latticegen

Release 0.0.4

Tobias A. de Jong

Nov 18, 2021

CONTENTS:

Simple python code to interactively generate visualizations of moire patterns of lattices, such as for example magic angle bilayer graphene.

CHAPTER

ONE

INSTALLATION

```
pip install latticegen
```

1.1 Lattice generation

Using latticegen, it is possible to generate different symmetry lattices with different appearances. These lattices can then be combined in moiré lattices, or compound lattices of the same lattice constant

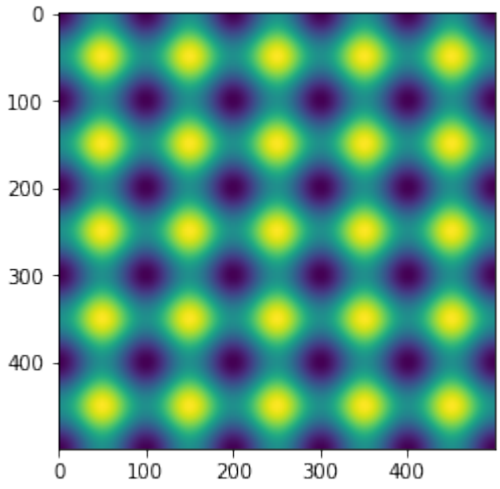
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import latticegen
```

1.1.1 A single lattice

First, let's look at generation of a single square lattice:

[2]:
```
lattice = latticegen.anylattice_gen(r_k=0.01, theta=0,
                                    order=1, symmetry=4)
plt.imshow(lattice.T)
```

[2]:
```
<matplotlib.image.AxesImage at 0x7f5a84673940>
```



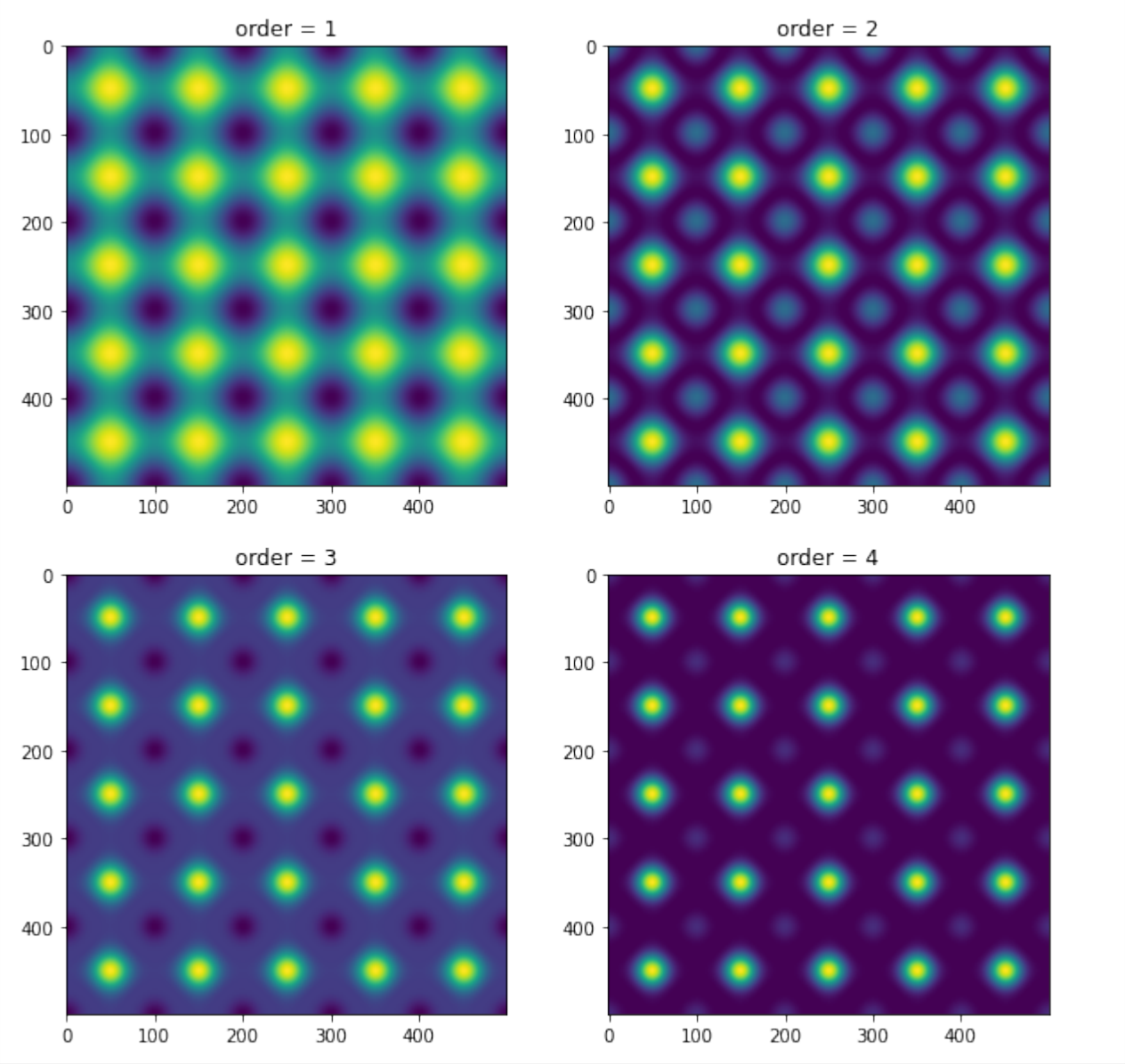
Here, r_k is one over the lattice constant, `theta` is the rotation angle of the lattice, and `symmetry=4` indicates a four-fold symmetric lattice.

Note: r_k is designed to be easily used with diffraction patterns, i.e. FFT transforms of images. If you just want a physical lattice, you might find *latticegen.physical_lattice_gen()* more intuitive.

1.1.2 The `order` parameter

To give more indication of what the `order` parameter, the maximum order of the Fourier/k-vector components does: The higher the order, the more well-resolved the atoms are as single spots. However, computational complexity increases fast.

[3]:
```
fig, ax = plt.subplots(ncols=2, nrows=2, figsize=[10,10])
for i in range(4):
    ax.flat[i].imshow(latticegen.anylattice_gen(r_k=0.01, theta=0,
                                    order=1+i, symmetry=4))
    ax.flat[i].set_title(f'order = {i+1}')
```



1.1.3 Different symmetries

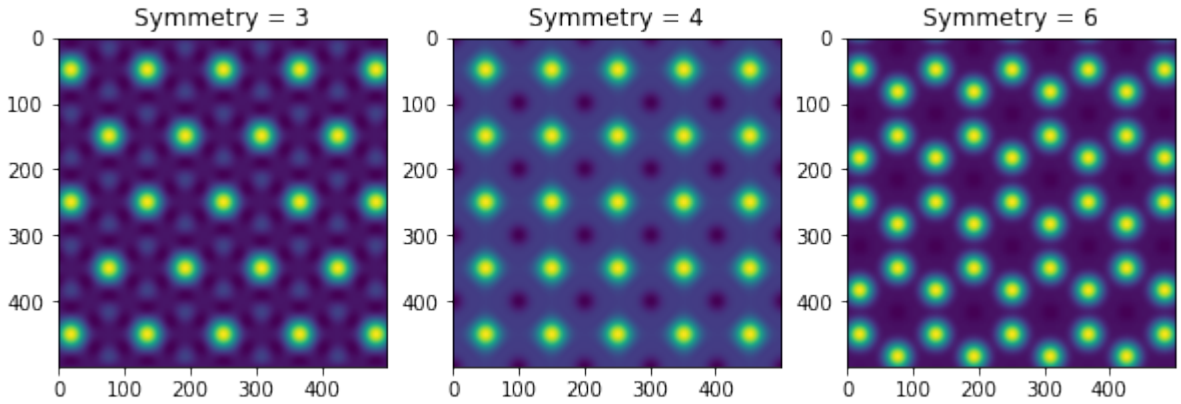
We can generate lattices of six-fold (triangular) symmetry and four-fold symmetry, as wel as an hexagonal lattice. These functions are also available separately as *trilattice_gen()*, *squarelattice_gen()* and *hexlattice_gen()*.

[4]:
```
fig, ax = plt.subplots(ncols=3, figsize=[10,4])
for i, sym in enumerate([3, 4, 6]):
    if sym == 6:
        data = latticegen.hexlattice_gen(r_k=0.01, theta=0,
                                         order=3)
    else:
        data = latticegen.anylattice_gen(r_k=0.01, theta=0,
                                         order=3, symmetry=sym)
    ax.flat[i].imshow(data)
```

(continues on next page)

(continued from previous page)

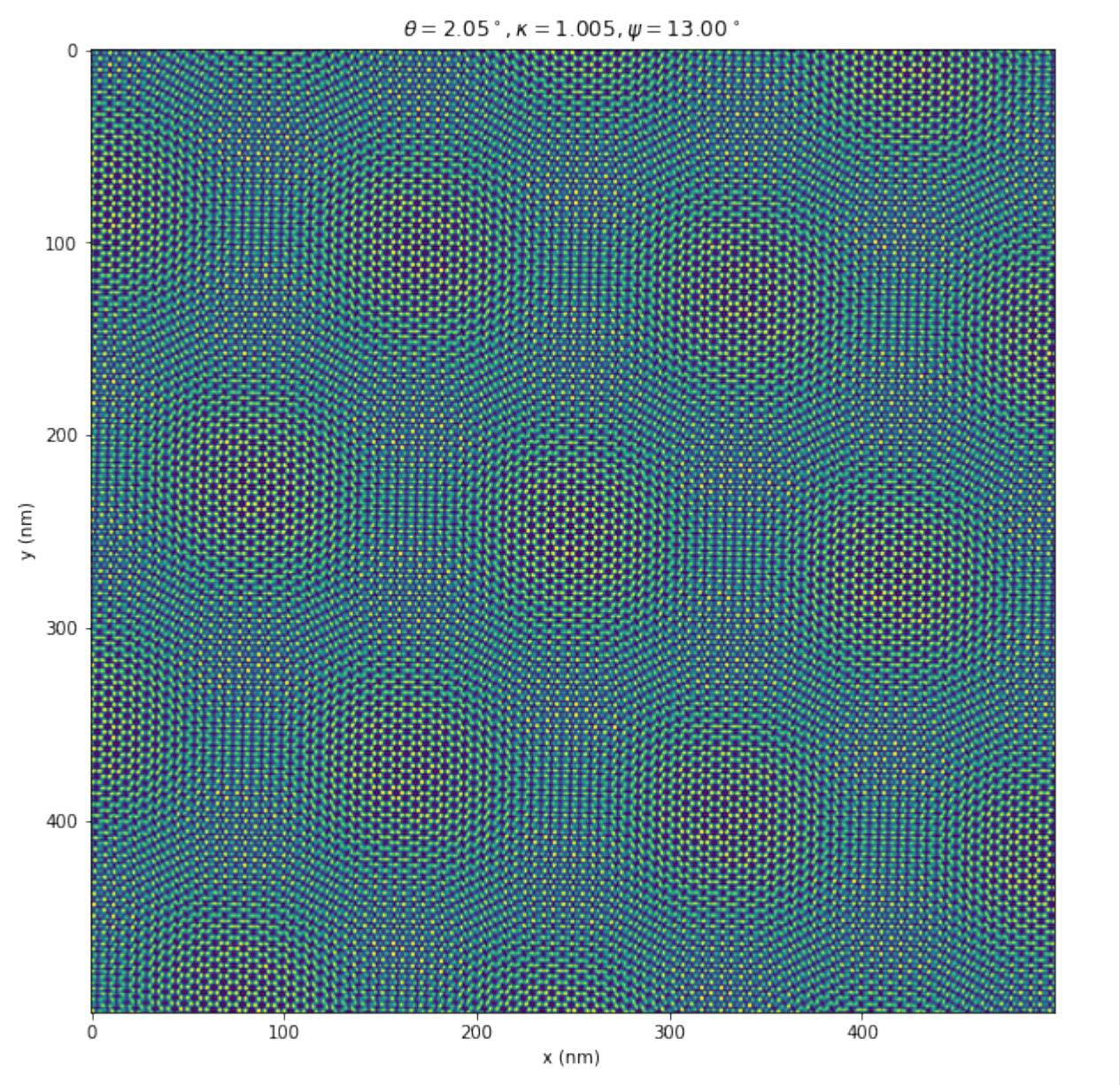
```
    ax.flat[i].set_title(f'Symmetry = {sym}')
```



1.1.4 A moiré superlattice of two lattices

Now, we can visualize what the moiré of two stacked lattices looks like and play around with the influence of deforming the top lattice. We by default drop back to `order=2` to keep things snappy.

[5]:
```
r_k = 0.2
theta=2.05
kappa=1.005
psi=13.
xi=0.

