TEMUL Toolkit Documentation

Eoghan O'Connell

Apr 04, 2022

CONTENTS

The TEMUL Toolkit is a suit of functions and classes for analysis and visualisation of atomic resolution images. It is mostly built upon the data structure of HyperSpy and Atomap.

CHAPTER

ONE

INTERACTIVE EXAMPLES

The easiest way to try the TEMUL Toolkit is via Binder: introductory Jupyter Notebook. To install the TEMUL Toolkit on your own computer, see the *Installation* instructions.

And there are more examples with Binder, just click the below button!

Click the button above to start some data analysis (it may take a few minutes to load). The "code_tutorials" folder contains walkthroughs of some of the documentation examples from this website. The "publication_examples" folder will allow you to analyse data from published scientific papers! Just navigate to whichever of these folders you want click on the ".ipynb" files.

CHAPTER

TWO

SOME PUBLISHED RESULTS USING THE TEMUL TOOLKIT

CHAPTER

THREE

NEWS

3.1 16/02/2021: Version 0.1.4 released

The polarisation, structure tools and fft mapping has now been refactored into the topotem module. The temul functionality remains the same i.e. `import temul.api as tml`.

3.2 16/02/2021: Version 0.1.3 released

First articles uses and citations for the TEMUL Toolkit! This version updated the Publication Examples folder with two newly published articles. The folder contains interactive and raw code on how to reproduce the data in the publications. Congrats to those involved!

- M. Hadjimichael, Y. Li *et al*, Metal-ferroelectric supercrystals with periodically curved metallic layers, Nature Materials 2020
- K. Moore *et al* Highly charged 180 degree head-to-head domain walls in lead titanate, Nature Communications Physics 2020

If you have a question or issue with using the publication examples, please make an issue on GitHub.

Code changes in this version:

- The `atom_deviation_from_straight_line_fit` function has been **corrected** and expanded. For a use case, see *Finding Polarisation Vectors*
- Corrected the `plot_polarisation_vectors` function's vector quiver key.
- Created the "polar_colorwheel" `plot_style` for `plot_polarisation_vectors` by using a HSV to RGB 2D colorwheel and mapping the angles and magnitudes to these values. Used code from PixStem for colorwheel visualisation.
- Fixed `norm` and `cmap` scaling for the colorbar for the "contour", "colorwheel" and "colormap" `plot_styles`. Now each of these `plot_styles` scale nicely, and colorbar `ticks` may be specified.
- Added invert_y_axis param for plot_polarisation_vectors function, useful for testing if angles are displaying correctly.
- `plot_polarisation_vectors` function now returns a Matplotlib `Axes` object, which can be used to further edit the layout/presentation of the plotted map.
- Added functions to correct for possible off-zone tilt in the atomic columns. Use with caution.

Documentation changes in this version:

- Added documentation for *how to find the polarisation vectors*.

- Added "code_tutorials" ipynb (interactive Jupyter Notebook) examples. See the GitHub repository for downloads.
- The *workflows* folder in "code_tutorials/workflows" also contains starting workflows for analysis of different materials. See the GitHub repository for downloads.
- Added "publication_examples" tutorial ipynb (interactive Jupyter Notebook) examples. See the GitHub repository for downloads.

3.3 03/11/2020: Version 0.1.2 released

This version contains minor changes from the 0.1.1 release. It removes pyCifRW as a dependency.

3.4 02/11/2020: Version 0.1.1 released

This version contains many changes to the TEMUL Toolkit.

- More parameters have been added to the polarisation module's `plot_polarisation_vectors` function. Check out the walkthrough *here* for more info!
- *Interactive double Gaussian filtering* with the `visualise_dg_filter` function in the signal_processing module. Thanks to Michael Hennessy for the help!
- The `calculate_atom_plane_curvature` function has been added, creating the lattice_structure_tools module.
- Strain, rotation, and c/a mapping can now be done *here*.
- Masked FFT filtering to obtain iFFTs. See *this guide* to see some code!
- Example walk-throughs for many features of the TEMUL Toolkit are now on this website! Check out the menu on the left to get started!

3.4.1 Installation

The TEMUL Toolkit can be installed easily with PIP (those using Windows may need to download VS C++ Build Tools, see below).

```
$ pip install temul-toolkit
```

Then, it can be imported with the name "temul". For example, to import most of the temul functionality use:

```
import temul.api as tml
```

Installation Problems & Notes

- If installing on Windows, you will need Visual Studio C++ Build Tools. Download it here. After downloading, choose the "C++ Build Tools" Workload and click install.
- If you want to use the *`temul.io.write_cif_from_dataframe()`* function, you will need to install pyCifRW version 4.3. This requires Visual Studio.
- If you wish to use the `temul.simulations` or `temul.model_refiner` modules, you will need to install PyPrismatic. This requires Visual Studio and other dependencies. **It is unfortunately not guaranteed to work**. If you want to help develop the `temul.model_refiner.Model_Refiner`, please create an issue and/or a pull request on the TEMUL github.

- If you're using any of the functions or classes that require element quantification:
  - navigate to the "temul/external" directory, copy the "atomap_devel_012" folder and paste that in your "site-packages" directory.
  - Then, when using atomap to create sublattices and quantify elements call atomap like this: `import atomap_devel_012.api as am`.
  - This development version is slowly being folded into the master branch here: https://gitlab.com/atomap/atomap/-/issues/93 and any help or tips on implementation are welcome!

3.4.2 Getting started

There are many aspects to the TEMUL Toolkit, such as polarisation analysis, element quantification, and automatic image simulation (through pyprismatic).

Checkout the tutorials in the table of contents above or on the left of the page. One can also view the extensive *documentation*, where each function is described and examples of their use given.

To use the vast majority of the temul functionality, import it from the api module:

```
import temul.api as tml
```

3.4.3 Analysis Workflows

The TEMUL Toolkit contains some basic analysis workflows for the various ferroelectric material-types described in *Finding Polarisation Vectors*.

The python scripts, jupyter notebooks and data can be downloaded from the TEMUL repository in the "code_tutorials/workflows" folder.

You can also interact with the data without needing any downloads. Just click the button below and navigate to that same folder, where you will find the python scripts and interactive python notebooks:

More will be added to these workflows in future. If you have any ideas, please create an issue on GitHub or create a pull request.

3.4.4 Finding Polarisation Vectors

There are several methods available in the TEMUL Toolkit and Atomap packages for finding polarisation vectors in atomic resolution images. These are briefly described here, followed by a use-case of each.

Current functions:

1. Using Atomap's get_polarization_from_second_sublattice Sublattice method. Great for "standard" polarised structures with two sublattices.
2. Using the TEMUL *temul.topotem.polarisation.find_polarisation_vectors()* function. Useful for structures that Atomap's `get_polarization_from_second_sublattice` can't handle.
3. Using the TEMUL *temul.topotem.polarisation.atom_deviation_from_straight_line_fit()* function. Useful for calculating polarisation from a single sublattice, similar to and based off: J. Gonnissen *et al*, Direct Observation of Ferroelectric Domain Walls in LiNbO3: Wall-Meanders, Kinks, and Local Electric Charges, 26, 42, 2016, DOI: 10.1002/adfm.201603489.

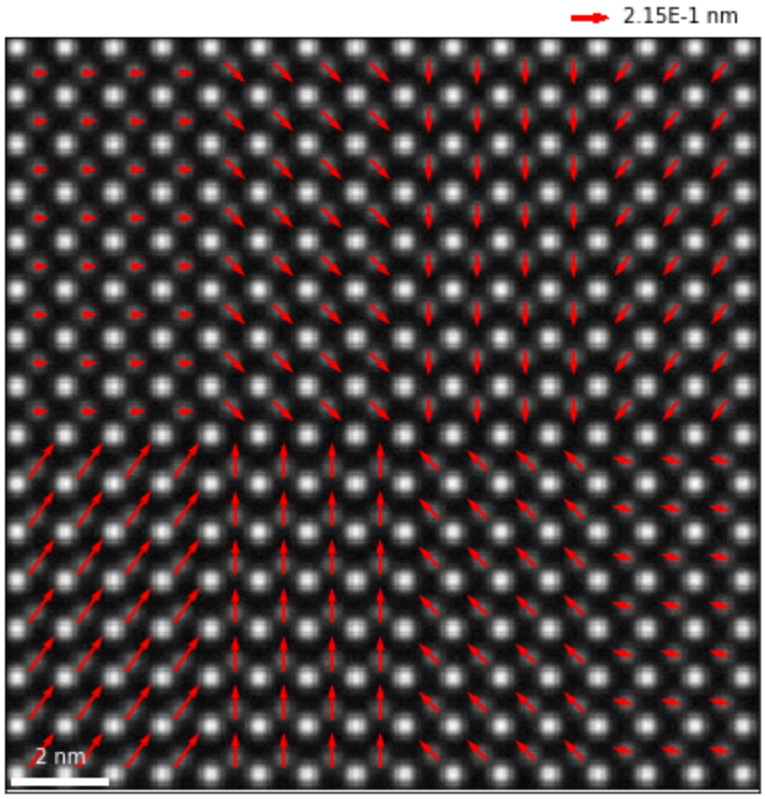
These methods are also available as python scripts and jupyter notebooks in the TEMUL repository in the "code_tutorials/workflows" folder. You can interact with the workflows without needing any downloads. Just click the button below and navigate to that same folder, where you will find the python scripts and interactive python notebooks:

For standard Polarised Structures (e.g., PTO)

Atomap's get_polarization_from_second_sublattice Sublattice method will be sufficent for most users when dealing with the classic PTO-style polarisation, wherein the atoms in a sublattice are polarised with respect to a second sublattice.

See the second section of this tutorial on how to plot this in many different ways using *temul.topotem.polarisation.plot_polarisation_vectors()*!

```
>>> import temul.api as tml
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset(image_noise=True)
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> sublatticeA.construct_zone_axes()
>>> za0, za1 = sublatticeA.zones_axis_average_distances[0:2]
>>> s_p = sublatticeA.get_polarization_from_second_sublattice(
...     za0, za1, sublatticeB)
>>> vector_list = s_p.metadata.vector_list
>>> x, y = [i[0] for i in vector_list], [i[1] for i in vector_list]
>>> u, v = [i[2] for i in vector_list], [i[3] for i in vector_list]
>>> sampling, units =  0.05, 'nm'
>>> tml.plot_polarisation_vectors(x, y, u, v, image=atom_lattice.image,
...                               sampling=sampling, units=units,
...                               unit_vector=False, save=None, scalebar=True,
...                               plot_style='vector', color='r',
...                               overlay=True, monitor_dpi=45)
```



For nonstandard Polarised Structures (e.g., Boracites)

When the above function can't isn't suitable, the TEMUL *temul.topotem.polarisation.find_polarisation_vectors()* function may be an option. It is useful for structures that Atomap's `get_polarization_from_second_sublattice` can't handle. It is a little more involved and requires some extra preparation when creating the sublattices.

See the second section of this tutorial on how to plot this in many different ways using *temul.topotem.polarisation.plot_polarisation_vectors()*!

```
>>> import temul.api as tml
>>> import atomap.api as am
>>> import numpy as np
>>> from temul.dummy_data import get_polarisation_dummy_dataset_bora
>>> signal = get_polarisation_dummy_dataset_bora(True).signal
>>> atom_positions = am.get_atom_positions(signal, separation=7)
>>> sublatticeA = am.Sublattice(atom_positions, image=signal.data)
>>> sublatticeA.find_nearest_neighbors()
>>> sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> zone_axis_001 = sublatticeA.zones_axis_average_distances[0]
```

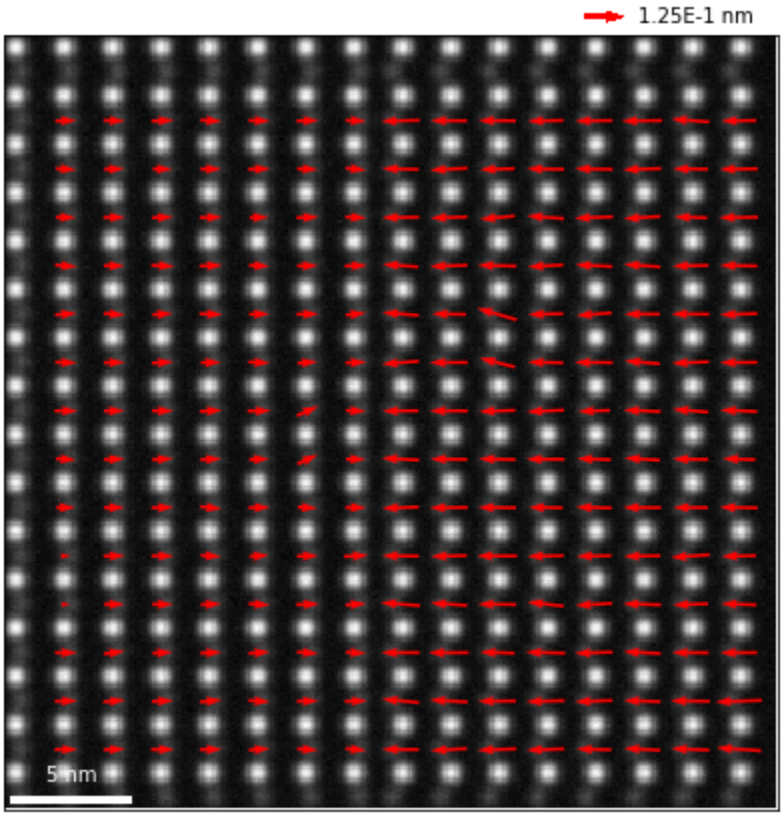
(continues on next page)

(continued from previous page)

```
>>> atom_positions2 = sublatticeA.find_missing_atoms_from_zone_vector(
...     zone_axis_001, vector_fraction=0.5)
>>> sublatticeB = am.Sublattice(atom_positions2, image=signal.data,
...                             color='blue')
>>> sublatticeB.find_nearest_neighbors()
>>> sublatticeB.refine_atom_positions_using_center_of_mass(percent_to_nn=0.2)
>>> atom_positions2_refined = np.array([sublatticeB.x_position,
...                                     sublatticeB.y_position]).T
>>> atom_positions2 = np.asarray(atom_positions2).T
```

We then use the original (ideal) positions "atom_positions2" and the refined positions "atom_positions2_refined" to calculate and visualise the polarisation in the structure. Don't forget to save these arrays for further use!

```
>>> u, v = tml.find_polarisation_vectors(atom_positions2,
...                                      atom_positions2_refined)
>>> x, y = sublatticeB.x_position.tolist(), sublatticeB.y_position.tolist()
>>> sampling, units =  0.1, 'nm'
>>> tml.plot_polarisation_vectors(x, y, u, v, image=signal.data,
...                           sampling=sampling, units=units, scalebar=True,
...                           unit_vector=False, save=None,
...                           plot_style='vector', color='r',
...                           overlay=True, monitor_dpi=45)
```



For single Polarised Sublattices (e.g., LNO)

When dealing with structures in which the polarisation must be extracted from a single sublattice (one type of chemical atomic column, the TEMUL *temul.topotem.polarisation.atom_deviation_from_straight_line_fit()* function may be an option. It is based off the description by J. Gonnissen *et al*, Direct Observation of Ferroelectric Domain Walls in LiNbO3: Wall-Meanders, Kinks, and Local Electric Charges, 26, 42, 2016, DOI: 10.1002/adfm.201603489.

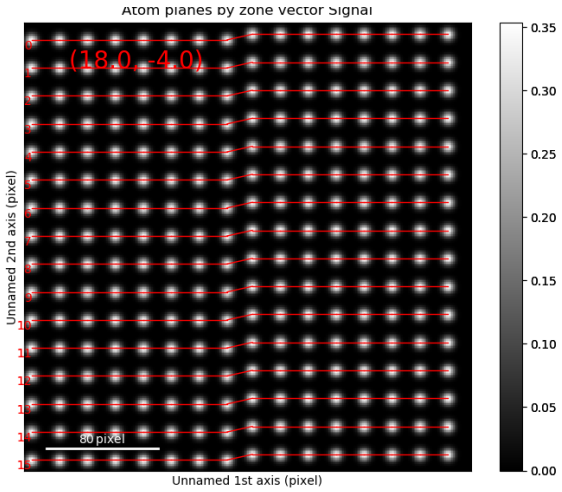
See the second section of this tutorial on how to plot this in many different ways using *temul.topotem.polarisation.plot_polarisation_vectors()*!

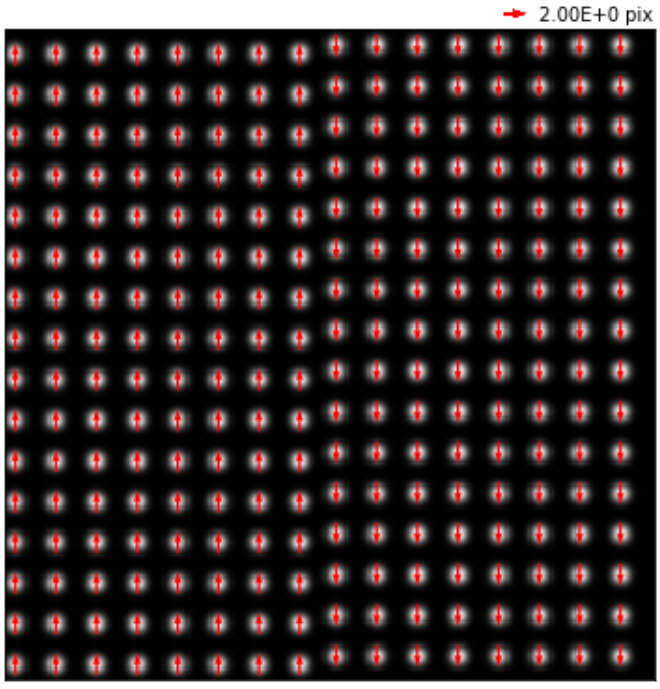
```
>>> import temul.api as tml
>>> import temul.dummy_data as dd
>>> sublattice = dd.get_polarised_single_sublattice()
>>> sublattice.construct_zone_axes(atom_plane_tolerance=1)
>>> sublattice.plot_planes()

Choose `n`: how many atom columns should be used to fit the line on each
side of the atom planes. If `n` is too large, the fitting will appear
incorrect.
```

(continues on next page)

(continued from previous page)

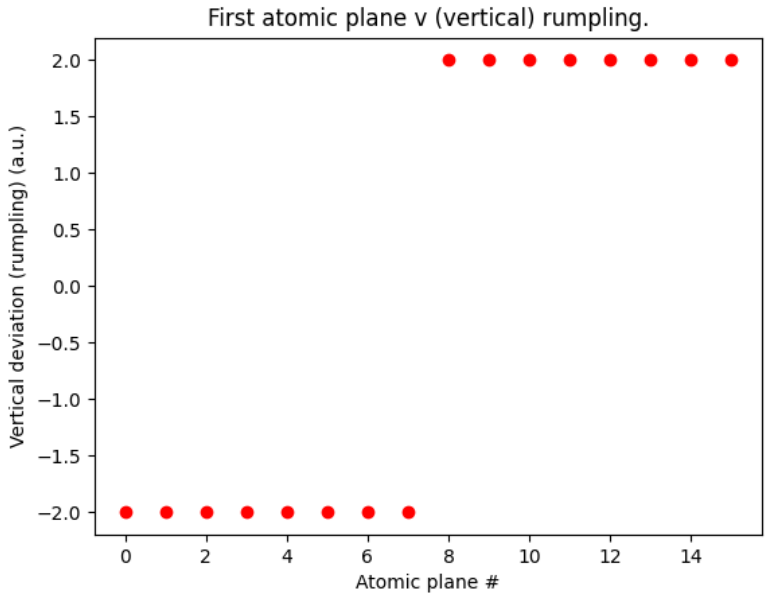
```
>>> n = 5
>>> x, y, u, v = tml.atom_deviation_from_straight_line_fit(
...     sublattice, 0, n)
>>> tml.plot_polarisation_vectors(x, y, u, v, image=sublattice.image,
...                               unit_vector=False, save=None,
...                               plot_style='vector', color='r',
...                               overlay=True, monitor_dpi=50)
```





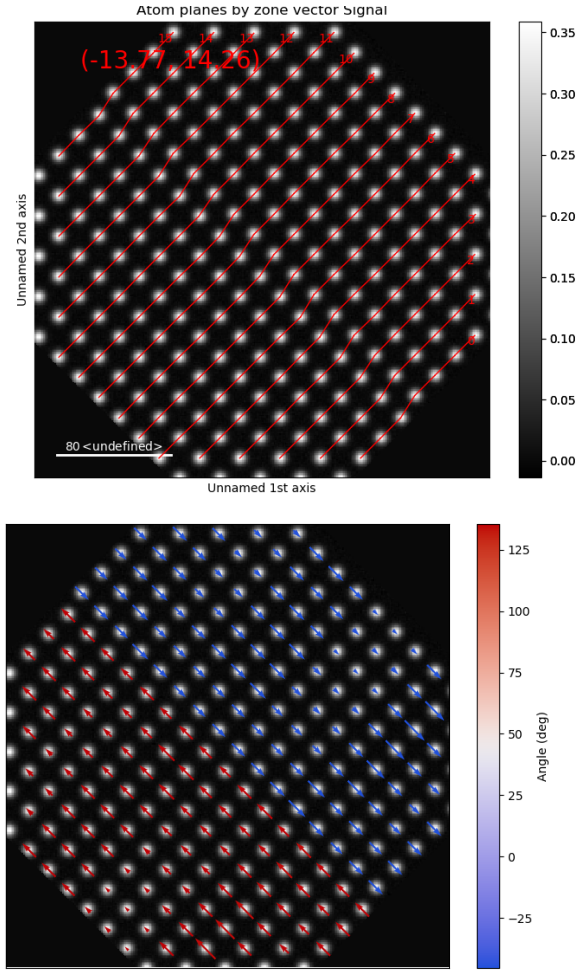
We can look at individual atomic column information by using `return_individual_atom_planes = True`.

```
>>> return_individual_atom_planes = True
>>> x, y, u, v = tml.atom_deviation_from_straight_line_fit(
...     sublattice, axis_number=0, n=5, second_fit_rigid=True, plot=False,
...     return_individual_atom_planes=return_individual_atom_planes)

To look at first atomic plane rumpling

>>> plt.figure()
>>> plt.plot(range(len(v[0])), v[0], 'ro')
>>> plt.title("First atomic plane v (vertical) rumpling.")
>>> plt.xlabel("Atomic plane #")
>>> plt.ylabel("Vertical deviation (rumpling) (a.u.)")
>>> plt.show()

We could also of course use a numpy array to handle the data.

>>> arr = np.array([x, y, u, v]).T
```



Let's look at some rotated data

```
>>> sublattice = dd.get_polarised_single_sublattice_rotated(
...     image_noise=True, rotation=45)
>>> sublattice.construct_zone_axes(atom_plane_tolerance=0.9)
>>> sublattice.plot_planes()
>>> n = 3  # plot the sublattice to see why 3 is suitable here!
>>> x, y, u, v = tml.atom_deviation_from_straight_line_fit(
...     sublattice, 0, n)
>>> tml.plot_polarisation_vectors(x, y, u, v, image=sublattice.image,
...                       vector_rep='angle', save=None, degrees=True,
...                       plot_style='colormap', cmap='cet_coolwarm',
...                       overlay=True, monitor_dpi=50)
```



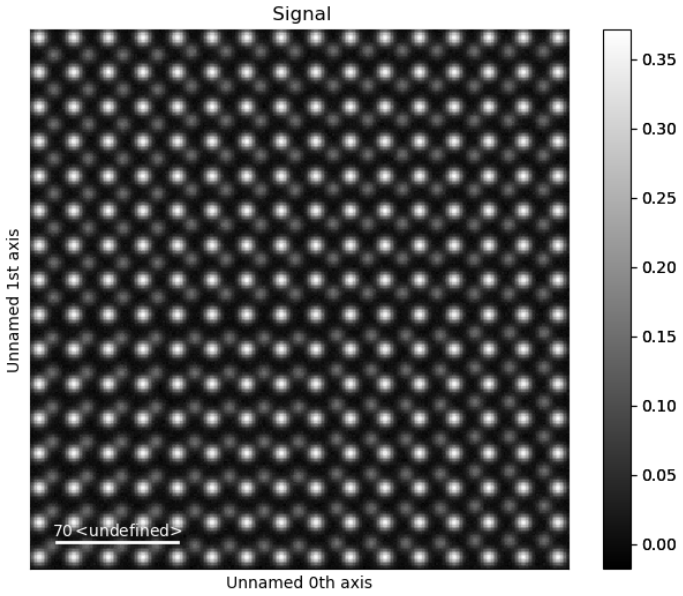
3.4.5 Plotting Polarisation and Movement Vectors

The *temul.topotem.polarisation* module allows one to visualise the polarisation/movement of atoms in an atomic resolution image. In this tutorial, we will use a dummy dataset to show the different ways the *temul.topotem.polarisation.plot_polarisation_vectors()* function can display data. In future, tutorials on published experimental data will also be available.

To go through the below examples in a live Jupyter Notebook session, click the button below and choose "code_tutorials/polarisation_vectors_tutorial.ipynb" (it may take a few minutes to load).

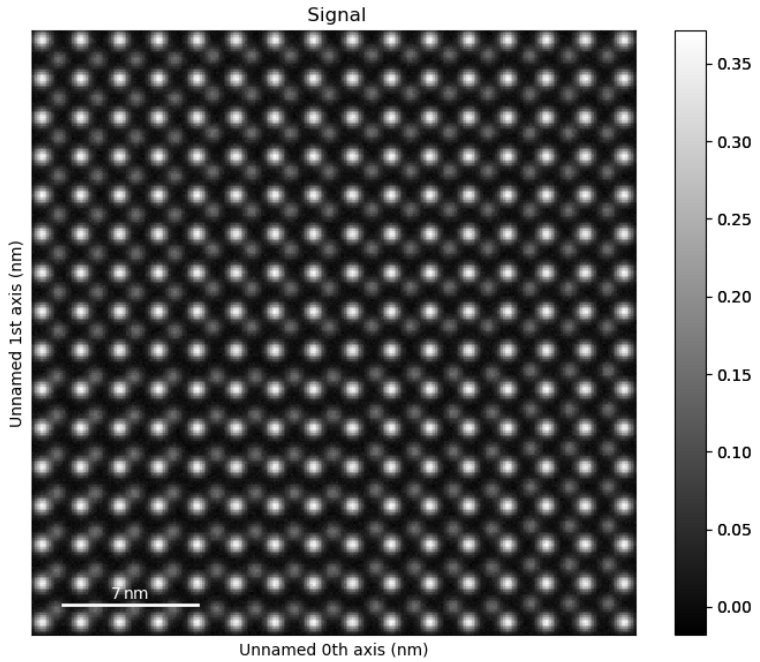
Prepare and Plot the dummy dataset

```
>>> import temul.api as tml
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset(image_noise=True)
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> image = sublatticeA.signal
>>> image.plot()
```

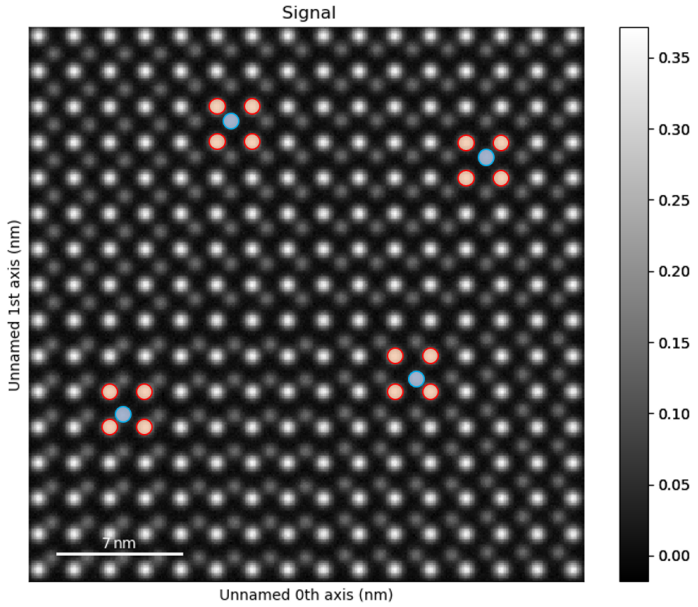


It is best when the image is calibrated. Your image may already be calibrated, but if not, use Hyperspy's `axes_manager` for calibration.

```
>>> sampling = 0.1  # example of 0.1 nm/pix
>>> units = 'nm'
>>> image.axes_manager[-1].scale = sampling
>>> image.axes_manager[-2].scale = sampling
>>> image.axes_manager[-1].units = units
>>> image.axes_manager[-2].units = units
>>> image.plot()
```



Zoom in on the image to see how the atoms look in the different regions.



Find the Vector Coordinates using Atomap

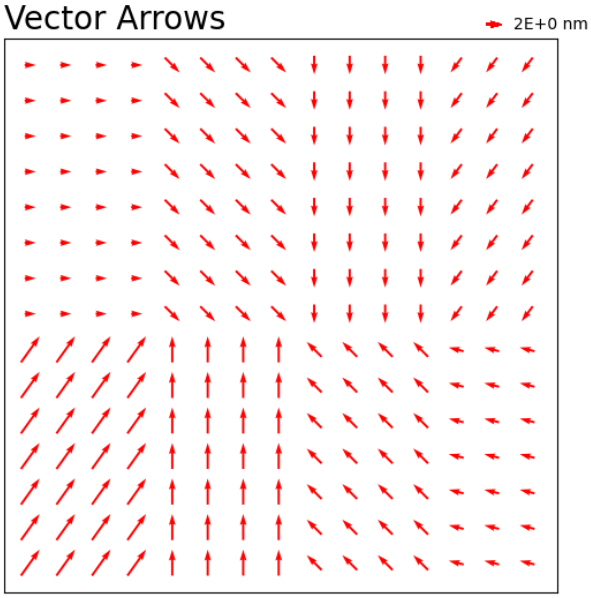
Using the Atomap package, we can easily get the polarisation vectors for regular structures.

```
>>> sublatticeA.construct_zone_axes()
>>> za0, za1 = sublatticeA.zones_axis_average_distances[0:2]
>>> s_p = sublatticeA.get_polarization_from_second_sublattice(
...     za0, za1, sublatticeB, color='blue')
>>> vector_list = s_p.metadata.vector_list
>>> x, y = [i[0] for i in vector_list], [i[1] for i in vector_list]
>>> u, v = [i[2] for i in vector_list], [i[3] for i in vector_list]
```

Now we can display all of the variations that *temul.topotem.polarisation.plot_polarisation_vectors()* gives us! You can specify sampling (scale) and units, or use a calibrated image so that they are automatically set.

