diffsims

Release 0.5.0

Duncan Johnstone, Phillip Crout

Jun 10, 2022

GETTING STARTED

diffsims is an open-source Python library for simulating diffraction.

If simulations performed using diffsims form a part of published work please cite the DOI at the top of this page. You can find demos in the diffsims-demos repository. See the documentation for installation instructions, the API reference, the changelog and more.

diffsims is released under the GPL v3 license.

CHAPTER

ONE

INSTALLATION

diffsims can be installed from Anaconda, the Python Package Index (`pip`), or from source, and supports Python >= 3.6.

We recommend you install it in a conda environment with the Miniconda distribution:

```
conda create --name diffsims-env python=3.10
conda activate diffsims-env
```

If you prefer a graphical interface to manage packages and environments, install the Anaconda distribution instead.

1.1 Anaconda

Anaconda provides the easiest installation. In the Anaconda Prompt, terminal or Command Prompt, install with:

```
conda install diffsims --channel conda-forge
```

If you at a later time need to update the package:

```
conda update diffsims
```

1.2 Pip

To install with `pip`, run the following in the Anaconda Prompt, terminal or Command Prompt:

```
pip install diffsims
```

If you at a later time need to update the package:

```
pip install --upgrade diffsims
```

1.3 Install from source

To install diffsims from source, clone the repository from GitHub:

```
git clone https://github.com/pyxem/diffsims.git
cd diffsims
pip install -e .
```

CHAPTER

TWO

USAGE

See the demos for how to use diffsims.

CHAPTER

THREE

API REFERENCE

This reference manual details the public modules, classes, and functions in diffsims, as generated from their docstrings. Many of the docstrings contain examples. See the user guide for other ways to use diffsims.

Caution: diffsims is in an alpha stage, so there may be breaking changes with each release.

The list of top modules:

crystallography	Generation of reciprocal lattice vectors (crystal plane, reflector, g, hkl) for a crystal structure.
generators	Generation of diffraction simulations and libraries, and lists of rotations.
libraries	Diffraction, structure and vector libraries.
sims	Diffraction simulations.
structure_factor	Calculation of scattering factors and structure factors.
utils	Diffraction utilities used by the other modules.

3.1 crystallography

Generation of reciprocal lattice vectors (crystal plane, reflector, g, hkl) for a crystal structure.

ReciprocalLatticeVector(phase[, xyz, hkl, hkil])	Reciprocal lattice vectors (hkl) for use in electron diffraction analysis and simulation.
ReciprocalLatticePoint(phase, hkl)	*[Deprecated]* Reciprocal lattice point (or crystal plane, reflector, g, etc.) with Miller indices, length of the reciprocal lattice vectors and other relevant structure_factor parameters.
get_equivalent_hkl(hkl, operations[, ...])	Return symmetrically equivalent Miller indices.
get_highest_hkl(lattice[, min_dspacing])	Return the highest Miller indices hkl of the plane with a direct space interplanar spacing greater than but closest to a lower threshold.
get_hkl(highest_hkl)	Return a list of planes from a set of highest Miller indices.

3.1.1 ReciprocalLatticeVector

Methods

angle_with(other[, use_symmetry])	Calculate angles between reciprocal lattice vectors, possibly using symmetrically equivalent vectors to find the smallest angle under symmetry.
calculate_structure_factor([scattering_params])	Populate structure_factor with the complex kinematical structure factor F_{hkl} for each vector.
calculate_theta(voltage)	Populate theta with the Bragg angle $theta_B$ in radians.
cross(other)	Cross product between reciprocal lattice vectors producing zone axes $[uvw]$ or $[UVTW]$ in the direct lattice.
deepcopy()	Get a deepcopy of the vectors.
draw_circle([projection, figure, ...])	Draw great or small circles with a given *opening_angle* to to the vectors in the stereographic projection.
dot(other)	Dot product of the vectors with other reciprocal lattice vectors.
dot_outer(other)	Outer dot product of the vectors with other reciprocal lattice vectors.
flatten()	A new instance with these reciprocal lattice vectors in a single column.
from_highest_hkl(phase, hkl)	Create a set of unique reciprocal lattice vectors from three highest indices.
from_min_dspacing(phase[, min_dspacing])	Create a set of unique reciprocal lattice vectors with a a direct space interplanar spacing greater than a lower threshold.
from_miller(miller)	Create a new instance from a `Miller` instance.
get_circle([opening_angle, steps])	Get vectors delineating great or small circle(s) with a given *opening_angle* about each vector.
get_hkl_sets()	Get unique sets of hkl for the vectors and the indices of vectors in each set.
print_table()	Table with indices, structure factor values and multiplicity of each set of hkl.
reshape(*shape)	A new instance with these reciprocal lattice vectors reshaped.
sanitise_phase()	Sanitise the `phase` inplace for calculation of structure factors.
scatter([projection, figure, axes_labels, ...])	Plot vectors in the stereographic projection.
squeeze()	A new instance with these reciprocal lattice vectors where singleton dimensions are removed.
stack(sequence)	A new instance from a sequence of reciprocal lattice vectors.
symmetrise([return_multiplicity, return_index])	Unique vectors symmetrically equivalent to the vectors.
to_miller()	Return the vectors as a `Miller` instance.
to_polar()	Return the azimuth ϕ, polar θ, and radial r spherical coordinates, the angles in radians.
transpose(*axes)	A new instance with the navigation shape of these reciprocal lattice vectors transposed.
unique([use_symmetry, return_index])	The unique vectors.

class diffsims.crystallography.**ReciprocalLatticeVector**(*phase*, *xyz=None*, *hkl=None*, *hkil=None*)

Bases: Vector3d

Reciprocal lattice vectors (hkl) for use in electron diffraction analysis and simulation.

All lengths are assumed to be given in Å or inverse Å.

This class extends `orix.vector.Vector3d` to reciprocal lattice vectors (hkl) specifically for diffraction experiments and simulations. It is thus different from `orix.vector.Miller`, which is a general class for Miller indices both in reciprocal *and* direct space. It supports relevant methods also supported in *Miller*, like obtaining a set of vectors from a minimal interplanar spacing.

Create a set of reciprocal lattice vectors from (hkl) or $(hkil)$.

The vectors are internally as cartesian coordinates in the `data` attribute.

Parameters

- **phase** (*orix.crystal_map.Phase*) – A phase with a crystal lattice and symmetry.
- **xyz** (*numpy.ndarray, list, or tuple, optional*) – Cartesian coordinates of indices of reciprocal lattice vector(s) `hkl`. Default is `None`. This, `hkl`, or `hkil` is required.
- **hkl** (*numpy.ndarray, list, or tuple, optional*) – Indices of reciprocal lattice vector(s). Default is `None`. This, `xyz`, or `hkil` is required.
- **hkil** (*numpy.ndarray, list, or tuple, optional*) – Indices of reciprocal lattice vector(s), often preferred over `hkl` in trigonal and hexagonal lattices. Default is `None`. This, `xyz`, or `hkl` is required.

Examples

```
>>> from diffpy.structure import Atom, Lattice, Structure
>>> from orix.crystal_map import Phase
>>> from diffsims.crystallography import ReciprocalLatticeVector
>>> phase = Phase(
...     "al",
...     space_group=225,
...     structure=Structure(
...         lattice=Lattice(4.04, 4.04, 4.04, 90, 90, 90),
...         atoms=[Atom("Al", [0, 0, 1])],
...     ),
... )
>>> rlv = ReciprocalLatticeVector(phase, hkl=[[1, 1, 1], [2, 0, 0]])
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

property allowed

Return whether vectors diffract according to diffraction selection rules assuming kinematic scattering theory.

Returns

allowed – Boolean array.

Return type

numpy.ndarray

Examples

```
>>> from diffpy.structure import Atom, Lattice, Structure
>>> from orix.crystal_map import Phase
>>> from diffsims.crystallography import ReciprocalLatticeVector
>>> phase = Phase(
...     "al",
...     space_group=225,
...     structure=Structure(
...         lattice=Lattice(4.04, 4.04, 4.04, 90, 90, 90),
...         atoms=[Atom("Al", [0, 0, 1])],
...     ),
... )
>>> rlv = ReciprocalLatticeVector(
...     phase, hkl=[[1, 0, 0], [2, 0, 0]]
... )
>>> rlv.allowed
array([False,  True])
```

angle_with(*other*, *use_symmetry=False*)

Calculate angles between reciprocal lattice vectors, possibly using symmetrically equivalent vectors to find the smallest angle under symmetry.

Parameters

- **other** (ReciprocalLatticeVector) – Other vectors of compatible shape to the vectors.
- **use_symmetry** (*bool, optional*) – Whether to consider equivalent vectors to find the smallest angle under symmetry. Default is False.

Returns

The angle between the vectors, in radians. If use_symmetry=True, the angles are the smallest under symmetry.

Return type

numpy.ndarray

property azimuth

Azimuth spherical coordinate, i.e. the angle $\phi \in [0, 2\pi]$ from the positive z-axis to a point on the sphere, according to the ISO 31-11 standard [SphericalWolfram].

Return type

numpy.ndarray

calculate_structure_factor(*scattering_params='xtables'*)

Populate *structure_factor* with the complex kinematical structure factor F_{hkl} for each vector.

Parameters

scattering_params (*str*) – Which atomic scattering factors to use, either "xtables" (default) or "lobato".

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

A unit cell with all asymmetric atom positions is required to calculate structure factors

```
>>> rlv.phase.structure
[Al   0.000000 0.000000 1.000000 1.0000]
>>> rlv.sanitise_phase()
>>> rlv.phase.structure
[Al   0.000000 0.000000 0.000000 1.0000,
 Al   0.000000 0.500000 0.500000 1.0000,
 Al   0.500000 0.000000 0.500000 1.0000,
 Al   0.500000 0.500000 0.000000 1.0000]
```

```
>>> rlv.calculate_structure_factor()
>>> rlv.structure_factor
array([8.46881663-1.55569638e-15j, 7.04777513-8.63103525e-16j])
```

Default atomic scattering factors are from the International Tables of Crystallography Vol. C Table 4.3.2.3. Alternative scattering factors are available from Lobato and Van Dyck Acta Cryst. (2014). A70, 636-649 https://doi.org/10.1107/S205327331401643X

```
>>> rlv.calculate_structure_factor("lobato")
>>> rlv.structure_factor
array([8.44934816-1.55212008e-15j, 7.0387957 -8.62003862e-16j])
```

calculate_theta(*voltage*)

Populate *theta* with the Bragg angle $theta_B$ in radians.

Assumes `phase.structure` lattice parameters and Debye-Waller factors are expressed in Ångströms.

Parameters
voltage (*float*) – Beam energy in V.

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

```
>>> rlv.calculate_theta(20e3)
>>> rlv.theta
array([0.0184036 , 0.02125105])
>>> rlv.calculate_theta(200e3)
```

(continues on next page)

(continued from previous page)

```
>>> rlv.theta
array([0.00537583, 0.00620749])
```

property coordinate_format

Vector coordinate format, either "hkl" or "hkil".

Return type
str

Examples

See *ReciprocalLatticeVector* for the creation of rlv

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.coordinate_format
'hkl'
>>> rlv.coordinate_format = "hkil"
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[ 1.  1. -2.  1.]
 [ 2.  0. -2.  0.]]
```

property coordinates

Miller or Miller-Bravais indices.

Returns
coordinates – Miller indices if coordiante_format is "hkl" or Miller-Bravais indices if it is "hkil".

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of rlv

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.coordinates
array([[1., 1., 1.],
       [2., 0., 0.]])
>>> rlv.coordinate_format = "hkil"
>>> rlv.coordinates
array([[ 1.,  1., -2.,  1.],
       [ 2.,  0., -2.,  0.]])
```

cross(*other*)

Cross product between reciprocal lattice vectors producing zone axes $[uvw]$ or $[UVTW]$ in the direct lattice.

Parameters
other (ReciprocalLatticeVector) – Other vectors of compatible shape to the vectors.

Returns
Direct lattice vector(s) $[uvw]$ or $UVTW$, depending on whether the vector's *coordinate_format* is hkl or hkil, respectively.

Return type
orix.vector.Miller

property data

deepcopy()

Get a deepcopy of the vectors.

Return type
ReciprocalLatticeVector

dim = 3

The number of dimensions for this object.

Type
int

dot(*other*)

Dot product of the vectors with other reciprocal lattice vectors.

Parameters
other (ReciprocalLatticeVector) – Other vectors of compatible shape to the vectors.

Return type
numpy.ndarray

dot_outer(*other*)

Outer dot product of the vectors with other reciprocal lattice vectors.

The dot product for every combination of the vectors are computed.

Parameters
other (ReciprocalLatticeVector) – Other vectors of compatible shape to the vectors.

Return type
numpy.ndarray

draw_circle(*projection='stereographic'*, *figure=None*, *opening_angle=1.5707963267948966*, *steps=100*, *reproject=False*, *axes_labels=None*, *hemisphere=None*, *show_hemisphere_label=None*, *grid=None*, *grid_resolution=None*, *figure_kwargs=None*, *reproject_plot_kwargs=None*, *return_figure=False*, ***kwargs*)

Draw great or small circles with a given *opening_angle* to to the vectors in the stereographic projection.

A vector must be present in the current hemisphere for its circle to be drawn.

Parameters

- **projection** (*str, optional*) – Which projection to use. The default is "stereographic", the only current option.

- **figure** (*matplotlib.figure.Figure, optional*) – Which figure to plot onto. Default is None, which creates a new figure.
- **opening_angle** (*float or numpy.ndarray, optional*) – Opening angle(s) around the vector(s). Default is $\pi/2$, giving a great circle. If an array is passed, its size must be equal to the number of vectors.
- **steps** (*int, optional*) – Number of vectors to describe each circle, default is 100.
- **reproject** (*bool, optional*) – Whether to reproject parts of the circle(s) visible on the other hemisphere. Reprojection is achieved by reflection of the circle(s) parts located on the other hemisphere in the projection plane. Ignored if hemisphere is "both". Default is False.
- **axes_labels** (*list of str, optional*) – Reference frame axes labels, defaults to [None, None, None].
- **hemisphere** (*str, optional*) – Which hemisphere(s) to plot the vectors in, defaults to "None", which means "upper" if a new figure is created, otherwise adds to the current figure's hemispheres. Options are "upper", "lower", and "both", which plots two projections side by side.
- **show_hemisphere_label** (*bool, optional*) – Whether to show hemisphere labels "upper" or "lower`. Default is True if *hemisphere* is "both", otherwise False.
- **grid** (*bool, optional*) – Whether to show the azimuth and polar grid. Default is whatever *axes.grid* is set to in `matplotlib.rcParams`.
- **grid_resolution** (*tuple, optional*) – Azimuth and polar grid resolution in degrees, as a tuple. Default is whatever is default in `stereographic_grid`.
- **figure_kwargs** (*dict, optional*) – Dictionary of keyword arguments passed to `matplotlib.pyplot.subplots()`.
- **reproject_plot_kwargs** (*dict, optional*) – Keyword arguments passed to `matplotlib.axes.Axes.plot()` to alter the appearance of parts of the circle(s) visible on the other hemisphere if *reproject* is True. These lines are dashed by default. Values used for circle(s) on the current hemisphere are used unless values are passed here.
- **return_figure** (*bool, optional*) – Whether to return the figure (default is False).
- **kwargs** – Keyword arguments passed to `matplotlib.axes.Axes.plot()` to alter the circles' appearance.

Returns

fig – The created figure, returned if *return_figure* is True.

Return type

matplotlib.figure.Figure

Notes

This is a somewhat customizable convenience method which creates a figure with axes using `StereographicPlot`, however, it is meant for quick plotting and prototyping. This figure and the axes can also be created using Matplotlib directly, which is more customizable.

See also:

`orix.plot.StereographicPlot`, `orix.vector.Vector3d.get_circle`

property dspacing

Direct lattice interplanar spacing $d = 1/g$.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

Lattice parameters are given in

```
>>> rlv.phase.structure.lattice
Lattice(a=4.04, b=4.04, c=4.04, alpha=90, beta=90, gamma=90)
```

so d is given in

```
>>> rlv.dspacing
array([2.33249509, 2.02      ])
```

flatten()

A new instance with these reciprocal lattice vectors in a single column.

Return type
ReciprocalLatticeVector

classmethod from_highest_hkl(*phase*, *hkl*)

Create a set of unique reciprocal lattice vectors from three highest indices.

Parameters

- **phase** (*orix.crystal_map.Phase*) – A phase with a crystal lattice and symmetry.
- **hkl** (*numpy.ndarray*, *list*, *or tuple*) – Three highest reciprocal lattice vector indices.

Examples