lattice1 = 0.7*latticegen.hexlattice_gen(r_k, xi, 2)
lattice2 = latticegen.hexlattice_gen(r_k, theta+xi, 2,
                                     kappa=kappa, psi=psi)

fig, ax = plt.subplots(figsize=[10,10])


data = (lattice1 + lattice2).compute()
im = ax.imshow(data.T,
               vmax=np.quantile(data,0.95),
               vmin=np.quantile(data,0.05),
               )
ax.set_xlabel('x (nm)')
ax.set_ylabel('y (nm)')
ax.set_title(f'$\\theta = {theta:.2f}^\\circ, \\kappa = {kappa:.3f}, \\psi = {psi:.2f}
↪^\\circ$');
```



1.2 Deformation and Dislocations

Perfectly periodic lattices are interesting, but in practical applications it is often desirable to study imperfect lattices. To compare to such pratical applications, `latticegens` lattice generation functions support general deformations via the `shift=` keyword argument.

This allows for two main classes of deformations: topologically trivial, deformations via a continuous deformation field and edge dislocations, where a discontinuous deformation field corresponds to a missing row of unit cells, yielding a topological point defect.

[1]:
```
import numpy as np
import matplotlib as mpl
```

(continues on next page)

(continued from previous page)

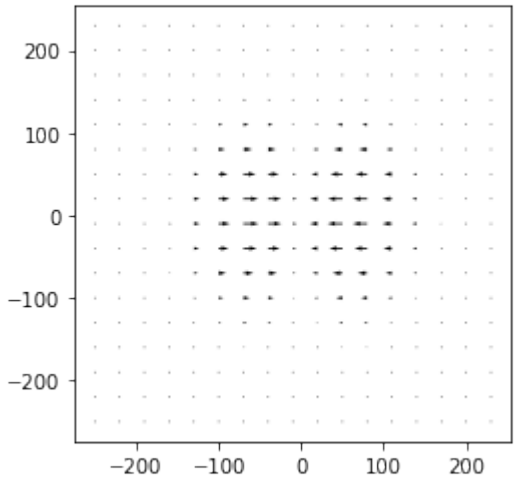
```
import matplotlib.pyplot as plt

