc=True)

Function to calculate virtual diffraction patterns

Parameters

• datacube (Datacube) – datacube class object which stores 4D-dataset needed for calculation

• method (str) – defines method used for diffraction pattern, options are ‘mean’, ‘median’, and ‘max’

• mode (str) – defines mode for selecting area in real space to use for virtual diffraction. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. Options:
  – ‘point’ uses singular point as detector
  – ‘circle’ or ‘circular’ uses round detector, like bright field
  – ‘annular’ or ‘annulus’ uses annular detector, like dark field
  – ‘mask’ flexible detector, any 2D array

• geometry (variable) – valid entries are determined by the mode, values in pixels argument, as follows. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. If mode is None the geometry will not be applied.
  – ‘point’: 2-tuple, (rx,ry), ints
  – ‘circle’ or ‘circular’: nested 2-tuple, ((rx,ry),radius),
  – ‘annular’ or ‘annulus’: nested 2-tuple, ((rx,ry),(radius_i,radius_o))
  – ‘rectangle’, ‘square’, ‘rectangular’: 4-tuple, (rxmin,rxmax,rymin,rymax)
  – mask: flexible detector, any boolean or floating point 2D array with the same shape as datacube.Rshape

• calibrated (bool) – if True, geometry is specified in units of ‘A’ instead of pixels. The datacube’s calibrations must have its “R_pixel_units” parameter set to “A”. If mode is None the geometry and calibration will not be applied.

• shift_center (bool) – if True, the diffraction patterns are shifted to account for beam shift or the changing of the origin through the scan. The datacube’s calibration[‘origin’] parameter must be set. Only ‘max’ and ‘mean’ supported for this option.

• verbose (bool) – if True, show progress bar

Returns

the diffraction image
Return type

*(VirtualDiffraction)*

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

Get a virtual image and store it in *datacube’s tree under ‘name’*. The kind of virtual image is specified by the *mode* argument.

Parameters

- **mode** *(str)* – defines geometry mode for calculating virtual image options:
  - ‘point’ uses singular point as detector
  - ‘circle’ or ‘circular’ uses round detector, like bright field
  - ‘annular’ or ‘annulus’ uses annular detector, like dark field
  - ‘mask’ flexible detector, any 2D array

- **geometry** *(variable)* – valid entries are determined by the *mode*, values in

- **argument** *(pixels)* –
  - ‘point’: 2-tuple, (qx,qy), ints
  - ‘circle’ or ‘circular’: nested 2-tuple, ((qx,qy),radius),
  - ‘annular’ or ‘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*

- **follows** *(as)* –
  - ‘point’: 2-tuple, (qx,qy), ints
  - ‘circle’ or ‘circular’: nested 2-tuple, ((qx,qy),radius),
  - ‘annular’ or ‘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 ‘Å^-1’ instead of pixels. The datacube’s calibrations must have its “*Q_pixel_units*” parameter set to “Å^-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.

- **verbose** *(bool)* – if True, show progress bar

- **dask** *(bool)* – if True, use dask arrays
• **return_mask** *(bool)* – if False (default) returns a virtual image as usual. If True, does not generate or return a virtual image, instead returning the mask that would be used in virtual image computation for any call to this function where `shift_center = False`. Otherwise, must be a 2-tuple of integers corresponding to a scan position (rx,ry); in this case, returns the mask that would be used for virtual image computation at this scan position with `shift_center` set to True. Setting `return_mask` to True does not add anything to the datacube’s tree.

• **name** *(str)* – the output object’s name

• **return_calc** *(bool)* – if True, returns the output

**Returns**

if `return_calc` is True, returns the VirtualImage

**Return type**

(Optional)

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

Moves a branch from one tree, starting at this node, onto another tree at target `node`.

**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) the new tree’s root node

**static log_new_node**(method)

Node subclass methods which generate and return a new node may be decorated with `@log_new_node`. This method creates a new Metadata dict stored inside `new_node.metadata` called `_fn_call_*`, where * is the name of the decorated method, which stores the args/kwags/params passed to the generating method.

**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

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

Display a diffraction space image with an overlaid mask representing a virtual detector.

**Parameters**

• **mode** – see py4DSTEM.process.get_virtual_image

• **geometry** – see py4DSTEM.process.get_virtual_image

• **scan_position** – if None, positions the unshifted detector over the mean or max diffraction pattern. Otherwise, must be a tuple (rx,ry) of ints, and a detector is positioned over the diffraction pattern at this position, including shifts if they would be applied for this dataset (i.e. if it contains the appropriate calibrations)

• **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.

• **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.

• **invert** *(bool)* – if True, invert the display mask

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

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)

**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

**set_scan_shape**(Rshape)

Reshape the data given the real space scan shape.

**Accepts:**

- Rshape (2-tuple)

**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.

**swap_Qxy()**

Swaps the diffraction space x and y coordinates.

**swap_RQ()**

Swaps the first and last two dimensions of the 4D datacube.
**swap_Rxy()**
Swaps the real space x and y coordinates.

**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

**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 -

```python
>>> .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

```python
>>> .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)**

Stores a diffraction-space shaped 2D data array.

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

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

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 a branch from one tree, starting at this node, onto another tree at target node.

   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) the new tree’s root node

static log_new_node(method)
Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside new_node.metadata called _fn_call_*, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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

(continues on next page)
The show, add, and get methods can be accessed directly with

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

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

### Metadata

**class** py4DSTEM.Metadata

```python
class 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:**

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

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

**__init__**

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

**Parameters**

- `name` *(Optional, string)*
- `copy` *(name=None)*

**to_h5**

```python
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**

```python
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 built. Nodes themselves contain the machinery for managing a tree hierarchy 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

```python
>>> 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

```python
>>> 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

```python
>>> 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:

```plaintext
>>> .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
```

(continues on next page)
The show, add, and get methods can be accessed directly with

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

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

**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:

```python
>>> 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:

```python
>>> 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

```python
__init__(name: str | None = 'node')
```

```python
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.

```python
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().

```python
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 a branch from one tree, starting at this node, onto another tree at target *node*.

**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) the new 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, 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).

**static log_new_node** (*method*)

Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside *new_node.metadata* called _fn_call_* where * is the name of the decorated method, which stores the args/kwarggs/params passed to the generating method.
classmethod from_h5(group)
    Takes an h5py Group which is open in read mode. Confirms that 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

**classmethod from_h5**(group)

Takes an h5py Group which is open in read mode. Confirms that 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 a branch from one tree, starting at this node, onto another tree at target *node*.

**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) the new tree’s root node
**static log_new_node** (*method*)

Node subclass methods which generate and return a new node may be decorated with `@log_new_node`. This method creates a new Metadata dict stored inside `new_node.metadata` called `_fn_call_*`, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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.

**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(grain='copy')        # remove/return a branch, copy root metadata
>>> .tree(grain=(node, True))  # as above
>>> .tree(grain=(node, False)) # as above, discard root metadata
>>> .tree(grain=(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).

**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

**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 a branch from one tree, starting at this node, onto another tree at target node.
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) the new tree’s root node

static log_new_node(method)
Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside new_node.metadata called _fn_call_* , where * is the name of the decorated method, which stores the args/kwgs/params passed to the generating 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 -

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

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.

**__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)

**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

**dim**(n)

Return the n’th dim vector

**classmethod from_h5**(group)

Takes an h5py Group which is open in read mode. Confirms that 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.

get_kernel(mode=‘flat’, returncalc=True, **kwargs)
Creates a kernel from the probe for cross-correlative template matching.
Precise behavior and valid keyword arguments depend on the mode selected. In each case, the center of the probe is shifted to the origin and the kernel normalized such that it sums to 1. In ‘flat’ mode, this is the only processing performed. In the remaining modes, some additional processing is performed which adds a ring of negative intensity around the central probe, which results in edge-filtering-like behavior during cross correlation. Valid modes, and the required additional kwargs, if any, for each, are:

• ‘flat’: creates a flat probe kernel. For bullseye or other structured probes, this mode is recommended. No required arguments, optional arg origin (2-tuple)

• ‘gaussian’: subtracts a gaussian with a width of standard deviation sigma, which is a required argument. Optional arg origin.

• ‘sigmoid’: subtracts an annulus with inner and outer radii of (ri,ro) and a sine-squared sigmoid radial profile from the probe template. Required arg: radii (2 tuple). Optional args origin (2-tuple)

• ‘sigmoid_log’: subtracts an annulus with inner and outer radii of (ri,ro) and a logistic sigmoid radial profile from the probe template. Required arg: radii (2 tuple). Optional args origin (2-tuple)

Returns
(2D array)

graft(node, merge_metadata=True)
Moves a branch from one tree, starting at this node, onto another tree at target node.

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) the new tree’s root node

static log_new_node(method)
Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside new_node.metadata called _fn_call_* , where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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 -

```python
>>> .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(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

```python
>>> .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().

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
classmethod `from_h5(group)`
Takes an h5py Group which is open in read mode. Confirms that 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 a branch from one tree, starting at this node, onto another tree at target `node`.

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)` the new tree’s root node

`static log_new_node(method)`
Node subclass methods which generate and return a new node may be decorated with `@log_new_node`. This method creates a new Metadata dict stored inside `new_node.metadata` called `_fn_call_*`, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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, 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

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

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

**RealSlice**

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

    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
    real slices

    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().

    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

**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 a branch from one tree, starting at this node, onto another tree at target node.

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) the new tree’s root node

**static log_new_node**(method)
Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside new_node.metadata called _fn_call_, where * is the name of the decorated method, which stores the args/kwags/params passed to the generating 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 -

```python
>>> .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

```python
>>> .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 data-cube.

**__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()`.

**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

**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 a branch from one tree, starting at this node, onto another tree at target node.

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) the new tree’s root node

static log_new_node(method)

Node subclass methods which generate and return a new node may be decorated with @log_new_node. This method creates a new Metadata dict stored inside new_node.metadata called _fn_call_*, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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 -

```python
>>> .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

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

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

### VirtualImage

**class** `py4DSTEM.VirtualImage`(*data: ndarray, name: str | None = 'virtualimage')`

Stores a real-space shaped 2D image with 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

**Returns**

A new VirtualImage 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()`.

**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
\textbf{dim}(n)
Return the n'th dim vector

\textbf{classmethod from_h5}(\textit{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:
\textit{group} (h5py Group)

\textbf{Returns}
(Node)

\textbf{get_dim}(n)
Return the n'th dim vector

\textbf{get_dim_name}(n)
Get the n'th dim vector name

\textbf{get_dim_units}(n)
Return the n'th dim vector units

\textbf{get_from_tree}(\textit{name})
Finds and returns an object from an EMD tree using the string key \textit{name}, with ‘/’ delimiters between ‘parent/child’ nodes. Search from the root node by adding a leading ‘/’; otherwise, searches from the current node.

\textbf{graft}(\textit{node}, merge\_metadata=True)
Moves a branch from one tree, starting at this node, onto another tree at target \textit{node}.
Accepts:
\textit{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.

\textbf{Returns}
(Node) the new tree’s root node

\textbf{static log_new_node}(\textit{method})
Node subclass methods which generate and return a new node may be decorated with \texttt{@log\_new\_node}. This method creates a new Metadata dict stored inside \texttt{new\_node.metadata} called \texttt{\_fn\_call\_*}, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating method.

\textbf{set_dim}(n: int, \textit{dim: list | ndarray}, units: str | None = None, name: str | None = None)
Sets the n'th dim vector, using \textit{dim} as described in the Array documentation. If \textit{units} and/or \textit{name} are passed, sets these values for the n’th dim vector.
Accepts:
\textit{n} (int): specifies which dim vector \textit{dim} (list or array): length must be either 2, or equal to the length of the n’th axis of the data array
\textit{units} (Optional, str): name: (Optional, str):

\textbf{set_dim_name}(n: int, \textit{name: str})
Sets the n’th dim vector name to \textit{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 -

```python
>>> .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

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

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

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)

```python
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.

```python
__init__(filepath, sync_block_IDs=True, hidden_stripe_noise_reduction=True)
```

```python
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.

```python
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: memmnap 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.download_file_from_google_drive(id_, destination, overwrite=False)

Downloads a file or collection of files from google drive to the destination file path.

Parameters

• id (str) – File ID for the desired file. May be: - the file id, i.e. for 
  https://drive.google.com/file/d/1bHv3u61Cr-y_GkdWhrJGh1lw2VKmt3UM/
  id='1bHv3u61Cr-y_GkdWhrJGh1lw2VKmt3UM'
  – the complete URL,
  – a special string denoting a sample dataset or collection of datasets. For a list of sample datasets and their keys, run get_sample_data_ids().

• destination (str or Path) – path file will be downloaded to. For collections of files, this should point to an existing directory; a subdirectory will be created inside this directory whose name will be given by id_, and the collection of files will be placed inside that subdirectory. If a subdirectory of this name already exists, aborts or deletes and overwrite the entire subdirectory, depending on the value of overwrite.

• overwrite (bool) – turn overwrite protection on/off
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.
- ****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) formatted HDF5 files.

Different file versions Precise behavior is determined 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)** – Strictly, 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) formatted 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 wraparoud errors.

Parameters
• **datacube** *(DataCube)* – data to background subtract
• **N_frames** *(int)* – number of random diffraction patterns to use
• **width** *(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**

```python
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(gaussian 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 binning 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

```python
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.

```python
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_Nsigma = mean(gaussian fit) + thresh_lower * std(gaussian fit)

For more info, see the electron_count docstring.

Parameters

• **datacube** – a 4D numpy.ndarray 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(gaussian 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

```python
>>> swap_Qxy(datacube).data.shape
(Rx, Ry, Qx, Qy)
```

`py4DSTEM.preprocess.preprocess.bin_data_diffraction(datacube, bin_factor)`
Performs diffraction space binning of data by bin_factor.

`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. 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 highest, 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)` – py4DSTEM 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 (bool): (optional) the bad pixel mask

`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')

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. If not None, 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 savgol filter smoothing is applied
- **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**

A 2- or 3-tuple of ndarrays:

- **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)

```python
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 elliptical 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. If not None, the center point in `p_ellipse` is ignored, and this argument is used as (qx0,qy0) instead.

**Returns**
(ndarray) 2D polar elliptical median background image

**utils**

```python
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

```python
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.

```python
get_shifted_ar(ar, xshift, yshift, periodic=True, bilinear=False)
```

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** – flag for bilinear image shifts

```python
def 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

```python
@functools.wraps(get_maxima_2D)
def 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

1.4. API
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**

`py4DSTEM.process.calibration.bragg_vectors.calibrate(braggpeaks, calibration, use_fitted_origin=True, **params)`

Determines which calibrations are present in `calibrations` (of origin, elliptical, pixel, rotational), and applies any it finds to `braggpeaks`.

Note that this function updates the data pointed to by whatever PointListArray that’s passed to it, so consider copying the PLA before running this function!

**Parameters**
- `braggpeaks` (PointListArray) –
- `calibration` (Calibration) –
- `use_fitted_origin` (bool) – determine if using fitted origin or measured origin

**Returns**
(PointListArray)
py4DSTEM.process.calibration.braggvectors.center_braggpeaks(braggpeaks, origin)
Shift the braggpeaks positions to center them about the origin.

Accepts:
- braggpeaks (PointListArray): the unshifted peak positions origin (2-tuple): (qx0,qy0) either as scalars or as (R_Nx,R_Ny).

Returns
- the centered Bragg peaks

Return type
(PointListArray)

py4DSTEM.process.calibration.braggvectors.correct_braggpeak_elliptical_distortions(braggpeaks, p_ellipse, centered=True)
Correct the elliptical distortions in a BraggPeaks instance.

Accepts:
- braggpeaks (PointListArray): the detected, unshifted bragg peaks p_ellipse (5-tuple): the ellipse parameters (x0,y0,a,b,theta) centered (bool): if True, assumes that the braggpeaks PointListArray has been centered, and uses (x0,y0)=(0,0). Otherwise, uses the (x0,y0) from p_ellipse

Returns
- the corrected Bragg peaks

Return type
(PointListArray)

py4DSTEM.process.calibration.braggvectors.calibrate_Bragg_peaks_pixel_size(braggpeaks: PointListArray, q_pixel_size: Number)
Calibrate the reciprocal length of Bragg peak positions.

Accepts:
- braggpeaks (PointListArray) the detected, unscaled bragg peaks q_pixel_size (float) Q pixel size in inverse Ångström

Returns
(PointListArray)

py4DSTEM.process.calibration.braggvectors.calibrate_bragg_peaks_rotation(braggpeaks: PointListArray, theta: float, flip: bool) → PointListArray
Calibrate rotation of Bragg peak positions, using either the R/Q rotation theta or the QR_rotation value inside a Calibration object.

Accepts:
- braggpeaks (PointListArray) the CENTERED Bragg peaks theta (float) the rotation between real and reciprocal space in radians flip (bool) whether there is a flip between real and reciprocal space
Returns

braggpeaks_rotated (PointListArray) the rotated Bragg peaks

Functions related to elliptical calibration, such as fitting elliptical distortions.

The user-facing representation of ellipses is in terms of the following:

- x0: the center of the ellipse
- 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

More details about the elliptical parameterization used can be found in the module docstring for the
module.

**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)

**Notes**

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.

**Examples**

```python
# Example usage
```

**References**

This function is based on the work by [Smith et al., 2020].

**License**

This code is licensed under the Apache License, Version 2.0.
\[ f(x,y; I_0,I_1,\sigma_0,\sigma_1,\sigma_2,c_{\text{bkgd}},x_0,y_0,A,B,C) = \]
\[ \text{Norm}(r; I_0,\sigma_0,0) + \]
\[ \text{Norm}(r; I_1,\sigma_1,R) \cdot \text{Theta}(r-R) \]
\[ \text{Norm}(r; I_1,\sigma_2,R) \cdot \text{Theta}(R-r) + c_{\text{bkgd}} \]

where

- \((x,y)\) are cartesian coordinates,
- \(r\) is the radial coordinate,
- \((I_0,I_1,\sigma_0,\sigma_1,\sigma_2,c_{\text{bkgd}},x_0,y_0,R,B,C)\) are parameters,
- \(\text{Norm}(x;I,s,u)\) is a gaussian in the variable \(x\) with maximum amplitude \(I\), standard deviation \(s\), and mean \(u\)
- \(\text{Theta}(x)\) 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

- \(I_0\): the intensity of the first gaussian function
- \(I_1\): the intensity of the Janus gaussian
- \(\sigma_0\): std of first gaussian
- \(\sigma_1\): inner std of Janus gaussian
- \(\sigma_2\): outer std of Janus gaussian
- \(c_{\text{bkgd}}\): a constant offset
- \(x_0,y_0\): 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 \((x_0,y_0)\)
- **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, \(p0 = (I_0,I_1,\sigma_0,\sigma_1,\sigma_2,c_{\text{bkgd}},x_0,y_0,A,B,C)\). Note that \(x_0,y_0\) are redundant; their guess values are the \(x_0,y_0\) 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:

- \( x_0 \): x center
- \( y_0 \): 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 \(\phi_{\text{known}}\) between a pair of the diffraction peaks after performing elliptical distortion correction.

Note that there are two possible angles which \(\phi_{\text{known}}\) might refer to, because the angle of interest is well defined up to a complementary angle. This function is written such that \(\phi_{\text{known}}\) should be the smaller of these two angles.

Parameters

- \(\text{data} \) (\(\text{ndarray}\)) –
- \(\text{p}_\text{ellipse} \) (5-tuple) – the ellipse parameters \((x_0,y_0,a,b,\theta)\)
- \(\text{r}_\text{inner} \) (\(\text{float}\)) – the fitting annulus inner radius
- \(\text{r}_\text{outer} \) (\(\text{float}\)) – the fitting annulus outer radius
- \(\phi_{\text{known}} \) (\(\text{float}\)) – the known angle between a pair of diffraction peaks, in radians
- \(\text{fitrad} \) (\(\text{float}\)) – the region about the fixed data point used to refine its position

Returns

A 2-tuple containing:

- \(\text{a}_{\text{constrained}} \) (\(\text{float}\)) the first semiaxis of the selected ellipse
- \(\text{b}_{\text{constrained}} \) (\(\text{float}\)) the second semiaxis of the selected ellipse

Return type

(2-tuple)
Options for the `mode` argument, their uses-cases, and their expected additional input arguments are:

- **“dc_no_beamstop”** - A datacube with no beamstop, and in which the center beam is brightest throughout.
  
  Args:
  
  data (DataCube)

- **“bragg_no_beamstop”** - A set of bragg peaks for data with no beamstop, and in which the center beam is brightest throughout.
  
  Args:
  
  data (PointListArray) Q_shape (Qx, Qy) from braggvector

- **“dc_beamstop”** - A datacube with a beamstop

  Args:
  
  data (DataCube) mask (2d array)

- **“bragg_beamstop”** - A set of bragg peaks for data with a beamstop

  Args:
  
  data (PointListArray) center_guess (2-tuple) radii (2-tuple) Q_Nx (int) Q_Ny (int)

Returns

(3 real space shaped arrays) qx0, qy0, mask

### 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)

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 \( qx0,qy0 \)

**Returns**
the origin, \((x,y)\) at each scan position

**Return type**
(2-tuple of \((R_{Nx},R_{Ny})\)-shaped ndarrays)

```python
py4DSTEM.process.calibration.origin.get_origin_from_braggpeaks(braggpeaks, Q_shape, center_guess=None, score_method=None, findcenter='CoM', bvm=None, **kwargs)
```

Gets the diffraction shifts using detected Bragg disk positions.

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 an 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**

- **braggpeaks** *(PointListArray)* – the Bragg peak positions
- **Q_shape** *(tuple of ints)* – the shape of diffraction space
- **center_guess** *(tuple of ints)* – initial guess for the center
- **score_method** *(string)* – 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
- **findcenter** *(str)* – specifies the method for determining the unscattered beam position options: ‘CoM’, or ‘max.’ Only used if center_guess is None. CoM (default) finds the center of mass of bragg vector map. ‘max’ uses the maximum value in the bragg vector map.
- **bvm** *(array or None)* – the bragg vector map. If None (default), the bvm is calculated

**Returns**
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.

**Return type**
(3-tuple)
Find the origin for a single diffraction pattern, assuming there is a beam stop.

**Parameters**

- **DP** *(np array)* – diffraction pattern
- **mask** *(np array)* – 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

**Returns**

qx0, qy0 (tuple) measured center position of diffraction pattern

Find the origin for each diffraction pattern, assuming there is a beam stop.

**Parameters**

- **datacube** *(DataCube)* –
- **mask** *(np array)* – 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

**Returns**

qx0, qy0 (tuple of np arrays) measured center position of each diffraction pattern

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**

- **braggpeaks** *(PointListArray)* –
- **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)

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 retrieved 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.find_outlier_shifts(xshifts, yshifts, n_sigma=10, edge_boundary=0)

Finds outliers in the shift matrices.

Gets a score function for each scan position Rx,Ry, given by the sum of the absolute values of the difference between the shifts at this position and all 8 NNs. Calculates a histogram of the scoring function, fits a gaussian to its initial peak, and sets a cutoff value to *n_sigma* times its standard deviation. Values beyond this cutoff are deemed outliers, as are scan positions within *edge_boundary* pixels of the edge of real space.

Accepts:

- **xshifts** (*R_Nx,R_Ny)-shaped array*) the shifts in x
- **yshifts** (*R_Nx,R_Ny)-shaped array*) the shifts in y
- **n_sigma** (*float*) the cutoff value for the score function, in number of std
- **edge_boundary** (*int*) number of pixels near the mask edge to mark as outliers

Returns:

- **mask** (*R_nx,R_ny)-shaped array of bools*) the outlier mask score
- **score** (*float*) the score cutoff value

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 $r(\text{thresh})$ 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, θ to 1*) – the lower limit of threshold values
- **thresh_upper** (*float, θ 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 $q_{\text{known}}$

**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.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=True, 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 constraint 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() docstring 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 \((N_{\text{feat}}, N_c)\), 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 \(N_{\text{feat}}\), 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 \((N_c, N_{\text{meas}})\). 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 coefficient 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 \(N_{\text{meas}}\), 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 \(\text{max\_iterations}\) NMF steps.

**Parameters**

- **max_iterations** (int) – the maximum number of NMF steps to take

**split**

- **split** (\(\text{sigma}=2\), \(\text{threshold\_split}=0.25\), \(\text{expand\_mask}=1\), \(\text{minimum\_pixels}=1\))

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 \(\text{sigma}\). This is then turned into a binary mask, by thresholding with \(\text{threshold\_split}\). Stray pixels are eliminated by performing a one pixel binary closing, then binary opening. The mask is then expanded by \(\text{expand\_mask}\) 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**

- **\(\text{sigma}\)** (float) – std of gaussian kernel used to smooth the class images before thresholding and splitting.

- **\(\text{threshold\_split}\)** (float) – used to threshold the class image to create a binary mask.

- **\(\text{expand\_mask}\)** (int) – number of pixels by which to expand the mask before separating into contiguous regions.

- **\(\text{minimum\_pixels}\)** (int) – if, after splitting, a potential new class contains fewer than this number of pixels, ignore it

**merge**

- **merge** (\(\text{threshBPs}=0.1\), \(\text{threshScanPosition}=0.1\), \(\text{return\_params}=\text{True}\))

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 \(\text{threshBPs}\) 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 merged 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

• 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)

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)

```python
py4DSTEM.process.classification.classutils.get_class_DP_without_Bragg_scattering(datacube, class_image, bragg_peaks, 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 bragg_peaks in each diffraction pattern before adding to the average image. Importantly, bragg_peaks 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
- **bragg_peaks** *(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, 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 initializing 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**

```python
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)
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, 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 = 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 in 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

- **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_number_peaks=3, inversion_symmetry=True, multiple_corr_reset=True, progress_bar: bool = True, return_orientation: bool = True)*

This function computes the orientation of any number of PointLists stored in a PointListArray, and returns an OrientationMap.

**match_single_pattern** *(bragg_peaks: PointList, num_matches_return: int = 1, 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_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
• **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 class containing all outputs fig, ax (handles): Figure handles for the plotting output

**Return type**

orientation *(Orientation)*
**calculate_strain**

```python
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,
    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 positions.

TODO: add robust fitting?

**Parameters**

- **bragg_peaks_array** (*PointListArray*) – All Bragg peaks
- **orientation_map** (*OrientationMap*) – Orientation map generated from ACOM
- **corr_kernel_size** (*float*) – Correlation kernel size - if user does not specify, uses self.corr_kernel_size.
- **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
- **min_num_peaks** (*int*) – Minimum number of peaks required.
- **rotation_range** (*float*) – Maximum rotation range in radians (for symmetry reduction).
- **progress_bar** (*bool*) – Show progress bar
- **mask_from_corr** (*bool*) – Use ACOM correlation signal for mask
- **corr_range** (*np.ndarray*) – Range of correlation signals for mask
- **corr_normalize** (*bool*) – Normalize correlation signal before masking

**Returns**

strain tensor

**Return type**

*strain_map* (*RealSlice*)

**symmetry_reduce_directions**

```python
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.

**Parameters**

- **file_name** (*str*) – Path to save .ang file.