```
>>> from diffpy.structure import Atom, Lattice, Structure
>>> from orix.crystal_map import Phase
>>> from diffsims.crystallography import ReciprocalLatticeVector
>>> phase = Phase(
...     "al",
...     space_group=225,
...     structure=Structure(
...         lattice=Lattice(4.04, 4.04, 4.04, 90, 90, 90),
...         atoms=[Atom("Al", [0, 0, 1])],
...     ),
... )
```

(continues on next page)

(continued from previous page)

```
>>> rlv = ReciprocalLatticeVector.from_highest_hkl(phase, [3, 3, 3])
>>> rlv
ReciprocalLatticeVector (342,), al (m-3m)
[[ 3.  3.  3.]
 [ 3.  3.  2.]
 [ 3.  3.  1.]
 ...
 [-3. -3. -1.]
 [-3. -3. -2.]
 [-3. -3. -3.]]
```

Vectors are included regardless of whether they are kinematically allowed or not

```
>>> rlv.allowed.all()
False
```

classmethod from_miller(*miller*)

Create a new instance from a `Miller` instance.

Parameters

miller (*orix.vector.Miller*) – Reciprocal lattice vectors $(hk(i)l)$.

Return type

ReciprocalLatticeVector

Examples

```
>>> from diffpy.structure import Atom, Lattice, Structure
>>> from orix.crystal_map import Phase
>>> from orix.vector import Miller
>>> from diffsims.crystallography import ReciprocalLatticeVector
>>> phase = Phase(
...     "al",
...     space_group=225,
...     structure=Structure(
...         lattice=Lattice(4.04, 4.04, 4.04, 90, 90, 90),
...         atoms=[Atom("Al", [0, 0, 1])],
...     ),
... )
>>> miller = Miller(hkl=[[1, 1, 1], [2, 0, 0]], phase=phase)
>>> miller
Miller (2,), point group m-3m, hkl
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv = ReciprocalLatticeVector.from_miller(miller)
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.to_miller()
Miller (2,), point group m-3m, hkl
```

(continues on next page)

(continued from previous page)

```
[[1. 1. 1.]
 [2. 0. 0.]]
```

classmethod from_min_dspacing(*phase*, *min_dspacing=0.7*)

Create a set of unique reciprocal lattice vectors with a a direct space interplanar spacing greater than a lower threshold.

Parameters

- **phase** (*orix.crystal_map.Phase*) – A phase with a crystal lattice and symmetry.
- **min_dspacing** (*float, optional*) – Smallest interplanar spacing to consider. Default is 0.7, in the unit used to define the lattice parameters in `phase`, which is assumed to be Ångström.

Examples

```
>>> from diffpy.structure import Atom, Lattice, Structure
>>> from orix.crystal_map import Phase
>>> from diffsims.crystallography import ReciprocalLatticeVector
>>> phase = Phase(
...     "al",
...     space_group=225,
...     structure=Structure(
...         lattice=Lattice(4.04, 4.04, 4.04, 90, 90, 90),
...         atoms=[Atom("Al", [0, 0, 1])],
...     ),
... )
>>> rlv = ReciprocalLatticeVector.from_min_dspacing(phase)
>>> rlv
ReciprocalLatticeVector (798,), al (m-3m)
[[ 5.  2.  2.]
 [ 5.  2.  1.]
 [ 5.  2.  0.]
 ...
 [-5. -2.  0.]
 [-5. -2. -1.]
 [-5. -2. -2.]]
>>> rlv.dspacing.min()
0.7032737300610338
```

Vectors are included regardless of whether they are kinematically allowed or not

```
>>> rlv = ReciprocalLatticeVector.from_min_dspacing(phase, 1)
>>> rlv.size
256
>>> rlv.allowed.all()
False
```

get_circle(*opening_angle=1.5707963267948966*, *steps=100*)

Get vectors delineating great or small circle(s) with a given *opening_angle* about each vector.

Used for plotting plane traces in stereographic projections.

Parameters

- **opening_angle** (*float or numpy.ndarray, optional*) – Opening angle(s) around the vector(s). Default is $\pi/2$, giving a great circle. If an array is passed, its size must be equal to the number of vectors.
- **steps** (*int, optional*) – Number of vectors to describe each circle, default is 100.

Returns
circles – Vectors delineating circles with the *opening_angle* about the vectors.

Return type
Vector3d

Notes

A set of *steps* number of vectors equal to each vector is rotated twice to obtain a circle: (1) About a perpendicular vector to the current vector at *opening_angle* and (2) about the current vector in a full circle.

get_hkl_sets()

Get unique sets of hkl for the vectors and the indices of vectors in each set.

Returns
hkl_sets – Dictionary with (h, k, l) as keys and a tuple with numpy.ndarray with integers of the vectors (possibly multi-dimensional) in each set. The keys (h, k, l) are rounded to six decimals so that applying integer values (h, k, l) as dictionary keys work.

Return type
defaultdict

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> hkl_sets = rlv.get_hkl_sets()
>>> hkl_sets
defaultdict(<class 'tuple'>, {(2.0, 0.0, 0.0): (array([1]),), (1.0, 1.0, 1.0):␣
↪(array([0]),)})
>>> hkl_sets[2, 0, 0]
(array([1]),)
>>> rlv[hkl_sets[2, 0, 0]]
ReciprocalLatticeVector (1,), al (m-3m)
[[2. 0. 0.]]
```

property gspacing

Reciprocal lattice vector spacing g.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

Lattice parameters are given in

```
>>> rlv.phase.structure.lattice
Lattice(a=4.04, b=4.04, c=4.04, alpha=90, beta=90, gamma=90)
```

so g is given in $^-1$

```
>>> rlv.gspacing
array([0.42872545, 0.4950495 ])
```

property h

First reciprocal lattice vector index.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.h
array([1., 2.])
```

property has_hexagonal_lattice

Whether the crystal lattice is hexagonal/trigonal.

Return type
bool

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.has_hexagonal_lattice
False
```

property hkil

Miller-Bravais indices.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.hkil
array([[ 1.,  1., -2.,  1.],
       [ 2.,  0., -2.,  0.]])
```

property hkl

Miller indices.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.hkl
array([[1., 1., 1.],
       [2., 0., 0.]])
```

property i

Third reciprocal lattice vector index in 4-index Miller-Bravais indices, equal to $-(h + k)$.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.i
array([-2., -2.])
```

property k

Second reciprocal lattice vector index.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.k
array([1., 0.])
```

property l

Third reciprocal lattice vector index, or fourth index in 4-index Miller Bravais indices.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.l
array([1., 0.])
```

property multiplicity

Number of symmetrically equivalent directions per vector.

Returns
mult

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.multiplicity
array([8, 6])
```

property ndim

The number of navigation dimensions of the instance.

For example, if *data* has shape (4, 5, 6), *ndim* is 3.

Type
int

property polar

Polar spherical coordinate, i.e. the angle $\theta \in [0, \pi]$ from the positive z-axis to a point on the sphere, according to the ISO 31-11 standard [SphericalWolfram].

Return type
numpy.ndarray

print_table()

Table with indices, structure factor values and multiplicity of each set of hkl.

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.print_table()
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    2.332     nan      nan       nan       8
 2 0 0    2.020     nan      nan       nan       6
>>> rlv.sanitise_phase()
>>> rlv.calculate_structure_factor()
>>> rlv.print_table()
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    2.332     8.5     71.7      100.0      8
 2 0 0    2.020     7.0     49.7       69.3      6
```

property radial

Radial spherical coordinate, i.e. the distance from a point on the sphere to the origin, according to the ISO 31-11 standard [SphericalWolfram].

Return type
numpy.ndarray

reshape(**shape*)

A new instance with these reciprocal lattice vectors reshaped.

Parameters
***shape** (*int*) – Multiple integers designating the new shape.

Return type
ReciprocalLatticeVector

sanitise_phase()

Sanitise the `phase` inplace for calculation of structure factors.

The phase is sanitised when it's `structure` has an expanded unit cell with all symmetrically atom positions filled, and the atoms have their `element` set to a string, e.g. "Al".

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.phase.structure
[Al   0.000000 0.000000 1.000000 1.0000]
>>> rlv.sanitise_phase()
>>> rlv.phase.structure
[Al   0.000000 0.000000 0.000000 1.0000,
 Al   0.000000 0.500000 0.500000 1.0000,
 Al   0.500000 0.000000 0.500000 1.0000,
 Al   0.500000 0.500000 0.000000 1.0000]
```

scatter(*projection='stereographic'*, *figure=None*, *axes_labels=None*, *vector_labels=None*, *hemisphere=None*, *reproject=False*, *show_hemisphere_label=None*, *grid=None*, *grid_resolution=None*, *figure_kwargs=None*, *reproject_scatter_kwargs=None*, *text_kwargs=None*, *return_figure=False*, ***kwargs*)

Plot vectors in the stereographic projection.

Parameters

- **projection** (*str, optional*) – Which projection to use. The default is "stereographic", the only current option.
- **figure** (*matplotlib.figure.Figure, optional*) – Which figure to plot onto. Default is None, which creates a new figure.
- **axes_labels** (*list of str, optional*) – Reference frame axes labels, defaults to [None, None, None].
- **vector_labels** (*list of str, optional*) – Vector text labels, which by default aren't added.
- **hemisphere** (*str, optional*) – Which hemisphere(s) to plot the vectors in, defaults to "None", which means "upper" if a new figure is created, otherwise adds to the current figure's hemispheres. Options are "upper", "lower", and "both", which plots two projections side by side.
- **reproject** (*bool, optional*) – Whether to reproject vectors onto the chosen hemisphere. Reprojection is achieved by reflection of the vectors located on the opposite hemisphere in the projection plane. Ignored if *hemisphere* is "both". Default is False.
- **show_hemisphere_label** (*bool, optional*) – Whether to show hemisphere labels "upper" or "lower". Default is True if *hemisphere* is "both", otherwise False.
- **grid** (*bool, optional*) – Whether to show the azimuth and polar grid. Default is whatever *axes.grid* is set to in `matplotlib.rcParams`.
- **grid_resolution** (*tuple, optional*) – Azimuth and polar grid resolution in degrees, as a tuple. Default is whatever is default in `stereographic_grid`.
- **figure_kwargs** (*dict, optional*) – Dictionary of keyword arguments passed to `matplotlib.pyplot.subplots()`.
- **reproject_scatter_kwargs** (*dict, optional*) – Dictionary of keyword arguments for the reprojected scatter points which is passed to `scatter()`, which passes these on to

matplotlib.axes.Axes.scatter(). The default marker style for reprojected vectors is "+". Values used for vector(s) on the visible hemisphere are used unless another value is passed here.

- **text_kwargs** (*dict, optional*) – Dictionary of keyword arguments passed to text(), which passes these on to matplotlib.axes.Axes.text().
- **return_figure** (*bool, optional*) – Whether to return the figure (default is False).
- **kwargs** (*dict, optional*) – Keyword arguments passed to scatter(), which passes these on to matplotlib.axes.Axes.scatter().

Returns
fig – The created figure, returned if *return_figure* is True.

Return type
matplotlib.figure.Figure

Notes

This is a somewhat customizable convenience method which creates a figure with axes using StereographicPlot, however, it is meant for quick plotting and prototyping. This figure and the axes can also be created using Matplotlib directly, which is more customizable.

See also:

orix.plot.StereographicPlot

property scattering_parameter

Scattering parameter $0.5 \cdot g$.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

Lattice parameters are given in

```
>>> rlv.phase.structure.lattice
Lattice(a=4.04, b=4.04, c=4.04, alpha=90, beta=90, gamma=90)
```

so the scattering parameters are given in $^-1$

```
>>> rlv.scattering_parameter
array([0.21436272, 0.24752475])
```

property shape

Shape of the object.

Type
tuple

property size

Total number of entries in this object.

Type
int

squeeze()

A new instance with these reciprocal lattice vectors where singleton dimensions are removed.

Return type
ReciprocalLatticeVector

classmethod stack(*sequence*)

A new instance from a sequence of reciprocal lattice vectors.

Parameters
sequence (*iterable of ReciprocalLatticeVector*) – One or more sets of compatible reciprocal lattice vectors.

Return type
ReciprocalLatticeVector

property structure_factor

Kinematical structure factors F.

Returns
structure_factor – Complex array. Filled with `None` if *calculate_structure_factor()* hasn't been called yet.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

Kinematical structure factors are by default not calculated

```
>>> rlv.structure_factor
array([nan+0.j, nan+0.j])
```

A unit cell with all asymmetric atom positions is required to calculate structure factors

```
>>> rlv.phase.structure
[Al   0.000000 0.000000 1.000000 1.0000]
>>> rlv.sanitise_phase()
>>> rlv.phase.structure
[Al   0.000000 0.000000 0.000000 1.0000,
 Al   0.000000 0.500000 0.500000 1.0000,
 Al   0.500000 0.000000 0.500000 1.0000,
 Al   0.500000 0.500000 0.000000 1.0000]
```

```
>>> rlv.calculate_structure_factor()
>>> rlv.structure_factor
array([8.46881663-1.55569638e-15j, 7.04777513-8.63103525e-16j])
```

symmetrise(*return_multiplicity=False*, *return_index=False*)

Unique vectors symmetrically equivalent to the vectors.

Parameters

- **return_multiplicity** (*bool, optional*) – Whether to return the multiplicity of each vector. Default is `False`.
- **return_index** (*bool, optional*) – Whether to return the index into the vectors for the returned symmetrically equivalent vectors. Default is `False`.

Returns

- *ReciprocalLatticeVector* – Flattened symmetrically equivalent vectors.
- **multiplicity** (*numpy.ndarray*) – Multiplicity of each vector. Returned if `return_multiplicity=True`.
- **idx** (*numpy.ndarray*) – Index into the vectors for the symmetrically equivalent vectors. Returned if `return_index=True`.

Examples

See *ReciprocalLatticeVector* for the creation of `rlv`

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.symmetrise()
ReciprocalLatticeVector (14,), al (m-3m)
[[ 1.  1.  1.]
 [-1.  1.  1.]
 [-1. -1.  1.]
 [ 1. -1.  1.]
 [ 1. -1. -1.]
 [ 1.  1. -1.]
 [-1.  1. -1.]
 [-1. -1. -1.]
 [ 2.  0.  0.]
 [ 0.  2.  0.]
 [-2.  0.  0.]
 [ 0. -2.  0.]
 [ 0.  0.  2.]
 [ 0.  0. -2.]]
>>> _, mult, idx = rlv.symmetrise(
...     return_multiplicity=True, return_index=True
... )
>>> mult
array([8, 6])
>>> idx
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
```

property theta

Twice the Bragg angle.

Returns
theta – Filled with None if *calculate_theta()* hasn't been called yet.

Return type
numpy.ndarray

Examples

See *ReciprocalLatticeVector* for the creation of rlv

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
```

Bragg angles are by default not calculated

```
>>> rlv.theta
array([nan, nan])
```

```
>>> rlv.calculate_theta(20e3)
>>> rlv.theta
array([0.0184036 , 0.02125105])
```

to_miller()

Return the vectors as a Miller instance.

Return type
orix.vector.Miller

Examples

See *ReciprocalLatticeVector* for the creation of rlv

```
>>> rlv
ReciprocalLatticeVector (2,), al (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]]
>>> rlv.to_miller()
Miller (2,), point group m-3m, hkl
[[1. 1. 1.]
 [2. 0. 0.]]
```

to_polar()

Return the azimuth ϕ, polar θ, and radial r spherical coordinates, the angles in radians. The coordinates are defined as in the ISO 31-11 standard [SphericalWolfram].

Returns
azimuth, polar, radial

Return type
numpy.ndarray

`transpose`(**axes*)

A new instance with the navigation shape of these reciprocal lattice vectors transposed.

If *ndim* is originally 2, then order may be undefined. In this case the first two dimensions will be transposed.

Parameters

***axes** (*int, optional*) – Transposed axes order. Only navigation axes need to be defined. May be undefined if the vectors only contain two navigation dimensions.

Return type

ReciprocalLatticeVector

`unique`(*use_symmetry=False*, *return_index=False*)

The unique vectors.