import latticegen
from latticegen.singularities import singularity_shift
```

```
Matplotlib is building the font cache; this may take a moment.
```

1.2.1 Continuous deformation field

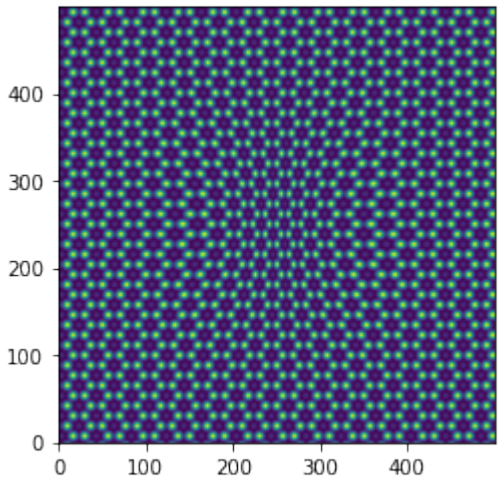
First, let's look at an example of a simple continuous deformation, in the variable `shift`. It should consist of two stacked arrays, each the same size/shape as the intended final lattice, (500x500) by default.

[2]:
```
S = 250
r_k = 0.05
xp, yp = np.mgrid[-S:S, -S:S]
xshift = 0.5*xp*np.exp(-0.5 * ((xp/(2*S/8))**2 + 1.2*(yp/(2*S/6))**2))
shift = np.stack([xshift, np.zeros_like(xshift)])
a = 30
plt.quiver(xp[::a,::a], yp[::a,::a],
           *-shift[:,::a,::a],  # There is a minus sign still wrong.
            units='xy', scale=1, angles='xy')
plt.gca().set_aspect('equal')
```



[3]:
```
lattice = latticegen.hexlattice_gen(r_k=r_k, theta=0,
                                    order=2, shift=shift)
plt.imshow(lattice.T, origin='lower')
```

[3]:
```
<matplotlib.image.AxesImage at 0x7f57ea8c1670>
```

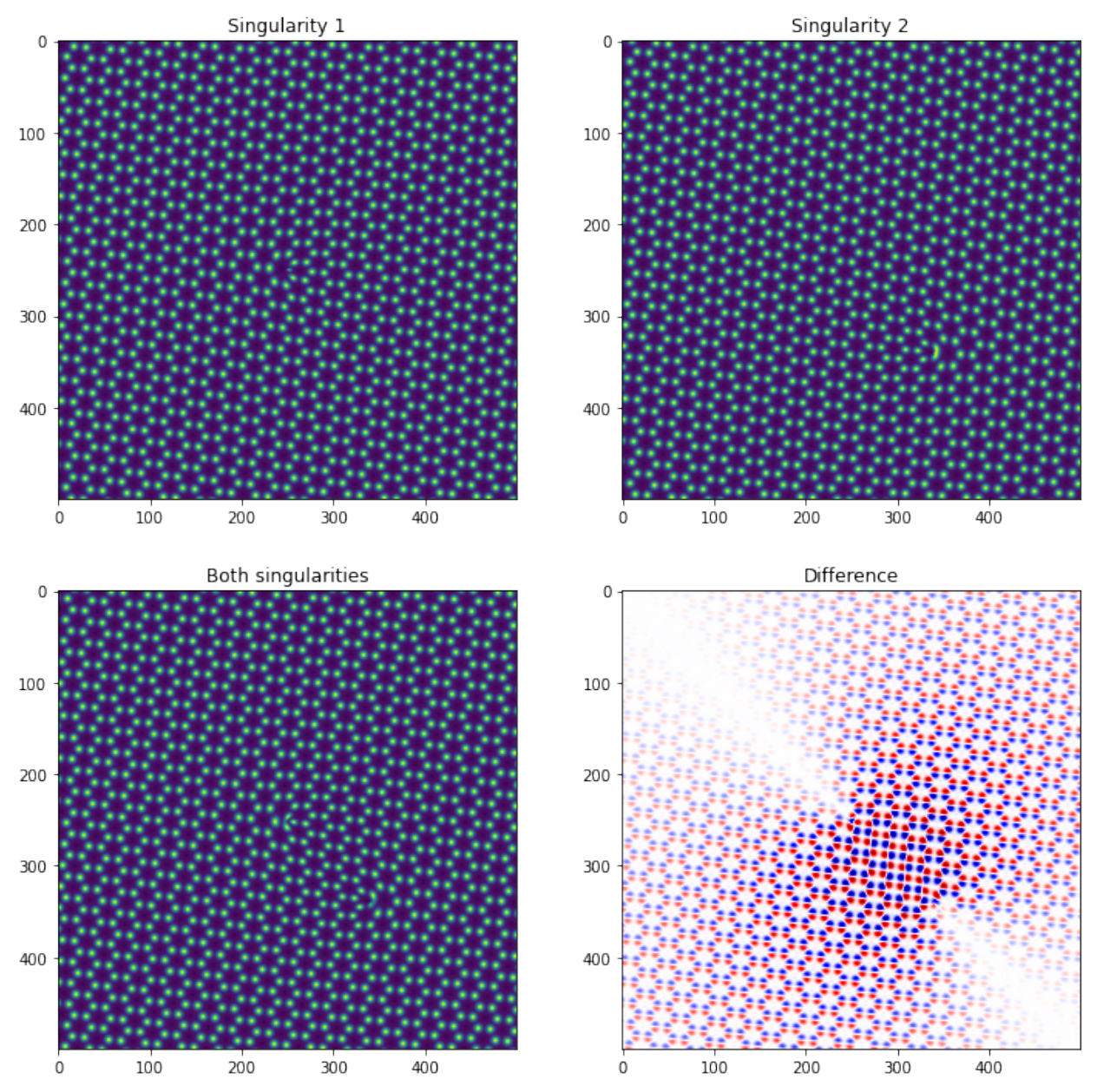


1.2.2 Edge Dislocations

In a two-dimensional lattice, the edge dislocation is the only fundamental dislocation. `latticegen` supports rendering such dislocations using *singularity_shift()* to generate a corresponding deformation array (*dis*continuous now).

[4]:
```
l1 = np.array([90,90])
shift = singularity_shift(r_k, 5)
singularity = latticegen.hexlattice_gen(r_k, 5, 3, shift=shift)
shift2 = singularity_shift(r_k, 5, position=l1, alpha=np.pi)
singularity2 = latticegen.hexlattice_gen(r_k, 5, 3, shift=shift2)
ssum = latticegen.hexlattice_gen(r_k, 5, 3, shift=shift2+shift)
fig, ax = plt.subplots(figsize=[12,12], ncols=2, nrows=2)
ax=ax.flat
ax[0].imshow(singularity.compute().T)
ax[0].set_title('Singularity 1')
ax[1].imshow(singularity2.compute().T)
ax[1].set_title('Singularity 2')
ax[2].imshow(ssum.compute().T)
ax[2].set_title('Both singularities')
ax[3].set_title('Difference')
ax[3].imshow(ssum.compute().T - latticegen.hexlattice_gen(r_k, 5, 3).T.compute(),
             cmap='seismic')
```

[4]:
```
<matplotlib.image.AxesImage at 0x7f57e3fc4e50>
```



1.2.3 Edge dislocation in a moiré lattice

As D.C. Cosma et al. describe, an atomic edge dislocation in a moiré lattice is magnified to a corresponding edge dislocation in the moiré lattice.

With this information a more complex example of the use of `latticegen` is shown below.

To illustrate the effect of the moiré lattice on the dislocation, we recreate an adapted version of the relevant panels from Figure 3 of De Jong et al.

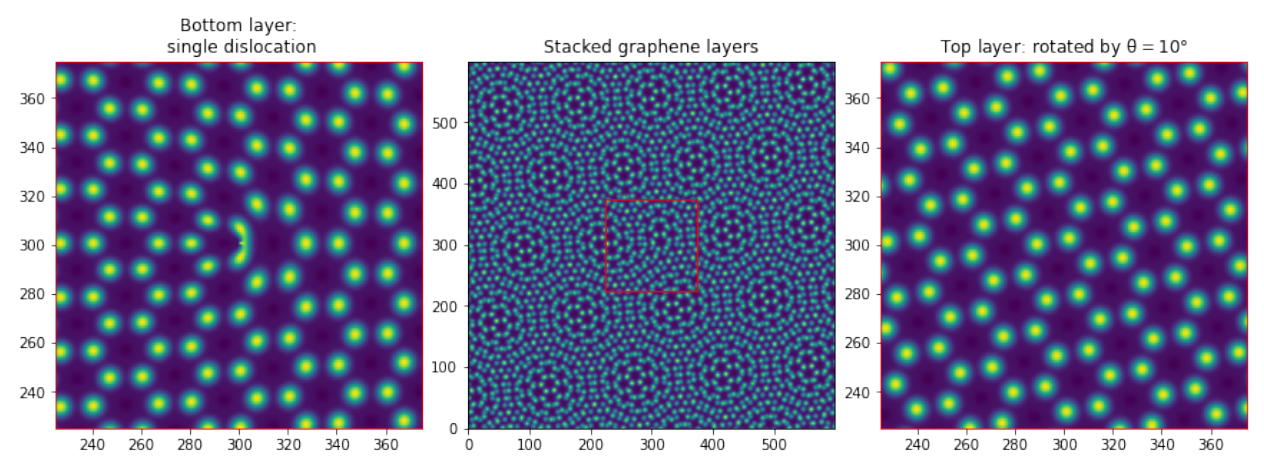
[5]:
```
S = 600  #Size of visualization in pixels.
r_k = 0.05
xi0 = 0
```

(continues on next page)

(continued from previous page)