**save_ang_file**

```python
save_ang_file(file_name, orientation_map, ind_orientation=0, pixel_size=1.0, pixel_units='px',
    transpose_xy=True, flip_x=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 unit 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 | None = None*) – 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*) – Select either perspective (true) or orthogonal (false) 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 = False, 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 = (12, 6), 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** (float) – 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.
• float) (bragg_k_broadening) – Broadening applied to bragg_peaks.
• figsize (list, tuple, np.ndarray) – Figure size for plot.
• returnfig (bool) – Return figure and axes handles if this is True.

Returns
fig, ax (optional) figure and axes handles

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, 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.
- `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 orientation 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 progress bar when calculating orientation images.

**Returns**

RGB images `fig, axs` (handles): Figure and axes handles 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,
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 orientation 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 handles 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.

---

**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=True, set_calibration=True, 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.

• **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
Optional figure and axis handles, if returnfig=True.

Return type
fig, ax (handles)

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 | ndarrays = (12.0, 6.0), 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 each other, 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
**generate_CBED**

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 \(1/\AA\).
• **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.

:return Probe (bool)* – If True, the probe (np.ndarray) will be returned in addition 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, verbose=True)*

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-absorptive”: Lobato factors plus an imaginary part equal to 0.1f, 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)

• \(k_{\text{max}}\) (float) – max scattering length to compute structure factors to. Setting this to 2x the \(k_{\text{max}}\) used in generating the beams in included in a simulation will retain all possible couplings

• \(\text{thermal}_\text{sigma}\) (float or dict{int->float}) – RMS atomic displacement 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 \(\langle u \rangle\) in papers) To specify different \(\langle u \rangle\) for each element, pass a dictionary with Z as the key, mapping to the appropriate float value

• \(\text{tol}_\text{structure}_\text{factor}\) (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)

• \(\text{recompute}_\text{kinematic}_\text{structure}_\text{factors}\) (bool) – When True, recomputes the kinematic structure factors using the same \(\text{tol}_\text{structure}_\text{factor}\), and with \(k_{\text{max}}\) set to half the \(k_{\text{max}}\) 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.)

• \(g\_\text{vec}_\text{precision}\) (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

• \(\text{factors}\) (See \text{WK}_\text{scattering}_\text{factors}.py for details on the Weickenmeier-Kohl form)

_init__(\text{positions, numbers, cell})

Parameters

• \text{positions} (np.array) – fractional coordinates of each atom in the cell

• \text{numbers} (np.array) – Z number for each atom in the cell

• \text{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

positions
fractional atomic coordinates

from_CIF(conventional_standard_structure=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

from_pymatgen_structure(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 Project 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 formulas 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

from_unitcell_parameters(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** (numpy float) – max scattering vector to include (1/Angstroms)

• **tol_structure_factor** (numpy float) – tolerance for removing low-valued structure factors

**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, zone_axis_cartesian: ndarray | None = None, proj_x_cartesian: ndarray | None = None, proj_x_axis: 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.

:func: 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

```python
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 Ångstroms
  • **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 diffraction spots

**Return type**

- radii_unique (np array)

### excitation_errors

```python
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

```python
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

```python
fig, ax (handles): Optional figure and axis handles, if returnfig=True.
```

Return type

```python
bragg_peaks_cali (BraggVectors)
```

---

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 in 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
• **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_number_peaks=3, inversion_symmetry=True, multiple_corr_reset=True, progress_bar: bool = True, return_orientation: bool = True*)

This function computes the orientation of any number of PointLists stored in a PointListArray, and returns an OrientationMap.

    py4DSTEM.process.diffraction.crystal_ACOM.match_single_pattern(*self, bragg_peaks: PointList, num_matches_return: int = 1, 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_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
• 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 class containing all outputs fig, ax (handles): Figure handles for the plotting output

Return type
orientation (Orientation)

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, 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
• bragg_peaks_array (PointListArray) – All Bragg peaks
• orientation_map (OrientationMap) – Orientation map generated from ACOM
• corr_kernel_size (float) – Correlation kernel size - if user does not specify, uses self.corr_kernel_size.
• 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
• min_num_peaks (int) – Minimum number of peaks required.
• `rotation_range` (float) – Maximum rotation range in radians (for symmetry reduction).

• `progress_bar` (bool) – Show progress bar

• `mask_from_corr` (bool) – Use ACOM correlation signal for mask

• `corr_range` (np.ndarray) – Range of correlation signals for mask

• `corr_normalize` (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)`

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 = None)`

`__init__ (has_valid_cache: bool = False, cached_U_gmh: array | None = None) → 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-absorptive": Lobato factors plus an imaginary part
  equal to 0.1*f, as a simple but inaccurate way to include absorption, per Hashimoto,

  "WK": Uses the Weickenmeier-Kohl parameterization for
  the elastic form factors, including Debye-Waller factor, with no absorption, as de-

  "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}*<br>(This is often written as <u>)in papers) To specify different <u> for each element, pass a<br>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)* –
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**: (*DynamicalMatrixCache*) 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 |
foil_normal_lattice: ~numpy.ndarray | None = None,
foil_normal_cartesian: ~numpy.ndarray |
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 addition 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=True, set_calibration=True, 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.
- **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
Optional figure and axis handles, if returnfig=True.

Return type
fig, ax (handles)

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 each other, 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 redefine 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='nls', intensity_power=0, mask_peaks=None)
Quantification of the phase of a crystal based on the crystal instances and the pointlistarray.

Parameters
- pointlistarray (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**

• **pointlistarray** *(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 

\[
\begin{align*}
&[[0,0], [0,1], [1,0], [0,1]], \\
&[0,0] \text{ is the first crystal and the first match within that crystal. } [0,1]
\end{align*}
\]

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 unit 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

---

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

```python
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 = (12, 6), 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.
- **float** (bragg_k_broadening) – Broadening applied to bragg_peaks.
- **figsize** (list, tuple, np.ndarray) – Figure size for plot.
- **returnfig** (bool) – Return figure and axes handles if this is True.

Returns
fig, ax (optional) figure and axes handles
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: float = 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
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, orientation_ind=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.
- **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 orientation 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 handles 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.
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 orientation 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 handles 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.

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

```python
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 diffraction 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)* –

```python
py4DSTEM.process.diffraction.flowlines.make_orientation_histogram(bragg_peaks: PointList | 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 a 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 y 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)

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 symmetry for colouring
• theta_offset (float) – Offset the angular 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.
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 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 symmetry for coloring.
- **theta_offset (float)** – Offset the angular coloring by this value in radians.
- **white_background (bool)** – For either color or grayscale 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)
```

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 bins.
- **white_background (bool)** – For either color or grayscale 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.

**Returns**
flowline images

**Return type**
im_flowline (array)

```python
py4DSTEM.process.diffraction.flowlines.orientation_correlation(orient_hist, radius_max=None)
```

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)

```python
py4DSTEM.process.diffraction.flowlines.plot_orientation_correlation(orient_corr,
prob_range=[0.1, 10.0],
inds_plot=None,
pixel_size=None,
pixel_units=None,
size_fig=[8, 6],
return_fig=False)
```

Plot the distance-angle (auto)correlations in orient_corr.

**Parameters**

• **orient_corr** *(array)* – 3D or 4D array containing correlation images as function of (dr,dtheta) 1st index represents each pair of rings.
• **prob_range** *(array)* – Plotting range in units of “multiples of random distribution”.
• **inds_plot** *(float)* – Which indices to plot for orient_corr. Set to “None” to plot all pairs.
• **pixel_size** *(float)* – Pixel size for x axis.
• **pixel_units** *(str)* – units of pixels.
• **size_fig** *(array)* – Size of the figure panels.
• **return_fig** *(bool)* – Whether to return figure axes.

**Returns**
fig, ax Figure and axes handles (optional).

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

**Dynamic objects:**

1.4. **API**
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._displayhook() – 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 getrecursionlimit() – control how often the interpreter checks for events 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

```python
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 multiplied into all peak intensities, or a vector with 1 value per peak to scale peaks individually.
- **normalize_intensity** – Normalize maximum output value to 1.

```python
diskdetection
py4DSTEM.process.diskdetection.braggvectormap.get_bvm(braggpeaks, Qshape, mode='centered', Q_pixel_size=1.0)
```

Gets a Bragg vector map, a 2D histogram of Bragg scattering vectors.

**Parameters**
- **braggpeaks** (PointListArray) – the Bragg vectors and intensities.
- **Qshape** (2 tuple) – diffraction space shape
- **mode** (str) – must be ‘raw’ or ‘centered’. TODO, sampling selection
- **braggpeaks** –

```python
py4DSTEM.process.diskdetection.braggvectormap.get_bragg_vector_map(braggpeaks, Q_Nx, Q_Ny, Q_pixel_size=1)
```

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, given braggpeak positions which have been centered about the origin. In the returned array braggvectormap, the origin is placed at (Q_Nx/2.,Q_Ny/2.)

**Parameters**
- **braggpeaks** (PointListArray) – Must have the coords ‘qx’, ‘qy’, ‘intensity’, the default coordinates from the bragg peak detection fns
• \textbf{Q\_Nx (ints)} – the size of diffraction space in pixels

• \textbf{Q\_Ny (ints)} – the size of diffraction space in pixels

• \textbf{Q\_pixel\_size (number)} – the size of the diffraction space pixels

\textbf{Returns}
the bragg vector map

\textbf{Return type}
(ndarray)

\texttt{py4DSTEM.process.diskdetection.braggvectormap.get\_bragg\_vector\_maxima\_map(braggpeaks, Q\_Nx, Q\_Ny)}

Calculates the Bragg vector maxima map from a PointListArray of Bragg peak positions, given braggpeak positions which have been centered about the origin. In the returned array braggvectormap, the origin is placed at \((Q\_Nx/2.,Q\_Ny/2.)\)

\textbf{Parameters}

• \textbf{braggpeaks (PointListArray)} – Must have the coords 'qx','qy','intensity', the default coordinates from the bragg peak detection fns

• \textbf{Q\_Nx (ints)} – the size of diffraction space in pixels

• \textbf{Q\_Ny (ints)} – the size of diffraction space in pixels

\textbf{Returns}
(2D ndarray, shape (Q\_Nx,Q\_Ny)) the bragg vector map

\texttt{py4DSTEM.process.diskdetection.braggvectormap.get\_weighted\_bragg\_vector\_map(braggpeaks, Q\_Nx, Q\_Ny, weights)}

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, given bragg peak positions which have been centered about the origin, weighting the peaks at each scan position according to the array weights. In the returned array braggvectormap, the origin is placed at \((Q\_Nx/2.,Q\_Ny/2.)\)

\textbf{Parameters}

• \textbf{braggpeaks (PointListArray)} – Must have the coords 'qx','qy','intensity', the default coordinates from the bragg peak detection fns

• \textbf{Q\_Nx (int)} – the size of diffraction space in pixels

• \textbf{Q\_Ny (int)} – the size of diffraction space in pixels

• \textbf{weights (2D array)} – The shape of weights must be (R\_Nx,R\_Ny)

\textbf{Returns}
the bragg vector map

\textbf{Return type}
(2D ndarray, shape (Q\_Nx,Q\_Ny))

\texttt{py4DSTEM.process.diskdetection.braggvectormap.get\_bragg\_vector\_map\_raw(braggpeaks, Q\_Nx, Q\_Ny, Q\_pixel\_size=1)}

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, where the peak positions have not been centered.

\textbf{Parameters}

• \textbf{braggpeaks (PointListArray)} – Must have the coords 'qx','qy','intensity', the default coordinates from the bragg peak detection fns
• Q_Nx (ints) – the size of diffraction space in pixels
• Q_Ny (ints) – the size of diffraction space in pixels
• Q_pixel_size (number) – the size of the diffraction space pixels

Returns
the bragg vector map

Return type
(2D ndarray, shape (Q_Nx, Q_Ny))

Calculates the Bragg vector maxima map from a PointListArray of Bragg peak positions, where the peak positions have not been centered.

Parameters
• braggpeaks (PointListArray) – Must have the coords 'qx', 'qy', 'intensity', the default coordinates from the bragg peak detection fn
• Q_Nx (ints) – the size of diffraction space in pixels
• Q_Ny (ints) – the size of diffraction space in pixels

Returns
the bragg vector map

Return type
(2D ndarray, shape (Q_Nx, Q_Ny))

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, where the peak positions have not been centered, and weighting the peaks at each scan position according to the array weights.

Parameters
• braggpeaks (PointListArray) – Must have the coords 'qx', 'qy', 'intensity', the default coordinates from the bragg peak detection fn
• Q_Nx (ints) – the size of diffraction space in pixels
• Q_Ny (ints) – the size of diffraction space in pixels
• weights (2D array) – The shape of weights must be (R_Nx, R_Ny)

Returns
the bragg vector map

Return type
(2D ndarray, shape (Q_Nx, Q_Ny))

Calculates the Bragg vector maxima map from a PointListArray of Bragg peak positions, given braggpeak positions which have been centered about the origin. In the returned array braggvectormap, the origin is placed at (Q_Nx/2., Q_Ny/2.)

Parameters
• **braggpeaks** (*PointListArray*) – Must have the coords ’qx’,’qy’,’intensity’, the default coordinates from the bragg peak detection fns

• **Q_Nx** (*ints*) – the size of diffraction space in pixels

• **Q_Ny** (*ints*) – the size of diffraction space in pixels

**Returns**

(2D ndarray, shape (Q_Nx,Q_Ny)) the bragg vector map

```python
py4DSTEM.process.diskdetection.braggvectormap.get_bvm_weighted(braggpeaks, Q_Nx, Q_Ny, weights)
```

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, given bragg peak positions which have been centered about the origin, weighting the peaks at each scan position according to the array weights. In the returned array braggvectormap, the origin is placed at (Q_Nx/2.,Q_Ny/2.)

**Parameters**

• **braggpeaks** (*PointListArray*) – Must have the coords ’qx’,’qy’,’intensity’, the default coordinates from the bragg peak detection fns

• **Q_Nx** (*int*) – the size of diffraction space in pixels

• **Q_Ny** (*int*) – the size of diffraction space in pixels

• **weights** (*2D array*) – The shape of weights must be (R_Nx,R_Ny)

**Returns**

the bragg vector map

**Return type**

(2D ndarray, shape (Q_Nx,Q_Ny))

```python
py4DSTEM.process.diskdetection.braggvectormap.get_bvm_raw(braggpeaks, Q_Nx, Q_Ny, Q_pixel_size=1)
```

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, where the peak positions have not been centered.

**Parameters**

• **braggpeaks** (*PointListArray*) – Must have the coords ’qx’,’qy’,’intensity’, the default coordinates from the bragg peak detection fns

• **Q_Nx** (*ints*) – the size of diffraction space in pixels

• **Q_Ny** (*ints*) – the size of diffraction space in pixels

• **Q_pixel_size** (*number*) – the size of the diffraction space p[ixels]

**Returns**

the bragg vector map

**Return type**

(2D ndarray, shape (Q_Nx,Q_Ny))

```python
py4DSTEM.process.diskdetection.braggvectormap.get_bvm_maxima_raw(braggpeaks, Q_Nx, Q_Ny)
```

Calculates the Bragg vector maxima map from a PointListArray of Bragg peak positions, where the peak positions have not been centered.

**Parameters**

• **braggpeaks** (*PointListArray*) – Must have the coords ’qx’,’qy’,’intensity’, the default coordinates from the bragg peak detection fns

• **Q_Nx** (*ints*) – the size of diffraction space in pixels
• **Q\_Ny** (ints) – the size of diffraction space in pixels

**Returns**
the bragg vector map

**Return type**
(2D ndarray, shape \((Q\_Nx, Q\_Ny)\))

py4DSTEM.process.diskdetection.braggvectormap.get\_bvm\_weighted\_raw(braggpeaks, \(Q\_Nx, Q\_Ny,\)
weights)

Calculates the Bragg vector map from a PointListArray of Bragg peak positions, where the peak positions have not been centered, and weighting the peaks at each scan position according to the array weights.

**Parameters**

• **braggpeaks** (PointListArray) – Must have the coords 'qx','qy','intensity', the default coordinates from the bragg peak detection fns
• **Q\_Nx** (ints) – the size of diffraction space in pixels
• **Q\_Ny** (ints) – the size of diffraction space in pixels
• **weights** (2D array) – The shape of weights must be \((R\_Nx,R\_Ny)\)

**Returns**
the bragg vector map

**Return type**
(2D ndarray, shape \((Q\_Nx,Q\_Ny)\))

py4DSTEM.process.diskdetection.diskdetection.find\_Bragg\_disks(data, template,
filter\_function=None, corr\_Power=1, sigma=2,
subpixel='multicorr', upsample\_factor=16, min\_Absolute\_Intensity=0,
min\_Relative\_Intensity=0.005, relative\_To\_Peak=0,
min\_Peak\_Spacing=60, edge\_Boundary=20,
max\_Num\_Peaks=70, CUDA=False, CUDA\_batched=True,
distributed=None, ML=False,
ml\_model\_path=None, ml\_num\_attempts=1,
ml\_batch\_size=8)

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

Behavior depends on *data*. If it is

• a DataCube: runs on all its diffraction patterns, and returns a
  BraggVectors instance
• a 2D array: runs on this array, and returns a QPoints instance
• a 3D array: runs slice the ar[i,:,:] slices of this array, and returns
  a len(ar.shape[0]) list of QPoints instances.
• a 3-tuple (DataCube, rx, ry), for numbers or length-N arrays (rx,ry):
  runs on the diffraction patterns in DataCube at positions (rx,ry), and returns a instance or length N
  list of instances of QPoints
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. If filter_function is None, this 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 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
- **data** (variable) – see above
- **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.
- **filter_function** (callable) – filtering function to apply to each diffraction pattern before peakfinding. 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 giving hybrid correlations.
- **sigma** (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

• **edgeBoundary** (*int*) – minimum acceptable distance for detected peaks from 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

Returns

the Bragg peak positions and correlation intensities. If

*data* is:

• a DataCube, returns a BraggVectors instance

• a 2D array, returns a QPoints instance

• a 3D array, returns a list of QPoints instances

• a (DataCube, rx, ry) 3-tuple, returns a list of QPoints instances

Return type

(variable)

Functions for finding Braggdisks using AI/ML method using tensorflow
Finds the Bragg disks in single DP by AI/ML method. This method utilizes FCU-Net to predict Bragg disks from diffraction images.

The input DP and Probes need to be aligned before the prediction. Detected peaks within edgeBoundary pixels of the diffraction plane edges are then discarded. Next, peaks with intensities less than minRelativeIntensity of the brightest peak in the correlation are discarded. Then peaks which are within a distance of minPeakSpacing of their nearest neighbor peak are found, and in each such pair the peak with the lesser correlation intensities is removed. Finally, if the number of peaks remaining exceeds maxNumPeaks, only the maxNumPeaks peaks with the highest correlation intensity are retained.

**Parameters**

- **DP (ndarray)** – a diffraction pattern
- **probe (ndarray)** – the vacuum probe template
- **num_attempts (int)** – Number of attempts to predict the Bragg disks. Recommended: 5 Ideally, the more num_attempts the better (confident) the prediction will be as the ML prediction utilizes Monte Carlo Dropout technique to estimate model uncertainty using Bayesian approach. Note: increasing num_attempts will increase the compute time significantly and it is advised to use GPU (CUDA) enabled environment for fast prediction with num_attempts > 1
- **int_window_radius (int)** – window radius (in pixels) for disk intensity integration over the predicted atomic potentials array

```python
def find_Bragg_disks_aimg_single_DP(DP, probe, num_attempts=5, int_window_radius=1, predict=True, sigma=0, edge_Boundary=20, minRelativeIntensity=0.005, minAbsoluteIntensity=0, relativeToPeak=0, minPeakSpacing=60, sub_pixel='multicorr', upsampling_factor=16, filter_function=None, peaks=None, model_path=None)
```
• **predict** (*bool*) – Flag to determine if ML prediction is opted.
• **edgeBoundary** (*int*) – minimum acceptable distance from the DP edge, in pixels
• **minRelativeIntensity** (*float*) – the minimum acceptable correlation peak intensity, relative to the intensity of the relativeToPeak’th peak
• **minAbsoluteIntensity** (*float*) – the minimum acceptable correlation peak intensity, on an absolute scale
• **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
• **maxNumPeaks** (*int*) – the maximum number of peaks to return
• **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')
• **filter_function** (*callable*) – filtering function to apply to each diffraction pattern before peakfinding. 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.
• **peaks** (*PointList*) – For internal use. If peaks is None, the PointList of peak positions is created here. If peaks is not None, it is the PointList that detected peaks are added to, and must have the appropriate coords ('qx','qy','intensity').
• **model_path** (*str*) – filepath for the model weights (Tensorflow model) to load from. By default, if the model_path is not provided, py4DSTEM will search for the latest model stored on cloud using metadata json file. It is not recommended to keep track of the model path and advised to keep this argument unchanged (None) to always search for the latest updated training model weights.

**Returns**
the Bragg peak positions and correlation intensities

**Return type**
(*PointList*)
find_Bragg_disks_aiml_selected(datacube, probe, Rx, Ry, num_attempts=5, int_window_radius=1, batch_size=1, predict=True, sigma=0, edge_Boundary=20, minRelativeIntensity=0.005, minAbsoluteIntensity=0, relativeToPeak=0, minPeakSpacing=60, maxNumPeaks=70, sub_pixel='multicorr', upsample_factor=16, filter_function=None, model_path=None)

Finds the Bragg disks in the diffraction patterns of datacube at scan positions (Rx,Ry) by AI/ML method. This method utilizes FCU-Net to predict Bragg disks from diffraction images.

Parameters

- **datacube** (datacube) – a diffraction datacube
- **probe** (ndarray) – the vacuum probe template
- **num_attempts** (int) – Number of attempts to predict the Bragg disks. Recommended: 5 Ideally, the more num_attempts the better (confident) the prediction will be as the ML prediction utilizes Monte Carlo Dropout technique to estimate model uncertainty using Bayesian approach. Note: increasing num_attempts will increase the compute time significantly and it is advised to use GPU (CUDA) enabled environment for fast prediction with num_attempts > 1
- **int_window_radius** (int) – window radius (in pixels) for disk intensity integration over the predicted atomic potentials array
- **predict** (bool) – Flag to determine if ML prediction is opted.
- **edgeBoundary** (int) – minimum acceptable distance from the DP edge, in pixels
- **minRelativeIntensity** (float) – the minimum acceptable correlation peak intensity, relative to the intensity of the relativeToPeak’th peak
- **minAbsoluteIntensity** (*float*) – the minimum acceptable correlation peak intensity, on an absolute scale
- **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
- **maxNumPeaks** (*int*) – the maximum number of peaks to return
- **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’)
- **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 dill.
- **peaks** (*PointList*) – For internal use. If peaks is None, the PointList of peak positions is created here. If peaks is not None, it is the PointList that detected peaks are added to, and must have the appropriate coords ("qx","qy","intensity").
- **model_path** (*str*) – filepath for the model weights (Tensorflow model) to load from. By default, if the model_path is not provided, py4DSTEM will search for the latest model stored on cloud using metadata json file. It is not recommended to keep track of the model path and advised to keep this argument unchanged (None) to always search for the latest updated training model weights.

Returns
the Bragg peak positions and correlation intensities at each scan position (Rx,Ry).

Return type
(n-tuple of PointLists, n=len(Rx))
Finds the Bragg disks in all diffraction patterns of datacube from AI/ML method. When hist = True, returns histogram of intensities in the entire datacube.

Parameters

- **datacube** (*datacube*) – a diffraction datacube
- **probe** (*ndarray*) – the vacuum probe template
- **num_attempts** (*int*) – Number of attempts to predict the Bragg disks. Recommended: 5. Ideally, the more num_attempts the better (confident) the prediction will be as the ML prediction utilizes Monte Carlo Dropout technique to estimate model uncertainty using Bayesian approach. Note: increasing num_attempts will increase the compute time significantly and it is advised to use GPU (CUDA) enabled environment for fast prediction with num_attempts > 1
- **int_window_radius** (*int*) – window radius (in pixels) for disk intensity integration over the predicted atomic potentials array
- **predict** (*bool*) – Flag to determine if ML prediction is opted.
• **batch_size** *(int)* – batch size for Tensorflow model.predict() function, by default `batch_size = 2`. Note: if you are using CPU for model.predict(), please use `batch_size < 2`. Future version will implement Dask parallelization implementation of the serial function to boost up the performance of Tensorflow CPU predictions. Keep in mind that this function will take significant amount of time to predict for all the DPs in a datacube.

• **edgeBoundary** *(int)* – minimum acceptable distance from the DP edge, in pixels

• **minRelativeIntensity** *(float)* – the minimum acceptable correlation peak intensity, relative to the intensity of the relativeToPeak’th peak

• **minAbsoluteIntensity** *(float)* – the minimum acceptable correlation peak intensity, on an absolute scale

• **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

• **maxNumPeaks** *(int)* – the maximum number of peaks to return

• **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'`)

• **global_threshold** *(bool)* – if True, applies global threshold based on minGlobalIntensity and metric

• **minGlobalThreshold** *(float)* – the minimum allowed peak intensity, relative to the selected metric (0-1), except in the case of ‘manual’ metric, in which the threshold value based on the minimum intensity that you want threshold out should be set.

• **metric** *(string)* – the metric used to compare intensities. ‘average’ compares peak intensity relative to the average of the maximum intensity in each diffraction pattern. ‘max’ compares peak intensity relative to the maximum intensity value out of all the diffraction patterns. ‘median’ compares peak intensity relative to the median of the maximum intensity peaks in each diffraction pattern. ‘manual’ Allows the user to threshold based on a predetermined intensity value manually determined. In this case, minIntensity should be an int.

• **name** *(str)* – name for the returned PointListArray

• **filter_function** *(callable)* – filtering function to apply to each diffraction pattern before peakfinding. 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 dill.

• **model_path** *(str)* – filepath for the model weights (Tensorflow model) to load from. By default, if the model_path is not provided, py4DSTEM will search for the latest model stored on cloud using metadata json file. It is not recommended to keep track of the model path and advised to keep this argument unchanged (None) to always search for the latest updated training model weights.
**Returns**
the Bragg peak positions and correlation intensities

**Return type**
(PointListArray)

```python
py4DSTEM.process.diskdetection.diskdetection_aiml.find_Bragg_disks_aiml(datacube, probe,
num_attempts=5,
int_window_radius=1,
predict=True,
batch_size=8,
sigma=0,
edgeBoundary=20,
minRelativeIntensity=0.005,
minAbsoluteIntensity=0,
relativeToPeak=0,
minPeakSpacing=60,
subpixel='multicorr',
upsample_factor=16,
name='braggpeaks_raw',
filter_function=None,
model_path=None,
distributed=None,
CUDA=True,
**kwargs)
```

Finds the Bragg disks in all diffraction patterns of datacube by AI/ML method. This method utilizes FCU-Net to predict Bragg disks from diffraction images.

**datacube (datacube):** a diffraction datacube
probe (ndarray): the vacuum probe template num_attempts (int): Number of attempts to predict the Bragg disks. Recommended: 5.

Ideally, the more num_attempts the better (confident) the prediction will be as the ML prediction utilizes Monte Carlo Dropout technique to estimate model uncertainty using Bayesian approach.

Note: increasing num_attempts will increase the compute time significantly and it is advised to use GPU (CUDA) enabled environment for fast prediction with num_attempts > 1

**int_window_radius (int):** window radius (in pixels) for disk intensity integration over the predicted atomic potentials array

predict (bool): Flag to determine if ML prediction is opted. batch_size (int): batch size for Tensorflow model.predict() function, by default batch_size = 2,

Note: if you are using CPU for model.predict(), please use batch_size < 2. Future version will implement Dask parallelization implementation of the serial function to boost up the performance of Tensorflow CPU predictions. Keep in mind that this function will take significant amount of time to predict for all the DPs in a datacube.

**edgeBoundary (int):** minimum acceptable distance from the DP edge, in pixels minRelativeIntensity (float): the minimum acceptable correlation peak intensity,

relative to the intensity of the relativeToPeak’th peak
**minAbsoluteIntensity** (float): the minimum acceptable correlation peak intensity, on an absolute scale

**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

**maxNumPeaks** (int): the maximum number of peaks to return

**subpixel** (str): Whether to use subpixel fitting, and which algorithm to use.

- **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')

**global_threshold** (bool): if True, applies global threshold based on minGlobalIntensity and metric

**minGlobalThreshold** (float): the minimum allowed peak intensity, relative to the selected metric (0-1), except in the case of 'manual' metric, in which the threshold value based on the minimum intensity that you want thresholder out should be set.

**metric** (string): the metric used to compare intensities. ‘average’ compares peak intensity relative to the average of the maximum intensity in each diffraction pattern. ‘max’ compares peak intensity relative to the maximum intensity value out of all the diffraction patterns. ‘median’ compares peak intensity relative to the median of the maximum intensity peaks in each diffraction pattern. ‘manual’ Allows the user to threshold based on a predetermined intensity value manually determined. In this case, minIntensity should be an int.

**name** (str): name for the returned PointListArray

**filter_function** (callable): filtering function to apply to each diffraction pattern before peakfinding. 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 dill.

**model_path** (str): filepath for the model weights (Tensorflow model) to load from. By default, if the model_path is not provided, py4DSTEM will search for the latest model stored on cloud using metadata json file. It is not recommended to keep track of the model path and advised to keep this argument unchanged (None) to always search for the latest updated training model weights.

**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

CUDA (bool): When True, py4DSTEM will use CUDA-enabled disk_detection_aiml function

Returns
the Bragg peak positions and correlation intensities

Return type
(PointListArray)

py4DSTEM.process.diskdetection.threshold.threshold_Braggpeaks(pointlistarray, minRelativeIntensity, relativeToPeak, minPeakSpacing, maxNumPeaks)

Takes a PointListArray of detected Bragg peaks and applies additional thresholding, returning the thresholded PointListArray. To skip a threshold, set that parameter to False.

Parameters
• pointlistarray (PointListArray) – The Bragg peaks. Must have co-
  ords=('qx','qy','intensity')
• minRelativeIntensity (float) – the minimum allowed peak intensity, relative to the
  brightest peak in each diffraction pattern
• relativeToPeak (int) – specifies the peak against which the minimum relative intensity
  is measured – 0=brightest maximum. 1=next brightest, etc.
• minPeakSpacing (int) – the minimum allowed spacing between adjacent peaks
• maxNumPeaks (int) – maximum number of allowed peaks per diffraction pattern

py4DSTEM.process.diskdetection.threshold.universal_threshold(pointlistarray, thresh, metric='maximum', minPeakSpacing=False, maxNumPeaks=False, name=None)

Takes a PointListArray of detected Bragg peaks and applies universal thresholding, returning the thresholded PointListArray. To skip a threshold, set that parameter to False.

Parameters
• pointlistarray (PointListArray) – The Bragg peaks. Must have co-
  ords=('qx','qy','intensity')
• thresh (float) – the minimum allowed peak intensity. The meaning of this threshold
  value is determined by the value of the ‘metric’ argument, below
• metric (string) – the metric used to compare intensities. Must be in (‘maxi-
  mum’,’average’,’median’,’manual’). In each case aside from ‘manual’, the intensity
  threshold is set to Val*thresh, where Val is given by
  – ’maximum’ - the maximum intensity in the entire pointlistarray
  – ’average’ - the average of the maximum intensities of each scan position in the
    pointlistarray
  – ’median’ - the medain of the maximum intensities of each scan position in the entire
    pointlistarray
  If metric is ‘manual’, the threshold is exactly minIntensity
• **minPeakSpacing** *(int)* – the minimum allowed spacing between adjacent peaks. optional, default is false

• **maxNumPeaks** *(int)* – maximum number of allowed peaks per diffraction pattern. optional, default is false

• **name** *(str, optional)* – a name for the returned PointListArray. If unspecified, takes the old PLA name and appends ‘_unithresh’.

**Returns**
Bragg peaks thresholded by intensity.

**Return type**
*(PointListArray)*

`py4DSTEM.process.diskdetection.threshold.get_pointlistarray_intensities(pointlistarray)`

Concatenates the Bragg peak intensities from a PointListArray of Bragg peak positions into one array and returns the intensities. This output can be used for understanding the distribution of intensities in your dataset for universal thresholding.

**Parameters**

- **pointlistarray** *(PointListArray)* –

**Returns**
all detected peak intensities

**Return type**
*(ndarray)*

`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

```python
>>> 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, where the fit function takes its first input in the form of a length 2 vector (ndarray) of (x,y) positions, followed by the remaining parameters, and the data to fit takes the form of an (n,m) shaped array. Robust fitting can be enabled to iteratively reject outlier data points, which have a root-mean-square error beyond the user-specified threshold.