Parameters

- **use_symmetry** (*bool, optional*) – Whether to consider equivalent vectors to compute the unique vectors. Default is `False`.
- **return_index** (*bool, optional*) – Whether to return the indices of the (flattened) data where the unique entries were found. Default is `False`.

Returns

- *ReciprocalLatticeVector* – Flattened unique vectors.
- **idx** (*numpy.ndarray*) – Indices of the unique data in the (flattened) array.

`property unit`

Unit reciprocal lattice vectors.

Return type

ReciprocalLatticeVector

`property x`

This vector's x data.

Type

numpy.ndarray

`property xyz`

This vector's components, useful for plotting.

Type

tuple of ndarray

`property y`

This vector's y data.

Type

numpy.ndarray

`property z`

This vector's z data.

Type

numpy.ndarray

3.1.2 ReciprocalLatticePoint

class diffsims.crystallography.**ReciprocalLatticePoint**(*phase*, *hkl*)

Bases: object

[Deprecated] Reciprocal lattice point (or crystal plane, reflector, g, etc.) with Miller indices, length of the reciprocal lattice vectors and other relevant structure_factor parameters.

Notes

Deprecated since version 0.5: Class `ReciprocalLatticePoint` is deprecated and will be removed in version 0.6. Use *ReciprocalLatticeVector* instead.

property allowed

Return whether planes diffract according to structure_factor selection rules assuming kinematical scattering theory.

calculate_structure_factor(*method=None*, *voltage=None*)

Populate *self.structure_factor* with the structure factor F for each plane.

Parameters

- **method** (*str, optional*) – Either "kinematical" for kinematical X-ray structure factors or "doyleturner" for structure factors using Doyle-Turner atomic scattering factors. If None (default), kinematical structure factors are calculated.
- **voltage** (*float, optional*) – Beam energy in V used when *method=doyleturner*.

calculate_theta(*voltage*)

Populate *self.theta* with the Bragg angle $theta_B$ for each plane.

Parameters

voltage (*float*) – Beam energy in V.

property dspacing

Return `np.ndarray` of direct lattice interplanar spacings.

classmethod from_highest_hkl(*phase*, *highest_hkl*)

Create a CrystalPlane object populated by unique Miller indices below, but including, a set of higher indices.

Parameters

- **phase** (*orix.crystal_map.phase_list.Phase*) – A phase container with a crystal structure and a space and point group describing the allowed symmetry operations.
- **highest_hkl** (*np.ndarray, list, or tuple of int*) – Highest Miller indices to consider (including).

classmethod from_min_dspacing(*phase*, *min_dspacing=0.5*)

Create a CrystalPlane object populated by unique Miller indices with a direct space interplanar spacing greater than a lower threshold.

Parameters

- **phase** (*orix.crystal_map.phase_list.Phase*) – A phase container with a crystal structure and a space and point group describing the allowed symmetry operations.
- **min_dspacing** (*float, optional*) – Smallest interplanar spacing to consider. Default is 0.5 Å.

property gspacing

Return np.ndarray of reciprocal lattice point spacings.

property h

Return np.ndarray of Miller index h.

property hkl

Return Vector3d of Miller indices.

property k

Return np.ndarray of Miller index k.

property l

Return np.ndarray of Miller index l.

property multiplicity

Return either *int* or np.ndarray of *int*.

property scattering_parameter

Return np.ndarray of scattering parameters s.

property shape

Return *tuple*.

property size

Return *int*.

property structure_factor

Return np.ndarray of structure factors F or None.

symmetrise(*antipodal=True*, *unique=True*, *return_multiplicity=False*)

Return planes with symmetrically equivalent Miller indices.

Parameters

- **antipodal** (*bool, optional*) – Whether to include antipodal symmetry operations. Default is True.
- **unique** (*bool, optional*) – Whether to return only distinct indices. Default is True. If True, zero-entries, which are assumed to be degenerate, are removed.
- **return_multiplicity** (*bool, optional*) – Whether to return the multiplicity of indices. This option is only available if *unique* is True. Default is False.

Returns

- *ReciprocalLatticePoint* – Planes with Miller indices symmetrically equivalent to the original planes.
- **multiplicity** (*np.ndarray*) – Multiplicity of the original Miller indices. Only returned if *return_multiplicity* is True.

Notes

Should be the same as EMsoft's CalcFamily in their symmetry.f90 module, although not entirely sure. Use with care.

property theta

Return np.ndarray of twice the Bragg angle.

unique(*use_symmetry=True*)

Return planes with unique Miller indices.

Parameters
use_symmetry (*bool, optional*) – Whether to use symmetry to remove the planes with indices symmetrically equivalent to another set of indices.

Return type
ReciprocalLatticePoint

3.1.3 Functions

diffsims.crystallography.**get_equivalent_hkl**(*hkl*, *operations*, *unique=False*, *return_multiplicity=False*)

Return symmetrically equivalent Miller indices.

Parameters

- **hkl** (*orix.vector.Vector3d, np.ndarray, list or tuple of int*) – Miller indices.
- **operations** (*orix.quaternion.symmetry.Symmetry*) – Point group describing allowed symmetry operations.
- **unique** (*bool, optional*) – Whether to return only unique Miller indices. Default is False.
- **return_multiplicity** (*bool, optional*) – Whether to return the multiplicity of the input indices. Default is False.

Returns

- **new_hkl** (*orix.vector.Vector3d*) – The symmetrically equivalent Miller indices.
- **multiplicity** (*np.ndarray*) – Number of symmetrically equivalent indices. Only returned if *return_multiplicity* is True.

diffsims.crystallography.**get_highest_hkl**(*lattice*, *min_dspacing=0.5*)

Return the highest Miller indices hkl of the plane with a direct space interplanar spacing greater than but closest to a lower threshold.

Parameters

- **lattice** (*diffpy.structure.Lattice*) – Crystal lattice.
- **min_dspacing** (*float, optional*) – Smallest interplanar spacing to consider. Default is 0.5 Å.

Returns
highest_hkl – Highest Miller indices.

Return type
np.ndarray

diffsims.crystallography.**get_hkl**(*highest_hkl*)

Return a list of planes from a set of highest Miller indices.

Parameters
highest_hkl (*orix.vector.Vector3d, np.ndarray, list, or tuple of int*) – Highest Miller indices to consider.

Returns
hkl – An array of Miller indices.

Return type
np.ndarray

3.2 generators

Generation of diffraction simulations and libraries, and lists of rotations.

3.2.1 diffraction_generator

Electron diffraction pattern simulation.

class diffsims.generators.diffraction_generator.**AtomicDiffractionGenerator**(*accelerating_voltage*, *detector*, *reciprocal_mesh=False*)

Bases: object

Computes electron diffraction patterns for an atomic lattice.

Parameters

- **accelerating_voltage** (*float, 'inf'*) – The accelerating voltage of the microscope in kV
- **detector** (*list of 1D float-type arrays*) – List of mesh vectors defining the (flat) detector size and sensor positions
- **reciprocal_mesh** (*bool, optional*) – If True then *detector* is assumed to be a reciprocal grid, else (default) it is assumed to be a real grid.

calculate_ed_data(*structure*, *probe*, *slice_thickness*, *probe_centre=None*, *z_range=200*, *precessed=False*, *dtype='float64'*, *ZERO=1e-14*, *mode='kinematic'*, ***kwargs*)

Calculates single electron diffraction image for particular atomic structure and probe.

Parameters

- **structure** (*Structure*) – The structure for upon which to perform the calculation
- **probe** (*instance of probeFunction*) – Function representing 3D shape of beam
- **slice_thickness** (*float*) – Discretisation thickness in the z-axis
- **probe_centre** (*ndarray (or iterable), shape [3] or [2]*) – Translation vector for the probe. Either of the same dimension of the space or the dimension of the detector. default=None focusses the probe at [0,0,0]
- **zrange** (*float*) – z-thickness to discretise. Only required if sample is not thick enough to fully resolve the Ewald-sphere. Default value is 200.

- **precessed** (*bool, float, or (float, int)*) – Dictates whether beam precession is simulated. If False or the float is 0 then no precession is computed. If <precessed> = (alpha, n) then the precession arc of tilt alpha (in degrees) is discretised into n projections. If n is not provided then default of 30 is used.
- **dtype** (*str or numpy.dtype*) – Defines the precision to use whilst computing diffraction image.
- **ZERO** (*float > 0*) – Rounding error permitted in computation of atomic density. This value is the smallest value rounded to 0. Default is 1e-14.
- **mode** (*str*) – Only <mode>='kinematic' is currently supported.
- **kwargs** (*dictionary*) – Extra key-word arguments to pass to child simulator. For kinematic: **GPU** (bool): Flag to use GPU if available, default is True. **pointwise** (bool): Flag to evaluate charge pointwise on voxels rather than average, default is False.

Returns
Diffraction data to be interpreted as a discretisation on the original detector mesh.

Return type
ndarray

class diffsims.generators.diffraction_generator.**DiffractionGenerator**(*accelerating_voltage*, *scattering_params='lobato'*, *precession_angle=0*, *shape_factor_model='lorentzian'*, *approximate_precession=True*, *minimum_intensity=1e-20*, ***kwargs*)

Bases: object

Computes electron diffraction patterns for a crystal structure.

1. Calculate reciprocal lattice of structure. Find all reciprocal points within the limiting sphere given by $frac2lambda$.
2. For each reciprocal point $mathbf{g}_{hkl}$ corresponding to lattice plane (hkl), compute the Bragg condition $sin(theta) = fraclambda2d_{hkl}$
3. The intensity of each reflection is then given in the kinematic approximation as the modulus square of the structure factor. $I_{hkl} = F_{hkl}F_{hkl}^{*}$

Parameters

- **accelerating_voltage** (*float*) – The accelerating voltage of the microscope in kV.
- **scattering_params** (*str*) – "lobato", "xtables" or None. If None is provided then atomic scattering is not taken into consideration.
- **precession_angle** (*float*) – Angle about which the beam is precessed in degrees. Default is no precession.
- **shape_factor_model** (*func or str*) – A function that takes excitation_error and *max_excitation_error* (and potentially kwargs) and returns an intensity scaling factor. If

None defaults to *shape_factor_models.linear*. A number of pre-programmed functions are available via strings.

- **approximate_precession** (*bool*) – When using precession, whether to precisely calculate average excitation errors and intensities or use an approximation.
- **minimum_intensity** (*float*) – Minimum intensity for a peak to be considered visible in the pattern (fractional from the maximum).
- **kwargs** – Keyword arguments passed to *shape_factor_model*.

Notes

When using precession and approximate_precession=True, the shape factor model defaults to Lorentzian; shape_factor_model is ignored. Only with *approximate_precession=False* the *custom shape_factor_model* is used.

calculate_ed_data(*structure*, *reciprocal_radius*, *rotation=(0, 0, 0)*, *with_direct_beam=True*, *max_excitation_error=0.01*, *shape_factor_width=None*, *debye_waller_factors={}*)

Calculates the Electron Diffraction data for a structure.

Parameters

- **structure** (*diffpy.structure.structure.Structure*) – The structure for which to derive the diffraction pattern. Note that the structure must be rotated to the appropriate orientation and that testing is conducted on unit cells (rather than supercells).
- **reciprocal_radius** (*float*) – The maximum radius of the sphere of reciprocal space to sample, in reciprocal Angstroms.
- **rotation** (*tuple*) – Euler angles, in degrees, in the rzxz convention. Default is (0, 0, 0) which aligns 'z' with the electron beam.
- **with_direct_beam** (*bool*) – If True, the direct beam is included in the simulated diffraction pattern. If False, it is not.
- **max_excitation_error** (*float*) – The cut-off for geometric excitation error in the z-direction in units of reciprocal Angstroms. Spots with a larger distance from the Ewald sphere are removed from the pattern. Related to the extinction distance and roungly equal to 1/thickness.
- **shape_factor_width** (*float*) – Determines the width of the reciprocal rel-rod, for fine-grained control. If not set will be set equal to max_excitation_error.
- **debye_waller_factors** (*dict of str:value pairs*) – Maps element names to their temperature-dependent Debye-Waller factors.

Returns

The data associated with this structure and diffraction setup.

Return type

diffsims.sims.diffraction_simulation.DiffractionSimulation

calculate_profile_data(*structure*, *reciprocal_radius=1.0*, *minimum_intensity=0.001*, *debye_waller_factors={}*)

Calculates a one dimensional diffraction profile for a structure.

Parameters

- **structure** (*diffpy.structure.structure.Structure*) – The structure for which to calculate the diffraction profile.

- **reciprocal_radius** (*float*) – The maximum radius of the sphere of reciprocal space to sample, in reciprocal angstroms.
- **minimum_intensity** (*float*) – The minimum intensity required for a diffraction peak to be considered real. Deals with numerical precision issues.
- **debye_waller_factors** (*dict of str:value pairs*) – Maps element names to their temperature-dependent Debye-Waller factors.

Returns
The diffraction profile corresponding to this structure and experimental conditions.

Return type
diffsims.sims.diffraction_simulation.ProfileSimulation

3.2.2 library_generator

Diffraction pattern library generator and associated tools.

class diffsims.generators.library_generator.**DiffractionLibraryGenerator**(*electron_diffraction_calculator*)

Bases: object

Computes a library of electron diffraction patterns for specified atomic structures and orientations.

get_diffraction_library(*structure_library*, *calibration*, *reciprocal_radius*, *half_shape*, *with_direct_beam=True*, *max_excitation_error=0.01*, *shape_factor_width=None*, *debye_waller_factors={}*)

Calculates a dictionary of diffraction data for a library of crystal structures and orientations.

Each structure in the structure library is rotated to each associated orientation and the diffraction pattern is calculated each time.

Angles must be in the Euler representation (Z,X,Z) and in degrees

Parameters

- **structure_library** (*difffsims:StructureLibrary Object*) – Dictionary of structures and associated orientations for which electron diffraction is to be simulated.
- **calibration** (*float*) – The calibration of experimental data to be correlated with the library, in reciprocal Angstroms per pixel.
- **reciprocal_radius** (*float*) – The maximum g-vector magnitude to be included in the simulations.
- **half_shape** (*tuple*) – The half shape of the target patterns, for 144x144 use (72,72) etc
- **with_direct_beam** (*bool*) – Include the direct beam in the library.
- **max_excitation_error** (*float*) – The extinction distance for reflections, in reciprocal Angstroms.
- **shape_factor_width** (*float*) – Determines the width of the shape functions of the reflections in Angstroms. If not set is equal to max_excitation_error.
- **debye_waller_factors** (*dict of str:value pairs*) – Maps element names to their temperature-dependent Debye-Waller factors.

Returns
diffraction_library – Mapping of crystal structure and orientation to diffraction data objects.

Return type
DiffractionLibrary

class diffsims.generators.library_generator.**VectorLibraryGenerator**(*structure_library*)

Bases: object

Computes a library of diffraction vectors and pairwise inter-vector angles for a specified StructureLibrary.

get_vector_library(*reciprocal_radius*)

Calculates a library of diffraction vectors and pairwise inter-vector angles for a library of crystal structures.

Parameters
reciprocal_radius (*float*) – The maximum g-vector magnitude to be included in the library.

Returns
vector_library – Mapping of phase identifier to phase information in dictionary format.

Return type
DiffractionVectorLibrary

3.2.3 rotation_list_generators

Provides users with a range of gridding functions

diffsims.generators.rotation_list_generators.**get_beam_directions_grid**(*crystal_system*, *resolution*, *mesh='spherified_cube_edge'*)

Produces an array of beam directions, within the stereographic triangle of the relevant crystal system. The way the array is constructed is based on different methods of meshing the sphere [Cajaravelli2015] and can be specified through the *mesh* argument.

Parameters

- **crystal_system** (*str*) – Allowed are: 'cubic','hexagonal','trigonal','tetragonal', 'orthorhombic','monoclinic','triclinic'
- **resolution** (*float*) – An angle in degrees representing the worst-case angular distance to a first nearest neighbor grid point.
- **mesh** (*str*) – Type of meshing of the sphere that defines how the grid is created. Options are: uv_sphere, normalized_cube, spherified_cube_corner (default), spherified_cube_edge, icosahedral, random.

Returns
rotation_list

Return type
list of tuples

diffsims.generators.rotation_list_generators.**get_fundamental_zone_grid**(*resolution=2*, *point_group=None*, *space_group=None*)

Generates an equispaced grid of rotations within a fundamental zone.

