TEMUL Toolkit Documentation

Release v0.1.3

Eoghan O'Connell

Jan 29, 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 1

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 2

Some published results using the TEMUL Toolkit

CHAPTER 3

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.

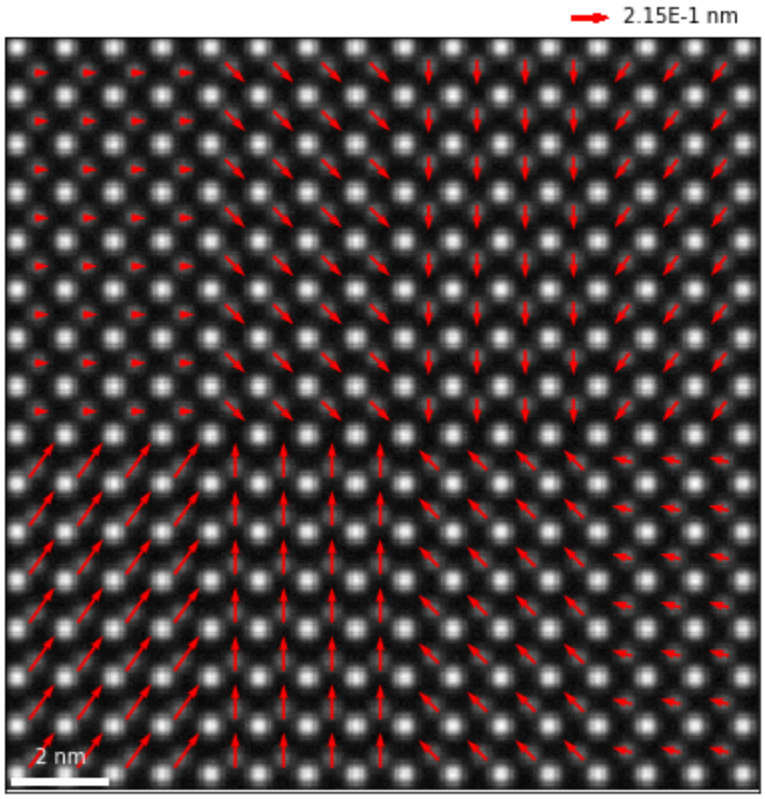
These methods are also available as python scripts and jupyter notebooks in the TEMUL repository in the "code_tutorials/workflows" folder. You can interact with the workflows without needing any downloads. Just click the button below and navigate to that same folder, where you will find the python scripts 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]
>>> atom_positions2 = sublatticeA.find_missing_atoms_from_zone_vector(
```

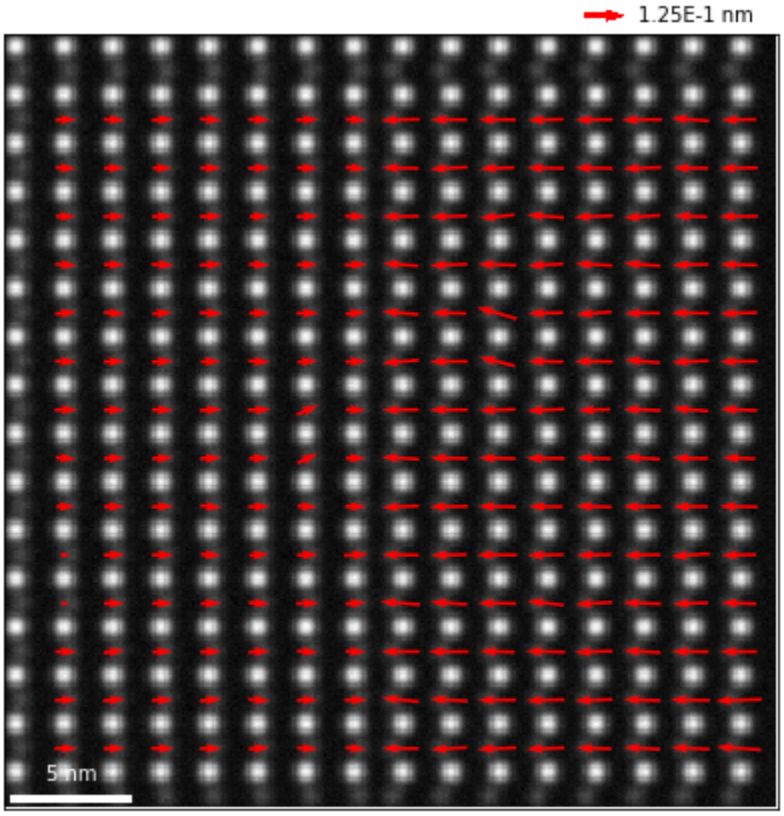
(continues on next page)

(continued from previous page)

```
...     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)
```

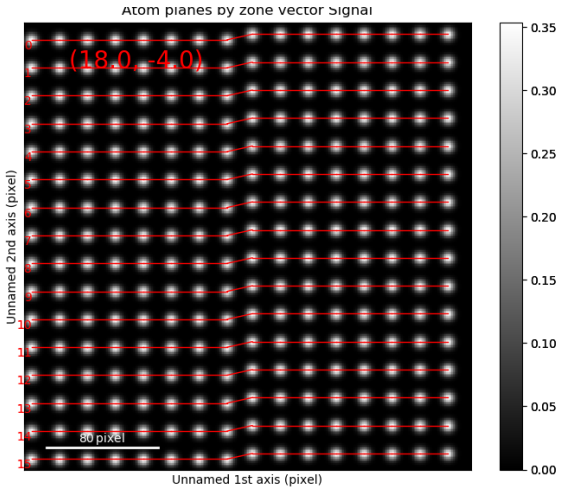


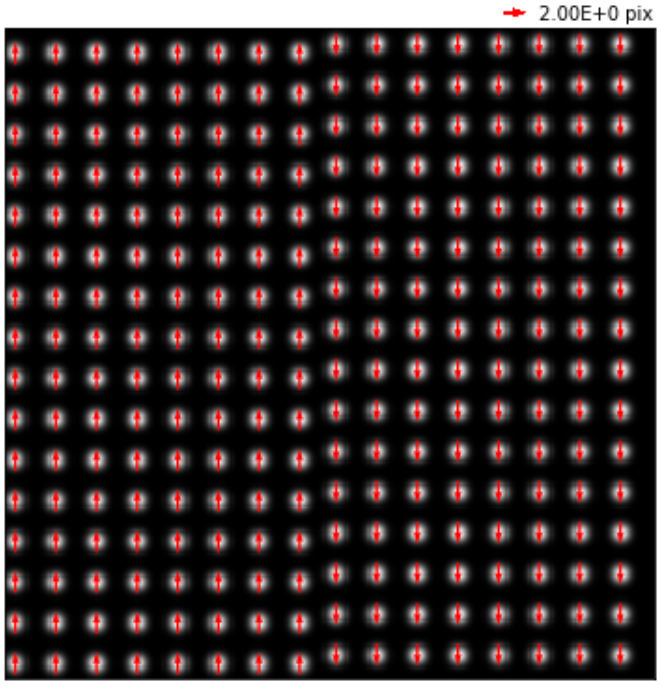
For single Polarised Sublattices (e.g., LNO)

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()`!

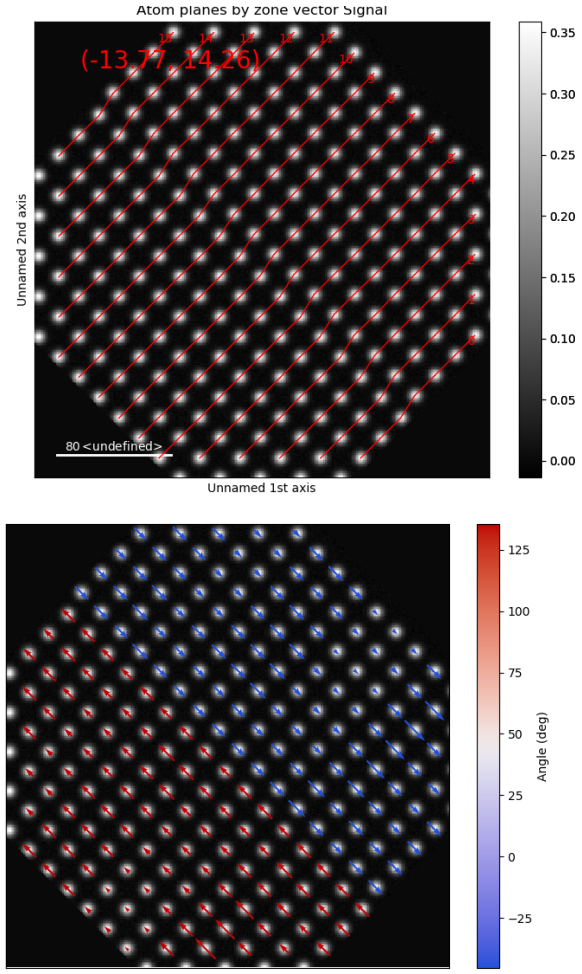
```
>>> import temul.api as tml
>>> import temul.dummy_data as dd
>>> sublattice = dd.get_polarised_single_sublattice()
>>> sublattice.construct_zone_axes(atom_plane_tolerance=1)
>>> sublattice.plot_planes()

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.