```
alphai=3
sshift = singularity_shift(r_k, xi0, S, alpha=2*np.pi*alphai/6)
lattice1 = 0.7 * latticegen.hexlattice_gen(r_k, xi0, 3, S,
                                           shift=sshift, chunks=400).compute()
lattice1 -= lattice1.min()

theta = 10
l2_shift = np.array([-1.5, -2])
lattice2 = latticegen.hexlattice_gen(r_k, xi0+theta, 3, S,
                                     shift=l2_shift, chunks=400).compute()
lattice2 -= lattice2.min()
moire = np.sqrt((lattice1)**2 + (lattice2)**2)

z = 4
r = slice((z-1)*S//z//2, (z+1)*S//z//2)
```

[6]:
```
fig, axs = plt.subplots(ncols=3, figsize=[12, 5], constrained_layout=True)

inset_extent = [r.start, r.stop, r.start, r.stop]

axs[0].imshow(lattice1[r,r].T,
              origin='lower',
              extent=inset_extent,
                  )
axs[1].imshow(moire.T,
               origin='lower',
          )
axs[2].imshow(lattice2[r,r].T,
                origin='lower',
                extent=inset_extent,
      )

axs[0].set_title('Bottom layer:\n single dislocation')
axs[1].set_title('Stacked graphene layers')
axs[2].set_title(f'Top layer: rotated by $ = ${theta}°')

rect = mpl.patches.Rectangle((r.start,r.start),
                             (r.stop-r.start), (r.stop-r.start),
                             edgecolor='red',
                             facecolor='none')
axs[1].add_patch(rect)

for ax in [axs[0], axs[2]]:
    for axis in ['top','bottom','left','right']:
        ax.spines[axis].set_color("red")
```



1.3 Schematics: Transparency tricks

Although `latticegen` is primarily focussed on generating numerical data to test algorithms (e.g. GPA) on, it is also possible to use it to generate more schematic renderings of lattices. To visualize bonds and combine different lattices, it is possible to play transparency tricks with `matplotlibs imshow()` and the diverging colormaps.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import latticegen
```

1.3.1 A hexagonal moiré lattice

By applying a hexagonal lattice of `order=1` itself as transparency, we obtain something nicely respresenting the bonds in the lattice.

Note: In `matplotlib`, `alpha` values are clipped between 0 and 1. Here, the lattices are scaled to $max = 1$ instead of using the `normalize` kwarg of `anylattice_gen()`, to obtain slightly moire transparent lattices overall.

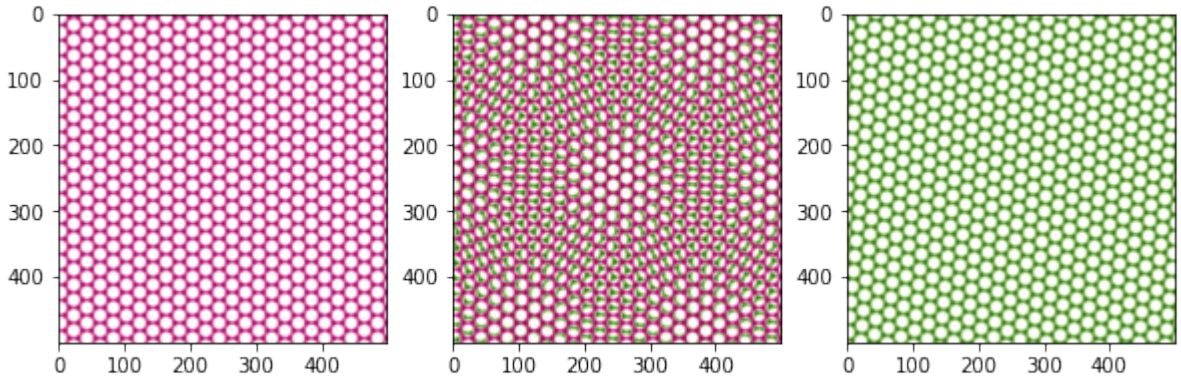
---

[2]:
```
r_k = 0.05
lattice1 = latticegen.hexlattice_gen(r_k, 0, order=1)
lattice1 = np.clip(lattice1 / lattice1.max(), 0, 1).compute()
lattice2 = latticegen.hexlattice_gen(r_k, 5, order=1)
lattice2 = np.clip(lattice2 / lattice2.max(), 0, 1).compute()

fig,axs = plt.subplots(ncols=3, figsize=[10,4])
for i in [0, 1]:
    axs[i].imshow(-lattice1.T, cmap='PiYG',
                        vmax=1,
                        vmin=-1,
                         alpha=lattice1.T
                       )
    axs[i + 1].imshow(lattice2.T, cmap='PiYG',
                      vmax=1,
```

(continues on next page)

(continued from previous page)

```
                  vmin=-1,
                  alpha=lattice2.T)
```

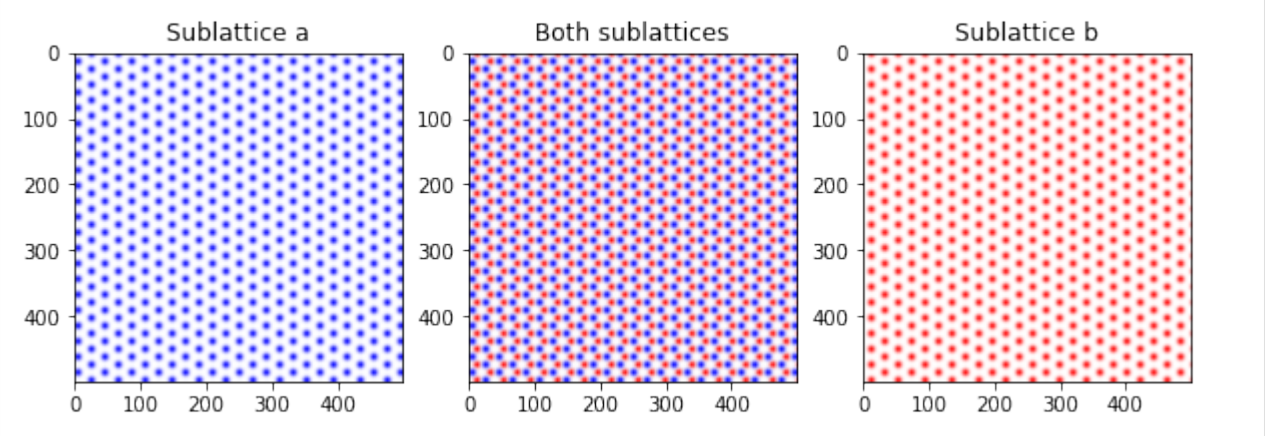


1.3.2 A diatomic lattice

By combining two separate trigonal lattices, we can illustrate a diatomic hexagonal lattice such as the insulator hexagonal Boron Nitride (hBN). A different colormap is used to generate different colors.

(Any combination of colors can be obtained by creating a custom colormap)

[3]:
```
r_hBN = r_k * (0.246 / 0.2504)
sublattice_a = latticegen.trilattice_gen(r_hBN, 0, order=1, normalize=True)
sublattice_a = sublattice_a.compute()
# Now add the second shifted sublattice lattice to get a hexagonal lattice
ks = latticegen.generate_ks(r_hBN, 0, sym=6)
x = np.array([ks[1], -ks[2]])
shift = (np.linalg.inv(x / r_hBN).T/(3*r_k)).sum(axis=0).T  # Don't ask, this works
sublattice_b = latticegen.trilattice_gen(r_hBN, 0, order=1,
                                         shift=shift, normalize=True)
sublattice_b = sublattice_b.compute()

fig, axs = plt.subplots(ncols=3, figsize=[10,4])
axs[0].set_title('Sublattice a')
axs[1].set_title('Both sublattices')
axs[2].set_title('Sublattice b')
for i in [0, 1]:
    axs[i].imshow(-sublattice_a.T, cmap='bwr',
                  vmax=1, vmin=-1,
                  alpha=sublattice_a.T)
    axs[i + 1].imshow(sublattice_b.T, cmap='bwr',
                      vmax=1, vmin=-1,
                      alpha=sublattice_b.T)
```