Parameters

- **resolution** (*float, optional*) – The characteristic distance between a rotation and its neighbour (degrees)
- **point_group** (*orix.quaternion.symmetry.Symmetry, optional*) – One of the 11 proper point groups, defaults to None

- **space_group** (*int, optional*) – Between 1 and 231, defaults to None

Returns
rotation_list – Grid of rotations lying within the specified fundamental zone

Return type
list of tuples

diffsims.generators.rotation_list_generators.**get_grid_around_beam_direction**(*beam_rotation*, *resolution*, *angular_range=(0, 360)*)

Creates a rotation list of rotations for which the rotation is about given beam direction.

Parameters

- **beam_rotation** (*tuple*) – A desired beam direction as a rotation (rzxz eulers), usually found via get_rotation_from_z_to_direction.
- **resolution** (*float*) – The resolution of the grid (degrees).
- **angular_range** (*tuple*) – The minimum (included) and maximum (excluded) rotation around the beam direction to be included.

Returns
rotation_list

Return type
list of tuples

Examples

```
>>> from diffsims.generators.zap_map_generator import get_rotation_from_z_to_
↪direction
>>> beam_rotation = get_rotation_from_z_to_direction(structure, [1, 1, 1])
>>> grid = get_grid_around_beam_direction(beam_rotation, 1)
```

diffsims.generators.rotation_list_generators.**get_list_from_orix**(*grid*, *rounding=2*)

Converts an orix sample to a rotation list

Parameters

- **grid** (*orix.quaternion.rotation.Rotation*) – A grid of rotations
- **rounding** (*int, optional*) – The number of decimal places to retain, defaults to 2

Returns
rotation_list – A rotation list

Return type
list of tuples

diffsims.generators.rotation_list_generators.**get_local_grid**(*resolution=2*, *center=None*, *grid_width=10*)

Generates a grid of rotations about a given rotation

Parameters

- **resolution** (*float, optional*) – The characteristic distance between a rotation and its neighbour (degrees)

- **center** (*euler angle tuple or orix.quaternion.rotation.Rotation, optional*) – The rotation at which the grid is centered. If None (default) uses the identity
- **grid_width** (*float, optional*) – The largest angle of rotation away from center that is acceptable (degrees)

Returns
rotation_list

Return type
list of tuples

3.2.4 sphere_mesh_generators

diffsims.generators.sphere_mesh_generators.**beam_directions_grid_to_euler**(*vectors*)

Convert list of vectors representing zones to a list of Euler angles in the bunge convention with the constraint that phi1=0.

Parameters
vectors (*numpy.ndarray (N, 3)*) – N 3-dimensional vectors to convert to Euler angles

Returns
grid – Euler angles in bunge convention corresponding to each vector in degrees.

Return type
numpy.ndarray (N, 3)

Notes

The Euler angles represent the orientation of the crystal if that particular vector were parallel to the beam direction [001]. The additional constraint of phi1=0 means that this orientation is uniquely defined for most vectors. phi1 represents the rotation of the crystal around the beam direction and can be interpreted as the rotation of a particular diffraction pattern.

diffsims.generators.sphere_mesh_generators.**get_cube_mesh_vertices**(*resolution*, *grid_type='spherified_corner'*)

Return the (x, y, z) coordinates of the vertices of a cube mesh on a sphere. To generate the mesh, a cube is made to surround the sphere. The surfaces of the cube are subdivided into a grid. The vectors from the origin to these grid points are normalized to unit length. The grid on the cube can be generated in three ways, see *grid_type* and reference [Cajaravelli2015].

Parameters

- **resolution** (*float*) – The maximum angle in degrees between first nearest neighbor grid points.
- **grid_type** (*str*) – The type of cube grid, can be either *normalized* or *spherified_edge* or *spherified_corner* (default). For details see notes.

Returns
points_in_cartesian – Rows are x, y, z where z is the 001 pole direction

Return type
numpy.ndarray (N,3)

Notes

The resolution determines the maximum angle between first nearest neighbor grid points, but to get an integer number of points between the cube face center and the edges, the number of grid points is rounded up. In practice this means that resolution is always an upper limit. Additionally, where on the grid this maximum angle will be will depend on the type of grid chosen. Resolution says something about the maximum angle but nothing about the distribution of nearest neighbor angles or the minimum angle - also this is fixed by the chosen grid.

In the normalized grid, the grid on the surface of the cube is linear. The maximum angle between nearest neighbors is found between the <001> directions and the first grid point towards the <011> directions. Points approaching the edges and corners of the cube will have a smaller angular deviation, so orientation space will be oversampled there compared to the cube faces <001>.

In the spherified_edge grid, the grid is constructed so that there are still two sets of perpendicular grid lines parallel to the {100} directions on each cube face, but the spacing of the grid lines is chosen so that the angles between the grid points on the line connecting the face centers (<001>) to the edges (<011>) are equal. The maximum angle is also between the <001> directions and the first grid point towards the <011> edges. This grid slightly oversamples the directions between <011> and <111>

The spherified_corner case is similar to the spherified_edge case, but the spacing of the grid lines is chosen so that the angles between the grid points on the line connecting the face centers to the cube corners (<111>) is equal. The maximum angle in this grid is from the corners to the first grid point towards the cube face centers.

References

diffsims.generators.sphere_mesh_generators.**get_icosahedral_mesh_vertices**(*resolution*)

Return the (x, y, z) coordinates of the vertices of an icosahedral mesh of a cube, see [Cajaravelli2015]. Method was adapted from meshzoo [Meshzoo].

Parameters

resolution (*float*) – The maximum angle in degrees between neighboring grid points. Since the mesh is generated iteratively, the actual maximum angle in the mesh can be slightly smaller.

Returns

points_in_cartesian – Rows are x, y, z where z is the 001 pole direction

Return type

numpy.ndarray (N,3)

References

diffsims.generators.sphere_mesh_generators.**get_random_sphere_vertices**(*resolution*, *seed=None*)

Create a mesh that randomly samples the surface of a sphere

Parameters

- **resolution** (*float*) – The expected mean angle between nearest neighbor grid points in degrees.
- **seed** (*int, optional*) – passed to np.random.default_rng(), defaults to None which will give a "new" random result each time

Returns

points_in_cartesian – Rows are x, y, z where z is the 001 pole direction

Return type

numpy.ndarray (N,3)

References

https://mathworld.wolfram.com/SpherePointPicking.html

diffsims.generators.sphere_mesh_generators.**get_uv_sphere_mesh_vertices**(*resolution*)

Return the vertices of a UV (spherical coordinate) mesh on a unit sphere [Cajaravelli2015]. The mesh vertices are defined by the parametrization:

$$x = sin(u)cos(v)$$
$$y = sin(u)sin(v)$$
$$z = cos(u)$$

Parameters

resolution (*float*) – An angle in degrees. The maximum angle between nearest neighbor grid points. In this mesh this occurs on the equator of the sphere. All elevation grid lines are separated by at most resolution. The step size of u and v are rounded up to get an integer number of elevation and azimuthal grid lines with equal spacing.

Returns

points_in_cartesian – Rows are x, y, z where z is the 001 pole direction

Return type

numpy.ndarray (N,3)

3.2.5 zap_map_generator

diffsims.generators.zap_map_generator.**corners_to_centroid_and_edge_centers**(*corners*)

Produces the midpoints and center of a trio of corners

Parameters

corners (*list of lists*) – Three corners of a streographic triangle

Returns

list_of_corners – Length 7, elements ca, cb, cc, mean, cab, cbc, cac where naming is such that ca is the first corner of the input, and cab is the midpoint between corner a and corner b.

Return type

list

diffsims.generators.zap_map_generator.**generate_directional_simulations**(*structure*, *simulator*, *direction_list*, *reciprocal_radius=1*, ***kwargs*)

Produces simulation of a structure aligned with certain axes

Parameters

- **structure** (*diffpy.structure.structure.Structure*) – The structure from which simulations need to be produced.
- **simulator** (DiffractionGenerator) – The diffraction generator object used to produce the simulations
- **direction_list** (*list of lists*) – A list of [UVW] indices, eg. [[1,0,0],[1,1,0]]
- **reciprocal_radius** (*float*) – Default to 1

Returns

direction_dictionary – Keys are zone axes, values are simulations

Return type
dict

diffsims.generators.zap_map_generator.**generate_zap_map**(*structure*, *simulator*, *system='cubic'*, *reciprocal_radius=1*, *density='7'*, ***kwargs*)

Produces a number of zone axis patterns for a structure

Parameters

- **structure** (*diffpy.structure.structure.Structure*) – The structure to be simulated.
- **simulator** (DiffractionGenerator) – The simulator used to generate the simulations
- **system** (*str*) – 'cubic', 'hexagonal', 'trigonal', 'tetragonal', 'orthorhombic', 'monoclinic'. Defaults to 'cubic'.
- **reciprocal_radius** (*float*) – The range of reciprocal lattice spots to be included. Default to 1.
- **density** (*str*) – '3' for the corners or '7' (corners + midpoints + centroids). Defaults to 7.
- **kwargs** – Keyword arguments to be passed to simulator.calculate_ed_data().

Returns
zap_dictionary – Keys are zone axes, values are simulations

Return type
dict

Examples

Plot all of the patterns that you have generated

```
>>> zap_map = generate_zap_map(structure,simulator,'hexagonal',density='3')
>>> for k in zap_map.keys():
>>>     pattern = zap_map[k]
>>>     pattern.calibration = 4e-3
>>>     plt.figure()
>>>     plt.imshow(pattern.get_diffraction_pattern(),vmax=0.02)
```

diffsims.generators.zap_map_generator.**get_rotation_from_z_to_direction**(*structure*, *direction*)

Finds the rotation that takes [001] to a given zone axis.

Parameters

- **structure** (*diffpy.structure.structure.Structure*) – The structure for which a rotation needs to be found.
- **direction** (*array like*) – [UVW] direction that the 'z' axis should end up point down.

Returns
euler_angles – 'rzxz' in degrees.

Return type
tuple

See also:

generate_zap_map, *get_grid_around_beam_direction()*

Notes

This implementation works with an axis arrangement that has +x as left to right, +y as bottom to top and +z as out of the plane of a page. Rotations are counter clockwise as you look from the tip of the axis towards the origin

---

3.3 libraries

Diffraction, structure and vector libraries.

3.3.1 diffraction_library

class diffsims.libraries.diffraction_library.**DiffractionLibrary**(**args*, ***kwargs*)

Bases: dict

Maps crystal structure (phase) and orientation to simulated diffraction data.

identifiers

A list of phase identifiers referring to different atomic structures.

Type
list of strings/ints

structures

A list of diffpy.structure.Structure objects describing the atomic structure associated with each phase in the library.

Type
list of diffpy.structure.Structure objects.

diffraction_generator

Diffraction generator used to generate this library.

Type
DiffractionGenerator

reciprocal_radius

Maximum g-vector magnitude for peaks in the library.

Type
float

with_direct_beam

Whether the direct beam included in the library or not.

Type
bool

get_library_entry(*phase=None*, *angle=None*)

Extracts a single DiffractionLibrary entry.

Parameters

- **phase** (*str*) – Key for the phase of interest. If unspecified the choice is random.
- **angle** (*tuple*) – The orientation of interest as a tuple of Euler angles following the Bunge convention [z, x, z] in degrees. If unspecified the choise is random (the first hit).

Returns
library_entries – Dictionary containing the simulation associated with the specified phase and orientation with associated properties.

Return type
dict

pickle_library(*filename*)

Saves a diffraction library in the pickle format.

Parameters
filename (*str*) – The location in which to save the file

See also:

load_DiffractionLibrary

diffsims.libraries.diffraction_library.**load_DiffractionLibrary**(*filename*, *safety=False*)

Loads a previously saved diffraction library.

Parameters

- **filename** (*str*) – The location of the file to be loaded.
- **safety** (*bool*) – Unpickling is risky, this variable requires you to acknowledge this. Default is False.

Returns
Previously saved Library.

Return type
DiffractionLibrary

See also:

DiffractionLibrary.pickle_library

3.3.2 structure_library

class diffsims.libraries.structure_library.**StructureLibrary**(*identifiers*, *structures*, *orientations*)

Bases: object

A dictionary containing all the structures and their associated rotations in the .struct_lib attribute.

identifiers

A list of phase identifiers referring to different atomic structures.

Type
list of strings/ints

structures

A list of diffpy.structure.Structure objects describing the atomic structure associated with each phase in the library.

Type
list of diffpy.structure.Structure objects.

orientations

A list over identifiers of lists of euler angles (as tuples) in the rzxz convention and in degrees.

Type
list

classmethod from_crystal_systems(*identifiers*, *structures*, *systems*, *resolution*, *equal='angle'*)

Creates a structure library from crystal system derived orientation lists

Parameters

- **identifiers** (*list of strings/ints*) – A list of phase identifiers referring to different atomic structures.
- **structures** (*list of diffpy.structure.Structure objects.*) – A list of diffpy.structure.Structure objects describing the atomic structure associated with each phase in the library.
- **systems** (*list*) – A list over indentifiers of crystal systems
- **resolution** (*float*) – resolution in degrees
- **equal** (*str*) – Default is 'angle'

Raises

NotImplementedError: – "This function has been removed in version 0.3.0, in favour of creation from orientation lists"

classmethod from_orientation_lists(*identifiers*, *structures*, *orientations*)

Creates a structure library from "manual" orientation lists

Parameters

- **identifiers** (*list of strings/ints*) – A list of phase identifiers referring to different atomic structures.
- **structures** (*list of diffpy.structure.Structure objects.*) – A list of diffpy.structure.Structure objects describing the atomic structure associated with each phase in the library.
- **orientations** (*list of lists of tuples*) – A list over identifiers of lists of euler angles (as tuples) in the rzxz convention and in degrees.

Return type

StructureLibrary

get_library_size(*to_print=False*)

Returns the the total number of orientations in the current StructureLibrary object. Will also print the number of orientations for each identifier in the library if the to_print==True

Parameters

to_print (*bool*) – Default is 'False'

Returns

size_library – Total number of entries in the current StructureLibrary object.

Return type

int

3.3.3 vector_library

class diffsims.libraries.vector_library.**DiffractionVectorLibrary**(**args*, ***kwargs*)

Bases: dict

Maps crystal structure (phase) to diffraction vectors.

The library is a dictionary mapping from a phase name to phase information. The phase information is stored as a dictionary with the following entries:

'indices'
[np.array] List of peak indices [hkl1, hkl2] as a 2D array.

'measurements'
[np.array] List of vector measurements [len1, len2, angle] in the same order as the indices. Lengths in reciprocal Angstrom and angles in radians.

identifiers
A list of phase identifiers referring to different atomic structures.

Type
list of strings/ints

structures
A list of diffpy.structure.Structure objects describing the atomic structure associated with each phase in the library.

Type
list of diffpy.structure.Structure objects.

reciprocal_radius
Maximum reciprocal radius used when generating the library.

Type
float

pickle_library(*filename*)
Saves a vector library in the pickle format.

Parameters
filename (*str*) – The location in which to save the file

diffsims.libraries.vector_library.**load_VectorLibrary**(*filename*, *safety=False*)

Loads a previously saved vectorlibrary.

Parameters

- **filename** (*str*) – The location of the file to be loaded
- **safety** (*bool (defaults to False)*) – Unpickling is risky, this variable requires you to acknowledge this.

Returns
Previously saved Library

Return type
VectorLibrary

See also:

VectorLibrary.pickle_library

3.4 sims

Diffraction simulations.

3.4.1 diffraction_simulation

class diffsims.sims.diffraction_simulation.**DiffractionSimulation**(*coordinates*, *indices=None*, *intensities=None*, *calibration=None*, *offset=(0.0, 0.0)*, *with_direct_beam=False*)

Bases: object

Holds the result of a kinematic diffraction pattern simulation.

Parameters

- **coordinates** (*array-like, shape [n_points, 2]*) – The x-y coordinates of points in reciprocal space.
- **indices** (*array-like, shape [n_points, 3]*) – The indices of the reciprocal lattice points that intersect the Ewald sphere.
- **intensities** (*array-like, shape [n_points,]*) – The intensity of the reciprocal lattice points.
- **calibration** (*float or tuple of float, optional*) – The x- and y-scales of the pattern, with respect to the original reciprocal angstrom coordinates.
- **offset** (*tuple of float, optional*) – The x-y offset of the pattern in reciprocal angstroms. Defaults to zero in each direction.

property calibrated_coordinates

Coordinates converted into pixel space.

Type
ndarray

property calibration

The x- and y-scales of the pattern, with respect to the original reciprocal angstrom coordinates.