>>> 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)
```





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)
```



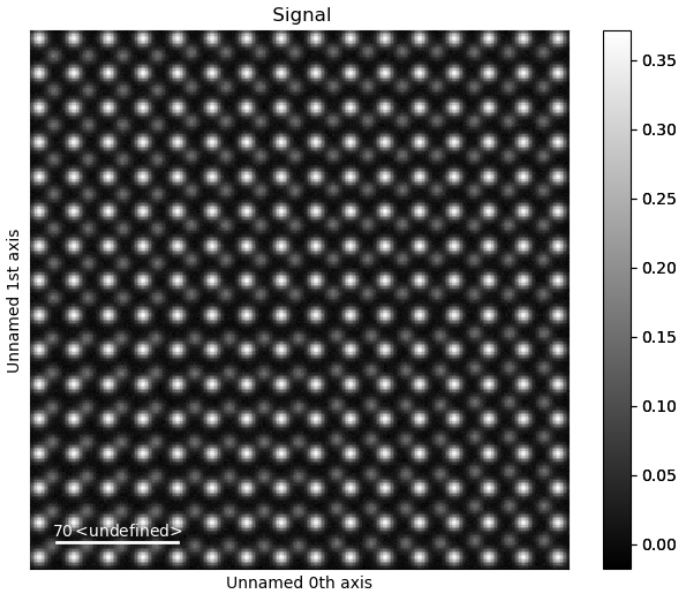
3.4.5 Plotting Polarisation and Movement Vectors

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).

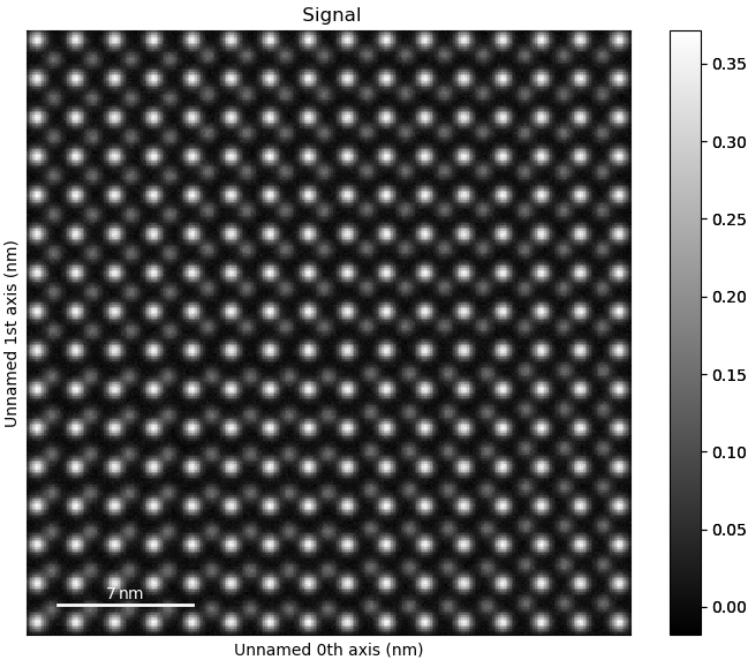
Prepare and Plot the dummy dataset

```
>>> 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()
```

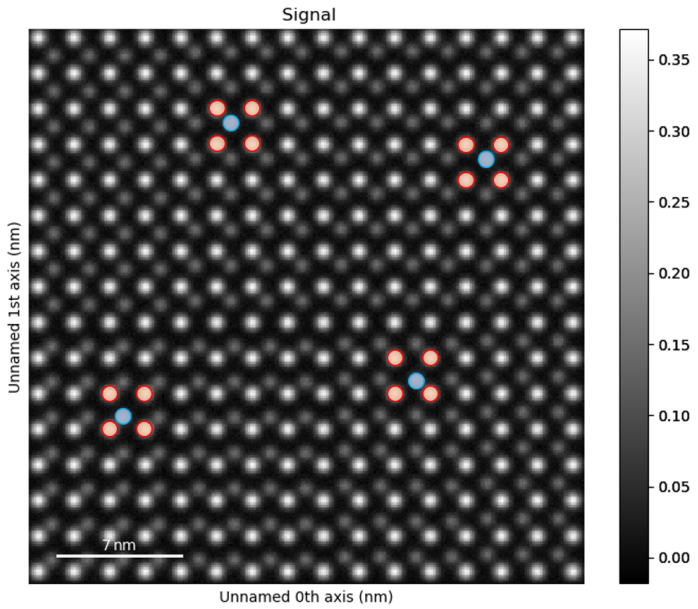


It is best when the image is calibrated. Your image may already be calibrated, but if not, use Hyperspy's `axes_manager` for calibration.

```
>>> 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

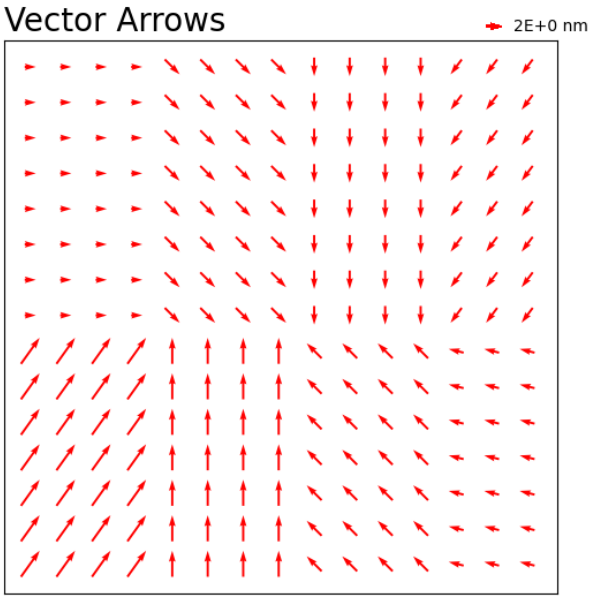
Using the Atomap package, we can easily get the polarisation vectors for regular structures.

```
>>> 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.

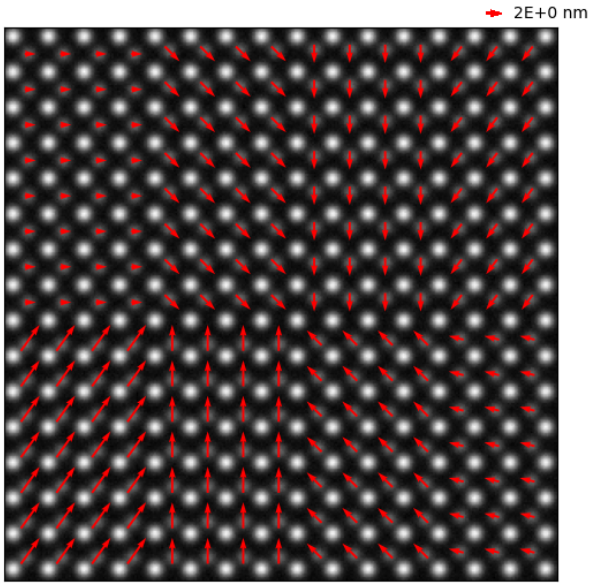
Vector magnitude plot with red arrows:

```
>>> 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)
```



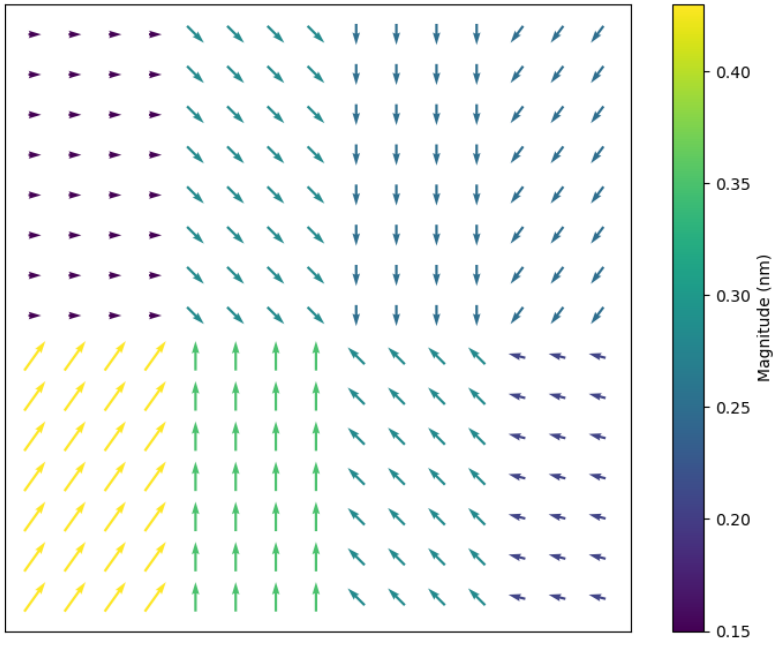
Vector magnitude plot with red arrows overlaid on the image, no title:

```
>>> 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)
```



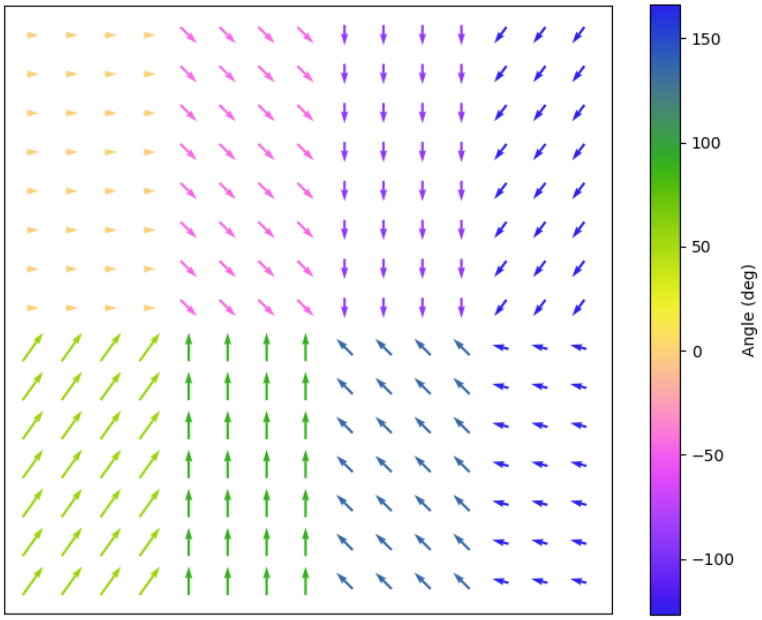
Vector magnitude plot with colormap viridis:

```
>>> 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')
```