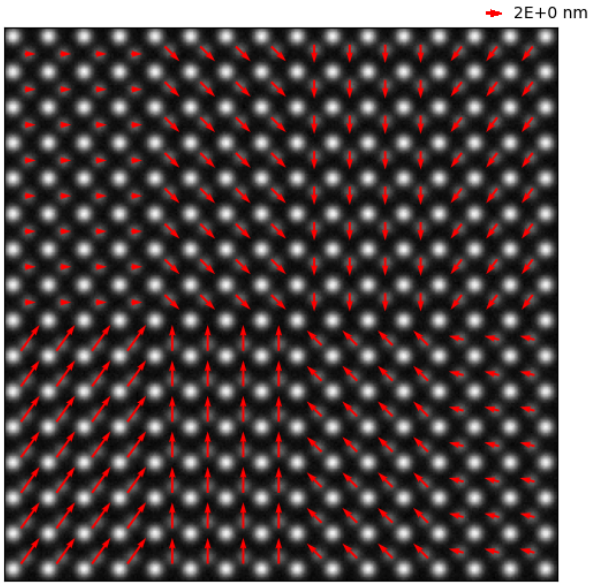
Vector magnitude plot with red arrows:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=False, save=None,
...                               plot_style='vector', color='r',
...                               overlay=False, title='Vector Arrows',
...                               monitor_dpi=50)
```



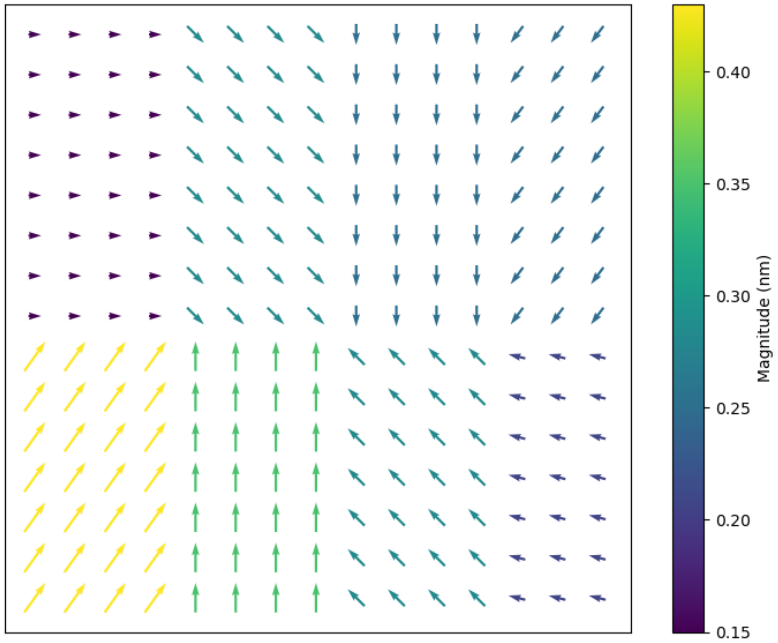
Vector magnitude plot with red arrows overlaid on the image, no title:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=False, save=None,
...                               plot_style='vector', color='r',
...                               overlay=True, monitor_dpi=50)
```



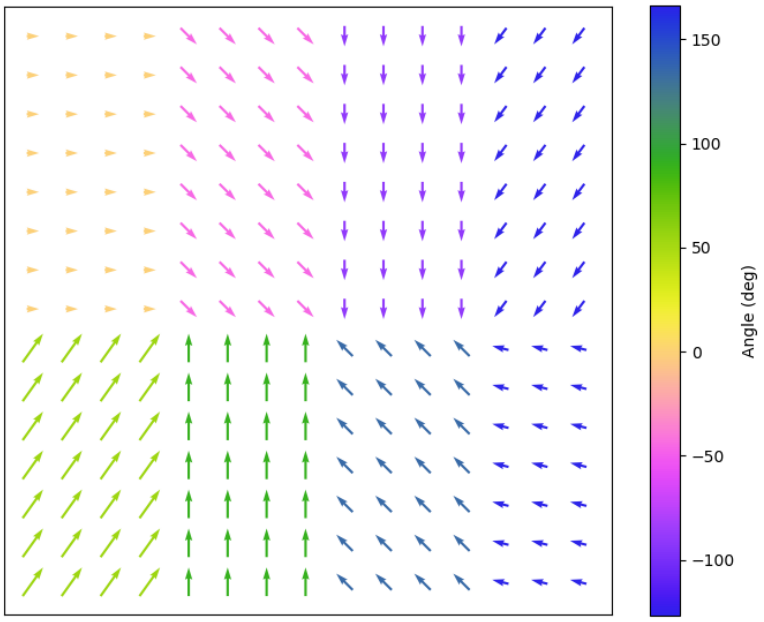
Vector magnitude plot with colormap viridis:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=False, save=None,
...                               plot_style='colormap', monitor_dpi=50,
...                               overlay=False, cmap='viridis')
```



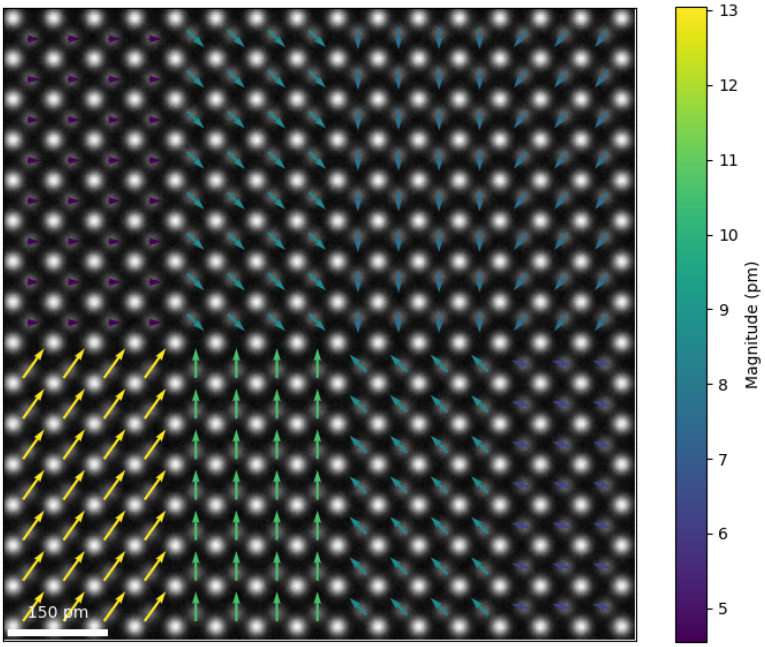
Vector angle plot with colormap viridis (`vector_rep='angle'`):

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=False, save=None,
...                               plot_style='colormap', monitor_dpi=50,
...                               overlay=False, cmap='cet_colorwheel',
...                               vector_rep="angle", degrees=True)
```



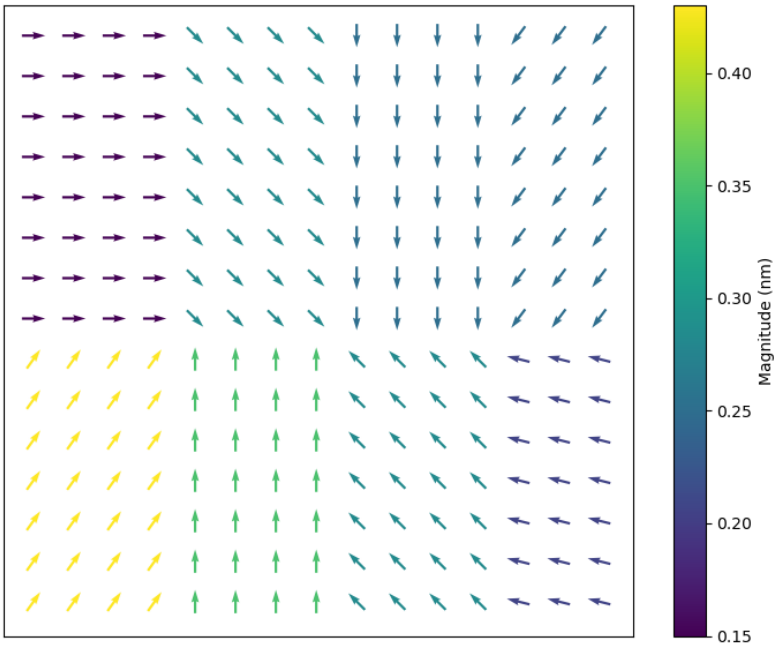
Colormap arrows with sampling specified in the parameters and with scalebar:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                               sampling=3.0321, units='pm', monitor_dpi=50,
...                               unit_vector=False, plot_style='colormap',
...                               overlay=True, save=None, cmap='viridis',
...                               scalebar=True)
```



Vector plot with colormap viridis and unit vectors:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=True, save=None, monitor_dpi=50,
...                               plot_style='colormap', color='r',
...                               overlay=False, cmap='viridis')
```



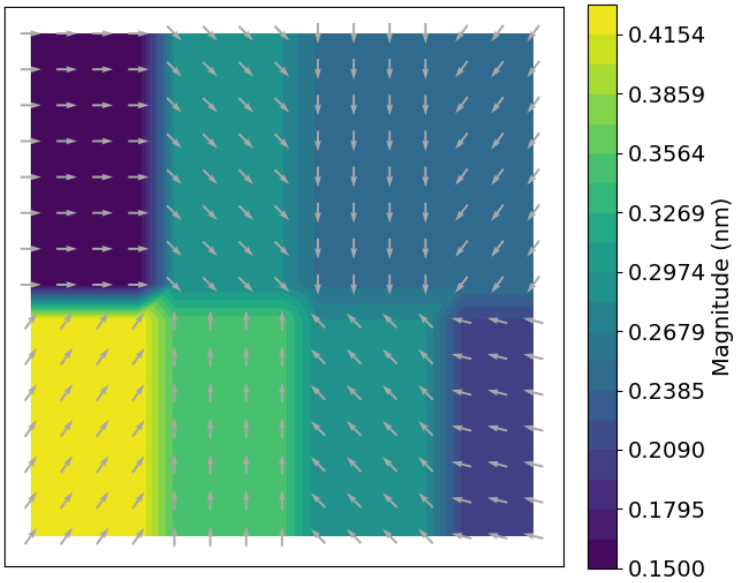
Change the vectors to unit vectors on a Matplotlib tricontourf map:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image, unit_vector=True,
```

(continues on next page)

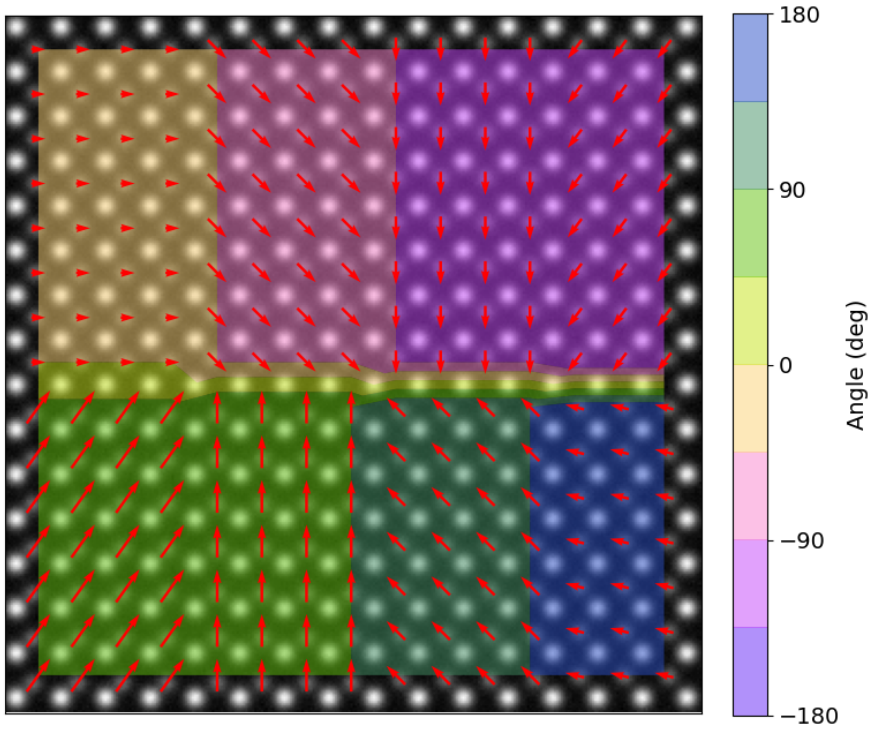
(continued from previous page)

```
...                             plot_style='contour', overlay=False,
...                             pivot='middle', save=None, monitor_dpi=50,
...                             color='darkgray', cmap='viridis')
```



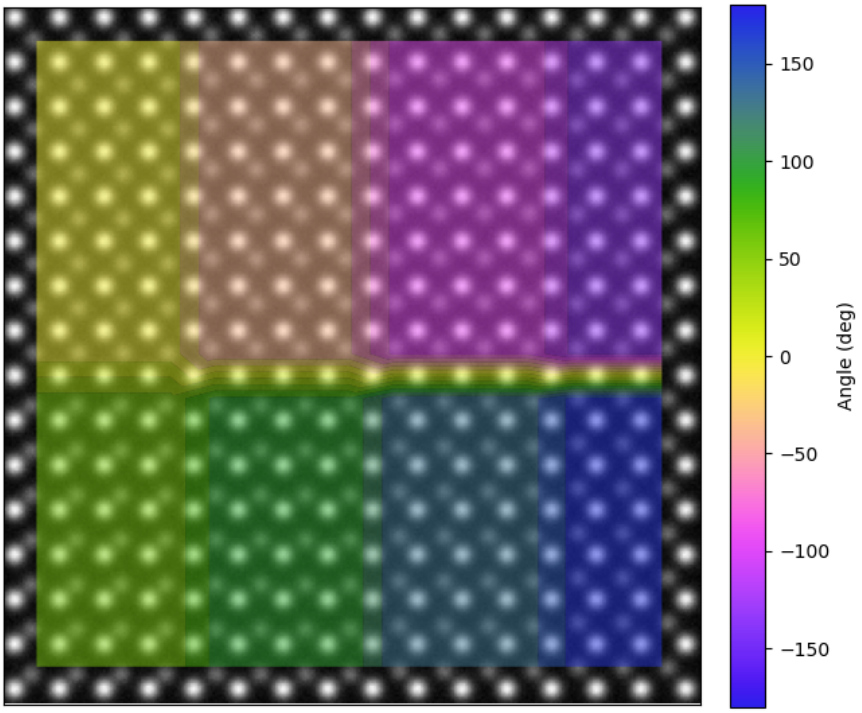
Plot a partly transparent angle tricontourf map with specified colorbar ticks and vector arrows:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                             unit_vector=False, plot_style='contour',
...                             overlay=True, pivot='middle', save=None,
...                             color='red', cmap='cet_colorwheel',
...                             monitor_dpi=50, remove_vectors=False,
...                             vector_rep="angle", alpha=0.5, levels=9,
...                             antialiased=True, degrees=True,
...                             ticks=[180, 90, 0, -90, -180])
```



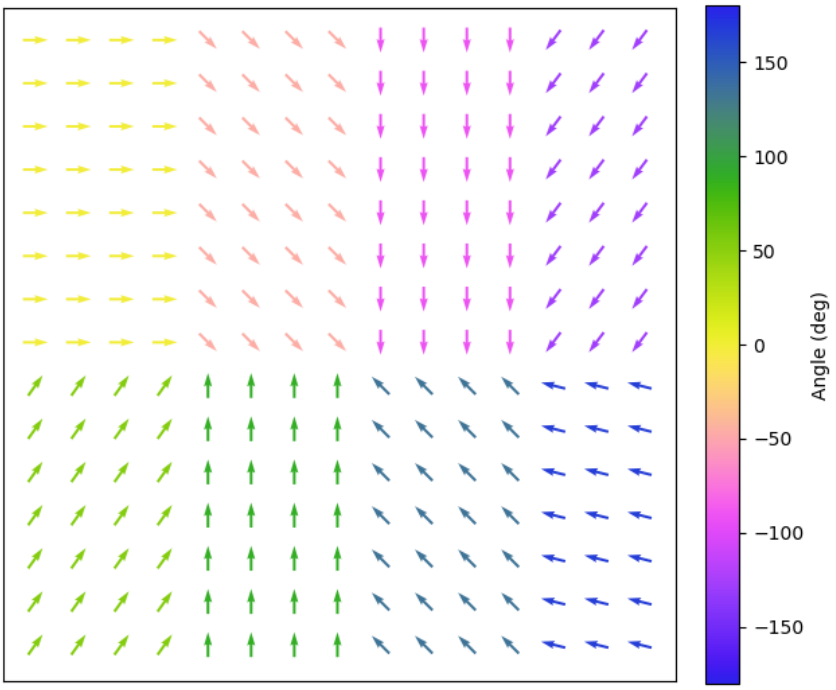
Plot a partly transparent angle tricontourf map with no vector arrows:

```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image, remove_vectors=True,
...                               unit_vector=True, plot_style='contour',
...                               overlay=True, pivot='middle', save=None,
...                               cmap='cet_colorwheel', alpha=0.5,
...                               monitor_dpi=50, vector_rep="angle",
...                               antialiased=True, degrees=True)
```



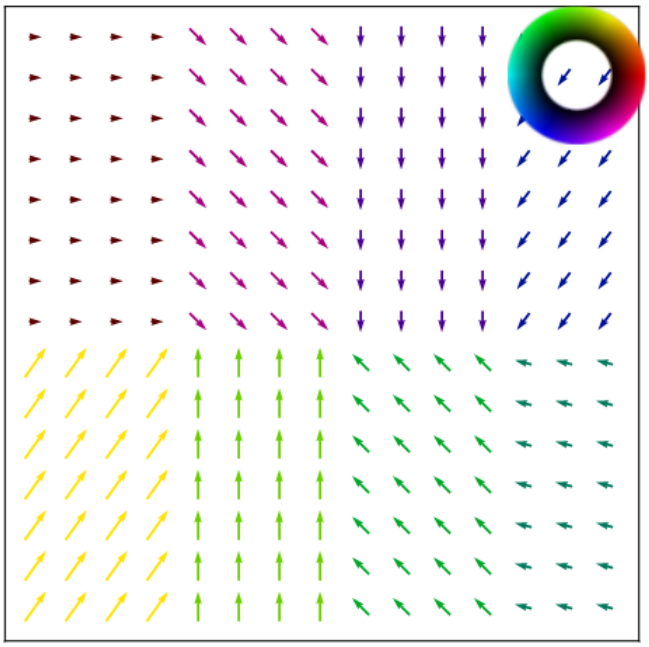
"colorwheel" plot of the vectors, useful for visualising vortexes:

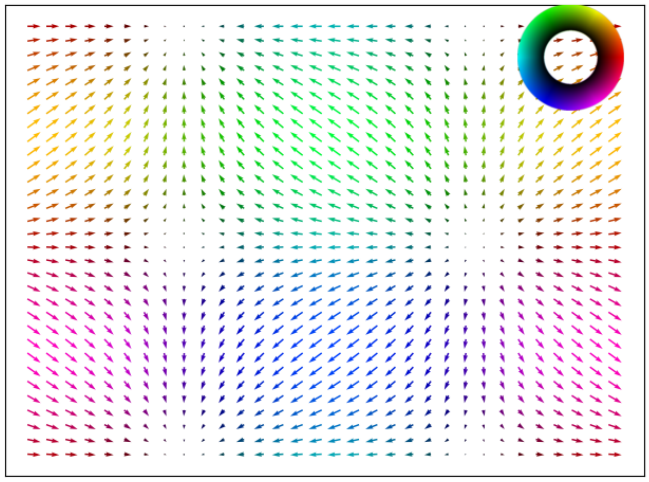
```
>>> import colorcet as cc  # can also just use cmap="cet_colorwheel"
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=True, plot_style="colorwheel",
...                               vector_rep="angle",
...                               overlay=False, cmap=cc.cm.colorwheel,
...                               degrees=True, save=None, monitor_dpi=50)
```



“polar_colorwheel” plot showing a 2D polar color wheel, also useful for vortexes:

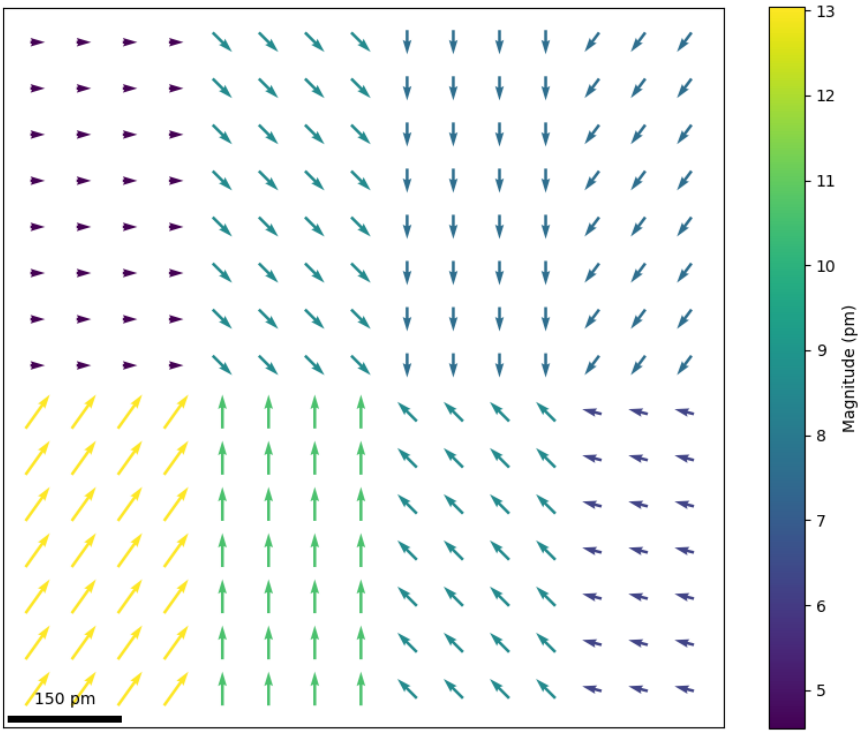
```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               plot_style="polar_colorwheel",
...                               unit_vector=False, overlay=False,
...                               save=None, monitor_dpi=50)

# This plot may show the effect of the second dimension more clearly.
# Example taken from Matplotlib's Quiver documentation.
>>> import numpy as np
>>> X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
>>> image_temp = np.ones_like(X)
>>> U = np.reshape(np.cos(X), 1024)
>>> V = np.reshape(np.sin(Y), 1024)
>>> X, Y = np.reshape(X, 1024), np.reshape(Y, 1024)
>>> ax = tml.plot_polarisation_vectors(X, Y, U, -V, image=image_temp,
...                               plot_style="polar_colorwheel",
...                               overlay=False, invert_y_axis=False,
...                               save=None, monitor_dpi=None)
>>> ax.invert_yaxis()
```





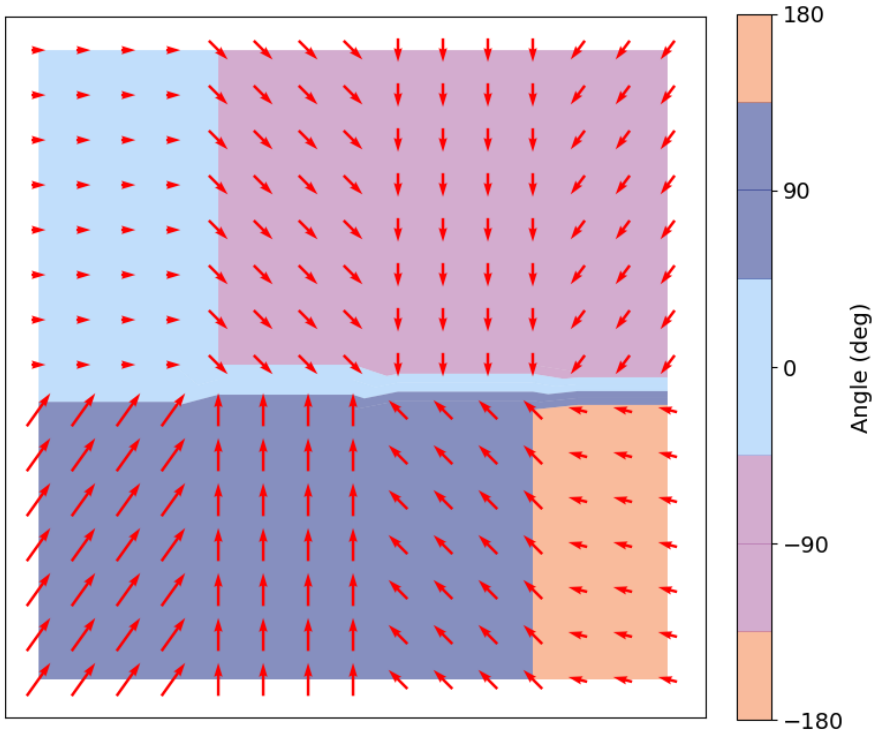
Plot with a custom scalebar. In this example, we need it to be dark, see matplotlib-scalebar for more custom features.

```
>>> scbar_dict = {"dx": 3.0321, "units": "pm", "location": "lower left",
...               "box_alpha":0.0, "color": "black", "scale_loc": "top"}
>>> tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                               sampling=3.0321, units='pm', monitor_dpi=50,
...                               unit_vector=False, plot_style='colormap',
...                               overlay=False, save=None, cmap='viridis',
...                               scalebar=scbar_dict)
```



Plot a tricontourf for quadrant visualisation using a custom matplotlib cmap:

```
>>> import temul.api as tml
>>> from matplotlib.colors import from_levels_and_colors
>>> zest = tml.hex_to_rgb(tml.color_palettes('zesty'))
>>> zest.append(zest[0])  # make the -180 and 180 degree colour the same
>>> expanded_zest = tml.expand_palette(zest, [1,2,2,2,1])
>>> custom_cmap, from_levels_and_colors(
...     levels=range(9), colors=tml.rgb_to_dec(expanded_zest))
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=False, plot_style='contour',
...                               overlay=False, pivot='middle', save=None,
...                               cmap=custom_cmap, levels=9, monitor_dpi=50,
...                               vector_rep="angle", alpha=0.5, color='r',
...                               antialiased=True, degrees=True,
...                               ticks=[180, 90, 0, -90, -180])
```

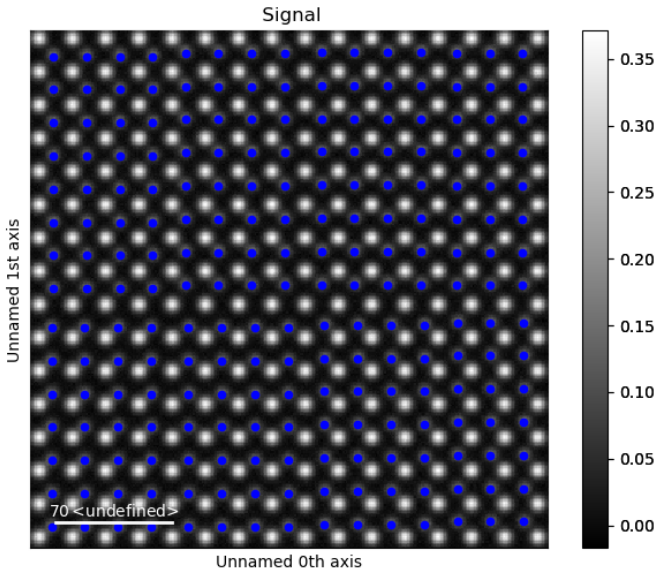


3.4.6 Plot Lattice Structure Maps

The *temul.topotem.polarisation* module allows one to easily visualise various lattice structure characteristics, such as strain, rotation of atoms along atom planes, and the *c/a* ratio in an atomic resolution image. In this tutorial, we will use a dummy dataset to show the different ways each map can be created. In future, tutorials on published experimental data will also be available.

Prepare and Plot the dummy dataset

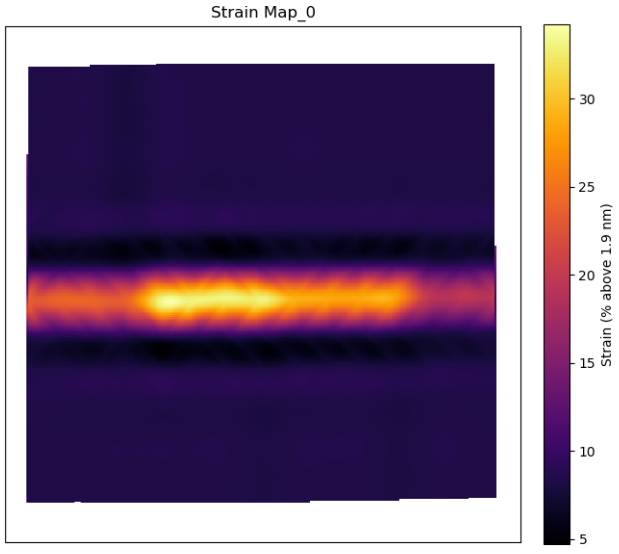
```
>>> import temul.api as tml
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset(image_noise=True)
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> sublatticeA.construct_zone_axes()
>>> sublatticeB.construct_zone_axes()
>>> sampling = 0.1  # example of 0.1 nm/pix
>>> units = 'nm'
>>> sublatticeB.plot()
```



Plot the Lattice Strain Map

By inputting the calculated or theoretical atom plane separation distance as the `theoretical_value` parameter in *`temul.topotem.polarisation.get_strain_map()`* below, we can plot a strain map. The distance l is calculated as the distance between each atom plane in the given zone axis. More details on this can be found on the Atomap website.

```
>>> theor_val = 1.9
>>> strain_map = tml.get_strain_map(sublatticeB, zone_axis_index=0,
...             units=units, sampling=sampling, theoretical_value=theor_val)
```



The outputted `strain_map` is a Hyperspy Signal2D. To learn more what can be done with Hyperspy, read their documentation!