Type
tuple of float

property coordinates

The coordinates of all unmasked points.

Type
ndarray

deepcopy()

property direct_beam_mask

If *with_direct_beam* is True, returns a True array for all points. If *with_direct_beam* is False, returns a True array with False in the position of the direct beam.

Type
ndarray

extend(*other*)

Add the diffraction spots from another DiffractionSimulation

get_as_mask(*shape*, *radius=6.0*, *negative=True*, *radius_function=None*, *direct_beam_position=None*, *in_plane_angle=0*, *mirrored=False*, **args*, ***kwargs*)

Return the diffraction pattern as a binary mask of type bool

Parameters

- **shape** (*2-tuple of ints*) – Shape of the output mask (width, height)
- **radius** (*float or array, optional*) – Radii of the spots in pixels. An array may be supplied of the same length as the number of spots.
- **negative** (*bool, optional*) – Whether the spots are masked (True) or everything else is masked (False)
- **radius_function** (*Callable, optional*) – Calculate the radius as a function of the spot intensity, for example np.sqrt. args and kwargs supplied to this method are passed to this function. Will override radius.
- **direct_beam_position** (*2-tuple of ints, optional*) – The (x,y) coordinate in pixels of the direct beam. Defaults to the center of the image.
- **in_plane_angle** (*float, optional*) – In plane rotation of the pattern in degrees
- **mirrored** (*bool, optional*) – Whether the pattern should be flipped over the x-axis, corresponding to the inverted orientation

Returns

mask – Boolean mask of the diffraction pattern

Return type

numpy.ndarray

get_diffraction_pattern(*shape=(512, 512)*, *sigma=10*, *direct_beam_position=None*, *in_plane_angle=0*, *mirrored=False*)

Returns the diffraction data as a numpy array with two-dimensional Gaussians representing each diffracted peak. Should only be used for qualitative work.

Parameters

- **shape** (*tuple of ints*) – The size of a side length (in pixels)
- **sigma** (*float*) – Standard deviation of the Gaussian function to be plotted (in pixels).
- **direct_beam_position** (*2-tuple of ints, optional*) – The (x,y) coordinate in pixels of the direct beam. Defaults to the center of the image.
- **in_plane_angle** (*float, optional*) – In plane rotation of the pattern in degrees
- **mirrored** (*bool, optional*) – Whether the pattern should be flipped over the x-axis, corresponding to the inverted orientation

Returns

diffraction-pattern – The simulated electron diffraction pattern, normalised.

Return type

numpy.array

Notes

If don't know the exact calibration of your diffraction signal using 1e-2 produces reasonably good patterns when the lattice parameters are on the order of 0.5nm and a the default size and sigma are used.

property indices

property intensities

The intensities of all unmasked points.

Type
ndarray

plot(*size_factor=1*, *direct_beam_position=None*, *in_plane_angle=0*, *mirrored=False*, *units='real'*, *show_labels=False*, *label_offset=(0, 0)*, *label_formatting={}*, *ax=None*, ***kwargs*)

A quick-plot function for a simulation of spots

Parameters

- **size_factor** (*float, optional*) – linear spot size scaling, default to 1
- **direct_beam_position** (*2-tuple of ints, optional*) – The (x,y) coordinate in pixels of the direct beam. Defaults to the center of the image.
- **in_plane_angle** (*float, optional*) – In plane rotation of the pattern in degrees
- **mirrored** (*bool, optional*) – Whether the pattern should be flipped over the x-axis, corresponding to the inverted orientation
- **units** (*str, optional*) – 'real' or 'pixel', only changes scalebars, falls back on 'real', the default
- **show_labels** (*bool, optional*) – draw the miller indices near the spots
- **label_offset** (*2-tuple, optional*) – the relative location of the spot labels. Does nothing if *show_labels* is False.
- **label_formatting** (*dict, optional*) – keyword arguments passed to *ax.text* for drawing the labels. Does nothing if *show_labels* is False.
- **ax** (*matplotlib Axes, optional*) – axes on which to draw the pattern. If *None*, a new axis is created
- ****kwargs** – passed to ax.scatter() method

Return type
ax,sp

Notes

spot size scales with the square root of the intensity.

rotate_shift_coordinates(*angle*, *center=(0, 0)*, *mirrored=False*)

Rotate, flip or shift patterns in-plane

Parameters

- **angle** (*float*) – In plane rotation angle in degrees
- **center** (*2-tuple of floats*) – Center coordinate of the patterns
- **mirrored** (*bool*) – Mirror across the x axis

property size

class diffsims.sims.diffraction_simulation.**ProfileSimulation**(*magnitudes*, *intensities*, *hkls*)

Bases: object

Holds the result of a given kinematic simulation of a diffraction profile.

Parameters

- **magnitudes** (*array-like, shape [n_peaks, 1]*) – Magnitudes of scattering vectors.
- **intensities** (*array-like, shape [n_peaks, 1]*) – The kinematic intensity of the diffraction peaks.
- **hkls** (*[{(h, k, l): mult}] {(h, k, l): mult} is a dict of Miller*) – indices for all diffracted lattice facets contributing to each intensity.

get_plot(*annotate_peaks=True*, *with_labels=True*, *fontsize=12*)

Plots the diffraction profile simulation for the
calculate_profile_data method in DiffractionGenerator.

Parameters

- **annotate_peaks** (*boolean*) – If True, peaks are annotaed with hkl information.
- **with_labels** (*boolean*) – If True, xlabels and ylabels are added to the plot.
- **fontsize** (*integer*) – Fontsize for peak labels.

3.5 structure_factor

Calculation of scattering factors and structure factors.

diffsims.structure_factor.**find_asymmetric_positions**(*positions*, *space_group*)

Return the asymmetric atom positions among a set of positions when considering symmetry operations defined by a space group.

Parameters

- **positions** (*list*) – A list of cartesian atom positions.
- **space_group** (*diffpy.structure.spacegroupmod.SpaceGroup*) – Space group describing the symmetry operations.

Returns
Asymmetric atom positions.

Return type
numpy.ndarray

diffsims.structure_factor.**get_atomic_scattering_parameters**(*element*, *unit=None*)

Return the eight atomic scattering parameters a_1-4, b_1-4 for elements with atomic numbers Z = 1-98 from Table 12.1 in [DeGraef2007], which are themselves from [Doyle1968] and [Smith1962].

Parameters

- **element** (*int or str*) – Element to return scattering parameters for. Either one-two letter string or integer atomic number.

- **unit** (*str, optional*) – Either "nm" or "Å"/"A". Whether to return parameters in terms of Å^-2 or nm^-2. If None (default), Å^-2 is used.

Returns

- **a** (*numpy.ndarray*) – The four atomic scattering parameters a_1-4.
- **b** (*numpy.ndarray*) – The four atomic scattering parameters b_1-4.

References

diffsims.structure_factor.**get_doyleturner_atomic_scattering_factor**(*atom*, *scattering_parameter*, *unit_cell_volume*)

Return the atomic scattering factor f for a certain atom and scattering parameter using Doyle-Turner atomic scattering parameters [Doyle1968].

Assumes structure's Debye-Waller factors are expressed in Ångströms.

This function is adapted from EMsoft.

Parameters

- **atom** (*diffpy.structure.atom.Atom*) – Atom with element type, Debye-Waller factor and occupancy number.
- **scattering_parameter** (*float*) – The scattering parameter s for these Miller indices describing the crystal plane in which the atom lies.
- **unit_cell_volume** (*float*) – Volume of the unit cell.

Returns

f – Scattering factor for this atom on this plane.

Return type

float

diffsims.structure_factor.**get_doyleturner_structure_factor**(*phase*, *hkl*, *scattering_parameter*, *voltage*, *return_parameters=False*)

Return the structure factor for a given family of Miller indices using Doyle-Turner atomic scattering parameters [Doyle1968].

Assumes structure's lattice parameters and Debye-Waller factors are expressed in Ångströms.

This function is adapted from EMsoft.

Parameters

- **phase** (*orix.crystal_map.phase_list.Phase*) – A phase container with a crystal structure and a space and point group describing the allowed symmetry operations.
- **hkl** (*numpy.ndarray or list*) – Miller indices.
- **scattering_parameter** (*float*) – Scattering parameter for these Miller indices.
- **voltage** (*float*) – Beam energy in V.
- **return_parameters** (*bool, optional*) – Whether to return a set of parameters derived from the calculation as a dictionary. Default is False.

Returns

- **structure_factor** (*float*) – Structure factor F.

- **params** (*dict*) – A dictionary with (key, item) (str, float) of parameters derived from the calculation. Only returned if *return_parameters=True*.

diffsims.structure_factor.**get_element_id_from_string**(*element_str*)

Get periodic element ID for elements Z = 1-98 from one-two letter string.

Parameters
element_str (*str*) – One-two letter string.

Returns
element_id – Integer ID in the periodic table of elements.

Return type
int

diffsims.structure_factor.**get_kinematical_atomic_scattering_factor**(*atom*, *scattering_parameter*)

Return the kinematical (X-ray) atomic scattering factor f for a certain atom and scattering parameter.

Assumes structure's Debye-Waller factors are expressed in Ångströms.

This function is adapted from EMsoft.

Parameters

- **atom** (*diffpy.structure.atom.Atom*) – Atom with element type, Debye-Waller factor and occupancy number.
- **scattering_parameter** (*float*) – The scattering parameter s for these Miller indices describing the crystal plane in which the atom lies.

Returns
f – Scattering factor for this atom on this plane.

Return type
float

diffsims.structure_factor.**get_kinematical_structure_factor**(*phase*, *hkl*, *scattering_parameter*)

Return the kinematical (X-ray) structure factor for a given family of Miller indices.

Assumes structure's lattice parameters and Debye-Waller factors are expressed in Ångströms.

This function is adapted from EMsoft.

Parameters

- **phase** (*orix.crystal_map.phase_list.Phase*) – A phase container with a crystal structure and a space and point group describing the allowed symmetry operations.
- **hkl** (*numpy.ndarray or list*) – Miller indices.
- **scattering_parameter** (*float*) – Scattering parameter for these Miller indices.

Returns
structure_factor – Structure factor F.

Return type
float

diffsims.structure_factor.**get_refraction_corrected_wavelength**(*phase*, *voltage*)

Return the refraction corrected relativistic electron wavelength in Ångströms for a given crystal structure and beam energy in V.

This function is adapted from EMsoft.

Parameters

- **phase** (*orix.crystal_map.Phase*) – A phase container with a crystal structure and a space and point group describing the allowed symmetry operations.
- **voltage** (*float*) – Beam energy in V.

Returns
wavelength – Refraction corrected relativistic electron wavelength in Ångströms.

Return type
float

3.6 pattern

3.6.1 detector_functions

diffsims.pattern.detector_functions.**add_dead_pixels**(*pattern*, *n=None*, *fraction=None*, *seed=None*)

Adds randomly placed dead pixels onto a pattern

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **n** (*int*) – The number of dead pixels, defaults to None
- **fraction** (*float*) – The fraction of dead pixels, defaults to None
- **seed** (*int or None*) – seed value for the random number generator

Returns
corrupted_pattern – The pattern, with dead pixels included

Return type
numpy.ndarray

diffsims.pattern.detector_functions.**add_detector_offset**(*pattern*, *offset*)

Adds/subtracts a fixed offset value from a pattern

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **offset** (*float or numpy.ndarray*) – Added through the pattern, broadcasting applies

Returns
corrupted_pattern – The pattern, with offset applied, pixels that would have been negative are instead 0.

Return type
np.ndarray

diffsims.pattern.detector_functions.**add_gaussian_noise**(*pattern*, *sigma*, *seed=None*)

Applies gaussian noise at each pixel within the pattern

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **sigma** (*float*) – The (absolute) deviation of the gaussian errors
- **seed** (*int or None*) – seed value for the random number generator

Return type
corrupted_pattern

diffsims.pattern.detector_functions.**add_gaussian_point_spread**(*pattern*, *sigma*)

Blurs intensities across space with a gaussian function

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **sigma** (*float*) – The standard deviation of the gaussian blur, in pixels

Returns
blurred_pattern – The blurred pattern (deterministic)

Return type
numpy.ndarray

diffsims.pattern.detector_functions.**add_linear_detector_gain**(*pattern*, *gain*)

Multiplies the pattern by a gain (which is not a function of the pattern)

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **gain** (*float or numpy.ndarray*) – Multiplied through the pattern, broadcasting applies

Returns
corrupted_pattern – The pattern, with gain applied

Return type
numpy.ndarray

diffsims.pattern.detector_functions.**add_shot_and_point_spread**(*pattern*, *sigma*, *shot_noise=True*, *seed=None*)

Adds shot noise (optional) and gaussian point spread (via a convolution) to a pattern

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **sigma** (*float*) – The standard deviation of the gaussian blur, in pixels
- **shot_noise** (*bool*) – Whether to include shot noise in the original signal, default True
- **seed** (*int or None*) – seed value for the random number generator (effects the shot noise only)

Returns
detector_pattern – A single sample of the pattern after accounting for detector properties

Return type
numpy.ndarray

See also:

add_shot_noise
adds only shot noise

add_gaussian_point_spread
adds only point spread

diffsims.pattern.detector_functions.**add_shot_noise**(*pattern*, *seed=None*)

Applies shot noise to a pattern

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector
- **seed** (*int or None*) – seed value for the random number generator

Returns

shotted_pattern – A single sample of the pattern after accounting for shot noise

Return type

numpy.ndarray

Notes

This function will (as it should) behave differently depending on the pattern intensity, so be mindful to put your intensities in physical units

diffsims.pattern.detector_functions.**constrain_to_dynamic_range**(*pattern*, *detector_max=None*)

Force the values within pattern to lie between [0,detector_max]

Parameters

- **pattern** (*numpy.ndarray*) – The diffraction pattern at the detector after corruption
- **detector_max** (*float*) – The maximum allowed value at the detector

Returns

within_range_pattern – The pattern, with values >=0 and =< detector_max

Return type

numpy.ndarray

3.7 utils

Diffraction utilities used by the other modules.

3.7.1 atomic_diffraction_generator_utils

Back end for computing diffraction patterns with a kinematic model.

diffsims.utils.atomic_diffraction_generator_utils.**get_diffraction_image**(*coordinates*, *species*, *probe*, *x*, *wavelength*, *precession*, *GPU=True*, *pointwise=False*, ***kwargs*)

Return kinematically simulated diffraction pattern

Parameters

- **coordinates** (*numpy.ndarray* [*float*], (n_atoms, 3)) – List of atomic coordinates
- **species** (*numpy.ndarray* [*int*], (n_atoms,)) – List of atomic numbers

- **probe** (*diffsims.ProbeFunction*) – Function representing 3D shape of beam
- **x** (*list* [*numpy.ndarray* [*float*]], of shapes [(nx,), (ny,), (nz,)]) – Mesh on which to compute the volume density
- **wavelength** (*float*) – Wavelength of electron beam
- **precession** (a pair (*float*, *int*)) – The float dictates the angle of precession and the int how many points are used to discretise the integration.
- **dtype** ((*str*, *str*)) – tuple of floating/complex datatypes to cast outputs to
- **ZERO** (*float* > 0, optional) – Rounding error permitted in computation of atomic density. This value is the smallest value rounded to 0.
- **GPU** (*bool*, optional) – Flag whether to use GPU or CPU discretisation. Default (if available) is True
- **pointwise** (*bool*, optional) – Optional parameter whether atomic intensities are computed point-wise at the centre of a voxel or an integral over the voxel. default=False

Returns

DP – The two-dimensional diffraction pattern evaluated on the reciprocal grid corresponding to the first two vectors of *x*.

Return type

numpy.ndarray [*dtype[0]*], (nx, ny, nz)

diffsims.utils.atomic_diffraction_generator_utils.**grid2sphere**(*arr*, *x*, *dx*, *C*)

Projects 3d array onto a sphere

Parameters

- **arr** (np.ndarray [*float*], (nx, ny, nz)) – Input function to be projected
- **x** (*list [np.ndarray [float]], of shapes [(nx,), (ny,), (nz,)]*) – Vectors defining mesh of <arr>
- **dx** (*list [np.ndarray [float]], of shapes [(3,), (3,), (3,)]*) – Basis in which to orient sphere. Centre of sphere will be at *C*dx[2]* and mesh of output array will be defined by the first two vectors
- **C** (*float*) – Radius of sphere

Returns

out – If y is the point on the line between *i*dx[0]+j*dx[1]* and *C*dx[2]* which also lies on the sphere of radius *C* from *C*dx[2]* then: *out[i,j] = arr(y)*. Interpolation on arr is linear.