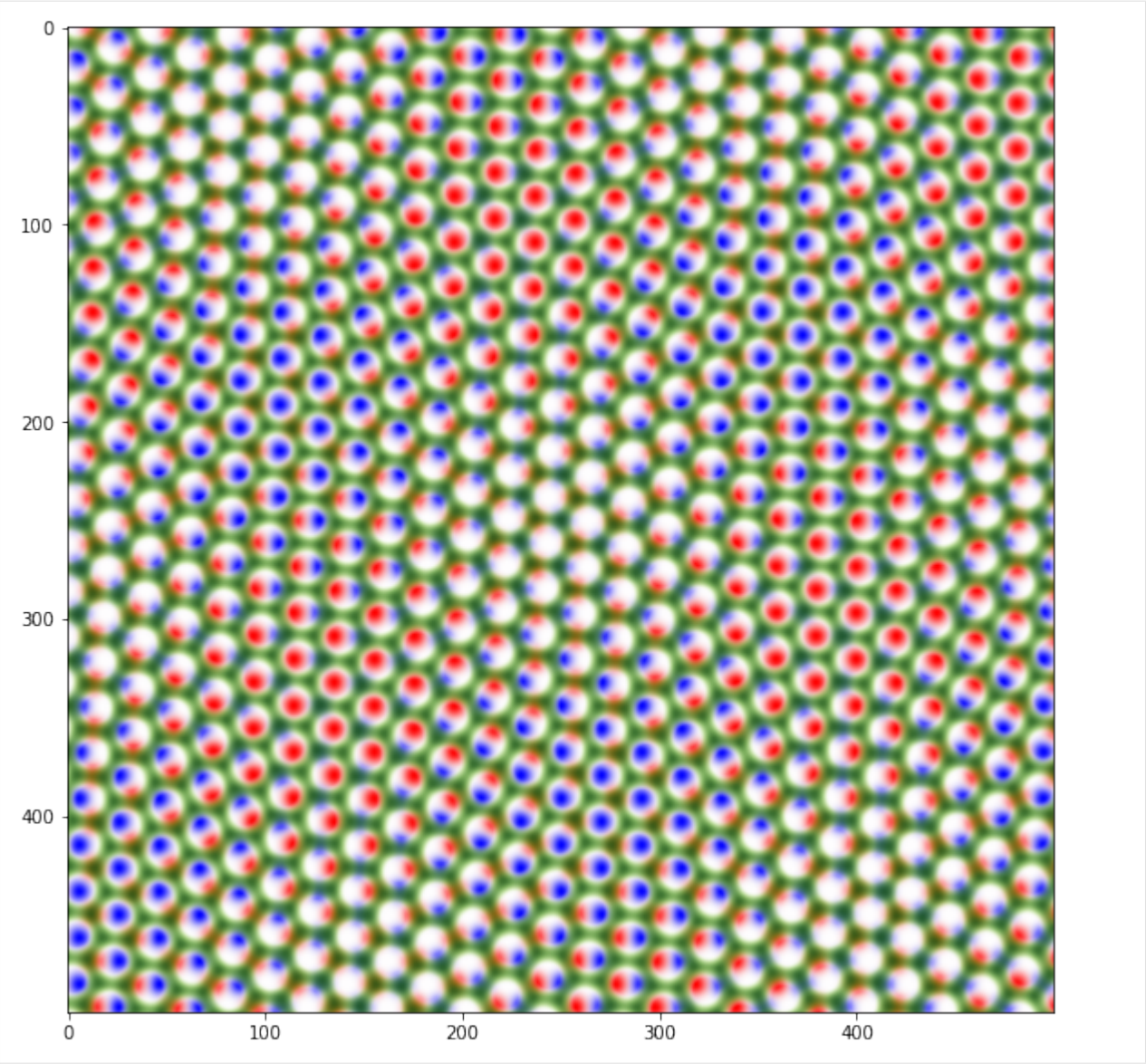


1.3.3 A moiré of a diatomic lattice and a hexagonal lattice

Putting both examples together to create an image of a moiré of a graphene lattice (green) on top of a hBN lattice (red/blue):

[4]:
```
plt.figure(figsize=[10,10])
plt.imshow(-sublattice_a.T, cmap='bwr',
           vmax=1, vmin=-1,
           alpha=sublattice_a.T)
plt.imshow(sublattice_b.T, cmap='bwr',
           vmax=1, vmin=-1,
           alpha=sublattice_b.T)
plt.imshow(lattice2.T, cmap='PiYG',
           vmax=1, vmin=-1,
           alpha=lattice2.T*0.9)
```

[4]:
```
<matplotlib.image.AxesImage at 0x7f40f579c2e0>
```



1.4 Quasi-crystals

1.4.1 Generalizing beyond the obvious symmetries

When using `anylattice_gen()`, we can choose any rotational symmetry. This generalizes beyond just 2-fold, 4-fold and 3/6-fold symmetry. So let's see what happens if we use 5, or 7, rotational symmetries were no periodic lattice / covering of the plane exists… Some pretty wild patterns might appear!

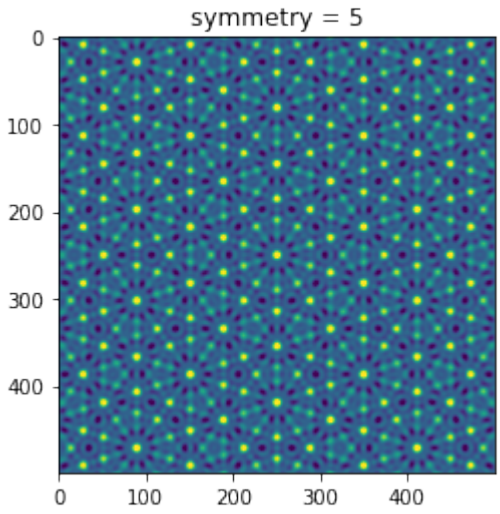
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import dask.array as da