Setting the `filename` parameter to any string will save the outputted plot and the .hspy signal (Hyperspy's hdf5 format).

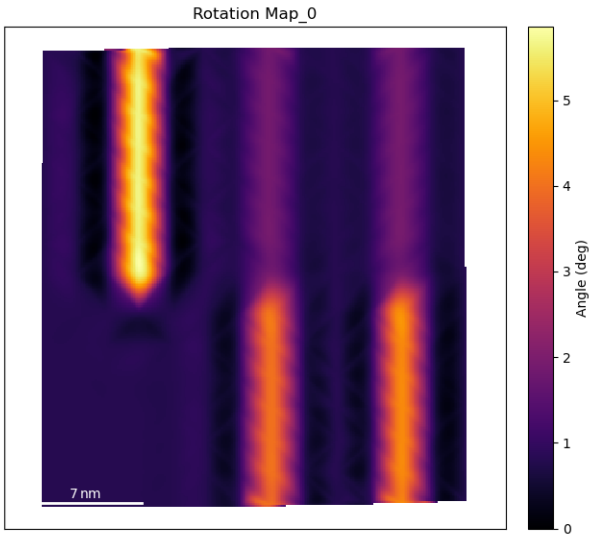
This applies to all structure maps discussed in this tutorial.

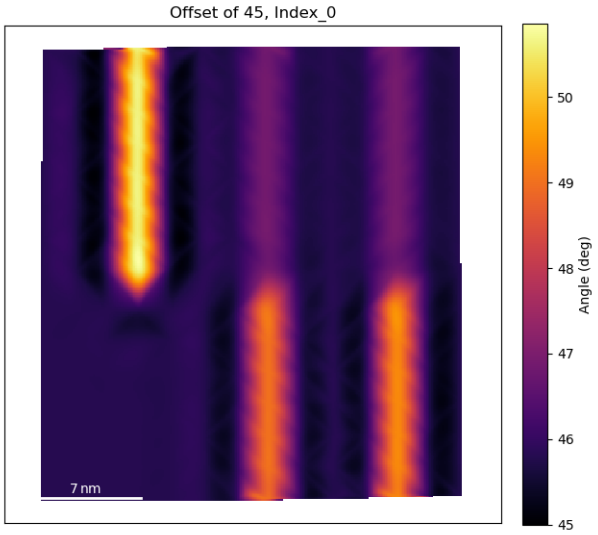
Setting `return_x_y_z=False` will return the strain map along with the x and y coordinates along with their corresponding strain values. One can then use these values externally, e.g., create a matplotlib tricontour plot). This applies to all structure maps discussed in this tutorial.

Plot the Lattice Atom Rotation Map

The *temul.topotem.polarisation.rotation_of_atom_planes()* function calculates the angle between successive atoms and the horizontal for all atoms in the given zone axis. See Atomap for other options.

```
>>> degrees=True
>>> rotation_map = tml.rotation_of_atom_planes(sublatticeB, 0,
...                     units=units, sampling=sampling, degrees=degrees)

'''
Use `angle_offset` to effectively change the angle of the horizontal axis
when calculating angles. Useful when the zone is not perfectly on the horizontal.
'''

>>> angle_offset = 45
>>> rotation_map = tml.rotation_of_atom_planes(sublatticeB, 0,
...                     units=units, sampling=sampling, degrees=degrees,
...                     angle_offset=angle_offset, title='Offset of 45, Index')
```

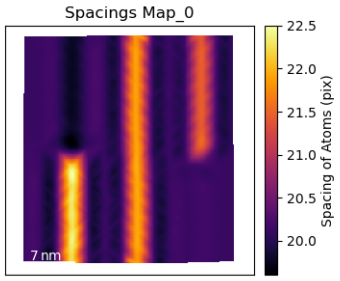


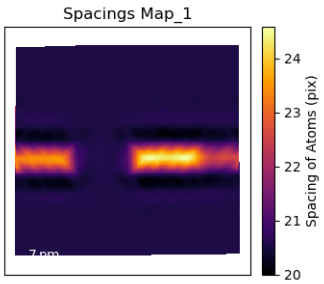


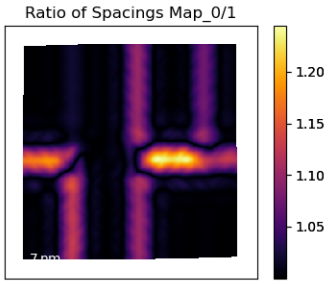
Plot the *c*/*a* Ratio

Using the *temul.topotem.polarisation.ratio_of_lattice_spacings()* function, we can visualise the ratio of two sublattice zone axes. Useful for plotting the *c*/*a* Ratio.

```
>>> ratio_map = tml.ratio_of_lattice_spacings(sublatticeB, 0, 1,
...                 units=units, sampling=sampling)
```







One can also use `ideal_ratio_one=False` to view the direction of tetragonality.

3.4.7 Calculation of Atom Plane Curvature

This tutorial follows the python scripts and jupyter notebooks found in the "publication_examples/PTO_supercrystal_hadjimichael" folder in the TEMUL repository. The data and scripts used below can be downloaded from there. You can also interact with the data without needing any downloads. Just click this button and navigate to that same folder, where you will find the python scripts and interactive python notebooks:

The *temul.topotem.lattice_structure_tools.calculate_atom_plane_curvature()* function has been adapted from the MATLAB script written by Dr. Marios Hadjimichael for the publication M. Hadjimichael, Y. Li *et al*, Metal-ferroelectric supercrystals with periodically curved metallic layers, Nature Materials 2020. This MATLAB script can also be found in the same folder.

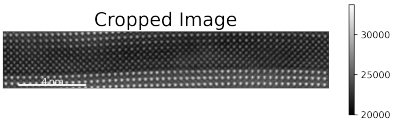
The *temul.topotem.lattice_structure_tools.calculate_atom_plane_curvature()* function in the *temul.topotem.lattice_structure_tools* module can be used to find the curvature of the displacement of atoms along an atom plane in a sublattice. Using the default parameter `func='strain_grad'`, the function will approximate the curvature as the strain gradient, as in cases where the first derivative is negligible. See "Landau and Lifshitz, Theory of Elasticity, Vol 7, pp 47-49, 1981" for more details. One can use any `func` input that can be used by `scipy.optimize.curve_fit`.

Import the Modules and Load the Data

```
>>> import temul.api as tml
>>> import atomap.api as am
>>> import hyperspy.api as hs
>>> import os
>>> path_to_data = os.path.join(os.path.abspath(''),
...             "publication_examples/PTO_supercrystal_hadjimichael/data")
>>> os.chdir(path_to_data)
```

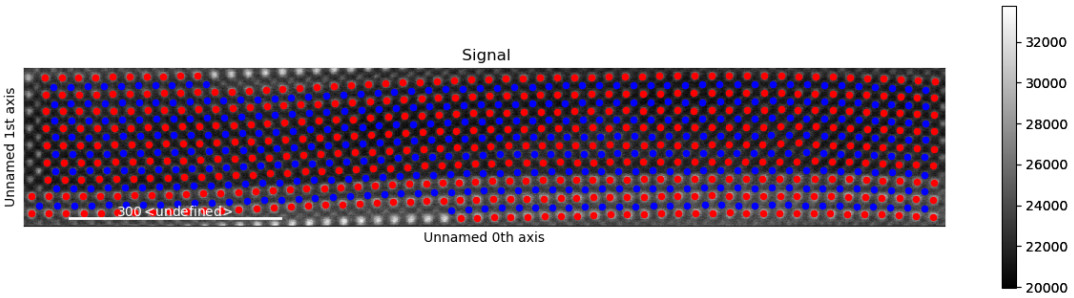
Open the PTO/SRO dataset

```
>>> image = hs.load('Cropped_PTO-SRO_Aligned.hspy')
>>> sampling = image.axes_manager[-1].scale #  nm/pix
>>> units = image.axes_manager[-1].units
>>> image.plot()
```



Open the pre-made PTO-SRO atom lattice.

```
>>> atom_lattice = am.load_atom_lattice_from_hdf5("Atom_Lattice_crop.hdf5")
>>> sublattice1 = atom_lattice.sublattice_list[0] #  Pb-Sr Sublattice
>>> sublattice2 = atom_lattice.sublattice_list[1] #  Ti-Ru Sublattice
>>> atom_lattice.plot()
```



Set up the Parameters

Plot the sublattice planes to see which `zone_vector_index` we use

```
>>> sublattice2.construct_zone_axes(atom_plane_tolerance=1)
>>> # sublattice2.plot_planes()
```

Set up parameters for `calculate_atom_plane_curvature`

```
>>> zone_vector_index = 0
>>> atom_planes = (2, 6) #  chooses the starting and ending atom planes
>>> vmin, vmax = 1, 2
>>> cmap = 'bwr' #  see matplotlib and colorcet for more colormaps
>>> title = 'Curvature Map'
>>> filename = None #  Set to a string if you want to save the map
```

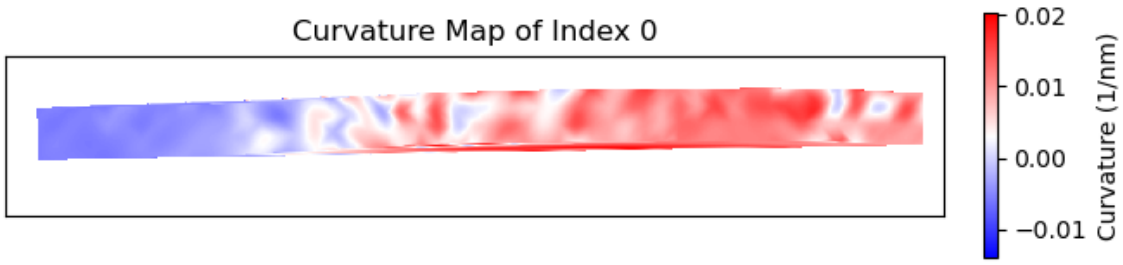
Set the extra initial fitting parameters

```
>>> p0 = [14, 10, 24, 173]
>>> kwargs = {'p0': p0, 'maxfev': 1000}
```

Calculate the Curvature of Atom Planes

We want to see the curvature in the SRO Sublattice

```
>>> curvature_map = tml.calculate_atom_plane_curvature(sublattice2, zone_vector_index,
...                     sampling=sampling, units=units, cmap=cmap, title=title,
...                     atom_planes=atom_planes, **kwargs)
```



When using `plot_and_return_fits=True`, the function will return the curve fittings, and plot each plane (plots not displayed).

```
>>> curvature_map, fittings = tml.calculate_atom_plane_curvature(sublattice2,
...                     zone_vector_index, sampling=sampling, units=units,
...                     cmap=cmap, title=title, atom_planes=atom_planes, **kwargs,
...                     plot_and_return_fits=True)
```

3.4.8 Analysis of PTO Domain Wall Junction

This tutorial follows the python scripts and jupyter notebooks found in the "TEMUL/publication_examples/PTO_Junction_moore" folder in the TEMUL repository. The data and scripts used below can be downloaded from there. Check out the publication: K. Moore *et al* Highly charged 180 degree head-to-head domain walls in lead titanate, Nature Communications Physics 2020.

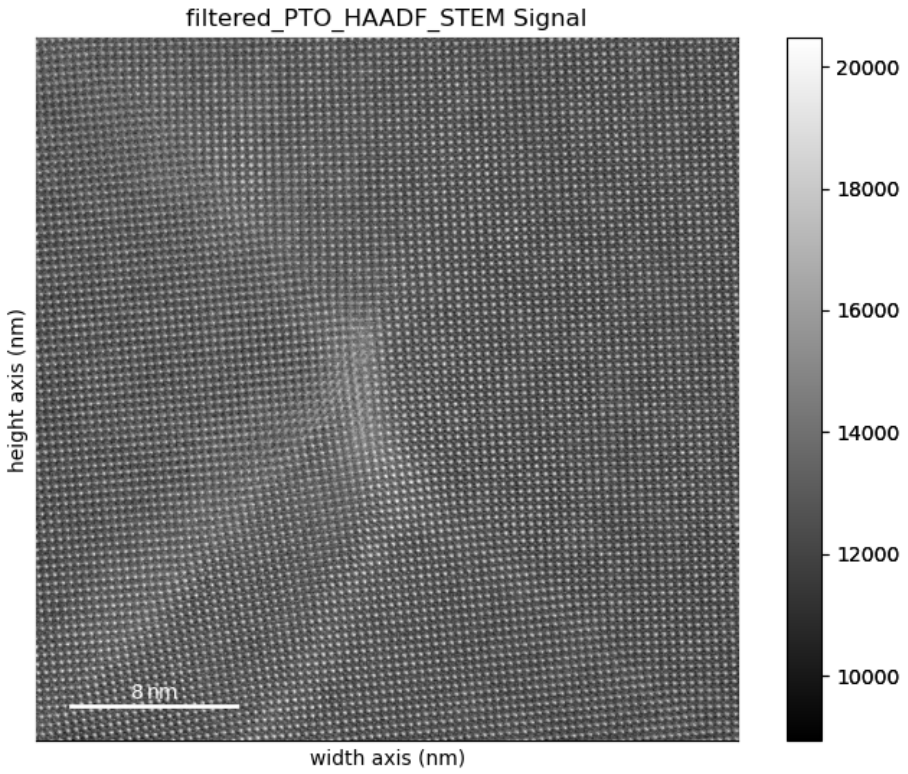
Use the notebook interactively now with MyBinder, just click the launch button below (There are some memory errors at the moment with the large .hdf5 files. It may work better if you run a Jupyter Notebook locally).

Import the Modules and Load the Data

```
>>> import temul.api as tml
>>> import atomap.api as am
>>> import hyperspy.api as hs
>>> import numpy as np
>>> import os
>>> path_to_data = os.path.join(os.path.abspath(''), "publication_examples/PTO_Junction_
↪moore/data")
>>> os.chdir(path_to_data)
```

Open the filtered PTO Junction dataset

```
>>> image = hs.load('filtered_PTO_HAADF_STEM.hspy')
>>> sampling = image.axes_manager[-1].scale  # nm/pix
>>> units = image.axes_manager[-1].units
>>> image.plot()
```



Open the pre-made PTO-SRO atom lattice.

```
>>> atom_lattice = am.load_atom_lattice_from_hdf5("Atom_Lattice_crop.hdf5", False)
>>> sublattice1 = atom_lattice.sublattice_list[0]  # Pb Sublattice
>>> sublattice2 = atom_lattice.sublattice_list[1]  # Ti Sublattice
>>> sublattice1.construct_zone_axes(atom_plane_tolerance=1)
```

Set up the Parameters

Set up parameters for plotting the strain, rotation, and c/a ratio maps: Note that sometimes the 0 and 1 axes are constructed first or second, so you may have to swap them.

```
>>> zone_vector_index_A = 0
>>> zone_vector_index_B = 1
>>> filename = None  # Set to a string if you want to save the map
```

Note: You can use `return_x_y_z=True` for each of the map functions below to get the raw x,y, and strain/rotation/ratio values for further plotting with matplotlib! Check the documentation

Load the line profile positions:

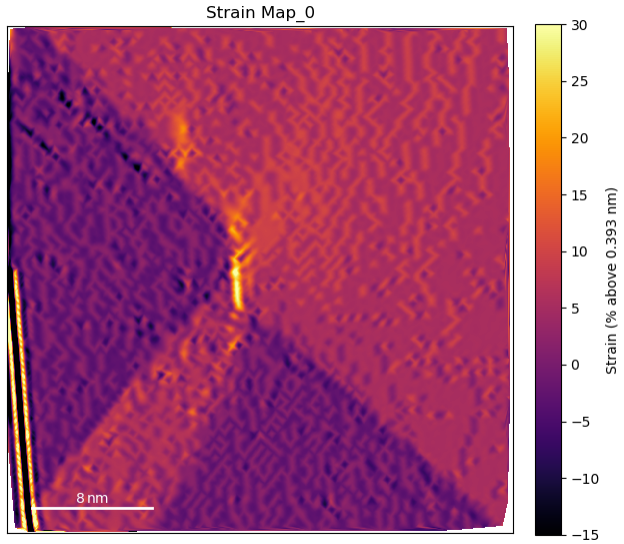
```
>>> line_profile_positions = np.load('line_profile_positions.npy')
```

Note: You can also choose your own line_profile_positions with *temul.topotem.fft_mapping.choose_points_on_image()* and use the `skimage.profile_line` for customisability.

Create the Lattice Strain Map

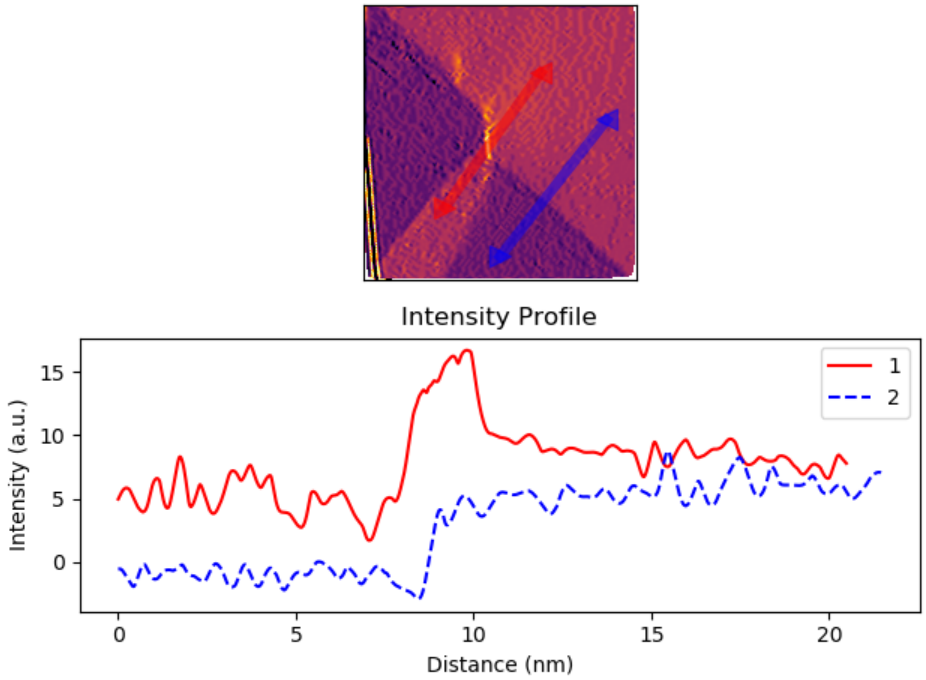
We want to see the strain map of the Pb Sublattice in the y-axis direction Note that sometimes the 0 and 1 axes directions are constructed vice versa.

```
>>> vmin = -15
>>> vmax = 30
>>> cmap = 'inferno'
>>> theoretical_value = round(3.929/10, 3)  # units of nm
>>> strain_map = tml.get_strain_map(sublattice1, zone_vector_index_B,
...                                 theoretical_value, sampling=sampling,
...                                 units=units, vmin=vmin, vmax=vmax, cmap=cmap)
```



Plot the line profiles with *temul.signal_plotting* functions and a kwarg dictionary. For more details on this function, see *this tutorial*.

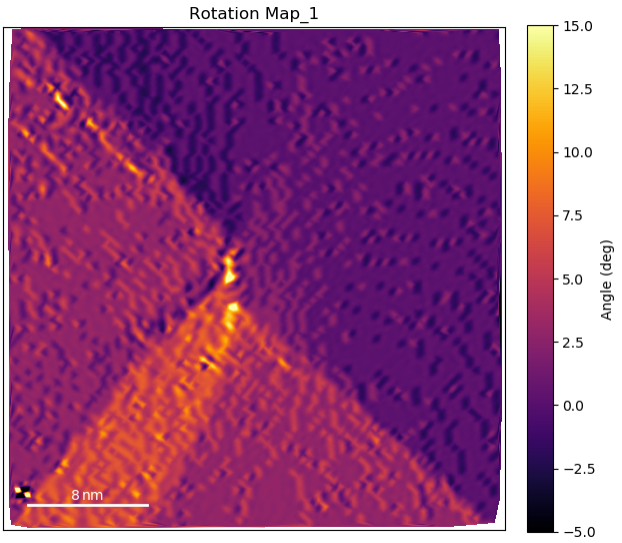
```
>>> kwargs = {'vmin': vmin, 'vmax': vmax, 'cmap': cmap}
>>> tml.compare_images_line_profile_one_image(strain_map, line_profile_positions,
...                                           linewidth=100, arrow='h', linetrace=0.05,
...                                           **kwargs)
```



Create the Lattice Rotation Map

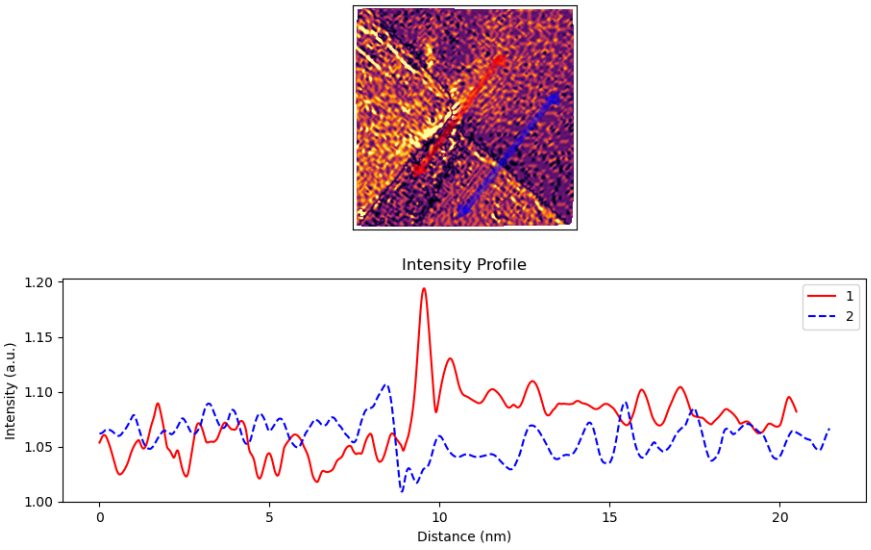
Now plot the rotation map of the Pb Sublattice in the x-axis direction to see the turning of the lattice across the junction.

```
>>> vmin = -5
>>> vmax = 15
>>> cmap = 'inferno'
>>> angle_offset = -2  # degrees
>>> rotation_map = tml.rotation_of_atom_planes(
...                     sublattice1, zone_vector_index_A, units=units,
...                     angle_offset, degrees=True, sampling=sampling,
...                     vmin=vmin, vmax=vmax, cmap=cmap)
```



Plot the line profiles with *temul.signal_plotting* functions and a kwarg dictionary. For more details on this function, see *this tutorial*.

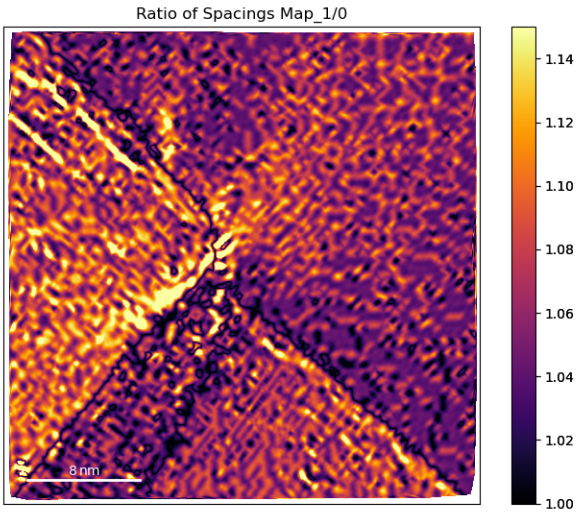
```
>>> kwargs = {'vmin': vmin, 'vmax': vmax, 'cmap': cmap}
>>> tml.compare_images_line_profile_one_image(
...     rotation_map, line_profile_positions, linewidth=100, arrow='h',
...     linetrace=0.05, **kwargs)
```



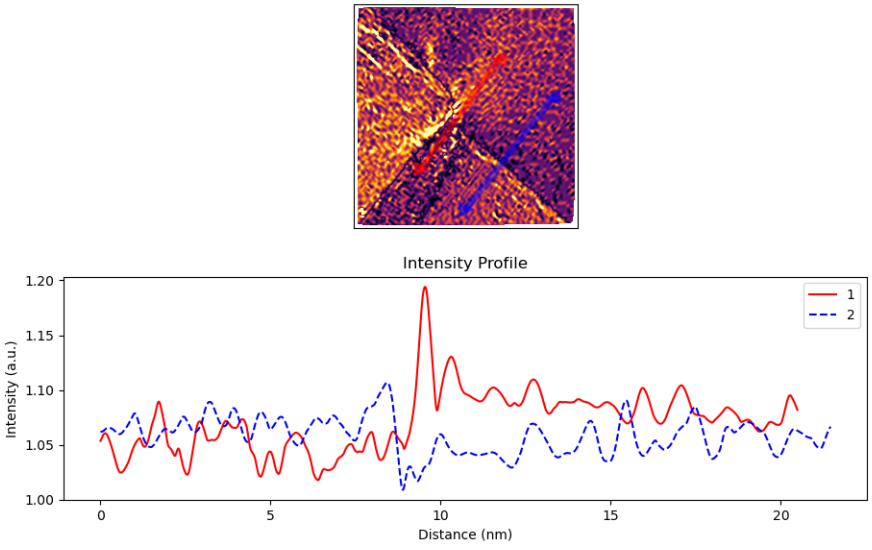
Create the Lattice c/a Ratio Map

Now plot the c/a ratio map of the Pb Sublattice

```
>>> vmin = 1
>>> vmax = 1.15
>>> cmap = 'inferno'
>>> ideal_ratio_one = True  # values under 1 will be divided by themselves
>>> ca_ratio_map = tml.ratio_of_lattice_spacings(
...                     sublattice1, zone_vector_index_B,
...                     zone_vector_index_A, ideal_ratio_one, sampling=sampling,
...                     units=units, vmin=vmin, vmax=vmax, cmap=cmap)
```




```
>>> kwargs = {'vmin': vmin, 'vmax': vmax, 'cmap': cmap}
>>> tml.compare_images_line_profile_one_image(
...     ca_ratio_map, line_profile_positions, linewidth=100, arrow='h',
...     linetrace=0.05, **kwargs)
```



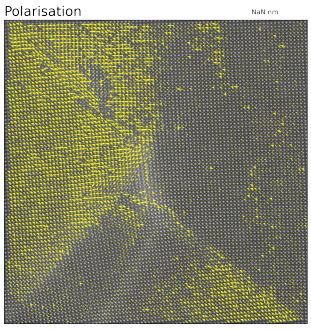
Map the Polarisation

In this case, the PTO structure near the junction is highly strained. Therefore, we can't use the the Atomap get_polarization_from_second_sublattice function.

```
>>> atom_positions_actual = np.array(
...     [sublattice2.x_position, sublattice2.y_position]).T
>>> atom_positions_ideal = np.load('atom_positions_orig_2.npy')
>>> u, v = tml.find_polarisation_vectors(
...     atom_positions_actual, atom_positions_ideal)
>>> x, y = atom_positions_actual.T.tolist()
```

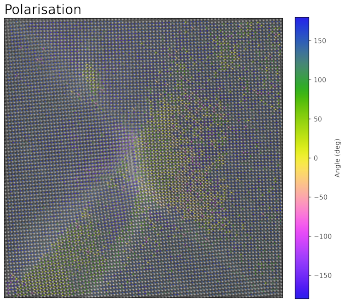
Plot the polarisation vectors (zoom in to get a better look, the top left is off zone).

```
>>> tml.plot_polarisation_vectors(
...     x=x, y=y, u=u, v=v, image=image.data,
...     sampling=sampling, units=units, unit_vector=False, overlay=True,
...     color='yellow', plot_style='vector', title='Polarisation',
...     monitor_dpi=250, save=None)
```



Plot the angle information as a colorwheel

```
>>> plt.style.use("grayscale")
>>> tml.plot_polarisation_vectors(
...     x=x, y=y, u=u, v=v, image=image.data, save=None,
...     sampling=sampling, units=units, unit_vector=True, overlay=True,
...     color='yellow', plot_style='colorwheel', title='Polarisation',
...     monitor_dpi=250, vector_rep='angle', degrees=True)
```

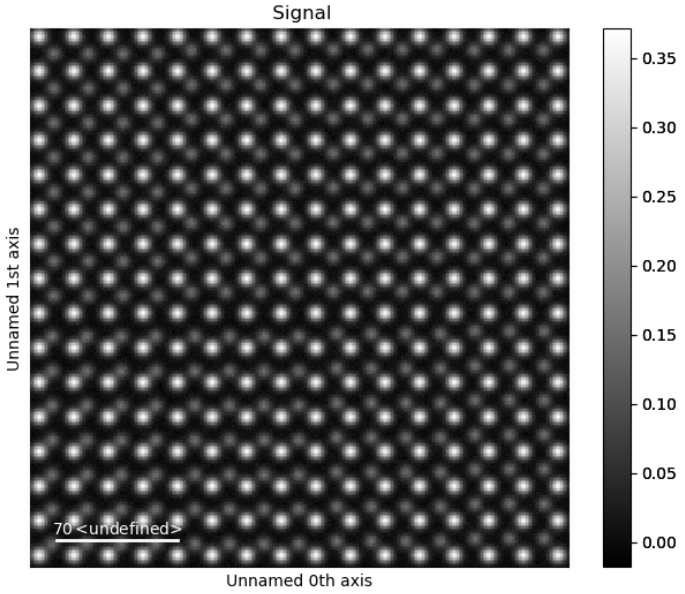


3.4.9 Masked FFT and iFFT

The *temul.signal_processing* module allows one to choose the masking coordinates with *temul.topotem.fft_mapping.choose_mask_coordinates()* and easily return the masked fast Fourier Transform (FFT) with *temul.topotem.fft_mapping.get_masked_ifft()*. This can useful in various scenarios, from understanding the diffraction space spots and how they relate to the real space structure, to revealing domain walls and finding initial atom positions for difficult images.

Load the Example Image

```
>>> import temul.api as tml
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset(image_noise=True)
>>> image = atom_lattice.sublattice_list[0].signal
>>> image.plot()
```

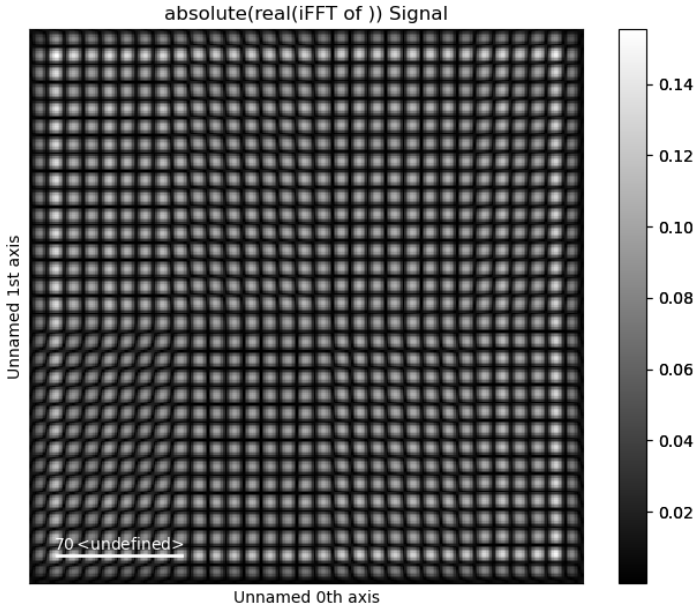


Choose the Mask Coordinates

```
>>> mask_coords = tml.choose_mask_coordinates(image, norm="log")
```

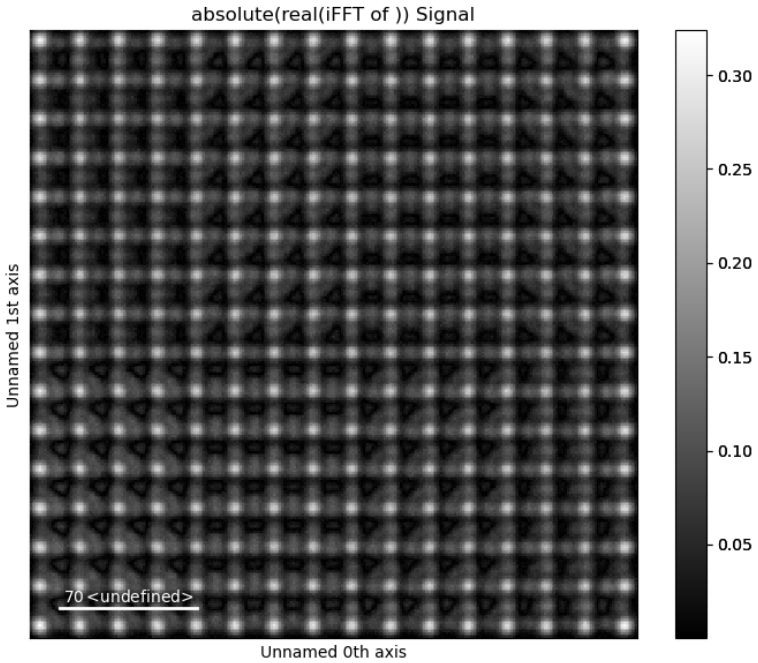
Plot the Masked iFFT

```
>>> mask_radius = 10  # pixels, default is also 10 pixels
>>> image_ifft = tml.get_masked_ifft(image, mask_coords,
...                                  mask_radius=mask_radius)
>>> image_ifft.plot()
```



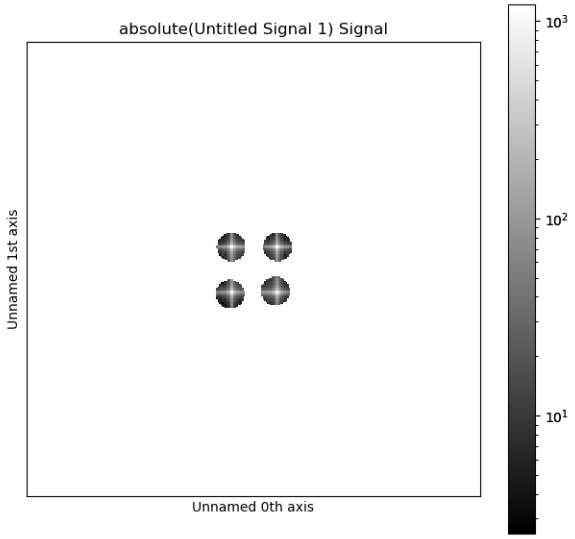
Reverse the masking with *keep_masked_area=False*

```
>>> image_ifft = tml.get_masked_ifft(image, mask_coords,
...                                  keep_masked_area=False)
>>> image_ifft.plot()
```



Plot the FFT with masks overlaid by using *plot_masked_fft=True*

```
>>> image_ifft = tml.get_masked_ifft(image, mask_coords,
...                                  plot_masked_fft=True)
```



If the input image is already a Fourier transform

```
>>> fft_image = image.fft(shift=True)  # Check out Hyperspy
>>> image_ifft = tml.get_masked_ifft(fft_image, mask_coords,
...                                  image_space='fourier')
```

Run FFT masking for Multiple Images

If you have multiple images, you can easily apply the mask to them all in a simple `for` loop. Of course, you can also save the images after plotting.

```
>>> from hyperspy.io import load
>>> for file in files:
...     image = load(file)
...     image_ifft = tml.get_masked_ifft(image, mask_coords)
...     image_ifft.plot()
```

3.4.10 Line Intensity Profile Comparisons

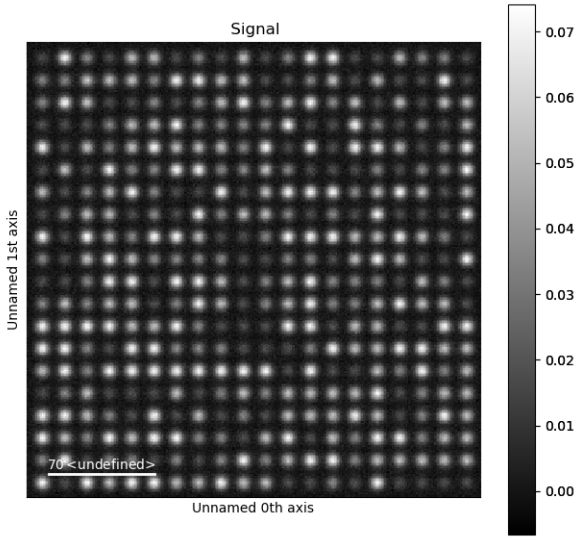
The *temul.signal_plotting* module allows one to draw line intensity profiles over images. The :py:func`temul.signal_plotting.compare_images_line_profile_one_image` can be used to draw two line profiles on one image for comparison. In future we hope to expand this function to allow for multiple line profiles on one image. The *temul.signal_plotting.compare_images_line_profile_two_images()* function allows you to draw a line profile on an image, and apply that same profile to another image (of the same shape). This can be useful for comparing subsequent images in series or comparing experimental and simulated images.

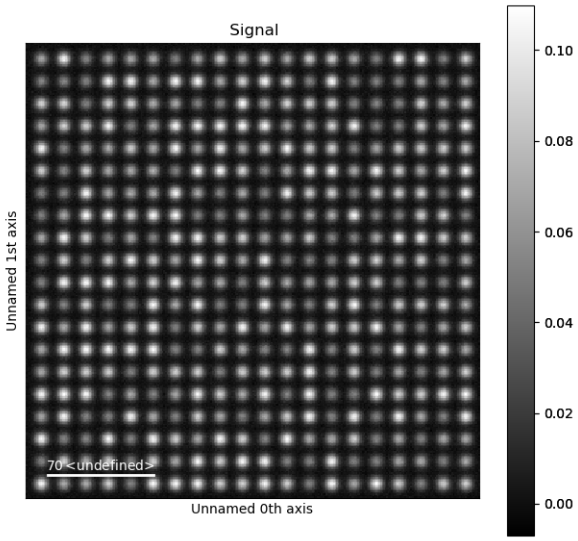
Check out the examples below for each comparison method.

Load the Example Images

Here we load some dummy data using a variation of Atomap's *temul.dummy_data.get_simple_cubic_signal()* function.

```
>>> from temul.dummy_data import get_simple_cubic_signal
>>> imageA = get_simple_cubic_signal(image_noise=True, amplitude=[1, 5])
>>> imageA.plot()
>>> imageB = get_simple_cubic_signal(image_noise=True, amplitude=[3, 7])
>>> imageB.plot()
>>> sampling, units = 0.1, 'nm'
```





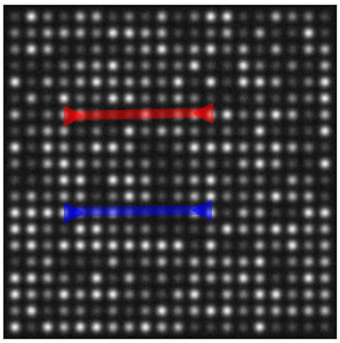
Compare two Line Profiles in one Image

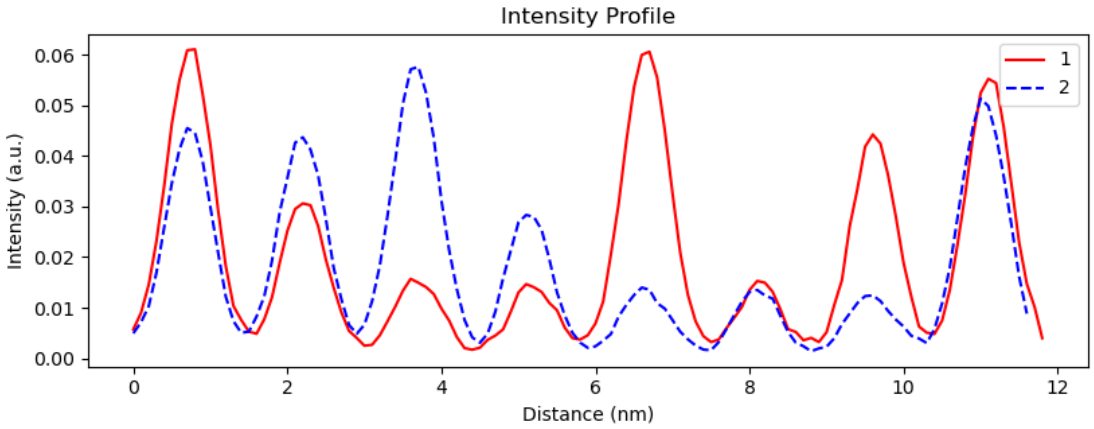
As with the *Masked FFT and iFFT* tutorial, we can choose points on the image. This time we use *temul.topotem.fft_mapping.choose_points_on_image()*. We need to choose four points for the *temul.signal_plotting.compare_images_line_profile_one_image()* function, as it draws two line profiles over one image.

```
>>> import temul.api as tml
>>> line_profile_positions = tml.choose_points_on_image(imageA)
>>> line_profile_positions
[[61.75132848177407, 99.25182885155715],
 [178.97030854763057, 96.60281235289372],
 [61.75132848177407, 186.0071191827843],
 [177.64580029829887, 184.6826109334526]]
```

Now run the comparison function to display the two line intensity profiles.

```
>>> tml.compare_images_line_profile_one_image(
...             imageA, line_profile_positions, linewidth=5,
...             sampling=sampling, units=units, arrow='h', linetrace=1)
```



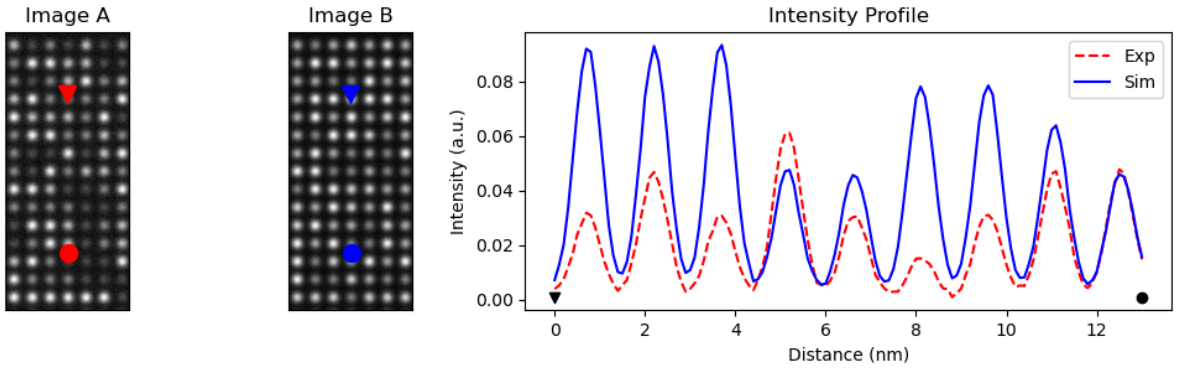


Compare two Images with Line Profile

Using *temul.topotem.fft_mapping.choose_points_on_image()*, we now choose two points on one image. Then, we plot this line intensity profile over the same position in two images.

```
>>> line_profile_positions = tml.choose_points_on_image(imageA)
>>> line_profile_positions
[[127.31448682369383, 46.93375300295452],
 [127.97674094835968, 176.7355614374623]]
```

```
>>> import numpy as np
>>> tml.compare_images_line_profile_two_images(imageA, imageB,
...             line_profile_positions, linewidth=5, reduce_func=np.mean,
...             sampling=sampling, units=units, crop_offset=50,
...             imageA_title="Image A", imageB_title="Image B")
```



3.4.11 Interactive Image Filtering

The *temul.signal_processing* module allows one to filter images with a double Gaussian (band-pass) filter. Apart from the base functions, it can be done interactively with the *temul.signal_processing.visualise_dg_filter()* function. In this tutorial, we will see how to use the function on experimental data.

Load the Experimental Image

Here we load an example experimental atomic resolution image stored in the TEMUL package.

```
>>> import temul.api as tml
>>> from temul.example_data import load_Se_implanted_MoS2_data
>>> image = load_Se_implanted_MoS2_data()
```

Interactively Filter the Experimental Image

Run the *temul.signal_processing.visualise_dg_filter()* function. There are lots of other parameters too for customisation.

```
>>> tml.visualise_dg_filter(image)
```

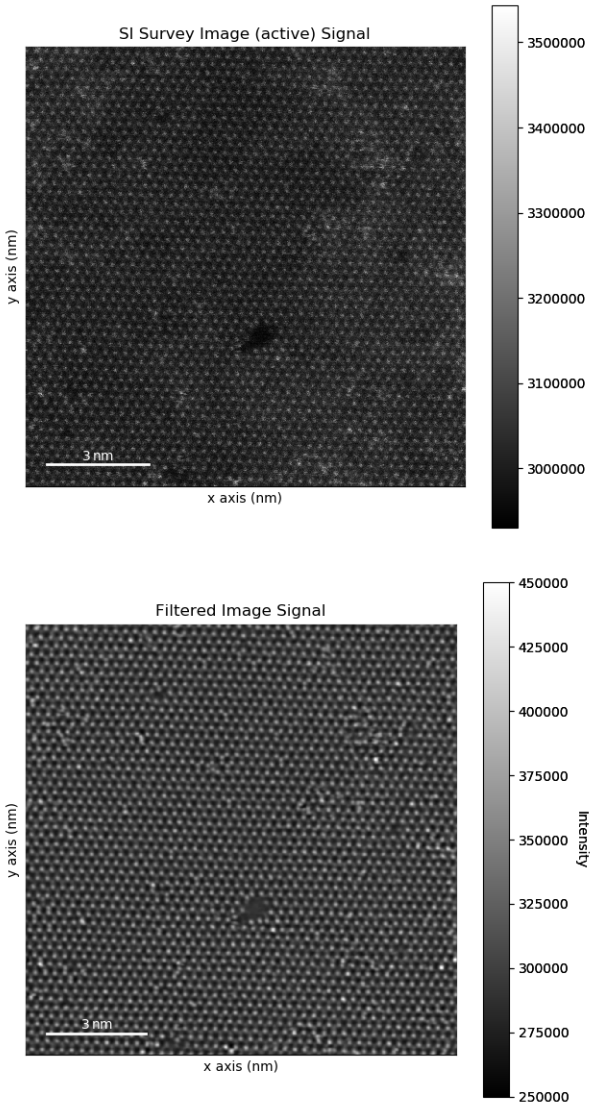
As we can see, an interactive window appears, showing the FFT ("FFT Widget") of the image with the positions of the inner and outer Gaussian full width at half maximums (FWHMs). The inital FWHMs can be changed with the `d_inner` and `d_outer` parameters (limits can also be changed).

To change the two FWHMs interactively just use the sliders at the bottom of the window. Reset can be used to reset the FWHMs to their initial value, and Filter will display the "Convolution" of the FFT and double Gaussian filter as well as the inverse FFT ("Filtered Image") of this convolution.

Return the Filtered Image

When we have suitable FWHMs for the inner and outer Gaussians, we can use the *temul.signal_processing.double_gaussian_fft_filter()* function the return a filtered image.

```
>>> filtered_image = tml.double_gaussian_fft_filter(image, 50, 150)
>>> image.plot()
>>> filtered_image.plot()
```



Details on the *temul.signal_processing.double_gaussian_fft_filter_optimised()* function can be found in the *API documentation*.

3.4.12 API documentation

Classes

Model Refiner

Modules

TopoTEM (Polarisation)

temul.topotem.polarisation.**angle_label**(*vector_rep='magnitude'*, *units='pix'*, *degrees=False*)

temul.topotem.polarisation.**atom_deviation_from_straight_line_fit**(*sublattice*, *axis_number*, *n*, *second_fit_rigid=True*, *plot=False*, *return_individual_atom_planes=False*)

Fits the atomic columns in an atom plane to two straight lines using the first `n` and second (last) `n` atomic columns. See Notes.

The slope of the first fitting will be used for the second fitting. Setting `second_fit_rigid` = False will reverse this behaviour.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **axis_number** (*int*) – The index of the zone axis (translation symmetry) found by the Atomap function `construct_zone_axes()`. For sublattices with heavily deviating atomic columns, you may need to use sublattice.construct_zone_axes(atom_plane_tolerance=1).
- **n** (*int*) – The number of columns used at the beginning and end of each atom plane to fit a straight line.
- **second_fit_rigid** (*bool, default True*) – Used to decide whether the first or second fitting's slope will be rigid during fitting. With `second_fit_rigid=True`, the slope of the second fitting will be defined as the slope as the first fitting. The y-intercept is free to move.
- **plot** (*bool, default False*) – Whether to plot the atom plane data, first and second fitting, and the line constructed halfway between the two.
- **return_individual_atom_planes** (*bool*) – If set to True, the returned lists will be lists of sublists. These sublists have information from each atom plane. For example, the returned *x_list* will have sublists which contain the *x* values from each atom plane. If set to False (default), the list of sublists is flattened.

Returns **x, y, u, v** – x, y are the original atom position coordinates `sublattice.x_position` and `sublattice.y_position` for the coordinates included in the chosen `axis_number`. u, v are the polarisation vector components pointing towards the halfway line from the atom position coordinates. These can be input to `plot_polarisation_vectors()` for visualisation. If `return_individual_atom_planes` is True, then the four lists will be made up of a sublists. See `return_individual_atom_planes` for more information.

Return type lists of equal length

See also:

get_xyuv_from_line_fit uses `_fit_line_clusters` to get xyuv from arr

Notes

Computes the distance of each atomic column from the line halfway between the two fitted lines, as described in [1]_. This is done for every sublattice atom plane along the chosen `axis_number`.

References

Examples

```
>>> import temul.api as tml
>>> import temul.dummy_data as dd
>>> sublattice = dd.get_polarised_single_sublattice()
>>> sublattice.construct_zone_axes(atom_plane_tolerance=1)
```

Choose `n`: how many atom columns should be used to fit the line on each side of the atom planes. If ``n is too large, the fitting will appear incorrect.

```
>>> n = 5
>>> x, y, u, v = tml.atom_deviation_from_straight_line_fit(
...     sublattice, 0, n)
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublattice.image,
...                                    unit_vector=False, save=None,
...                                    plot_style='vector', color='r',
...                                    overlay=True, monitor_dpi=50)
```

Plot with angle and up/down. Note that the data ranges from -90 to +90 degrees, so the appropriate diverging cmap should be chosen.

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublattice.image,
...                         vector_rep='angle', save=None, degrees=True,
...                         plot_style='colormap', cmap='cet_coolwarm',
...                         overlay=True, monitor_dpi=50)
```

Get each atomic column in a list of sublists

```
>>> return_individual_atom_planes = True
>>> x, y, u, v = tml.atom_deviation_from_straight_line_fit(
...     sublattice, axis_number=0, n=5, second_fit_rigid=True, plot=False,
...     return_individual_atom_planes=return_individual_atom_planes)
```

look at first atomic plane rumpling

```
>>> _ = plt.figure()
>>> _ = plt.plot(range(len(v[0])), v[0], 'ro')
>>> _ = plt.title("First atomic plane v (vertical) rumpling.")
>>> _ = plt.xlabel("Atomic plane #")
>>> _ = plt.ylabel("Vertical deviation (rumpling) (a.u.)")
>>> _ = plt.close()
```

could also use numpy array to handle the data

```
>>> arr = np.array([x, y, u, v]).T
```

Let's look at some rotated data

```
>>> sublattice = dd.get_polarised_single_sublattice_rotated(
...     image_noise=True, rotation=45)
>>> sublattice.construct_zone_axes(atom_plane_tolerance=0.9)
>>> # sublattice.plot_planes()
>>> n = 3  # plot the sublattice to see why 3 is suitable here!
>>> x, y, u, v = tml.atom_deviation_from_straight_line_fit(
...     sublattice, 0, n)
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublattice.image,
...                         vector_rep='angle', save=None, degrees=True,
...                         plot_style='colormap', cmap='cet_coolwarm',
...                         overlay=True, monitor_dpi=50)
```

temul.topotem.polarisation.**atom_to_atom_distance_grouped_mean**(*sublattice*, *zone_axis_index*, *aggregation_axis='y'*, *slice_thickness=10*, *sampling=None*, *units='pix'*)

Average the atom to atom distances along the chosen zone_axis_index parallel to the chosen axis ('x' or 'y').

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **zone_axis_index** (*int*) – The zone axes you wish to average along.
- **aggregation_axis** (*string, default "y"*) – Axis parallel to which the atom to atom distances will be averaged.
- **slice_thickness** (*float, default 10*) – thickness of the slices used for aggregation.
- **sampling** (*float, default None*) – Pixel sampling of the image for calibration.
- **units** (*string, default "pix"*) – Units of the sampling.

Returns

- *Slice thickness groupings and means of each grouping. Groupings can be*
- *thought of as bins in a histogram of grouped means.*

Example

```
>>> import numpy as np
>>> from atomap.dummy_data import get_distorted_cubic_sublattice
>>> import matplotlib.pyplot as plt
>>> from temul.topotem.polarisation import (
...     atom_to_atom_distance_grouped_mean)
>>> sublatticeA = get_distorted_cubic_sublattice()
>>> sublatticeA.construct_zone_axes(atom_plane_tolerance=1)
>>> sublatticeA.plot()
>>> groupings, grouped_means = atom_to_atom_distance_grouped_mean(
...     sublatticeA, 0, 'y', 40)
```

You can then plot these as below: plt.figure() plt.plot(groupings, grouped_means, 'k.') plt.show()

Average parallel to the x axis instead:

```
>>> groupings, grouped_means = atom_to_atom_distance_grouped_mean(
...     sublatticeA, 0, 'x', 40)
```

You can then plot these as below: plt.figure() plt.plot(groupings, grouped_means, 'k.') plt.show()

temul.topotem.polarisation.**combine_atom_deviations_from_zone_axes**(*sublattice*, *image=None*, *axes=None*, *sampling=None*, *units='pix'*, *plot_style='vector'*, *overlay=True*, *unit_vector=False*, *degrees=False*, *save='atom_deviation'*, *title=''*, *color='yellow'*, *cmap=None*, *pivot='middle'*, *angles='xy'*, *scale_units='xy'*, *scale=None*, *headwidth=3.0*, *headlength=5.0*, *headaxislength=4.5*, *monitor_dpi=96*, *no_axis_info=True*, *scalebar=False*)

Combine the atom deviations of each atom for all zone axes. Good for plotting vortexes and seeing the total deviation from a perfect structure.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **plot_polarisation_vectors()** (*For the remaining parameters see*) –

Returns

- **Four lists** (*x, y, u, and v where x,y are the original atom position*)
- *coordinates (simply sublattice.x_position, sublattice.y_position) and*
- *u,v are the polarisation vector components pointing to the new coordinate.*
- These can be input to `plot_polarisation_vectors()` for visualisation.

Examples

```
>>> import atomap.api as am
>>> from temul.topotem.polarisation import (plot_polarisation_vectors,
...     combine_atom_deviations_from_zone_axes)
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> x,y,u,v = combine_atom_deviations_from_zone_axes(
...     sublatticeA, save=None)
>>> ax = plot_polarisation_vectors(x, y, u, v, save=None,
...     image=sublatticeA.image)
```

You can also choose the axes:

```
>>> x,y,u,v = combine_atom_deviations_from_zone_axes(
...     sublatticeA, axes=[0,1], save=None)
```

(continues on next page)

(continued from previous page)

```
>>> ax = plot_polarisation_vectors(x, y, u, v, save=None,
...     image=sublatticeA.image)
```

temul.topotem.polarisation.**correct_off_tilt_vectors**(*u*, *v*, *method='com'*)

Useful if your image is off-tilt (electron beam is not perfectly parallel to the atomic columns).

Parameters

- **u** (*1D numpy arrays*) – horizontal and vertical components of the (polarisation) vectors.
- **v** (*1D numpy arrays*) – horizontal and vertical components of the (polarisation) vectors.
- **method** (*string, default "com"*) – method used to correct the vector components. "com" is via the center of mass of the vectors. "av" is via the average vector.

Returns u_corr, v_corr

Return type corrected 1D numpy arrays

Examples

```
>>> from temul.topotem.polarisation import correct_off_tilt_vectors
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> sublatticeA.construct_zone_axes()
>>> za0, za1 = sublatticeA.zones_axis_average_distances[0:2]
>>> s_p = sublatticeA.get_polarization_from_second_sublattice(
...     za0, za1, sublatticeB, color='blue')
>>> vector_list = s_p.metadata.vector_list
>>> x, y = [i[0] for i in vector_list], [i[1] for i in vector_list]
>>> u, v = [i[2] for i in vector_list], [i[3] for i in vector_list]
```

Correct for some tilt using the correct_off_tilt_vectors function:

```
>>> u_com, v_com = correct_off_tilt_vectors(u, v, method="com")
```

Use the average vector instead: (be careful that you're not just applying this on previously corrected data!)

```
>>> u_av, v_av = correct_off_tilt_vectors(u, v, method="av")
```

temul.topotem.polarisation.**corrected_vectors_via_average**(*u*, *v*)

temul.topotem.polarisation.**corrected_vectors_via_center_of_mass**(*u*, *v*)

temul.topotem.polarisation.**delete_atom_planes_from_sublattice**(*sublattice*, *zone_axis_index=0*, *atom_plane_tolerance=0.5*, *divisible_by=3*, *offset_from_zero=0*, *opposite=False*)

Delete atom_planes from a zone axis. Can choose whether to delete every second, third etc. atom plane, and the offset from the zero index.

Parameters

- **sublattice** (*Atomap Sublattice object*) –

- **zone_axis_index** (*int, default 0*) – The zone axis you wish to specify. You are indexing sublattice.zones_axis_average_distances[zone_axis_index]
- **atom_plane_tolerance** (*float, default 0.5*) – float between 0.0 and 1.0. Closer to 1 means it will find more zones. See sublattice.construct_zone_axes() for more information.
- **divisible_by** (*int, default 3*) – If divisible_by is 2, every second atom_plane is deleted, If divisible_by is 4, every fourth atom_plane is deleted, etc.
- **offset_from_zero** (*int, default 0*) – The atom_plane from which you start deleting. If offset_from_zero is 4, the fourth atom_plane will be the first deleted.
- **opposite** (*bool, default False*) – If this is set to True, the atom_plane specified by divisible_by will be kept and all others deleted.

Examples

```
>>> from temul.topotem.polarisation import (
...     delete_atom_planes_from_sublattice)
>>> import atomap.dummy_data as dd
>>> atom_lattice = dd.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.construct_zone_axes()
>>> zone_vec_list = sublatticeA.zones_axis_average_distances[0:2]
```

Plot the planes visually using, it may take some time in large images: zones01_A = sublatticeA.get_all_atom_planes_by_zone_vector(zone_vec_list) zones01_A.plot()

```
>>> delete_atom_planes_from_sublattice(
...         sublatticeA, zone_axis_index=0,
...         divisible_by=3, offset_from_zero=1)
```

zones01_B = sublatticeA.get_all_atom_planes_by_zone_vector(zone_vec_list) zones01_A.plot()

temul.topotem.polarisation.**find_polarisation_vectors**(*atom_positions_A*, *atom_positions_B*, *save=None*)

Calculate the vectors from atom_positions_A to atom_positions_B.

Parameters

- **atom_positions_A** (*list*) – Atom positions list in the form [[x1,y1], [x2,y2], [x3,y3]...].
- **atom_positions_B** (*list*) – Atom positions list in the form [[x1,y1], [x2,y2], [x3,y3]...].
- **save** (*string, default None*) – If set to a string, the array will be saved.

Returns two lists

Return type u and v components of the vector from A to B

Examples

```
>>> from temul.topotem.polarisation import find_polarisation_vectors
>>> pos_A = [[1,2], [3,4], [5,8], [5,2]]
>>> pos_B = [[1,1], [5,2], [3,1], [6,2]]
>>> u, v = find_polarisation_vectors(pos_A, pos_B, save=None)
```

convert to the [[u1,v1], [u2,v2], [u3,v3]...] format

```
>>> import numpy as np
>>> vectors = np.asarray([u, v]).T
```

temul.topotem.polarisation.**find_polarization_vectors**(*atom_positions_A*, *atom_positions_B*, *save=None*)

Alias function for find_polarisation_vectors.

temul.topotem.polarisation.**full_atom_plane_deviation_from_straight_line_fit**(*sublattice*, *axis_number: int = 0*, *save: str = ''*)

Fit the atoms in an atom plane to a straight line and find the deviation of each atom position from that straight line fit. To plot all zones see `plot_atom_deviation_from_all_zone_axes()`.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **axis_number** (*int, default 0*) – The index of the zone axis (translation symmetry) found by the Atomap function `construct_zone_axes()`.
- **save** (*string, default ''*) – If set to `save=None`, the array will not be saved.

Returns

- **Four lists** (*x, y, u, and v where x,y are the original atom position*)
- *coordinates (simply sublattice.x_position, sublattice.y_position) and*
- *u,v are the polarisation vector components pointing to the new coordinate.*
- These can be input to `plot_polarisation_vectors()` for visualisation.

Examples

```
>>> import atomap.api as am
>>> from temul.topotem.polarisation import (
...     full_atom_plane_deviation_from_straight_line_fit,
...     plot_polarisation_vectors)
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> x,y,u,v = full_atom_plane_deviation_from_straight_line_fit(
...     sublatticeA, save=None)
>>> ax = plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, save=None, monitor_dpi=50)
```

temul.topotem.polarisation.**get_angles_from_uv**(*u*, *v*, *degrees=False*, *angle_offset=None*)
Calculate the angle of a vector given the uv components.

Parameters

- **u** (*list or 1D NumPy array*) –
- **v** (*list or 1D NumPy array*) –
- **degrees** (*bool, default False*) – Change between degrees and radian. Default is radian.
- **angle_offset** (*float, default None*) – Rotate the angles by the given amount. The function assumes that if you set `degrees=False` then the provided `angle_offset` is in radians, and if you set `degrees=True` then the provided `angle_offset` is in degrees.

Returns

Return type 1D NumPy array

temul.topotem.polarisation.**get_average_polarisation_in_regions**(*x*, *y*, *u*, *v*, *image*, *divide_into=8*)
This function splits the image into the given number of regions and averages the polarisation of each region.

Parameters

- **x** (*list or 1D NumPy array*) – xy coordinates on the image
- **y** (*list or 1D NumPy array*) – xy coordinates on the image
- **u** (*list or 1D NumPy array*) – uv vector components
- **v** (*list or 1D NumPy array*) – uv vector components
- **image** (*2D NumPy array*) –
- **divide_into** (*int, default 8*) – The number used to divide the image up. If 8, then the image will be split into an 8x8 grid.

Returns **x_new, y_new, u_new, v_new** – x_new and y_new are the central coordinates of the divided regions. u_new and v_new are the averaged polarisation vectors.

Return type lists of equal length

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import atomap.api as am
>>> from temul.topotem.polarisation import (
...    combine_atom_deviations_from_zone_axes,
...    plot_polarisation_vectors, get_average_polarisation_in_regions)
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
```

Get and plot the original polarisation vectors:

```
>>> x, y, u, v = combine_atom_deviations_from_zone_axes(sublatticeA,
...     save=None)
>>> ax = plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                   unit_vector=False, save=None,
...                   plot_style='vector', color='r',
...                   overlay=False, title='Actual Vector Arrows',
...                   monitor_dpi=50)
```

Get and plot the new, averaged polarisation vectors

```
>>> x_new, y_new, u_new, v_new = get_average_polarisation_in_regions(
...     x, y, u, v, image=sublatticeA.image, divide_into=8)
>>> ax = plot_polarisation_vectors(x_new, y_new, u_new, v_new,
...         monitor_dpi=50, image=sublatticeA.image, save=None, color='r',
...         overlay=False, title='Averaged Vector Arrows')
```

temul.topotem.polarisation.**get_average_polarisation_in_regions_square**(*x*, *y*, *u*, *v*, *image*, *divide_into=4*)

Same as `get_average_polarisation_in_regions()` but works for non-square (rectangular) images.

Parameters

- **x** (*list or 1D NumPy array*) – xy coordinates on the image
- **y** (*list or 1D NumPy array*) – xy coordinates on the image
- **u** (*list or 1D NumPy array*) – uv vector components
- **v** (*list or 1D NumPy array*) – uv vector components
- **image** (*2D NumPy array*) –
- **divide_into** (*int, default 8*) – The number used to divide the image up. If 8, then the image will be split into an 8x8 grid.

Returns **x_new, y_new, u_new, v_new** – x_new and y_new are the central coordinates of the divided regions. u_new and v_new are the averaged polarisation vectors.

Return type lists of equal length

Examples

```
>>> import atomap.api as am
>>> from temul.topotem.polarisation import (
...     combine_atom_deviations_from_zone_axes, plot_polarisation_vectors,
...     get_average_polarisation_in_regions_square)
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
```

Get and plot the original polarisation vectors of a non-square image

```
>>> image = sublatticeA.image[0:200]
>>> x, y, u, v = combine_atom_deviations_from_zone_axes(sublatticeA,
```

(continues on next page)

(continued from previous page)

```
...     save=None)
>>> ax = plot_polarisation_vectors(x, y, u, v, image=image, save=None,
...                   color='r', overlay=False, monitor_dpi=50,
...                   title='Actual Vector Arrows')
```

Get and plot the new, averaged polarisation vectors for a non-square image

```
>>> coords = get_average_polarisation_in_regions_square(
...     x, y, u, v, image=image, divide_into=8)
>>> x_new, y_new, u_new, v_new = coords
>>> ax = plot_polarisation_vectors(x_new, y_new, u_new, v_new, image=image,
...                   color='r', overlay=False, monitor_dpi=50,
...                   title='Averaged Vector Arrows', save=None)
```

temul.topotem.polarisation.**get_average_polarization_in_regions**(*x*, *y*, *u*, *v*, *image*, *divide_into=8*)

Alias function for get_average_polarisation_in_regions

temul.topotem.polarisation.**get_average_polarization_in_regions_square**(*x*, *y*, *u*, *v*, *image*, *divide_into=4*)

Alias function for get_average_polarization_in_regions_square

temul.topotem.polarisation.**get_divide_into**(*sublattice*, *averaging_by*, *sampling*, *zone_axis_index_A*, *zone_axis_index_B*)

Calculate the `divide_into` required to get an averaging of `averaging_by`. `divide_into` can then be used in temul.topotem.polarisation.get_average_polarisation_in_regions. Also finds unit cell size and the number of unit cells in the (square) image along the x axis.

Parameters

- **sublattice** (*Atomap Sublattice*) –
- **averaging_by** (*int or float*) – How many unit cells should be averaged. If `averaging_by=2`, 2x2 unit cells will be averaged when passing `divide_into` to temul.topotem.polarisation.get_average_polarisation_in_regions.
- **sampling** (*float*) – Pixel sampling of the image for calibration.
- **zone_axis_index_A** (*int*) – Sublattice zone axis indices which should represent the sides of the unit cell.
- **zone_axis_index_B** (*int*) – Sublattice zone axis indices which should represent the sides of the unit cell.

Returns

Return type divide_into, unit_cell_size, num_unit_cells

Examples

```
>>> from temul.topotem.polarisation import get_divide_into
>>> from atomap.dummy_data import get_simple_cubic_sublattice
>>> sublattice = get_simple_cubic_sublattice()
>>> sublattice.construct_zone_axes()
>>> cell_info = get_divide_into(sublattice, averaging_by=2, sampling=0.1,
...                   zone_axis_index_A=0, zone_axis_index_B=1)
>>> divide_into = cell_info[0]
```

(continues on next page)

(continued from previous page)

```
>>> unit_cell_size = cell_info[1]
>>> num_unit_cells = cell_info[2]
>>> sublattice.plot()  # You can count the unit cells to check
```

temul.topotem.polarisation.**get_strain_map**(*sublattice*, *zone_axis_index*, *theoretical_value*, *sampling=None*, *units='pix'*, *vmin=None*, *vmax=None*, *cmap='inferno'*, *title='Strain Map'*, *plot=False*, *filename=None*, *return_x_y_z=False*, ***kwargs*)

Calculate the strain across a zone axis of a sublattice.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **zone_axis_index** (*int*) – The zone axis you wish to specify. You are indexing `sublattice.zones_axis_average_distances[zone_axis_index]`.
- **theoretical_value** (*float*) – The theoretical separation between the atoms across (not along) the specified zone.
- **sampling** (*float, default None*) – Pixel sampling of the image for calibration.
- **units** (*string, default "pix"*) – Units of the sampling.
- **vmin** (*see Matplotlib for details*) –
- **vmax** (*see Matplotlib for details*) –
- **cmap** (*see Matplotlib for details*) –
- **title** (*string, default "Strain Map"*) –
- **plot** (*bool*) – Set to true if the figure should be plotted.
- **filename** (*string, optional*) – If filename is set, the strain signal and plot will be saved. `plot` must be set to True.
- **return_x_y_z** (*bool, default False*) – If this is set to True, the `strain_signal` (map), as well as separate lists of the x, y, and strain values.
- ****kwargs** (Matplotlib keyword arguments passed to `imshow()`.) –

Returns

Return type Strain map as a Hyperspy Signal2D

Examples

```
>>> import atomap.api as am
>>> from temul.topotem.polarisation import get_strain_map
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> strain_map = get_strain_map(sublatticeA, zone_axis_index=0, units='nm',
...                             theoretical_value=1.9, sampling=0.1)
```

temul.topotem.polarisation.**get_vector_magnitudes**(*u*, *v*, *sampling=None*)

Calculate the magnitude of a vector given the uv components.

Parameters

- **u** (*list or 1D NumPy array*) –
- **v** (*list or 1D NumPy array*) –
- **sampling** (*float, default None*) – If sampling is set, the vector magnitudes (in pix) will be scaled by sampling (nm/pix).

Returns

Return type 1D NumPy array

Examples

```
>>> from temul.topotem.polarisation import get_vector_magnitudes
>>> import numpy as np
>>> u, v = [4,3,2,5,6], [8,5,2,1,1] # list input
>>> vector_mags = get_vector_magnitudes(u,v)
>>> u, v = np.array(u), np.array(v) # numpy input also works
>>> vector_mags = get_vector_magnitudes(u,v)
>>> sampling = 0.0321
>>> vector_mags = get_vector_magnitudes(u,v, sampling=sampling)
```

temul.topotem.polarisation.**get_xyuv_from_line_fit**(*arr*, *n*, *second_fit_rigid=True*, *plot=False*)

Fits the data in an array to two straight lines using the first `n` and second (last) `n` array value pairs. Computes the distance of each array value pair from the line halfway between the two fitted lines.

The slope of the first fitting will be used for the second fitting. Setting `second_fit_rigid` = False will reverse this behaviour.

Parameters

- **arr** (*2D array-like*) – Array-like (e.g., NumPy 2D array) in the form [[x1, y1], [x2, y2]...]
- **n** (*int*) – The number of arr value pairs used at the beginning and end of the arr to fit a straight line.
- **second_fit_rigid** (*bool, default True*) – Used to decide whether the first or second fitting's slope will be rigid during fitting. With `second_fit_rigid=True`, the slope of the second fitting will be defined as the slope as the first fitting. The y-intercept is free to move.
- **plot** (*bool, default False*) – Whether to plot the arr data, first and second fitting, and the line constructed halfway between the two.

Returns **x, y, u, v** – x, y are the original arr coordinates. u, v are the vector components pointing towards the halfway line from the arr coordinates. These can be input to `plot_polarisation_vectors()` for visualisation.

Return type lists of equal length

See also:

atom_deviation_from_straight_line_fit Gets xyuv for Sublattice object

Examples

```
>>> arr = np.array([[1, 2], [2, 2], [3, 2], [4, 2], [5, 2.05],
...                 [6, 1], [7, 1], [8, 1], [9, 1], [10, 0.75]])
>>> x, y, u, v = get_xyuv_from_line_fit(
...     arr, n=5, second_fit_rigid=True, plot=False)
```

Use the lower (second) cluster to fit the data, making the first cluster rigid

```
>>> x, y, u, v = get_xyuv_from_line_fit(
...     arr, n=5, second_fit_rigid=False, plot=False)
```

temul.topotem.polarisation.**plot_atom_deviation_from_all_zone_axes**(*sublattice*, *image=None*, *sampling=None*, *units='pix'*, *plot_style='vector'*, *overlay=True*, *unit_vector=False*, *degrees=False*, *save='atom_deviation'*, *title=''*, *color='yellow'*, *cmap=None*, *pivot='middle'*, *angles='xy'*, *scale_units='xy'*, *scale=None*, *headwidth=3.0*, *headlength=5.0*, *headaxislength=4.5*, *monitor_dpi=96*, *no_axis_info=True*, *scalebar=False*)

Plot the atom deviation from a straight line fit for all zone axes constructed by an Atomap sublattice object.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **plot_polarisation_vectors()** (*For all other parameters see*) –

Examples

```
>>> import atomap.dummy_data as dd
>>> import temul.api as tml
>>> atom_lattice = dd.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> tml.plot_atom_deviation_from_all_zone_axes(
...     sublatticeA, save=None)
```

temul.topotem.polarisation.**plot_polarisation_vectors**(*x*, *y*, *u*, *v*, *image*, *sampling=None*, *units='pix'*, *plot_style='vector'*, *overlay=True*, *unit_vector=False*, *vector_rep='magnitude'*, *degrees=False*, *angle_offset=None*, *save='polarisation_image'*, *title=''*, *color='yellow'*, *cmap=None*, *alpha=1.0*, *image_cmap='gray'*, *monitor_dpi=96*, *no_axis_info=True*, *invert_y_axis=True*, *ticks=None*, *scalebar=False*, *antialiased=False*, *levels=20*, *remove_vectors=False*, *quiver_units='width'*, *pivot='middle'*, *angles='xy'*, *scale_units='xy'*, *scale=None*, *headwidth=3.0*, *headlength=5.0*, *headaxislength=4.5*, *width=None*, *minshaft=1*, *minlength=1*)

Plot the polarisation vectors. These can be found with `find_polarisation_vectors()` or Atomap's `get_polarization_from_second_sublattice()` function.

Parameters

- **details.** (*See matplotlib's quiver function for more*) –
- **x** (*list or 1D NumPy array*) – xy coordinates of the vectors on the image.
- **y** (*list or 1D NumPy array*) – xy coordinates of the vectors on the image.
- **u** (*list or 1D NumPy array*) – uv vector components.
- **v** (*list or 1D NumPy array*) – uv vector components.
- **image** (*2D NumPy array*) – image is used to fit the image. Will flip the y axis, as used for electron microscopy data (top left point is (0, 0) coordinate).
- **sampling** (*float, default None*) – Pixel sampling (pixel size) of the image for calibration.
- **units** (*string, default "pix"*) – Units used to display the magnitude of the vectors.
- **plot_style** (*string, default "vector"*) – Options are "vector", "colormap", "contour", "colorwheel", "polar_colorwheel". Note that "colorwheel" will automatically plot the colorbar as an angle. Also note that "polar_colorwheel" will automatically generate a 2D RGB (HSV) list of colors that match with the vector components (uv).
- **overlay** (*bool, default True*) – If set to True, the `image` will be plotting behind the arrows
- **unit_vector** (*bool, default False*) – Change the vectors magnitude to unit vectors for plotting purposes. Magnitude will still be displayed correctly for colormaps etc.
- **vector_rep** (*str, default "magnitude"*) – How the vectors are represented. This can be either their `magnitude` or `angle`. One may want to use `angle` when plotting a contour map, i.e. view the contours in terms of angles which can be useful for visualising regions of different polarisation.
- **degrees** (*bool, default False*) – Change between degrees and radian. Default is radian. If `plot_style="colorwheel"`, then setting `degrees=True` will convert the angle unit to degree from the default radians.
- **angle_offset** (*float, default None*) – If using `vector_rep="angle"` or `plot_style="contour"`, this angle will rotate the vector angles displayed by the

given amount. Useful when you want to offset the angle of the atom planes relative to the polarisation.

- **save** (*string*) – If set to `save=None`, the image will not be saved.
- **title** (*string, default ""*) – Title of the plot.
- **color** (*string, default "r"*) – Color of the arrows when `plot_style="vector"` or `"contour"`.
- **cmap** (matplotlib colormap, default `"viridis"`) – Matplotlib cmap used for the vector arrows.
- **alpha** (*float, default 1.0*) – Transparency of the matplotlib `cmap`. For `plot_style="colormap"` and `plot_style="colorwheel"`, this alpha applies to the vector arrows. For `plot_style="contour"` this alpha applies to the tricontourf map.
- **image_cmap** (*matplotlib colormap, default 'gray'*) – Matplotlib cmap that will be used for the overlay image.
- **monitor_dpi** (*int, default 96*) – The DPI of the monitor, generally 96 pixels. Used to scale the image so that large images render correctly. Use a smaller value to enlarge too-small images. `monitor_dpi=None` will ignore this param.
- **no_axis_info** (*bool, default True*) – This will remove the x and y axis labels and ticks from the plot if set to True.
- **invert_y_axis** (*bool, default True*) – If set to true, this will flip the y axis, effectively setting the top left corner of the image as the (0, 0) origin, as in scanning electron microscopy images.
- **ticks** (*colorbar ticks, default None*) – None or list of ticks or Locator If None, ticks are determined automatically from the input.
- **scalebar** (*bool or dict, default False*) – Add a matplotlib-scalebar to the plot. If set to True the scalebar will appear similar to that given by Hyperspy's `plot()` function. A custom scalebar can be included as a dictionary and more custom options can be found in the matplotlib-scalebar package. See below for an example.
- **antialiased** (*bool, default False*) – Applies only to `plot_style="contour"`. Essentially removes the border between regions in the tricontourf map.
- **levels** (*int, default 20*) – Number of Matplotlib tricontourf levels to be used.
- **remove_vectors** (*bool, default False*) – Applies only to `plot_style="contour"`. If set to True, do not plot the vector arrows.
- **quiver_units** (*string, default 'width'*) – The units parameter from the matplotlib quiver function, not to be confused with the `units` parameter above for the image units.
- **parameters** (*ax.quiver*) – See matplotlib's quiver function for the remaining parameters.

Returns **ax** – Matplotlib Axes object

Return type Axes

Examples

```
>>> import temul.api as tml
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> sublatticeA.construct_zone_axes()
>>> za0, za1 = sublatticeA.zones_axis_average_distances[0:2]
>>> s_p = sublatticeA.get_polarization_from_second_sublattice(
...     za0, za1, sublatticeB, color='blue')
>>> vector_list = s_p.metadata.vector_list
>>> x, y = [i[0] for i in vector_list], [i[1] for i in vector_list]
>>> u, v = [i[2] for i in vector_list], [i[3] for i in vector_list]
```

vector plot with red arrows:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, save=None,
...                            plot_style='vector', color='r',
...                            overlay=False, title='Vector Arrows',
...                            monitor_dpi=50)
```

vector plot with red arrows overlaid on the image:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, save=None,
...                            plot_style='vector', color='r',
...                            overlay=True, monitor_dpi=50)
```

vector plot with colormap viridis:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, save=None,
...                            plot_style='colormap', monitor_dpi=50,
...                            overlay=False, cmap='viridis')
```

vector plot with colormap viridis, with `vector_rep="angle"`:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, save=None,
...                            plot_style='colormap', monitor_dpi=50,
...                            overlay=False, cmap='cet_colorwheel',
...                            vector_rep="angle", degrees=True)
```

colormap arrows with sampling applied and with scalebar:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            sampling=3.0321, units='pm', monitor_dpi=50,
...                            unit_vector=False, plot_style='colormap',
...                            overlay=True, save=None, cmap='viridis',
...                            scalebar=True)
```

vector plot with colormap viridis and unit vectors:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=True, save=None, monitor_dpi=50,
...                            plot_style='colormap', color='r',
...                            overlay=False, cmap='viridis')
```

Change the vectors to unit vectors on a tricontourf map:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=True, plot_style='contour',
...                            overlay=False, pivot='middle', save=None,
...                            color='darkgray', cmap='plasma',
...                            monitor_dpi=50)
```

Plot a partly transparent angle tricontourf map with vector arrows:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, plot_style='contour',
...                            overlay=True, pivot='middle', save=None,
...                            color='red', cmap='cet_colorwheel',
...                            monitor_dpi=50, remove_vectors=False,
...                            vector_rep="angle", alpha=0.5, levels=9,
...                            antialiased=True, degrees=True,
...                            ticks=[180, 90, 0, -90, -180])
```

Plot a partly transparent angle tricontourf map with no vector arrows:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=True, plot_style='contour',
...                            overlay=True, pivot='middle', save=None,
...                            cmap='cet_colorwheel',
...                            monitor_dpi=50, remove_vectors=True,
...                            vector_rep="angle", alpha=0.5,
...                            antialiased=True, degrees=True)
```

“colorwheel” plot of the vectors, useful for vortexes:

```
>>> import colorcet as cc
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=True, plot_style="colorwheel",
...                            vector_rep="angle",
...                            overlay=False, cmap=cc.cm.colorwheel,
...                            degrees=True, save=None, monitor_dpi=50,
...                            ticks=[180, 90, 0, -90, -180])
```

“polar_colorwheel” plot showing a 2D polar color wheel:

```
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            plot_style="polar_colorwheel",
...                            unit_vector=False, overlay=False,
...                            save=None, monitor_dpi=50)
```

Plot with a custom scalebar, for example here we need it to be dark, see matplotlib-scalebar for more custom features.

```
>>> scbar_dict = {"dx": 3.0321, "units": "pm", "location": "lower left",
...               "box_alpha":0.0, "color": "black", "scale_loc": "top"}
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            sampling=3.0321, units='pm', monitor_dpi=50,
...                            unit_vector=False, plot_style='colormap',
...                            overlay=False, save=None, cmap='viridis',
...                            scalebar=scbar_dict)
```

Plot a contourf for quadrant visualisation using a custom matplotlib cmap:

```
>>> import temul.api as tml
>>> from matplotlib.colors import from_levels_and_colors
>>> zest = tml.hex_to_rgb(tml.color_palettes('zesty'))
>>> zest.append(zest[0])  # make the -180 and 180 degree colour the same
>>> expanded_zest = tml.expand_palette(zest, [1,2,2,2,1])
>>> custom_cmap, _ = from_levels_and_colors(
...     levels=range(9), colors=tml.rgb_to_dec(expanded_zest))
>>> ax = tml.plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, plot_style='contour',
...                            overlay=False, pivot='middle', save=None,
...                            cmap=custom_cmap, levels=9, monitor_dpi=50,
...                            vector_rep="angle", alpha=0.5, color='r',
...                            antialiased=True, degrees=True,
...                            ticks=[180, 90, 0, -90, -180])
```

temul.topotem.polarisation.**plot_polarization_vectors**(*x*, *y*, *u*, *v*, *image*, *sampling=None*, *units='pix'*, *plot_style='vector'*, *overlay=True*, *unit_vector=False*, *vector_rep='magnitude'*, *degrees=False*, *angle_offset=None*, *save='polarisation_image'*, *title=''*, *color='yellow'*, *cmap=None*, *alpha=1.0*, *image_cmap='gray'*, *monitor_dpi=96*, *no_axis_info=True*, *invert_y_axis=True*, *ticks=None*, *scalebar=False*, *antialiased=False*, *levels=20*, *remove_vectors=False*, *quiver_units='width'*, *pivot='middle'*, *angles='xy'*, *scale_units='xy'*, *scale=None*, *headwidth=3.0*, *headlength=5.0*, *headaxislength=4.5*, *width=None*, *minshaft=1*, *minlength=1*)

Alias function name for *plot_polarisation_vectors*.

temul.topotem.polarisation.**ratio_of_lattice_spacings**(*sublattice*, *zone_axis_index_A*, *zone_axis_index_B*, *ideal_ratio_one=True*, *sampling=1*, *units='pix'*, *vmin=None*, *vmax=None*, *cmap='inferno'*, *title='Spacings Map'*, *filename=None*, ***kwargs*)

Create a ratio map between two zone axes. Useful to see the tetragonality or shearing of unit cells.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **zone_axis_index_A** (*int*) – The zone axes you wish to specify. You are indexing `sublattice.zones_axis_average_distances[zone_axis_index]`. The signal created from zone_axis_index_A will be divided by the signal created from zone_axis_index_B.

- **zone_axis_index_B** (*int*) – The zone axes you wish to specify. You are indexing sublattice.zones_axis_average_distances[zone_axis_index]. The signal created from zone_axis_index_A will be divided by the signal created from zone_axis_index_B.
- **ideal_ratio_one** (*bool, default True*) – If set to true this will force negative ratio values to be positive. Useful for seeing the overall tetragonality of a lattice.
- **sampling** (*float, default None*) – Pixel sampling of the image for calibration.
- **units** (*string, default "pix"*) – Units of the sampling.
- **vmin** (*see Matplotlib for details*) –
- **vmax** (*see Matplotlib for details*) –
- **cmap** (*see Matplotlib for details*) –
- **title** (*string, default "Spacings Map"*) –
- **filename** (*string, optional*) – If filename is set, the strain signal and plot will be saved.
- ****kwargs** (Matplotlib keyword arguments passed to imshow().) –

Returns

- *Ratio of lattice spacings map as a Hyperspy Signal2D. It will also plot the*
- *two lattice spacing maps.*

Examples

```
>>> import atomap.api as am
>>> from temul.topotem.polarisation import ratio_of_lattice_spacings
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> ratio_map = ratio_of_lattice_spacings(sublatticeA, 0, 1)
```

Use ideal_ratio_one=False to view the direction of tetragonality

```
>>> ratio_map = ratio_of_lattice_spacings(sublatticeA, 0, 1,
...                                       ideal_ratio_one=False)
```

temul.topotem.polarisation.**rotation_of_atom_planes**(*sublattice*, *zone_axis_index*, *angle_offset=None*, *degrees=False*, *sampling=None*, *units='pix'*, *vmin=None*, *vmax=None*, *cmap='inferno'*, *title='Rotation Map'*, *plot=False*, *filename=None*, *return_x_y_z=False*, ***kwargs*)

Calculate a map of the angles between each atom along the atom planes of a zone axis.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **zone_axis_index** (*int*) – The zone axis you wish to specify. You are indexing sublattice.zones_axis_average_distances[zone_axis_index].
- **angle_offset** (*float, default None*) – The angle which can be considered zero degrees. Useful when the atomic planes are at an angle.

- **degrees** (*bool, default False*) – Setting to False will return angle values in radian. Setting to True will return angle values in degrees.
- **sampling** (*float, default None*) – Pixel sampling of the image for calibration.
- **units** (*string, default "pix"*) – Units of the sampling.
- **vmin** (*see Matplotlib for details*) –
- **vmax** (*see Matplotlib for details*) –
- **cmap** (*see Matplotlib for details*) –
- **title** (*string, default "Rotation Map"*) –
- **plot** (*bool*) – Set to true if the figure should be plotted.
- **filename** (*string, optional*) – If filename is set, the signal and plot will be saved. `plot` must be set to True.
- **return_x_y_z** (*bool, default False*) – If this is set to True, the rotation_signal (map), as well as separate lists of the x, y, and angle values.
- ****kwargs** (Matplotlib keyword arguments passed to `imshow()`.) –

Returns

Return type Rotation map as a Hyperspy Signal2D

Examples

```
>>> import atomap.api as am
>>> from temul.topotem.polarisation import rotation_of_atom_planes
>>> atom_lattice = am.dummy_data.get_polarization_film_atom_lattice()
>>> sublatticeA = atom_lattice.sublattice_list[1]
>>> sublatticeA.find_nearest_neighbors()
>>> _ = sublatticeA.refine_atom_positions_using_center_of_mass()
>>> sublatticeA.construct_zone_axes()
>>> rotation_map = rotation_of_atom_planes(sublatticeA, 3, degrees=True)
```

Use `angle_offset` to effectively change the angle of the horizontal axis when calculating angles:

```
>>> rotation_map = rotation_of_atom_planes(sublatticeA, 3, angle_offset=45,
...                                        degrees=True)
```

Use the return_x_y_z parameter when you want to either plot with a different style (e.g., contour map), or you want the angle information:

```
>>> rotation_map, x, y, angles = rotation_of_atom_planes(
...     sublatticeA, 3, degrees=True, return_x_y_z=True)
>>> mean_angle = np.mean(angles)  # useful for offsetting polar. plots
```

temul.topotem.fft_mapping.**choose_mask_coordinates**(*image*, *norm='log'*)

Pick the mask locations for an FFT. See get_masked_ifft() for examples and commit 5ba307b5af0b598bedc0284aa989d44e23fdde4d on Atomap for more details.

Parameters

- **image** (*Hyperspy 2D Signal*) –

- **norm** (*str, default "log"*) – How to scale the intensity value for the displayed image. Options are "linear" and "log".

Returns

Return type list of pixel coordinates

temul.topotem.fft_mapping.**choose_points_on_image**(*image*, *norm='linear'*, *distance_threshold=4*)

temul.topotem.fft_mapping.**get_masked_ifft**(*image*, *mask_coords*, *mask_radius=10*, *image_space='real'*, *keep_masked_area=True*, *plot_masked_fft=False*)

Creates an inverse fast Fourier transform (iFFT) from an image and mask coordinates. Use `choose_mask_coordinates` to manually choose mask coordinates in the FFT.

Parameters

- **image** (*Hyperspy 2D Signal*) –
- **mask_coords** (*list of pixel coordinates*) – Pixel coordinates of the masking locations. See the example below for two simple coordinates found using `choose_mask_coordinates`.
- **mask_radius** (*int, default 10*) – Radius in pixels of the mask.
- **image_space** (*str, default "real"*) – If the input image is in Fourier/diffraction/reciprocal space already, set space="fourier".
- **keep_masked_area** (*bool, default True*) – If True, this will set the mask at the mask_coords. If False, this will set the mask as everything other than the mask_coords. Can be thought of as inversing the mask.
- **plot_masked_fft** (*bool, default False*) – If True, the mask used to filter the FFT will be plotted. Good for checking that the mask is doing what you want. Can fail sometimes due to matplotlib plotting issues.

Returns

Return type Inverse FFT as a Hyperspy 2D Signal

Examples

```
>>> from temul.dummy_data import get_simple_cubic_signal
>>> import temul.api as tml
>>> image = get_simple_cubic_signal()
>>> mask_coords = tml.choose_mask_coordinates(image) #  manually
>>> mask_coords = [[170.2, 170.8],[129.8, 130]]
>>> image_ifft = tml.get_masked_ifft(image, mask_coords)
>>> image_ifft.plot()
```

Plot the masked fft:

```
>>> image_ifft = tml.get_masked_ifft(image, mask_coords,
...                                  plot_masked_fft=True)
```

Use unmasked fft area and plot the masked fft:

```
>>> image_ifft = tml.get_masked_ifft(
...     image, mask_coords, plot_masked_fft=True, keep_masked_area=False)
>>> image_ifft.plot()
```

If the input image is already a Fourier transform:

```
>>> fft_image = image.fft(shift=True)
>>> image_ifft = tml.get_masked_ifft(fft_image, mask_coords,
...     image_space='fourier')
>>> image_ifft.plot()
```

temul.topotem.lattice_structure_tools.**calculate_atom_plane_curvature**(*sublattice*, *zone_vector_index*, *func='strain_grad'*, *atom_planes=None*, *sampling=None*, *units='pix'*, *vmin=None*, *vmax=None*, *cmap='inferno'*, *title='Curvature Map'*, *filename=None*, *plot=False*, *return_fits=False*, ***kwargs*)

Calculates the curvature of the sublattice atom planes along the direction given by `zone_vector_index`. In the case of [1]_. below, the curvature is the inverse of the radius of curvature, and is effectively equal to the second derivative of the displacement direction of the atoms. Because the first derivative is negligible, the curvature can be calculated as the strain gradient[2]. With the parameter `func="strain_grad"`, this function calculates the strain gradient of the atom planes of a Atomap Sublattice object.

Parameters

- **sublattice** (*Atomap Sublattice object*) –
- **zone_vector_index** (*int*) – The index of the zone axis (translation symmetry) found by the Atomap function `construct_zone_axes()`.
- **func** (*'strain_grad' or function*) – Function that can be used by `scipy.optimize.curve_fit`. If func='strain_grad', then the `temul.topotem.lattice_structure_tools.sine_wave_function_strain_gradient` function will be used.
- **atom_planes** (*tuple, optional*) – The starting and ending atom plane to be computed. Useful if only a section of the image should be fitted with sine waves. Given in the form e.g., (0, 3).
- **sampling** (*float, optional*) – sampling of an image in units of units/pix
- **units** (*string, default "pix"*) – Units of sampling, for display purposes.
- **vmin** (*see Matplotlib documentation, default None*) –
- **vmax** (*see Matplotlib documentation, default None*) –
- **cmap** (*see Matplotlib documentation, default None*) –
- **title** (*string, default 'Curvature Map'*) –
- **filename** (*string, default None*) – Name of the file to be saved.
- **plot** (*bool, default False*) – If set to True, each atom plane fitting will be plotted along with its respective atom positions. The fitting parameters (popt) will be returned as a list.

[2] Landau and Lifshitz, Theory of Elasticity, Vol 7, pp 47-49, 1981

- **return_fits** (*bool, default False*) – If set to True, each atom plane fitting will be plotted along with its respective atom positions. The fitting parameters (popt) will be returned as a list.
- ****kwargs** – keyword arguments to be passed to `scipy.optimize.curve_fit`.

Examples

```
>>> from temul.dummy_data import sine_wave_sublattice
>>> import temul.api as tml
>>> sublattice = sine_wave_sublattice()
>>> sublattice.construct_zone_axes(atom_plane_tolerance=1)
>>> sublattice.plot()
>>> sampling = 0.05 #  nm/pix
>>> cmap='bwr'
>>> curvature_map = tml.calculate_atom_plane_curvature(sublattice,
...         zone_vector_index=0, sampling=sampling, units='nm', cmap=cmap)
```

Just compute several atom planes:

```
>>> curvature_map = tml.calculate_atom_plane_curvature(sublattice, 0,
...         atom_planes=(0,3), sampling=sampling, units='nm', cmap=cmap)
```

You can also provide initial fitting estimations via scipy's curve_fit:

```
>>> p0 = [2, 1, 1, 15]
>>> kwargs = {'p0': p0}
>>> curvature_map, fittings = tml.calculate_atom_plane_curvature(
...     sublattice, zone_vector_index=0, atom_planes=(0,3),
...     sampling=sampling, units='nm', cmap=cmap, return_fits=True,
...     **kwargs)
```

Returns

Return type Curvature Map as a Hyperspy Signal2D

References

`temul.topotem.lattice_structure_tools.`**sine_wave_function_strain_gradient**(*x*, *a*, *b*, *c*, *d*)

Element Tools

`temul.element_tools.`**atomic_radii_in_pixels**(*sampling*, *element_symbol*)

Get the atomic radius of an element in pixels, scaled by an image sampling

Parameters

- **sampling** (*float, default None*) – sampling of an image in units of nm/pix
- **element_symbol** (*string, default None*) – Symbol of an element from the periodic table of elements

Returns

Return type Half the colavent radius of the input element in pixels

Examples

```
>>> import atomap.api as am
>>> from temul.element_tools import atomic_radii_in_pixels
>>> image = am.dummy_data.get_simple_cubic_signal()
```

pretend it is a 5x5 nm image

```
>>> image_sampling = 5/len(image.data) # units nm/pix
>>> radius_pix_Mo = atomic_radii_in_pixels(image_sampling, 'Mo')
>>> radius_pix_Mo
4.62
```

```
>>> radius_pix_S = atomic_radii_in_pixels(image_sampling, 'C')
>>> radius_pix_S
2.28
```

temul.element_tools.**combine_element_lists**(*lists*)

Reduce multiple element_lists into one list of strings from a list of lists, useful for the `Model Refiner flattened_element_list`.

temul.element_tools.**get_and_return_element**(*element_symbol*)

From the elemental symbol, e.g., 'H' for Hydrogen, provides Hydrogen as a periodictable.core.Element object for further use.

Parameters **element_symbol** (*string*) – Symbol of an element from the periodic table of elements e.g., "C", "H"

Returns

Return type A periodictable.core.Element object

Examples

```
>>> from temul.element_tools import get_and_return_element
>>> Moly = get_and_return_element(element_symbol='Mo')
>>> print(Moly.symbol)
Mo
```

```
>>> print(Moly.covalent_radius)
1.54
```

```
>>> print(Moly.number)
42
```

temul.element_tools.**get_individual_elements_from_element_list**(*element_list*, *split_symbol=['_', '.']*)

Examples

Single list

```
>>> import temul.element_tools as tml_el
>>> element_list = ['Mo_0', 'Ti_3', 'Ti_9', 'Ge_2']
>>> get_individual_elements_from_element_list(
...     element_list, split_symbol=['_', '.'])
['Ge', 'Mo', 'Ti']
```

some complex atomic_columns

```
>>> element_list = ['Mo_0', 'Ti_3.Re_7', 'Ti_9.Re_3', 'Ge_2']
>>> get_individual_elements_from_element_list(
...     element_list, split_symbol=['_', '.'])
['Ge', 'Mo', 'Re', 'Ti']
```

multiple lists in element_list. Used in Model_Refiner if you have more than one sublattice.

```
>>> element_list = [['Ti_7_0', 'Ti_9.Re_3', 'Ge_2'], ['B_9', 'B_2.Fe_8']]
>>> get_individual_elements_from_element_list(
...     element_list, split_symbol=['_', '.'])
['B', 'Fe', 'Ge', 'Re', 'Ti']
```

temul.element_tools.**split_and_sort_element**(*element*, *split_symbol=['_', '.']*)

Extracts info from input atomic column element configuration Split an element and its count, then sort the element for use with other functions.

Parameters

- **element** (*string, default None*) – element species and count must be separated by the first string in the split_symbol list. separate elements must be separated by the second string in the split_symbol list.
- **split_symbol** (*list of strings, default ['_', '.']*) – The symbols used to split the element into its name and count. The first string '_' is used to split the name and count of the element. The second string is used to split different elements in an atomic column configuration.

Returns

- *list of a list with element_split, element_name, element_count, and*
- *element_atomic_number.*
- *See examples below*

Examples

```
>>> from temul.element_tools import split_and_sort_element
```

simple atomic column

```
>>> split_and_sort_element(element='S_1')
[['S', '1'], 'S', 1, 16]]
```

complex atomic column

```
>>> info = split_and_sort_element(element='O_6.Mo_3.Ti_5')
```

Intensity Tools

temul.intensity_tools.**get_pixel_count_from_image_slice**(*atom*, *image_data*, *percent_to_nn=0.4*)

Fid the number of pixels in an area when calling _get_image_slice_around_atom()

Parameters

- **atom** (*atomap.AtomPosition*) –
- **image_data** (*Numpy 2D array*) –
- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.

Returns

Return type The number of pixels in the image_slice

Examples

```
>>> from temul.intensity_tools import get_pixel_count_from_image_slice
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> sublattice.find_nearest_neighbors()
>>> atom0 = sublattice.atom_list[0]
>>> pixel_count = get_pixel_count_from_image_slice(atom0, sublattice.image)
```

temul.intensity_tools.**get_sublattice_intensity**(*sublattice*, *intensity_type='max'*, *remove_background_method=None*, *background_sub=None*, *num_points=3*, *percent_to_nn=0.4*, *mask_radius=None*)

Finds the intensity for each atomic column using either max, mean, min, total or all of them at once.

The intensity values are taken from the area defined by percent_to_nn.

Results are stored in each Atom_Position object as amplitude_max_intensity, amplitude_mean_intensity, amplitude_min_intensity and/or amplitude_total_intensity which can most easily be accessed through the sublattice object. See the examples in get_atom_column_amplitude_max_intensity.

Parameters

- **sublattice** (*sublattice object*) – The sublattice whose intensities you are finding.
- **intensity_type** (*string, default "max"*) – Determines the method used to find the sublattice intensities. The available methods are “max”, “mean”, “min”, “total” and “all”.
- **remove_background_method** (*string, default None*) – Determines the method used to remove the background_sub intensities from the image. Options are “average” and “local”.
- **background_sub** (*sublattice object, default None*) – The sublattice used if remove_background_method is used.
- **num_points** (*int, default 3*) – If remove_background_method=”local”, num_points is the number of nearest neighbour values averaged from background_sub

- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float*) – Radius of the atomic column in pixels. If chosen, percent_to_nn must be None.

Returns

Return type Numpy array, shape depending on intensity_type

Examples

```
>>> from temul.intensity_tools import get_sublattice_intensity
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> sublattice.find_nearest_neighbors()
>>> intensities_all = get_sublattice_intensity(
...     sublattice=sublattice,
...     intensity_type="all",
...     remove_background_method=None,
...     background_sub=None)
```

Return the summed intensity around the atom:

>>> intensities_total = get_sublattice_intensity(# ... sublattice=sublattice, # ... intensity_type=”total”, # ... remove_background_method=None, # ... background_sub=None)

Return the max intensity around the atom with local background subtraction:

>>> intensities_total_local = get_sublattice_intensity(# ... sublattice=sublattice, # ... intensity_type=”max”, # ... remove_background_method=”local”, # ... background_sub=sublattice)

Return the maximum intensity around the atom with average background subtraction:

```
>>> intensities_max_average = get_sublattice_intensity(
...     sublattice=sublattice,
...     intensity_type="max",
...     remove_background_method="average",
...     background_sub=sublattice)
```

temul.intensity_tools.**remove_average_background**(*sublattice*, *intensity_type*, *background_sub*, *percent_to_nn=0.4*, *mask_radius=None*)

Remove the average background from a sublattice intensity using a background sublattice.

Parameters

- **sublattice** (*sublattice object*) – The sublattice whose intensities are of interest.
- **intensity_type** (*string*) – Determines the method used to find the sublattice intensities. The available methods are “max”, “mean”, “min” and “all”.
- **background_sub** (*sublattice object*) – The sublattice used to find the average background.
- **percent_to_nn** (*float, default 0.4*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float*) – Radius of the atomic column in pixels. If chosen, percent_to_nn must be None.

Returns

Return type Numpy array, shape depending on `intensity_type`

Examples

```
>>> from temul.intensity_tools import remove_average_background
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> # import atomap.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> sublattice.find_nearest_neighbors()
>>> intensities_all = remove_average_background(
...     sublattice, intensity_type="all",
...     background_sub=sublattice)
>>> intensities_max = remove_average_background(
...     sublattice, intensity_type="max",
...     background_sub=sublattice)
```

temul.intensity_tools.**remove_local_background**(*sublattice*, *background_sub*, *intensity_type*, *num_points=3*, *percent_to_nn=0.4*, *mask_radius=None*)

Remove the local background from a sublattice intensity using a background sublattice.

Parameters

- **sublattice** (*sublattice object*) – The sublattice whose intensities are of interest.
- **intensity_type** (*string*) – Determines the method used to find the sublattice intensities. The available methods are "max", "mean", "min", "total" and "all".
- **background_sub** (*sublattice object*) – The sublattice used to find the local backgrounds.
- **num_points** (*int, default 3*) – The number of nearest neighbour values averaged from background_sub
- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float*) – Radius of the atomic column in pixels. If chosen, `percent_to_nn` must be None.

Returns

Return type Numpy array, shape depending on `intensity_type`

Examples

```
>>> from temul.intensity_tools import remove_local_background
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> sublattice.find_nearest_neighbors()
>>> intensities_max = remove_local_background(
...     sublattice, intensity_type="max",
...     background_sub=sublattice)
```

Model Creation

temul.model_creation.**assign_z_height**(*sublattice*, *lattice_type*, *material*)

temul.model_creation.**assign_z_height_to_sublattice**(*sublattice*, *z_bond_length*, *material=None*, *fractional_coordinates=True*, *atom_layout='bot'*)

Set the z_heights for each atom position in a sublattice.

Parameters **sublattice** (*Atomap Sublattice object*) –

Examples

See example_scripts : Model Creation Example

temul.model_creation.**auto_generate_sublattice_element_list**(*material_type*, *elements='Au'*, *max_number_atoms_z=10*)

Example

```
>>> element_list = auto_generate_sublattice_element_list(
...                     material_type='nanoparticle',
...                     elements='Au',
...                     max_number_atoms_z=10)
```

temul.model_creation.**change_sublattice_atoms_via_intensity**(*sublattice*, *image_diff_array*, *darker_or_brighter*, *element_list*, *verbose=False*)

Change the elements in a sublattice object to a higher or lower combined atomic (Z) number. The aim is to change the sublattice elements so that the experimental image agrees with the simulated image in a realistic manner. See image_difference_intensity()

Get the index in sublattice from the image_difference_intensity() output, which is the image_diff_array input here. Then, depending on whether the image_diff_array is for atoms that should be brighter or darker, set a new element to that sublattice atom_position

Parameters

- **sublattice** (*Atomap Sublattice object*) – the elements of this sublattice will be changed
- **image_diff_array** (*Numpy 2D array*) – Contains (p, x, y, intensity) where p = index of Atom_Position in sublattice x = Atom_Position.pixel_x y = Atom_Position.pixel_y intensity = calculated intensity of atom in sublattice.image
- **darker_or_brighter** (*int*) – if the element should have a lower combined atomic Z number, darker_or_brighter = 0. if the element should have a higher combined atomic Z number, darker_or_brighter = 1 In other words, the image_diff_array will change the given sublattice elements to darker or brighter spots by choosing 0 and 1, respectively.
- **element_list** (*list*) – list of element configurations
- **verbose** (*bool, default False*) – If set to True then atom warnings will be printed as output.

Examples

```
>>> from temul.model_creation import change_sublattice_atoms_via_intensity
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> # import atomap.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> for i in range(0, len(sublattice.atom_list)):
...     sublattice.atom_list[i].elements = 'Mo_1'
...     sublattice.atom_list[i].z_height = [0.5]
>>> element_list = ['H_0', 'Mo_1', 'Mo_2']
>>> image_diff_array = np.array([[5, 2, 2, 20],[1, 2, 4, 7]])
>>> # This will change the 5th atom in the sublattice to a lower atomic Z
>>> # number, i.e., 'H_0' in the given element_list
>>> change_sublattice_atoms_via_intensity(sublattice=sublattice,
...                                image_diff_array=image_diff_array,
...                                darker_or_brighter=0,
...                                element_list=element_list)
Changing some atoms
```

temul.model_creation.**change_sublattice_pseudo_inplace**(*new_atom_positions*, *old_sublattice*)

Create and return a new Sublattice object which is a copy of old_sublattice except with extra atom positions set with new_atom_positions.

Parameters

- **new_atom_positions** (*NumPy array*) – In the form [[x0, y0], [x1, y1], [x2, y2], …]
- **old_sublattice** (*Atomap Sublattice*) –

Examples

```
>>> from temul.model_creation import change_sublattice_pseudo_inplace
>>> from temul.external.atomap_devel_012.api import Sublattice
>>> import numpy as np
>>> atom_positions = [[1, 5], [2, 4]]
>>> image_data = np.random.random((7, 7))
>>> sublattice_A = Sublattice(atom_positions, image_data)
>>> new_atom_positions = [[4, 3], [6, 6]]
>>> sublattice_B = change_sublattice_pseudo_inplace(
...     new_atom_positions, sublattice_A)
>>> sublattice_B.atom_positions
[[1, 2, 4, 6], [5, 4, 3, 6]]
```

```
>>> sublattice_A.atom_positions
[[1, 2], [5, 4]]
```

It also copies information such as elements

```
>>> sublattice_A.atom_list[0].elements = 'Ti_1'
>>> sublattice_A = change_sublattice_pseudo_inplace(
...     new_atom_positions, sublattice_A)
>>> sublattice_A.atom_list[0].elements
'Ti_1'
```

Returns

- *An Atomap Sublattice object that combines the new_atom_positions with*
- *the information from the old_sublattice.*

temul.model_creation.**compare_count_atoms_in_sublattice_list**(*counter_list*)

Compare the count of atomap elements in two counter_lists gotten by count_atoms_in_sublattice_list()

If the counters are the same, then the original atom_lattice is the same as the refined atom_lattice. It will return the boolean value True. This can be used to stop refinement loops if neccessary.

Parameters **counter_list** (*list of two Counter objects*) –

Returns

- *boolean True if the counters are equal,*
- *boolean False is the counters are not equal.*

Examples

```
>>> from temul.model_creation import (
...     count_atoms_in_sublattice_list,
...     compare_count_atoms_in_sublattice_list)
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> atom_lattice = dummy_data.get_simple_atom_lattice_two_sublattices()
>>> sub1 = atom_lattice.sublattice_list[0]
>>> sub2 = atom_lattice.sublattice_list[1]
```

```
>>> for i in range(0, len(sub1.atom_list)):
...     sub1.atom_list[i].elements = 'Ti_2'
>>> for i in range(0, len(sub2.atom_list)):
...     sub2.atom_list[i].elements = 'Cl_1'
>>> added_atoms = count_atoms_in_sublattice_list(
...     sublattice_list=[sub1, sub2],
...     filename=atom_lattice.name)
```

```
>>> at_lat_before = dummy_data.get_simple_atom_lattice_two_sublattices()
>>> no_added_atoms = count_atoms_in_sublattice_list(
...     sublattice_list=at_lat_before.sublattice_list,
...     filename=at_lat_before.name)
```

```
>>> compare_count_atoms_in_sublattice_list([added_atoms, no_added_atoms])
False
```

This function can also be used to stop a refinement loop by using an if break loop: # >>> if compare_count_atoms_in_sublattice_list(counter_list) is True: # >>> break

temul.model_creation.**convert_numpy_z_coords_to_z_height_string**(*z_coords*)

Convert from the output of return_z_coordinates(), which is a 1D numpy array, to a long string, with which I have set up sublattice.atom_list[i].z_height.

Examples

>>> from temul.model_creation import (# ... return_z_coordinates, # ... convert_numpy_z_coords_to_z_height_string) # >>> Au_NP_z_coord = return_z_coordinates(z_thickness=20, # z_bond_length=1.5) # >>> Au_NP_z_height_string = convert_numpy_z_coords_to_z_height_string(# ... Au_NP_z_coord)

temul.model_creation.**correct_background_elements**(*sublattice*)

temul.model_creation.**count_all_individual_elements**(*individual_element_list*, *dataframe*)

Perform count_element_in_pandas_df() for all elements in a dataframe. Specify the elements you wish to count in the individual_element_list.

Parameters

- **individual_element_list** (*list*) –
- **dataframe** (*pandas dataframe*) – The dataframe must have column headers as elements or element configurations

Returns

Return type dict object with each key=individual element and value=element count

Examples

```
>>> import pandas as pd
>>> from temul.model_creation import count_all_individual_elements
>>> header = ['Se_1', 'Mo_1']
>>> counts = [[9, 4], [8, 6]]
>>> df = pd.DataFrame(data=counts, columns=header)
>>> individual_element_list = ['Mo', 'S']
>>> element_count = count_all_individual_elements(
...     individual_element_list, dataframe=df)
```

temul.model_creation.**count_atoms_in_sublattice_list**(*sublattice_list*, *filename=None*)

Count the elements in a list of Atomap sublattices

Parameters

- **sublattice_list** (*list of atomap sublattice(s)*) –
- **filename** (*string, default None*) – name with which the image will be saved

Returns

Return type Counter object

Examples

```
>>> from temul.model_creation import count_atoms_in_sublattice_list
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> # import atomap.dummy_data as dummy_data
>>> atom_lattice = dummy_data.get_simple_atom_lattice_two_sublattices()
>>> sub1 = atom_lattice.sublattice_list[0]
>>> sub2 = atom_lattice.sublattice_list[1]
```

```
>>> for i in range(0, len(sub1.atom_list)):
...     sub1.atom_list[i].elements = 'Ti_2'
>>> for i in range(0, len(sub2.atom_list)):
...     sub2.atom_list[i].elements = 'Cl_1'
>>> added_atoms = count_atoms_in_sublattice_list(
...     sublattice_list=[sub1, sub2])
```

Compare before and after

```
>>> at_lat_before = dummy_data.get_simple_atom_lattice_two_sublattices()
>>> no_added_atoms = count_atoms_in_sublattice_list(
...    sublattice_list=at_lat_before.sublattice_list)
```

temul.model_creation.**count_element_in_pandas_df**(*element*, *dataframe*)

Count the number of a single element in a dataframe

Parameters

- **element** (*string*) – element symbol
- **dataframe** (*pandas dataframe*) – The dataframe must have column headers as elements or element configurations

Returns

Return type Counter object

Examples

```
>>> import pandas as pd
>>> from temul.model_creation import count_element_in_pandas_df
>>> header = ['Se_1', 'Mo_1', 'S_2']
>>> counts = [[9, 4, 3], [8, 6, 2]]
>>> df = pd.DataFrame(data=counts, columns=header)
>>> Se_count = count_element_in_pandas_df(element='Se', dataframe=df)
>>> Se_count
Counter({0: 9, 1: 8})
```

temul.model_creation.**create_dataframe_for_cif**(*sublattice_list*, *element_list*)

Outputs a dataframe from the inputted sublattice(s), which can then be input to temul.io.write_cif_from_dataframe().

Parameters

- **sublattice_list** (*list of Atomap Sublattice objects*) –
- **element_list** (*list of strings*) – Each string must be an element symbol from the periodic table.

temul.model_creation.**find_middle_and_edge_intensities**(*sublattice*, *element_list*, *standard_element*, *scaling_exponent*, *largest_element_intensity=None*, *split_symbol=['_', '.']*)

Create a list which represents the peak points of the intensity distribution for each atom.

works for nanoparticles as well, doesn't matter what scaling_exponent you use for nanoparticle. Figure this out!

If the max_element_intensity is set, then the program assumes that the standard element is the largest available element combination, and scales the middle and limit intensity lists so that the middle_intensity_list[-1] == max_element_intensity

temul.model_creation.**find_middle_and_edge_intensities_for_background**(*elements_from_sub1*, *elements_from_sub2*, *sub1_mode*, *sub2_mode*, *element_list_sub1*, *element_list_sub2*, *middle_intensity_list_sub1*, *middle_intensity_list_sub2*)

temul.model_creation.**get_max_number_atoms_z**(*sublattice*)

temul.model_creation.**get_most_common_sublattice_element**(*sublattice*, *info='element'*)

Find the most common element configuration in a sublattice.

Parameters **sublattice** (*Atomap Sublattice object*) –

Returns

Return type Most common element configuration in the sublattice

Examples

```
>>> from temul.model_creation import get_most_common_sublattice_element
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> # import atomap.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> for i, atom in enumerate(sublattice.atom_list):
...     if i % 3 == 0:
...         atom.elements = 'Ti_3'
...         atom.z_height = '0.3, 0.6, 0.9'
...     else:
...         atom.elements = 'Ti_2'
...         atom.z_height = '0.3, 0.6'
>>> get_most_common_sublattice_element(sublattice, info='element')
'Ti_2'
```

```
>>> get_most_common_sublattice_element(sublattice, info='z_height')
'0.3, 0.6'
```

temul.model_creation.**get_positions_from_sublattices**(*sublattice_list*)

temul.model_creation.**image_difference_intensity**(*sublattice*, *sim_image*, *element_list*, *filename=None*, *percent_to_nn=0.4*, *mask_radius=None*, *change_sublattice=False*, *verbose=False*)

Find the differences in a sublattice's atom_position intensities. Change the elements of these atom_positions depending on this difference of intensities.

The aim is to change the sublattice elements so that the experimental image agrees with the simulated image in a realistic manner.

Parameters

- **sublattice** (*Atomap Sublattice object*) – Elements of this sublattice will be refined

- **sim_image** (*HyperSpy 2D signal*) – The image you wish to refine with, usually an image simulation of the sublattice.image
- **element_list** (*list*) – list of element configurations, used for refinement
- **filename** (*string, default None*) – name with which the image will be saved
- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float, default None*) – Radius of the mask around each atom. If this is not set, the radius will be the distance to the nearest atom in the same sublattice times the percent_to_nn value. Note: if mask_radius is not specified, the Atom_Position objects must have a populated nearest_neighbor_list.
- **change_sublattice** (*bool, default False*) – If change_sublattice is set to True, all incorrect element assignments will be corrected inplace.
- **verbose** (*bool, default False*) – If set to True then atom warnings will be printed as output.

Example

```
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> # import atomap.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> sim_image = dummy_data.get_simple_cubic_with_vacancies_signal()
>>> for i in range(0, len(sublattice.atom_list)):
...     sublattice.atom_list[i].elements = 'Mo_1'
...     sublattice.atom_list[i].z_height = [0.5]
>>> element_list = ['H_0', 'Mo_1', 'Mo_2']
>>> image_difference_intensity(sublattice=sublattice,
...                            sim_image=sim_image,
...                            element_list=element_list)
```

with some image noise and plotting the images

```
>>> sublattice = dummy_data.get_simple_cubic_sublattice(
...     image_noise=True)
>>> sim_image = dummy_data.get_simple_cubic_with_vacancies_signal()
>>> for i in range(0, len(sublattice.atom_list)):
...     sublattice.atom_list[i].elements = 'Mo_1'
...     sublattice.atom_list[i].z_height = [0.5]
>>> element_list = ['H_0', 'Mo_1', 'Mo_2']
>>> image_difference_intensity(sublattice=sublattice,
...                            sim_image=sim_image,
...                            element_list=element_list)
```

temul.model_creation.**image_difference_position**(*sublattice*, *sim_image*, *pixel_threshold*, *comparison_sublattice_list=None*, *filename=None*, *percent_to_nn=0.4*, *mask_radius=None*, *num_peaks=5*, *inplace=True*, *verbose=False*)

Find new atomic coordinates by comparing experimental to simulated image. Changes the sublattice inplace using change_sublattice_pseudo_inplace.

The aim is to change the sublattice elements so that the experimental image agrees with the simulated image in a realistic manner. See also image_difference_intensity function and the Model_Refiner class.

Parameters

- **sublattice** (*Atomap sublattice object*) –
- **sim_image** (*simulated image used for comparison with sublattice image*) –
- **pixel_threshold** (*int*) – minimum pixel distance from current sublattice atoms. If the new atomic coordinates are greater than this distance, they will be created. Choose a pixel_threshold that will not create new atoms in unrealistic positions.
- **filename** (*string, default None*) – name with which the image will be saved
- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float, default None*) – Radius of the mask around each atom. If this is not set, the radius will be the distance to the nearest atom in the same sublattice times the percent_to_nn value. Note: if mask_radius is not specified, the Atom_Position objects must have a populated nearest_neighbor_list.
- **num_peaks** (*int, default 5*) – number of new atoms to add
- **add_sublattice** (*bool, default False*) – If set to True, a new sublattice will be created and returned. The reason it is set to False is so that one can check if new atoms would be added with the given parameters.
- **sublattice_name** (*string, default 'sub_new'*) – the outputted sublattice object name and sublattice.name the new sublattice will be given
- **inplace** (*bool, default True*) – If set to True, the input sublattice will be changed inplace and the sublattice returned. If set to False, these changes will be output only to a new sublattice.
- **verbose** (*bool*) – Setting to True will print out some info as the function is running.

Returns

- Atomap Sublattice object if inplace=False. See the inplace
- *parameter for details.*

Examples

```
>>> from temul.model_creation import (image_difference_position,
...                                   change_sublattice_pseudo_inplace)
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_with_vacancies_sublattice(
...                                               image_noise=True)
>>> sim_image = dummy_data.get_simple_cubic_signal()
>>> for i in range(0, len(sublattice.atom_list)):
...         sublattice.atom_list[i].elements = 'Mo_1'
...         sublattice.atom_list[i].z_height = '0.5'
>>> old_atoms = len(sublattice.atom_list)
```

```
>>> sublattice = image_difference_position(
...     sublattice=sublattice, sim_image=sim_image, pixel_threshold=10,
...     percent_to_nn=None, mask_radius=5, num_peaks=5, inplace=True,
...     verbose=False)
>>> new_atoms = len(sublattice.atom_list)
```

One can now sort these atom positions into elements.

temul.model_creation.**image_difference_position_new_sub**(*sublattice_list*, *sim_image*, *pixel_threshold*, *filename=None*, *percent_to_nn=0.4*, *mask_radius=None*, *num_peaks=5*, *add_sublattice=False*, *sublattice_name='sub_new'*)

Find new atomic coordinates by comparing experimental to simulated image. Create a new sublattice to store the new atomic coordinates.

The aim is to change the sublattice elements so that the experimental image agrees with the simulated image in a realistic manner.

Parameters

- **sublattice_list** (*list of atomap sublattice objects*) –
- **sim_image** (*simulated image used for comparison with sublattice image*) –
- **pixel_threshold** (*int*) – minimum pixel distance from current sublattice atoms. If the new atomic coordinates are greater than this distance, they will be created. Choose a pixel_threshold that will not create new atoms in unrealistic positions.
- **filename** (*string, default None*) – name with which the image will be saved
- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float, default None*) – Radius of the mask around each atom. If this is not set, the radius will be the distance to the nearest atom in the same sublattice times the percent_to_nn value. Note: if mask_radius is not specified, the Atom_Position objects must have a populated nearest_neighbor_list.
- **num_peaks** (*int, default 5*) – number of new atoms to add
- **add_sublattice** (*bool, default False*) – If set to True, a new sublattice will be created and returned. The reason it is set to False is so that one can check if new atoms would be added with the given parameters.
- **sublattice_name** (*string, default 'sub_new'*) – the outputted sublattice object name and sublattice.name the new sublattice will be given

Returns

Return type Atomap Sublattice object

Examples

```
>>> from temul.model_creation import image_difference_position
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> # import atomap.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_with_vacancies_sublattice(
...                                             image_noise=True)
>>> sim_image = dummy_data.get_simple_cubic_signal()
>>> for i in range(0, len(sublattice.atom_list)):
...         sublattice.atom_list[i].elements = 'Mo_1'
...         sublattice.atom_list[i].z_height = '0.5'
>>> # Check without adding a new sublattice
>>> image_difference_position_new_sub(sublattice_list=[sublattice],
```

(continues on next page)

(continued from previous page)

```
...                             sim_image=sim_image,
...                             pixel_threshold=1,
...                             percent_to_nn=None,
...                             mask_radius=5,
...                             num_peaks=5,
...                             add_sublattice=False)
>>> # Add a new sublattice
>>> # if you have problems with mask_radius, increase it!
>>> # Just a gaussian fitting issue, could turn it off!
>>> sub_new = image_difference_position_new_sub(
...                                     sublattice_list=[sublattice],
...                                     sim_image=sim_image,
...                                     pixel_threshold=10,
...                                     num_peaks=5,
...                                     add_sublattice=True)
New Atoms Found! Adding to a new sublattice
```

temul.model_creation.**print_sublattice_elements**(*sublattice*, *number_of_lines='all'*)

temul.model_creation.**return_xyz_coordinates**(*x*, *y*, *z_thickness*, *z_bond_length*, *number_atoms_z=None*, *fractional_coordinates=True*, *atom_layout='bot'*)

Produce xyz coordinates for an xy coordinate given the z-dimension information.

Parameters

- **x** (*float*) – atom position coordinates.
- **y** (*float*) – atom position coordinates.
- **return_z_coordinates()** (*for other parameters see*) –

Returns

Return type 2D numpy array with columns x, y, z

Examples

>>> from temul.model_creation import return_xyz_coordinates # >>> x, y = 2, 3 # >>> atom_coords = return_xyz_coordinates(x, y, # … z_thickness=10, # … z_bond_length=1.5, # … number_atoms_z=5)

temul.model_creation.**return_z_coordinates**(*z_thickness*, *z_bond_length*, *number_atoms_z=None*, *max_number_atoms_z=None*, *fractional_coordinates=True*, *atom_layout='bot'*)

Produce fractional z-dimension coordinates for a given thickness and bond length.

Parameters

- **z_thickness** (*number*) – Size (Angstrom) of the z-dimension.
- **z_bond_length** (*number*) – Size (Angstrom) of the bond length between adjacent atoms in the z-dimension.
- **number_atoms_z** (*integer, default None*) – number of atoms in the z-dimension. If this is set to an interger value, it will override the use of z_thickness.
- **centered_atoms** (*bool, default True*) – If set to True, the z_coordinates will be centered about 0.5. If set to False, the z_coordinate will start at 0.0.

Returns

Return type 1D numpy array

Examples

>>> from temul.model_creation import return_z_coordinates # >>> Au_NP_z_coord = return_z_coordinates(z_thickness=20, # z_bond_length=1.5)

temul.model_creation.**scaling_z_contrast**(*numerator_sublattice*, *numerator_element*, *denominator_sublattice*, *denominator_element*, *intensity_type*, *method*, *remove_background_method*, *background_sublattice*, *num_points*, *percent_to_nn=0.4*, *mask_radius=None*, *split_symbol='_'*)

Find new atomic coordinates by comparing experimental to simulated image. Create a new sublattice to store the new atomic coordinates.

The aim is to change the sublattice elements so that the experimental image agrees with the simulated image in a realistic manner. See [1]_.

Parameters

- **numerator_sublattice** (*Sublattice object*) – Sublattice from which the numerator intensity will be calculated.
- **numerator_element** (*string*) – Element from which the numerator atomic number will be taken.
- **denominator_sublattice** (*Sublattice object*) – Sublattice from which the denominator intensity will be calculated.
- **denominator_element** (*string*) – Element from which the denominator atomic number will be taken.
- **intensity_type** (*string*) – Determines the method used to find the sublattice intensities. The available methods are “max”, “mean”, “min”, “total” and “all”.
- **method** (*string*) – Method used to calculate the intensity of the sublattices. Options are “mean” or “mode”.
- **remove_background_method** (*string*) – Determines the method used to remove the background_sublattice intensities from the image. Options are “average” and “local”.
- **background_sub** (*Sublattice object*) – The sublattice to be used if remove_background_method is used.
- **num_points** (*int*) – If remove_background_method=”local”, num_points is the number of nearest neighbour values averaged from background_sublattice.
- **percent_to_nn** (*float, default 0.40*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*float*) – Radius of the atomic column in pixels. If chosen, percent_to_nn must be None.
- **split_symbol** (*string, default '_'*) – The symbols used to split the element into its name and count. The first string ‘_’ is used to split the name and count of the element. The second string has not been implemented within this function.

Returns

- *Four floats*
- *scaling_ratio, scaling_exponent, sublattice0_intensity_method,*

- *sublattice1_intensity_method*

References

Examples

```
>>> from temul.model_creation import image_difference_position
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_with_vacancies_sublattice(
...                                                 image_noise=True)
>>> sim_image = dummy_data.get_simple_cubic_signal()
>>> for i in range(0, len(sublattice.atom_list)):
...         sublattice.atom_list[i].elements = 'Mo_1'
...         sublattice.atom_list[i].z_height = '0.5'
```

Check without adding a new sublattice

```
>>> image_difference_position_new_sub(sublattice_list=[sublattice],
...                            sim_image=sim_image,
...                            pixel_threshold=1,
...                            percent_to_nn=None,
...                            mask_radius=5,
...                            num_peaks=5,
...                            add_sublattice=False)
```

Add a new sublattice

```
>>> sub_new = image_difference_position_new_sub(
...                                    sublattice_list=[sublattice],
...                                    sim_image=sim_image,
...                                    pixel_threshold=10,
...                                    num_peaks=5,
...                                    add_sublattice=True)
New Atoms Found! Adding to a new sublattice
```

temul.model_creation.**sort_sublattice_intensities**(*sublattice*, *intensity_type='max'*, *element_list=[]*, *scalar_method='mode'*, *middle_intensity_list=None*, *limit_intensity_list=None*, *remove_background_method=None*, *background_sublattice=None*, *num_points=3*, *intensity_list_real=False*, *percent_to_nn=0.4*, *mask_radius=None*)

Image Simulation Functions

Signal Processing

temul.signal_processing.**calibrate_intensity_distance_with_sublattice_roi**(*image*, *cropping_area*, *separation*, *reference_image=None*, *scalebar_true=False*, *percent_to_nn=0.2*, *mask_radius=None*, *refine=True*, *filename=None*)

Calibrates the intensity of an image by using the brightest sublattice. The mean intensity of that sublattice is set to 1.

Parameters

- **image** (*Hyperspy Signal2D*) – Image you wish to calibrate.
- **cropping_area** (*list of 2 floats*) – The best method of choosing the area is by using the function "choose_points_on_image(image.data)". Choose two points on the image. First point is top left of area, second point is bottom right.
- **separation** (*int, default 8*) – Pixel separation between atoms as used by Atomap.
- **reference_image** (*Hyperspy Signal2D*) – Image with which `image` is compared.
- **scalebar_true** (*bool, default True*) – If set to True, the function assumes that `image.axes_manager` is calibrated to a unit other than pixel.
- **mask_radius** (*int, default None*) – Radius in pixels of the mask.
- **percent_to_nn** (*float, default 0.2*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **refine** (*bool, default False*) – If set to True, the atom positions found for the calibration will be refined.
- **filename** (*str, default None*) – If set to a string, the image will be saved.

Returns

Return type Nothing, but the mean intensity of the brightest sublattice is set to 1.

Examples

```
>>> from temul.dummy_data import get_simple_cubic_signal
>>> import temul.api as tml
>>> import matplotlib.pyplot as plt
>>> image = get_simple_cubic_signal()
>>> image.plot()
>>> crop_a = tml.choose_points_on_image(image.data) #  manually
>>> crop_a = [[10,10],[100,100]] #  use above line if trying yourself!
```

tml.calibrate_intensity_distance_with_sublattice_roi(# image, crop_a, 10)

temul.signal_processing.**compare_two_image_and_create_filtered_image**(*image_to_filter*, *reference_image*, *delta_image_filter*, *max_sigma=6*, *cropping_area=[[0, 0], [50, 50]]*, *separation=8*, *filename=None*, *percent_to_nn=0.4*, *mask_radius=None*, *refine=False*)

Gaussian blur an image for comparison with a reference image. Good for finding the best gaussian blur for a simulation by comparing to an experimental image. See measure_image_errors() and load_and_compare_images().

Parameters

- **image_to_filter** (*Hyperspy Signal2D*) – Image you wish to automatically filter.
- **reference_image** (*Hyperspy Signal2D*) – Image with which `image_to_filter` is compared.
- **delta_image_filter** (*float*) – The increment of the Gaussian sigma used.
- **max_sigma** (*float, default 6*) – The largest (limiting) Gaussian sigma used.
- **cropping_area** (*list of 2 floats, default [[0,0], [50,50]]*) – The best method of choosing the area is by using the function "choose_points_on_image(image.data)". Choose two points on the image. First point is top left of area, second point is bottom right.
- **separation** (*int, default 8*) – Pixel separation between atoms as used by Atomap.
- **filename** (*str, default None*) – If set to a string, the plotted and filtered image will be saved.
- **percent_to_nn** (*float, default 0.4*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*int, default None*) – Radius in pixels of the mask. If set, then set `percent_to_nn=None`.
- **refine** (*bool, default False*) – If set to True, the `calibrate_intensity_distance_with_sublattice_roi` calibration will refine the atom positions for each calibration. May make the function very slow depending on the size of `image_to_filter` and `cropping_area`.

Returns

Return type Hyperspy Signal2D (filtered image) and float (ideal Gaussian sigma)

Examples

```
>>> from scipy.ndimage.filters import gaussian_filter
>>> import temul.example_data as example_data
>>> import matplotlib.pyplot as plt
>>> from temul.signal_processing import (
...     compare_two_image_and_create_filtered_image)
>>> experiment = example_data.load_Se_implanted_MoS2_data() # example
>>> experiment.data = gaussian_filter(experiment.data, sigma=4)
>>> simulation = example_data.load_Se_implanted_MoS2_data()
```

filt_image, ideal_sigma = compare_two_image_and_create_filtered_image(simulation, experiment, 0.25, cropping_area=[[5,5], [200, 200]], separation=11, mask_radius=4, percent_to_nn=None, max_sigma=10)

temul.signal_processing.**crop_image_hs**(*image*, *cropping_area*, *scalebar_true=True*, *filename=None*)
Crop a Hyperspy Signal2D by providing the `cropping_area`. See the example below.

Parameters

- **image** (*Hyperspy Signal2D*) – Image you wish to crop
- **cropping_area** (*list of 2 floats*) – The best method of choosing the area is by using the function "choose_points_on_image(image.data)". Choose two points on the image. First point is top left of area, second point is bottom right.
- **scalebar_true** (*bool, default True*) – If set to True, the function assumes that `image.axes_manager` is calibrated to a unit other than pixel.
- **filename** (*str, default None*) – If set to a string, the images and cropping variables will be saved.

Returns

Return type Hyperspy Signal2D

Examples

```
>>> from temul.dummy_data import get_simple_cubic_signal
>>> import temul.api as tml
>>> import matplotlib.pyplot as plt
>>> image = get_simple_cubic_signal()
>>> image.plot()
```

Choose two points on the image, top left and bottom right of crop

```
>>> cropping_area = tml.choose_points_on_image(image.data)
>>> cropping_area = [[5,5],[50,50]] # use above line if trying yourself!
>>> # image_cropped = tml.crop_image_hs(image, cropping_area, False)
>>> # image_cropped.plot()
```

temul.signal_processing.**distance_vector**(*x1*, *y1*, *x2*, *y2*)

temul.signal_processing.**double_gaussian_fft_filter**(*image*, *fwhm_neg*, *fwhm_pos*, *neg_min=0.9*)
Filter an image with a bandpass-like filter.

Parameters

- **image** (*Hyperspy Signal2D*) –
- **fwhm_neg** (*float*) – Initial guess in pixels of full width at half maximum (fwhm) of inner (negative) and outer (positive) Gaussian to be applied to fft, respectively. Use the visualise_dg_filter function to find the optimium values.
- **fwhm_pos** (*float*) – Initial guess in pixels of full width at half maximum (fwhm) of inner (negative) and outer (positive) Gaussian to be applied to fft, respectively. Use the visualise_dg_filter function to find the optimium values.
- **neg_min** (*float, default 0.9*) – Effective amplitude of the negative Gaussian.

Examples

```
>>> import temul.api as tml
>>> from temul.example_data import load_Se_implanted_MoS2_data
>>> image = load_Se_implanted_MoS2_data()
```

Use the visualise_dg_filter to find suitable FWHMs

```
>>> tml.visualise_dg_filter(image)
```

then use these values to carry out the double_gaussian_fft_filter

```
>>> filtered_image = tml.double_gaussian_fft_filter(image, 50, 150)
>>> image.plot()
>>> filtered_image.plot()
```

temul.signal_processing.**double_gaussian_fft_filter_optimised**(*image*, *d_inner*, *d_outer*, *delta=0.05*, *sampling=None*, *units=None*, *filename=None*)

Filter an image with an double Gaussian (band-pass) filter. The function will automatically find the optimum magnitude of the negative inner Gaussian.

Parameters

- **image** (*Hyperspy Signal2D*) – Image to be filtered.
- **d_inner** (*float*) – Inner diameter of the FFT spots. Effectively the diameter of the negative Gaussian.
- **d_outer** (*float*) – Outer diameter of the FFT spots. Effectively the diameter of the positive Gaussian.
- **delta** (*float, default 0.05*) – Increment of the automatic filtering with the negative Gaussian. Setting this very small will slow down the function, but too high will not allow the function to calculate negative Gaussians near zero.
- **sampling** (*float*) – image sampling in units/pixel. If set to None, the image.axes_manager will be used.
- **units** (*str*) – Real space units. `sampling` should then be the value of these units/pixel. If set to None, the image.axes_manager will be used.
- **filename** (*str, default None*) – If set to a string, the following files will be plotted and saved: negative Gaussian optimumisation, negative Gaussian, positive Gaussian, double Gaussian, FFT and double Gaussian convolution, filtered image, filtered variables table.

Returns

Return type Hyperspy Signal2D

Examples

```
>>> import temul.example_data as example_data
>>> import temul.api as tml
>>> image = example_data.load_Se_implanted_MoS2_data()
>>> image.plot()
>>> filtered_image = tml.double_gaussian_fft_filter(image, 7.48, 14.96)
>>> filtered_image.plot()
```

temul.signal_processing.**fit_1D_gaussian_to_data**(*xdata*, *amp*, *mu*, *sigma*)

Fitting function for a single 1D gaussian distribution

Parameters

- **xdata** (*numpy 1D array*) – values input as the x coordinates of the gaussian distribution
- **amp** (*float*) – amplitude of the gaussian in y-axis
- **mu** (*float*) – mean value of the gaussianin x-axis, corresponding to y-axis amplitude.
- **sigma** (*float*) – standard deviation of the gaussian distribution

Returns

Return type gaussian distibution of xdata array

Examples

```
>>> from temul.signal_processing import (
...     get_xydata_from_list_of_intensities,
...     fit_1D_gaussian_to_data)
>>> amp, mu, sigma = 10, 10, 0.5
>>> sub1_inten = np.random.normal(mu, sigma, 1000)
>>> xdata, ydata = get_xydata_from_list_of_intensities(sub1_inten,
...     hist_bins=50)
>>> gauss_fit_01 = fit_1D_gaussian_to_data(xdata, amp, mu, sigma)
```

temul.signal_processing.**get_cell_image**(*s*, *points_x*, *points_y*, *method='Voronoi'*, *max_radius='Auto'*, *reduce_func=<function amin>*, *show_progressbar=True*)

The same as atomap's integrate, except instead of summing the region around an atom, it removes the value from all pixels in that region. For example, with this you can remove the local background intensity by setting reduce_func=np.min.

Parameters

- **reduce_func** (*ufunc, default np.min*) – function used to reduce the pixel values around each atom to a float.
- **function.** (*For the other parameters see Atomap's integrate*) –

Returns

Return type Numpy array with the same shape as s

Examples

```
>>> from temul.dummy_data import (
...     get_simple_cubic_sublattice_positions_on_vac)
>>> from temul.signal_processing import get_cell_image
>>> sublattice = get_simple_cubic_sublattice_positions_on_vac()
>>> cell_image = get_cell_image(sublattice.image, sublattice.x_position,
...     sublattice.y_position)
```

Plot the `cell_image` which shows, in this case, the background intensity

```
>>> import matplotlib.pyplot as plt
>>> im = plt.imshow(cell_image)
```

Convert it to a Hyperspy Signal2D object:

```
>>> import hyperspy.api as hs
>>> cell_image = hs.signals.Signal2D(cell_image)
>>> cell_image.plot()
```

temul.signal_processing.**get_fitting_tools_for_plotting_gaussians**(*element_list*, *scaled_middle_intensity_list*, *scaled_limit_intensity_list*, *fit_bright_first=True*, *gaussian_amp=5*, *gauss_sigma_division=100*)

Creates a list of parameters and details for fitting the intensities of a sublattice with multiple Gaussians.

temul.signal_processing.**get_scaled_middle_limit_intensity_list**(*sublattice*, *middle_intensity_list*, *limit_intensity_list*, *sublattice_scalar*)

Returns the middle and limit lists scaled to the actual intensities in the sublattice. Useful for get_fitting_tools_for_plotting_gaussians().

temul.signal_processing.**get_xydata_from_list_of_intensities**(*sublattice_intensity_list*, *hist_bins=100*)

Output x and y data for a histogram of intensities

Parameters

- **sublattice_intensity_list** (*list*) – See get_subattice_intensity() for more information
- **hist_bins** (*int, default 100*) – number of bins to sort the intensities into must be a better way of doing this? maybe automate the binning choice

Returns

- *Two numpy 1D arrays corressponding to the x and y values of a histogram*
- *of the sublattice intensities.*

Examples

```
>>> from temul.signal_processing import get_xydata_from_list_of_intensities
>>> amp, mu, sigma = 10, 10, 0.5
>>> sub1_inten = np.random.normal(mu, sigma, 1000)
>>> xdata, ydata = get_xydata_from_list_of_intensities(sub1_inten,
...     hist_bins=50)
```

temul.signal_processing.**load_and_compare_images**(*imageA*, *imageB*, *filename=None*)

Load two images with hyperspy and compare their mean square error and structural simularity index.

Parameters

- **imageA** (*str, path to file*) – filename of the images to be loaded and compared
- **imageB** (*str, path to file*) – filename of the images to be loaded and compared
- **filename** (*str, default None*) – name with which the image will be saved

Returns

Return type Two floats (mean standard error and structural simularity index)

temul.signal_processing.**make_gaussian**(*size*, *fwhm*, *center=None*)

Make a square gaussian kernel.

Parameters

- **size** (*int*) – The length of a side of the square
- **fwhm** (*float*) – The full-width-half-maximum of the Gaussian, which can be thought of as an effective radius.
- **center** (*array, default None*) – The location of the center of the Gaussian. None will set it to the center of the array.

Returns

Return type 2D Numpy array

Examples

```
>>> from temul.signal_processing import make_gaussian
>>> import matplotlib.pyplot as plt
>>> array = make_gaussian(15, 5)
>>> im = plt.imshow(array)
```

temul.signal_processing.**make_gaussian_pos_neg**(*size*, *fwhm_neg*, *fwhm_pos*, *neg_min=0.9*, *center=None*)

See double_gaussian_fft_filter for details

temul.signal_processing.**mean_and_std_nearest_neighbour_distances**(*sublattice*, *nearest_neighbours=5*, *sampling=None*)

Calculates mean and standard deviation of the distance from each atom to its nearest neighbours.

Parameters

- **sublattice** (*Atomap Sublattice object*) –

- **nearest_neighbours** (*int, default 5*) – The number of nearest neighbours used to calculate the mean distance from an atom. As in atomap, choosing 5 gets the 4 nearest neighbours.
- **sampling** (*float, default None*) – The image sampling in units/pixel. If set to None then the values returned are given in pixels. This may be changed in future versions if Atomap's Sublattice pixel attribute is updated.

Returns Two lists

Return type list of mean distances, list of standard deviations.

Examples

```
>>> from temul.dummy_data import get_simple_cubic_sublattice
>>> from temul.signal_processing import (
...     mean_and_std_nearest_neighbour_distances)
>>> sublattice = get_simple_cubic_sublattice()
>>> mean, std = mean_and_std_nearest_neighbour_distances(sublattice)
>>> mean_scaled, _ = mean_and_std_nearest_neighbour_distances(sublattice,
...     sampling=0.0123)
```

temul.signal_processing.**measure_image_errors**(*imageA*, *imageB*, *filename=None*)

Measure the Mean Squared Error (mse) and Structural Similarity Index (ssm) between two images.

Parameters

- **imageA** (*2D NumPy array, default None*) – Two images between which to measure mse and ssm
- **imageB** (*2D NumPy array, default None*) – Two images between which to measure mse and ssm
- **filename** (*str, default None*) – name with which the image will be saved

Returns

Return type two floats (mse_number, ssm_number)

Examples

```
>>> from temul.dummy_data import get_simple_cubic_signal
>>> imageA = get_simple_cubic_signal().data
>>> imageB = get_simple_cubic_signal(image_noise=True).data
>>> mse_number, ssm_number = measure_image_errors(imageA, imageB)
```

Showing the ideal case of both images being exactly equal:

```
>>> imageB = imageA
>>> mse_number, ssm_number = measure_image_errors(imageA, imageA)
>>> print("MSE: {} and SSM: {}".format(mse_number, ssm_number))
MSE: 0.0 and SSM: 1.0
```

temul.signal_processing.**mse**(*imageA*, *imageB*)

Measure the mean squared error between two images of the same shape.

Parameters

- **imageA** (*array-like*) – The images must be the same shape.
- **imageB** (*array-like*) – The images must be the same shape.

Returns

Return type Mean squared error

temul.signal_processing.**plot_gaussian_fit**(*xdata*, *ydata*, *function*, *amp*, *mu*, *sigma*, *gauss_art='r--'*, *gauss_label='Gauss Fit'*, *plot_data=True*, *data_art='ko'*, *data_label='Data Points'*, *plot_fill=False*, *facecolor='r'*, *alpha=0.5*)

```
>>> from temul.signal_processing import (fit_1D_gaussian_to_data,
...                                      plot_gaussian_fit,
...                                      return_fitting_of_1D_gaussian)
>>> amp, mu, sigma = 10, 10, 0.5
>>> sub1_inten = np.random.normal(mu, sigma, 1000)
>>> xdata, ydata = get_xydata_from_list_of_intensities(sub1_inten,
...     hist_bins=50)
>>> popt_gauss, _ = return_fitting_of_1D_gaussian(
...                     fit_1D_gaussian_to_data,
...                     xdata, ydata, amp, mu, sigma)
>>> plot_gaussian_fit(xdata, ydata, function=fit_1D_gaussian_to_data,
...           amp=popt_gauss[0], mu=popt_gauss[1], sigma=popt_gauss[2],
...           gauss_art='r--', gauss_label='Gauss Fit',
...           plot_data=True, data_art='ko', data_label='Data Points',
...           plot_fill=True, facecolor='r', alpha=0.5)
```

temul.signal_processing.**plot_gaussian_fitting_for_multiple_fits**(*sub_ints_all*, *fitting_tools_all_subs*, *element_list_all_subs*, *marker_list*, *hist_bins=150*, *plotting_style='hist'*, *filename='Fit of Intensities'*, *mpl_cmaps_list=['viridis']*)

plots Gaussian distributions for intensities of a sublattice, over the given parameters (fitting tools).

Examples

sub_ints_all = [sub1_ints, sub2_ints] marker_list = [[‘Sub1’, ‘.’],[‘Sub2’, ‘x’]]

middle_intensity_list_real_sub1, limit_intensity_list_real_sub1 = make_middle_limit_intensity_list_real(
sublattice=sub1, middle_intensity_list=middle_intensity_list_sub1, limit_intensity_list=limit_intensity_list_sub1, method=method, sublattice_scalar=sub1_mode)

middle_intensity_list_real_sub2, limit_intensity_list_real_sub2 = make_middle_limit_intensity_list_real(
sublattice=sub2, middle_intensity_list=middle_intensity_list_sub2, limit_intensity_list=limit_intensity_list_sub2, method=method, sublattice_scalar=sub2_mode)

element_list_all_subs = [element_list_sub1, element_list_sub2]

fitting_tools_all_subs = [

get_fitting_tools_for_plotting_gaussians(element_list_sub1, middle_intensity_list_real_sub1, limit_intensity_list_real_sub1),

get_fitting_tools_for_plotting_gaussians(element_list_sub2, middle_intensity_list_real_sub2, limit_intensity_list_real_sub2)]

plot_gaussian_fitting_for_multiple_fits(sub_ints_all, fitting_tools_all_subs, element_list_all_subs, marker_list, hist_bins=500, filename='Fit of Intensities900')

temul.signal_processing.**remove_image_intensity_in_data_slice**(*atom*, *image_data*, *percent_to_nn=0.5*)

Remove intensity from the area around an atom in a sublattice

temul.signal_processing.**return_fitting_of_1D_gaussian**(*function*, *xdata*, *ydata*, *amp*, *mu*, *sigma*)

Use the initially found centre (mean/mode) value of a sublattice histogram (e.g., Mo_1 in an Mo sublattice) as an input mean for a gaussian fit of the data.

Parameters

- **xdata** (*see scipy.optimize.curve_fit*) –
- **ydata** (*see scipy.optimize.curve_fit*) –
- **amp** (*see fit_1D_gaussian_to_data() for more details*) –
- **mu** (*see fit_1D_gaussian_to_data() for more details*) –
- **sigma** (*see fit_1D_gaussian_to_data() for more details*) –

Returns

- *optimised parameters (popt) and estimated covariance (pcov) of the*
- *fitted gaussian function.*

Examples

```
>>> from temul.signal_processing import (
...     get_xydata_from_list_of_intensities,
...     return_fitting_of_1D_gaussian,
...     fit_1D_gaussian_to_data)
>>> amp, mu, sigma = 10, 10, 0.5
>>> sub1_inten = np.random.normal(mu, sigma, 1000)
>>> xdata, ydata = get_xydata_from_list_of_intensities(sub1_inten,
...     hist_bins=50)
>>> popt_gauss, _ = return_fitting_of_1D_gaussian(
...                     fit_1D_gaussian_to_data,
...                     xdata, ydata,
...                     amp, mu, sigma)
```

temul.signal_processing.**toggle_atom_refine_position_automatically**(*sublattice*, *min_cut_off_percent*, *max_cut_off_percent*, *range_type='internal'*, *method='mode'*, *percent_to_nn=0.05*, *mask_radius=None*, *filename=None*)

Sets the 'refine_position' attribute of each Atom Position in a sublattice using a range of intensities.

Parameters

- **sublattice** (*Atomap Sublattice object*) –

- **min_cut_off_percent** (*float, default None*) – The lower end of the intensity range is defined as `min_cut_off_percent` * `method` value of max intensity list of `sublattice`.
- **max_cut_off_percent** (*float, default None*) – The upper end of the intensity range is defined as `max_cut_off_percent` * `method` value of max intensity list of `sublattice`.
- **range_type** (*str, default 'internal'*) – "internal" provides the `refine_position` attribute for each `Atom Position` as True if the intensity of that Atom Position lies between the lower and upper limits defined by `min_cut_off_percent` and `max_cut_off_percent`. "external" provides the `refine_position` attribute for each `Atom Position` as True if the intensity of that Atom Position lies outside the lower and upper limits defined by `min_cut_off_percent` and `max_cut_off_percent`.
- **method** (*str, default 'mode'*) – The method used to aggregate the intensity of the sublattice positions max intensity list. Options are "mode" and "mean"
- **percent_to_nn** (*float, default 0.05*) – Determines the boundary of the area surrounding each atomic column, as fraction of the distance to the nearest neighbour.
- **mask_radius** (*int, default None*) – Radius in pixels of the mask.
- **filename** (*str, default None*) – If set to a string, the Atomap `refine_position` image will be saved.

Returns

Return type list of the `AtomPosition.refine_position=False` attribute.

Examples

```
>>> from temul.dummy_data import (
...     get_simple_cubic_sublattice_positions_on_vac)
>>> import temul.api as tml
>>> sublattice = get_simple_cubic_sublattice_positions_on_vac()
>>> sublattice.find_nearest_neighbors()
>>> sublattice.plot()
>>> min_cut_off_percent = 0.75
>>> max_cut_off_percent = 1.25
>>> false_list_sublattice =  tml.toggle_atom_refine_position_automatically(
...         sublattice, min_cut_off_percent, max_cut_off_percent,
...         range_type='internal', method='mode', percent_to_nn=0.05)
>>> len(false_list_sublattice) # check how many atoms will not be refined
12
```

Visually check which atoms will not be refined (red dots)

```
>>> sublattice.toggle_atom_refine_position_with_gui()
```

temul.signal_processing.**visualise_dg_filter**(*image*, *d_inner=7.7*, *d_outer=21*, *slider_min=0.1*, *slider_max=300*, *slider_step=0.1*, *plot_lims=(0, 1)*, *figsize=(15, 7)*)

Parameters

- **image** (*Hyperspy Signal2D*) – This image.axes_manager scale should be calibrated.

- **d_inner** (*float, default 7.7*) – Initial 'guess' of full width at half maximum (fwhm) of inner (negative) gaussian to be applied to fft. Can be changed with sliders during visualisation.
- **d_outer** (*float, default 14*) – Initial 'guess' of full width at half maximum (fwhm) of outer (positive) gaussian to be applied to fft. Can be changed with sliders during visualisation.
- **slider_min** (*float, default 0.1*) – Minimum value on sliders
- **slider_max** (*float, default 300*) – Maximum value on sliders
- **slider_step** (*float, default 0.1*) – Step size on sliders
- **plot_lims** (*tuple, default (0, 1)*) – Used to plot a smaller section of the FFT image, which can be useful if the information is very small (far away!). Default plots the whole image.

Examples

```
>>> import temul.api as tml
>>> from temul.example_data import load_Se_implanted_MoS2_data
>>> image = load_Se_implanted_MoS2_data()
>>> tml.visualise_dg_filter(image)
```

Signal Plotting

class temul.signal_plotting.**Sublattice_Hover_Intensity**(*image*, *sublattice*, *sublattice_positions*, *background_sublattice*)

User can hover over sublattice overlaid on STEM image to display the x,y location and intensity of that point.

scaled(*points*)

setup_annotation()

Draw and hide the annotation box.

snap(*x*, *y*)

Return the value in self.tree closest to x, y.

temul.signal_plotting.**color_palettes**(*pallette*)

Color sequences that are useful for creating matplotlib colormaps. Info on "zesty" and other options: venngage.com/blog/color-blind-friendly-palette/ Info on "r_safe": Google: r-plot-color-combinations-that-are-colorblind-accessible

Parameters **palette** (*str*) – Options are "zesty" (4 colours), and "r_safe" (12 colours).

Returns

Return type list of hex colours

temul.signal_plotting.**compare_images_line_profile_one_image**(*image*, *line_profile_positions*, *linewidth=1*, *sampling='Auto'*, *units='pix'*, *arrow=None*, *linetrace=None*, ***kwargs*)

Plots two line profiles on one image with the line profile intensities in a subfigure. See skimage PR PinkShnack for details on implementing profile_line in skimage: https://github.com/scikit-image/scikit-image/pull/4206

Parameters

- **image** (*2D Hyperspy signal*) –

- **line_profile_positions** (*list of lists*) – two line profile coordinates. Use atomap's am.add_atoms_with_gui() function to get these. The first two dots will trace the first line profile etc. Could be extended to n positions with a basic loop.
- **linewidth** (*int, default 1*) – see profile_line for parameter details.
- **sampling** (*float, default 'Auto'*) –

  if set to 'Auto' the function will attempt to find the sampling of image from image.axes_manager[0].scale.

  arrow [string, default None] If set, arrows will be plotting on the image. Options are 'h' and 'v' for horizontal and vertical arrows, respectively.
- **linetrace** (*int, default None*) – If set, the line profile will be plotted on the image. The thickness of the linetrace will be linewidth*linetrace. Name could be improved maybe.
- **kwargs** (*Matplotlib keyword arguments passed to imshow()*) –

temul.signal_plotting.**compare_images_line_profile_two_images**(*imageA*, *imageB*, *line_profile_positions*, *reduce_func=<function mean>*, *filename=None*, *linewidth=1*, *sampling='auto'*, *units='nm'*, *crop_offset=20*, *title='Intensity Profile'*, *imageA_title='Experiment'*, *imageB_title='Simulation'*, *marker_A='v'*, *marker_B='o'*, *arrow_markersize=10*, *figsize=(10, 3)*, *imageB_intensity_offset=0*)

Plots two line profiles on two images separately with the line profile intensities in a subfigure. See skimage PR PinkShnack for details on implementing profile_line in skimage: https://github.com/scikit-image/scikit-image/pull/4206

Parameters

- **imageA** (*2D Hyperspy signal*) –
- **imageB** (*2D Hyperspy signal*) –
- **line_profile_positions** (*list of lists*) – one line profile coordinate. Use atomap's am.add_atoms_with_gui() function to get these. The two dots will trace the line profile. See Examples below for example.
- **filename** (*string, default None*) – If this is set to a name (string), the image will be saved with that name.
- **reduce_func** (*ufunc, default np.mean*) – See skimage's `profile_line` reduce_func parameter for details.
- **linewidth** (*int, default 1*) – see profile_line for parameter details.
- **sampling** (*float, default 'auto'*) – if set to 'auto' the function will attempt to find the sampling of image from image.axes_manager[0].scale.
- **units** (*string, default 'nm'*) –
- **crop_offset** (*int, default 20*) – number of pixels away from the `line_profile_positions` coordinates the image crop will be taken.
- **title** (*string, default "Intensity Profile"*) – Title of the plot
- **marker_A** (*Matplotlib marker*) –

- **marker_B** (*Matplotlib marker*) –
- **arrow_markersize** (*Matplotlib markersize*) –
- **figsize** (*see Matplotlib for details*) –
- **imageB_intensity_offset** (*float, default 0*) – Adds a y axis offset for comparison purposes.

Examples

```
>>> import atomap.api as am
>>> import temul.api as tml
>>> imageA = am.dummy_data.get_simple_cubic_signal(image_noise=True)
>>> imageB = am.dummy_data.get_simple_cubic_signal()
>>> # line_profile_positions = tml.choose_points_on_image(imageA)
>>> line_profile_positions = [[81.58, 69.70], [193.10, 184.08]]
>>> tml.compare_images_line_profile_two_images(
...     imageA, imageB, line_profile_positions,
...     linewidth=3, sampling=0.012, crop_offset=30)
```

To use the new skimage functionality try the `reduce_func` parameter:

```
>>> import numpy as np
>>> reduce_func = np.sum # can be any ufunc!
>>> tml.compare_images_line_profile_two_images(
...     imageA, imageB, line_profile_positions, reduce_func=reduce_func,
...     linewidth=3, sampling=0.012, crop_offset=30)
```

```
>>> reduce_func = lambda x: np.sum(x**0.5)
>>> tml.compare_images_line_profile_two_images(
...     imageA, imageB, line_profile_positions, reduce_func=reduce_func,
...     linewidth=3, sampling=0.012, crop_offset=30)
```

Offseting the y axis of the second image can sometimes be useful:

```
>>> import temul.example_data as example_data
>>> imageA = example_data.load_Se_implanted_MoS2_data()
>>> imageA.data = imageA.data/np.max(imageA.data)
>>> imageB = imageA.deepcopy()
>>> line_profile_positions = [[301.42, 318.9], [535.92, 500.82]]
>>> tml.compare_images_line_profile_two_images(
...     imageA, imageB, line_profile_positions, reduce_func=None,
...     imageB_intensity_offset=0.1)
```

temul.signal_plotting.**create_rgb_array**()

temul.signal_plotting.**expand_palette**(*palette*, *expand_list*)

Essentially multiply the palette so that it has the number of instances of each color that you want.

Parameters

- **palette** (*list*) – Color palette in hex, rgb or dec
- **expand_list** (*list*) – List of integers that will be used to duplicate colours in the palette.

Returns

Return type List of expanded palette

Examples

```
>>> import temul.api as tml
>>> zest = tml.color_palettes('zesty')
>>> expanded_palette = tml.expand_palette(zest, [1,2,2,2])
```

temul.signal_plotting.**get_cropping_area**(*line_profile_positions*, *crop_offset=20*)

By inputting the top-left and bottom-right coordinates of a rectangle, this function will add a border buffer (*crop_offset*) which can be used for cropping of regions in a plot. See `compare_images_line_profile_two_images` for use-case

temul.signal_plotting.**get_polar_2d_colorwheel_color_list**(*u*, *v*)

make the color_list from the HSV/RGB colorwheel. This color_list will be the same length as u and as v. It works by indexing the angle of the RGB (hue in HSV) array, then indexing the magnitude (r) in the RGB (value in HSV) array, leaving only a single RGB color for each vector.

temul.signal_plotting.**hex_to_rgb**(*hex_values*)

Change from hexidecimal color values to rgb color values. Grabs starting two, middle two, last two values in hex, multiplies by 16^1 and 16^0 for the first and second, respectively.

Parameters **hex_values** (*list*) – A list of hexidecimal color values as strings e.g., '#F5793A'

Returns

Return type list of tuples

Examples

```
>>> import temul.api as tml
>>> tml.hex_to_rgb(color_palettes('zesty'))
[(245, 121, 58), (169, 90, 161), (133, 192, 249), (15, 32, 128)]
```

Create a matplotlib cmap from a palette with the help of matplotlib.colors.from_levels_and_colors()

```
>>> from matplotlib.colors import from_levels_and_colors
>>> zest = tml.hex_to_rgb(tml.color_palettes('zesty'))
>>> zest.append(zest[0])  # make the top and bottom colour the same
>>> cmap, norm = from_levels_and_colors(
...     levels=[0,1,2,3,4,5], colors=tml.rgb_to_dec(zest))
```

temul.signal_plotting.**plot_atom_energies**(*sublattice_list*, *image=None*, *vac_or_implants=None*, *elements_dict_other=None*, *filename='energy_map'*, *cmap='plasma'*, *levels=20*, *colorbar_fontsize=16*)

Used to plot the energies of atomic column configurations above 0 as calculated by DFT.

Parameters

- **sublattice_list** (*list of Atomap Sublattices*) –
- **image** (*array-like, default None*) – The first sublattice image is used if `image=None`.
- **vac_or_implants** (*string, default None*) – `vac_or_implants` options are "implants" and "vac".

- **elements_dict_other** (*dict, default None*) – A dictionary of {element_config1: energy1, element_config2: energy2, } The default is Se antisites in monolayer MoS2.
- **filename** (*string, default "energy_map"*) – Name with which to save the plot.
- **cmap** (*Matplotlib colormap, default "plasma"*) –
- **levels** (*int, default 20*) – Number of Matplotlib contour map levels.
- **colorbar_fontsize** (*int, default 16*) –

Returns

- *The x and y coordinates of the atom positions and the*
- *atom energy.*

temul.signal_plotting.**rgb_to_dec**(*rgb_values*)

Change RGB color values to decimal color values (between 0 and 1). Required for use with matplotlib. See Example in `hex_to_rgb` below.

Parameters **rgb_values** (*list of tuples*) –

Returns

Return type Decimal color values (RGB but scaled from 0 to 1 rather than 0 to 255)

Input/Output (io)

temul.io.**batch_convert_emd_to_image**(*extension_to_save*, *top_level_directory*, *glob_search='**/*'*, *overwrite=True*)

Convert all .emd files to the chosen extension_to_save file format in the specified directory and all subdirectories.

Parameters

- **extension_to_save** (*string*) – the image file extension to be used for saving the image. See Hyperspy documentation for information on file writing extensions available: http://hyperspy.org/hyperspy-doc/current/user_guide/io.html
- **top_level_directory** (*string*) – The top-level directory in which the emd files exist. The default glob_search will search this directory and all subdirectories.
- **glob_search** (*string*) – Glob search string, see glob for more details: https://docs.python.org/2/library/glob.html Default will search this directory and all subdirectories.
- **overwrite** (*bool, default True*) – Overwrite if the extension_to_save file already exists.

temul.io.**convert_vesta_xyz_to_prismatic_xyz**(*vesta_xyz_filename*, *prismatic_xyz_filename*, *delimiter='||'*, *header=None*, *skiprows=[0, 1]*, *engine='python'*, *occupancy=1.0*, *rms_thermal_vib=0.05*, *edge_padding=None*, *header_comment="Let's make a file!"*, *save=True*)

Convert from Vesta outputted xyz file format to the prismatic-style xyz format. Lose some information from the .cif or .vesta file but okay for now. Develop your own converter if you need rms and occupancy! Lots to do.

delimiter=' || ' # ase xyz delimiter=' || ' # vesta xyz

Parameters

- **vesta_xyz_filename** (*string*) – name of the vesta outputted xyz file. See vesta > export > xyz

- **prismatic_xyz_filename** (*string*) – name to be given to the outputted prismatic xyz file
- **delimiter** (*pandas.read_csv input parameters*) – See pandas.read_csv for documentation Note that the delimiters here are only available if you use engine='python'
- **header** (*pandas.read_csv input parameters*) – See pandas.read_csv for documentation Note that the delimiters here are only available if you use engine='python'
- **skiprows** (*pandas.read_csv input parameters*) – See pandas.read_csv for documentation Note that the delimiters here are only available if you use engine='python'
- **engine** (*pandas.read_csv input parameters*) – See pandas.read_csv for documentation Note that the delimiters here are only available if you use engine='python'
- **occupancy** (*see prismatic documentation*) – if you want a file format that will retain these atomic attributes, use a format other than vesta xyz. Maybe .cif or .vesta keeps these?
- **rms_thermal_vib** (*see prismatic documentation*) – if you want a file format that will retain these atomic attributes, use a format other than vesta xyz. Maybe .cif or .vesta keeps these?
- **header_comment** (*string*) – header comment for the file.
- **save** (*bool, default True*) – whether to output the file as a prismatic formatted xyz file with the name of the file given by "prismatic_xyz_filename".

Returns

Return type The converted file format as a pandas dataframe

Examples

See example_data for the vesta xyz file.

```
>>> from temul.io import convert_vesta_xyz_to_prismatic_xyz
>>> prismatic_xyz = convert_vesta_xyz_to_prismatic_xyz(
...     'temul/example_data/prismatic/example_MoS2_vesta_xyz.xyz',
...     'temul/example_data/prismatic/MoS2_hex_prismatic.xyz',
...     delimiter='   |    |  ', header=None, skiprows=[0, 1],
...     engine='python', occupancy=1.0, rms_thermal_vib=0.05,
...     header_comment="Let's do this!", save=True)
```

temul.io.**create_dataframe_for_xyz**(*sublattice_list*, *element_list*, *x_size*, *y_size*, *z_size*, *filename*, *header_comment='top_level_comment'*)

Creates a Pandas Dataframe and a .xyz file (usable with Prismatic) from the inputted sublattice(s).

Parameters

- **sublattice_list** (*list of Atomap Sublattice objects*) –
- **element_list** (*list of strings*) – Each string must be an element symbol from the periodic table.
- **x_size** (*floats*) – Dimensions of the x,y,z axes in Angstrom.
- **y_size** (*floats*) – Dimensions of the x,y,z axes in Angstrom.
- **z_size** (*floats*) – Dimensions of the x,y,z axes in Angstrom.
- **filename** (*string*) – Name with which the .xyz file will be saved.

- **header_comment** (*string, default 'top_level_comment'*) –

Example

```
>>> import temul.external.atomap_devel_012.dummy_data as dummy_data
>>> sublattice = dummy_data.get_simple_cubic_sublattice()
>>> for i in range(0, len(sublattice.atom_list)):
...     sublattice.atom_list[i].elements = 'Mo_1'
...     sublattice.atom_list[i].z_height = '0.5'
>>> element_list = ['Mo_0', 'Mo_1', 'Mo_2']
>>> x_size, y_size = 50, 50
>>> z_size = 5
>>> dataframe = create_dataframe_for_xyz([sublattice], element_list,
...                          x_size, y_size, z_size,
...                          filename='dataframe',
...                          header_comment='Here is an Example')
```

temul.io.**dm3_stack_to_tiff_stack**(*loading_file*, *loading_file_extension='.dm3'*, *saving_file_extension='.tif'*, *crop=False*, *crop_start=20.0*, *crop_end=80.0*)

Save an image stack filetype to a different filetype, e.g., dm3 to tiff.

Parameters

- **filename** (*string*) – Name of the image stack file
- **loading_file_extension** (*string*) – file extension of the filename
- **saving_file_extension** (*string*) – file extension you wish to save as
- **crop** (*bool, default False*) – if True, the image will be cropped in the navigation space, defined by the frames given in crop_start and crop_end
- **crop_start** (*float, default 20.0, 80.0*) – the start and end frame of the crop
- **crop_end** (*float, default 20.0, 80.0*) – the start and end frame of the crop

temul.io.**load_data_and_sampling**(*filename*, *file_extension=None*, *invert_image=False*, *save_image=True*)

temul.io.**load_prismatic_mrc_with_hyperspy**(*prismatic_mrc_filename*, *save_name='calibrated_data_'*)

We are aware this is currently producing errors with new versions of Prismatic.

Open a prismatic .mrc file and save as a hyperspy object. Also plots saves a png.

Parameters **prismatic_mrc_filename** (*string*) – name of the outputted prismatic .mrc file.

Returns

Return type Hyperspy Signal 2D

Examples

```
>>> from temul.io import load_prismatic_mrc_with_hyperspy
>>> load_prismatic_mrc_with_hyperspy("temul/example_data/prismatic/"
...         "prism_2Doutput_prismatic_simulation.mrc")
<Signal2D, title: , dimensions: (|1182, 773)>
```

temul.io.**save_individual_images_from_image_stack**(*image_stack*, *output_folder='individual_images'*)

Save each image in an image stack. The images are saved in a new folder. Useful for after running an image series through Rigid Registration.

Parameters

- **image_stack** (*rigid registration image stack object*) –
- **output_folder** (*string*) – Name of the folder in which all individual images from the stack will be saved.

temul.io.**write_cif_from_dataframe**(*dataframe*, *filename*, *chemical_name_common*, *cell_length_a*, *cell_length_b*, *cell_length_c*, *cell_angle_alpha=90*, *cell_angle_beta=90*, *cell_angle_gamma=90*, *space_group_name_H_M_alt='P 1'*, *space_group_IT_number=1*)

Write a cif file from a Pandas Dataframe. This Dataframe can be created with temul.model_creation.create_dataframe_for_cif().

Parameters

- **dataframe** (*dataframe object*) – pandas dataframe containing rows of atomic position information
- **chemical_name_common** (*string*) – name of chemical
- **cell_length_a** (*float*) – lattice dimensions in angstrom
- **_cell_length_b** (*float*) – lattice dimensions in angstrom
- **_cell_length_c** (*float*) – lattice dimensions in angstrom
- **cell_angle_alpha** (*float*) – lattice angles in degrees
- **cell_angle_beta** (*float*) – lattice angles in degrees
- **cell_angle_gamma** (*float*) – lattice angles in degrees
- **space_group_name_H-M_alt** (*string*) – space group name
- **space_group_IT_number** (*float*) –

Dummy Data

temul.dummy_data.**get_distorted_cubic_signal_adjustable**(*image_noise=False*, *y_offset=2*)

Generate a test image signal of a distorted cubic atomic structure.

Parameters

- **image_noise** (*default False*) – If True, will add Gaussian noise to the image.
- **y_offset** (*int, default 2*) – The magnitude of distortion of the cubic signal.

Returns

Return type HyperSpy Signal2D

Examples

```
>>> from temul.dummy_data import get_distorted_cubic_signal_adjustable
>>> s = get_distorted_cubic_signal_adjustable(y_offset=2)
>>> s.plot()
```

temul.dummy_data.**get_distorted_cubic_sublattice_adjustable**(*image_noise=False*, *y_offset=2*)

Generate a test sublattice of a distorted cubic atomic structure.

Parameters

- **image_noise** (*default False*) – If True, will add Gaussian noise to the image.
- **y_offset** (*int, default 2*) – The magnitude of distortion of the cubic signal.

Returns

Return type Atomap Sublattice object

Examples

```
>>> from temul.dummy_data import get_distorted_cubic_sublattice_adjustable
>>> sublattice = get_distorted_cubic_sublattice_adjustable(y_offset=2)
>>> sublattice.plot()
```

temul.dummy_data.**get_polarisation_dummy_dataset**(*image_noise=False*)

Get an Atom Lattice with two sublattices resembling a perovskite film.

Similar to a perovskite oxide thin film, where the B cations are shifted in the film.

Parameters **image_noise** (*bool, default False*) –

Returns **simple_atom_lattice**

Return type Atom_Lattice object

Examples

```
>>> import numpy as np
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset()
>>> atom_lattice.plot()
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> sublatticeA.construct_zone_axes()
>>> za0, za1 = sublatticeA.zones_axis_average_distances[0:2]
>>> s_p = sublatticeA.get_polarization_from_second_sublattice(
...     za0, za1, sublatticeB, color='blue')
>>> s_p.plot()
>>> vector_list = s_p.metadata.vector_list
>>> x, y = [i[0] for i in vector_list], [i[1] for i in vector_list]
>>> u, v = [i[2] for i in vector_list], [i[3] for i in vector_list]
>>> u, v = -np.asarray(u), -np.asarray(v)
```

You can they use the plot_polarisation_vectors function to visualise:

```
>>> import temul.api as tml
>>> ax = plot_polarisation_vectors(x, y, u, v, image=sublatticeA.image,
...                            unit_vector=False, plot_style="vector",
...                            overlay=True, color='yellow',
...                            degrees=False, save=None, monitor_dpi=50)
```

temul.dummy_data.**get_polarisation_dummy_dataset_bora**(*image_noise=False*)

temul.dummy_data.**get_polarised_single_sublattice**(*image_noise=False*)

temul.dummy_data.**get_polarised_single_sublattice_rotated**(*image_noise=False*, *rotation=45*)

temul.dummy_data.**get_simple_cubic_signal**(*image_noise=False*, *amplitude=1*, *with_vacancies=False*)

Generate a test image signal of a simple cubic atomic structure.

Parameters

- **image_noise** (*bool, default False*) – If set to True, will add Gaussian noise to the image.
- **amplitude** (*int, list of ints, default 1*) – If amplitude is set to an int, that int will be applied to all atoms in the sublattice. If amplitude is set to a list, the atoms will be a distribution set by np.random.randint between the min and max int.
- **with_vacancies** (*bool, default False*) – If set to True, the returned signal or sublattice will have some vacancies.

Returns **signal**

Return type HyperSpy 2D

Examples

```
>>> import atomap.api as am
>>> s = am.dummy_data.get_simple_cubic_signal()
>>> s.plot()
```

temul.dummy_data.**get_simple_cubic_sublattice**(*image_noise=False*, *amplitude=1*, *with_vacancies=False*)

Generate a test sublattice of a simple cubic atomic structure.

Parameters

- **image_noise** (*bool, default False*) – If set to True, will add Gaussian noise to the image.
- **amplitude** (*int, list of ints, default 1*) – If amplitude is set to an int, that int will be applied to all atoms in the sublattice. If amplitude is set to a list, the atoms will be a distribution set by np.random.randint between the min and max int.
- **with_vacancies** (*bool, default False*) – If set to True, the returned signal or sublattice will have some vacancies.

Returns

Return type Atomap Sublattice object

Examples

```
>>> from temul.dummy_data import get_simple_cubic_sublattice
>>> sublattice = get_simple_cubic_sublattice()
>>> sublattice.plot()
```

If you want different atom amplitudes, use `amplitude`

```
>>> sublattice = get_simple_cubic_sublattice(
...     amplitude=[1, 5])
```

Do not set `amplitude` to two consecutive numbers, as only amplitudes of the lower number (2 below) will be set, see numpy.random.randint for info.

```
>>> sublattice = get_simple_cubic_sublattice(
...     amplitude=[2,3])
```

temul.dummy_data.**get_simple_cubic_sublattice_positions_on_vac**(*image_noise=False*)

Create a simple cubic structure similar to `get_simple_cubic_sublattice` above but the atom positions are also overlaid on the vacancy positions.

temul.dummy_data.**make_polarised_sublattice_bora_return**(*image_noise=False*)

temul.dummy_data.**polarisation_colorwheel_test_dataset**(*cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *plot_XY=True*, *degrees=False*, *normalise=False*)

Check how the arrows will be plotted on a colorwheel. Note that for STEM images, the y axis is reversed. This is taken into account in the plot_polarisation_vectors function, but not here.

Parameters **image_noise** (*default False*) – If True, will add Gaussian noise to the image.

Examples

```
>>> from temul.dummy_data import polarisation_colorwheel_test_dataset
```

Use a cyclic colormap for better understanding of vectors

```
>>> polarisation_colorwheel_test_dataset(cmap='hsv')
```

For more cyclic colormap options (and better colormaps), use colorcet

```
>>> import colorcet as cc
>>> polarisation_colorwheel_test_dataset(cmap=cc.cm.colorwheel)
```

Plot with degrees rather than the default radians

```
>>> polarisation_colorwheel_test_dataset(degrees=True)
```

To just plot the top and bottom arrows as "diverging" the middle of the colormap should be the same as the edges, such as colorcet's CET_C4.

```
>>> polarisation_colorwheel_test_dataset(cmap=cc.cm.CET_C4)
```

To just plot the left and right arrows as "diverging" the halfway points of the colormap should be the same as the edges, such as colorcet's CET_C4s.

```
>>> polarisation_colorwheel_test_dataset(cmap=cc.cm.CET_C4s)
```

temul.dummy_data.**sine_wave_sublattice**()

Example Data

temul.example_data.**load_Se_implanted_MoS2_data**()
Load an ADF image of Se implanted monolayer MoS2.

Example

```
>>> import temul.example_data as example_data
>>> s = example_data.load_Se_implanted_MoS2_data()
>>> s.plot()
```

temul.example_data.**load_Se_implanted_MoS2_simulation**()
Get the simulated image of an MoS2 monolayer

Example

```
>>> import temul.example_data as example_data
>>> s = example_data.load_Se_implanted_MoS2_simulation()
>>> s.plot()
```

temul.example_data.**load_example_Au_nanoparticle**()
Get the emd STEM image of an example Au nanoparticle

Example

```
>>> import temul.example_data as example_data
>>> s = example_data.load_example_Au_nanoparticle()
>>> s.plot()
```

temul.example_data.**load_example_Cu_nanoparticle_sim**()
Get the hspy simulated image of an example Cu nanoparticle

Example

```
>>> import temul.example_data as example_data
>>> s = example_data.load_example_Cu_nanoparticle_sim()
>>> s.plot()
```

temul.example_data.**path_to_example_data_MoS2_hex_prismatic**()
Get the path of the xyz file for monolayer MoS2

Example

```
>>> import temul.example_data as example_data
>>> path_xyz_file = example_data.path_to_example_data_MoS2_hex_prismatic()
```

temul.example_data.**path_to_example_data_MoS2_vesta_xyz**()
Get the path of the vesta xyz file for bilayer MoS2

Example

```
>>> import temul.example_data as example_data
>>> path_vesta_file = example_data.path_to_example_data_MoS2_vesta_xyz()
```

CHAPTER

FOUR

INSTALLATION

The TEMUL Toolkit can be installed easily with PIP (those using Windows may need to download VS C++ Build Tools, see below).

```
$ pip install temul-toolkit
```

Then, it can be imported with the name "temul". For example, to import most of the temul functionality use:

```
import temul.api as tml
```

- If installing on Windows, you will need Visual Studio C++ Build Tools. Download it here. After downloading, choose the "C++ Build Tools" Workload and click install.
- If you want to use the *temul.io.write_cif_from_dataframe()* function, you will need to install pyCifRW version 4.3. This requires Visual Studio.
- If you wish to use the `temul.simulations` or `temul.model_refiner` modules, you will need to install PyPrismatic. This requires Visual Studio and other dependencies. **It is unfortunately not guaranteed to work**. If you want to help develop the `temul.model_refiner.Model_Refiner`, please create an issue and/or a pull request on the TEMUL github.
- If you're using any of the functions or classes that require element quantification:
  - navigate to the "temul/external" directory, copy the "atomap_devel_012" folder and paste that in your "site-packages" directory.
  - Then, when using atomap to create sublattices and quantify elements call atomap like this: `import atomap_devel_012.api as am`.
  - This development version is slowly being folded into the master branch here: https://gitlab.com/atomap/atomap/-/issues/93 and any help or tips on implementation are welcome!

CHAPTER

FIVE

GETTING STARTED

There are many aspects to the TEMUL Toolkit, such as polarisation analysis, element quantification, and automatic image simulation (through pyprismatic).

Checkout the tutorials in the table of contents above or on the left of the page. One can also view the extensive *documentation*, where each function is described and examples of their use given.

To use the vast majority of the temul functionality, import it from the api module:

```
import temul.api as tml
```

CHAPTER

SIX

CODE DOCUMENTATION

See the *API documentation* for examples and a full list of modules and functions.

CHAPTER

SEVEN

CITE

To cite the latest TEMUL Toolkit version, use the following DOI:

For example: Eoghan O'Connell, Michael Hennessy, & Eoin Moynihan. (2021). PinkShnack/TEMUL: (Version 0.1.3). Zenodo. http://doi.org/10.5281/zenodo.4543963

If you wish to cite an older release of the TEMUL Toolkit, click on the above badge to find the relevant version.

CHAPTER

EIGHT

CONTRIBUTE

- Issue Tracker
- Source Code

CHAPTER

NINE

SUPPORT

If you are having issues, please let us know in the issue tracker on GitHub.

CHAPTER

TEN

LICENSE

The project is licensed under the GPL-3.0 License.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

M

P

R

S

T

V

W