Return type

np.ndarray [float], (nx, ny)

diffsims.utils.atomic_diffraction_generator_utils.**normalise**(*arr*)

diffsims.utils.atomic_diffraction_generator_utils.**precess_mat**(*alpha*, *theta*)

Generates rotation matrices for precession curves.

Parameters

- **alpha** (*float*) – Angle (in degrees) of precession tilt
- **theta** (*float*) – Angle (in degrees) along precession curve

Returns

R – Rotation matrix associated to the tilt of *alpha* away from the vertical axis and a rotation of *theta* about the vertical axis.

Return type
numpy.ndarray [*float*], (3, 3)

3.7.2 atomic_scattering_params

3.7.3 discretise_utils

Utils for converting lists of atoms to a discretised volume

diffsims.utils.discretise_utils.**do_binning**(*x*, *loc*, *Rmax*, *d*, *GPU*)

Utility function which takes in a mesh, atom locations, atom radius and minimal grid-spacing and returns a binned array of atom indices.

Parameters

- **x** (*list [np.ndarray [float]], of shape [(nx,), (ny,), ...]*) – Dictates the range of the box over which to bin atoms.
- **loc** (*np.ndarray, (n, 3)*) – Atoms to bin.
- **Rmax** (*float > 3*) – Maximum radius of an atom (rounded up to 3).
- **d** (*list of float > 0*) – The finest permitted binning.
- **GPU** (*bool*) – If *True* then constrains to memory of GPU rather than RAM.

Returns

- **subList** (*np.ndarray [int]*) – *subList[i0,i1,i2]* is a list of indices *[j0, j1, …, jn, -1,…]* such that the only atoms which are contained in the box: *[x[0].min()+i0*r,x[0].min(),+(i0+1)*r] x [x[1].min()+i1*r,x[1].min(),+(i1+1)*r]…*
- **r** (*np.ndarray [float]*) – Size of each bin.
- **mem** (*int*) – Upper limit of memory in bytes.

diffsims.utils.discretise_utils.**get_atoms**(*Z*, *returnFunc=True*, *dtype='f8'*)

This function returns an approximation of the atom with atomic number Z using a list of Gaussians.

Parameters

- **Z** (*int*) – Atomic number of atom
- **returnFunc** (*bool, optional*) – If True (default) then returns functions for real/reciprocal space discretisation else returns the vectorial representation of the approximating Gaussians.

Returns
obj1, obj2 – Continuous atom is represented by: .. math:: ymapsto sum_i a[i]*exp(-b[i]*|y|^2)

Return type
numpy.ndarray or function

This is data table 3 from 'Robust Parameterization of Elastic and Absorptive Electron Atomic Scattering Factors' by L.-M. Peng, G. Ren, S. L. Dudarev and M. J. Whelan, 1996

diffsims.utils.discretise_utils.**get_discretisation**(*loc*, *Z*, *x*, *GPU=False*, *ZERO=None*, *dtype=('f8', 'c16')*, *pointwise=False*, *FT=False*, ***kwargs*)

Parameters

- **loc** (*numpy.ndarray*, (n, 3)) – Atoms to bin

- **Z** (*str*, *int*, or *numpy.ndarray* [*str* or *int*], (n,)) – atom labels, either string or atomic masses.
- **x** (*list* [*numpy.ndarray* [*float*]]) – Dictates mesh over which to discretise. Volume will be discretised at points *[x[0][i],x[1][j],...]*
- **GPU** (*bool*, optional) – If *True* (default) then attempts to use the GPU.
- **ZERO** (*float* > 0) – Approximation threshold
- **dtype** ((*str*, *str*), optional) – Real and complex data precisions to use, default=('float64', 'complex128')
- **pointwise** (*bool*, optional) – If *True* (default) then computes pointwise atomic potentials on mesh points, else averages the potential over cube of same size as the discretisation.
- **FT** (*bool*, optional) – If *True* then computes the Fourier transform directly on the reciprocal mesh, otherwise (default) computes the volume potential

Returns
out – Discretisation of atoms defined by *loc*/*Z* on mesh defined by *x*.

Return type
numpy.ndarray, (x[0].size, x[1].size, x[2].size)

diffsims.utils.discretise_utils.**rebin**(*x*, *loc*, *r*, *k*, *mem*)

Bins each location into a grid subject to memory constraints.

Parameters

- **x** (*list [np.ndarray [float]], of shape [(nx,), (ny,), ...]*) – Dictates the range of the box over which to bin atoms.
- **loc** (*np.ndarray, (n, 3)*) – Atoms to bin.
- **r** (*float or [float, float, float]*) – Mesh size (in each direction).
- **k** (*int*) – Integer such that the radius of the atom is <= *k*r*. Consequently, each atom will appear in approximately *8k^3* bins.
- **mem** (*int*) – Upper limit of number of bytes permitted for mesh. If not possible then raises a *MemoryError*.

Returns
subList – *subList[i0,i1,i2]* is a list of indices *[j0, j1, ..., jn, -1,...]* such that the only atoms which are contained in the box: *[x[0].min()+i0*r,x[0].min(),+(i0+1)*r] x [x[1].min()+i1*r,x[1].min(),+(i1+1)*r]...* are the atoms with locations *loc[j0], ..., loc[jn]*.

Return type
np.ndarray [int]

3.7.4 fourier_transform

Created on 31 Oct 2019

Module provides optimised fft and Fourier transform approximation.

@author: Rob Tovey

diffsims.utils.fourier_transform.**convolve**(*arr1*, *arr2*, *dx=None*, *axes=None*)

Performs a centred convolution of input arrays

Parameters

- **arr1** (*numpy.ndarray*) – Arrays to be convolved. If dimensions are not equal then 1s are appended to the lower dimensional array. Otherwise, arrays must be broadcastable.
- **arr2** (*numpy.ndarray*) – Arrays to be convolved. If dimensions are not equal then 1s are appended to the lower dimensional array. Otherwise, arrays must be broadcastable.
- **dx** (float > 0, list of float, or *None* , optional) – Grid spacing of input arrays. Output is scaled by *dx**max(arr1.ndim, arr2.ndim)*. default=`None` applies no scaling
- **axes** (tuple of ints or *None*, optional) – Choice of axes to convolve. default=`None` convolves all axes

diffsims.utils.fourier_transform.**fast_abs**(*x*, *y=None*)

Fast computation of abs of an array

Parameters

- **x** (*numpy.ndarray*) – Input
- **y** (*numpy.ndarray* or *None*, optional) – If *y* is not *None*, used as preallocated output

Returns
y – Array equal to *abs(x)*

Return type
numpy.ndarray

diffsims.utils.fourier_transform.**fast_fft_len**(*n*)

Returns the smallest integer greater than input such that the fft can be computed efficiently at this size

Parameters
n (*int*) – minimum size

Returns
N – smallest integer greater than n which permits efficient ffts.

Return type
int

diffsims.utils.fourier_transform.**fftn**(*a*, *s=None*, *axes=None*, *norm=None*, **_)

diffsims.utils.fourier_transform.**fftshift_phase**(*x*)

Fast implementation of fft_shift: fft(fftshift_phase(x)) = fft_shift(fft(x))

Note two things: - this is an in-place manipulation of the (3D) input array - the input array must have even side lengths. This is softly guaranteed by fast_fft_len but will raise error if not true.

diffsims.utils.fourier_transform.**from_recip**(*y*)

Converts Fourier frequencies to spatial coordinates.

Parameters
y (*list* [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...]) – List (or equivalent) of vectors which define a mesh in the dimension equal to the length of *x*

Returns
x – List of vectors defining a mesh such that for a function, *f*, defined on the mesh given by *y*, ifft(f) is defined on the mesh given by *x*. 0 will be in the middle of *x*.

Return type
list [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...]

diffsims.utils.fourier_transform.**get_DFT**(*X=None*, *Y=None*)

Returns discrete analogues for the Fourier/inverse Fourier transform pair defined from grid *X* to grid *Y* and back again.

Parameters

- **X** (*list* [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...], optional) – Mesh on real space
- **Y** (*list* [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...], optional) – Corresponding mesh on Fourier space
- **other** (*If either X or Y is None then it is inferred from the*) –

Returns

- **DFT** (*function(f, axes=None)*) – If f is a function on X then *DFT(f)* is the Fourier transform of f on Y. *axes* parameter can be used to specify which axes to transform.
- **iDFT** (*function(f, axes=None)*) – If f is a function on Y then *iDFT(f)* is the inverse Fourier transform of f on X. *axes* parameter can be used to specify which axes to transform.

diffsims.utils.fourier_transform.**get_recip_points**(*ndim*, *n=None*, *dX=inf*, *rX=0*, *dY=inf*, *rY=1e-16*)

Returns a minimal pair of real and Fourier grids which satisfy each given requirement.

Parameters

- **ndim** (*int*) – Dimension of domain
- **n** (*int*, list of length *ndim*, or *None* , optional) – Sugested number of pixels (per dimension). default=`None` infers this from other parameters. If enough other constraints are given to define a discretisation then this will be shrunk if possible.
- **dX** (*float* > 0 or list of *float* of length *ndim*, optional) – Maximum grid spacing (per dimension). default=`numpy.inf` infers this from other parameters
- **rX** (*float* > 0 or list of *float* of length *ndim*, optional) – Minimum grid range (per dimension). default=`None` infers this from other parameters. In this case, range is maximal span, i.e. diameter.
- **dY** (*float* > 0 or list of *float* of length *ndim*) – Maximum grid spacing (per dimension) in Fourier domain. default=`None` infers this from other parameters
- **rY** (*float* > 0 or list of *float* of length *ndim*) – Minimum grid range (per dimension) in Fourier domain. default=`None` infers this from other parameters. In this case, range is maximal span, i.e. diameter.

Returns

- **x** (*list* [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...]) – Real mesh of points, centred at 0 with at least *n* pixels, resolution higher than *dX*, and range greater than *rX*.
- **y** (*list* [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...]) – Fourier mesh of points, centred at 0 with at least *n* pixels, resolution higher than *dY*, and range greater than *rY*.

diffsims.utils.fourier_transform.**ifftn**(*a*, *s=None*, *axes=None*, *norm=None*, **_)

diffsims.utils.fourier_transform.**plan_fft**(*A*, *n=None*, *axis=None*, *norm=None*, **_)

Plans an fft for repeated use. Parameters are the same as for *pyfftw*'s *fftn* which are, where possible, the same as the *numpy* equivalents. Note that some functionality is only possible when using the *pyfftw* backend.

Parameters

- **A** (*numpy.ndarray*, of dimension *d*) – Array of same shape to be input for the fft

- **n** (iterable or *None*, *len(n) == d*, optional) – The output shape of fft (default=`None` is same as *A.shape*)
- **axis** (*int*, iterable length *d*, or *None*, optional) – The axis (or axes) to transform (default=`None` is all axes)
- **overwrite** (*bool*, optional) – Whether the input array can be overwritten during computation (default=False)
- **planner** (*{0, 1, 2, 3}, optional*) – Amount of effort put into optimising Fourier transform where 0 is low and 3 is high (default=`1`).
- **threads** (*int*, *None*) – Number of threads to use (default=`None` is all threads)
- **auto_align_input** (*bool*, optional) – If *True* then may re-align input (default=`True`)
- **auto_contiguous** (*bool*, optional) – If *True* then may re-order input (default=`True`)
- **avoid_copy** (*bool*, optional) – If *True* then may over-write initial input (default=`False`)
- **norm** (*{None, 'ortho'}, optional*) – Indicate whether fft is normalised (default=`None`)

Returns

- **plan** (*function*) – Returns the Fourier transform of *B*, *plan() == fftn(B)*
- **B** (*numpy.ndarray*, *A.shape*) – Array which should be modified inplace for fft to be computed. If possible, *B is A*.

Example

A = numpy.zeros((8,16)) plan, B = plan_fft(A)

B[:,:] = numpy.random.rand(8,16) numpy.fft.fftn(B) == plan()

B = numpy.random.rand(8,16) numpy.fft.fftn(B) != plan()

diffsims.utils.fourier_transform.**plan_ifft**(*A*, *n=None*, *axis=None*, *norm=None*, ***_*)

Plans an ifft for repeated use. Parameters are the same as for *pyfftw*'s *ifftn* which are, where possible, the same as the *numpy* equivalents. Note that some functionality is only possible when using the *pyfftw* backend.

Parameters

- **A** (*numpy.ndarray*, of dimension *d*) – Array of same shape to be input for the ifft
- **n** (iterable or *None*, *len(n) == d*, optional) – The output shape of ifft (default=`None` is same as *A.shape*)
- **axis** (*int*, iterable length *d*, or *None*, optional) – The axis (or axes) to transform (default=`None` is all axes)
- **overwrite** (*bool*, optional) – Whether the input array can be overwritten during computation (default=False)
- **planner** (*{0, 1, 2, 3}, optional*) – Amount of effort put into optimising Fourier transform where 0 is low and 3 is high (default=`1`).
- **threads** (*int*, *None*) – Number of threads to use (default=`None` is all threads)
- **auto_align_input** (*bool*, optional) – If *True* then may re-align input (default=`True`)
- **auto_contiguous** (*bool*, optional) – If *True* then may re-order input (default=`True`)
- **avoid_copy** (*bool*, optional) – If *True* then may over-write initial input (default=`False`)

- **norm** (*{None, 'ortho'}, optional*) – Indicate whether ifft is normalised (default=`None`)

Returns

- **plan** (*function*) – Returns the inverse Fourier transform of *B*, *plan() == ifftn(B)*
- **B** (*numpy.ndarray*, *A.shape*) – Array which should be modified inplace for ifft to be computed. If possible, *B is A*.

diffsims.utils.fourier_transform.**to_recip**(*x*)

Converts spatial coordinates to Fourier frequencies.

Parameters

x (*list* [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...]) – List (or equivalent) of vectors which define a mesh in the dimension equal to the length of *x*

Returns

y – List of vectors defining a mesh such that for a function, *f*, defined on the mesh given by *x*, *fft(f)* is defined on the mesh given by *y*

Return type

list [*numpy.ndarray* [*float*]], of shape [(nx,), (ny,), ...]

3.7.5 generic_utils

Created on 31 Oct 2019

Generic tools for all areas of code.

@author: Rob Tovey

class diffsims.utils.generic_utils.**GLOBAL_BOOL**(*val*)

Bases: object

An object which behaves like a bool but can be changed in-place by *set* or by calling as a function.

set(*val*)

diffsims.utils.generic_utils.**get_grid**(*sz*, *tpb=None*)

diffsims.utils.generic_utils.**to_mesh**(*x*, *dx=None*, *dtype=None*)

Generates dense meshes from grid vectors, broadly:

to_mesh(x)[i,j,...] = (x[0][i], x[1][j], ...)

Parameters

- **x** (*list* [*numpy.ndarray*], of shape [(nx,), (ny,), ...]) – List of grid vectors
- **dx** (*list* [*numpy.ndarray*] or *None*, optional) – Basis in which to expand mesh, default is the canonical basis
- **dtype** (*str* or *None*, optional) – String representing the *numpy* type of output, default inherits from *x*

Returns

X – X[i,j,..., k] = x[0][i]*dx[0][k] + x[1][j]*dx[1][k] + ...

Return type

numpy.ndarray [dtype], (x[0].size, x[1].size, ..., len(x))

3.7.6 kinematic_simulation_utils

Created on 1 Nov 2019

Back end for computing diffraction patterns with a kinematic model.

@author: Rob Tovey

diffsims.utils.kinematic_simulation_utils.**get_diffraction_image**(*coordinates*, *species*, *probe*, *x*, *wavelength*, *precession*, *GPU=True*, *pointwise=False*, ***kwargs*)

Return kinematically simulated diffraction pattern

Parameters

- **coordinates** (*numpy.ndarray* [*float*], (n_atoms, 3)) – List of atomic coordinates
- **species** (*numpy.ndarray* [*int*], (n_atoms,)) – List of atomic numbers
- **probe** (*diffsims.ProbeFunction*) – Function representing 3D shape of beam
- **x** (*list* [*numpy.ndarray* [*float*]], of shapes [(nx,), (ny,), (nz,)]) – Mesh on which to compute the volume density
- **wavelength** (*float*) – Wavelength of electron beam
- **precession** (a pair (*float*, *int*)) – The float dictates the angle of precession and the int how many points are used to discretise the integration.
- **dtype** ((*str*, *str*)) – tuple of floating/complex datatypes to cast outputs to
- **ZERO** (*float* > 0, optional) – Rounding error permitted in computation of atomic density. This value is the smallest value rounded to 0.
- **GPU** (*bool*, optional) – Flag whether to use GPU or CPU discretisation. Default (if available) is True
- **pointwise** (*bool*, optional) – Optional parameter whether atomic intensities are computed point-wise at the centre of a voxel or an integral over the voxel. default=False

Returns

DP – The two-dimensional diffraction pattern evaluated on the reciprocal grid corresponding to the first two vectors of *x*.