from latticegen import anylattice_gen, generate_ks, combine_ks
```

(continues on next page)

(continued from previous page)

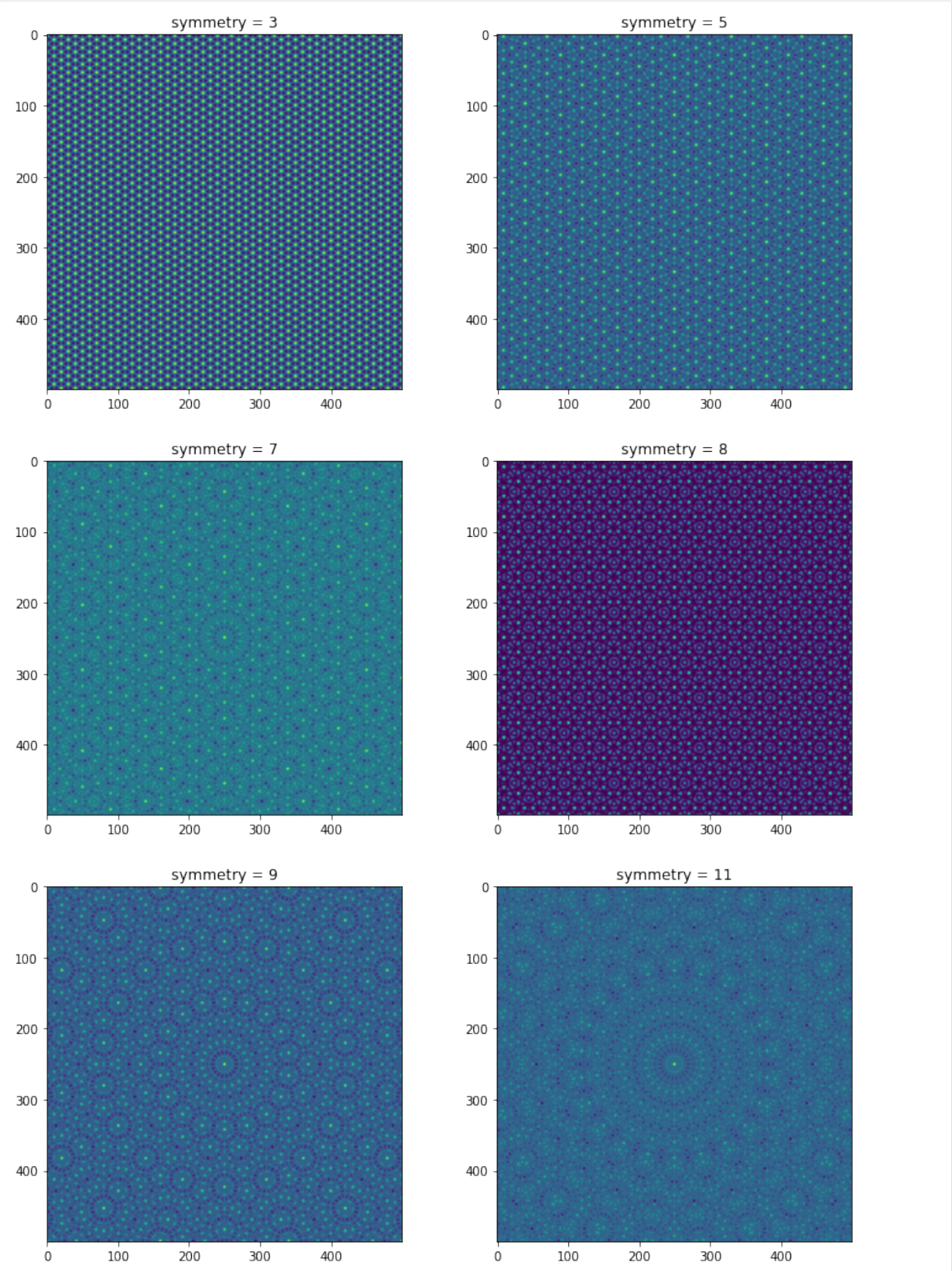
```
from latticegen.transformations import *
```

[2]:
```
data = anylattice_gen(0.05, 0, order=2, symmetry=5).compute()
plt.imshow(data.T,
               vmax=np.quantile(data,0.99),
               vmin=np.quantile(data,0.01),
              )
plt.title('symmetry = 5');
```



This pattern *looks* like it repeats, but it never actually precisely does… We can do this for all kind of different symmetries of course:

[3]:
```
fig, ax = plt.subplots(ncols=2, nrows=3, figsize=[12,18])
for i,sym in enumerate([3,5,7,8,9,11]):
    data = anylattice_gen(0.1, 0, order=2, symmetry=sym).compute()
    ax.flat[i].imshow(data.T)
    ax.flat[i].set_title(f'symmetry = {sym}')
```

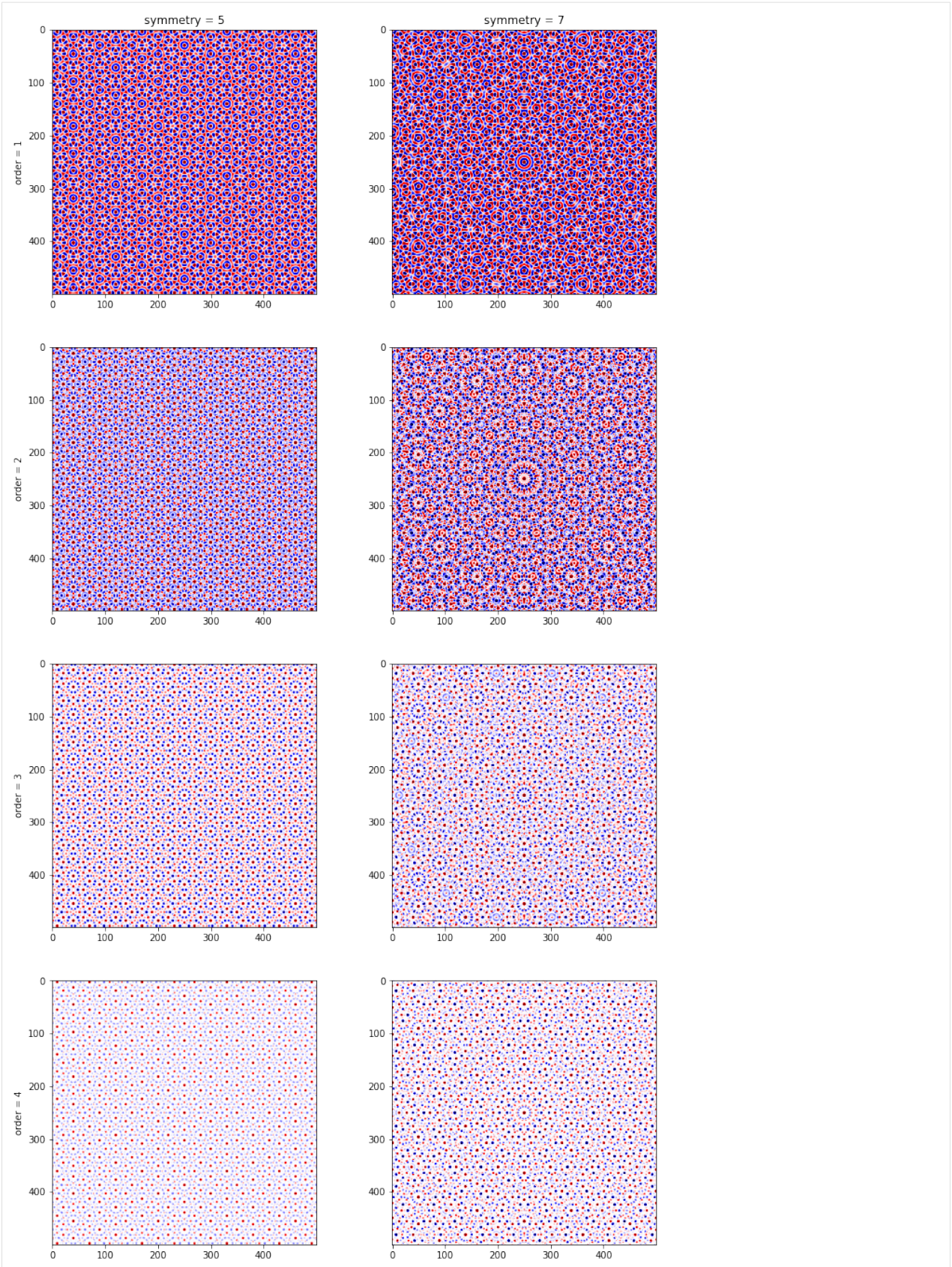

symmetry = 3
symmetry = 5
symmetry = 7
symmetry = 8
symmetry = 9
symmetry = 11
0
100
200
300
400

The 8-fold and 9-fold symmetric patterns are actually closely related to moiré patterns of square and 6-fold symmetric lattices...

1.4.2 Effect of the order parameter

Now we can of course also explore what influence the order parameter has on the patterns. (This is due how the basis vectors are combined. Checkout `combine_ks()` to see how.)

[4]:
```
fig, ax = plt.subplots(ncols=2, nrows=4, figsize=[12,24])
ax = ax.T
for i in np.arange(1,5):
    data = anylattice_gen(0.1, 0, i, symmetry=5).compute()
    mean = data.mean()
    dv = max(data.max()-np.quantile(data,0.99), data.mean()-np.quantile(data,0.01))
    im = ax[0, i-1].imshow(data.T, cmap='seismic',
                   vmax=mean+dv, vmin=mean-dv,
                  )
    ax[0, 0].set_title('symmetry = 5')
    ax[0, i-1].set_ylabel(f'order = {i}')
    data = anylattice_gen(0.1, 0, i, symmetry=7).compute()
    im = ax[1, i-1].imshow(data.T, cmap='seismic',
                vmax=mean+dv, vmin=mean-dv,
               )
    ax[1,0].set_title('symmetry = 7')
```


symmetry = 5
symmetry = 7
order = 1
order = 2
order = 3
order = 4
0
100
200
300
400

1.5 API

1.5.1 Latticegeneration module

Code for generating (quasi-)lattices and the corresponding k-vectors

latticegen.latticegeneration.**anylattice_gen**(*r_k*, *theta*, *order*, *symmetry=6*, *size=500*, *kappa=1.0*, *psi=0.0*, *shift=array([0, 0])*, *normalize=False*, *chunks=(- 1, - 1)*)

Generate a regular lattice of any symmetry. The lattice is generated from the *symmetry 360/symmetry* degree rotated k-vectors of length *r_k*, further rotated by *theta* degrees. Size either an int for a size*size lattice, or tuple (N,M) for a rectangle N*M. Optionally, the lattice can be strained by a factor kappa in direction psi[1].

With higher order frequency components upto order *order*

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants. 1/r_k is the line spacing in pixels in the resulting image.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **symmetry** (*int*) – symmetry of the lattice.
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **kappa** (*float, default: 1*) – strain/deformation magnitude. 1 corresponds to no strain.
- **psi** (*float, default: 0*) – Principal strain direction with respect to horizontal.
- **shift** (*iterable or array, optional*) – shift of the lattice in pixels. Either a pair (x,y) global shift, or an (2xNxM) array where (NxM) corresponds to *size*.
- **normalize** (*bool, default: False*) – if true, normalize the output values to the interval [0,1].
- **chunks** (*int or pair of int, optional*) – dask chunks in which to divide the returned *lattice*.

Returns **lattice** – The generated lattice.

Return type Dask array

See also:

generate_ks, *physical_lattice_gen*

References

[1] T. Benschop et al., 2020, https://doi.org/10.1103/PhysRevResearch.3.013153

latticegen.latticegeneration.**anylattice_gen_np**(*r_k*, *theta*, *order=1*, *symmetry=6*, *size=50*, *kappa=1.0*, *psi=0.0*, *shift=array([0, 0])*)

Generate a regular lattice of any symmetry in pure numpy.

The lattice is generated from the *symmetry 360/symmetry* degree rotated k-vectors of length r_k, further rotated by *theta* degrees. Size either an int for a size*size lattice, or tuple (N,M) for a rectangle N*M. Optionally, the lattice can be strained by a factor kappa in direction psi[1].

With higher order frequency components upto order *order*

The generated lattice gets returned as a numpy array. Only usable for small values of size and order.

See also:

anylattice_gen

latticegen.latticegeneration.**combine_ks**(*kvecs*, *order=1*, *return_counts=False*)

Generate all possible different sums of kvecs upto order.

Parameters

- **kvecs** (*array-like 2xN*) – k vectors to combine.
- **order** (*int, default: 1*) – Number of different *kvecs* to combine for each resulting *tks*.
- **return_counts** (*bool, default False*) – if True, also return the number of possible combinations for each different combination.

Returns

- **tks** (*(2xM) array of float*) – All possible unique combinations of *kvecs* upto *order*
- **counts** (*(1xM) array of int, optional*) – Number of combinations for each vectors in tks.

latticegen.latticegeneration.**generate_ks**(*r_k*, *theta*, *kappa=1.0*, *psi=0.0*, *sym=6*)

Generate k-vectors from given parameters.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants. 1/r_k is the line spacing in pixels in the resulting image.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **kappa** (*float, default: 1*) – strain/deformation magnitude. 1 corresponds to no strain. Larger values corresponds to stretching along the *psi* direction in real space, so compression along the same direction in k-space.
- **psi** (*float, default: 0*) – Principal strain direction with respect to horizontal in degrees.
- **sym** (*int, default 6*) – Rotational symmetry of the unstrained lattice.

Returns **ks**

Return type np.array (2x(*sym* + 1))

latticegen.latticegeneration.**hexlattice_gen**(*r_k*, *theta*, *order*, *size=500*, *kappa=1.0*, *psi=0.0*, *shift=array([0, 0])*, ***kwargs*)

Generate a regular hexagonal lattice. The lattice is generated from the six 60 degree rotated k-vectors of length

r_k, further rotated by *theta* degrees. Optionally, the lattice can be strained by a factor kappa in direction psi [1]. Rendered with higher order frequency components upto order *order*.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants. 1/r_k is the line spacing in pixels in the resulting image.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **kappa** (*float, default: 1*) – strain/deformation magnitude. 1 corresponds to no strain.