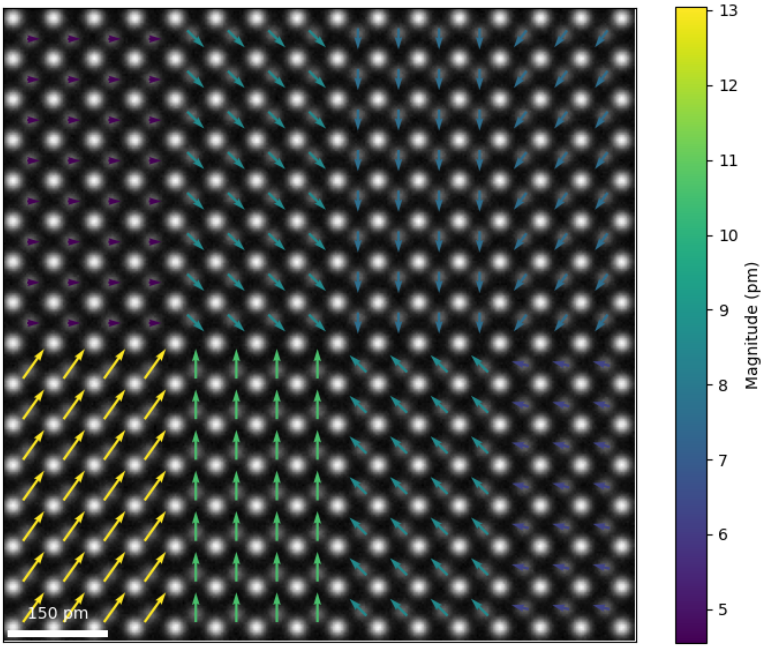
Vector angle plot with colormap viridis (`vector_rep='angle'`):

```
>>> 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)
```



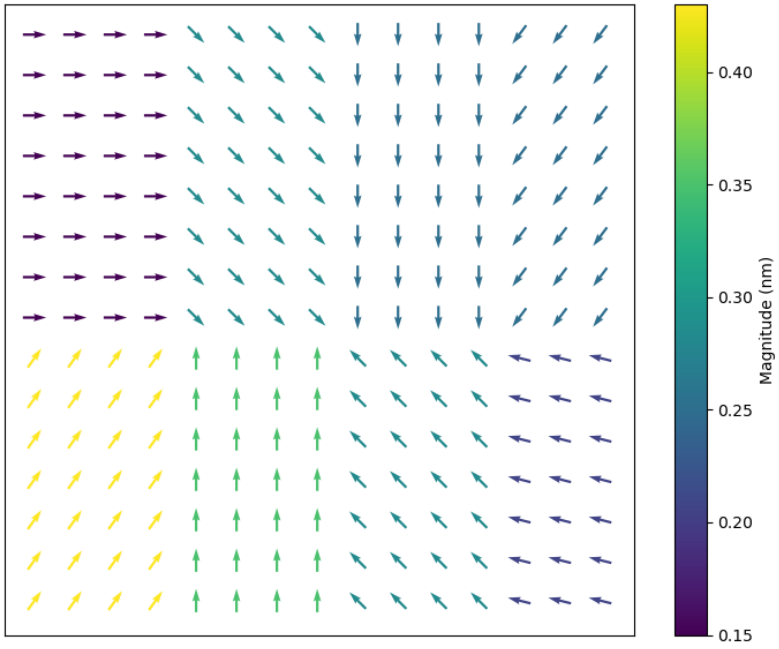
Vector plot with colormap viridis and unit vectors:

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

(continues on next page)

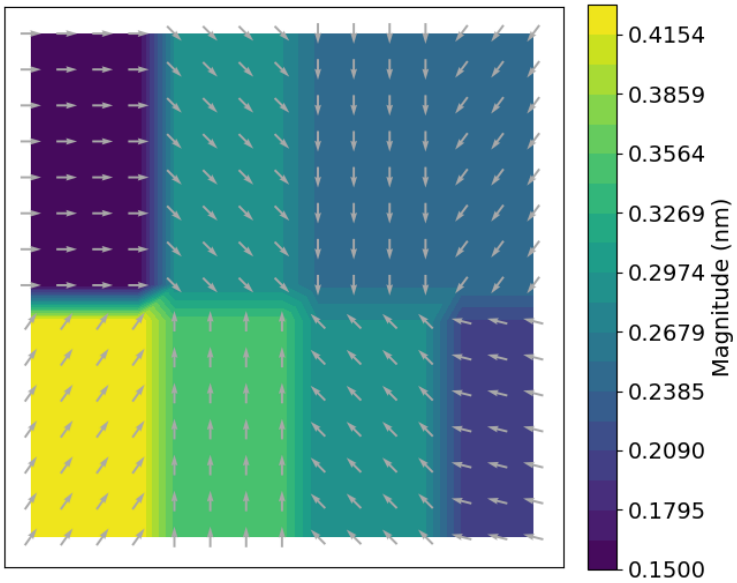
(continued from previous page)

```
...                             unit_vector=True, save=None, monitor_dpi=50,
...                             plot_style='colormap', color='r',
...                             overlay=False, cmap='viridis')
```



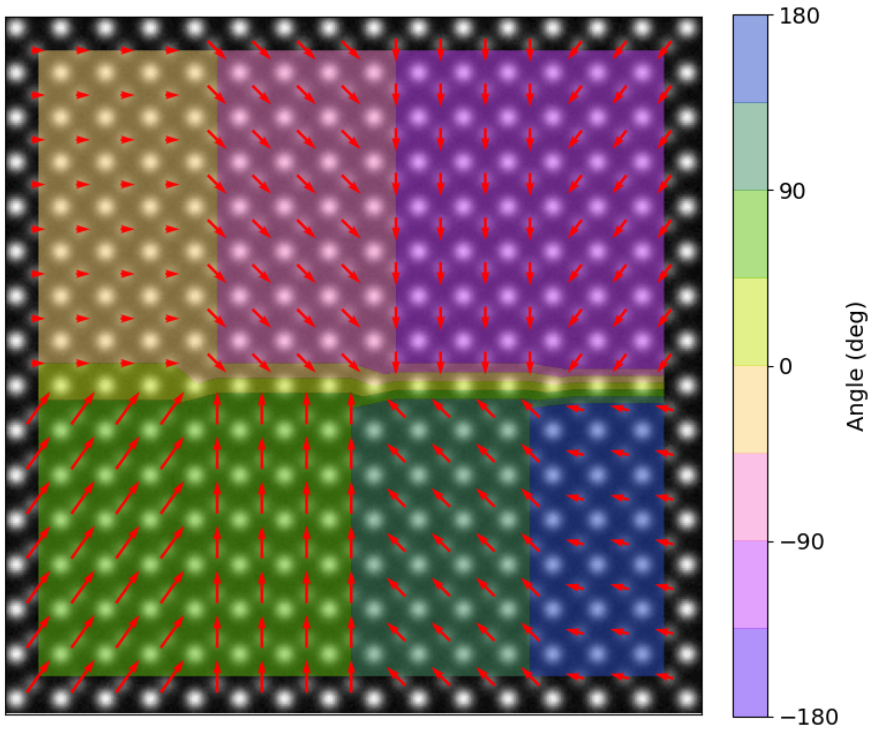
Change the vectors to unit vectors on a Matplotlib tricontourf map:

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



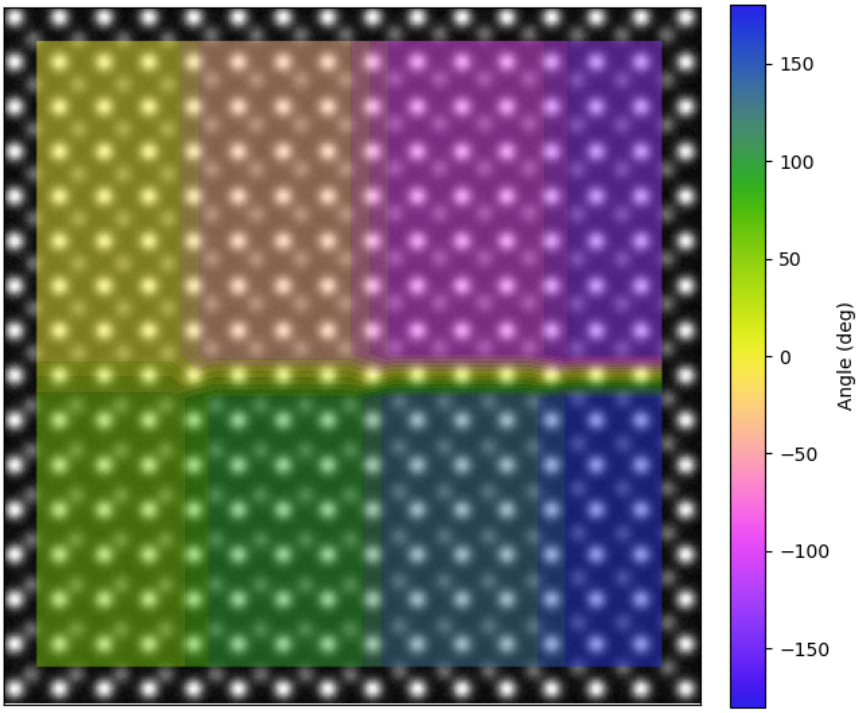
Plot a partly transparent angle tricontourf map with specified colorbar ticks and vector arrows:

```
>>> 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])
```



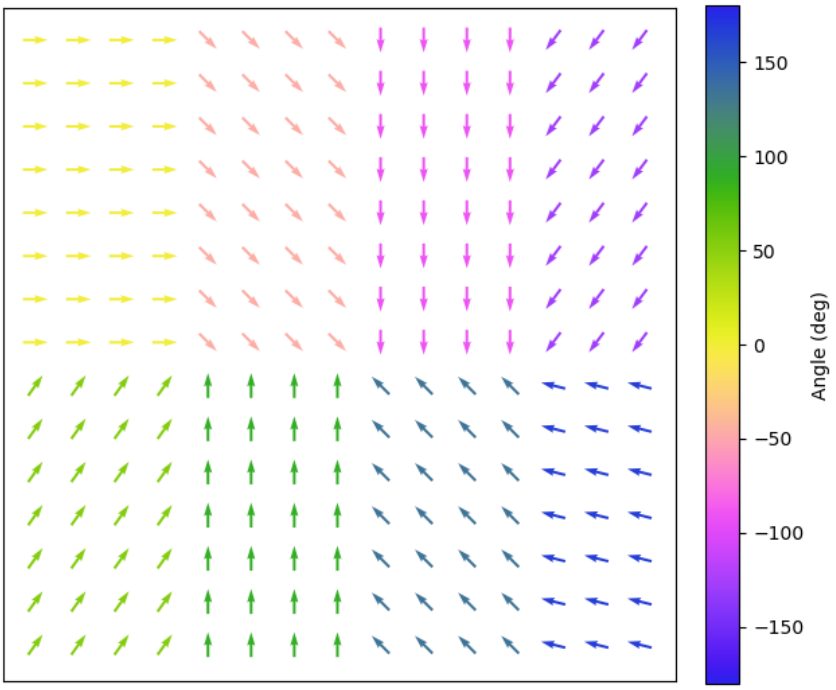
Plot a partly transparent angle tricontourf map with no vector arrows:

```
>>> 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)
```



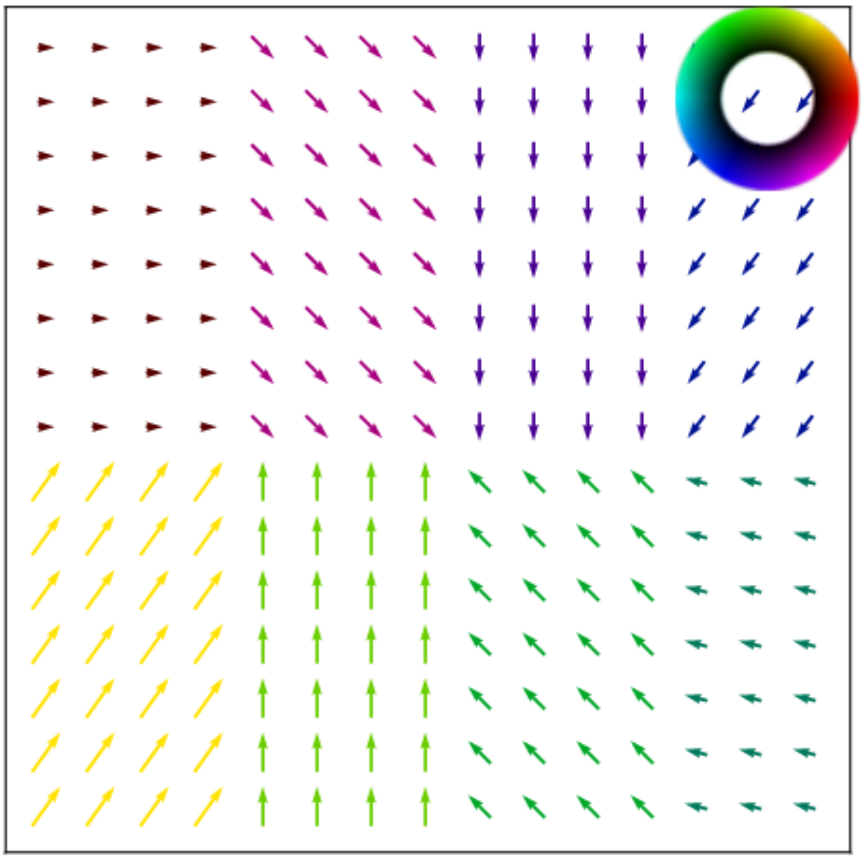
"colorwheel" plot of the vectors, useful for visualising vortexes:

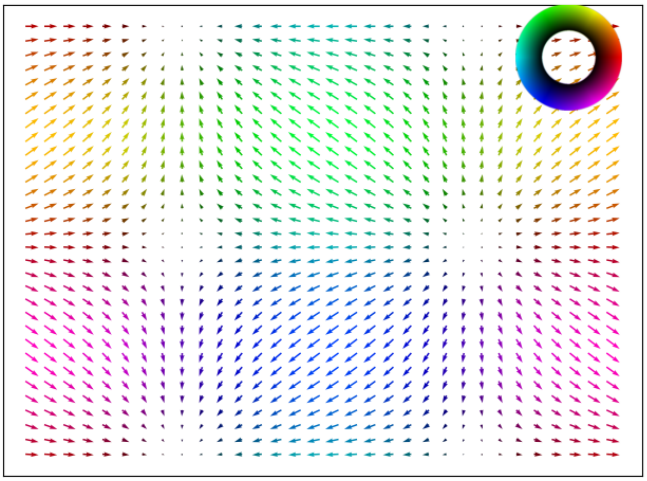
```
>>> import colorcet as cc  # can also just use cmap="cet_colorwheel"
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               unit_vector=True, plot_style="colorwheel",
...                               vector_rep="angle",
...                               overlay=False,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



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

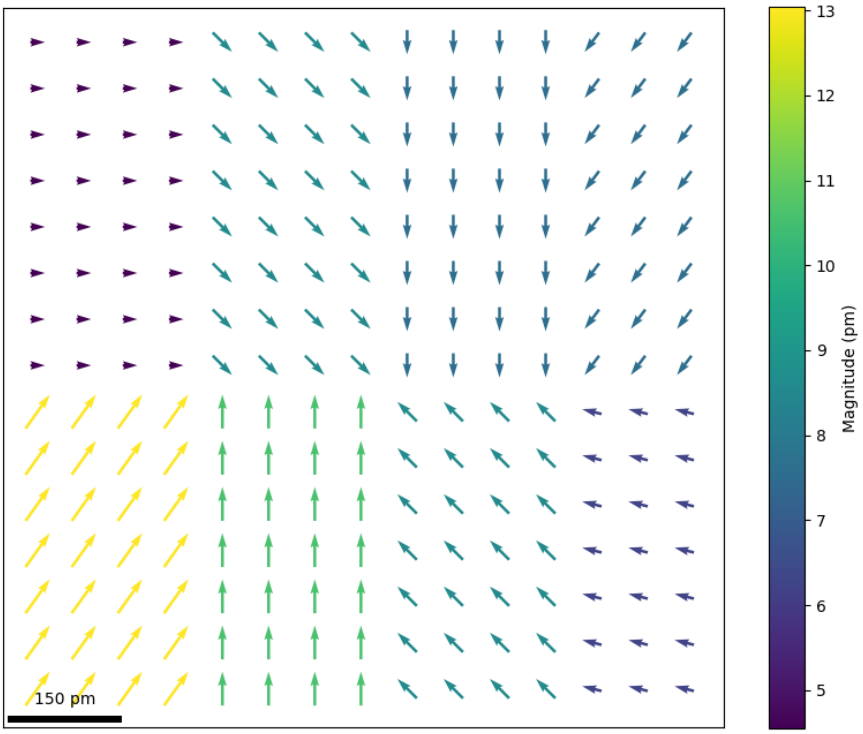
```
>>> tml.plot_polarisation_vectors(x, y, u, v, image=image,
...                               plot_style="polar_colorwheel",
...                               unit_vector=False, overlay=False,
...                               save=None, monitor_dpi=50)

# 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()
```





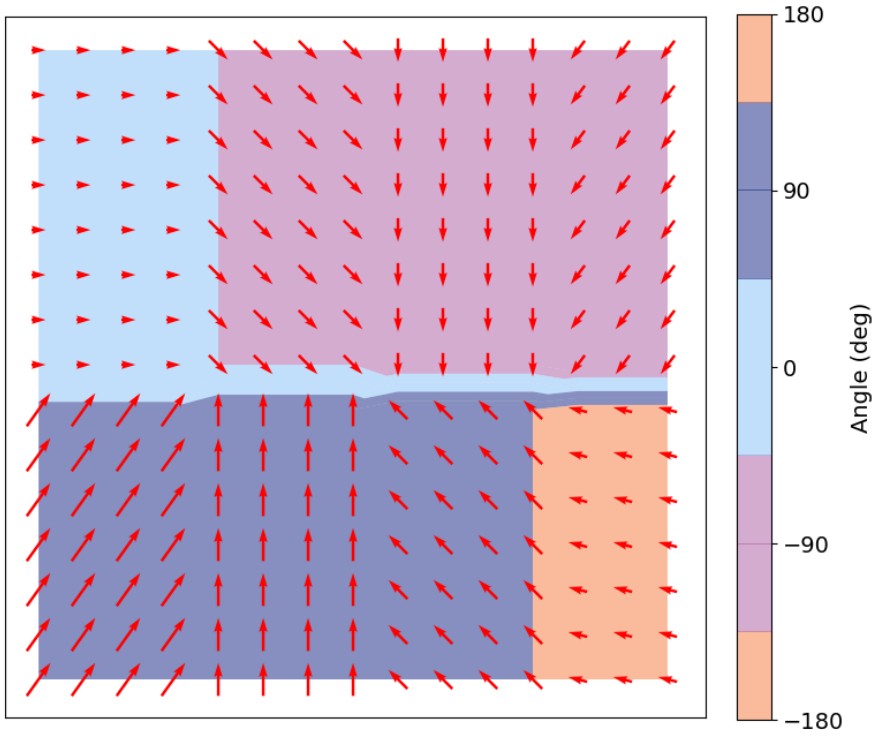
Plot with a custom scalebar. In this example, we need it to be dark, see matplotlib-scalebar for more custom features.

```
>>> 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])
```

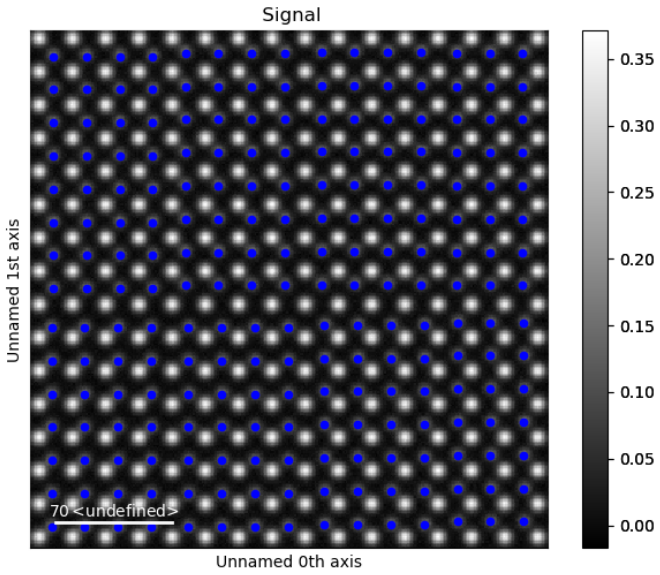


3.4.6 Plot Lattice Structure Maps

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

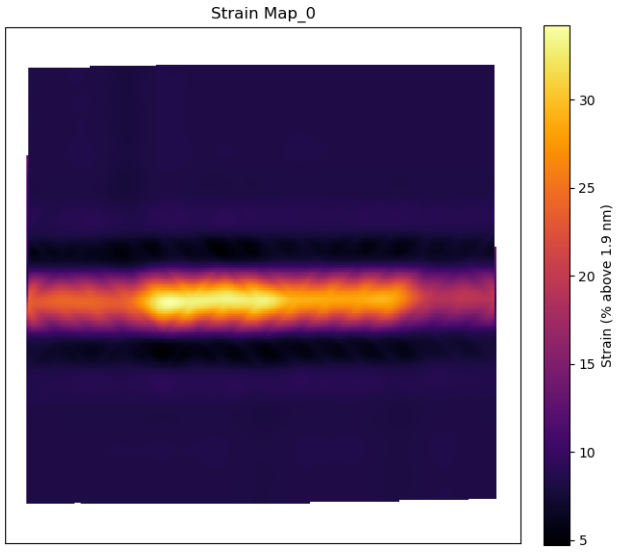
```
>>> import temul.api as tml
>>> from temul.dummy_data import get_polarisation_dummy_dataset
>>> atom_lattice = get_polarisation_dummy_dataset(image_noise=True)
>>> sublatticeA = atom_lattice.sublattice_list[0]
>>> sublatticeB = atom_lattice.sublattice_list[1]
>>> sublatticeA.construct_zone_axes()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

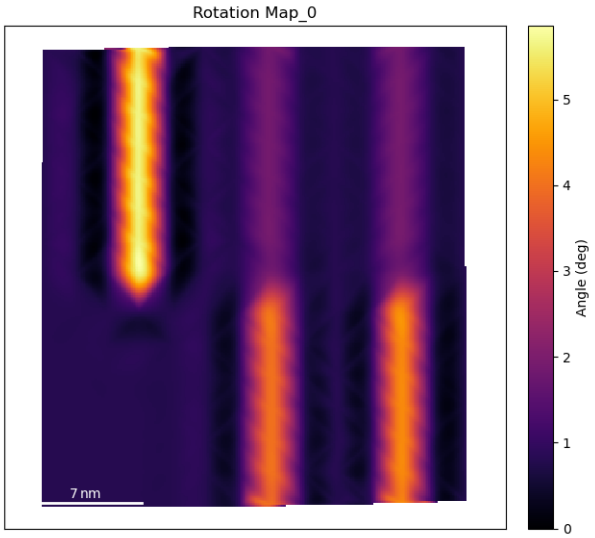
format). This applies to all structure maps discussed in this tutorial.

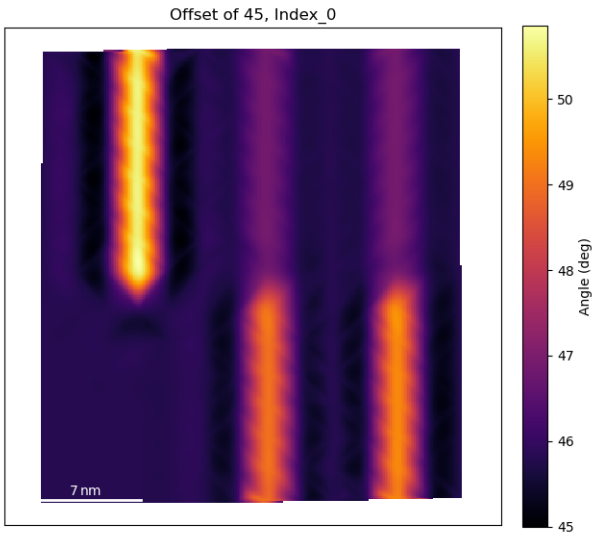
Setting `return_x_y_z=False` will return the strain map along with the x and y coordinates along with their corresponding strain values. One can then use these values externally, e.g., create a matplotlib tricontour plot). This applies to all structure maps discussed in this tutorial.

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')
```

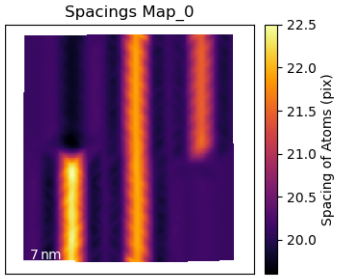


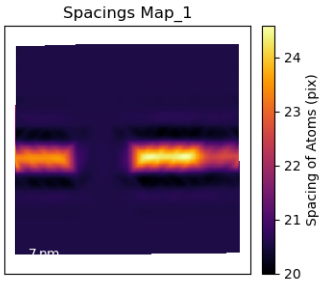


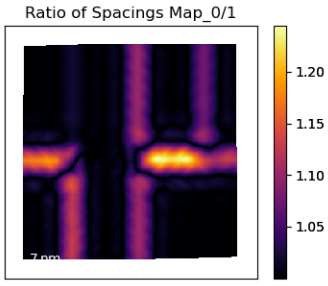
Plot the *c*/*a* Ratio

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.

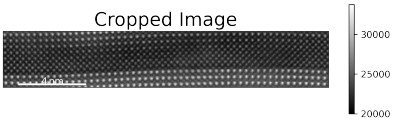
The `temul.topotem.lattice_structure_tools.calculate_atom_plane_curvature()` function in the `temul.topotem.lattice_structure_tools` module can be used to find the curvature of the displacement of atoms along an atom plane in a sublattice. Using the default parameter `func='strain_grad'`, the function 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)
```

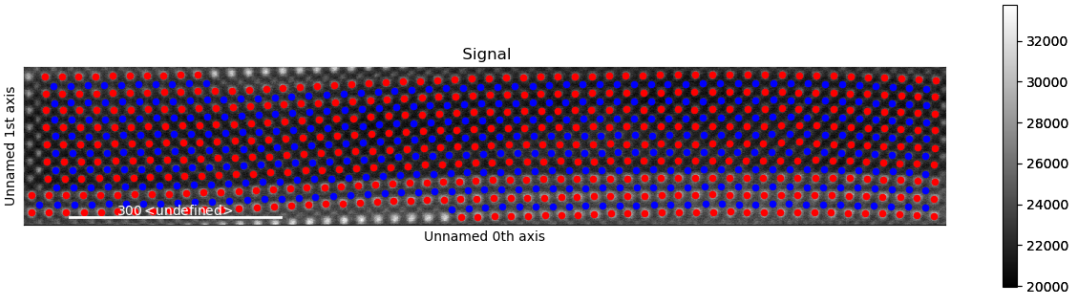
Open the PTO/SRO dataset

```
>>> 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
```

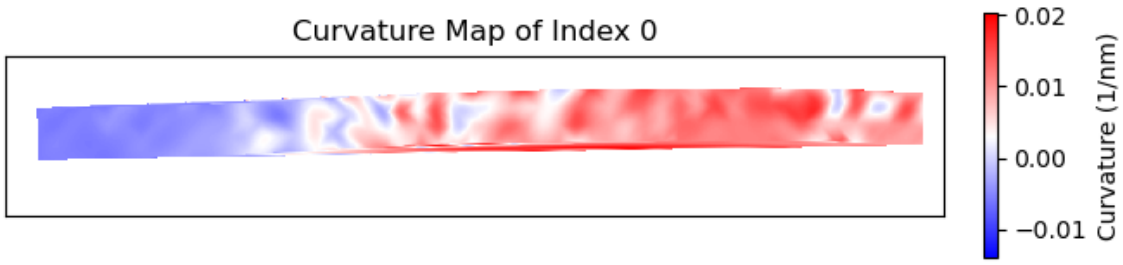
Set the extra initial fitting parameters

```
>>> 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.

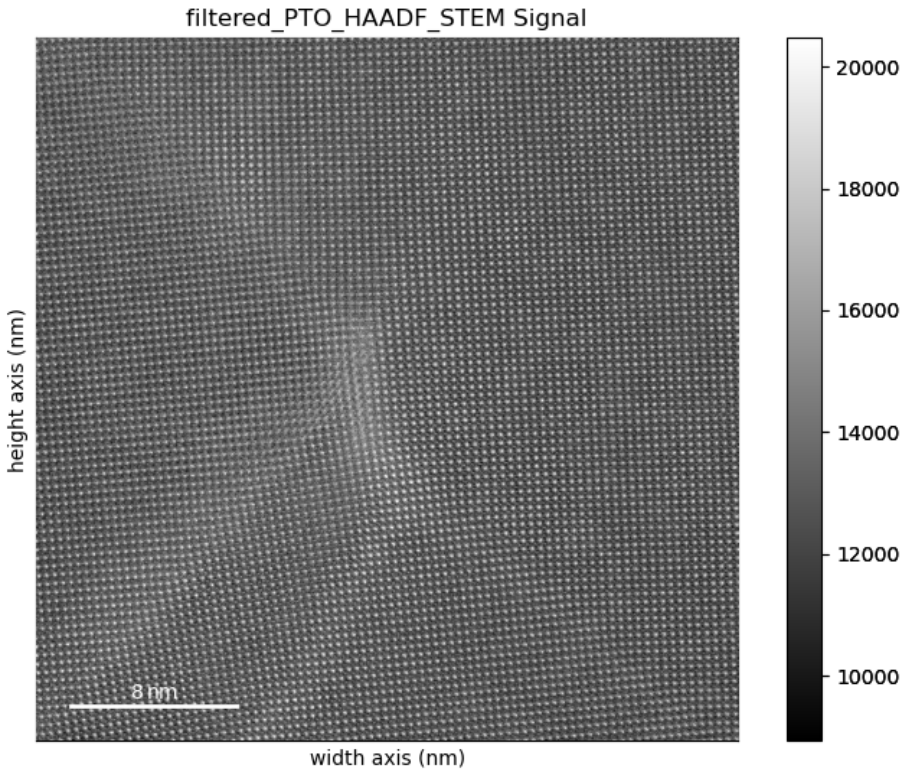
Use the notebook interactively now with MyBinder, just click the launch button below (There are some memory errors at the moment with the large .hdf5 files. It may work better if you run a Jupyter Notebook locally).

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:

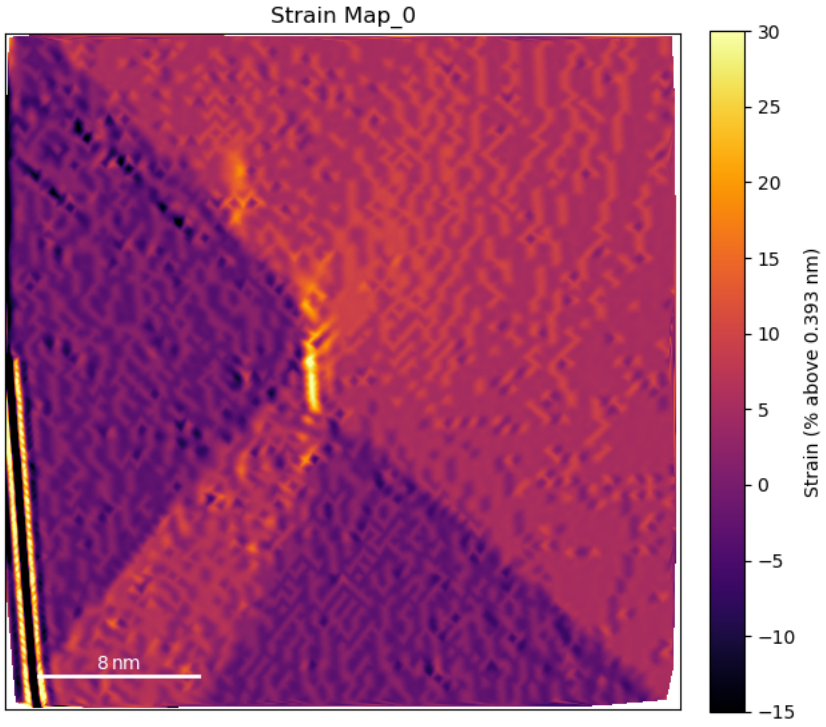
```
>>> line_profile_positions = np.load('line_profile_positions.npy')
```

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

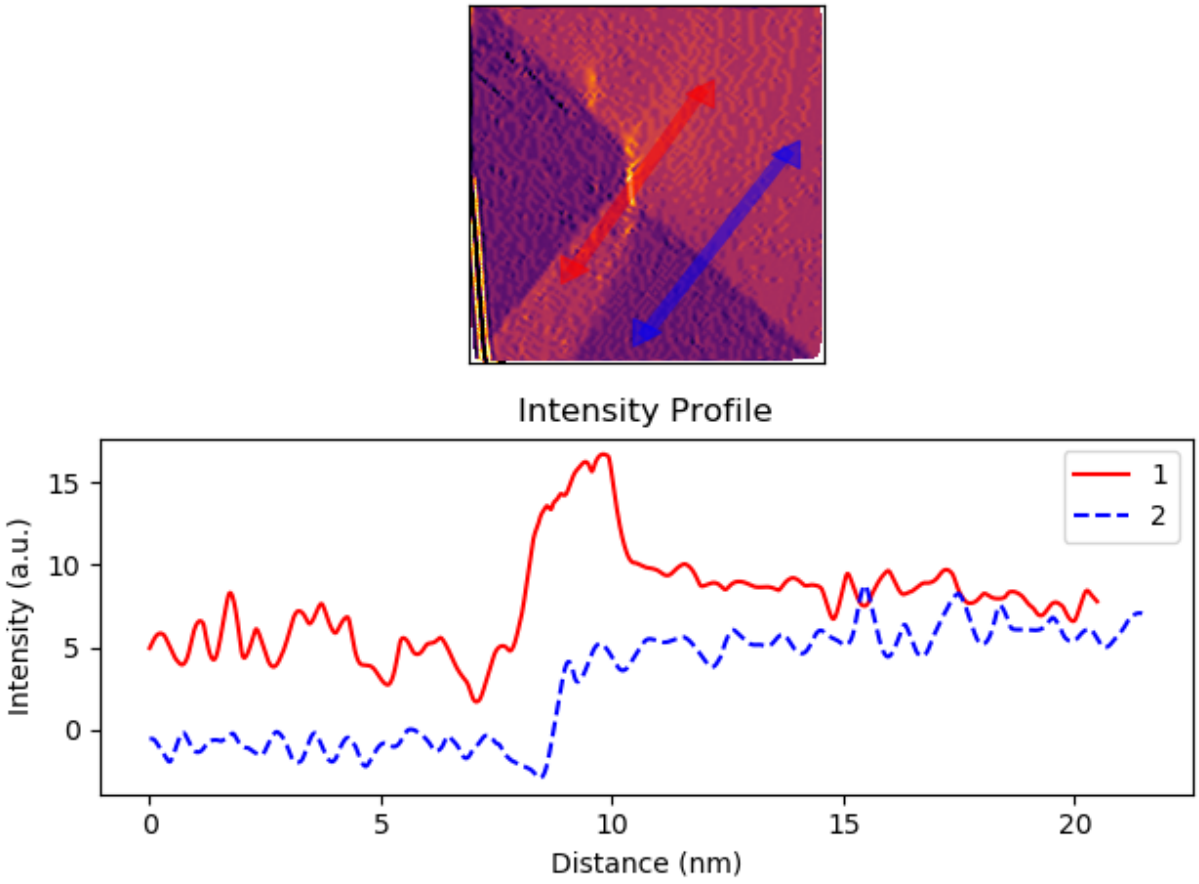
We want to see the strain map of the Pb Sublattice in the y-axis direction Note that sometimes the 0 and 1 axes directions are constructed vice versa.

```
>>> 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*.

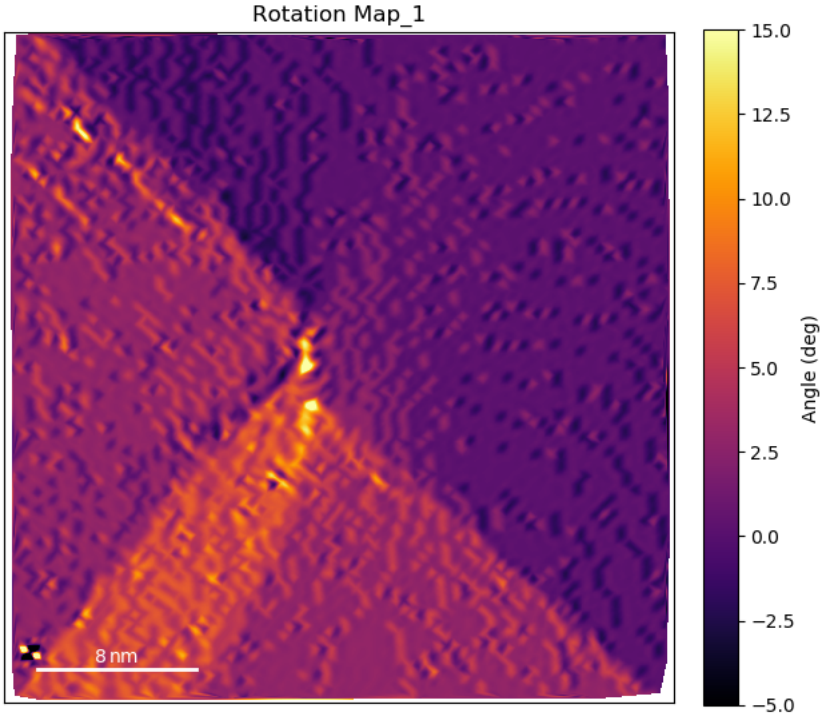
```
>>> kwargs = {'vmin': vmin, 'vmax': vmax, 'cmap': cmap}
>>> tml.compare_images_line_profile_one_image(strain_map, line_profile_positions,
...                                linewidth=100, arrow='h', linetrace=0.05,
...                                **kwargs)
```



Create the Lattice Rotation Map

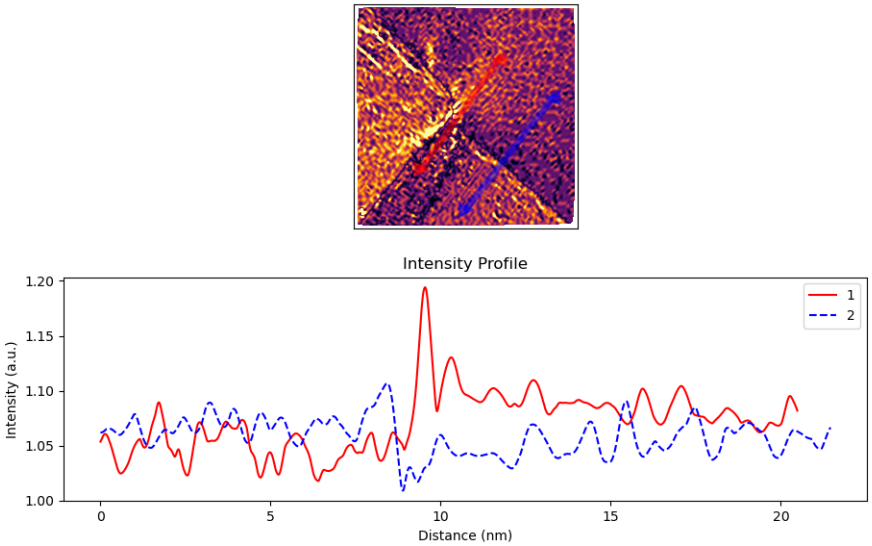
Now plot the rotation map of the Pb Sublattice in the x-axis direction to see the turning of the lattice across the junction.

```
>>> 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*.

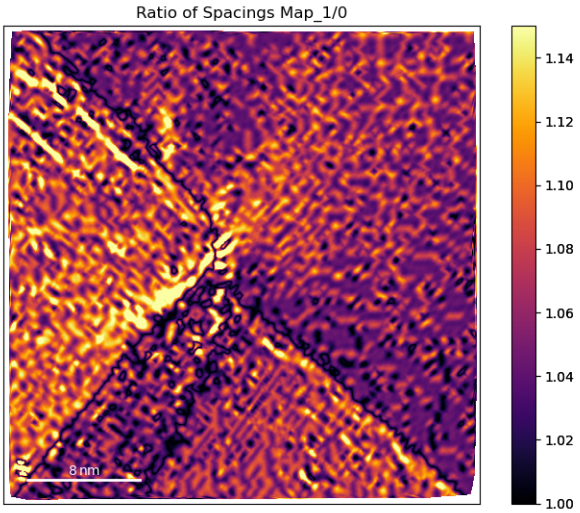
```
>>> kwargs = {'vmin': vmin, 'vmax': vmax, 'cmap': cmap}
>>> tml.compare_images_line_profile_one_image(
...     rotation_map, line_profile_positions, linewidth=100, arrow='h',
...     linetrace=0.05, **kwargs)
```



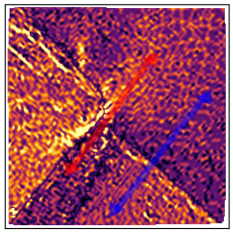
Create the Lattice c/a Ratio Map

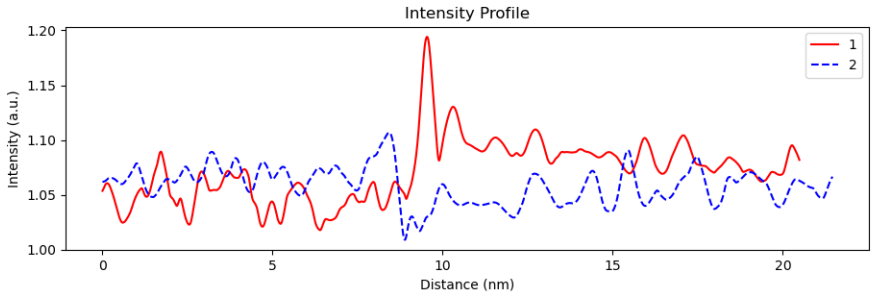
Now plot the c/a ratio map of the Pb Sublattice

```
>>> 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)
```





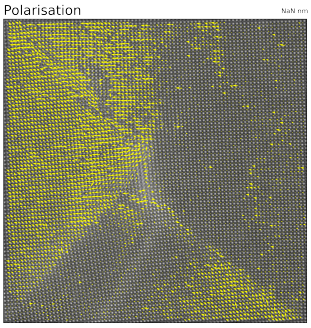
Map the Polarisation

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()
```

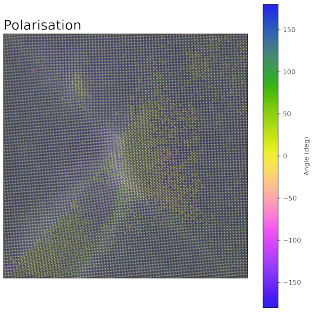
Plot the polarisation vectors (zoom in to get a better look, the top left is off zone).

```
>>> 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)
```



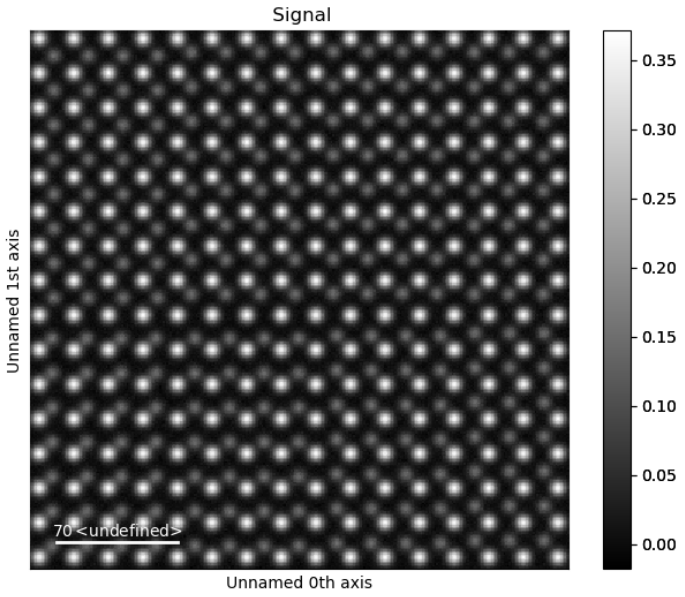
3.4.9 Masked FFT and iFFT

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 Trans-

form (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()
```

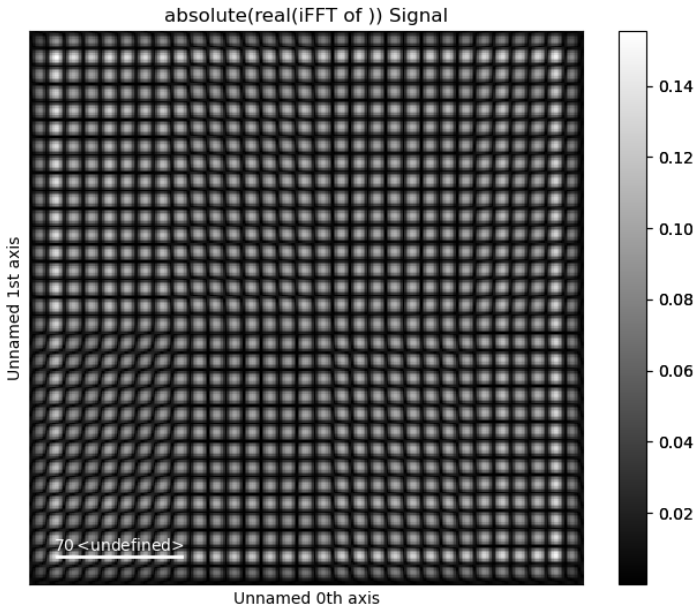


Choose the Mask Coordinates

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

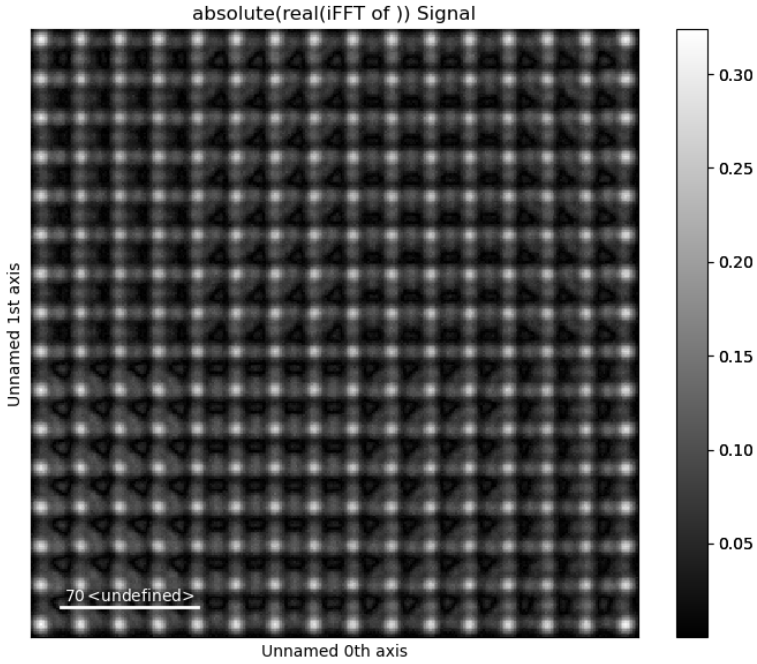
Plot the Masked iFFT

```
>>> 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()
```



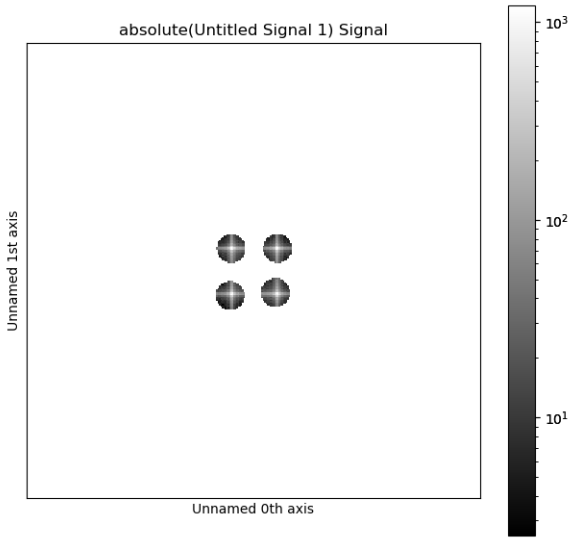
Reverse the masking with *keep_masked_area=False*

```
>>> 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

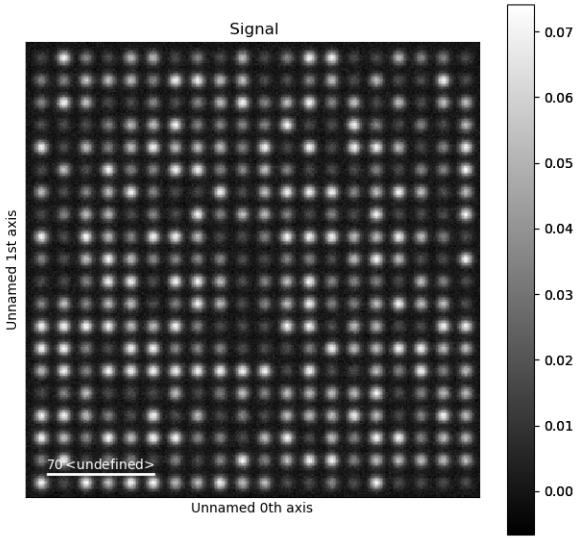
The *temul.signal_plotting* module allows one to draw line intensity profiles over images. The :py:func‘temul.signal_plotting.compare_images_line_profile_one_image‘ can be used to draw two line profiles on one image for comparison. In future we hope to expand this function to allow for multiple line 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.