Return type

numpy.ndarray [*dtype[0]*], (nx, ny, nz)

diffsims.utils.kinematic_simulation_utils.**grid2sphere**(*arr*, *x*, *dx*, *C*)

Projects 3d array onto a sphere.

Parameters

- **arr** (*np.ndarray [float], (nx, ny, nz)*) – Input function to be projected
- **x** (*list [np.ndarray [float]], of shapes [(nx,), (ny,), (nz,)]*) – Vectors defining mesh of <arr>
- **dx** (*list [np.ndarray [float]], of shapes [(3,), (3,), (3,)]*) – Basis in which to orient sphere. Centre of sphere will be at *C*dx[2]* and mesh of output array will be defined by the first two vectors.
- **C** (*float*) – Radius of sphere.

Returns
out – If y is the point on the line between *i*dx[0]+j*dx[1]* and *C*dx[2]* which also lies on the sphere of radius *C* from *C*dx[2]* then: *out[i,j] = arr(y)*. Interpolation on arr is linear.

Return type
np.ndarray [float], (nx, ny)

diffsims.utils.kinematic_simulation_utils.**normalise**(*arr*)

diffsims.utils.kinematic_simulation_utils.**precess_mat**(*alpha*, *theta*)
Generates rotation matrices for precession curves.

Parameters
- **alpha** (*float*) – Angle (in degrees) of precession tilt
- **theta** (*float*) – Angle (in degrees) along precession curve

Returns
R – Rotation matrix associated to the tilt of *alpha* away from the vertical axis and a rotation of *theta* about the vertical axis.

Return type
numpy.ndarray [*float*], (3, 3)

3.7.7 lobato_scattering_params

3.7.8 probe_utils

Created on 5 Nov 2019

@author: Rob Tovey

class diffsims.utils.probe_utils.**ProbeFunction**(*func=None*)
Bases: object

Object representing a probe function.

Parameters
func (*function*) – Function which takes in an array, *r*, of shape *[nx, ny, nz, 3]* and returns an array of shape *[nx, ny, nz]*. *r[...,0]* corresponds to the *x* coordinate, *r[..., 1]* to *y* etc. If not provided (or *None*) then the *__call__* and *FT* methods must be overrided.

__call__(*x*, *out=None*, *scale=None*)
Returns *func(x)*scale*. If *out* is provided then it is used as preallocated storage. If *scale* is not provided then it is assumed to be 1. If *x* is a list of arrays it is converted into a mesh first.

Parameters
- **x** (*numpy.ndarray, (nx, ny, nz, 3) or list of arrays of shape*) – [(nx,), (ny,), (nz,)] Mesh points at which to evaluate the probe density.
- **out** (*numpy.ndarray, (nx, ny, nz), optional*) – If provided then computation is performed inplace.
- **scale** (*numpy.ndarray, (nx, ny, nz), optional*) – If provided then the probe density is scaled by this before being returned.

Returns
out – An array equal to *probe(x)*scale*.

Return type
numpy.ndarray, (nx, ny, nz)

FT(*y*, *out=None*)

Returns the Fourier transform of func on the mesh *y*. Again, if *out* is provided then computation is *inplace*. If *y* is a list of arrays then it is converted into a mesh first. If this function is not overridden then an approximation is made using *func* and the *fft*.

Parameters

- **y** (*numpy.ndarray, (nx, ny, nz, 3) or list of arrays of shape*) – [(nx,), (ny,), (nz,)] Mesh of Fourier coordinates at which to evaluate the probe density.
- **out** (*numpy.ndarray, (nx, ny, nz), optional*) – If provided then computation is performed inplace.

Returns
out – An array equal to *FourierTransform(probe)(y)*.

Return type
numpy.ndarray, (nx, ny, nz)

class diffsims.utils.probe_utils.**BesselProbe**(*r*)

Bases: *ProbeFunction*

Probe function given by a radially scaled Bessel function of the first kind.

Parameters
r (*float*) – Width of probe at the surface of the sample. More specifically, the smallest 0 of the probe.

__call__(*x*, *out=None*, *scale=None*)

If *X = sqrt(x[...,0]**2+x[...,1]**2)/r* then returns *J_1(X)/X*scale*. If *out* is provided then this is computed inplace. If *scale* is not provided then it is assumed to be 1. If *x* is a list of arrays it is converted into a mesh first.

Parameters

- **x** (*numpy.ndarray, (nx, ny, nz, 3) or list of arrays of shape*) – [(nx,), (ny,), (nz,)] Mesh points at which to evaluate the probe density. As a plotting utility, if a lower dimensional mesh is provided then the remaining coordinates are assumed to be 0 and so only the respective 1D/2D slice of the probe is returned.
- **out** (*numpy.ndarray, (nx, ny, nz), optional*) – If provided then computation is performed inplace.
- **scale** (*numpy.ndarray, (nx, ny, nz), optional*) – If provided then the probe density is scaled by this before being returned.

Returns
out – An array equal to *probe(x)*scale*. If *ny=0* or *nz=0* then array is of smaller dimension.

Return type
numpy.ndarray, (nx, ny, nz)

FT(*y*, *out=None*)

If *Y = sqrt(y[...,0]**2 + y[...,1]**2)*r* then returns an indicator function on the disc *Y < 1, y[2]==0*. Again, if *out* is provided then computation is inplace. If *y* is a list of arrays then it is converted into a mesh first.

Parameters

- **y** (*numpy.ndarray, (nx, ny, nz, 3) or list of arrays of shape*) – [(nx,), (ny,), (nz,)] Mesh of Fourier coordinates at which to evaluate the probe density. As a plotting utility, if a lower dimensional mesh is provided then the remaining coordinates are assumed to be 0 and so only the respective 1D/2D slice of the probe is returned.
- **out** (*numpy.ndarray, (nx, ny, nz), optional*) – If provided then computation is performed inplace.

Returns
out – An array equal to *FourierTransform(probe)(y)*. If *ny=0* or *nz=0* then array is of smaller dimension.

Return type
numpy.ndarray, (nx, ny, nz)

3.7.9 scattering_params

Scattering Paramaters as Tabulated in "Advanced Computing in Electron Microscopy - Second Edition (2010) - Earl.J.Kirkland" ISBN 978-1-4419-6532-5 Pages 253-260 Appendix C

This transcription comes from scikit-ued (MIT license) - https://pypi.python.org/pypi/scikit-ued

3.7.10 shape_factor_models

diffsims.utils.shape_factor_models.**atanc**(*excitation_error*, *max_excitation_error*, *minima_number=5*)

Intensity with arctan(s)/s profile that closely follows sin(s)/s but is smooth for s!=0.

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct
- **minima_number** (*int*) – The minima_number'th minima in the corresponding sinx/x lies at max_excitation_error from 0

Returns
intensity

Return type
array-like or float

diffsims.utils.shape_factor_models.**binary**(*excitation_error*, *max_excitation_error*)

Returns a unit intensity for all reflections

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct

Returns
intensities

Return type
array-like or float

diffsims.utils.shape_factor_models.**linear**(*excitation_error*, *max_excitation_error*)

Returns an intensity linearly scaled with by the excitation error

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct

Returns
intensities

Return type
array-like or float

diffsims.utils.shape_factor_models.**lorentzian**(*excitation_error*, *max_excitation_error*)

Lorentzian intensity profile that should approximate the two-beam rocking curve. This is equation (6) in reference [1].

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct

Returns
intensity_factor – Vector representing the rel-rod factor for each reflection

Return type
array-like or float

References

[1] L. Palatinus, P. Brázda, M. Jelínek, J. Hrdá, G. Steciuk, M. Klementová, Specifics of the data processing of precession electron diffraction tomography data and their implementation in the program PETS2.0, Acta Crystallogr. Sect. B Struct. Sci. Cryst. Eng. Mater. 75 (2019) 512–522. doi:10.1107/S2052520619007534.

diffsims.utils.shape_factor_models.**lorentzian_precession**(*excitation_error*, *max_excitation_error*, *r_spot*, *precession_angle*)

Intensity profile factor for a precessed beam assuming a Lorentzian intensity profile for the un-precessed beam. This is equation (10) in reference [1].

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct
- **r_spot** (*array-like or float*) – The distance (reciprocal) from each reflection to the origin
- **precession_angle** (*float*) – The beam precession angle in radians; the angle the beam makes with the optical axis.

Returns
intensity_factor – Vector representing the rel-rod factor for each reflection

Return type
array-like or float

References

[1] L. Palatinus, P. Brázda, M. Jelínek, J. Hrdá, G. Steciuk, M. Klementová, Specifics of the data processing of precession electron diffraction tomography data and their implementation in the program PETS2.0, Acta Crystallogr. Sect. B Struct. Sci. Cryst. Eng. Mater. 75 (2019) 512–522. doi:10.1107/S2052520619007534.

diffsims.utils.shape_factor_models.**sin2c**(*excitation_error*, *max_excitation_error*, *minima_number=5*)

Intensity with sin^2(s)/s^2 profile, after Howie-Whelan rel-rod

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct
- **minima_number** (*int*) – The minima_number'th minima lies at max_excitation_error from 0

Returns
intensity

Return type
array-like or float

diffsims.utils.shape_factor_models.**sinc**(*excitation_error*, *max_excitation_error*, *minima_number=5*)

Returns an intensity with a sinc profile

Parameters

- **excitation_error** (*array-like or float*) – The distance (reciprocal) from a reflection to the Ewald sphere
- **max_excitation_error** (*float*) – The distance at which a reflection becomes extinct
- **minima_number** (*int*) – The minima_number'th minima lies at max_excitation_error from 0

Returns
intensity

Return type
array-like or float

3.7.11 sim_utils

diffsims.utils.sim_utils.**acceleration_voltage_to_relativistic_mass**(*acceleration_voltage*)

Get relativistic mass of electron as function of acceleration voltage.

Parameters
acceleration_voltage (*float*) – In Volt

Returns
mr – Relativistic electron mass

Return type
float

Example

```
>>> import diffsims.utils.sim_utils as sim_utils
>>> mr = sim_utils.acceleration_voltage_to_relativistic_mass(200000) # 200 kV
```

diffsims.utils.sim_utils.**acceleration_voltage_to_velocity**(*acceleration_voltage*)

Get relativistic velocity of electron from acceleration voltage.

Parameters
acceleration_voltage (*float*) – In Volt

Returns
v – In m/s

Return type
float

Example

```
>>> import diffsims.utils.sim_utils as sim_utils
>>> v = sim_utils.acceleration_voltage_to_velocity(200000) # 200 kV
>>> round(v)
208450035
```

diffsims.utils.sim_utils.**acceleration_voltage_to_wavelength**(*acceleration_voltage*)

Get electron wavelength from the acceleration voltage.

Parameters
acceleration_voltage (*float or array-like*) – In Volt

Returns
wavelength – In meters

Return type
float or array-like

diffsims.utils.sim_utils.**beta_to_bst**(*beam_deflection*, *acceleration_voltage*)

Calculate Bs * t values from beam deflection (beta).

Parameters

- **beam_deflection** (*NumPy array*) – In radians
- **acceleration_voltage** (*float*) – In Volts

Returns
bst – In Tesla * meter

Return type
NumPy array

Examples

```
>>> import numpy as np
>>> import diffsims.utils.sim_utils as sim_utils
>>> data = np.random.random((100, 100))  # In radians
>>> acceleration_voltage = 200000  # 200 kV (in Volt)
>>> bst = sim_utils.beta_to_bst(data, 200000)
```

diffsims.utils.sim_utils.**bst_to_beta**(*bst*, *acceleration_voltage*)

Calculate beam deflection (beta) values from Bs * t.

Parameters

- **bst** (*NumPy array*) – Saturation induction Bs times thickness t of the sample. In Tesla*meter
- **acceleration_voltage** (*float*) – In Volts

Returns

beta – Beam deflection in radians

Return type

NumPy array

Examples

```
>>> import numpy as np
>>> import diffsims.utils.sim_utils as sim_utils
>>> data = np.random.random((100, 100))  # In Tesla*meter
>>> acceleration_voltage = 200000  # 200 kV (in Volt)
>>> beta = sim_utils.bst_to_beta(data, acceleration_voltage)
```

diffsims.utils.sim_utils.**diffraction_scattering_angle**(*acceleration_voltage*, *lattice_size*, *miller_index*)

Get electron scattering angle from a crystal lattice.

Returns the total scattering angle, as measured from the middle of the direct beam (0, 0, 0) to the given Miller index.

Miller index: h, k, l = miller_index Interplanar distance: d = a / (h**2 + k**2 + l**2)**0.5 Bragg's law: theta = arcsin(electron_wavelength / (2 * d)) Total scattering angle (phi): phi = 2 * theta

Parameters

- **acceleration_voltage** (*float*) – In Volt
- **lattice_size** (*float or array-like*) – In meter
- **miller_index** (*tuple*) – (h, k, l)

Returns

angle – Scattering angle in radians.

Return type

float

diffsims.utils.sim_utils.**et_to_beta**(*et*, *acceleration_voltage*)

Calculate beam deflection (beta) values from E * t.

Parameters

- **et** (*NumPy array*) – Electric field times thickness t of the sample.
- **acceleration_voltage** (*float*) – In Volts

Returns
beta – Beam deflection in radians

Return type
NumPy array

Examples

```
>>> import numpy as np
>>> import diffsims.utils.sim_utils as sim_utils
>>> data = np.random.random((100, 100))
>>> acceleration_voltage = 200000  # 200 kV (in Volt)
>>> beta = sim_utils.et_to_beta(data, acceleration_voltage)
```

diffsims.utils.sim_utils.**get_atomic_scattering_factors**(*g_hkl_sq*, *coeffs*, *scattering_params*)

Calculate atomic scattering factors for n atoms.

Parameters

- **g_hkl_sq** (*numpy.ndarray*) – One-dimensional array of g-vector lengths squared.
- **coeffs** (*numpy.ndarray*) – Three-dimensional array [n, 5, 2] of coefficients corresponding to the n atoms.
- **scattering_params** (*str*) – Type of scattering factor calculation to use. One of 'lobato', 'xtables'.

Returns
scattering_factors – The calculated atomic scattering parameters.

Return type
numpy.ndarray

diffsims.utils.sim_utils.**get_electron_wavelength**(*accelerating_voltage*)

Calculates the (relativistic) electron wavelength in Angstroms for a given accelerating voltage in kV.

Parameters
accelerating_voltage (*float or 'inf'*) – The accelerating voltage in kV. Values *numpy.inf* and 'inf' are also accepted.

Returns
wavelength – The relativistic electron wavelength in Angstroms.

Return type
float

diffsims.utils.sim_utils.**get_holz_angle**(*electron_wavelength*, *lattice_parameter*)

Converts electron wavelength and lattice paramater to holz angle :param electron_wavelength: In nanometers :type electron_wavelength: scalar :param lattice_parameter: In nanometers :type lattice_parameter: scalar

Returns
scattering_angle – Scattering angle in radians

Return type
scalar

Examples

```
>>> import diffsims.utils.sim_utils as sim_utils
>>> lattice_size = 0.3905 # STO-(001) in nm
>>> wavelength = 2.51/1000 # Electron wavelength for 200 kV
>>> angle = sim_utils.get_holz_angle(wavelength, lattice_size)
```

diffsims.utils.sim_utils.**get_intensities_params**(*reciprocal_lattice*, *reciprocal_radius*)

Calculates the variables needed for get_kinematical_intensities

Parameters

- **reciprocal_lattice** (*diffpy.Structure.Lattice*) – The reciprocal crystal lattice for the structure of interest.
- **reciprocal_radius** (*float*) – The radius of the sphere in reciprocal space (units of reciprocal Angstroms) within which reciprocal lattice points are returned.