- **psi** (*float, default: 0*) – Principal strain direction with respect to horizontal in degrees.
- **shift** (*iterable or array, optional*) – shift of the lattice in pixels. Either a pair (x,y) global shift, or an (2xNxM) array where (NxM) corresponds to *size*.
- ****kwargs** (*dict*) – Keyword arguments to be passed to *anylattice_gen*

Returns **lattice** – The generated lattice.

Return type Dask array

See also:

anylattice_gen

latticegen.latticegeneration.**hexlattice_gen_fast**(*r_k*, *theta*, *order*, *size=500*, *kappa=1.0*, *psi=0.0*, *shift=array([0, 0])*)

Generate a regular hexagonal lattice.

Speed optimized version, losing some mathematical precision. Tested to be accurate down to max(1e-10*r_k, 1e-10) w.r.t. the regular function. The lattice is generated from the six 60 degree rotated k-vectors of length r_k, further rotated by *theta* degrees. Size either an int for a size*size lattice, or tuple (N,M) for a rectangle N*M. Optionally, the lattice can be strained by a factor kappa in direction psi[1].

With higher order frequency components upto order *order*

The generated lattice gets returned as a dask array.

See also:

anylattice_gen

latticegen.latticegeneration.**physical_lattice_gen**(*a_0*, *theta*, *order*, *pixelspernm=10*, *symmetry='hexagonal'*, *size=500*, *epsilon=None*, *delta=0.16*, ***kwargs*)

Generate a physical lattice

Wraps anylattice_gen. Using a lattice constant *a_0* in nm and a resolution in pixels per nm, generate a rendering of a lattice of *size* pixels. Optionally, the lattice can be strained by a factor kappa in direction psi[1].

With higher order frequency components upto order *order*

Parameters

- **a_0** (*float*) – lattice constant in nm
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **pixelspernm** (*float, default: 10*) – number of pixels per nanometer.
- **symmetry** (*{'hexagonal', 'trigonal', 'square'}*) – symmetry of the lattice.
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **epsilon** (*float*) – Lattice strain
- **delta** (*float, default=0.16*) – Poisson ratio for lattice strain. Default value corresponds to graphene.
- ****kwargs** (*dict*) – Keyword arguments to be passed to *anylattice_gen*

Returns **lattice** – The generated lattice.

Return type Dask array

See also:

anylattice_gen

latticegen.latticegeneration.**squarelattice_gen**(*r_k*, *theta*, *order*, *size=500*, *kappa=1.0*, *psi=0.0*, *shift=array([0, 0])*, ***kwargs*)

Generate a regular square lattice.

The lattice is generated from the four 90 degree rotated k-vectors of length *r_k*, further rotated by *theta* degrees. Optionally, the lattice can be strained by a factor *kappa* in direction *psi`[1]. Rendered with higher order frequency components upto order `order*.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **kappa** (*float, default: 1*) – strain/deformation magnitude. 1 corresponds to no strain.
- **psi** (*float, default: 0*) – Principal strain direction with respect to horizontal.
- **shift** (*iterable or array, optional*) – shift of the lattice in pixels. Either a pair (x,y) global shift, or an (2xNxM) array where (NxM) corresponds to *size*.
- ****kwargs** (*dict*) – Keyword arguments to be passed to *anylattice_gen*

Returns **lattice** – The generated lattice.

Return type Dask array

See also:

anylattice_gen

latticegen.latticegeneration.**trilattice_gen**(*r_k*, *theta*, *order*, *size=500*, *kappa=1.0*, *psi=0.0*, *shift=array([0, 0])*, ***kwargs*)

Generate a regular trigonal lattice.

The lattice is generated from the six 60 degree rotated k-vectors of length *r_k*, further rotated by *theta* degrees. Optionally, the lattice can be strained by a factor *kappa* in direction *psi`[1]. Rendered with higher order frequency components upto order `order*.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **kappa** (*float, default: 1*) – strain/deformation magnitude. 1 corresponds to no strain.
- **psi** (*float, default: 0*) – Principal strain direction with respect to horizontal.
- **shift** (*iterable or array, optional*) – shift of the lattice in pixels. Either a pair (x,y) global shift, or an (2xNxM) array where (NxM) corresponds to *size*.
- ****kwargs** (*dict*) – Keyword arguments to be passed to *anylattice_gen*

Returns **lattice** – The generated lattice.

Return type Dask array

See also:

anylattice_gen, *hexlattice_gen*

1.5.2 Singularities module

Code to generate edge dislocations/singularities in lattices

latticegen.singularities.**gen_dists_image**(*peaks*, *imageshape*, *rmax=55*)

Generate squared distance to peaks

Parameters

- **peaks** (*(N,2) array_like*) – peak locations in image
- **imageshape** (*pair of int*) – shape of the original image
- **rmax** (*float, default 55*) – maximum radius to use around each peak

Returns **res** – Array of shape *imageshape*, with the squared distance to the nearest peak in float, with a maximum value of *rmax*.

Return type array_like

latticegen.singularities.**hexlattice_gen_singularity**(*r_k*, *theta*, *order*, *size=250*, *position=[0, 0]*, *shift=array([0, 0])*, ***kwargs*)

Generate a hexagonal lattice with a singularity.

Singularity is shifted *position* from the center. Not yet equivalent to hexlattice_gen_singularity_legacy.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **kappa** (*float, default: 1*) – strain/deformation magnitude. 1 corresponds to no strain.
- **position** (*iterable, default [0, 0]*) – [x, y] position of the singularity in pixels with respect to the center.
- **shift** (*iterable or array, optional*) – shift of the lattice. Either a pair (x,y) global shift, or an (2xNxM) array where (NxM) corresponds to *size*.
- ****kwargs** (*dict*) – Keyword arguments to be passed to *hexlattice_gen*

Returns **lattice** – The generated lattice with singularity

Return type Dask array

See also:

hexlattice_gen, *singularity_shift*

latticegen.singularities.**hexlattice_gen_singularity_legacy**(*r_k*, *theta*, *order*, *size=250*)

Generate a regular hexagonal lattice of *2*size* times *2*size*. The lattice is generated from the six 60 degree rotations of *k0*, further rotated by *theta* degrees. With higher order frequency components upto order *order* and containing a singularity in the center. *theta != 0* does not yield a correct lattice.

The generated lattice gets returned as a dask array.

latticegen.singularities.**refine_peaks**(*image*, *peaks*)