**Parameters**

- **function** – First input should be a length 2 array xy, where (xy[0],xy[1]) are the (x,y) coordinates
- **data** – Data to fit, in an (n,m) shaped ndarray
- **data_mask** – Optional parameter. If specified, must be a boolean array of the same shape as data, specifying which elements of data to use in the fit
- **return_ar** – Optional parameter. If False, only the fit parameters and covariance matrix are returned. If True, return an array of the same shape as data with the fit values. Defaults to True
- **popt** – Optional parameter for input. If specified, should be a tuple of initial guesses for the fit parameters.
• **robust** – Optional parameter. If set to True, fit will be repeated with outliers removed.

• **robust_steps** – Optional parameter. Number of robust iterations performed after initial fit.

• **robust_thresh** – Optional parameter. Threshold for including points, in units of root-mean-square (standard deviations) error of the predicted values after fitting.

**Returns**

• **popt**: optimal fit parameters to function

• **pcov**: the covariance matrix

• **fit_ar**: optional. If return_ar==True, fit_ar is returned, and is an array of the same shape as data, containing the fit values

**Return type**

(3-tuple) A 3-tuple containing

```python
py4DSTEM.process.latticevectors.fit

fit_lattice_vectors(braggpeaks, x0=0, y0=0, minNumPeaks=5)
```

Fits lattice vectors g1,g2 to braggpeaks given some known (h,k) indexing.

**Parameters**

• **braggpeaks** (**PointList**) – A 6 coordinate PointList containing the data to fit. Coords are ‘qx’, ‘qy’ (the bragg peak positions), ‘intensity’ (used as a weighting factor when fitting), ‘h’, ‘k’ (indexing). May optionally also contain ‘index_mask’ (bool), indicating which peaks have been successfully indexed and should be used.

• **x0** (**float**) – x-coord of the origin

• **y0** (**float**) – y-coord of the origin

• **minNumPeaks** (**int**) – if there are fewer than minNumPeaks peaks found in braggpeaks which can be indexed, return None for all return parameters

**Returns**

• **x0**: (float) the x-coord of the origin of the best-fit lattice.

• **y0**: (float) the y-coord of the origin

• **g1x**: (float) x-coord of the first lattice vector

• **g1y**: (float) y-coord of the first lattice vector

• **g2x**: (float) x-coord of the second lattice vector

• **g2y**: (float) y-coord of the second lattice vector

• **error**: (float) the fit error

**Return type**

(7-tuple) A 7-tuple containing

```python
py4DSTEM.process.latticevectors.fit

fit_lattice_vectors_all_DPs(braggpeaks, x0=0, y0=0, minNumPeaks=5)
```

Fits lattice vectors g1,g2 to each diffraction pattern in braggpeaks, given some known (h,k) indexing.

**Parameters**
• **braggpeaks** (*PointList*) – A 6 coordinate PointList containing the data to fit. Coords are ‘qx’, ‘qy’ (the bragg peak positions), ‘intensity’ (used as a weighting factor when fitting), ‘h’, ‘k’ (indexing). May optionally also contain ‘index_mask’ (bool), indicating which peaks have been successfully indexed and should be used.

• **x0** (*float*) – x-coord of the origin

• **y0** (*float*) – y-coord of the origin

• **minNumPeaks** (*int*) – if there are fewer than minNumPeaks peaks found in braggpeaks which can be indexed, return None for all return parameters

**Returns**

A RealSlice g1g2map containing the following 8 arrays:

• `g1g2_map.get_slice('x0')` x-coord of the origin of the best fit lattice

• `g1g2_map.get_slice('y0')` y-coord of the origin

• `g1g2_map.get_slice('glx')` x-coord of the first lattice vector

• `g1g2_map.get_slice('g1x')` x-coord of the first lattice vector

• `g1g2_map.get_slice('g2x')` x-coord of the second lattice vector

• `g1g2_map.get_slice('g2y')` y-coord of the second lattice vector

• `g1g2_map.get_slice('error')` the fit error

• `g1g2_map.get_slice('mask')` 1 for successful fits, 0 for unsuccessful fits

**Return type**

(RealSlice)

py4DSTEM.process.latticevectors.fit.fit_lattice_vectors_masked(braggpeaks, mask, x0=0, y0=0, minNumPeaks=5)

Fits lattice vectors g1,g2 to each diffraction pattern in braggpeaks corresponding to a scan position for which masks==True.

**Parameters**

• **braggpeaks** (*PointList*) – A 6 coordinate PointList containing the data to fit. Coords are ‘qx’, ‘qy’ (the bragg peak positions), ‘intensity’ (used as a weighting factor when fitting), ‘h’, ‘k’ (indexing). May optionally also contain ‘index_mask’ (bool), indicating which peaks have been successfully indexed and should be used.

• **mask** (*boolean array*) – real space shaped (R_Nx,R_Ny); fit lattice vectors where mask is True

• **x0** (*float*) – x-coord of the origin

• **y0** (*float*) – y-coord of the origin

• **minNumPeaks** (*int*) – if there are fewer than minNumPeaks peaks found in braggpeaks which can be indexed, return None for all return parameters

**Returns**

A RealSlice g1g2map containing the following 8 arrays:

• `g1g2_map.get_slice('x0')` x-coord of the origin of the best fit lattice

• `g1g2_map.get_slice('y0')` y-coord of the origin

• `g1g2_map.get_slice('glx')` x-coord of the first lattice vector
• g1g2_map.get_slice('g1y') y-coord of the first lattice vector
• g1g2_map.get_slice('g2x') x-coord of the second lattice vector
• g1g2_map.get_slice('g2y') y-coord of the second lattice vector
• g1g2_map.get_slice('error') the fit error
• g1g2_map.get_slice('mask') 1 for successful fits, 0 for unsuccessful fits

Return type

(RealSlice)

py4DSTEM.process.latticevectors.index.get_selected_lattice_vectors(gx, gy, i0, i1, i2)

From a set of reciprocal lattice points (gx,gy), and indices in those arrays which specify the center beam, the first basis lattice vector, and the second basis lattice vector, computes and returns the lattice vectors g1 and g2.

Parameters

• gx (1d array) – the reciprocal lattice points x-coords
• gy (1d array) – the reciprocal lattice points y-coords
• i0 (int) – index in the (gx,gy) arrays specifying the center beam
• i1 (int) – index in the (gx,gy) arrays specifying the first basis lattice vector
• i2 (int) – index in the (gx,gy) arrays specifying the second basis lattice vector

Returns

(2-tuple of 2-tuples) A 2-tuple containing
• g1: (2-tuple) the first lattice vector, (g1x,g1y)
• g2: (2-tuple) the second lattice vector, (g2x,g2y)

py4DSTEM.process.latticevectors.index.index_bragg_directions(x0, y0, gx, gy, g1, g2)

From an origin (x0,y0), a set of reciprocal lattice vectors gx,gy, and an pair of lattice vectors g1=(g1x,g1y), g2=(g2x,g2y), find the indices (h,k) of all the reciprocal lattice directions.

The approach is to solve the matrix equation

alpha = beta * M

where alpha is the 2xN array of the (x,y) coordinates of N measured bragg directions, beta is the 2x2 array of the two lattice vectors u,v, and M is the 2xN array of the h,k indices.

Parameters

• x0 (float) – x-coord of origin
• y0 (float) – y-coord of origin
• gx (1d array) – x-coord of the reciprocal lattice vectors
• gy (1d array) – y-coord of the reciprocal lattice vectors
• g1 (2-tuple of floats) – g1x,g1y
• g2 (2-tuple of floats) – g2x,g2y

Returns

• h: (ndarray of ints) first index of the bragg directions
• k: (ndarray of ints) second index of the bragg directions
• **bragg_directions**: *(PointList)* a 4-coordinate PointList with the indexed bragg directions; coords ‘qx’ and ‘qy’ contain bragg_x and bragg_y coords ‘h’ and ‘k’ contain h and k.

**Return type**

(3-tuple) A 3-tuple containing

```python
py4DSTEM.process.latticevectors.index.generate_lattice(ux, uy, vx, vy, x0, y0, Q_Nx, Q_Ny, 
    h_max=None, k_max=None)
```

Returns a full reciprocal lattice stretching to the limits of the diffraction pattern by making linear combinations of the lattice vectors up to (±h_max,±k_max).

This can be useful when there are false peaks or missing peaks in the braggvectormap, which can cause errors in the strain finding routines that rely on those peaks for indexing. This allows us to create a reference lattice that has all combinations of the lattice vectors all the way out to the edges of the frame, and excluding any erroneous intermediate peaks.

**Parameters**

- **ux** *(float)* – x-coord of the u lattice vector
- **uy** *(float)* – y-coord of the u lattice vector
- **vx** *(float)* – x-coord of the v lattice vector
- **vy** *(float)* – y-coord of the v lattice vector
- **x0** *(float)* – x-coord of the lattice origin
- **y0** *(float)* – y-coord of the lattice origin
- **Q_Nx** *(int)* – diffraction pattern size in the x-direction
- **Q_Ny** *(int)* – diffraction pattern size in the y-direction
- **h_max** *(int)* – maximal indices for generating the lattice (the lattice is always trimmed to fit inside the pattern so you can overestimate these, or leave unspecifed and they will be automatically found)
- **k_max** *(int)* – maximal indices for generating the lattice (the lattice is always trimmed to fit inside the pattern so you can overestimate these, or leave unspecifed and they will be automatically found)

**Returns**

A 4-coordinate PointList, (‘qx’,’qy’,’h’,’k’), containing points corresponding to linear combinations of the u and v vectors, with associated indices

**Return type**

*(PointList)*

```python
py4DSTEM.process.latticevectors.index.add_indices_to_braggpeaks(braggpeaks, lattice,
    maxPeakSpacing, qx_shift=0,
    qy_shift=0, mask=None)
```

Using the peak positions (qx,qy) and indices (h,k) in the PointList lattice, identify the indices for each peak in the PointListArray braggpeaks. Return a new braggpeaks_indexed PointListArray, containing a copy of braggpeaks plus three additional data columns – ‘h’,’k’, and ‘index_mask’ – specifying the peak indices with the ints (h,k) and indicating whether the peak was successfully indexed or not with the bool index_mask. If mask is specified, only the locations where mask is True are indexed.

**Parameters**

- **braggpeaks** *(PointListArray)* – the braggpeaks to index. Must contain the coordinates ‘qx’, ‘qy’, and ‘intensity’
• **lattice** *(PointList)* – the positions \((q_x, q_y)\) of the \((h,k)\) lattice points. Must contain the coordinates ‘\(q_x\)’, ‘\(q_y\)’, ‘\(h\)’, and ‘\(k\)’

• **maxPeakSpacing** *(float)* – Maximum distance from the ideal lattice points to include a peak for indexing

• **qx_shift** *(number)* – the shift of the origin in the lattice PointList relative to the \(\text{bragg-peaks}\) PointListArray

• **qy_shift** *(number)* – the shift of the origin in the lattice PointList relative to the \(\text{bragg-peaks}\) PointListArray

• **mask** *(bool)* – Boolean mask, same shape as the pointlistarray, indicating which locations should be indexed. This can be used to index different regions of the scan with different lattices

**Returns**

The original braggpeaks pointlistarray, with new coordinates ‘\(h\)’, ‘\(k\)’, containing the indices of each indexable peak.

**Return type**

*(PointListArray)*

---

**py4DSTEM.process.latticevectors.index.brass_vector_intensity_map_by_index** *(braggpeaks, h, k, symmetric=False)*

Returns a correlation intensity map for an indexed \((h,k)\) Bragg vector. Used to obtain a darkfield image corresponding to the \((h,k)\) reflection or a brightfield image when \(h=k=0\)

**Parameters**

• **braggpeaks** *(PointListArray)* – must contain the coordinates ‘\(h\)’, ‘\(k\)’, and ‘intensity’

• **h** *(int)* – indices for the reflection to generate an intensity map from

• **k** *(int)* – indices for the reflection to generate an intensity map from

• **symmetric** *(bool)* – if set to true, returns sum of intensity of \((h,k)\), \((-h,k)\), \((h,-k)\), \((-h,-k)\)

**Returns**

a map of the intensity of the \((h,k)\) Bragg vector correlation. Same shape as the pointlistarray.

**Return type**

*(numpy array)*

---

**py4DSTEM.process.latticevectors.initialguess.get_radon_scores** *(braggvectormap, mask=None, N_angles=200, sigma=2, minSpacing=2, minRelativeIntensity=0.05)*

Calculates a score function, score(angle), representing the likelihood that angle is a principle lattice direction of the lattice in braggvectormap.

The procedure is as follows: If mask is not None, ignore any data in braggvectormap where mask is False. Useful for removing the unscattered beam, which can dominate the results. Take the Radon transform of the (masked) Bragg vector map. For each angle, get the corresponding slice of the sinogram, and calculate its score. If we let \(R_\theta(r)\) be the sinogram slice at angle \(\theta\), and where \(r\) is the sinogram position coordinate, then the score of the slice is given by

\[
score(\theta) = \frac{\text{sum}_i (R_\theta(r_i))}{N_i}
\]

Here, \(r_i\) are the positions \(r\) of all local maxima in \(R_\theta(r)\), and \(N_i\) is the number of such maxima. Thus the score is large when there are few maxima which are high intensity.

**Parameters**
• **braggvectormap** (*ndarray*) – the Bragg vector map

• **mask** (*ndarray of bools*) – ignore data in braggvectormap wherever mask==False

• **N_angles** (*int*) – the number of angles at which to calculate the score

• **sigma** (*float*) – smoothing parameter for local maximum identification

• **minSpacing** (*float*) – if two maxima are found in a radon slice closer than minSpacing, the dimmer of the two is removed

• **minRelativeIntensity** (*float*) – maxima in each radon slice dimmer than minRelativeIntensity compared to the most intense maximum are removed

**Returns**

• **scores**: (*ndarray, len N_angles, floats*) the scores for each angle

• **thetas**: (*ndarray, len N_angles, floats*) the angles, in radians

• **sinogram**: (*ndarray*) the radon transform of braggvectormap*mask

**Return type**

(3-tuple) A 3-tuple containing

```python
py4DSTEM.process.latticevectors.initialguess.get_lattice_directions_from_scores(thetas, scores, sigma=2, minSpacing=2, minRelativeIntensity=0.05, index1=0, index2=0)
```

Get the lattice directions from the scores of the radon transform slices.

**Parameters**

• **thetas** (*ndarray*) – the angles, in radians

• **scores** (*ndarray*) – the scores

• **sigma** (*float*) – gaussian blur for local maxima identification

• **minSpacing** (*float*) – minimum spacing for local maxima identification

• **minRelativeIntensity** (*float*) – minimum intensity, relative to the brightest maximum, for local maxima identification

• **index1** (*int*) – specifies which local maximum to use for the first lattice direction, in order of maximum intensity

• **index2** (*int*) – specifies the local maximum for the second lattice direction

**Returns**

• **theta1**: (*float*) the first lattice direction, in radians

• **theta2**: (*float*) the second lattice direction, in radians

**Return type**

(2-tuple) A 2-tuple containing
Gets the lengths of the two lattice vectors from their angles and the sinogram.

First, finds the spacing between peaks in the sinogram slices projected down the u- and v- directions, u_proj and v_proj. Then, finds the lengths by taking:

\[ |u| = \frac{v_{proj}}{\sin(u_{\theta}-v_{\theta})} \]
\[ |v| = \frac{u_{proj}}{\sin(u_{\theta}-v_{\theta})} \]

The most important thresholds for this function are spacing_thresh, which discards any detected spacing between adjacent radon projection peaks which deviate from the median spacing by more than this fraction, and minRelativeIntensity, which discards detected maxima (from which spacings are then calculated) below this threshold relative to the brightest maximum.

**Parameters**

- **u_theta** (*float*) – the angle of u, in radians
- **v_theta** (*float*) – the angle of v, in radians
- **thetas** (*ndarray*) – the angles corresponding to the sinogram
- **sinogram** (*ndarray*) – the sinogram
- **spacing_thresh** (*float*) – ignores spacings which are greater than spacing_thresh times the median spacing
- **sigma** (*float*) – gaussian blur for local maxima identification
- **minSpacing** (*float*) – minimum spacing for local maxima identification
- **minRelativeIntensity** (*float*) – minimum intensity, relative to the brightest maximum, for local maxima identification

**Returns**

- **u_length** (*float*) the length of u, in pixels
- **v_length** (*float*) the length of v, in pixels

**Return type**

(2-tuple) A 2-tuple containing

py4DSTEM.process.latticevectors.strain.get_reference_g1g2(g1g2_map, mask)

Gets a pair of reference lattice vectors from a region of real space specified by mask. Takes the median of the lattice vectors in g1g2_map within the specified region.

**Parameters**

- **g1g2_map** (*RealSlice*) – the lattice vector map; contains 2D arrays in g1g2_map.data under the keys ‘g1x’, ‘g1y’, ‘g2x’, and ‘g2y’. See documentation for fit_lattice_vectors_all_DPs() for more information.
- **mask** (*ndarray of bools*) – use lattice vectors from g1g2_map scan positions wherever mask==True

**Returns**

- **g1**: (2-tuple) first reference lattice vector (x,y)
• \( g2 \): (2-tuple) second reference lattice vector (x,y)

**Return type**

(2-tuple of 2-tuples) A 2-tuple containing

py4DSTEM.process.latticevectors.strain.get_strain_from_reference_g1g2(g1g2_map, g1, g2)

Gets a strain map from the reference lattice vectors \( g1, g2 \) and lattice vector map \( g1g2\_map \).

Note that this function will return the strain map oriented with respect to the x/y axes of diffraction space - to rotate the coordinate system, use get_rotated_strain_map(). Calibration of the rotational misalignment between real and diffraction space may also be necessary.

**Parameters**

• **g1g2_map** (RealSlice) – the lattice vector map; contains 2D arrays in \( g1g2\_map\_data \) under the keys ‘g1x’, ‘g1y’, ‘g2x’, and ‘g2y’. See documentation for fit_lattice_vectors_all_DPs() for more information.

• **g1** (2-tuple) – first reference lattice vector (x,y)

• **g2** (2-tuple) – second reference lattice vector (x,y)

**Returns**

(RealSlice) the strain map; contains the elements of the infinitessimal strain matrix, in the following 5 arrays:

• \( \text{strain\_map.get\_slice('e_xx')} \): change in lattice x-components with respect to x

• \( \text{strain\_map.get\_slice('e_yy')} \): change in lattice y-components with respect to y

• \( \text{strain\_map.get\_slice('e_xy')} \): change in lattice x-components with respect to y

• \( \text{strain\_map.get\_slice('theta')} \): rotation of lattice with respect to reference

• \( \text{strain\_map.get\_slice('mask')} \): 0/False indicates unknown values

Note 1: the strain matrix has been symmetrized, so \( e_{xy} \) and \( e_{yx} \) are identical

py4DSTEM.process.latticevectors.strain.get_strain_from_reference_region(g1g2_map, mask)

Gets a strain map from the reference region of real space specified by mask and the lattice vector map \( g1g2\_map \).

Note that this function will return the strain map oriented with respect to the x/y axes of diffraction space - to rotate the coordinate system, use get_rotated_strain_map(). Calibration of the rotational misalignment between real and diffraction space may also be necessary.

**Parameters**

• **g1g2_map** (RealSlice) – the lattice vector map; contains 2D arrays in \( g1g2\_map\_data \) under the keys ‘g1x’, ‘g1y’, ‘g2x’, and ‘g2y’. See documentation for fit_lattice_vectors_all_DPs() for more information.

• **mask** (ndarray of bools) – use lattice vectors from \( g1g2\_map\) scan positions wherever mask==True

**Returns**

(RealSlice) the strain map; contains the elements of the infinitessimal strain matrix, in the following 5 arrays:

• \( \text{strain\_map.get\_slice('e_xx')} \): change in lattice x-components with respect to x

• \( \text{strain\_map.get\_slice('e_yy')} \): change in lattice y-components with respect to y

• \( \text{strain\_map.get\_slice('e_xy')} \): change in lattice x-components with respect to y
\begin{itemize}
  \item \texttt{strain_map.get_slice('theta')}: rotation of lattice with respect to reference
  \item \texttt{strain_map.get_slice('mask')}: 0/False indicates unknown values
\end{itemize}

Note 1: the strain matrix has been symmetrized, so \( e_{xy} \) and \( e_{yx} \) are identical

\begin{verbatim}
py4DSTEM.process.latticevectors.strain.get_rotated_strain_map(unrotated_strain_map, xaxis_x, xaxis_y, flip_theta)
\end{verbatim}

Starting from a strain map defined with respect to the xy coordinate system of diffraction space, i.e. where \( e_{xx} \) and \( e_{yy} \) are the compression/tension along the \( Q_x \) and \( Q_y \) directions, respectively, get a strain map defined with respect to some other right-handed coordinate system, in which the x-axis is oriented along \((xaxis_x, xaxis_y)\).

**Parameters**

\begin{itemize}
  \item \texttt{xaxis_x (float)} – diffraction space (x,y) coordinates of a vector along the new x-axis
  \item \texttt{xaxis_y (float)} – diffraction space (x,y) coordinates of a vector along the new x-axis
  \item \texttt{unrotated_strain_map (RealSlice)} – a RealSlice object containing 2D arrays of the infinitessimal strain matrix elements, stored at
    \begin{itemize}
      \item \texttt{unrotated_strain_map.get_slice('e_xx')}
      \item \texttt{unrotated_strain_map.get_slice('e_xy')}
      \item \texttt{unrotated_strain_map.get_slice('e_yy')}
      \item \texttt{unrotated_strain_map.get_slice('theta')}
    \end{itemize}
\end{itemize}

**Returns**

\begin{verbatim}
(RealSlice) the rotated counterpart to unrotated_strain_map, with the rotated_strain_map.get_slice('e_xx') element oriented along the new coordinate system
\end{verbatim}

\textbf{phase}

Module for reconstructing phase objects from 4DSTEM datasets using iterative methods.

\begin{verbatim}
class py4DSTEM.process.phase.iterative_base_class.PhaseReconstruction
\end{verbatim}

Base phase reconstruction class. Defines various common functions and properties for all subclasses to inherit, as well as sets up various abstract methods each subclass must define.

\begin{verbatim}
abstract preprocess()
\end{verbatim}

Abstract method subclasses must define which prepares measured intensities.

For DPC, this includes:
- Fitting diffraction intensitie’s CoM and rotation
- Preparing Fourier-coordinates and operators

For Ptychography, this includes:
- Centering diffraction intensities using fitted CoM
- Padding diffraction intensities to region of interest dimensions
- Preparing initial guess for scanning positions
- Preparing initial guesses for the objects and probes arrays

\begin{verbatim}
abstract reconstruct()
\end{verbatim}

Abstract method subclasses must define which performs the reconstruction by calling the subclass \_forward(), \_adjoint(), and \_update() methods.

\begin{verbatim}
abstract visualize()
\end{verbatim}

Abstract method subclasses must define to postprocess and display results.

\begin{verbatim}
tune_angle_and_defocus(angle_guess=None, defocus_guess=None, transpose=None, angle_step_size=1, defocus_step_size=20, num_angle_values=5, num_defocus_values=5, max_iter=5, plot_reconstructions=True, plot_convergence=True, return_values=False, **kwargs)
\end{verbatim}
Run reconstructions over a parameters space of angles and defocus values.