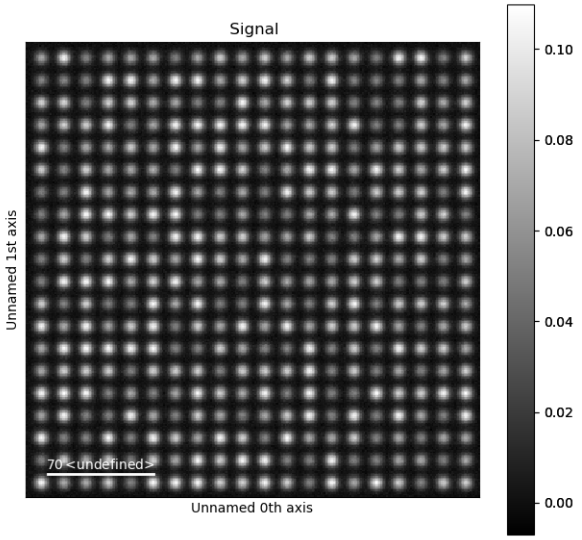
Check out the examples below for each comparison method.

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)
```

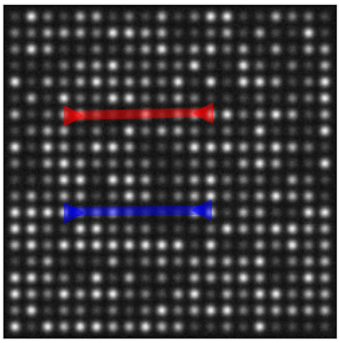
(continues on next page)

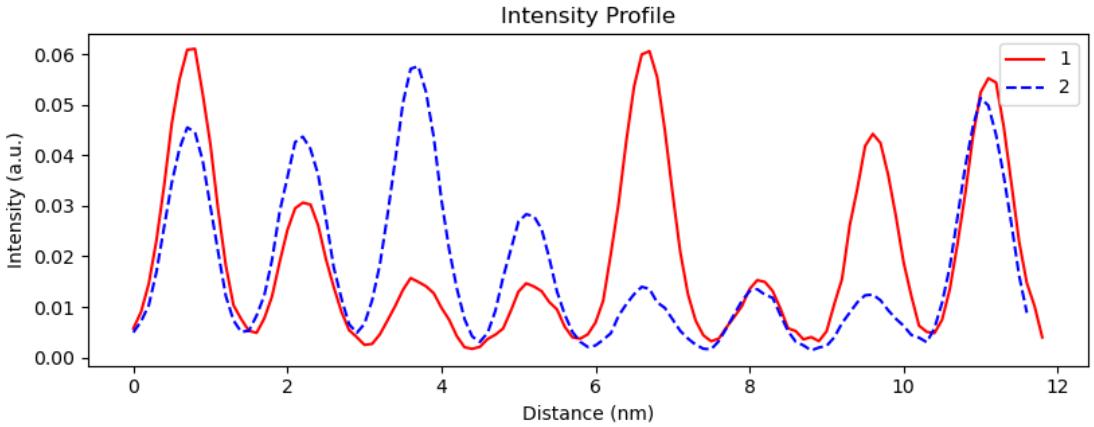
(continued from previous page)

```
>>> 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)
```



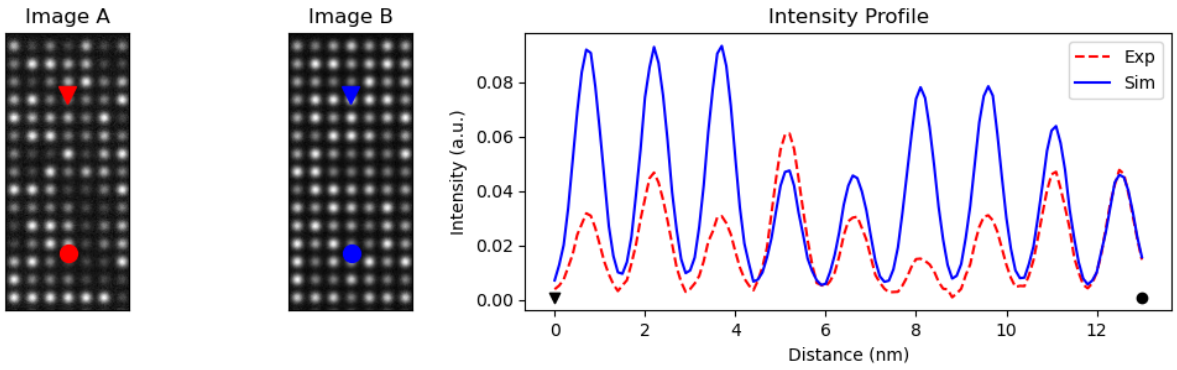


Compare two Images with Line Profile

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)
```

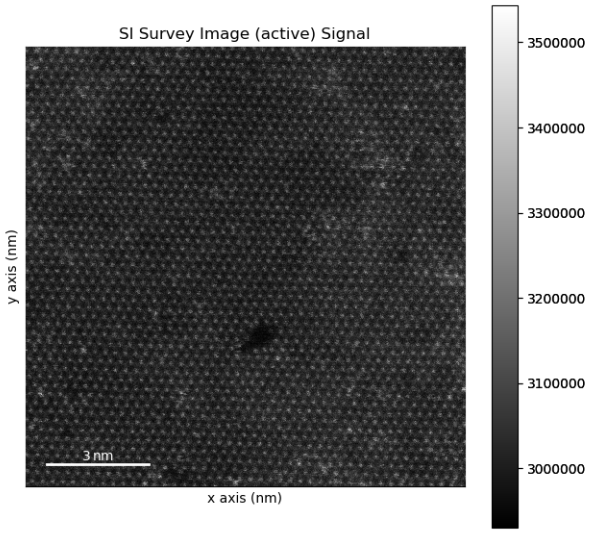
As we can see, an interactive window appears, showing the FFT ("FFT Widget") of the image with the positions of the inner and outer Gaussian full width at half maximums (FWHMs). The inital FWHMs can be changed with the `d_inner` and `d_outer` parameters (limits can also be changed).

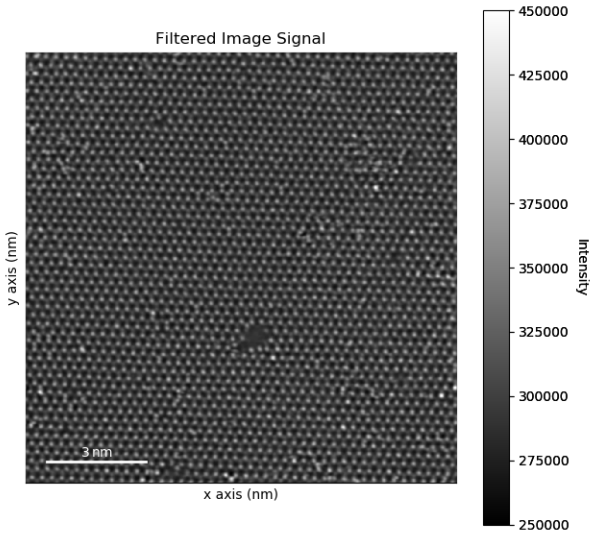
To change the two FWHMs interactively just use the sliders at the bottom of the window. Reset can be used to reset the FWHMs to their initial value, and Filter will display the "Convolution" of the FFT and double Gaussian filter as well as the inverse FFT ("Filtered Image") of this convolution.

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)

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**(*self*, *image_data*, *percent_to_nn=0.4*)

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

Parameters

- **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 summed intensity around the atom with local background subtraction:

```
>>> intensities_total_local = get_sublattice_intensity(
...     sublattice=sublattice,
...     intensity_type="total",
...     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_total = remove_local_background(
...     sublattice, intensity_type="total",
...     background_sub=sublattice)
>>> intensities_max = remove_local_background(
...     sublattice, intensity_type="max",
...     background_sub=sublattice)
```

Model Creation

Image Simulation Functions

Signal Processing

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

- **imageB** (*imageA,*) –
- **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_B** (*marker_A,*) –
- **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
- **header, skiprows, engine** (*delimiter,*) – See pandas.read_csv for documentation Note that the delimiters here are only available if you use engine='python'
- **rms_thermal_vib** (*occupancy,*) – 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.
- **y_size, z_size** (*x_size,*) – 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_end** (*crop_start,*) – 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_b, _cell_length_c** (*cell_length_a,*) – lattice dimensions in angstrom
- **cell_angle_beta, cell_angle_gamma** (*cell_angle_alpha,*) – lattice angles in degrees
- **space_group_name_H-M_alt** (*string*) – space group name
- **space_group_IT_number** (*float*) –

Dummy Data

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 4

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 5

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 6

Code Documentation

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

CHAPTER 7

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 8

Contribute

CHAPTER 9

Support

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

CHAPTER 10

License

The project is licensed under the GPL-3.0 License.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

S

T

W