Returns

- **unique_hkls** (*array-like*) – The unique plane families which lie in the given reciprocal sphere.
- **multiplicites** (*array-like*) – The multiplicites of the given unqiue planes in the sphere.
- **g_hkls** (*list*) – The g vector length of the given hkl in the sphere.

diffsims.utils.sim_utils.**get_interaction_constant**(*accelerating_voltage*)

Calculates the interaction constant, sigma, for a given accelerating voltage.

Parameters
accelerating_voltage (*float*) – The accelerating voltage in V.

Returns
sigma – The relativistic electron wavelength in m.

Return type
float

diffsims.utils.sim_utils.**get_kinematical_intensities**(*structure*, *g_indices*, *g_hkls_array*, *debye_waller_factors=None*, *scattering_params='lobato'*, *prefactor=1*)

Calculates peak intensities.

The peak intensity is a combination of the structure factor for a given peak and the position the Ewald sphere intersects the relrod. In this implementation, the intensity scales linearly with proximity.

Parameters

- **structure** (*diffpy.structure.Structure*) – The structure for which to derive the structure factors.
- **g_indices** (*numpy.ndarray*) – Indicies of spots to be considered.
- **g_hkls_array** (*numpy.ndarray*) – Coordinates of spots to be considered.
- **debye_waller_factors** (*dict*) – Maps element names to their temperature-dependent Debye-Waller factors.
- **scattering_params** (*str*) – “lobato”, “xtables” or None
- **prefactor** (*float or numpy.ndarray*) – Multiplciation factor for structure factor.

Returns
peak_intensities – The intensities of the peaks.

Return type
numpy.ndarray

diffsims.utils.sim_utils.**get_points_in_sphere**(*reciprocal_lattice*, *reciprocal_radius*)

Finds all reciprocal lattice points inside a given reciprocal sphere. Utilised within the DiffractionGenerator.

Parameters

- **reciprocal_lattice** (*diffpy.Structure.Lattice*) – The reciprocal crystal lattice for the structure of interest.
- **reciprocal_radius** (*float*) – The radius of the sphere in reciprocal space (units of reciprocal Angstroms) within which reciprocal lattice points are returned.

Returns

- **spot_indices** (*numpy.array*) – Miller indices of reciprocal lattice points in sphere.
- **cartesian_coordinates** (*numpy.array*) – Cartesian coordinates of reciprocal lattice points in sphere.
- **spot_distances** (*numpy.array*) – Distance of reciprocal lattice points in sphere from the origin.

diffsims.utils.sim_utils.**get_scattering_params_dict**(*scattering_params*)

Get scattering parameter dictionary from name.

Parameters
scattering_params (*string*) – Name of scattering factors. One of 'lobato', 'xtables'.

Returns
scattering_params_dict – Dictionary of scattering parameters mapping from element name.

Return type
dict

diffsims.utils.sim_utils.**get_unique_families**(*hkls*)

Returns unique families of Miller indices, which must be permutations of each other.

Parameters
hkls (*list*) – List of Miller indices ([h, k, l])

Returns
pretty_unique – A dict with unique hkl and multiplicity {hkl: multiplicity}.

Return type
dict

diffsims.utils.sim_utils.**get_vectorized_list_for_atomic_scattering_factors**(*structure*, *debye_waller_factors*, *scattering_params*)

Create a flattened array of coeffs, fcoords and occus for vectorized computation of atomic scattering factors.

Note: The dimensions of the returned objects are not necessarily the same size as the number of atoms in the structure as each partially occupied specie occupies its own position in the flattened array.

Parameters

- **structure** (*diffpy.structure.Structure*) – The atomic structure for which scattering factors are required.
- **debye_waller_factors** (*dist*) – Debye-Waller factors for atoms in the structure.
- **scattering_params** (*string*) – The type of scattering params to use. “lobato”, “xtables”, and None are supported.

Returns

- **coeffs** (*numpy.ndarray*) – Coefficients of atomic scattering factor parameterization for each atom.
- **fcoords** (*numpy.ndarray*) – Fractional coordinates of each atom in structure.
- **occus** (*numpy.ndarray*) – Occupancy of each atomic site.
- **dwfactors** (*numpy.ndarray*) – Debye-Waller factors for each atom in the structure.

diffsims.utils.sim_utils.**is_lattice_hexagonal**(*latt*)

Determines if a diffpy lattice is hexagonal or trigonal. :param latt: The diffpy lattice object to be determined as hexagonal or not. :type latt: diffpy.Structure.lattice

Returns
is_true – True if hexagonal or trigonal.

Return type
bool

diffsims.utils.sim_utils.**scattering_angle_to_lattice_parameter**(*electron_wavelength*, *angle*)

Convert scattering angle data to lattice parameter sizes.

Parameters

- **electron_wavelength** (*float*) – Wavelength of the electrons in the electron beam. In nm. For 200 kV electrons: 0.00251 (nm)
- **angle** (*NumPy array*) – Scattering angle, in radians.

Returns
lattice_parameter – Lattice parameter, in nanometers

Return type
NumPy array

Examples

```
>>> import diffsims.utils.sim_utils as sim_utils
>>> angle_list = [0.1, 0.1, 0.1, 0.1] # in radians
>>> wavelength = 2.51/1000 # Electron wavelength for 200 kV
>>> lattice_size = sim_utils.scattering_angle_to_lattice_parameter(
...     wavelength, angle_list)
```

diffsims.utils.sim_utils.**simulate_kinematic_scattering**(*atomic_coordinates*, *element*, *accelerating_voltage*, *simulation_size=256*, *max_k=1.5*, *illumination='plane_wave'*, *sigma=20*, *scattering_params='lobato'*)

Simulate electron scattering from arrangement of atoms comprising one elemental species.

Parameters

- **atomic_coordinates** (*array*) – Array specifying atomic coordinates in structure.
- **element** (*string*) – Element symbol, e.g. "C".
- **accelerating_voltage** (*float*) – Accelerating voltage in keV.
- **simulation_size** (*int*) – Simulation size, n, specifies the n x n array size for the simulation calculation.
- **max_k** (*float*) – Maximum scattering vector magnitude in reciprocal angstroms.
- **illumination** (*string*) – Either 'plane_wave' or 'gaussian_probe' illumination
- **sigma** (*float*) – Gaussian probe standard deviation, used when illumination == 'gaussian_probe'
- **scattering_params** (*string*) – Type of scattering factor calculation to use. One of 'lobato', 'xtables'.

Returns
simulation – ElectronDiffraction simulation.

Return type
ElectronDiffraction

diffsims.utils.sim_utils.**tesla_to_am**(*data*)

Convert data from Tesla to A/m

Parameters
data (*NumPy array*) – Data in Tesla

Returns
output_data – In A/m

Return type
NumPy array

Examples

```
>>> import numpy as np
>>> import diffsims.utils.sim_utils as sim_utils
>>> data_T = np.random.random((100, 100))  # In tesla
>>> data_am = sim_utils.tesla_to_am(data_T)
```

diffsims.utils.sim_utils.**uvtw_to_uvw**(*uvtw*)

Convert 4-index direction to a 3-index direction.

Parameters
uvtw (*array-like with 4 floats*) –

Returns
uvw

Return type
tuple of 4 floats

3.7.12 vector_utils

diffsims.utils.vector_utils.**get_angle_cartesian**(*a*, *b*)

Compute the angle between two vectors in a cartesian coordinate system.

Parameters

- **a** (*array-like with 3 floats*) – The two directions to compute the angle between.
- **b** (*array-like with 3 floats*) – The two directions to compute the angle between.

Returns

angle – Angle between *a* and *b* in radians.

Return type

float

diffsims.utils.vector_utils.**get_angle_cartesian_vec**(*a*, *b*)

Compute the angles between two lists of vectors in a cartesian coordinate system.

Parameters

- **a** (*np.array()*) – The two lists of directions to compute the angle between in Nx3 float arrays.
- **b** (*np.array()*) – The two lists of directions to compute the angle between in Nx3 float arrays.

Returns

angles – List of angles between *a* and *b* in radians.

Return type

np.array()

diffsims.utils.vector_utils.**vectorised_spherical_polars_to_cartesians**(*z*)

Converts an array of spherical polars into an array of (x,y,z) = r(cos(psi)sin(theta),sin(psi)sin(theta),cos(theta))

Parameters

z (*np.array*) – With rows of r : the radius value, r = sqrt(x**2+y**2+z**2) psi : The azimuthal angle generally (0,2pi]) theta : The elevation angle generally (0,pi)

Returns

xyz – With rows of x,y,z

Return type

np.array

CHAPTER

FOUR

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

4.1 2022-06-10 - version 0.5.0

4.1.1 Added

- Extra parameters in diffraction pattern's plot method for drawing miller index labels next to the diffraction spots.
- Option to use None for `scattering_params` which ignores atomic scattering.
- Python 3.10 support.
- Class `ReciprocalLatticeVector` for handling generation, handling and plotting of vectors. This class replaces `ReciprocalLatticePoint`, which is deprecated.

4.1.2 Changed

- Minimal version of dependencies orix >= 0.9, numpy >= 1.17 and tqdm >= 4.9.
- The Laue group representing the rotation list sampling of "hexagonal" from 6/m to 6/mmm.
- Loosened the angle tolerance in `DiffractionLibrary.get_library_entry()` from `1e-5` to `1e-2`.

4.1.3 Deprecated

- Class `ReciprocalLatticePoint` is deprecated and will be removed in v0.6. Use `ReciprocalLatticeVector` instead.

4.2 2021-04-16 - version 0.4.2

4.2.1 Added

- Simulations now have a .get_as_mask() method (#154, #158)
- Python 3.9 testing (#161)

4.2.2 Changed

- Simulations now use a fractional (rather than absolute) min_intensity (#161)

4.2.3 Fixed

- Precession simulations (#161)

4.3 2021-03-15 - version 0.4.1

4.3.1 Changed

- *get_grid_beam_directions* default meshing changed to "spherified_cube_edge" from "spherified_cube_corner"

4.3.2 Fixed

- *get_grid_beam_directions* now behaves correctly for the triclinic and monoclinic cases

4.4 2021-01-11 - version 0.4.0

4.4.1 Added

- API reference documentation via Read The Docs: https://diffsims.readthedocs.io
- New module: *sphere_mesh_generators*
- New module: *detector_functions*
- New module: *ring_pattern_utils*
- beam precession is now supported in simulating electron diffraction patterns
- plot method for *DiffractionSimulation*
- more shape factor functions have been added
- This project now keeps a Changelog

4.4.2 Changed

- *get_grid_beam_directions*, now works based off of meshes
- the arguments in the *DiffractionGenerator* constructor and the *DiffractionLibraryGenerator.get_diffraction_library* function have been shuffled so that the former captures arguments related to "the instrument/physics" while the latter captures arguments relevant to "the sample/material".
- CI is now provided by github actions

4.4.3 Removed

- Python 3.6 testing

4.4.4 Fixed

- ReciprocalLatticePoint handles having only one point/vector

CHAPTER

FIVE

CONTRIBUTOR GUIDE

This guide is intended to get new developers started with contributing to diffsims.

Many potential contributors will be scientists with much expert knowledge but potentially little experience with open-source code development. This guide is primarily aimed at this audience, helping to reduce the barrier to contribution.

We have a Code of Conduct that must be honoured by contributors.

5.1 Start using diffsims

The best way to start understanding how diffsims works is to use it.

For developing the code the home of diffsims is on GitHub and you'll see that a lot of this guide boils down to using that platform well. so visit the following link and poke around the code, issues, and pull requests (PRs): diffsims on GitHub.

It's probably also worth visiting the GitHub guides to get a feel for the terminology.

In brief, to give you a hint on the terminology to search for, the contribution pattern is:

1. Setup git/GitHub if you don't have it.
2. Fork diffsims on GitHub.
3. Checkout your fork on your local machine.
4. Create a new branch locally where you will make your changes.
5. Push the local changes to your own github fork.
6. Create a PR to the official diffsims repository.

Note: You cannot mess up the main diffsims project. So when you're starting out be confident to play, get it wrong, and if it all goes wrong you can always get a fresh install of diffsims!

PS: If you choose to develop in Windows/Mac you may find the Github Desktop useful.

5.2 Questions?

Open source projects are all about community - we put in much effort to make good tools available to all and most people are happy to help others start out. Everyone had to start at some point and the philosophy of these projects centers around the fact that we can do better by working together.

Much of the conversation happens in 'public' using the 'issues' pages on GitHub – doing things in public can be scary but it ensures that issues are identified and logged until dealt with. This is also a good place to make a proposal for some new feature or tool that you want to work on.

5.3 Good coding practice

The most important aspects of good coding practice are: (1) to work in manageable branches, (2) develop a good code style, (3) write tests for new functions, and (4) document what the code does. Tips on these points are provided below.

5.3.1 Use git to work in manageable branches

Git is an open source "version control" system that enables you to can separate out your modifications to the code into many versions (called branches) and switch between them easily. Later you can choose which version you want to have integrated into diffsims.

You can learn all about Git here!

The most important thing is to separate your contributions so that each branch is a small advancement on the "master" code or on another branch.

5.3.2 Get the style right

diffsims closely follows the Style Guide for Python Code - these are just some rules for consistency that you can read all about in the Python Style Guide.

Please run the latest version of black on your newly added and modified files prior to each PR.

5.3.3 Run and write tests

All functionality in diffsims is tested via the pytest framework. The tests reside in the `diffsims.tests` module. Tests are short functions that call functions in diffsims and compare resulting output values with known answers. Good tests should depend on as few other features as possible so that when they break we know exactly what caused it.

Install necessary dependencies to run the tests:

```
pip install --editable .[tests]
```

Some useful fixtures are available in the `conftest.py` file.

To run the tests:

```
pytest --cov --pyargs diffsims
```

The `--cov` flag makes coverage.py print a nice report in the terminal. For an even nicer presentation, you can use `coverage.py` directly:

```
coverage html
```

Then, you can open the created `htmlcov/index.html` in the browser and inspect the coverage in more detail.

Useful hints on testing:

- When comparing integers, it's fine to use `==`. When comparing floats use something like assert `np.allclose(shifts, shifts_expected, atol=0.2)`.
- `@pytest.mark.parametrize()` is a convenient decorator to test several parameters of the same function without having to write to much repetitive code, which is often error-prone. See pytest documentation for more details.

5.3.4 Build and write documentation

Docstrings – written at the start of a function – give essential information about how it should be used, such as which arguments can be passed to it and what the syntax should be. The docstrings follow the NumPy specification, as shown in this example.

We use Sphinx for documenting functionality. Install necessary dependencies to build the documentation:

```
pip install -e .[doc]
```

Then, build the documentation from the `doc` directory:

```
cd doc
make html
```

The documentation's HTML pages are built in the `doc/build/html` directory from files in the reStructuredText (reST) plaintext markup language. They should be accessible in the browser by typing `file:///your-absolute/path/to/diffsims/doc/build/html/index.html` in the address bar.

5.4 Continuous integration (CI)

We use GitHub Actions to ensure that diffsims can be installed on Windows, macOS and Linux. After a successful installation, the CI server runs the tests. After the tests return no errors, code coverage is reported to Coveralls.

5.5 Learn more

1. The Python programming language, for beginners.

CHAPTER

SIX

RELATED PROJECTS

Related, open-source projects that users of diffsims might find useful:

- pyxem: Python library for multi-dimensional diffraction microscopy. Uses diffsims.
- orix: Python library for handling crystal orientation mapping data.
- kikuchipy: Python library for processing and analysis of electron backscatter diffraction (EBSD) patterns. Uses diffsims.

BIBLIOGRAPHY

[Cajaravelli2015] O. S. Cajaravelli, "Four Ways to Create a Mesh for a Sphere," https://medium.com/game-dev-daily/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4.

[Meshzoo] The *meshzoo.sphere* module, https://github.com/nschloe/meshzoo/blob/master/meshzoo/sphere.py.

[DeGraef2007] M. De Graef, M. E. McHenry, "Structure of Materials," Cambridge University Press (2007).

[Doyle1968] P. A. Doyle, P. S. Turner, "Relativistic Hartree-Fock X-ray and electron scattering factors," *Acta Cryst.* **24** (1968), doi: https://doi.org/10.1107/S0567739468000756.

[Smith1962] G. Smith, R. Burge, "The analytical representation of atomic scattering amplitudes for electrons," *Acta Cryst.* **A15** (1962), doi: https://doi.org/10.1107/S0365110X62000481.

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

H

I

K

L

T

U

V

W

X

Y

Z