**Parameters**

- **angle_guess** *(float (degrees), optional)* – initial starting guess for rotation angle between real and reciprocal space if None, uses current initialized values
- **defocus_guess** *(float (A), optional)* – initial starting guess for defocus if None, uses current initialized values
- **angle_step_size** *(float (degrees), optional)* – size of change of rotation angle between real and reciprocal space for each step in parameter space
- **defocus_step_size** *(float (A), optional)* – size of change of defocus for each step in parameter space
- **num_angle_values** *(int, optional)* – number of values of angle to test, must be >= 1.
- **num_defocus_values** *(int, optional)* – number of values of defocus to test, must be >= 1
- **max_iter** *(int, optional)* – number of iterations to run in ptychographic reconstruction
- **plot_reconstructions** *(bool, optional)* – if True, plot phase of reconstructed objects
- **plot_convergence** *(bool, optional)* – if True, plots error for each iteration for each reconstruction.
- **return_values** *(bool, optional)* – if True, returns objects, convergence

**Returns**

- **objects** *(list)* – reconstructed objects
- **convergence** *(np.ndarray)* – array of convergence values from reconstructions

**plot_position_correction**(scale_arrows=1, verbose=True, **kwargs)

Function to plot changes to probe positions during ptychography reconstruction.

**Parameters**

- **scale** *(float, optional)* – scaling of quiver arrows
- **verbose** *(bool, optional)* – if True, prints AffineTransformation if positions have been updated

**plot_fourier_probe**(probe=None, cbar=True, scalebar=True, pixelsize=None, pixelunits=None, **kwargs)

Plot probe in fourier space.

**Parameters**

- **probe** *(complex array, optional)* – if None is specified, uses the **probe_fourier** property
- **cbar** *(bool, optional)* – if True, adds colorbar
- **scalebar** *(bool, optional)* – if True, adds scalebar to probe
- **pixelunits** *(str, optional)* – units for scalebar, default is A^-1
- **pixelsize** *(float, optional)* – default is probe reciprocal sampling
Module for reconstructing phase objects from 4DSTEM datasets using iterative methods, namely DPC.

**class** py4DSTEM.process.phase.iterative_dpc.DPCReconstruction

**Parameters**

- **datacube** (*DataCube*) – Input 4D diffraction pattern intensities
- **dp_mask** (*ndarray, optional*) – Mask for datacube intensities (Qx,Qy)
- **energy** (*float, optional*) – The electron energy of the wave functions in eV
- **verbose** (*bool, optional*) – If True, class methods will inherit this and print additional information
- **device** (*str, optional*) – Calculation device will be performed on. Must be 'cpu' or 'gpu'

**Assigns** –

- **self._xp** (*Callable*) – Array computing module
- **self._asnumpy** (*Callable*) – Array conversion module to numpy
- **self._region_of_interest_shape** – None, i.e. same as diffraction intensities (Qx,Qy)
- **self._preprocessed** (*bool*) – Flag to signal object has not yet been preprocessed

**__init__**(*datacube: DataCube, dp_mask: ndarray | None = None, energy: float | None = None, verbose: bool = True, device: str = 'cpu')
force_com_rotation: float | None = None, force_com_transpose: bool | None = None,
force_com_shifts: float | None = None, plot_center_of_mass: str = 'default', plot_rotation: bool = True, **kwargs)

DPC preprocessing step. Calls the base class methods:
_extract_intensities_and_calibrations_from_datacube(), _calculate_intensities_center_of_mass(), and _solve_for_center_of_mass_relative_rotation()

Parameters

- **rotation_angles_deg** (np.darray, optional) – Array of angles in degrees to perform curl minimization over
- **maximize_divergence** (bool, optional) – If True, the divergence of the CoM gradient vector field is maximized
- **fit_function** (str, optional) – 2D fitting function for CoM fitting. One of 'plane', 'parabola', 'bezier_two'
- **com_rotation** (force) – Force relative rotation angle between real and reciprocal space
- **force_comtranspose** (bool (optional)) – Force whether diffraction intensities need to be transposed.
- **force_com_shifts** (tuple of ndarrays (CoMx, CoMy)) – Force CoM fitted shifts
- **plot_center_of_mass** (str, optional) – If ‘default’, the corrected CoM arrays will be displayed If ‘all’, the computed and fitted CoM arrays will be displayed
- **plot_rotation** (bool, optional) – If True, the CoM curl minimization search result will be displayed
- **Mutates** –
- -------- _
- **self._preprocessed** (bool) – Flag to signal object has been preprocessed

Returns

self – Self to accommodate chaining

Return type

DPCReconstruction

reconstruct(reset: bool | None = None, padding_factor: float = 2, max_iter: int = 64, step_size: float = 1.0, stopping_criterion: float = 1e-06, progress_bar: bool = True, gaussian_filter_sigma: float | None = None, gaussian_filter_iter: int = inf, butterworth_filter_iter: int = inf, q_lowpass: float | None = None, q_highpass: float | None = None, store_iterations: bool = False)

Performs Iterative DPC Reconstruction:
Parameters

- **reset** *(bool, optional)* – If True, previous reconstructions are ignored
- **padding_factor** *(float, optional)* – Factor to pad object by to reduce periodic artifacts
- **max_iter** *(int, optional)* – Maximum number of iterations
- **step_size** *(float, optional)* – Reconstruction update step size
- **stopping_criterion** *(float, optional)* – Step size below which reconstruction exits
- **progress_bar** *(bool, optional)* – If True, reconstruction progress bar will be printed
- **gaussian_filter_sigma** *(float, optional)* – Standard deviation of gaussian kernel
- **gaussian_filter_iter** *(int, optional)* – Number of iterations to run using object smoothness constraint
- **butterworth_filter_iter** *(int, optional)* – Number of iterations to run using high-pass butterworth filter
- **q_lowpass** *(float)* – Cut-off frequency in A^-1 for low-pass butterworth filter
- **q_highpass** *(float)* – Cut-off frequency in A^-1 for high-pass butterworth filter
- **store_iterations** *(bool, optional)* – If True, all reconstruction iterations will be stored
- **Assigns** –
  - --------
  - **self._object_phase** *(xp.ndarray)* – Reconstructed phase object, on calculation device
  - **self.object_phase** *(np.ndarray)* – Reconstructed phase object, as a numpy array
  - **self.error** *(float)* – RMS error
  - **self.object_phase_iterations** – Reconstructed phase objects at each iteration as numpy arrays
  - **optional** – Reconstructed phase objects at each iteration as numpy arrays
  - **self.error_iterations** – RMS errors at each iteration
  - **optional** – RMS errors at each iteration

Returns

**self** – Self to accommodate chaining

Return type

*DPCReconstruction*

**visualize**(iterations_grid: Tuple[int, int] | None = None, plot_convergence: bool = True, char: bool = False, **kwargs)

Displays reconstructed phase object.

Parameters

- **plot_convergence** *(bool, optional)* – If true, the RMS error plot is displayed
• **iterations_grid** (*Tuple[int, int]*) – Grid dimensions to plot reconstruction iterations

• **cbar** (*bool, optional*) – If true, displays a colorbar

**Returns**

* self – Self to accommodate chaining

**Return type**

*DPCReconstruction*

**property sampling**

Sampling [Å]

Module for reconstructing phase objects from 4DSTEM datasets using iterative methods, namely mixed-state ptychography.
class py4DSTEM.process.phase.iterative_mixedstate_ptychography.MixedStatePtychographicReconstruction(
    datacube: DataCube, 
    energy: float, 
    num_probes: int | None = None, 
    semi_angle_cutoff: float | None = None, 
    rolloff: float = 2.0, 
    vacuum_probe_intensity: ndarray | None = None, 
    polar_parameters: Mapping[str, float] | None = None, 
    diffraction_intensities_shape: Tuple[int, int] | None = None, 
    reshaping_method: str = '/quotesingle.ts1 fourier/quotesingle.ts1', 
    probe_roi_shape: Tuple[int, int] | None = None, 
    object_padding_px: Tuple[int, int] | None = None, 
    dp_mask: ndarray | None = None, 
    initial_object_guess: ndarray | None = None, 
    initial_probe_guess: ndarray | None = None, 
    initial_scan_positions: ndarray | None = None, 
    verbose: bool = True, 
    device: str = '/quotesingle.ts1 cpu/quotesingle.ts1', 
    **kwargs)

Mixed-State Ptychographic Reconstruction Class.

Diffraction intensities dimensions: \((R_x, R_y, Q_x, Q_y)\) Reconstructed probe dimensions: \((N, S_x, S_y)\) Reconstructed object dimensions: \((P_x, P_y)\)

such that \((S_x, S_y)\) is the region-of-interest (ROI) size of our \(N\) probes and \((P_x, P_y)\) is the padded-object size we position our ROI around in.

**Parameters**

- **datacube** (*DataCube*) – Input 4D diffraction pattern intensities
- **energy** (*float*) – The electron energy of the wave functions in eV
- **num_probes** (*int, optional*) – Number of mixed-state probes
- **semiangle_cutoff** (*float, optional*) – Semiangle cutoff for the initial probe guess
- **rolloff** (*float, optional*) – Semiangle rolloff for the initial probe guess
- **vacuum_probe_intensity** (*np.ndarray, optional*) – Vacuum probe to use as intensity aperture for initial probe guess
- **polar_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.
- **diffraction_intensities_shape** (*Tuple[int, int], optional*) – Pixel dimensions \((Q_x', Q_y')\) of the resampled diffraction intensities If None, no resampling of diffraction intensities is performed
- **reshaping_method** (*str, optional*) – Method to use for reshaping, either ‘bin’, ‘bilinear’, or ‘fourier’ (default)
- **probe_roi_shape** – Padded diffraction intensities shape. If None, no padding is performed
- **(int – Padded diffraction intensities shape. If None, no padding is performed
- **int) – Padded diffraction intensities shape. If None, no padding is performed
- **optional** – Padded diffraction intensities shape. If None, no padding is performed
- **object_padding_px** (*Tuple[int, int], optional*) – Pixel dimensions to pad object with If None, the padding is set to half the probe ROI dimensions
- **dp_mask** (*ndarray, optional*) – Mask for datacube intensities \((Q_x, Q_y)\)
- **initial_object_guess** (*np.ndarray, optional*) – Initial guess for complex-valued object of dimensions \((P_x, P_y)\) If None, initialized to 1.0j
- **initial_probe_guess** (*np.ndarray, optional*) – Initial guess for complex-valued probe of dimensions \((S_x, S_y)\). If None, initialized to ComplexProbe with semiangle_cutoff, energy, and aberrations
- **initial_scan_positions** (*np.ndarray, optional*) – Probe positions in Å for each diffraction intensity If None, initialized to a grid scan
- **verbose** (*bool, optional*) – If True, class methods will inherit this and print additional information
- **device** (*str, optional*) – Calculation device will be perfomed on. Must be ‘cpu’ or ‘gpu’
- **kwargs** – Provide the aberration coefficients as keyword arguments.
__init__(datacube: DataCube, energy: float, num_probes: int | None = None, num_energy: int | None = None, rolloff: float = 2.0, vacuum_probe_intensity: ndarray | None = None, object_padding_px: Tuple[int, int] | None = None, rolloff: float = 2.0, polar_parameters: Mapping[str, float] | None = None, **kwargs)

preprocess(fit_function: str = 'plane', plot_center_of_mass: str = 'default', plot_rotation: bool = True, maximize_divergence: bool = False, rotation_angles_deg: np.darray | None = None, plot_probe_overlaps: bool = True, force_com_rotation: float | None = None, force_com_transpose: float | None = None, force_com_shifts: float | None = None, **kwargs)

Ptychographic preprocessing step. Calls the base class methods:

_extract_intensities_and_calibrations_from_datacube_, _compute_center_of_mass(), _solve_CoM_rotation(), _normalize_diffraction_intensities() _calculate_scan_positions_in_px()

Additionally, it initializes an (Px, Py) array of 1.0j and a complex probe using the specified polar parameters.

Parameters

- **fit_function** (str, optional) – 2D fitting function for CoM fitting. One of ‘plane’, ‘parabola’, ‘bezier_two’
- **plot_center_of_mass** (str, optional) – If ‘default’, the corrected CoM arrays will be displayed. If ‘all’, the computed and fitted CoM arrays will be displayed
- **plot_rotation** (bool, optional) – If True, the CoM curl minimization search result will be displayed
- **maximize_divergence** (bool, optional) – If True, the divergence of the CoM gradient vector field is maximized
- **rotation_angles_deg** (np.darray, optional) – Array of angles in degrees to perform curl minimization over
- **plot_probe_overlaps** (bool, optional) – If True, initial probe overlaps scanned over the object will be displayed
- **force_com_rotation** (float (degrees), optional) – Force relative rotation angle between real and reciprocal space
- **force_com_transpose** (bool, optional) – Force whether diffraction intensities need to be transposed.
- **force_com_shifts** (tuple of ndarrays (CoMx, CoMy)) – Amplitudes come from diffraction patterns shifted with the CoM in the upper left corner for each probe unless shift is overwritten.
Returns

self – Self to accommodate chaining

Return type

PtychographicReconstruction

reconstruct(max_iter: int = 64, reconstruction_method: str = 'gradient-descent',
reconstruction_parameter: float = 1.0, max_batch_size: int | None = None, seed_random: int | None = None, step_size: float = 0.9, normalization_min: float = 0.001, positions_step_size: float = 0.9, pure_phase_object_iter: int = 0, fix_com: bool = True, orthogonalize_probe: bool = True, fix_probe_iter: int = 0, fix_probe_fourier_amplitude_iter: int = 0, fix_positions_iter: int = inf, global_affine_transformation: bool = True, probe_support_relative_radius: float = 1.0, probe_support_supergaussian_degree: float = 10.0, gaussian_filter_sigma: float | None = None, gaussian_filter_iter: int = inf, butterworth_filter_iter: int = inf, q_lowpass: float | None = None, q_highpass: float | None = None, store_iterations: bool = False, progress_bar: bool = True, reset: bool | None = None)

Ptychographic reconstruction main method.

Parameters

- **max_iter** (int, optional) – Maximum number of iterations to run
- **reconstruction_method** (str, optional) – Specifies which reconstruction algorithm to use, one of: “generalized-projection”, “DM_AP” (or “difference-map_alternating-projections”), “RAAR” (or “relaxed-averaged-alternating-reflections”), “RRR” (or “relax-reflect-reflect”), “SUPERFLIP” (or “charge-flipping”), or “GD” (or “gradient_descent”)
- **reconstruction_parameter** (float, optional) – Reconstruction parameter for various reconstruction methods above.
- **max_batch_size** (int, optional) – Max number of probes to update at once
- **seed_random** (int, optional) – Seeds the random number generator, only applicable when max_batch_size is not None
- **step_size** (float, optional) – Update step size
- **normalization_min** (float, optional) – Probe normalization minimum as a fraction of the maximum overlap intensity
- **positions_step_size** (float, optional) – Positions update step size
- **pure_phase_object** (bool, optional) – If True, object amplitude is set to unity
- **fix_com** (bool, optional) – If True, fixes center of mass of probe
- **fix_probe_iter** (int, optional) – Number of iterations to run with a fixed probe before updating probe estimate
- **fix_probe_amplitude** (int, optional) – Number of iterations to run with a fixed probe amplitude
- **fix_positions_iter** (int, optional) – Number of iterations to run with fixed positions before updating positions estimate
- **global_affine_transformation** (bool, optional) – If True, positions are assumed to be a global affine transform from initial scan
- **probe_support_relative_radius** (float, optional) – Radius of probe supergaussian support in scaled pixel units, between (0,1]
• **probe_support_supergaussian_degree** (*float, optional*) – Degree super-gaussian support is raised to, higher is sharper cutoff

• **gaussian_filter_sigma** (*float, optional*) – Standard deviation of gaussian kernel

• **gaussian_filter_iter** (*int, optional*) – Number of iterations to run using object smoothness constraint

• **butterworth_filter_iter** (*int, optional*) – Number of iterations to run using high-pass butteworth filter

• **q_lowpass** (*float*) – Cut-off frequency in Å^-1 for low-pass butterworth filter

• **q_highpass** (*float*) – Cut-off frequency in Å^-1 for high-pass butterworth filter

• **store_iterations** (*bool, optional*) – If True, reconstructed objects and probes are stored at each iteration

• **progress_bar** (*bool, optional*) – If True, reconstruction progress is displayed

• **reset** (*bool, optional*) – If True, previous reconstructions are ignored

**Returns**

self – Self to accommodate chaining

**Return type**

*PtychographicReconstruction*

**visualize**(*iterations_grid* (*Tuple[int, int]*) | None = None, **plot_convergence** *bool* = True, **plot_probe** *bool* = True, **object_mode** *str* = 'phase', **cbar** *bool* = True, **padding** *int* = 0, **relative_error** *bool* = True, **kwargs**)

Displays reconstructed object and probe.

**Parameters**

• **plot_convergence** (*bool, optional*) – If true, the RMS error plot is displayed

• **iterations_grid** (*Tuple[int, int]*) – Grid dimensions to plot reconstruction iterations

• **cbar** (*bool, optional*) – If true, displays a colorbar

• **plot_probe** (*bool*) – If true, the reconstructed probe intensity is also displayed

• **object_mode** (*str*) – Specifies the attribute of the object to plot. One of ‘phase’, ‘amplitude’, ‘intensity’

• **relative_error** (*bool*) – Sets the error to be relative to the first iteration. TODO - update to be relative to empty object wave error (RMS of all measurements).

**Returns**

self – Self to accommodate chaining

**Return type**

*PtychographicReconstruction*

**plot_fourier_probe**(*probe* = None, **scalebar** = True, **pixelsize** = None, **pixelunits** = None, **kwargs**)

Plot probe in fourier space

**Parameters**

• **probe** (*complex array, optional*) – if None is specified, uses the *probe_fourier* property
• **scalebar** (*bool, optional*) – if True, adds scalebar to probe
• **pixelunits** (*str, optional*) – units for scalebar, default is Å⁻¹
• **pixelsize** (*float, optional*) – default is probe reciprocal sampling

Module for reconstructing phase objects from 4DSTEM datasets using iterative methods, namely multislice ptychography.
class py4STEM.process.phase.iterative_multislice_ptychography.MultislicePtychographicReconstruction:

datacube: DataCube,
energy: float,
num_slices: int,
slice_thicknesses: float | Sequence[float],
semi_angle_cutoff: float | None = None,
rolloff: float = 2.0,
vacuum_probe_intensity: ndarray | None = None,
polar_parameters: Mapping[str, float] | None = None,
diffraction_intensities_shape: Tuple[int, int] | None = None,
reshaping_method: str = '/quotesingle.ts1 fourier',
probe_roi_shape: Tuple[int, int] | None = None,
object_padding_px: Tuple[int, int] | None = None,
dp_mask: ndarray | None = None,
initial_object_guess: ndarray | None = None,
initial_probe_guess: ndarray | None = None,
initial_scan_positions: ndarray | None = None,
verbose: bool = True,
device: str = '/quotesingle.ts1 cpu',
**kwargs)
Multislice Ptychographic Reconstruction Class.

Diffraction intensities dimensions : $(Rx, Ry, Qx, Qy)$ Reconstructed probe dimensions : $(Sx, Sy)$ Reconstructed object dimensions : $(T, Px, Py)$

such that $(Sx, Sy)$ is the region-of-interest (ROI) size of our probe and $(Px, Py)$ is the padded-object size we position our ROI around in each of the $T$ slices.

Parameters

- **datacube** (*DataCube*) – Input 4D diffraction pattern intensities
- **energy** (*float*) – The electron energy of the wave functions in eV
- **num_slices** (*int*) – Number of slices to use in the forward model
- **slice_thicknesses** (*float or Sequence[float]*) – Slice thicknesses. If float, all slices are assigned the same thickness
- **semiangle_cutoff** (*float, optional*) – Semiangle cutoff for the initial probe guess
- **rolloff** (*float, optional*) – Semiangle rolloff for the initial probe guess
- **vacuum_probe_intensity** (*np.ndarray, optional*) – Vacuum probe to use as intensity aperture for initial probe guess
- **polar_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.
- **diffraction_intensities_shape** (*Tuple[int, int], optional*) – Pixel dimensions $(Qx', Qy')$ of the resampled diffraction intensities If None, no resampling of diffraction intensities is performed
- **reshaping_method** (*str, optional*) – Method to use for reshaping, either 'bin', 'bilinear', or 'fourier' (default)
- **probe_roi_shape** – Padded diffraction intensities shape. If None, no padding is performed
- **(int – Padded diffraction intensities shape. If None, no padding is performed**
- **int) – Padded diffraction intensities shape. If None, no padding is performed
- **optional – Padded diffraction intensities shape. If None, no padding is performed**
- **object_padding_px** (*Tuple[int, int], optional*) – Pixel dimensions to pad object with If None, the padding is set to half the probe ROI dimensions
- **dp_mask** (*ndarray, optional*) – Mask for datacube intensities $(Qx, Qy)$
- **initial_object_guess** (*np.ndarray, optional*) – Initial guess for complex-valued object of dimensions $(Px, Py)$ If None, initialized to 1.0j
- **initial_probe_guess** (*np.ndarray, optional*) – Initial guess for complex-valued probe of dimensions $(Sx, Sy)$. If None, initialized to ComplexProbe with semiangle_cutoff, energy, and aberrations
- **initial_scan_positions** (*np.ndarray, optional*) – Probe positions in Å for each diffraction intensity If None, initialized to a grid scan
- **verbose** (*bool, optional*) – If True, class methods will inherit this and print additional information
device (str, optional) – Calculation device will be performed on. Must be ‘cpu’ or ‘gpu’

**kwargs – Provide the aberration coefficients as keyword arguments.

__init__(datacube: DataCube, energy: float, num_slices: int, slice_thicknesses: float | Sequence[float],
         semiangle_cutoff: float | None = None, rolloff: float = 2.0, vacuum_probe_intensity: ndarray | None = None,
         polar_parameters: Mapping[str, float] | None = None, diffraction_intensities_shape: Tuple[int, int] | None = None,
         reshaping_method: str = 'fourier', probe_roi_shape: Tuple[int, int] | None = None,
         object_padding_px: Tuple[int, int] | None = None,
         dp_mask: ndarray | None = None, initial_object_guess: ndarray | None = None,
         initial_probe_guess: ndarray | None = None, initial_scan_positions: ndarray | None = None,
         verbose: bool = True, **kwargs)

preprocess(fit_function: str = 'plane', plot_center_of_mass: str = 'default', plot_rotation: bool = True,
           maximize_divergence: bool = False, rotation_angles_deg: np.darray | float = array([-89., -88., -87.,
                                  82., 83., 84., 85., 86., 87., 88., 89.]), plot_probe_overlaps: bool = True,
           force_com_rotation: float = None, force_com_transpose: float = None, force_com_shifts: float = None, **kwargs)

Ptychographic preprocessing step. Calls the base class methods:

 extract_intensities_and_calibrations_from_datacube, compute_center_of_mass(), solve_CoM_rotation(), normalize_diffraction_intensities() calculate_scan_positions_in_px()

Additionally, it initializes an (T,Px,Py) array of 1.0j and a complex probe using the specified polar parameters.

Parameters

  * fit_function (str, optional) – 2D fitting function for CoM fitting. One of ‘plane’, ‘parabola’, ‘bezier_two’
  * plot_center_of_mass (str, optional) – If ‘default’, the corrected CoM arrays will be displayed If ‘all’, the computed and fitted CoM arrays will be displayed
  * plot_rotation (bool, optional) – If True, the CoM curl minimization search result will be displayed
  * maximize_divergence (bool, optional) – If True, the divergence of the CoM gradient vector field is maximized
  * rotation_angles_deg (np.darray, optional) – Array of angles in degrees to perform curl minimization over
  * plot_probe_overlaps (bool, optional) – If True, initial probe overlaps scanned over the object will be displayed
  * force_com_rotation (float (degrees), optional) – Force relative rotation angle between real and reciprocal space
  * force_com_transpose (bool, optional) – Force whether diffraction intensities need to be transposed.
• **force_com_shifts** *(tuple of ndarrays (CoMx, CoMy))* – Amplitudes come from diffraction patterns shifted with the CoM in the upper left corner for each probe unless shift is overwritten.

**Returns**

self – Self to accommodate chaining

**Return type**

`MultislicePtychographicReconstruction`

**reconstruct** *(max_iter: int = 64, reconstruction_method: str = ‘gradient-descent’,
reconstruction_parameter: float = 1.0, max_batch_size: int | None = None, seed_random: int | None = None, step_size: float = 0.9, normalization_min: float = 0.001, positions_step_size: float = 0.9, pure_phase_object_iter: int = 0, fix_com: bool = True, fix_probe_iter: int = 0,
fix_probe_fourier_amplitude_iter: int = 0, fix_positions_iter: int = inf,
global_affine_transformation: bool = True, probe_support_relative_radius: float = 1.0,
probe_support_supergaussian_degree: float = 10.0, gaussian_filter_sigma: float | None = None, butterworth_filter_iter: int = inf, q_lowpass: float | None = None,
kz_regularization_filter_iter: int = inf, kz_regularization_gamma: float | None = None, identical_slices_iter: int = 0, store_iterations: bool = False, progress_bar: bool = True, reset: bool | None = None)*

Ptychographic reconstruction main method.

**Parameters**

• **max_iter** *(int, optional)* – Maximum number of iterations to run

• **reconstruction_method** *(str, optional)* – Specifies which reconstruction algorithm to use, one of: “generalized-projection”, “DM-AP” (or “difference-map-alternating-projections”), “RAAR” (or “relaxed-averaged-alternating-reflections”), “RRR” (or “relax-reflect-reflect”), “SUPERFLIP” (or “charge-flipping”), or “GD” (or “gradient_descent”)

• **reconstruction_parameter** *(float, optional)* – Reconstruction parameter for various reconstruction methods above.

• **reconstruction_parameter** – Tuning parameter to interpolate b/w DM-AP and DM-RAAR

• **max_batch_size** *(int, optional)* – Max number of probes to update at once

• **seed_random** *(int, optional)* – Seeds the random number generator, only applicable when max_batch_size is not None

• **step_size** *(float, optional)* – Update step size

• **normalization_min** *(float, optional)* – Probe normalization minimum as a fraction of the maximum overlap intensity

• **positions_step_size** *(float, optional)* – Positions update step size

• **pure_phase_object** *(bool, optional)* – If True, object amplitude is set to unity

• **fix_com** *(bool, optional)* – If True, fixes center of mass of probe

• **fix_probe_iter** *(int, optional)* – Number of iterations to run with a fixed probe before updating probe estimate

• **fix_probe_amplitude** *(int, optional)* – Number of iterations to run with a fixed probe amplitude

• **fix_positions_iter** *(int, optional)* – Number of iterations to run with fixed positions before updating positions estimate
• `global_affine_transformation` *(bool, optional)* – If True, positions are assumed to be a global affine transform from initial scan

• `probe_support_relative_radius` *(float, optional)* – Radius of probe supergaussian support in scaled pixel units, between (0,1]

• `probe_support_supergaussian_degree` *(float, optional)* – Degree supergaussian support is raised to, higher is sharper cutoff

• `gaussian_filter_sigma` *(float, optional)* – Standard deviation of gaussian kernel

• `gaussian_filter_iter` *(int, optional)* – Number of iterations to run using object smoothness constraint

• `butterworth_filter_iter` *(int, optional)* – Number of iterations to run using high-pass butterworth filter

• `q_lowpass` *(float)* – Cut-off frequency in Å⁻¹ for low-pass butterworth filter

• `q_highpass` *(float)* – Cut-off frequency in Å⁻¹ for high-pass butterworth filter

• `kz_regularization_filter_iter` *(int, optional)* – Number of iterations to run using kz regularization filter

• `kz_regularization_gamma` – `kz` regularization strength

• `float` – `kz` regularization strength

• `optional` – `kz` regularization strength

• `identical_slices_iter` *(int, optional)* – Number of iterations to run using identical slices

• `store_iterations` *(bool, optional)* – If True, reconstructed objects and probes are stored at each iteration

• `progress_bar` *(bool, optional)* – If True, reconstruction progress is displayed

• `reset` *(bool, optional)* – If True, previous reconstructions are ignored

Returns

self – Self to accommodate chaining

Return type

`MultislicePtychographicReconstruction`

`visualize(iterations_grid: Tuple[int, int] | None = None, plot_convergence: bool = True, plot_probe: bool = True, object_mode: str = 'phase', cbar: bool = True, padding: int = 0, relative_error: bool = True, **kwargs)`

Displays reconstructed object and probe.

Parameters

• `plot_convergence` *(bool, optional)* – If true, the RMS error plot is displayed

• `iterations_grid` *(Tuple[int, int])* – Grid dimensions to plot reconstruction iterations

• `cbar` *(bool, optional)* – If true, displays a colorbar

• `plot_probe` *(bool)* – If true, the reconstructed probe intensity is also displayed

• `object_mode` *(str)* – Specifies the attribute of the object to plot. One of ‘phase’, ‘amplitude’, ‘intensity’
• `relative_error (bool)` – Sets the error to be relative to the first iteration. TODO -
  update to be relative to empty object wave error (RMS of all measurements).

**Returns**

- `self` – Self to accommodate chaining

**Return type**

`PtychographicReconstruction`

Module for reconstructing phase objects from 4DSTEM datasets using iterative methods, namely overlap tomography.
class py4DSTEM.process.phase.iterative_overlap_tomography.OverlapTomographicReconstruction:

datacube: Sequence[DataCube],
energy: float,
num_slices: int,
tilt_angles_degrees: Sequence[float],
semi_angle_cutoff: float
| None = None,
rolloff: float = 2.0,
vacuum_probe_intensity: ndarray | None = None,
polar_parameters: Mapping[str, float] | None = None,
diffraction_intensities_shape: Tuple[int, int] | None = None,
resizing_method: str = 'fourier',
probe_roi_shape: Tuple[int, int] |
Overlap Tomographic Reconstruction Class.

List of diffraction intensities dimensions : (Rx,Ry,Qx,Qy) Reconstructed probe dimensions : (Sx,Sy) Reconstructed object dimensions : (Px,Py,Py) such that (Sx,Sy) is the region-of-interest (ROI) size of our probe and (Px,Py,Py) is the padded-object electrostatic potential volume, where x-axis is the tilt.

Parameters

- **datacube** *(List of DataCubes)* – Input list of 4D diffraction pattern intensities
- **energy** *(float)* – The electron energy of the wave functions in eV
- **num_slices** *(int)* – Number of slices to use in the forward model
- **tilt_angles_deg** *(Sequence[float])* – List of tilt angles in degrees,
- **semiangle_cutoff** *(float, optional)* – Semiangle cutoff for the initial probe guess
- **rolloff** *(float, optional)* – Semiangle rolloff for the initial probe guess
- **vacuum_probe_intensity** *(np.ndarray, optional)* – Vacuum probe to use as intensity aperture for initial probe guess
- **polar_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.
- **diffraction_intensities_shape** *(Tuple[int,int], optional)* – Pixel dimensions (Qx’,Qy’) of the resampled diffraction intensities If None, no resampling of diffraction intensities is performed
- **reshaping_method** *(str, optional)* – Method to use for reshaping, either ’bin’, ’bilinear’, or ’fourier’ (default)
- **probe_roi_shape** – Padded diffraction intensities shape. If None, no padding is performed
- **(int** – Padded diffraction intensities shape. If None, no padding is performed
- **int** – Padded diffraction intensities shape. If None, no padding is performed
- **optional** – Padded diffraction intensities shape. If None, no padding is performed
- **object_padding_px** *(Tuple[int,int], optional)* – Pixel dimensions to pad object with If None, the padding is set to half the probe ROI dimensions
- **dp_mask** *(ndarray, optional)* – Mask for datacube intensities (Qx,Qy)
- **initial_object_guess** *(np.ndarray, optional)* – Initial guess for complex-valued object of dimensions (Px,Py,Py) If None, initialized to 1.0
- **initial_probe_guess** *(np.ndarray, optional)* – Initial guess for complex-valued probe of dimensions (Sx,Sy). If None, initialized to ComplexProbe with semiangle_cutoff, energy, and aberrations
- **initial_scan_positions** *(list of np.ndarray, optional)* – Probe positions in Å for each diffraction intensity per tilt If None, initialized to a grid scan centered along tilt axis
- **verbose** *(bool, optional)* – If True, class methods will inherit this and print additional information
• **device** *(str, optional)* – Calculation device will be perfomed on. Must be ‘cpu’ or ‘gpu’

• **kwargs** – Provide the aberration coefficients as keyword arguments.

```python
__init__(datacube: Sequence[DataCube], energy: float, num_slices: int, tilt_angles_degrees: Sequence[float], semiangle_cutoff: float | None = None, rolloff: float = 2.0, vacuum_probe_intensity: ndarray | None = None, polar_parameters: Mapping[str, float] | None = None, diffraction_intensities_shape: Tuple[int, int] | None = None, reshaping_method: str = 'fourier', probe_roi_shape: Tuple[int, int] | None = None, object_padding_px: Tuple[int, int] | None = None, dp_mask: ndarray | None = None, initial_object_guess: ndarray | None = None, initial_probe_guess: ndarray | None = None, initial_scan_positions: Sequence[ndarray] | None = None, verbose: bool = True, device: str = 'cpu', **kwargs)
```

```
preprocess(fit_function: str='plane', plot_probe_overlaps: bool=True, rotation_real_space_degrees: float | None = None, diffraction_patterns_rotate_degrees: float | None = None, diffraction_patterns_transpose: bool | None = None, force_com_shifts: Sequence[float] | None = None, progress_bar: bool = True, **kwargs)
```

Ptychographic preprocessing step.

Additionally, it initializes an (Px,Py, Py) array of 1.0 and a complex probe using the specified polar parameters.

**Parameters**

• **fit_function** *(str, optional)* – 2D fitting function for CoM fitting. One of ‘plane’, ‘parabola’, ‘bezier_two’

• **plot_probe_overlaps** *(bool, optional)* – If True, initial probe overlaps scanned over the object will be displayed

• **rotation_real_space_degrees** *(float (degrees), optional)* – In plane rotation around z axis between x axis and tilt axis in real space (forced to be in xy plane)

• **diffraction_patterns_rotate_degrees** *(float, optional)* – Relative rotation angle between real and reciprocal space

• **diffraction_patterns_transpose** *(bool, optional)* – Whether diffraction intensities need to be transposed.

• **force_com_shifts** *(list of tuple of ndarrays (CoMx, CoMy))* – Amplitudes come from diffraction patterns shifted with the CoM in the upper left corner for each probe unless shift is overwritten. One tuple per tilt.

**Returns**

self – Self to accommodate chaining

**Return type**

OverlapTomographicReconstruction

```
reconstruct(max_iter: int = 64, reconstruction_method: str = 'gradient-descent', reconstruction_parameter: float = 1.0, max_batch_size: int | None = None, seed_random: int | None = None, step_size: float = 0.9, normalization_min: float = 0.001, positions_step_size: float = 0.9, fix_com: bool = False, fix_probe_iter: int = 0, fix_probe_fourier_amplitude_iter: int = 0, fix_positions_iter: int = inf, global_affine_transformation: bool = True, probe_support_relative_radius: float = 1.0, probe_support_supergaussian_degree: float = 10.0, gaussian_filter_sigma: float | None = None, gaussian_filter_iter: int = inf, butterworth_filter_iter: int = inf, q_lowpass: float | None = None, q_highpass: float | None = None, collective_tilt_updates: bool = False, store_iterations: bool = False, progress_bar: bool = True, reset: bool | None = None)
```

Ptychographic reconstruction main method.
Parameters

- **max_iter** *(int, optional)* – Maximum number of iterations to run
- **reconstruction_method** *(str, optional)* – Specifies which reconstruction algorithm to use, one of: “generalized-projection”, “DM_AP” (or “difference-map_alternating-projections”), “RAAR” (or “relaxed-averaged-alternating-reflections”), “RRR” (or “relax-reflect-reflect”), “SUPERFLIP” (or “charge-flipping”), or “GD” (or “gradient_descent”)
- **reconstruction_parameter** *(float, optional)* – Reconstruction parameter for various reconstruction methods above.
- **reconstruction_parameter** – Tuning parameter to interpolate b/w DM-AP and DM-RAAR
- **max_batch_size** *(int, optional)* – Max number of probes to update at once
- **seed_random** *(int, optional)* – Seeds the random number generator, only applicable when max_batch_size is not None
- **step_size** *(float, optional)* – Update step size
- **normalization_min** *(float, optional)* – Probe normalization minimum as a fraction of the maximum overlap intensity
- **positions_step_size** *(float, optional)* – Positions update step size
- **fix_com** *(bool, optional)* – If True, fixes center of mass of probe
- **fix_probe_iter** *(int, optional)* – Number of iterations to run with a fixed probe before updating probe estimate
- **fix_probe_amplitude** *(int, optional)* – Number of iterations to run with a fixed probe amplitude
- **fix_positions_iter** *(int, optional)* – Number of iterations to run with fixed positions before updating positions estimate
- **global_affine_transformation** *(bool, optional)* – If True, positions are assumed to be a global affine transform from initial scan
- **probe_support_relative_radius** *(float, optional)* – Radius of probe supergaussian support in scaled pixel units, between (0,1]
- **probe_support_supergaussian_degree** *(float, optional)* – Degree supergaussian support is raised to, higher is sharper cutoff
- **gaussian_filter_sigma** *(float, optional)* – Standard deviation of gaussian kernel
- **gaussian_filter_iter** *(int, optional)* – Number of iterations to run using object smoothness constraint
- **butterworth_filter_iter** *(int, optional)* – Number of iterations to run using high-pass butteworth filter
- **q_lowpass** *(float)* – Cut-off frequency in A^-1 for low-pass butterworth filter
- **q_highpass** *(float)* – Cut-off frequency in A^-1 for high-pass butterworth filter
- **store_iterations** *(bool, optional)* – If True, reconstructed objects and probes are stored at each iteration
- **progress_bar** *(bool, optional)* – If True, reconstruction progress is displayed
• **reset** *(bool, optional)* – If True, previous reconstructions are ignored

**Returns**

self – Self to accommodate chaining

**Return type**

`MultislicePtychographicReconstruction`

**visualize** *(iterations_grid: Tuple[int, int]|None=None, plot_convergence: bool = True, plot_probe: bool = True, cbar: bool = True, plot_probe: bool = True, relative_error: bool = True, projection_angle_deg: float | None = None, projection_axes: Tuple[int, int] = (0, 2), x_lims=(None, None), y_lims=(None, None), **kwargs)*

Displays reconstructed object and probe.

**Parameters**

• **plot_convergence** *(bool, optional)* – If true, the RMS error plot is displayed

• **iterations_grid** *(Tuple[int, int])* – Grid dimensions to plot reconstruction iterations

• **cbar** *(bool, optional)* – If true, displays a colorbar

• **plot_probe** *(bool)* – If true, the reconstructed probe intensity is also displayed

• **relative_error** *(bool)* – Sets the error to be relative to the first iteration. TODO - update to be relative to empty object wave error (RMS of all measurements).

**Returns**

self – Self to accommodate chaining

**Return type**

`PtychographicReconstruction`

Module for reconstructing virtual parallax (also known as tilted-shifted bright field) images by aligning each virtual BF image.

**class** `py4DSTEM.process.phase.iterative_parallax.ParallaxReconstruction` *(datacube: DataCube, energy: float, dp_mean: ndarray | None = None, verbose: bool = False, device: str = 'cpu')*

Iterative parallax reconstruction class.

**Parameters**

• **datacube** *(DataCube)* – Input 4D diffraction pattern intensities

• **energy** *(float)* – The electron energy of the wave functions in eV

• **dp_mean** *(ndarray, optional)* – Mean diffraction pattern If None, get_dp_mean() is used

• **verbose** *(bool, optional)* – If True, class methods will inherit this and print additional information

• **device** *(str, optional)* – Calculation device will be performed on. Must be ‘cpu’ or ‘gpu’

**__init__** *(datacube: DataCube, energy: float, dp_mean: ndarray | None = None, verbose: bool = False, device: str = 'cpu')*
py4dstem, Release 0.14.0

preprocess(object_padding_px: Tuple[int, int] = (32, 32), edge_blend: int = 16, threshold_intensity: float = 0.8, normalize_images: bool = True, normalize_order=0, defocus_guess: float | None = None, rotation_guess: float | None = None, plot_average_bf: bool = True, **kwargs)

Iterative parallax reconstruction preprocessing method.

Parameters

- **object_padding_px** *(Tuple[int, int], optional)* – Pixel dimensions to pad object with. If None, the padding is set to half the probe ROI dimensions
- **edge_blend** *(int, optional)* – Pixels to blend image at the border
- **threshold** *(float, optional)* – Fraction of max of dp_mean for bright-field pixels
- **normalize_images** *(bool, optional)* – If True, bright images normalized to have a mean of 1
- **normalize_order** *(integer, optional)* – Polynomial order for normalization. 0 means constant, 1 means linear, etc. Higher orders not yet implemented.
- **defocus_guess** *(float, optional)* – Initial guess of defocus value (defocus dF) in A. If None, first iteration is assumed to be in-focus
- **rotation_guess** *(float, optional)* – Initial guess of defocus value in degrees. If None, first iteration assumed to be 0
- **plot_average_bf** *(bool, optional)* – If True, plots the average bright-field image, using defocus_guess

Returns

* self – Self to accommodate chaining

Return type

* ParallaxReconstruction

**tune_angle_and_defocus**(angle_guess=None, defocus_guess=None, angle_step_size=5, defocus_step_size=100, num_angle_values=5, num_defocus_values=5, return_values=False, plot_reconstructions=True, plot_convergence=True, **kwargs)

Run parallax reconstruction over a parameters space of pre-determined angles and defocus.

Parameters

- **angle_guess** *(float (degrees), optional)* – initial starting guess for rotation angle between real and reciprocal space if None, uses 0
- **defocus_guess** *(float (A), optional)* – initial starting guess for defocus (defocus dF) if None, uses 0
- **angle_step_size** *(float (degrees), optional)* – size of change of rotation angle between real and reciprocal space for each step in parameter space
- **defocus_step_size** *(float (A), optional)* – size of change of defocus for each step in parameter space
- **num_angle_values** *(int, optional)* – number of values of angle to test, must be >= 1.
- **num_defocus_values** *(int, optional)* – number of values of defocus to test, must be >= 1
- **plot_reconstructions** *(bool, optional)* – if True, plot phase of reconstructed objects
• **plot_convergence** (*bool*, *optional*) – if True, makes 2D plot of error metrix

• **return_values** (*bool*, *optional*) – if True, returns objects, convergence

**Returns**

• **objects** (*list*) – reconstructed objects

• **convergence** (*np.ndarray*) – array of convergence values from reconstructions

**reconstruct**(*max_alignment_bin*: *int* | *None* = *None*, *min_alignment_bin*: *int* = *1*, *max_iter_at_min_bin*: *int* = *2*, *upsample_factor*: *int* = *8*, *regularizer_matrix_size*: *Tuple*[int, int] = *(1, 1)*, *regularize_shifts*: *bool* = *True*, *running_average*: *bool* = *True*, *progress_bar*: *bool* = *True*, *plot_aligned_bf*: *bool* = *True*, *plot_convergence*: *bool* = *True*, *reset*: *bool* | *None* = *None*, **kwargs)

Iterative Parallax Reconstruction main reconstruction method.

**Parameters**

• **max_alignment_bin** (*int*, *optional*) – Maximum bin size for bright field alignment. If None, the bright field disk radius is used

• **min_alignment_bin** (*int*, *optional*) – Minimum bin size for bright field alignment

• **max_iter_at_min_bin** (*int*, *optional*) – Number of iterations to run at the smallest bin size

• **upsample_factor** (*int*, *optional*) – DFT upsample factor for subpixel alignment

• **regularizer_matrix_size** (*Tuple*[int, int], *optional*) – Bernstein basis degree used for regularizing shifts

• **regularize_shifts** (*bool*, *optional*) – If True, the cross-correlated shifts are constrained to a spline interpolation

• **running_average** (*bool*, *optional*) – If True, the bright field reference image is updated in a spiral from the origin

• **progress_bar** (*bool*, *optional*) – If True, progress bar is displayed

• **plot_aligned_bf** (*bool*, *optional*) – If True, the aligned bright field image is plotted at each bin level

• **plot_convergence** (*bool*, *optional*) – If True, the convergence error is also plotted

• **reset** (*bool*, *optional*) – If True, the reconstruction is reset

**Returns**

self – Self to accommodate chaining

**Return type**

BFReconstruction

**aberration_fit**(*plot_CTF_compare*: *bool* = *False*, *plot_dk*: *float* = *0.005*, *plot_k_sigma*: *float* = *0.02*)

Fit aberrations to the measured image shifts.

**Parameters**

• **plot_CTF_compare** (*bool*, *optional*) – If True, the fitted CTF is plotted against the reconstructed frequencies

• **plot_dk** (*float*, *optional*) – Reciprocal bin-size for polar-averaged FFT
• **plot_k_sigma** *(float, optional)* – sigma to gaussian blur polar-averaged FFT by

**aberration_correct** *(plot_corrected_phase: bool = True, k_info_limit: float | None = None, k_info_power: float = 1.0, Wiener_filter=False, Wiener_signal_noise_ratio=1.0, Wiener_filter_low_only=False, **kwargs)*

CTF correction of the phase image using the measured defocus aberration.

**Parameters**

• **plot_corrected_phase** *(bool, optional)* – If True, the CTF-corrected phase is plotted

• **k_info_limit** *(float, optional)* – maximum allowed frequency in butterworth filter

• **k_info_power** *(float, optional)* – power of butterworth filter

• **Wiener_filter** *(bool, optional)* – Use Wiener filtering instead of CTF sign correction.

• **Wiener_signal_noise_ratio** *(float, optional)* – Signal to noise radio at k = 0 for Wiener filter

• **Wiener_filter_low_only** *(bool, optional)* – Apply Wiener filtering only to the CTF portions before the 1st CTF maxima.

**depth_section** *(depth_angstroms=array([-250, -150, -50, 50, 150, 250]), plot_depth_sections=True, k_info_limit: float | None = None, k_info_power: float = 1.0, progress_bar=True, **kwargs)*

CTF correction of the BF image using the measured defocus aberration.

**Parameters**

• **depth_angstroms** *(np.array)* – Specify the depths

• **k_info_limit** *(float, optional)* – maximum allowed frequency in butterworth filter

• **k_info_power** *(float, optional)* – power of butterworth filter

**Returns**

• **stack_depth** – stack of phase images at different depths with shape [depth Nx Ny]

**Return type**

np.array

**visualize** *(**kwargs)*

Visualization function for bright field average

**Returns**

• **self** – Self to accommodate chaining

**Return type**

BFReconstruction

Module for reconstructing phase objects from 4DSTEM datasets using iterative methods, namely (single-slice) ptychography.
class py4DSTEM.process.phase.iterative_ptychography.PtychographicReconstruction(
    datacube: DataCube,
    energy: float,
    semiangle_cutoff: float | None = None,
    rolloff: float = 2.0,
    vacuum_probe_intensity: ndarray | None = None,
    polar_parameters: Mapping[str, float] | None = None,
    diffraction_intensities_shape: Tuple[int, int] | None = None,
    reshaping_method: str = 'fourier',
    probe_roi_shape: Tuple[int, int] | None = None,
    object_padding_px: Tuple[int, int] | None = None,
    dp_mask: ndarray | None = None,
    initial_object_guess: ndarray | None = None,
    initial_probe_guess: ndarray | None = None,
    initial_scan_positions: ndarray | None = None,
    verbose: bool = True,
    device: str = 'cpu',
    **kwargs
)
Iterative Ptychographic Reconstruction Class.

Diffraction intensities dimensions: \((R_x, R_y, Q_x, Q_y)\) Reconstructed probe dimensions: \((S_x, S_y)\) Reconstructed object dimensions: \((P_x, P_y)\)

such that \((S_x, S_y)\) is the region-of-interest (ROI) size of our probe and \((P_x, P_y)\) is the padded-object size we position our ROI around in.

**Parameters**

- **datacube** (*DataCube*) – Input 4D diffraction pattern intensities
- **energy** (*float*) – The electron energy of the wave functions in eV
- **semiangle_cutoff** (*float*, *optional*) – Semiangle cutoff for the initial probe guess
- **rolloff** (*float*, *optional*) – Semiangle rolloff for the initial probe guess
- **vacuum_probe_intensity** (*np.ndarray*, *optional*) – Vacuum probe to use as intensity aperture for initial probe guess
- **polar_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.
- **diffraction_intensities_shape** (*Tuple[int, int]*, *optional*) – Pixel dimensions \((Q_x', Q_y')\) of the resampled diffraction intensities. If None, no resampling of diffraction intensities is performed.
- **reshaping_method** (*str*, *optional*) – Method to use for reshaping, either ‘bin’, ‘bilinear’, or ‘fourier’ (default)
- **probe_roi_shape** – Padded diffraction intensities shape. If None, no padding is performed.
- **object_padding_px** (*Tuple[int, int]*, *optional*) – Pixel dimensions to pad object with. If None, the padding is set to half the probe ROI dimensions.
- **dp_mask** (*ndarray*, *optional*) – Mask for datacube intensities \((Q_x, Q_y)\)
- **initial_object_guess** (*np.ndarray*, *optional*) – Initial guess for complex-valued object of dimensions \((P_x, P_y)\). If None, initialized to 1.0j.
- **initial_probe_guess** (*np.ndarray*, *optional*) – Initial guess for complex-valued probe of dimensions \((S_x, S_y)\). If None, initialized to ComplexProbe with semiangle_cutoff, energy, and aberrations.
- **initial_scan_positions** (*np.ndarray*, *optional*) – Probe positions in Å for each diffraction intensity. If None, initialized to a grid scan.
- **verbose** (*bool*, *optional*) – If True, class methods will inherit this and print additional information.
- **device** (*str*, *optional*) – Calculation device will be performed on. Must be ‘cpu’ or ‘gpu’.
- **kwargs** – Provide the aberration coefficients as keyword arguments.
__init__(datacube: DataCube, energy: float, semiangle_cutoff: float | None = None, rolloff: float = 2.0, vacuum_probe_intensity: ndarray | None = None, polar_parameters: Mapping[str, float] | None = None, diffraction_intensities_shape: Tuple[int, int] | None = None, reshaping_method: str = 'fourier', probe_roi_shape: Tuple[int, int] | None = None, object_padding_px: Tuple[int, int] | None = None, dp_mask: ndarray | None = None, initial_object_guess: ndarray | None = None, initial_probe_guess: ndarray | None = None, initial_scan_positions: ndarray | None = None, verbose: bool = True, **kwargs)

preprocess(fit_function: str = 'plane', plot_center_of_mass: str = 'default', plot_rotation: bool = True, maximize_divergence: bool = False, rotation_angles_deg: np.darray, plot_probe_overlaps: bool = True, force_com_rotation: float | None = None, force_com_transpose: float | None = None, force_com_shifts: float | None = None, **kwargs)

Ptychographic preprocessing step. Calls the base class methods:

- _extract_intensities_and_calibrations_from_datacube,
- _compute_center_of_mass(),
- _solve_CoM_rotation(), _normalize_diffraction_intensities(),
- _calculate_scan_positions_in_px()

Additionally, it initializes an (Px,Py) array of 1.0j and a complex probe using the specified polar parameters.

Parameters

- **fit_function** (str, optional) – 2D fitting function for CoM fitting. One of ‘plane’, ‘parabola’, ‘bezier_two’
- **plot_center_of_mass** (str, optional) – If ‘default’, the corrected CoM arrays will be displayed. If ‘all’, the computed and fitted CoM arrays will be displayed
- **plot_rotation** (bool, optional) – If True, the CoM curl minimization search result will be displayed
- **maximize_divergence** (bool, optional) – If True, the divergence of the CoM gradient vector field is maximized
- **rotation_angles_deg** (np.darray, optional) – Array of angles in degrees to perform curl minimization over
- **plot_probe_overlaps** (bool, optional) – If True, initial probe overlaps scanned over the object will be displayed
- **force_com_rotation** (float (degrees), optional) – Force relative rotation angle between real and reciprocal space
- **force_com_transpose** (bool, optional) – Force whether diffraction intensities need to be transposed.
- **force_com_shifts** (tuple of ndarrays (CoMx, CoMy)) – Amplitudes come from diffraction patterns shifted with the CoM in the upper left corner for each probe unless shift is overwritten.

Returns

- **self** – Self to accommodate chaining
Return type

_PtychographicReconstruction_

**reconstruct**(*max_iter: int = 64, reconstruction_method: str = 'gradient-descent',
    reconstruction_parameter: float = 1.0, max_batch_size: int | None = None, seed_random: int | None = None, step_size: float = 0.9, normalization_min: float = 0.001, positions_step_size: float = 0.9, pure_phase_object_iter: int = 0, fix_com: bool = True, fix_probe_iter: int = 0, fix_probe_fourier_amplitude_iter: int = 0, positions_step_iter: int = inf,

Ptychographic reconstruction main method.

**Parameters**

- **max_iter** (*int, optional*) – Maximum number of iterations to run
- **reconstruction_method** (*str, optional*) – Specifies which reconstruction algorithm to use, one of: “generalized-projection”, “DM_AP” (or “difference-map_alternating-projections”), “RAAR” (or “relaxed-averaged_alternating-reflections”), “RRR” (or “relax-reflect-reflect”), “SUPERFLIP” (or “charge-flipping”), or “GD” (or “gradient_descent”)
- **reconstruction_parameter** (*float, optional*) – Reconstruction parameter for various reconstruction methods above.
- **max_batch_size** (*int, optional*) – Max number of probes to update at once
- **seed_random** (*int, optional*) – Seeds the random number generator, only applicable when max_batch_size is not None
- **step_size** (*float, optional*) – Update step size
- **normalization_min** (*float, optional*) – Probe normalization minimum as a fraction of the maximum overlap intensity
- **positions_step_size** (*float, optional*) – Positions update step size
- **pure_phase_object** (*bool, optional*) – If True, object amplitude is set to unity
- **fix_com** (*bool, optional*) – If True, fixes center of mass of probe
- **fix_probe_iter** (*int, optional*) – Number of iterations to run with a fixed probe before updating probe estimate
- **fix_probe_amplitude** (*int, optional*) – Number of iterations to run with a fixed probe amplitude
- **fix_positions_iter** (*int, optional*) – Number of iterations to run with fixed positions before updating positions estimate
- **global_affine_transformation** (*bool, optional*) – If True, positions are assumed to be a global affine transform from initial scan
- **probe_support_relative_radius** (*float, optional*) – Radius of probe supergaussian support in scaled pixel units, between (0,1]
- **probe_support_supergaussian_degree** (*float, optional*) – Degree supergaussian support is raised to, higher is sharper cutoff
• `gaussian_filter_sigma (float, optional)` – Standard deviation of gaussian kernel
• `gaussian_filter_iter (int, optional)` – Number of iterations to run using object smoothness constraint
• `butterworth_filter_iter (int, optional)` – Number of iterations to run using high-pass butterworth filter
• `q_lowpass (float)` – Cut-off frequency in A^-1 for low-pass butterworth filter
• `q_highpass (float)` – Cut-off frequency in A^-1 for high-pass butterworth filter
• `store_iterations (bool, optional)` – If True, reconstructed objects and probes are stored at each iteration
• `progress_bar (bool, optional)` – If True, reconstruction progress is displayed
• `reset (bool, optional)` – If True, previous reconstructions are ignored

Returns
self – Self to accommodate chaining

Return type
`PtychographicReconstruction`

`visualize(iterations_grid: Tuple[int, int] | None = None, plot_convergence: bool = True, plot_probe: bool = True, object_mode: str = 'phase', cbar: bool = True, padding: int = 0, relative_error: bool = True, **kwargs)`

Displays reconstructed object and probe.

Parameters
• `plot_convergence (bool, optional)` – If true, the RMS error plot is displayed
• `iterations_grid (Tuple[int, int])` – Grid dimensions to plot reconstruction iterations
• `cbar (bool, optional)` – If true, displays a colorbar
• `plot_probe (bool)` – If true, the reconstructed probe intensity is also displayed
• `object_mode (str)` – Specifies the attribute of the object to plot. One of ‘phase’, ‘amplitude’, ‘intensity’
• `relative_error (bool)` – Sets the error to be relative to the first iteration. TODO - update to be relative to empty object wave error (RMS of all measurements).

Returns
self – Self to accommodate chaining

Return type
`PtychographicReconstruction`

Module for reconstructing phase objects from 4DSTEM datasets using iterative methods, namely joint ptychography.
class py4STEM.process.phase.iterative_simultaneous_ptychography.SimultaneousPtysographicReconstruction
Iterative Simultaneous Ptychographic Reconstruction Class.

Diffraction intensities dimensions : (Rx,Ry,Qx,Qy) (for each measurement) Reconstructed probe dimensions : (Sx,Sy) Reconstructed electrostatic dimensions : (Px,Py) Reconstructed magnetic dimensions : (Px,Py)

such that (Sx,Sy) is the region-of-interest (ROI) size of our probe and (Px,Py) is the padded-object size we position our ROI around in.

Parameters

- **datacube** *(Sequence[DataCube])* – Tuple of input 4D diffraction pattern intensities
- **energy** *(float)* – The electron energy of the wave functions in eV
- **simultaneous_measurements_mode** *(str, optional)* – One of ‘-+’, ‘-0+’, ‘0+’, where -/0/+ refer to the sign of the magnetic potential
- **semiangle_cutoff** *(float, optional)* – Semiangle cutoff for the initial probe guess
- **rolloff** *(float, optional)* – Semiangle rolloff for the initial probe guess
- **vacuum_probe_intensity** *(np.ndarray, optional)* – Vacuum probe to use as intensity aperture for initial probe guess
- **polar_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.
- **diffraction_intensities_shape** *(Tuple[int,int], optional)* – Pixel dimensions (Qx’,Qy’) of the resampled diffraction intensities If None, no resampling of diffraction intensities is performed
- **reshaping_method** *(str, optional)* – Method to use for reshaping, either ‘bin’, ‘bilinear’, or ‘fourier’ (default)
- **probe_roi_shape** – Padded diffraction intensities shape. If None, no padding is performed
- **(int)** – Padded diffraction intensities shape. If None, no padding is performed
- **int** – Padded diffraction intensities shape. If None, no padding is performed
- **optional** – Padded diffraction intensities shape. If None, no padding is performed
- **object_padding_px** *(Tuple[int,int], optional)* – Pixel dimensions to pad objects with If None, the padding is set to half the probe ROI dimensions
- **dp_mask** *(ndarray, optional)* – Mask for datacube intensities (Qx,Qy)
- **initial_object_guess** *(np.ndarray, optional)* – Initial guess for complex-valued object of dimensions (Px,Py) If None, initialized to 1.0j
- **initial_probe_guess** *(np.ndarray, optional)* – Initial guess for complex-valued probe of dimensions (Sx,Sy). If None, initialized to ComplexProbe with semiangle_cutoff, energy, and aberrations
- **initial_scan_positions** *(np.ndarray, optional)* – Probe positions in Å for each diffraction intensity If None, initialized to a grid scan
- **verbose** *(bool, optional)* – If True, class methods will inherit this and print additional information
- **device** *(str, optional)* – Calculation device will be performed on. Must be ‘cpu’ or ‘gpu’
**kwargs – Provide the aberration coefficients as keyword arguments.

```
__init__(datacube: Sequence[DataCube], energy: float, simultaneous_measurements_mode: str = '+', 
    semiangle_cutoff: float | None = None, rolloff: float = 2.0, vacuum_probe_intensity: ndarray | None = None,
    polar_parameters: Mapping[str, float] | None = None, diffraction_intensities_shape: Tuple[int, int] | None = None,
    reshaping_method: str = 'fourier', probe_roi_shape: Tuple[int, int] | None = None, 
    object_padding_pixels: Tuple[int, int] | None = None, dp_mask: ndarray | None = None,
    initial_object_guess: ndarray | None = None, initial_probe_guess: ndarray | None = None,
    initial_scan_positions: ndarray | None = None, device: str = 'cpu', 
    **kwargs)
```

**preprocess**

```
preprocess(fit_function: str = 'plane', plot_rotation: bool = True, maximize_divergence: bool = False, 
       87., 88., 89.]), plot_probe_overlaps: bool = True, force_com_rotation: float | None = None, 
    force_com_transpose: float | None = None, force_com_shifts: float | None = None, 
    **kwargs)
```

Ptychographic preprocessing step. Calls the base class methods:

```
_extract_intensities_and_calibrations_from_datacube, _compute_center_of_mass(), _solve_CoM_rotation(), _normalize_diffraction_intensities(), _calculate_scan_positions_in_px()
```

Additionally, it initializes an (Px,Py) array of 1.0j and a complex probe using the specified polar parameters.

**Parameters**

- **fit_function** (str, optional) – 2D fitting function for CoM fitting. One of 'plane', 'parabola', 'bezier_two'
- **plot_rotation** (bool, optional) – If True, the CoM curl minimization search result will be displayed
- **maximize_divergence** (bool, optional) – If True, the divergence of the CoM gradient vector field is maximized
- **rotation_angles_deg** (np.ndarray, optional) – Array of angles in degrees to perform curl minimization over
- **plot_probe_overlaps** (bool, optional) – If True, initial probe overlaps scanned over the object will be displayed
- **force_com_rotation** (float (degrees), optional) – Force relative rotation angle between real and reciprocal space
- **force_com_transpose** (float, optional) – Force whether diffraction intensities need to be transposed
- **force_com_shifts** (sequence of tuples of ndarrays (CoMx, CoMy)) – Amplitudes come from diffraction patterns shifted with the CoM in the upper left corner for each probe unless shift is overwritten.

**Returns**

- **self** – Self to accommodate chaining
Return type

PtychographicReconstruction

reconstruct(max_iter: int = 64, reconstruction_method: str = 'gradient-descent',
reconstruction_parameter: float = 1.0, max_batch_size: int | None = None, seed_random: int | None = None, step_size: float = 0.9, normalization_min: float = 0.001, positions_step_size: float = 0.9, pure_phase_object_iter: int = 0, fix_com: bool = True, fix_probe_iter: int = 0, warmup_iter: int = 0, fix_probe_fourier_amplitude_iter: int = 0, fix_positions_iter: int = inf, global_affine_transformation: bool = True, probe_support_relative_radius: float = 10.0, gaussian_filter_sigma: float | None = None, butterworth_filter_iter: int = inf, q_lowpass: float | None = None, q_highpass: float | None = None, store_iterations: bool = False, progress_bar: bool = True, reset: bool | None = None)

Ptychographic reconstruction main method.

Parameters

- **max_iter** (int, optional) – Maximum number of iterations to run
- **reconstruction_method** (str, optional) – Specifies which reconstruction algorithm to use, one of: “generalized-projection”, “DM_AP” (or “difference-map_alternating-projections”), “RAAR” (or “relaxed-averaged-alternating-reflections”), “RRR” (or “relax-reflect-reflect”), “SUPERFLIP” (or “charge-flipping”), or “GD” (or “gradient_descent”)
- **reconstruction_parameter** (float, optional) – Reconstruction parameter for various reconstruction methods above.
- **max_batch_size** (int, optional) – Max number of probes to update at once
- **seed_random** (int, optional) – Seeds the random number generator, only applicable when max_batch_size is not None
- **step_size** (float, optional) – Update step size
- **normalization_min** (float, optional) – Probe normalization minimum as a fraction of the maximum overlap intensity
- **positions_step_size** (float, optional) – Positions update step size
- **pure_phase_object** (bool, optional) – If True, object amplitude is set to unity
- **fix_com** (bool, optional) – If True, fixes center of mass of probe
- **fix_probe_iter** (int, optional) – Number of iterations to run with a fixed probe before updating probe estimate
- **fix_probe_amplitude** (int, optional) – Number of iterations to run with a fixed probe amplitude
- **fix_positions_iter** (int, optional) – Number of iterations to run with fixed positions before updating positions estimate
- **global_affine_transformation** (bool, optional) – If True, positions are assumed to be a global affine transform from initial scan
- **probe_support_relative_radius** (float, optional) – Radius of probe supergaussian support in scaled pixel units, between (0,1]
- **probe_support_supergaussian_degree** (float, optional) – Degree supergaussian support is raised to, higher is sharper cutoff
• `gaussian_filter_sigma` *(float, optional)* – Standard deviation of gaussian kernel

• `gaussian_filter_iter` *(int, optional)* – Number of iterations to run using object smoothness constraint

• `butterworth_filter_iter` *(int, optional)* – Number of iterations to run using high-pass butteworth filter

• `q_highpass` *(float)* – Cut-off frequency for high-pass filter

• `store_iterations` *(bool, optional)* – If True, reconstructed objects and probes are stored at each iteration

• `progress_bar` *(bool, optional)* – If True, reconstruction progress is displayed

• `reset` *(bool, optional)* – If True, previous reconstructions are ignored

Returns

`self` – Self to accommodate chaining

Return type

`PtychographicReconstruction`

`visualize` *(iterations_grid: Tuple[int, int] | None = None, plot_convergence: bool = True, plot_probe: bool = True, object_mode: str = \"phase\", cbar: bool = True, padding: int = 0, relative_error: bool = True, **kwargs)*

Displays reconstructed object and probe.

Parameters

• `plot_convergence` *(bool, optional)* – If true, the RMS error plot is displayed

• `iterations_grid` *(Tuple[int, int]*) – Grid dimensions to plot reconstruction iterations

• `cbar` *(bool, optional)* – If true, displays a colorbar

• `plot_probe` *(bool)* – If true, the reconstructed probe intensity is also displayed

• `object_mode` *(str)* – Specifies the attribute of the object to plot. One of ‘phase’, ‘amplitude’, ‘intensity’

• `relative_error` *(bool)* – Sets the error to be relative to the first iteration. TODO - update to be relative to empty object wave error (RMS of all measurements).

Returns

`self` – Self to accommodate chaining

Return type

`PtychographicReconstruction`

```python
```

Symbols for the polar representation of all optical aberrations up to the fifth order.

```python
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].

- **seминgle_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].

- **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 complex probe in the center of the region of interest.
visualize(**kwargs)

Plots the probe amplitude.

py4DSTEM.process.phase.utils.spatial_frequencies(gpts: Tuple[int, int], sampling: Tuple[float, float])

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.projection(u: ndarray, v: ndarray, xp)

Projection of vector u onto vector v.

py4DSTEM.process.phase.utils.orthogonalize(V: ndarray, xp)

Non-normalized QR decomposition using repeated projections.

py4DSTEM.process.phase.utils.fourier_translation_operator(positions: ~numpy.ndarray, shape: tuple, xp=<module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/py4dstem/envs/latest/lib/python3.8/site-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/python3.8/site-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)

Affine Transform Class.

Simplified version of AffineTransform from tike: https://github.com/AdvancedPhotonSource/tike/blob/f9004a32fda5e49fa63b987e9ff3c8447d59950/src/py4DSTEM/ptycho/position.py

AffineTransform() -> Identity

Parameters
• scale0 (float) – x-scaling
• scale1 (float) – y-scaling
• shear1 (float) – gamma shear
• angle (float) – heta rotation angle
• t0 (float) – x-translation
• t1 (float) – y-translation

__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)

classmethod fromarray(T: ndarray)

Return an Affine Transform 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/py4dstem/envs/latest/lib/python3.8/site-packages/numpy/__init__.py>)

Use least squares to estimate the global affine transformation.
Use RANSAC to estimate the global affine transformation.

```python
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_builds/py4dstem/envs/latest/lib/python3.8/site-packages/numpy/__init__.py'>)
```

Computes fourier ring correlation (FRC) of 2 arrays. Arrays must be 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 performed 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

```python
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 Å (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 performed 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

---

**probe**

```python
py4DSTEM.process.probe.kernel.get_kernel(probe, mode='flat', **kwargs)
```

Creates a kernel from the probe for cross-correlative template matching.

Precise behavior and valid keyword arguments depend on the **mode** selected. In each case, the center of the probe is shifted to the origin and the kernel normalized such that it sums to 1. In ‘flat’ mode, this is the only processing performed. In the remaining modes, some additional processing is performed which adds a ring of negative intensity around the central probe, which results in edge-smoothing-like behavior during cross correlation. Valid modes are:

- **‘flat’**: creates a flat probe kernel. For bullseye or other structured probes, this mode is recommended.

- **‘gaussian’**: subtracts a gaussian with a width of standard deviation ‘sigma’

- **‘sigmoid’**: subtracts an annulus with inner and outer radii of (ri,ro) and a sine-squared sigmoid radial profile from the probe template.

- **‘sigmoid_log’**: subtracts an annulus with inner and outer radii of (ri,ro) and a logistic sigmoid radial profile from the probe template.

Each mode accepts ‘center’ (2-tuple) as a kwarg to manually specify the center of the probe, which is otherwise autodetected. Modes which accept additional kwargs and those arguments are:

- **‘gaussian’**: 
  
  sigma (number)

- **‘sigmoid’**: 
  
  radii (2-tuple)
• ‘sigmoid_log’:
  radii (2-tuple)

Accepts:
  **kwargs: depend on mode, see above

Returns
  (2D array)

py4DSTEM.process.probe.kernel.get_probe_kernel(probe, origin=None, bilinear=True)

Creates a convolution kernel from an average probe, by normalizing, then shifting the center of the probe to the corners of the array.

Parameters
  • probe (ndarray) – the diffraction pattern corresponding to the probe over vacuum
  • origin (2-tuple or None) – if None (default), finds the origin using get_probe_radius.
    Otherwise, should be a 2-tuple (x0, y0) specifying the origin position
  • bilinear (bool) – By default probe is shifted via a Fourier transform. Setting this to True overrides it and uses bilinear shifting. Not recommended!

Returns
  the convolution kernel corresponding to the probe, in real space

Return type
  (ndarray)

py4DSTEM.process.probe.kernel.get_probe_kernel_edge_gaussian(probe, sigma, origin=None, bilinear=True)

Creates a convolution kernel from an average 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 or None) – if None (default), finds the origin using get_probe_radius.
    Otherwise, should be a 2-tuple (x0, y0) specifying the origin position
  • bilinear (bool) – By default probe is shifted via a Fourier transform. Setting this to True overrides it and uses bilinear shifting. Not recommended!

Returns
  (ndarray) the convolution kernel corresponding to the probe

py4DSTEM.process.probe.kernel.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, from the probe center
- **origin** (2-tuple or None) – if None (default), finds the origin using get_probe_radius. Otherwise, should be a 2-tuple (x0,y0) specifying the origin position
- **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
the convolution kernel corresponding to the probe

Return type
(ndarray)

```python
py4DSTEM.process.probe.probe.get_vacuum_probe(data, **kwargs)
```

Takes some data and computes a vacuum probe, using a method selected based on the type and shape of `data`, and on other arguments passed. In each case, points outside the center disk are set to zero.

**Parameters**
- **data** (variable) – behavior and additional arguments depend on the type of `data`. If `data` is a
  - DataCube: computes a probe using all or some subset of the diffraction patterns in the datacube, aligning and averaging those patterns. The whole datacube is used if the ROI argument is not passed. If ROI is passed, uses a subset of diffraction patterns based on the ROI argument’s value, which may be a either an R-space shaped boolean mask, or a 4-tuple representing (Rxmin,Rxmax,Rymin,Rymax) of a rectangular region.
  - 3D array: averages the stack with no alignment
  - 2D array: uses this array as the probe.
  - None: makes a synthetic probe. Additional required arguments are
    - radius (number): the probe radius
    - width (number): the width of the region where the probe intensity drops off from its maximum to 0.
    - Qshape (2 tuple): the shape of diffraction space

Returns
(Probe) a Probe instance

```python
py4DSTEM.process.probe.probe.get_probe_from_vacuum_4Dscan(datacube, mask_threshold=0.2, mask_expansion=12, mask_opening=3, verbose=False, align=True)
```

Averages all diffraction patterns in a datacube, assumed to be taken over vacuum, to create and average vacuum probe. Optionally (default) aligns the patterns.

Values outside the average probe are zeroed, using a binary mask determined by the optional parameters mask_threshold, mask_expansion, and mask_opening. An initial binary mask is created using a threshold of less than mask_threshold times the maximal probe value. A morphological opening of mask_opening pixels is performed to eliminate stray pixels (e.g. from x-rays), followed by a dilation of mask_expansion pixels to ensure the entire probe is captured.

**Parameters**
- **datacube** (DataCube) – a vacuum scan
• **mask_threshold** *(float)* – threshold determining mask which zeros values outside of probe

• **mask_expansion** *(int)* – number of pixels by which the zeroing mask is expanded to capture the full probe

• **probe** *(expanded to capture the full)* –

• **mask_opening** *(int)* – size of binary opening used to eliminate stray bright pixels

• **verbose** *(bool)* – if True, prints progress updates

• **align** *(bool)* – if True, aligns the probes before averaging

Returns

the average probe

Return type

(ndarray of shape (datacube.Q_Nx,datacube.Q_Ny))

```python
py4DSTEM.process.probe.probe.get_probe_from_4Dscan_ROI_lims(datacube, ROI, mask_threshold=0.2,
                           mask_expansion=12,
                           mask_opening=3, verbose=False, align=True)
```

Averages all diffraction patterns within a specified ROI of a datacube to create an average vacuum probe. Optionally (default) aligns the patterns.

See documentation for get_average_probe_from_vacuum_scan for more detailed discussion of the algorithm.

Parameters

1. **datacube** *(DataCube)* – a vacuum scan

2. **ROI** *(len 4 list or tuple)* – the limits (rx_min, rx_max, ry_min, ry_max) of the selected region.

3. **mask_threshold** *(float)* – threshold determining mask which zeros values outside of probe

4. **mask_expansion** *(int)* – number of pixels by which the zeroing mask is expanded to capture the full probe

5. **mask_opening** *(int)* – size of binary opening used to eliminate stray bright pixels

6. **verbose** *(bool)* – if True, prints progress updates

7. **align** *(bool)* – if True, aligns the probes before averaging

8. **DP_mask** *(array)* – array of same shape as diffraction pattern to mask probes

Returns

the average probe

Return type

(ndarray of shape (datacube.Q_Nx,datacube.Q_Ny))

```python
py4DSTEM.process.probe.probe.get Probe from 4Dscan ROI mask(datacube, ROI, mask_threshold=0.2,
                       mask_expansion=12,
                       mask_opening=3, verbose=False, align=True, DP_mask=1)
```

Averages all diffraction patterns within a specified ROI of a datacube to create an average vacuum probe. Optionally (default) aligns the patterns.

See documentation for get_average_probe_from_vacuum_scan for more detailed discussion of the algorithm.

Parameters
• **datacube** ([`DataCube`)] – a vacuum scan

• **ROI** ([`ndarray` of `dtype=bool` and shape (`datacube.R_Nx`, `datacube.R_Ny`))] – An array of boolean variables shaped like the real space scan. Only scan positions where ROI=True are used to create the average probe.

• **mask_threshold** (`float`) – threshold determining mask which zeros values outside of probe

• **mask_expansion** (`int`) – number of pixels by which the zeroing mask is expanded to capture the full probe

• **mask_opening** (`int`) – size of binary opening used to eliminate stray bright pixels

• **verbose** (`bool`) – if True, prints progress updates

• **align** (`bool`) – if True, aligns the probes before averaging

• **DP_mask** (`array`) – array of same shape as diffraction pattern to mask probes

**Returns**
the average probe

**Return type**
(`ndarray` of shape (`datacube.Q_Nx`, `datacube.Q_Ny`))

`py4DSTEM.process.probe.probe.get_probe_from_vacuum_3Dstack(data, mask_threshold=0.2, mask_expansion=12, mask_opening=3)`

Averages all diffraction patterns in a 3D stack of diffraction patterns, assumed to be taken over vacuum, to create and average vacuum probe. No alignment is performed - i.e. it is assumed that the beam was stationary during acquisition of the stack.

Values outside the average probe are zeroed, using a binary mask determined by the optional parameters mask_threshold, mask_expansion, and mask_opening. An initial binary mask is created using a threshold of less than mask_threshold times the maximal probe value. A morphological opening of mask_opening pixels is performed to eliminate stray pixels (e.g. from x-rays), followed by a dilation of mask_expansion pixels to ensure the entire probe is captured.

**Parameters**

• **data** (`array`) – a 3D stack of vacuum diffraction patterns, shape (`Q_Nx`, `Q_Ny`, `N`)

• **mask_threshold** (`float`) – threshold determining mask which zeros values outside of probe

• **mask_expansion** (`int`) – number of pixels by which the zeroing mask is expanded to capture the full probe

• **mask_opening** (`int`) – size of binary opening used to eliminate stray bright pixels

**Returns**
the average probe

**Return type**
(`array` of shape (`Q_Nx`, `Q_Ny`))

`py4DSTEM.process.probe.probe.get_probe_from_vacuum_2Dimage(data, mask_threshold=0.2, mask_expansion=12, mask_opening=3)`

A single image of the probe over vacuum is processed by zeroing values outside the central disk, using a binary mask determined by the optional parameters mask_threshold, mask_expansion, and mask_opening. An initial
binary mask is created using a threshold of less than mask_threshold time the maximal probe value. A morphological opening of mask_opening pixels is performed to eliminate stray pixels (e.g. from x-rays), followed by a dilation of mask_expansion pixels to ensure the entire probe is captured.

**Parameters**
- `data` *(array)* – a 2D array of the vacuum diffraction pattern, shape (Q_Nx,Q_Ny)
- `mask_threshold` *(float)* – threshold determining mask which zeros values outside of probe
- `mask_expansion` *(int)* – number of pixels by which the zeroing mask is expanded to capture the full probe
- `mask_opening` *(int)* – size of binary opening used to eliminate stray bright pixels

**Returns**
- (array of shape (Q_Nx,Q_Ny)) the average probe

---

`py4DSTEM.process.probe.probe.get_probe_synthetic(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**
- the probe

**Return type**
- (ndarray of shape (Q_Nx,Q_Ny))

---

`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**
- `datacube` – 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

**Coeffs 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`

```python
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

```python
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`

```python
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.

```python
py4DSTEM.process.rdf.rdf.get_phi(radialIntensity, scatter, q_arr)
```
```python
ymean scale*scatter.fe**2
```

```python
py4DSTEM.process.rdf.rdf.get_mask(left, right, midpoint, slopes, q_arr)
```
```python
start is float stop is float midpoint is float slopes is [float,float]
```

```python
py4DSTEM.process.rdf.rdf.get_rdf(phi, q_arr)
```
phi can be masked or not masked

### utils

```python
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.

```python
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.

```python
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 performed 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 performed 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 representation 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 \((x_0,y_0,A,B,C)\):

\[
A(x-x_0)^2 + B(x-x_0)(y-y_0) + C(y-y_0)^2 = 1
\]

It is possible to convert between \((a,b,\theta) \leftrightarrow (A,B,C)\) using the convert_ellipse_params() and convert_ellipse_params_r() methods.

Transformation from cartesian to polar-elliptical space is done using

\[
x = x_0 + a*r*cos(phi)*cos(\theta) + b*r*sin(phi)*sin(\theta)\]
\[
y = y_0 + a*r*cos(phi)*sin(\theta) - b*r*sin(phi)*cos(\theta)
\]

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)
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)
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 Bragg Vector calibration routines, as it is much faster to perform this on the peak positions than the entire datacube.

```python
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)

```python
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

**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)
Create a hard circular mask, for use in DPC integration 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

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 initial correlation peak. By using the matrix multiply DFT to do the Fourier upsampling, the efficiency is greatly improved. This is adapted from the subfunction 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:

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:

```
coops = np.mgrid[0:ar.shape[0],0:ar.shape[1]] radial_sums = radial_reduction(ar, x0,y0, co-ords=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
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

Generates Fourier coordinates for a (Nx,Ny)-shaped 2D array. Specifying the pixelSize argument sets a unit size.

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 performed on. Must be ‘cpu’ or ‘gpu’

Finds and returns the center of mass of array ar.

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.

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 tessellation
- **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, bandlimitNyquist=None, bandlimit_power=2, dtype=<class 'numpy.float32'>)
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

**Returns**
the resized array (2D/4D numpy array)

**virtualdiffraction**

py4DSTEM.process.virtualdiffraction.get_virtual_diffraction(datacube, method=None, mode=None, geometry=None, calibrated=False, shift_center=False, verbose=True, return_mask=False)

Function to calculate virtual diffraction

**Parameters**
- **datacube** *(Datacube)* – datacube class object which stores 4D-dataset needed for calculation
- **method** *(str)* – defines method used for diffraction pattern, options are ‘mean’, ‘median’, and ‘max’
- **mode** *(str)* – defines mode for selecting area in real space to use for virtual diffraction. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. Options:
  - ‘point’ uses singular point as detector
  - ‘circle’ or ‘circular’ uses round detector, like bright field
  - ‘annular’ or ‘annulus’ uses annular detector, like dark field
  - ‘mask’ flexible detector, any 2D array
- **geometry** *(variable)* – valid entries are determined by the mode, values in pixels argument, as follows. The default is None, which means no geometry will be applied and the whole datacube will be used for the calculation. If mode is None the geometry will not be applied.
  - ‘point’: 2-tuple, (rx,ry), qx and qy are each single float or int to define center
  - ‘circle’ or ‘circular’: nested 2-tuple, ((rx,ry),radius), qx, qy and radius, are each single float or int
  - ‘annular’ or ‘annulus’: nested 2-tuple, ((rx,ry),(radius_i,radius_o)), qx, qy, radius_i, and radius_o are each single float or integer
  - ‘rectangle’, ‘square’, ‘rectangular’: 4-tuple, (xmin,xmax,ymin,ymax)
  - **mask**: flexible detector, any boolean or floating point 2D array with the same shape as datacube.Rshape
**calibrated** (bool) – if True, geometry is specified in units of ‘A’ instead of pixels. The datacube’s calibrations must have its “R_pixel_units” parameter set to “A”. If mode is None the geometry and calibration will not be applied.

**shift_center** (bool) – if True, the diffraction pattern is shifted to account for beam shift or the changing of the origin through the scan. The datacube’s calibration[‘origin’] parameter must be set Only ‘max’ and ‘mean’ supported for this option.

**verbose** (bool) – if True, show progress bar

**return_mask** (bool) – if False (default) returns a virtual image as usual. If True, does not generate or return a virtual image, instead returning the mask that would be used in virtual diffraction computation.

**Returns**
the diffraction image

**Return type**
(2D array)

### virtualimage

**py4DSTEM.process.virtualimage.get_virtual_image**(datacube, mode, geometry, centered=False, calibrated=False, shift_center=False, verbose=True, dask=False, return_mask=False, test_config=False)

Function to calculate virtual image

**Parameters**

- **datacube** (Datacube) – datacube class object which stores 4D-dataset needed for calculation
- **mode** (str) – defines geometry mode for calculating virtual image. Options:
  - ‘point’ uses singular point as detector
  - ‘circle’ or ‘circular’ uses round detector, like bright field
  - ‘annular’ or ‘annulus’ uses annular detector, like dark field
  - ‘mask’ flexible detector, any 2D array
- **geometry** (variable) – valid entries are determined by the mode, values in pixels argument, as follows:
  - ‘point’: 2-tuple, (qx, qy), qx and qy are each single float or int to define center
  - ‘circle’ or ‘circular’: nested 2-tuple, ((qx, qy),radius), qx, qy and radius, are each single float or int
  - ‘annular’ or ‘annulus’: nested 2-tuple, ((qx,qy), (radius_i,radius_o))
  - ‘rectangle’, ‘square’, ‘rectangular’: a 4-tuple, (xmin,xmax,ymin,ymax)
  - mask: flexible detector, any boolean or floating point 2D array with the same shape as datacube.Qshape
- **centered** (bool) – if False (default), the origin is in the upper left corner. If True, the mean measured origin in the datacube calibrations is set as center. The measured origin
is set with datacube.calibration.set_origin(). In this case, for example, a centered bright field image could be defined by geometry = ((0,0), R). For mode="mask", has no effect.

- **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.

- **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 (centered = True). 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.

- **verbose (bool)** – if True, show progress bar

- **dask (bool)** – if True, use dask arrays

- **return_mask (bool or tuple)** – if False (default) returns a virtual image as usual. If True, does not generate or return a virtual image, instead returning the mask that would be used in virtual image computation for any call to this function where shift_center = False. Otherwise, must be a 2-tuple of integers corresponding to a scan position (rx,ry); in this case, returns the mask that would be used for virtual image computation at this scan position with shift_center set to True.

- **test_config** – if True, returns the Boolean value of (centered, calibrated, shift_center). Does not compute the virtual image.

Returns
(2D array) virtual image

**py4DSTEM.process.virtualimage.get_calibrated_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)** –
- **DataCube** –
- **attr (any object with a .calibration)** –

:paramref :param or None) Used to retrieve the center positions. If None: :paramref confirms that: :paramref centered and calibrated are False then passes: :paramref mode: see py4DSTEM.process.virtualimage.get_virtual_image :paramref geometry: see py4DSTEM.process.virtualimage.get_virtual_image :paramref centered: see py4DSTEM.process.virtualimage.get_virtual_image :paramref calibrated: see py4DSTEM.process.virtualimage.get_virtual_image

Returns
(tuple) the geometry in detector pixels

**py4DSTEM.process.virtualimage.make_detector (shape, mode, geometry)**

Function to return 2D mask

Parameters

- **shape (tuple)** – defines shape of mask, for example (Q_Nx, Q_Ny) where Q_Nx and Q_Ny are mask sizes
- **mode (str)** – defines geometry mode for calculating virtual image options:
'point' uses singular point as detector
'circle' or 'circular' uses round detector, like bright field
'annular' or 'annulus' uses annular detector, like dark field
'rectangle', 'square', 'rectangular', uses rectangular detector
'mask' flexible detector, any boolean or floating point 2D array with
the same shape as datacube.Qshape or datacube.Rshape for virtual image or diffraction image respectively

- geometry (variable) – valid entries are determined by the mode, values in pixels argument, as follows:
  - 'point': 2-tuple, (qx,qy),
    qx and qy are each single float or int to define center
  - 'circle' or 'circular': nested 2-tuple, ((qx,qy),radius),
    qx, qy and radius, are each single float or int
  - 'annular' or 'annulus': nested 2-tuple, ((qx,qy),(radius_i,radius_o)),
    qx, qy, radius_i, and radius_o are each single float or integer
  - 'rectangle', 'square', 'rectangular': 4-tuple, (xmin,xmax,ymin,ymax)

Returns
virtual detector in the form of a 2D mask (array)

py4DSTEM.process.virtualimage.make_bragg_mask(Qshape, g1, g2, radius, origin, max_q, return_sum=True, **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 (number) – 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

Returns
(2 or 3D array) the mask

py4DSTEM.process.virtualimage.get_virtual_image_pointlistarray(peaks, mode=None, geometry=None)

Make a virtual image from a pointlist array. TODO - implement more virtual detectors.

Parameters
- peaks (PointListArray) – List of all peaks and intensities.
- mode (str) – defines geometry mode for calculating virtual image. Options:
- 'circular' or 'circle' uses round detector, like bright field
- 'annular' or 'annulus' uses annular detector, like dark field

**geometry**(variable) –
valid entries are determined by the mode, values in pixels
argument, as follows:

- 'circle' or 'circular': nested 2-tuple, ((qx,qy),radius),
  qx, qy and radius, are each single float or int
- 'annular' or 'annulus': nested 2-tuple, ((qx,qy),(radius_i,radius_o)),
  qx, qy, radius_i, and radius_o are each single float or integer
- Note that (qx,qy) can be skipped, which assumes peaks centered at (0,0)

Returns
the calculated virtual image

**Return type**
im_virtual (2D numpy array)

```python
py4DSTEM.process.virtualimage.get_virtual_image_braggvectors(bragg_peaks, mode=None, geometry=None)
```

Function to calculate virtual images from braggvectors / pointlist arrays. TODO - implement these detectors for braggvectors

**Parameters**

- **bragg_peaks**(BraggVectors) – BraggVectors class object which stores bragg peaks
- **mode**(str) – defines geometry mode for calculating virtual image. Options:
  - 'circular' or 'circle' uses round detector, like bright field
  - 'annular' or 'annulus' uses annular detector, like dark field
- **geometry**(variable) –
  valid entries are determined by the mode, values in pixels
  argument, as follows:
  
  - 'circle' or 'circular': nested 2-tuple, ((qx,qy),radius),
    qx, qy and radius, are each single float or int
  - 'annular' or 'annulus': nested 2-tuple, ((qx,qy),(radius_i,radius_o)),
    qx, qy, radius_i, and radius_o are each single float or integer
  - Note that (qx,qy) can be skipped, which assumes peaks centered at (0,0)

Returns
the calculated virtual image

**Return type**
im_virtual (2D numpy array)
wholepatternfit

class py4DSTEM.process.wholepatternfit.wp_models.WPFModelPrototype(name: str, params: dict)

Prototype class for a component 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)

class py4DSTEM.process.wholepatternfit.wp_models.DCBackground(background_value=0.0, name='DC Background')

__init__(background_value=0.0, name='DC Background')

class py4DSTEM.process.wholepatternfit.wp_models.GaussianBackground(sigma, intensity, global_center=True, x0=0.0, y0=0.0, name='Gaussian Background')

__init__(sigma, intensity, global_center=True, x0=0.0, y0=0.0, name='Gaussian Background')

class py4DSTEM.process.wholepatternfit.wp_models.GaussianRing(radius, sigma, intensity, global_center=True, x0=0.0, y0=0.0, name='Gaussian Ring')

__init__(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)

__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.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 = [], include_indices: list | None = None, 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 = [], name='Complex Overlapped Disk Lattice', verbose=False)


py4DSTEM.process.wholepatternfit.wpf_viz.show_lattice_points(self, returnfig=False, *args, **kwargs)

Plotting utility to show the initial lattice points.

1.4.6 visualize

Table of Contents

- visualize
  - show
  - overlay
  - virtualimage
  - vis_RQ
  - vis_grid
  - vis_special

show

py4DSTEM.visualize.show(ar, figsize=(8, 8), 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, **kwargs)

General visualization function for 2D arrays.

The simplest use of this function is:

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

will overlay a single red square, and
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:

```python
>>> 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

```python
>>> 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:

```python
>>> 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

```python
>>> 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

```python
>>> 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', 'polar_elliptical_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:

```python
>>> 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:

```python
>>> 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 minimum 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.
• ****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.

```python
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, scalebarwidth=None, scalebartextloc='above', scalebartextsize=12, gridsizing=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 receive 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 arr[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 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.

**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:**
- **braggdirections** (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-elliptical 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
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_theta_lines (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
    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`

`py4DSTEM.visualize.virtualimage.position_detector(data, mode, geometry, centered, calibrated, shift_center, invert=False, color='r', alpha=0.7, **kwargs)`
Display a diffraction space image with an overlaid mask representing a virtual detector.

Parameters

- `data` (DataCube, DiffractionSlice, array, tuple) —
  behavior depends on the argument type:
  - DataCube - check to see if this datacube has a mean, max, or median diffraction pattern, and if found, uses it (order of preference as written here). If not found, raises an exception.
  - DiffractionSlice - use the first slice array - use this array. This mode only works when centered, calibrated, and shift_center are False. Otherwise, use the tuple entry (array, Calibration)
  - tuple - must be either:
    - (DataCube, rx, ry) for rx,ry integers.
      Use the diffraction pattern at this scan position. `shift_center` is auto set to True in this mode.

- `mode` — see py4DSTEM.process.get_virtual_image
- `geometry` — see py4DSTEM.process.get_virtual_image
- `centered` — see py4DSTEM.process.get_virtual_image
- `calibrated` — see py4DSTEM.process.get_virtual_image
- `shift_center` — see py4DSTEM.process.get_virtual_image; if True, `data` should be a 3-tuple (DataCube, rx, ry)
- `invert` — if True, invert the mask
- `**kwargs` — all additional arguments are passed on to `show`
General visualization function for 2D arrays.

The simplest use of this function is:

```python
>>> 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:

```python
>>> 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:

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

will overlay a single red square, and

```python
>>> 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:

```python
>>> 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

```python
>>> 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:

```python
>>> 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

```python
>>> 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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 minimum 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.

• **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.

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
space
x0,y0 (numbers) the origin / vector tail position,
in reciprocal space
py4dstem, Release 0.14.0

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.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.visualize.vis_RQ.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)
py4DSTEM.visualize.vis_RQ.show(ar, figsize=(8, 8), cmap='gray', 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, **kwargs)

General visualization function for 2D arrays.

The simplest use of this function is:

```python
>>> 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:

```python
>>> 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:

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

will overlay a single red square, and

```python
>>> 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:

```python
>>> 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

```python
>>> 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:
```python
>>> 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
```python
>>> 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
```python
>>> 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:
```python
>>> 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:
```python
>>> 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 colormap; 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 minimum 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’. Whenever 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 calibraton or pixeisize/pixelunits for scalebar. If False, no scalebar is added.
- ****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 diffraction space vector.

```python
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 diffraction space vector.

```python
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,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
diffraction 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 num-
bbers) 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) linwidth
(number) alpha (number)
General visualization function for 2D arrays.

The simplest use of this function is:

```python
>>> 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:

```python
>>> 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 val-
ues 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 re-
quested overlay. For example:

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

will overlay a single red square, and

```python
>>> 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:

```python
>>> 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

```python
>>> 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:
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’. Whenever 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 calibration or pixelsize/pixelunits for scalebar. If False, no scalebar is added.

• **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 instead. if title is a list of strings (ex: ['title 1','title 2'])), each array has corre-
sponding 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 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.
**vis_special**

`py4DSTEM.visualize.vis_special.Complex2RGB(complex_array, vmin=None, vmax=None, hue_start=90)`  
Function to turn a complex array into rgb for plotting  
:param complex_array: complex array  
:type complex_array: 2D array  
:param vmin: minimum absolute value  
:type vmin: float, optional  
:param vmax: maximum absolute value  
:type vmax: float, optional  

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 minimum display value to saturate the lower 2% of pixels

**Parameters**

- **hue_start** (float, optional) – phase offset (degrees)

**Returns**

rgb array for plotting

`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:**

- **braggdirections** (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, uppright, 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` (color)
- `width` (number)
- `label` (str)
- `labelsixe` (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.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.visualize.vis_special.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.visualize.vis_special.get_selected_lattice_vectors\( (gx, gy, i0, i1, i2) \)`
From a set of reciprocal lattice points \((gx, gy)\), and indices in those arrays which specify the center beam, the first basis lattice vector, and the second basis lattice vector, computes and returns the lattice vectors \(g1\) and \(g2\).

**Parameters**
- \( gx \): \((1d \ array)\) – the reciprocal lattice points x-coords
- \( gy \): \((1d \ array)\) – the reciprocal lattice points y-coords
- \( i0 \): (int) – index in the \((gx, gy)\) arrays specifying the center beam
- \( i1 \): (int) – index in the \((gx, gy)\) arrays specifying the first basis lattice vector
- \( i2 \): (int) – index in the \((gx, gy)\) arrays specifying the second basis lattice vector

**Returns**
(2-tuple of 2-tuples) A 2-tuple containing
- \( g1 \): (2-tuple) the first lattice vector, \((g1x, g1y)\)
- \( g2 \): (2-tuple) the second lattice vector, \((g2x, g2y)\)

`py4DSTEM.visualize.vis_special.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 tessellated 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 tessellation may be directly laid over data.

**Parameters**
- \( voronoi \): (scipy.spatial.Voronoi) – the voronoi tessellation
- \( nx \): (int) – the x field-of-view of the tessellated region
- \( ny \): (int) – the y field-of-view of the tessellated 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.visualize.vis_special.hsv_to_rgb\( (hsv) \)`
Convert HSV values to RGB.
Parameters

**hsv** (*..., 3*) *array-like* – All values assumed to be in range [0, 1]

Returns

Colors converted to RGB values in range [0, 1]

Return type

(*..., 3*) *~numpy.ndarray*

py4DSTEM.visualize.vis_special.make_axes_locatable(axes)

This function accepts a set of reciprocal lattice points (gx, gy) and three indices (i0, i1, i2). Using those indices as, respectively, the origin, the endpoint of g1, and the endpoint of g2, this function computes the basis lattice vectors g1, g2, visualizes them, and returns them. To compute these vectors without visualizing, use latticevectors.get_selected_lattice_vectors().

Returns

g1, g2 if returnfig==True

g1, g2, fig, ax

Return type

if returnfig==False

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=(8, 8), 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, **kwargs)

General visualization function for 2D arrays.

The simplest use of this function is:

```python
>>> 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:

```python
>>> 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:

```python
>>> show(ar, rectangle={'lims':(10,20,10,20), 'color':'r'})
``` will overlay a single red square, and

```python
>>> show(ar, annulus={
    'center':[(28,68),(92,160)],
    'radii':[16,24], (12,36)],
    'fill':True,
    'alpha':[0.9,0.3],
    'color':[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 informa-
tion 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:
```python
>>> 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
```python
>>> 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:
```python
>>> 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
```python
>>> 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
```python
>>> 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 visual-
ize 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:
```python
>>> 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:

```python
>>> 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 minimum 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’. Whenever 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.

• **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.

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

```python
def py4DSTEM.visualize.vis_special.show_bragg_indexing(ar, braggdirections, voffset=5, hoffset=0, color='w', size=20, points=True, pointcolor='r', pointsize=50, returnfig=False, **kwargs)
```

Shows an array with an overlay describing the Bragg directions

**Accepts:**
- `ar (array)` the image `bragg_directions` (PointList) the bragg scattering directions; must have coordinates ‘qx’, ‘qy’, ‘h’, and ‘k’. Optionally may also have ‘l’.

```python
def py4DSTEM.visualize.vis_special.show_class_BPs(ar, x, y, s, s2, color='r', color2='y', **kwargs)
```

```python
def py4DSTEM.visualize.vis_special.show_complex(ar_complex, vmin=None, vmax=None, 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 horizontally 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 minimumum display value to saturate the lower 2% of pixels
- **cbar** (*bool, optional*) – if True, include color wheel
- **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 for 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 instead. 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

gerobordercolor
    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:

```python
>>> 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

ger_s function which returns a set of point sizes as a function of index i

ger_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 line profile.

`py4DSTEM.visualize.vis_special.show_lattice_vectors(ar, x0, y0, g1, g2, color='r', width=1, labelsize=20, labelcolor='w', returnfig=False, **kwargs)`

Adds the vectors g1,g2 to an image, with tail positions at (x0,y0). g1 and g2 are 2-tuples (gx,gy).

`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)`

Show the measured, fit, and residuals of the origin positions.

Parameters
    data (DataCube or Calibration or (3,2) – ((qx0_meas,qy0_meas),(qx0_fit,qy0_fit),(qx0_residuals,qy0_residuals)) –

`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, returfig=False, **kwargs)

Show enumerated index labels for a set of points

py4DSTEM.visualize.vis_special.show_qprofile(q, intensity, ymax=None, figsize=(12, 4), returfig=False, color='k', xlabel='q (pixels)', ylabel='Intensity (A.U.)', labelsize=16, ticklabels=14, grid='on', 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 xlabl (str) ylablabel size of x and y labels ticklabels size grid ‘off’ or ‘on’ 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 suppresses 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(strainmap, vrange_exx, vrange_theta, vrange_exy=None, vrange_eyy=None, flip_theta=False, bkgrd=True, show_chars=('exx', 'eyy', 'exy', 'theta'), bordercolor='k', borderwidth=1, titlesize=24, ticklabels=16, ticknumber=5, unilabs=24, show_axes=True, axes_x0=0, axes_y0=0, axes_x1=1, axes_y1=0, axes_length=10, axes_width=1, axes_color='r', axes_space='Q', labelaxes=True, QR_rotation=0, axes_labelsize=12, axes_labelcolor='r', axes_plots='exx', cmap='RdBu_r', layout=0, figsize=(12, 12), returnfig=False)

Display a strain map, showing the 4 strain components (e_xx,e_yy,e_xxy,theta), and masking each image with strainmap.get_slice('mask')

Parameters
- **strainmap** (RealSlice) –
- **vrange_exx** (length 2 list or tuple) –

1.4. API 287
• **vrange_theta** *(length 2 list or tuple)* –
• **vrange_exy** *(length 2 list or tuple)* –
• **vrange_eyy** *(length 2 list or tuple)* –
• **flip_theta** *(bool)* – if True, take negative of angle
• **bkgrd** *(bool)* –
• **show_cbars** *(tuple of strings)* – Show colorbars for the specified axes. Must be a tuple containing any, all, or none of ('exx','eyy','exy','theta').
• **bordercolor** *(color)* –
• **borderwidth** *(number)* –
• **titlesize** *(number)* –
• **ticklabelsize** *(number)* –
• **ticknumber** *(number)* – number of ticks on colorbars
• **unitlabelsize** *(number)* –
• **show_axes** *(bool)* –
• **axes_x0** *(number)* –
• **axes_y0** *(number)* –
• **xaxis_x** *(number)* –
• **xaxis_y** *(number)* –
• **axes_length** *(number)* –
• **axes_width** *(number)* –
• **axes_color** *(color)* –
• **xaxis_space** *(string)* – must be ‘Q’ or ‘R’
• **labelaxes** *(bool)* –
• **QR_rotation** *(number)* –
• **axes_labelsize** *(number)* –
• **axes_labelcolor** *(color)* –
• **axes_plots** *(tuple of strings)* – controls if coordinate axes showing the orientation of the strain matrices are overlaid over any of the plots. Must be a tuple of strings containing any, all, or none of ('exx','eyy','exy','theta').
• **cmap** *(colormap)* –
• **layout=0** *(int)* – determines the layout of the grid which the strain components will be plotted in. Must be in (0,1,2). 0=(2x2), 1=(1x4), 2=(4x1).
• **figsize** *(length 2 tuple of numbers)* –
• **returnfig** *(bool)* –

`py4DSTEM.visualize.vis_special.show_voronoi(ar, x, y, color_points='r', color_lines='w', max_dist=None, returnfig=False, **kwargs)`
1.4.7 emd

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 cali-
brations for each axis of the array, and names and units for those axis calibrations.

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

```python
>>> 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:

```python
>>> ar = Array(
>>>     np.ones((20,20,256,256)),
>>>     name = 'test_array',
>>>     units = 'intensity',
>>>     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 0.01Å^-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,

```python
>>> 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.

```python
>>> plt.scatter(ardims[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

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

```python
>>> 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
- **d ims** (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.

- **slice labels** (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 **slice labels** list, with the syntax array[‘label’] or array.get_slice(’label’). If **slice labels** 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
*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
*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

---

```python
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

---

```python
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:**

```python
>>> 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*) –
```python
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)
```

```python
class emdfile.Node(name: str | None = '/quotesingle.ts1'
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 built. Nodes themselves
contain the machinery for managing a tree hierarchy 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

```python
>>> 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

```python
>>> 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

```python
def 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:

```python
def .tree()  # show tree from current node
def .tree(show=True)  # show from root
def .tree(show=False)  # show from current node
def .tree(add=node)  # add a child node
def .tree(get='path')  # return a '/' delimited child node

def .tree(get='path')  # as above, starting at root

def .tree(cut=True)  # remove/return a branch, keep root metadata

def .tree(cut=False)  # remove/return a branch, discard root metadata

def .tree(cut='copy')  # remove/return a branch, copy root metadata

def .tree(graft=node)  # remove/graft a branch, keep root metadata

def .tree(graft=(node,True))  # as above

def .tree(graft=(node,False))  # as above, discard root metadata

def .tree(graft=(node,'copy'))  # as above, copy root metadata
```

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

```python
def .tree(arg)
```

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

**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:

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

```python
def x = node.metadata['md1']
def 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().

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 a branch from one tree, starting at this node, onto another tree at target node.
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) the new 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, keep root metadata
>>> .tree(graft=(node,True))  # as above

(continues on next page)
The show, add, and get methods can be accessed directly with

```python
>>> .tree(graft=(node, False))  # as above, discard root metadata
>>> .tree(graft=(node, 'copy'))  # as above, copy root metadata
```

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

**static log_new_node** (*method*)
Node subclass methods which generate and return a new node may be decorated with `@log_new_node`. This method creates a new Metadata dict stored inside `new_node.metadata` called `_fn_call_*`, where * is the name of the decorated method, which stores the args/kwargs/params passed to the generating 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 its 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')

class emdfile.RootedNode

RootedNodes are nodes that are required to have a root. When __init__ is run, if a root doesn't already exist, it
creates one with its own name and attaches to it.

__init__()

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. Specific behavior depends on the data, mode, tree, and emdpath arguments.

Calling

```python
>>> 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 determine 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:

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

is equivalent to

```python
>>> 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 repository here

1.6.2 Installation

Currently there are no pip or conda packages and it must be install in one of two ways:

```bash
git clone https://github.com/sezelt/py4D-browser.git
cd py4D-browser
python setup.py
```

Alternatively,

```bash
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 contribute? 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

---

GNU GENERAL PUBLIC LICENSE  
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for  
software and other kinds of works.

The licenses for most software and other practical works are designed  
to take away your freedom to share and change the works. By contrast,  
the GNU General Public License is intended to guarantee your freedom to  
share and change all versions of a program--to make sure it remains free  
software for all its users. We, the Free Software Foundation, use the  
GNU General Public License for most of our software; it applies also to  
any other work released this way by its authors. You can apply it to  
your programs, too.

When we speak of free software, we are referring to freedom, not  
price. Our General Public Licenses are designed to make sure that you  
have the freedom to distribute copies of free software (and charge for  
them if you wish), that you receive source code or can get it if you  
want it, that you can change the software or use pieces of it in new  
free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you  
these rights or asking you to surrender the rights. Therefore, you have  
certain responsibilities if you distribute copies of the software, or if  
you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether  

(continues on next 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
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.


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
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
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.


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
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
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.
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
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.


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
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 subdivide 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 sublicences 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,
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.


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
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

1.8. License
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.
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.

- Additional funding has been provided by the US Department of Energy, Office of Science, Basic Energy Sciences.

- 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

py4DSTEM
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>py4DSTEM.io</td>
<td>75</td>
</tr>
<tr>
<td>py4DSTEM.io.filereaders</td>
<td>75</td>
</tr>
<tr>
<td>py4DSTEM.io.filereaders.empad</td>
<td>75</td>
</tr>
<tr>
<td>py4DSTEM.io.filereaders.read_K2</td>
<td>75</td>
</tr>
<tr>
<td>py4DSTEM.io.filereaders.read_mib</td>
<td>76</td>
</tr>
<tr>
<td>py4DSTEM.io.google_drive_downloader</td>
<td>78</td>
</tr>
<tr>
<td>py4DSTEM.io.google_drive_downloader.gdown</td>
<td>78</td>
</tr>
<tr>
<td>py4DSTEM.io.importfile</td>
<td>79</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy</td>
<td>79</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy.h5py</td>
<td>79</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy.legacy12</td>
<td>79</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy.legacy13</td>
<td>79</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy.read_legacy_12</td>
<td>79</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy.read_legacy_13</td>
<td>80</td>
</tr>
<tr>
<td>py4DSTEM.io.legacy.read_utils</td>
<td>80</td>
</tr>
<tr>
<td>py4DSTEM.io.parsefiletype</td>
<td>81</td>
</tr>
<tr>
<td>py4DSTEM.preprocess.darkreference</td>
<td>81</td>
</tr>
<tr>
<td>py4DSTEM.preprocess.electroncount</td>
<td>83</td>
</tr>
<tr>
<td>py4DSTEM.preprocess.preprocess</td>
<td>85</td>
</tr>
<tr>
<td>py4DSTEM.preprocess.radialbkgrd</td>
<td>87</td>
</tr>
<tr>
<td>py4DSTEM.preprocess.utils</td>
<td>88</td>
</tr>
<tr>
<td>py4DSTEM.process</td>
<td>90</td>
</tr>
<tr>
<td>py4DSTEM.process.calibration</td>
<td>90</td>
</tr>
<tr>
<td>py4DSTEM.process.calibration.braggveectors</td>
<td>90</td>
</tr>
<tr>
<td>py4DSTEM.process.calibration.ellipse</td>
<td>92</td>
</tr>
<tr>
<td>py4DSTEM.process.calibration.origin</td>
<td>94</td>
</tr>
<tr>
<td>py4DSTEM.process.calibration.qpixelsize</td>
<td>99</td>
</tr>
<tr>
<td>py4DSTEM.process.calibration.rotation</td>
<td>100</td>
</tr>
<tr>
<td>py4DSTEM.process.classification</td>
<td>101</td>
</tr>
<tr>
<td>py4DSTEM.process.classification.braggvectorsclassification</td>
<td>101</td>
</tr>
<tr>
<td>py4DSTEM.process.classification.classutils</td>
<td>108</td>
</tr>
<tr>
<td>py4DSTEM.process.classification.featureization</td>
<td>110</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction</td>
<td>115</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.crystal_calibrate</td>
<td>141</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.crystal_phase</td>
<td>143</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.crystal_viz</td>
<td>144</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.flowlines</td>
<td>151</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.sys</td>
<td>155</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.utils</td>
<td>156</td>
</tr>
<tr>
<td>py4DSTEM.process.diffraction.WK_scattering_factors</td>
<td>115</td>
</tr>
<tr>
<td>py4DSTEM.process.diskdetection</td>
<td>157</td>
</tr>
<tr>
<td>py4DSTEM.process.diskdetection.braggvectormap</td>
<td>157</td>
</tr>
<tr>
<td>py4DSTEM.process.diskdetection.diskdetection</td>
<td>161</td>
</tr>
<tr>
<td>py4DSTEM.process.diskdetection.diskdetection_aiml</td>
<td>163</td>
</tr>
<tr>
<td>py4DSTEM.process.diskdetection.threshold</td>
<td>172</td>
</tr>
<tr>
<td>py4DSTEM.process.fit</td>
<td>173</td>
</tr>
<tr>
<td>py4DSTEM.process.fit.fit</td>
<td>173</td>
</tr>
<tr>
<td>py4DSTEM.process.latticevectors</td>
<td>174</td>
</tr>
<tr>
<td>py4DSTEM.process.latticevectors.fit</td>
<td>174</td>
</tr>
<tr>
<td>py4DSTEM.process.latticevectors.index</td>
<td>176</td>
</tr>
<tr>
<td>py4DSTEM.process.latticevectors.initialguess</td>
<td>178</td>
</tr>
<tr>
<td>py4DSTEM.process.latticevectors.strain</td>
<td>180</td>
</tr>
<tr>
<td>py4DSTEM.process.phase</td>
<td>182</td>
</tr>
<tr>
<td>py4DSTEM.process.phase.iterative_base_class</td>
<td>182</td>
</tr>
<tr>
<td>py4DSTEM.process.phase.iterative_constraints</td>
<td>184</td>
</tr>
<tr>
<td>py4DSTEM.process.phase.iterative_dpc</td>
<td>184</td>
</tr>
<tr>
<td>py4DSTM.process.phase.iterative_multislice_ptychography</td>
<td>187</td>
</tr>
<tr>
<td>py4DSTEM.process.phase.iterative_overlap_tomography</td>
<td>193</td>
</tr>
<tr>
<td>py4DSTM.process.phase.iterative_parallax</td>
<td>199</td>
</tr>
<tr>
<td>py4DSTM.process.phase.iterative_pptychography</td>
<td>204</td>
</tr>
</tbody>
</table>

319
py4DSTEM.process.phase.iterative_simultaneous_ptychography, 207
py4DSTEM.process.phase.utils, 212
py4DSTEM.process.probe, 217
py4DSTEM.process.probe.kernel, 222
py4DSTEM.process.probe.probe, 224
py4DSTEM.process.rdf.amorph, 227
py4DSTEM.process.rdf.rdf, 229
py4DSTEM.process.utils, 229
py4DSTEM.process.utils.cross_correlate, 229
py4DSTEM.process.utils.elliptical_coords, 230
py4DSTEM.process.utils.masks, 234
py4DSTEM.process.utils.multicorr, 235
py4DSTEM.process.utils.utils, 236
py4DSTEM.process.virtualdiffraction, 239
py4DSTEM.process.virtualimage, 240
py4DSTEM.process.wholepatternfit, 244
py4DSTEM.process.wholepatternfit.wp_models, 244
py4DSTEM.process.wholepatternfit.wpf, 246
py4DSTEM.process.wholepatternfit.wpf_viz, 246
## Symbols

<table>
<thead>
<tr>
<th>Function</th>
<th>Source</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(emdfile.Array method)</code></td>
<td></td>
<td>290</td>
</tr>
<tr>
<td><code>__init__(emdfile.Custom method)</code></td>
<td></td>
<td>292</td>
</tr>
<tr>
<td><code>__init__(emdfile.Metadata method)</code></td>
<td></td>
<td>292</td>
</tr>
<tr>
<td><code>__init__(emdfile.Node method)</code></td>
<td></td>
<td>294</td>
</tr>
<tr>
<td><code>__init__(emdfile.PointList method)</code></td>
<td></td>
<td>297</td>
</tr>
<tr>
<td><code>__init__(emdfile.PointListArray method)</code></td>
<td></td>
<td>297</td>
</tr>
<tr>
<td><code>__init__(emdfile.Root method)</code></td>
<td></td>
<td>298</td>
</tr>
<tr>
<td><code>__init__(emdfile.RootedNode method)</code></td>
<td></td>
<td>298</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Array method)</code></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.BraggVectors method)</code></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Calibration method)</code></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Custom method)</code></td>
<td></td>
<td>36</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Data method)</code></td>
<td></td>
<td>38</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.DataCube method)</code></td>
<td></td>
<td>38</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.DiffractionSlice method)</code></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Metadata method)</code></td>
<td></td>
<td>53</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Node method)</code></td>
<td></td>
<td>55</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.PointList method)</code></td>
<td></td>
<td>57</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.PointListArray method)</code></td>
<td></td>
<td>59</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.Probe method)</code></td>
<td></td>
<td>62</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.QPoints method)</code></td>
<td></td>
<td>65</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.RealSlice method)</code></td>
<td></td>
<td>67</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.VirtualDiffraction method)</code></td>
<td></td>
<td>70</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.VirtualImage method)</code></td>
<td></td>
<td>72</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.io.filereaders.read_K2.K2DataArray method)</code></td>
<td></td>
<td>76</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method)</code></td>
<td></td>
<td>102</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.classification.featureization.Featurization method)</code></td>
<td></td>
<td>111</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.diffraction.crystal.Crystal method)</code></td>
<td></td>
<td>128</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.diffraction.crystal_bloch.DynamicalMatrixCache method)</code></td>
<td></td>
<td>135</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.diffraction.crystal_phase.CrystalPhase method)</code></td>
<td></td>
<td>143</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.diffraction.utils.Orientation method)</code></td>
<td></td>
<td>156</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.diffraction.utils.OrientationMap method)</code></td>
<td></td>
<td>156</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_dpc.DPCReconstruction method)</code></td>
<td></td>
<td>184</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_mixedstate_ptychography.Ptychography method)</code></td>
<td></td>
<td>189</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_multislice_ptychography.Ptychography method)</code></td>
<td></td>
<td>196</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_overlap_tomography.OverlapTomography method)</code></td>
<td></td>
<td>202</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_parallax.ParallaxReconstruction method)</code></td>
<td></td>
<td>204</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_pychography.Pychography method)</code></td>
<td></td>
<td>209</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.iterative_simultaneous_ptychography.Ptychography method)</code></td>
<td></td>
<td>209</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.ComplexProbe method)</code></td>
<td></td>
<td>218</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.AffineTransform method)</code></td>
<td></td>
<td>220</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.ComplexOverlapKernelDiskLattice method)</code></td>
<td></td>
<td>245</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.DCBackground method)</code></td>
<td></td>
<td>244</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.GaussianBackground method)</code></td>
<td></td>
<td>244</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.GaussianRing method)</code></td>
<td></td>
<td>244</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.KernelDiskLattice method)</code></td>
<td></td>
<td>246</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.SyntheticDiskLattice method)</code></td>
<td></td>
<td>245</td>
</tr>
<tr>
<td><code>__init__(py4DSTEM.process.phase.utils.WPFModelPrototype method)</code></td>
<td></td>
<td>244</td>
</tr>
<tr>
<td><code>aberration_correct()</code></td>
<td>(py4DSTEM.process.phase.iterative_parallax.ParallaxReconstruction method)</td>
<td>207</td>
</tr>
</tbody>
</table>
add() (py4DSTEM.DataCube method), 38
add() (py4DSTEM.PointList method), 57
add() (py4DSTEM.QPoints method), 65
add_annuli() (in module py4DSTEM.visualize.overlay), 253
add_bragg_index_labels() (in module py4DSTEM.visualize.overlay), 253
add_bragg_index_labels() (in module py4DSTEM.visualize.vis_special), 276
add_cartesian_grid() (in module py4DSTEM.visualize.overlay), 253
add_circles() (in module py4DSTEM.visualize.overlay), 253
add_data_by_field() (emdfile.PointList method), 297
add_data_by_field() (py4DSTEM.PointList method), 58
add_data_by_field() (py4DSTEM.QPoints method), 65
add_ellipses() (in module py4DSTEM.visualize.overlay), 253
add_ellipses() (in module py4DSTEM.visualize.vis_special), 276
add_features() (py4DSTEM.process.classification.featureization.Featu- 
ration.featureization.Featureization method), 112
add_fields() (emdfile.PointList method), 297
add_fields() (py4DSTEM.PointList method), 57
add_fields() (py4DSTEM.PointListArray method), 60
add_fields() (py4DSTEM.QPoints method), 65
add_grid_overlay() (in module py4DSTEM.visualize.overlay), 254
add_grid_overlay() (in module py4DSTEM.visualize.vis_grid), 269
add_indices_to_braggpeaks (in module py4DSTEM.process.latticevectors.index), 177
add_indices_to_braggpeaks() (py4DSTEM.BraggVectors method), 30
add_pointlabels() (in module py4DSTEM.visualize.overlay), 254
add_pointlabels() (in module py4DSTEM.visualize.vis_special), 276
add_points() (in module py4DSTEM.visualize.overlay), 254
add_points() (in module py4DSTEM.visualize.vis_special), 276
add_polarelliptical_grid() (in module py4DSTEM.visualize.overlay), 254
add_rectangles() (in module py4DSTEM.visualize.overlay), 254
add_rtheta_grid() (in module py4DSTEM.process.classification.braggvec-
 tors.BraggVectors method), 254
add_scalebar() (in module py4DSTEM.visualize.overlay), 254
add_scalebar() (in module py4DSTEM.visualize.vis_special), 276
dto_2D_array_from_floats() (in module py4DSTEM.process.utils.utils), 253
dto_tree() (emdfile.Node method), 295
dto_tree() (py4DSTEM.Array method), 28
dto_tree() (py4DSTEM.BraggVectors method), 30
dto_tree() (py4DSTEM.DataCube method), 38
dto_tree() (py4DSTEM.DiffractionSlice method), 51
dto_tree() (py4DSTEM.Node method), 55
dto_tree() (py4DSTEM.PointList method), 58
dto_tree() (py4DSTEM.PointListArray method), 60
dto_tree() (py4DSTEM.Probe method), 62
dto_tree() (py4DSTEM.QPoints method), 65
dto_tree() (py4DSTEM.RealSlice method), 67
dto_tree() (py4DSTM.VirtualDiffraction method), 70
dto_tree() (py4DSTEM.VirtualImage method), 72
add_vector() (in module py4DSTEM.visualize.overlay), 254
add_vector() (in module py4DSTEM.visualize.vis_special), 277
AffineTransform (class in py4DSTEM.process.phase.utils), 220
align_and_shift_images() (in module py4DSTEM.process.phase.utils.cross_correlate), 230
align_images_fourier() (in module py4DSTEM.process.phase.utils.cross_correlate), 230
angular_sampling (py4DSTEM.process.phase.iterative_base_class.Phase
property), 184
Array (class in emdfile), 289
Array (class in py4DSTEM), 25
asarray() (py4DSTEM.process.phase.utils.AffineTransform method), 220
asarray3() (py4DSTEM.process.phase.utils.AffineTransform method), 220
astuple() (py4DSTEM.process.phase.utils.AffineTransform method), 220
atomic_colors() (in module py4DSTEM.process.diffraction.crystal_viz), 150
ax_addaxes() (in module py4DSTEM.visualize.vis_RQ), 260
ax_addaxes() (in module py4DSTEM.visualize.vis_special), 277
elliptical_resample_datacube() (in module py4DSTEM.process.utils.elliptical_coords), 232
estimate_global_transformation() (in module py4DSTEM.process.phase.utils), 220
estimate_global_transformation_ransac() (in module py4DSTEM.process.phase.utils), 220
excitation_errors() (py4DSTEM.process.diffraction.crystal.Crystal class method), 131
fit_lattice_vectors_all_DPs() (py4DSTEM.BraggVectors method), 31
fit_lattice_vectors_masked() (in module py4DSTEM.process.latticevectors.fit), 175
fit_origin() (in module py4DSTEM.process.calibration.origin), 97
fit_origin() (py4DSTEM.BraggVectors method), 31
fit_scattering_factor() (in module py4DSTEM.process.rdf.rdf), 229
fit_stack() (in module py4DSTEM.process.rdf.amorph), 227
fourier_resample() (in module py4DSTEM.process.phase.utils), 238
fourier_ring_correlation() (in module py4DSTEM.process.phase.utils), 221
fourier_translation_operator() (in module py4DSTEM.process.phase.utils), 219
from_bragg_vectors() (py4DSTEM.process.classification.featurization.Featurization class method), 111
from_CIF() (py4DSTEM.process.diffraction.crystal.Crystal class method), 128
from_h5() (emdfile.Metadata class method), 293
from_h5() (emdfile.Node class method), 296
from_h5() (py4DSTEM.Array class method), 28
from_h5() (py4DSTEM.BraggVectors class method), 32
from_h5() (py4DSTEM.Calibration class method), 36
from_h5() (py4DSTEM.Custom class method), 36
from_h5() (py4DSTEM.DataCube class method), 41
from_h5() (py4DSTEM.DiffractionSlice class method), 51
from_h5() (py4DSTEM.Metadata class method), 53
from_h5() (py4DSTEM.Node class method), 56
from_h5() (py4DSTEM.PointList class method), 58
from_h5() (py4DSTEM.PointListArray class method), 60
from_h5() (py4DSTEM.Probe class method), 62
from_h5() (py4DSTEM.QPoints class method), 65
from_h5() (py4DSTEM.RealSlice class method), 68
from_h5() (py4DSTEM.VirtualDiffraction class method), 70
from_h5() (py4DSTEM.VirtualImage class method), 73
from_pymatgen_structure() (py4DSTEM.process.diffraction.crystal.Crystal class method), 129
from_unitcell_parameters() (py4DSTEM.process.diffraction.crystal.Crystal class method), 129
fromarray() (py4DSTEM.process.phase.utils.AffineTransform class method), 220

G
GaussianBackground (class in py4DSTEM.process.wholepatternfit.wp_models),
get_darkreference() (in module py4DSTEM.preprocess.darkreference), 81
get_dim() (emdfile.Array method), 291
get_dim() (py4DSTEM.Array method), 27
get_dim() (py4DSTEM.DataCube method), 42
get_dim() (py4DSTEM.DiffractionSlice method), 51
get_dim() (py4DSTEM.Probe method), 62
get_dim() (py4DSTEM.RealSlice method), 68
get_dim() (py4DSTEM.VirtualDiffraction method), 70
get_dim() (py4DSTEM.VirtualImage method), 73
get_dim_name() (emdfile.Array method), 292
get_dim_name() (py4DSTEM.Array method), 27
get_dim_name() (py4DSTEM.DataCube method), 42
get_dim_name() (py4DSTEM.DiffractionSlice method), 51
get_dim_name() (py4DSTEM.Probe method), 62
get_dim_name() (py4DSTEM.RealSlice method), 68
get_dim() (py4DSTEM.VirtualDiffraction method), 70
get_dim() (py4DSTEM.VirtualImage method), 73
get_dim_units() (emdfile.Array method), 291
get_dim_units() (py4DSTEM.Array method), 27
get_dim_units() (py4DSTEM.DataCube method), 42
get_dim_units() (py4DSTEM.DiffractionSlice method), 51
get_dim_units() (py4DSTEM.Probe method), 62
get_dim_units() (py4DSTEM.RealSlice method), 68
get_dim_units() (py4DSTEM.VirtualDiffraction method), 70
get_dim_units() (py4DSTEM.VirtualImage method), 73
get_dp_max() (py4DSTEM.DataCube method), 42
get_dp_mean() (py4DSTEM.DataCube method), 43
get_dp_median() (py4DSTEM.DataCube method), 44
get_dp_from_indexed_peaks() (in module py4DSTEM.process.calibration.apixelsize), 99
get_ewp_filter_function() (in module py4DSTEM.process.calibration.apixelsize), 238
get_from_tree() (emdfile.Node method), 295
get_from_tree() (py4DSTEM.Array method), 28
get_from_tree() (py4DSTEM.BraggVectors method), 28
get_from_tree() (py4DSTEM.Custom method), 37
get_from_tree() (py4DSTEM.DataCube method), 45
get_from_tree() (py4DSTEM.DiffractionSlice method), 51
get_from_tree() (py4DSTEM.Node method), 55
get_from_tree() (py4DSTEM.PointList method), 58
get_from_tree() (py4DSTEM.PointListArray method), 61
get_from_tree() (py4DSTEM.Probe method), 62
get_from_tree() (py4DSTEM.QPoints method), 66
get_from_tree() (py4DSTEM.RealSlice method), 68
get_from_tree() (py4DSTEM.VirtualDiffraction method), 70
get_from_tree() (py4DSTEM.VirtualImage method), 73
get_hdr_bits() (in module py4DSTEM.io.filereaders.read_mib), 78
get_initial_classes() (in module py4DSTEM.process.classification.braggvectorclassification), 108
get_initial_classes_by_cooccurrence() (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method), 103
get_initial_classes_from_images() (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method), 103
get_kernel() (in module py4DSTEM.process.probe.kernel), 222
get_kernel() (py4DSTEM.Probe method), 63
get_lattice_directions_from_scores() (in module py4DSTEM.process.latticevectors.initialguess), 179
get_lattice_vector_lengths() (in module py4DSTEM.process.latticevectors.initialguess), 179
get_mask() (in module py4DSTEM.process.rdf.rdf), 229
get_masked_peaks() (py4DSTEM.BraggVectors method), 32
get_maxima_1D() (in module py4DSTEM.process.utils.utils), 237
get_maxima_2D() (in module py4DSTEM.preprocess.utils), 89
get_mib_depth() (in module py4DSTEM.io.filereaders.read_mib), 78
get_mib_memmap() (in module py4DSTEM.io.filereaders.read_mib), 77
get_N_dataobjects() (in module py4DSTEM.io.legacy.read_utils), 81
get_nice_spacing() (in module py4DSTEM.visualize.overlay), 254
get_origin() (in module py4DSTEM.process.calibration.origin), 95
generate_beamstop() (in module py4DSTEM.process.calibration.origin), 97
generate_beamstop_braggpeaks() (in module py4DSTEM.process.calibration.origin), 97
generate_from_braggpeaks() (in module py4DSTEM.process.calibration.origin), 96
generate_single_dp() (in module py4DSTEM.process.calibration.origin), 95
generate_single_dp_beamstop() (in module py4DSTEM.process.calibration.origin), 96
get_phi() (in module py4DSTEM.process.rdf.rdf), 229
get_pointlist() (emdfile.PointListArray method), 297
get_pointlist() (emdfile.PointListArray method), 297
py4DSTEM.process.latticevectors.index, 176
py4DSTEM.process.latticevectors.initialguess() (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification attribute), 103
py4DSTEM.process.latticevectors.strain, 180
py4DSTEM.process.phase, 182
py4DSTEM.process.phase.iterative_base_class, 182
py4DSTEM.process.phase.iterative_constraints, 184
py4DSTEM.process.phase.iterative_dpc, 184
py4DSTEM.process.phase.iterative_multislice_ptychography, 193
py4DSTEM.process.phase.iterative_overlap_tomography, 199
py4DSTEM.process.phase.iterative_parallax, 204
py4DSTEM.process.phase.iterative_ptychography, 207
py4DSTEM.process.phase.iterative_simultaneous_ptychography, 212
py4DSTEM.process.phase.utils, 217
py4DSTEM.process.probe, 222
py4DSTEM.process.probe.kernel, 222
py4DSTEM.process.probe.probe, 224
py4DSTEM.process.rdf.amorph, 227
py4DSTEM.process.rdf.rdf, 229
py4DSTEM.process.utils, 229
py4DSTEM.process.utils.cross_correlate, 229
py4DSTEM.process.utils.elliptical_coords, 230
py4DSTEM.process.utils.masks, 234
py4DSTEM.process.utils.multicorr, 235
py4DSTEM.process.utils.utils, 236
py4DSTEM.process.virtualdiffraction, 239
py4DSTEM.process.virtualimage, 240
py4DSTEM.process.wholepatternfit, 244
py4DSTEM.process.wholepatternfit.wf_models, 244
py4DSTEM.process.wholepatternfit.wpf, 246
py4DSTEM.process.wholepatternfit.wpf_viz, 246
MultislicePtychographicReconstruction (class in py4DSTEM.process.phase.iterative_multislice_ptychography), 193

N
N_feat (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification attribute), 103
N_meas (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification attribute), 103
NMF() (py4DSTEM.process.classification.featureization.Featureization method), 113
Orientation (class in py4DSTEM.process.diffraction.utils), 156
OrientationCorrelation() (in module py4DSTEM.process.diffraction.flowlines), 156
orientation_plan() (in module py4DSTEM.process.diffraction.crystal_ACOM), 132
OrientationMap (class in py4DSTEM.process.diffraction.utils), 156
OrientationMap (in module py4DSTEM.process.phase.utils), 219
OverlapTomographicReconstruction (class in py4DSTEM.process.phase.iterative_overlap_tomography), 199

P
pad_data_diffraction() (in module py4DSTEM.preprocess.preprocess), 87
pad_Q() (py4DSTEM.DataCube method), 48
ParallaxReconstruction (class in py4DSTEM.process.phase.iterative_parallax), 204
parse_hdr() (in module py4DSTEM.io.filereaders.read_mib), 77
PCA() (py4DSTEM.process.classification.featureization.Featureization method), 113
PhaseReconstruction (class in py4DSTEM.process.phase.iterative_base_class), 182
plot_all_phase_maps() (py4DSTEM.process.diffraction.crystal_phase.CrystalPhase method), 143
plot_diffraction_pattern() (in module py4DSTEM.process.diffraction.crystal_viz), 147
plot_fiber_orientation_maps() (in module py4DSTEM.process.diffraction.crystal_viz), 149
plot_fiber_orientation_maps() (py4DSTEM.process.diffraction.crystal_phase.CrystalPhase method), 122

Q
plot_fourier_probe()
(py4DSTEM.process.phase.iterative_base_class.PhaseReconstruction method), 183

plot_fourier_probe()
(py4DSTEM.process.phase.iterative_mixedstate_ptychography.MixedStatePtychographicReconstruction method), 192

plot_orientation_correlation() (in module py4DSTEM.process.diffraction.flowlines), 155

plot_orientation_maps() (in module py4DSTEM.process.diffraction.crystal_vizu), 148

plot_orientation_maps()
(py4DSTEM.process.diffraction.crystal.Crystal method), 121

plot_orientation_plan() (in module py4DSTEM.process.diffraction.crystal_vizu), 147

plot_orientation_plan() (py4DSTEM.process.diffraction.crystal.Crystal method), 121

plot_orientation_zones() (in module py4DSTEM.process.diffraction.crystal_vizu), 146

plot_orientation_zones()
(py4DSTEM.process.diffraction.crystal.Crystal method), 121

plot_position_correction() (py4DSTEM.process.phase.iterative_base_class.PhaseReconstruction method), 183

plot_ring_pattern() (in module py4DSTEM.process.diffraction.crystal_vizu), 151

plot_scattering_intensity() (in module py4DSTEM.process.diffraction.crystal_vizu), 146

plot_scattering_intensity()
(py4DSTEM.process.diffraction.crystal.Crystal method), 120

plot_strains() (in module py4DSTEM.process.rdf.amorph), 228

plot_structure() (in module py4DSTEM.process.diffraction.crystal_vizu), 144

plot_structure()
(py4DSTEM.process.diffraction.crystal.Crystal method), 119

plot_structure_factors() (in module py4DSTEM.process.diffraction.crystal_vizu), 145

plot_structure_factors()
(py4DSTEM.process.diffraction.crystal.Crystal method), 119

plot_symmetries() (in module py4DSTEM.process.rdf.amorph), 229

PointList (class in py4DSTEM), 57
PointListArray (class in py4DSTEM), 59

PointList (class in emdfile), 297
PointListArray (class in py4DSTEM), 59

polar_aliases (in module py4DSTEM.process.phase.utils), 217
polar_coordinates()
(py4DSTEM.process.phase.utils.ComplexProbe method), 218
polar_symbols (in module py4DSTEM.process.phase.utils), 217
position_detector() (in module py4DSTEM.visualize.virtualimage), 255
position_detector() (py4DSTEM.DataCube method), 48
positions (py4DSTEM.process.diffraction.crystal.Crystal attribute), 128
positions (py4DSTEM.process.phase.iterative_base_class.PhaseReconstruction property), 184
preprocess() (py4DSTEM.process.phase.iterative_base_class.PhaseReconstruction method), 182
preprocess() (py4DSTEM.process.phase.iterative_dpc.DPCReconstruction method), 184
preprocess() (py4DSTEM.process.phase.iterative_mixedstate_ptychography.MixedStatePtychographicReconstruction method), 190
preprocess() (py4DSTEM.process.phase.iterative_multislice_ptychography.MultislicePtychographicReconstruction method), 196
preprocess() (py4DSTEM.process.phase.iterative_overlap_tomography.OverlapTomographicReconstruction method), 202
preprocess() (py4DSTEM.process.phase.iterative_parallax.ParallaxReconstruction method), 204
preprocess() (py4DSTEM.process.phase.iterative_ptychography.PtychographicReconstruction method), 210
preprocess() (py4DSTEM.process.phase.iterative_simultaneous_ptychography.SimultaneousPtychographicReconstruction method), 215
print_h5_tree() (in module emdfile), 298
print_h5_tree() (in module py4DSTEM), 18
print_v13h5_tree() (in module py4DSTEM.io.legacy.read_legacy_13), 80
print_v13h5pyFile_tree() (in module py4DSTEM.io.legacy.read_legacy_13), 80

Probe (class in py4DSTEM), 62
probe_fourier() (py4DSTEM.process.phase.iterative_base_class.PhaseReconstruction property), 183

Projection (in module py4DSTEM.process.phase.utils), 219
PtychographicReconstruction (class in py4DSTEM.process.phase.iterative_ptychography), 207

py4DSTEM.io module, 75
py4DSTEM.io.filereaders module, 75
py4DSTEM.io.filereaders.empad module, 75
show_circles() (in module py4DSTEM.visualize), 252
show_class_BPs() (in module py4DSTEM.visualize.vis_special), 284
show_class_BPs_grid() (in module py4DSTEM.visualize.vis_special), 284
show_complex() (in module py4DSTEM.visualize.vis_special), 284
show_DP_grid() (in module py4DSTEM.visualize.vis_grid), 274
show_ellipses() (in module py4DSTEM.visualize), 252
show_elliptical_fit() (in module py4DSTEM.visualize.vis_special), 285
show_grid_overlay() (in module py4DSTEM.visualize.vis_grid), 274
show_hist() (in module py4DSTEM.visualize), 250
show_image_grid() (in module py4DSTEM.visualize.vis_grid), 274
show_kernel() (in module py4DSTEM.visualize.vis_special), 285
show_lattice_points() (in module py4DSTEM.process.wholepatternfit.wpf_viz), 246
show_lattice_vectors() (in module py4DSTEM.visualize.vis_special), 286
show_max_peak_spacing() (in module py4DSTEM.visualize.vis_special), 286
show_origin_fit() (in module py4DSTEM.visualize.vis_special), 286
show_origin_meas() (in module py4DSTEM.visualize.vis_special), 286
show_pointlabels() (in module py4DSTEM.visualize.vis_special), 286
show_points() (in module py4DSTEM.visualize.vis_grid), 275
show_RQ() (in module py4DSTEM.visualize.vis_RQ), 269
show_qprofile() (in module py4DSTEM.visualize.vis_special), 287
show_rectangles() (in module py4DSTEM.visualize), 267
show_RQ() (in module py4DSTEM.visualize.vis_RQ), 267
show_RQ_axes() (in module py4DSTEM.visualize.vis_RQ), 267
show_RQ_vector() (in module py4DSTEM.visualize.vis_RQ), 267
show_RQ_vectors() (in module py4DSTEM.visualize.vis_RQ), 268
show_selected_dp() (in module py4DSTEM.visualize.vis_special), 287
show_strain() (in module py4DSTEM.visualize.vis_special), 287
show_tree() (emdfile.Node method), 295
show_tree() (py4DSTEM.Array method), 28
show_tree() (py4DSTEM.BraggVectors method), 34
show_tree() (py4DSTEM.Custom method), 37
show_tree() (py4DSTEM.DataCube method), 49
show_tree() (py4DSTEM.DiffractionSlice method), 52
show_tree() (py4DSTEM.Node method), 55
show_tree() (py4DSTEM.PointList method), 59
show_tree() (py4DSTEM.PointListArray method), 61
show_tree() (py4DSTEM.Probe method), 64
show_tree() (py4DSTEM.QPoints method), 66
show_tree() (py4DSTEM.RealSlice method), 69
show_tree() (py4DSTEM.VirtualDiffraction method), 71
show_tree() (py4DSTEM.VirtualImage method), 74
show_voronoi() (in module py4DSTEM.visualize.vis_special), 288
SimultaneousPtychographicReconstruction (class (in py4DSTEM.process.phase.iterative_simultaneous_ptychography), 212
sort() (emdfile.PointList method), 296
sort() (py4DSTEM.PointList method), 57
sort() (py4DSTEM.QPoints method), 66
sort_orientation_maps() (in module py4DSTEM.process.diffraction.utils), 156
spatial_frequencies() (in module py4DSTEM.process.phase.utils), 219
spatial_separation() (in py4DSTEM.process.classification.featureization.Featurization method), 114
split() (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method), 104
split_by_class_index() (py4DSTEM.process.classification.braggvectorclassification.BraggVectorClassification method), 105
subdivide_into_batches() (in module py4DSTEM.process.phase.utils), 219
swap_Qxy() (in module py4DSTEM.preprocess.preprocess), 85
swap_Qxy() (py4DSTEM.DataCube method), 49
swap_RQ() (in module py4DSTEM.preprocess.preprocess), 85
swap_RQ() (py4DSTEM.DataCube method), 49
swap_Rxy() (in module py4DSTEM.preprocess.preprocess), 85
symmetry_reduce_directions() (in module py4DSTEM.process.diffraction.crystal_ACOM), 135
Index