Refine peak locations using a bivariate spline

Uses scipy.interpolate.RectBivariateSpline to interpolate image, and scipy.optimize.minimize with bounds to find the interpolated maximum.

Parameters

- **image** (*2D array*) – image to interpolate
- **peaks** (*(2,N) array of ints*) – list of peaks to refine.

Returns **interppeaks** – The refined peak locations.

Return type (2,N) array of floats

latticegen.singularities.**refined_singularity**(*r_k*, *theta=0*, *order=3*, *S=500*, *remove_center_atom=True*, *position=array([0., 0.])*, ***kwargs*)

Created a refined singularity without distorted atoms in the center

Generate a lattice with a dislocation, then extract atom positions using *subpixel_peak_local_max*. Optionally remove the atom at the position of the dislocation, add a gaussian blob at each atom position.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **S** (*int, default: 500*) – Size of the resulting lattice in pixels. The returned lattice will be square.
- **position** (*iterable, default [0, 0]*) – [x, y] position of the singularity in pixels with respect to the center.
- **remove_center_atom** (*bool, default=True*) – whether to remove the atom at the center of the dislocation
- ****kwargs** (*dict*) – Keyword arguments to be passed to *hexlattice_gen*

Returns **lattice** – The generated refined lattice with singularity

Return type numpy array

See also:

hexlattice_gen_singularity

latticegen.singularities.**singularity_shift**(*r_k*, *theta*, *size=500*, *position=[0, 0]*, *alpha=0.0*, *symmetry=6*)

Generate the shift of an edge dislocation.

Generate the shift / displacement field for a hexagonal lattice of *size*, with the Burgers vector at angle *alpha* (radians) w.r.t. *theta* (in degrees). Burgers vector has length *np.sin(2*np.pi/symmetry) / r_k*, to create a first order edge dislocation.

Parameters

- **r_k** (*float*) – length of lattice vectors in k-space. Larger *r_k* correspond to smaller real space lattice constants.
- **theta** (*float*) – Angle of the first lattice vector with respect to positive horizontal.
- **order** (*int*) – Order upto which to generate higher frequency components by combining lattice vectors
- **size** (*int, or pair of int, default: 500*) – Size of the resulting lattice in pixels. if int, the returned lattice will be square.
- **position** (*iterable, default: [0, 0]*) – [x, y] position of the singularity in pixels with respect to the center.
- **alpha** (*float, default 0.0*) – angle in radians of the Burgers vector with respect to the first lattice vector.
- **symmetry** (*int, default: 6*) – symmetry of the lattice.

Returns **shift** – The generated shift corresponding to a singularity, which can be passed to anylattice_gen

Return type ndarray

See also:

anylattice_gen

latticegen.singularities.**subpixel_peak_local_max**(*image*, ***kwargs*)
A subpixel accurate local peak find

Wraps skimage.feature.peak_local_max to return subpixel accurate peak locations from a BivariateSpline interpolation.

Parameters

- **image** (*2D array*) – image to find maxima in
- ****kwargs** (*dict*) – keyword arguments passed to skimage.feature.peak_local_max. indices=False is not supported.

Returns Containing the coordinates of the peaks

Return type (N,2) array

1.5.3 Transformations module

Common code for geometrical transformations

latticegen.transformations.**a_0_to_r_k**(*a_0*, *symmetry=6*)
Transform realspace lattice constant to r_k

Where $r_k = (a_0sin(2pi/symmetry))^{-1}$. i.e. r_k is 1/(2pi) the reciprocal lattice constant.

Parameters

- **a_0** (*float*) – realspace lattice constant
- **symmetry** (*integer*) – symmetry of the lattice

Returns **r_k** – length of k-vector in frequency space

Return type float

latticegen.transformations.**apply_transformation_matrix**(*vecs*, *matrix*)
Apply transformation matrix to a list of vectors.

Apply transformation matrix *matrix* to a list of vectors *vecs*. *vecs* can either be a list of vectors, or a NxM array, where N is the number of M-dimensional vectors.

Parameters

- **vecs** (*2D array or iterable*) – list of vectors to be transformed
- **matrix** (*2D array*) – Array corresponding to the transformation matrix

Returns Transformed vectors

Return type 2D array

latticegen.transformations.**epsilon_to_kappa**(*r_k*, *epsilon*, *delta=0.16*)
Convert frequency r_k and strain epsilon to corresponding r_k and kappa as consumed by functions in *latticegeneration*.

Returns

- **r_k2** (*float*)
- **kappa** (*float*)

See also:

latticegeneration.generate_ks

latticegen.transformations.**r_k_to_a_0**(*r_k*, *symmetry=6*)
Transform r_k to a realspace lattice constant a_0

Parameters

- **r_k** (*float*) – length of k-vector in frequency space as used by lattigeneration functions
- **symmetry** (*integer*) – symmetry of the lattice

Returns **a_0** – realspace lattice constant

Return type float

latticegen.transformations.**rotate**(*vecs*, *angle*)
Rotate 2D vectors *vecs* by *angle* radians around the origin.

latticegen.transformations.**rotation_matrix**(*angle*)
Create a rotation matrix

an array corresponding to the 2D transformation matrix of a rotation over *angle*.

Parameters **angle** (*float*) – rotation angle in radians

Returns 2D transformation matrix corresponding to the rotation

Return type ndarray (2x2)

latticegen.transformations.**scaling_matrix**(*kappa*, *dims=2*)
Create a scaling matrix

Creates a numpy array containing the *dims*-dimensional scaling matrix scaling the first dimension by *kappa*.

Parameters

- **kappa** (*float*) – scaling factor of first dimension
- **dims** (*int, default: 2*) – number of dimensions

Returns scaling matrix corresponding to scaling the first dimension by a factor *kappa*

Return type ndarray (dims x dims)

latticegen.transformations.**strain_matrix**(*epsilon*, *delta=0.16*, *axis=0*, *diff=True*)
Create a scaling matrix corresponding to uniaxial strain

Only works for the 2D case.

Parameters

- **epsilon** (*float*) – applied strain
- **delta** (*float, default 0.16*) – Poisson ratio. default value corresponds to graphene
- **axis** (*{0, 1}*) – Axis along which to apply the strain.
- **diff** (*bool, default True*) – Whether to apply in diffraction or real space.

Returns scaling matrix corresponding to *epsilon* strain along *axis*

Return type ndarray (2 x 2)

latticegen.transformations.**wrapToPi**(*x*)
Wrap all values of *x* to the interval [-pi,pi)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX