xcdskd

Release v0.1-29-g8cf606c

Aimo Winkelmann

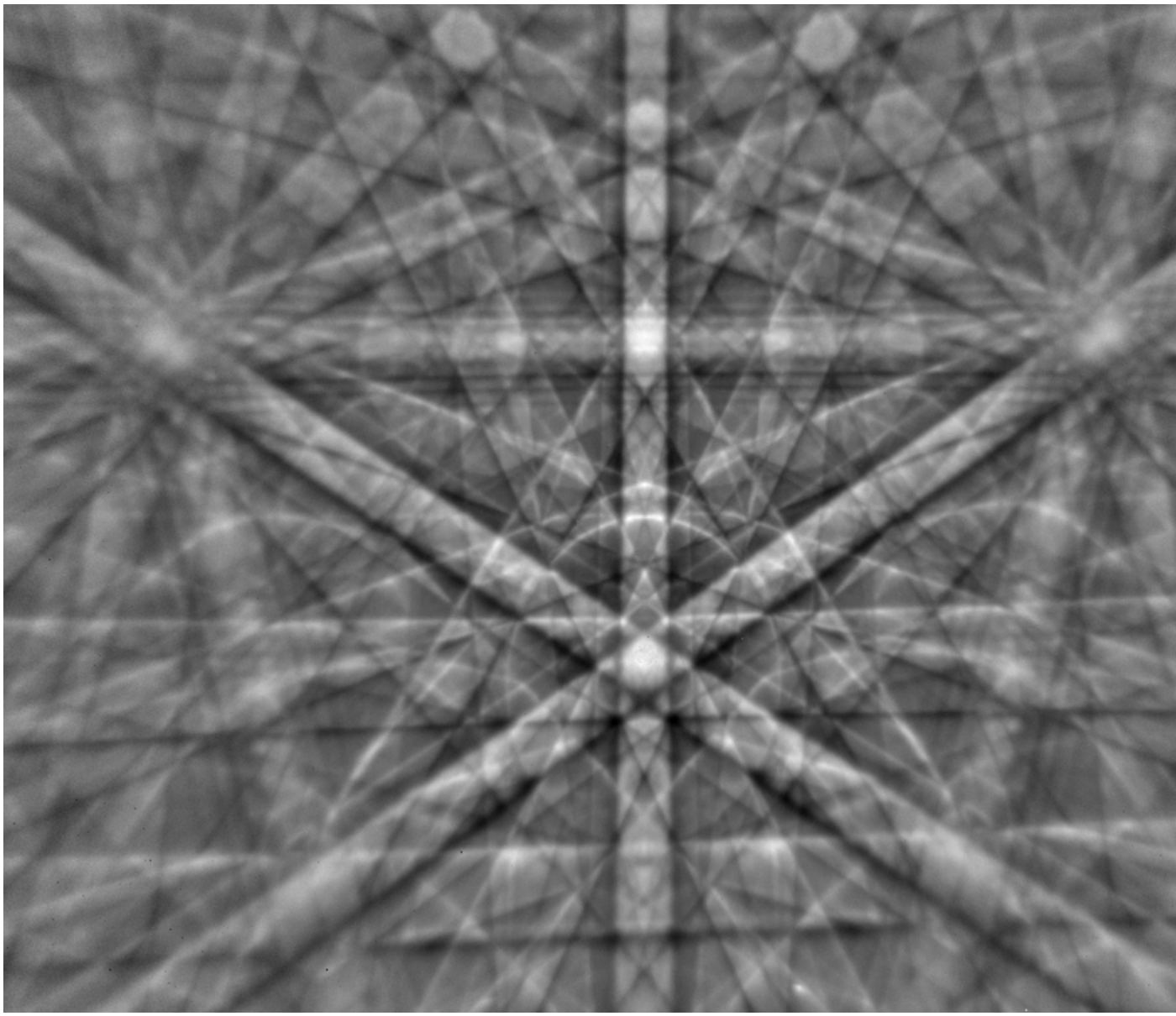
Nov 04, 2020

Contents

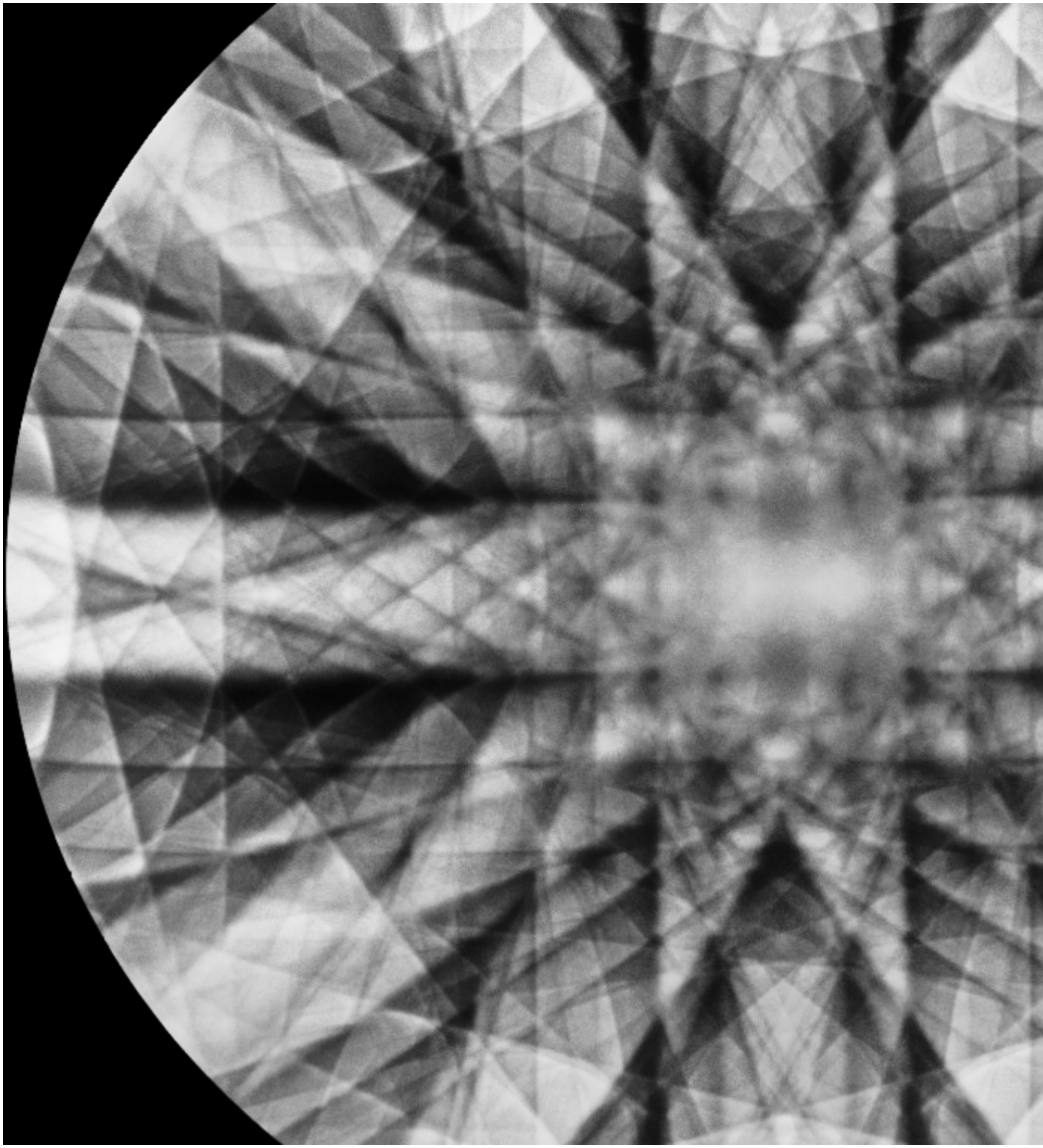
1 Getting Started

1.1 Kikuchi Diffraction Methods in the SEM

EBSD Pattern:



ECP:



Literature:

- L. Reimer *Scanning Electron Microscopy Physics of Image Formation and Microanalysis*[1], Springer Verlag, 1998

[1] https://doi.org/10.1007/978-3-540-38967-5

2 Data Acquisition

2.1 Tools for HDF5

Accessing CTF files with Python

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
```

Importing and Plotting EBSD Data from CTF

To access EBSD data in CTF files, we use the ebsdtools from Philippe Pinard at https://github.com/ppinard/ebsdtools

To read a CTF file, we need ebsdtools.hkl.tango.ctfFile, which we have updated for Python3 and saved as *ctfFile3.py*.

[2]:
```
from aloe.ext.ctfFile3 import Ctf
```

[3]:
```
filename="../data/ctf/GaN_polytype_mixing.ctf"
ctf = Ctf(filename)
mapdata = ctf.getPixelArray(key='bc', phase=('=', 1), bc=('>', 10))
print(np.array(mapdata))
```

```
[153 154 153 ..., 155 156 157]
```

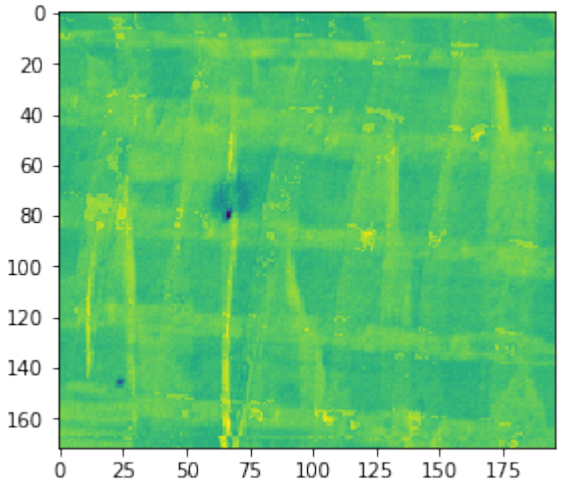
[4]:
```
print(ctf.getPhases())
```

```
{1: {'id': 1, 'cell_length_a': 3.189, 'cell_length_b': 3.189, 'cell_length_c': 5.185,
 ↪'cell_angle_alpha': 90.0, 'cell_angle_beta': 90.0, 'cell_angle_gamma': 120.0, 'chemical_
 ↪name_mineral': 'Ga N', 'symmetry_Int_Tables_number': 186, 'publ_author_name': 'Mod. Phys.
 ↪ Lett. B [MPLBET], (1999), vol. 13, pages 285-290'}, 2: {'id': 2, 'cell_length_a': 4.364,
 ↪ 'cell_length_b': 4.364, 'cell_length_c': 4.364, 'cell_angle_alpha': 90.0, 'cell_angle_
 ↪beta': 90.0, 'cell_angle_gamma': 90.0, 'chemical_name_mineral': 'Ga N', 'symmetry_Int_
 ↪Tables_number': 216, 'publ_author_name': 'ICSD[67781]'}}
```

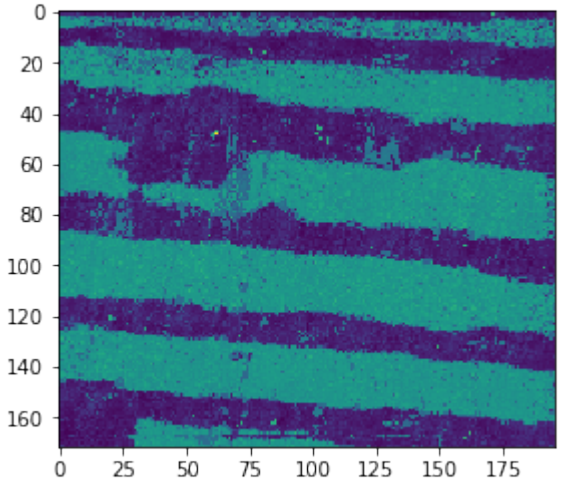
[5]:
```
ctf_map_width=ctf.getWidth()
ctf_map_height=ctf.getHeight()
print(ctf_map_height, ctf_map_width)
```

```
172 196
```

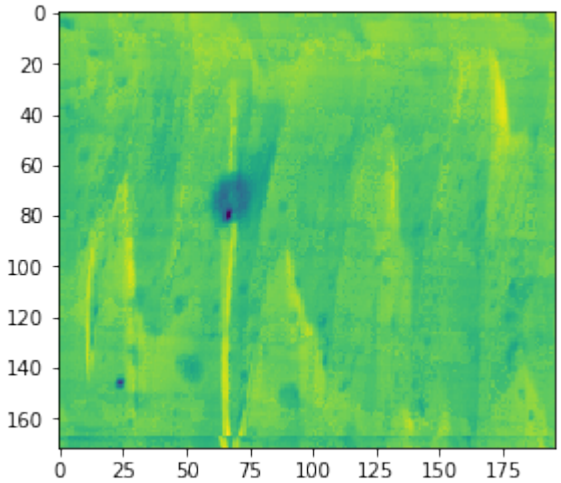
[6]:
```
bc=np.array(ctf.getPixelArray(key='bc', phase=('=', 1), bc=('>', 10)))
plt.imshow(bc.reshape(ctf_map_height, ctf_map_width))
plt.show()
```



[7]:
```
mad=np.array(ctf.getPixelArray(key='mad'))
plt.imshow(mad.reshape(ctf_map_height, ctf_map_width))
plt.show()
```



[8]:
```
bs=np.array(ctf.getPixelArray(key='bs'))
plt.imshow(bs.reshape(ctf_map_height, ctf_map_width))
plt.show()
```



HDF5: Convert *Directory* of Images

In this notebook, we show how to load and save a set of images into a HDF5 file for subsequent processing.

Initialization Code

[7]:
```
%matplotlib inline
import sys
from PIL import Image

import matplotlib.pyplot as plt
import numpy as np
import h5py

from aloe.image.downsample import downsample
from aloe.plots import plot_image
```

Loading the Images

The patterns are assumed to be in a *directory* as separate files, from which we load them and save them into a HDF5 file.

[13]:
```
example_dir = "../../../xcdskd_example_data/GaN_Dislocations_1/Map_Patterns/"
image_dir = example_dir +"/Images/"
image_ext = ".tiff"
ctf_filename = example_dir+"Map Data 1.ctf"

#static background image
background_filename=example_dir+"StaticBackground.tiff"

output_dir = example_dir
HDF5FileName = output_dir +"GaN_Dislocations_1.hdf5"
```

No changes of filenames should be needed from here on.

[14]:
```
# get the infos directly form the related CTF file
from ctfFile3 import Ctf
ctf = Ctf(ctf_filename)
```

[15]:
```
ctf_map_width=ctf.getWidth()
ctf_map_height=ctf.getHeight()

ctf_size = ctf.getSize()
print(ctf_size)
print(ctf_map_height, ctf_map_width, ctf_map_height*ctf_map_width)

ctf_xstep = ctf.getXStep()
ctf_ystep = ctf.getYStep()
print(ctf_xstep, ctf_ystep)


ctf_xcells = ctf.getXCells()
ctf_ycells = ctf.getYCells()
print(ctf_xcells, ctf_ycells) # -> x is width and y is height
```

```
2600
50 52 2600
```

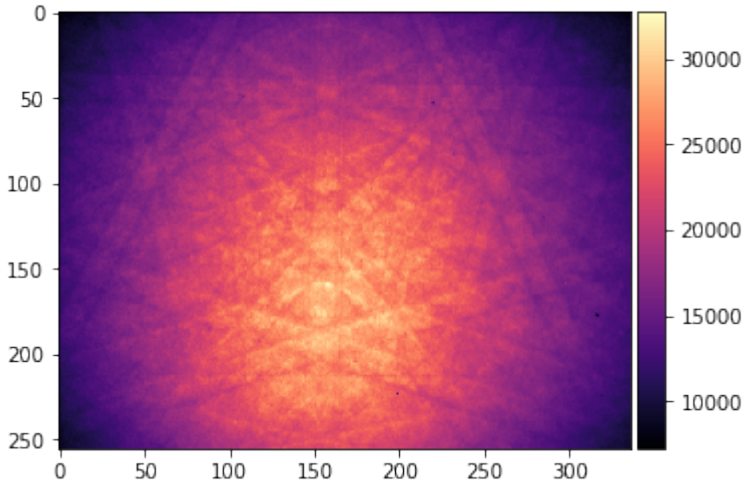
(continues on next page)

(continued from previous page)

```
0.05 0.05
52 50
```

[16]:
```
# get image dimensions from first image
img_filename=image_dir+'0_0.tiff'
img = Image.open(img_filename)
print('pattern dims, mode, bytes:', img.size, img.mode, len(img.getdata()))
pattern_width, pattern_height = img.size
print("pattern width, pattern height: ", pattern_width, pattern_height)

img_arr=np.array(img)

plot_image(img_arr, cmap='magma')
```

```
pattern dims, mode, bytes: (336, 256) I;16 86016
pattern width, pattern height:  336 256
```



Enter the map parameters from CTF:

[17]:
```
# map parameters
img_width  = pattern_width
img_height = pattern_height
map_width  = ctf_xcells
map_height = ctf_ycells
xstepmu=ctf_xstep
ystepmu=ctf_ystep
NImages=ctf_size
```

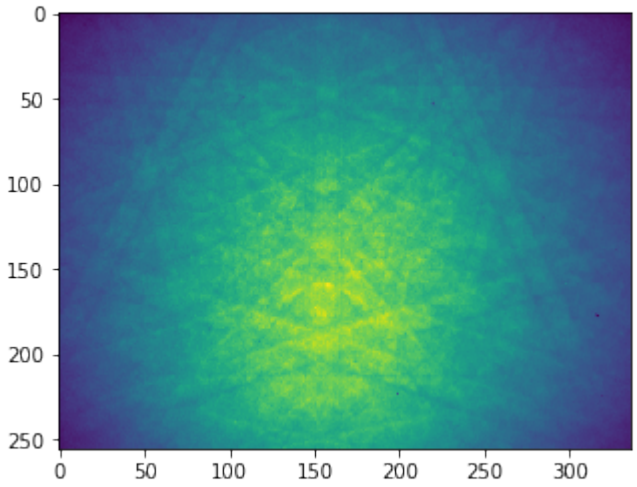
Test downsampling of the data to reduce file size if possible:

[18]:
```
# test downsampling
plt.figure()
plt.imshow(img_arr)
plt.show()
print(img_arr)

# bin by a factor of 2
binning=1

# sum the pixel intensities to retain INTEGER values in binned array (save memory in hdf5)
# we need to be sure not to exceed the 16bit integer range in the WHOLE MAP if using np.
↪sum!
```

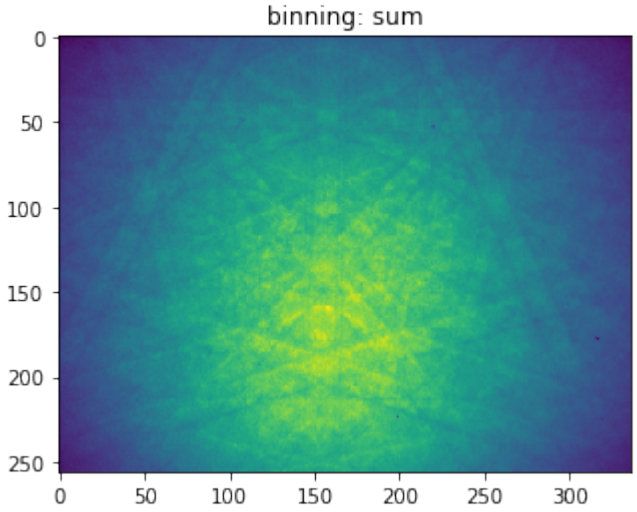
(continues on next page)

(continued from previous page)

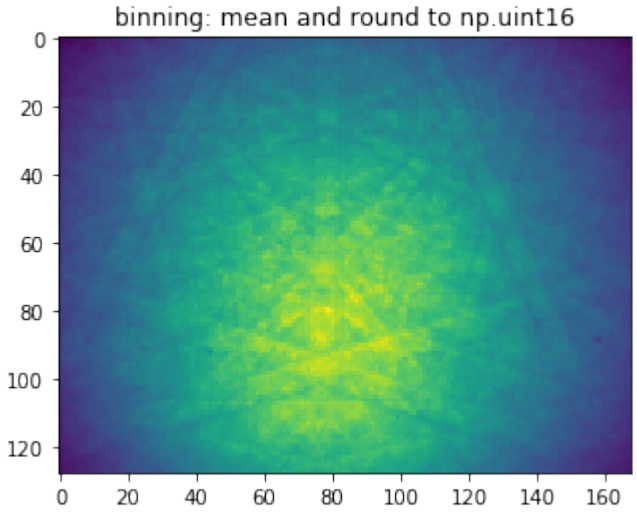
```
binning_estimator=np.sum
img_arr_binned=downsample(img_arr, binning, estimator=binning_estimator)
plt.figure()
plt.imshow(img_arr_binned)
plt.title('binning: sum')
plt.show()
print(img_arr_binned)

# estimator=np.nanmean will result in float values (32bit)
# to save memory, we can use rounding and conversion to 16bit integers
binning_estimator=np.nanmean
img_arr_binned=np.rint(downsample(img_arr,2, estimator=binning_estimator)).astype(np.
↪uint16)
plt.figure()
plt.imshow(img_arr_binned)
plt.title('binning: mean and round to np.uint16')
plt.show()
print(img_arr_binned)
```




```
[[7520 7976 7928 ..., 8128 8232 8432]
 [7704 7912 8184 ..., 8216 8272 8184]
 [8136 7880 8080 ..., 8216 8240 8104]
 ...,
 [8568 9088 8896 ..., 9680 9480 9552]
 [8768 9016 9000 ..., 9600 9792 9208]
 [8696 8832 8976 ..., 9504 9352 9320]]
```




```
[[7520 7976 7928 ..., 8128 8232 8432]
 [7704 7912 8184 ..., 8216 8272 8184]
 [8136 7880 8080 ..., 8216 8240 8104]
 ...,
 [8568 9088 8896 ..., 9680 9480 9552]
 [8768 9016 9000 ..., 9600 9792 9208]
 [8696 8832 8976 ..., 9504 9352 9320]]
```




```
[[ 7778  8048  8108 ...,  8486  8240  8280]
 [ 8064  8184  8380 ...,  8608  8376  8166]
 [ 8156  8320  8828 ...,  8858  8796  8438]
 ...,
 [ 9080  9234  9536 ..., 10136  9726  9762]
 [ 8870  9050  9298 ...,  9818  9704  9514]
 [ 8828  9064  9362 ..., 10028  9668  9418]]
```

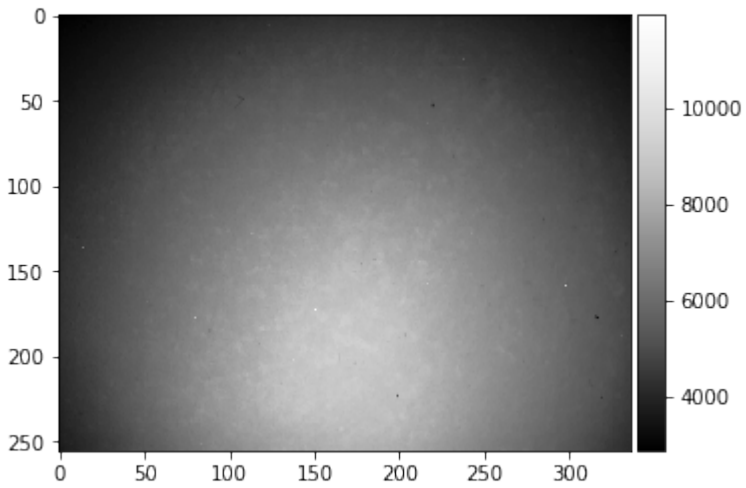
[20]:
```
# flatfielding background extra file
print('Loading background from file: ', background_filename)
img = Image.open(background_filename) #.convert('LA')
print(img.size, img.mode, len(img.getdata()))
bg_binning=1

bg_img_arr =np.rint(downsample(np.array(img), bg_binning,
                                    estimator=np.nanmean)).astype(np.uint16)
```

(continues on next page)

(continued from previous page)

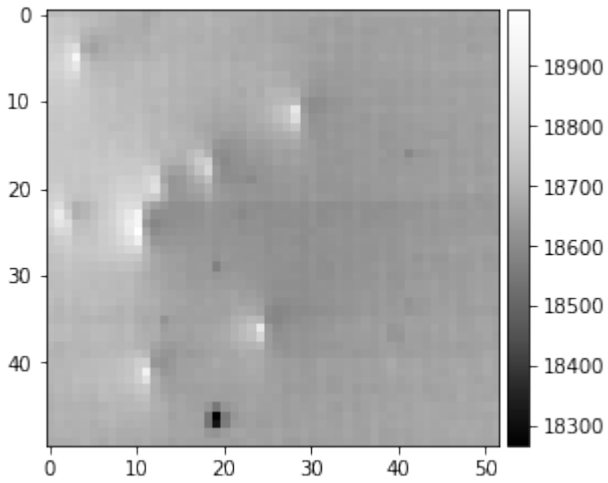
```
print(bg_img_arr)
print(bg_img_arr.shape)
plot_image(bg_img_arr)
```

```
Loading background from file:  ../../../xcdskd_example_data/GaN_Dislocations_1/Map_
↪Patterns/StaticBackground.tiff
(336, 256) I;16 86016
[[2884 2920 2920 ..., 2994 2995 2992]
 [2910 2944 2954 ..., 3005 3009 2992]
 [2931 2955 2954 ..., 3050 3024 3036]
 ...,
 [3725 3774 3783 ..., 4260 4243 4211]
 [3714 3761 3722 ..., 4297 4266 4221]
 [3704 3707 3737 ..., 4279 4220 4204]]
(256, 336)
```



[23]:
```
from skimage.io import imread
def load_row_col_tiff(row, col, prefix=''):
    """
    reads an image file row_col.tiff (e.g. saved by the AZTEC EBSD software)
    """
    img_filename = prefix + str(row) + '_' + str(col) + '.tiff'
    img_arr  = imread(img_filename)
    return img_arr
```

[25]:
```
# make total bse map
bse_map = np.zeros((map_height, map_width))
for iw in range(map_width):
    for ih in range(map_height):
        bse_map[ih, iw] = np.nanmean(load_row_col_tiff(ih, iw, prefix=image_dir))

plot_image(bse_map)
```



[26]:
```
# export to hdf5
binning=1
img_height = pattern_height // binning
img_width = pattern_width  // binning

binning_estimator=np.nanmean

iStart=0
iEnd=NImages

DataGroup='Scan/EBSD/Data/'
HeaderGroup='Scan/EBSD/Header/'

# create new HDF5 File
f5=h5py.File(HDF5FileName,"w") # overwrite if exists
f5["Manufacturer"]="xcdskd"
f5["Version"]="0.01"

# h5ebsd HEADER INFO Data

# create empty datasets for h5ebsd format
# (empty datasets take no space)
f5.create_dataset(DataGroup+"X BEAM", (NImages,) ,dtype=np.int32,compression='gzip')
f5.create_dataset(DataGroup+"Y BEAM", (NImages,) ,dtype=np.int32,compression='gzip')
f5.create_dataset(DataGroup+"PCX" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"PCY" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"DD" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"phi1", (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"PHI" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"phi2", (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"X Position", (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"Y Position", (NImages,) ,dtype=np.float32,compression='gzip')

f5[DataGroup+"bse_map"] = bse_map

f5[HeaderGroup+"NCOLS"] = map_width
f5[HeaderGroup+"NROWS"] = map_height
f5[HeaderGroup+"PatternHeight"] = img_height
f5[HeaderGroup+"PatternWidth"] = img_width
f5[HeaderGroup+"X Resolution"] = xstepmu  # microns
f5[HeaderGroup+"Y Resolution"] = ystepmu  # microns
```

(continues on next page)

(continued from previous page)

```
# StaticBackground
#f5.create_dataset(DataGroup+"StaticBackground", (img_height,img_width), dtype=np.uint16)
f5[DataGroup+"StaticBackground"]=np.rint(downsample(bg_img_arr, binning,
                                    estimator=binning_estimator)).astype(np.uint16)

print(f5[DataGroup+"StaticBackground"].shape)

# create empty data set for Patterns, recompress (max=9) using hdf5-gzip filter
dset_patterns=f5.create_dataset(DataGroup+'RawPatterns',
                                (NImages,img_height,img_width),
                                dtype=np.uint16,
                                chunks=(1,img_height,img_width),
                                compression='gzip',compression_opts=9)

print(dset_patterns.shape)
print(map_width, map_height)
i=0
try:
    for iw in range(map_width):
        for ih in range(map_height):
            img_filename = str(ih)+'_'+str(iw)+'.tiff'
            img = Image.open(image_dir+img_filename)

            img_arr = np.rint(downsample(np.array(img), binning,
                                estimator=binning_estimator)).astype(np.uint16)
            dset_patterns[i]=img_arr
            # save scan indices = map position index of image i
            f5[DataGroup+"X BEAM"][i]=iw
            f5[DataGroup+"Y BEAM"][i]=ih

            # some dummy defaults for the projection center
            f5[DataGroup+"DD"][i]  = 1.0
            f5[DataGroup+"PCX"][i] = 0.5
            f5[DataGroup+"PCY"][i] = 0.5
            i+=1

            # live update progress info
            if (i % 25 == 0):
                progress=100.0*(i+1)/NImages
                sys.stdout.write("\rtotal patterns: %5i current:%5i progress: %4.2f%%"
                                 %(NImages,i,progress) )
                sys.stdout.flush()


    # find map indices and save index maps
    npatterns=dset_patterns.shape[0]
    XIndex = f5[DataGroup+"X BEAM"]
    YIndex = f5[DataGroup+"Y BEAM"]
    print('creating index map: ', npatterns)
    map_index=[]
    index_lookup = -1*np.ones((map_height,map_width), dtype=np.int)
    for index in range(npatterns):
        bx=XIndex[index]
        by=YIndex[index]
        map_index.append([index, bx, by])
        index_lookup[by, bx]=index
    map_index=np.array(map_index)
```

(continues on next page)

(continued from previous page)

```
    print(map_index)
    #np.savetxt('./index_lookup.dat', index_lookup, fmt=' %9i ')
    f5[HeaderGroup+"maplist"] = map_index
    f5[HeaderGroup+"indexmap"] = index_lookup

finally:
    f5.close()
```

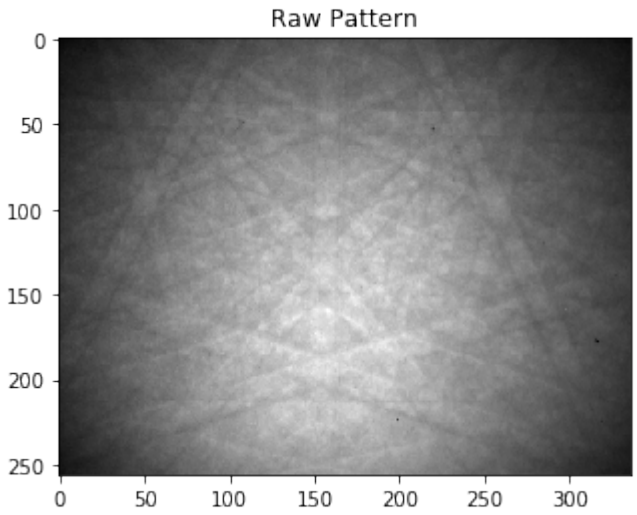
```
(256, 336)
(2600, 256, 336)
52 50
total patterns:  2600 current: 2600 progress: 100.04%creating index map:  2600
[[   0    0    0]
 [   1    0    1]
 [   2    0    2]
 ...,
 [2597   51   47]
 [2598   51   48]
 [2599   51   49]]
```

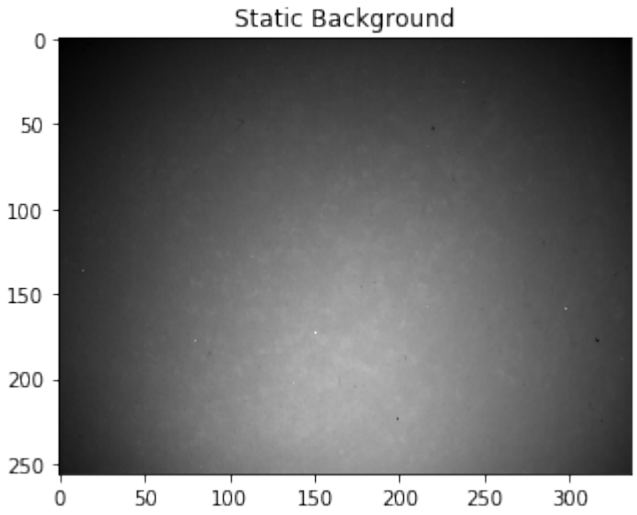
[13]:
```
f5.close()
```

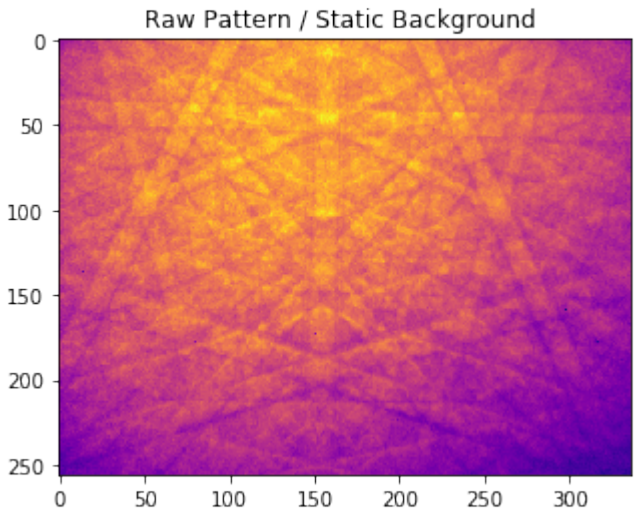
Test: Load the Pattern Data from the HDF5

[27]:
```
# which pattern to load
pattern_number = 1000

# open file for reading
f5=h5py.File(HDF5FileName, 'r')

img=f5[DataGroup+'RawPatterns'][pattern_number]
print(np.max(img))
print(img.shape)

bg=f5[DataGroup+'StaticBackground']
print(bg.shape)

ff=img/bg

plt.figure()
plt.imshow(img, cmap='gray')
plt.title('Raw Pattern')
plt.show()

plt.figure()
plt.imshow(bg, cmap='gray')
plt.title('Static Background')
plt.show()


plt.figure()
plt.imshow(img/bg, cmap='plasma')
plt.title('Raw Pattern / Static Background ')
plt.show()

f5.close()
```

```
32152
(256, 336)
(256, 336)
```







[ ]:

Conversion of Image Zip to HDF5

Initialization Code

[2]:
```
%matplotlib inline
import sys
from zipfile import ZipFile
from PIL import Image # if not present:  pip install pillow

import matplotlib.pyplot as plt
import numpy as np
import h5py
```

[3]:
```
def downsample(myarr,factor,estimator=np.nanmean):
    """
    Downsample a 2D array by averaging over *factor* pixels in each axis.
    Crops upper edge if the shape is not a multiple of factor.

    This code is pure np and should be fast.

    keywords:
        estimator - default to mean.  You can downsample by summing or
            something else if you want a different estimator
            (e.g., downsampling error: you want to sum & divide by sqrt(n))
    """
    ys,xs = myarr.shape
    crarr = myarr[:ys-(ys % int(factor)),:xs-(xs % int(factor))]
    dsarr = estimator( np.concatenate([[crarr[i::factor,j::factor]
        for i in range(factor)]
        for j in range(factor)]), axis=0)
    return dsarr
```

Loading the Images

The patterns are assumed to be in a zip file, from which we extract them and save them into a HDF5 file.

[4]:
```
# filename of the zip file, relative to current directory
zip_filename = './GaN_Polytypes/polytypeImages.zip'

zip_image_dir = '' # in this zip, the images were saved at the root level
zip_image_ext = '.tiff'

zip_background_name = None # none if background is extra file outside of zip
background_filename='./GaN_Polytypes/StaticBackground.tiff'
```

[ ]:
```
# test to see how importing works...
with ZipFile(zip_filename) as archive:
    for idx, entry in enumerate(archive.namelist()):
        if idx<5:
            if (zip_image_dir in entry) and (zip_image_ext in entry):
                # now we have an image filename in entry...
                img_name=entry.replace(zip_image_dir,'')
                img_name=img_name.replace(zip_image_ext,'')
                numbers=img_name.split('_')
                img_row=numbers[0]
                img_col=numbers[1]
```

(continues on next page)

(continued from previous page)

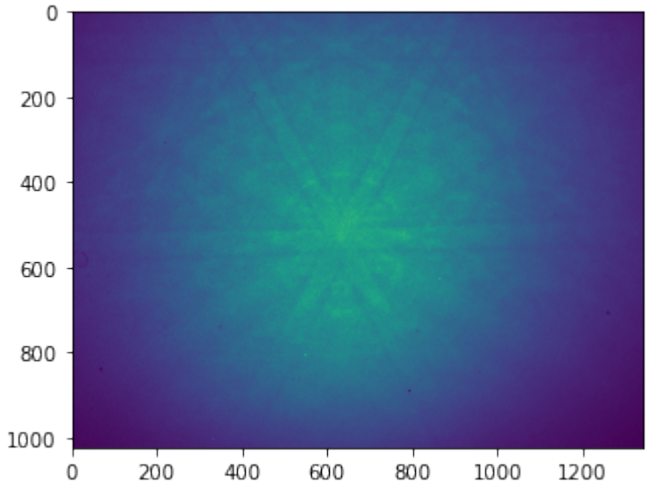
```
                print('pattern name in zip:', entry)
                print('extracted row, column numbers in map: ', img_row, img_col)
                with archive.open(entry) as file:
                    img = Image.open(file) #.convert('LA')
                    print('pattern dims, mode, bytes:', img.size, img.mode, len(img.
↪getdata()))
                    img_arr =np.array(img)
                    print(img_arr)
                    print()
                    plt.figure()
                    plt.imshow(img_arr)
```

Enter the map parameters from CTF etc, change manually for now:

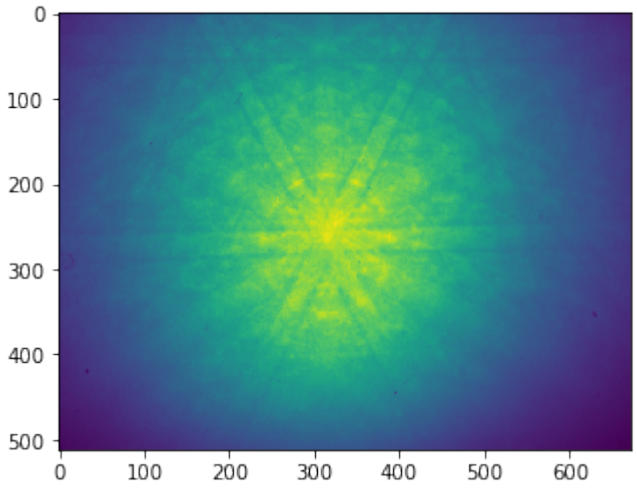
[10]:
```
# map parameters
img_width  = img.size[0] # last img that is loaded in above cell
img_height = img.size[1]
map_width  = 196 # CTF: XCells 196
map_height = 172 # CTF: YCells 172
xstepmu=0.15 # CTF: XStep 0.1500
ystepmu=0.15 # CTF: YStep 0.1500
NImages=map_width*map_height
```

Test downsampling of the data to reduce file size if possible:

[6]:
```
# test downsampling
plt.figure()
plt.imshow(img_arr)
plt.show()
print(img_arr)

# bin by a factor of 2
binning=2

# sum the pixel intensities to retain INTEGER values in binned array (save memory in hdf5)
# we need to be sure not to exceed the 16bit integer range in the WHOLE MAP if using np.
↪sum!
binning_estimator=np.sum
img_arr_binned=downsample(img_arr,2, estimator=binning_estimator)
plt.figure()
plt.imshow(img_arr_binned)
plt.title('binning: sum')
plt.show()
print(img_arr_binned)

# estimator=np.nanmean will result in float values (32bit)
# to save memory, we can use rounding and conversion to 16bit integers
binning_estimator=np.nanmean
img_arr_binned=np.rint(downsample(img_arr,2, estimator=binning_estimator)).astype(np.
↪uint16)
plt.figure()
plt.imshow(img_arr_binned)
plt.title('binning: mean and round to np.uint16')
plt.show()
print(img_arr_binned)
```




```
[[4336 4264 4128 ..., 4200 4064 4312]
 [4328 4296 4112 ..., 4120 4296 4320]
 [4280 4248 4296 ..., 4064 4224 4112]
 ...,
 [2824 2976 3000 ..., 2584 2584 2512]
 [2880 2904 2928 ..., 2592 2544 2608]
 [2800 2840 2936 ..., 2584 2488 2528]]
```




```
[[17224 16664 17240 ..., 16384 16600 16992]
 [17040 17008 17104 ..., 16576 16536 16688]
 [17152 16808 17064 ..., 16840 16760 16936]
 ...,
 [11624 11632 11552 ..., 10336 10248 10184]
 [11680 11624 11704 ..., 10248 10240 10224]
 [11424 11600 11680 ..., 10120 10272 10168]]
```

[7]:
```
# if flatfielding background in zip
if not (zip_background_name is None):
    with ZipFile(zip_filename) as archive:
        with archive.open(zip_background_name) as file:
            img = Image.open(file) #.convert('LA')
            print(img.size, img.mode, len(img.getdata()))
            bg_img_arr =np.array(img)
            print(bg_img_arr)
            plt.figure()
```

(continues on next page)

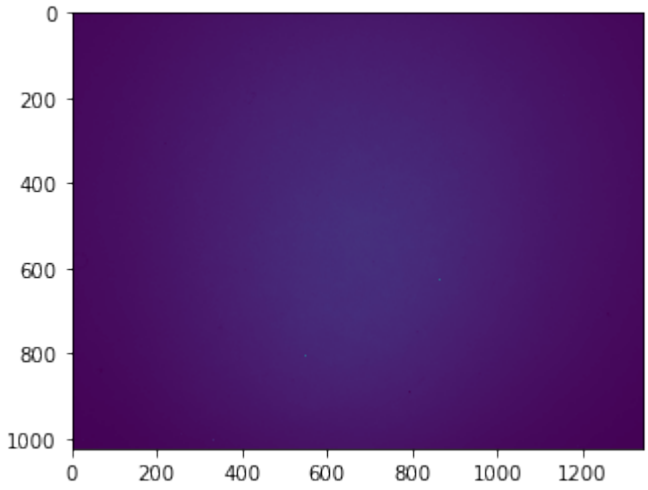
(continued from previous page)

```
            plt.imshow(bg_img_arr)
```

[8]:
```
# flatfielding background extra file
print('Loading background from file: ', background_filename)
img = Image.open(background_filename) #.convert('LA')
print(img.size, img.mode, len(img.getdata()))
bg_img_arr =np.array(img)
print(bg_img_arr)
plt.figure()
plt.imshow(bg_img_arr)
```

```
Loading background from file:  ./GaN_Polytypes/StaticBackground.tiff
(1344, 1024) I;16 1376256
[[1870 1873 1872 ..., 1887 1882 1898]
 [1861 1875 1876 ..., 1899 1902 1886]
 [1881 1880 1876 ..., 1906 1891 1895]
 ...,
 [1654 1665 1671 ..., 1647 1654 1642]
 [1658 1673 1675 ..., 1650 1647 1651]
 [1652 1668 1676 ..., 1653 1660 1632]]
```

[8]:
```
<matplotlib.image.AxesImage at 0xb58d208>
```



[9]:
```
# export to hdf5
binning=2
img_height = img_height // binning
img_width = img_width  // binning
binning_estimator=np.nanmean

iStart=0
iEnd=NImages

HDF5FileName='GaN_poly2.hdf5'

DataGroup='Scan 0/EBSD/Data/'
HeaderGroup='Scan 0/EBSD/Header/'

# create new HDF5 File
f5=h5py.File(HDF5FileName,"w") # overwrite if exists
f5["Manufacturer"]="pattern_zip_to_hdf5"
```

(continues on next page)

(continued from previous page)

```
f5["Version"]="0.1"



# h5ebsd HEADER INFO Data

# create empty datasets for h5ebsd format
# (empty datasets take no space)
f5.create_dataset(DataGroup+"BEAM X", (NImages,) ,dtype=np.int32,compression='gzip')
f5.create_dataset(DataGroup+"BEAM Y", (NImages,) ,dtype=np.int32,compression='gzip')
f5.create_dataset(DataGroup+"PCX" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"PCY" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"DD" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"phi1", (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"PHI" , (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"phi2", (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"X Position", (NImages,) ,dtype=np.float32,compression='gzip')
f5.create_dataset(DataGroup+"Y Position", (NImages,) ,dtype=np.float32,compression='gzip')

f5[HeaderGroup+"NCOLS"]=map_width
f5[HeaderGroup+"NROWS"]=map_height
f5[HeaderGroup+"PatternHeight"]=img_height
f5[HeaderGroup+"PatternWidth"]=img_width
f5[HeaderGroup+"X Resolution"]=xstepmu  # microns
f5[HeaderGroup+"Y Resolution"]=ystepmu  # microns

# staticBackround
#f5.create_dataset(DataGroup+"StaticBackground", (img_height,img_width), dtype=np.uint16)
f5[DataGroup+"StaticBackground"]=np.rint(downsample(bg_img_arr, binning,
                                  estimator=binning_estimator)).astype(np.uint16)

# create empty data set for Patterns, recompress (max=9) using hdf5-gzip filter
dset_patterns=f5.create_dataset(DataGroup+'RawPatterns',
                                (NImages,img_height,img_width),
                                dtype=np.uint16,
                                chunks=(1,img_height,img_width),
                                compression='gzip',compression_opts=9)

i=0
with ZipFile(zip_filename) as archive:
    for idx, entry in enumerate(archive.namelist()):
        if (zip_image_dir in entry) and (zip_image_ext in entry):
            #if i>100:
            #    break
            # now we have an image filename in entry...
            img_name=entry.replace(zip_image_dir,'')
            img_name=img_name.replace(zip_image_ext,'')
            numbers=img_name.split('_')
            img_row=int(numbers[0])
            img_col=int(numbers[1])
            with archive.open(entry) as file:
                img = Image.open(file)
                img_arr = np.rint(downsample(np.array(img), binning,
                            estimator=binning_estimator)).astype(np.uint16)
                dset_patterns[i]=img_arr
            # save scan indices = map position index of image i
            f5[DataGroup+"BEAM X"][i]=img_col    #i % MapWidth
            f5[DataGroup+"BEAM Y"][i]=img_row    #i // MapWidth
```

(continues on next page)

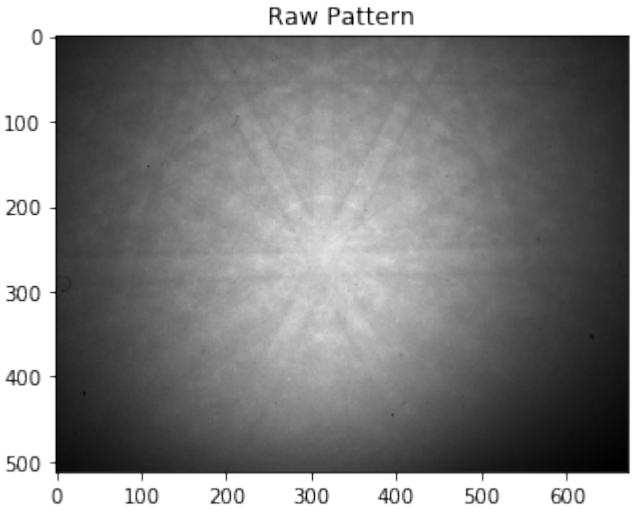
(continued from previous page)

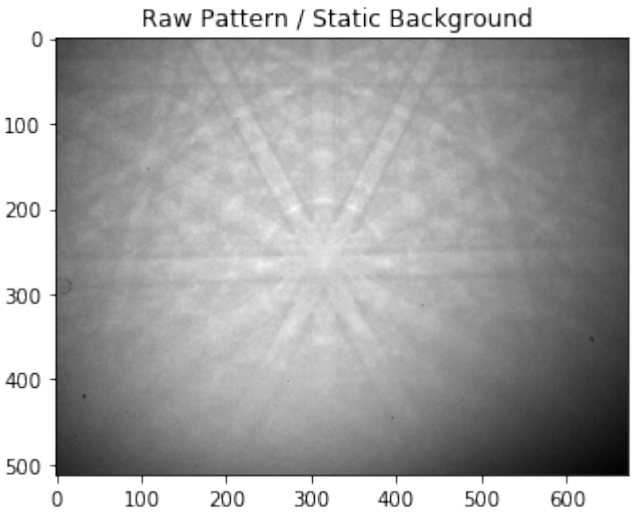
```
            # some dummy defaults for the projection center
            f5[DataGroup+"DD"][i]  = 1.0
            f5[DataGroup+"PCX"][i] = 0.5
            f5[DataGroup+"PCY"][i] = 0.5
            i+=1

        # live update progress info
        progress=100.0*(i+1)/NImages
        sys.stdout.write("\rtotal patterns: %5i current:%5i progress: %4.2f%%"
                         %(NImages,i,progress) )
        sys.stdout.flush()

f5.close()
```

```
total patterns: 33712 current:33712 progress: 100.00%
```

Example: Loading the Pattern Data from the HDF5

[11]:
```
# which pattern to load
pattern_number = 1000

# open file for reading
f5=h5py.File(HDF5FileName, 'r')

img=f5[DataGroup+'RawPatterns'][pattern_number]
bg=f5[DataGroup+'StaticBackground']

ff=img/bg

plt.figure()
plt.imshow(img, cmap='gray')
plt.title('Raw Pattern')
plt.show()

plt.figure()
plt.imshow(img/bg, cmap='gray')
plt.title('Raw Pattern / Static Background ')
plt.show()

f5.close()
```





Filtering Raw Patterns in HDF5

Process raw EBSD images in HDF5 file and save the Kikuchi signal.

[22]:
```
import os
data_dir = "../../../xcdskd_example_data/GaN_Dislocations_1"
h5FileNameFull=os.path.abspath(data_dir+"/hdf5/GaN_Dislocations_1.hdf5")
#print(h5FileNameFull)
nbin=1   # binning factor for pattern processing
```

[23]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[24]:
```
import numpy as np
import skimage.io
import time
import sys
```

(continues on next page)

(continued from previous page)

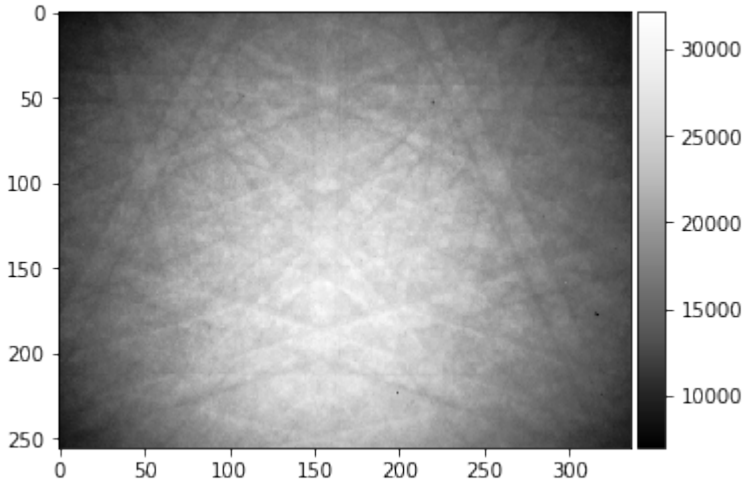
```
import h5py
from shutil import copyfile

from aloe.image import kikufilter

from aloe.plots import plot_image
```

[25]:
```
# close HDF5 file if still open
if 'f' in locals():
    f.close()

# split off the extension
h5FileName, h5FileExt = os.path.splitext(h5FileNameFull)
h5FilePath, h5File =os.path.split(h5FileNameFull)


# copy to new file
src = h5FileName+h5FileExt
dst = h5FileName+'_processed'+h5FileExt
copyfile(src, dst)
# use new filename from here to work on
h5FileName = h5FileName+'_processed'



f=h5py.File(h5FileName+h5FileExt, "a") # append mode
#f=h5py.File(h5FileName+h5FileExt, "r")
#print('HDF5 full file name: '+h5FileNameFull)
#print('HDF5 File: '+h5FileName+h5FileExt)
#print('HDF5 Path: '+h5FilePath)
```

[26]:
```
DataGroup="/Scan/EBSD/Data/"
HeaderGroup="/Scan/EBSD/Header/"
Patterns = f[DataGroup+"RawPatterns"]
XIndex =   f[DataGroup+"X BEAM"]
YIndex =   f[DataGroup+"Y BEAM"]
MapWidth = f[HeaderGroup+"NCOLS"].value
MapHeight= f[HeaderGroup+"NROWS"].value
PatternHeight=f[HeaderGroup+"PatternHeight"].value
PatternWidth =f[HeaderGroup+"PatternWidth"].value
print('Pattern Height: ', PatternHeight)
print('Pattern Width : ', PatternWidth)
PatternAspect=float(PatternWidth)/float(PatternHeight)
print('Pattern Aspect: '+str(PatternAspect))
print('Map Height: ', MapHeight)
print('Map Width : ', MapWidth)
```

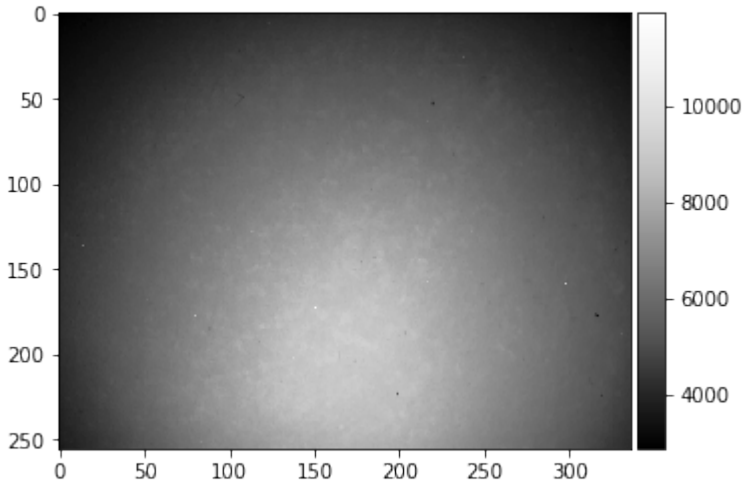
```
Pattern Height:  256
Pattern Width :  336
Pattern Aspect: 1.3125
Map Height:  50
Map Width :  52
```

[27]:
```
RawPattern=Patterns[1000,:,:]
plot_image(RawPattern)
#skimage.io.imsave('raw_pattern.tiff', RawPattern, plugin='tifffile')
```

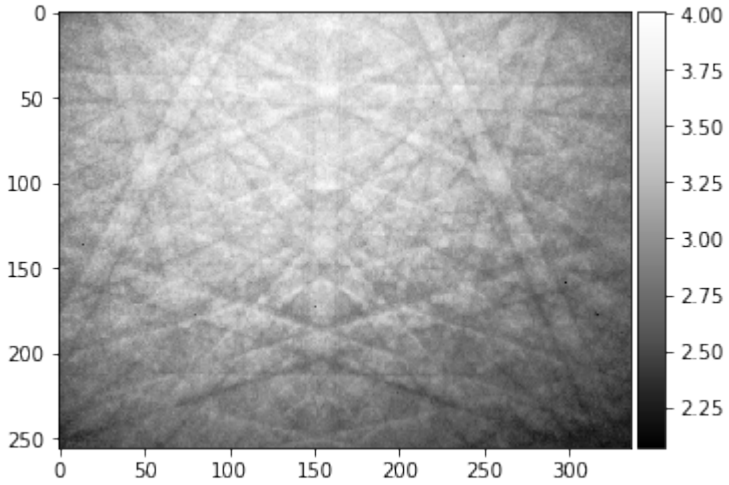


[28]:
```
# TRY OPTIONAL: make static background from average of all patterns in map
#StaticBackground=np.mean(Patterns[:], axis=0)
#f[HeaderGroup+"StaticBackground"] = StaticBackground
```

[29]:
```
StaticBackground=f[DataGroup+"StaticBackground"]
plot_image(StaticBackground)
#np.savetxt('StaticBackground.txt',StaticBackground )
```

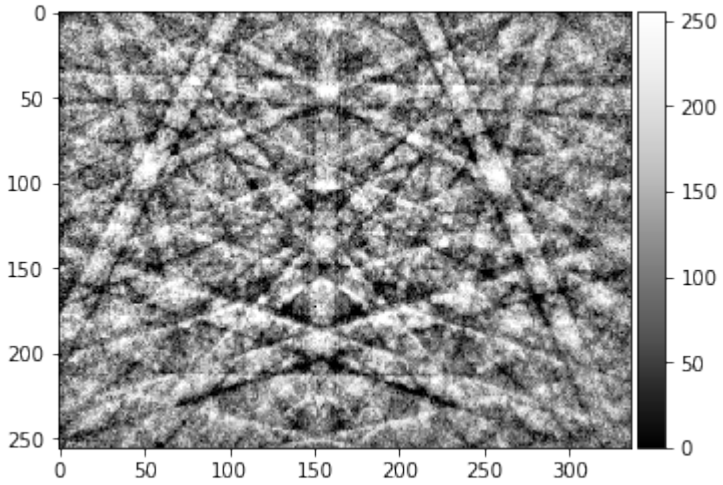


[30]:
```
PatternFlat=RawPattern/StaticBackground
plot_image(PatternFlat)
#from scipy import misc
#misc.imsave('Pattern_Flatfielded.png',RawPattern)
#scipy.misc.toimage(image_array, cmin=0.0, cmax=...).save('outfile.jpg')
```



We test the parameters for the processing of the raw patterns in the HDF5 file:

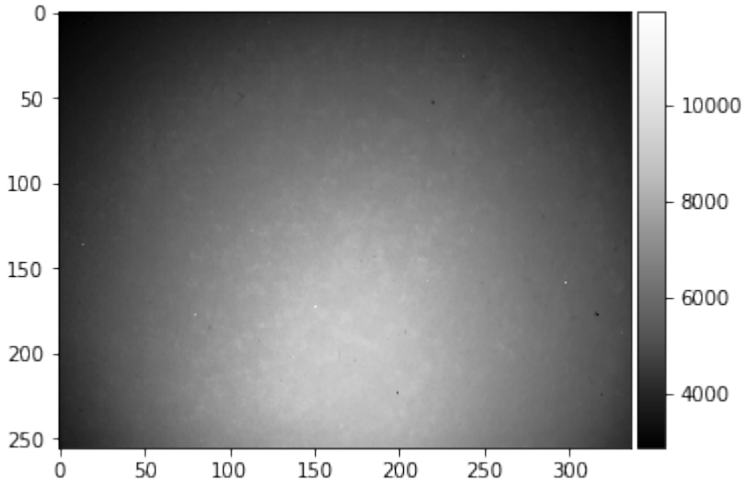
[31]:
```
kpattern=kikufilter.process_ebsp(raw_pattern=RawPattern,
                 static_background=StaticBackground, sigma=20,
                 clow=1.0, chigh=99.0, dtype=np.uint8, binning=nbin)
plot_image(kpattern)
print(kpattern.dtype)
```

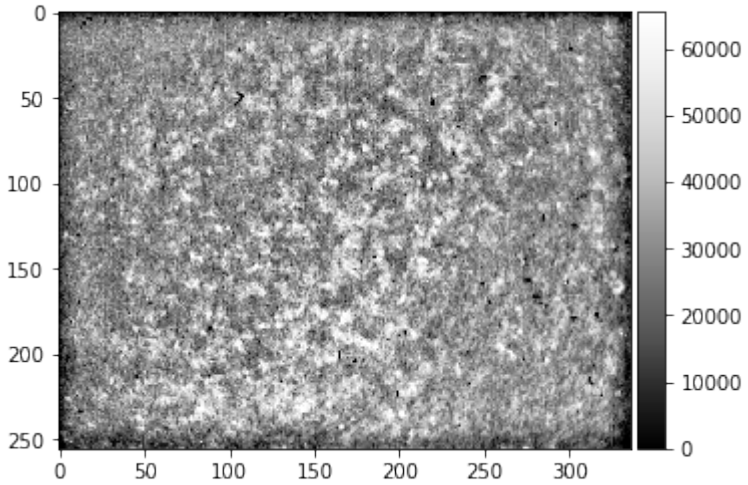



```
uint8
```

We also check the static background:

[32]:
```
bgpattern=kikufilter.process_ebsp(raw_pattern=StaticBackground,
                 static_background=None, sigma=15,
                 clow=0.5, chigh=99.5, dtype=np.uint16, binning=nbin)
plot_image(StaticBackground)
plot_image(bgpattern)
print(bgpattern.dtype)
```





uint16

Save Processed Patterns in HDF5

[33]:
```
nPatterns=MapWidth*MapHeight
print('Number of patterns: ',nPatterns)

# store *processed* patterns in HDF5
# create empty data set, compress (max=9) using hdf5-gzip filter
dataset_name=DataGroup+'Patterns'
if not (dataset_name in f):
    print('create: ', dataset_name)
    dset_processed=f.create_dataset(dataset_name,
                            (nPatterns,PatternHeight//nbin,PatternWidth//nbin),
                            dtype=np.uint8,
                            chunks=(1,PatternHeight//nbin,PatternWidth//nbin))
                            #compression='gzip',compression_opts=9)
else:
    # overwrite existing data
    print('data set already existing: overwriting ', dataset_name)
    dset_processed=f[dataset_name]

#nPatterns=5
```

(continues on next page)

(continued from previous page)

```
# this is very slow...
for p in range(nPatterns):
    img_raw = np.copy(Patterns[p])
    img = kikufilter.process_ebsp(raw_pattern=img_raw,
             static_background=StaticBackground, sigma=15,
             clow=1, chigh=99, dtype=np.uint8, binning=nbin)

    # update HDF5 data set
    dset_processed[p]=img

    # live update progress info
    if ( p % 10 == 0 ):
        progress=100.0*(p+1)/nPatterns
        sys.stdout.write("\rtotal patterns: %5i current:%5i progress: %4.2f%%"
                         % (nPatterns, p, progress) )
        sys.stdout.flush()

f.close()
```

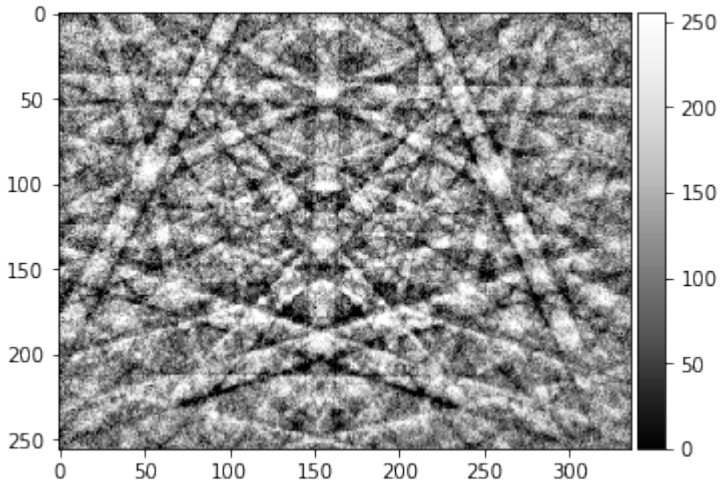
```
Number of patterns:  2600
create:  /Scan/EBSD/Data/Patterns
total patterns:  2600 current: 2590 progress: 99.65%
```

[34]:
```
f.close()
```

Test: Plotting the processed patterns

[35]:
```
f=h5py.File(h5FileName+h5FileExt, "r")
```

[36]:
```
kPatterns = f[DataGroup+"Patterns"]
plot_image(kPatterns[2500])
```



[37]:
```
indexmap = f[HeaderGroup+"indexmap"]


def get_npa(x, y, patterns, indexmap, nn=0):
    """ get neighbor pattern average
```

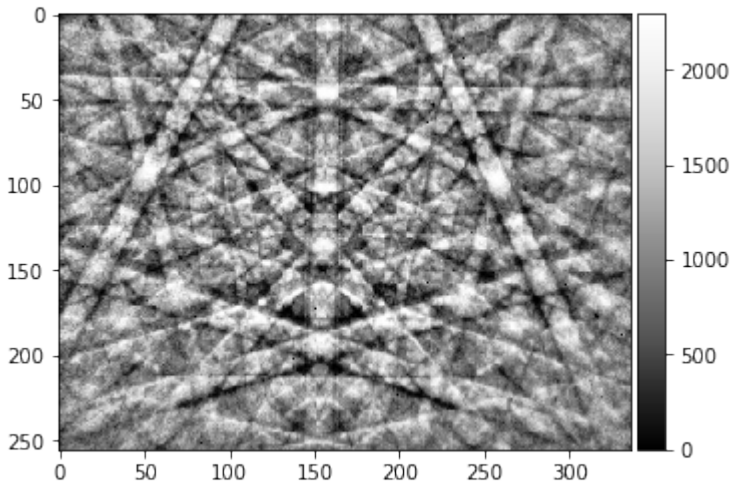
(continues on next page)

(continued from previous page)

```
    use -nn..+nn neighbors,
    i.e. at nn=1, there will be 8 neighbors = 9 pattern average

    patterns in assumed to be 1D array of 2D patterns
    index of pattern as function of x,y is in indexmap
    """
    ref_pattern = np.copy(patterns[indexmap[x,y]]).astype(np.int64)
    npa = np.zeros_like(ref_pattern, dtype=np.int64)
    for ix in range(x-nn,x+nn+1):
        for iy in range(y-nn, y+nn+1):
            print(ix, iy)
            if ((ix<0) or (ix>=patterns.shape[1]) or (iy<0) or (iy>=patterns.shape[0])):
                npa = npa + ref_pattern
            else:
                npa = npa + patterns[indexmap[ix,iy]]
    return npa
```

[38]:
```
img_npa = get_npa(24, 24, kPatterns, indexmap, nn=1)
plot_image(img_npa)
```

```
23 23
23 24
23 25
24 23
24 24
24 25
25 23
25 24
25 25
```



[ ]:
```
```

2.2 Experimental Details

ACQUIRE - PARSE - FILTER - MINE - REPRESENT - REFINE - INTERACT

Initialization code

[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

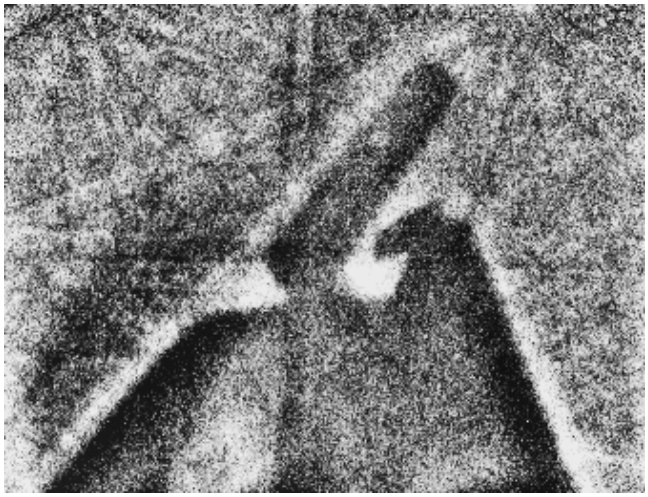
```
import numpy as np
from IPython.core.display import Image, display
```

Effects Gallery

Shadowing by Topography

Batman returns:

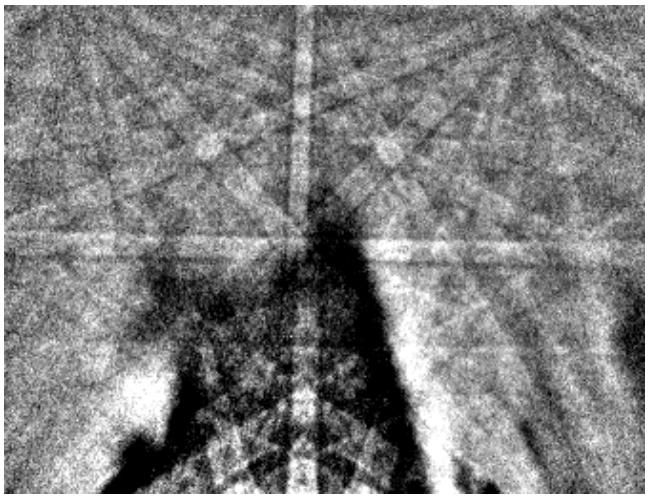
[3]:
```
display(Image(filename='./effects_gallery/batman_img000129.png', width=320))
```



Transmission Kikuchi Diffraction via Topography

Si electrons transmitted by GaN nanowire:

[4]:
```
display(Image(filename='./effects_gallery/transmit_Si_GaN_img001341.png', width=320))
```

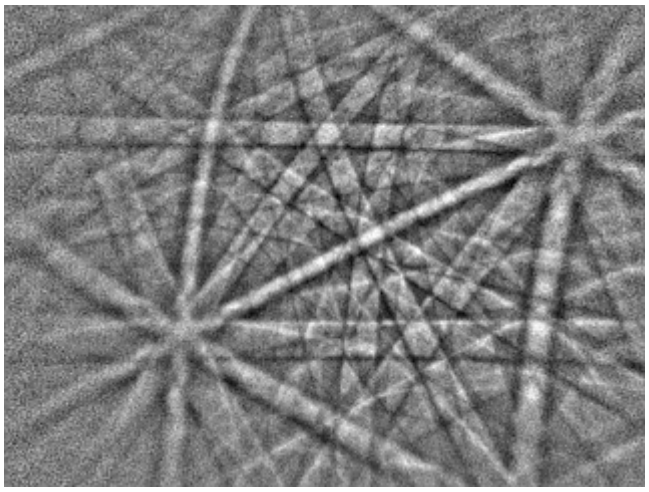


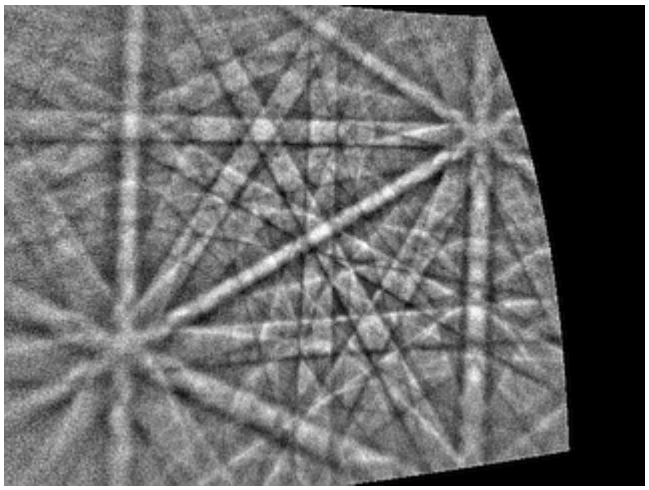
Magnetic Field Distortions

SEMs with immersion lenses have significant magnetic fields in the region near the pole piece, including between sample and phosphor screen. These fields distort the trajectories of the electrons

scattered from the sample. Kikuchi patterns measured under these conditions need to be "unwarped" to provide correct crystallographic information in the sample coordinate system.

[5]:
```
display(Image(filename='./effects_gallery/mf.png', width=320))
display(Image(filename='./effects_gallery/mf_unwarped.png', width=320))
```





[ ]:

3 Image Processing

3.1 Kikuchi Pattern Processing

Basics of Kikuchi Pattern Processing

Initialization code

[1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from IPython.core.display import Image, display

import skimage
```

(continues on next page)

(continued from previous page)

```
import skimage.io

# aloe routines
from aloe.jutils import show_source
from aloe.plots import plot_image
from aloe.image.kikufilter import make_signal_bg_fft, remove_bg_fft
from aloe.image.kikufilter import img_to_uint
from aloe.image.downsample import downsample
from aloe.image.kikufilter import process_ebsp
```

Motivation

- data acquisition
- data preparation / filtering / cleaning

Electrons scattered from the sample in an SEM carry crystallographic and other information which is conveyed in the intensity distribution measured on a two-dimensional electron detector, the electron backscatter pattern (EBSP). Useful information is distributed in the EBSP on different spatial and intensity scales, i.e. we often observe a relatively low-intensity signal varying over a few pixels (the Kikuchi diffraction signal) superimposed on a relatively large, dominating "background" signal that varies slowly on the scale of the detector dimensions (large spatial dimension).

For pure image processing purposes, we can artificially partition the as measured EBSP as the sum of a Kikuchi signal and the background signal, acted on by a DETECTOR_RESPONSE function:

EBSP = DETECTOR_RESPONSE @ (KIKUCHI + BACKGROUND)

The camera electronics has to be able to capture the large intensity variation from the edges to the middle of the image. At the same time, the camera also has to sufficiently resolve the remaining small signal variations in the small, a-few-pixel-sized regions where the Kikuchi bands and lines are observed, which we aim to extract in for the subsequent crystallographic analysis.

EBSP pattern processing serves the purpose to process the measured raw pattern data in such a way that the crystallographic information stored in the Kikuchi diffraction pattern can be extracted in the fastest way by the subsequent "indexing" steps.

Beyond the mere extraction of the crystallographic orientation from the Kikuchi pattern, it would also be desirable to define a specific data processing pipeline which helps to extract meaningful data on the physical processes that govern the signal formation and thus extract additional information about the sample scattering the electrons. This aim can be reached only partially, i.e. the clear separation of image processing and extraction of physical process data is not always possible. This concerns questions such as the details of the energy and momentum distribution and trajectories of electrons scattered inside the sample. These can be assumed to be different for those electrons that form the Kikuchi diffraction patterns vs. the "background" electrons.

The aim of the image processing of the raw EBSD patterns is to extract the information which is necessary as input for the next data analysis steps. Mainly, this concerns the extraction of the diffraction information, but we can also be interested in the angle-dependent backscattered intensity independent of any diffraction effects.

In dependence on the requirements of the subsequent processing steps, the image processing needs to conserve detail to different extents.

80x60: For example, the line detection via the Hough/Radon transforms will benefit from a different image processing compared to quantitative pattern matching using simulated patterns based on electron diffraction physics. The most important aspect for the HT approach is to be able to detect any possible Kikuchi bands which are present, even in rather noisy raw data, and to do this with the highest speed possible in oder to make high indexing rates possible. As long as the band detection

works reliably, we can live with noisy data, which would be otherwise of insufficient quality for a quantitative analyis. This is also true for pattern matching approaches for low-resolution orientation determination, which can be stable against even noisier data than the Hough transform approach.

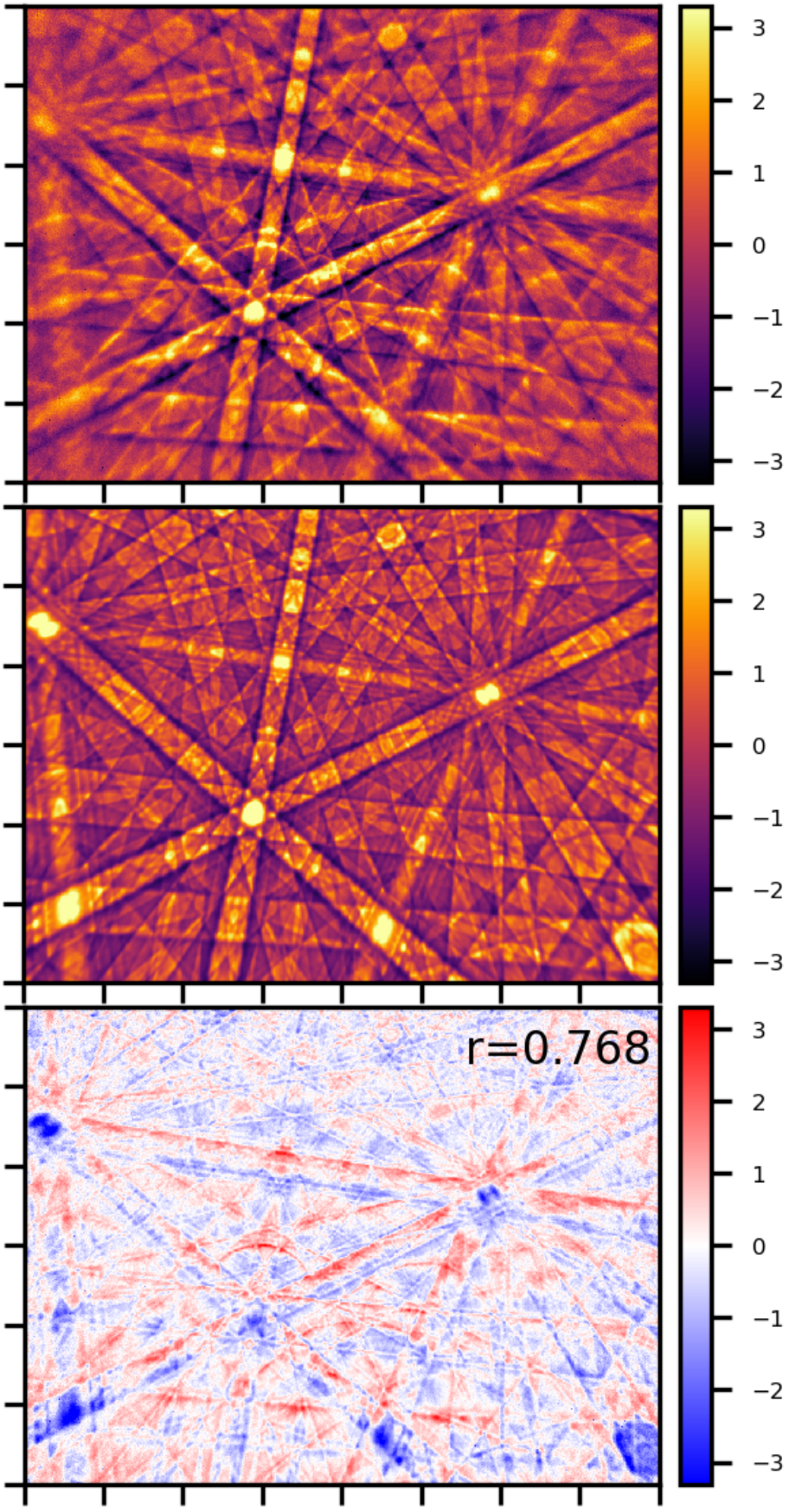
800x600: In comparison, the image processing pipeline for a pattern matching approach for phase identification or high-resolution orientation measurements has to ensure that the quantitive physical information, which we describe by the theoretical simulation model, will not result in distorted or biased data after the image processing procedure. We have to be able to describe the effects of the image data processing relative to the image model based on scattering physics of electrons in the sample.

The quantitative image matching relies on fine details for which the Hough transformation is at best insensitive or in the worst case even negatively influenced (e.g. if beyond an optimum value, pattern resolutions can actually be too high for the Hough transform, because highly resolved patterns can produce additional peaks in the Hough transform which can confuse a peak detection algorithm.)

Example 1: Pattern matching for advanced phase analysis and high-precision orientation determination

Below, we compare a processed, low-noise experimental Kikuchi pattern form a silicon sample (800x600, top) with a simulation using the dynamical theory of electron diffraction (middle) and their difference (bottom). The experimental pattern was measured at 15kV acceleration voltage, the simulated pattern fits best if we assume a mean energy of about 14keV for the backscattered electrons which contribute specifically to the Kikuchi pattern. The value of the normalized cross correlation coeffcient $r = 0.768$ indicates a very good fit between experiment and simulation. The patterns have been normalized to a mean value of $\mu = 0.0$ and a standard deviation $\sigma = 1.0$ for consistent comparison between experiment and simulation. The difference plot in the bottom panel shows that (compared to the simulation) the bands in the experimental pattern have slightly higher intensity (red) on their top side than on the bottom side (blue). This specific difference between experiment and simulation is related to the so-called "excess-deficiency effect", which is not treated in the simplified simulation shown here.

[2]:
```
display(Image(filename="./data/Si/Si_ROI_0_E_14000_W500_B1.5.png", width=500))
```

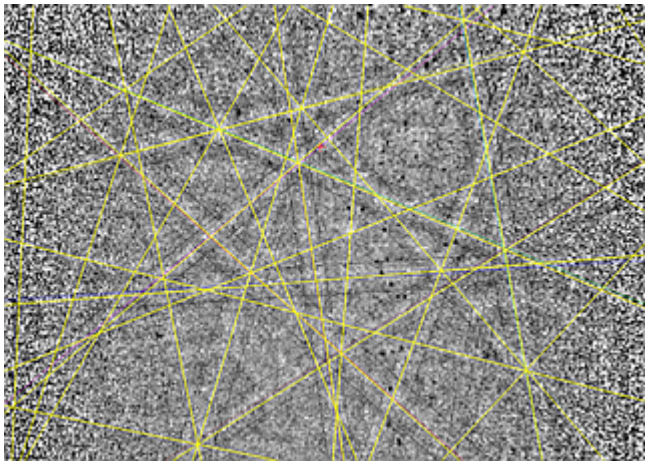

3
2
1
0
−1
−2
−3
3
2
1
0
−1
−2
−3
r=0.768
3
2
1
0
−1
−2
−3

Example 2: Line detection for fast indexing

Below we show an example pattern from a conventional EBSD indexing approach. The final indexing lines are displayed on the (noisy) Kikuchi signal in the original measured resolution, which is 320x240. The signal-to-noise ratio is suffcient for fast orientation measurements. Internally, the Hough transform line detection and subsequent indexing algorithms use binned and binarized images, i.e. at 80x60 pixels, which is shown at the bottom.

The use of the binned data in the indexing stage suggests that the resolution of the Hough transform line detection process can limit the resolution of the orientation determination, and not necessarily the resolution of the measured EBSD patterns. In this way, increasing the resolution of your measured EBSD pattern does not always improve the orientation data!

[3]:
```
display(Image(filename='./data/ocean_116_176_indexed.png'))
display(Image(filename='./data/ocean_116_176_binary.png'))
```



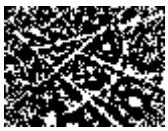


Image Processing

In an 8bit pattern, we have available a range of 256 grey values from 0 to 255 which can be adjusted by setting the gain and offset of the camera: the offset determines the pixel values with 0 intensity, and the gain allows to scale the intensity so that the highest intensity values are at 255.

The full dynamic range is often not filled in each pattern of a map, because the total BSE signal itself can vary at different places in the map, e.g. different phases can have different backscattering coefficients. In order to capture these variations, some patterns might have too little intensity for the full range in weakly backscattering phases in the map, while others have a range of intensity values that fits the full range of available channels in strongly backscattering regions. Also, the range of the noise needs to be considered.

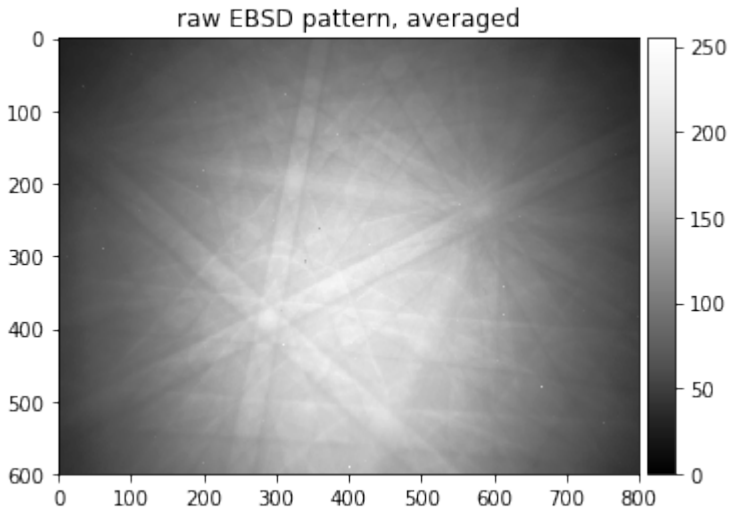
We see that the camera parameters have to be set to match the signal variations which are present (a) inside a pattern and (b) inside a map. As this is not perfectly possible at all times, compromises have to be made, underexposing patterns in some regions and overexposing the patterns in others. A 16bit camera providing 2^16 gray levels might help if the dynamic range of the sample (map and patterns) is too high to be efficiently captured in an 8bit setup. Note that many cameras can only 12bit dynamic range, so although the image format might use 16bit, these might not be actually fully used by the camera.

Low noise patterns for HR work

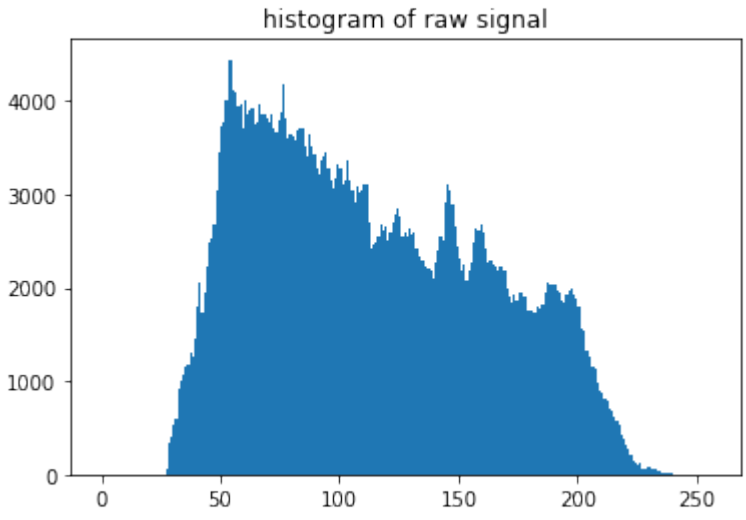
This is an average pattern of 10 patterns from a single grain, saved as ASCII text floating point numbers. The imported data will have 64bit floating point resolution in memory; note that importing this e.g. as 8bit integers (np.uint8) would reduce the intensity resolution available by the pattern averaging (the same applies escpecially for the static background pattern, see below).

[4]:
```
raw_pattern = np.loadtxt("./data/Si/Si_15000_10_rot.dat")
raw_pattern = 255 * raw_pattern/np.max(raw_pattern)
print(raw_pattern.dtype)
plot_image(raw_pattern, [0,255], title='raw EBSD pattern, averaged')
```

```
float64
```



[5]:
```
n, bins, patches = plt.hist(np.ravel(raw_pattern), 256, range=[0,256])
plt.title('histogram of raw signal')
plt.show()
```



[6]:
```
print('pattern height, width: ', raw_pattern.shape)
print('maximum grey value in image: ', np.max(raw_pattern))
print('minimum grey value in image: ', np.min(raw_pattern))
print(raw_pattern)
```

```
pattern height, width:  (600, 800)
maximum grey value in image:  255.0
minimum grey value in image:  27.2055352055
[[ 29.45421245  29.57875458  29.44037444 ...,  29.01139601  28.99063899
   29.21204721]
 [ 29.48188848  29.33658934  29.03907204 ...,  29.19129019  28.92144892
   29.3019943 ]
 [ 29.34350834  29.52340252  29.48880749 ...,  29.7032967   29.20512821
   29.24664225]
 ...,
 [ 40.60765161  39.86039886  40.22018722 ...,  39.07855108  38.47659748
   39.12006512]
 [ 40.06796907  40.16483516  40.44851445 ...,  38.93325193  38.72568173
   39.02319902]
 [ 40.89133089  40.53846154  40.4969475  ...,  39.45909646  38.78795279
   38.75335775]]
```

Deciphering the Pattern Information

The approach of the "dynamic" background correction is to to filter each pattern seperately without any additional data needed. Note that "dynamic" refers to this pattern-individual approach and is not related to the "dynamical simulations" of Kikuchi patterns.

A simple approach to obtain only the large scale variations (i.e. within a scale of many pixels) is to apply a Fourier transform filter that keeps only the low frequency components (low pass filter). The "filterfft" routines in the "BGK" function is accomplishing this low-pass FFT filter:

For pattern processing by fast Fourier transform[2] (FFT) filtering, we will use the "filterfft" routines, developed originally by Connelly Barnes http://www.connellybarnes.com/work/, with the code available at http://www.connellybarnes.com/code/python/filterfft :

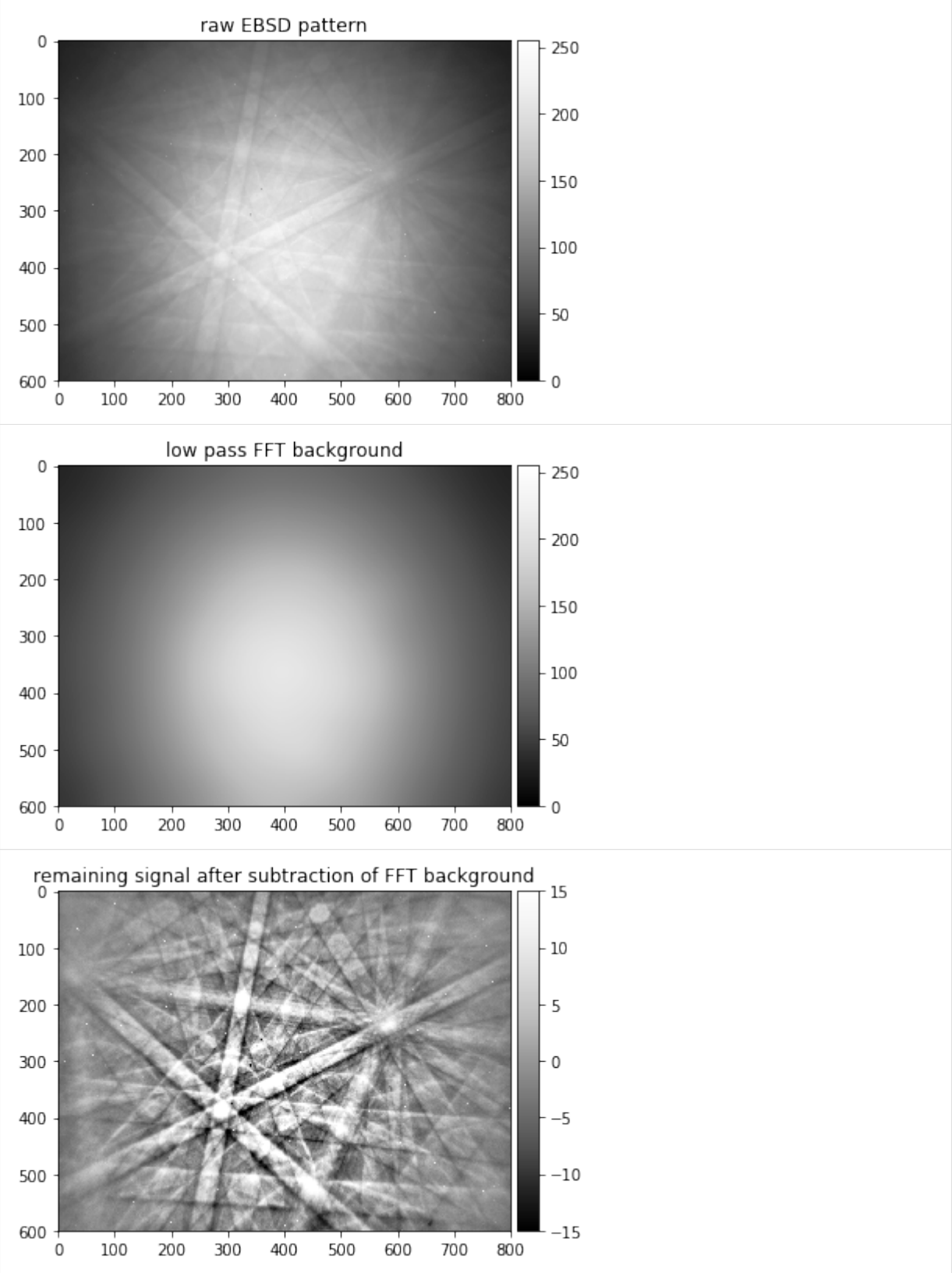
[7]:
```
show_source(make_signal_bg_fft)
```

```
<IPython.core.display.HTML object>
```

[8]:
```
#raw_pattern=raw_pattern1
print(raw_pattern.shape[0])
raw_pattern_signal, raw_pattern_bg = make_signal_bg_fft(raw_pattern, sigma = 40)

plot_image(raw_pattern, [0,255], title='raw EBSD pattern')
plot_image(raw_pattern_bg, [0,255], title='low pass FFT background')
plot_image(raw_pattern_signal, [-15,15], title='remaining signal after subtraction of FFT␣
↪background')
```

```
600
```

[2] https://en.wikipedia.org/wiki/Fast_Fourier_transform



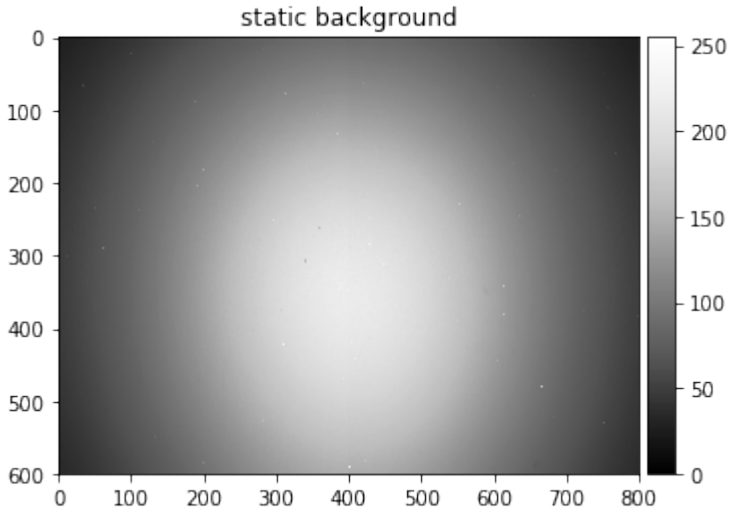
Unfortunately, we see that while we empasize the Kikuchi diffraction signal, there are some disturbing features remaining. In this example, we see “hot pixels” that always give a high signal (white pixels).

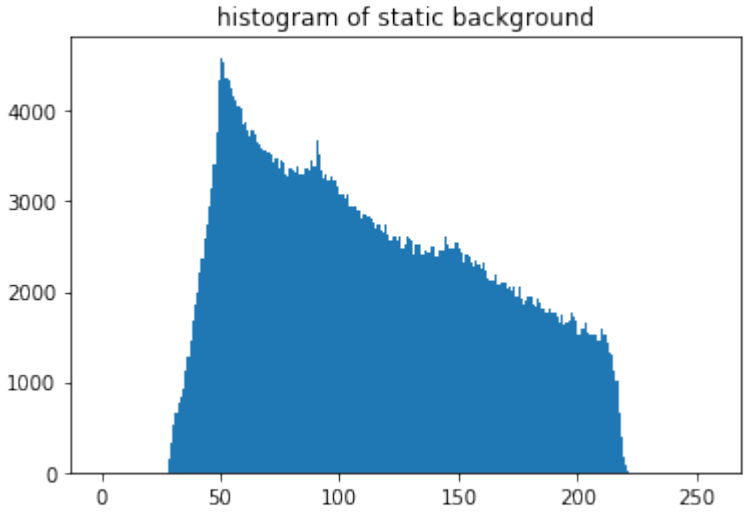
To remove those effects which are constant, we can divide them out by taking a reference image with low noise and without any diffraction features. This is called *Static Background* correction.

The Static Background

[9]:
```
static_bg=np.loadtxt('./data/Si/Al_StaticBackground_BAM.dat')
static_bg = 255 * static_bg/np.max(static_bg)
print(static_bg.shape)
plot_image(static_bg, [0,255], title='static background')

# histogram
n, bins, patches = plt.hist(np.ravel(static_bg), 256, range=[0,256])
plt.title('histogram of static background')
plt.show()
```

```
(600, 800)
```





In order to see the different contributions to the experimental static background, we can deconstruct it into a smooth part by FFT filtering (just like for the dynamic background correction) and a small scale part.

We can discern different contributions to the detector response, which can be attributed to the phosphor screen (dark spots caused by dust), and to the CCD chip used for capturing the image from the phosphor screen. The CCD chip is combined out of two halves, which can have a slightly different response. In addition there are "hot pixels" of high intensity, vertical lines (thermal curtaining). All of these effects are well known properties of CCD cameras which can be corrected for.

If we assume that these detector contributions stay constant over time, i.e. they are contributing to the

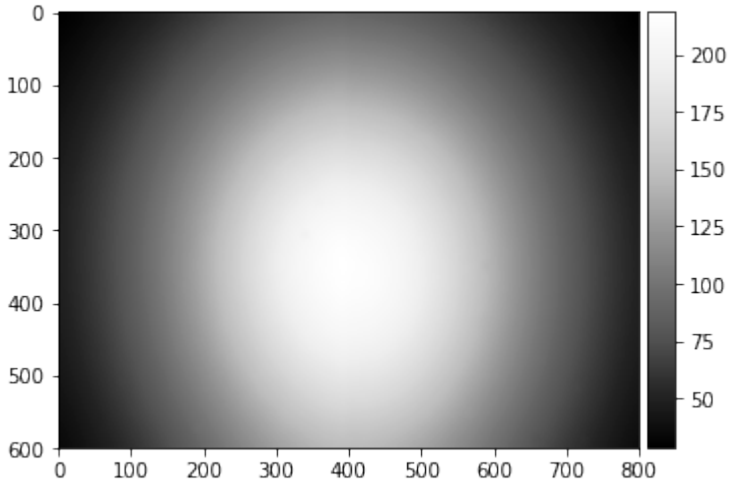
measured data in the same way, we can divide out the static contributions.

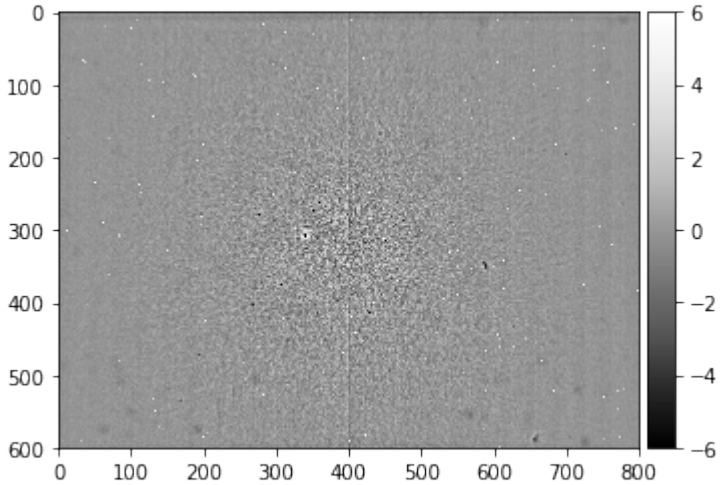
The large scale smooth intensity varaiations which are caused by the angular distribution of the backscattered electrons and the gnomonic projection effects on the detected solid angle per pixel are handled well by the dynamic background correction discussed next.

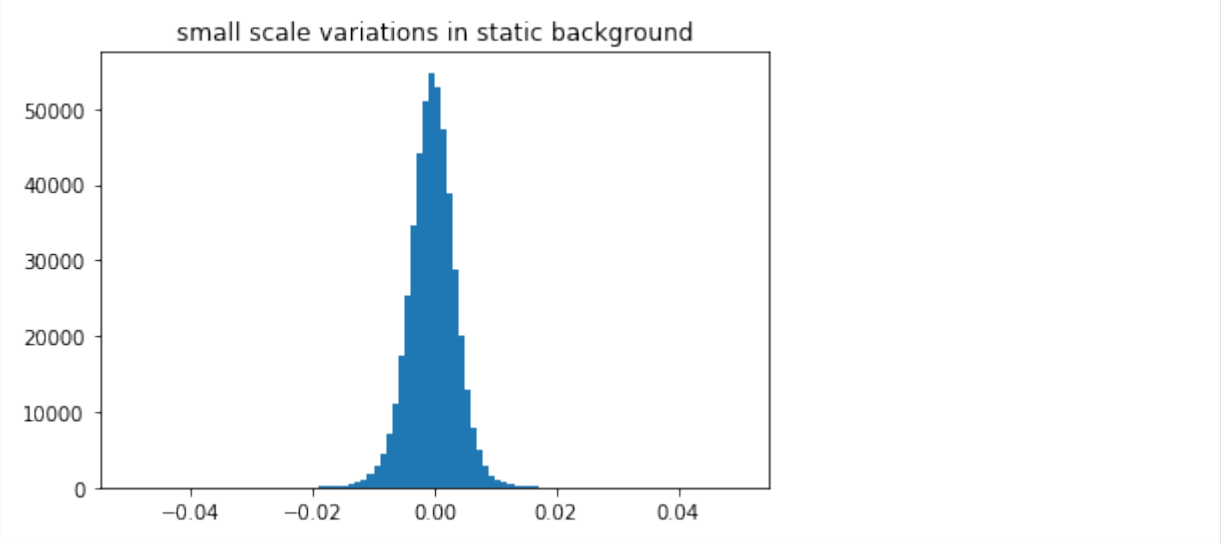
Properties of the static background

Independent of the method we have used to acquire a static background image, we should check this static background for any residual diffraction features and the remaining noise. As the magnitude of these effects might be hard to see in the static background data, we apply a dynamic background correction to the static background and partition the static background in a large scale and in a small scale varying part.

[10]:
```
screen_static, static_bg_smooth = make_signal_bg_fft(static_bg, sigma = 5, bgscale=1.0)

plot_image(static_bg_smooth)
plot_image(screen_static,[-6,6])

small_scale_variations=screen_static/static_bg_smooth
n, bins, patches = plt.hist(np.ravel(small_scale_variations), 100, range=[-0.05,0.05])
plt.title('small scale variations in static background')
plt.show()
```







The fine details of the static background show the hot pixels seen previously, but we can also reveal black spots and smudges due to contamination on the phosphor screen. We also have electronic hardware effects of the camera (horizontal lines in the top region), we see two halves of the CCD chip, possibly exposed differently and divided by the central vertical line, as well as additional banding noise[3] of the digital image sensor (the vertical stripes).

If the black or white spots are very frequent, the histogram of the small scale variations will show asymmetries, with more values on the lower side corresponding to the dark spots formed by the contamination on the screen and the horizontal artefact in the top lines.

We can test that both contributions combined actually result in exactly the original static background saved. In this way, we have a well-defined segmentation of the image data into a background and an additive signal. If we combine both contributions and subtract them form the original data, we should obtain a zero difference image. This can be tested by comparison of two arrays using the function `numpy.testing.assert_almost_equal`.

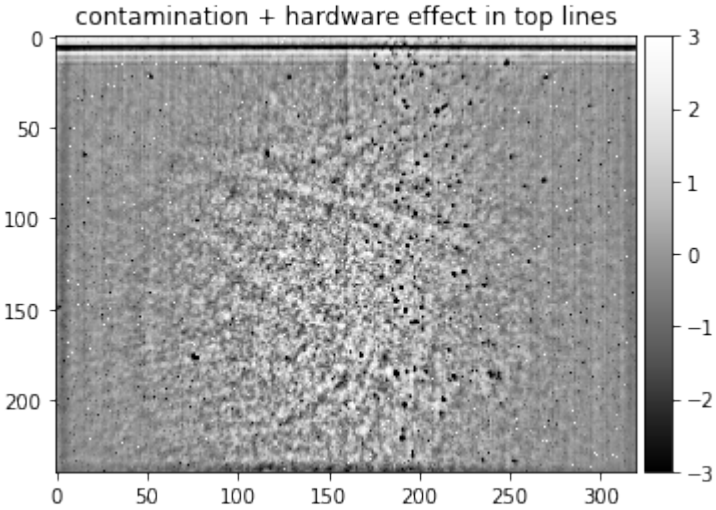
[11]:
```
all_recombined = screen_static + static_bg_smooth
np.testing.assert_almost_equal(static_bg, all_recombined)
```

Examples of Detector Effects

Detector with severe contamination, a hardware problem for the top lines, different exposure in CCD halves, hot pixels, banding noise, and visible Kikuchi band features due to insufficient averaging when taking the static background:

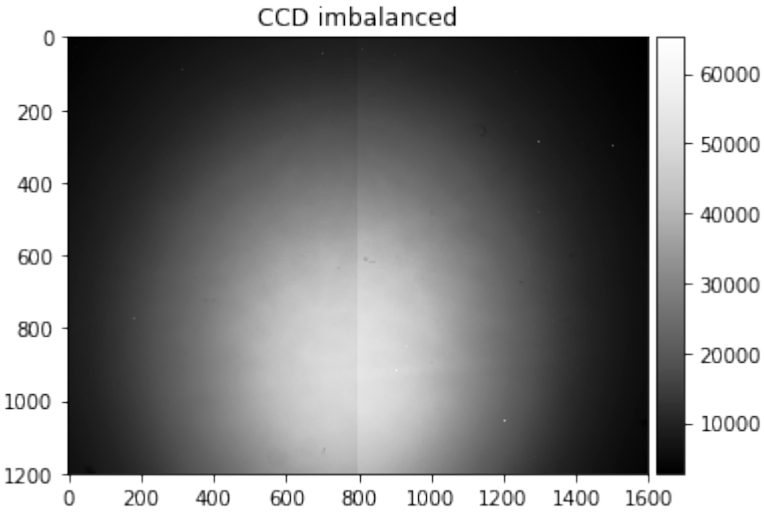
[12]:
```
static_ocean = np.loadtxt("./data/static_ocean.dat")
plot_image(static_ocean, [-3,3], title = "contamination + hardware effect in top lines")
```

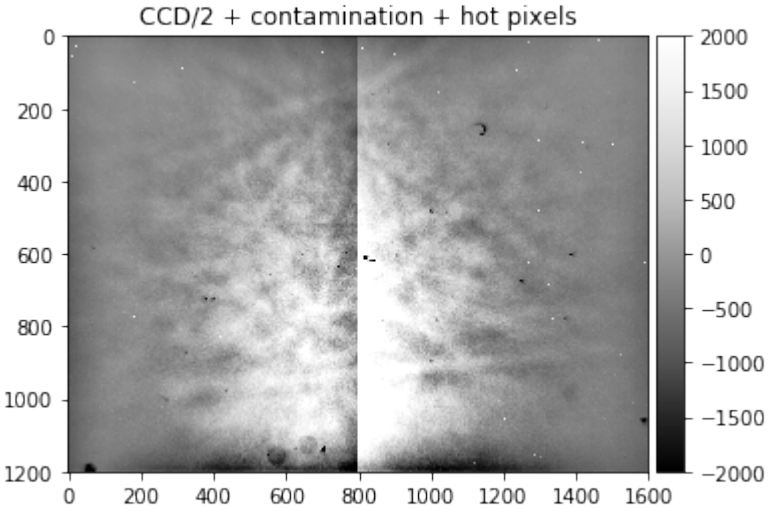
[3] https://photo.stackexchange.com/questions/41874/what-causes-banding-noise



Detector with severely unbalanced exposure in the CCD halves, taken from the AstroEBSD example data set at https://zenodo.org/record/1214829 (due to the limited number of grains in the map, some Kikuchi features are still visible, illustrating the limited possibility to extract a static background afterwards if not enough grains with different orientations can be averaged):

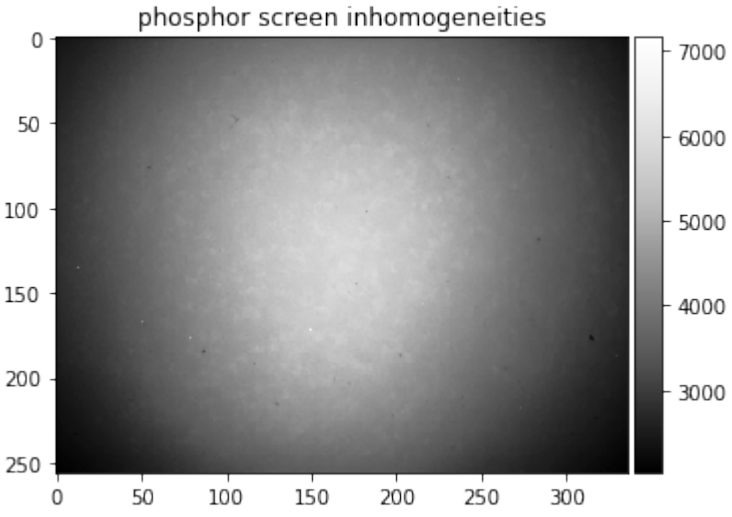
[13]:
```
astro_map_average = np.loadtxt("./data/static_astro.dat")
static_astro, static_astro_smooth  =  make_signal_bg_fft(astro_map_average, sigma = 100,␣
↪bgscale=1.0)
plot_image(astro_map_average, title= "CCD imbalanced")
plot_image(static_astro, [-2000,2000], title= "CCD/2 + contamination + hot pixels")
```

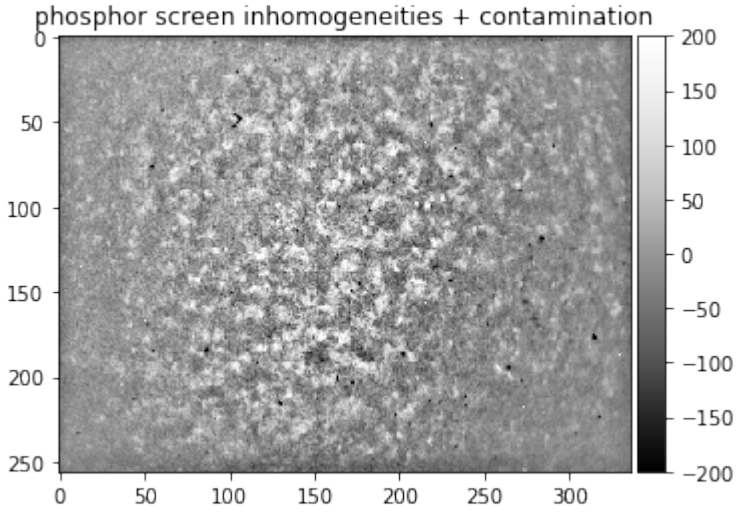




Phosphor Inhomogeneities:

[14]:
```
static_soup = np.loadtxt("./data/static_wedding_soup.dat")
static_fine, static_astro_smooth  =  make_signal_bg_fft(static_soup, sigma = 10, bgscale=1.
↪0)
plot_image(static_soup, title= "phosphor screen inhomogeneities")
plot_image(static_fine, [-200,200], title= "phosphor screen inhomogeneities + contamination
↪")
```



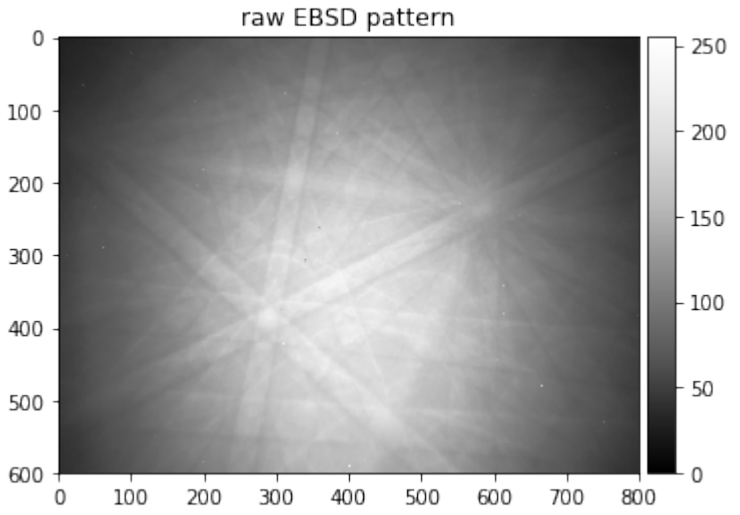


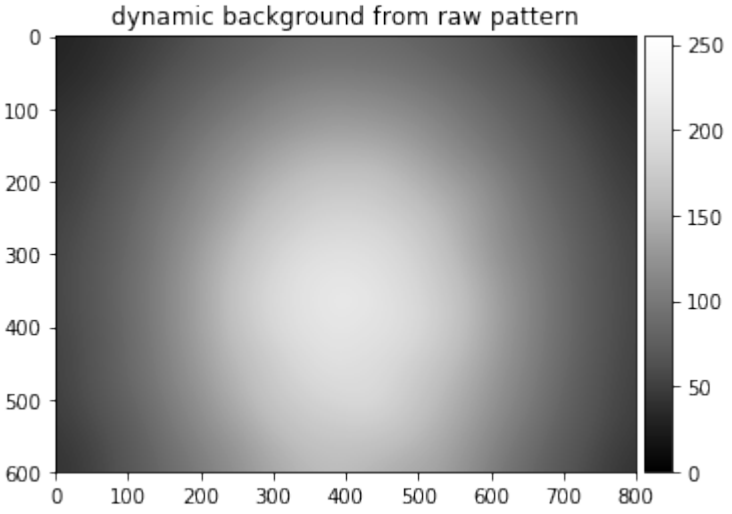
The Dynamic Background

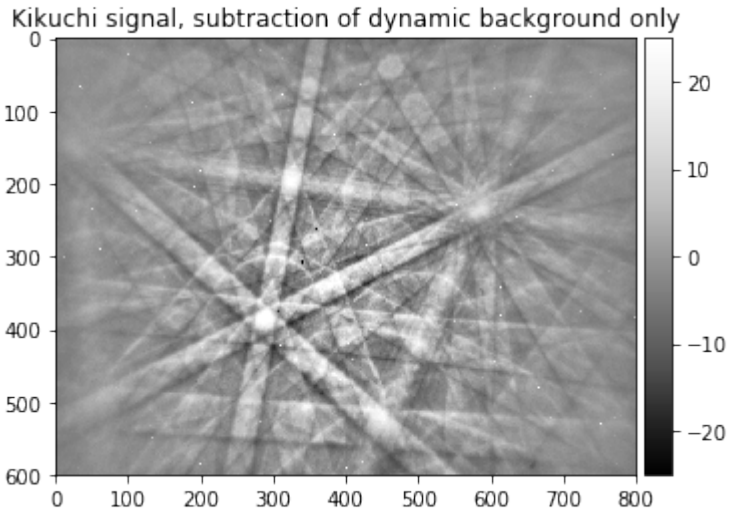
omit static background, live with detector response

i.e. post processing of raw data for which no static background was measured

[15]:
```
raw_pattern_signal, raw_pattern_bg=make_signal_bg_fft(raw_pattern, sigma=40, bgscale=1.0)
signal_relative=raw_pattern_signal/raw_pattern_bg

plot_image(raw_pattern, [0,255], title='raw EBSD pattern')
plot_image(raw_pattern_bg, [0,255], title='dynamic background from raw pattern')
plot_image(raw_pattern_signal, [-25,25], title='Kikuchi signal, subtraction of dynamic␣
↪background only')
```



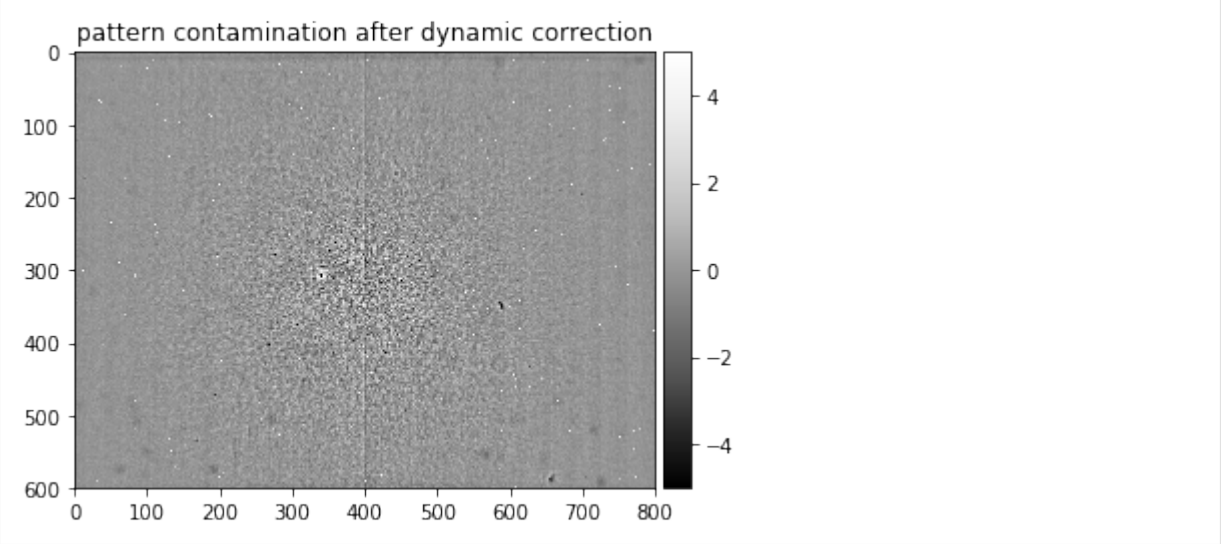




Pattern contamination after dynamic background correction

If only the dynamic background correction is used, the resulting pattern will still contain a contamination by detector features which have been "averaged away" by the filtering process (i.e. the small scale features of the static background image):

[16]:
```
plot_image(screen_static,[-5,5], title='pattern contamination after dynamic correction')
```



We see that the detector response is present in addition to the actual Kikuchi pattern signal. In many cases, the indexing process can live with such data.

Static Background Correction and Flat Fielding

Static Background Division

Imagine that you take a picture from a white piece of paper. You know that the paper is perfectly and uniformly white in reality. If you see any disturbing features in the final image data (maybe some pixels are darker than others; maybe some pixels show stange colors), you would assign these features to the optical and electronic system of your camera. If these effects are removed from your imperfect image, it should appear perfectly white again and give a correct representation of the physical reality. In a digital camera, this correction process is largely hidden from the user, but you can easily imagine that different pixels on the image sensor can have slightly different sensitivities, or there could be dust on some pixels which also reduces the pixel signal.

In EBSD, we also need to remove the relative variations of the detector response in the image. We can achieve this by division of each single pattern by the same constant "static background". This procedure is also known in the contexts of digital imaging as flat field correction[4] with the aim to remove artifacts from 2D images due to the pixel-to-pixel sensitivity of the detector and other influences in the imaging system.

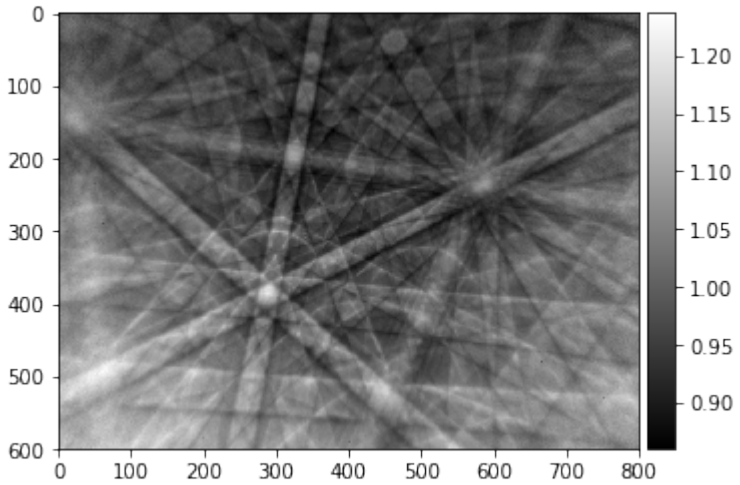
After the correction for the detector response, we can handle the remaining large-scale intensity and angular variation from pattern to pattern individually for each pattern by a "dynamic background" correction. Note that, ideally, the detector response could be taken with an almost constant intensity over the whole image by an even illumination of the phosphor.

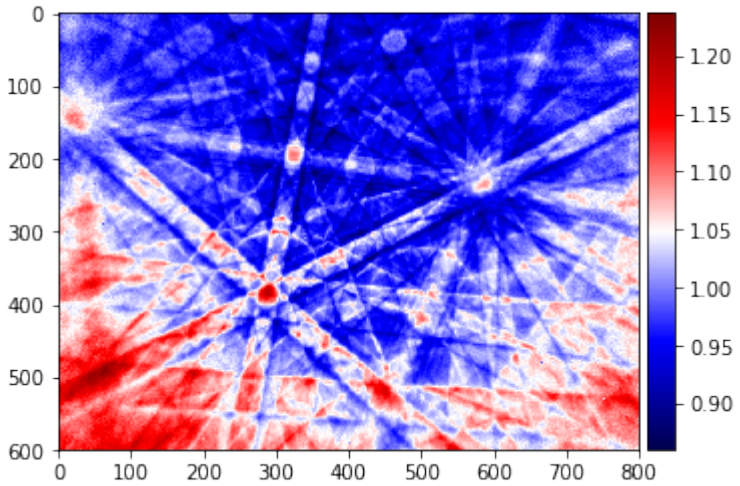
Experimentally, an average static background over a large sample area is often used for background correction. If the Kikuchi features are averaged out, the static background corrects for two different effects: (a) the detector contamination, dead pixels, hot pixels etc and (b) the average, smooth variation of the background signal. Note the difference between (a) and (b): (a) are the constant effects of the detector, (b) also changes the signal that we measure.

By background division we keep at least a relative size of the signal. Subtraction does not allow to conclude on the size of the background that was subtracted.

[4] https://en.wikipedia.org/wiki/Flat-field_correction

[17]:
```
pattern_div=raw_pattern/static_bg
#pattern_div=screen_static
plot_image(pattern_div, cmap='gray')
plot_image(pattern_div, cmap='seismic')
```





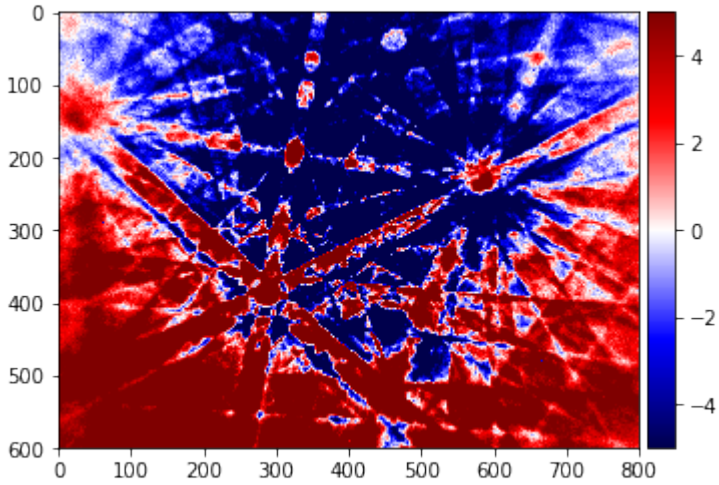
Static Background Subtraction

In contrast to the division, the subtraction of the static background is problematic.

We do not know the correct absolue scale of the static signal, so simply subtracting it will not lead to a good signal, because either the background or the raw pattern will dominate the result, depending on which is larger.

We need to scale the static background to vary approximately in the same limits as the raw pattern. The necessary scaling factor will be related to the relative exposure times and signal sizes of the raw data and the background. But note that this factor will in principle change from one pattern to the next, as different phases can produce more or less signal, with, in addition, a different angular distribution.

[18]:
```
pattern_sub=raw_pattern-static_bg
plot_image(pattern_sub, [-5,5], cmap='seismic')
```

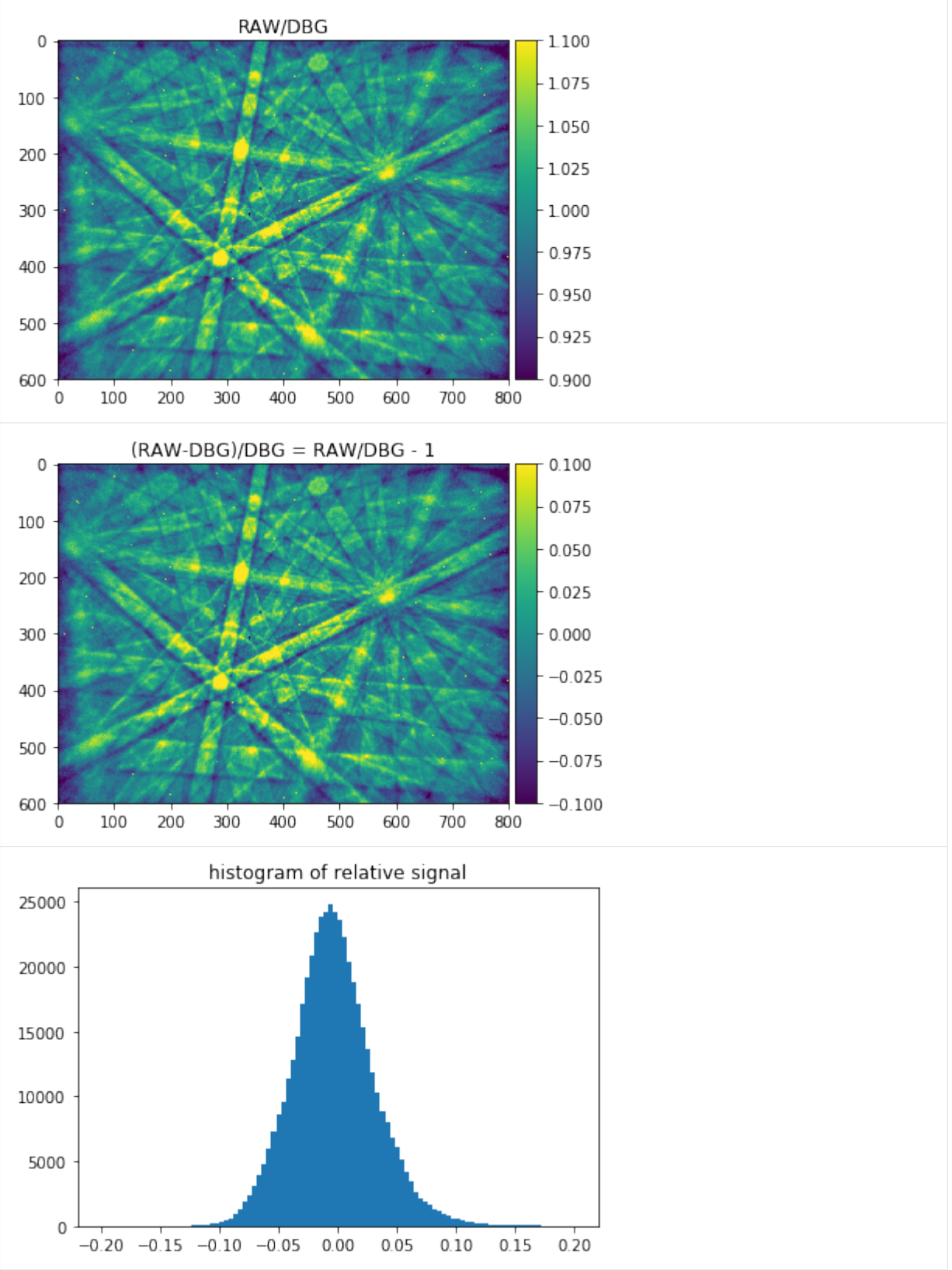


Dynamic Background Correction

With the constant detector response corrected by the static background, we will now try to remove the remaining large-area variations which are larger than our Kikuchi band features which we want to emphasize as much as possible. A possible approach is to use a Fourier transformation filter to suppress the image components which correspond to the large spatial extensions, which relate to low spatial frequencies, i.e. we need to do a high pass filter on the image.

Estimating the relative signal size in the raw uncorrected pattern data

[19]:
```
signal_amp = 0.1
plot_image(raw_pattern/raw_pattern_bg,  [1-signal_amp,1+signal_amp],cmap="viridis", title=
↪'RAW/DBG')
plot_image(signal_relative, [-signal_amp,+signal_amp], cmap="viridis", title='(RAW-DBG)/
↪DBG = RAW/DBG - 1')

#plot_image(signal_relative, [-0.1,0.1], title='signal relative to dynamic background')

n, bins, patches = plt.hist(np.ravel(signal_relative), 100, range=[-2*signal_amp,+2*signal_
↪amp])
plt.title('histogram of relative signal')
plt.show()
```



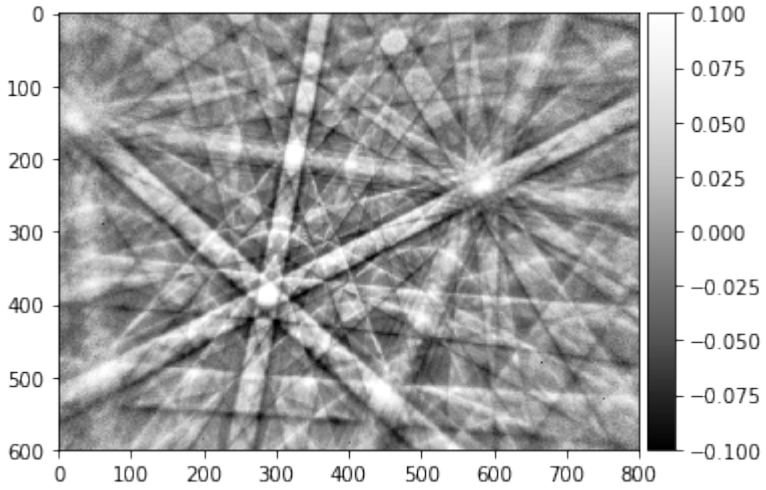
The histogram above shows the quantitative distribution of the relative signal. We see that the relative signal is within a few percent relative to the smooth background, e.g. the Kikuchi signal we aim to pick up is on the order of 10% of the background signal (this depends on the material an can range from a few percent to a maximum of about 20% as seen here for a single-crystal silicon sample).

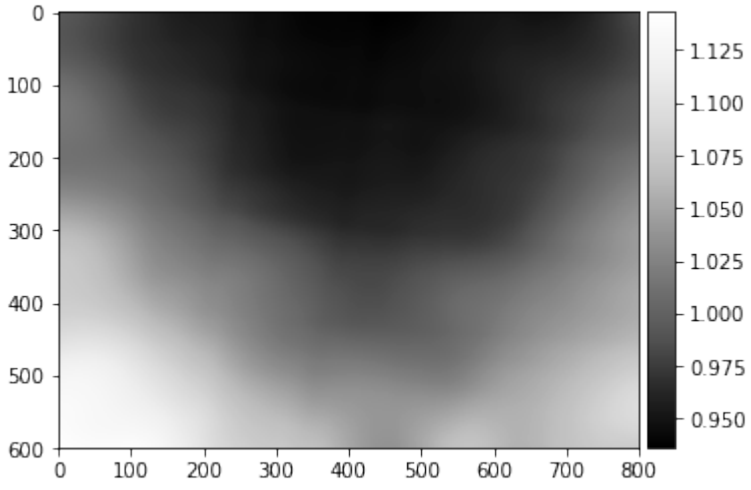
The result for the purely dynamical background correction is first subtraction of the background and

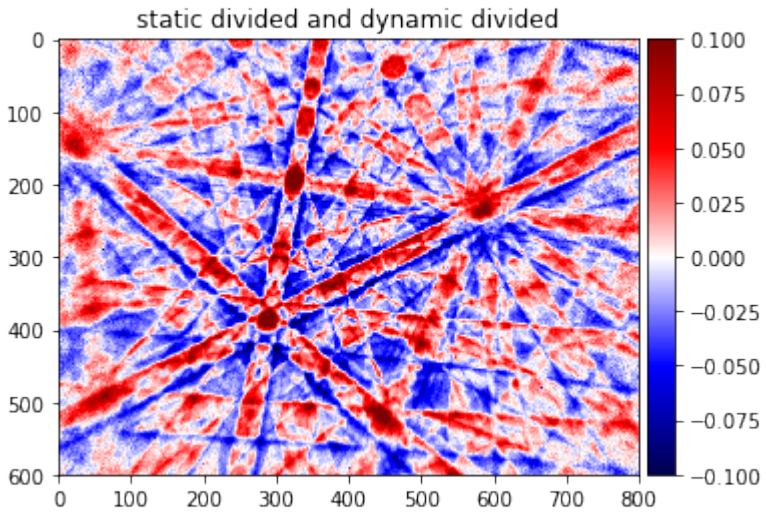
then divide to get the relative contribution.

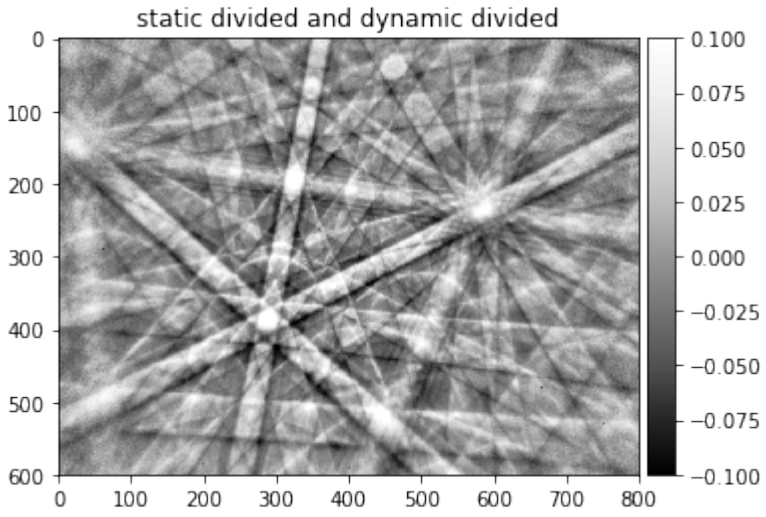
Additonal dynamic background correction of the statically corrected pattern

[20]:
```
pattern_div_sub, pattern_div_bg = make_signal_bg_fft(pattern_div, sigma=50)

plot_image(pattern_div_sub,[-signal_amp,signal_amp])
plot_image(pattern_div_bg)


img_corrected=pattern_div_sub/pattern_div_bg
plot_image(img_corrected,[-signal_amp, signal_amp], cmap='seismic', title='static divided␣
↪and dynamic divided')
plot_image(img_corrected,[-signal_amp, signal_amp], cmap='Greys_r', title='static divided␣
↪and dynamic divided')
np.savetxt('./results/img_corrected.dat', img_corrected)
```









Contrast stretching

http://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_equalize.html

further image processing by routines optimzed for 8bit or 16bit integers

memory considerations compared to 32bit float values

[21]:
```
# get values at 3% and 97% frequency
p_low, p_high = np.percentile(img_corrected, (3, 97))

img_contrast_stretch = skimage.exposure.rescale_intensity(img_corrected, in_range=(p_low,␣
↪p_high))
plot_image(img_contrast_stretch)
```

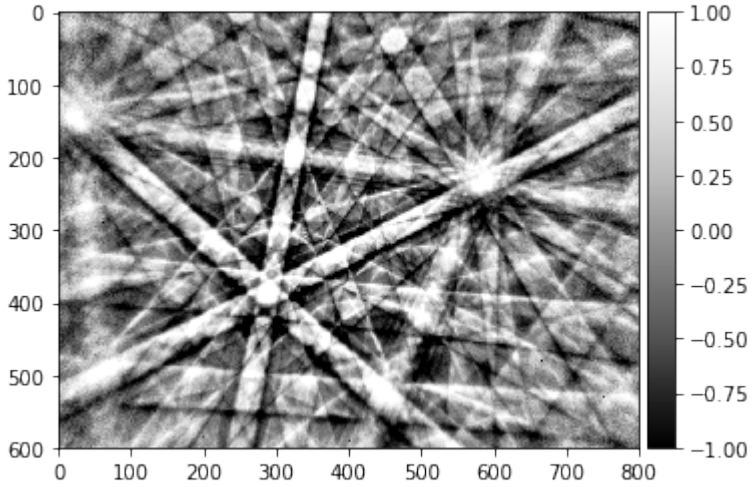


Image Data Type Conversion

Computations on images can lead to arrays of floats with 32bit or even more, which can be memory consuming, slower to compute with and which cannot be directly saved into common image formats that can be viewed without additional tools.

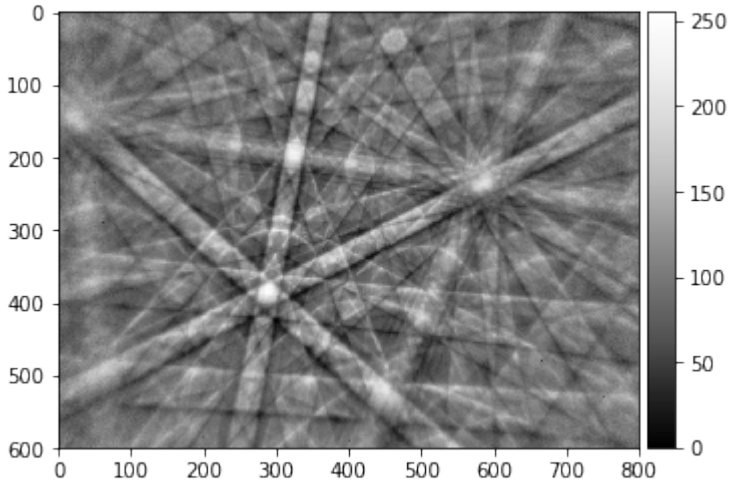
However, in many cases the data analysis is not limited by the floating point precision but by the dynamical range of the resulting signal. Often a range of 256 (8bits) or 65536 (16bits) integer values is sufficient to represent the data. This is why a rescaling and rounding of floating point data to 8 or 16 bit integer values can be useful as the final step of image processing, or even just for saving the image in 8bit formats like PNG or JPEG or 16bit TIFF formats.

The function "img_to_uint" in `aloe.image.kikufilter` can rescale a floating point array to 0..255 (or 65535), with 0 and 255/65535 at a certain contrast percentile. Setting these percentiles e.g. to 0.5 and 99.5 of all floating point pixel values, the lowest 0.5% and the highest 0.5% of the images will be scaled to "black (0)" and "white (255/65535)" respectively. This can reduce the influence of a few outliers with exceptionally high or low floating point values, because the outliers will be in the respective lower or higher precentages of floating point array.

[22]:
```
show_source(img_to_uint)
```

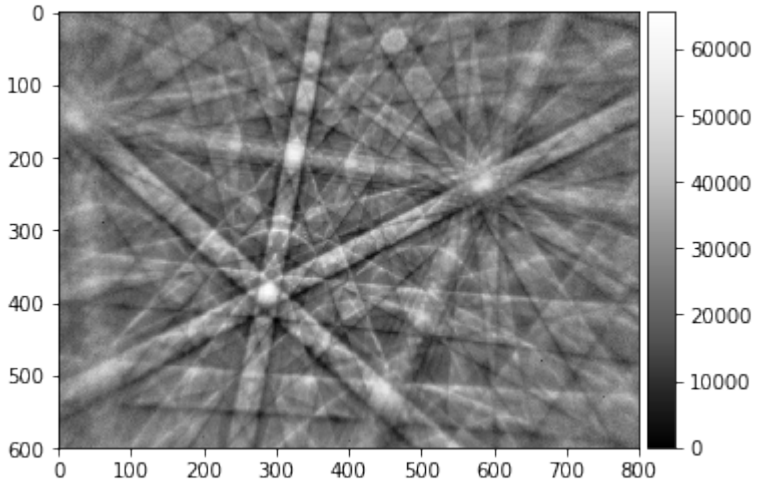
```
<IPython.core.display.HTML object>
```

[23]:
```
img8=img_to_uint(img_corrected, clow=0.001, chigh=99.999)
plot_image(img8)
print(img8.dtype)
skimage.io.imsave('./results/img_corrected_8bit.png', img8)
```



uint8

[24]:
```
img16=img_to_uint(img_corrected, dtype=np.uint16, clow=0.001, chigh=99.999)
plot_image(img16)
print(img16.dtype)
# note: the eye cannot see 65000 grey levels...
```



uint16

To save a 16bit image in TIFF format, we can use skimage, which contains the tifffile.py[5] module from https://www.lfd.uci.edu/~gohlke/code/tifffile.py.html:

[25]:
```
# check for available plugins
skimage.io.find_available_plugins()
```

[25]:
```
{'fits': ['imread', 'imread_collection'],
 'gdal': ['imread', 'imread_collection'],
 'gtk': ['imshow'],
 'imageio': ['imread', 'imsave', 'imread_collection'],
 'imread': ['imread', 'imsave', 'imread_collection'],
 'matplotlib': ['imshow', 'imread', 'imshow_collection', 'imread_collection'],
 'pil': ['imread', 'imsave', 'imread_collection'],
 'qt': ['imshow', 'imsave', 'imread', 'imread_collection'],
 'simpleitk': ['imread', 'imsave', 'imread_collection'],
```

(continues on next page)

[5] https://www.lfd.uci.edu/~gohlke/code/tifffile.py.html

(continued from previous page)

```
 'tifffile': ['imread', 'imsave', 'imread_collection']}
```

[26]:
```
skimage.io.imsave('./results/img_corrected_16bit.tif', img16, plugin='tifffile')

#check that saved image gives the same data when loaded back
im_saved = skimage.io.imread('./results/img_corrected_16bit.tif', plugin='tifffile')
print(im_saved.dtype)

np.testing.assert_array_equal(im_saved,img16)
#print(im_saved-img16)
```

```
uint16
```

Contrast Limited Adaptive Histogram Equalization (CLAHE)

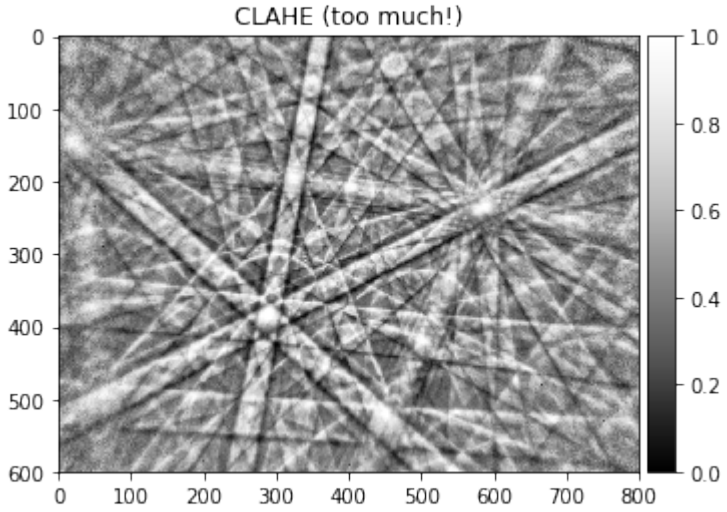
http://scikit-image.org/docs/dev/api/skimage.exposure.html#skimage.exposure.equalize_adapthist

- can change the intensity distribution considerably if not used properly
- intensity massage will affect agreement with simulations; use as mild image processing as possible
- might be better used for band detection, binarization

Note this will change the scaling of different regions in the image, so we have to be careful not to distort the relative intensity in the Kikuchi features. The kernel size should be larger than the Kikuchi features, as not to rescale the intensity in the Kikuchi band profile.

[27]:
```
local_scale = 40 # divide image size by scale to get local kernel size, e.g. 1/10 of image␣
↪dimensions
ksize=np.copy(img16.shape).astype(np.int) // local_scale
print(ksize)
img_adapteq = skimage.exposure.equalize_adapthist(img16, kernel_size=ksize, clip_limit=0.
↪01, nbins=256)
print(img16.dtype)
print(img_adapteq.dtype)
plot_image(img_adapteq, title='CLAHE (too much!)')


img16_clahe=img_to_uint(img_adapteq, dtype=np.uint16, clow=0.001, chigh=99.999)
skimage.io.imsave('./results/img16_clahe.tif', img16_clahe, plugin='tifffile')

#toimage(img_adapteq).save('img_adapteq.png')
```

```
[15 20]
uint16
float64
```



Binning of Pattern Data

- indexing: speed e.g. 80x60 internal image resolution

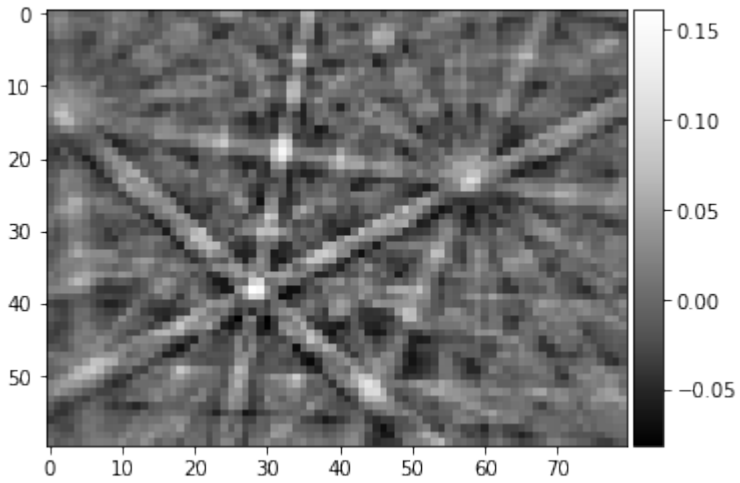
We use the downsample[6] function from Adam Ginsburg's Python Code (agpy)[7] to combine the intensity values from several pixels.

With a binning factor of 4, the intensity of groups of 4x4 pixels is combined into a single pixel. This means that an image of dimension (height=1024, width=1344) is reduced to a dimension of (256, 336). While the resolution of the image is reduced, the noise (e.g. Poisson counting noise) will also be improved as more counts are assigned to a pixel.

Note that the binning will mean that the "center" of the 4x4 pixel region is the new center that is assigned to the measured intensity. This changes the angular calibration of the pixels ("pattern center").

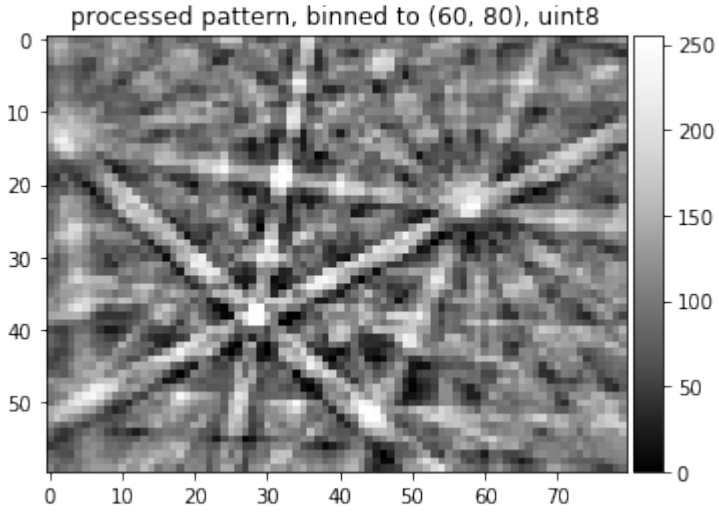
[28]:
```
binning=10
img_binned=downsample(img_corrected, binning)
plot_image(img_binned)

# save the binned down image:
#scipy.misc.toimage(img_binned, cmin=0.0, cmax=255).save('img_binned.png')
print(img_binned.shape)
#print(img_binned)
```

[6] https://github.com/keflavich/image_tools/blob/master/image_tools/downsample.py
[7] http://pythonhosted.org/agpy/image_tools.html




```
(60, 80)
```

[29]:
```
img8_binned=img_to_uint(img_binned)
plot_image(img8_binned, title='processed pattern, binned to '+str(img8_binned.shape)+', '+␣
↪str(img8_binned.dtype))
skimage.io.imsave('./results/img8_binned.tif', img8_binned, plugin='tifffile')
```



Final Pipeline Function

We can combine the image processing steps discussed so far into a function, which can also be accessed in the kikufilter module:

[30]:
```
show_source(process_ebsp)
```

```
<IPython.core.display.HTML object>
```

Summary

The various steps discussed above have been used to implement a simple image processing pipeline which enables us to extract the Kikuchi diffraction information from the raw measured electron backscattering patterns (EBSP).

The basic steps are:

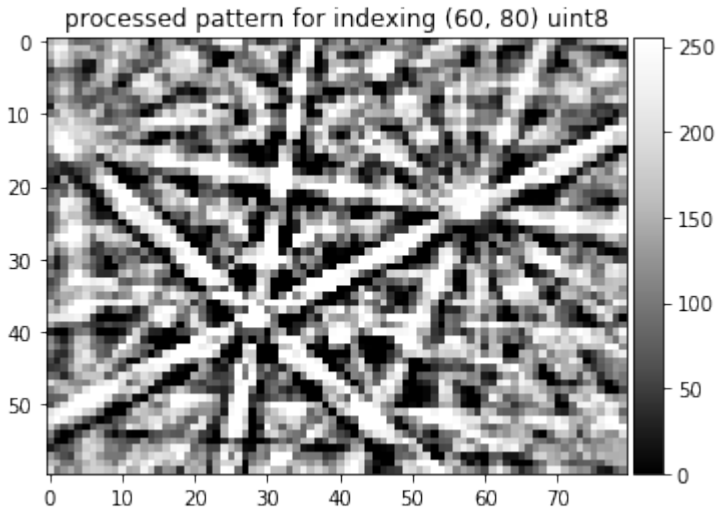
1. divide raw pattern by static background (without any Kikuchi features!) to remove detector contamination and hardware effects
2. correct the remaining large-scale intensity variations, e.g. by FFT filtering (= "dynamic background")
3. process pattern further depending on the application

The binned and the unbinned pattern at the end of the filtering pipeline so far are preconditioned to be used for subsequent application in:

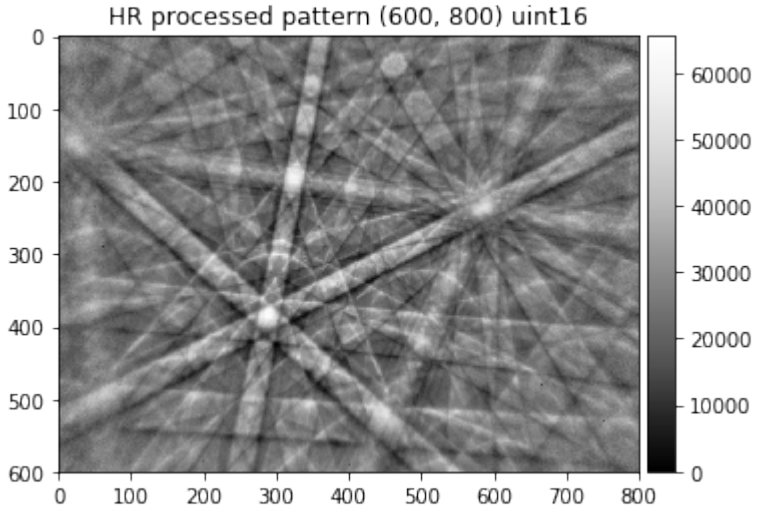
- line detection and conventional indexing by the Hough or Radon transforms
- indexing by pattern matching from a pattern dictionary
- phase ID and orientation refinement from high resolution, low noise patterns, possibly starting from initial guess data obtained by the two previous approaches

The extraction of the Kikuchi patterns from the raw data now works by calling a single function (with sensible default parameters):

[31]:
```
kikuchi_indexing=process_ebsp(raw_pattern, static_bg, dtype=np.uint8,
                              binning=10, sigma=3, clow=10, chigh=90)

img=kikuchi_indexing
plot_image(img, title='processed pattern for indexing '+str(img.shape)+' '+ str(img.dtype))
skimage.io.imsave('./results/img_final_indexing_8bit.png', img)
```



[32]:
```
kikuchi_detailed=process_ebsp(raw_pattern, static_bg, dtype=np.uint16, sigma=40,
                              clow=0.001, chigh=99.999)

img=kikuchi_detailed
plot_image(img, title='HR processed pattern '+str(img.shape)+' '+ str(img.dtype))
skimage.io.imsave('./results/img_final_detailed_16bit.tif', img, plugin='tifffile')
```



Kikuchi Band Detection using the Hough Transform

Motivation

The detection of Kikuchi bands is a first step towards the extraction of quantitative crystallographic information form a measured Kikuchi pattern.

Initialization code

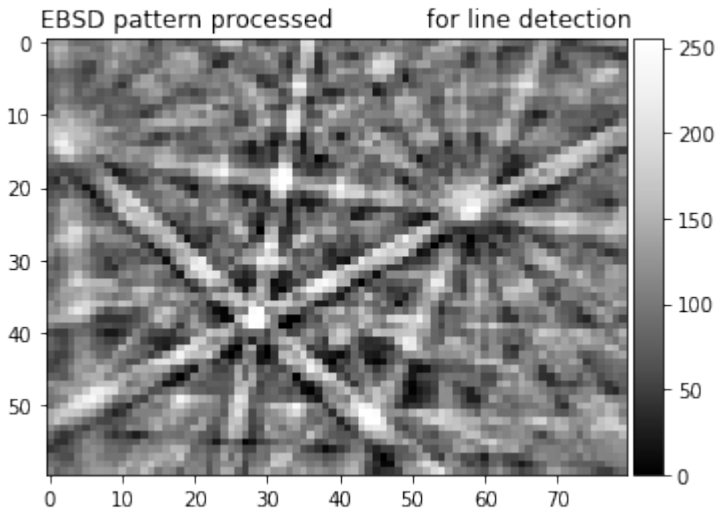
[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import cm

import numpy as np

import skimage.io
from skimage import exposure
from skimage.morphology import disk
from skimage.filters import rank
from skimage.filters import threshold_otsu
from skimage.transform import (hough_line, hough_line_peaks,
    probabilistic_hough_line)
from scipy.ndimage.filters import correlate

from aloe.plots import plot_image
from aloe.image.kikufilter import img_to_uint
from aloe.image.downsample import downsample
```

Loading the Filtered Data

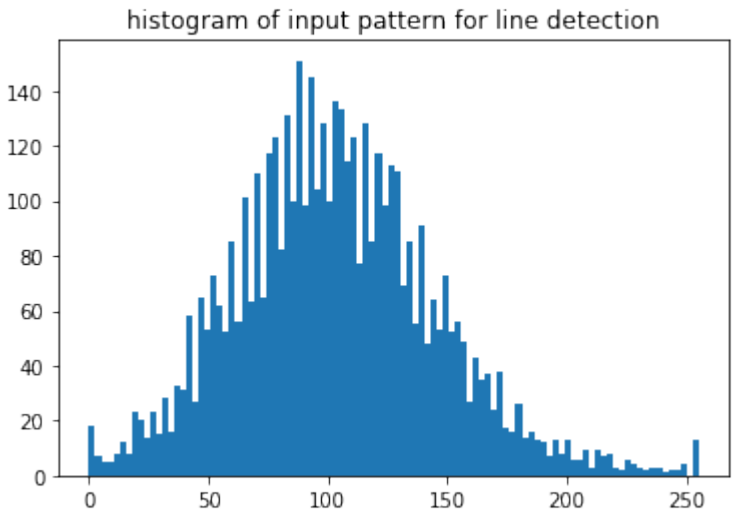
From the previous pattern processing step, we can load a single binned pattern, in 8bit range:

[2]:
```
pattern=skimage.io.imread('./data/Si/Si_img8_binned.tif',
                          plugin='tifffile')
print(pattern.shape, pattern.dtype)
plot_image(pattern, title='EBSD pattern processed \
           for line detection')
```

```
(60, 80) uint8
```



[3]:
```
n, bins, patches = plt.hist(np.ravel(pattern), bins=100)
plt.title('histogram of input pattern for line detection')
plt.show()
```



8bit pattern, 255 gray scales, relatively low contrast, large background offset camera: gain + offset settings, cannot be optimized for each pattern in the map, compromise for whole map with possibly strongly varying signal

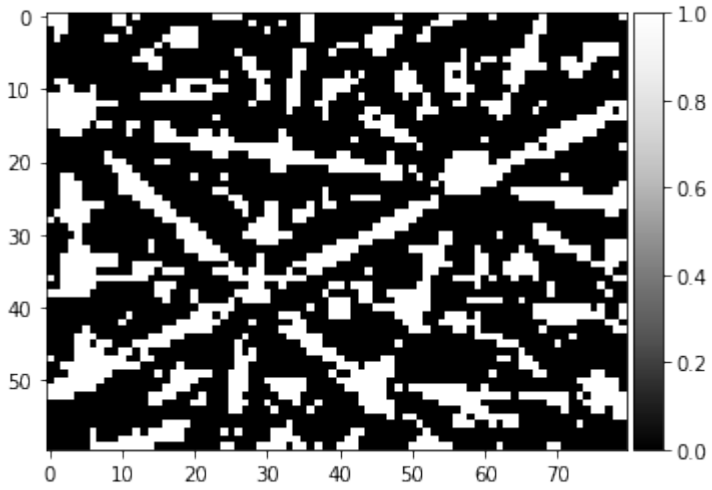
Hough Transform Band Detection

[4]:
```
def otsu(img, factor=1.15):
    """ return True/False array
        from thresholding the image
    """
    thresh = factor*threshold_otsu(img)
    binary = img > thresh
    return binary

binary=otsu(pattern).astype(int)
```

(continues on next page)

(continued from previous page)

```
plot_image(binary)

#skimage.io.imsave('./results/img8_binary.png', 255*binary)
```



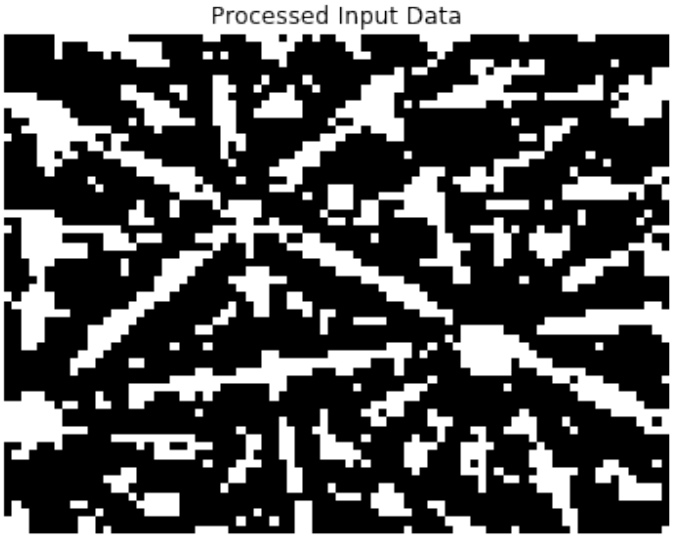
[5]:
```
# Classic straight-line Hough transform from binary data.
image=np.flipud(binary)
background=np.ones_like(image)

ang=np.deg2rad(np.linspace(-90,90,150))
h, theta, d = hough_line(image, theta=ang)
h_masked = np.ma.masked_equal(h ,0)
h_masked=h_masked/np.max(h_masked)

# butterfly mask convolution
# 9x9 mask Krieger-Lassen
k = np.array([[-10, -15, -22, -22, -22, -22, -22, -15, -10],
              [  1,  -6, -13, -22, -22, -22, -13,  -6,   1],
              [  3,   6,   4,  -3, -22,  -3,   4,   6,   3],
              [  3,  11,  19,  28,  42,  28,  19,  11,   3],
              [  3,  11,  27,  42,  42,  42,  27,  11,   3],
              [  3,  11,  19,  28,  42,  28,  19,  11,   3],
              [  3,   6,   4,  -3, -22,  -3,   4,   6,   3],
              [  1,  -6, -13, -22, -22, -22, -13,  -6,   1],
              [-10, -15, -22, -22, -22, -22, -22, -15, -10] ])

h = correlate(h_masked, k, mode='nearest') #.astype(np.int64)
a=(h-np.min(h))/(np.max(h)-np.min(h))

hpeaks=hough_line_peaks(a, theta, d,
        min_distance=10, min_angle=15,
        threshold=0.5*np.max(a), num_peaks=12)
```

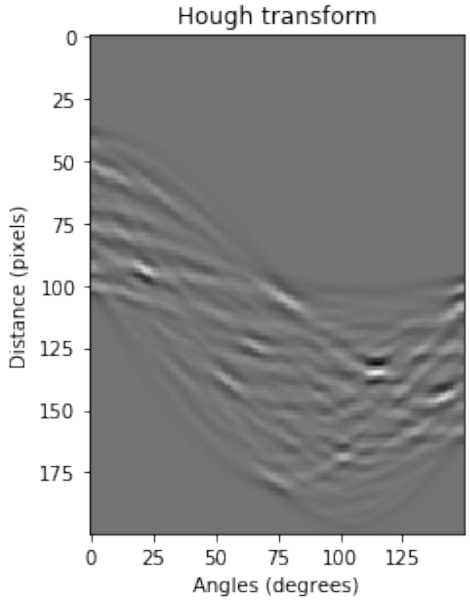
[6]:
```
# Generating figure 1.
fig, (ax0, ax1, ax2) = plt.subplots(3, 1, figsize=(6, 13))

ax0.imshow(image, cmap=cm.gray)
ax0.set_title('Processed Input Data')
ax0.set_axis_off()

#ax1.imshow(np.log(1 + h), extent=[np.rad2deg(theta[-1]),
```

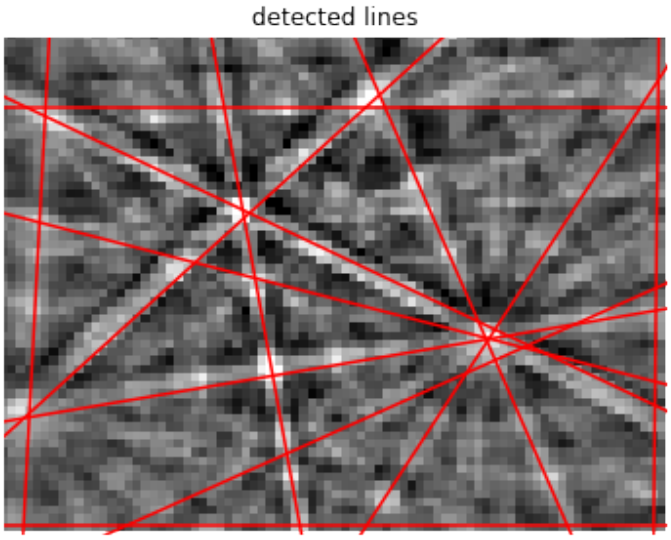
(continues on next page)

(continued from previous page)

```
#            np.rad2deg(theta[0]),
#           d[-1], d[0]], cmap=cm.gray, aspect=1)
ax1.imshow(a,cmap=cm.gray)
ax1.set_title('Hough transform')
ax1.set_xlabel('Angles (degrees)')
ax1.set_ylabel('Distance (pixels)')
ax1.axis('image')

ax2.imshow(np.flipud(pattern), cmap=cm.gray)
row1, col1 = pattern.shape
linecount=1
houghkl=[]
houghlines=[]
for _, angle, dist in zip(*hpeaks):
    y0 = (dist - 0 * np.cos(angle)) / np.sin(angle)
    y1 = (dist - col1 * np.cos(angle)) / np.sin(angle)
    ax2.plot((0, col1), (y0, y1), '-r')
    houghlines.append([linecount, dist, np.degrees(angle),
                       0, y0, col1,y1 ])
    #print(linecount, dist, np.rad2deg(angle),gpc)
    linecount=linecount+1
ax2.axis((0, col1, row1, 0))
ax2.set_title('detected lines')
ax2.set_axis_off()
plt.tight_layout()
plt.show()
#plt.savefig('hough_simple.png')
```


Processed Input Data


Hough transform
Distance (pixels)
0
25
50
75
100
125
150
175
0
25
50
75
100
125
Angles (degrees)


detected lines

Map Explorer

Data: **GaN_Dislocations_1.hdf5**

Basic Interactive EBSD Map Data Viewer Example. In a Jupyter Notebook, we can nicely mix "programming" (typing commands) and "interactivity" (doing things with the mouse etc in a user interface).

Note: The interactive features only work when running Jupyter notebook, not in the HTML documentation.

[17]:
```
ebsdpatternfile = "../../../../xcdskd_reference_data/GaN_Dislocations_1/hdf5/GaN_
↪Dislocations_1.hdf5"
```

[18]:
```
%load_ext autoreload
%autoreload 2
%matplotlib notebook
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[19]:
```
import matplotlib.pyplot as plt
import numpy as np

from pathlib import Path
from skimage.io import imread, imsave

from aloe.plots import plot_image
from aloe.io.patternproviders import EBSDDataManager
from aloe.io.patternproviders import H5PatternDataProviderCPRCTF as H5PatternDataProvider
from aloe.io.mapexplorer import MapExplorer
```

Connect the Data Sources

[20]:
```
hdf5_file = Path(ebsdpatternfile)
if hdf5_file.is_file():
    # file exists
    h5srv = H5PatternDataProvider(ebsdpatternfile, scan = 'Scan/', pattern_name =
↪'RawPatterns')
else:
    print('ERROR: file not found: ', ebsdpatternfile)
```

```
2600
```

The Base Map

We will select the spatial position on this map. Ideally, the signal should be from exactly the same position as the EBSP, e.g. a BSE map (or arBSE color maps) derived from the raw EBSD images themselves.

[21]:
```
print(h5srv.bse_total)
plot_image(h5srv.bse_total)
```

```
<HDF5 dataset "bse_map": shape (50, 52), type "<f8">
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

EBSD Data

The EBSDDataManager provides the common functionality for getting patterns and pattern calibration and orientation data etc from various different underlying data sources (e.g. HDF5 in different formats).

[22]:
```
ebsd = EBSDDataManager(h5srv)
```

[23]:
```
pattern_data = ebsd.get_pattern_data(30, 20)
print(pattern_data)
```

```
{'pattern': array([[ 7816.,  8016.,  8080., ...,  8464.,  8208.,  7992.],
       [ 7888.,  7896.,  8200., ...,  8008.,  8088.,  8184.],
       [ 7784.,  7880.,  7864., ...,  8272.,  8312.,  8088.],
       ...,
       [ 8808.,  8744.,  8928., ...,  9560.,  9576.,  8944.],
       [ 8832.,  8624.,  8720., ...,  9472.,  9488.,  9184.],
       [ 8592.,  8608.,  8800., ...,  9600.,  9160.,  9096.]], dtype=float32), 'euler_rad':
 ↪ array([ 0.,  0.,  0.], dtype=float32), 'xtilt_rad': 0.0, 'pc_brkr': array([ 0.5,  0.5, ␣
 ↪1. ], dtype=float32)}
```

[24]:
```
# plot the pattern from the returned data set "pattern_data" (dictionary)
plot_image(pattern_data['pattern'])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Interactive Map Explorer

now we should be ready to start the Map Explorer...

Left click in map to set new reference position. Select a Map Position with the Mouse and the pattern will update.

The last Slider adjusts the number of neighbors to be averaged. Averaging will not be done while real-time updating the mouse pointer (i.e. click the position to be averaged, this will move the blue circle, which indicates the reference position for averaging).

[25]:
```
ebsd.pattern_preprocessing = False # set to True for removing the background (very very␣
 ↪slow)

explorer = MapExplorer(ebsd, h5srv.bse_total)
explorer.init_plot()
explorer.init_widgets()
explorer.show_widgets()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
HBox(children=(VBox(children=(IntSlider(value=26, max=51), IntSlider(value=25, max=49),␣
 ↪IntSlider(value=0, con...
```

Using the Values which where selected in the Map Explorer

The interactive map explorer allows us to select a good map position or other parameters and then we can continue with these selected parameters in "programming" style work when necessary:

[26]:
```
# We can also continue to work with the values that the user has set interactively above:
iy = explorer.iy_slider.value
ix = explorer.ix_slider.value
ebsd.nap = explorer.neighbor_slider.value
print(iy, ix, ebsd.nap)
```

```
25 9 1
```

[27]:
```
current_pattern = ebsd.get_nap(ix, iy, invert=False)
print(' +/- Neighbors used for average: ', ebsd.nap)
plot_image(current_pattern)
```

```
 +/- Neighbors used for average:  1
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[28]:
```
ebsd.nap = 4 # set new value
current_pattern = ebsd.get_nap(ix, iy, invert=False)
print(' +/- Neighbors used for average: ', ebsd.nap)
plot_image(current_pattern)
```

```
 +/- Neighbors used for average:  4
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[29]:
```
# switch on background treament
ebsd.pattern_preprocessing = True
ebsd.nap = 2

current_pattern = ebsd.get_nap(ix, iy, invert=False)
print(' +/- Neighbors used for average: ', ebsd.nap)
plot_image(current_pattern)
```

```
 +/- Neighbors used for average:  2
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[30]:
```
# invert pattern (e.g. use for fitting inverted experimental patterns)
current_pattern = ebsd.get_nap(ix, iy, invert=True)
print(' +/- Neighbors used for average: ', ebsd.nap)
plot_image(current_pattern)
```

```
 +/- Neighbors used for average:  2
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```

3.2 BSE Imaging

SEM 2D BSE Imaging: Fe AstroEBSD example

This is a template file to demonstrate angle resolved BSE imaging via signal extraction from raw, saved, EBSD patterns.

A key advantage when using saved raw EBSD patterns is that we can partition the raw signal into a "Kikuchi pattern" and a "background" signal, which will give different types of contrasts. This partition is impossible when using conventional semiconductor diode detection etc. for angle-resolved signal BSE collection. We cannot distinguish between "Kikuchi signal" and "background signal" in the current from the diode (point-signal, 0D as compared to 2D signal on the phosphor screen).

Here, we use the AstroEBSD example data set from Ben Britton, which is publicly available as a HDF5 file from https://doi.org/10.5281/zenodo.1214828

[1]:
```
# directory with the HDF5 EBSD pattern file:
data_dir = "../../../xcdskd_reference_data/Ben_Astro/"

# filename of the HDF5 file:
hdf5_filename =  "Demo_Ben.h5"

# verbose output
output_verbose = False
```

Necessary packages

[2]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
# ignore divide by Zero
np.seterr(divide='ignore', invalid='ignore')

import time, sys, os
import h5py
import skimage.io

from aloe.plots import normalizeChannel, make2Dmap, get_vrange
from aloe.plots import plot_image, plot_SEM, plot_SEM_RGB
from aloe.image import arbse
from aloe.image.downsample import downsample
from aloe.image.kikufilter import process_ebsp
```

[3]:
```
# make result dirs and filenames
h5FileNameFull=os.path.abspath(data_dir + hdf5_filename)
h5FileName, h5FileExt = os.path.splitext(h5FileNameFull)
h5FilePath, h5File = os.path.split(h5FileNameFull)
timestr = time.strftime("%Y%m%d-%H%M%S")
h5ResultFile="arBSE_" + hdf5_filename


# close HDF5 file if still open
if 'f' in locals():
    f.close()
f=h5py.File(h5FileName+h5FileExt, "r")

ResultsDir = h5FilePath+"/arBSE_" + timestr + "/"
CurrentDir = os.getcwd()
#print('Current Directory: '+CurrentDir)
```

(continues on next page)

(continued from previous page)

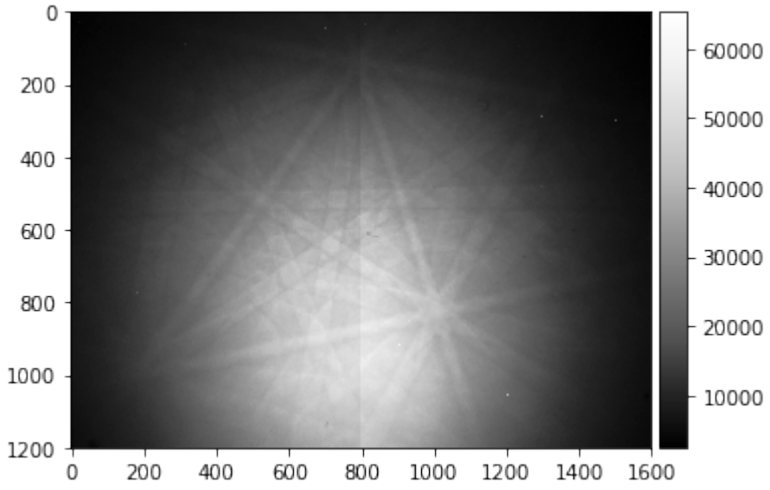
```
#print('Results Directory: '+ResultsDir)
if not os.path.isdir(ResultsDir):
    os.makedirs(ResultsDir)
os.chdir(ResultsDir)

if output_verbose:
    print('HDF5 full file name: ', h5FileNameFull)
    print('HDF5 File: ', h5FileName+h5FileExt)
    print('HDF5 Path: ', h5FilePath)
    print('Results Directory: ', ResultsDir)
    print('Results File: ', h5ResultFile)
```

[4]:
```
DataGroup="/Demo_Ben/EBSD/Data/"
HeaderGroup="/Demo_Ben/EBSD/Header/"
Patterns = f[DataGroup+"RawPatterns"]
#StaticBackground=f[DataGroup+"StaticBackground"]
XIndex =   f[DataGroup+"X BEAM"]
YIndex =   f[DataGroup+"Y BEAM"]
MapWidth = f[HeaderGroup+"NCOLS"].value
MapHeight= f[HeaderGroup+"NROWS"].value
PatternHeight=f[HeaderGroup+"PatternHeight"].value
PatternWidth =f[HeaderGroup+"PatternWidth"].value
print('Pattern Height: ', PatternHeight)
print('Pattern Width : ', PatternWidth)
PatternAspect=float(PatternWidth)/float(PatternHeight)
print('Pattern Aspect: '+str(PatternAspect))
print('Map Height: ', MapHeight)
print('Map Width : ', MapWidth)

step_map_microns = f[HeaderGroup+"XSTEP"].value
print('Map Step Size (microns): ', step_map_microns)
```

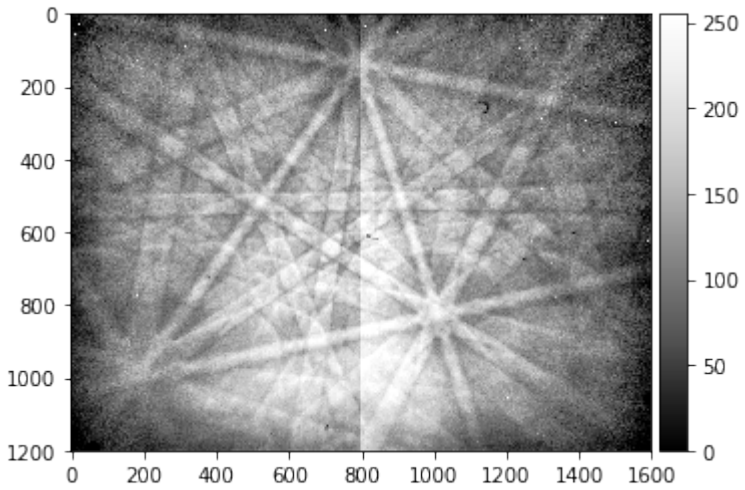
```
Pattern Height:  1200
Pattern Width :  1600
Pattern Aspect: 1.3333333333333333
Map Height:  83
Map Width :  110
Map Step Size (microns):  0.1497860178
```

Check Optional Pattern Processing

[5]:
```
ipattern = 215
raw_pattern=Patterns[ipattern,:,:]
plot_image(raw_pattern)
skimage.io.imsave('pattern_raw_' + str(ipattern) + '.tiff', raw_pattern, plugin='tifffile')
```



[6]:
```
# note the CCD halves and static background dust, hot pixels
processed_pattern = process_ebsp(raw_pattern, binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_' + str(ipattern) + '.tiff', processed_pattern,␣
↪plugin='tifffile')
```



[7]:
```
background_static = np.loadtxt(data_dir + "background_static.txt")
```

Making a static background from the EBSD map itself (can be useful with polycrystalline samples):

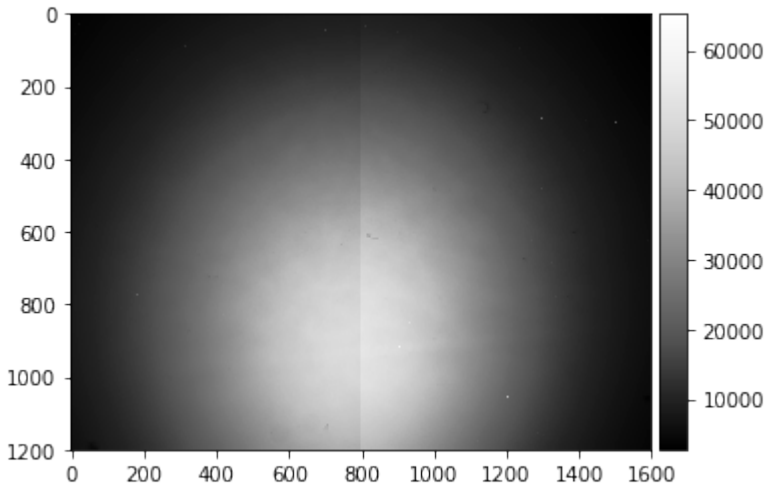
[8]:
```
calc_bg = False
if calc_bg:
    # avoid loading 35GB into RAM if you don't have 35GB RAM...
    # use incremental "updating average"
    # https://math.stackexchange.com/questions/106700/incremental-averageing
    tstart = time.time()
    npatterns = Patterns.shape[0]
    current_average = np.copy(Patterns[0]).astype(np.float64)
    for i in range(1, npatterns):
        current_average = current_average + (Patterns[i] - current_average)/i
        # update time info every 100 patterns
        if (i % 100 == 0):
            progress=100.0*(i+1)/Patterns.shape[0]
            tup = time.time()
```

(continues on next page)

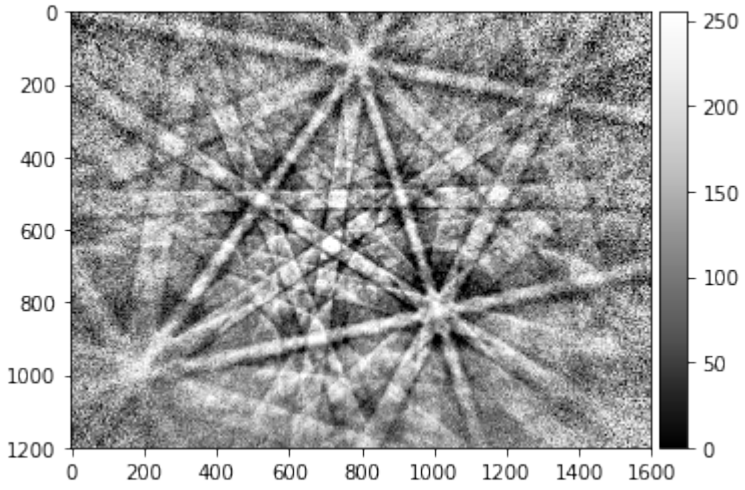
(continued from previous page)

```
            togo = (100.0-progress)*(tup-tstart)/(60.0*progress)
            sys.stdout.write("\rtotal map points:%5i current:%5i progress: %4.2f%% -> %6.
↪1f min to go" % (Patterns.shape[0],i+1,progress,togo))
            sys.stdout.flush()

    background_static = current_average
```

[9]:
```
plot_image(background_static)
#skimage.io.imsave('background_static.tiff', background_static, plugin='tifffile') # this␣
↪is 16bit only
np.savetxt('background_static.txt', background_static)
```



[10]:
```
# note the CCD halves and static background dust, hot pixels
processed_pattern = process_ebsp(raw_pattern, static_background=background_static,␣
↪binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_static_' + str(ipattern) + '.tiff', processed_pattern,
↪ plugin='tifffile')
```



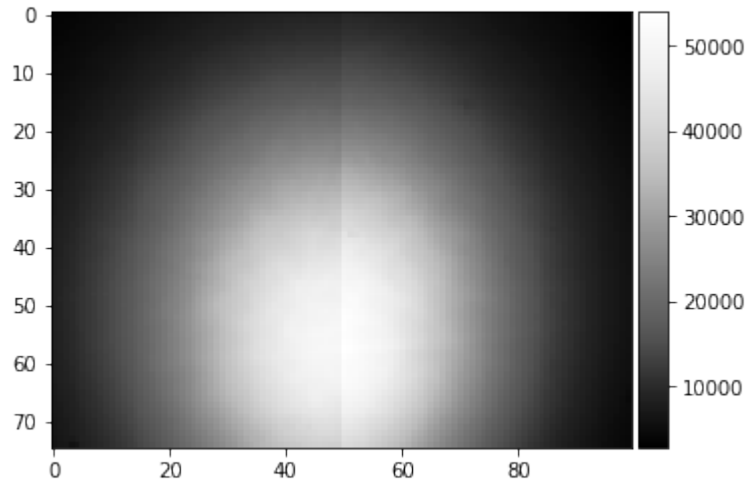
Specification of Image Pre-Processing Functions

[11]:
```
prebinning=16
background_static_binned = downsample(background_static, prebinning)
```

(continues on next page)

(continued from previous page)

```
def pipeline_process(pattern, prebinning=1, kikuchi=False):
    if prebinning>1:
        pattern = downsample(pattern, prebinning)
    if kikuchi:
        return process_ebsp(pattern, static_background=background_static_binned, binning=1)
    else:
        return pattern

def process_kikuchi(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=True)

def process_bin(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=False)

plot_image(background_static_binned)
print(background_static_binned.shape)
```

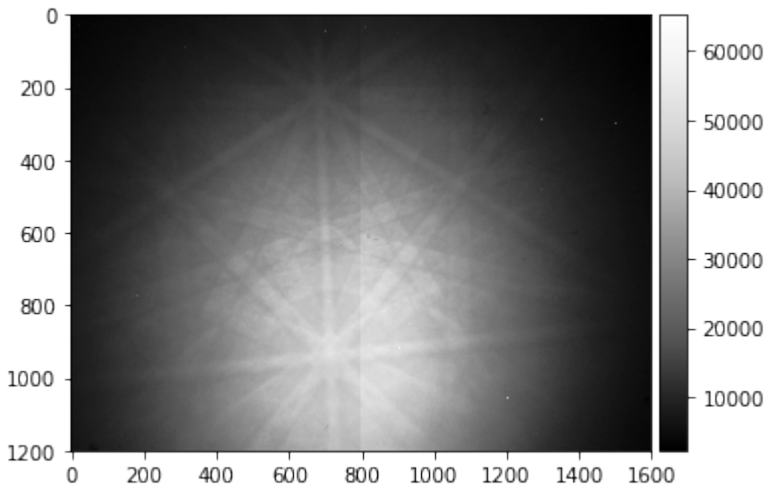


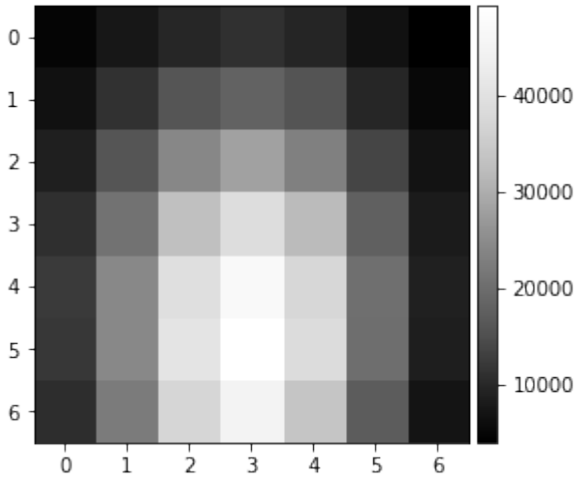

```
(75, 100)
```

vBSE Array

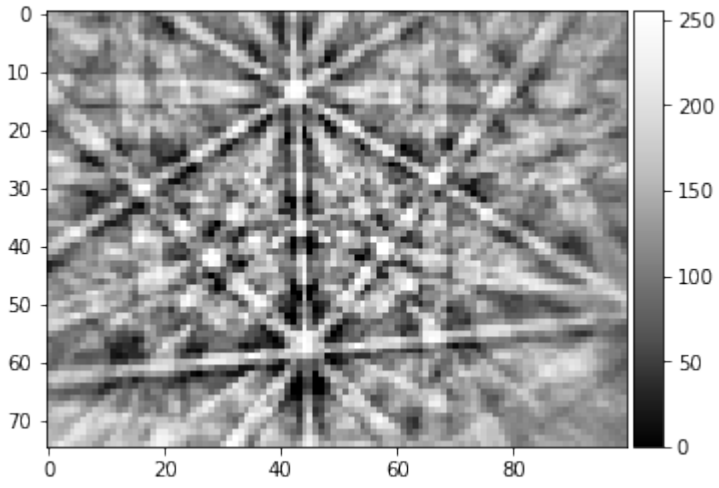
We convert the raw pattern into a 7x7 array of vBSE sensor intensities.

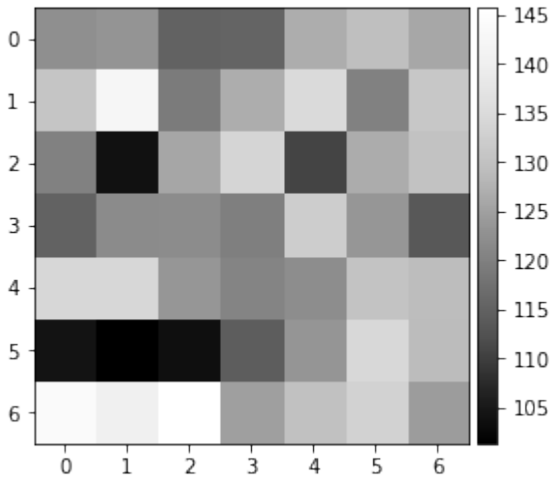
[12]:
```
pattern = pipeline_process(Patterns[1000], kikuchi=False)
vbse = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vbse)
```





[13]:
```
pattern = process_kikuchi(Patterns[1000])
vkiku = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vkiku)
```





vBSE Detector Signals: Calculation & Saving

This should take a few minutes, depending on your computer and file access speed.

Virtual BSE Imaging

Imaging the raw intensity in the respective area of the 2D detector (e.g. phosphor screen). Neglects gnomonic projection effect on intensities.

[14]:
```
# calculate the vBSE signals in 7x7 array
vbse_array = arbse.make_vbse_array(Patterns)

# make vBSE map of the total screen intensity
bse_total = np.sum(np.sum(vbse_array[:,:,:], axis=1), axis=1)
bse_map = make2Dmap(bse_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points: 9130 current: 9130 finished -> total calculation time :    5.4 min
```

[15]:
```
# save the results in an extra hdf5
print()
print(h5ResultFile)
try:
    h5f = h5py.File(h5ResultFile, 'a')
    h5f.create_dataset('vbse', data=vbse_array)
    h5f.create_dataset('/maps/bse_total', data=bse_map)
finally:
    h5f.close()
```

```

arBSE_Demo_Ben.h5
```

Virtual Orientation Imaging via Kikuchi Pattern Signals

If we process the raw images to obtain only the Kikuchi pattern, we have a modified 2D intensity which can be expected to show increased sensitivity to orientation effects (i.e. changes related to the Kikuchi bands). In a more advanced approach, we could select, for example, specific Kikuchi bands or zone axes to extract imaging signals.

[16]:
```
# calculate the vKikuchi signals from processed raw data
vkiku_array = arbse.make_vbse_array(Patterns, process=process_kikuchi)

# make vBSE map of the total screen intensity
kiku_total = np.sum(np.sum(vkiku_array[:,:,:], axis=1), axis=1)
kiku_map = make2Dmap(kiku_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points: 9130 current: 9130 finished -> total calculation time :    5.1 min
```

[17]:
```
# save the results in an extra hdf5
print()
print(h5ResultFile)
try:
    h5f = h5py.File(h5ResultFile, 'a')
    h5f.create_dataset('vkiku', data=vkiku_array)
    h5f.create_dataset('/maps/kiku_total', data=kiku_map)
finally:
    h5f.close()
```

```

arBSE_Demo_Ben.h5
```

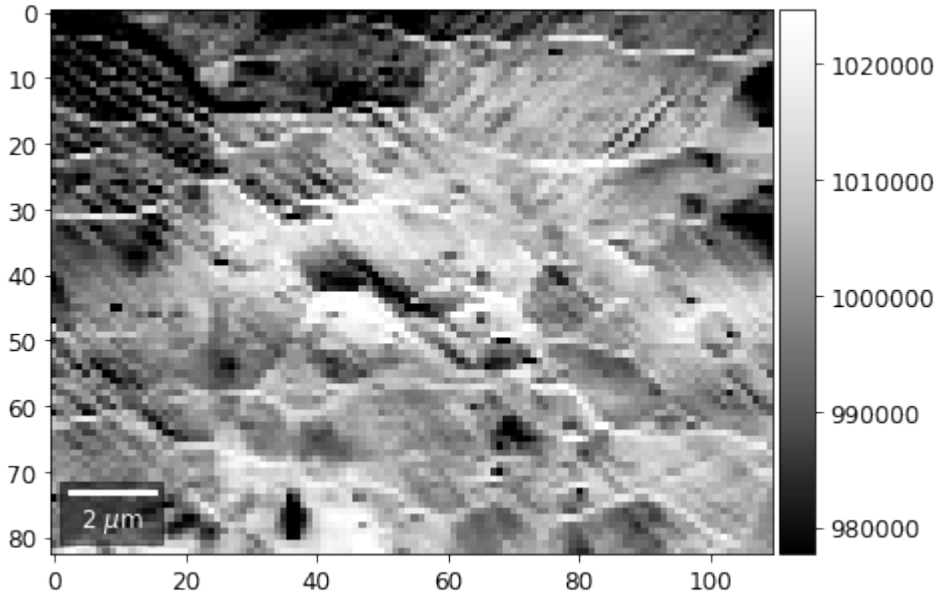
vBSE Signals: Plotting

[18]:
```
# plot from HDF5 results
h5f = h5py.File(h5ResultFile, 'r')
vFSD= h5f['vbse']
bse = h5f['/maps/bse_total']
kiku = h5f['/maps/kiku_total']
```

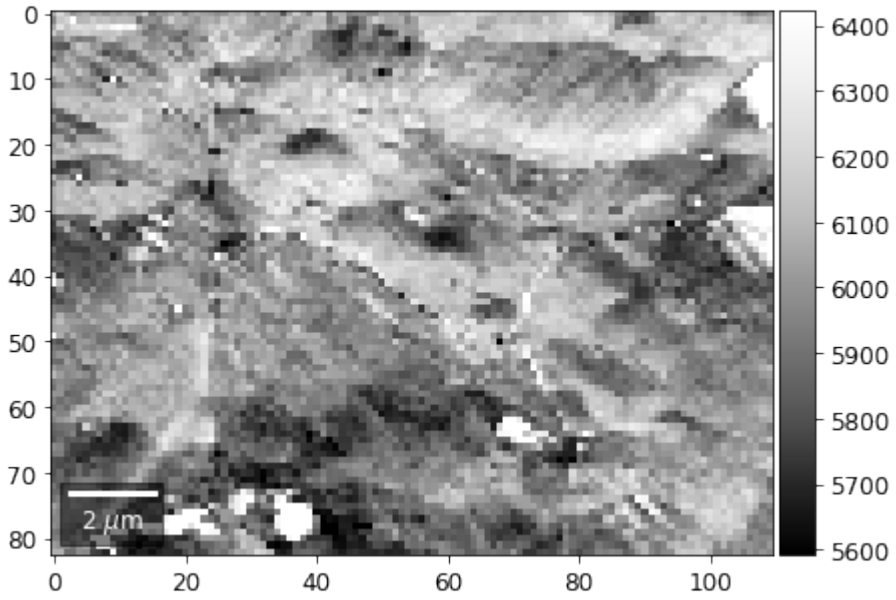
Total Signal on Screen

Total sum of the 7x7 arrays, for the raw pattern and the Kikuchi pattern at each map point:

[19]:
```
plot_SEM(bse, cmap='Greys_r', microns=step_map_microns)
```



[20]:
```
plot_SEM(kiku, cmap='Greys_r', microns=step_map_microns)
```



Intensity in Rows and Columns of the vBSE array

We can calculate additional images from the vBSE data set of 7x7 ROIs derived from the original patterns. As a first example, we plot the intensities of each of the 7 rows and then of each of the 7 columns:

Rows

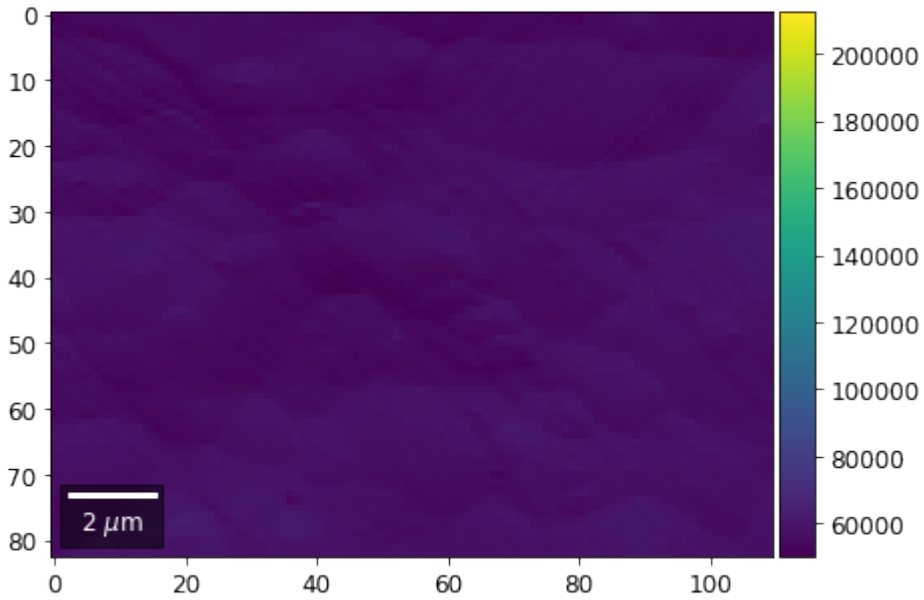
[21]:
```
# signal: sum of row
vmin=40000000
vmax=0

bse_rows = []

# (1) get full range for all images
for row in range(7):
    signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    minv, maxv = get_vrange(signal, stretch=3.0)
    if (minv<vmin):
        vmin=minv
    if (maxv>vmax):
        vmax=maxv

# (2) make plots with same range for comparisons of absolute BSE values
vrange=[vmin, vmax]
print('Range of Values: ', vrange)
#vrange=None
for row in range(7):
    signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    bse_rows.append(signal_map)
```

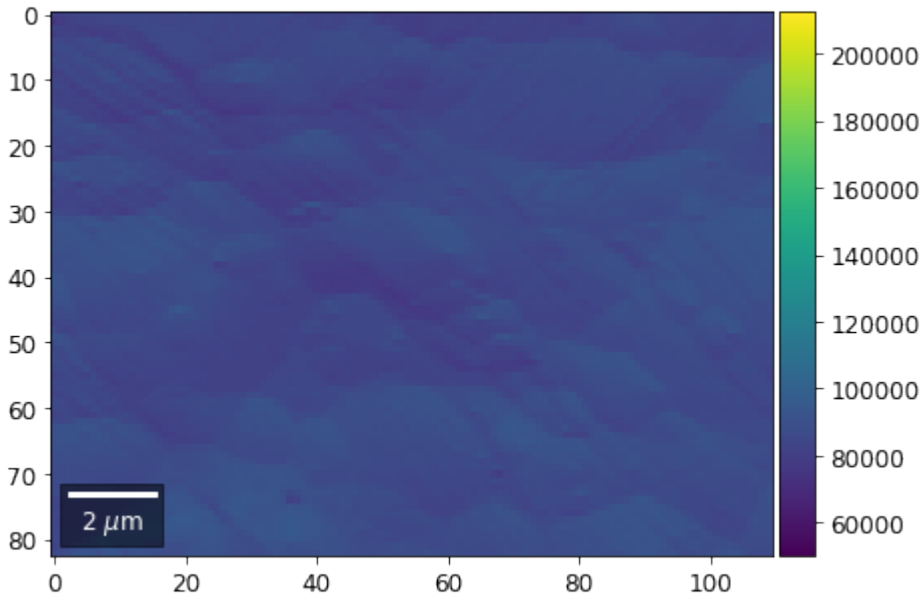
(continues on next page)

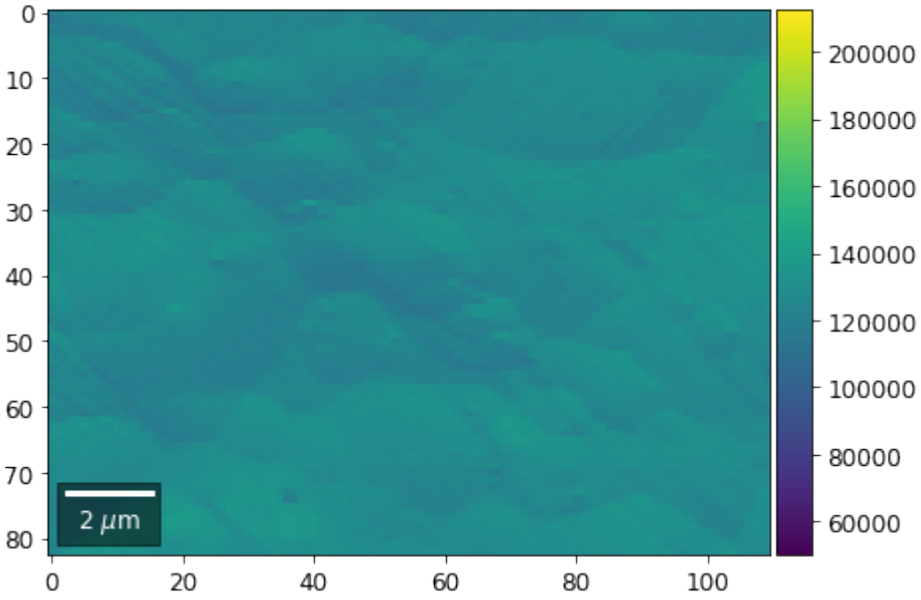
(continued from previous page)

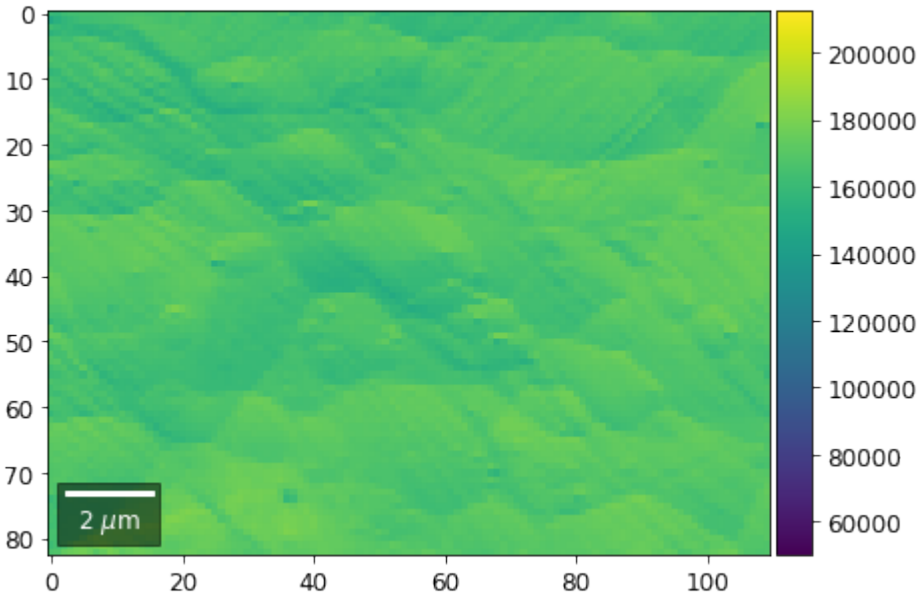
```
    plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_absolute_'+str(row),
             rot180=True, microns=step_map_microns)
```

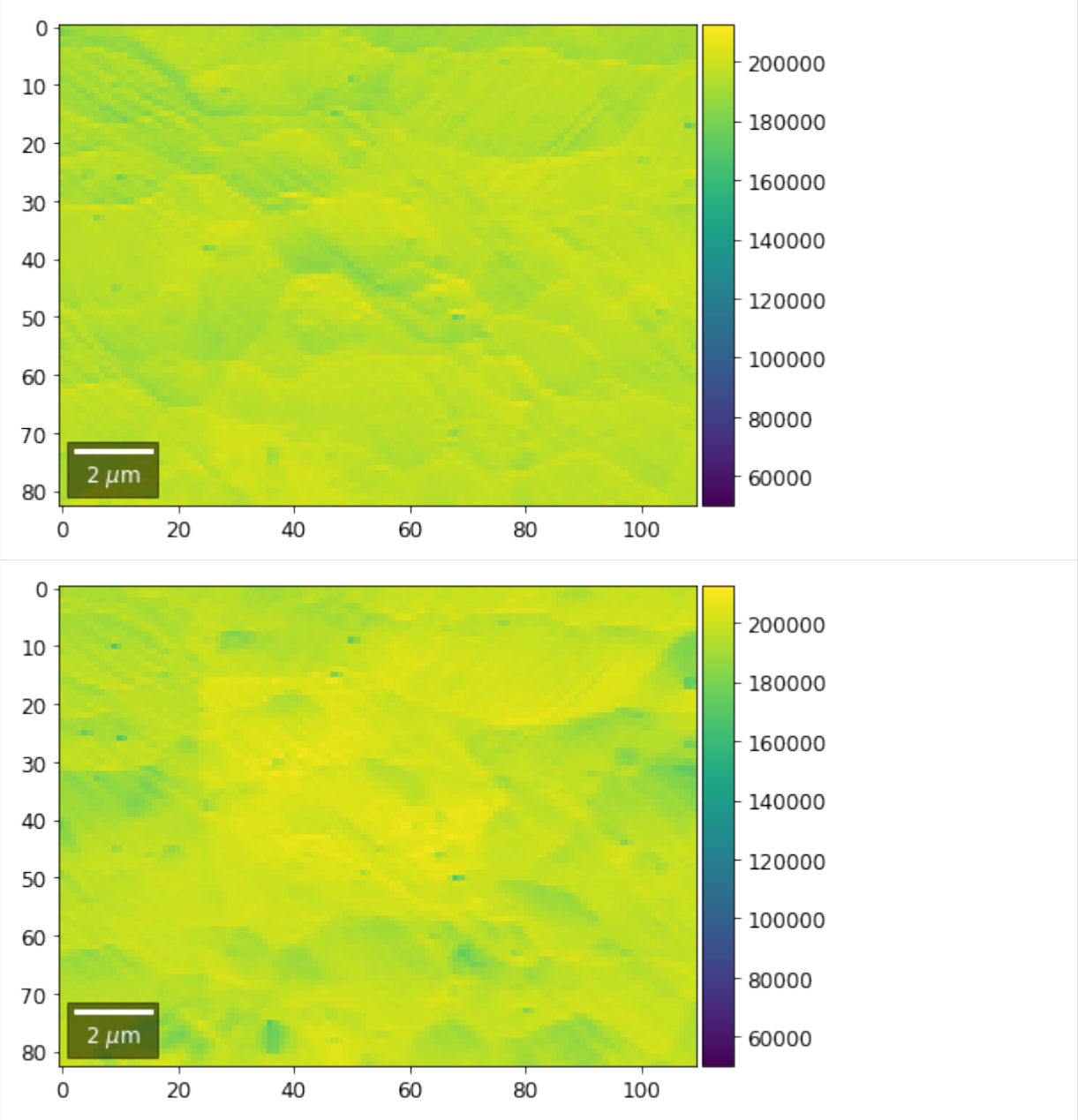
```
Range of Values:  [50122.067973821489, 212743.65306487455]
```

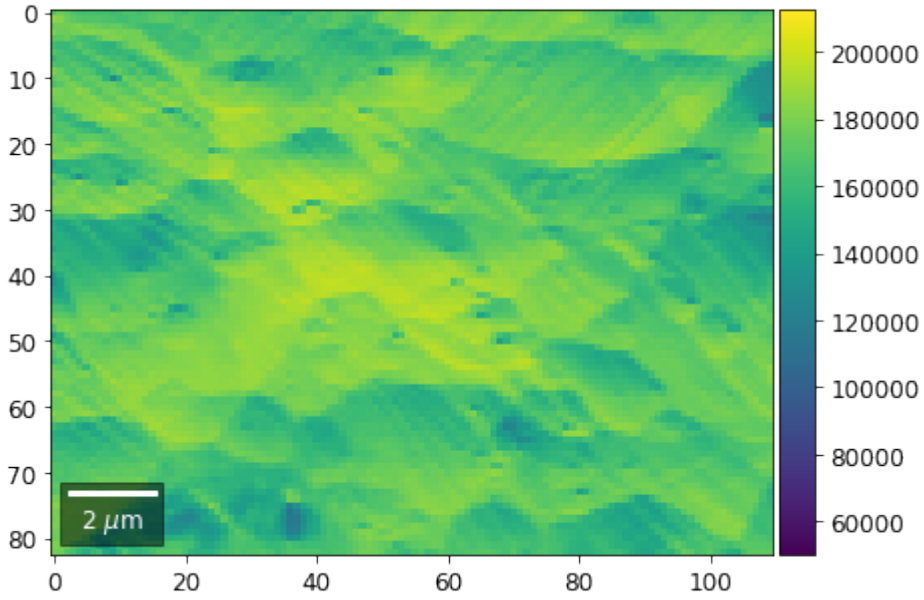




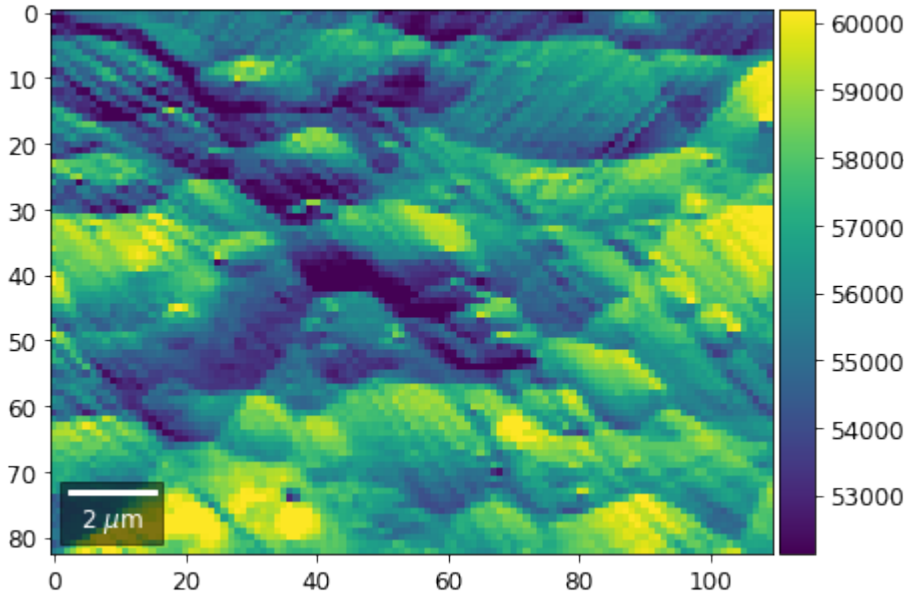

0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
200000
180000
160000
140000
120000
100000
80000
60000

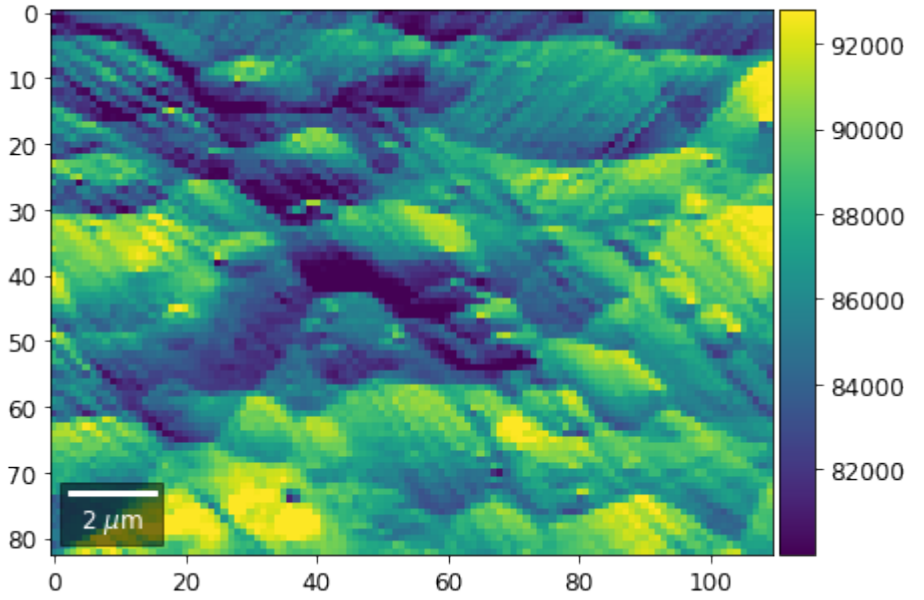

0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
200000
180000
160000
140000
120000
100000
80000
60000

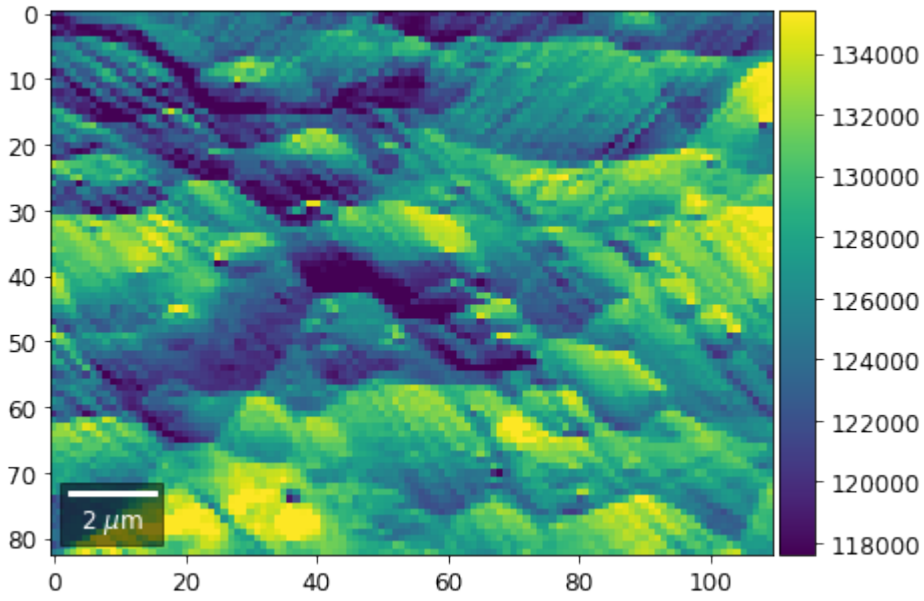

0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
200000
180000
160000
140000
120000
100000
80000
60000

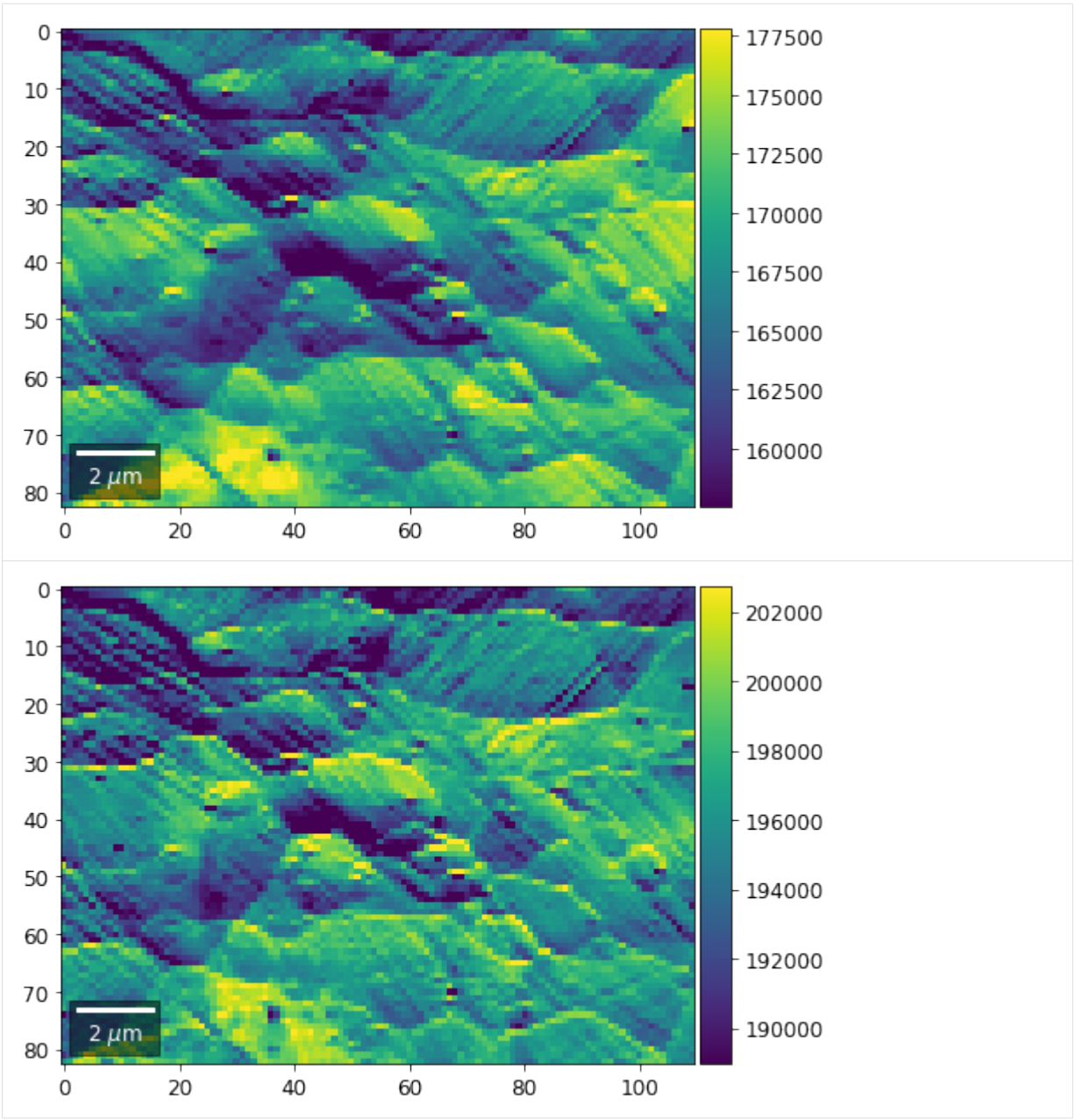


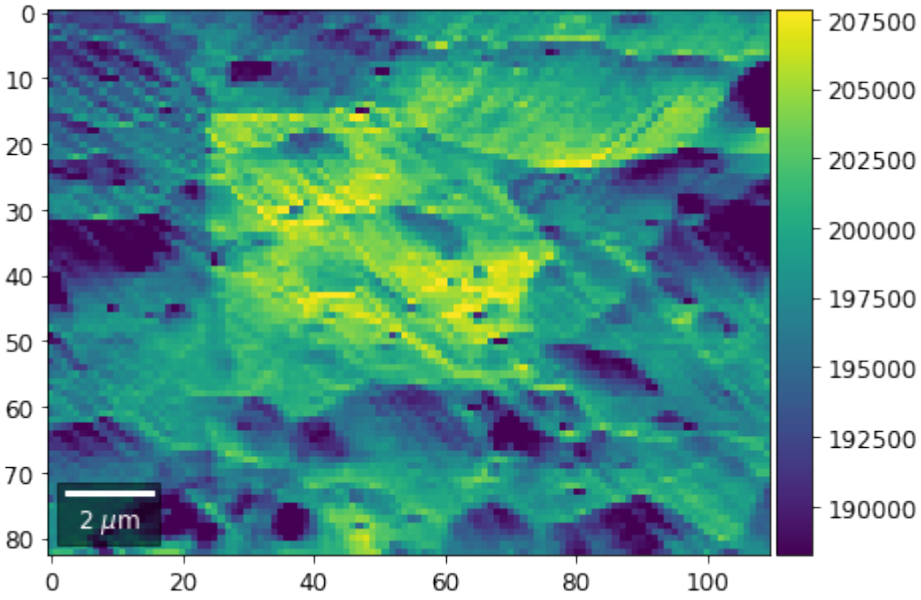
[22]:
```
# (3) make plots with  individual ranges for better contrast
vrange=None
for row in range(7):
    signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    bse_rows.append(signal_map)
    plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_individual_'+str(row),
             rot180=True, microns=step_map_microns)
```

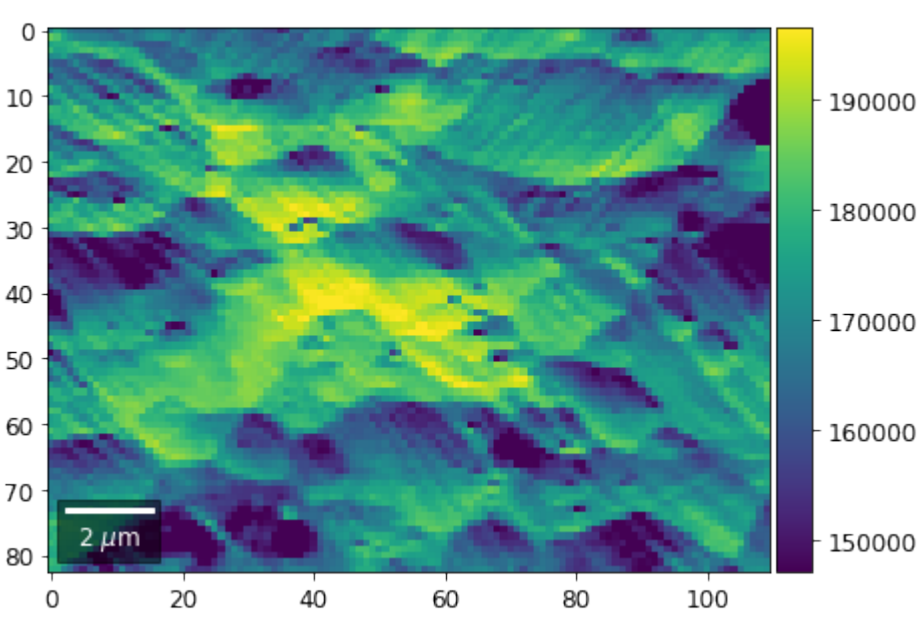



0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
92000
90000
88000
86000
84000
82000


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
134000
132000
130000
128000
126000
124000
122000
120000
118000


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
177500
175000
172500
170000
167500
165000
162500
160000
202000
200000
198000
196000
194000
192000
190000





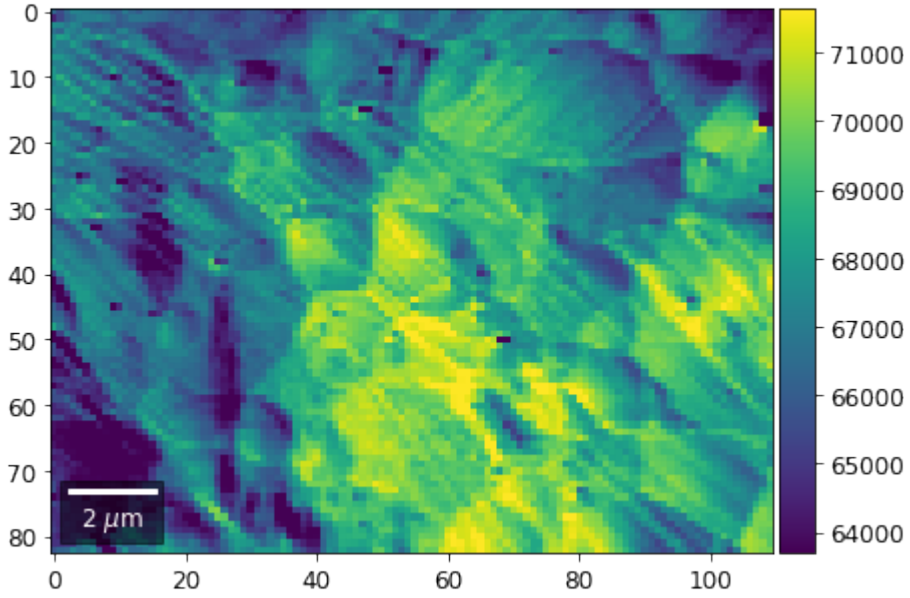
Columns

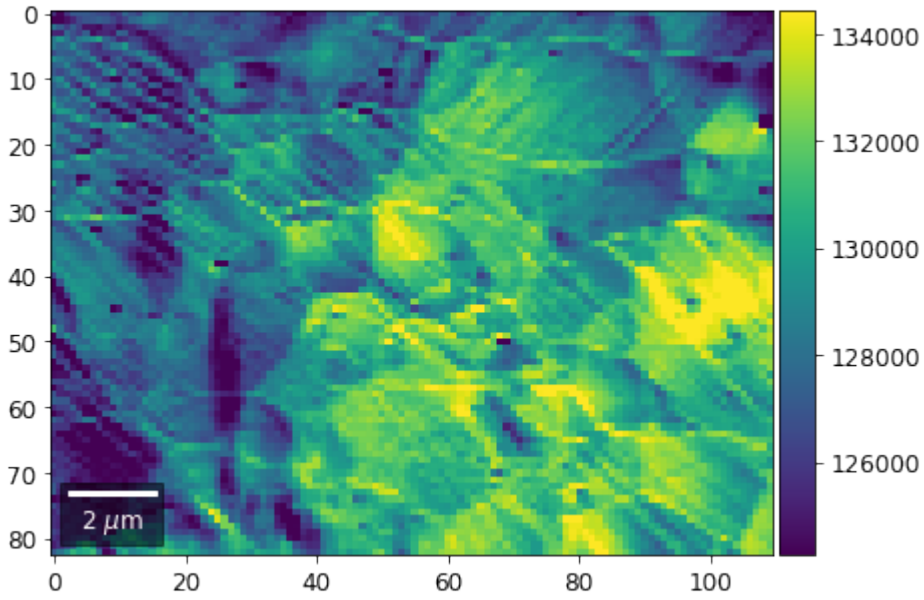
[23]:
```
# signal: sum of column
vmin=400000
vmax=0

bse_cols = []

# (1) get full range for all images
for col in range(7):
    signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    minv, maxv = get_vrange(signal)
    if (minv<vmin):
        vmin=minv
```

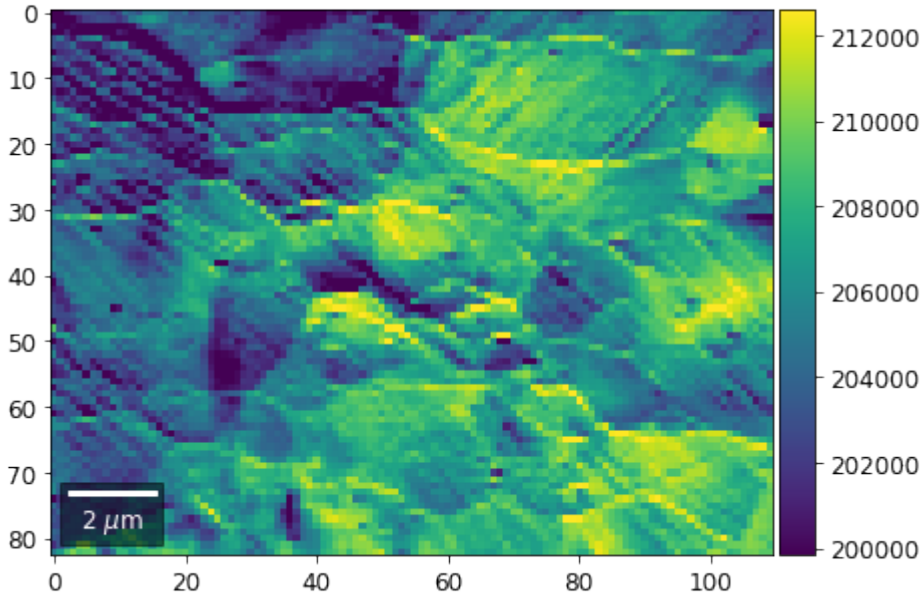
(continues on next page)

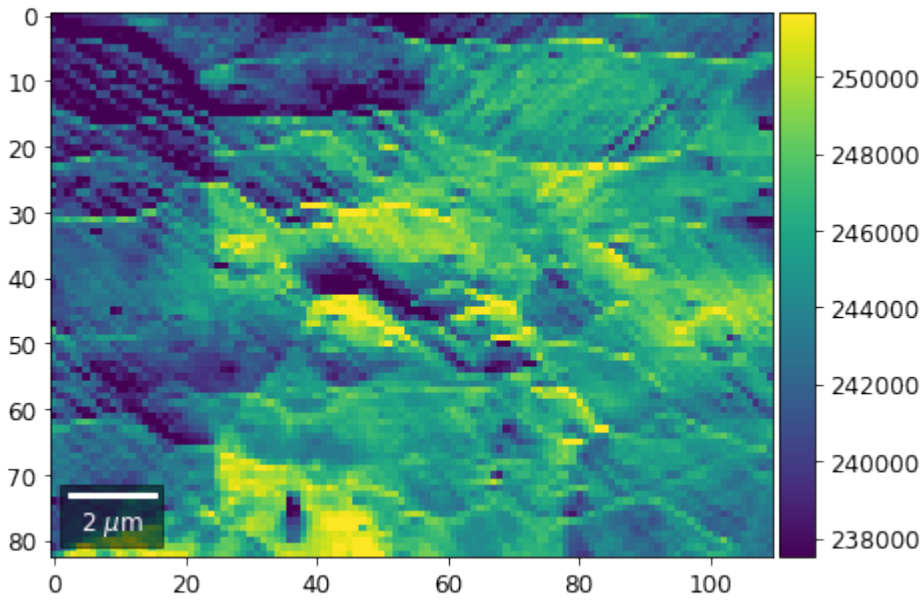
(continued from previous page)

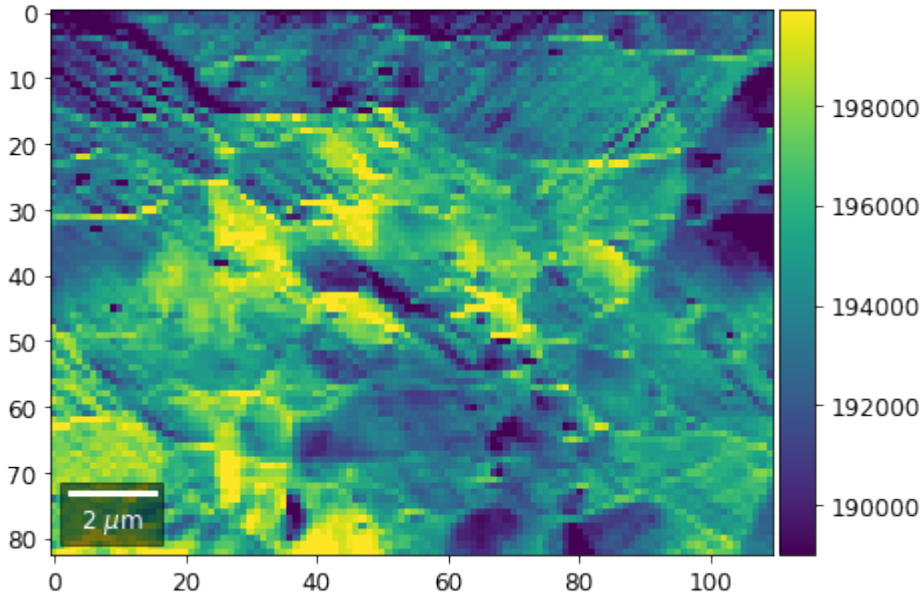
```
    if (maxv>vmax):
        vmax=maxv

# (2) make plots with same range for comparisons of absolute BSE values
#vrange=[vmin, vmax]
vrange=None # no fixed scale
for col in range(7):
    signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    bse_cols.append(signal_map)
    plot_SEM(signal_map, vrange=vrange, filename='vFSD_col_'+str(col),
             rot180=True, microns=step_map_microns)
```

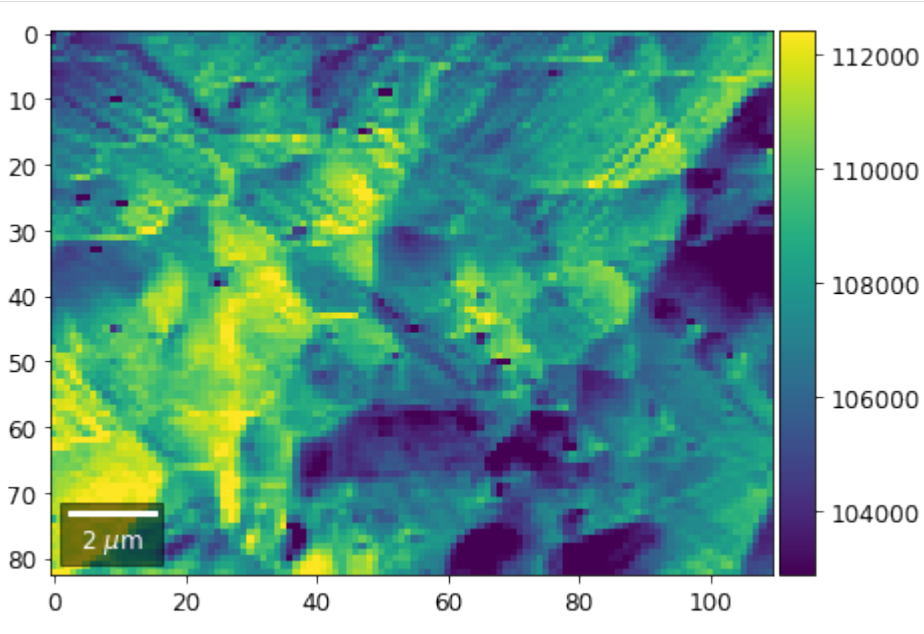


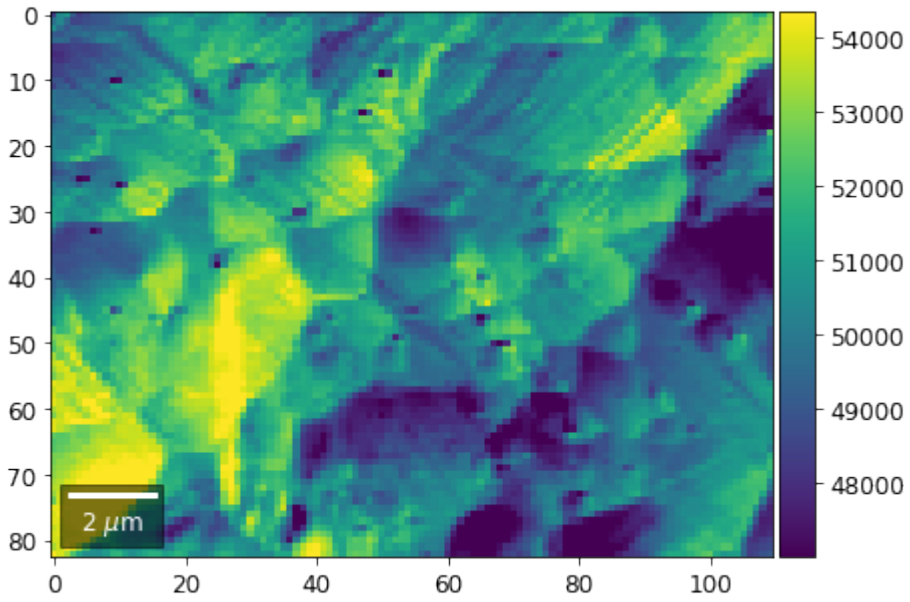



0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
212000
210000
208000
206000
204000
202000
200000


0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
250000
248000
246000
244000
242000
240000
238000


0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
198000
196000
194000
192000
190000

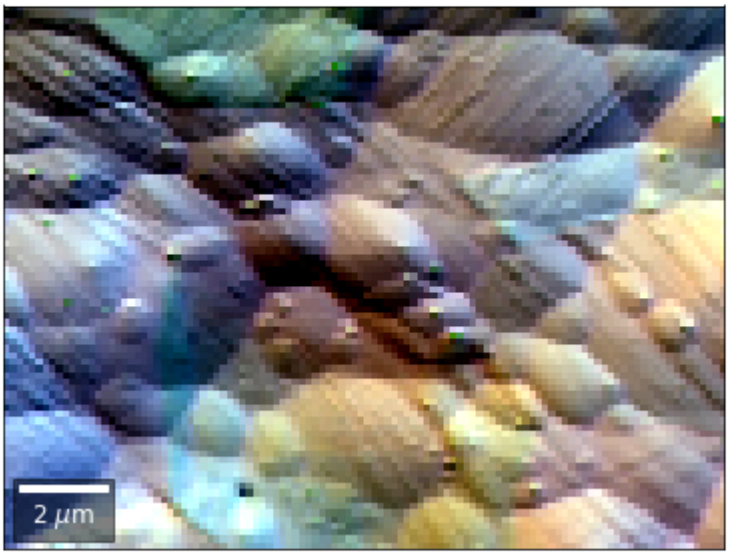

0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
112000
110000
108000
106000
104000

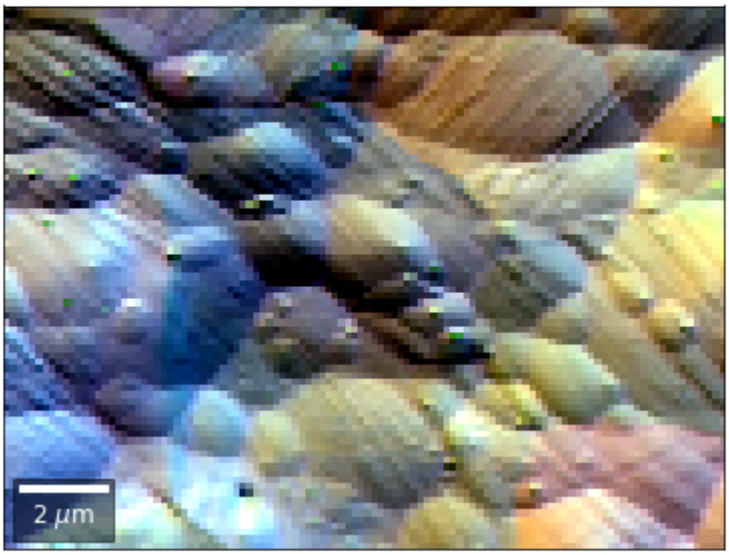


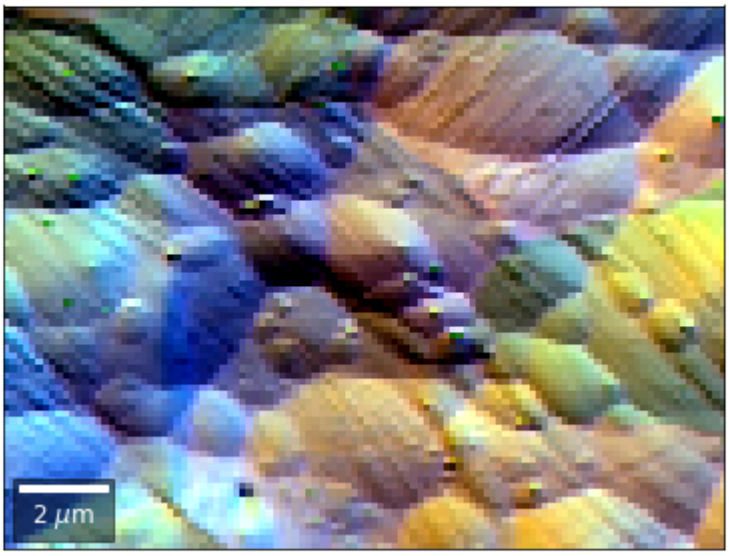
vBSE Color Imaging

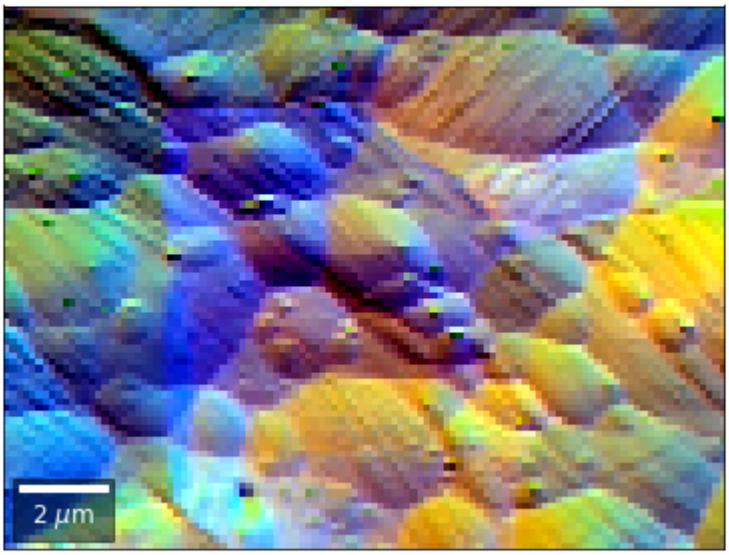
We can also form color images by assigning red, green, and blue channels to the left, middle, and right vBSE sensors of a row:

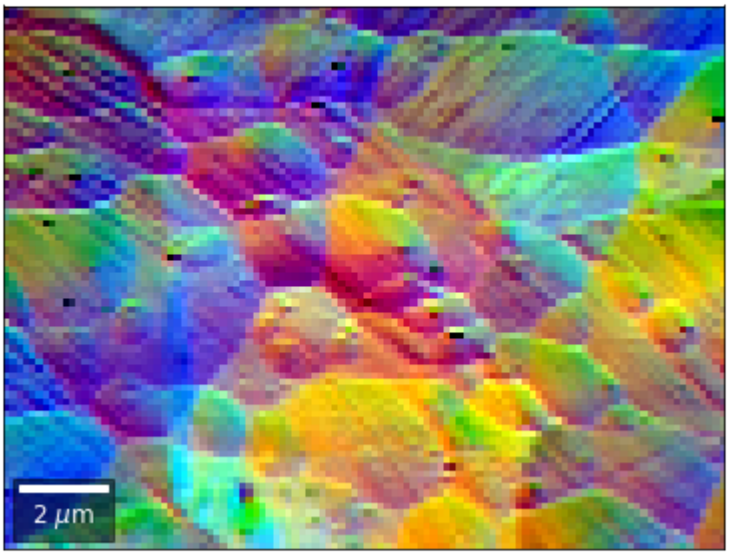
[24]:
```
# rgb direct
rgb_direct = []

for row in range(7):
    signal = vFSD[:,row,0]
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,3]
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,6]
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                     filename='vFSD_RGB_row_'+str(row),
                     rot180=False, microns=step_map_microns,
                     add_bright=0, contrast=0.8)

    rgb_direct.append(rgb)
```

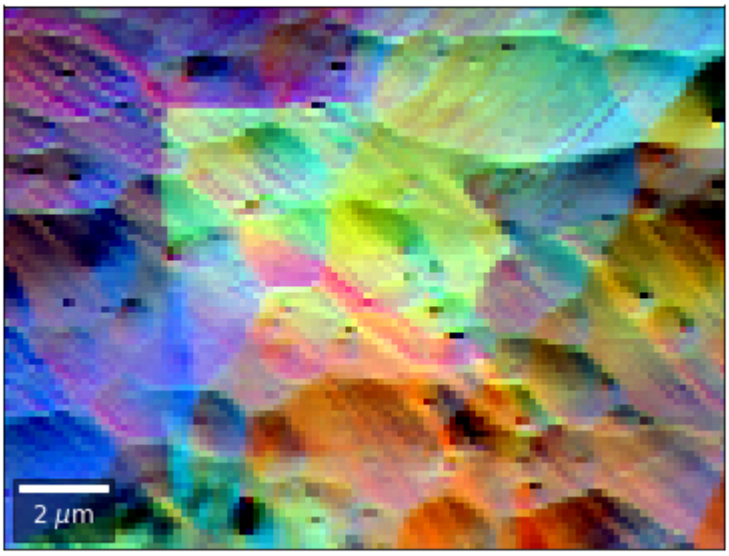

2 μm

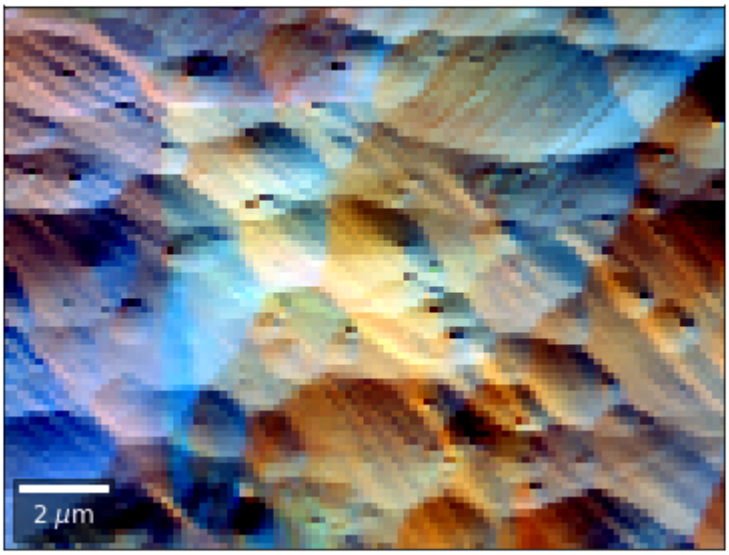

2 μm


2 μm


2 μm

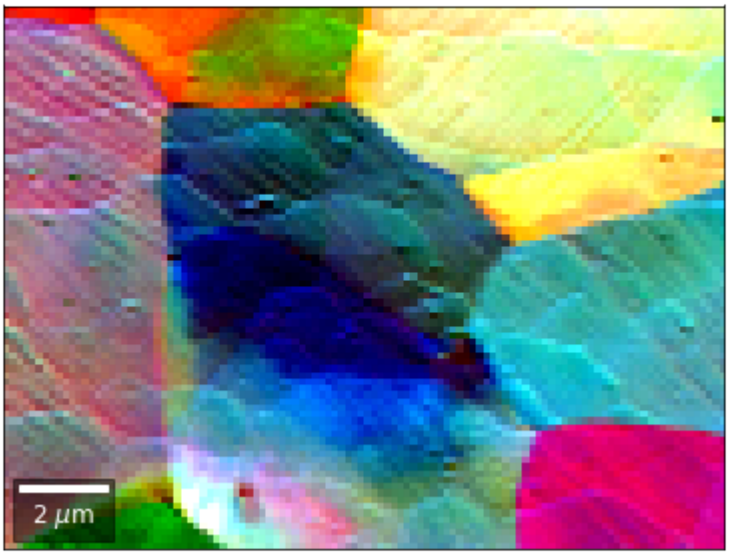

2 μm

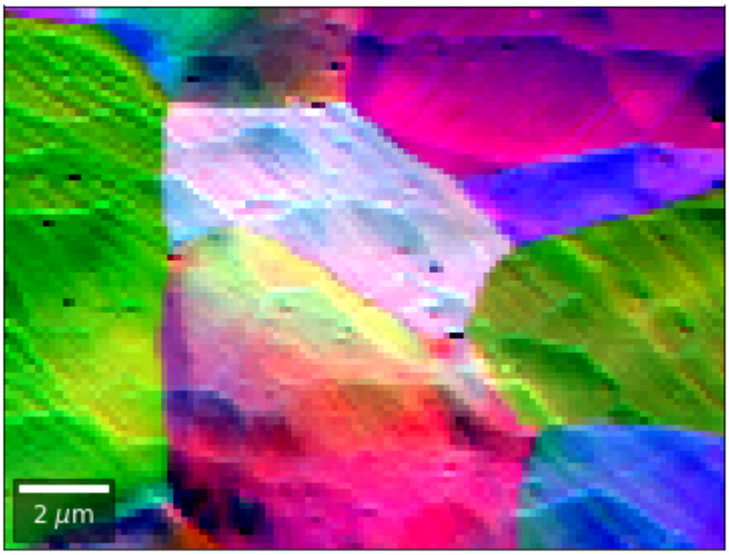

2 μm

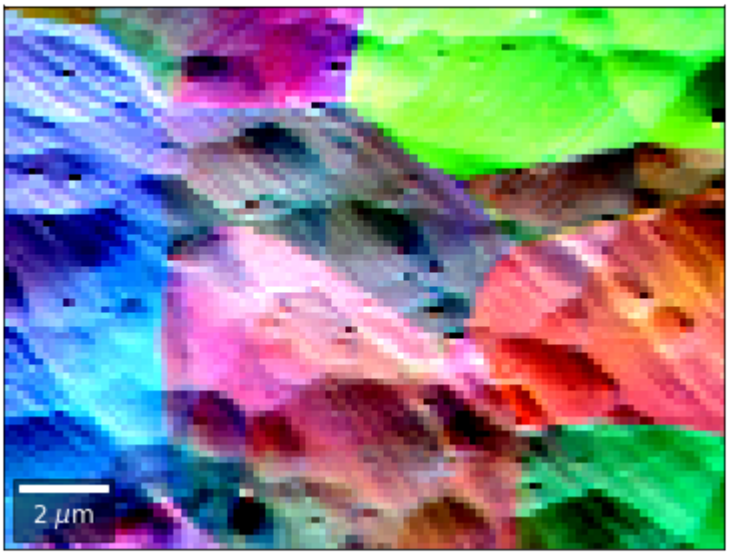


Differential color signals can be formed by calculating the relative changes to the ROI in the previous row:

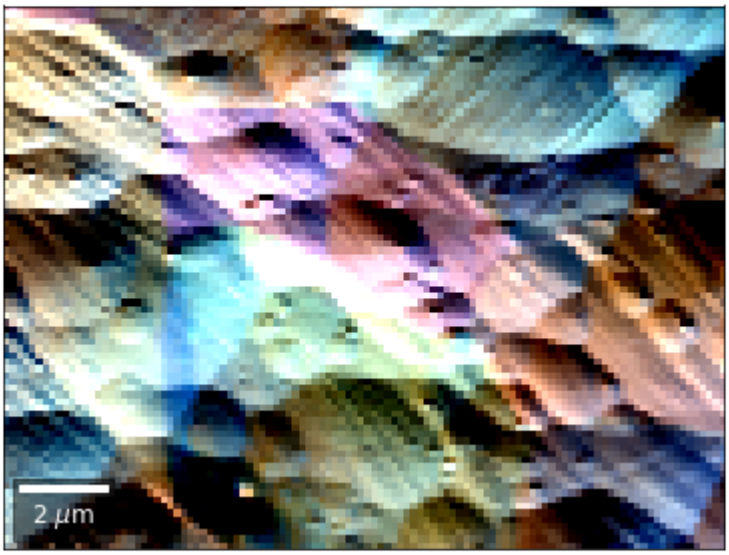
[25]:
```
# relative change to previous row
for row in range(1,7):
    drow = -1
    signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                     filename='vFSD_RGB_drow_'+str(row),
                     microns=step_map_microns,
                     rot180=False, add_bright=0, contrast=1.2)
```




2 μm

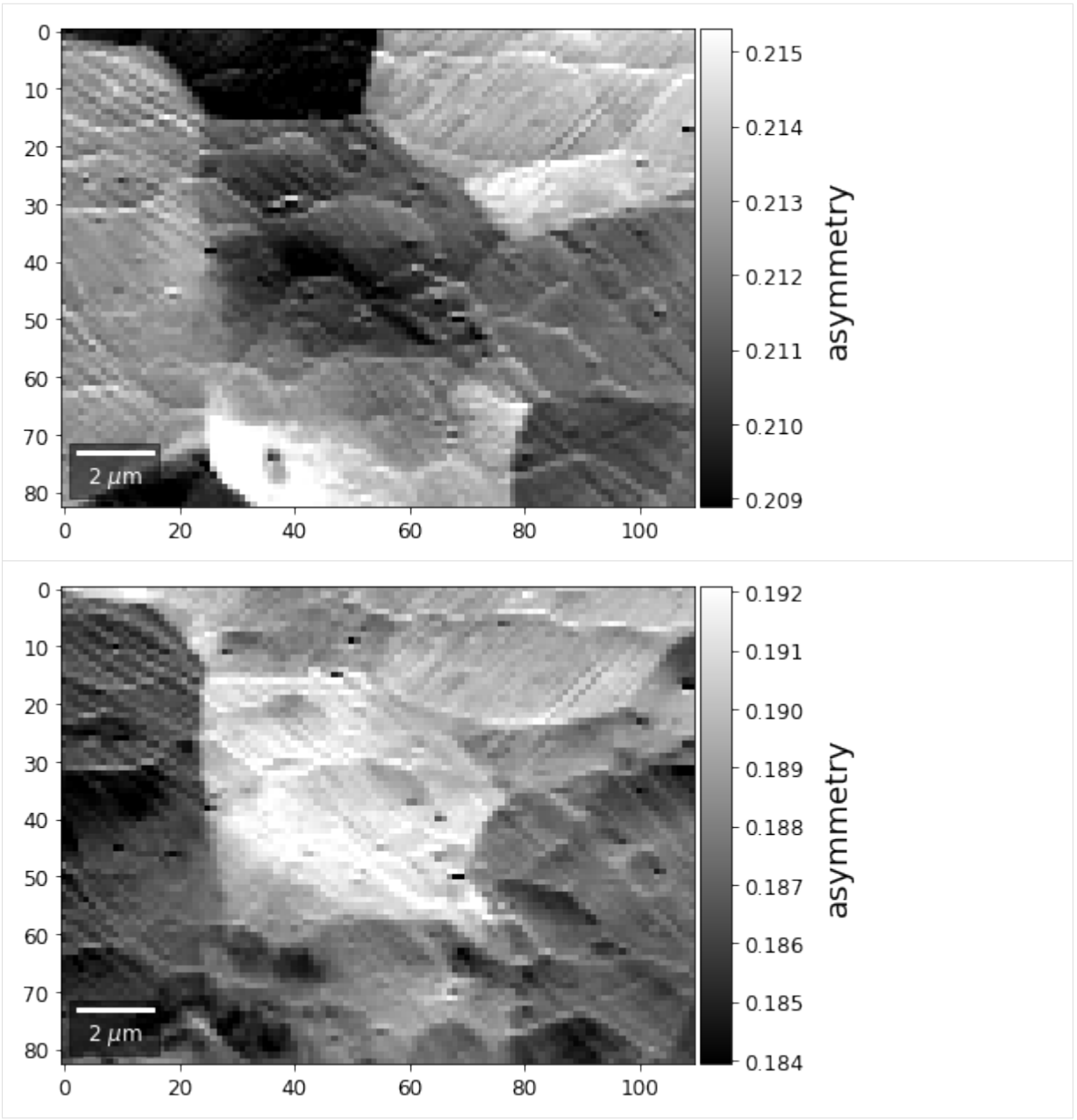

2 μm

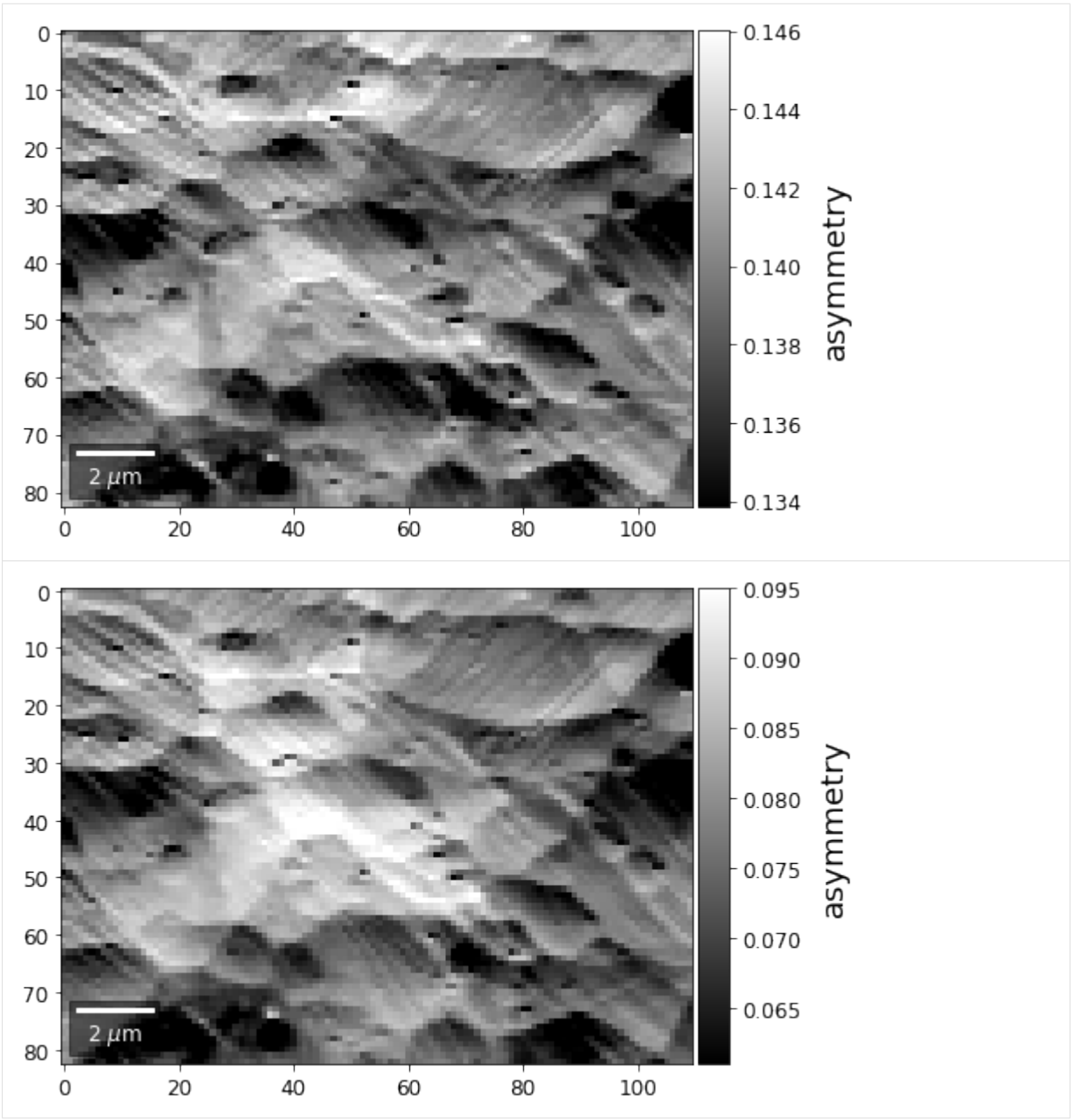

2 μm

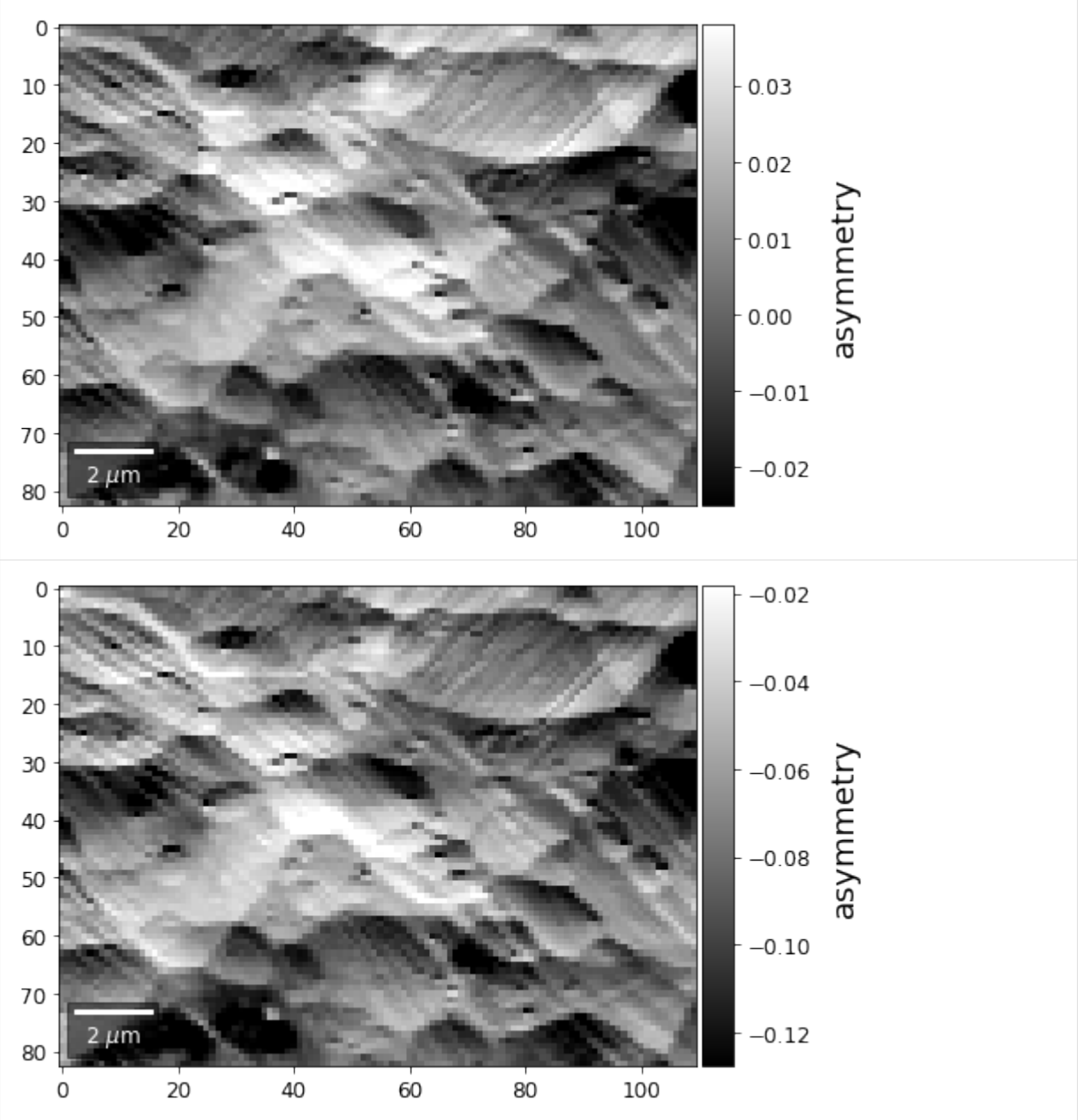


The normalized asymmetry between complete consecutive rows:

[26]:
```
# signal:asymmetry to previous row
for row in range(1,7):
    drow=-1
    signal = arbse.asy(np.sum(vFSD[:,row,:], axis=1) , np.sum(vFSD[:,row+drow,:], axis=1))
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    plot_SEM(signal_map, vrange=None, cmap='gray',
             colorbarlabel='asymmetry', microns=step_map_microns,
             filename='vFSD_row_asy_'+str(row))
```


0
10
20
30
40
50
60
70
80
2 μm
0
20
40
60
80
100
0.215
0.214
0.213
0.212
0.211
0.210
0.209
asymmetry
0
10
20
30
40
50
60
70
80
2 μm
0
20
40
60
80
100
0.192
0.191
0.190
0.189
0.188
0.187
0.186
0.185
0.184
asymmetry


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 μm
0.146
0.144
0.142
0.140
0.138
0.136
0.134
asymmetry
0.095
0.090
0.085
0.080
0.075
0.070
0.065
asymmetry



[27]:
```
h5f.close()
```

Center of Mass Imaging

We can interpret the 2D image intensity as a mass density on a plane. The statistical moments of the density distribution (mean, variance, ...) can be used as signal sources. In the example below, we use the image center of mass as a signal source.

COM of Raw Patterns

[28]:
```
# calculate the center-of-mass for each pattern, use binning for speed
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_bin)
```

```
total points: 9130 current: 9130 finished -> total calculation time :    5.1 min
```

[29]:
```
# save the results in an extra hdf5
print()
print(h5ResultFile)
try:
    h5f = h5py.File(h5ResultFile, 'a')
    h5f.create_dataset('/COM/COMxp_vbse', data=COMxp)
    h5f.create_dataset('/COM/COMyp_vbse', data=COMyp)
finally:
    h5f.close()
```

```

arBSE_Demo_Ben.h5
```

COM of Kikuchi Patterns

This should be seen with caution, as the background removal process is never perfect and will tend to leave some residual intensity, so that the Kikuchi COM is correlated with the raw pattern COM (which is dominated by the smooth background intensity).

[30]:
```
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_kikuchi)
```

```
total points: 9130 current: 9130 finished -> total calculation time :    5.1 min
```

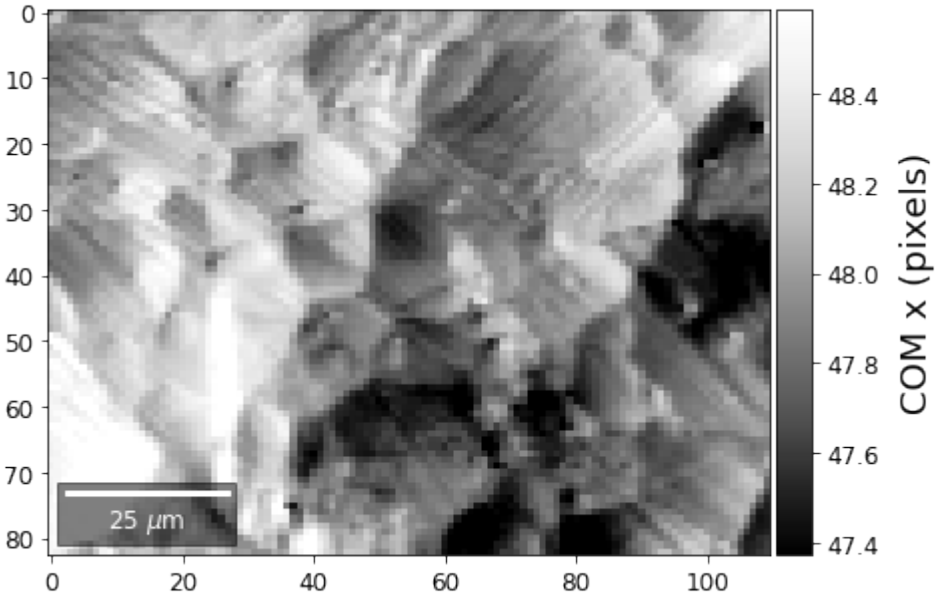
[31]:
```
# save the results in an extra hdf5
print()
print(h5ResultFile)
try:
    h5f = h5py.File(h5ResultFile, 'a')
    h5f.create_dataset('/COM/COMxp_kiku', data=COMxp)
    h5f.create_dataset('/COM/COMyp_kiku', data=COMyp)
finally:
    h5f.close()
```

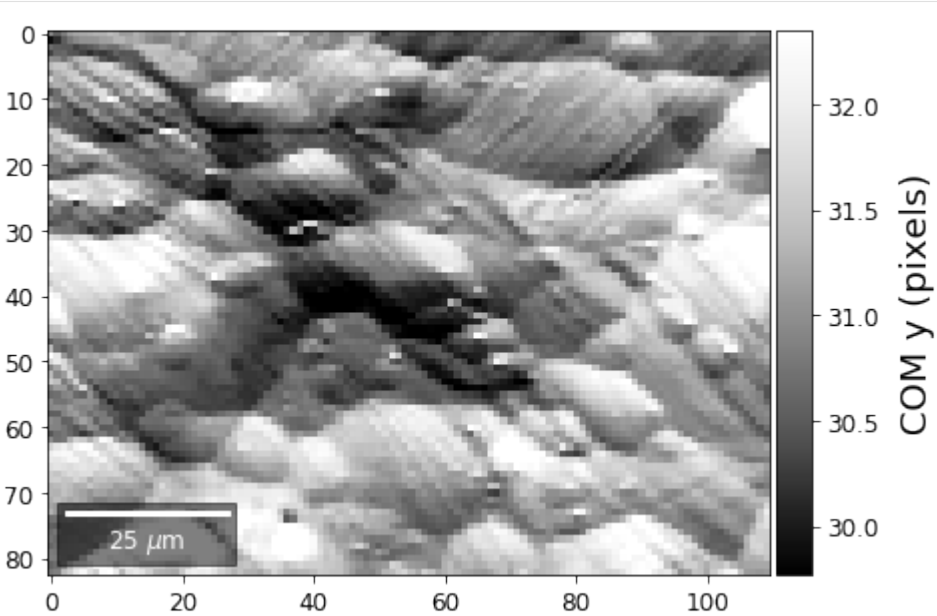
```

arBSE_Demo_Ben.h5
```

[32]:
```
h5f = h5py.File(h5ResultFile, 'r')
COMxp = h5f['/COM/COMxp_vbse']
COMyp = h5f['/COM/COMyp_vbse']
```

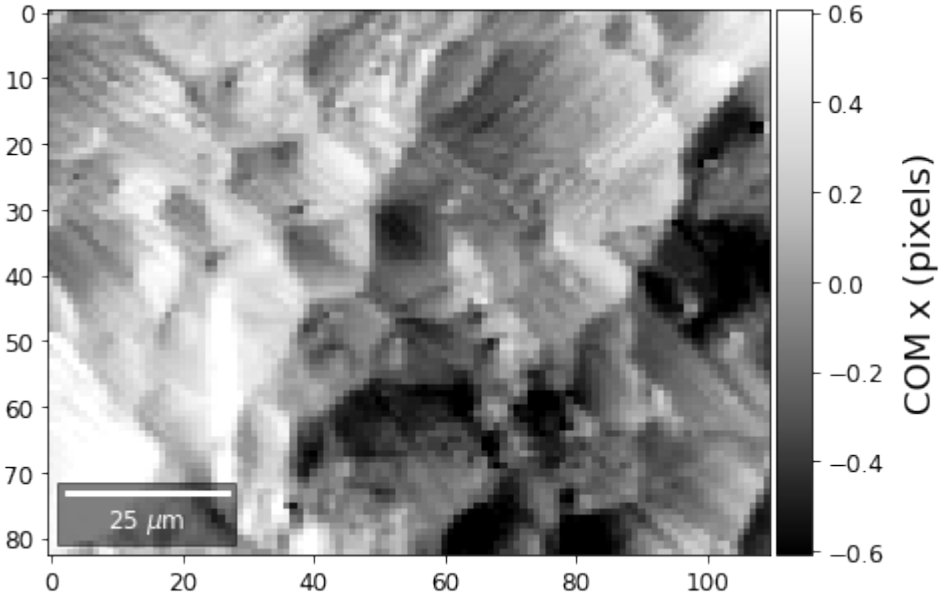
First, we calculate where the COMs are in x,y in pixels in the patterns:

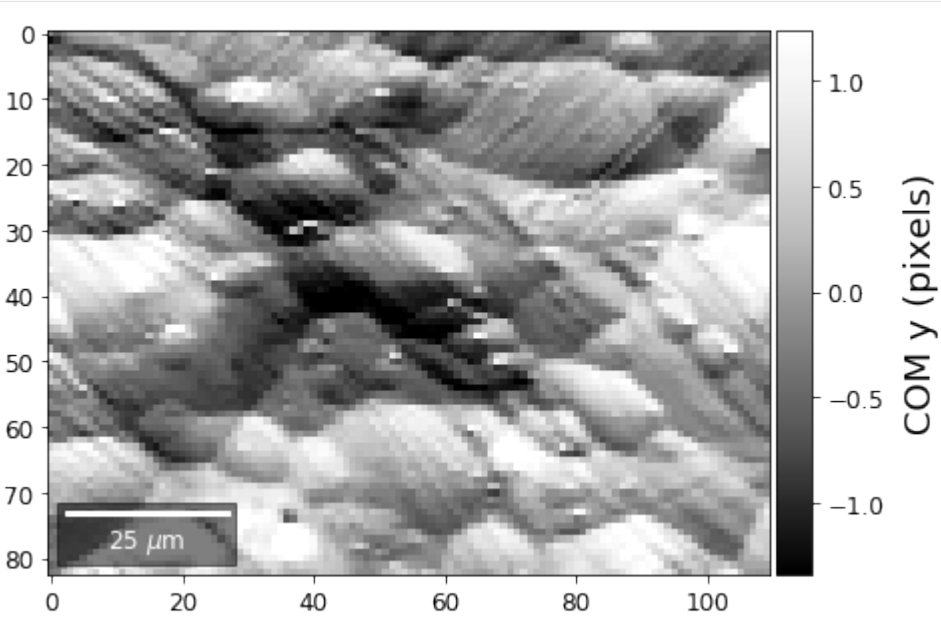
[33]:
```
comx_map0=make2Dmap(COMxp[:],XIndex,YIndex,MapHeight,MapWidth)
comy_map0=make2Dmap(COMyp[:],XIndex,YIndex,MapHeight,MapWidth)

plot_SEM(comx_map0, colorbarlabel='COM x (pixels)', cmap='Greys_r')
plot_SEM(comy_map0, colorbarlabel='COM y (pixels)', cmap='Greys_r')
```





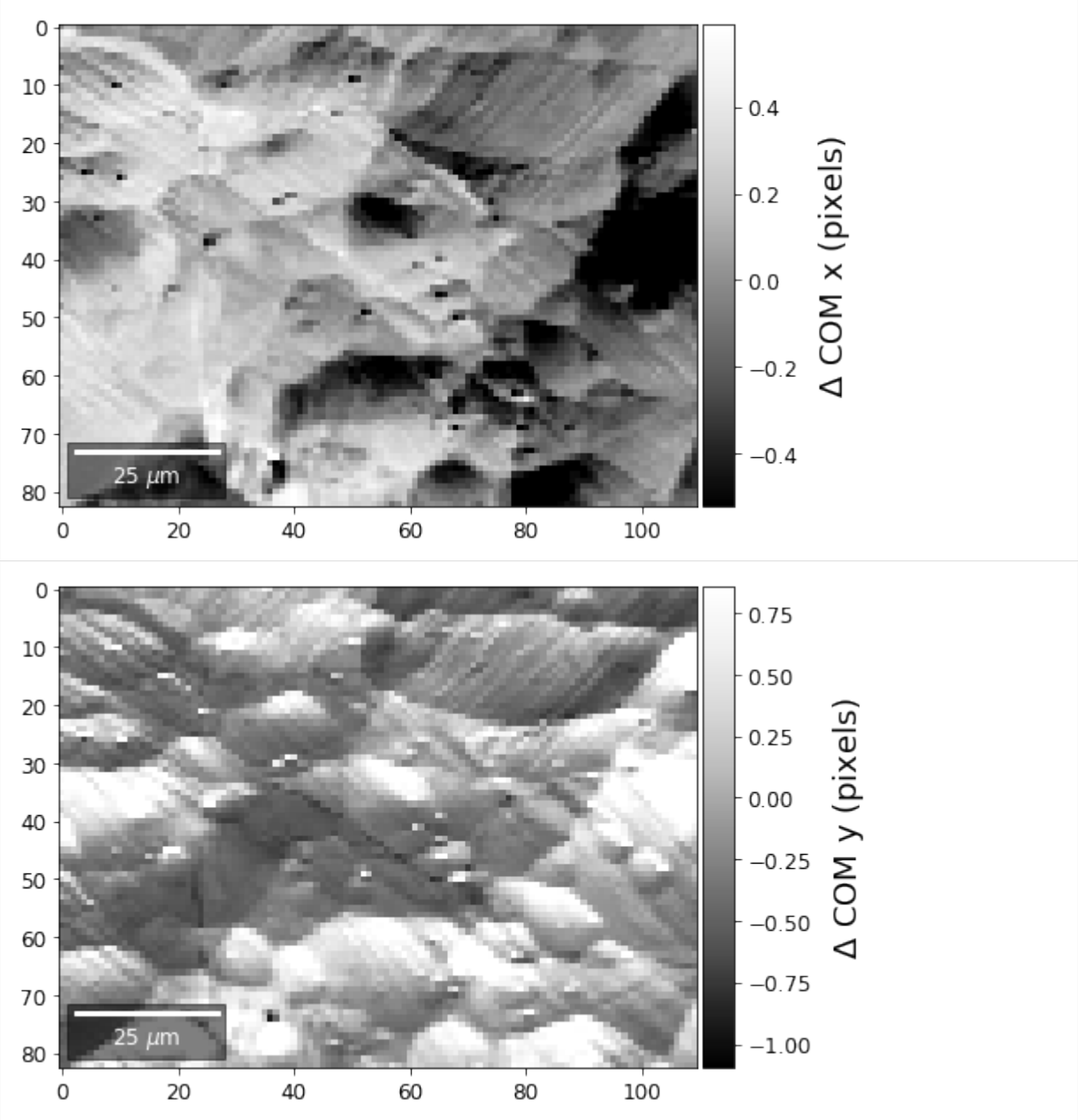
[34]:
```
meanx=np.mean(COMxp)
meany=np.mean(COMyp)

comx_map=make2Dmap(COMxp[:]-meanx,XIndex,YIndex,MapHeight,MapWidth)
comy_map=make2Dmap(COMyp[:]-meany,XIndex,YIndex,MapHeight,MapWidth)

plot_SEM(comx_map, colorbarlabel='COM x (pixels)', filename='comx', cmap='Greys_r')
plot_SEM(comy_map, colorbarlabel='COM y (pixels)', filename='comy', cmap='Greys_r')
```





[35]:
```
COMxp = h5f['/COM/COMxp_kiku']
COMyp = h5f['/COM/COMyp_kiku']
```

[36]:
```
meanx = np.mean(COMxp)
meany = np.mean(COMyp)

comx_map = make2Dmap(COMxp[:] - meanx, XIndex, YIndex, MapHeight, MapWidth)
comy_map = make2Dmap(COMyp[:] - meany, XIndex, YIndex, MapHeight, MapWidth)

plot_SEM(comx_map, colorbarlabel='$\Delta$ COM x (pixels)', filename='comx', cmap='Greys_r
↪')
plot_SEM(comy_map, colorbarlabel='$\Delta$ COM y (pixels)', filename='comy', cmap='Greys_r
↪')
```



Fourier Transform Based Imaging

With the help of the Fast Fourier Transform (FFT), we can extract information on spatial frequencies (wave vectors) from an image. This can be used to derive imaging signals which are based on ranges of specific wave vectors present in the Kikuchi pattern.

In the example shown below, we determine the intensity in the four quadrants of the FFT spectrum magnitude. The points corresponding to the low spatial frequencies (large spatial extensions) are removed to suppress the influence of the background signal.

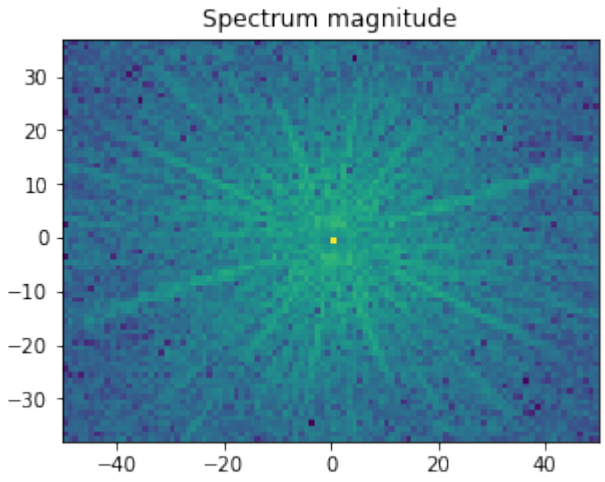
[37]:
```
from scipy import fftpack
```

[38]:
```
image = process_kikuchi(Patterns[0])
M, N = image.shape
F = fftpack.fftn(image)
```

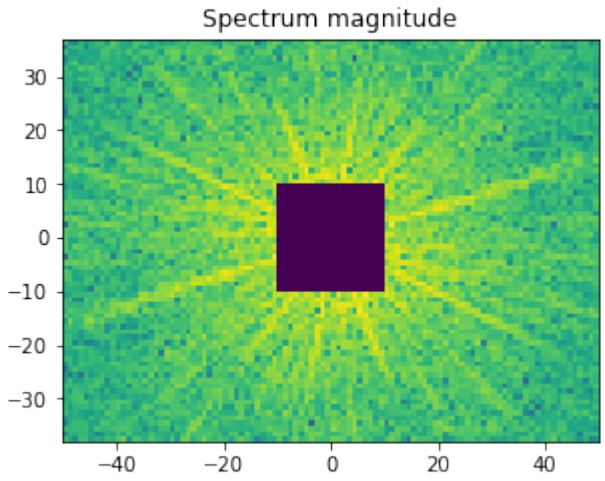
(continues on next page)

(continued from previous page)

```
F_magnitude = np.abs(F)
F_magnitude = fftpack.fftshift(F_magnitude)


magnitude = np.log(1 + F_magnitude)

f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(magnitude, cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude');
```

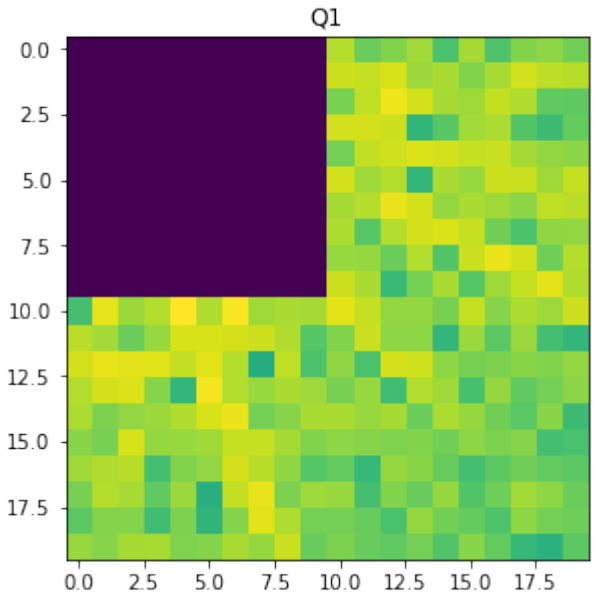


[39]:
```
# Set block around center of spectrum to zero
K = 10
F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

magnitude = np.log(1 + F_magnitude)

f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(magnitude, cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude');
```

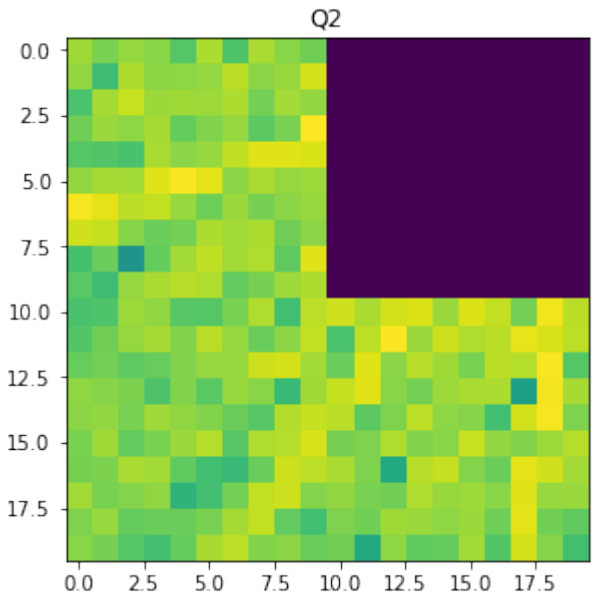


[40]:
```
W=20
Q1 = F_magnitude[M // 2     : M // 2 + W, N // 2     : N // 2 + W]
Q2 = F_magnitude[M // 2     : M // 2 + W, N // 2 - W: N // 2     ]
Q3 = F_magnitude[M // 2 - W: M // 2     , N // 2 - W: N // 2     ]
Q4 = F_magnitude[M // 2 - W: M // 2     , N // 2     : N // 2 + W]
```

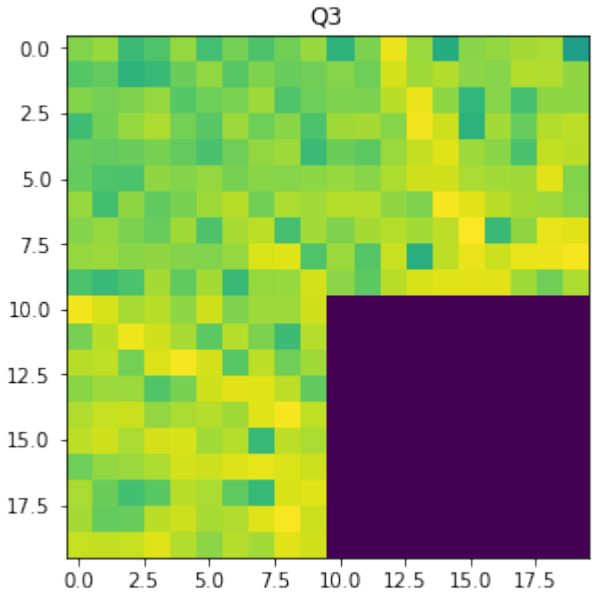
[41]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(np.log(1 + Q1), cmap='viridis',)
ax.set_title('Q1');
```



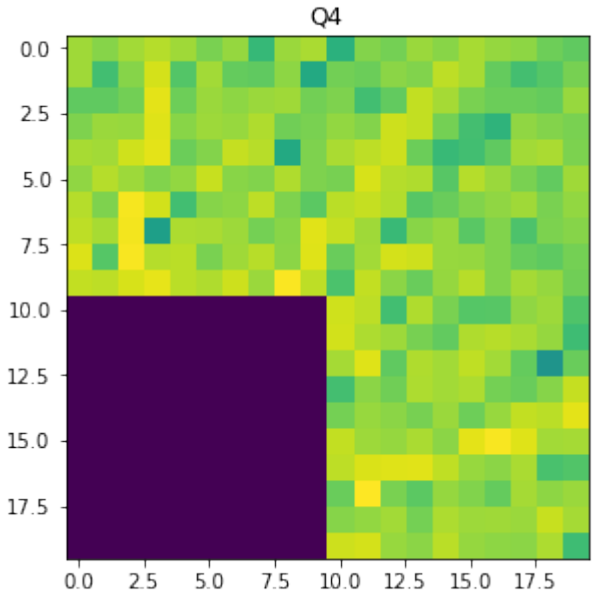
[42]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(np.log(1 + Q2), cmap='viridis',)
ax.set_title('Q2');
```



[43]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(np.log(1 + Q3), cmap='viridis',)
ax.set_title('Q3');
```



[44]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(np.log(1 + Q4), cmap='viridis',)
ax.set_title('Q4');
```



[45]:
```
def quadfft(image, K=5, W=20):
    """
    calculate FFT magnitude quadrants

    remove +/- K points in center (low spatial frequencies)
    limit to +/- W points away from center (low pass, avoid noise at higher frequencies)

    """
    M, N = image.shape
    F = fftpack.fftn(image)

    F_magnitude = np.abs(F)
```

(continues on next page)

(continued from previous page)

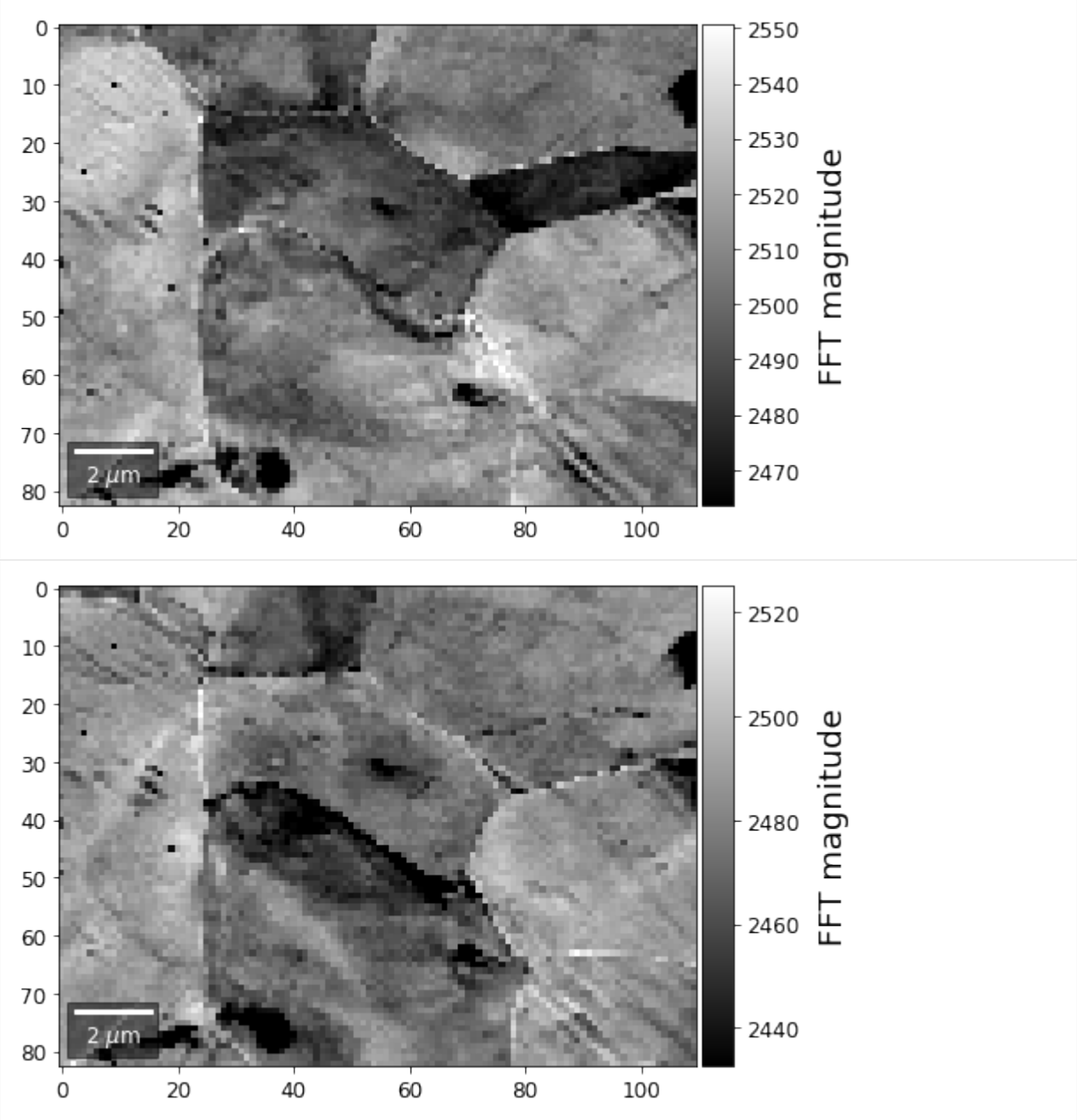
```
    F_magnitude = fftpack.fftshift(F_magnitude)
    F_magnitude = np.log(1 + F_magnitude)

    # Set block +/-K around center of spectrum to zero
    F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

    Q1 = F_magnitude[M // 2    : M // 2 + W, N // 2    : N // 2 + W]
    Q2 = F_magnitude[M // 2    : M // 2 + W, N // 2 - W: N // 2    ]
    Q3 = F_magnitude[M // 2 - W: M // 2    , N // 2 - W: N // 2    ]
    Q4 = F_magnitude[M // 2 - W: M // 2    , N // 2    : N // 2 + W]

    SQ1 = np.sum(Q1)
    SQ2 = np.sum(Q2)
    SQ3 = np.sum(Q3)
    SQ4 = np.sum(Q4)

    return np.array([SQ1,SQ2,SQ3,SQ4])
```

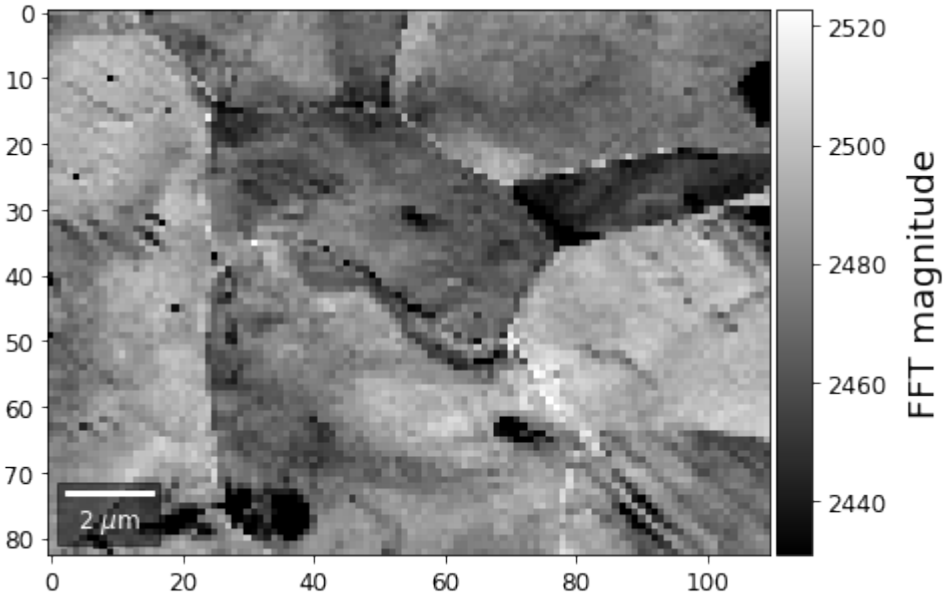
[46]:
```
def calc_quadfft(patterns, process=None, K=10, W=20):
    """
    calc magnitude sum in FFT quadrants of image
    "process" is an optional image pre-processing function
    """
    npatterns=patterns.shape[0]
    quad_fft_magnitude=np.zeros((npatterns, 4))
    tstart = time.time()
    for i in range(npatterns):
        # get current pattern
        if process is None:
            quad_fft_magnitude[i,:] = quadfft(patterns[i,:,:], K=K, W=W)
        else:
            quad_fft_magnitude[i,:] = quadfft(process(patterns[i,:,:]), K=K, W=W)

        # update time info every 10 patterns
        if (i % 100 == 0):
            progress=100.0*(i+1)/npatterns
            tup = time.time()
            togo = (100.0-progress)*(tup-tstart)/(60.0*progress)
            sys.stdout.write("\rtotal map points:%5i current:%5i progress: %4.2f%% -> %6.
↪1f min to go"
                % (npatterns,i+1,progress,togo)  )
            sys.stdout.flush()

    return quad_fft_magnitude
```

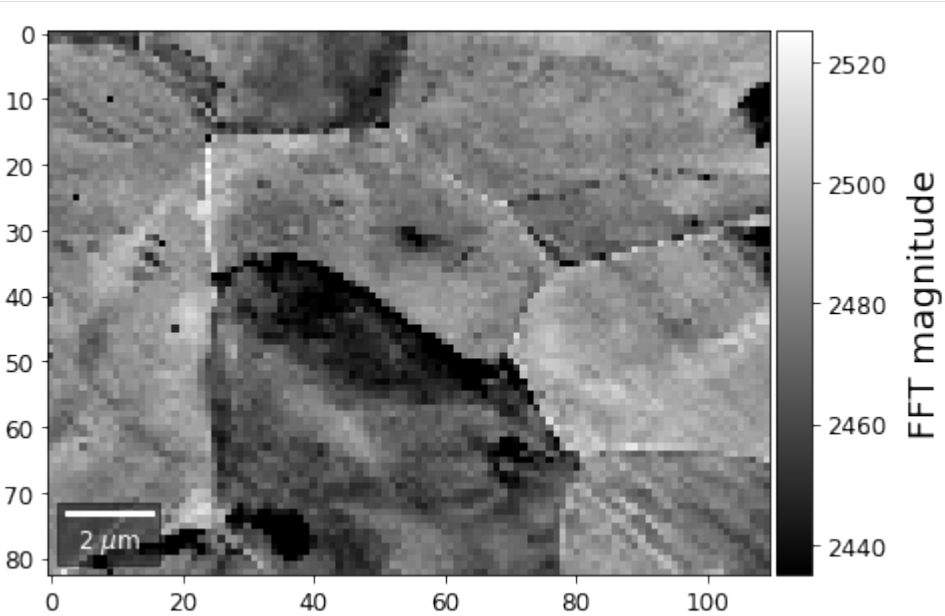
[47]:
```
qfftmag =  calc_quadfft(Patterns, process=process_kikuchi)
```

```
total map points: 9130 current: 9101 progress: 99.68% ->    0.0 min to go
```

[48]:
```
for q, quadrant in enumerate(qfftmag.T):
    signal_map = make2Dmap(quadrant,XIndex,YIndex,MapHeight,MapWidth)
    plot_SEM(signal_map, vrange=None, cmap='gray',
             colorbarlabel='FFT magnitude', microns=step_map_microns,
             filename='qFFT_'+str(q))
```

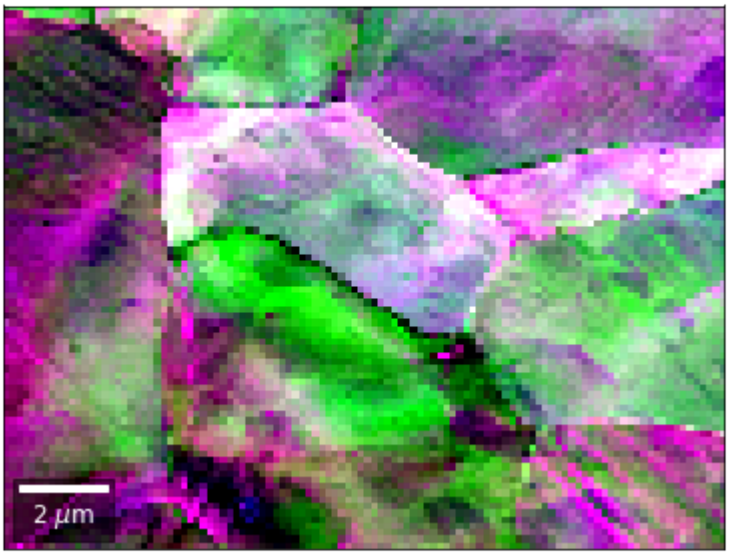

0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
2550
2540
2530
2520
2510
2500
2490
2480
2470
FFT magnitude
0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
2520
2500
2480
2460
2440
FFT magnitude





[49]:
```
signal = qfftmag[:,1] /qfftmag[:,0]
red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

signal = qfftmag[:,2] /qfftmag[:,0]
green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

signal = qfftmag[:,3] /qfftmag[:,0]
blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
```

[50]:
```
rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                     filename='qfft_RGB', microns=step_map_microns,
                     rot180=False, add_bright=0, contrast=1.0)
```

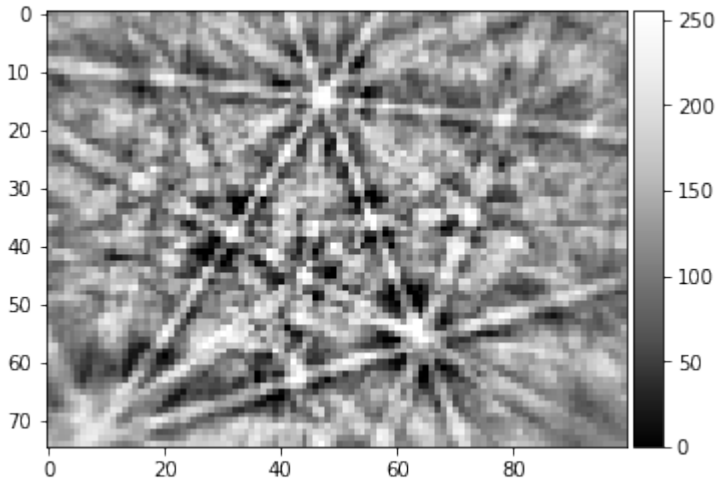


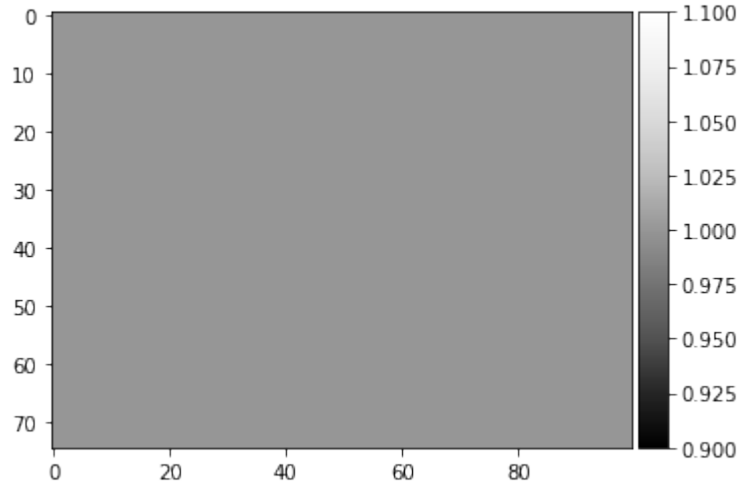
Radon Transform Imaging

(. . . without pattern indexing ;-)

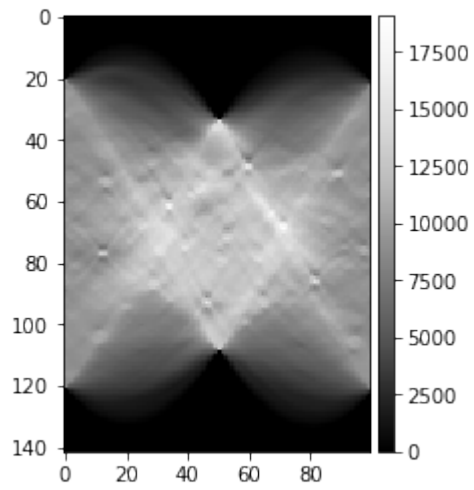
[51]:
```
from skimage.transform import radon
```

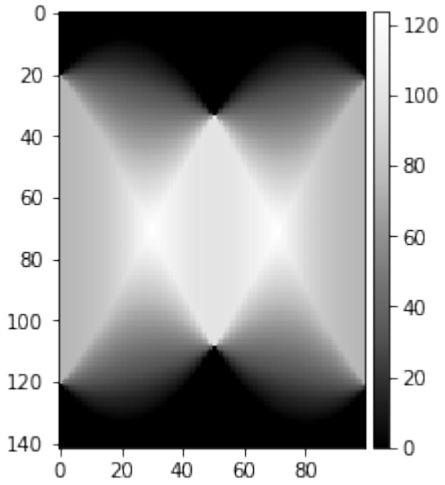
[52]:
```
image = process_kikuchi(Patterns[0])
image_uniform = np.ones_like(image)

img_height, img_width = image.shape

plot_image(image)
plot_image(image_uniform)
```



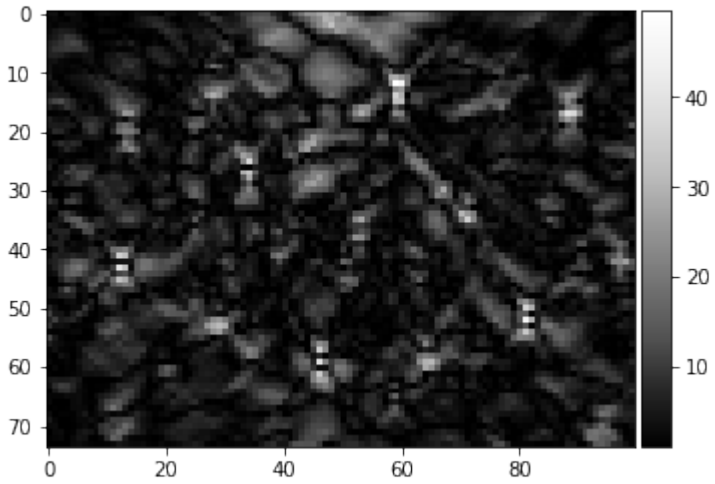


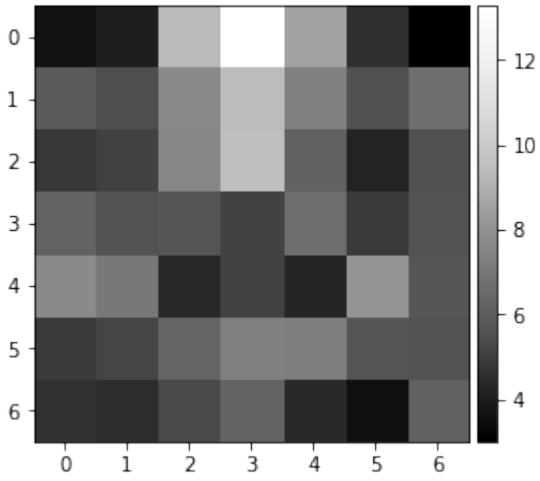
[53]:
```
theta = np.linspace(0., 180., max(image.shape), endpoint=False)
sinogram = radon(image, theta=theta, circle=False)
sinogram_uniform = radon(image_uniform, theta=theta, circle=False)
plot_image(sinogram)
plot_image(sinogram_uniform)

sino = sinogram / sinogram_uniform
h,w = sino.shape
print(h,w, img_height)
sino_center = sino[h//2-img_height//2: h//2+img_height//2,:]

mean_radon = np.nanmean(sino_center)
print(mean_radon)
sino_center = np.sqrt(1.0+(sino_center - mean_radon)**2 )

sino77=arbse.rebin_array(sino_center)
plot_image(sino_center)
plot_image(sino77)
```






```
142 100 75
123.749473208
```



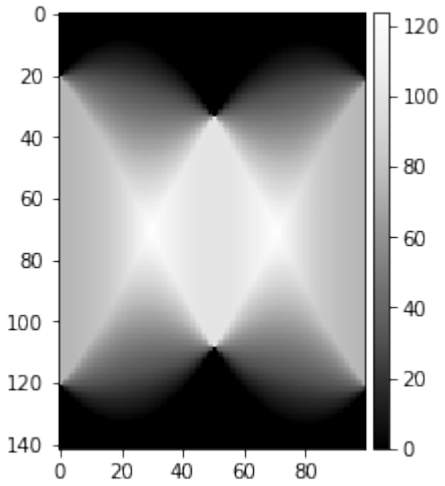


[54]:
```
theta = np.linspace(0., 180., max(image.shape), endpoint=False)

image_uniform = np.ones_like(background_static_binned)
sinogram_uniform = radon(image_uniform, theta=theta, circle=False)
```

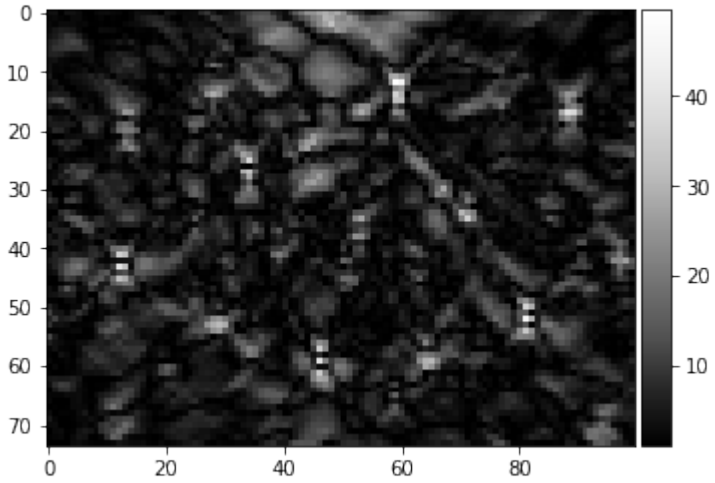
(continues on next page)

(continued from previous page)

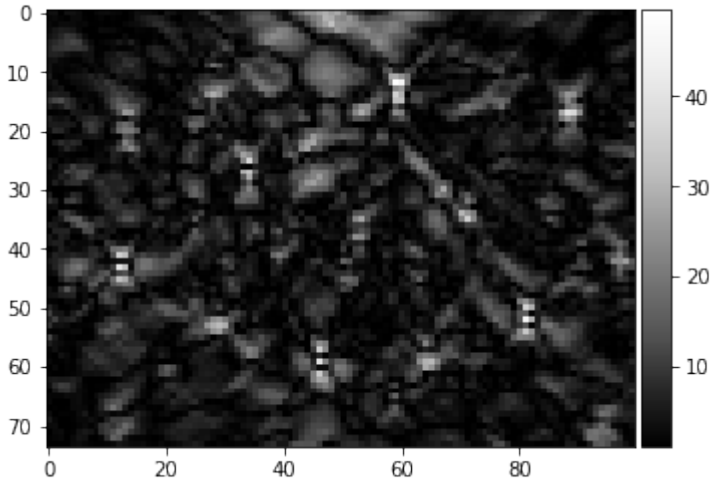
```
plot_image(sinogram_uniform)

def calc_radon(image, theta, sinogram_reference=None):
    """ calculate radon vbse array """

    sinogram = radon(image, theta=theta, circle=False)
    #if sinogram_reference is None:
    #    image_uniform = np.ones_like(image)
    #    sinogram_reference = radon(image_uniform, theta=theta, circle=False)

    img_height, img_width = image.shape

    sino = sinogram / sinogram_reference
    h,w = sino.shape

    sino_center = sino[h//2-img_height//2: h//2+img_height//2,:]
    mean_radon = np.nanmean(sino_center)
    sino_center = np.sqrt(1.0+(sino_center - mean_radon)**2 )
    return sino_center
```



[55]:
```
kiku = process_kikuchi(Patterns[0])
sino = calc_radon(kiku, theta, sinogram_reference=sinogram_uniform)
plot_image(sino)

def process_radon(pattern):
    kiku = process_kikuchi(pattern)
    sino = calc_radon(kiku, theta, sinogram_reference=sinogram_uniform)
    return sino
```



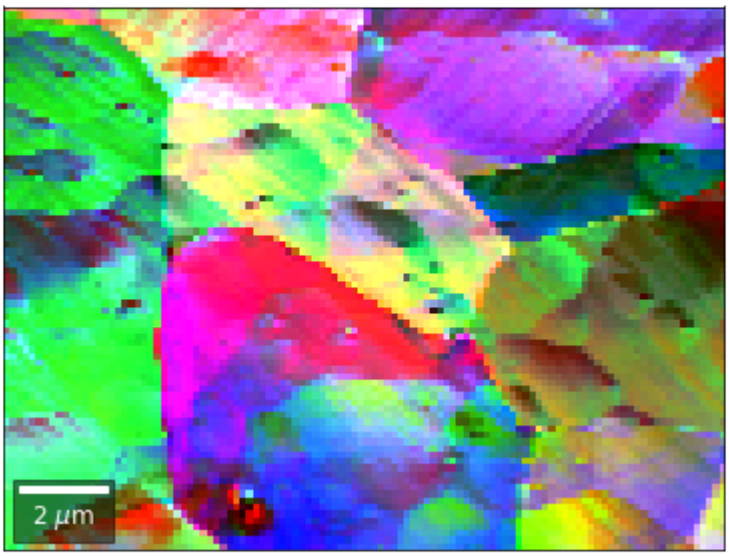
[56]:
```
radon_test = process_radon(Patterns[0])
plot_image(radon_test)
```

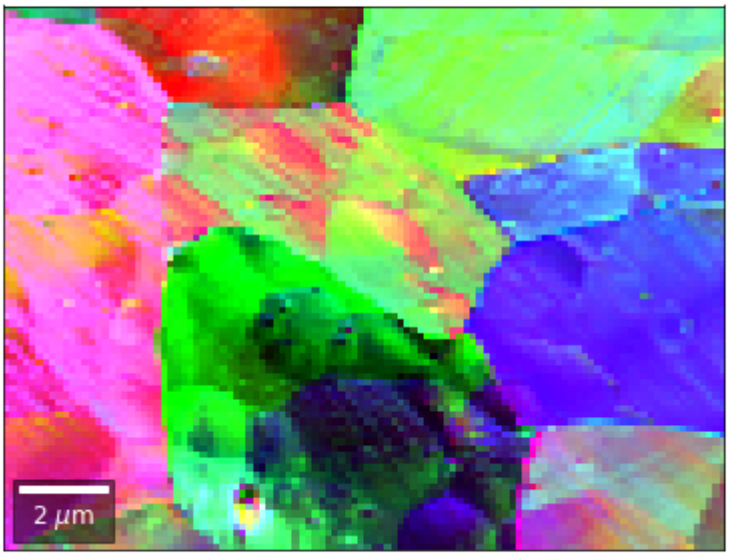


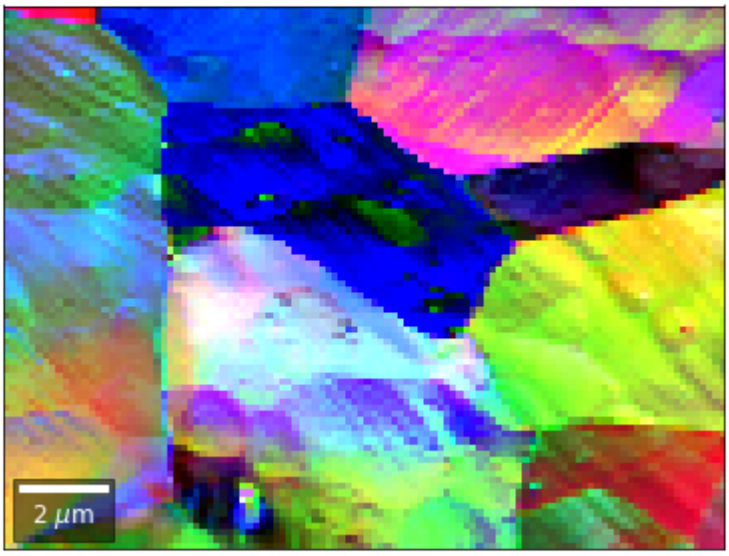
[57]:
```
vbse_radon =  arbse.make_vbse_array(Patterns, process=process_radon)
```

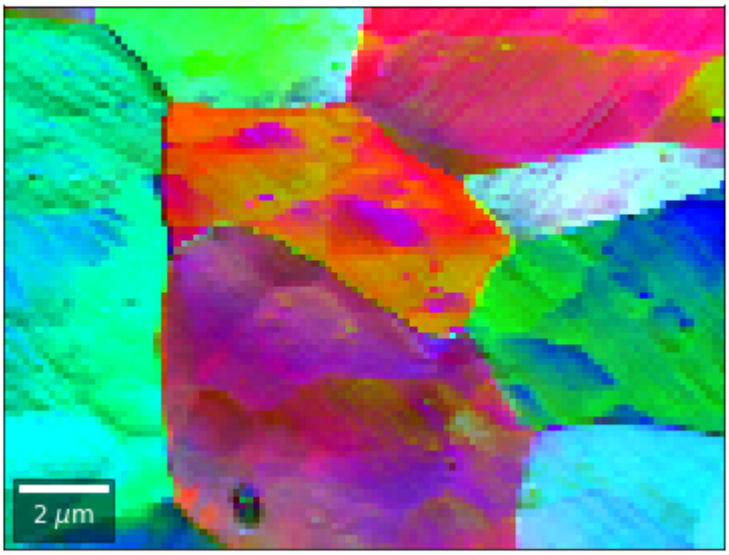
```
total points: 9130 current: 9130 finished -> total calculation time :   16.2 min
```

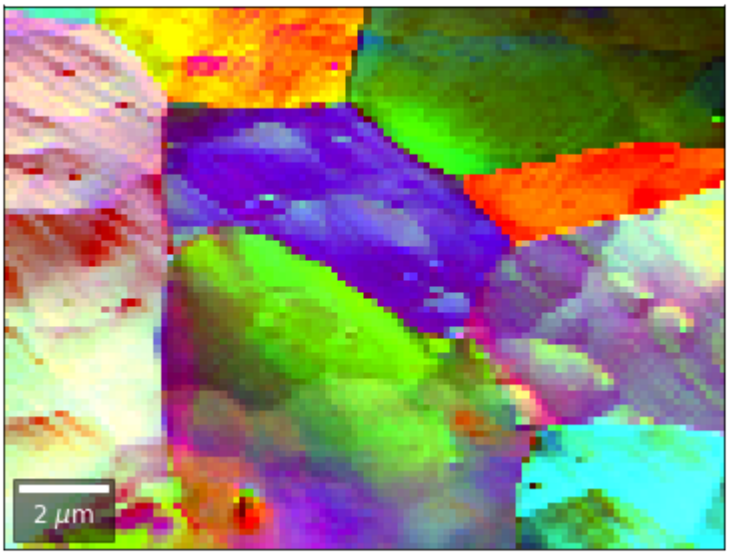
[58]:
```
vFSD = vbse_radon
# relative change to previous row
for row in range(1,7):
    drow = -1
    signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                     filename='vRadon_RGB_drow_'+str(row),
                     microns=step_map_microns,
                     rot180=False, add_bright=0, contrast=1.2)
```

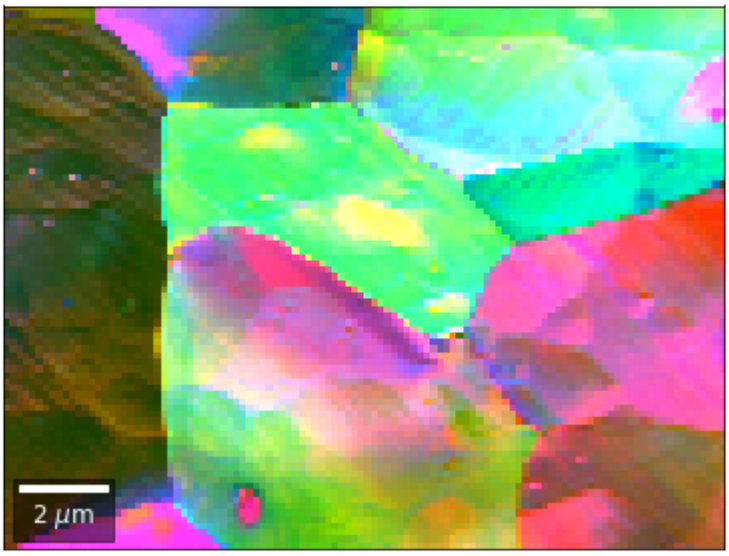

2 μm


2 μm

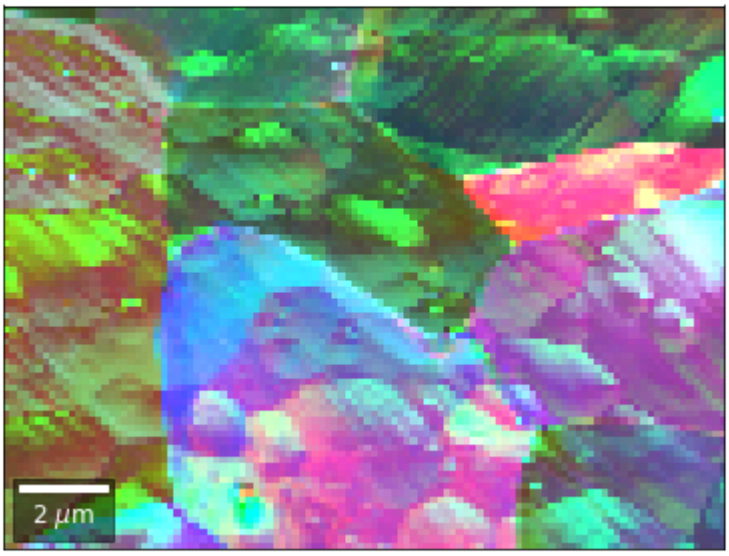

2 μm

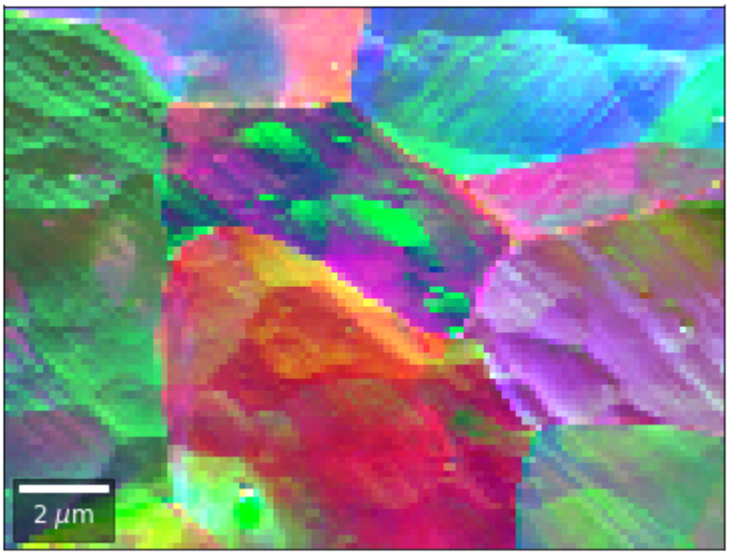

2 μm

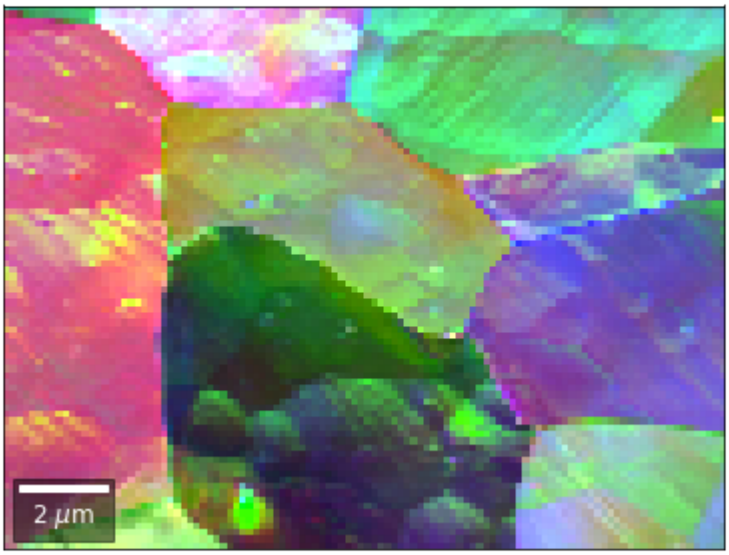

2 μm

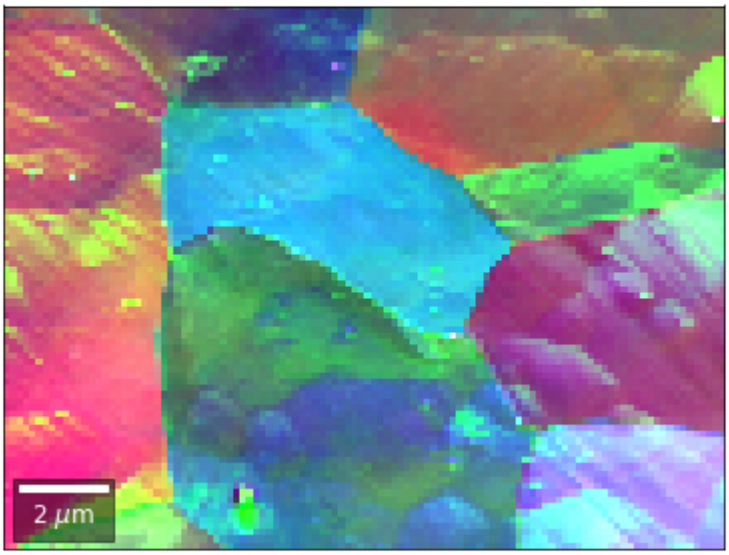

2 μm

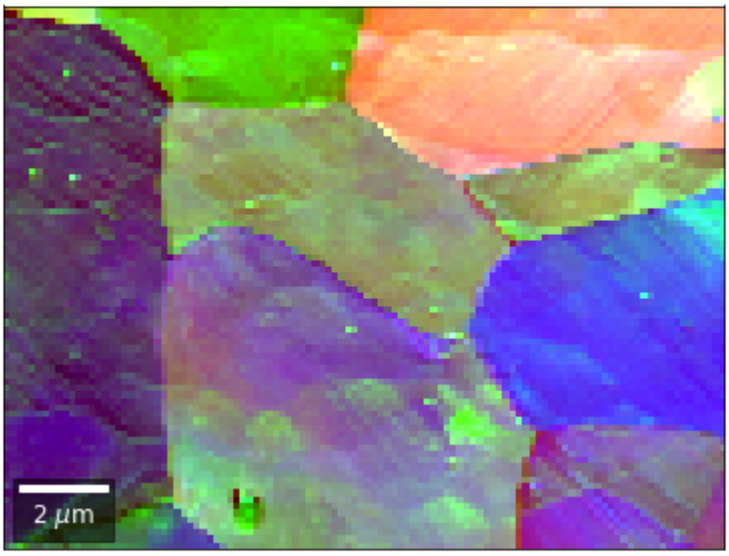
[59]:
```
# rgb direct
rgb_direct = []

for row in range(7):
    signal = vFSD[:,row,0]
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,3]
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = vFSD[:,row,6]
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                     filename='vRadon_RGB_'+str(row),
                     rot180=False, microns=step_map_microns,
                     add_bright=0, contrast=0.8)

    rgb_direct.append(rgb)
```

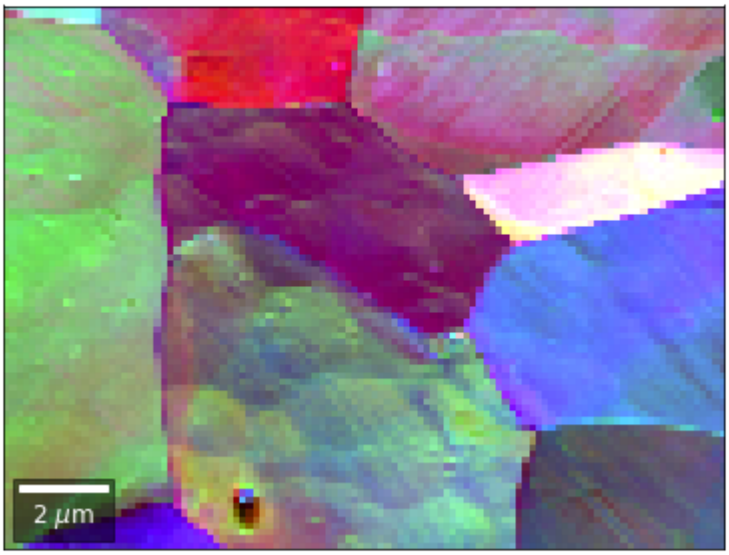


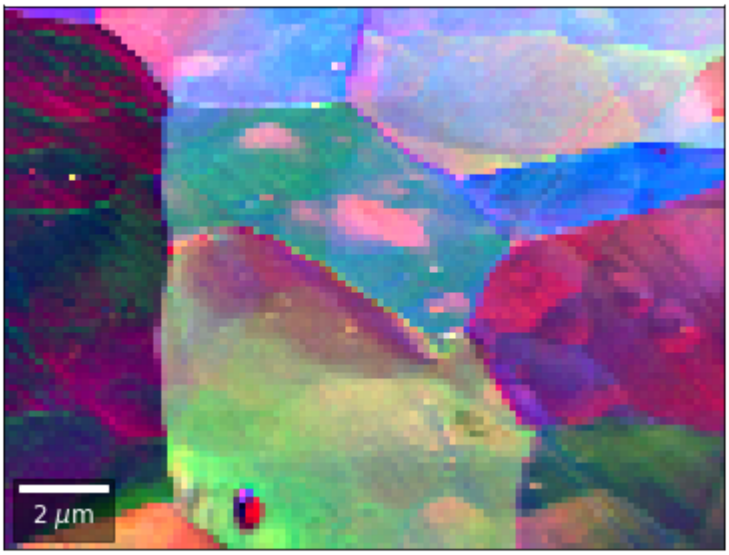



2 μm


2 μm


2 μm





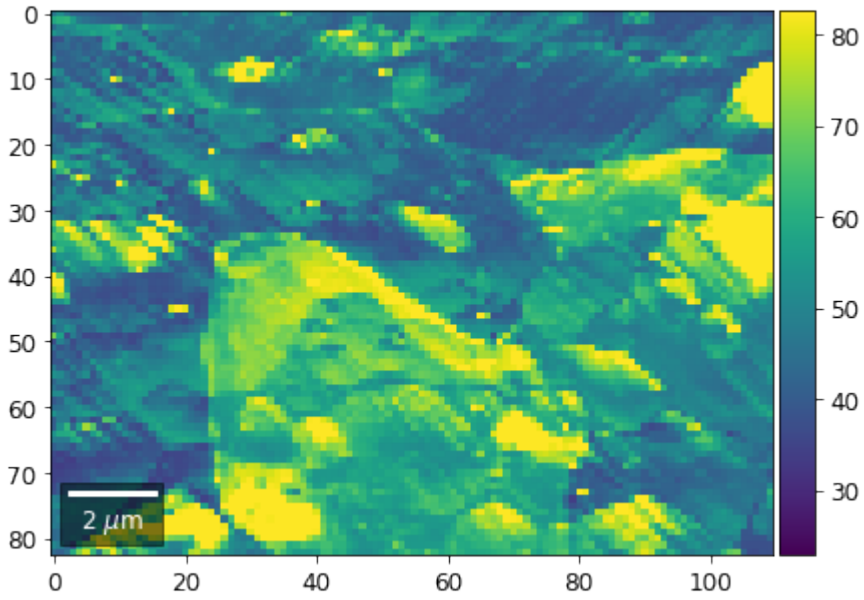
[60]:
```
# signal: sum of row
vmin=40000000
vmax=0

bse_rows = []

# (1) get full range for all images
for row in range(7):
    signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    minv, maxv = get_vrange(signal, stretch=3.0)
    if (minv<vmin):
        vmin=minv
    if (maxv>vmax):
        vmax=maxv

# (2) make plots with same range for comparisons of absolute BSE values
vrange=[minv, maxv]
print('Range of Values: ', vrange)
```

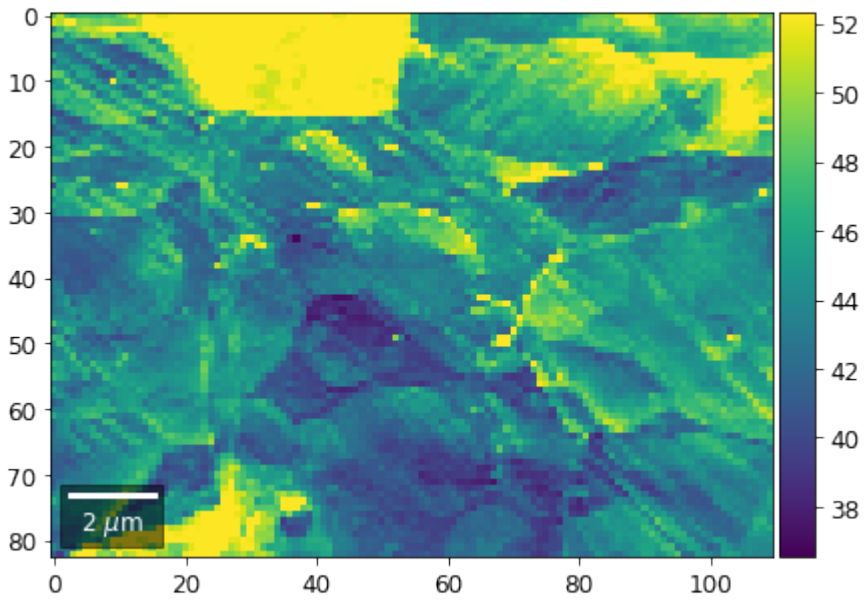
(continues on next page)

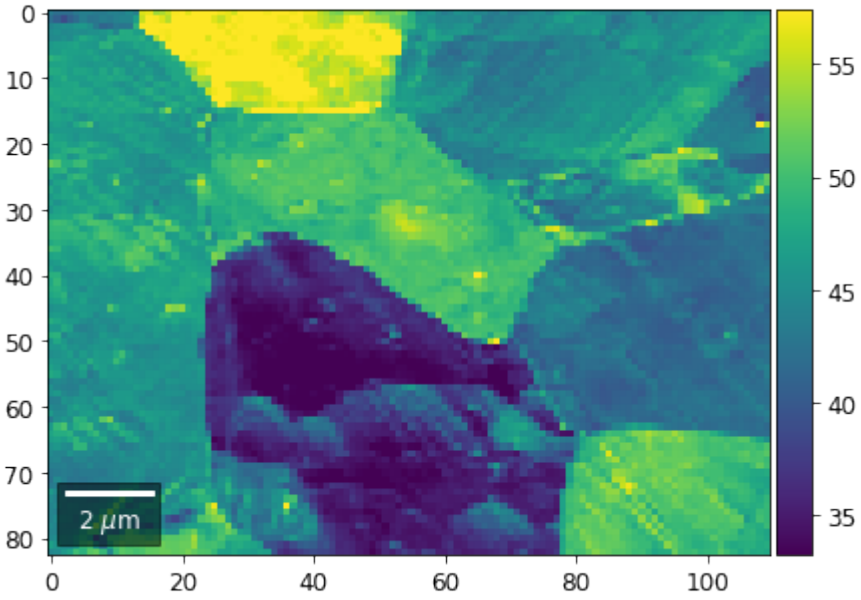
(continued from previous page)

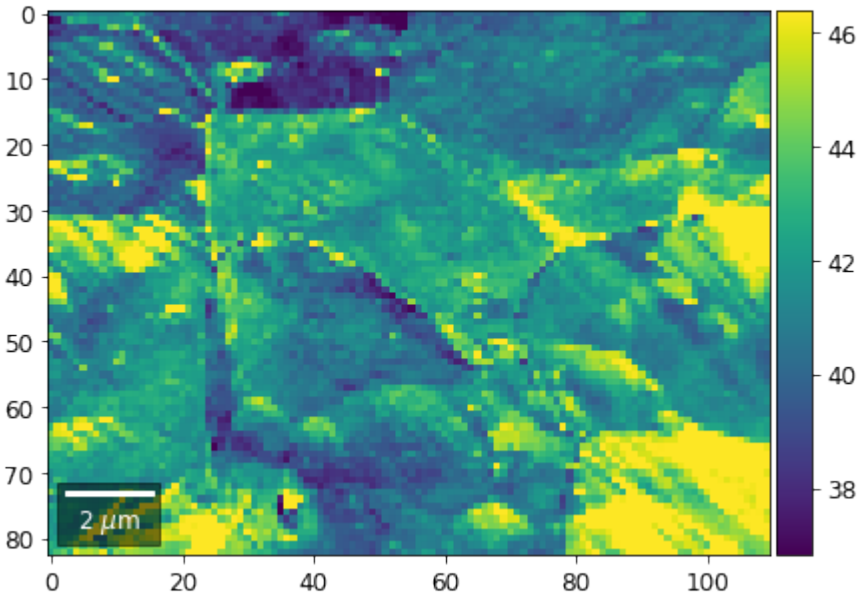
```
vrange=None
for row in range(7):
    signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    bse_rows.append(signal_map)
    plot_SEM(signal_map, vrange=vrange, filename='vRadon_row_'+str(row),
             rot180=True, microns=step_map_microns)
```

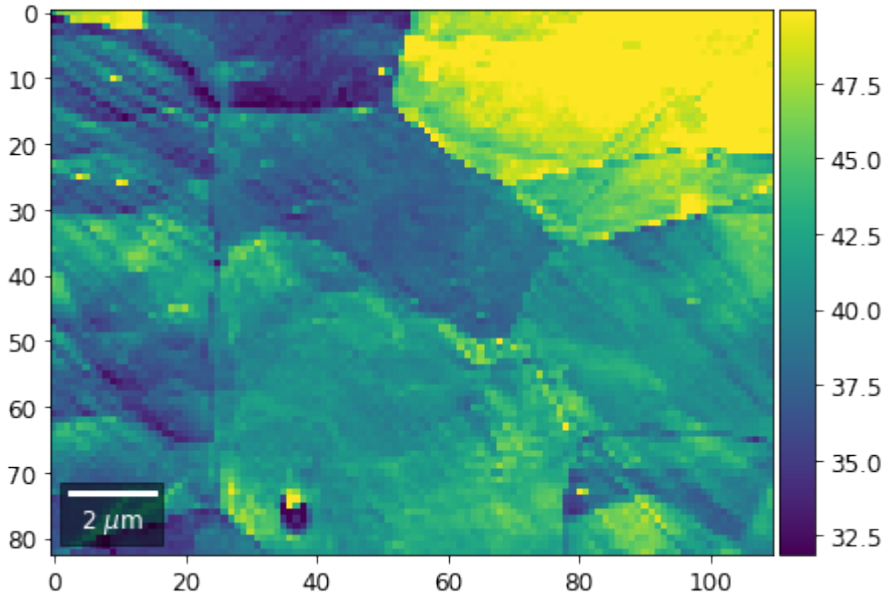
```
Range of Values:  [24.123688464536887, 52.246895984591987]
```

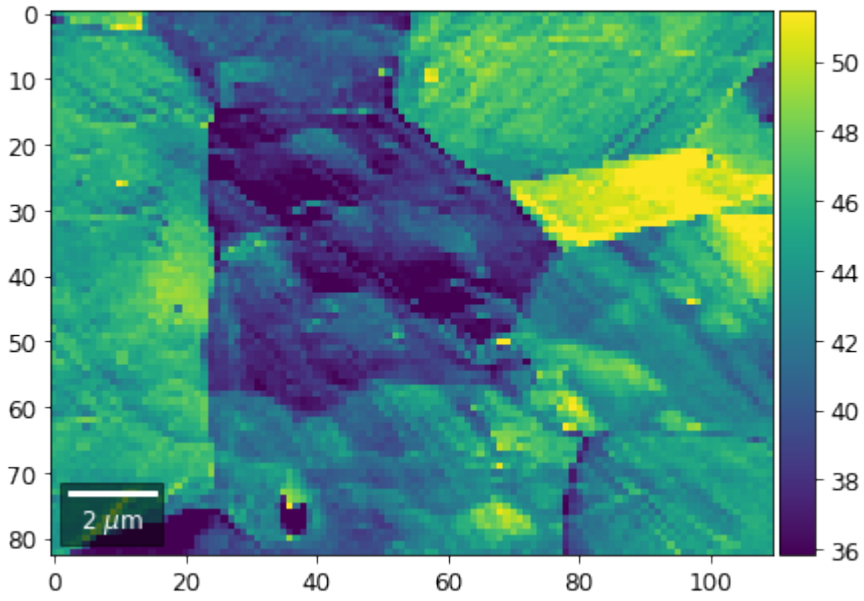


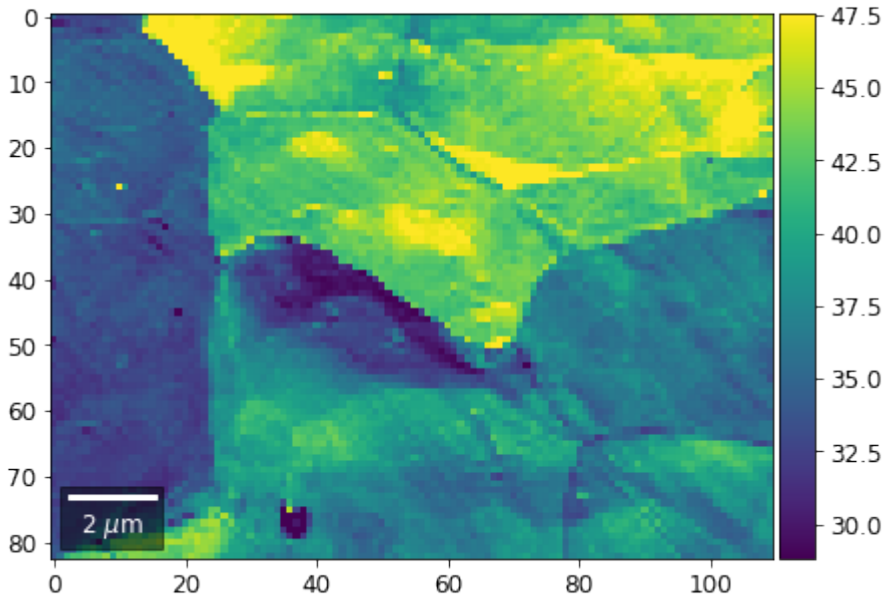



0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
55
50
45
40
35
2 µm

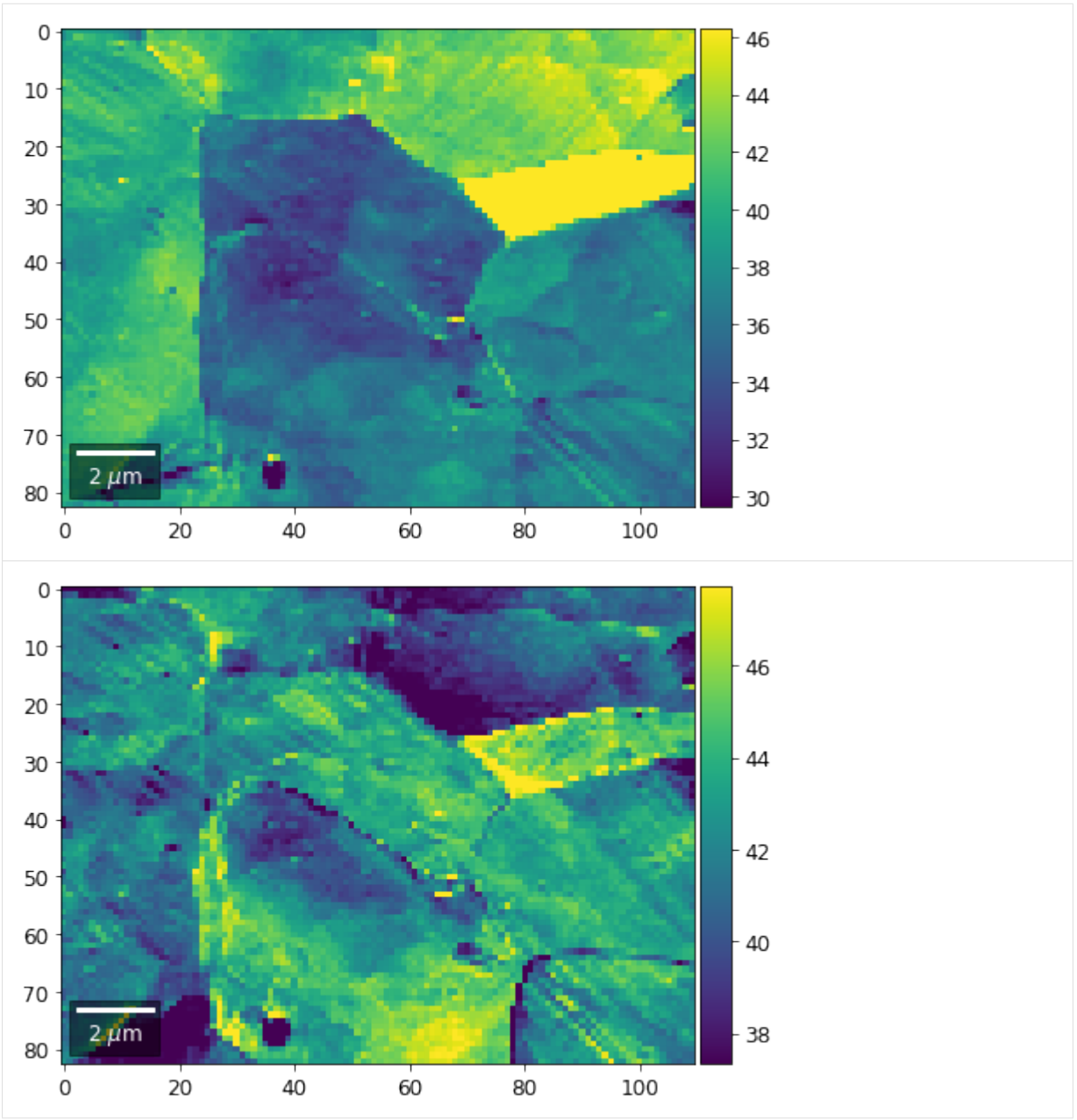

0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
46
44
42
40
38
2 µm

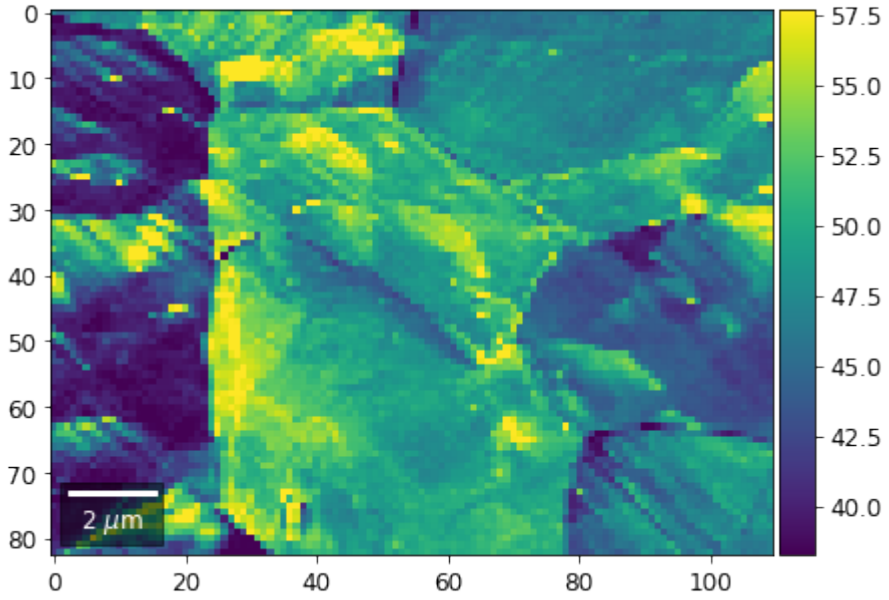

0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
47.5
45.0
42.5
40.0
37.5
35.0
32.5

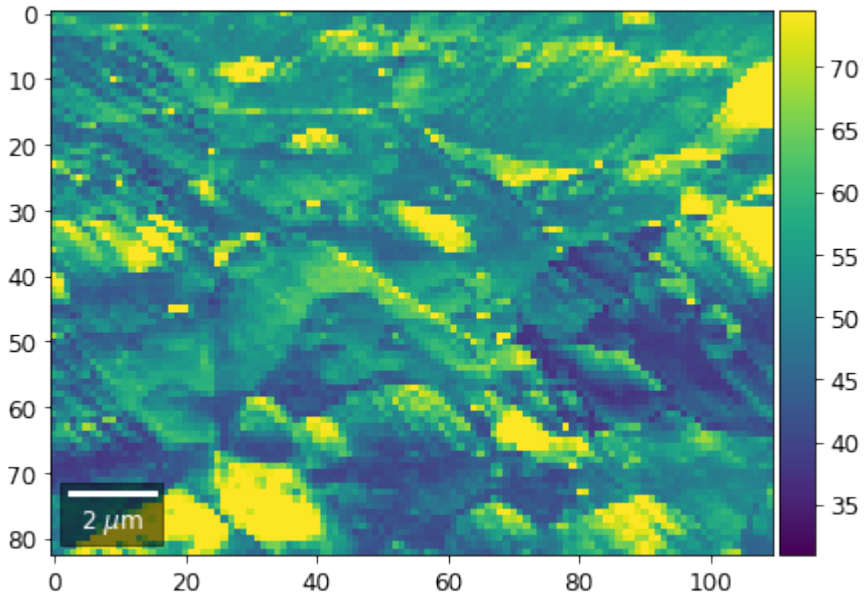

0
10
20
30
40
50
60
70
80
2 µm
0
20
40
60
80
100
50
48
46
44
42
40
38
36

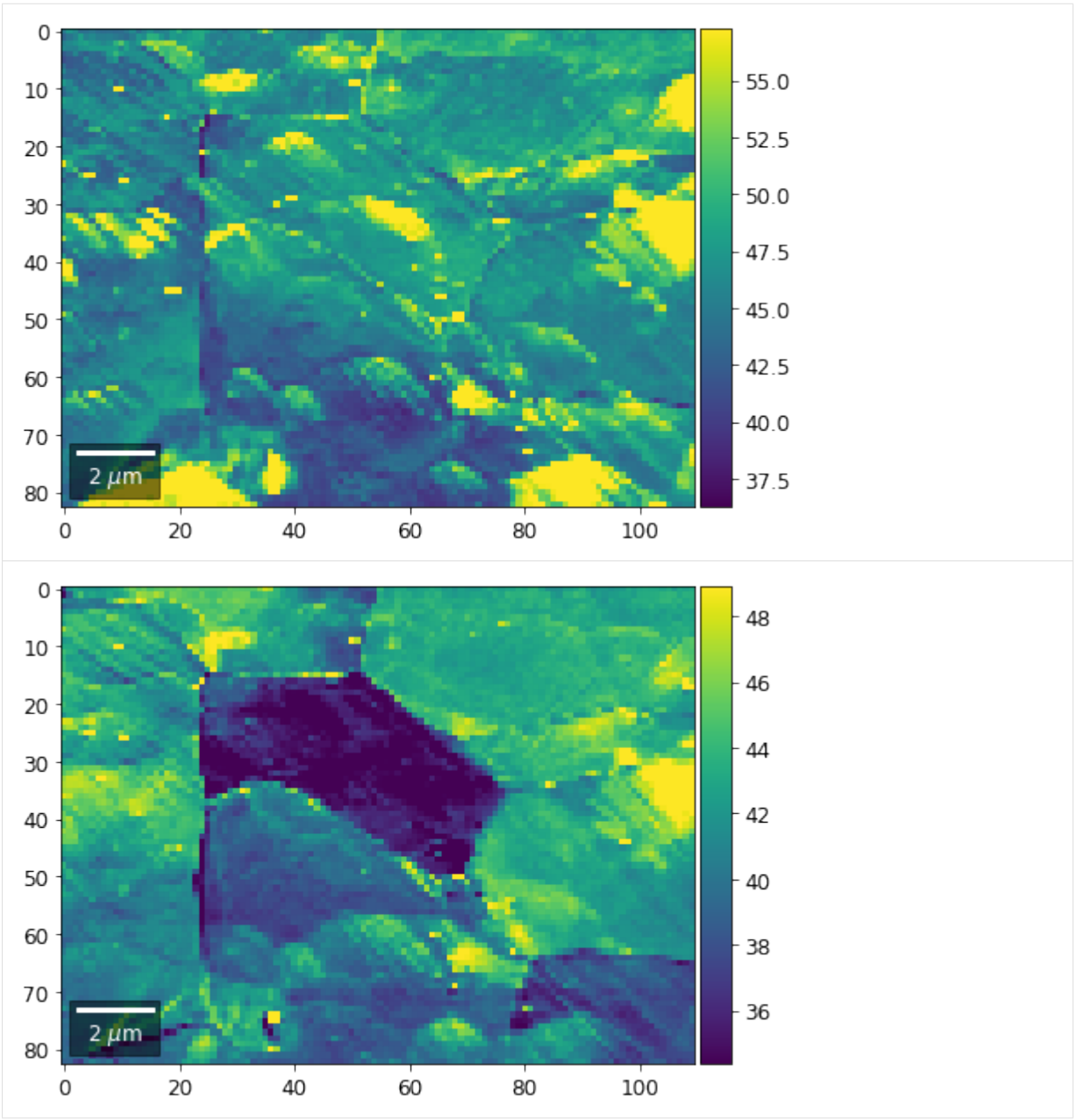


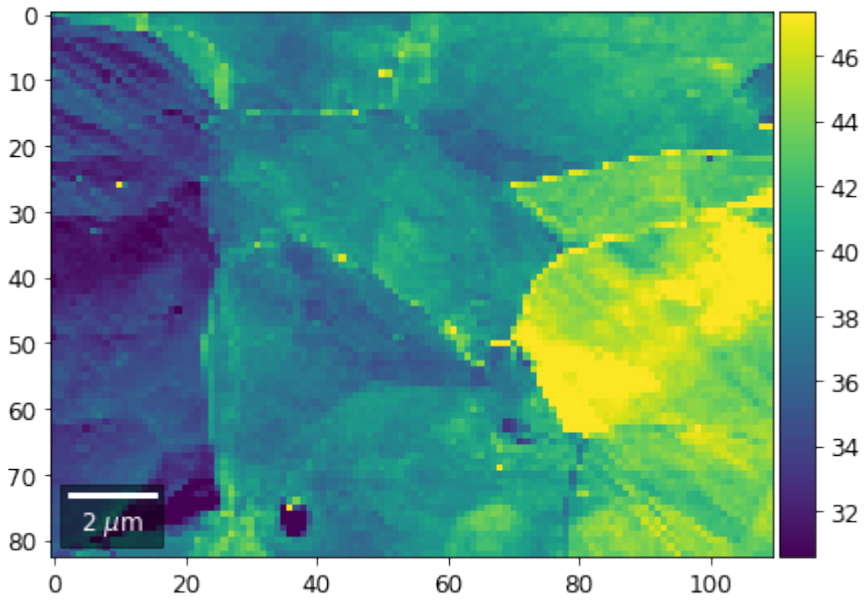
[61]:
```
# signal: sum of column
vmin=400000
vmax=0

bse_cols = []

# (1) get full range for all images
for col in range(7):
    signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    minv, maxv = get_vrange(signal)
    if (minv<vmin):
        vmin=minv
    if (maxv>vmax):
        vmax=maxv

# (2) make plots with same range for comparisons of absolute BSE values
#vrange=[vmin, vmax]
vrange=None # no fixed scale

for col in range(7):
    signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
    signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
    bse_cols.append(signal_map)
    plot_SEM(signal_map, vrange=vrange, filename='vRadon_col_'+str(col),
             rot180=True, microns=step_map_microns)
```


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
46
44
42
40
38
36
34
32
30
0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
46
44
42
40
38


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
57.5
55.0
52.5
50.0
47.5
45.0
42.5
40.0
2 μm


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
70
65
60
55
50
45
40
35
2 μm


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
2 µm
55.0
52.5
50.0
47.5
45.0
42.5
40.0
37.5
48
46
44
42
40
38
36



[ ]:

[ ]:

SEM 2D BSE Imaging: Fe fcc Precipitates in bcc Matrix

Example Data: Steel_FCC_in_BCC_2752

This is an example notebook to demonstrate angle resolved BSE imaging via signal extraction from raw, saved, EBSD patterns.

A key advantage when using saved raw EBSD patterns is that we can partition the raw signal into a "Kikuchi pattern" and a "background" signal, which will give different types of contrasts. This partition is impossible when using conventional semiconductor diode detection etc. for angle-resolved signal BSE collection. We cannot distinguish between "Kikuchi signal" and "background signal" in the electron current from the diode because the diode signal is a point-signal, as compared to a full 2D signal which is available on the phosphor screen.

[1]:
```
# directory with the HDF5 EBSD pattern file:
data_dir = "../../../xcdskd_reference_data/Steel_FCC_in_BCC/"

# filename of the HDF5 file:
hdf5_filename =  "Steel_FCC_in_BCC_2752.h5"

# Options
output_verbose = False # print explicit filenames
```

Necessary packages

[2]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
# ignore divide by Zero
np.seterr(divide='ignore', invalid='ignore')
```

(continues on next page)

(continued from previous page)

```
import time, sys, os
import h5py
import skimage.io

from aloe.plots import normalizeChannel, make2Dmap, get_vrange
from aloe.plots import plot_image, plot_SEM, plot_SEM_RGB
from aloe.image import arbse
from aloe.image.downsample import downsample
from aloe.image.kikufilter import process_ebsp
from aloe.io.progress import print_progress_line
```

[3]:
```
# load background from different experiment with same detector
background_static_txt_full = np.loadtxt("../data/patterns/static_bam_ef.txt")

# make result dirs and filenames
h5FileNameFull=os.path.abspath(data_dir + hdf5_filename)
h5FileName, h5FileExt = os.path.splitext(h5FileNameFull)
h5FilePath, h5File = os.path.split(h5FileNameFull)
timestr = time.strftime("%Y%m%d-%H%M%S")
h5ResultFile="arBSE_" + hdf5_filename


# close HDF5 file if still open
if 'f' in locals():
    f.close()
f=h5py.File(h5FileName+h5FileExt, "r")

ResultsDir = h5FilePath+"/arBSE_" + timestr + "/"
CurrentDir = os.getcwd()
#print('Current Directory: '+CurrentDir)
#print('Results Directory: '+ResultsDir)
if not os.path.isdir(ResultsDir):
    os.makedirs(ResultsDir)
os.chdir(ResultsDir)

if output_verbose:
    print('HDF5 full file name: ', h5FileNameFull)
    print('HDF5 File: ', h5FileName+h5FileExt)
    print('HDF5 Path: ', h5FilePath)
    print('Results Directory: ', ResultsDir)
    print('Results File: ', h5ResultFile)
```

[4]:
```
DataGroup="/Scan/EBSD/Data/"
HeaderGroup="/Scan/EBSD/Header/"
Patterns = f[DataGroup+"RawPatterns"]
#StaticBackground=f[DataGroup+"StaticBackground"]
XIndex =   f[DataGroup+"X BEAM"]
YIndex =   f[DataGroup+"Y BEAM"]
MapWidth = f[HeaderGroup+"NCOLS"].value
MapHeight= f[HeaderGroup+"NROWS"].value
PatternHeight=f[HeaderGroup+"PatternHeight"].value
PatternWidth =f[HeaderGroup+"PatternWidth"].value
print('Pattern Height: ', PatternHeight)
print('Pattern Width : ', PatternWidth)
PatternAspect=float(PatternWidth)/float(PatternHeight)
print('Pattern Aspect: '+str(PatternAspect))
```

(continues on next page)

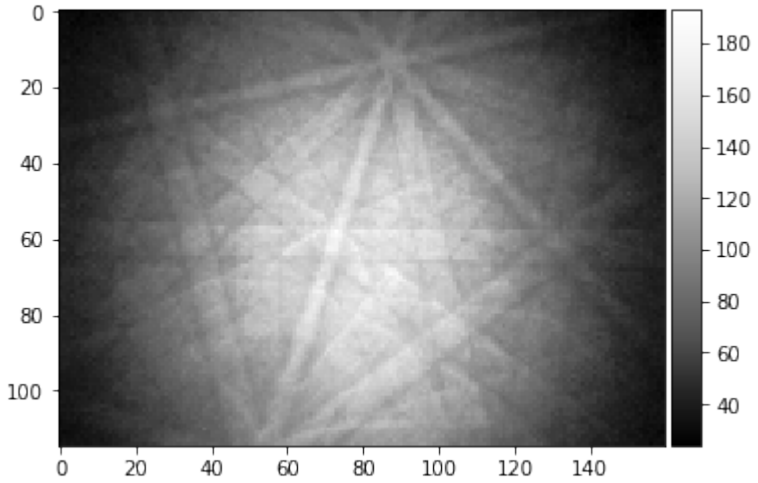
(continued from previous page)

```
print('Map Height: ', MapHeight)
print('Map Width : ', MapWidth)

step_map_microns = f[HeaderGroup+"XSTEP"].value
print('Map Step Size (microns): ', step_map_microns)
```

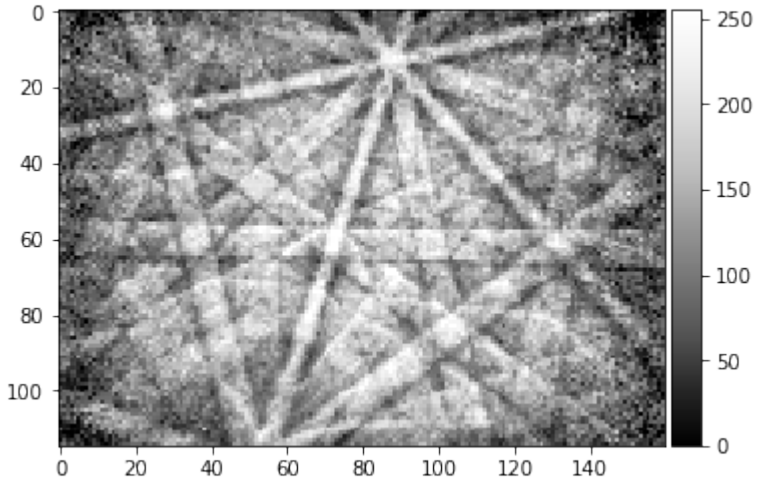
```
Pattern Height:  115
Pattern Width :  160
Pattern Aspect: 1.391304347826087
Map Height:  300
Map Width :  400
Map Step Size (microns):  0.121037425197
```

Check Optional Pattern Processing

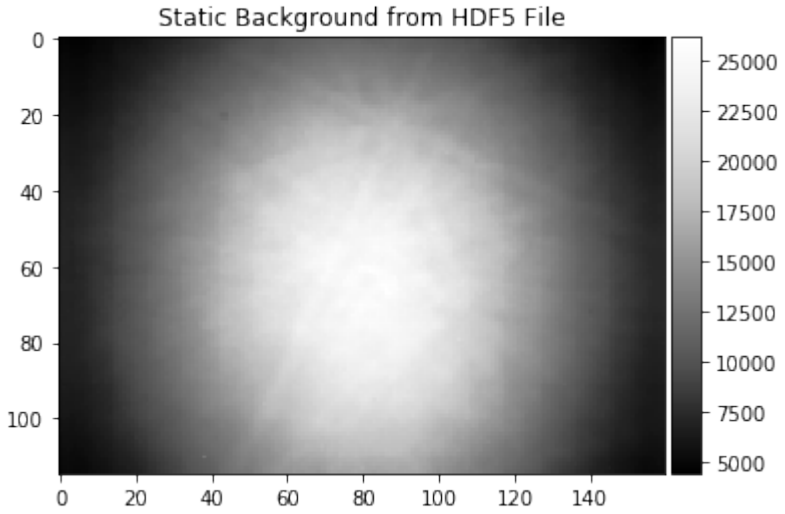
[5]:
```
ipattern = 215
raw_pattern=Patterns[ipattern,:,:]
plot_image(raw_pattern)
skimage.io.imsave('pattern_raw_' + str(ipattern) + '.tiff', raw_pattern, plugin='tifffile')
```



[6]:
```
# make processed pattern without static background
processed_pattern = process_ebsp(raw_pattern, binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_' + str(ipattern) + '.tiff', processed_pattern,␣
↪plugin='tifffile')
```



[7]:
```
try:
    # get static background from HDF5, cut off first lines to fit to Patterns
    background_static_file = f[HeaderGroup+"StaticBackground"][0,5:,:]
    plot_image(background_static_file, title="Static Background from HDF5 File")
except:
    background_static_file = None
```



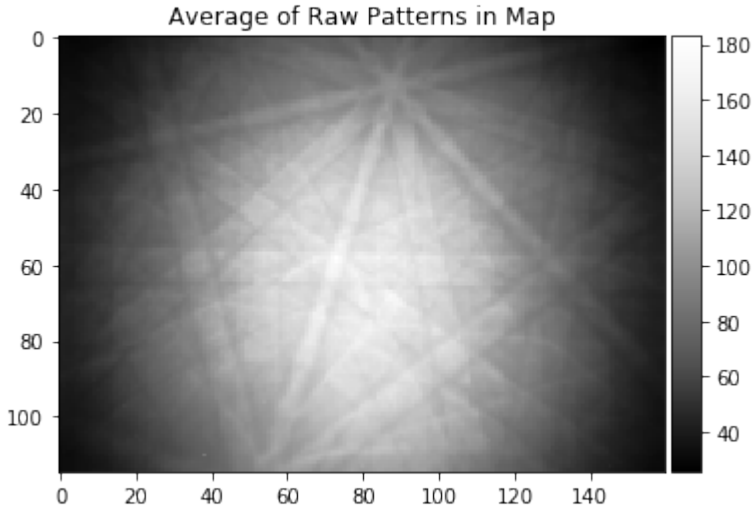
Static Background from Pattern Average in the Map

We can also approximate a static background from the EBSD map itself; or even use an extra map that was taken explicitly for making a background, e.g. from the sample holder. For polycrystalline samples with a large enough number of grains in random orientations, the Kikuchi patterns from all grains will average out when taking the average of all pattern. For single crystalline samples, or samples with a low number of different orientation present in the map area measured, the average of all patterns will stay contain Kikuchi diffraction features. These features in the static background will tend to produce artififacts when the raw data is processed using the background with diffraction features.

[8]:
```
calc_bg = True
if calc_bg:
    # avoid loading 35GB into RAM if you don't have 35GB RAM...
    # use incremental "updating average"
    # https://math.stackexchange.com/questions/106700/incremental-averageing
    tstart = time.time()
    npatterns = Patterns.shape[0]
    current_average = np.copy(Patterns[0]).astype(np.float64)
    for i in range(1, npatterns):
        current_average = current_average + (Patterns[i] - current_average)/i
        print_progress_line(tstart, i-1, npatterns-1, every=100)

    background_static_from_map = current_average
```

```
total points:119999 current:119999 finished -> total calculation time :    0.4 min
```

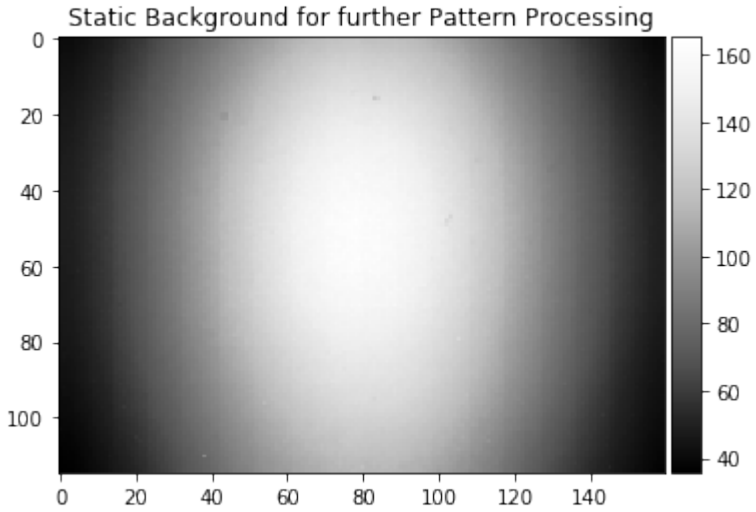
[9]:
```
plot_image(background_static_from_map, title="Average of Raw Patterns in Map")
```



Because we have an bcc-fcc orientation relationship between all the grains in this map, the pattern average still shows all the features which are common to both the bcc parent grain and the fcc inclusions.

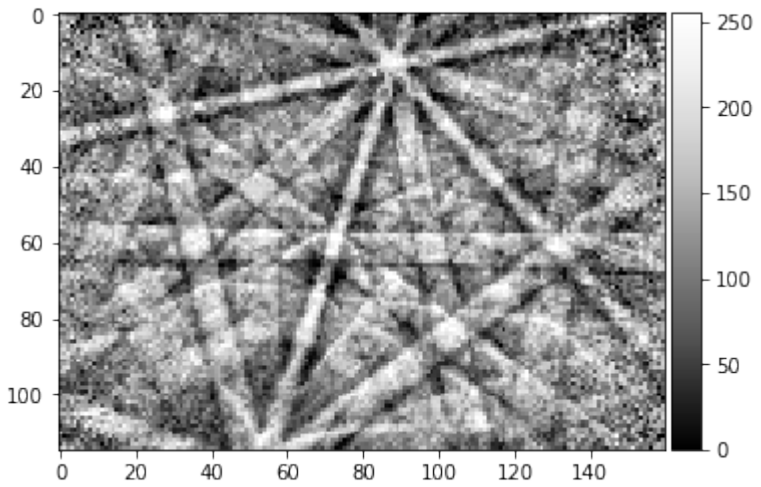
[10]:
```
# assign static background from file
background_static = background_static_txt_full[5:,:]
print(background_static.shape)
print(background_static)
plot_image(background_static, title="Static Background for further Pattern Processing")
```

```
(115, 160)
[[ 41.187  41.367  41.491 ...,  39.766  39.829  40.037]
 [ 41.283  41.374  41.392 ...,  39.608  39.749  39.836]
 [ 41.607  42.042  41.783 ...,  40.158  40.186  40.333]
 ...,
 [ 36.24   37.008  37.093 ...,  39.125  38.512  38.229]
 [ 36.347  36.62   36.757 ...,  38.399  38.03   37.787]
 [ 35.654  36.221  36.438 ...,  38.408  37.231  37.011]]
```



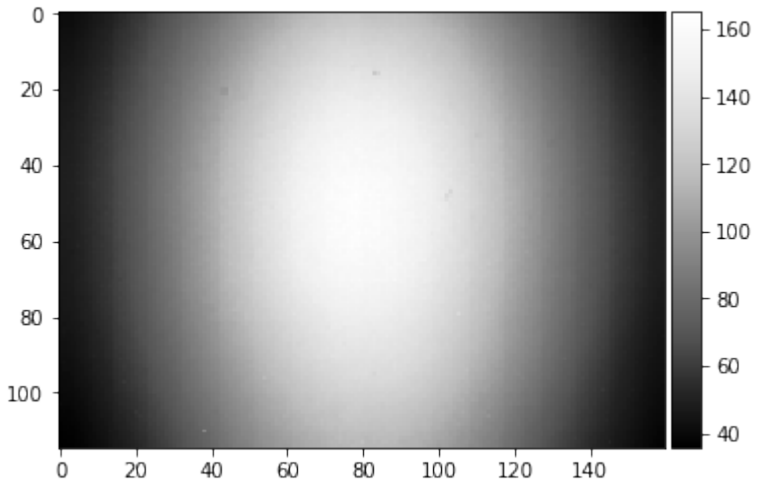
[11]:
```
#skimage.io.imsave('background_static.tiff', background_static, plugin='tifffile') # this␣
↪is 16bit only
np.savetxt('background_static.txt', background_static)
```

[12]:
```
# note the CCD halves and static background dust, hot pixels
processed_pattern = process_ebsp(raw_pattern, static_background=background_static,␣
↪binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_static_' + str(ipattern) + '.tiff', processed_pattern,
↪ plugin='tifffile')
```



Specification of Image Pre-Processing Functions

[13]:
```
prebinning=1
background_static_binned = downsample(background_static, prebinning)

def pipeline_process(pattern, prebinning=1, kikuchi=False):
    if prebinning>1:
        pattern = downsample(pattern, prebinning)
    if kikuchi:
        return process_ebsp(pattern, static_background=background_static_binned, binning=1)
    else:
        return pattern

def process_kikuchi(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=True)

def process_bin(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=False)

plot_image(background_static_binned)
print(background_static_binned.shape)
```

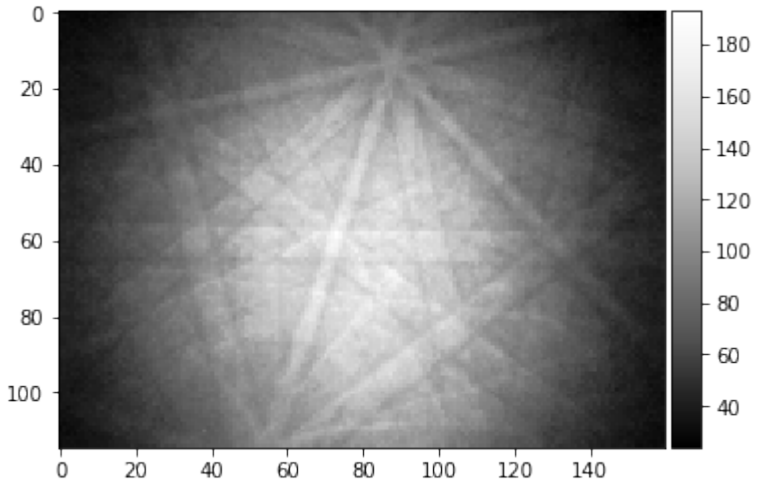


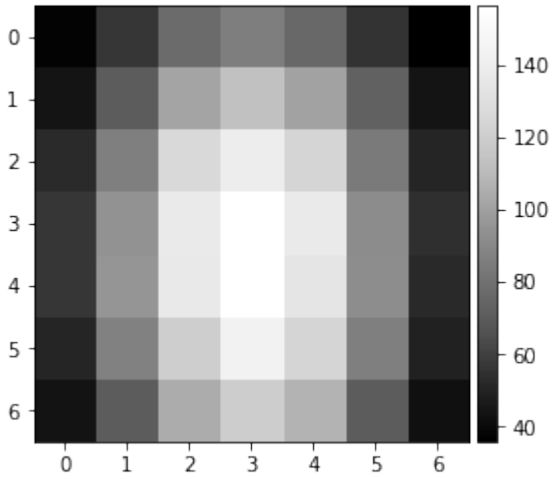

```
(115, 160)
```

vBSE Arrays

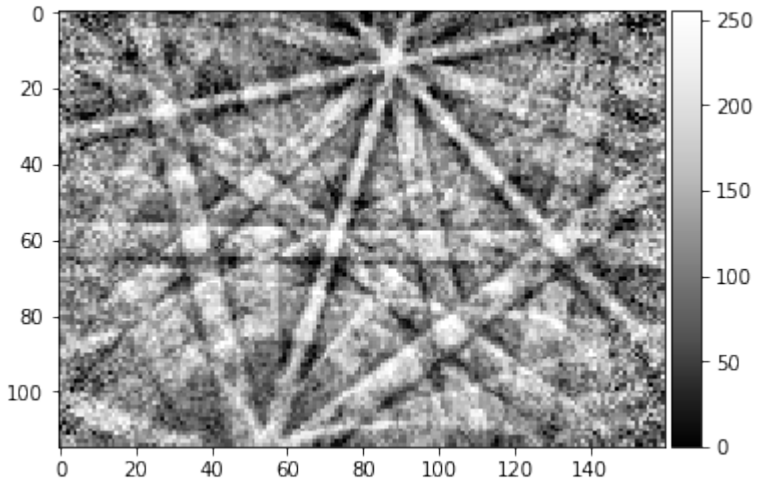
We convert the raw pattern into a 7x7 array of vBSE sensor intensities.

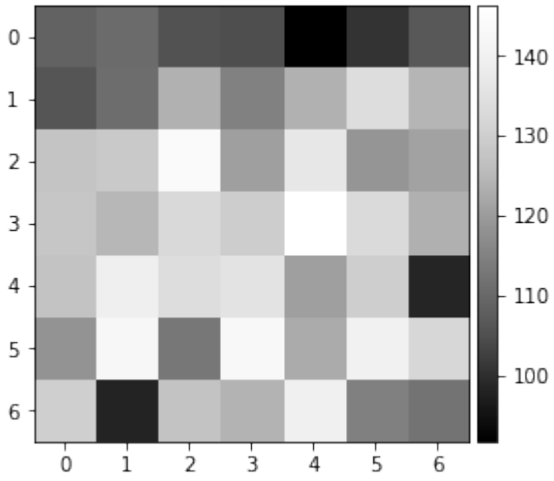
[14]:
```
pattern = pipeline_process(Patterns[1000], kikuchi=False)
vbse = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vbse)
```





[15]:
```
pattern = process_kikuchi(Patterns[1000])
vkiku = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vkiku)
```





vBSE Detector Signals: Calculation & Saving

This should take a few minutes, depending on your computer and file access speed.

Virtual BSE Imaging

Imaging the raw intensity in the respective area of the 2D detector (e.g. phosphor screen). Neglects gnomonic projection effect on intensities.

[16]:
```
# calculate the vBSE signals in 7x7 array
vbse_array = arbse.make_vbse_array(Patterns)

# make vBSE map of the total screen intensity
bse_total = np.sum(np.sum(vbse_array[:,:,:], axis=1), axis=1)
bse_map = make2Dmap(bse_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points:120000 current:120000 finished -> total calculation time :    0.7 min
```

[17]:
```
# save the results in the h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vbse', data=vbse_array)
    h5f.create_dataset('/maps/bse_total', data=bse_map)
```

```
arBSE_Steel_FCC_in_BCC_2752.h5
```

Virtual Orientation Imaging via Kikuchi Pattern Signals

If we process the raw images to obtain only the Kikuchi pattern, we have a modified 2D intensity which can be expected to show increased sensitivity to orientation effects (i.e. changes related to the Kikuchi bands). In a more advanced approach, we could select, for example, specific Kikuchi bands or zone axes to extract imaging signals.

[18]:
```
# calculate the vKikuchi signals from processed raw data
vkiku_array = arbse.make_vbse_array(Patterns, process=process_kikuchi)

# make vBSE map of the total screen intensity
kiku_total = np.sum(np.sum(vkiku_array[:,:,:], axis=1), axis=1)
kiku_map = make2Dmap(kiku_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points:120000 current:120000 finished -> total calculation time :    6.9 min
```

[19]:
```
# save the results in an extra hdf5
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vkiku', data=vkiku_array)
    h5f.create_dataset('/maps/kiku_total', data=kiku_map)
```

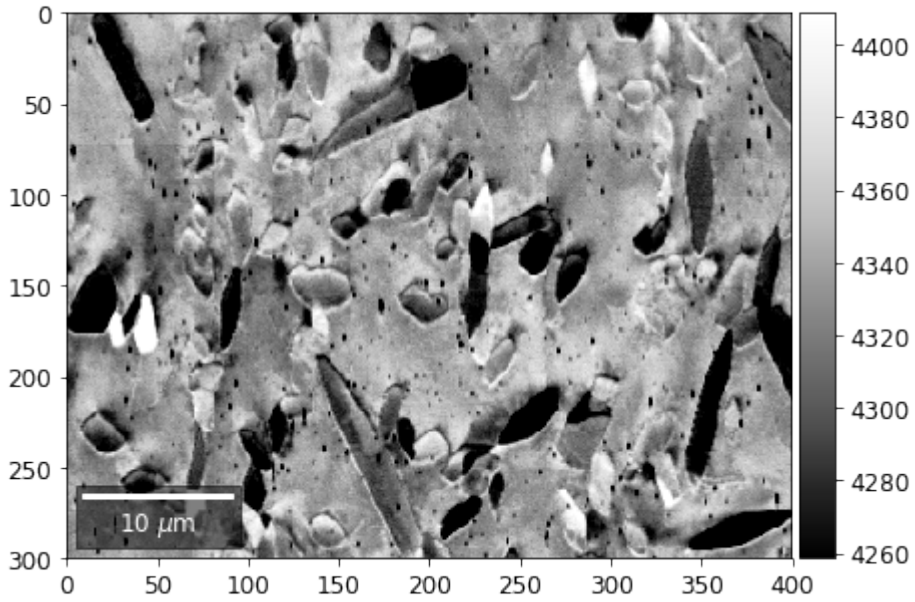
```
arBSE_Steel_FCC_in_BCC_2752.h5
```

vBSE Signals: Plotting

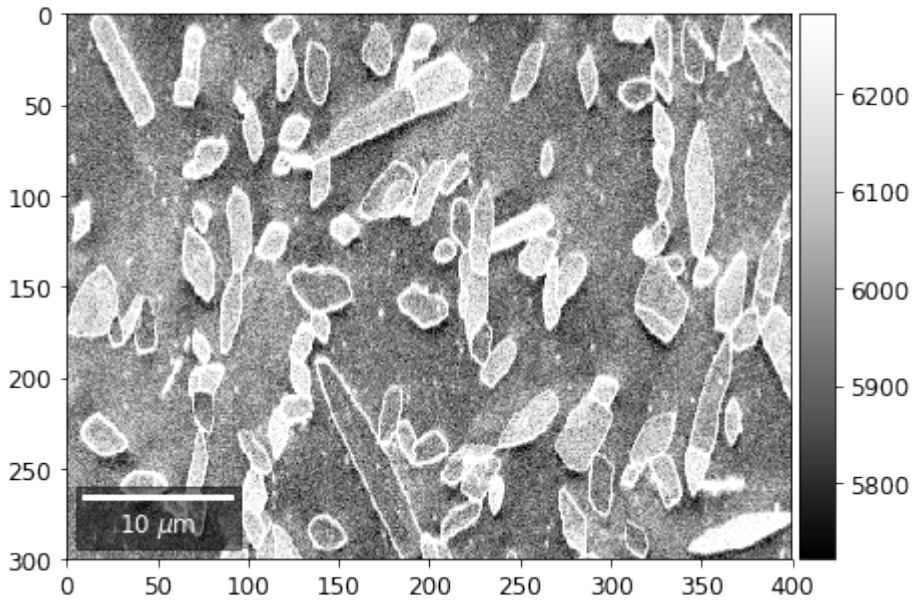
Total Signal on Screen

Total sum of the 7x7 arrays, for the raw pattern and the Kikuchi pattern at each map point:

[20]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    bse = h5f['/maps/bse_total']
    plot_SEM(bse, cmap='Greys_r', microns=step_map_microns)
```



[21]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    kiku = h5f['/maps/kiku_total']
    plot_SEM(kiku, cmap='Greys_r', microns=step_map_microns)
```



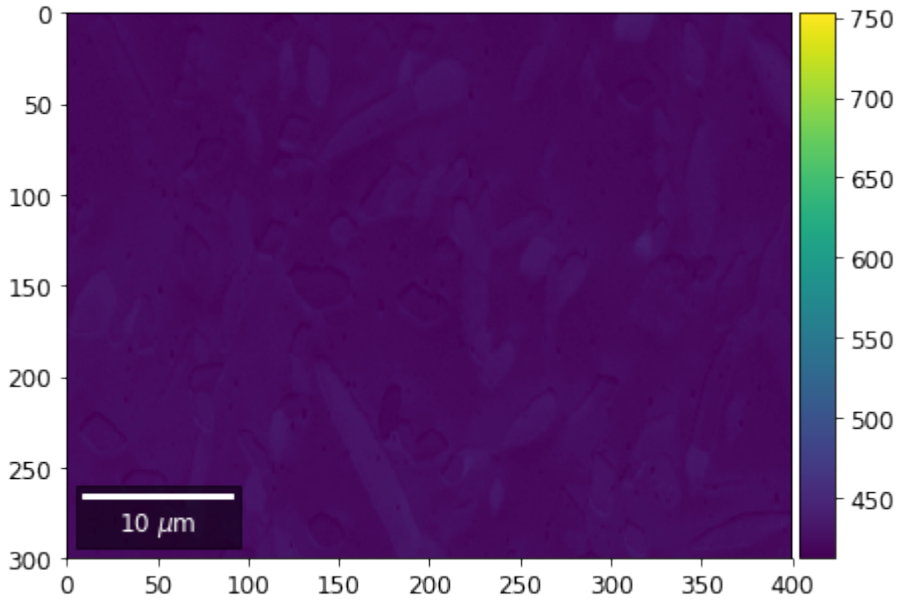
Intensity in Rows and Columns of the vBSE array

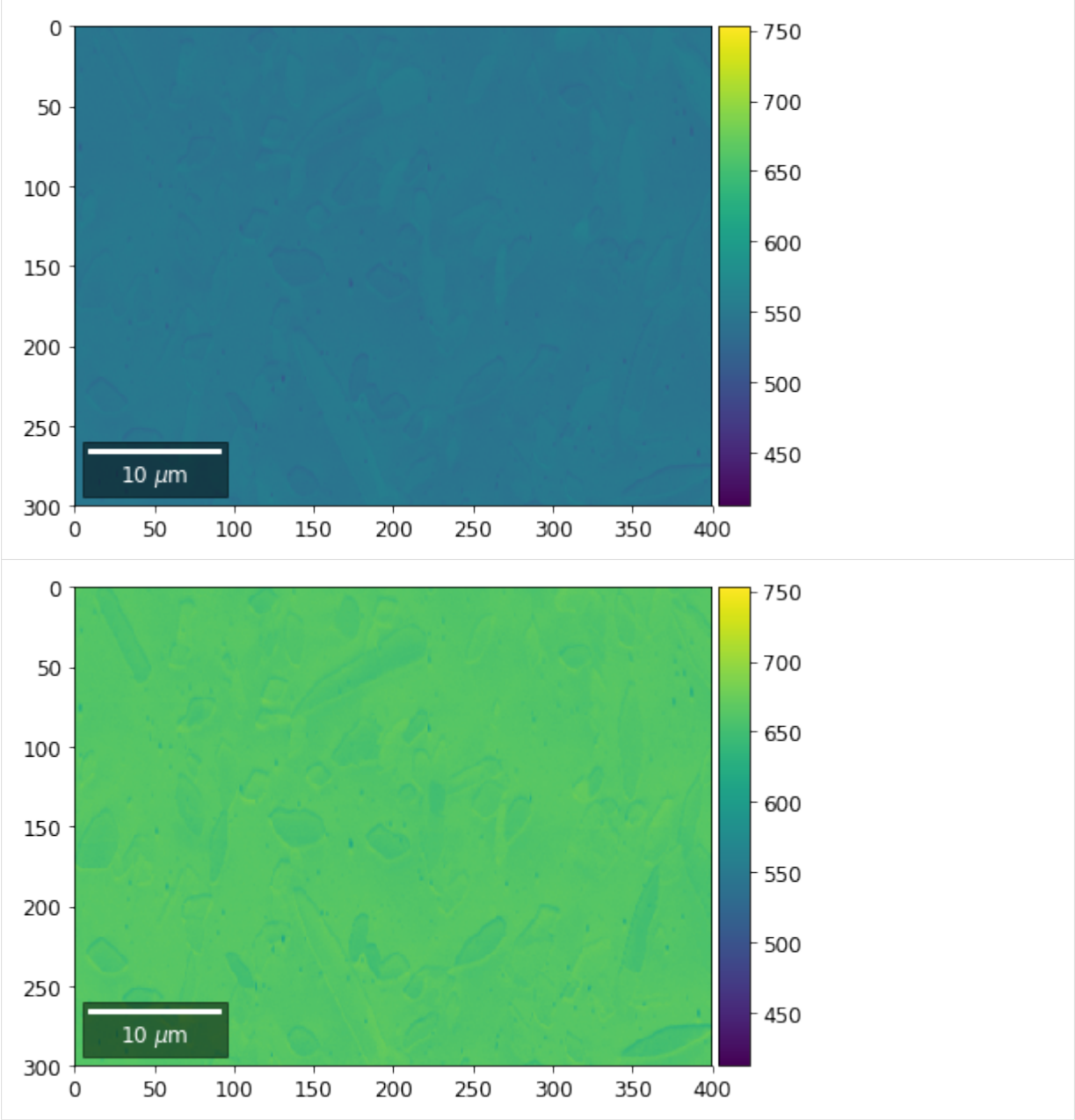
We can calculate additional images from the vBSE data set of 7x7 ROIs derived from the original patterns. As a first example, we plot the intensities of each of the 7 rows and then of each of the 7 columns:

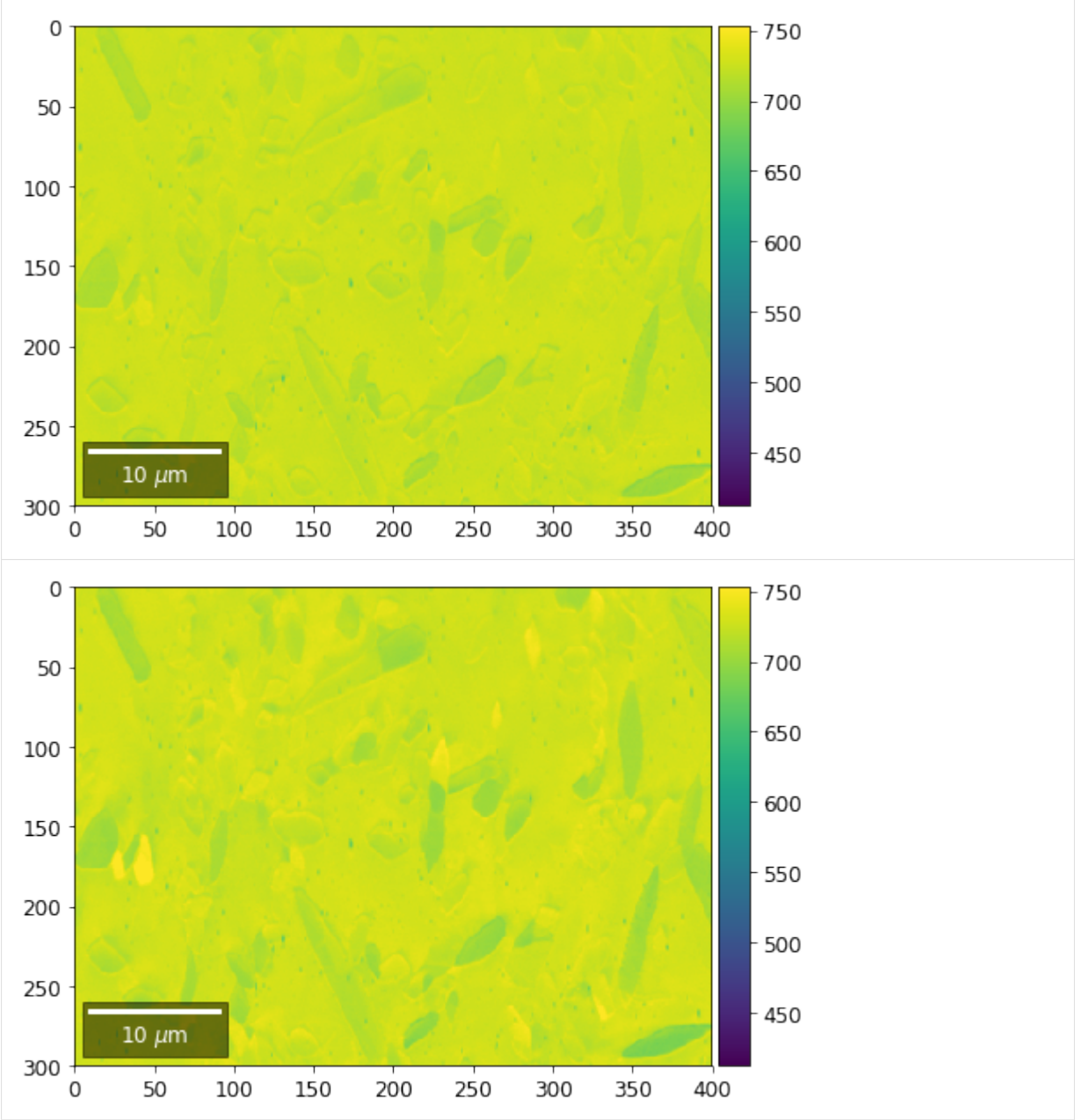
Rows

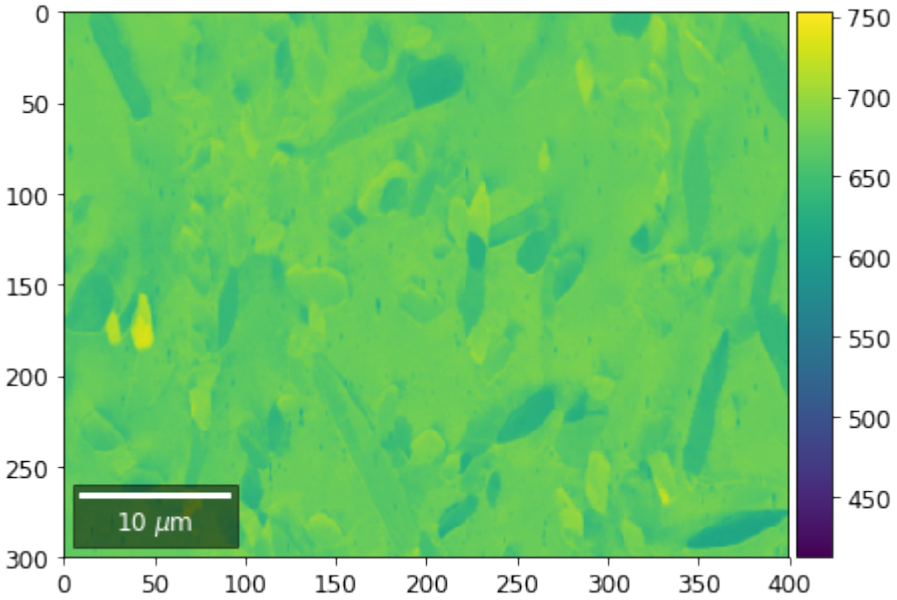
[22]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']

    # signal: sum of row
    vmin=40000000
    vmax=0

    bse_rows = []

    # (1) get full range for all images
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        minv, maxv = get_vrange(signal, stretch=3.0)
        if (minv<vmin):
            vmin=minv
        if (maxv>vmax):
            vmax=maxv

    # (2) make plots with same range for comparisons of absolute BSE values
    vrange=[vmin, vmax]
    print('Range of Values: ', vrange)
    #vrange=None
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_rows.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_absolute_'+str(row),
                 rot180=True, microns=step_map_microns)
```

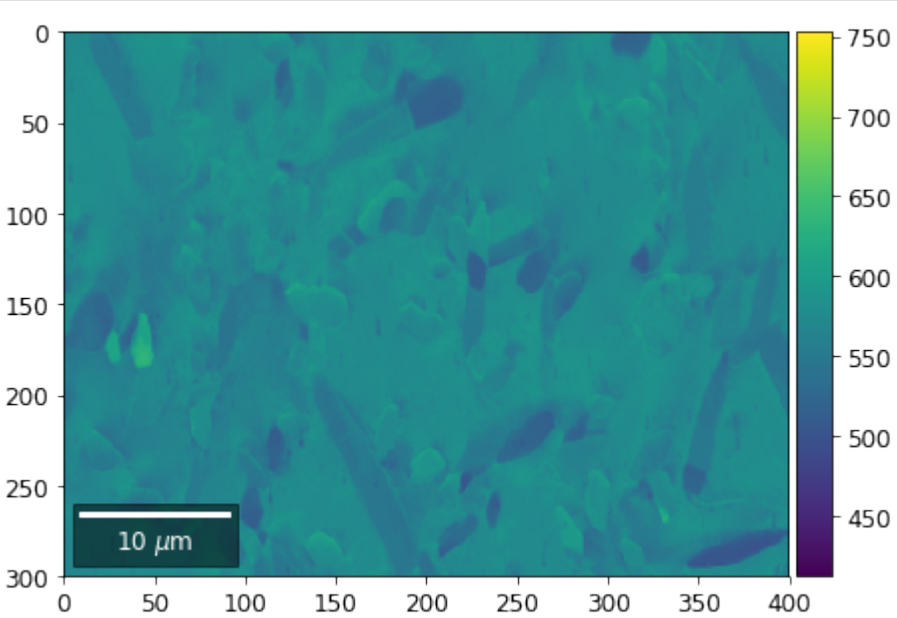
```
Range of Values:  [413.22316568814273, 753.16897562400345]
```



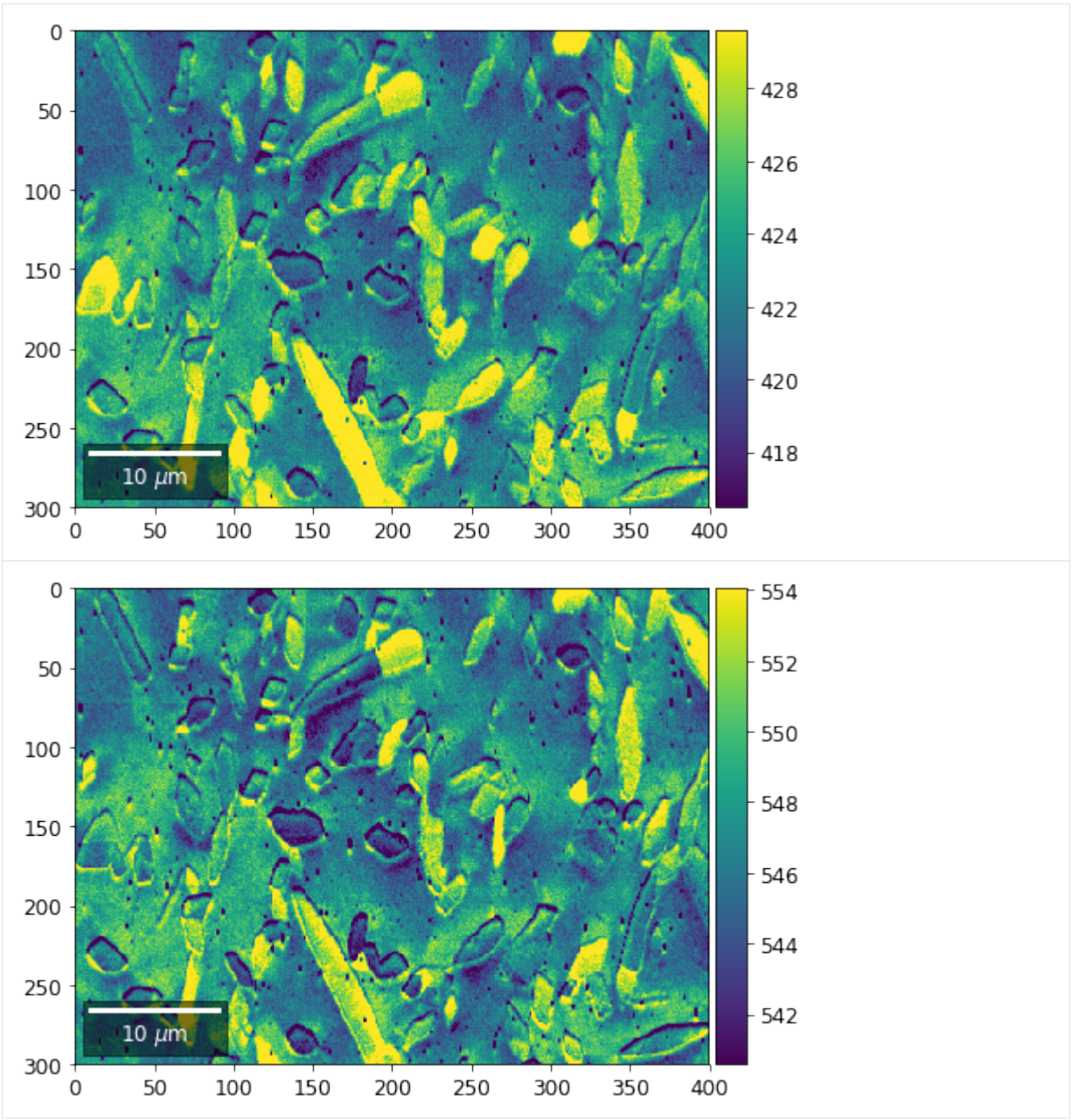

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
750
700
650
600
550
500
450

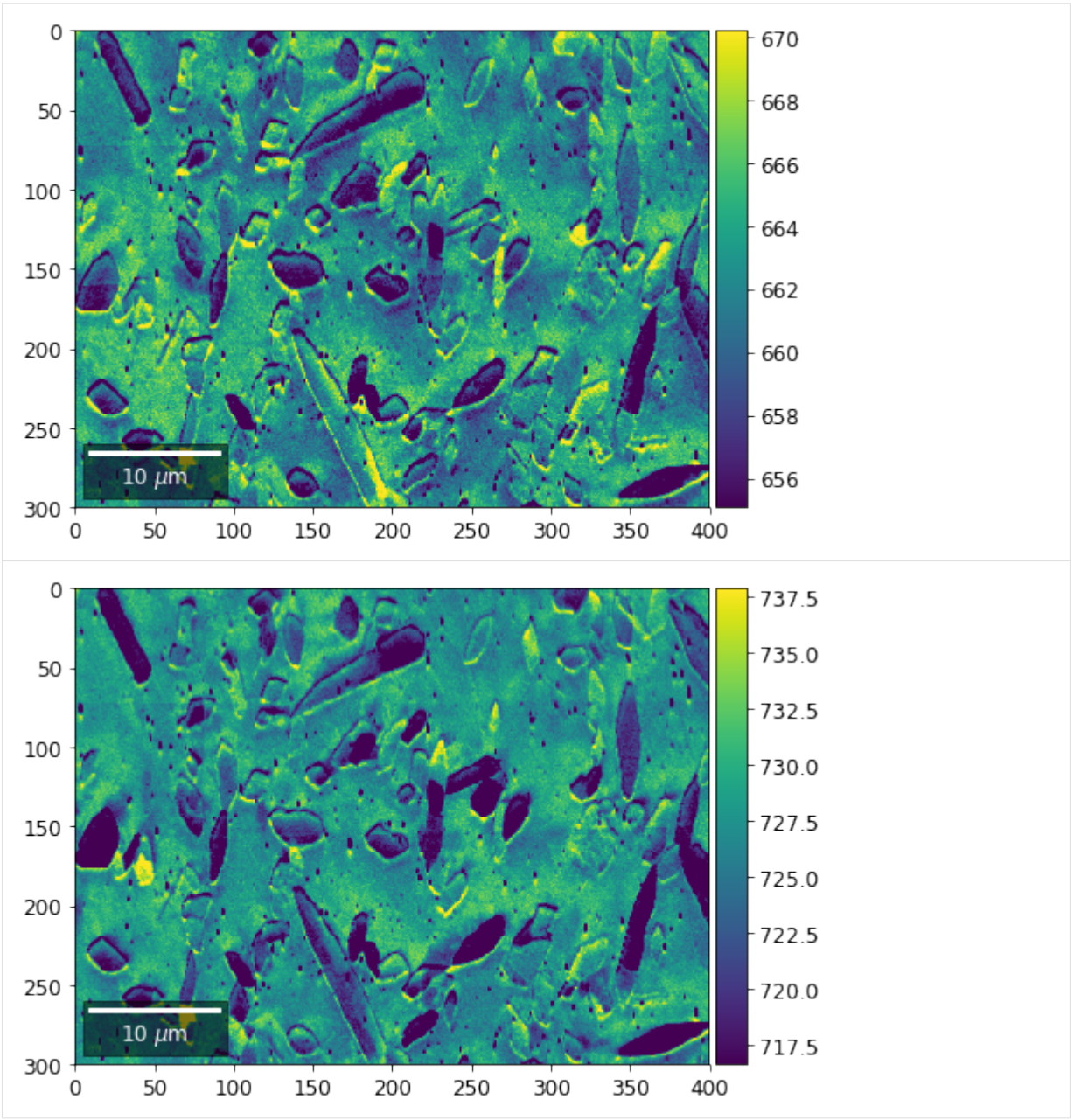

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
750
700
650
600
550
500
450

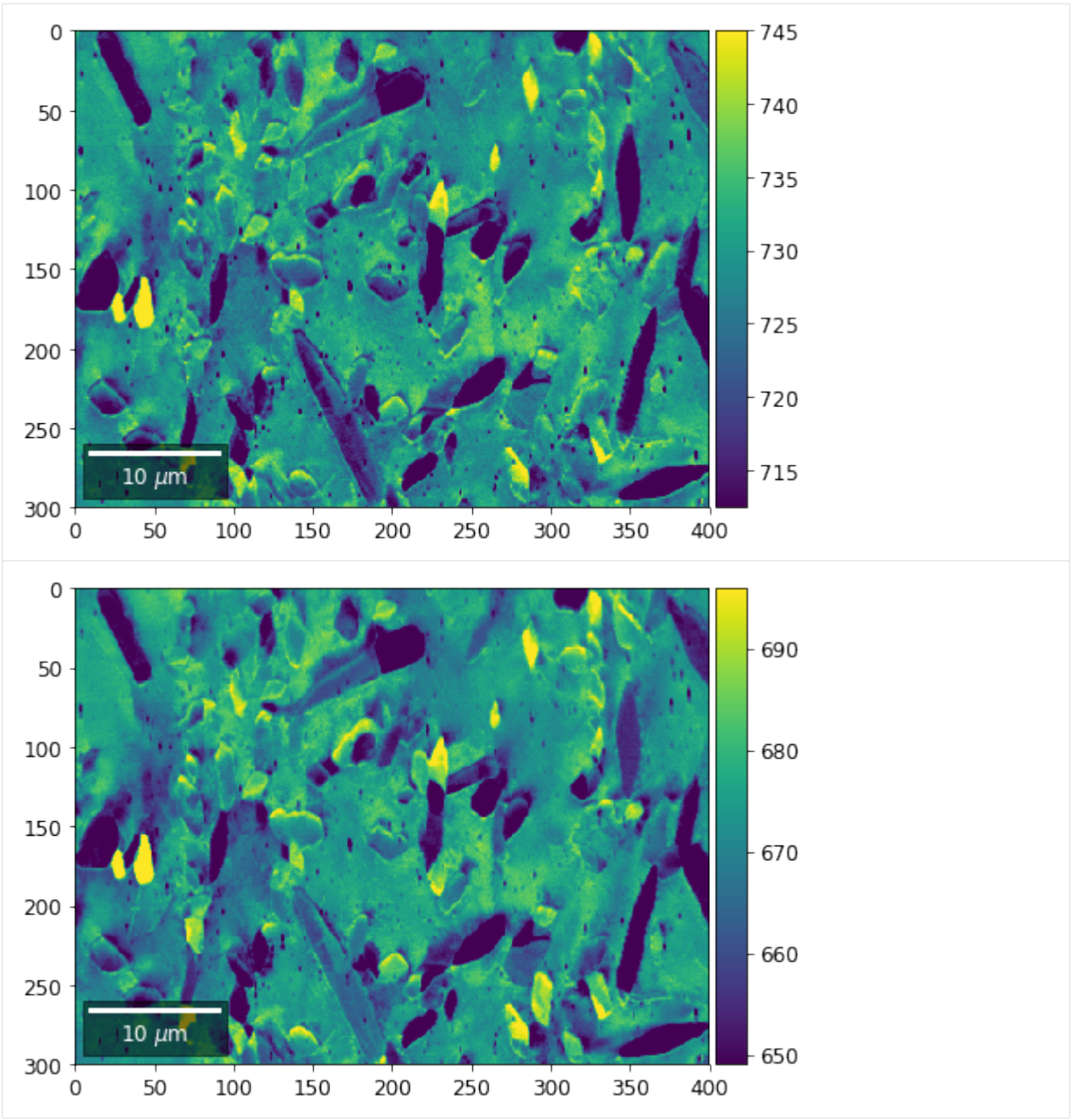


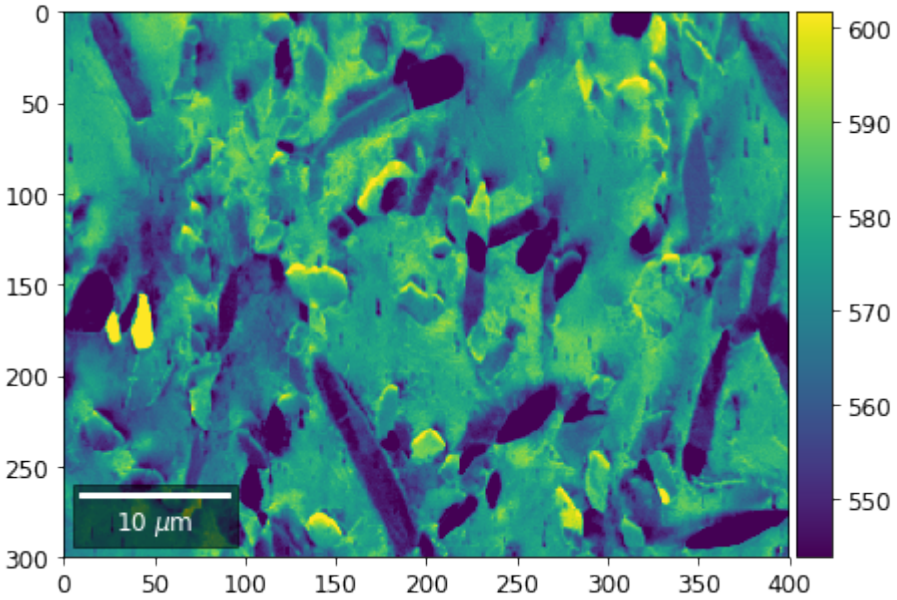


[23]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']
    # (3) make plots with  individual ranges for better contrast
    vrange=None
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_rows.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_individual_'+str(row),
                 rot180=True, microns=step_map_microns)
```


0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
428
426
424
422
420
418
554
552
550
548
546
544
542

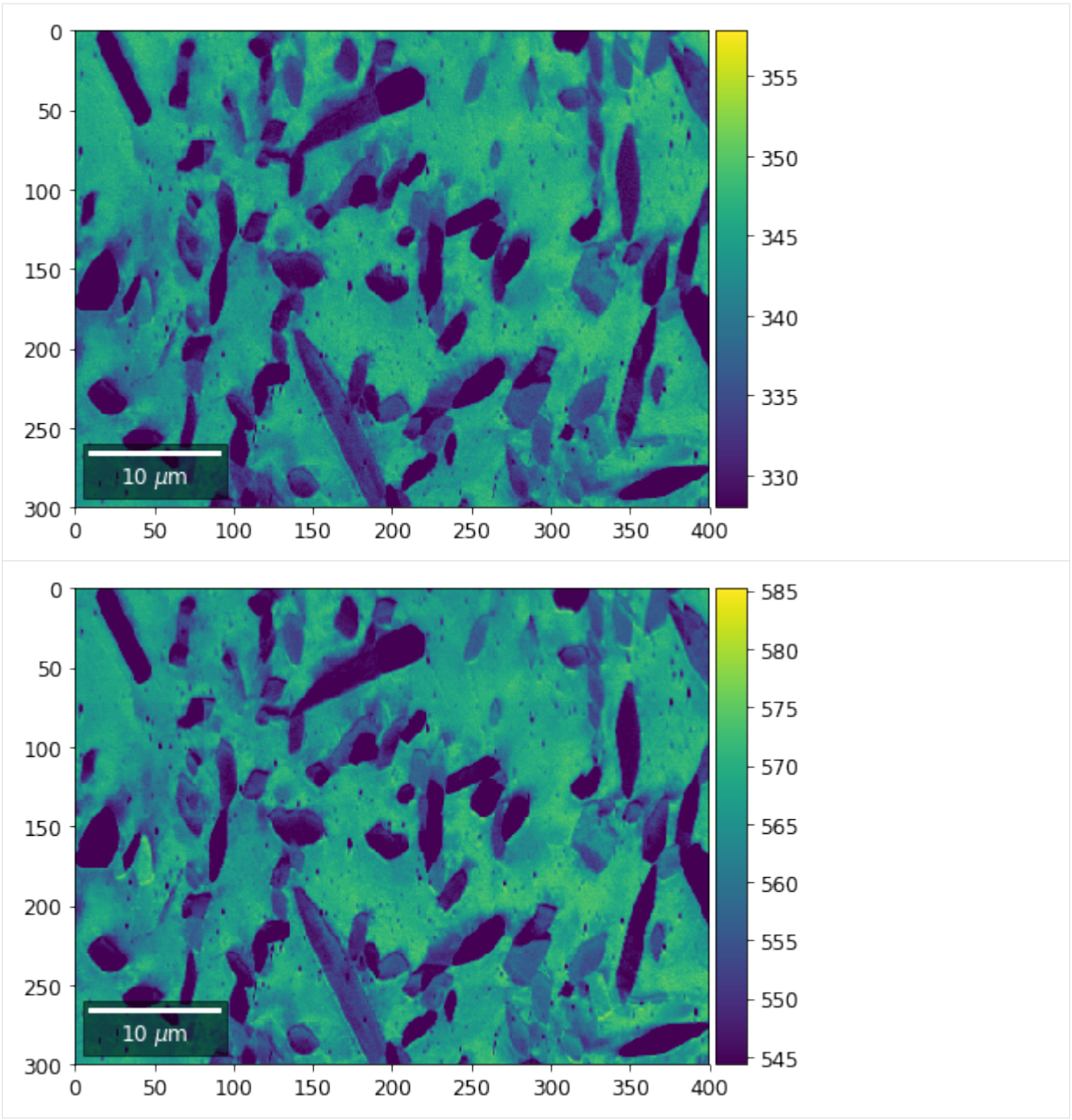

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
670
668
666
664
662
660
658
656
737.5
735.0
732.5
730.0
727.5
725.0
722.5
720.0
717.5

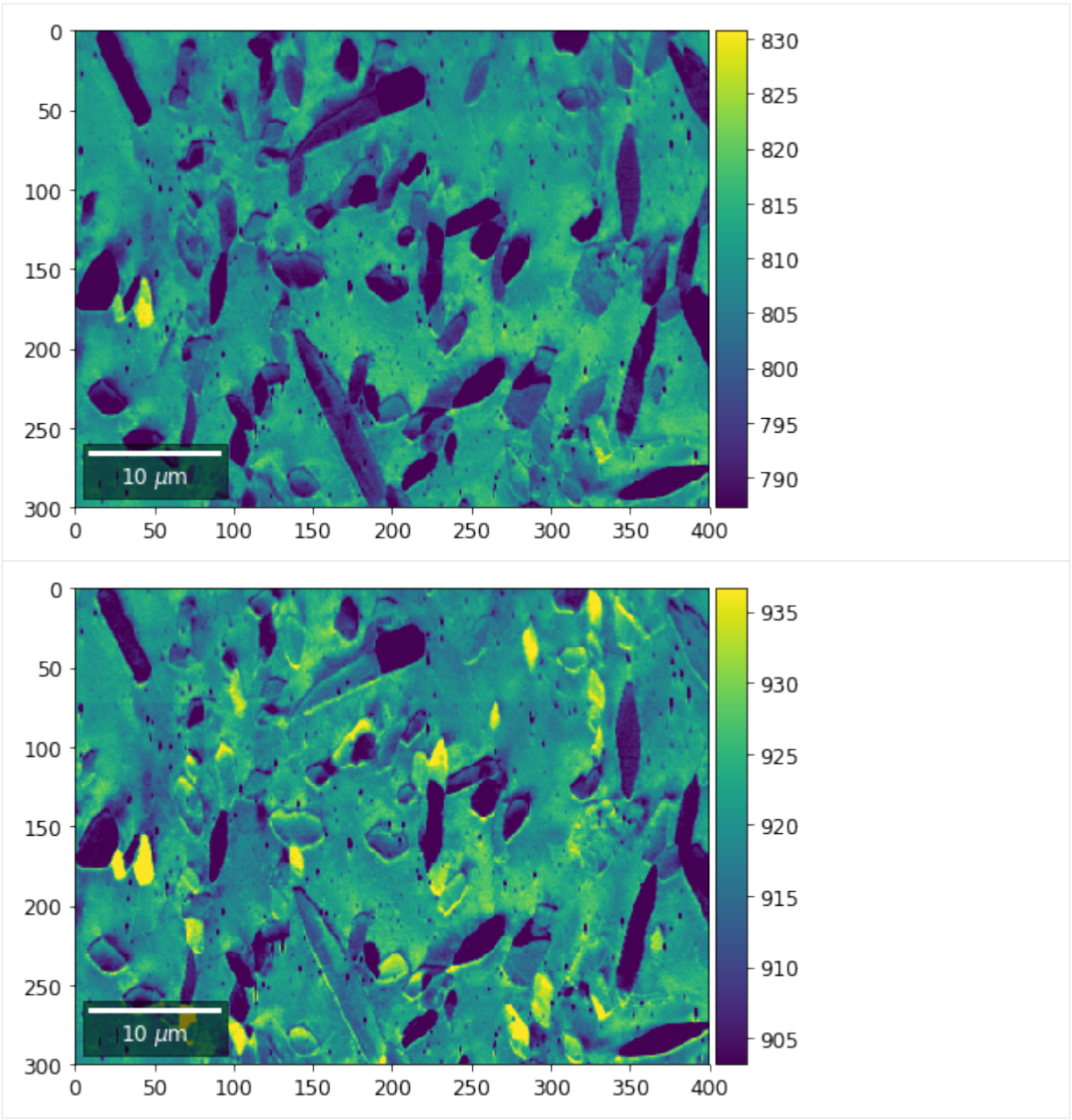

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
745
740
735
730
725
720
715
690
680
670
660
650

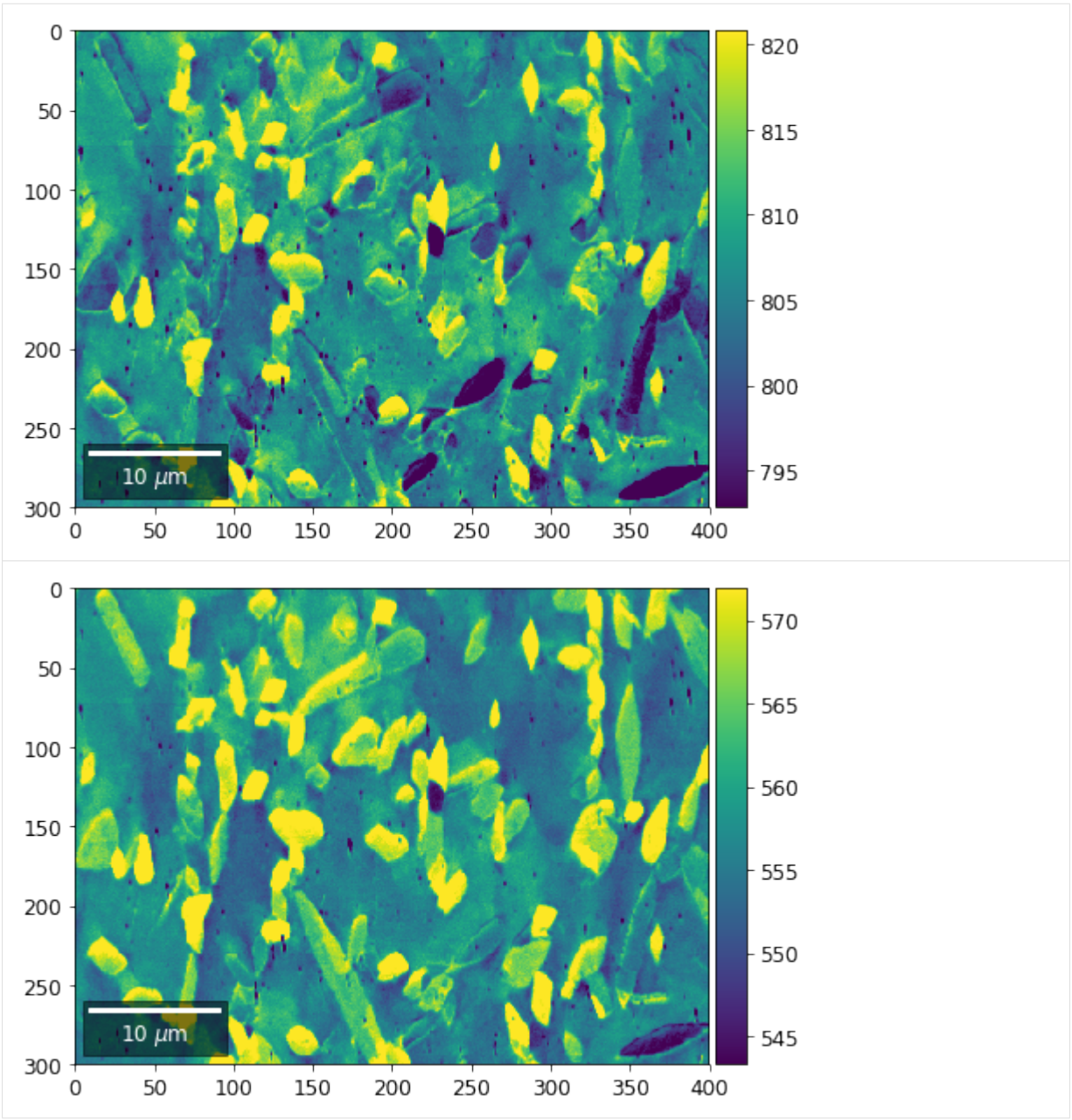


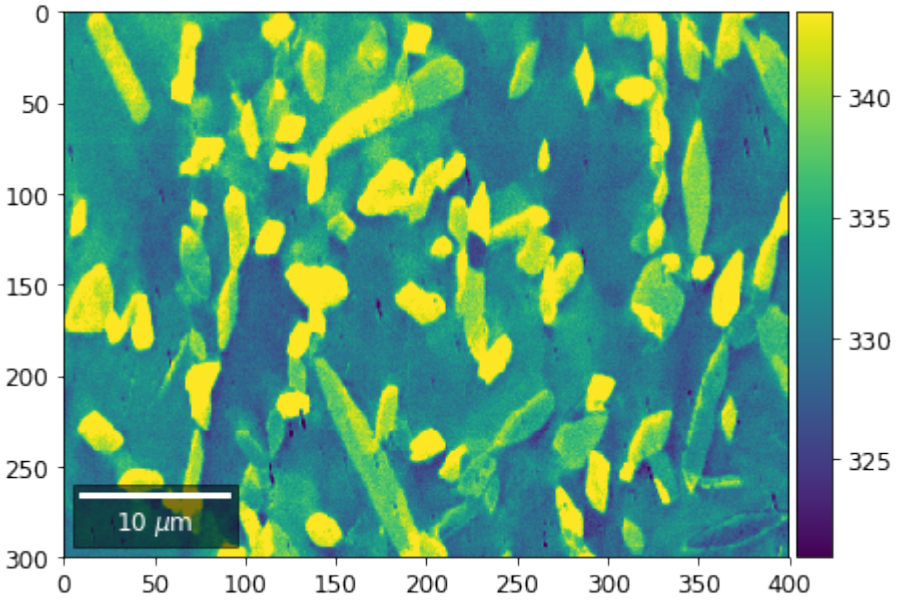
Columns

[24]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']
    # signal: sum of column
    vmin=400000
    vmax=0

    bse_cols = []

    # (1) get full range for all images
    for col in range(7):
        signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        minv, maxv = get_vrange(signal)
        if (minv<vmin):
            vmin=minv
        if (maxv>vmax):
            vmax=maxv

    # (2) make plots with same range for comparisons of absolute BSE values
    #vrange=[vmin, vmax]
    vrange=None # no fixed scale
    for col in range(7):
        signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_cols.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_col_'+str(col),
                 rot180=True, microns=step_map_microns)
```


0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
355
350
345
340
335
330
585
580
575
570
565
560
555
550
545


0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
830
825
820
815
810
805
800
795
790
935
930
925
920
915
910
905

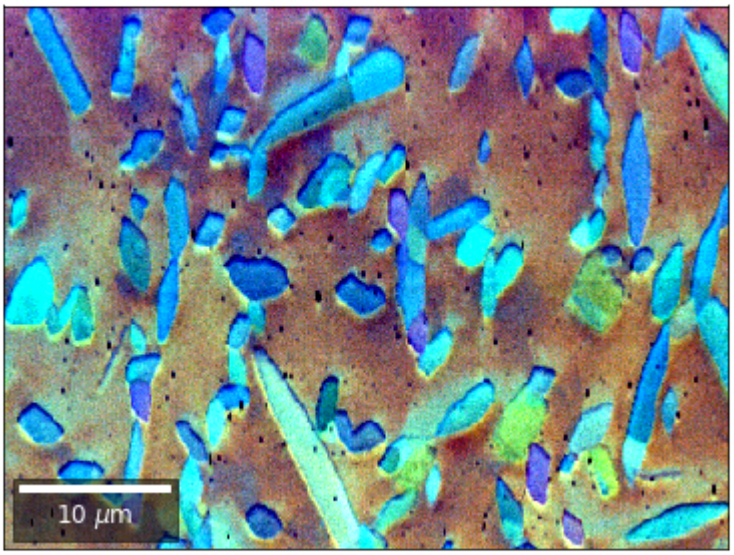

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
820
815
810
805
800
795
10 μm
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
570
565
560
555
550
545
10 μm

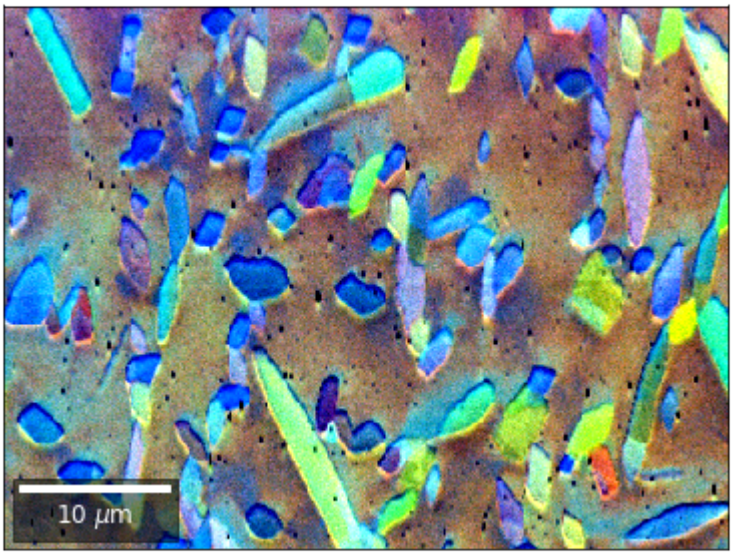


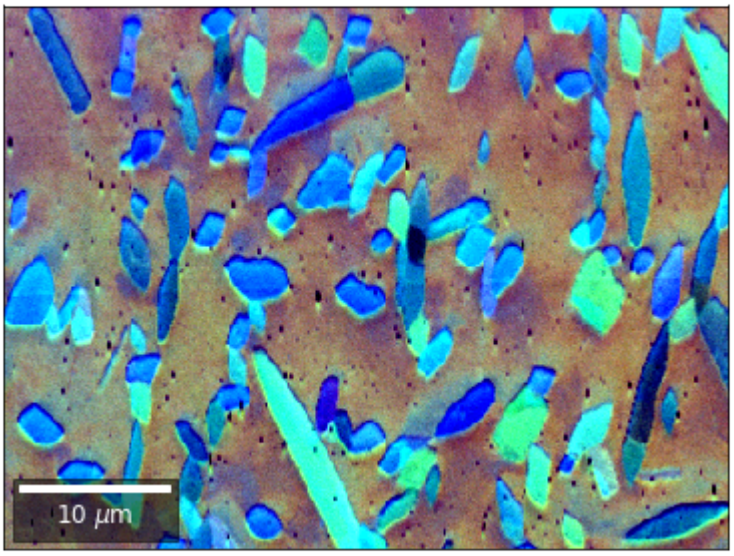
vBSE Color Imaging

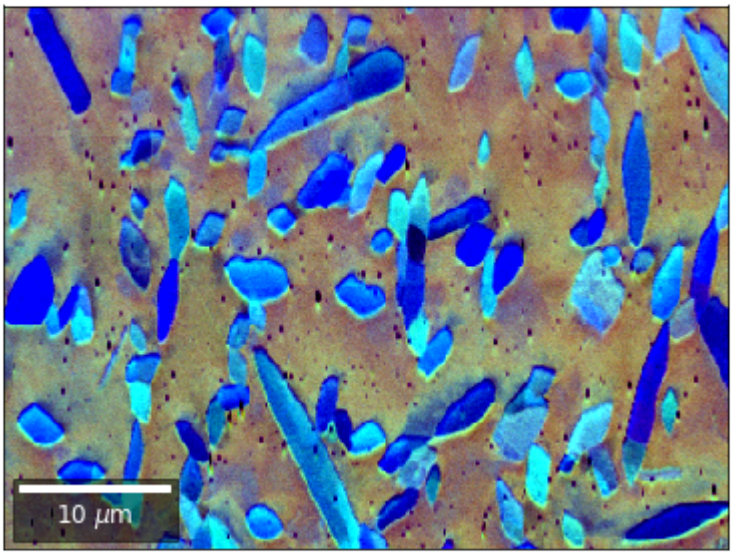
We can also form color images by assigning red, green, and blue channels to the left, middle, and right vBSE sensors of a row:

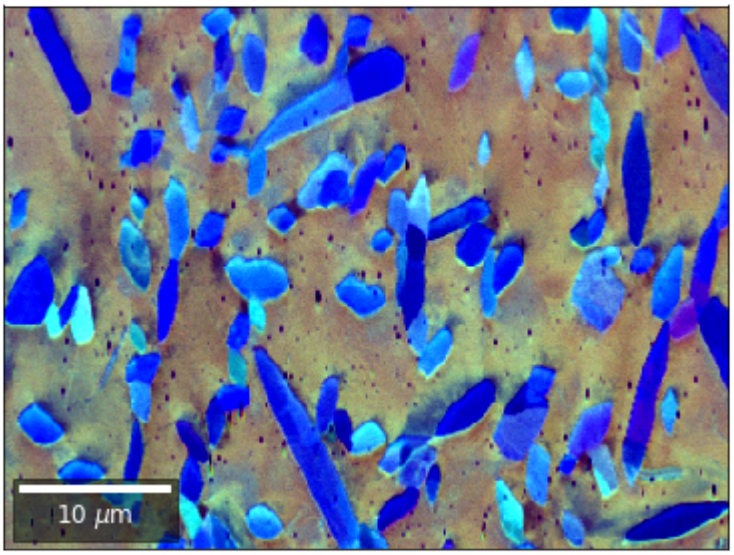
[25]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']# rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vFSD_RGB_row_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=0.8)

        rgb_direct.append(rgb)
```

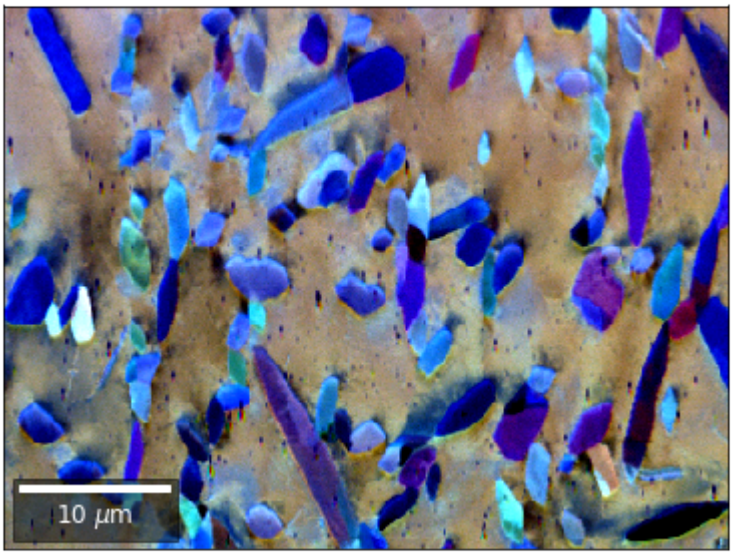

10 µm

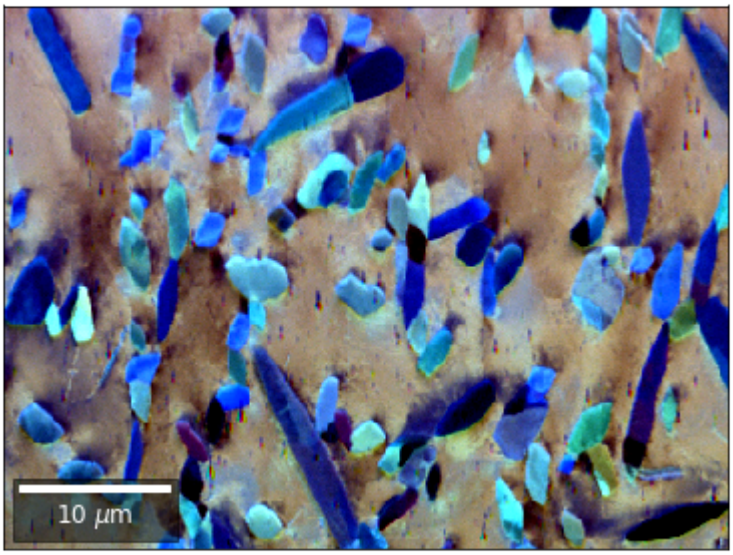

10 µm


10 µm


10 µm

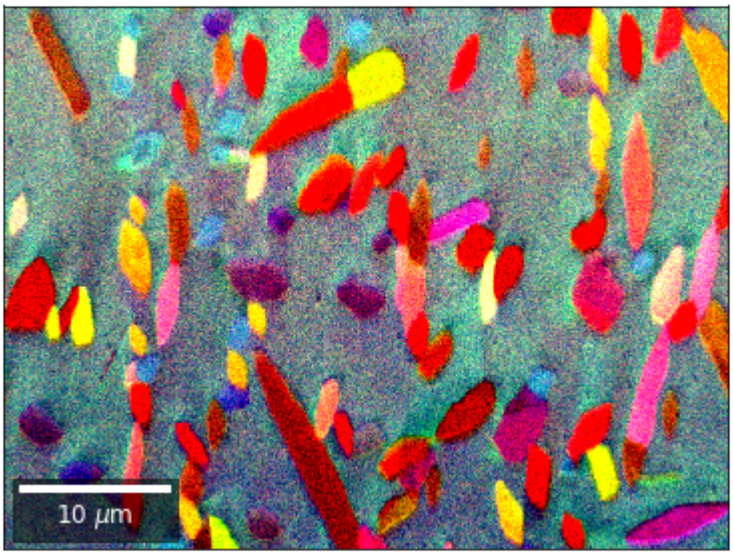

10 µm

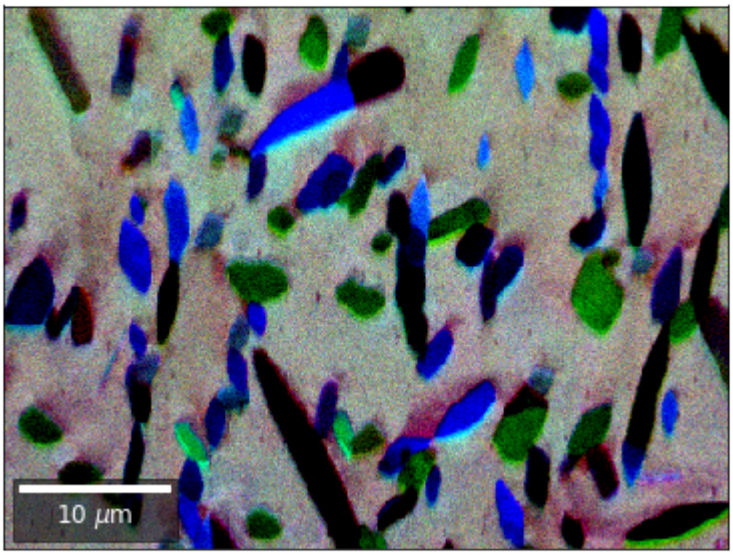

10 µm

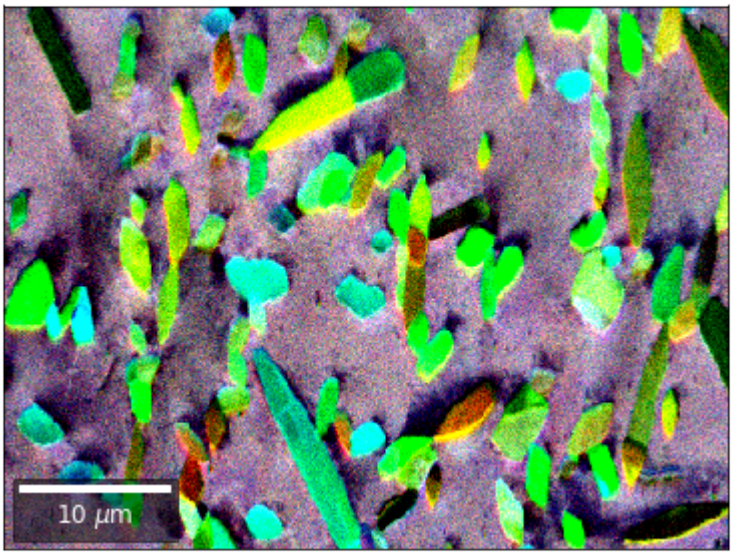


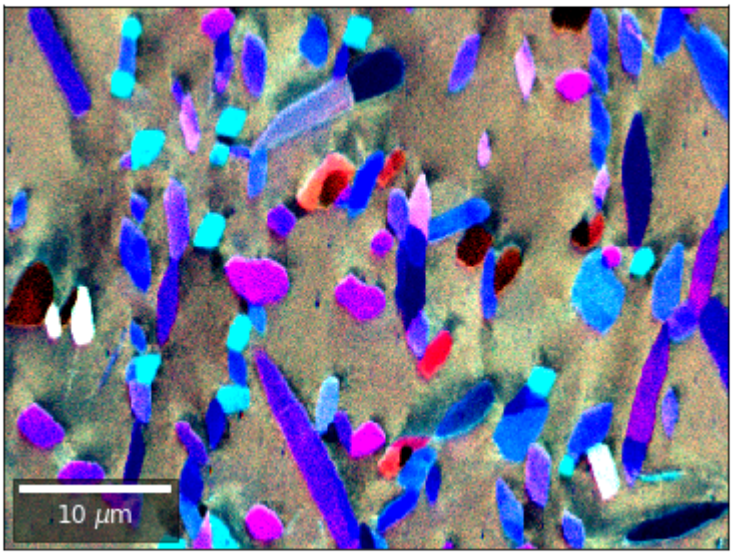
Differential color signals can be formed by calculating the relative changes to the ROI in the previous row:

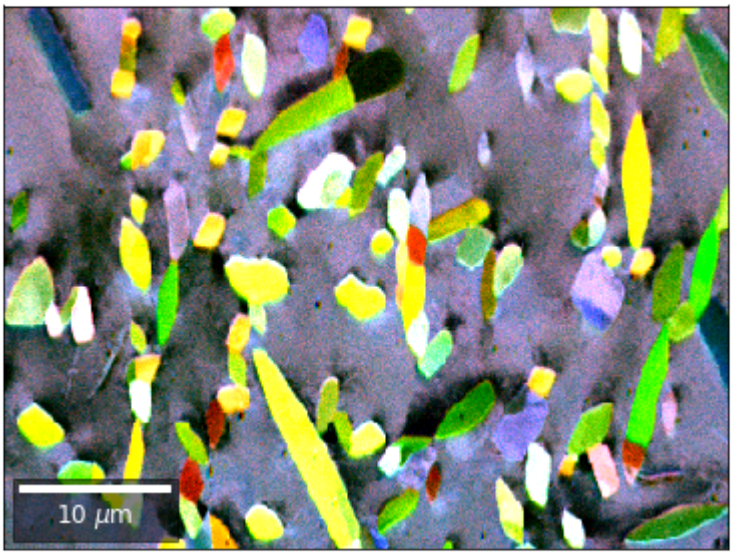
[26]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']# rgb direct# relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vFSD_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

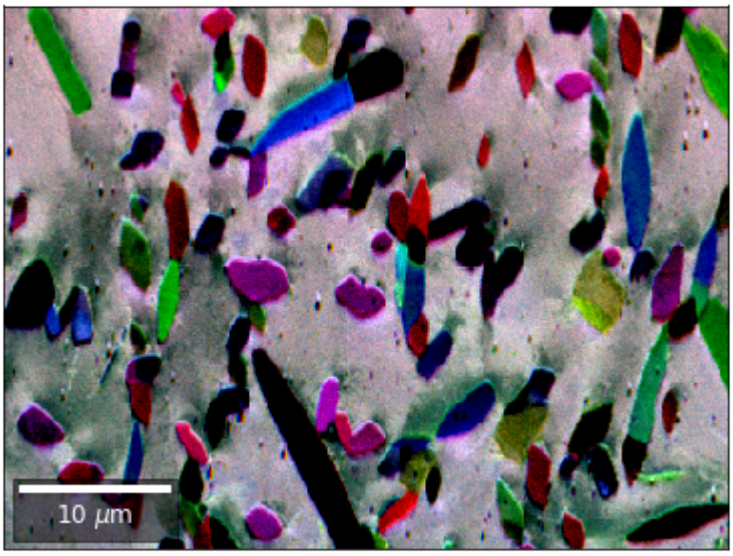

10 μm


10 μm


10 μm


10 μm


10 μm

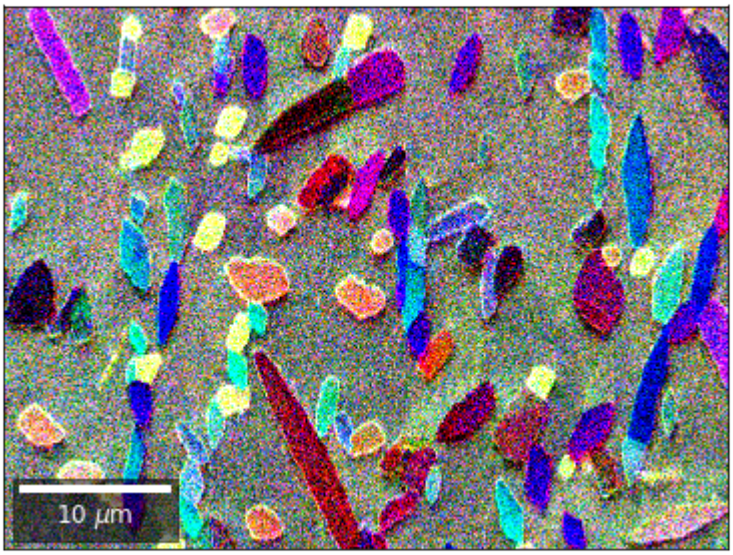

10 μm

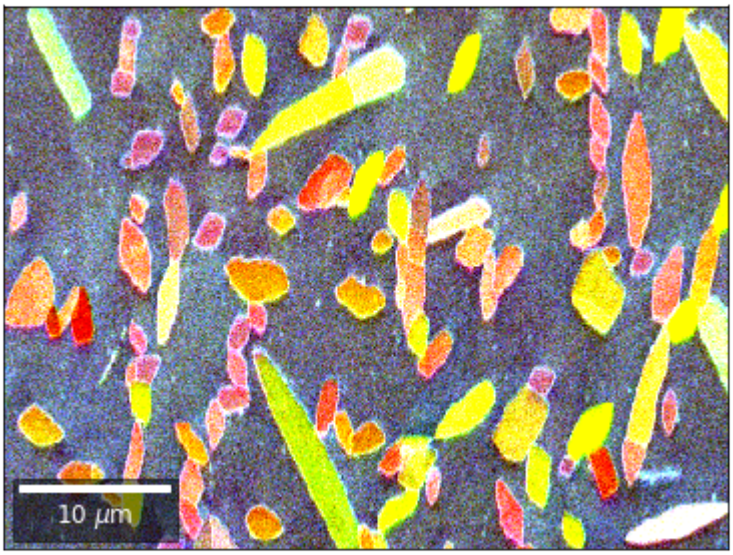
Kikuchi vBSE Imaging

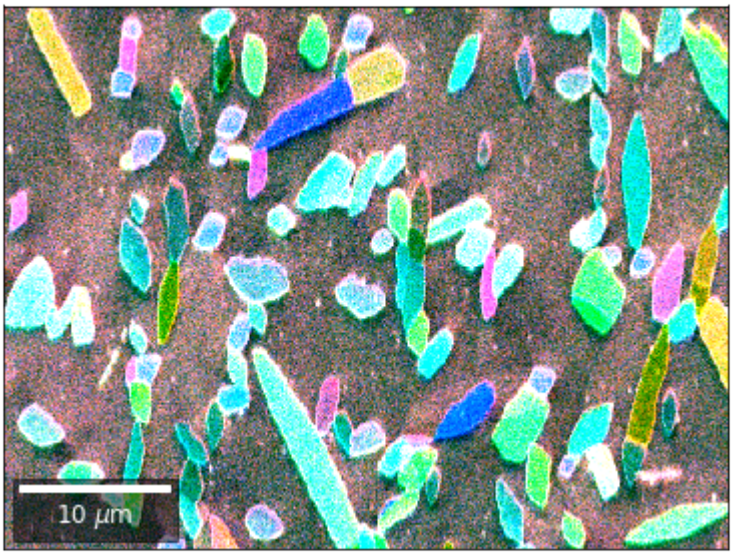
Kikuchi Pattern Array ROI as RGB

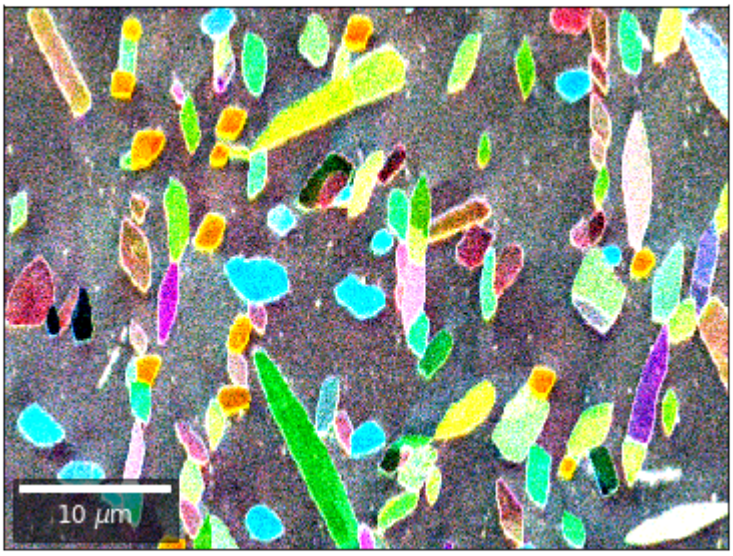
Not possible with simple BSE diodes!!!

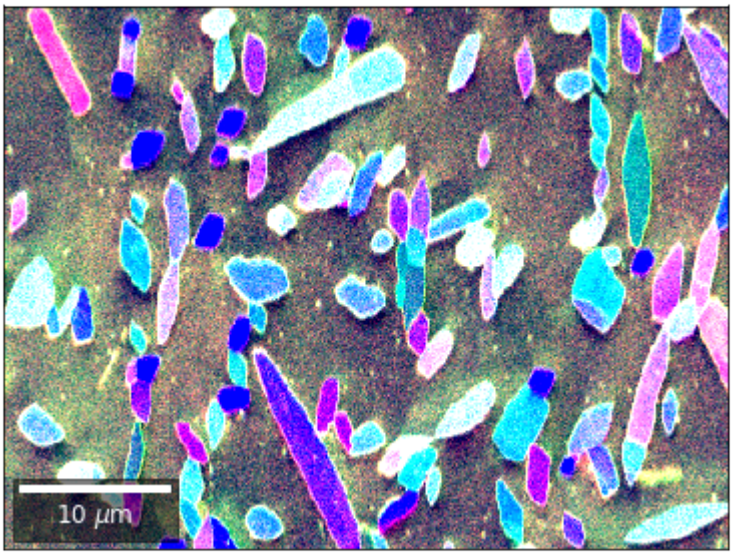
[27]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vkiku']

    # rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vKiku_RGB_row_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=1.2)

        rgb_direct.append(rgb)
```

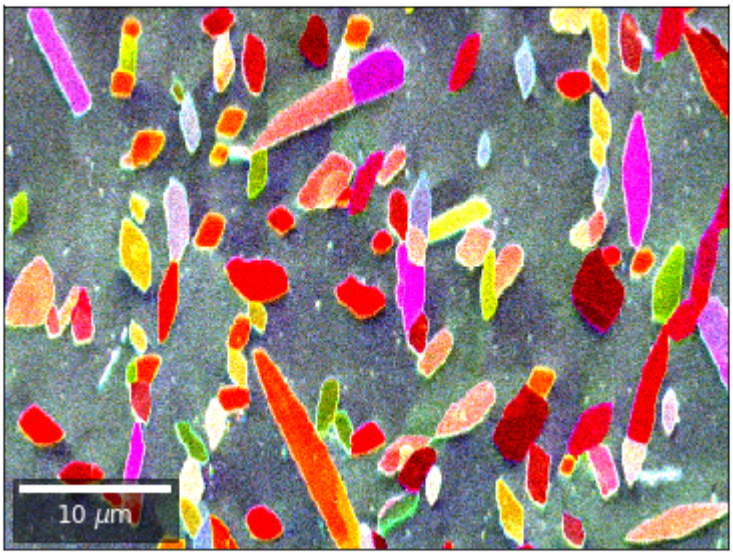


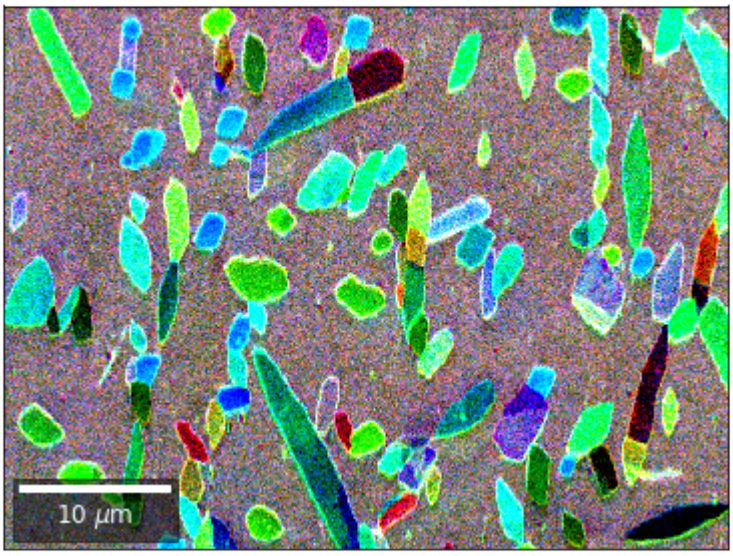

10 µm


10 µm


10 µm


10 μm


10 μm

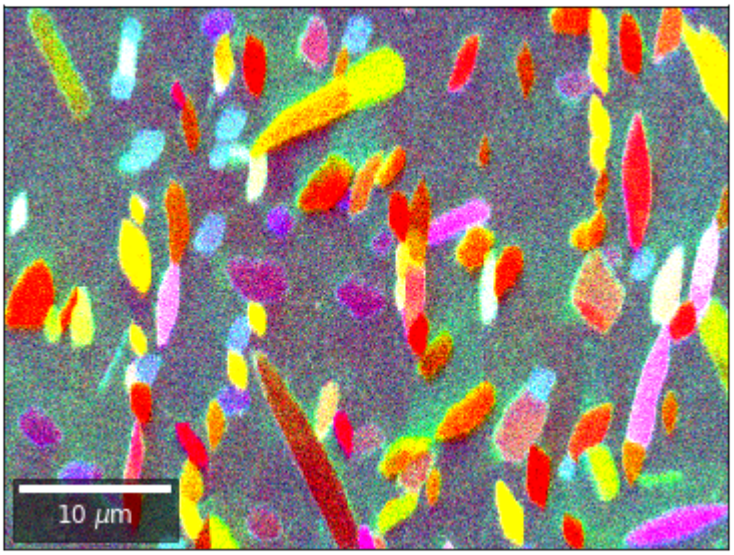

10 μm

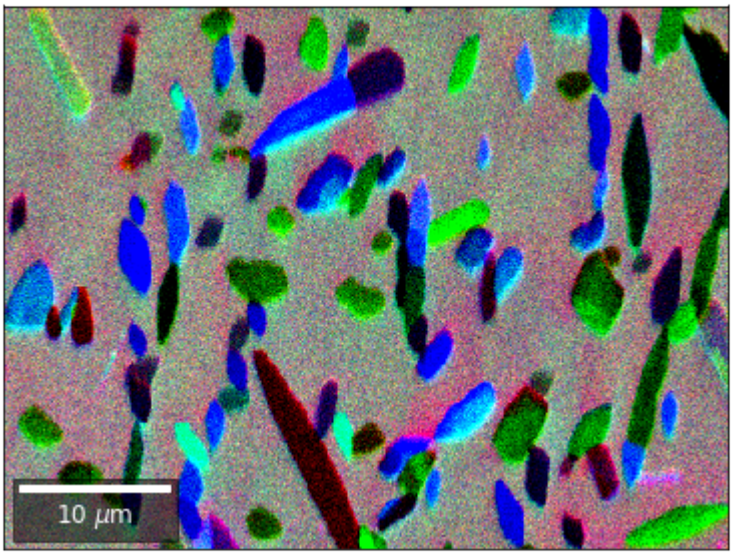
Differential Kikuchi Imaging

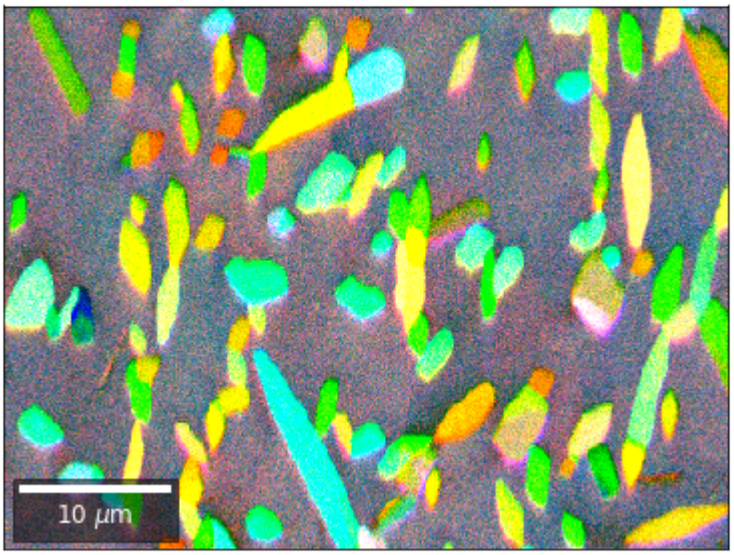
We determine the relative change between Kikuchi Array ROIs and use them as RGB values. The normalization to a reference ROI reduces the noise that is purely due to the variation of the background and the background processing on the complete pattern.

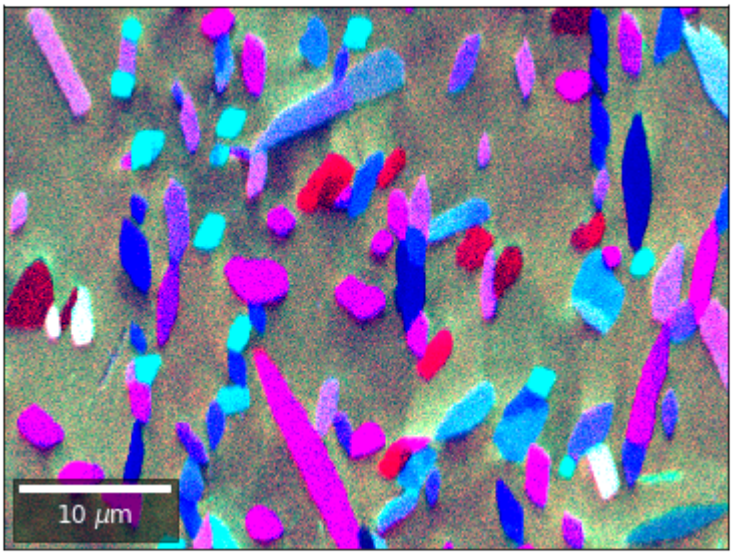
The colors represent orientation changes via the corresponding changes in ROIs of the Kikuchi patterns and the 7x7 array.

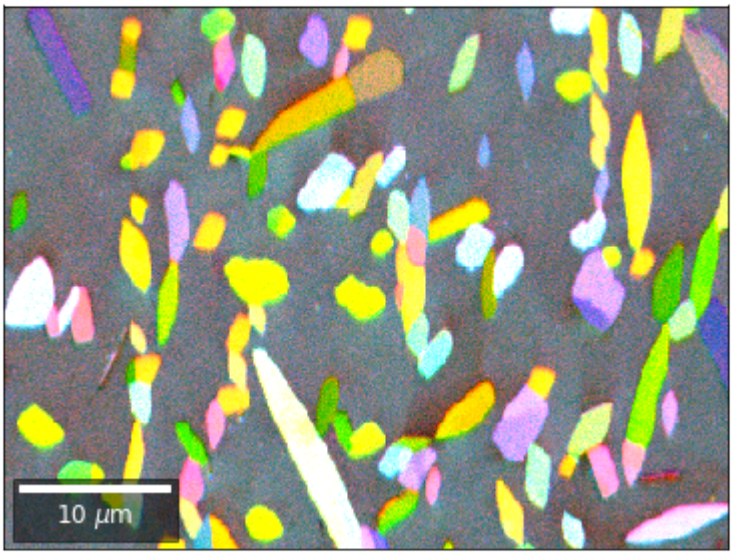
[28]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vkiku']
    # relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vKiku_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

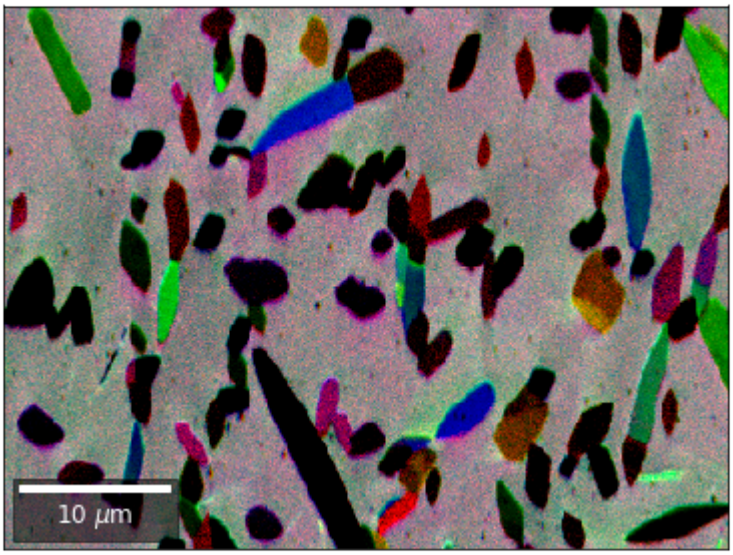



10 μm


10 μm


10 μm





Center of Mass Imaging

We can interpret the 2D image intensity as a mass density on a plane. The statistical moments of the density distribution (mean, variance, ...) can be used as signal sources. In the example below, we use the image center of mass as a signal source.

COM of Raw Patterns

[29]:
```
# calculate the center-of-mass for each pattern, use binning for speed
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_bin)
```

```
total points:120000 current:120000 finished -> total calculation time :    0.6 min
```

[30]:
```
# save the results in h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
```

(continues on next page)

(continued from previous page)

```
    h5f.create_dataset('/COM/COMxp_vbse', data=COMxp)
    h5f.create_dataset('/COM/COMyp_vbse', data=COMyp)
```

```
arBSE_Steel_FCC_in_BCC_2752.h5
```

COM of Kikuchi Patterns

This should be seen with caution, as the background removal process is never perfect and will tend to leave some residual intensity, so that the Kikuchi COM is correlated with the raw pattern COM (which is dominated by the smooth background intensity).

[31]:
```
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_kikuchi)
```

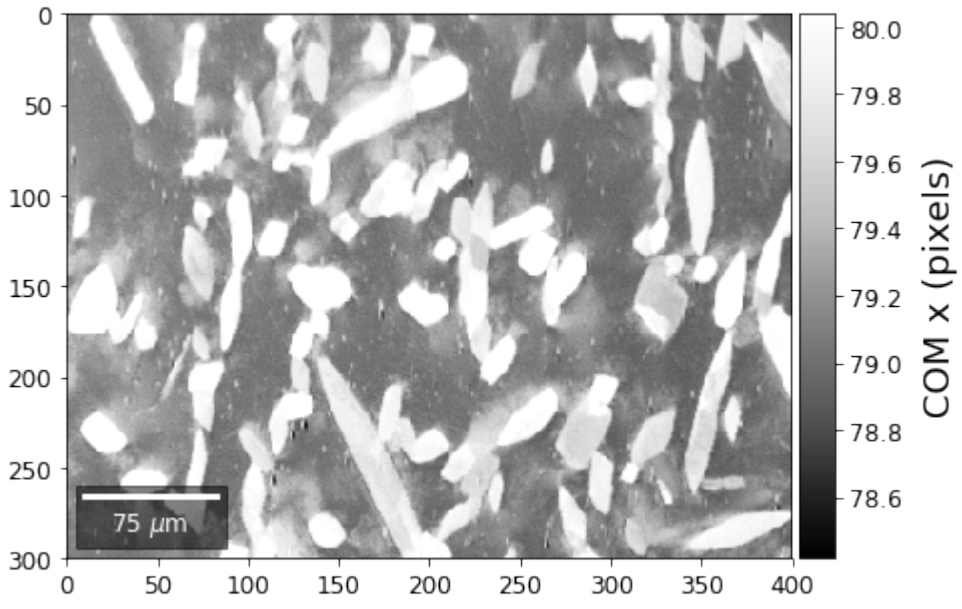
```
total points:120000 current:120000 finished -> total calculation time :    4.8 min
```

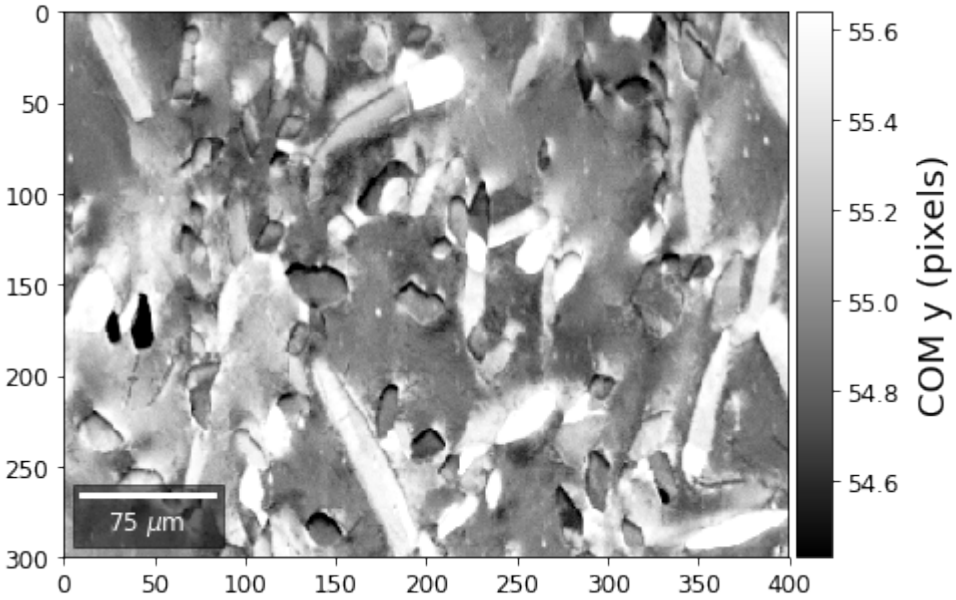
[32]:
```
# append to current h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('/COM/COMxp_kiku', data=COMxp)
    h5f.create_dataset('/COM/COMyp_kiku', data=COMyp)
```

```
arBSE_Steel_FCC_in_BCC_2752.h5
```

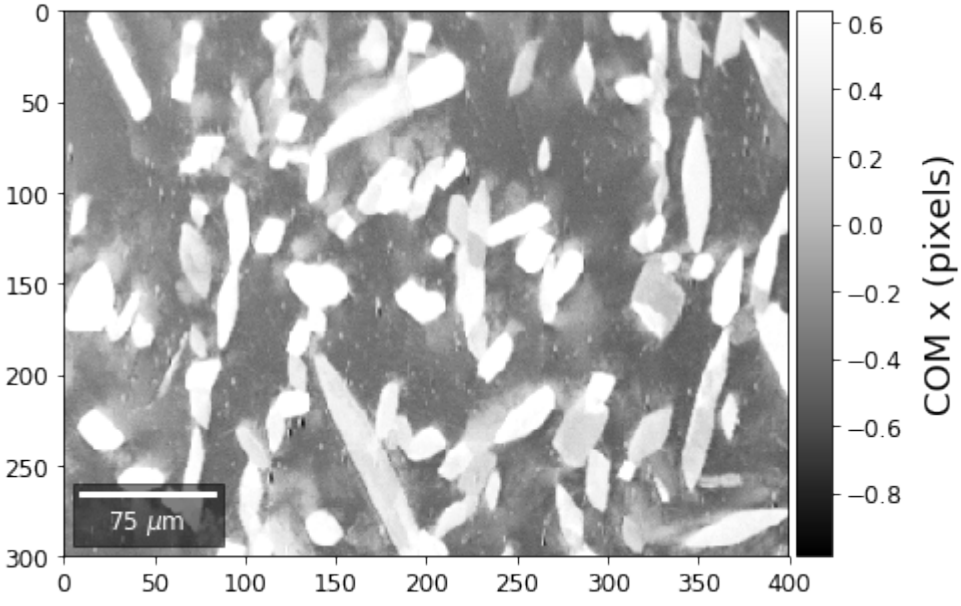
First, we calculate where the COMs are in x,y in pixels in the patterns:

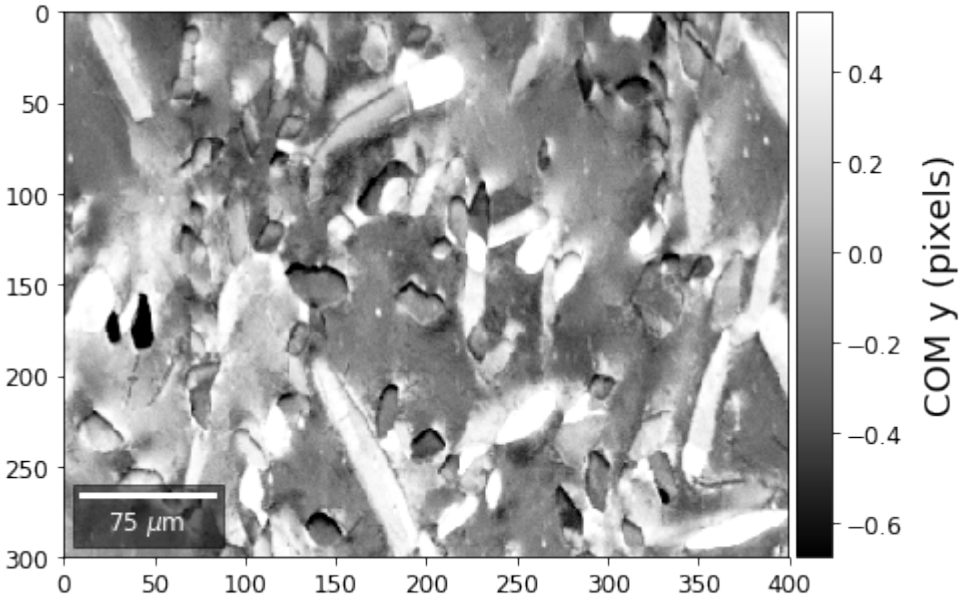
[33]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_vbse']
    COMyp = h5f['/COM/COMyp_vbse']

    comx_map0=make2Dmap(COMxp[:],XIndex,YIndex,MapHeight,MapWidth)
    comy_map0=make2Dmap(COMyp[:],XIndex,YIndex,MapHeight,MapWidth)

    plot_SEM(comx_map0, colorbarlabel='COM x (pixels)', cmap='Greys_r')
    plot_SEM(comy_map0, colorbarlabel='COM y (pixels)', cmap='Greys_r')
```



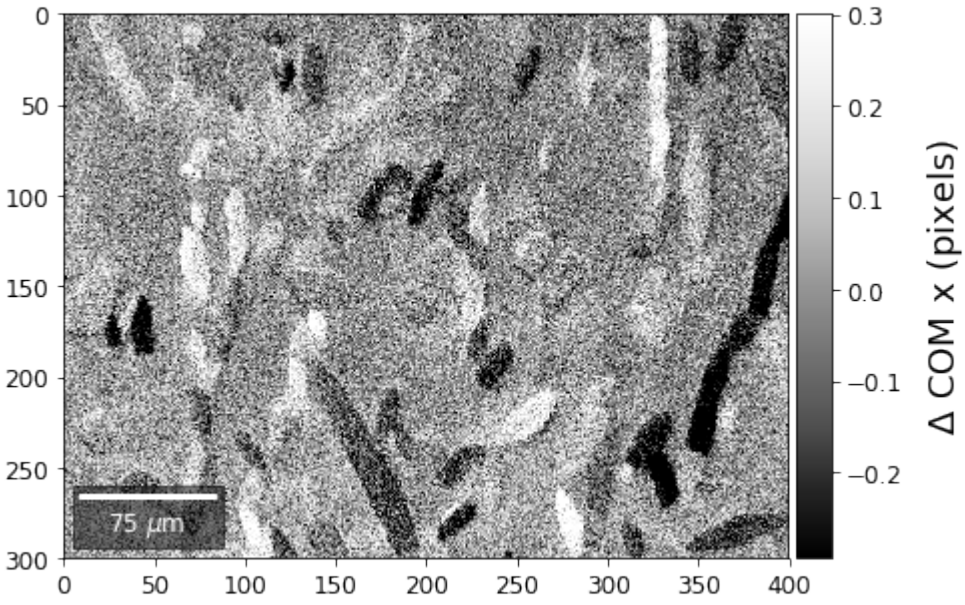


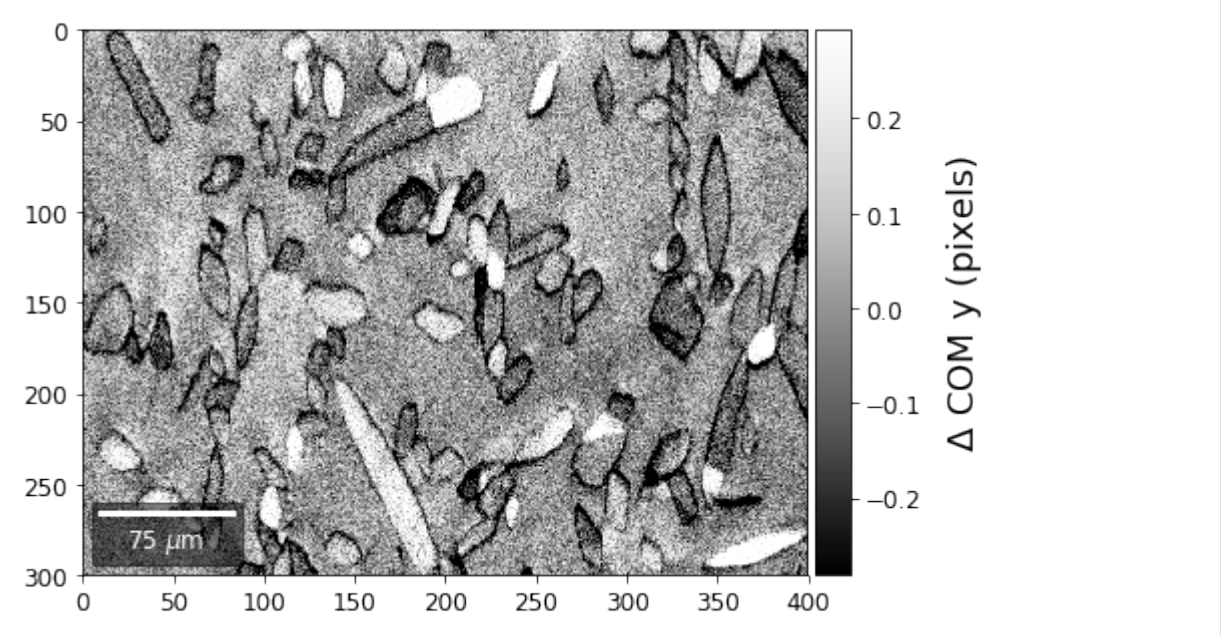
[34]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_vbse']
    COMyp = h5f['/COM/COMyp_vbse']

    meanx=np.mean(COMxp)
    meany=np.mean(COMyp)

    comx_map=make2Dmap(COMxp[:]-meanx,XIndex,YIndex,MapHeight,MapWidth)
    comy_map=make2Dmap(COMyp[:]-meany,XIndex,YIndex,MapHeight,MapWidth)

    plot_SEM(comx_map, colorbarlabel='COM x (pixels)', filename='comx', cmap='Greys_r')
    plot_SEM(comy_map, colorbarlabel='COM y (pixels)', filename='comy', cmap='Greys_r')
```





[35]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_kiku']
    COMyp = h5f['/COM/COMyp_kiku']
    meanx = np.mean(COMxp)
    meany = np.mean(COMyp)

    comx_map = make2Dmap(COMxp[:] - meanx, XIndex, YIndex, MapHeight, MapWidth)
    comy_map = make2Dmap(COMyp[:] - meany, XIndex, YIndex, MapHeight, MapWidth)

    plot_SEM(comx_map, colorbarlabel='$\Delta$ COM x (pixels)', filename='comx_kiku', cmap=
↪'Greys_r')
    plot_SEM(comy_map, colorbarlabel='$\Delta$ COM y (pixels)', filename='comy_kiku', cmap=
↪'Greys_r')
```





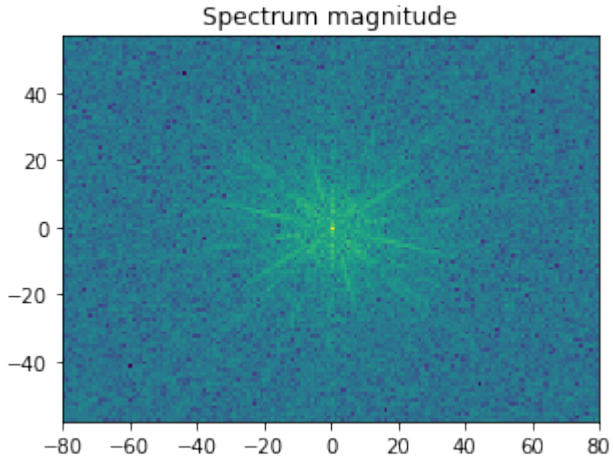
Fourier Transform Based Imaging

With the help of the Fast Fourier Transform (FFT), we can extract information on spatial frequencies (wave vectors) from an image. This can be used to derive imaging signals which are based on ranges of specific wave vectors present in the Kikuchi pattern.

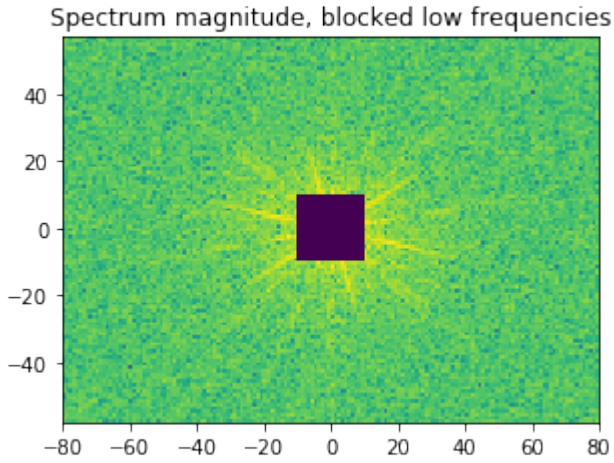
In the example shown below, we determine the intensity in the four quadrants of the FFT spectrum magnitude. The points corresponding to the low spatial frequencies (large spatial extensions) are removed to suppress the influence of the background signal.

[36]:
```
from scipy import fftpack
```

[37]:
```
image = process_kikuchi(Patterns[0])
M, N = image.shape
F = fftpack.fftn(image)

F_magnitude = np.abs(F)
F_magnitude = fftpack.fftshift(F_magnitude)


magnitude = np.log(1 + F_magnitude)

f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(magnitude, cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude');
```



[38]:
```
# Set block around center of spectrum to zero
K = 10
F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

magnitude = np.log(1 + F_magnitude)

f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(magnitude, cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude, blocked low frequencies');
```



[39]:
```
def get_fft_sector_mask(image, isector, rmin=0.1, rmax=0.4,
                        width_rad=np.pi/8, offset_rad=0.0):
    """
    get specified 45 deg sector mask from fft magnitude spectrum of image
    """
    F = fftpack.fftn(image)
    F_magnitude = np.abs(F)
    F_magnitude = fftpack.fftshift(F_magnitude)

    phi_start = (isector % 4) * np.pi/4 + offset_rad
    phi_end   = phi_start + width_rad
    sector_mask = np.ones_like(F_magnitude)
    M, N = F_magnitude.shape
```

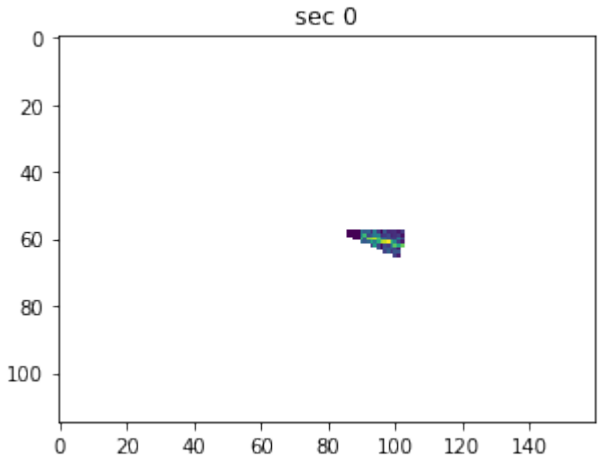
(continues on next page)

(continued from previous page)

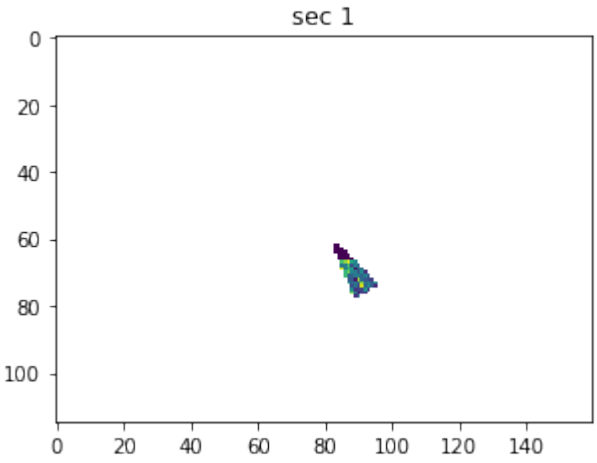
```
    M2 = M // 2
    N2 = N // 2
    for ix in range(N):
        for iy in range(M):
            rx = ix-N2
            ry = iy-M2
            phi = np.arctan2(ry, rx)
            r = np.sqrt(rx**2 + ry**2)
            if (phi>phi_start) and (phi<phi_end) and (r<= M2 * rmax ) and (r>=M2*rmin) :
                sector_mask[iy,ix] = 0

    #profile_quality_map = np.ma.MaskedArray(profile_quality_map, mask=calcmask)
    return sector_mask

def get_fft_sector(spectrum2d, mask):
    return np.ma.MaskedArray(spectrum2d, mask=mask)
```

[40]:
```
mask_0 = get_fft_sector_mask(Patterns[0], 0)
mask_1 = get_fft_sector_mask(Patterns[0], 1)
mask_2 = get_fft_sector_mask(Patterns[0], 2)
mask_3 = get_fft_sector_mask(Patterns[0], 3)

sec_0 = get_fft_sector(F_magnitude, mask_0)
sec_1 = get_fft_sector(F_magnitude, mask_1)
sec_2 = get_fft_sector(F_magnitude, mask_2)
sec_3 = get_fft_sector(F_magnitude, mask_3)
```

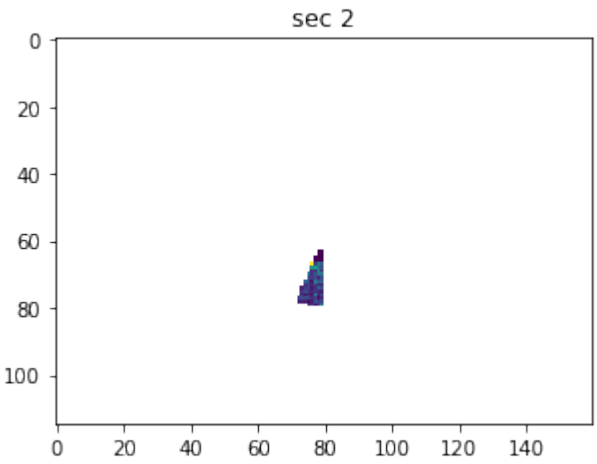
[41]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_0, cmap='viridis',)
ax.set_title('sec 0');
```



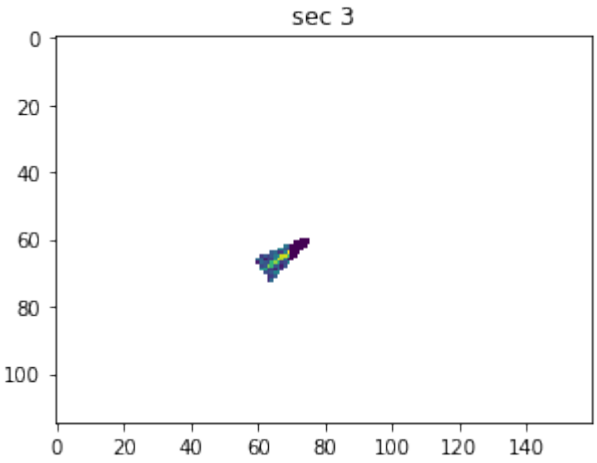
[42]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_1, cmap='viridis',)
ax.set_title('sec 1');
```



[43]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_2, cmap='viridis',)
ax.set_title('sec 2');
```



[44]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_3, cmap='viridis',)
ax.set_title('sec 3');
```



[45]:
```
def secfft(image, masks, K=5):
```

(continues on next page)

(continued from previous page)

```
    """
    calculate FFT magnitude sectors

    remove +/- K points in center (low spatial frequencies)
    limit to +/- W points away from center (low pass, avoid noise at higher frequencies)

    """
    M, N = image.shape
    mask_0, mask_1, mask_2, mask_3 = masks

    F = fftpack.fftn(image)

    F_magnitude = np.abs(F)
    F_magnitude = fftpack.fftshift(F_magnitude)
    #F_magnitude = np.log(1 + F_magnitude)

    # Set block +/-K around center of spectrum to zero
    F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

    sec_0 = get_fft_sector(F_magnitude, mask_0)
    sec_1 = get_fft_sector(F_magnitude, mask_1)
    sec_2 = get_fft_sector(F_magnitude, mask_2)
    sec_3 = get_fft_sector(F_magnitude, mask_3)

    s0 = np.sum(sec_0)
    s1 = np.sum(sec_1)
    s2 = np.sum(sec_2)
    s3 = np.sum(sec_3)

    return np.array([s0, s1, s2, s3])



def calc_secfft(patterns, process=None, K=4):
    """
    calc magnitude sum in FFT quadrants of image
    "process" is an optional image pre-processing function
    """
    npatterns = patterns.shape[0]
    fft_sec_magnitudes = np.zeros((npatterns, 4))

    mask_0 = get_fft_sector_mask(patterns[0], 0)
    mask_1 = get_fft_sector_mask(patterns[0], 1)
    mask_2 = get_fft_sector_mask(patterns[0], 2)
    mask_3 = get_fft_sector_mask(patterns[0], 3)

    masks = (mask_0, mask_1, mask_2, mask_3)

    tstart = time.time()
    for i in range(npatterns):

        # get current pattern
        if process is None:
            fft_sec_magnitudes[i,:] = secfft(patterns[i,:,:], masks, K=K)
        else:
            fft_sec_magnitudes[i,:] = secfft(process(patterns[i,:,:]), masks, K=K)

        print_progress_line(tstart, i, npatterns)
```

(continues on next page)

(continued from previous page)

```
    return fft_sec_magnitudes
```

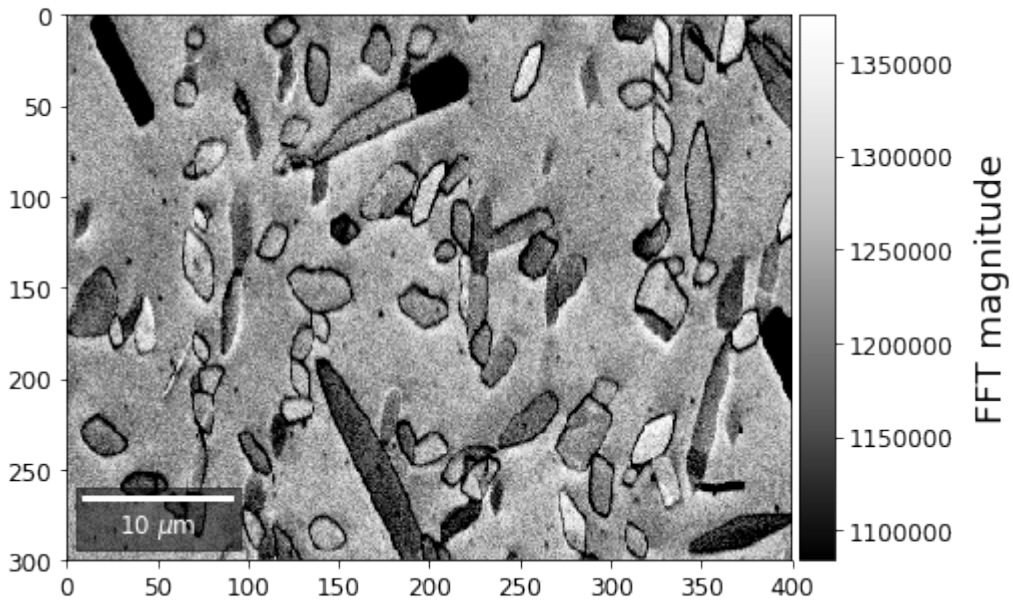
[46]:
```
secfftmag =  calc_secfft(Patterns, process=process_kikuchi)
```

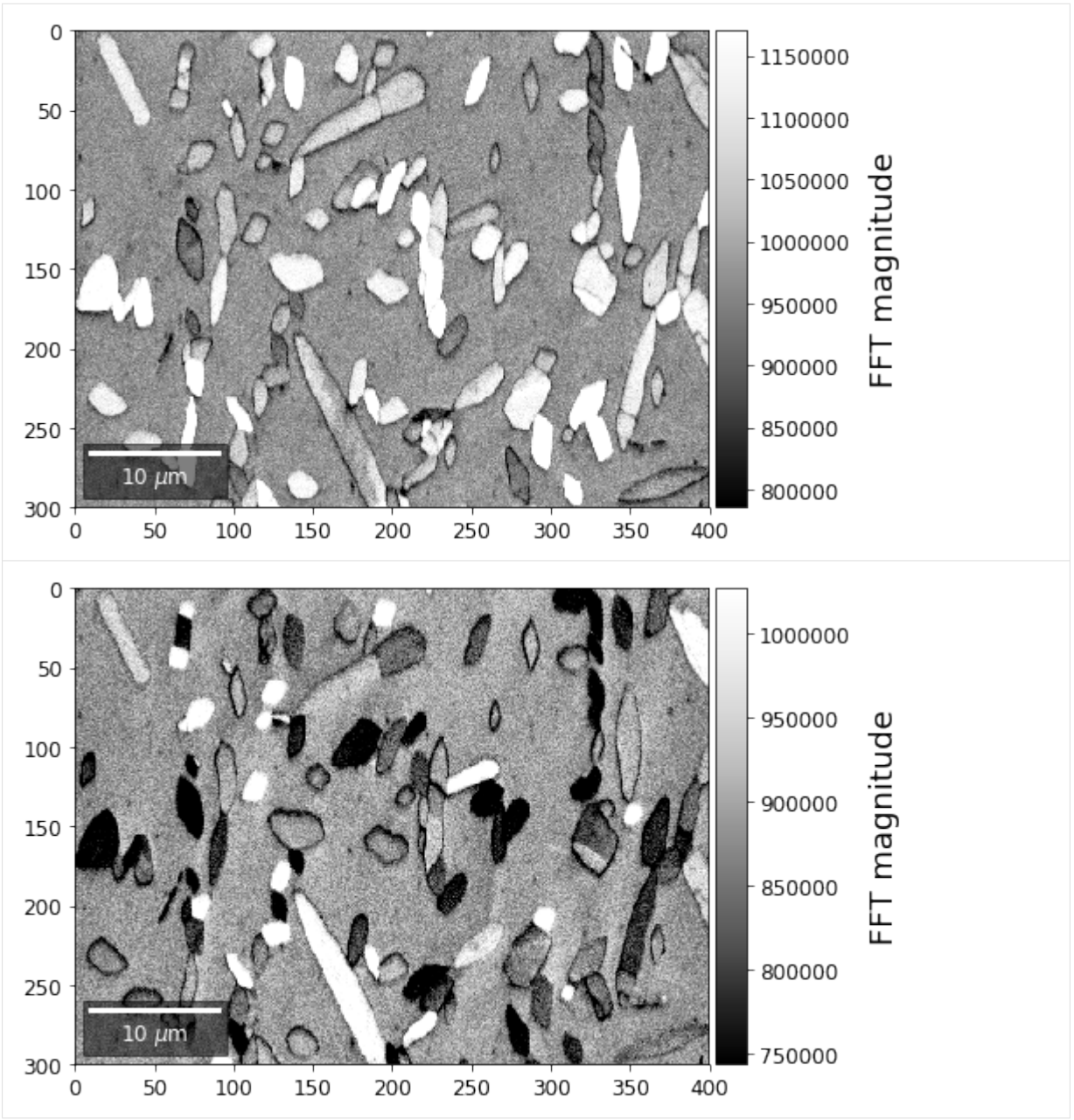
```
total points:120000 current:120000 finished -> total calculation time :    7.2 min
```

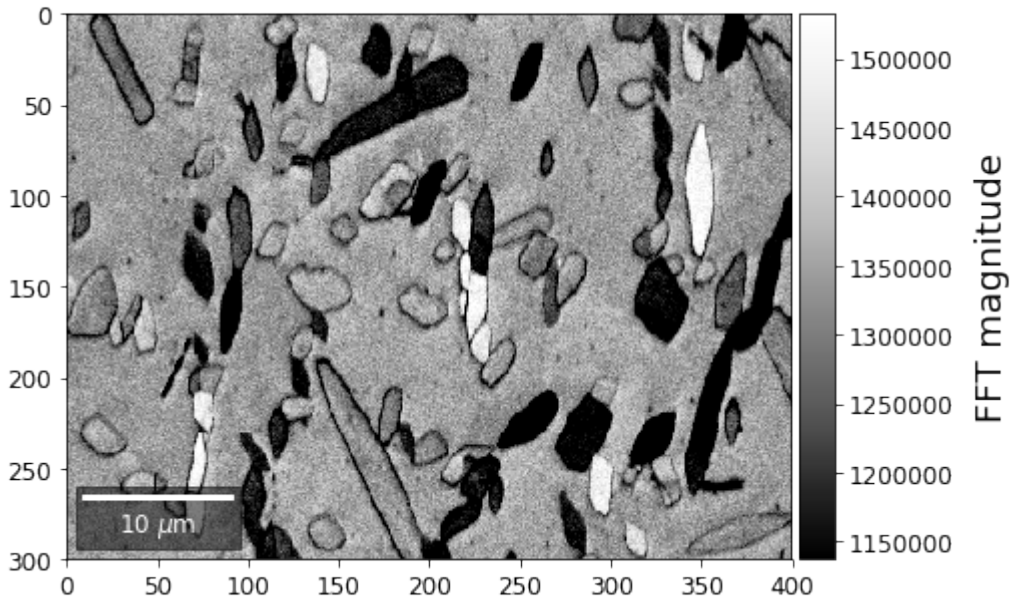
[47]:
```
# append to current h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('/fft_sectors', data=secfftmag)
```

```
arBSE_Steel_FCC_in_BCC_2752.h5
```

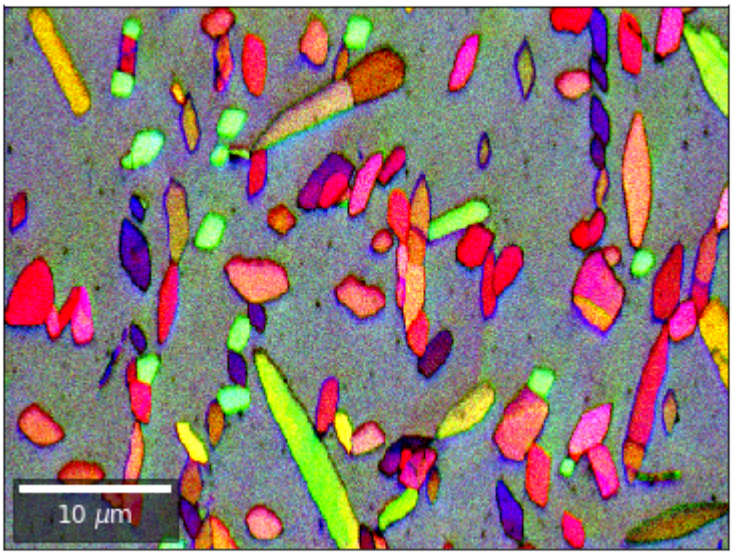
[48]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    secfftmag = np.copy(h5f['fft_sectors'])
    for q, sector in enumerate(secfftmag.T):
        signal_map = make2Dmap(sector,XIndex,YIndex,MapHeight,MapWidth)
        plot_SEM(signal_map, vrange=None, cmap='Greys_r',
                 colorbarlabel='FFT magnitude', microns=step_map_microns,
                 filename='secFFT_'+str(q))
```




0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
1150000
1100000
1050000
1000000
950000
900000
850000
800000
FFT magnitude
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
10 μm
1000000
950000
900000
850000
800000
750000
FFT magnitude



[49]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    secfftmag = h5f['fft_sectors']

    signal = secfftmag[:,1]
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,2]
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,0]
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='fft_sectors_RGB', microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.0)
```

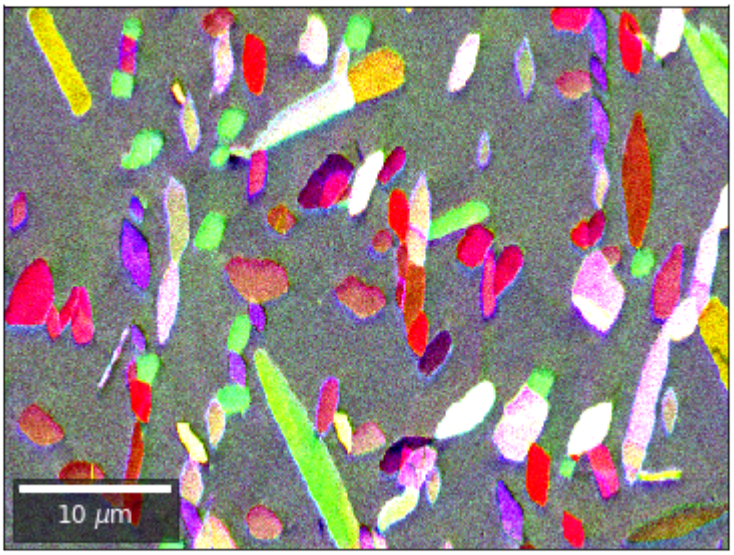


[50]:
```
# normalize to reference sector
```

(continues on next page)

(continued from previous page)

```
with h5py.File(h5ResultFile, 'r') as h5f:
    secfftmag = h5f['fft_sectors']

    ref_signal = secfftmag[:,3]

    signal = secfftmag[:,1] / ref_signal
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,2] / ref_signal
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,0] / ref_signal
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='fft_sectors_rel_RGB', microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.0)
```



Radon Transform Imaging

(. . . without pattern indexing)

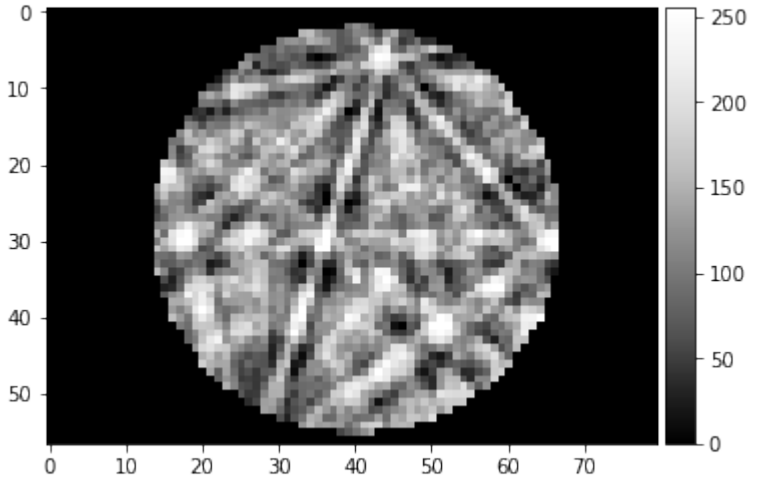
[51]:
```
from skimage.transform import radon
from aloe.image.nxcc import mask_pattern_disk
```

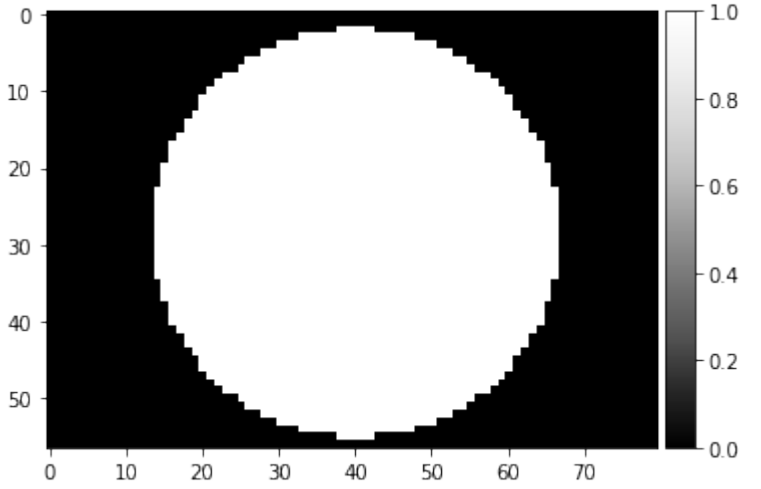
[52]:
```
prebinning_radon = 2 # downsample patterns before Radon (speed)

background_static_radon = downsample(background_static, prebinning_radon) # need␣
↪downsampled background for processing

# angles for Radon (constant for each pattern)
theta = np.linspace(0., 180., max(image.shape) // 2, endpoint=False)

def calc_radon(image, theta, sinogram_reference=None):
    """ calculate radon vbse array """
```

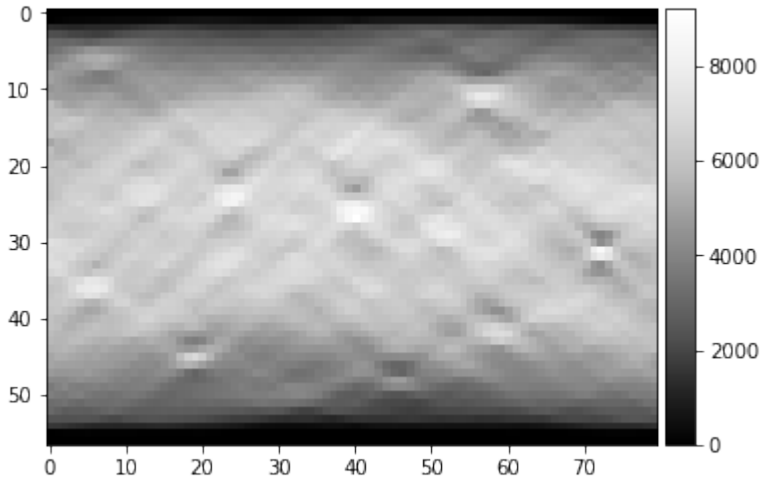
(continues on next page)

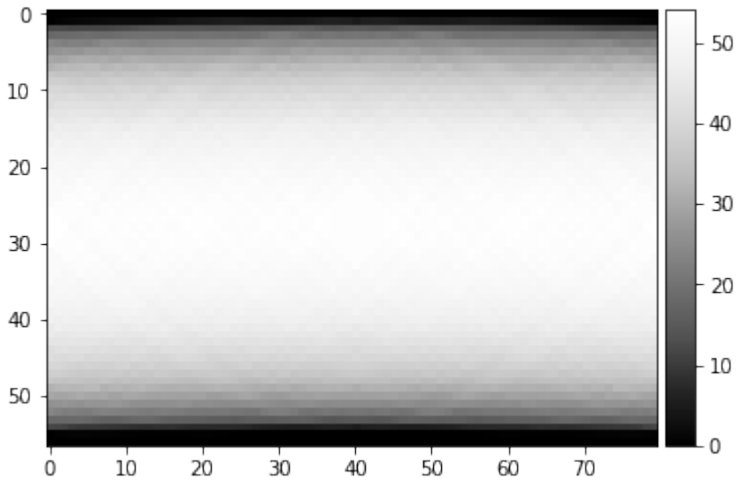
(continued from previous page)

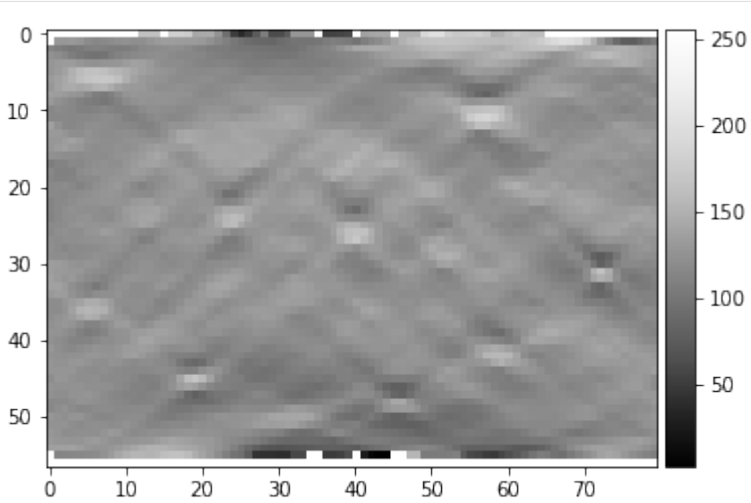
```
    sinogram = radon(image, theta=theta, circle=True)

    if sinogram_reference is not None:
        sino = sinogram / sinogram_reference
    else:
        sino = sinogram

    mean_radon = np.nanmean(sino)
    sino = (sino - mean_radon)**2

    # clip 2 lines
    return sino[2:-2,:]


def process_kikuchi_radon(pattern):
    return process_ebsp(downsample(pattern, prebinning_radon), static_
↪background=background_static_radon)


def process_radon(pattern):
    processed_pattern = process_ebsp(downsample(pattern, prebinning_radon), static_
↪background=background_static_radon)
    kiku, npix = mask_pattern_disk(processed_pattern, rmax=0.333)
    sino = calc_radon(kiku, theta, sinogram_reference=sinogram_uniform)
    return sino


image, npix = mask_pattern_disk(process_kikuchi_radon(Patterns[0]), rmax=0.333)
image_uniform, npix = mask_pattern_disk(np.ones_like(image), rmax=0.333)

img_height, img_width = image.shape

plot_image(image)
plot_image(image_uniform)

sinogram = radon(image, theta=theta, circle=True)
sinogram_uniform = radon(image_uniform, theta=theta, circle=True)
plot_image(sinogram)
plot_image(sinogram_uniform)

sino = sinogram / sinogram_uniform
sino77=arbse.rebin_array(sino)
plot_image(sino)
plot_image(sino77)
```

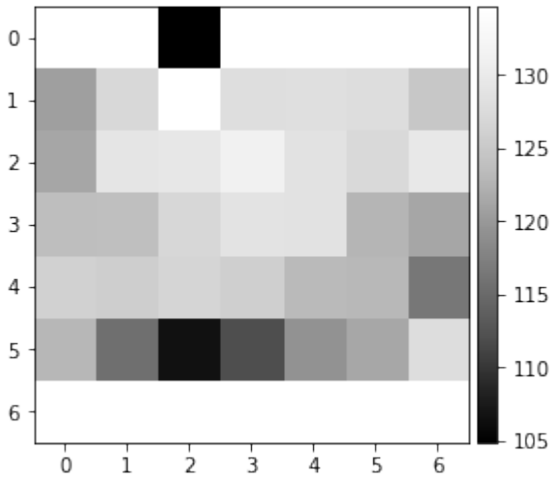

0
10
20
30
40
50
0
10
20
30
40
50
60
70
250
200
150
100
50
0


0
10
20
30
40
50
0
10
20
30
40
50
60
70
1.0
0.8
0.6
0.4
0.2
0.0

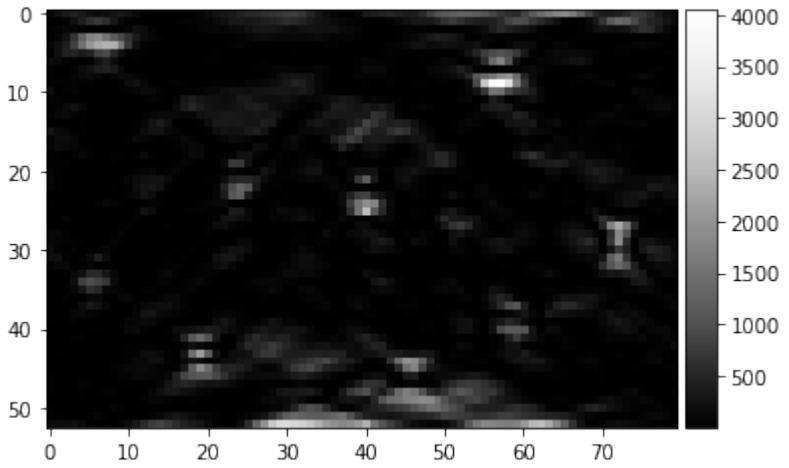

0
10
20
30
40
50
0
10
20
30
40
50
60
70
8000
6000
4000
2000
0







[53]:
```
radon_test = process_radon(Patterns[0])
plot_image(radon_test)
```



Calculate Radon arrays for all patterns:

[54]:
```
vbse_radon =  arbse.make_vbse_array(Patterns, process=process_radon)
```

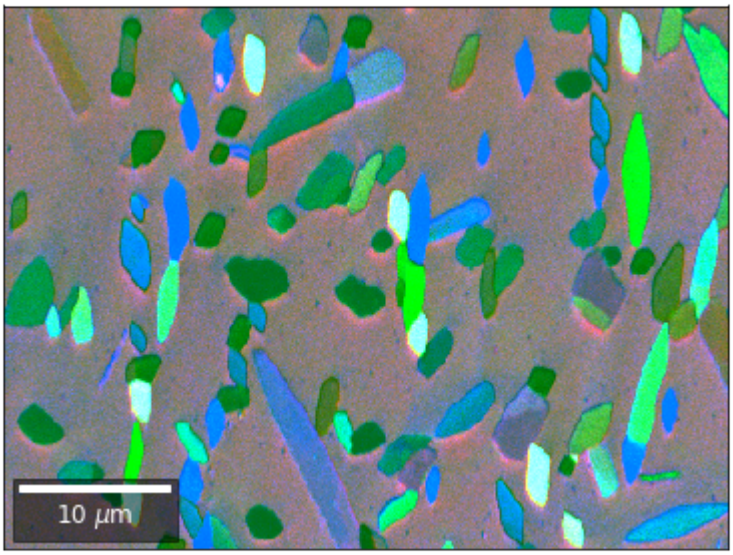
```
total points:120000 current:120000 finished -> total calculation time :   35.3 min
```

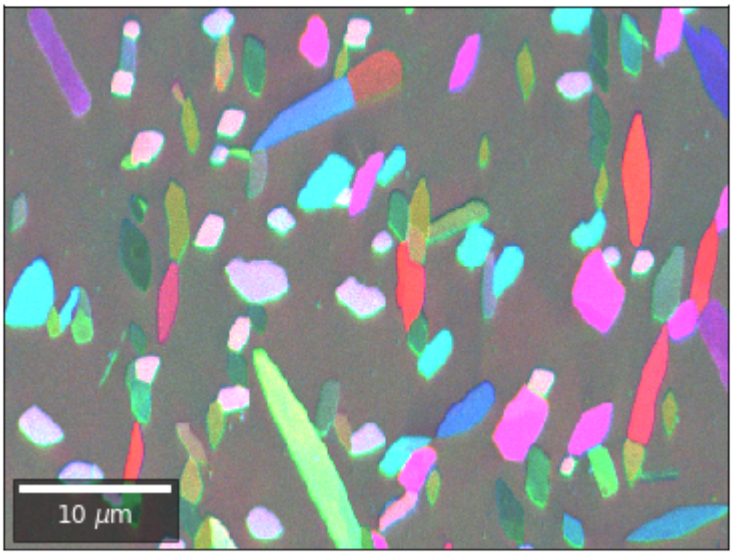
[55]:
```
# save to current hdf5 results file
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vbse_radon', data=vbse_radon)
```

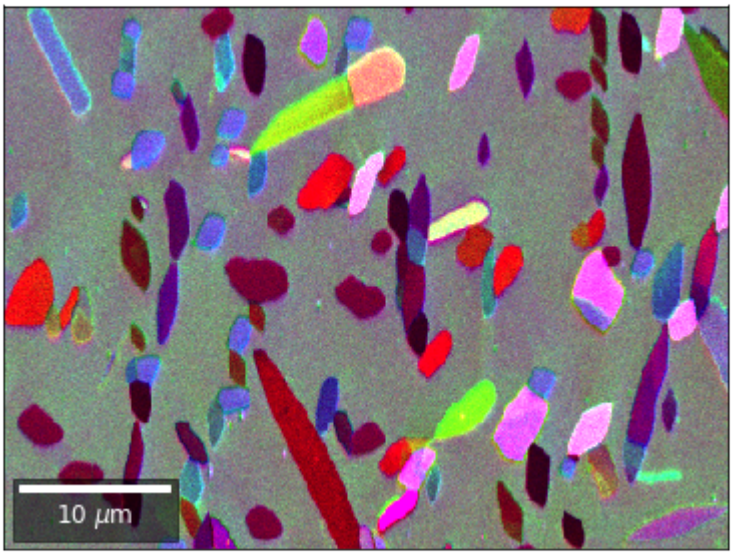
```
arBSE_Steel_FCC_in_BCC_2752.h5
```

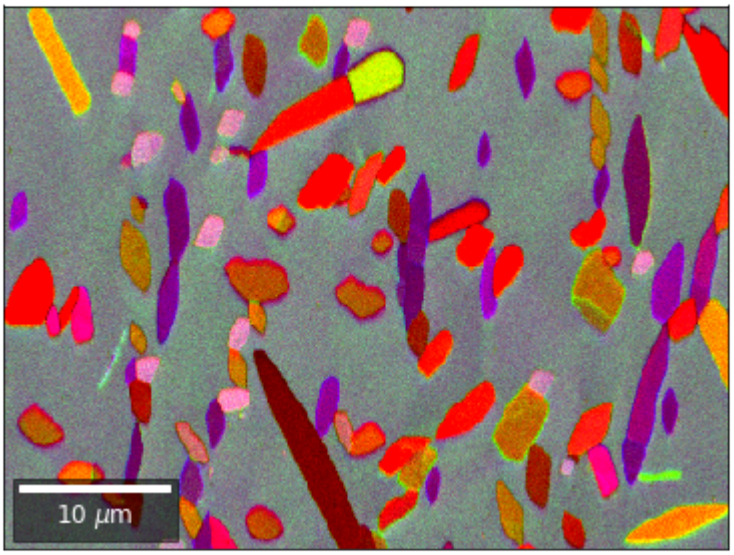
Radon 7x7 array, for each row, RGB color from element in column 1, 4, 7

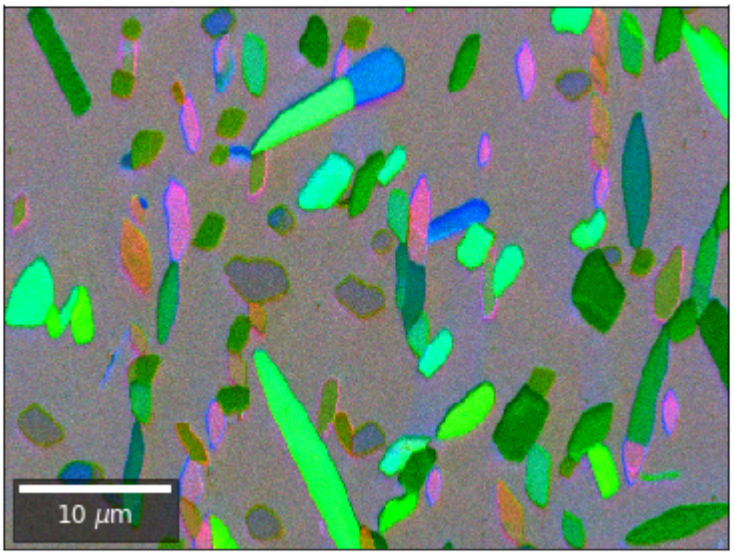
[56]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD = h5f['vbse_radon']
    # rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vRadon_RGB_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=0.8)

        rgb_direct.append(rgb)
```

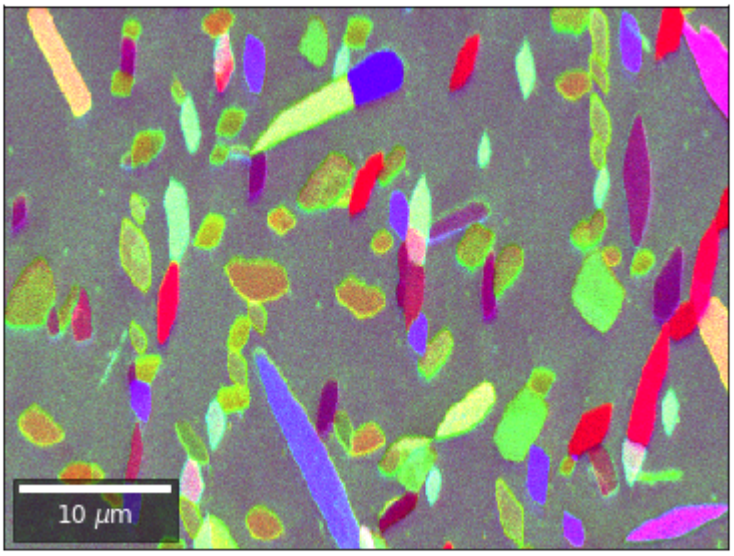

10 μm

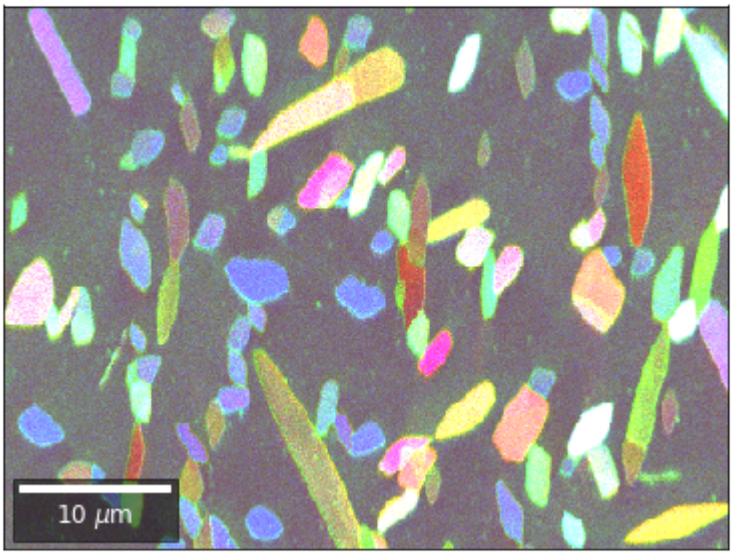

10 μm


10 μm


10 μm

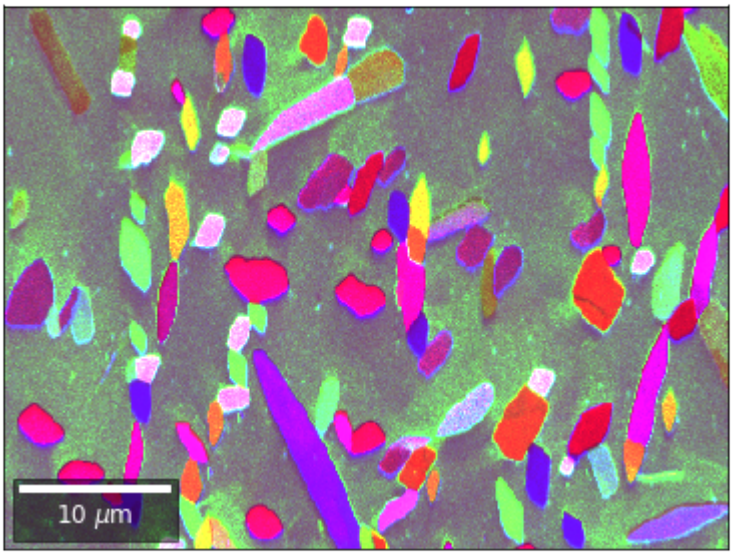

10 μm

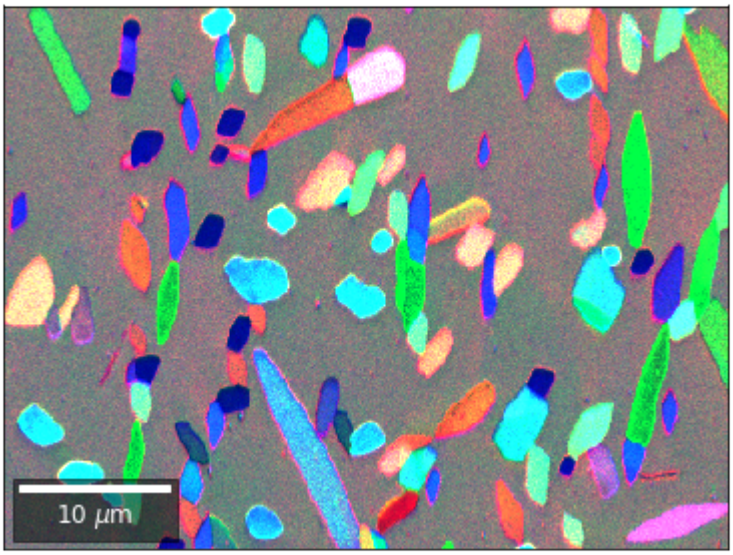

10 μm

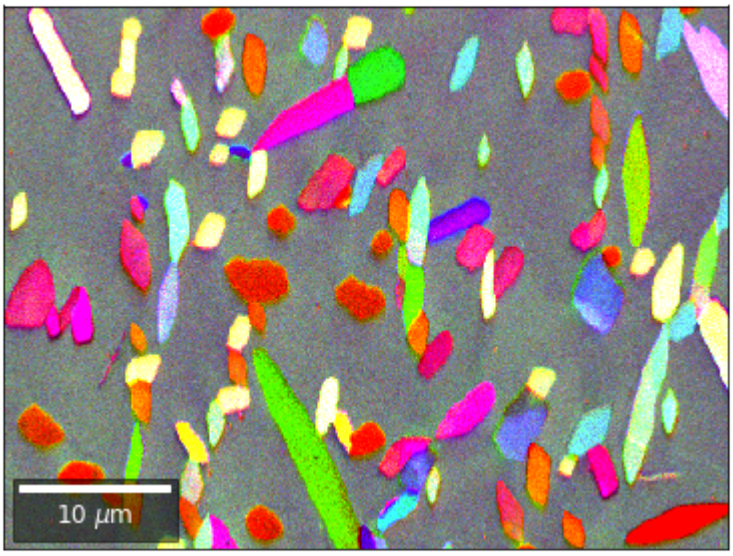


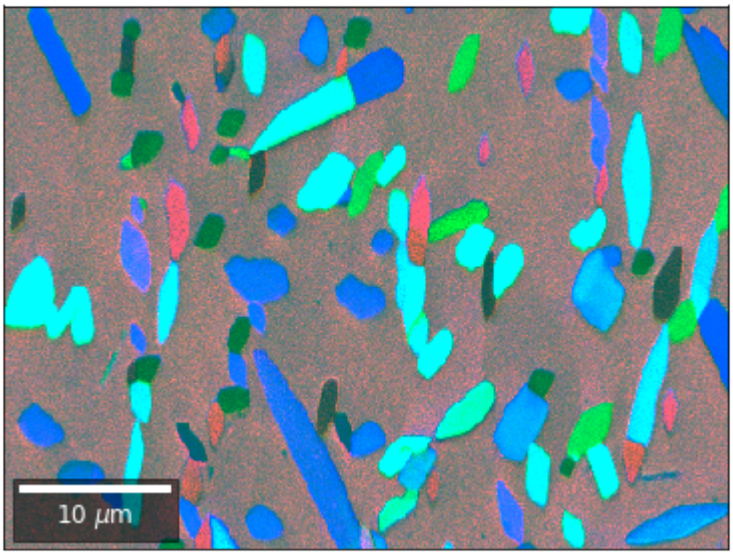
Radon 7x7 array: For each row, RGB color signal from CHANGE of elements in column 1, 4, 7 relative to previous row¶

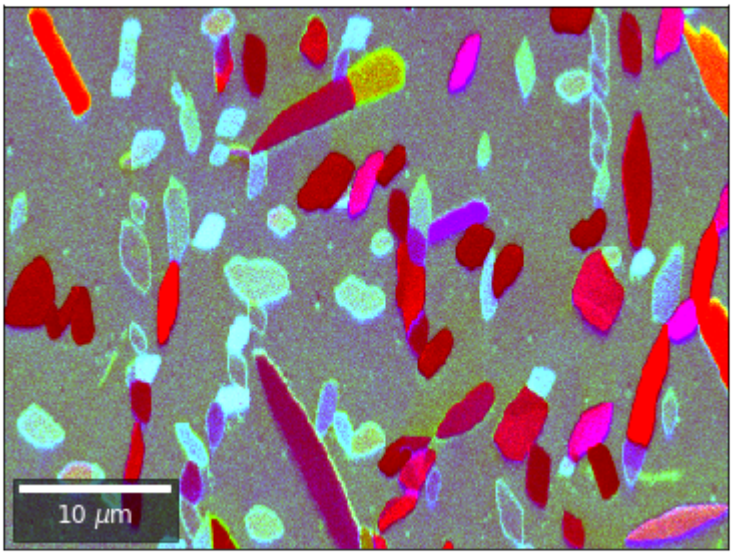
[57]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD = h5f['vbse_radon']
    # relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vRadon_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

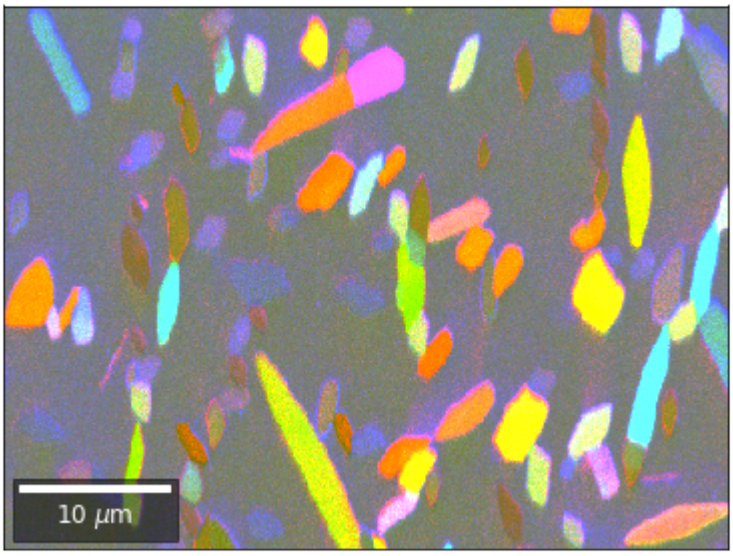

10 μm


10 μm


10 μm


10 μm


10 μm


10 μm

SEM 2D BSE Imaging: GaN Stripes

Example Data: GaN_Stripes

This is an example notebook to demonstrate angle resolved BSE imaging via signal extraction from raw, saved, EBSD patterns of a GaN thin film sample.

A key advantage when using saved raw EBSD patterns is that we can partition the raw signal into a "Kikuchi pattern" and a "background" signal, which will give different types of contrasts. This partition is impossible when using conventional semiconductor diode detection etc. for angle-resolved signal BSE collection. We cannot distinguish between "Kikuchi signal" and "background signal" in the electron current from the diode because the diode signal is a point-signal, as compared to a full 2D signal which is available on the phosphor screen.

[1]:
```
# directory with the HDF5 EBSD pattern file:
data_dir = "../../../xcdskd_reference_data/GaN_Stripes/"

# filename of the HDF5 file:
hdf5_filename =  "GaN_Stripes.h5"

# verbose output
output_verbose = False
```

Necessary packages

[2]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
# ignore divide by Zero
np.seterr(divide='ignore', invalid='ignore')

import time, sys, os
import h5py
import skimage.io

from aloe.plots import normalizeChannel, make2Dmap, get_vrange
from aloe.plots import plot_image, plot_SEM, plot_SEM_RGB
from aloe.image import arbse
from aloe.image.downsample import downsample
from aloe.image.kikufilter import process_ebsp
from aloe.io.progress import print_progress_line
```

[3]:
```
# load background from different experiment with same detector
background_static_txt_full = np.loadtxt("../data/patterns/static_bam_ef.txt")


# make result dirs and filenames
h5FileNameFull=os.path.abspath(data_dir + hdf5_filename)
h5FileName, h5FileExt = os.path.splitext(h5FileNameFull)
h5FilePath, h5File = os.path.split(h5FileNameFull)
timestr = time.strftime("%Y%m%d-%H%M%S")
h5ResultFile="arBSE_" + hdf5_filename


# close HDF5 file if still open
if 'f' in locals():
    f.close()
```

(continues on next page)

(continued from previous page)

```
f=h5py.File(h5FileName+h5FileExt, "r")

ResultsDir = h5FilePath+"/arBSE_" + timestr + "/"
CurrentDir = os.getcwd()
#print('Current Directory: '+CurrentDir)
#print('Results Directory: '+ResultsDir)
if not os.path.isdir(ResultsDir):
    os.makedirs(ResultsDir)
os.chdir(ResultsDir)

if output_verbose:
    print('HDF5 full file name: ', h5FileNameFull)
    print('HDF5 File: ', h5FileName+h5FileExt)
    print('HDF5 Path: ', h5FilePath)
    print('Results Directory: ', ResultsDir)
    print('Results File: ', h5ResultFile)
```

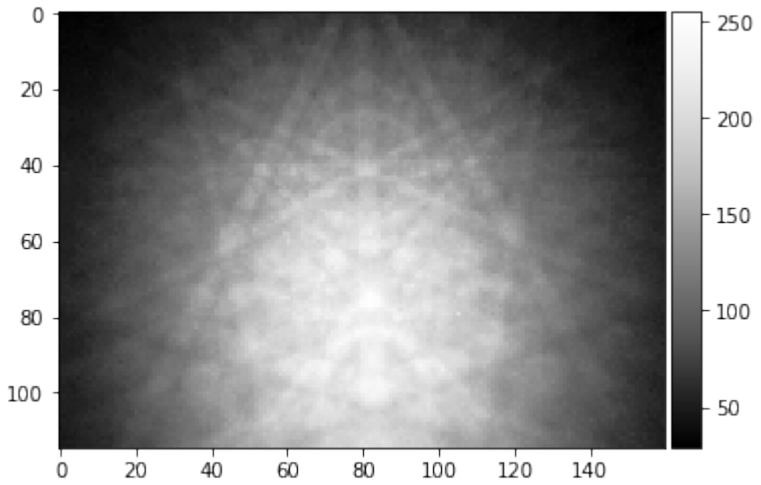
[4]:
```
DataGroup="/Scan/EBSD/Data/"
HeaderGroup="/Scan/EBSD/Header/"
Patterns = f[DataGroup+"RawPatterns"]
#StaticBackground=f[DataGroup+"StaticBackground"]
XIndex =   f[DataGroup+"X BEAM"]
YIndex =   f[DataGroup+"Y BEAM"]
MapWidth = f[HeaderGroup+"NCOLS"].value
MapHeight= f[HeaderGroup+"NROWS"].value
PatternHeight=f[HeaderGroup+"PatternHeight"].value
PatternWidth =f[HeaderGroup+"PatternWidth"].value
print('Pattern Height: ', PatternHeight)
print('Pattern Width : ', PatternWidth)
PatternAspect=float(PatternWidth)/float(PatternHeight)
print('Pattern Aspect: '+str(PatternAspect))
print('Map Height: ', MapHeight)
print('Map Width : ', MapWidth)

step_map_microns = f[HeaderGroup+"XSTEP"].value
print('Map Step Size (microns): ', step_map_microns)
```

```
Pattern Height:  115
Pattern Width :  160
Pattern Aspect: 1.391304347826087
Map Height:  300
Map Width :  400
Map Step Size (microns):  0.0963639064104
```

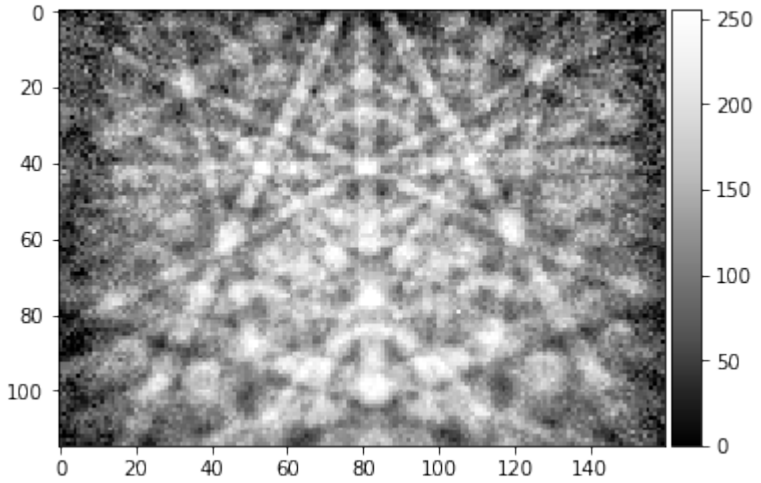
Check Optional Pattern Processing

[5]:
```
ipattern = 215
raw_pattern=Patterns[ipattern,:,:]
print(raw_pattern.shape)
plot_image(raw_pattern)
skimage.io.imsave('pattern_raw_' + str(ipattern) + '.tiff', raw_pattern, plugin='tifffile')
```

```
(115, 160)
```



[6]:
```
# make processed pattern without static background
processed_pattern = process_ebsp(raw_pattern, binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_' + str(ipattern) + '.tiff', processed_pattern,␣
↪plugin='tifffile')
```



How NOT to get the static background for the current measurement

[7]:
```
try:
    # get static background from HDF5, cut off first lines to fit to Patterns
    background_static_file = f[HeaderGroup+"StaticBackground"][0,5:,:]
    plot_image(background_static_file, title="Static Background from HDF5 File")
except:
    background_static_file = None
```

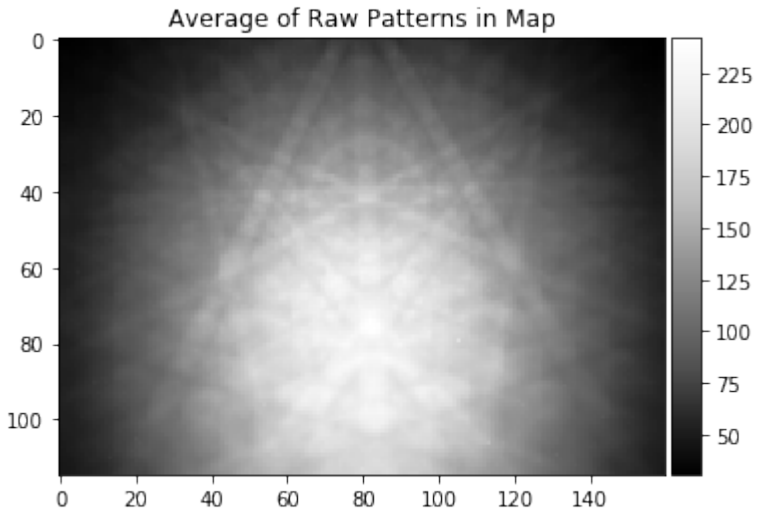
Static Background from Pattern Average in the Map

We can also approximate a static background from the EBSD map itself; or even use an extra map that was taken explicitly for making a background, e.g. from the sample holder. For polycrystalline samples with a large enough number of grains in random orientations, the Kikuchi patterns from all grains will average out when taking the average of all pattern. For single crystalline samples, or samples with a low number of different orientation present in the map area measured, the average of all patterns will stay contain Kikuchi diffraction features. These features in the static background will tend to produce artififacts when the raw data is processed using the background with diffraction features.

[8]:
```
calc_bg = True
if calc_bg:
    # avoid loading 35GB into RAM if you don't have 35GB RAM...
    # use incremental "updating average"
    # https://math.stackexchange.com/questions/106700/incremental-averageing
    tstart = time.time()
    npatterns = Patterns.shape[0]
    current_average = np.copy(Patterns[0]).astype(np.float64)
    for i in range(1, npatterns):
        current_average = current_average + (Patterns[i] - current_average)/i
        print_progress_line(tstart, i-1, npatterns-1, every=100)

    background_static_from_map = current_average
```

```
total points:119999 current:119999 finished -> total calculation time :    0.4 min
```

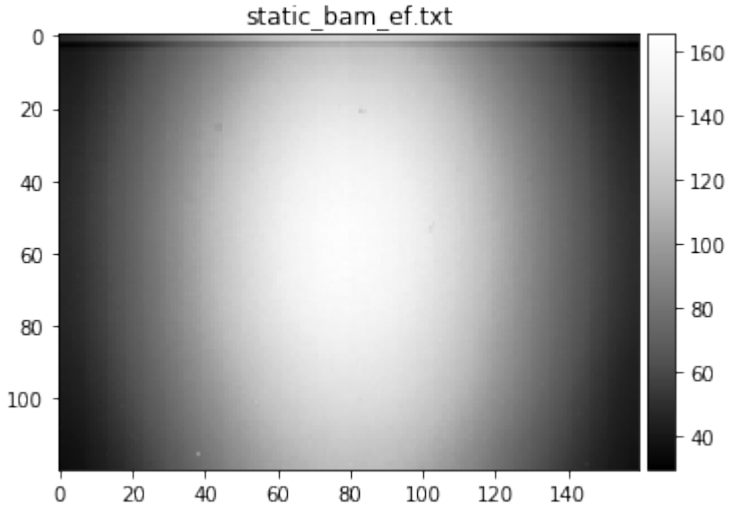
[9]:
```
plot_image(background_static_from_map, title="Average of Raw Patterns in Map")
```



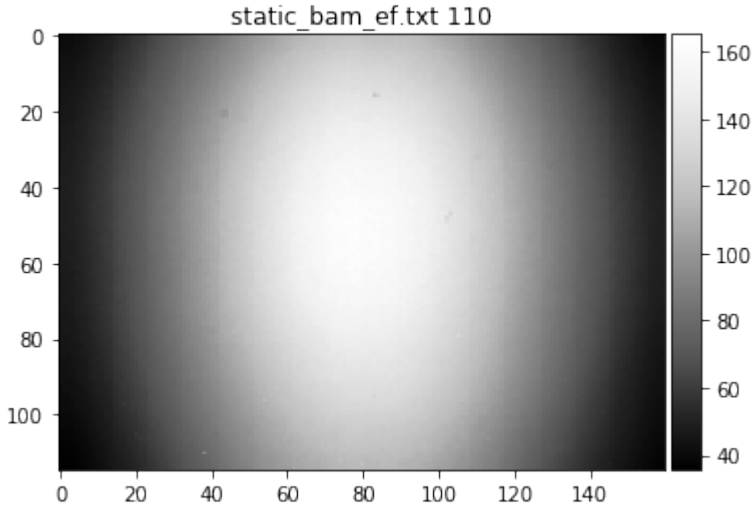
The pattern average does not give a smooth enough background because we are dealing with a sample are that shows only a very limited range of orientation and pattern changes.

We try with a background from a different experiment, which is better, but still shows some bands. Ideally, no Kikuchi bands should be seen in the static background.

[10]:
```
# load background from different experiment with same detector
#background_static_txt = np.loadtxt("../data/patterns/static_bam_ef.txt")
plot_image(background_static_txt_full, title="static_bam_ef.txt")
print(background_static_txt_full.shape)
# we need to cut off the upper  lines because of artifact due to camera hardware limits
background_static_txt = background_static_txt_full[5:,:]
print(background_static_txt.shape)
plot_image(background_static_txt, title="static_bam_ef.txt 110")
```

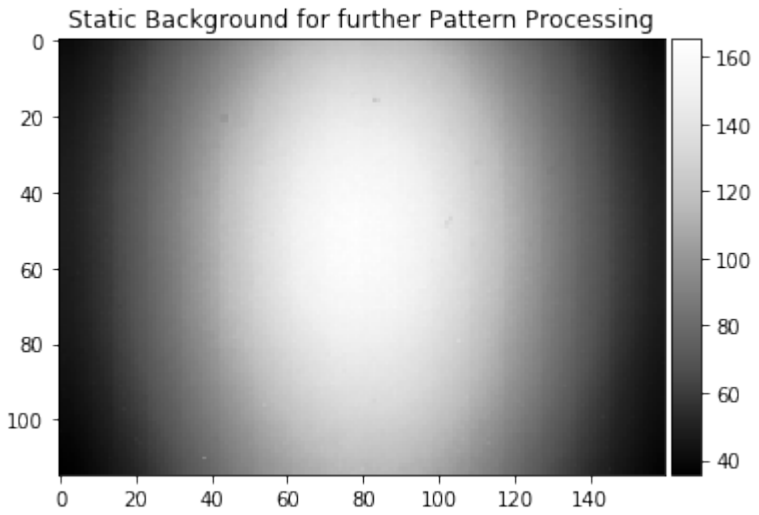



```
(120, 160)
(115, 160)
```



[11]:
```
# assign static background

#background_static = background_static_file
background_static = background_static_txt
print(background_static.shape)
print(background_static)
plot_image(background_static, title="Static Background for further Pattern Processing")
```

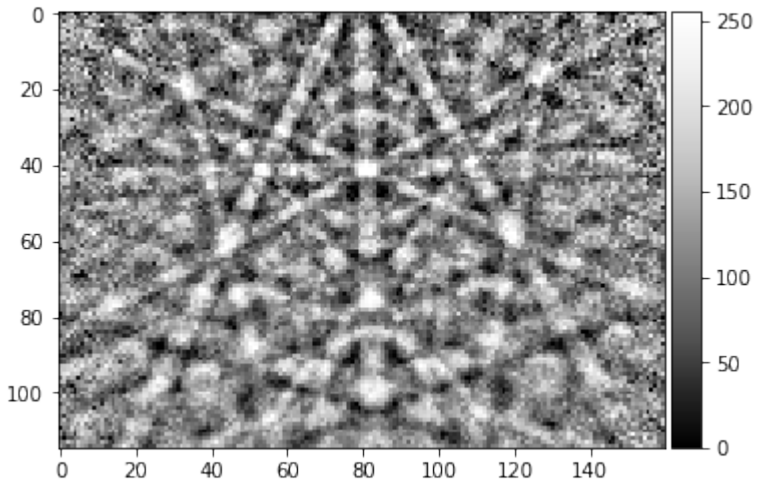
```
(115, 160)
[[ 41.187  41.367  41.491 ...,  39.766  39.829  40.037]
 [ 41.283  41.374  41.392 ...,  39.608  39.749  39.836]
 [ 41.607  42.042  41.783 ...,  40.158  40.186  40.333]
 ...,
 [ 36.24   37.008  37.093 ...,  39.125  38.512  38.229]
 [ 36.347  36.62   36.757 ...,  38.399  38.03   37.787]
 [ 35.654  36.221  36.438 ...,  38.408  37.231  37.011]]
```



[12]:
```
#skimage.io.imsave('background_static.tiff', background_static, plugin='tifffile') # this␣
↪will be 16bit only
np.savetxt('background_static.txt', background_static)
```

We test the dynamic and static background correction for an example pattern. Note that we use "sigma=5" in order to adjust the dynamic background correction taking a smaller local region for smoothing. Check what happens when changing the "sigma" parameter.

[13]:
```
processed_pattern = process_ebsp(raw_pattern, static_background=background_static,␣
↪binning=1, sigma=5)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_static_' + str(ipattern) + '.tiff', processed_pattern,
↪ plugin='tifffile')
```



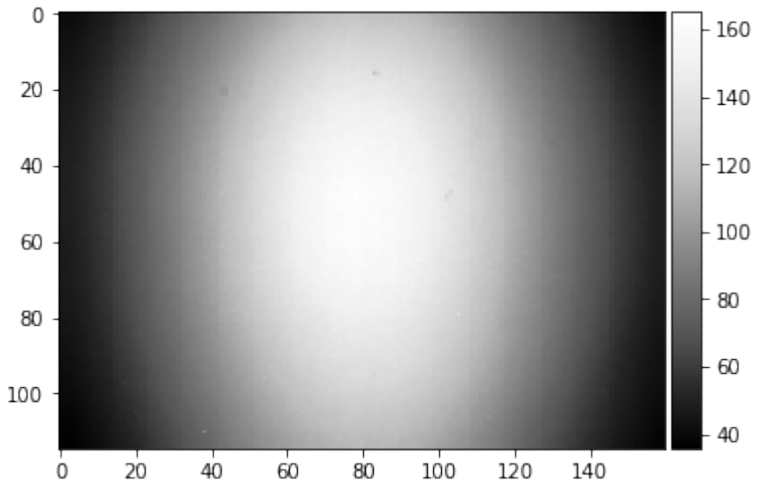
Specification of Image Pre-Processing Functions

[14]:
```
prebinning=1
background_static_binned = downsample(background_static, prebinning)

def pipeline_process(pattern, prebinning=1, kikuchi=False):
    if prebinning>1:
        pattern = downsample(pattern, prebinning)
    if kikuchi:
```

(continues on next page)

(continued from previous page)

```
        return process_ebsp(pattern, static_background=background_static_binned, binning=1,
↪ sigma=5)
    else:
        return pattern

def process_kikuchi(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=True)

def process_bin(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=False)

plot_image(background_static_binned)
print(background_static_binned.shape)
```

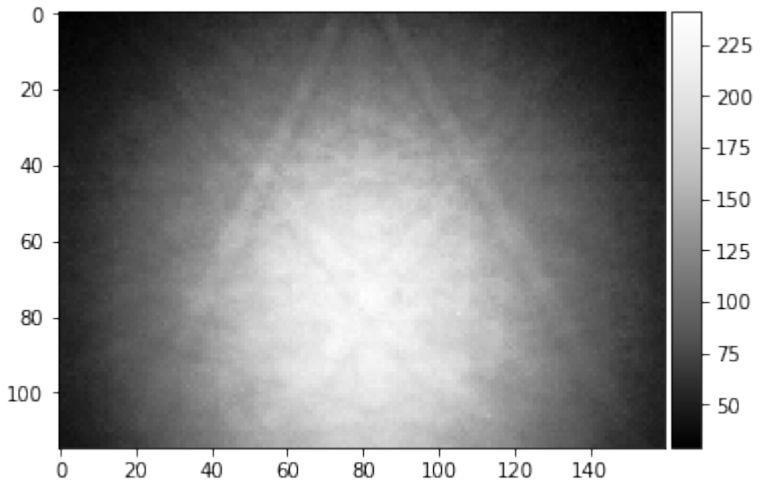


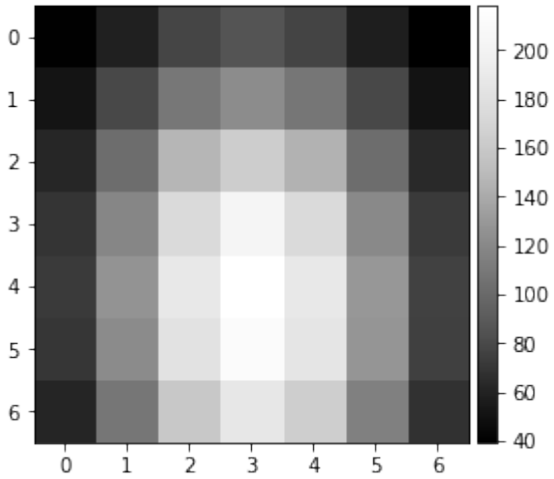

```
(115, 160)
```

vBSE Array

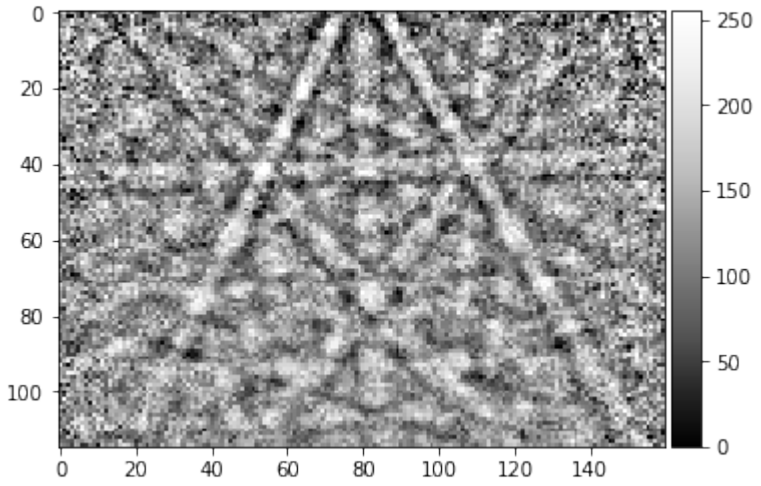
We convert the raw pattern into a 7x7 array of vBSE sensor intensities.

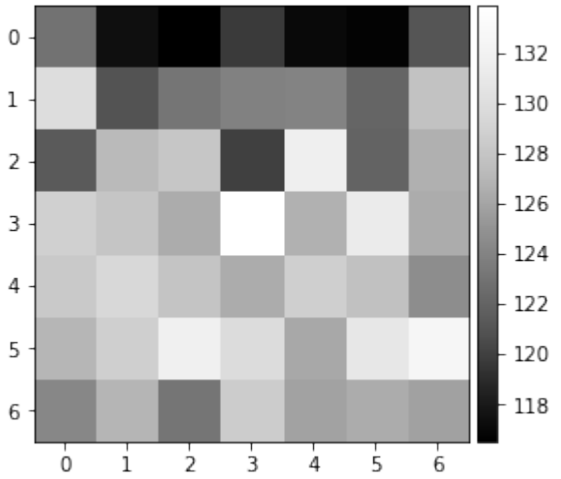
[15]:
```
pattern = pipeline_process(Patterns[1000], kikuchi=False)
vbse = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vbse)
```





[16]:
```
pattern = process_kikuchi(Patterns[1000])
vkiku = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vkiku)
```





vBSE Detector Signals: Calculation & Saving

This should take a few minutes, depending on your computer and file access speed.

Virtual BSE Imaging

Imaging the raw intensity in the respective area of the 2D detector (e.g. phosphor screen). Neglects gnomonic projection effect on intensities.

[17]:
```
# calculate the vBSE signals in 7x7 array
vbse_array = arbse.make_vbse_array(Patterns)

# make vBSE map of the total screen intensity
bse_total = np.sum(np.sum(vbse_array[:,:,:], axis=1), axis=1)
bse_map = make2Dmap(bse_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points:120000 current:120000 finished -> total calculation time :    0.8 min
```

[18]:
```
# save the results in the h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vbse', data=vbse_array)
    h5f.create_dataset('/maps/bse_total', data=bse_map)
```

```
arBSE_GaN_Stripes.h5
```

Virtual Orientation Imaging via Kikuchi Pattern Signals

If we process the raw images to obtain only the Kikuchi pattern, we have a modified 2D intensity which can be expected to show increased sensitivity to orientation effects (i.e. changes related to the Kikuchi bands). In a more advanced approach, we could select, for example, specific Kikuchi bands or zone axes to extract imaging signals.

[19]:
```
# calculate the vKikuchi signals from processed raw data
vkiku_array = arbse.make_vbse_array(Patterns, process=process_kikuchi)

# make vBSE map of the total screen intensity
```

(continues on next page)

(continued from previous page)

```
kiku_total = np.sum(np.sum(vkiku_array[:,:,:], axis=1), axis=1)
kiku_map = make2Dmap(kiku_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points:120000 current:120000 finished -> total calculation time :    6.8 min
```

[20]:
```
# save the results in an extra hdf5
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vkiku', data=vkiku_array)
    h5f.create_dataset('/maps/kiku_total', data=kiku_map)
```

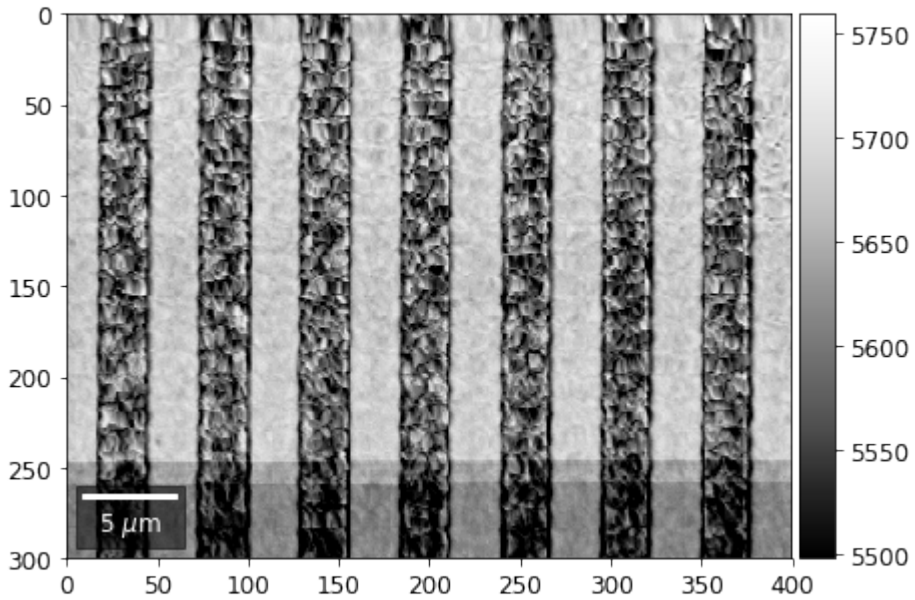
```
arBSE_GaN_Stripes.h5
```

vBSE Signals: Plotting

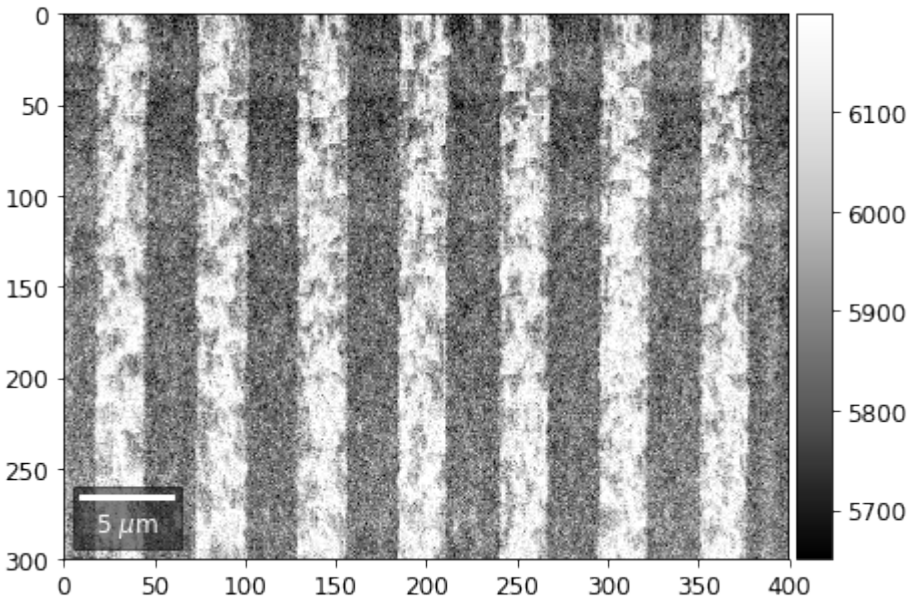
Total Signal on Screen

Total sum of the 7x7 arrays, for the raw pattern and the Kikuchi pattern at each map point:

[21]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    bse = h5f['/maps/bse_total']
    plot_SEM(bse, cmap='Greys_r', microns=step_map_microns)
```



[22]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    kiku = h5f['/maps/kiku_total']
    plot_SEM(kiku, cmap='Greys_r', microns=step_map_microns)
```



Intensity in Rows and Columns of the vBSE array

We can calculate additional images from the vBSE data set of 7x7 ROIs derived from the original patterns. As a first example, we plot the intensities of each of the 7 rows and then of each of the 7 columns:

Rows

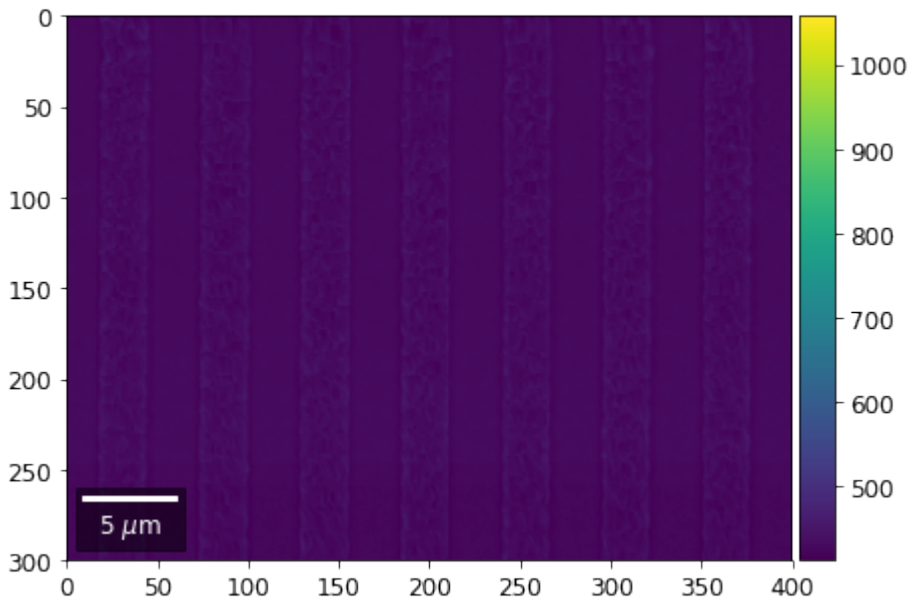
[23]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']

    # signal: sum of row
    vmin=40000000
    vmax=0

    bse_rows = []

    # (1) get full range for all images
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        minv, maxv = get_vrange(signal, stretch=3.0)
        if (minv<vmin):
            vmin=minv
        if (maxv>vmax):
            vmax=maxv

    # (2) make plots with same range for comparisons of absolute BSE values
    vrange=[vmin, vmax]
    print('Range of Values: ', vrange)
    #vrange=None
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
```

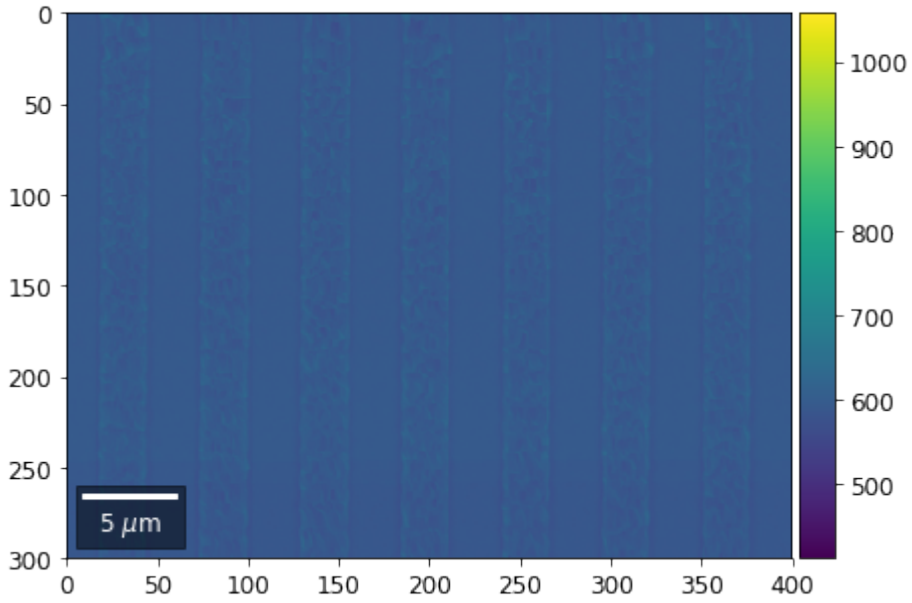
(continues on next page)

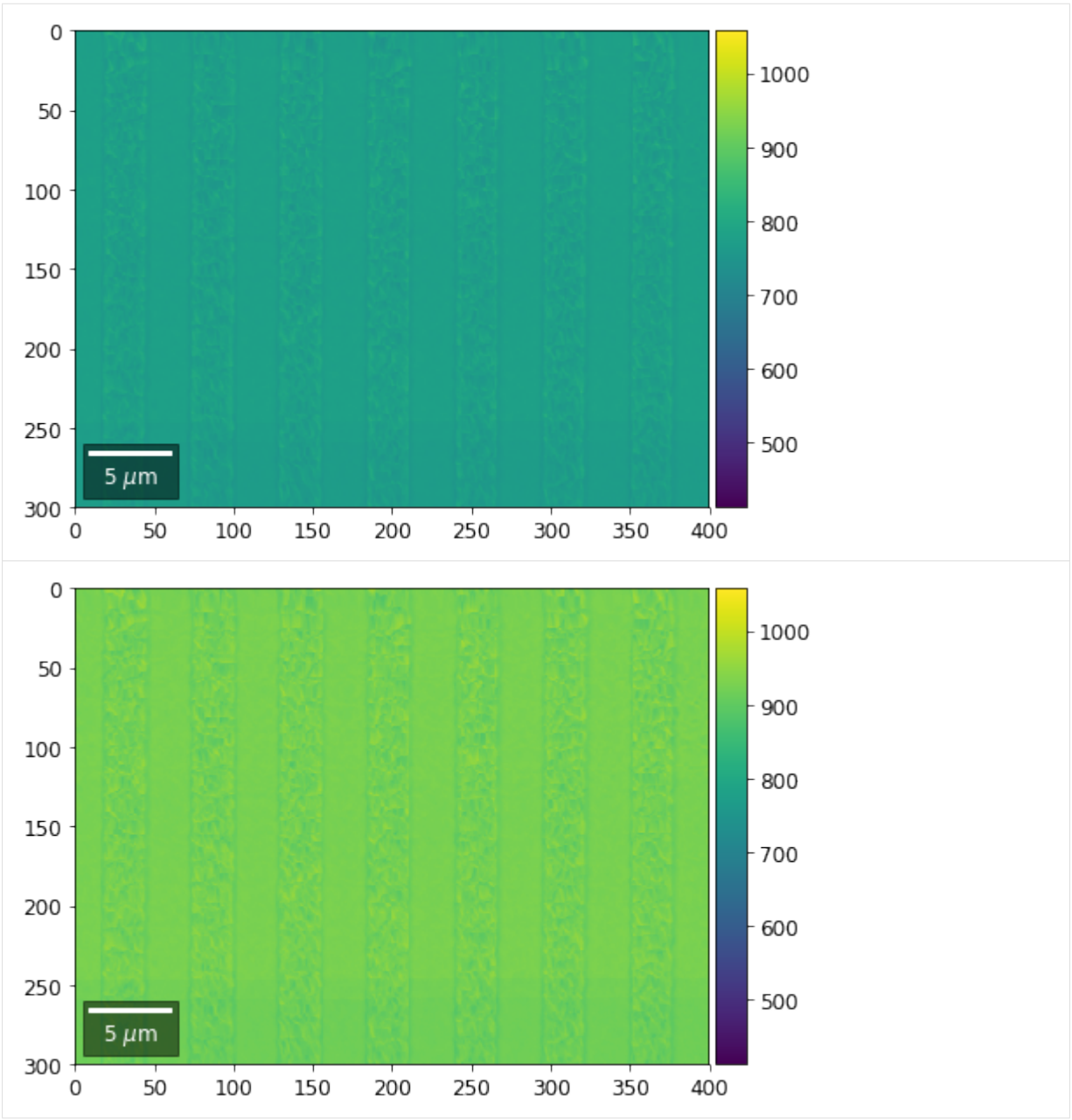
(continued from previous page)

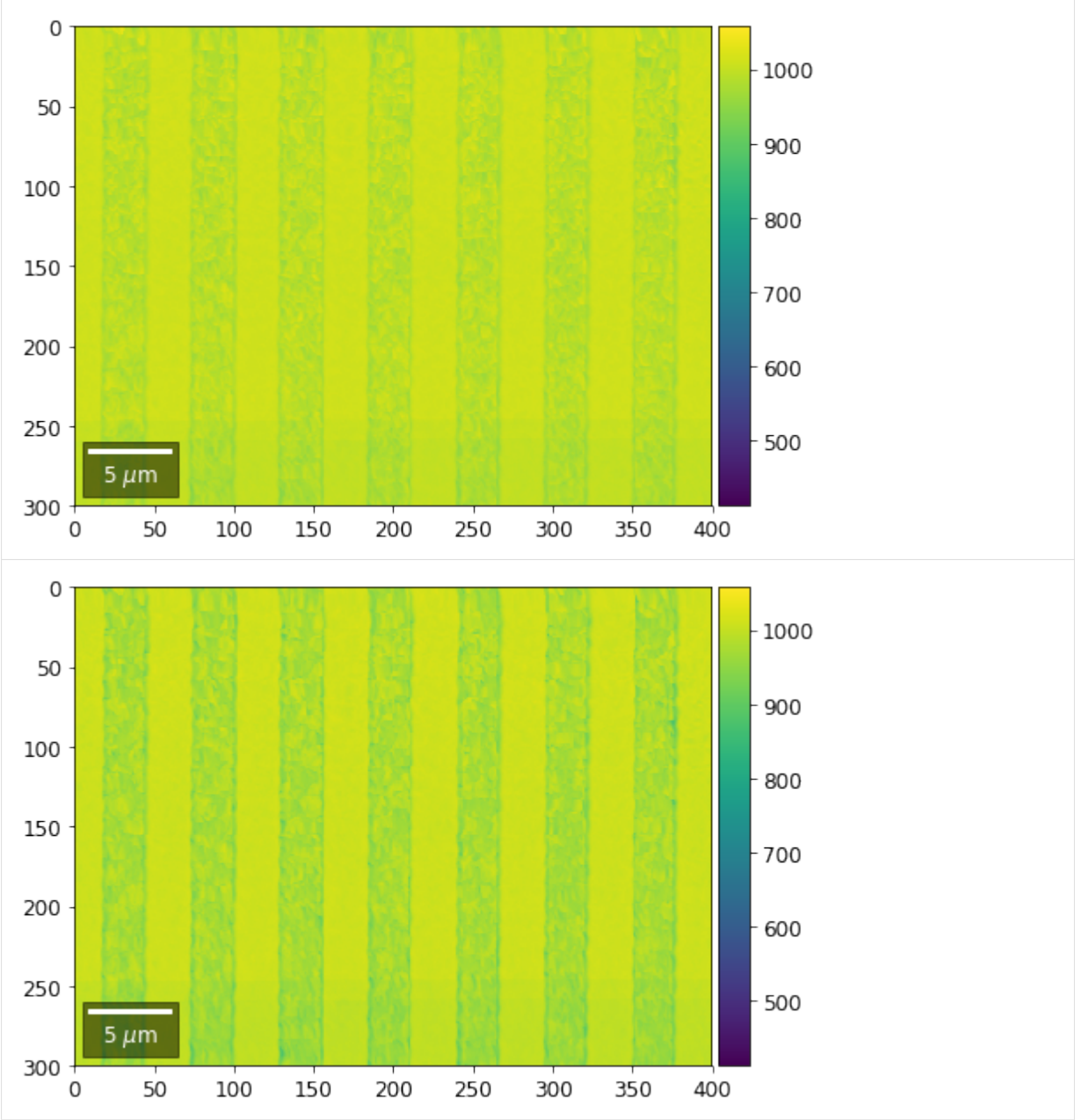
```
        bse_rows.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_absolute_'+str(row),
                 rot180=True, microns=step_map_microns)
```

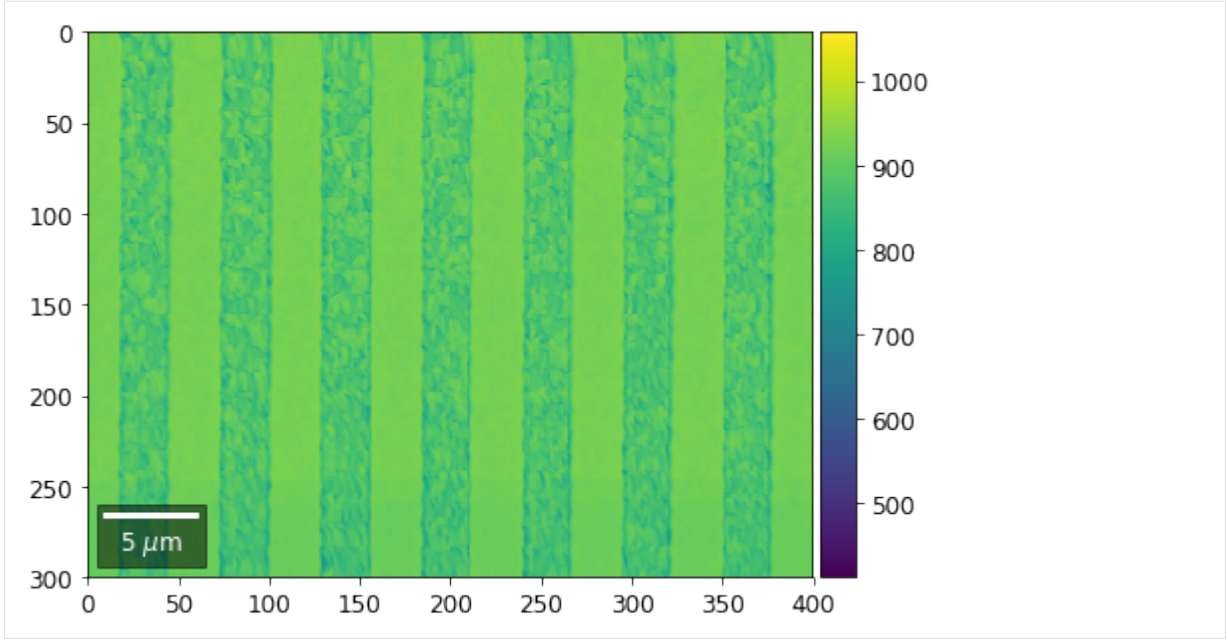
```
Range of Values:  [412.95351701880963, 1058.7636650741454]
```



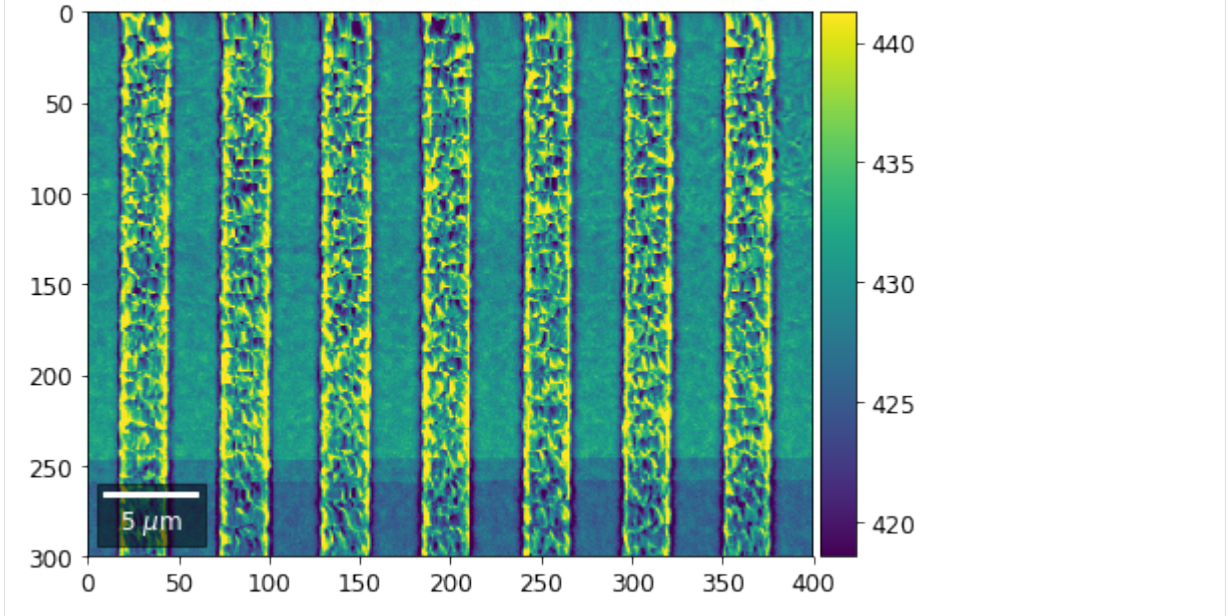


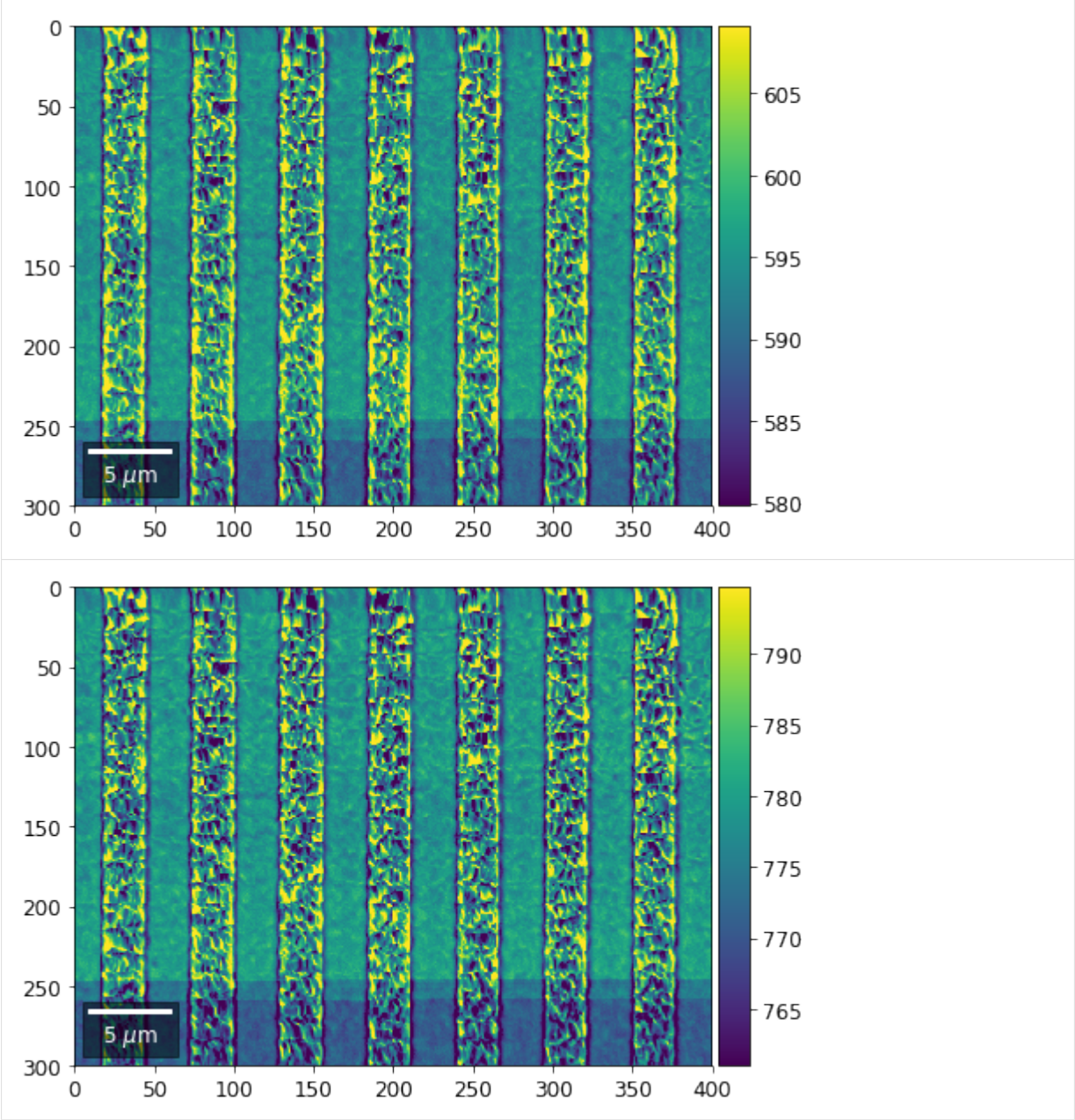

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1000
900
800
700
600
500

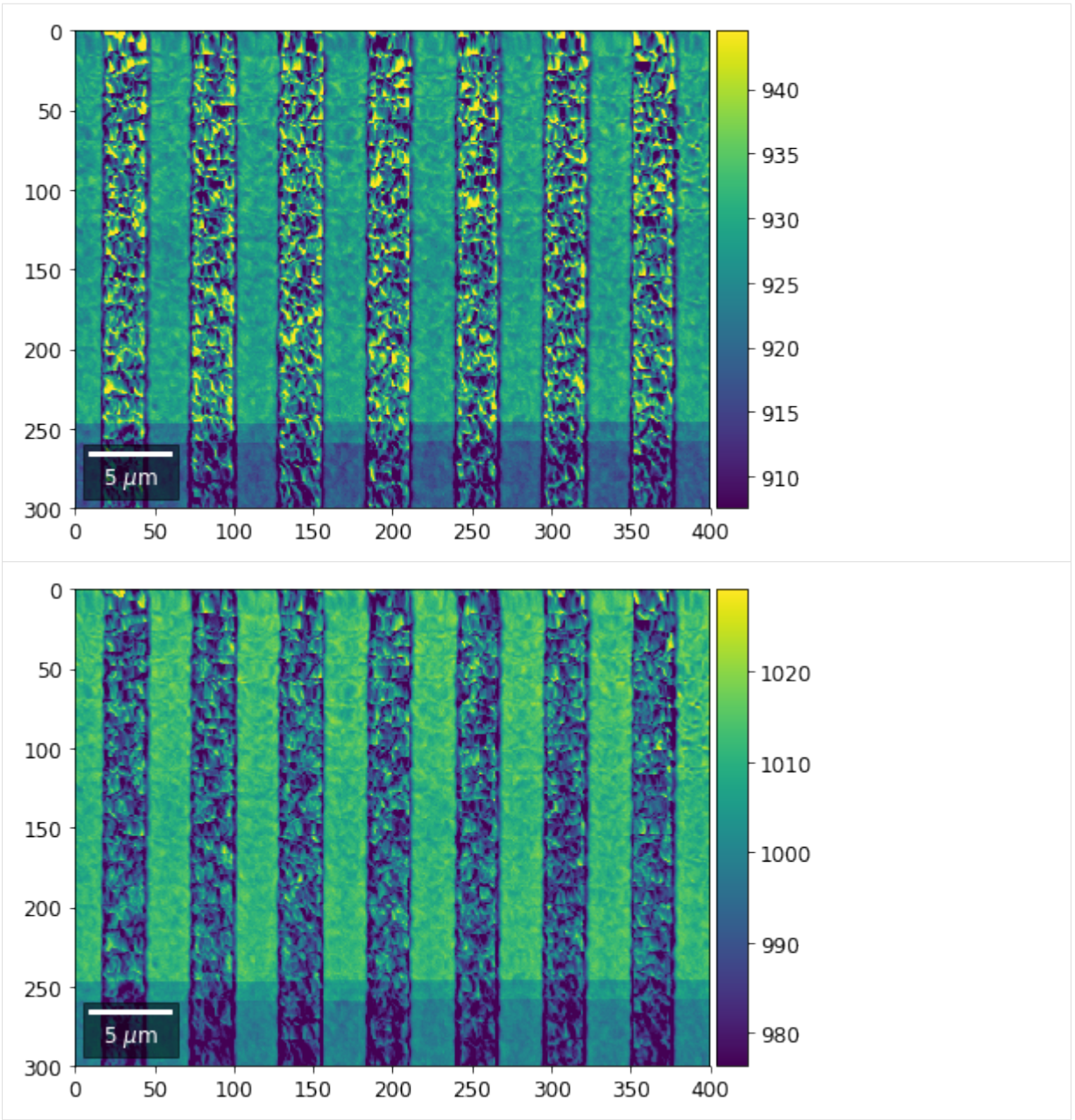

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1000
900
800
700
600
500

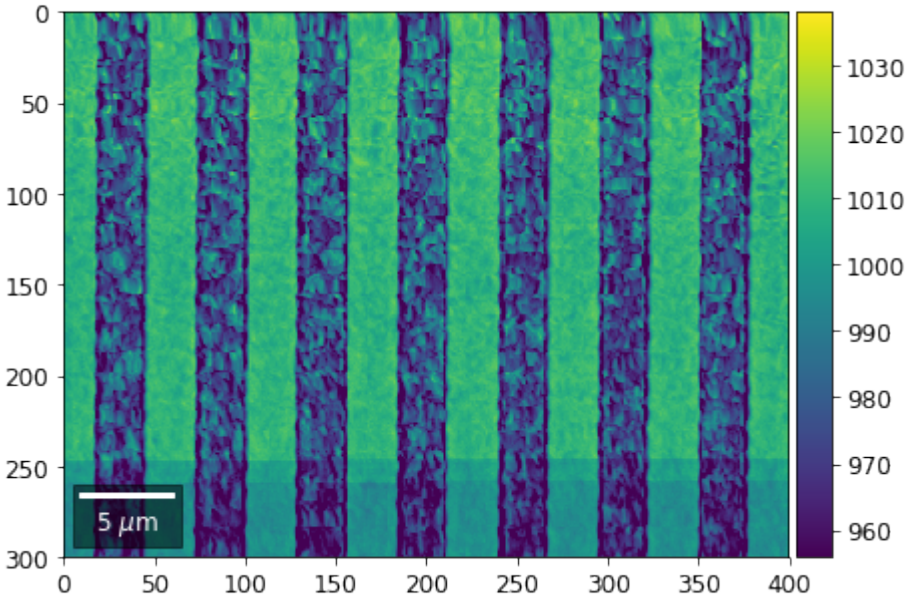


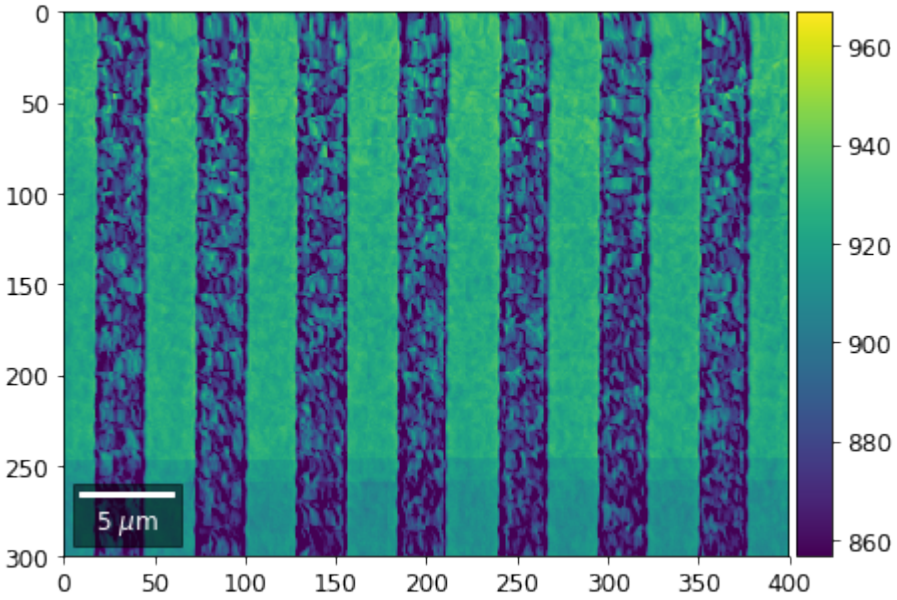
[24]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']
    # (3) make plots with  individual ranges for better contrast
    vrange=None
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_rows.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_individual_'+str(row),
                 rot180=True, microns=step_map_microns)
```




0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
605
600
595
590
585
580
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
790
785
780
775
770
765


0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
940
935
930
925
920
915
910
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1020
1010
1000
990
980





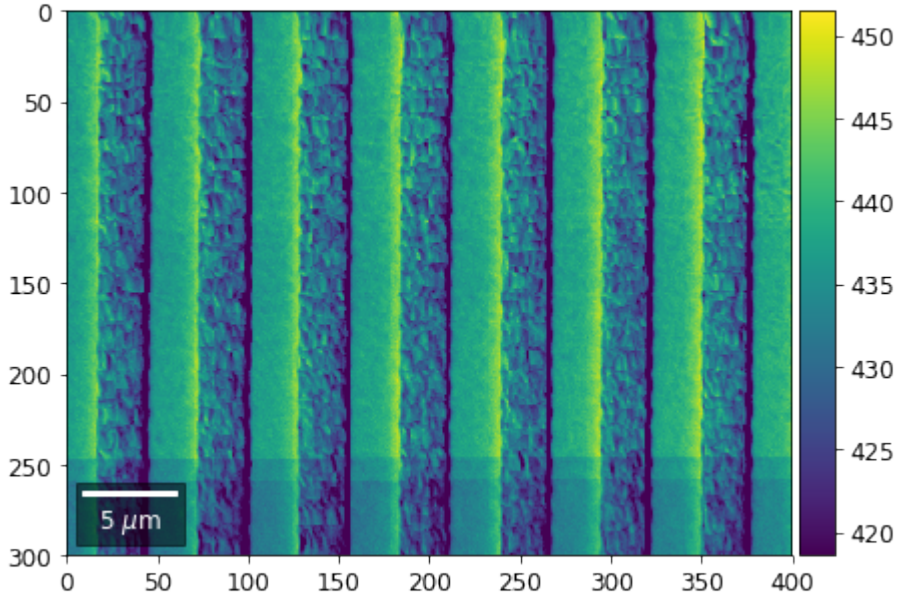
Columns

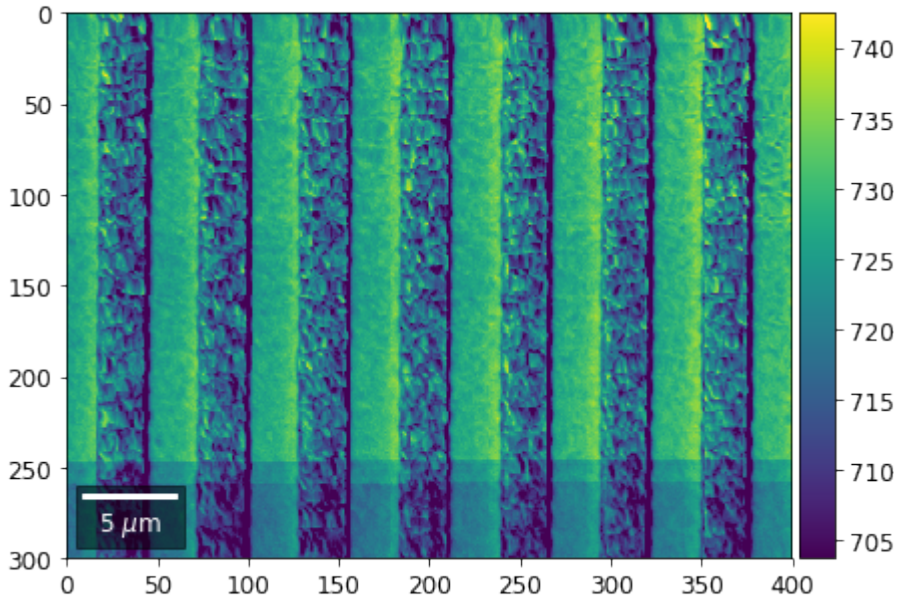
[25]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']
    # signal: sum of column
    vmin=400000
    vmax=0

    bse_cols = []

    # (1) get full range for all images
    for col in range(7):
        signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        minv, maxv = get_vrange(signal)
```

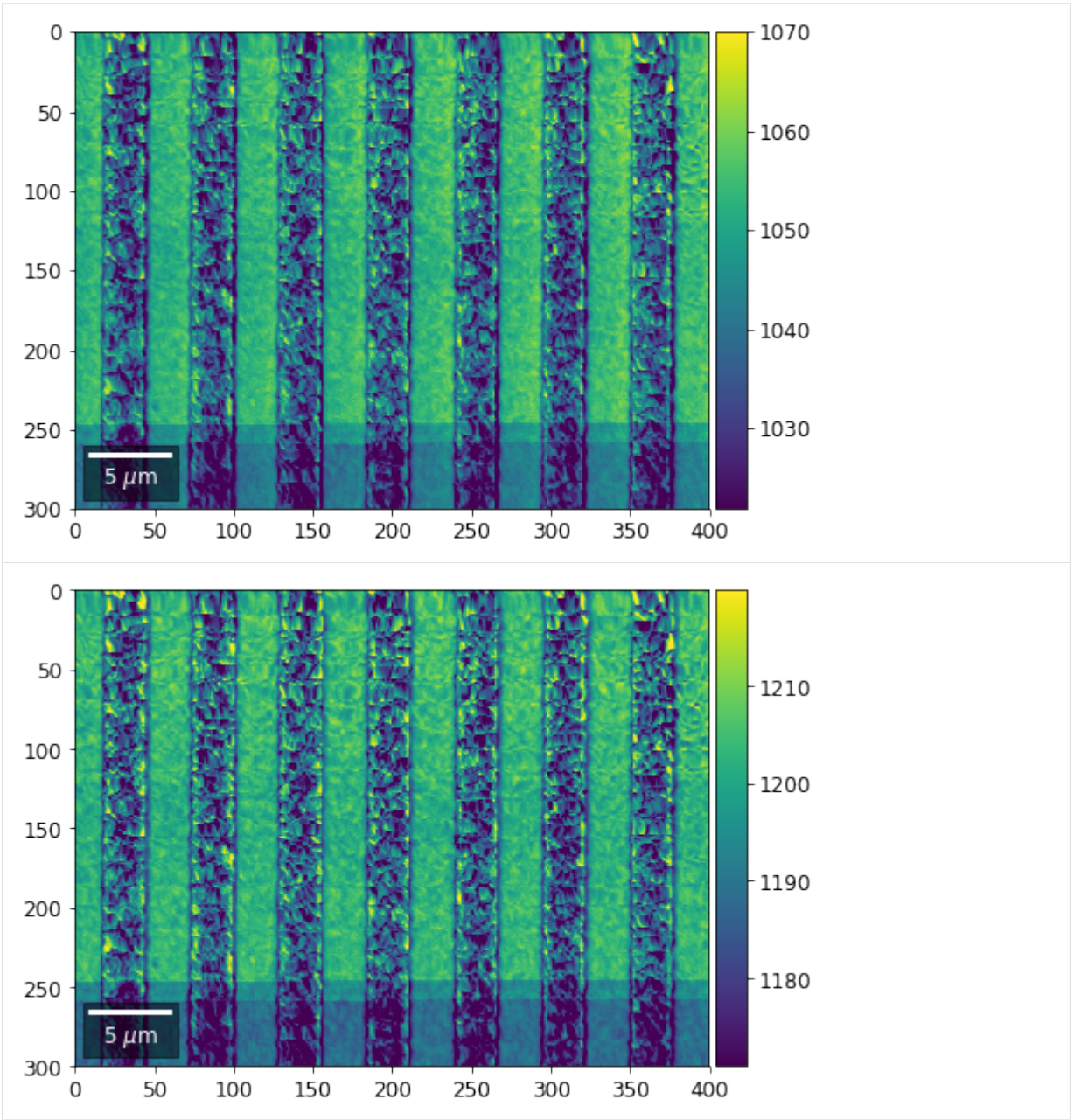
(continues on next page)

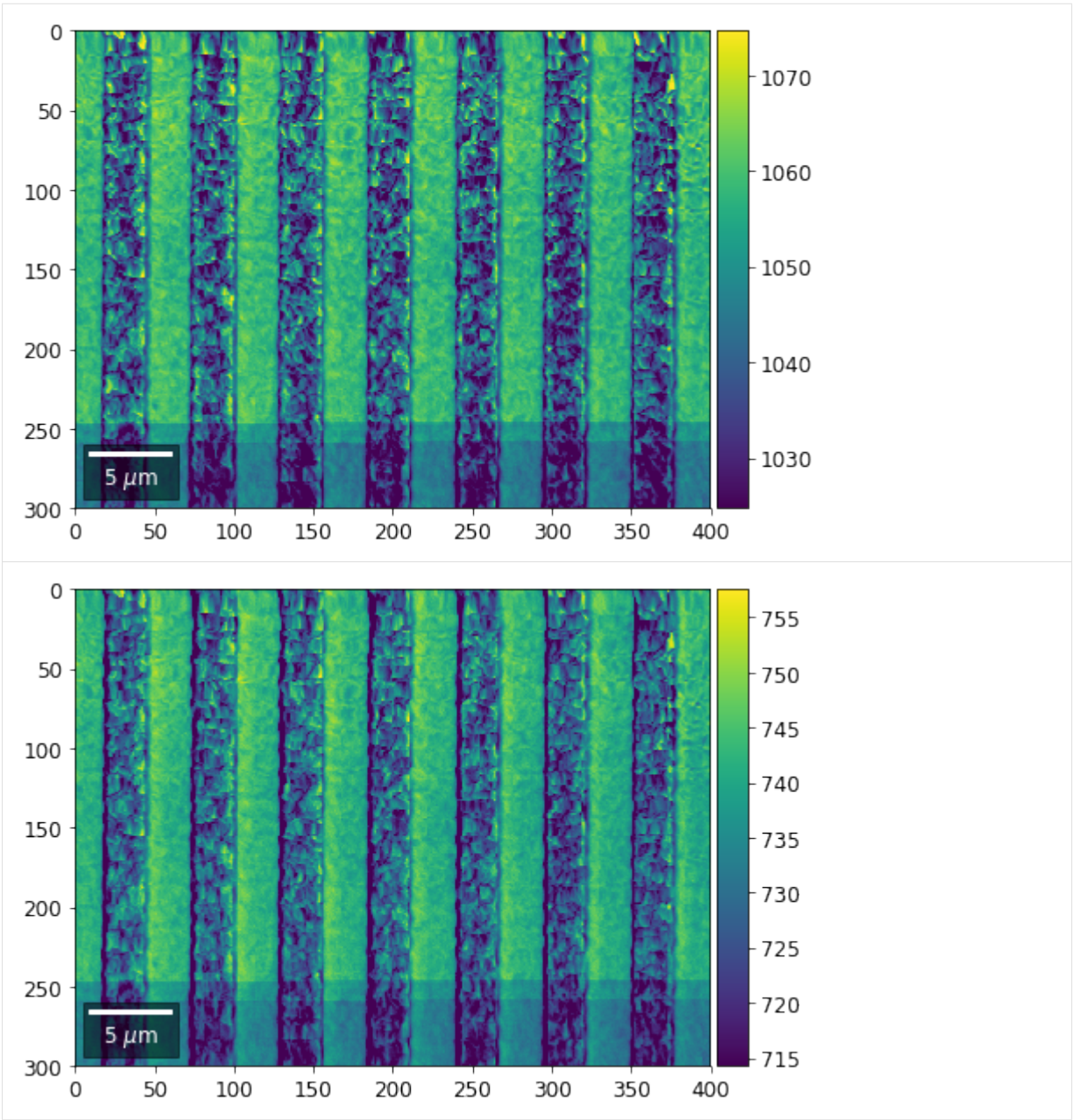
(continued from previous page)

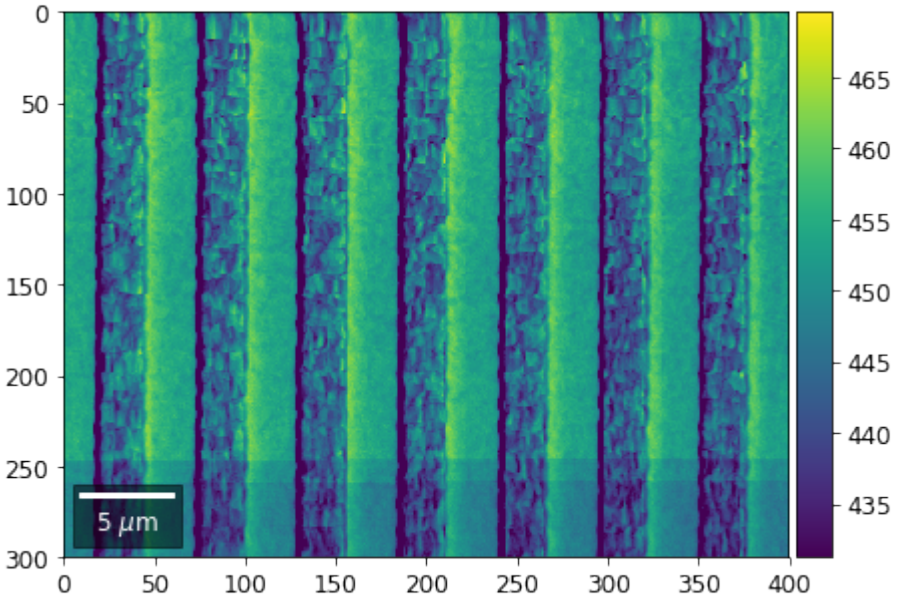
```
        if (minv<vmin):
            vmin=minv
        if (maxv>vmax):
            vmax=maxv

    # (2) make plots with same range for comparisons of absolute BSE values
    #vrange=[vmin, vmax]
    vrange=None # no fixed scale
    for col in range(7):
        signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_cols.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_col_'+str(col),
                 rot180=True, microns=step_map_microns)
```






0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1070
1060
1050
1040
1030
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1210
1200
1190
1180

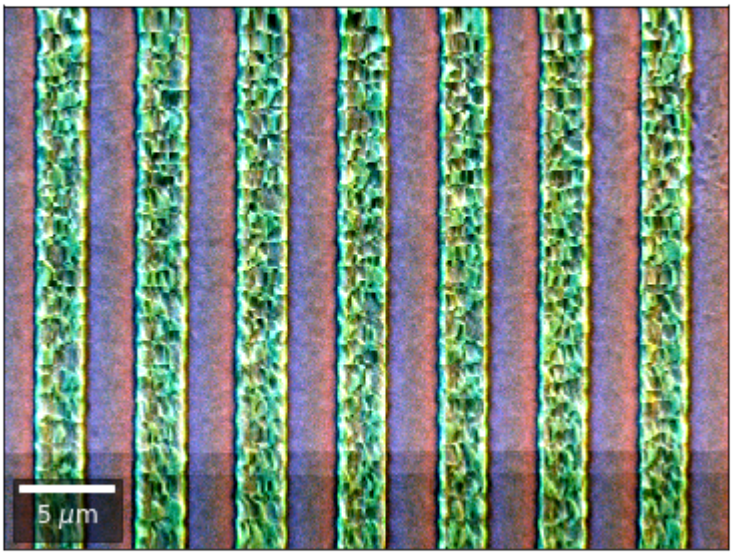

0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1070
1060
1050
1040
1030
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
755
750
745
740
735
730
725
720
715

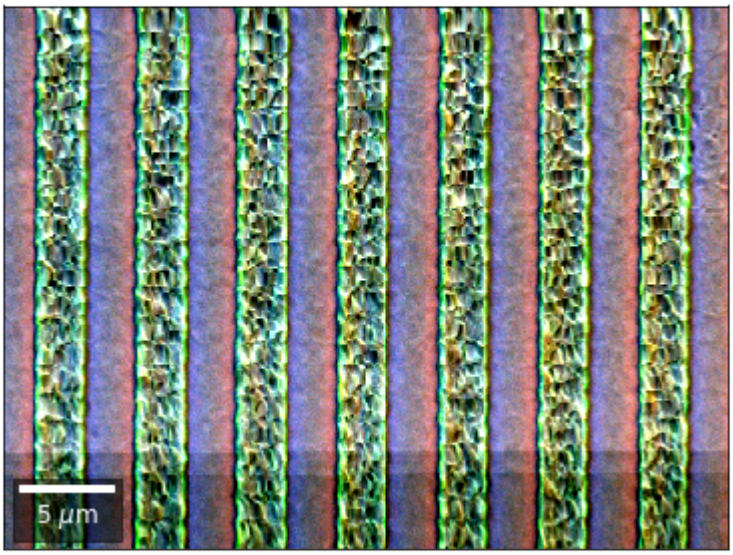


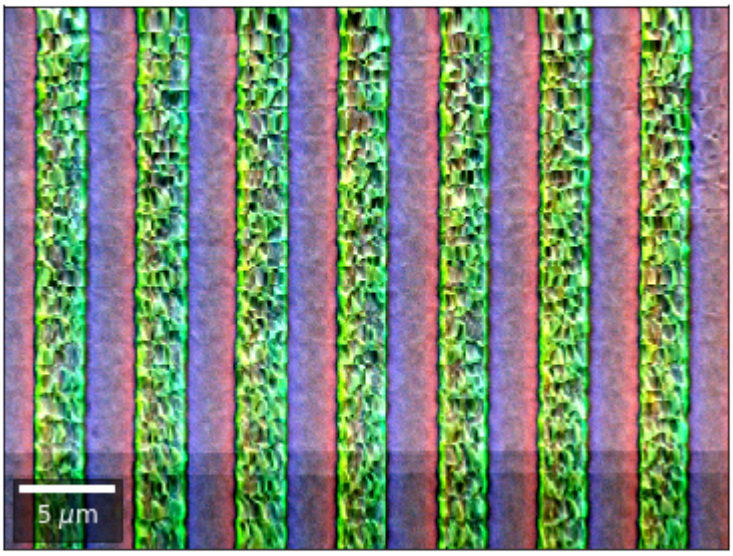
vBSE Color Imaging

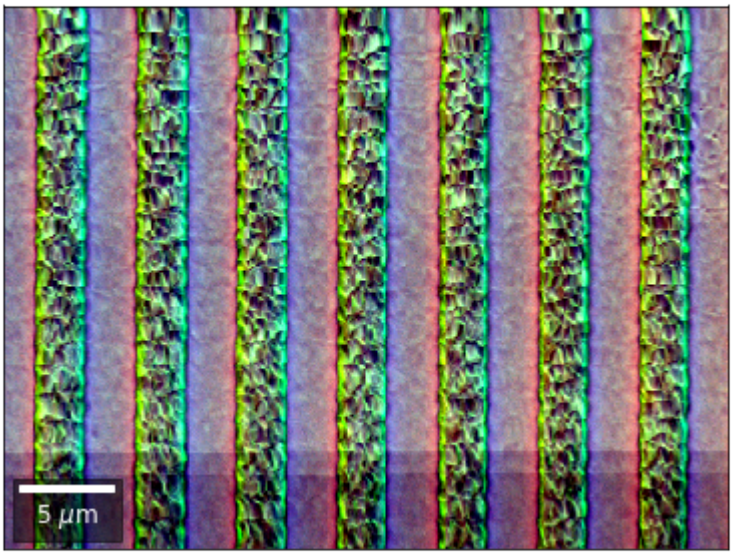
We can also form color images by assigning red, green, and blue channels to the left, middle, and right vBSE sensors of a row:

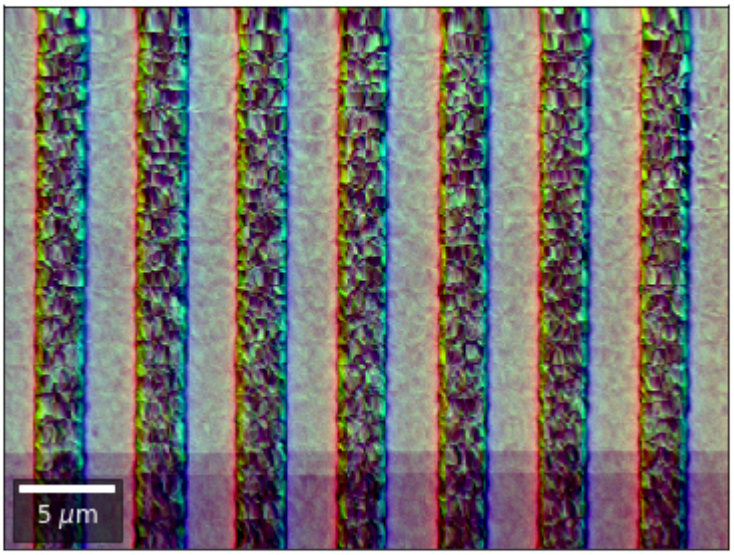
[26]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']# rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vFSD_RGB_row_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=0.8)

        rgb_direct.append(rgb)
```

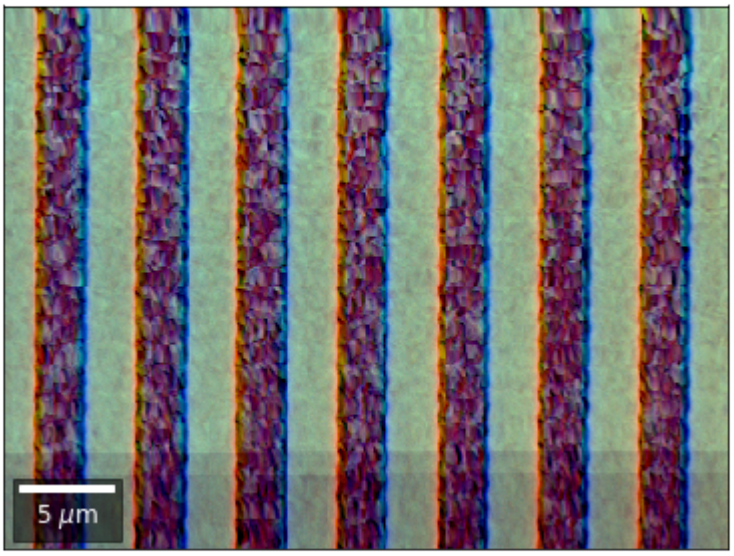

5 µm

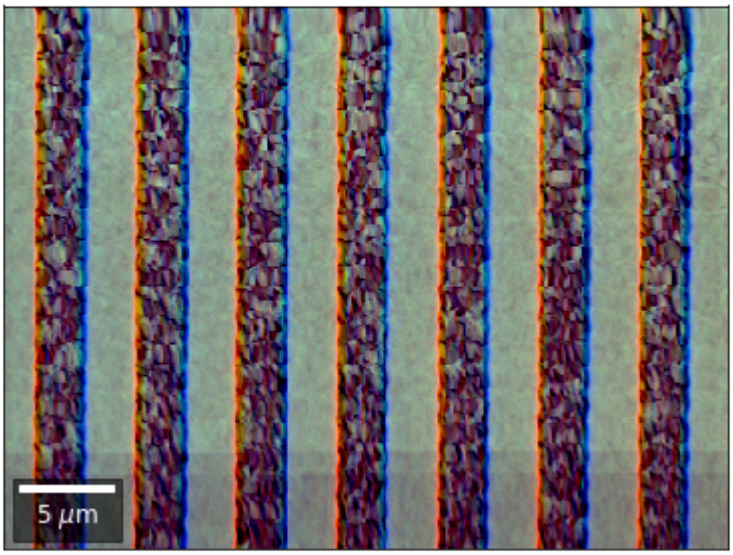

5 µm


5 µm


5 µm

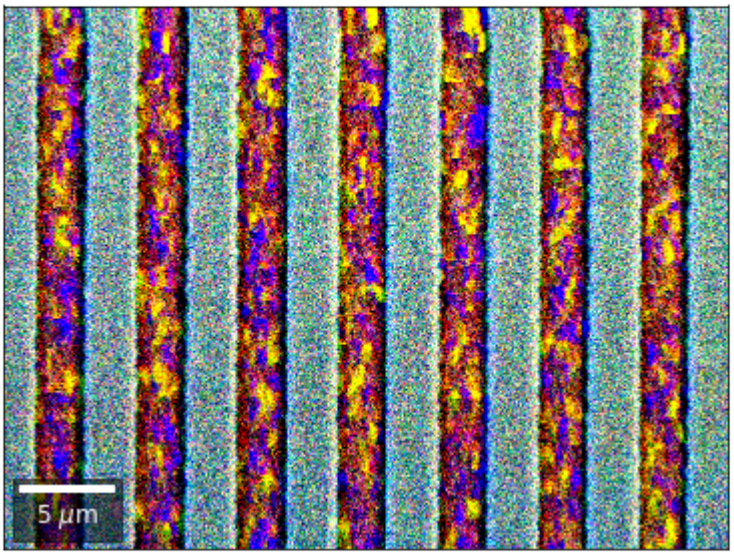

5 µm

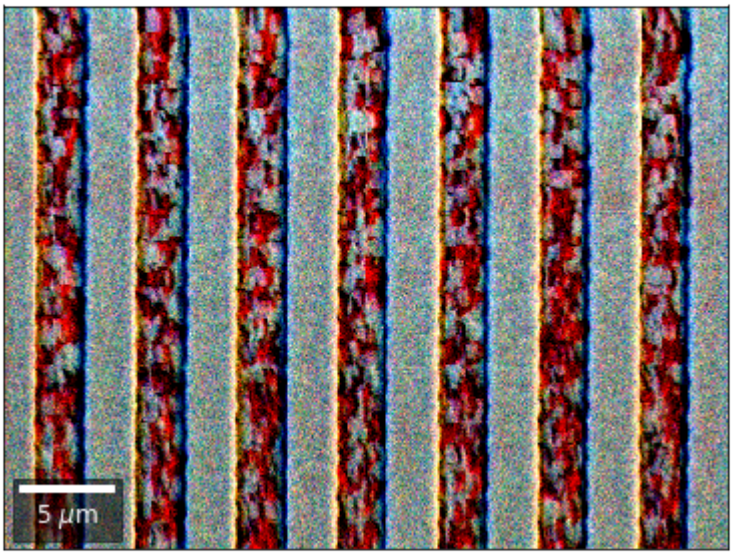

5 µm

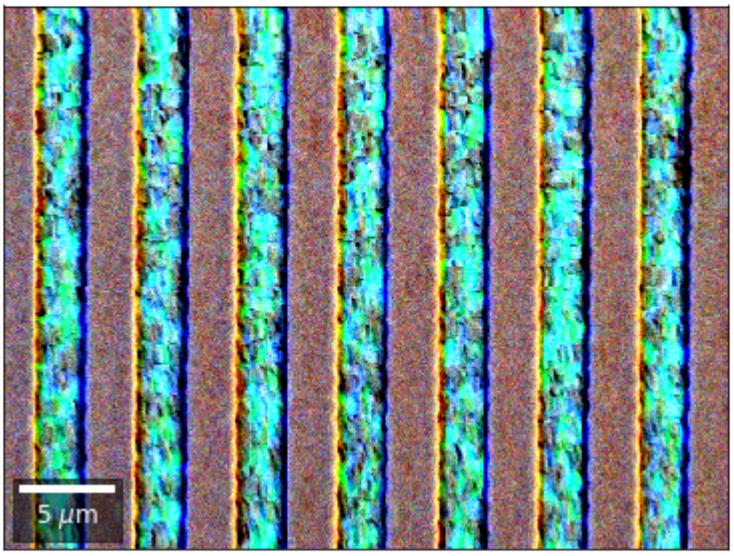


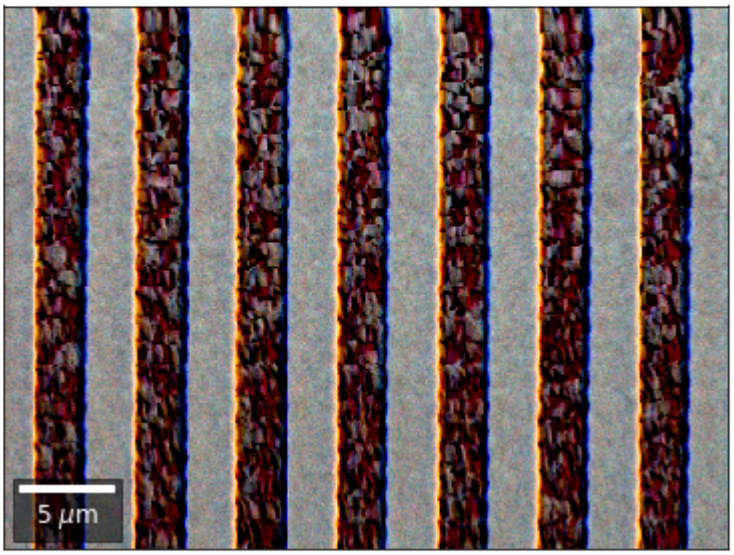
Differential color signals can be formed by calculating the relative changes to the ROI in the previous row:

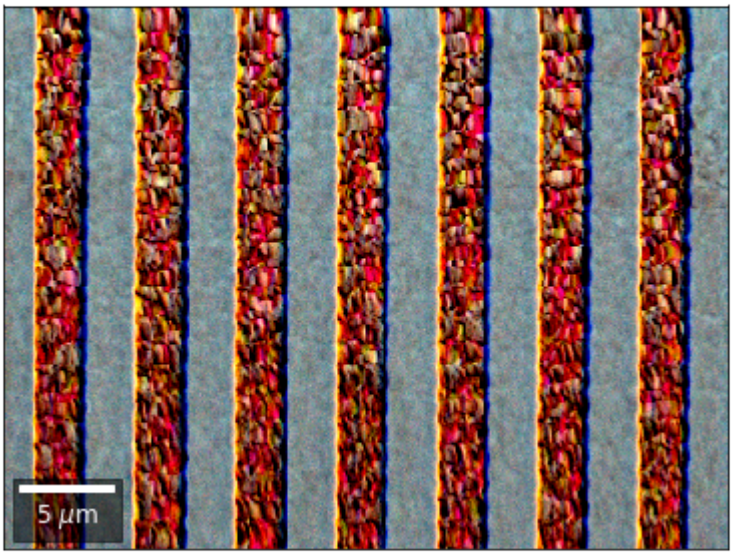
[27]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']# rgb direct# relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vFSD_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

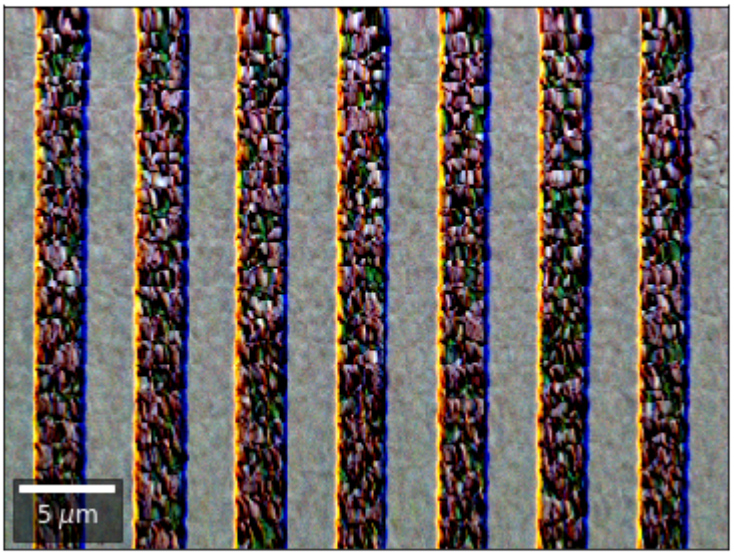

5 µm


5 µm


5 µm


5 µm


5 µm

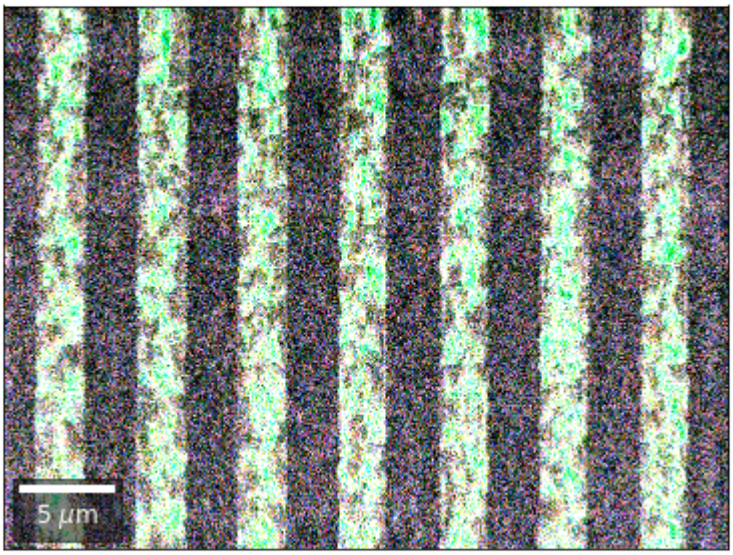

5 µm

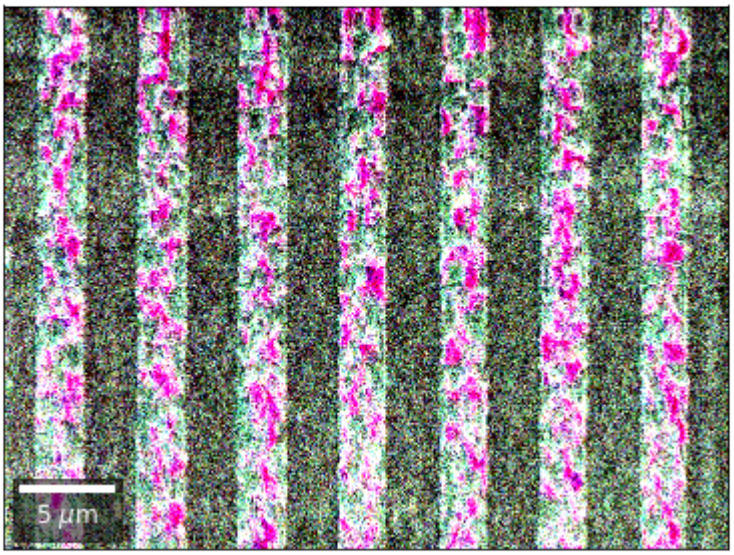
Kikuchi vBSE Imaging

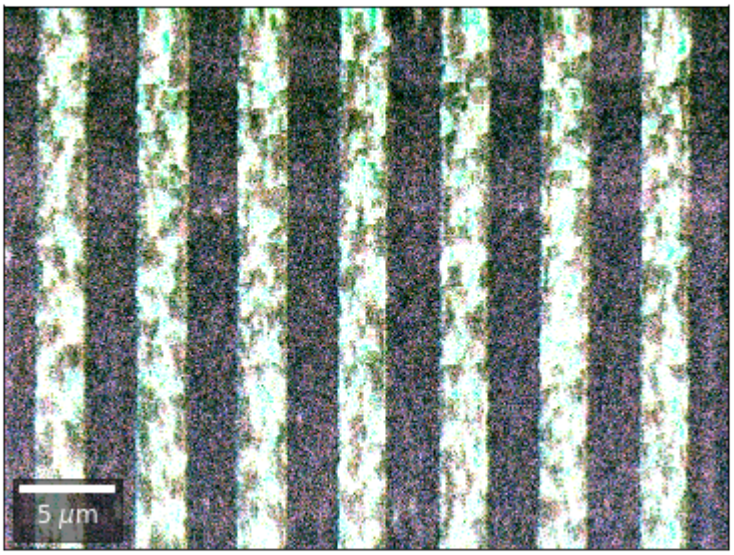
Kikuchi Pattern Array ROI as RGB

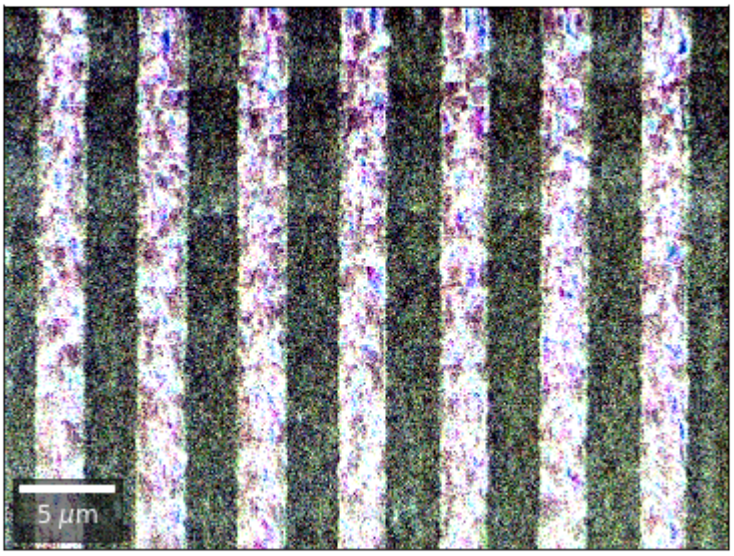
Not possible with simple BSE diodes!!!

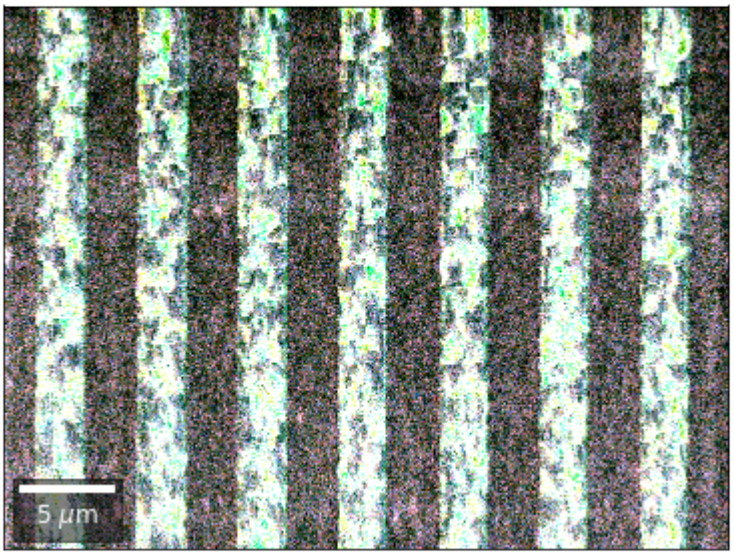
[28]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vkiku']

    # rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vKiku_RGB_row_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=1.2)

        rgb_direct.append(rgb)
```

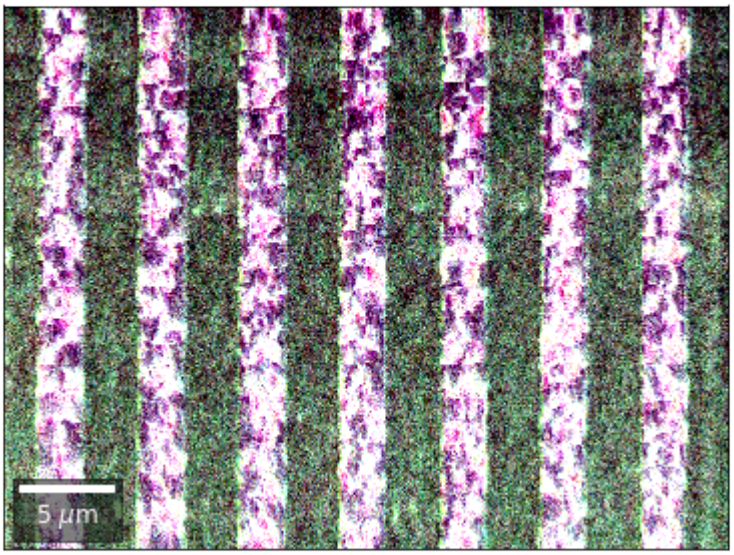


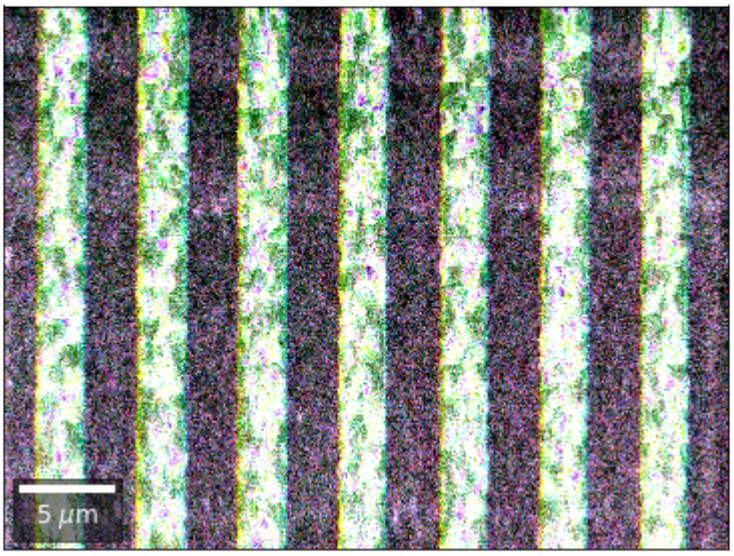

5 µm


5 µm


5 µm


5 µm


5 µm

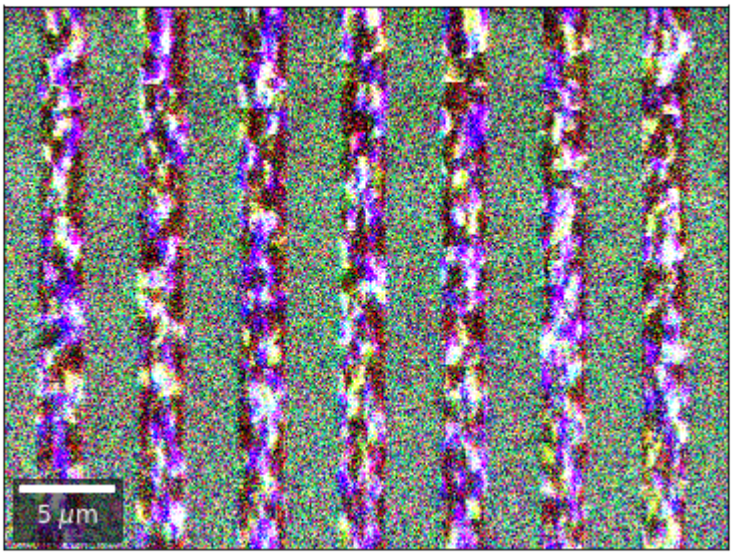

5 µm

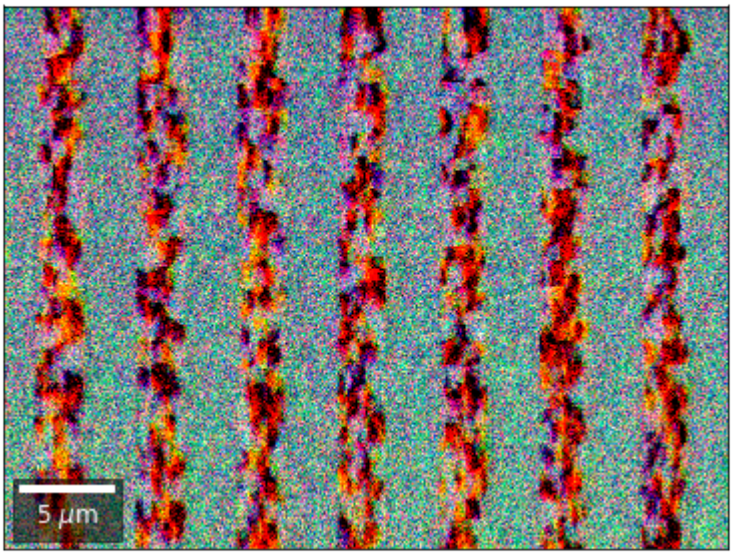
Differential Kikuchi Imaging

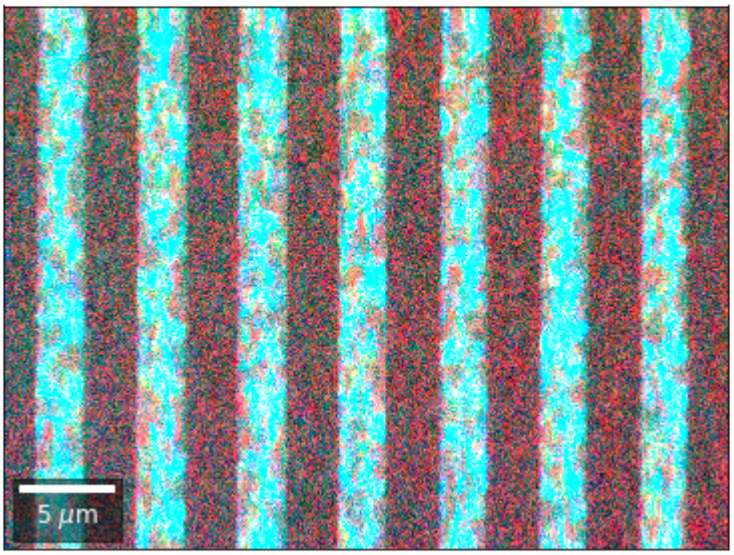
We determine the relative change between Kikuchi Array ROIs and use them as RGB values. The normalization to a reference ROI reduces the noise that is purely due to the variation of the background and the background processing on the complete pattern.

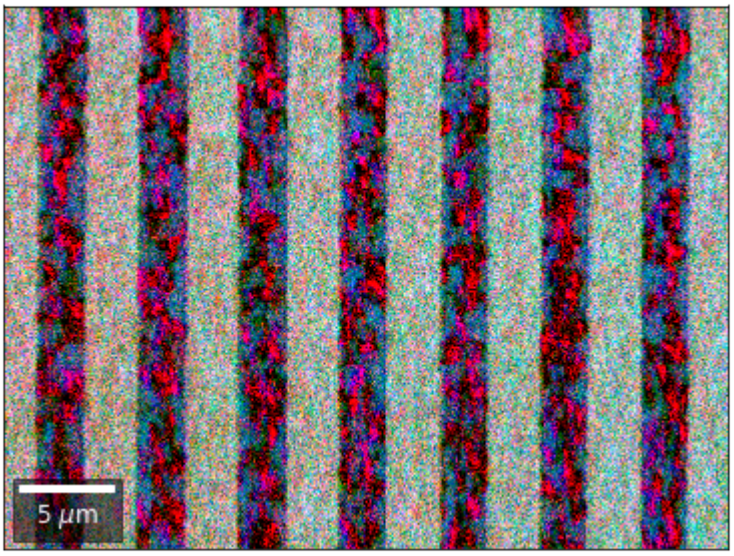
The colors represent orientation changes via the corresponding changes in ROIs of the Kikuchi patterns and the 7x7 array.

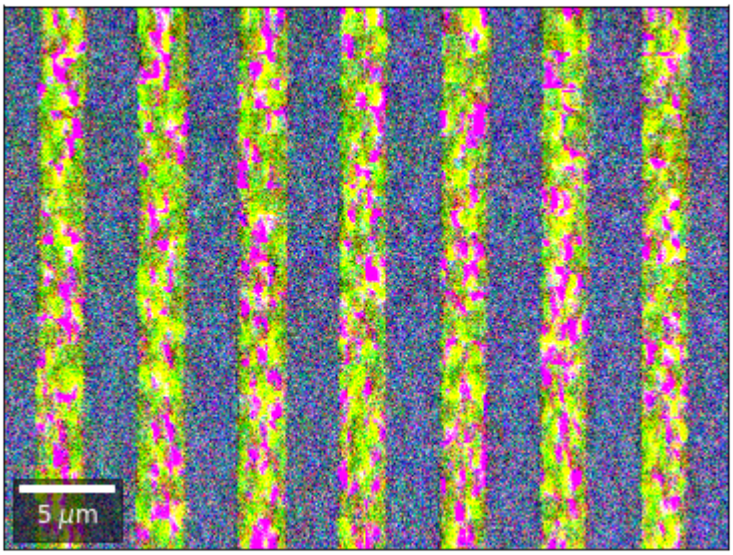
[29]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vkiku']
    # relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vKiku_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

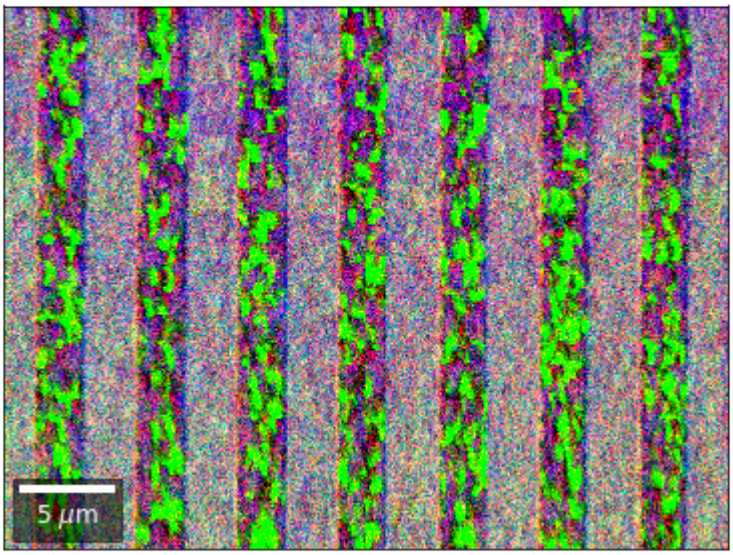



5 µm


5 µm


5 µm





Center of Mass Imaging

We can interpret the 2D image intensity as a mass density on a plane. The statistical moments of the density distribution (mean, variance, ...) can be used as signal sources. In the example below, we use the image center of mass as a signal source.

COM of Raw Patterns

[30]:
```
# calculate the center-of-mass for each pattern, use binning for speed
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_bin)
```

```
total points:120000 current:120000 finished -> total calculation time :    0.5 min
```

[31]:
```
# save the results in h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
```

(continues on next page)

(continued from previous page)

```
    h5f.create_dataset('/COM/COMxp_vbse', data=COMxp)
    h5f.create_dataset('/COM/COMyp_vbse', data=COMyp)
```

```
arBSE_GaN_Stripes.h5
```

COM of Kikuchi Patterns

This should be seen with caution, as the background removal process is never perfect and will tend to leave some residual intensity, so that the Kikuchi COM is correlated with the raw pattern COM (which is dominated by the smooth background intensity).

[32]:
```
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_kikuchi)
```

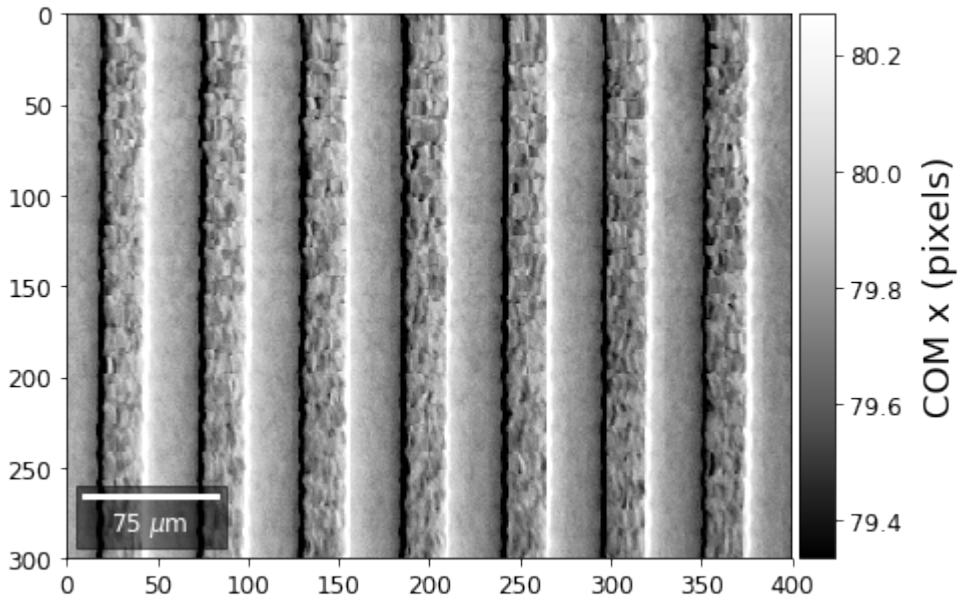
```
total points:120000 current:120000 finished -> total calculation time :    6.6 min
```

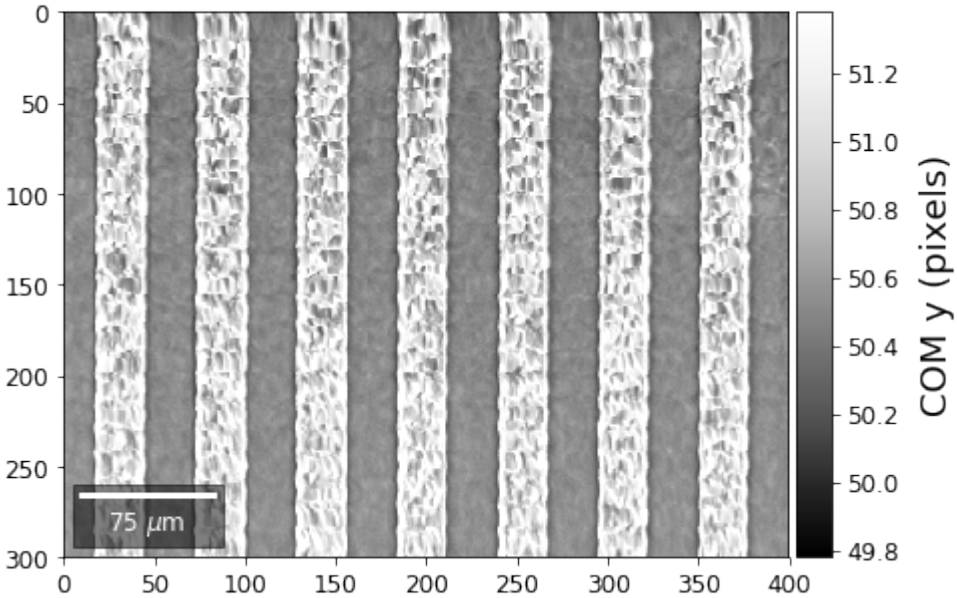
[33]:
```
# append to current h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('/COM/COMxp_kiku', data=COMxp)
    h5f.create_dataset('/COM/COMyp_kiku', data=COMyp)
```

```
arBSE_GaN_Stripes.h5
```

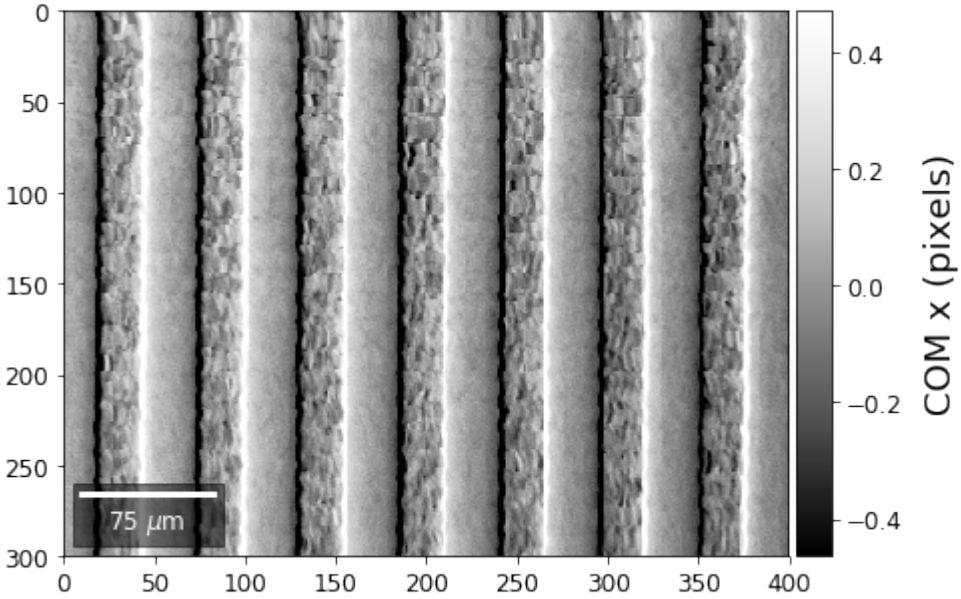
First, we calculate where the COMs are in x,y in pixels in the patterns:

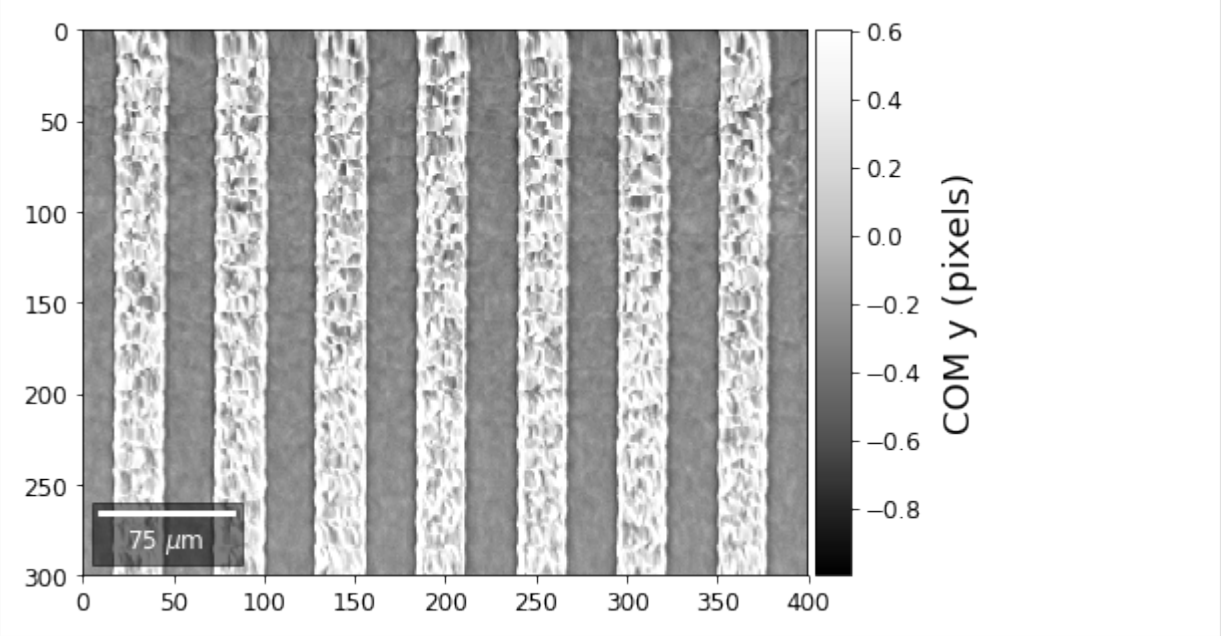
[34]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_vbse']
    COMyp = h5f['/COM/COMyp_vbse']

    comx_map0=make2Dmap(COMxp[:],XIndex,YIndex,MapHeight,MapWidth)
    comy_map0=make2Dmap(COMyp[:],XIndex,YIndex,MapHeight,MapWidth)

    plot_SEM(comx_map0, colorbarlabel='COM x (pixels)', cmap='Greys_r')
    plot_SEM(comy_map0, colorbarlabel='COM y (pixels)', cmap='Greys_r')
```



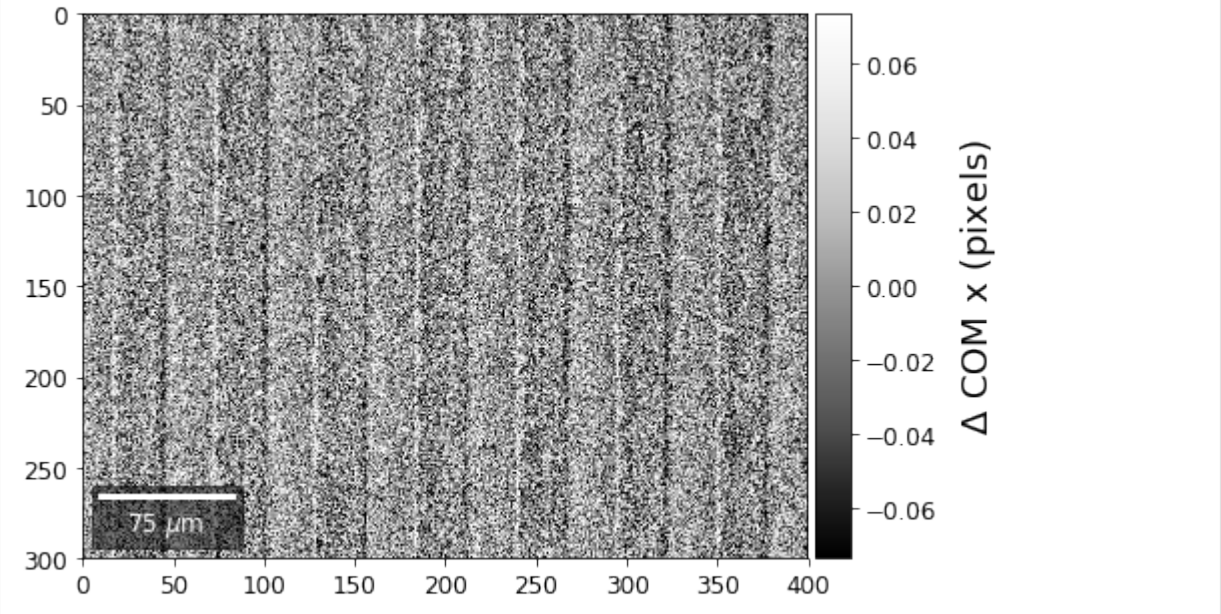


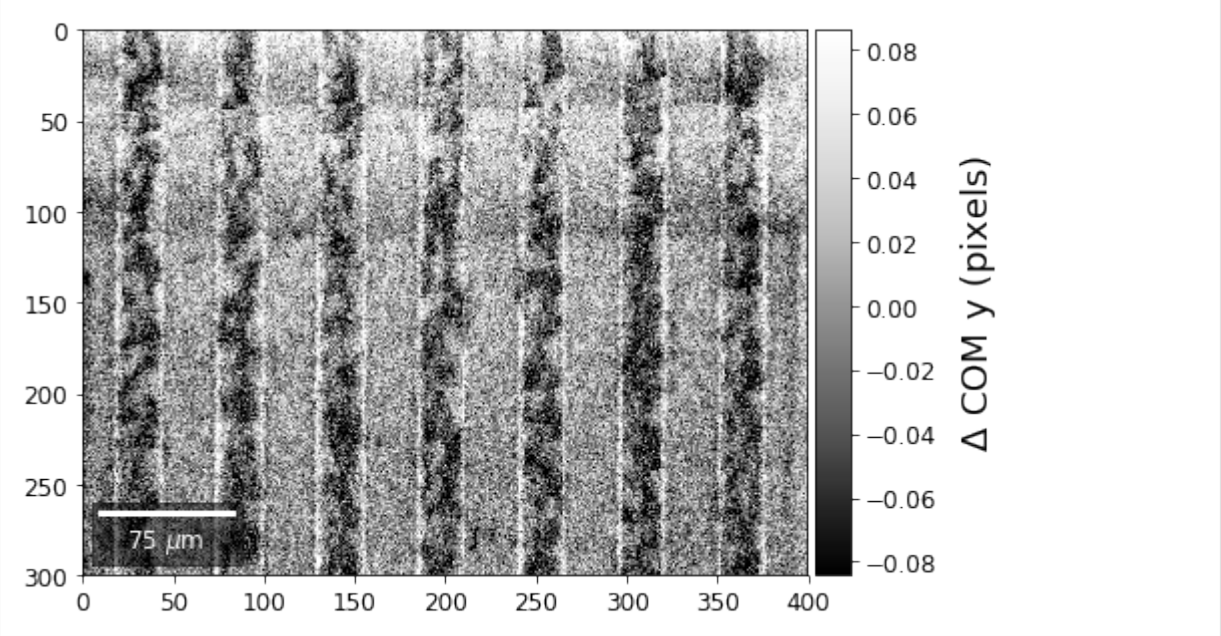
[35]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_vbse']
    COMyp = h5f['/COM/COMyp_vbse']

    meanx=np.mean(COMxp)
    meany=np.mean(COMyp)

    comx_map=make2Dmap(COMxp[:]-meanx,XIndex,YIndex,MapHeight,MapWidth)
    comy_map=make2Dmap(COMyp[:]-meany,XIndex,YIndex,MapHeight,MapWidth)

    plot_SEM(comx_map, colorbarlabel='COM x (pixels)', filename='comx', cmap='Greys_r')
    plot_SEM(comy_map, colorbarlabel='COM y (pixels)', filename='comy', cmap='Greys_r')
```





[36]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_kiku']
    COMyp = h5f['/COM/COMyp_kiku']
    meanx = np.mean(COMxp)
    meany = np.mean(COMyp)

    comx_map = make2Dmap(COMxp[:] - meanx, XIndex, YIndex, MapHeight, MapWidth)
    comy_map = make2Dmap(COMyp[:] - meany, XIndex, YIndex, MapHeight, MapWidth)

    plot_SEM(comx_map, colorbarlabel='$\Delta$ COM x (pixels)', filename='comx_kiku', cmap=
↪'Greys_r')
    plot_SEM(comy_map, colorbarlabel='$\Delta$ COM y (pixels)', filename='comy_kiku', cmap=
↪'Greys_r')
```





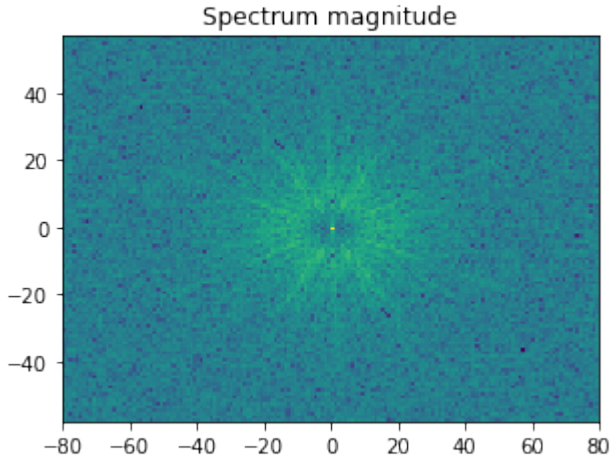
Fourier Transform Based Imaging

With the help of the Fast Fourier Transform (FFT), we can extract information on spatial frequencies (wave vectors) from an image. This can be used to derive imaging signals which are based on ranges of specific wave vectors present in the Kikuchi pattern.

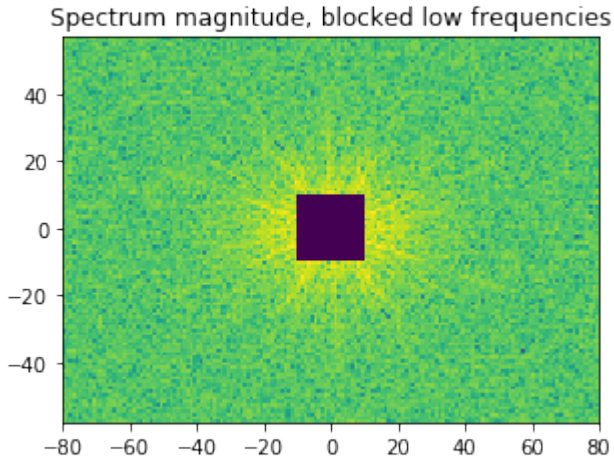
In the example shown below, we determine the intensity in the four quadrants of the FFT spectrum magnitude. The points corresponding to the low spatial frequencies (large spatial extensions) are removed to suppress the influence of the background signal.

[37]:
```
from scipy import fftpack
```

[38]:
```
image = process_kikuchi(Patterns[0])
M, N = image.shape
F = fftpack.fftn(image)

F_magnitude = np.abs(F)
F_magnitude = fftpack.fftshift(F_magnitude)


magnitude = np.log(1 + F_magnitude)

f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(magnitude, cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude');
```



[39]:
```
# Set block around center of spectrum to zero
K = 10
F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

magnitude = np.log(1 + F_magnitude)

f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(magnitude, cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude, blocked low frequencies');
```



[40]:
```
def get_fft_sector_mask(image, isector, rmin=0.1, rmax=0.4,
                        width_rad=np.pi/8, offset_rad=0.0):
    """
    get specified 45 deg sector mask from fft magnitude spectrum of image
    """
    F = fftpack.fftn(image)
    F_magnitude = np.abs(F)
    F_magnitude = fftpack.fftshift(F_magnitude)

    phi_start = (isector % 4) * np.pi/4 + offset_rad
    phi_end   = phi_start + width_rad
    sector_mask = np.ones_like(F_magnitude)
    M, N = F_magnitude.shape
```

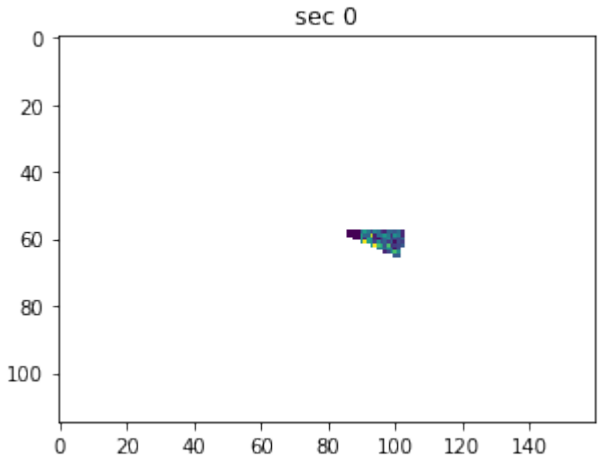
(continues on next page)

(continued from previous page)

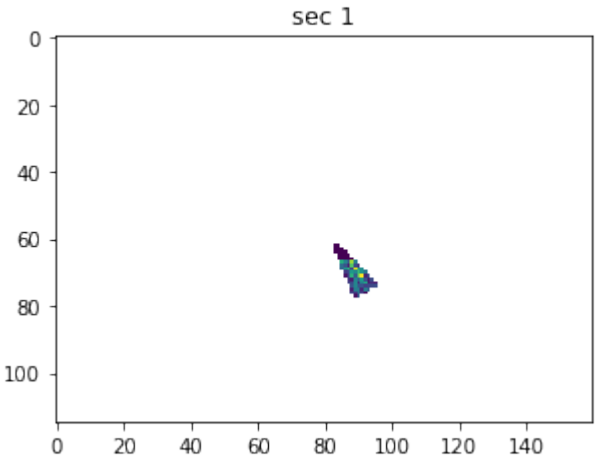
```
    M2 = M // 2
    N2 = N // 2
    for ix in range(N):
        for iy in range(M):
            rx = ix-N2
            ry = iy-M2
            phi = np.arctan2(ry, rx)
            r = np.sqrt(rx**2 + ry**2)
            if (phi>phi_start) and (phi<phi_end) and (r<= M2 * rmax ) and (r>=M2*rmin) :
                sector_mask[iy,ix] = 0

    #profile_quality_map = np.ma.MaskedArray(profile_quality_map, mask=calcmask)
    return sector_mask

def get_fft_sector(spectrum2d, mask):
    return np.ma.MaskedArray(spectrum2d, mask=mask)
```

[41]:
```
mask_0 = get_fft_sector_mask(Patterns[0], 0)
mask_1 = get_fft_sector_mask(Patterns[0], 1)
mask_2 = get_fft_sector_mask(Patterns[0], 2)
mask_3 = get_fft_sector_mask(Patterns[0], 3)

sec_0 = get_fft_sector(F_magnitude, mask_0)
sec_1 = get_fft_sector(F_magnitude, mask_1)
sec_2 = get_fft_sector(F_magnitude, mask_2)
sec_3 = get_fft_sector(F_magnitude, mask_3)
```

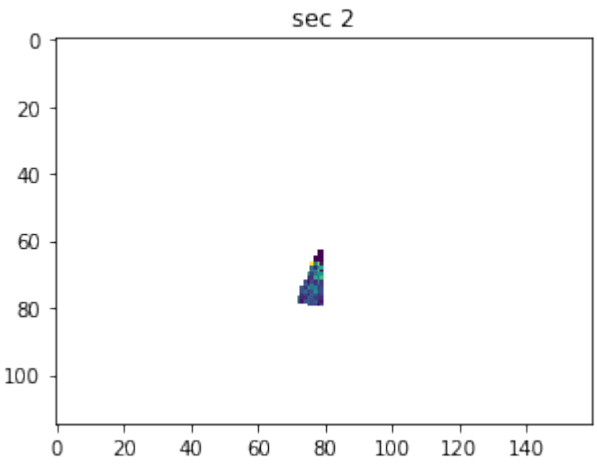
[42]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_0, cmap='viridis',)
ax.set_title('sec 0');
```



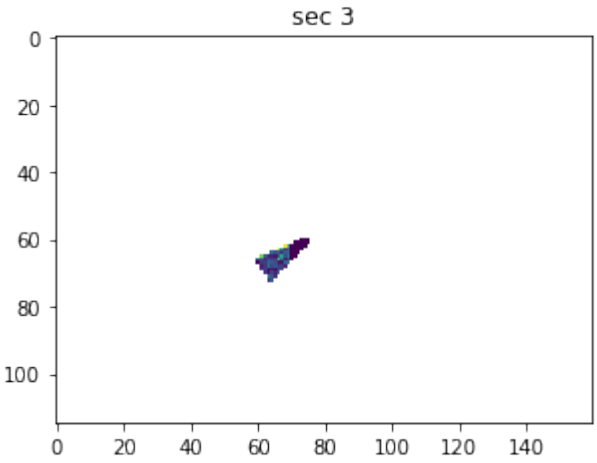
[43]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_1, cmap='viridis',)
ax.set_title('sec 1');
```



[44]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_2, cmap='viridis',)
ax.set_title('sec 2');
```



[45]:
```
f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(sec_3, cmap='viridis',)
ax.set_title('sec 3');
```



[46]:
```
def secfft(image, masks, K=5):
```

(continues on next page)

(continued from previous page)

```
    """
    calculate FFT magnitude sectors

    remove +/- K points in center (low spatial frequencies)
    limit to +/- W points away from center (low pass, avoid noise at higher frequencies)

    """
    M, N = image.shape
    mask_0, mask_1, mask_2, mask_3 = masks

    F = fftpack.fftn(image)

    F_magnitude = np.abs(F)
    F_magnitude = fftpack.fftshift(F_magnitude)
    #F_magnitude = np.log(1 + F_magnitude)

    # Set block +/-K around center of spectrum to zero
    F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

    sec_0 = get_fft_sector(F_magnitude, mask_0)
    sec_1 = get_fft_sector(F_magnitude, mask_1)
    sec_2 = get_fft_sector(F_magnitude, mask_2)
    sec_3 = get_fft_sector(F_magnitude, mask_3)

    s0 = np.sum(sec_0)
    s1 = np.sum(sec_1)
    s2 = np.sum(sec_2)
    s3 = np.sum(sec_3)

    return np.array([s0, s1, s2, s3])



def calc_secfft(patterns, process=None, K=4):
    """
    calc magnitude sum in FFT quadrants of image
    "process" is an optional image pre-processing function
    """
    npatterns = patterns.shape[0]
    fft_sec_magnitudes = np.zeros((npatterns, 4))

    mask_0 = get_fft_sector_mask(patterns[0], 0)
    mask_1 = get_fft_sector_mask(patterns[0], 1)
    mask_2 = get_fft_sector_mask(patterns[0], 2)
    mask_3 = get_fft_sector_mask(patterns[0], 3)

    masks = (mask_0, mask_1, mask_2, mask_3)

    tstart = time.time()
    for i in range(npatterns):

        # get current pattern
        if process is None:
            fft_sec_magnitudes[i,:] = secfft(patterns[i,:,:], masks, K=K)
        else:
            fft_sec_magnitudes[i,:] = secfft(process(patterns[i,:,:]), masks, K=K)

        print_progress_line(tstart, i, npatterns)
```

(continues on next page)

(continued from previous page)

```
    return fft_sec_magnitudes
```

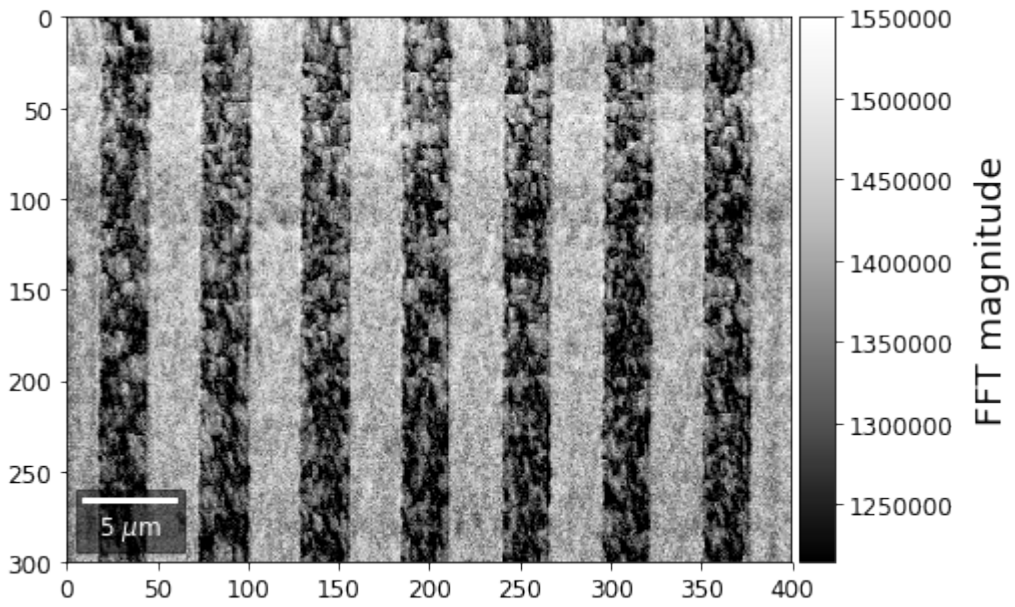
[47]:
```
secfftmag =  calc_secfft(Patterns, process=process_kikuchi)
```

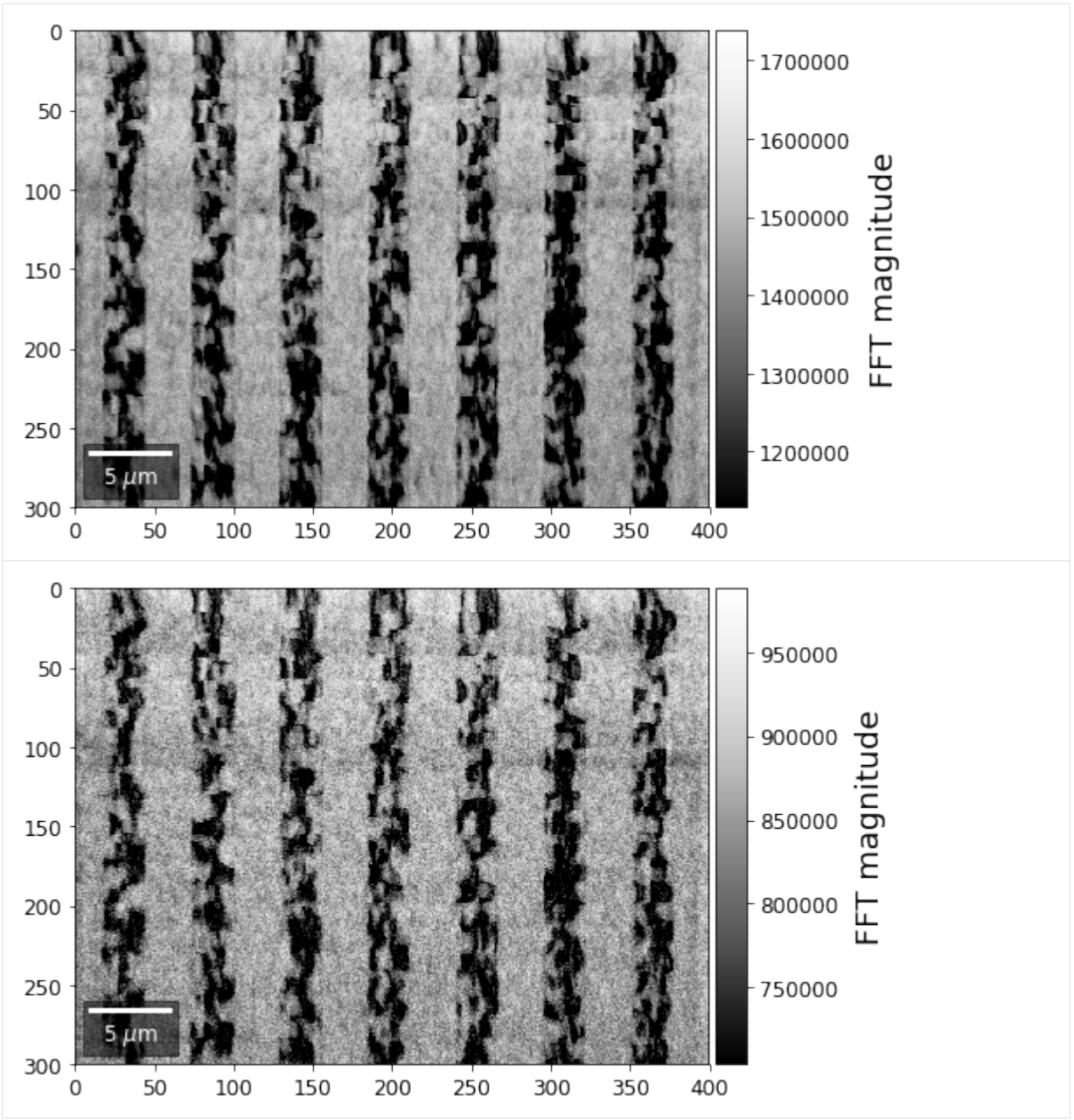
```
total points:120000 current:120000 finished -> total calculation time :    7.2 min
```

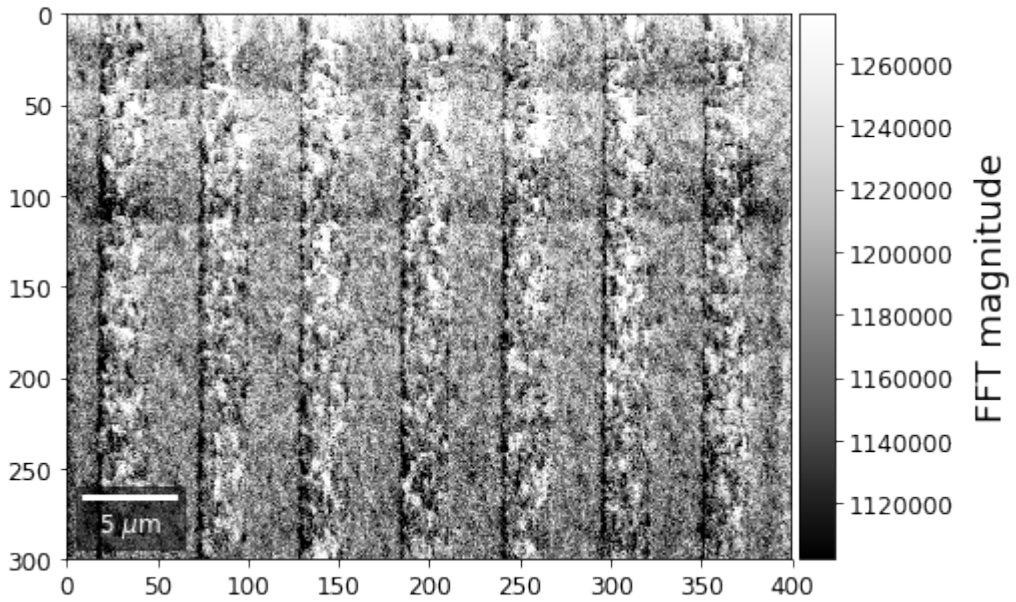
[48]:
```
# append to current h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('/fft_sectors', data=secfftmag)
```

```
arBSE_GaN_Stripes.h5
```

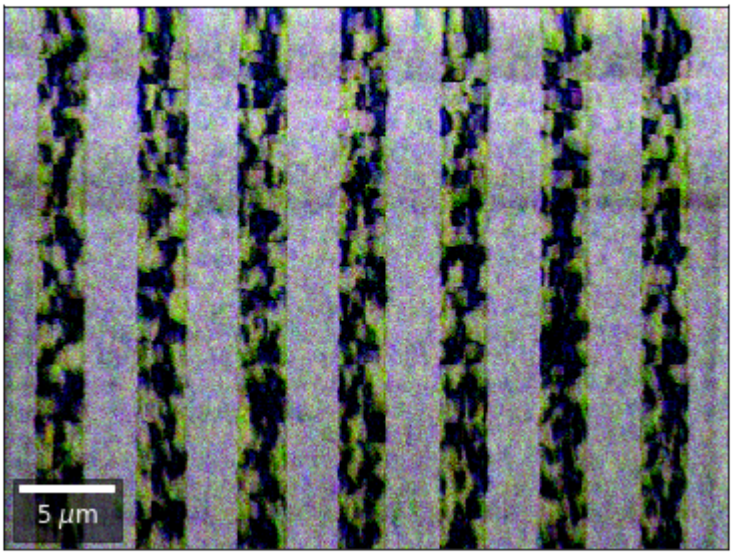
[49]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    secfftmag = np.copy(h5f['fft_sectors'])
    for q, sector in enumerate(secfftmag.T):
        signal_map = make2Dmap(sector,XIndex,YIndex,MapHeight,MapWidth)
        plot_SEM(signal_map, vrange=None, cmap='Greys_r',
                 colorbarlabel='FFT magnitude', microns=step_map_microns,
                 filename='secFFT_'+str(q))
```




0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
1700000
1600000
1500000
1400000
1300000
1200000
FFT magnitude
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350
400
5 μm
950000
900000
850000
800000
750000
FFT magnitude



[50]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    secfftmag = h5f['fft_sectors']

    signal = secfftmag[:,1]
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,2]
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,0]
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='fft_sectors_RGB', microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.0)
```

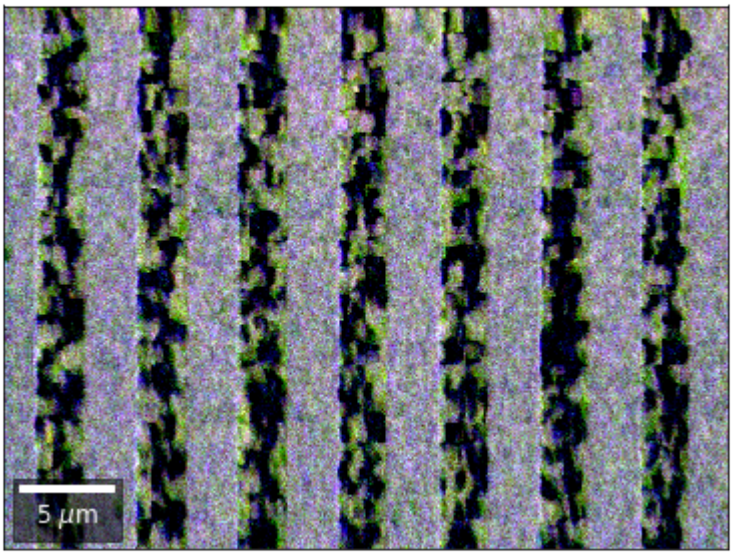


[51]:
```
# normalize to reference sector
```

(continues on next page)

(continued from previous page)

```
with h5py.File(h5ResultFile, 'r') as h5f:
    secfftmag = h5f['fft_sectors']

    ref_signal = secfftmag[:,3]

    signal = secfftmag[:,1] / ref_signal
    red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,2] / ref_signal
    green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    signal = secfftmag[:,0] / ref_signal
    blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

    rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='fft_sectors_rel_RGB', microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.0)
```



Radon Transform Imaging

(. . . without pattern indexing ;-)

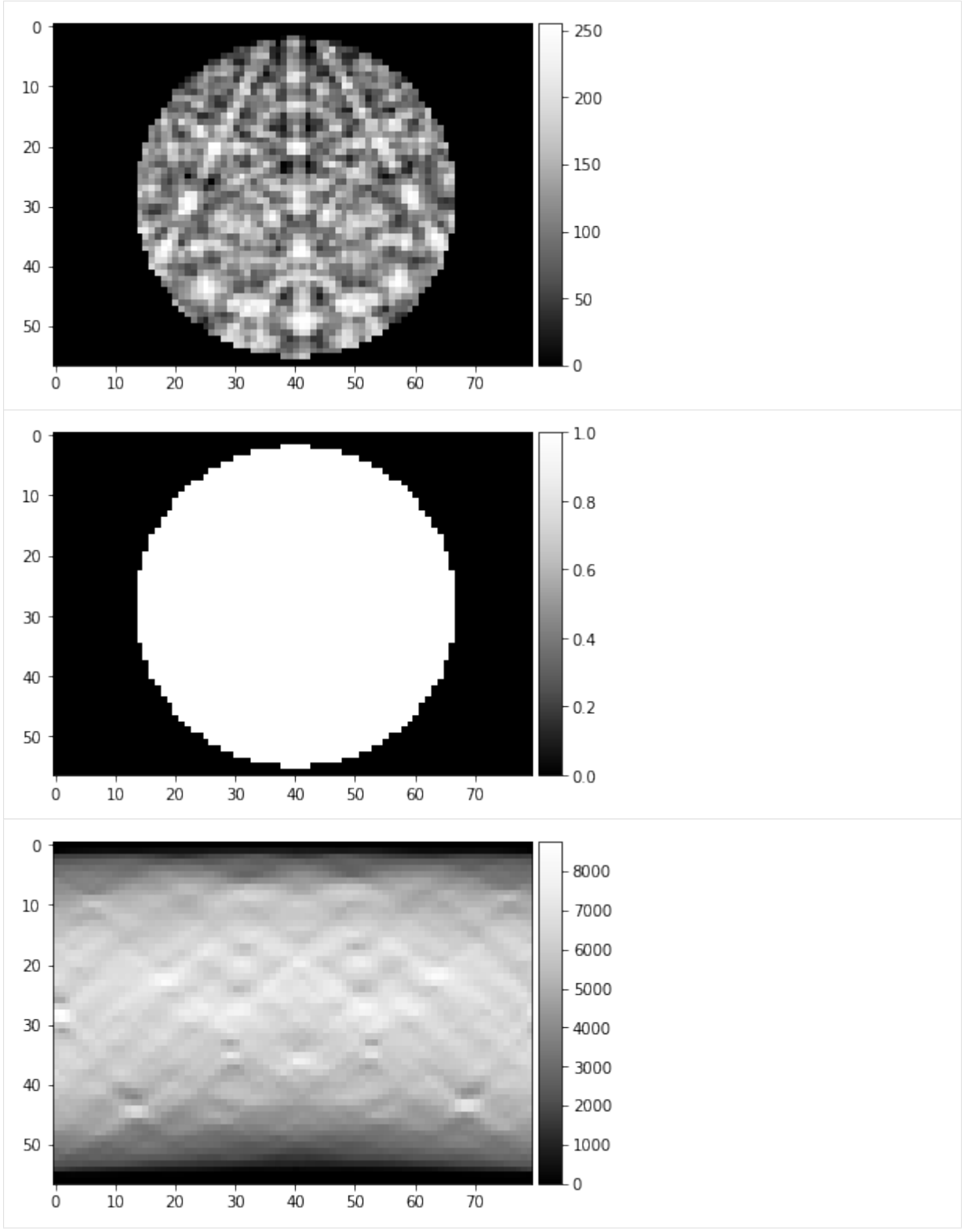
[52]:
```
from skimage.transform import radon
from aloe.image.nxcc import mask_pattern_disk
```

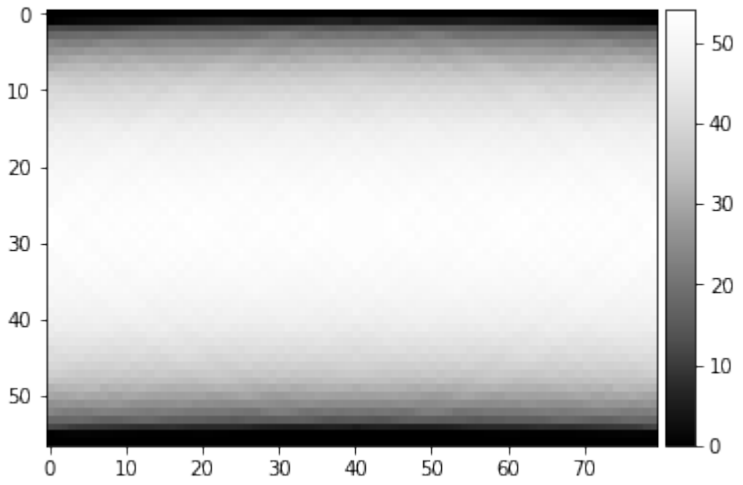
[53]:
```
prebinning_radon = 2 # downsample patterns before Radon (speed)

background_static_radon = downsample(background_static, prebinning_radon) # need␣
↪downsampled background for processing

# angles for Radon (constant for each pattern)
theta = np.linspace(0., 180., max(image.shape) // 2, endpoint=False)

def calc_radon(image, theta, sinogram_reference=None):
    """ calculate radon vbse array """
```

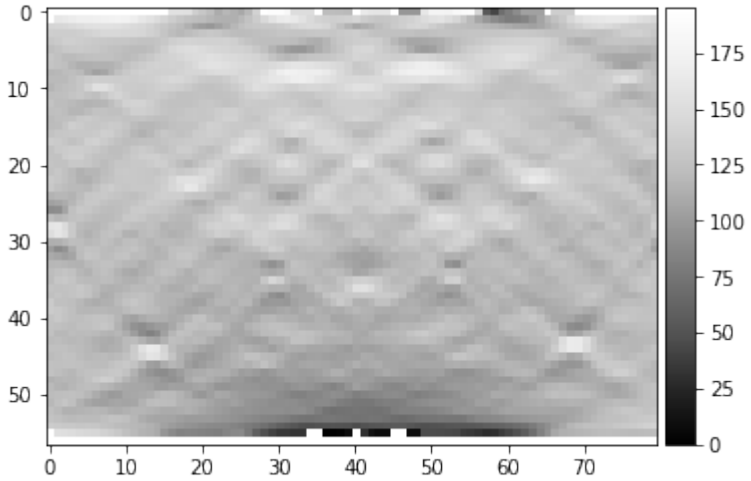
(continues on next page)

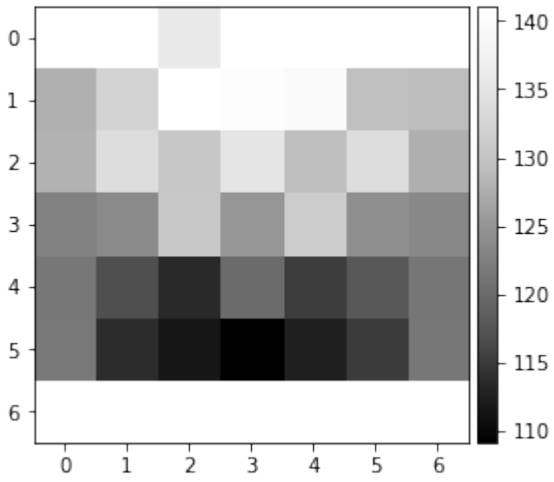
(continued from previous page)

```
    sinogram = radon(image, theta=theta, circle=True)

    if sinogram_reference is not None:
        sino = sinogram / sinogram_reference
    else:
        sino = sinogram

    mean_radon = np.nanmean(sino)
    sino = (sino - mean_radon)**2

    # clip 2 lines
    return sino[2:-2,:]


def process_kikuchi_radon(pattern):
    return process_ebsp(downsample(pattern, prebinning_radon), static_
↪background=background_static_radon)


def process_radon(pattern):
    processed_pattern = process_ebsp(downsample(pattern, prebinning_radon), static_
↪background=background_static_radon)
    kiku, npix = mask_pattern_disk(processed_pattern, rmax=0.333)
    sino = calc_radon(kiku, theta, sinogram_reference=sinogram_uniform)
    return sino


image, npix = mask_pattern_disk(process_kikuchi_radon(Patterns[0]), rmax=0.333)
image_uniform, npix = mask_pattern_disk(np.ones_like(image), rmax=0.333)

img_height, img_width = image.shape

plot_image(image)
plot_image(image_uniform)

sinogram = radon(image, theta=theta, circle=True)
sinogram_uniform = radon(image_uniform, theta=theta, circle=True)
plot_image(sinogram)
plot_image(sinogram_uniform)

sino = sinogram / sinogram_uniform
sino77=arbse.rebin_array(sino)
plot_image(sino)
plot_image(sino77)
```

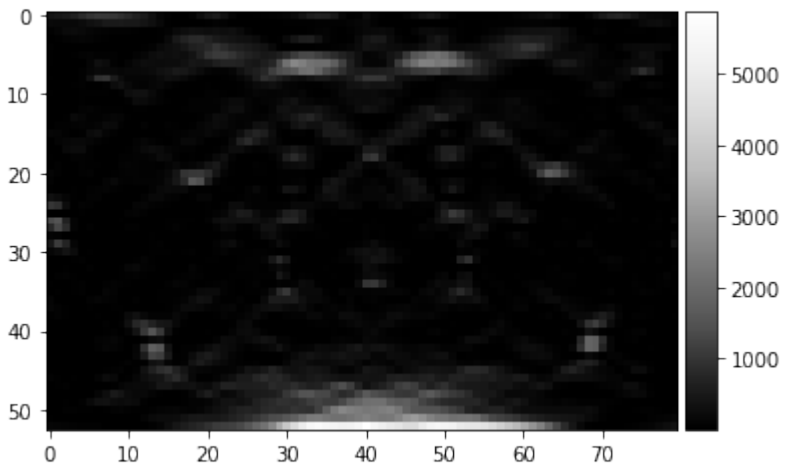

0
10
20
30
40
50
0 10 20 30 40 50 60 70
250
200
150
100
50
0
1.0
0.8
0.6
0.4
0.2
0.0
8000
7000
6000
5000
4000
3000
2000
1000
0







[54]:
```
radon_test = process_radon(Patterns[0])
plot_image(radon_test)
```



Calculate Radon arrays for all patterns:

[55]:
```
vbse_radon =  arbse.make_vbse_array(Patterns, process=process_radon)
```

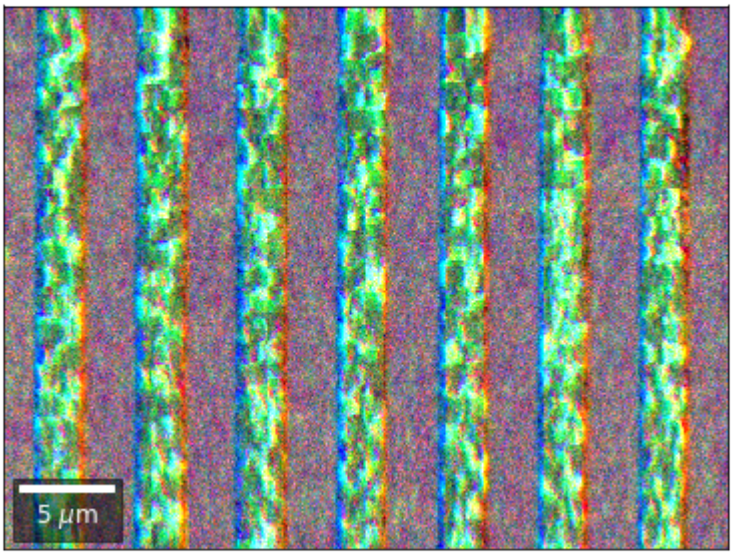
```
total points:120000 current:120000 finished -> total calculation time :   33.9 min
```

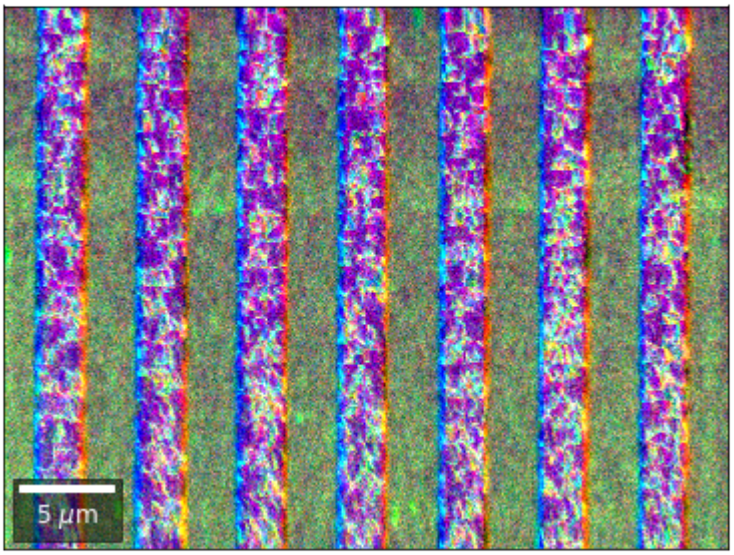
[56]:
```
# save to current hdf5 results file
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vbse_radon', data=vbse_radon)
```

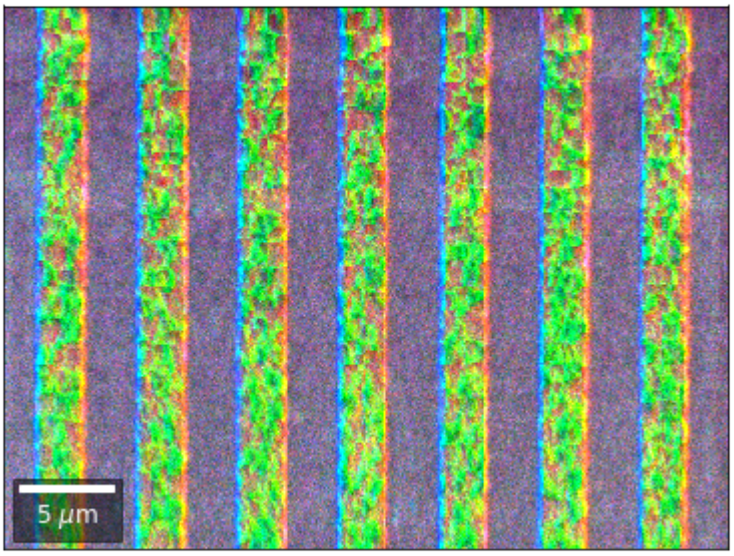
```
arBSE_GaN_Stripes.h5
```

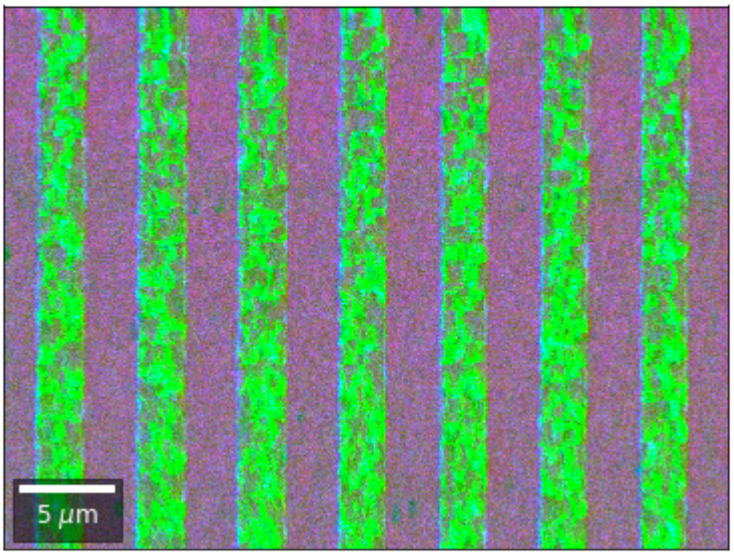
Radon 7x7 array, for each row, RGB color from element in column 1, 4, 7

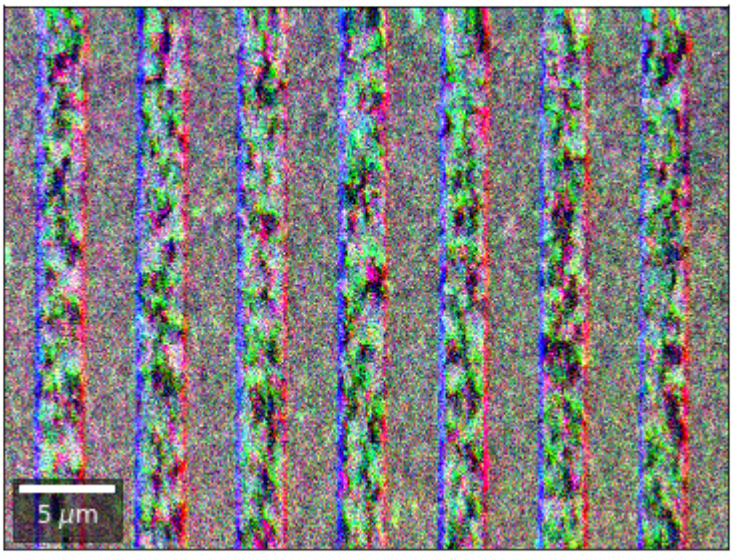
[57]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD = h5f['vbse_radon']
    # rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vRadon_RGB_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=0.8)

        rgb_direct.append(rgb)
```

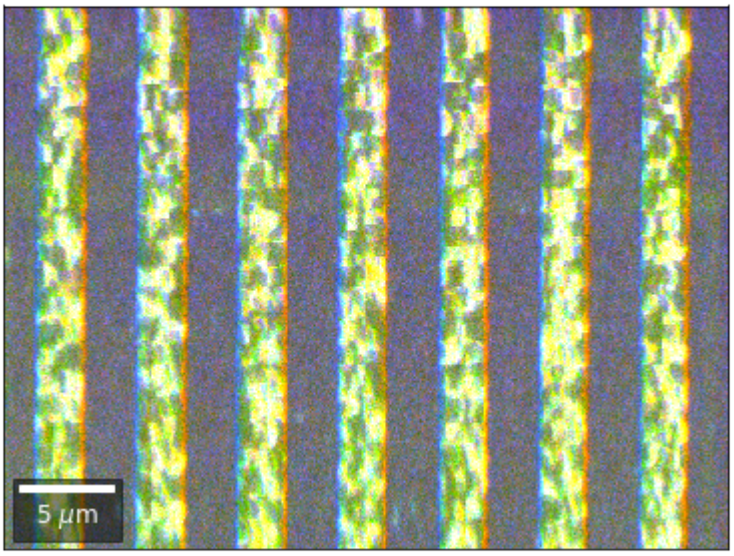

5 µm

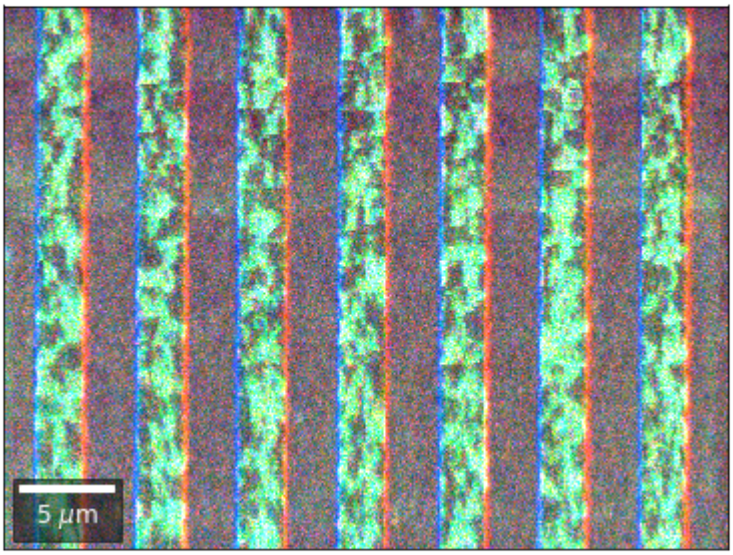

5 µm


5 µm


5 µm

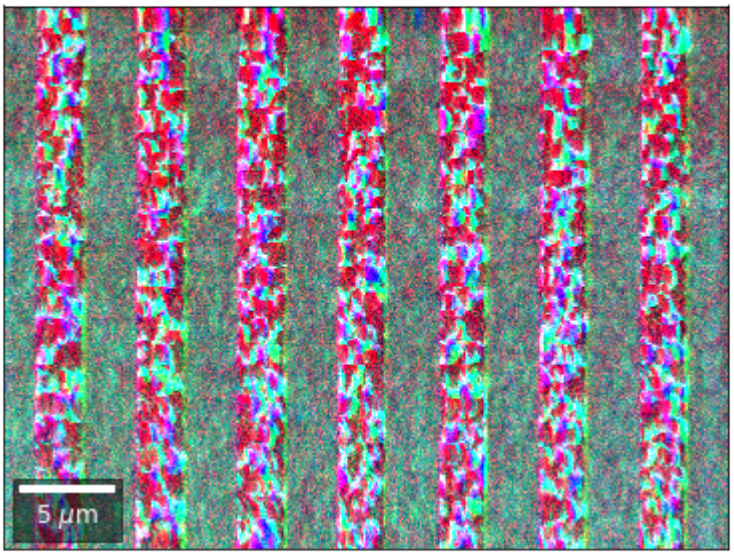

5 µm

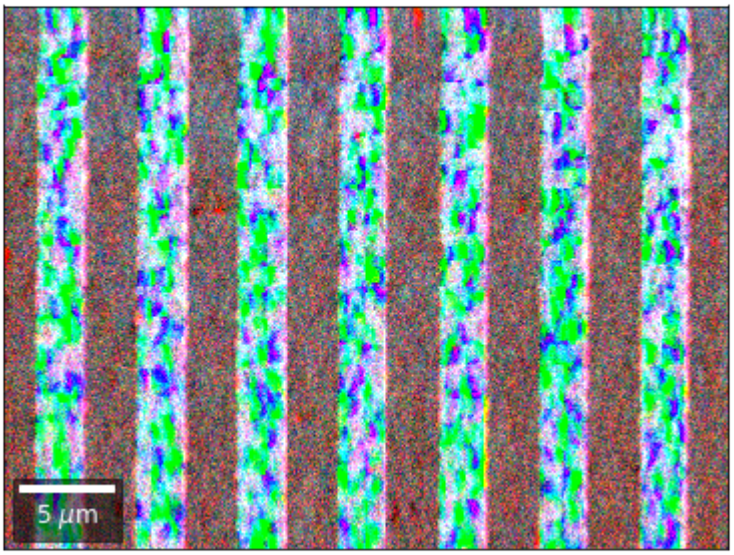

5 µm

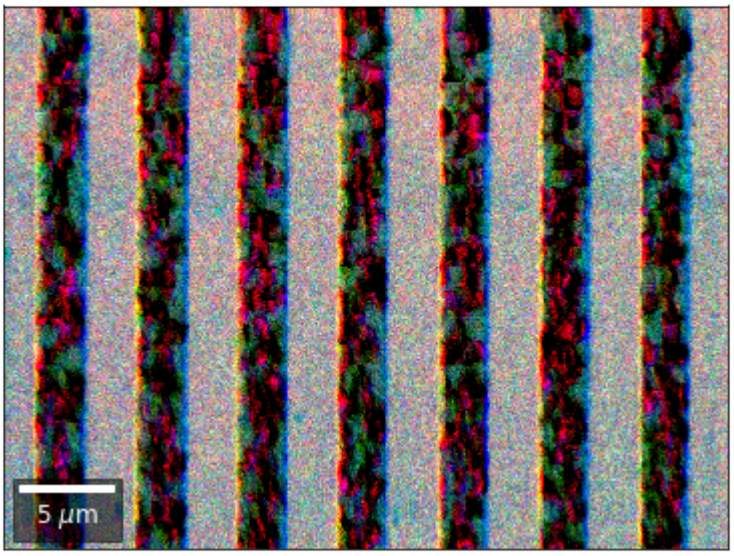


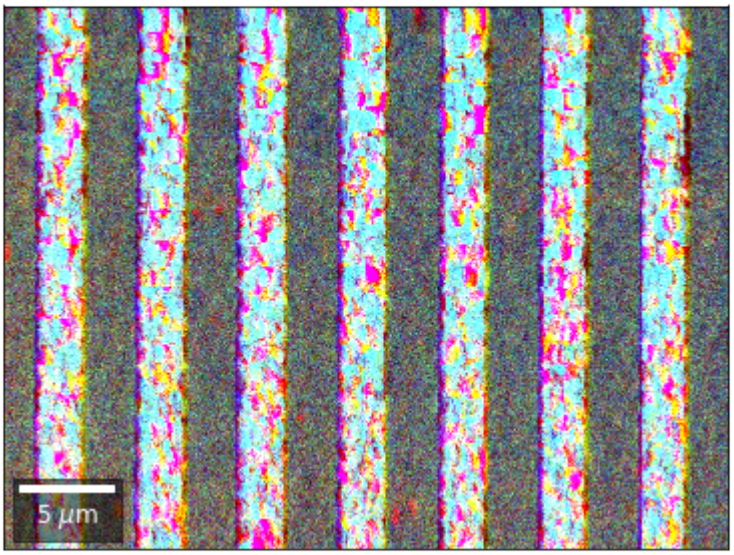
Radon 7x7 array: For each row, RGB color signal from CHANGE of elements in column 1, 4, 7 relative to previous row¶

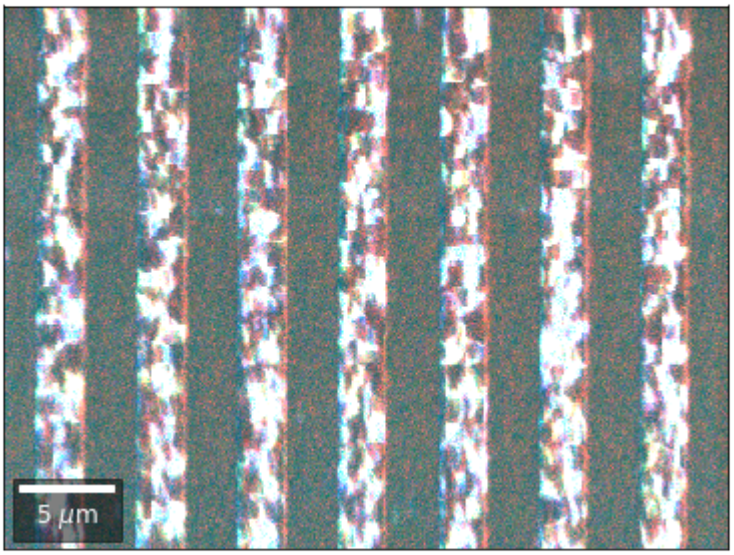
[58]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD = h5f['vbse_radon']
    # relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vRadon_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

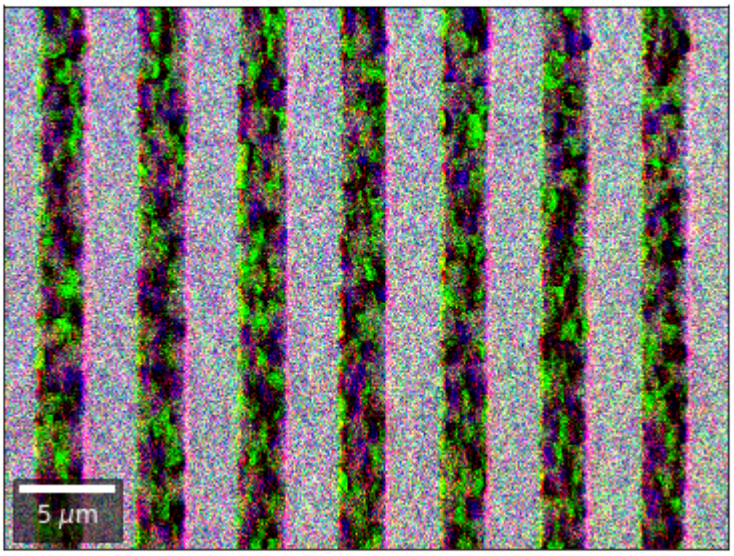

5 µm


5 µm


5 µm


5 µm


5 µm


5 µm

[ ]:

SEM 2D BSE Imaging: GaN Dislocations

Example Data: GaN_Dislocations_1

This is an example notebook to demonstrate angle resolved BSE imaging via signal extraction from raw, saved, EBSD patterns of a GaN thin film sample containing dislocations. As we will see below, the BSE signal from the raw EBSD patterns clearly shows the presence of dislocations in the GaN sample. This is because suitable incident beam channeling conditions have been carefully chosen before the actual measurement, i.e. the dislocation are usually not seen in an arbitrary measurement geometry.

The raw EBSD signal is dominated by the background BSE contribution, which varies according to the changes of the incident beam conditions relative to the strained and rotated crystal structure near the dislocations. For this specific experimental measurement, the analysis of the processed Kikuchi patterns also shows that the expected influences of the dislocations on the outgoing Kikuchi signal are not seen by the 7x7 virtual BSE detector arrangement (i.e. we see basically only noise using the 7x7 array approach).

[1]:
```
# directory with the HDF5 EBSD pattern file:
data_dir = "../../../xcdskd_reference_data/GaN_Dislocations_1/hdf5/"

# filename of the HDF5 file:
hdf5_filename =  "GaN_Dislocations_1.hdf5"

# verbose output
output_verbose = False
```

Necessary packages

[2]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
# ignore divide by Zero
np.seterr(divide='ignore', invalid='ignore')

import time, sys, os
import h5py
import skimage.io

from aloe.plots import normalizeChannel, make2Dmap, get_vrange
from aloe.plots import plot_image, plot_SEM, plot_SEM_RGB
from aloe.image import arbse
from aloe.image.downsample import downsample
from aloe.image.kikufilter import process_ebsp
from aloe.io.progress import print_progress_line
```

[3]:
```
# make result dirs and filenames
h5FileNameFull=os.path.abspath(data_dir + hdf5_filename)
h5FileName, h5FileExt = os.path.splitext(h5FileNameFull)
h5FilePath, h5File = os.path.split(h5FileNameFull)
timestr = time.strftime("%Y%m%d-%H%M%S")
h5ResultFile="arBSE_" + hdf5_filename
```

(continues on next page)

(continued from previous page)

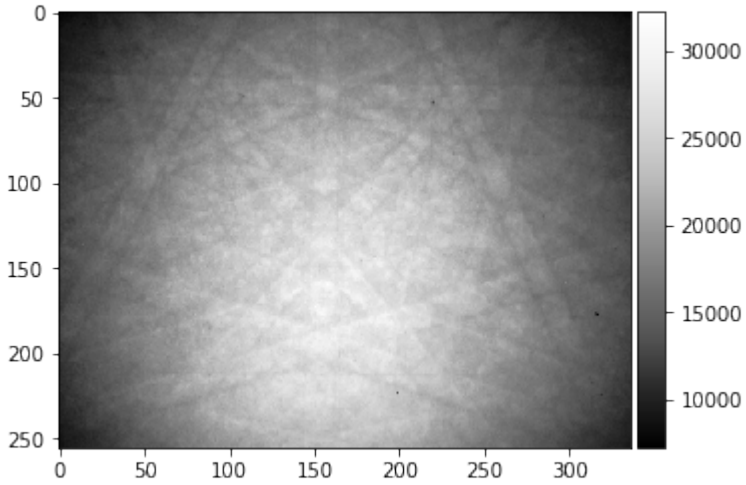
```
# close HDF5 file if still open
if 'f' in locals():
    f.close()
f=h5py.File(h5FileName+h5FileExt, "r")

ResultsDir = h5FilePath+"/arBSE_" + timestr + "/"
CurrentDir = os.getcwd()
#print('Current Directory: '+CurrentDir)
#print('Results Directory: '+ResultsDir)
if not os.path.isdir(ResultsDir):
    os.makedirs(ResultsDir)
os.chdir(ResultsDir)

if output_verbose:
    print('HDF5 full file name: ', h5FileNameFull)
    print('HDF5 File: ', h5FileName+h5FileExt)
    print('HDF5 Path: ', h5FilePath)
    print('Results Directory: ', ResultsDir)
    print('Results File: ', h5ResultFile)
```

[4]:
```
DataGroup="/Scan/EBSD/Data/"
HeaderGroup="/Scan/EBSD/Header/"
Patterns = f[DataGroup+"RawPatterns"]
#StaticBackground=f[DataGroup+"StaticBackground"]
XIndex =   f[DataGroup+"X BEAM"]
YIndex =   f[DataGroup+"Y BEAM"]
MapWidth = f[HeaderGroup+"NCOLS"].value
MapHeight= f[HeaderGroup+"NROWS"].value
PatternHeight=f[HeaderGroup+"PatternHeight"].value
PatternWidth =f[HeaderGroup+"PatternWidth"].value
print('Pattern Height: ', PatternHeight)
print('Pattern Width : ', PatternWidth)
PatternAspect=float(PatternWidth)/float(PatternHeight)
print('Pattern Aspect: '+str(PatternAspect))
print('Map Height: ', MapHeight)
print('Map Width : ', MapWidth)

step_map_microns = f[HeaderGroup+"X Resolution"].value
print('Map Step Size (microns): ', step_map_microns)
```

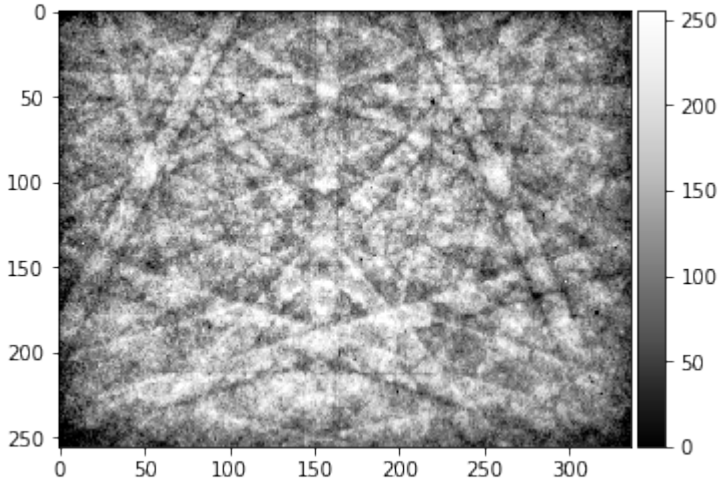
```
Pattern Height:  256
Pattern Width :  336
Pattern Aspect: 1.3125
Map Height:  50
Map Width :  52
Map Step Size (microns):  0.05
```

Check Optional Pattern Processing

[5]:
```
ipattern = 215
raw_pattern=Patterns[ipattern,:,:]
plot_image(raw_pattern)
skimage.io.imsave('pattern_raw_' + str(ipattern) + '.tiff', raw_pattern, plugin='tifffile')
```



[6]:
```
# make processed pattern without static background
processed_pattern = process_ebsp(raw_pattern, binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_' + str(ipattern) + '.tiff', processed_pattern,␣
↪plugin='tifffile')
```



[7]:
```
try:
    # get static background from HDF5, cut off first lines to fit to Patterns
    background_static_file = f[HeaderGroup+"StaticBackground"]
    plot_image(background_static_file, title="Static Background from HDF5 File")
except:
    print('WARNING: No static background found in HDF5 file.')
    background_static_file = None
```

```
WARNING: No static background found in HDF5 file.
```

Static Background from Pattern Average in the Map

We can also approximate a static background from the EBSD map itself; or even use an extra map that was taken explicitly for making a background, e.g. from the sample holder. For polycrystalline samples with a large enough number of grains in random orientations, the Kikuchi patterns from all grains will average out when taking the average of all pattern. For single crystalline samples, or samples with a low number of different orientation present in the map area measured, the average of all patterns will stay contain Kikuchi diffraction features. These features in the static background will tend to produce artififacts when the raw data is processed using the background with diffraction

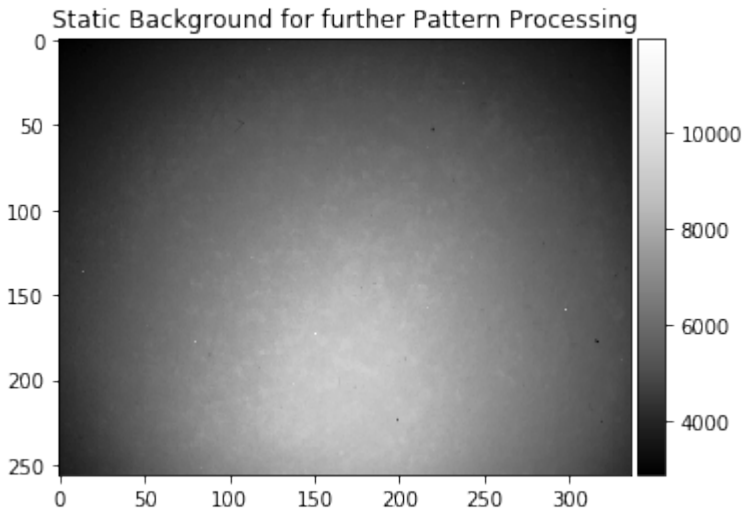
features.

[8]:
```
# load background from different experiment
#background_static_txt = np.loadtxt(data_dir + "background_static.txt")
#plot_image(background_static_txt, title="background_static.txt")
```

[9]:
```
# assign static background
background_static_image = skimage.io.imread("../StaticBackground.tiff") #, plugin='tifffile
↪')
```

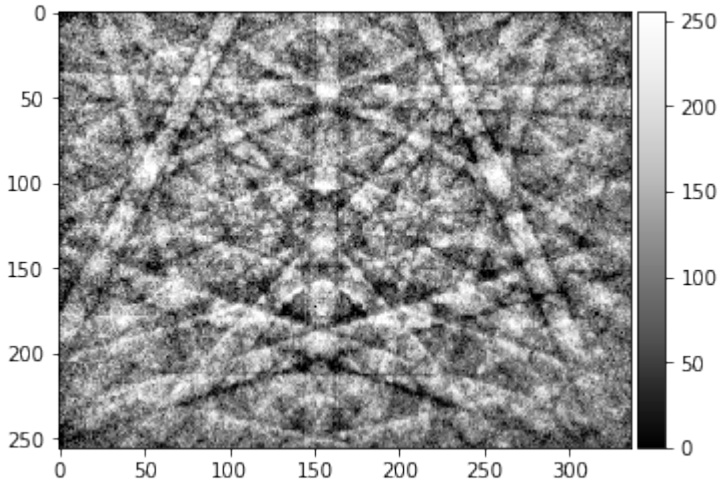
[10]:
```
#background_static = background_static_file
#background_static = background_static_txt
background_static = background_static_image
print(background_static.shape)
print(background_static)
plot_image(background_static, title="Static Background for further Pattern Processing")
```

```
(256, 336)
[[2884 2920 2920 ..., 2994 2995 2992]
 [2910 2944 2954 ..., 3005 3009 2992]
 [2931 2955 2954 ..., 3050 3024 3036]
 ...,
 [3725 3774 3783 ..., 4260 4243 4211]
 [3714 3761 3722 ..., 4297 4266 4221]
 [3704 3707 3737 ..., 4279 4220 4204]]
```



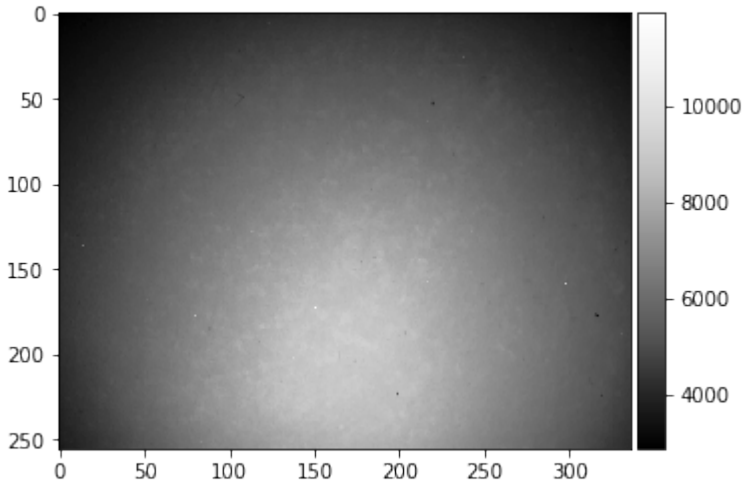
[11]:
```
#skimage.io.imsave('background_static.tiff', background_static, plugin='tifffile') # this␣
↪will be 16bit only
np.savetxt('background_static.txt', background_static)
```

[12]:
```
# note the CCD halves and static background dust, hot pixels
processed_pattern = process_ebsp(raw_pattern, static_background=background_static,␣
↪binning=1)
plot_image(processed_pattern)
skimage.io.imsave('pattern_processed_static_' + str(ipattern) + '.tiff', processed_pattern,
↪ plugin='tifffile')
```



Specification of Image Pre-Processing Functions

[13]:
```
prebinning=1
background_static_binned = downsample(background_static, prebinning)

def pipeline_process(pattern, prebinning=1, kikuchi=False):
    if prebinning>1:
        pattern = downsample(pattern, prebinning)
    if kikuchi:
        return process_ebsp(pattern, static_background=background_static_binned, binning=1)
    else:
        return pattern

def process_kikuchi(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=True)

def process_bin(pattern):
    return pipeline_process(pattern, prebinning=prebinning, kikuchi=False)

plot_image(background_static_binned)
print(background_static_binned.shape)
```

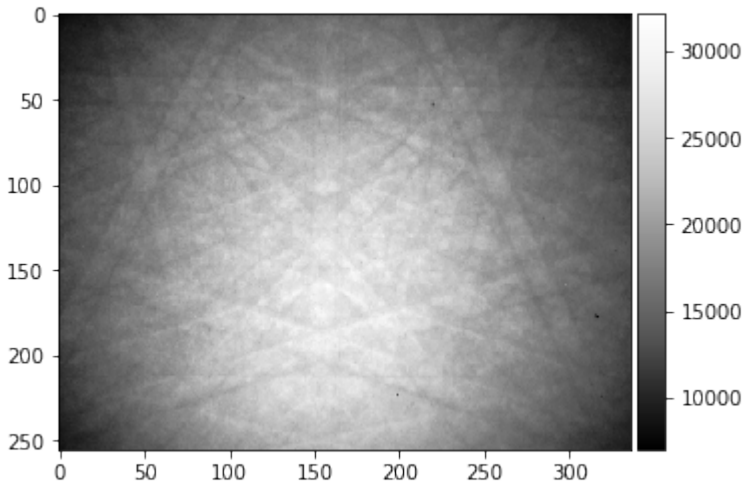


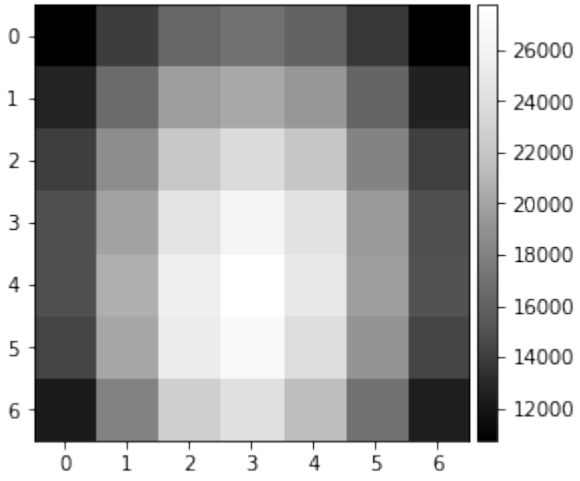

```
(256, 336)
```

vBSE Array

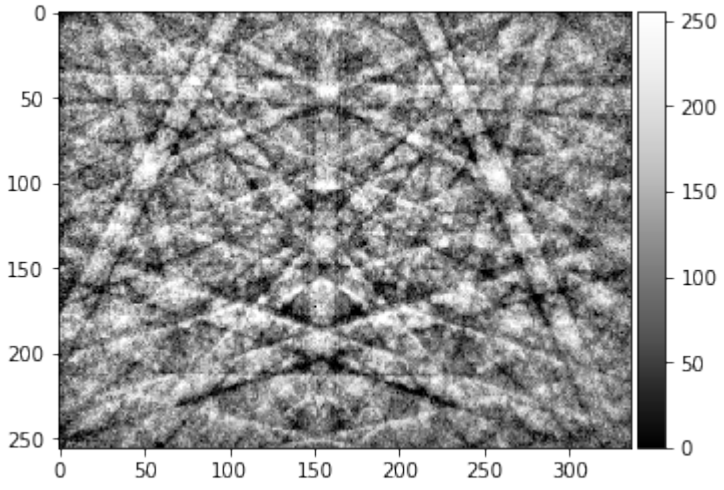
We convert the raw pattern into a 7x7 array of vBSE sensor intensities.

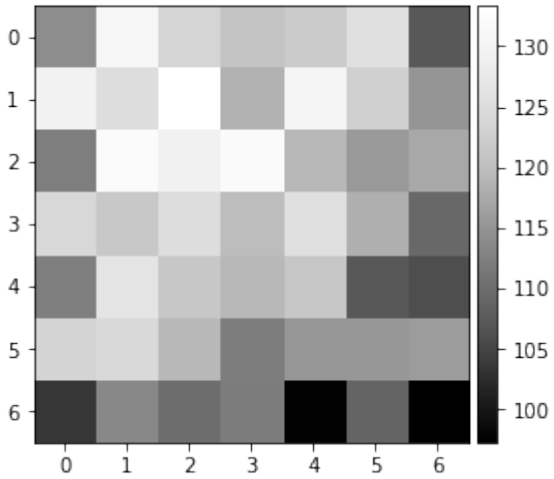
[14]:
```
pattern = pipeline_process(Patterns[1000], kikuchi=False)
vbse = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vbse)
```





[15]:
```
pattern = process_kikuchi(Patterns[1000])
vkiku = arbse.rebin_array(pattern)
plot_image(pattern)
plot_image(vkiku)
```





vBSE Detector Signals: Calculation & Saving

This should take a few minutes, depending on your computer and file access speed.

Virtual BSE Imaging

Imaging the raw intensity in the respective area of the 2D detector (e.g. phosphor screen). Neglects gnomonic projection effect on intensities.

[16]:
```
# calculate the vBSE signals in 7x7 array
vbse_array = arbse.make_vbse_array(Patterns)

# make vBSE map of the total screen intensity
bse_total = np.sum(np.sum(vbse_array[:,:,:], axis=1), axis=1)
bse_map = make2Dmap(bse_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points: 2600 current: 2600 finished -> total calculation time :    0.1 min
```

[17]:
```
# save the results in the h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
```

(continues on next page)

(continued from previous page)

```
    h5f.create_dataset('vbse', data=vbse_array)
    h5f.create_dataset('/maps/bse_total', data=bse_map)
```

```
arBSE_GaN_Dislocations_1.hdf5
```

Virtual Orientation Imaging via Kikuchi Pattern Signals

If we process the raw images to obtain only the Kikuchi pattern, we have a modified 2D intensity which can be expected to show increased sensitivity to orientation effects (i.e. changes related to the Kikuchi bands). In a more advanced approach, we could select, for example, specific Kikuchi bands or zone axes to extract imaging signals.

[18]:
```
# calculate the vKikuchi signals from processed raw data
vkiku_array = arbse.make_vbse_array(Patterns, process=process_kikuchi)

# make vBSE map of the total screen intensity
kiku_total = np.sum(np.sum(vkiku_array[:,:,:], axis=1), axis=1)
kiku_map = make2Dmap(kiku_total, XIndex, YIndex, MapHeight, MapWidth)
```

```
total points: 2600 current: 2600 finished -> total calculation time :    0.8 min
```

[19]:
```
# save the results in an extra hdf5
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('vkiku', data=vkiku_array)
    h5f.create_dataset('/maps/kiku_total', data=kiku_map)
```

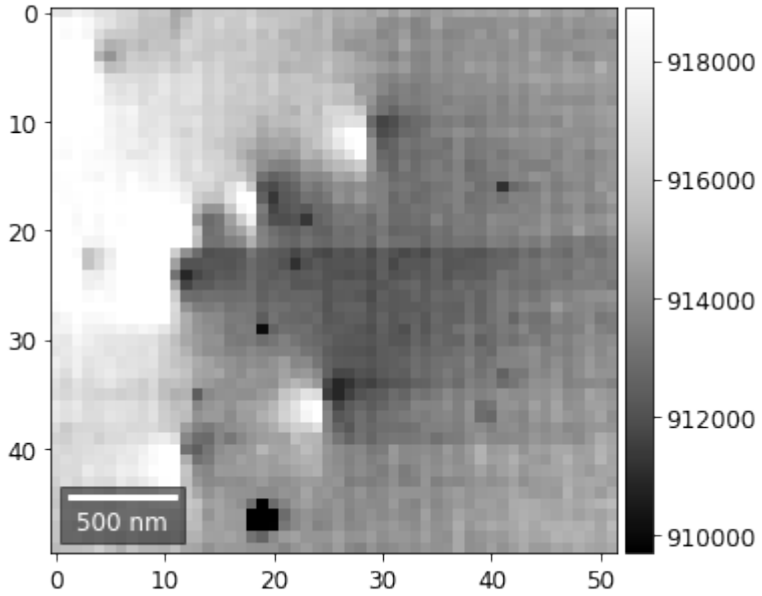
```
arBSE_GaN_Dislocations_1.hdf5
```

vBSE Signals: Plotting

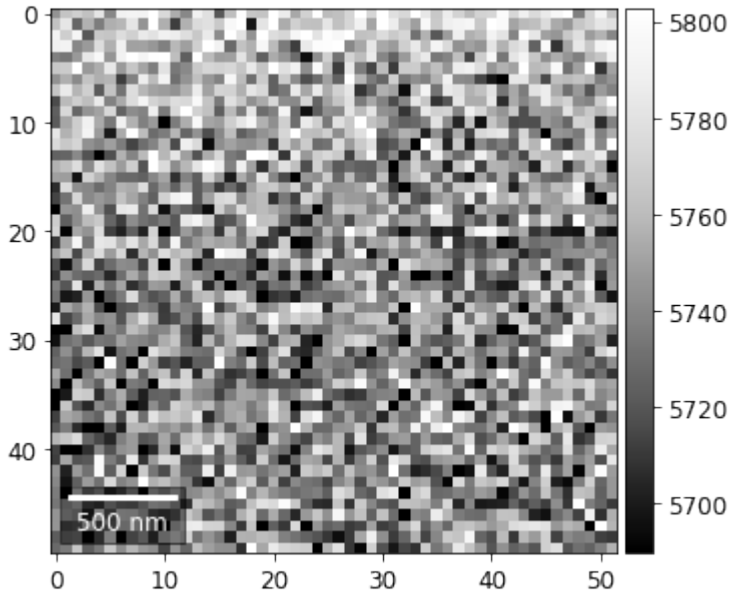
Total Signal on Screen

Total sum of the 7x7 arrays, for the raw pattern and the Kikuchi pattern at each map point:

[20]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    bse = h5f['/maps/bse_total']
    plot_SEM(bse, cmap='Greys_r', microns=step_map_microns)
```



[21]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    kiku = h5f['/maps/kiku_total']
    plot_SEM(kiku, cmap='Greys_r', microns=step_map_microns)
```



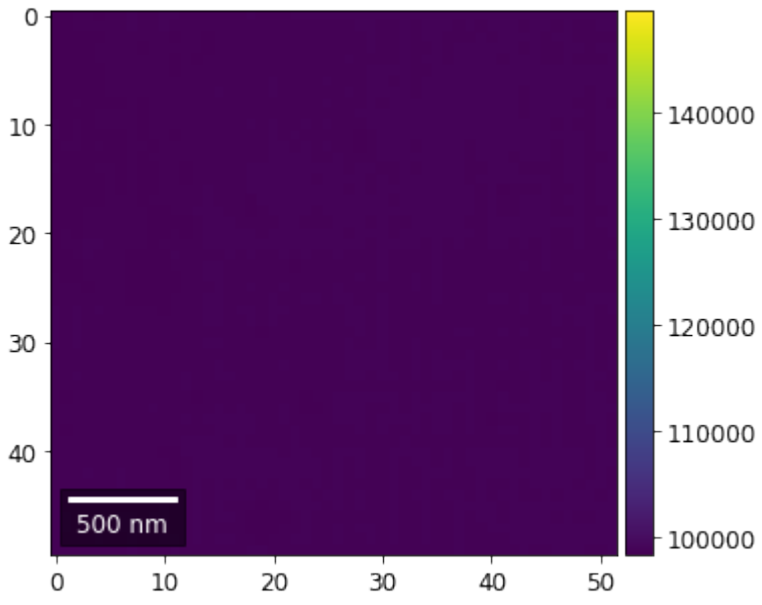
Intensity in Rows and Columns of the vBSE array

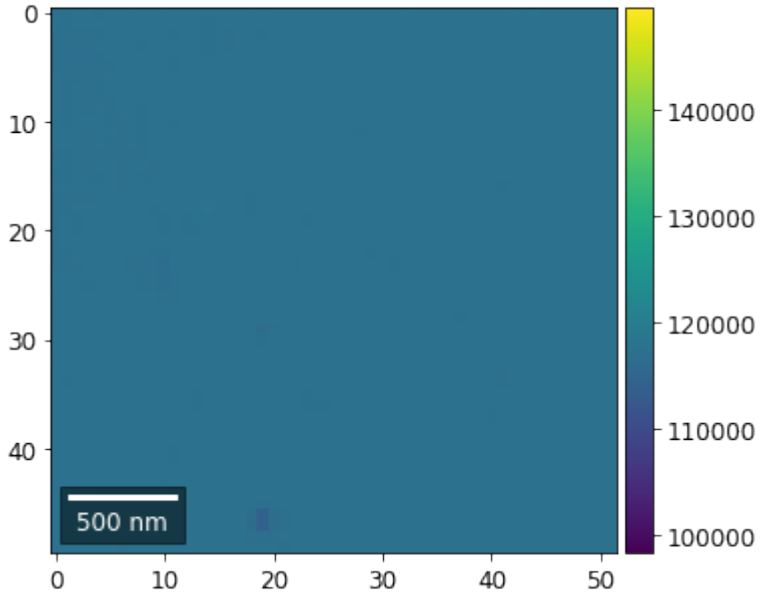
We can calculate additional images from the vBSE data set of 7x7 ROIs derived from the original patterns. As a first example, we plot the intensities of each of the 7 rows and then of each of the 7 columns:

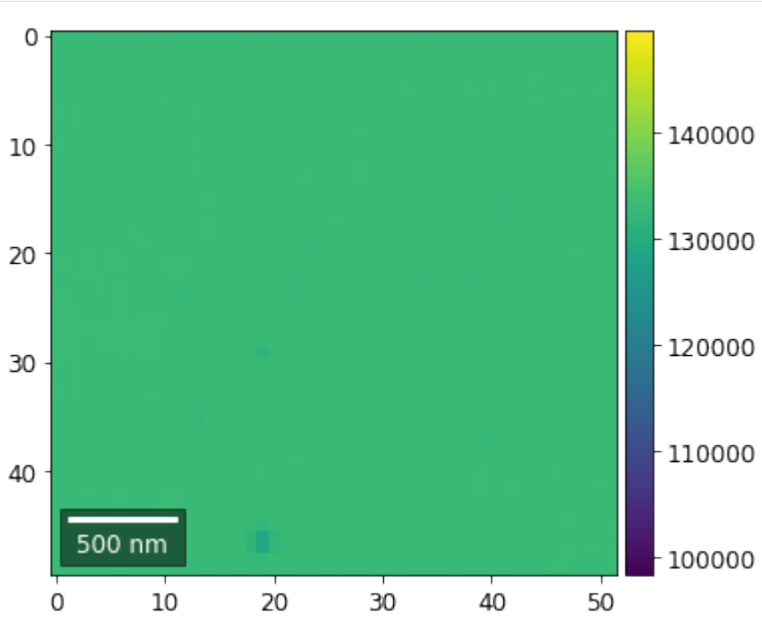
Rows

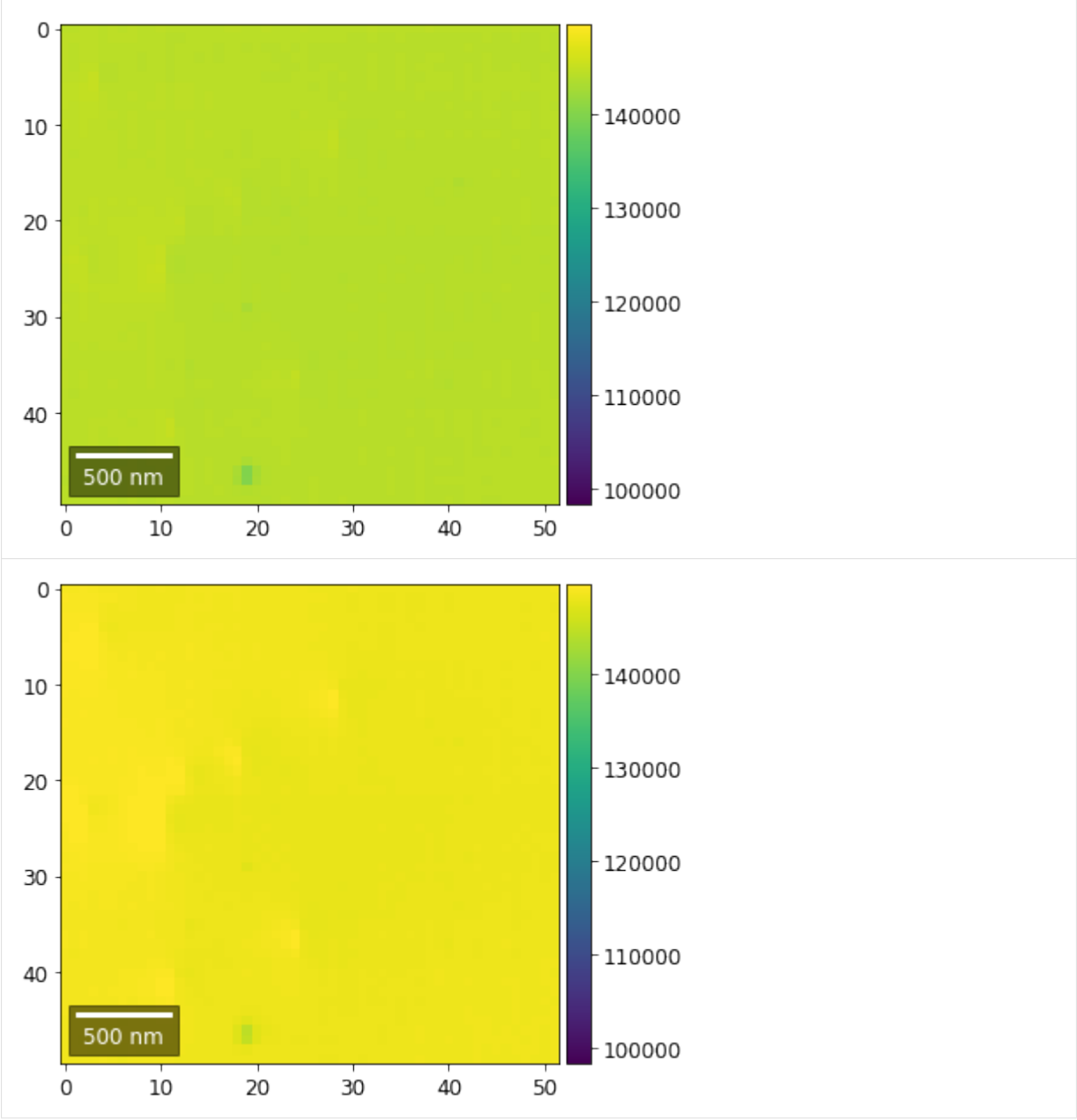
[22]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']

    # signal: sum of row
    vmin=40000000
    vmax=0

    bse_rows = []

    # (1) get full range for all images
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        minv, maxv = get_vrange(signal, stretch=3.0)
        if (minv<vmin):
            vmin=minv
        if (maxv>vmax):
            vmax=maxv

    # (2) make plots with same range for comparisons of absolute BSE values
    vrange=[vmin, vmax]
    print('Range of Values: ', vrange)
    #vrange=None
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_rows.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_absolute_'+str(row),
                 rot180=True, microns=step_map_microns)
```

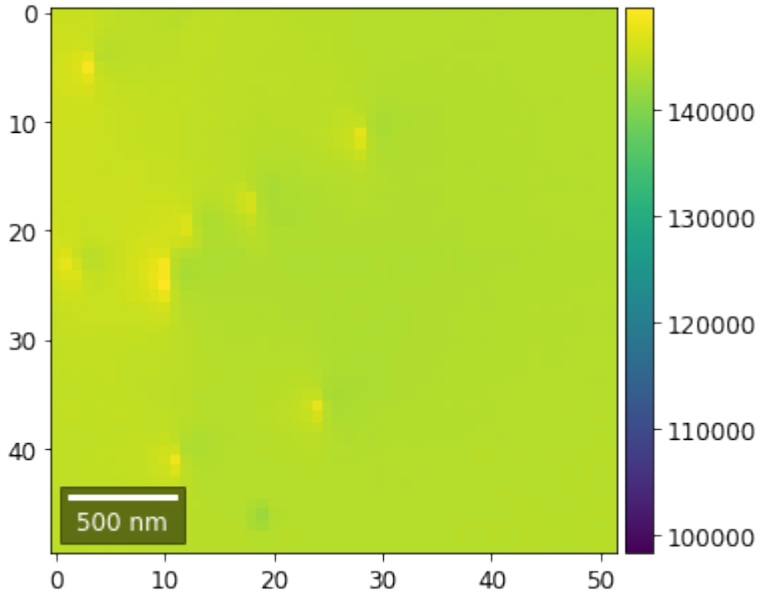
```
Range of Values:  [98411.729767687546, 149717.25413584543]
```

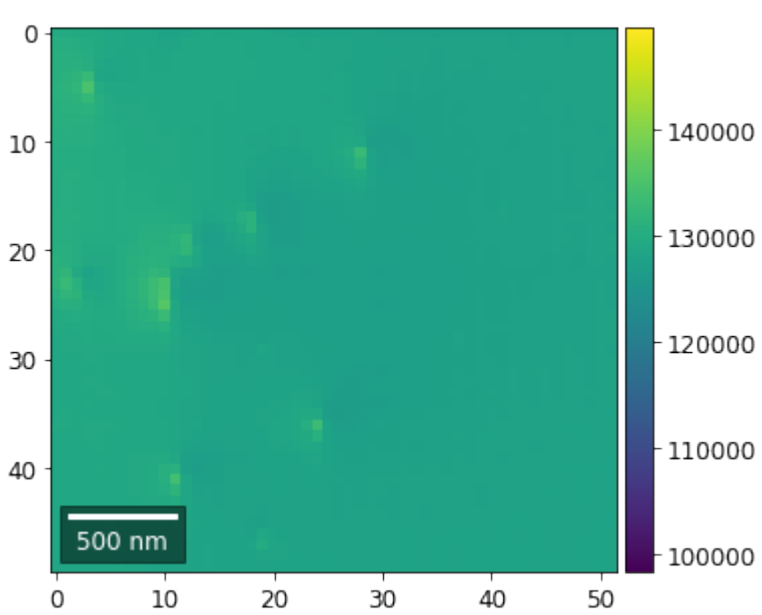



0
10
20
30
40
500 nm
0
10
20
30
40
50
140000
130000
120000
110000
100000

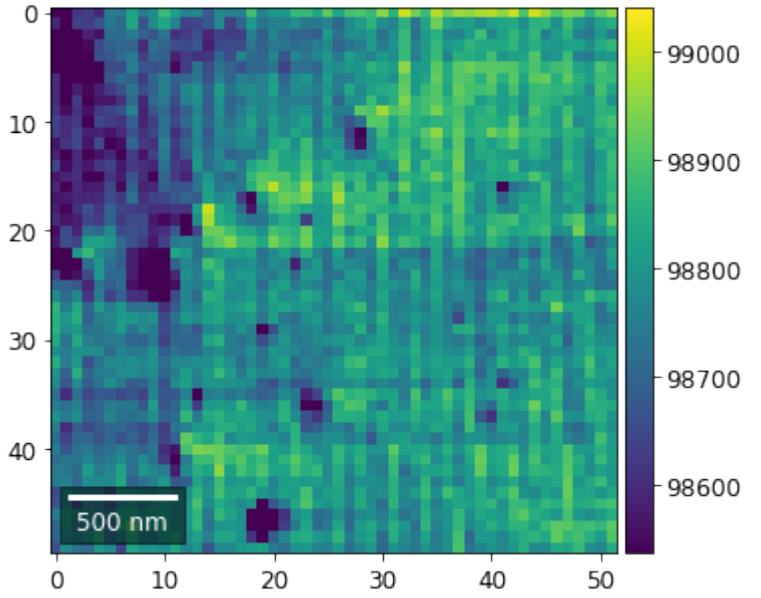

0
10
20
30
40
500 nm
0
10
20
30
40
50
140000
130000
120000
110000
100000

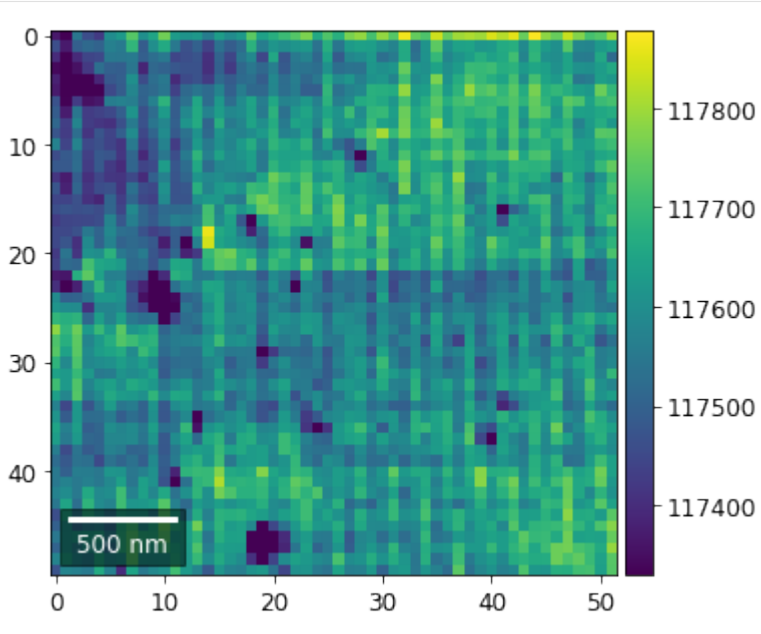

0
10
20
30
40
500 nm
0
10
20
30
40
50
140000
130000
120000
110000
100000
0
10
20
30
40
500 nm
0
10
20
30
40
50
140000
130000
120000
110000
100000

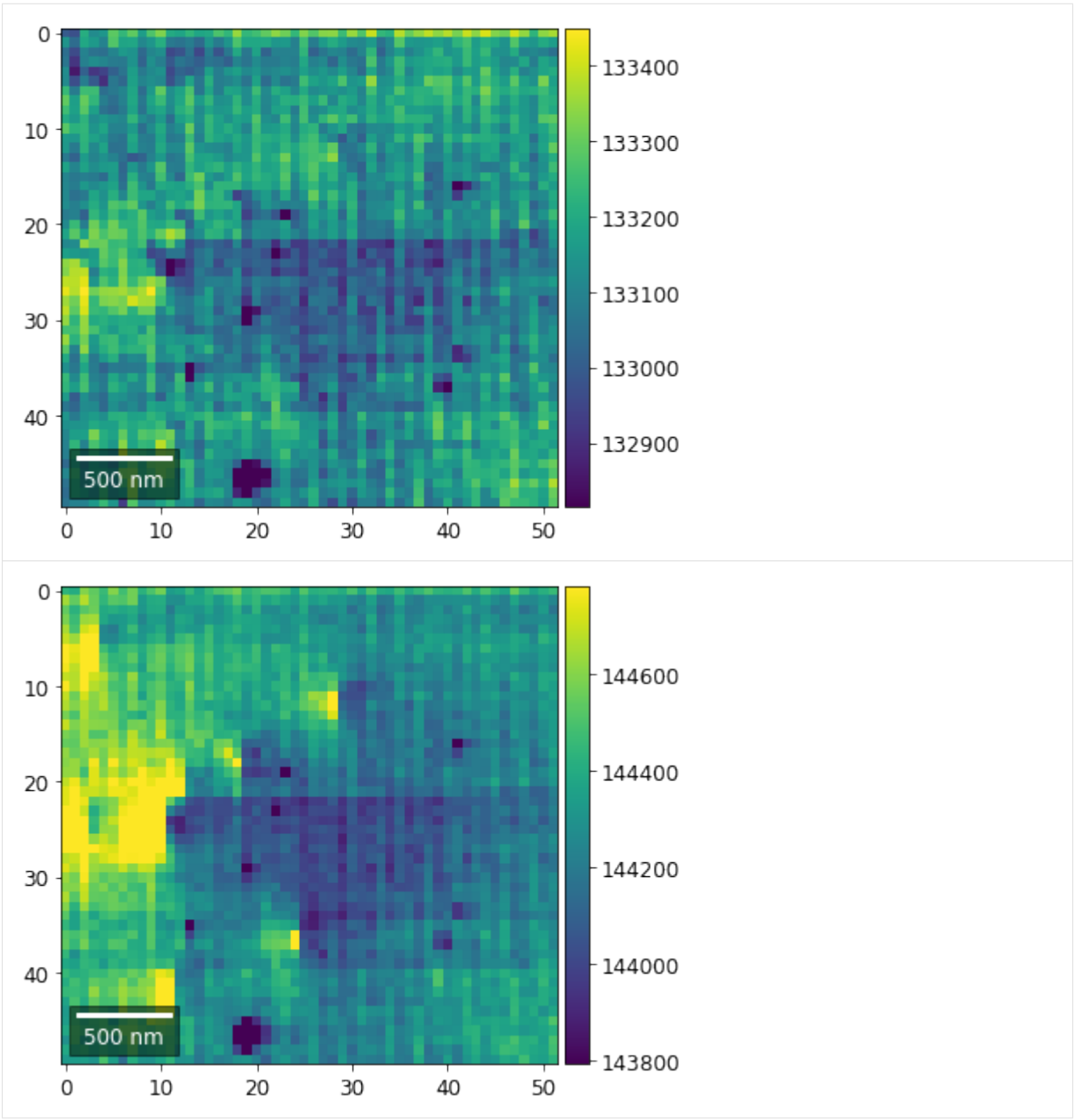


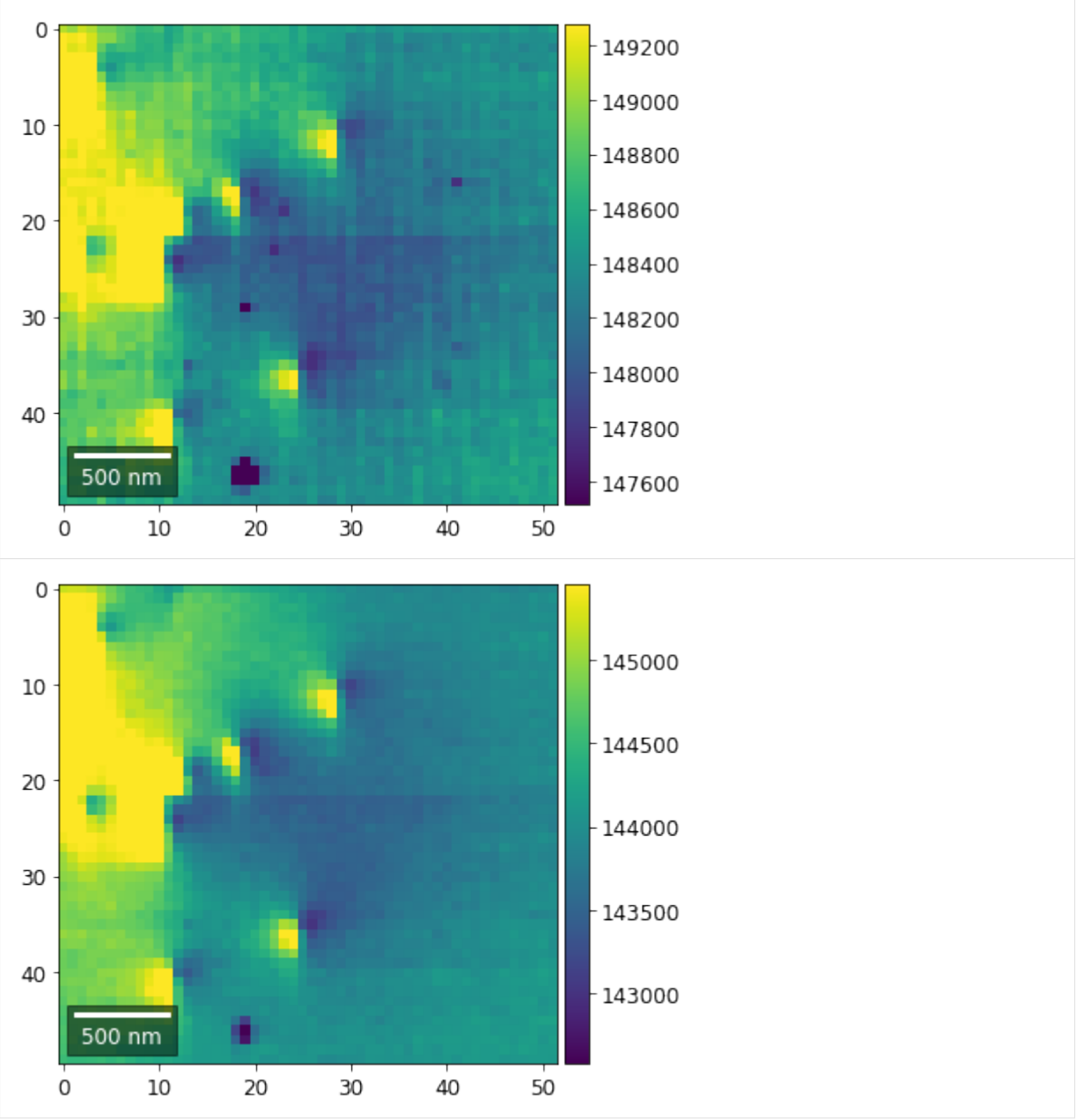


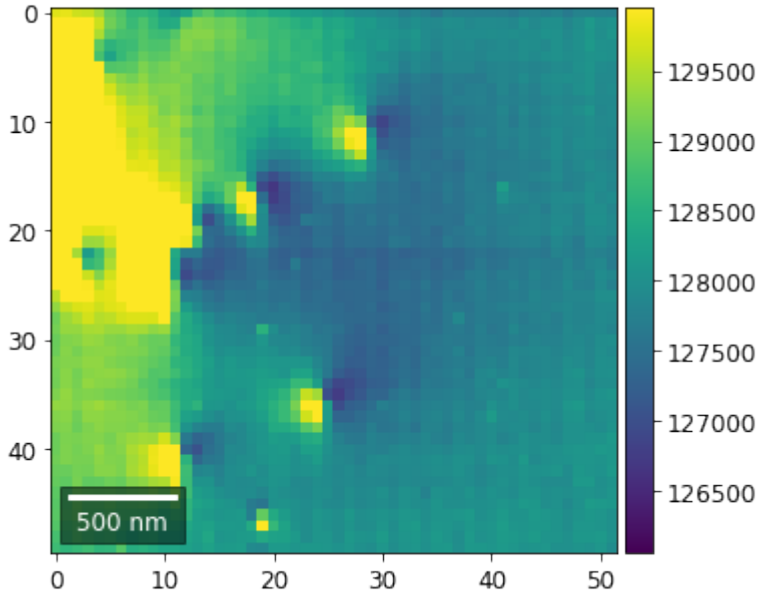
[23]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']
    # (3) make plots with  individual ranges for better contrast
    vrange=None
    for row in range(7):
        signal = np.sum(vFSD[:,row,:], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_rows.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_row_individual_'+str(row),
                 rot180=True, microns=step_map_microns)
```


0
10
20
30
40
500 nm
0
10
20
30
40
50
99000
98900
98800
98700
98600


0
10
20
30
40
500 nm
0
10
20
30
40
50
117800
117700
117600
117500
117400

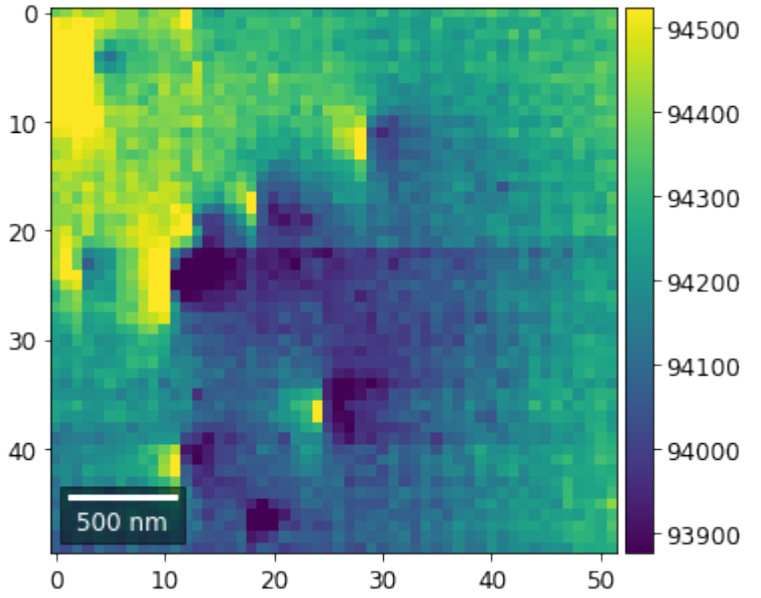

0
10
20
30
40
500 nm
0
10
20
30
40
50
133400
133300
133200
133100
133000
132900
0
10
20
30
40
500 nm
0
10
20
30
40
50
144600
144400
144200
144000
143800

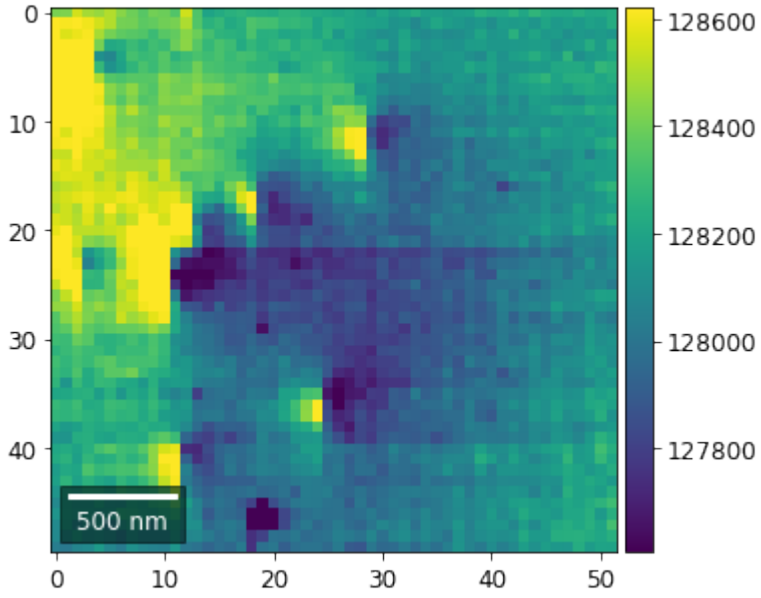

0
10
20
30
40
500 nm
0
10
20
30
40
50
149200
149000
148800
148600
148400
148200
148000
147800
147600
0
10
20
30
40
500 nm
0
10
20
30
40
50
145000
144500
144000
143500
143000

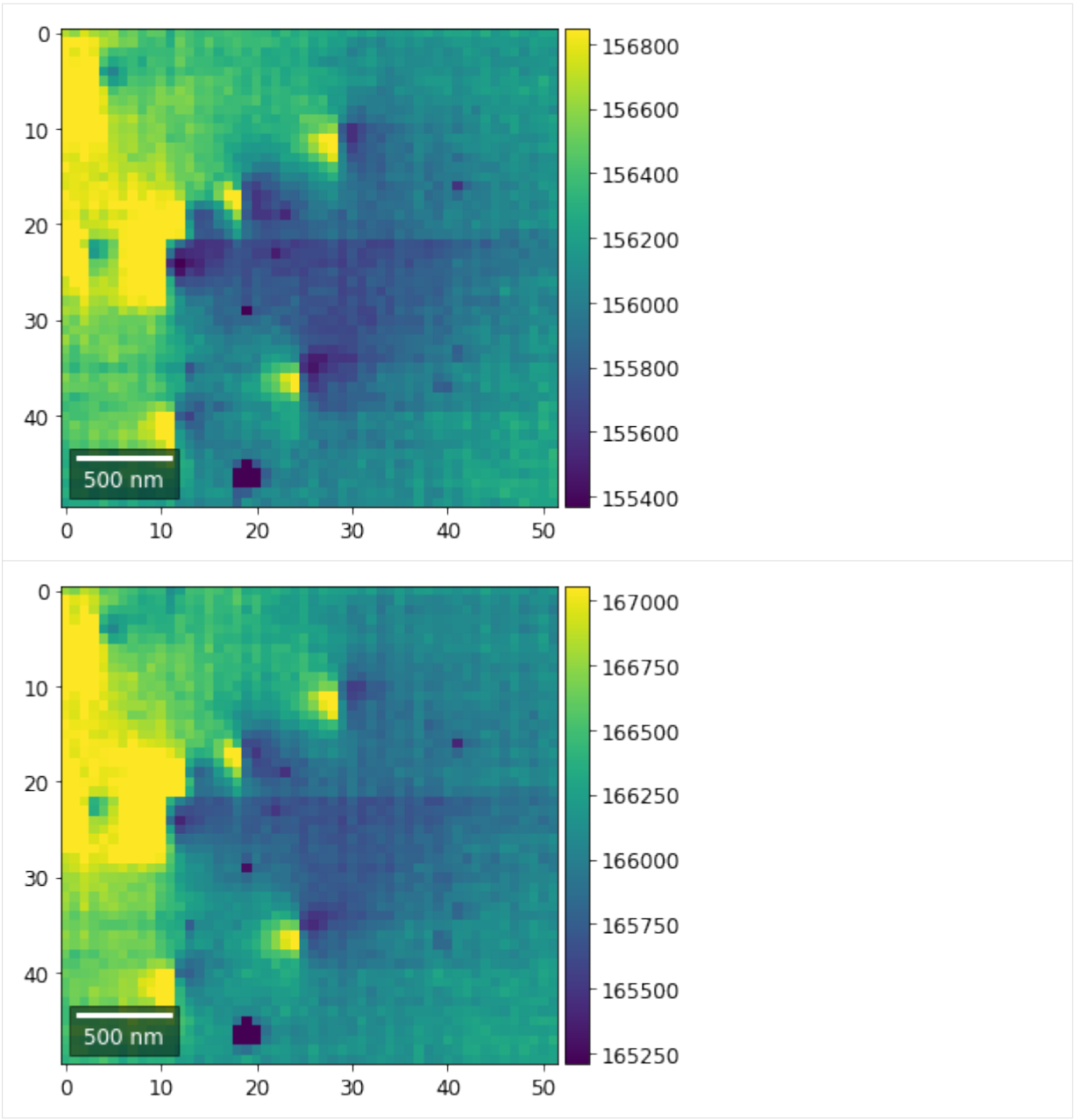


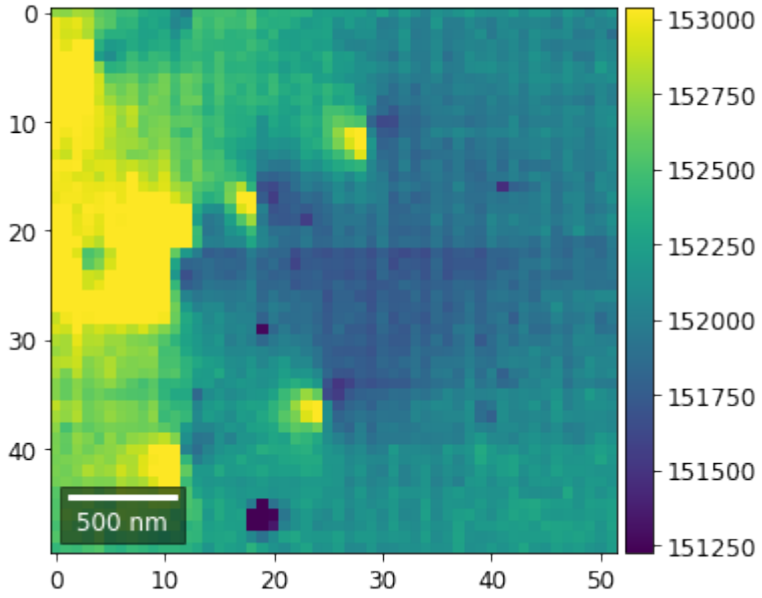
Columns

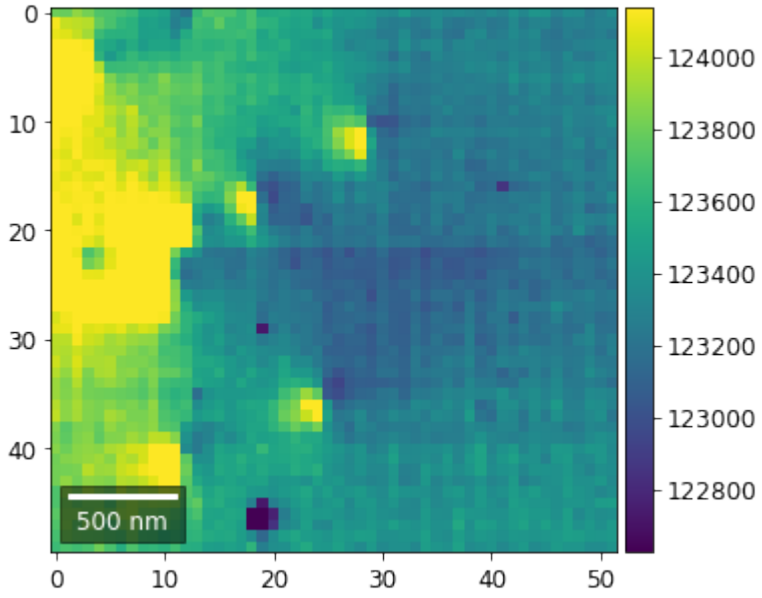
[24]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']
    # signal: sum of column
    vmin=400000
    vmax=0

    bse_cols = []

    # (1) get full range for all images
    for col in range(7):
        signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        minv, maxv = get_vrange(signal)
        if (minv<vmin):
            vmin=minv
        if (maxv>vmax):
            vmax=maxv

    # (2) make plots with same range for comparisons of absolute BSE values
    #vrange=[vmin, vmax]
    vrange=None # no fixed scale
    for col in range(7):
        signal = np.sum(vFSD[:,:,col], axis=1) #/vFSD[:,row+drow,0]
        signal_map = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)
        bse_cols.append(signal_map)
        plot_SEM(signal_map, vrange=vrange, filename='vFSD_col_'+str(col),
                 rot180=True, microns=step_map_microns)
```

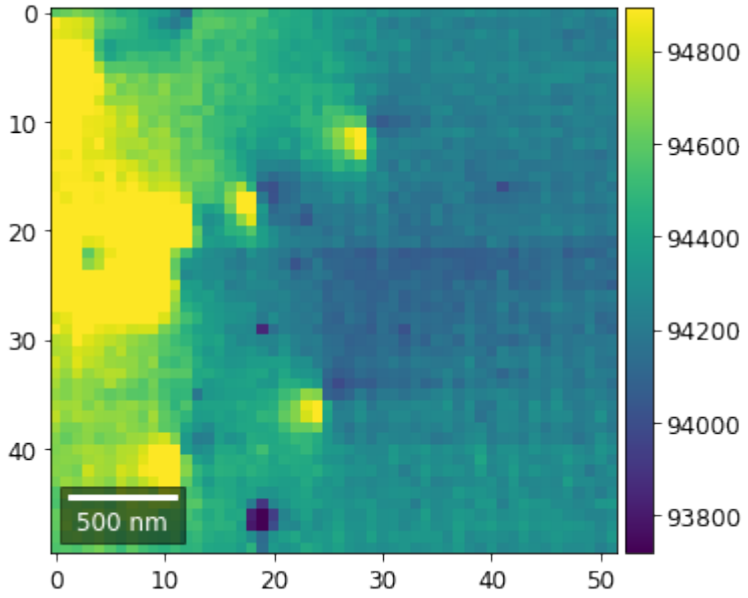

0
10
20
30
40
500 nm
0
10
20
30
40
50
94500
94400
94300
94200
94100
94000
93900


0
10
20
30
40
500 nm
0
10
20
30
40
50
128600
128400
128200
128000
127800


0
10
20
30
40
500 nm
0
10
20
30
40
50
156800
156600
156400
156200
156000
155800
155600
155400
0
10
20
30
40
500 nm
0
10
20
30
40
50
167000
166750
166500
166250
166000
165750
165500
165250


0
10
20
30
40
500 nm
0
10
20
30
40
50
153000
152750
152500
152250
152000
151750
151500
151250

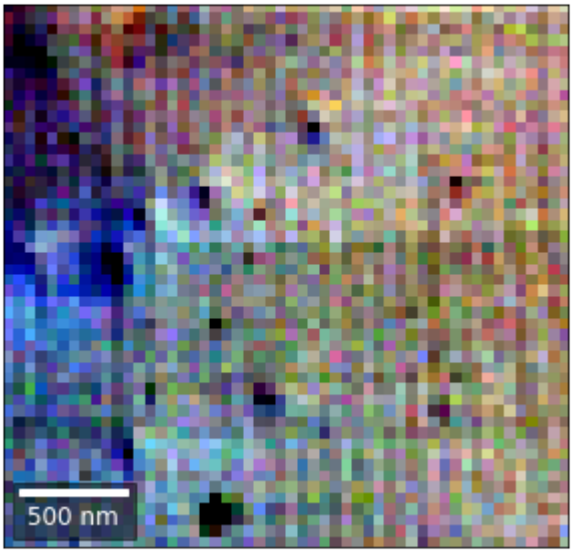

0
10
20
30
40
500 nm
0
10
20
30
40
50
124000
123800
123600
123400
123200
123000
122800

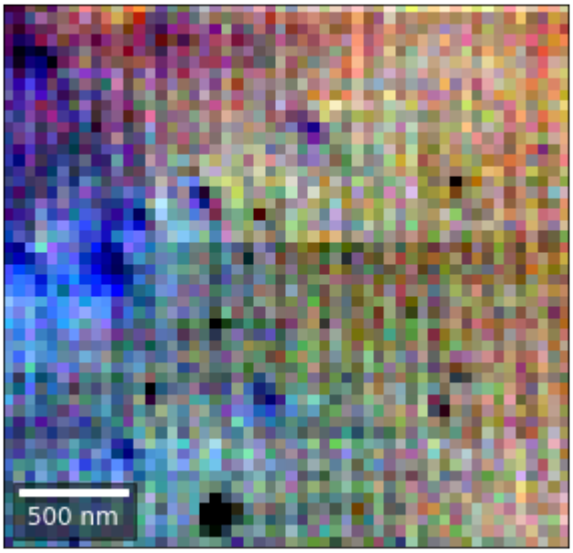


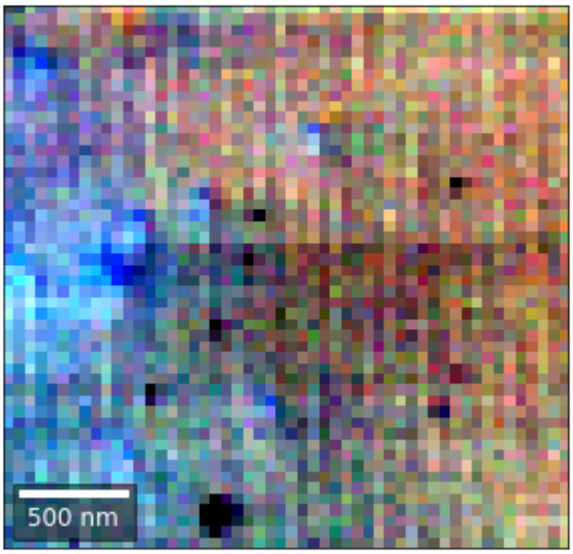
vBSE Color Imaging

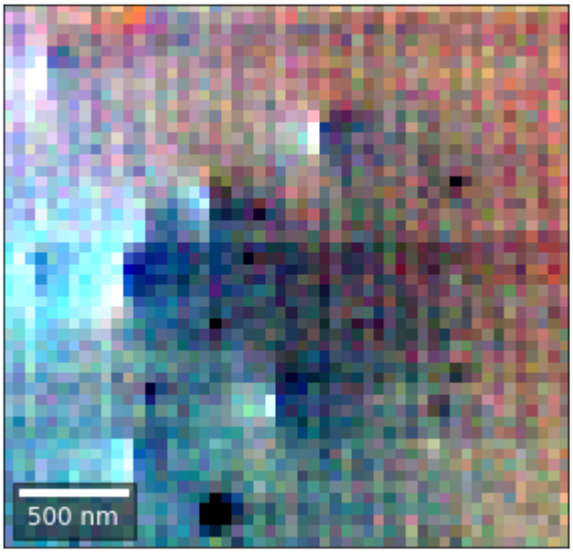
We can also form color images by assigning red, green, and blue channels to the left, middle, and right vBSE sensors of a row:

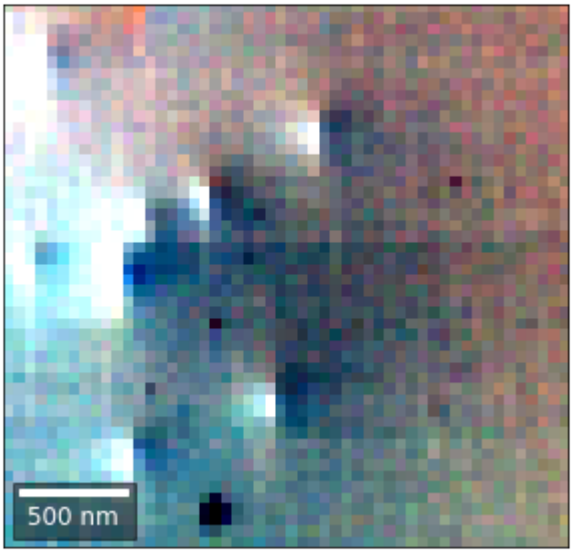
[25]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']# rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vFSD_RGB_row_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=0.8)

        rgb_direct.append(rgb)
```

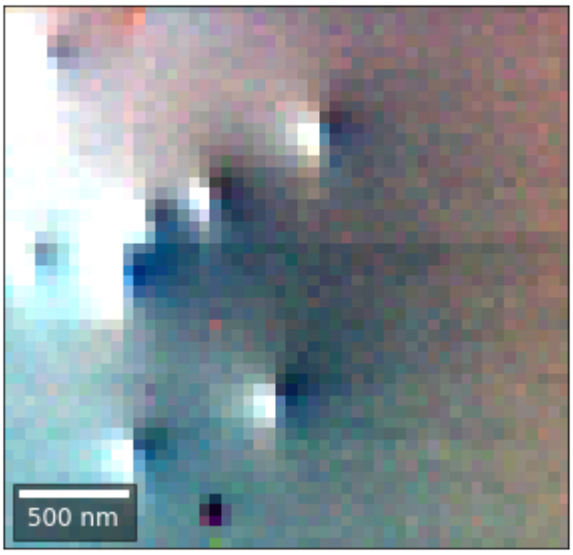

500 nm

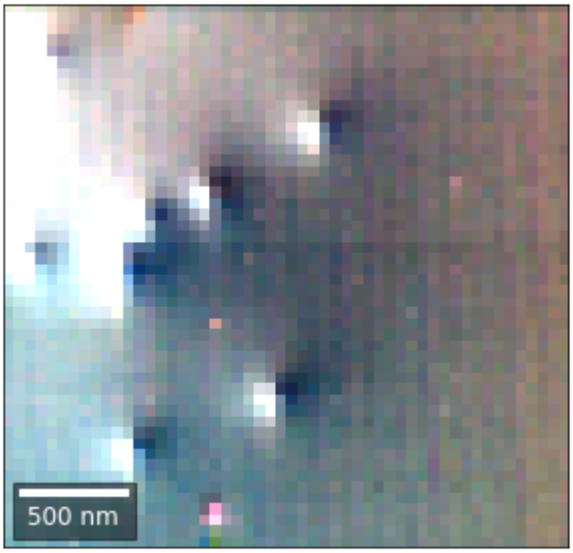

500 nm


500 nm


500 nm

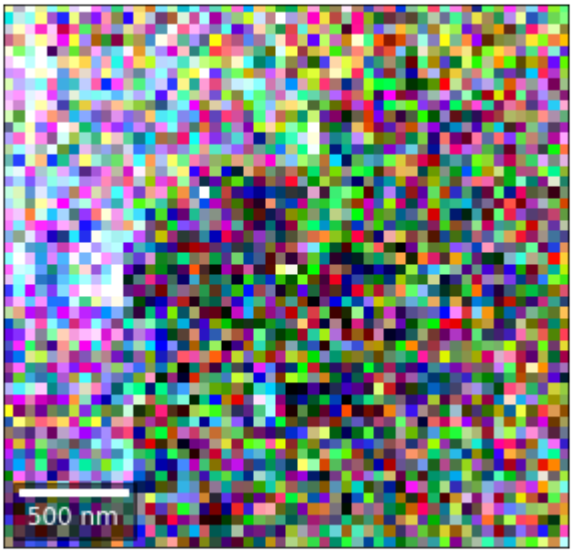

500 nm

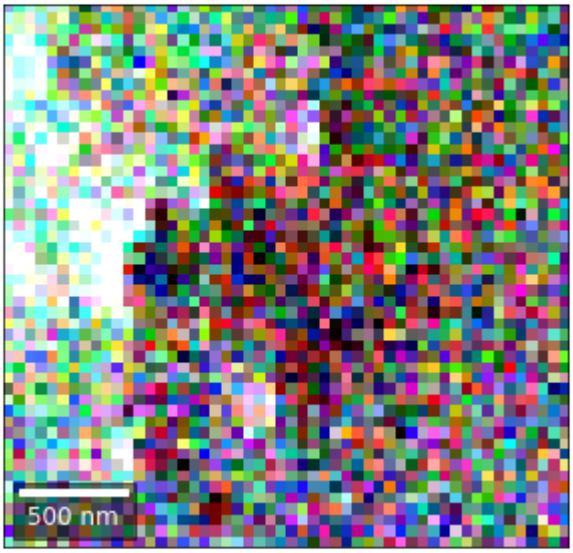

500 nm

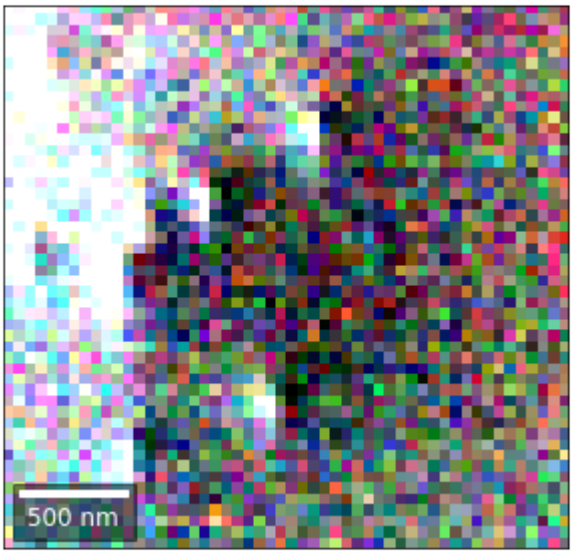


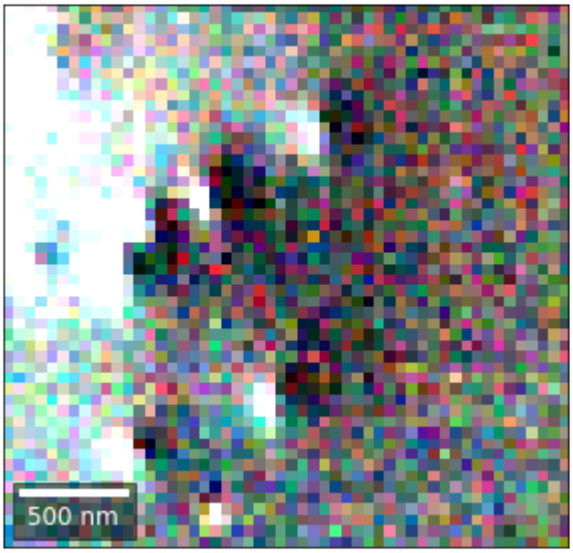
Differential color signals can be formed by calculating the relative changes to the ROI in the previous row:

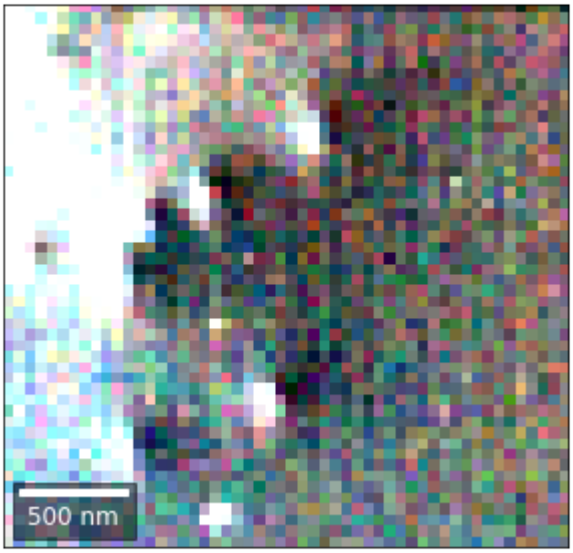
[26]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vbse']# rgb direct# relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vFSD_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

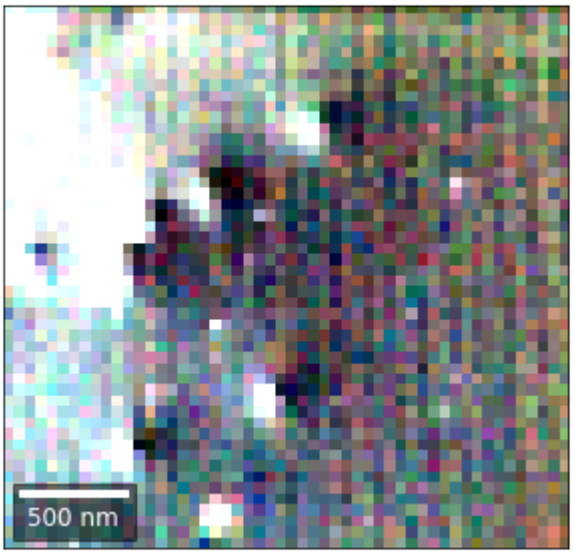

500 nm


500 nm


500 nm


500 nm


500 nm

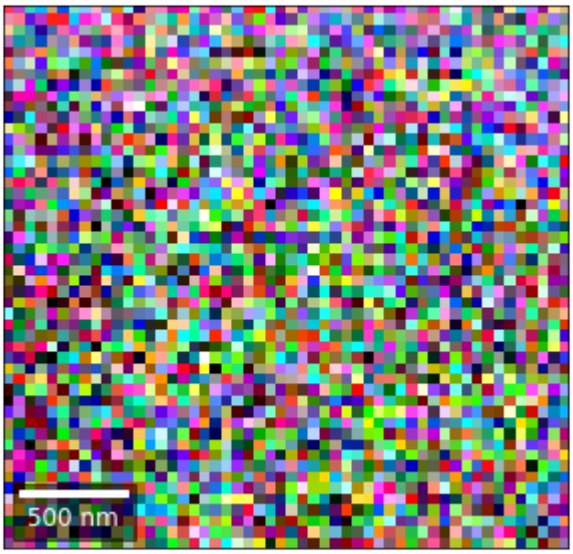

500 nm

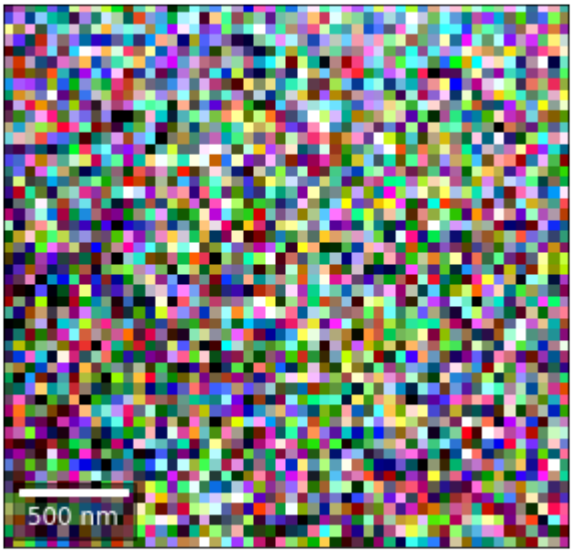
Kikuchi vBSE Imaging

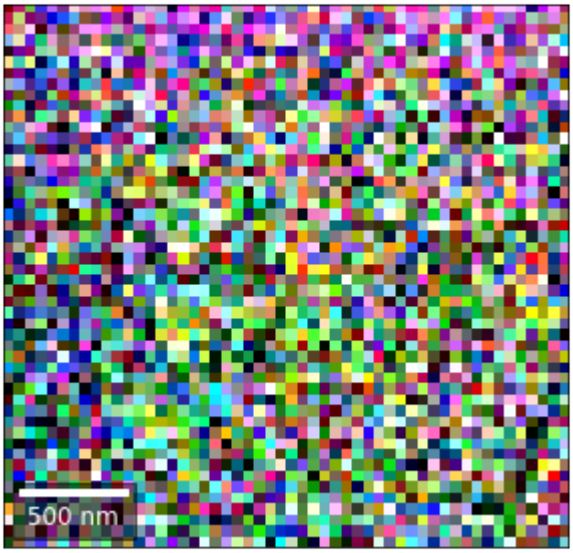
Kikuchi Pattern Array ROI as RGB

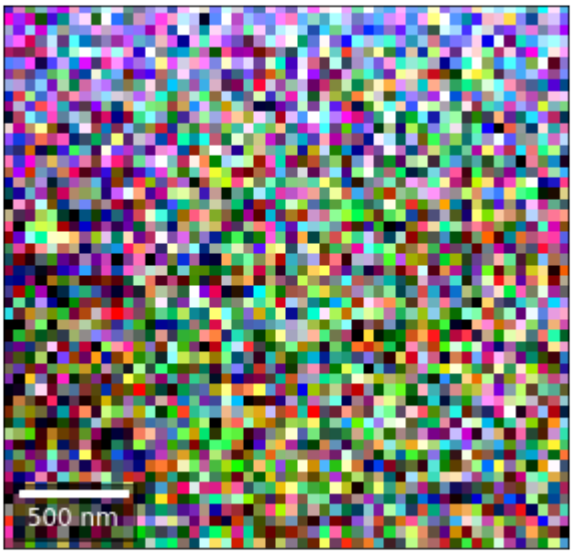
Not possible with simple BSE diodes!!!

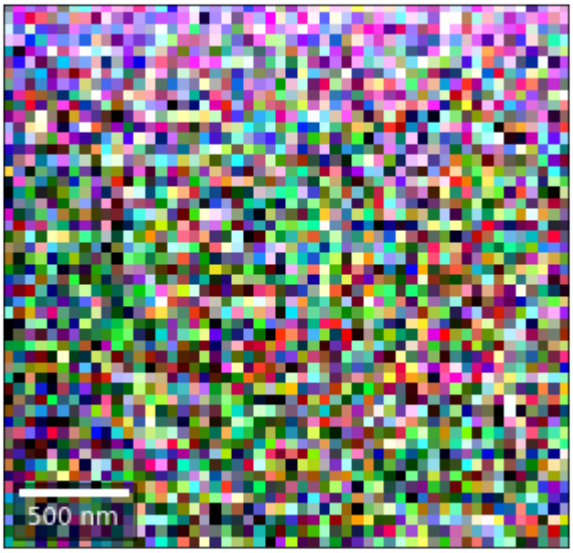
[27]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vkiku']

    # rgb direct
    rgb_direct = []

    for row in range(7):
        signal = vFSD[:,row,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,3]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,6]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vKiku_RGB_row_'+str(row),
                         rot180=False, microns=step_map_microns,
                         add_bright=0, contrast=1.2)

        rgb_direct.append(rgb)
```

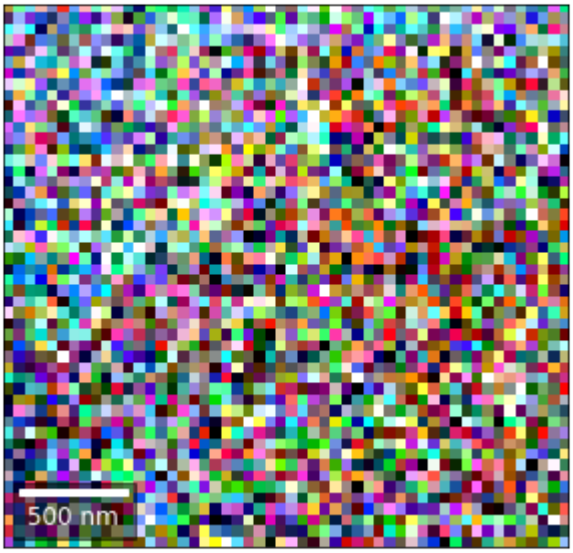


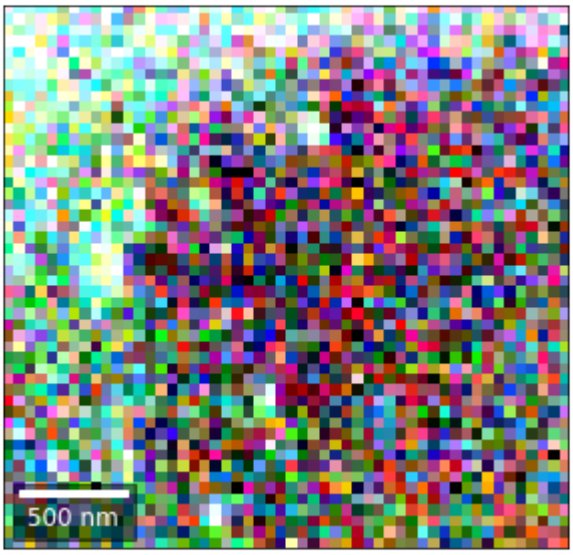

500 nm


500 nm


500 nm


500 nm


500 nm

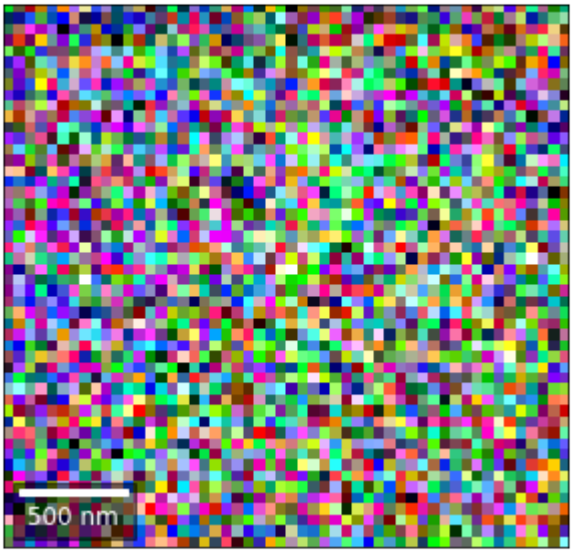

500 nm

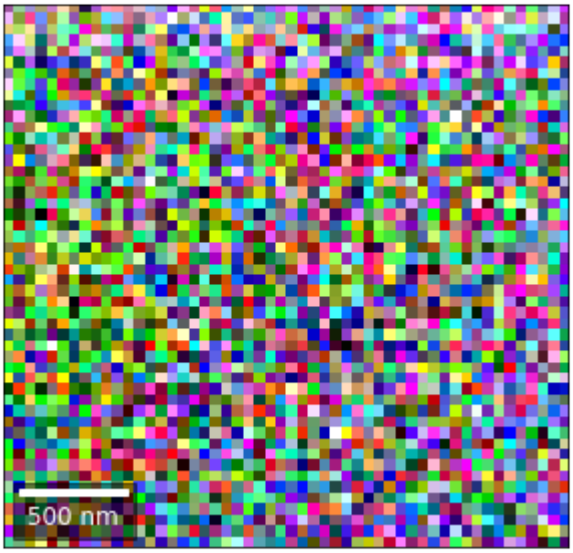
Differential Kikuchi Imaging

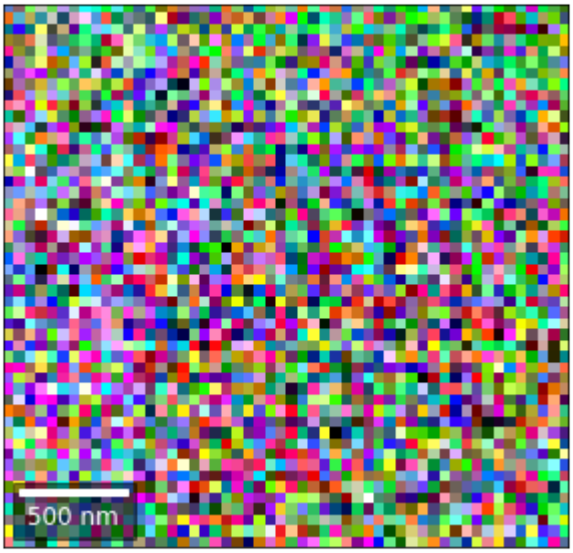
We determine the relative change between Kikuchi Array ROIs and use them as RGB values. The normalization to a reference ROI reduces the noise that is purely due to the variation of the background and the background processing on the complete pattern.

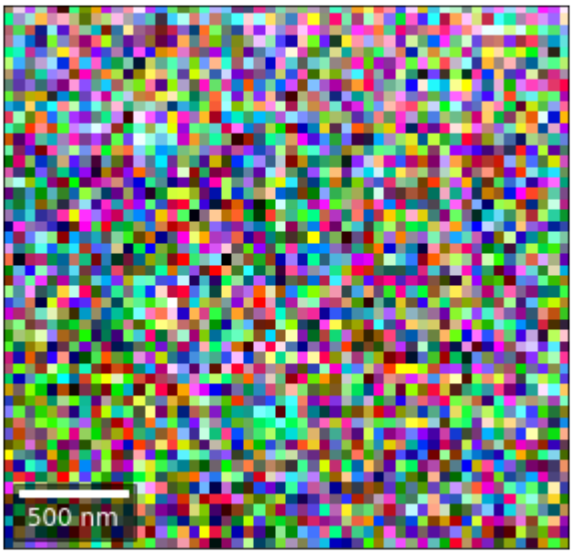
The colors represent orientation changes via the corresponding changes in ROIs of the Kikuchi patterns and the 7x7 array.

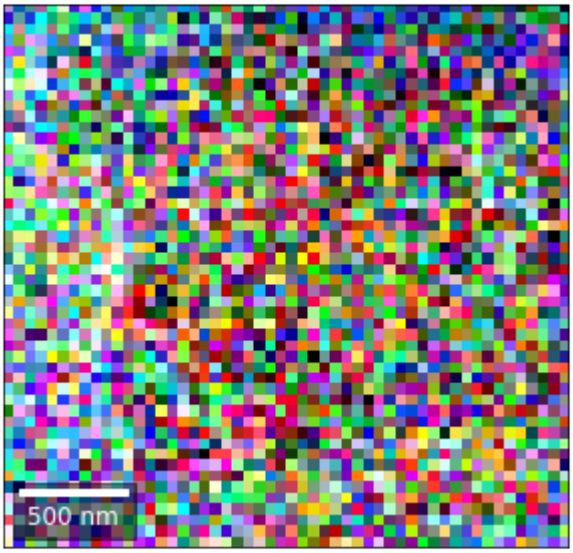
[28]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    vFSD= h5f['vkiku']
    # relative change to previous row
    for row in range(1,7):
        drow = -1
        signal = vFSD[:,row,0]/vFSD[:,row+drow,0]
        red = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,2]/vFSD[:,row+drow,2]
        green = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        signal = vFSD[:,row,4]/vFSD[:,row+drow,4]
        blue = make2Dmap(signal,XIndex,YIndex,MapHeight,MapWidth)

        rgb=plot_SEM_RGB(red, green, blue, MapHeight, MapWidth,
                         filename='vKiku_RGB_drow_'+str(row),
                         microns=step_map_microns,
                         rot180=False, add_bright=0, contrast=1.2)
```

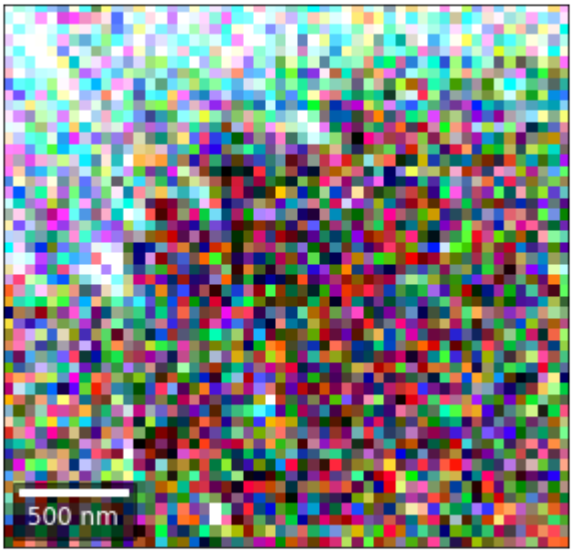



500 nm


500 nm


500 nm





Center of Mass Imaging

We can interpret the 2D image intensity as a mass density on a plane. The statistical moments of the density distribution (mean, variance, ...) can be used as signal sources. In the example below, we use the image center of mass as a signal source.

COM of Raw Patterns

[29]:
```
# calculate the center-of-mass for each pattern, use binning for speed
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_bin)
```

```
total points: 2600 current: 2600 finished -> total calculation time :    0.1 min
```

[30]:
```
# save the results in h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
```

(continues on next page)

(continued from previous page)

```
    h5f.create_dataset('/COM/COMxp_vbse', data=COMxp)
    h5f.create_dataset('/COM/COMyp_vbse', data=COMyp)
```

```
arBSE_GaN_Dislocations_1.hdf5
```

COM of Kikuchi Patterns

This should be seen with caution, as the background removal process is never perfect and will tend to leave some residual intensity, so that the Kikuchi COM is correlated with the raw pattern COM (which is dominated by the smooth background intensity).

[31]:
```
COMxp, COMyp = arbse.calc_COM_px(Patterns, process=process_kikuchi)
```

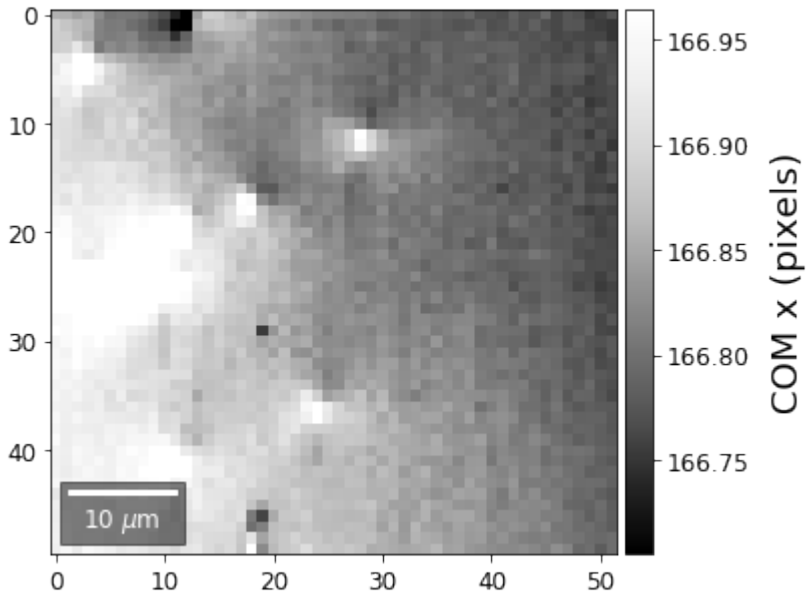
```
total points: 2600 current: 2600 finished -> total calculation time :    0.7 min
```

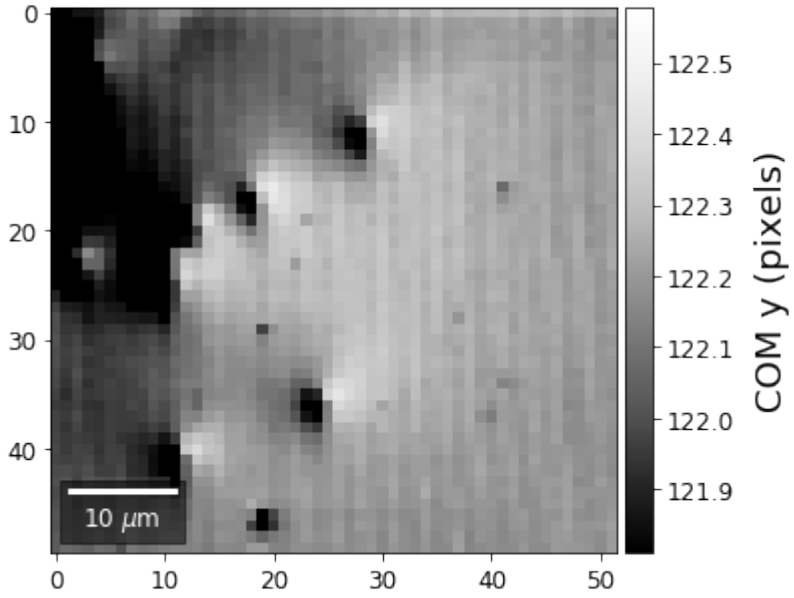
[32]:
```
# append to current h5ResultFile
print(h5ResultFile)
with h5py.File(h5ResultFile, 'a') as h5f:
    h5f.create_dataset('/COM/COMxp_kiku', data=COMxp)
    h5f.create_dataset('/COM/COMyp_kiku', data=COMyp)
```

```
arBSE_GaN_Dislocations_1.hdf5
```

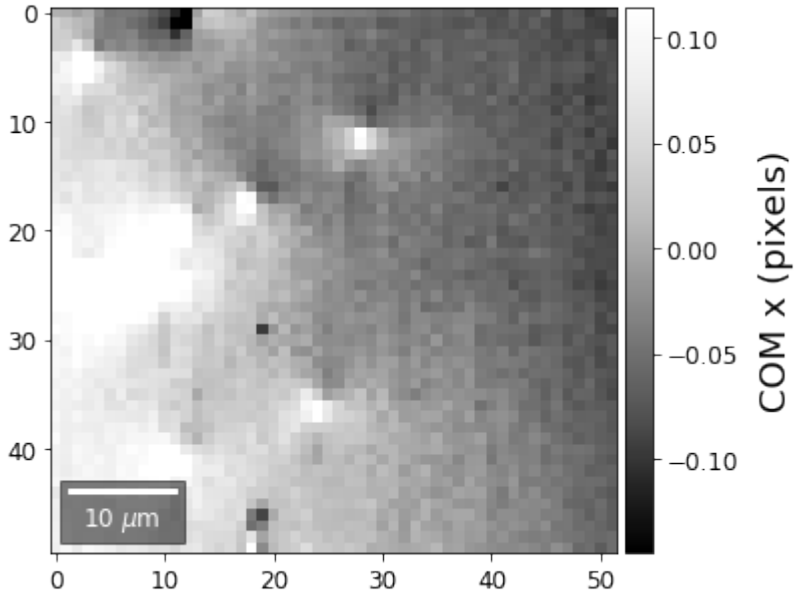
First, we calculate where the COMs are in x,y in pixels in the patterns:

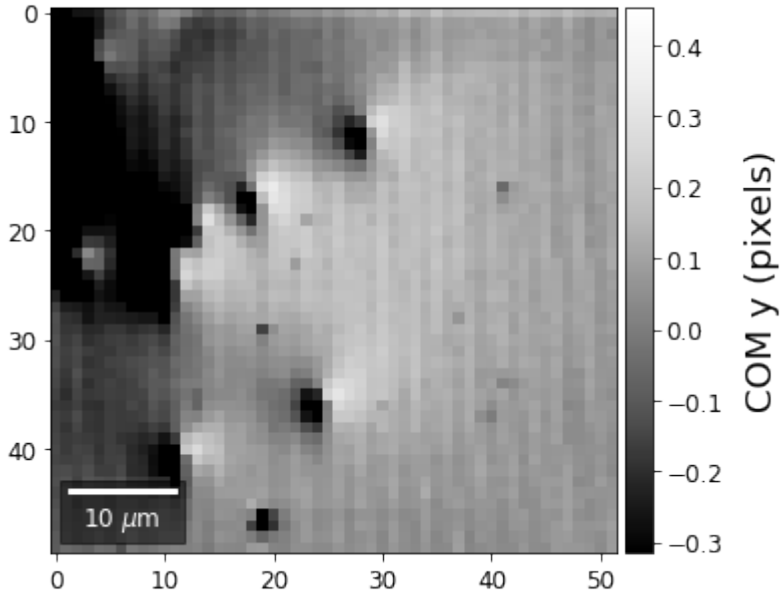
[33]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_vbse']
    COMyp = h5f['/COM/COMyp_vbse']

    comx_map0=make2Dmap(COMxp[:],XIndex,YIndex,MapHeight,MapWidth)
    comy_map0=make2Dmap(COMyp[:],XIndex,YIndex,MapHeight,MapWidth)

    plot_SEM(comx_map0, colorbarlabel='COM x (pixels)', cmap='Greys_r')
    plot_SEM(comy_map0, colorbarlabel='COM y (pixels)', cmap='Greys_r')
```



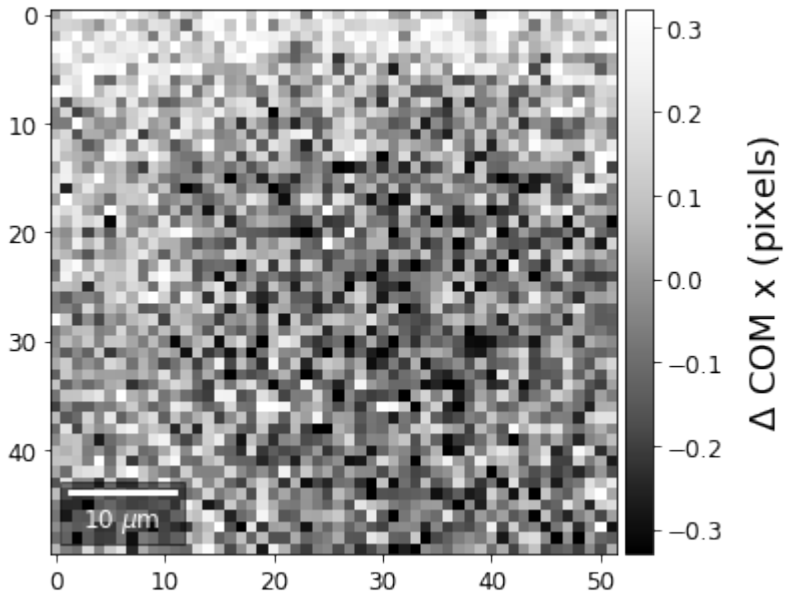


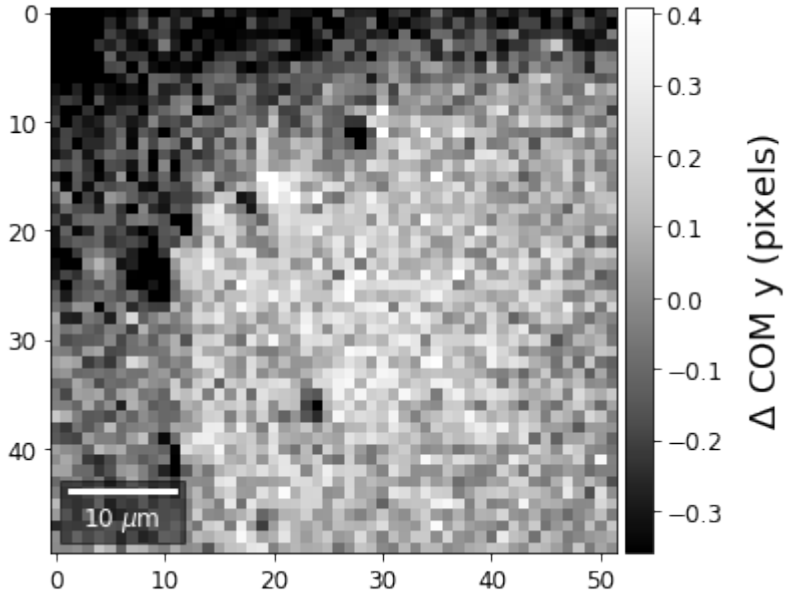
[34]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_vbse']
    COMyp = h5f['/COM/COMyp_vbse']

    meanx=np.mean(COMxp)
    meany=np.mean(COMyp)

    comx_map=make2Dmap(COMxp[:]-meanx,XIndex,YIndex,MapHeight,MapWidth)
    comy_map=make2Dmap(COMyp[:]-meany,XIndex,YIndex,MapHeight,MapWidth)

    plot_SEM(comx_map, colorbarlabel='COM x (pixels)', filename='comx', cmap='Greys_r')
    plot_SEM(comy_map, colorbarlabel='COM y (pixels)', filename='comy', cmap='Greys_r')
```





[35]:
```
with h5py.File(h5ResultFile, 'r') as h5f:
    COMxp = h5f['/COM/COMxp_kiku']
    COMyp = h5f['/COM/COMyp_kiku']
    meanx = np.mean(COMxp)
    meany = np.mean(COMyp)

    comx_map = make2Dmap(COMxp[:] - meanx, XIndex, YIndex, MapHeight, MapWidth)
    comy_map = make2Dmap(COMyp[:] - meany, XIndex, YIndex, MapHeight, MapWidth)

    plot_SEM(comx_map, colorbarlabel='$\Delta$ COM x (pixels)', filename='comx_kiku', cmap=
↪'Greys_r')
    plot_SEM(comy_map, colorbarlabel='$\Delta$ COM y (pixels)', filename='comy_kiku', cmap=
↪'Greys_r')
```





4 Simulation

5 Data Analysis

5.1 Image Similarity Measures

The Normalized Cross Correlation Coefficient

In this section we summarize some basic properties of the normalized cross correlation coefficient (NCC). This will be useful for the quantification of image similarity and for statistical tests of signifance based the observed values of the NCC.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np
import numpy.ma as ma

from scipy.stats import norm
from scipy.optimize import curve_fit
from scipy.integrate import trapz, simps

from skimage.io import imread
from aloe.plots import plot_image

from PIL import Image

# for fitting probability densities to a normal distribution
def gauss(x,a,x0,sigma):
    """
    gaussian function for input x values
    with amplitude=a, mean=x0, stddev=sigma
    """
```

(continues on next page)

(continued from previous page)

```
    return a*np.exp(-(x-x0)**2/(2*sigma**2))


# seed the random number generator so the randomness below is repeatable
np.random.seed(17139)
```

Definition

A description of various useful interpretations of the correlation coefficient is given by Rodgers and Nicewander in "Thirteeen Ways to Look at the Correlation Coefficent"[8]. An extensive treatment of the statistical use of correlation coefficients is given in D.C. Howell, "Statistical Methods for Psychology"[9].

The Pearson Correlation Coefficient[10], or normalized cross correlation coeffcient (NCC) is defined as:

$r = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}$

This can also be written as:

$r = r_{xy} = \sum_{i=1}^{n} \frac{1}{\sqrt{n-1}} \left(\frac{x_i-\bar{x}}{s_x}\right) \cdot \frac{1}{\sqrt{n-1}} \left(\frac{y_i-\bar{y}}{s_y}\right)$

sample mean: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

The normalization to $(n-1)$ degrees of freedom in the alternative form of r above is related to a corresponding definition of the sample standard deviation s: $s_x = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2}$

Notes:

- The numerical calculation of the standard deviation in Numpy can use $n-0$ or $n-1$, which is controlled by the parameter ddof=0/1. https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html *The average squared deviation is normally calculated as x.sum() / N, where N = len(x). If, however, ddof is specified, the divisor N - ddof is used instead.*
- To check the correct implementation, the NCC of a sample with itself needs to return 1.0

[2]:
```
def norm_data(data):
    """
    normalize data to have mean=0 and standard_deviation=1
    """
    mean_data=np.mean(data)
    std_data=np.std(data, ddof=1)
    #return (data-mean_data)/(std_data*np.sqrt(data.size-1))
    return (data-mean_data)/(std_data)


def ncc(data0, data1):
    """
    normalized cross-correlation coefficient between two data sets

    Parameters
    ----------
    data0, data1 :  numpy arrays of same size
    """
    return (1.0/(data0.size-1)) * np.sum(norm_data(data0)*norm_data(data1))
```

[8] http://dx.doi.org/10.1080/00031305.1988.10475524
[9] http://www.worldcat.org/oclc/913018446
[10] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

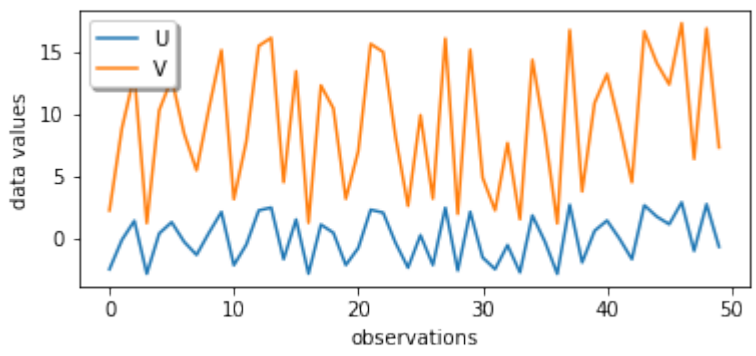
Simple Example

We will use U and V as the names for the random variables in this initial example (to avoid confusion with x and y in a two-dimensional plot, i.e. U and V are both independent variables or observations).

Initialize Some Random Data

[3]:
```
ndata=50
U_true= 3 *(-1.0 + 2.0*np.random.rand(ndata)) # vary between -3 and 3
print('U_true:')
print(U_true)
V_true=2.8*U_true+9.2 # V depends linearly on the *random* U
print('V_true:')
print(V_true)
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.plot(U_true, label='U')
ax.plot(V_true, label='V')
ax.legend(loc='upper left', shadow=True, fancybox=True)
ax.set_ylabel('data values')
ax.set_xlabel('observations')
plt.show()
```

```
U_true:
[-2.48878401 -0.13364715  1.39470182 -2.85237912  0.38434844  1.28474665
 -0.28210026 -1.33874788  0.44022736  2.10698153 -2.16631113 -0.51309775
  2.23026637  2.45915207 -1.67981306  1.50222604 -2.84405916  1.09486124
  0.43667326 -2.1551639  -0.78956172  2.28637227  2.05495757 -0.38035358
 -2.35889753  0.24364416 -2.15194566  2.44176342 -2.58892038  2.11739048
 -1.54875215 -2.48611461 -0.55459832 -2.74094723  1.82820669 -0.25800181
 -2.86068972  2.68440632 -1.94363372  0.59279678  1.42118888 -0.03309715
 -1.68236838  2.64292534  1.73635566  1.1210353   2.87912719 -1.01469519
  2.72985889 -0.68166506]
V_true:
[ 2.23140478  8.82578798 13.10516508  1.21333847 10.27617563 12.79729062
  8.41011928  5.45150593 10.43263662 15.09954829  3.13432885  7.76332631
 15.44474585 16.0856258   4.49652344 13.40623291  1.23663435 12.26561147
 10.42268513  3.16554107  6.98922719 15.60184237 14.9538812   8.13500998
  2.59508692  9.88220364  3.17455214 16.03693757  1.95102293 15.12869333
  4.86349399  2.2388791   7.64712469  1.52534776 14.31897873  8.47759492
  1.19006878 16.71633769  3.75782558 10.85983099 13.17932886  9.10732798
  4.48936852 16.60019095 14.06179585 12.33889883 17.26155614  6.35885346
 16.8436049   7.29133783]
```



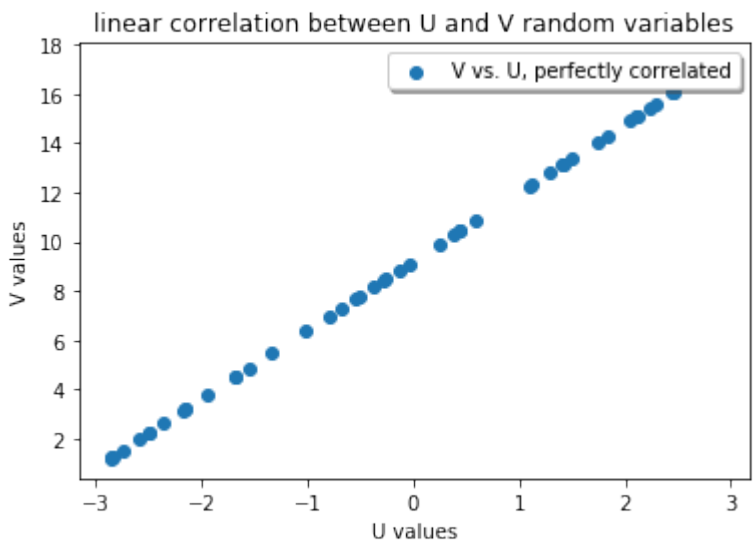
Although each of the line plots by itself looks rather random, when we compare U and V, we see that V is rising when U is rising and V is falling when U is falling, just with a different amplitude and with

an underlying offset. A high/low value in the U data set *correlates with* a high/low value in the V data set.

Correlation Plot

To see the relationship between U and V, we plot U as a function of V. Note the use of a scatter plot instead of a line plot; we just like to see the relationship between the related data *points*.

[4]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(U_true,V_true, label= "V vs. U, perfectly correlated")
ax.legend(loc='upper right', shadow=True, fancybox=True)
ax.set_ylabel('V values')
ax.set_xlabel('U values')
ax.set_title('linear correlation between U and V random variables')
plt.show()
```



As we see above, if U has a specific observed random value, V has a completely determined value, which does not depend on the position of the observation in the U and V data set, the order of the observation of the (U,V) pairs does not matter. In the plot that would mean that the lower values of U could have been observed before the higher values of U; in fact, any order of observations would still create the line that we see.
In this way, the line that we see is just a result of this perfect correlation, for non-perfect correlation, this plot will look different, as we will see below.

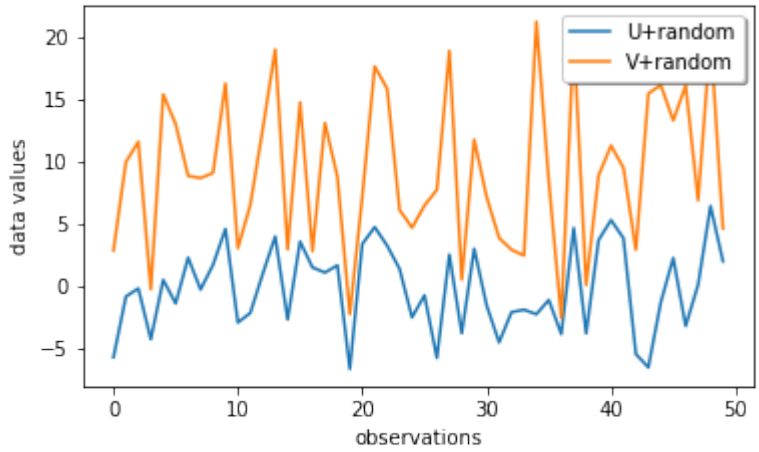
The NCC is 1.0 for these two data sets, indicating that apart from scaling and offset, they are "similar". Note that the NCC is symmetric, i.e. exchanging the data sets does not change the NCC:

[5]:
```
ncc1=ncc(U_true,V_true)
print(ncc1)
ncc1_rev=ncc(V_true,U_true)
print(ncc1_rev)
```

```
0.9999999999999999
0.9999999999999999
```

Now we add additional random variations on both U and V and see how the NCC changes.

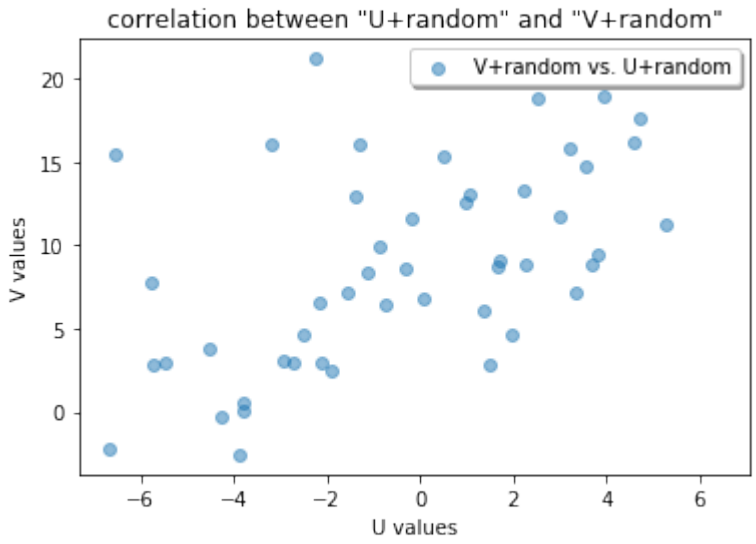
[6]:
```
stddev=3.0
U_exp = U_true + np.random.normal(scale=stddev,size=ndata)
V_exp = V_true + np.random.normal(scale=stddev,size=ndata)


fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.plot(U_exp, label='U+random')
ax.plot(V_exp, label='V+random')
ax.legend(loc='upper right', shadow=True, fancybox=True)
ax.set_ylabel('data values')
ax.set_xlabel('observations')
plt.show()
```



Now we check the correlation between the U and V data with random errors, and we see that we don not have a nice line anymore!

[7]:
```
ncc2=ncc(U_exp,V_exp)
print(ncc2)


fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(U_exp,V_exp, alpha=0.5, label= "V+random vs. U+random")
ax.legend(loc='upper right', shadow=True, fancybox=True)
ax.set_ylabel('V values')
ax.set_xlabel('U values')
ax.set_title('correlation between "U+random" and "V+random"')
plt.show()
```

```
0.5843524495386023
```



Statistical Distribution of the Cross Correlation Coefficient

We now compare the distribution of the NCC values for a large set of experiments (y) with the theory (x). So x and y correspond to U and V in the simple example above.

We can start by looking at the result of totally random data in x and y, where the different NCCs should be distributed around zero.

Note: Try to set the "stddev" value below to different values and observe what happens if x and y become increasingly spread. The surprising result is that the spread of the NCC histogram does *not* change with the standard deviation of the distribution of the actual values in the x and y data sets! Then what determines the shape of this curve? The answer will follow below...

[8]:
```
def get_correlated_samples(nsamples=50, r=0.5, means=[1,1], stddevs= [1.0, 1.0]):
    """
    get two correlated random data sets
    """
    sd = np.diag(stddevs) # SD in a diagonal matrix
    observations = np.random.normal(0, 1, (2, nsamples))

    cor_matrix = np.array([[1.0, r],
                           [r, 1.0]]) # correlation matrix [2 x 2]

    cov_matrix = np.dot(sd, np.dot(cor_matrix, sd)) # covariance matrix

    Chol = np.linalg.cholesky(cov_matrix) # Cholesky decomposition
    sam_eq_mean = Chol.dot(observations)  # Generating random MVN (0, cov_matrix)

    s = sam_eq_mean.transpose() + means  # adding the means column wise
    samples = s.transpose()              # Transposing back
    return samples


def get_r_sample(nruns=1000, nsamples=50, r=0.5,
                 means=[0, 0], stddevs=[1.0, 1.0]):
    """ get correlated x,y datasets with defined correlation r """
    samples=np.zeros(nruns)
    for i in range(nruns):
```

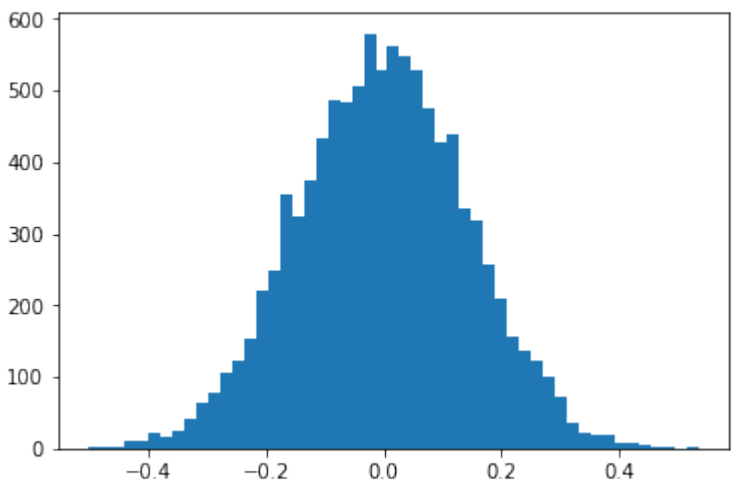
(continues on next page)

(continued from previous page)

```
        x,y = get_correlated_samples(nsamples=nsamples, r=r,
                                     means=means, stddevs=stddevs)
        samples[i]=ncc(x,y)
    return samples


def get_ncc_sample_r0(nruns=1000,stddev=1.0):
    """
    get ncc samples from population with correlation=0
    """
    samples=np.zeros(nruns)
    for i in range(nruns):
        x = np.random.normal(scale=stddev,size=ndata)
        y = np.random.normal(scale=stddev,size=ndata)
        samples[i]=ncc(x,y)
    return samples
```

We will first plot a histogram for a range of trials where we create two datasets of ndata points and calculate their NCC. The NCC will not be constant, as our data sets are random. The NCC can be different from zero in the specific trials, as we can have accidentally correlated values for a dataset of limited size, i.e. there is some chance for three random values to be correlated with NCC=0.1 to three other random values. We expect that the variation of the NCC values around zero will become sharper for larger data sets, i.e. there is a much smaller chance for 100 random values to show NCC=0.1 to 100 other random values.

[9]:
```
nruns=10000
stddev=37.0
ncc_samples=np.zeros(nruns)
for i in range(nruns):
    x = np.random.normal(scale=stddev,size=ndata)
    y = np.random.normal(scale=stddev,size=ndata)
    ncc_samples[i]=ncc(x,y)

counts, binval, _ = plt.hist(ncc_samples,bins=51)
```

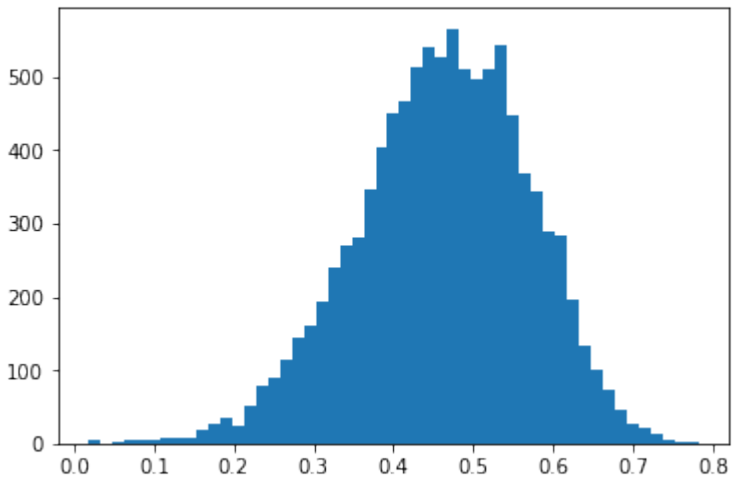


Now we compare experiments for the same underlying true x and y data (taken from the U and V data sets above), however with two different amounts of randomness in the experimental x and y. Because of the correlation between x and y, the NCC distribution is centered around a nonzero value and the distribution histogram of the NCC becomes *asymmetric* relative to the most probable NCC value.

First, we add Gaussian noise with stddev=10.0 to the true, perfectly correlated values, and the NCC

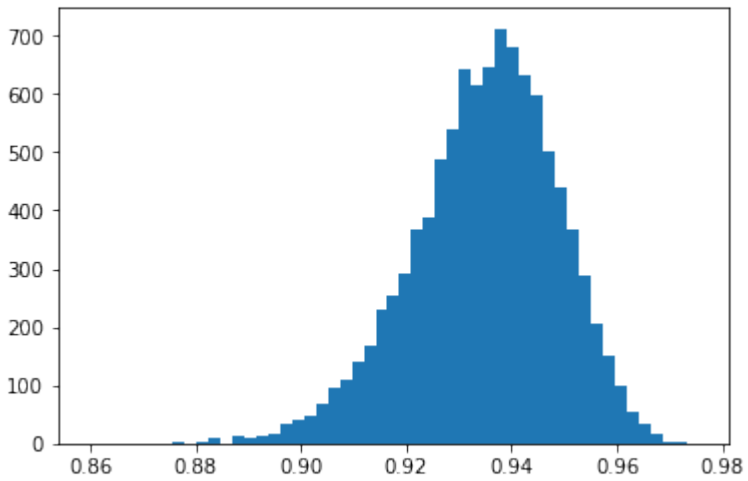
values will decrease any vary asymmetrically around about 0.4:

[10]:
```
nruns=10000
stddev=10.0

x_true = U_true
y_true = V_true

ncc_samples=np.zeros(nruns)
for i in range(nruns):
    x = x_true
    y = y_true + np.random.normal(scale=stddev,size=ndata)
    ncc_samples[i]=ncc(x,y)

counts, binval, _= plt.hist(ncc_samples,bins=51)
```



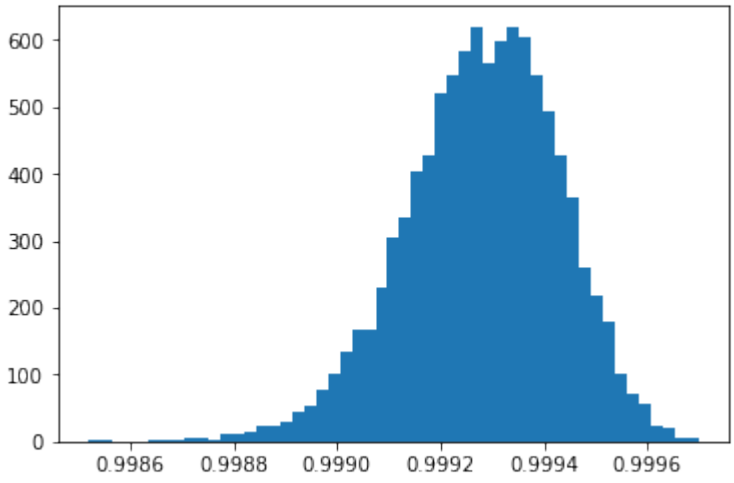
With less random noise (stddev=2.0) on the true, perfectly correlated values, the most probable NCC moves nearer to 1.0:

[11]:
```
nruns=10000
stddev=2.0

ncc_samples=np.zeros(nruns)
for i in range(nruns):
    x2 = x_true
    y2 = y_true + np.random.normal(scale=stddev,size=ndata)
    ncc_samples[i]=ncc(x2,y2)

counts, binval, _= plt.hist(ncc_samples,bins=51)
```



With even less noise, stddev=0.2, the NCC values approach 1.0. Note that the NCC still varies, but it can never be larger than 1.0.

[12]:
```
nruns=10000
stddev=0.2

ncc_samples=np.zeros(nruns)
for i in range(nruns):
    x2 = x_true
    y2 = y_true + np.random.normal(scale=stddev,size=ndata)
    ncc_samples[i]=ncc(x2,y2)

counts, binval, _= plt.hist(ncc_samples,bins=51)
```



Degrees Of Freedom:

We check what happens to the distribution of NCC values when we repeat some values in the x and y data sets.

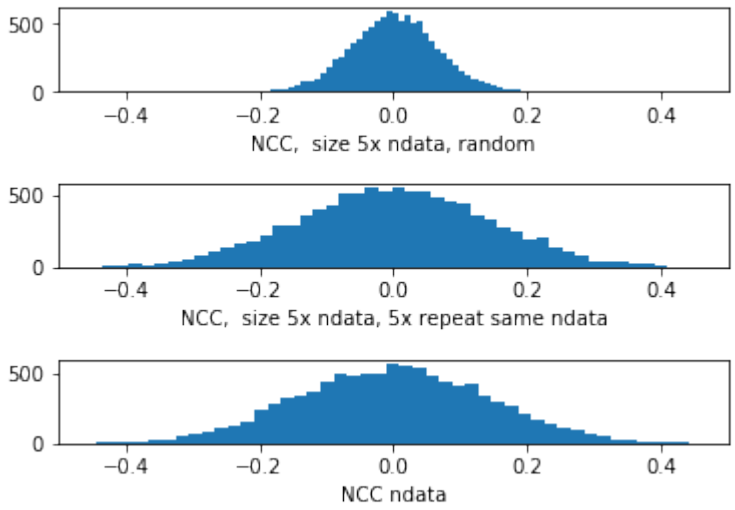
[13]:
```
nruns=10000
stddev=50.0

# sample size 5 times of original data, all random
```

(continues on next page)

(continued from previous page)

```
# ~ 5 times more dof
ncc_samples=np.zeros(nruns)
for i in range(nruns):
    x = np.random.normal(scale=stddev,size=5*ndata)
    y = np.random.normal(scale=stddev,size=5*ndata)
    ncc_samples[i]=ncc(x,y)

# repeat 5 samples of ndata
ncc_samples5=np.zeros(nruns)
for i in range(nruns):
    x = np.random.normal(scale=stddev,size=ndata)
    y = np.random.normal(scale=stddev,size=ndata)
    x=np.ravel([x,x,x,x,x])
    y=np.ravel([y,y,y,y,y])
    ncc_samples5[i]=ncc(x,y)

# compare to result for original sample size/dof
ncc_samples1=np.zeros(nruns)
for i in range(nruns):
    x = np.random.normal(scale=stddev,size=ndata)
    y = np.random.normal(scale=stddev,size=ndata)
    ncc_samples1[i]=ncc(x,y)
```

[14]:
```
fig = plt.figure()
plt.subplots_adjust(hspace = 1.1)
ax = fig.add_subplot(311)
counts, binval, _ = plt.hist(ncc_samples,bins=51)
ax.set_xlim(-0.5,0.5)
ax.set_xlabel('NCC,  size 5x ndata, random')

ax = fig.add_subplot(312)
counts, binval, _ = plt.hist(ncc_samples5,bins=51)
ax.set_xlim(-0.5,0.5)
ax.set_xlabel('NCC,  size 5x ndata, 5x repeat same ndata')

ax = fig.add_subplot(313)
counts, binval, _ = plt.hist(ncc_samples1,bins=51)
ax.set_xlim(-0.5,0.5)
ax.set_xlabel('NCC ndata')

plt.show()
```



The variation of ncc_samples5 (middle) is the same as that of the initial set with size ndata (bottom), while a data set with 5×ndata truly random data points shows a reduced variation in the distribution of the NCC values. We cannot reduce the variation of the NCC by simply repeating some values.

Since in our data set of 5× repeated ndata samples, not all observations are independent, the sample size in this case is **NOT** equivalent to the degrees of freedom in the data set. From the plots above, we would expect that, compared to 5× ndata, the effective sample size should be ndata, as this gives the same distribution as our 5× repeated ndata points.

The correlation in the 5× repeated ndata points reduces the effective sample size, which is signaled by the increased width of the NCC curve around zero, compared to 5×ndata independent random values.

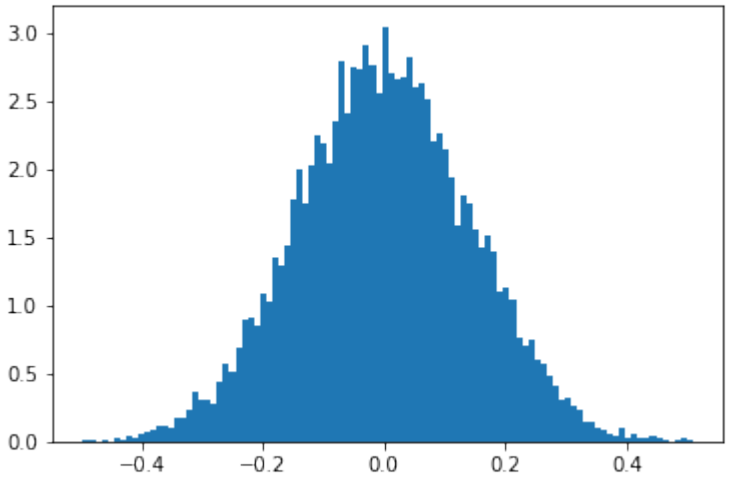
Will this observation enable us to define at least an *effective* sample size (DOF) via the distribution of the NCC? (WIP)

Standard Deviation and Degrees Of Freedom for Zero Correlation

Fit to Model for NCC distribution around $r = 0$:

[15]:
```
ncc0=get_ncc_sample_r0(nruns=10000, stddev=0.7)
```

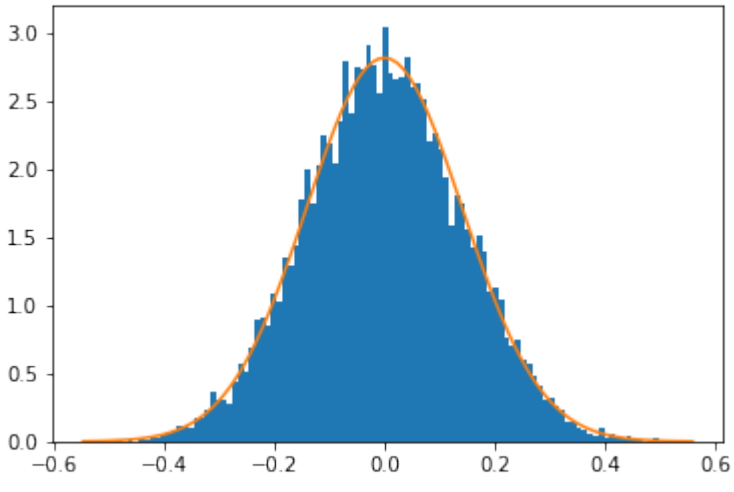
[16]:
```
ax = plt.subplot(111)
counts, binval, patches = plt.hist(ncc0,bins=100, density=True)
```



[17]:
```
zmean,zstd=norm.fit(ncc0)
print(zmean,zstd)
dof0=2+(1.0/zstd)*(1.0/zstd)
print(dof0)
print(ndata)
print('?!')
```

```
1.3458304497317376e-05 0.14160349116255136
51.87145953120684
50
?!
```

[18]:
```
plt.hist(ncc0, bins=100, density=True)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, zmean, zstd)
plt.plot(x, y)
plt.show()
```



For **uncorrelated** data sets (mean value of NCC is o), we can extract the initial degrees of freedom (the independent data points N) from the standard deviation σ of a normal distribution fitted to the histogram of the NCC values (see Howell):

$\sigma_r = \frac{1}{\sqrt{N-2}}$

Given only the histogram above, we can estimate the ndata=50 defined for the random data sets at the beginning of this chapter. In addition, note that this result does not depend on the standard deviation or mean of the uncorrelated data sets, which seems a little like magic, doesn't it?

Can we also achieve something similar when the data is correlated, i.e. the mean NCC is $r > 0$? It turns out that this is possible after transforming the NCC r values so that they become distributed normally also for $r > 0$, as will be discussed next.

Z transformation

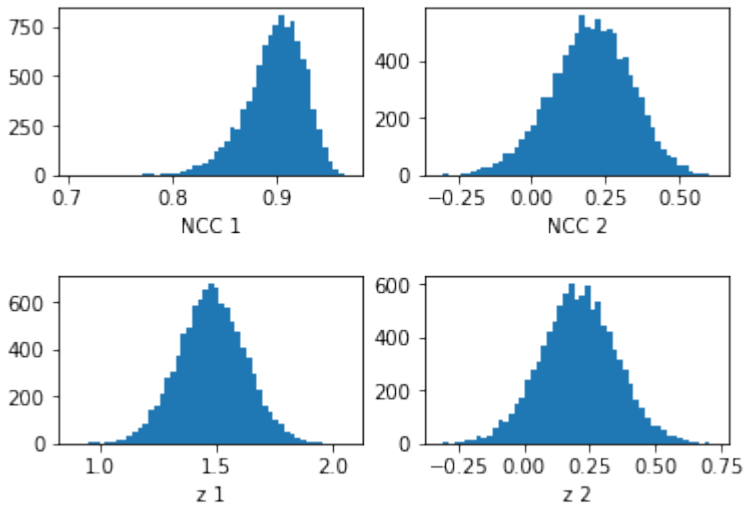
Fisher (1921) has shown that one can convert the NCC r to a value z that is approximately normal distributed (see Howell, Chapter 9, "Correlation and Regression"). The calculation of z will enable us to compare the variation of the NCC at *different* levels of the NCC, e.g. we can answer questions like "Is the correlation between two data sets significantly different from the correlation between a second pair of data sets" (where the data sets can have a different number of observations etc and thus a different statistical variation of the NCC values).

$z = 0.5 \cdot \ln \frac{|1+r|}{|1-r|}$

[19]:
```
def z_transform(r):
    """
    Fisher's z transform for the correlation coefficient
    """
    return 0.5*np.log((1.0+r)/(1.0-r))
```

[20]:
```
ncc1 = get_r_sample(nruns=10000, r=0.9, stddevs=[0.1,200])
z1=z_transform(ncc1)
ncc2 = get_r_sample(nruns=10000, r=0.2, stddevs=[3,17])
z2=z_transform(ncc2)
```

[21]:
```
fig = plt.figure()
plt.subplots_adjust(hspace = 0.6)

ax = fig.add_subplot(221)
counts, binval, _= ax.hist(ncc1,bins=51, density=False)
#ax.set_xlim(-0.5,0.8)
ax.set_xlabel('NCC 1')

ax = fig.add_subplot(222)
counts, binval, _= ax.hist(ncc2,bins=51, density=False)
#ax.set_xlim(-0.5,0.8)
ax.set_xlabel('NCC 2')

ax = fig.add_subplot(223)
counts, binval, _= ax.hist(z1,bins=51, density=False)
#ax.set_xlim(0.49,0.8)
ax.set_xlabel('z 1')

ax = fig.add_subplot(224)
counts, binval, _= ax.hist(z2,bins=51, density=False)
#ax.set_xlim(0.49,0.8)
ax.set_xlabel('z 2')
plt.show()
```



Note, in the plots above, the noise can be different between upper and lower rows because of the different binning of the NCC r values vs. z-transformed values.

Estimation of DOF from correlated data

The Degrees Of Freedom N for non-zero correlation can be estimated from the standard deviation of the z-transformed NCC values. For **correlated** data sets (mean value of NCC <> 0), z is approximately normally distributed with standard error σ_z (see Howell, Chapter 9, "Correlation and Regression"):

$\sigma_z = \frac{1}{\sqrt{N-3}}$

Like in the case of zero correlation, we can try to recover the initial Degrees of Freedom from the standard deviation of z as:

$N = \frac{1}{\sigma_z^2} + 3$

[22]:
```
def get_DOF_from_zstd(zstd):
    return 3+(1.0/zstd)**2

zmean1,zstd1=norm.fit(z1)
print('z1 mean, std:', zmean1,zstd1)
dof1=get_DOF_from_zstd(zstd1)
print('DOF1 from sigma:', dof1)

zmean2,zstd2=norm.fit(z2)
print('z2 mean, std:', zmean2,zstd2)
dof2=get_DOF_from_zstd(zstd2)
print('DOF2 from sigma:', dof2)
```

```
z1 mean, std: 1.4800939597596763 0.14494532687212142
DOF1 from sigma: 50.598313381070234
z2 mean, std: 0.20615464743747453 0.14421090214204885
DOF2 from sigma: 51.084356972551404
```

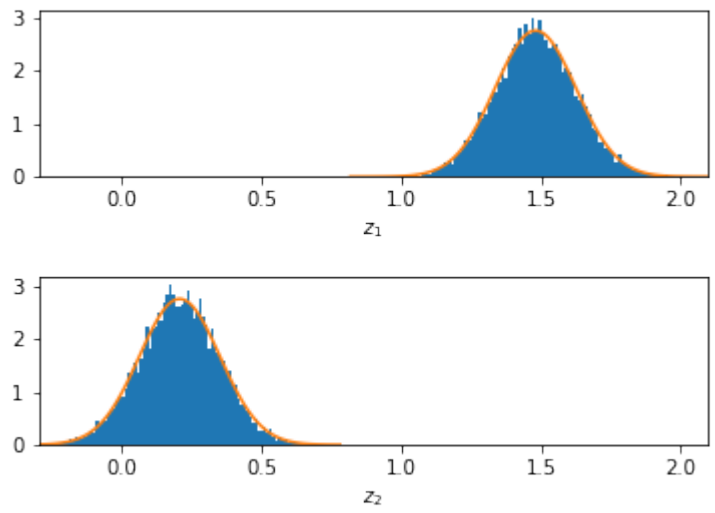
For our independent random data points with defined correlation , we nicely obtain values near 50, the initial ndata, for arbitrary mean and standard deviation in the two data sets! Statistical magic again...

[23]:
```
fig=plt.figure()
plt.subplots_adjust(hspace = 0.6)
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(211)
ax.hist(z1, bins=100, density=True)
xmin, xmax = ax.get_xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, zmean1, zstd1)
ax.plot(x, y)
ax.set_xlabel('$z_1$')
ax.set_xlim([-0.3, 2.1])


ax = fig.add_subplot(212)
ax.hist(z2, bins=100, density=True)
xmin, xmax = ax.get_xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, zmean2, zstd2)
ax.plot(x, y)
ax.set_xlabel('$z_2$')
ax.set_xlim([-0.3, 2.1])
plt.show()
```



In the two plots above, we can see that the z-transformation nicely scales the distributions to have the same standard deviation (which is determined by the DOF), even with strongly different values of the correlation coefficients. As the standard deviations are the same, we obtain the same estimate for ndata.

Example Data: Kikuchi Pattern Fits

We now look at an example for the distribution of the values of the NCC for a range of different "experiments". An "experiment" will correspond to the intensity values in an image that was flattened to a 1D array. The example data is from an EBSD measurement for a large number (16752) of different Kikuchi patterns (200x142 = 28400 pixels = length of 1D array data set like in U or V above). The aim was to discriminate between two different possible "theories", which show a slightly different NCC when compared to a single experimental pattern.

For the 16752 pattern fits, a different NCC value is obtained relative to both theories (r_0, r_1), and we have to decide which of the two theories is the better fit and possibly give a confidence level for this discrimination.

[24]:
```
mtxt=np.loadtxt(fname='./data/K2C_0802_200x142.MTXT',skiprows=1)
print(mtxt.shape)
```

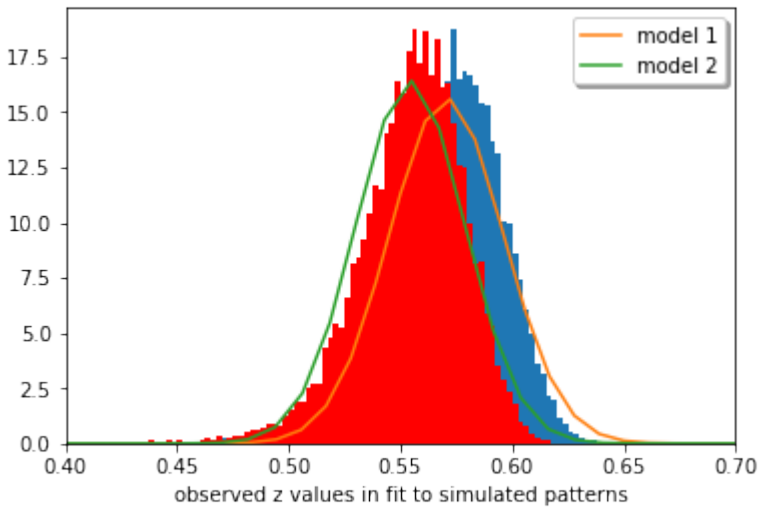
```
(16752, 17)
```

[25]:
```
r0=mtxt[:,15]
r1=mtxt[:,16]

z0=z_transform(r0)
z1=z_transform(r1)

#save
zxc0=z0
zxc1=z1
```

[26]:
```
zmean0,zstd0=norm.fit(z0)
print(zmean0,zstd0)

dof0=get_DOF_from_zstd(zstd0)
print(dof0)

zmean1,zstd1=norm.fit(z1)
print(zmean1,zstd1)
dof1=get_DOF_from_zstd(zstd1)
print(dof1)
```

```
0.5704894030485244 0.02553383706134989
1536.796825594422
0.5544172786435242 0.024283068143748792
1698.8712701227655
```

[27]:
```
counts0, bin_edges0, bars0 =  plt.hist(z0, bins=400,
                                range=[0.0, 1.0], density=True)

xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, zmean0, zstd0)
plt.plot(x, y, label = 'model 1')

counts1, bin_edges1, bars1 = plt.hist(z1, bins=400,
                        range=[0.0, 1.0], density=True,color='r')

xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, zmean1, zstd1)
plt.plot(x, y, label = 'model 2')
plt.xlim(0.4,0.7)
plt.xlabel('observed z values in fit to simulated patterns')
plt.legend(loc='upper right', shadow=True, fancybox=True)
plt.show()
```

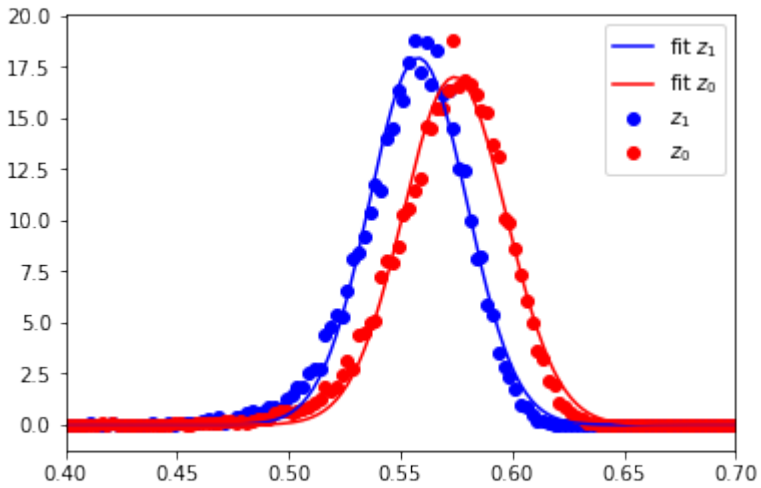


Alternative description of the statistical distribution: Gaussian peak fitting

Instead of using the mean and standard deviation as estimators, we can also use a peak fit to the distribution of NCC values.

[28]:
```
z1= 0.5*(bin_edges1[1:] + bin_edges1[:-1])
print('area under curve: ',trapz(counts1, z1))

popt,pcov = curve_fit(gauss,z1,counts1,p0=[np.max(counts1),np.mean(z1),1.0])

a_fit, mean_fit, sigma_fit = popt
print(a_fit, mean_fit, sigma_fit)
dofdz=get_DOF_from_zstd(sigma_fit)
print('DOF: ',dofdz)

plt.scatter(z1,counts1,label='$z_1$',color='b')
plt.plot(z1,gauss(z1,*popt),'b',label='fit $z_1$')


z0= 0.5*(bin_edges0[1:] + bin_edges0[:-1])
print('area under curve: ',trapz(counts0, z0))

popt,pcov = curve_fit(gauss,z0,counts0,p0=[np.max(counts0),np.mean(z0),1.0])

a_fit, mean_fit, sigma_fit = popt
print(a_fit, mean_fit, sigma_fit)
dofdz=3+(1.0/sigma_fit)**2
print('DOF: ',dofdz)

plt.scatter(z0,counts0,label='$z_0$',color='r')
plt.plot(z1,gauss(z0,*popt),'r',label='fit $z_0$')
#plt.xlim(0.75,0.87)
plt.xlim(0.4,0.7)
plt.legend()
plt.show()
```

```
area under curve:  1.0000000000000002
17.93916300442126 0.5579065730213564 -0.021820875063898286
DOF:  2103.175919211425
area under curve:  1.0000000000000002
17.003931263802883 0.5741550518076884 -0.02305540709794952
DOF:  1884.2842072483918
```



From the standard deviation of the distribution of the NCC values, we would naively estimate the effective DOF to be in the range of 2000 for the patterns of 200x142=28400 pixels.

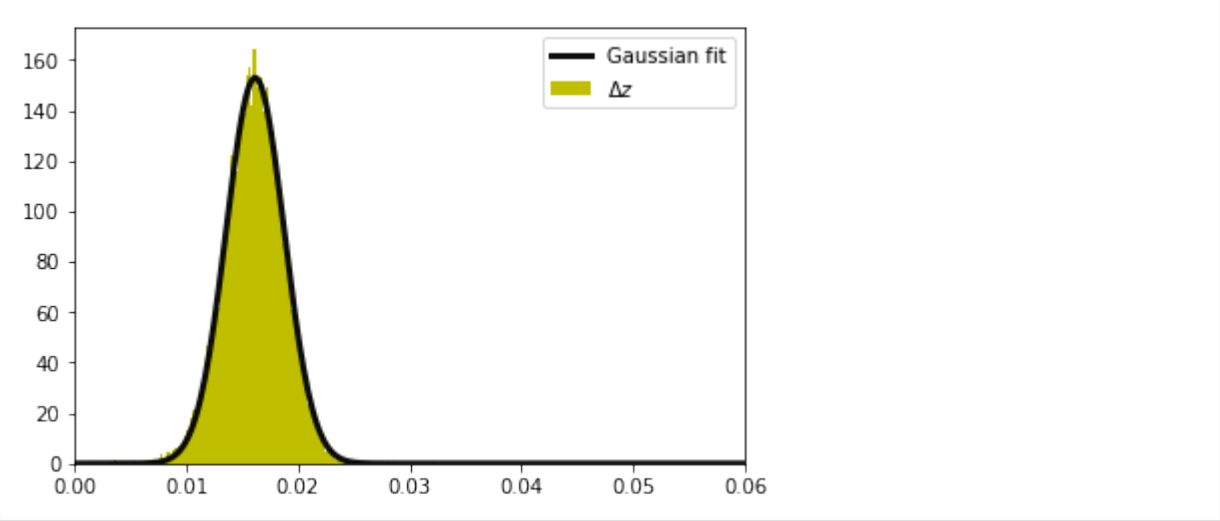
In a test scenario, we could assume the null hypothesis that both NCCs are equal, and thus test for the difference to be zero.

The overlap of the curves does not mean that we actually have data points (patterns) where the *difference* in z is zero. This is because the values in both curves are strongly correlated, i.e. if we have a value in the high end of the tail of th z_0 curve, the corresponding z_1 value will be in the high end tail of the z_1 curve, i.e. not in the low end which would be required to make the difference $z_1 - z_0$ zero.

To see the correlation effect, we additionally analyze the distribution of the *differences* of the z-transformed r values:

[29]:
```
dz=np.abs(zxc1-zxc0)
counts, bin_edges, bars = plt.hist(dz, bins=500, range=[0.0, 0.1],
                         density=True, color='y',label='$\Delta z$')

z = 0.5*(bin_edges[1:] + bin_edges[:-1])
zc=counts
print('area under curve: ',trapz(zc, z))

popt,pcov = curve_fit(gauss,z,zc,p0=[np.max(zc),np.mean(z),1.0])

a_fit, mean_fit, sigma_fit = popt
print(a_fit, mean_fit, np.abs(sigma_fit))
dofdz=3+(1.0/(np.sqrt(2)*sigma_fit))**2
print('DOF: ',dofdz)

#plt.scatter(z,zc,label='$\Delta z$', color='y')
plt.plot(z,gauss(z,*popt),'k',label='Gaussian fit',
         lw=3.0, alpha=0.95)
plt.xlim(0.0,0.06)
plt.legend()
plt.show()
```

```
area under curve:  0.9999999999999999
152.89324508417772 0.01616606770568351 0.0025878698651480166
DOF:  74662.51038720872
```



The estimated DOF from the z difference is unrealistically too large because the two distributions from which the z difference was obtained are not independent. Thus we *cannot* simply apply $\sigma^2_{diff} = \sigma^2_0 + \sigma^2_1 = 2\sigma^2_m$ to estimate the mean sigma σ_m of the initial from the distribution of the differences σ_{diff}. The sigma of the z difference is much smaller than expected for independent random z_0 and z_1.

If we *assume* that the fitted plot above is the true frequency distribution, clearly all or almost all experimentally observed values are away from 0.0, which would allow us to reject the null hypothesis at the 99.9...% level.

The problem is that we do not know the theoretical distribution beforehand, because our data points (image pixel intensities) are not independent and we do not know the "effective DOF" in a specific Kikuchi pattern. Otherwise we can estimate the theoretical distribution of the NCC values and also their difference from the DOF, like shown above. Once an estimation for the effective DOF in a Kikuchi pattern is avalaible, the usual testing scenarios are straightforward.

Since we compare the same experimental pattern with two theories that are also correlated, the change in NCC between the two comparison is smaller than would be expected for two independent theoretical patterns (i.e. the two theoretical patterns look very similar also).

To be mathematically correct, the "spatial" correlation between the image pixels, as well as the mutual correlation of the simulated theory-patterns need to be included somehow in the test scenario. We can know the correlation between the theoretical patterns, but we also have to estimate the spatial correlation in each of the theoretical patterns. (Hoteling, Schneider, see D.C. Howell, "Statistical Methods for Psychology").

Application as an Image Similarity Measure

The NCC (a.k.a. Pearson correlation coefficient, r) has several useful properties for the quantitative comparison of EBSD patterns.

- normalized patterns on well defined scale (mean=0.0 and standard deviation=1.0)
- inversion of contrast is trivial: multiply the normalized pattern by -1

A number of similarity and dissimilarity measures for images are discussed, for example, in A. Goshtasby "Image Registration" (Springer, 2012)[11], who summarizes the question of choosing a similarity/dissimilarity measure for photographic images (p.57):

> Each similarity/dissimilarity measure has its strengths and weaknesses. A measure that performs well on one type of images may perform poorly on another type of images.

[11] http://www.springer.com/de/book/9781447124573

> Therefore, an absolute conclusion cannot be reached about the superiority of one measure against another. However, the experimental results obtained on various image types and various image differences reveal that Pearson correlation coefficient, Tanimoto measure, minimum ratio, L_1 norm, square L_2 norm, and intensity ratio variance overall perform better than other measures. If the images are captured under different exposures of a camera or under different lighting of a scene, the results show that Pearson correlation coefficient, Tanimoto measure, normalized square L_2 norm, and incremental sign distance perform better than others.

The **Normalized Inner Product** (NIP) (also called **Normalized Dot Product**) which is not discusssed by Gotashby in the reference above, has also been used as a similarity measure for pattern matching and indexing of EBSD and ECP under various experimental conditions and pattern qualities, and including a corresponding NIP-based error analysis of orientation determination and phase discrimination. See for example:

- [1] Y.H. Chen et al., *A Dictionary Approach to Electron Backscatter Diffraction Indexing*, Microscopy and Microanalysis 21 (2015) 739[12]
- [2] S. Singh, M. De Graef, *Dictionary Indexing of Electron Channeling Patterns.* Microscopy and Microanalysis 23 (2017) 1[13]
- [3] S. Singh, F. Ram and M. De Graef , *Application of forward models to crystal orientation refinement*, J. Appl. Cryst. (2017) 50[14]
- [4] F. Ram, S. Wright, S. Singh, and M. De Graef , *Error analysis of the crystal orientations obtained by the dictionary approach to EBSD indexing*, Ultramicroscopy 181 (2017) 17[15]
- [5] K. Marquardt, M. De Graef, S. Singh, H. Marquardt, A. Rosenthal, S. Koizuimi *Quantitative electron backscatter diffraction (EBSD) data analyses using the dictionary indexing (DI) approach: Overcoming indexing difficulties on geological materials* American Mineralogist 102 (2017) 1843[16]
- [6] F. Ram and M. De Graef , *Phase differentiation by electron backscatter diffraction using the dictionary indexing approach*, Acta Materialia 144 (2018) 352[17]
- [7] S. Singh, Y. Guo, B. Winiarski, T. L. Burnett, P. J. Withers, M. De Graef *High resolution low kV EBSD of heavily deformed and nanocrystalline Aluminium by dictionary-based indexing* Scientific Reports 8 (2018) 10991[18]

In the following, we will demonstrate some key differences between the NCC and the NIP, which is defined according to [1,4] as:

$$\rho = \frac{\langle \mathbf{X}, \mathbf{Y} \rangle}{||\mathbf{X}|| \cdot ||\mathbf{Y}||}$$

with $\langle \mathbf{X}, \mathbf{Y} \rangle$ being the dot product (inner product) of the image vectors $\mathbf{X}$ and $\mathbf{Y}$.

Note that, compared to the NCC, the NIP does neither involve the removal of the mean from the image nor does it involve a normalization according to the image "energy" (standard deviation = 1.0). These features will make the NIP much less useful as an image similarity measure when images are compared which vary in intensity and mean level. We will also show the different reaction of the NCC and NIP when comparing experimental data to pure noise, which will show the the NIP is an unreliable predictor of *vanishing* agreement.

[30]:
```
def norm_dot(img1, img2):
    """
    return normalized dot product of the arrays img1, img2
```

(continues on next page)

[12] https://doi.org/10.1017/S1431927615000756
[13] https://doi.org/%2010.1017/S1431927616012769
[14] https://doi.org/10.1107/S1600576717014200
[15] https://doi.org/10.1016/j.ultramic.2017.04.016
[16] https://doi.org/10.2138/am-2017-6062
[17] https://doi.org/10.1016/j.actamat.2017.10.069
[18] https://doi.org/10.1038/s41598-018-29315-8

(continued from previous page)

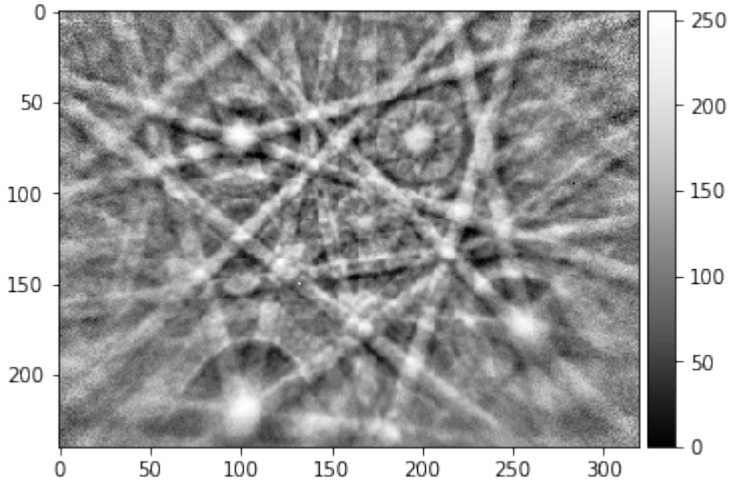
```
    """
    # make 1D value lists
    v1 = np.ravel(img1)
    v2 = np.ravel(img2)

    # get the norms of the vectors
    norm1 = np.linalg.norm(v1)
    norm2 = np.linalg.norm(v2)
    #print('norms of NDP vectors: ', norm1, norm2)

    ndot = np.dot( v1/norm1, v2/norm2)
    return ndot
```

To get a feeling for the typical relative values for a low-noise experimental image and a suffciently good simulation, we compare the NCC and the NIP of two images:

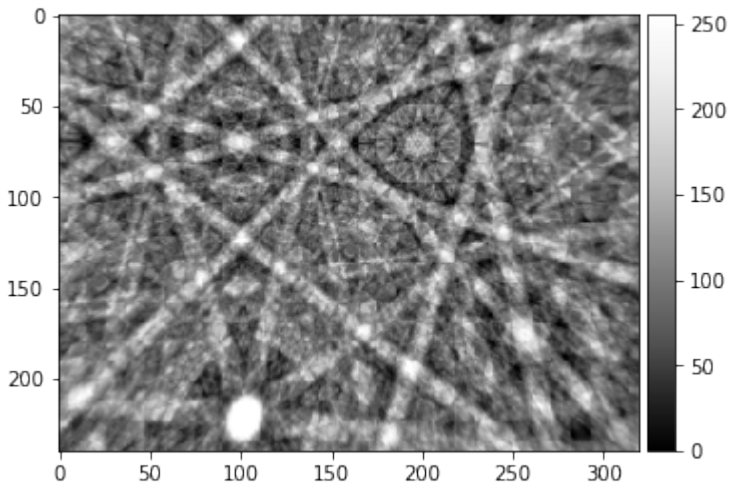
[31]:
```
def print_similarity_measures(img1, img2, nc0=None, nd0=None):
    nd = norm_dot(img1, img2)
    nc = ncc(img1, img2)
    print('NCC: ', nc, ' NDP: ', nd)
    if not ((nc0 is None) or (nd0 is None)):
        print('dNCC: ', nc-nc0, ' dNDP: ', nd-nd0)
    return
```

As a first test, we check the similarity of an image with itself, which should result in a value of 1.0 in both cases:

[32]:
```
img_exp = np.array(Image.open('./data/ocean_grain_final.png').convert(mode='L'), dtype=np.
↪float64)

plot_image(img_exp)
print(img_exp.shape, img_exp.dtype)
print_similarity_measures(img_exp, img_exp)

img_sim = np.array(Image.open('./data/ocean_dynsim.png').convert(mode='L'), dtype=np.
↪float64)
plot_image(img_sim)
print(img_sim.shape, img_sim.dtype)
print_similarity_measures(img_sim, img_sim)
```




```
(240, 320) float64
NCC:  0.9999999999999997  NDP:  1.0
```




```
(240, 320) float64
NCC:  1.0000000000000002  NDP:  1.0
```

We now check the similarity between the experimental pattern and the simulated pattern, and obtain a NCC near 0.7, which usually indicates a very good fit; the relevant NIP is 0.966 for the two loaded images:

[33]:
```
# initial images
test_exp = img_exp
test_sim = img_sim
ndot_0 = norm_dot(test_exp, test_sim)
ncc_0  = ncc(test_exp,test_sim)
print('NCC: ', ncc_0)
print('NIP: ', ndot_0)
```

```
NCC:  0.6934039295128853
NIP:  0.966085369229993
```

[34]:
```
# scale both: the ncc and ndp stay at their initial values
test_exp = 2.0*img_exp
test_sim = 2.0*img_sim
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128853  NDP:  0.966085369229993
```

[35]:
```
# scale both differently: the ncc and ndp stay at their initial values
test_exp = 2.0*img_exp
test_sim = 5.0*img_sim
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128853  NDP:  0.9660853692299929
```

[36]:
```
# offset
test_exp = img_exp+100
test_sim = img_sim+100
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128853  NDP:  0.9903671259223557
```

An offset which is large enough will drive the NDP towards 1.0, because the relative variations in the image vector length due to the image intensity variations will become neglible:

[37]:
```
test_exp = img_exp+100000
test_sim = img_sim+100000
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128853  NDP:  0.9999999579164437
```

[38]:
```
test_exp = img_exp-100
test_sim = img_sim
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128853  NDP:  0.6221259676934416
```

[39]:
```
test_exp = img_exp
test_sim = img_sim-1000
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128853  NDP:  -0.9477149135723966
```

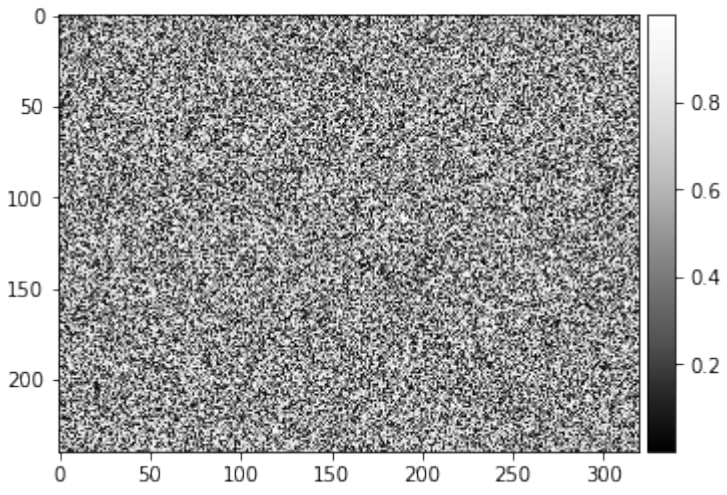
[40]:
```
test_exp = -1.0*img_exp+8000
test_sim = -1.0*img_sim-1000
print_similarity_measures(test_exp, test_sim)
```

```
NCC:  0.6934039295128855  NDP:  -0.9992710340851816
```

Comparison of Random Images

For checking the behaviour of the image simalrity for totally random images, we create images with uniformly distributed random float values from 0 to 1 and then calculate the NCC and NIP. As a first check, we make sure that the NCC and NIP of random noise with itself is also 1.0:

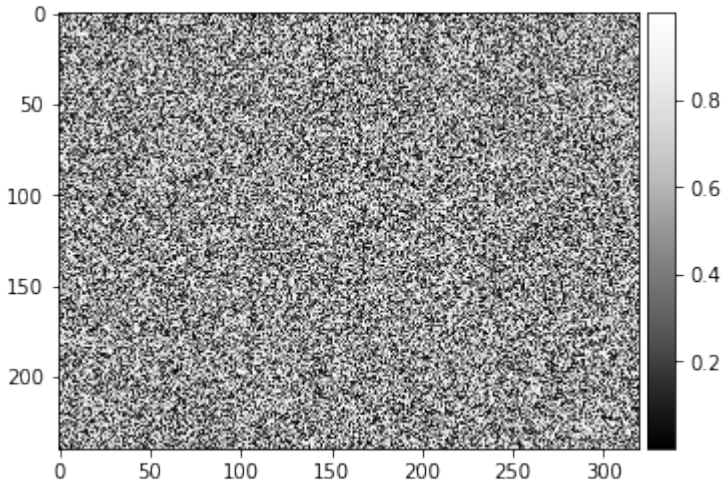
[41]:
```
img_random = np.random.rand(img_exp.shape[0], img_exp.shape[1])
plot_image(img_random)
print_similarity_measures(img_random, img_random)
```

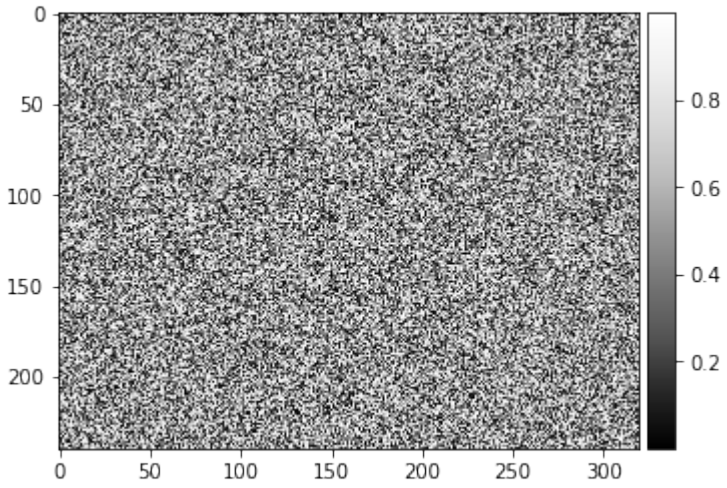



```
NCC:  0.9999999999999997  NDP:  1.0
```

We expect that two different random images should be completely dissimilar, which should be reflected in the values of the NCC and NIP, which should be different from 1.0. The calculation for two different random images gives a value near 0.0 for the NCC, which is consistent with our sense of similarity. However, we obtain but a value near 0.75 for the NIP, so this indicates that "complete dissimilarity" is signalled by a value of the NIP near 3/4 (for an explanation, where this value comes from, see below):

[42]:
```
# random images
img1_random = np.random.rand(img_exp.shape[0], img_exp.shape[1])
img2_random = np.random.rand(img_exp.shape[0], img_exp.shape[1])

plot_image(img1_random)
plot_image(img2_random)

print_similarity_measures(img1_random, img2_random)
```






```
NCC:  -0.0052766729699176345  NDP:  0.7487770197475101
```

[43]:
```
nruns=10000

def compare_random_images(nruns, offset=0.0, scale=1.0):
    nc_list=np.zeros(nruns, dtype=np.float32)
    nd_list=np.zeros(nruns, dtype=np.float32)
    for i in range(nruns):
        # 0..1 float random numbers
        img1_random = np.random.rand(img_exp.shape[0], img_exp.shape[1])
        img2_random = np.random.rand(img_exp.shape[0], img_exp.shape[1])

        # note: difference for images 0..1 values as compared -1,1
        test_exp =  offset + scale * img1_random
        test_sim =  offset + scale * img2_random
```

(continues on next page)

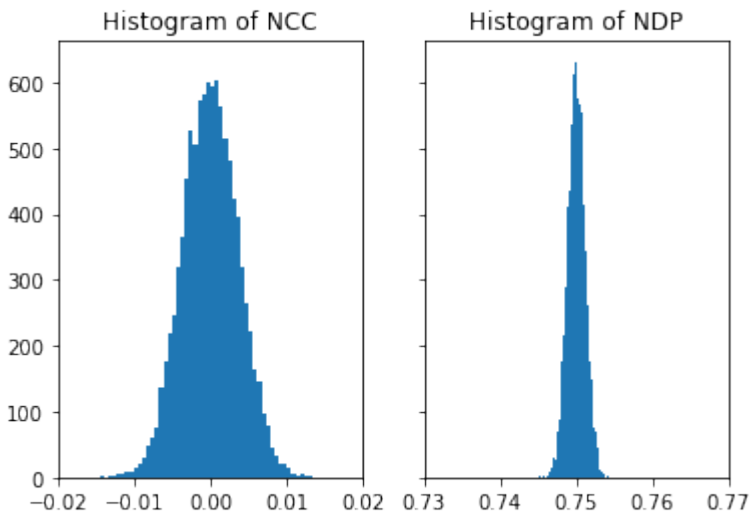
(continued from previous page)

```
        nc_list[i] = ncc(test_exp, test_sim)
        nd_list[i] = norm_dot(test_exp, test_sim)
    return nc_list, nd_list

nc_vals, nd_vals = compare_random_images(nruns)
```

When normalizing each image with 8bit intensities from 0..255 (or 0..65535 for 16bit), the resulting (random) unit image vectors reside only in one quadrant of the high-dimensional sphere so we obtain a value of 3/4 for the expection value of the NDP, not zero like for the NCC.

see, e.g. a discussion here: https://math.stackexchange.com/questions/2422001/expected-dot-product-of-two-random-vectors

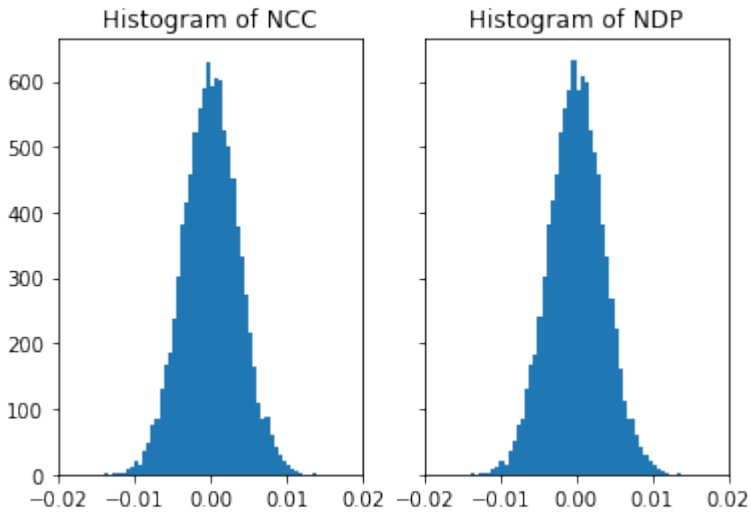
[44]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.hist(nc_vals, bins=50)
ax1.set_title('Histogram of NCC')
ax1.set_xlim([-0.02, 0.02])
ax2.hist(nd_vals, bins=50)
ax2.set_title('Histogram of NDP')
ax2.set_xlim([-0.02+0.75, 0.02+0.75])
plt.show()
```



If we use a symmetrical range around 0.0 for the image entries, we effectively calculate the NCC, and thus NDP=NCC in this limit:

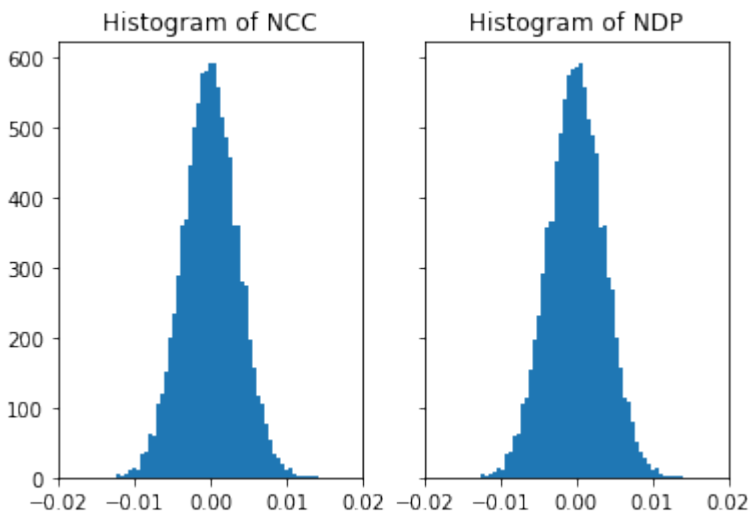
[45]:
```
# this is for vectors with entries -1...1
nc_vals, nd_vals = compare_random_images(nruns, offset=-1.0, scale=2.0)
```

[46]:
```
# now we have the same histogram for both (i.e. ncc=ndp in this limit, since we have the␣
↪mean=0.0)
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.hist(nc_vals, bins=50)
ax1.set_title('Histogram of NCC')
ax1.set_xlim([-0.02, 0.02])
ax2.hist(nd_vals, bins=50)
ax2.set_title('Histogram of NDP')
ax2.set_xlim([-0.02, 0.02])
plt.show()
```



[47]:
```
# this is for vectors with entries -0.5...0.5
nc_vals, nd_vals = compare_random_images(nruns, offset=-0.5, scale=1.0)
```

[48]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.hist(nc_vals, bins=50)
ax1.set_title('Histogram of NCC')
ax1.set_xlim([-0.02, 0.02])
ax2.hist(nd_vals, bins=50)
ax2.set_title('Histogram of NDP')
ax2.set_xlim([-0.02, 0.02])
plt.show()
```

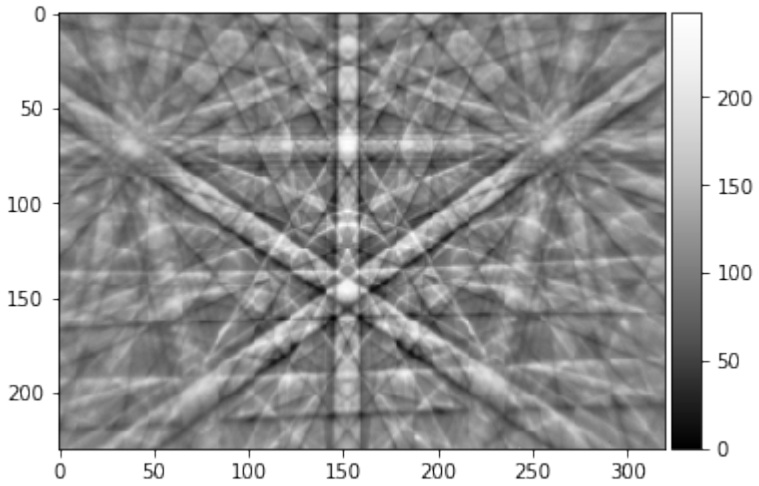


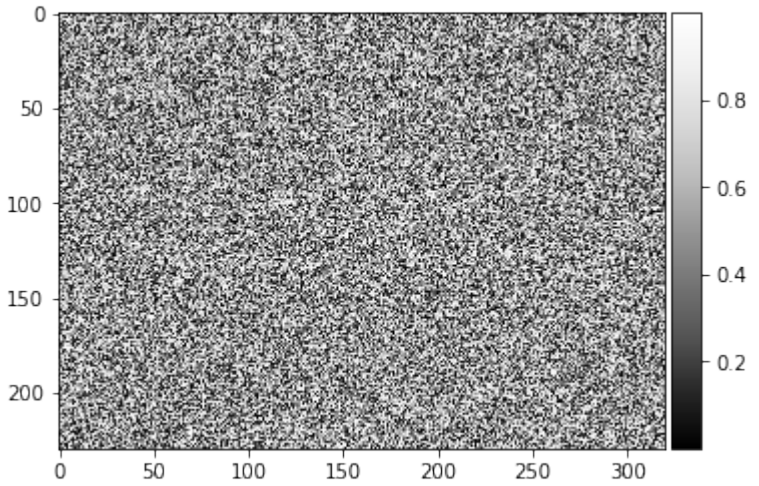
Comparison of Kikuchi Patterns to Random Noise

In this section, we will show the different reaction of the NCC and NIP when comparing experimental data to pure noise. Compared to the NCC, the the NIP is an unreliable predictor of *vanishing* agreement with a test pattern because different Kikuchi patterns show a different NIP distribution when compared to the same type of noise. In contrast, the NCC values are centered around 0.0 for the test patterns when compared to pure noise.

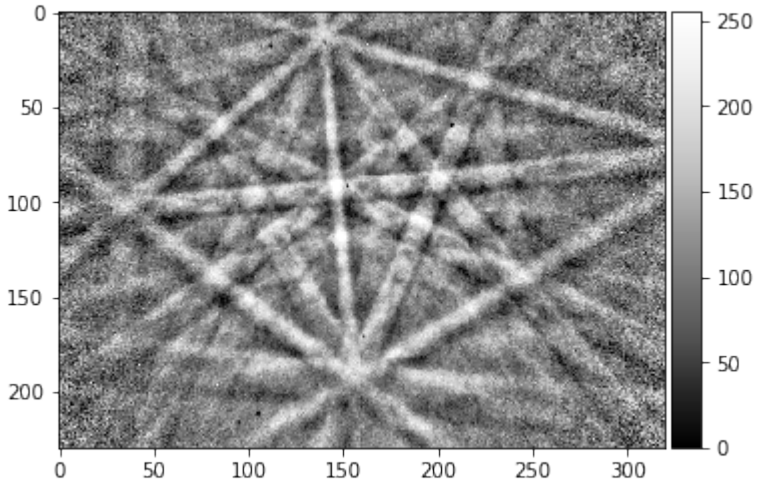
[49]:
```
img_exp_1 = np.array(Image.open('./data/Si_15000_BAM_320.png').convert(mode='L'), dtype=np.
↪uint8)
img_exp_2 = np.array(Image.open('./data/ocean_grain_final.png').convert(mode='L'),␣
↪dtype=np.uint8)
img_exp_3 = imread("../data/patterns/Ni_Example3.png")
print(img_exp_1.shape, img_exp_1.dtype)
print(img_exp_2.shape, img_exp_2.dtype)
print(img_exp_3.shape, img_exp_3.dtype)

# pure noise! uniform [0..1]
img_noise = np.random.rand(img_exp_1.shape[0], img_exp_1.shape[1])

plot_image(img_exp_1)
plot_image(img_noise)
print_similarity_measures(img_exp_1, img_noise)

plot_image(img_exp_3)
plot_image(img_noise)
print_similarity_measures(img_exp_3, img_noise)
```

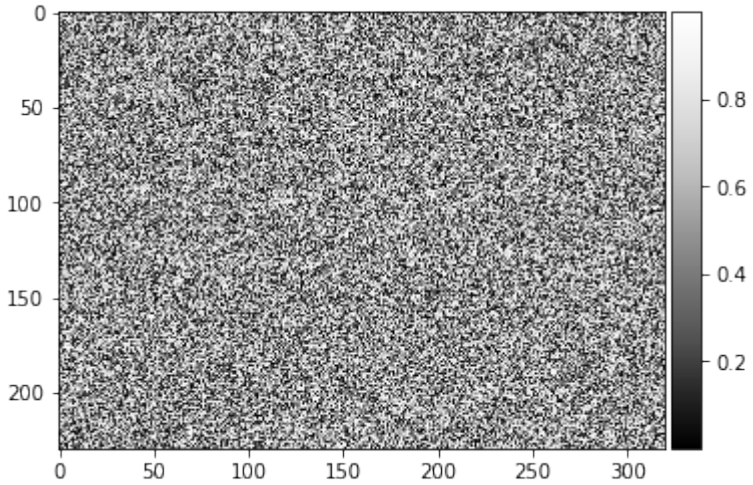
```
(230, 320) uint8
(240, 320) uint8
(230, 320) uint8
```






```
NCC:  -0.006287200297652947  NDP:  0.8397218358602418
```






```
NCC:  -0.003651187369326606  NDP:  0.8156612789563865
```

The higher NIP value of the first, higher quality, pattern (NIP=0.84) seems to indicate a better agreement with noise than the second image (NIP=0.81). It is highly counter-intuitive that extactly the same pure noise has a higher similarity to the better image, as compared to a lower similarity of the pure noise with a more noisy pattern. In contrast, the NCC is practically near 0.0 for both images, indicating that none of both image is in some way more similar to the noise.

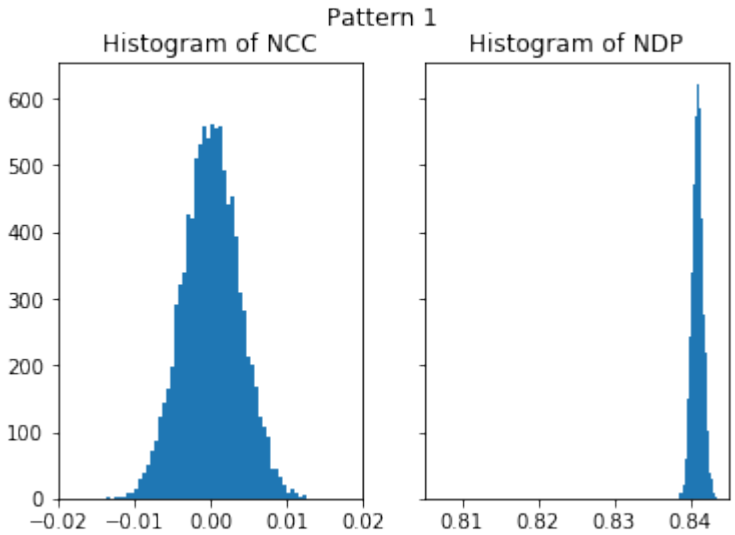
[50]:
```
nruns=10000

def compare_kik_noise(nruns, img_kik, offset=0.0, scale=1.0):
    """ compare Kikuchi pattern with noise
    """
    nc_list=np.zeros(nruns, dtype=np.float32)
    nd_list=np.zeros(nruns, dtype=np.float32)
    for i in range(nruns):
        # 0..1 float random numbers
        img_noise = np.random.rand(img_kik.shape[0], img_kik.shape[1])

        # note: difference for images 0..1 values as compared -1,1
        test_kik   =  offset + scale * img_kik
        test_noise =  offset + scale * img_noise

        nc_list[i] = ncc(test_kik, test_noise)
        nd_list[i] = norm_dot(test_kik, test_noise)
```

(continues on next page)

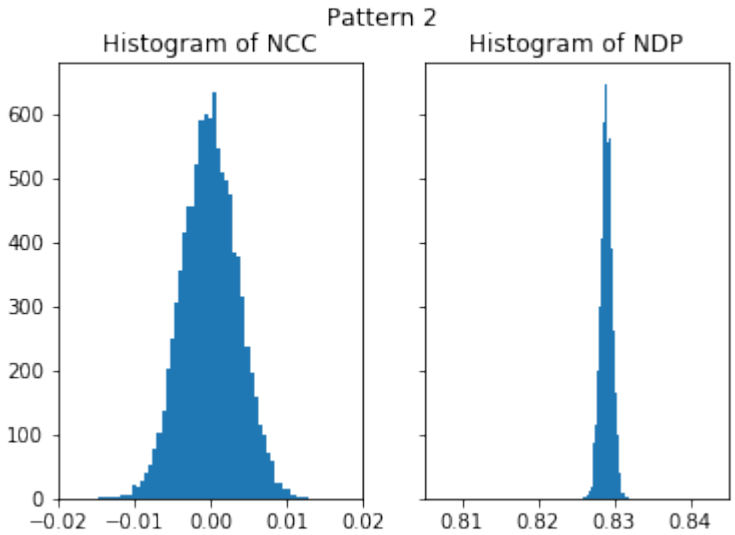
(continued from previous page)

```
    return nc_list, nd_list

def plot_histograms(nc_vals, nd_vals, title):
    fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
    ax1.hist(nc_vals, bins=50)
    ax1.set_title('Histogram of NCC')
    ax1.set_xlim([-0.02, 0.02])
    ax2.hist(nd_vals, bins=50)
    ax2.set_title('Histogram of NDP')
    ax2.set_xlim([-0.02+0.825, 0.02+0.825])
    plt.suptitle(title)
    plt.show()
```

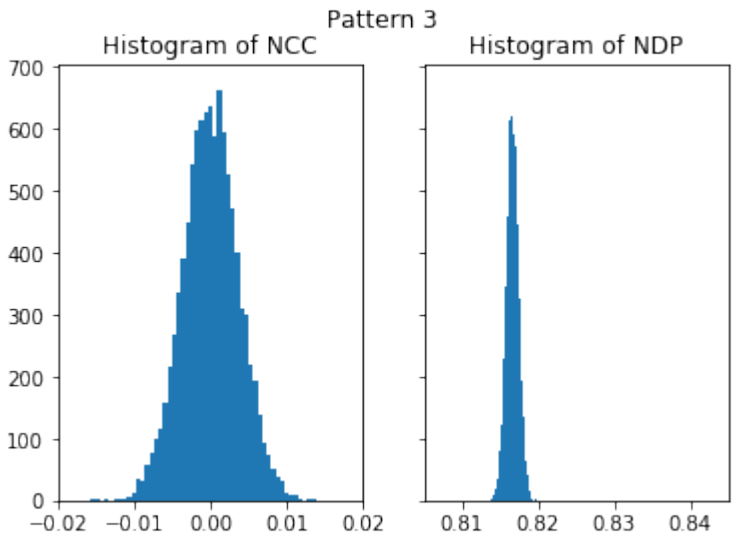
[51]:
```
nc_vals_1, nd_vals_1 = compare_kik_noise(nruns, img_exp_1)
plot_histograms(nc_vals_1, nd_vals_1, 'Pattern 1')
```



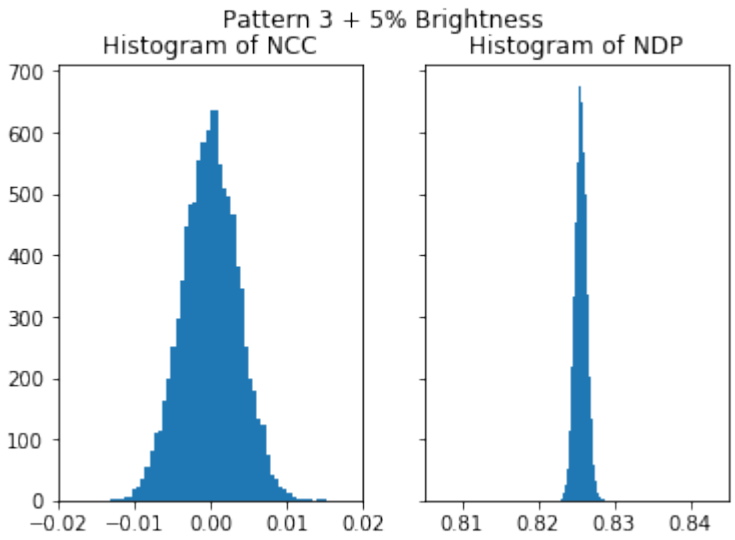
[52]:
```
nc_vals_2, nd_vals_2 = compare_kik_noise(nruns, img_exp_2)
plot_histograms(nc_vals_2, nd_vals_2, 'Pattern 2')
```



[53]:
```
nc_vals_3, nd_vals_3 = compare_kik_noise(nruns, img_exp_3)
plot_histograms(nc_vals_3, nd_vals_3, 'Pattern 3')
```

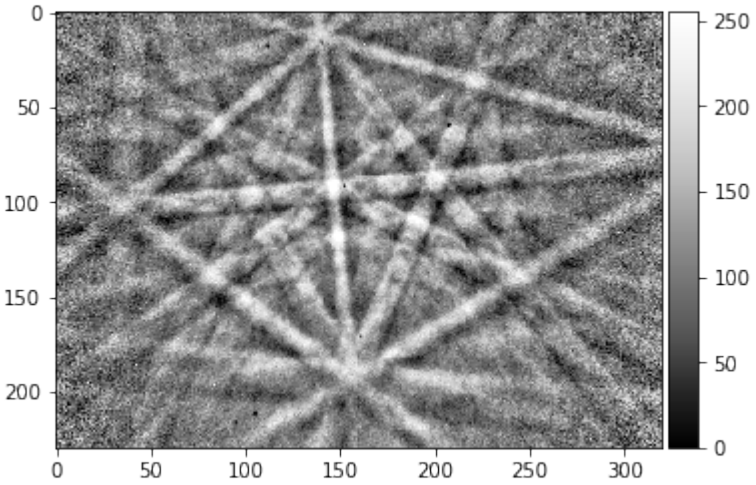


[54]:
```
nc_vals_4, nd_vals_4 = compare_kik_noise(nruns, img_exp_3 + 0.05*255)
plot_histograms(nc_vals_4, nd_vals_4, 'Pattern 3 + 5% Brightness')
```

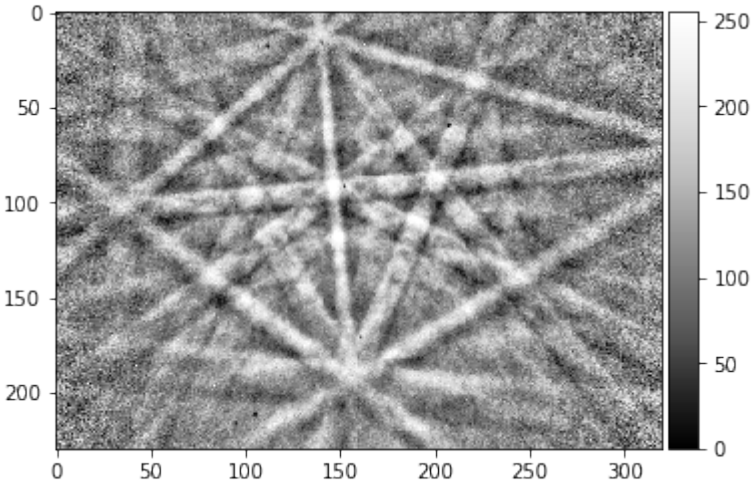


To illustrate the effect of 5% additional brightness:

[55]:
```
#original pattern data
plot_image(img_exp_3, limits=[0,255])
```



[56]:
```
# pattern gray levels upshifted by 5%
plot_image(img_exp_3 + 0.05*255, limits=[0,255])
```



Summary: Similarity Measure

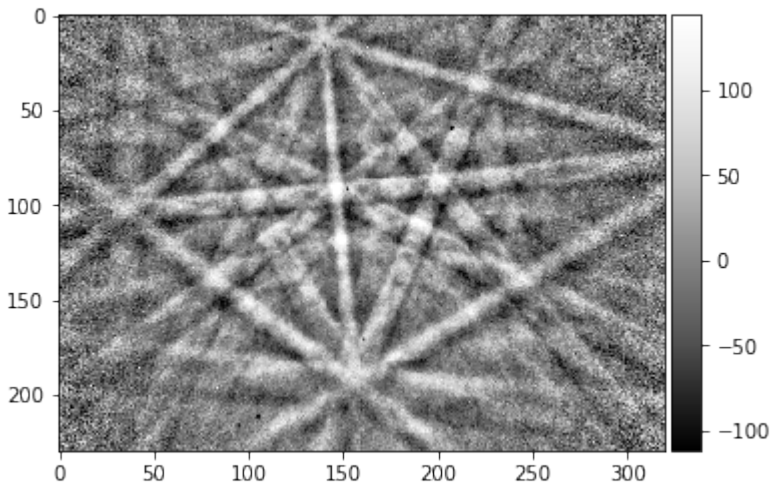
The NCC is stable under changes of brightness and contrast, the NIP shows properties which can make its use highly unreliable for a comparison of patterns which have been obtained under varying conditions.

Equivalence of FFT convolution and Normalized Cross Correlation Coefficient

principle: the signals must be periodic and thus only shifted in the "unit cell", i.e. sine or cosine must be cut at exactly the same places and padded with zeros

See also: https://stackoverflow.com/questions/46457866/how-do-i-scale-an-fft-based-cross-correlation-such-that-its-

[57]:
```
kiku_exp = imread("../data/patterns/Ni_Example3.png")
kiku_exp = kiku_exp - np.nanmean(kiku_exp)
plot_image(kiku_exp)
motif_h, motif_w = kiku_exp.shape
kiku_exp.shape
```

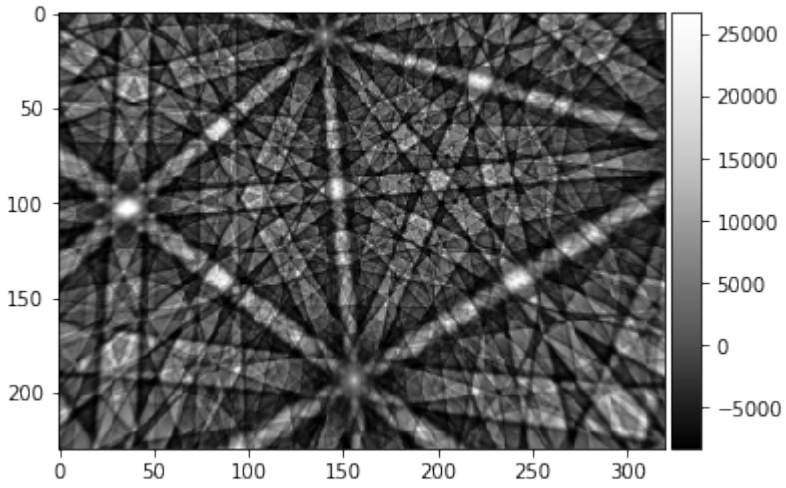


[57]:
```
(230, 320)
```

For comparison, we load a simulated pattern. To take into account that we do not know the absolute scale of experiment relative to the theory, we scale the theory by an arbitrary factor.

Because we subtract the mean from the experiment and the simulation, we now have now two signals, which vary around a mean value of zero.

[58]:
```
kiku_sim = 137.314 * (imread("../data/patterns/Ni_Example3_sim.png", asgray=True)[:,:,0]).
↪astype(np.float32)
kiku_sim = kiku_sim - np.nanmean(kiku_sim)
plot_image(kiku_sim)
kiku_sim.shape
```



[58]:
```
(230, 320)
```

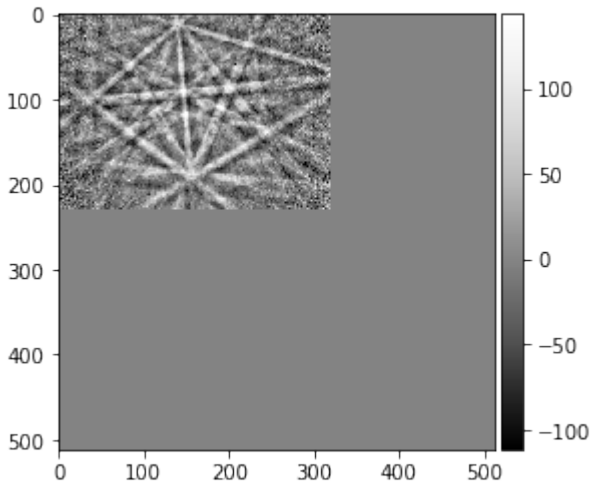
For reference, we calculate the Pearson normalized cross correlation coefficient, like defined at the top of the notebook:

[59]:
```
r_ncc = ncc(kiku_exp, kiku_sim)
print(r_ncc)
```

```
0.5401485802133111
```

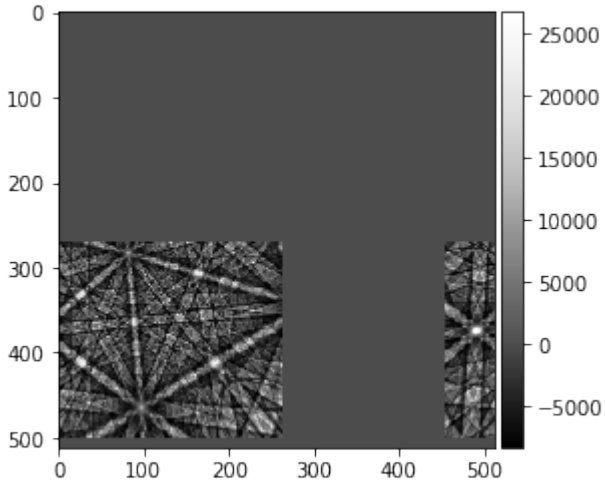
[60]:
```
def fill_fft_cell(image, shift, cell_size=512):
    cell = np.zeros((cell_size, cell_size), dtype=np.float32)
    shift_row = shift[0] % cell_size
    shift_col = shift[1] % cell_size
    for r in range(shift_row, shift_row + image.shape[0]):
        for c in range(shift_col, shift_col + image.shape[1]):
            cell[r % cell_size, c % cell_size] = image[r-shift_row, c-shift_col]

    return cell
```

[61]:
```
reference = fill_fft_cell(kiku_exp, [0, 0])
plot_image(reference)
```



[62]:
```
# note that this works also for "backfolding" images! FFT periodicity!
shift_row = np.random.randint(0, 512)
shift_col = np.random.randint(0, 512)
print("known shift of simulation (row, col): ", shift_row, shift_col)

shifted_sim = fill_fft_cell(kiku_sim, [shift_row, shift_col])
plot_image(shifted_sim)
```

```
known shift of simulation (row, col):  271 454
```

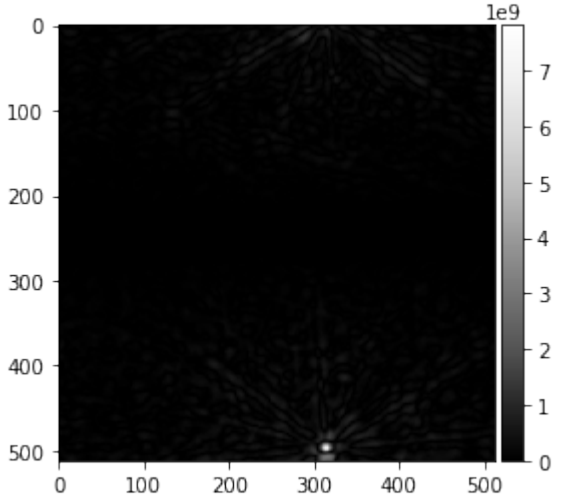


[63]:
```
image1 = reference
```

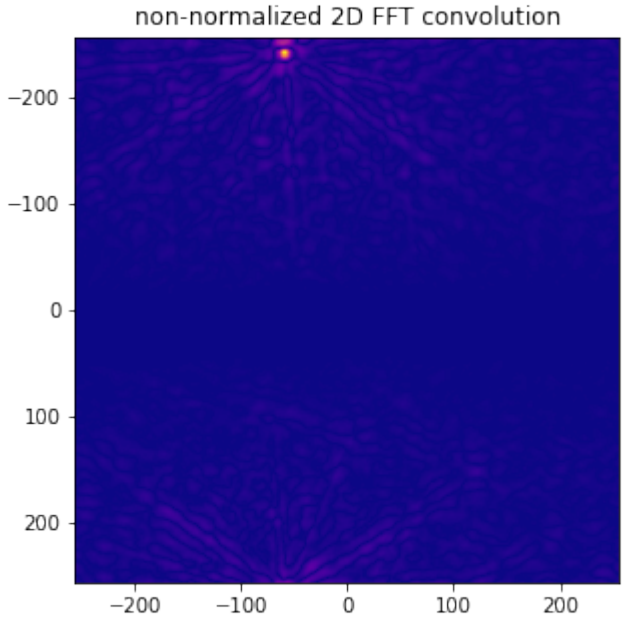
(continues on next page)

(continued from previous page)

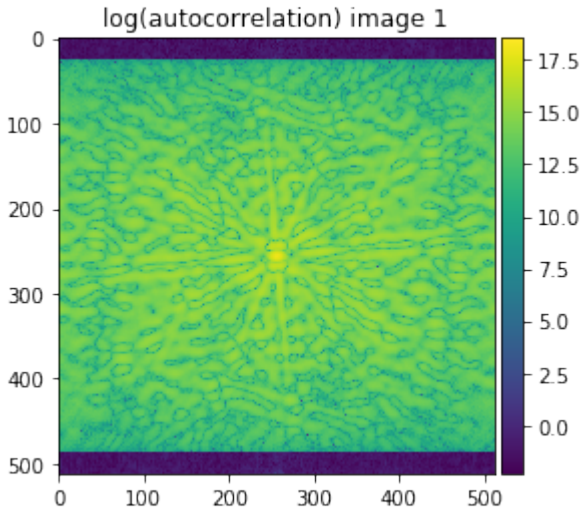
```
image2 = shifted_sim
M, N = image1.shape

# fftshift puts the zero frequencies to the middle of the array
xc_fft = np.fft.fftshift(np.abs( np.fft.ifft2( np.fft.fft2(image1) * np.fft.fft2(image2).
↪conjugate()) ))
plot_image(xc_fft)

f, ax = plt.subplots(figsize=(5, 5))
ax.imshow(np.flipud(np.fliplr(xc_fft)), cmap='plasma',
          extent=(-N // 2, +N // 2, M // 2, -M // 2))
ax.set_title('non-normalized 2D FFT convolution');

max_xcfft = np.unravel_index(np.argmax(xc_fft, axis=None), xc_fft.shape)
row_max = (256 - max_xcfft[0]) % 512
col_max = (256 - max_xcfft[1]) % 512
print ('position of maximum ( rows, cols, [0,0] at center, fftshift): ', row_max, col_max)
print ('xc fft max', np.max(xc_fft))
```

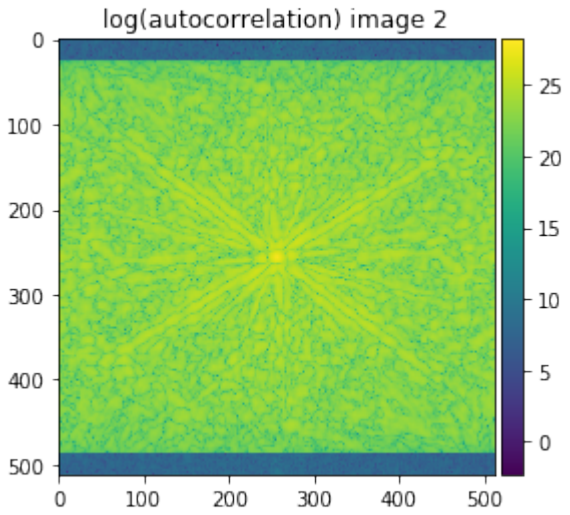



```
position of maximum ( rows, cols, [0,0] at center, fftshift):  271 454
xc fft max 7820401700.0
```

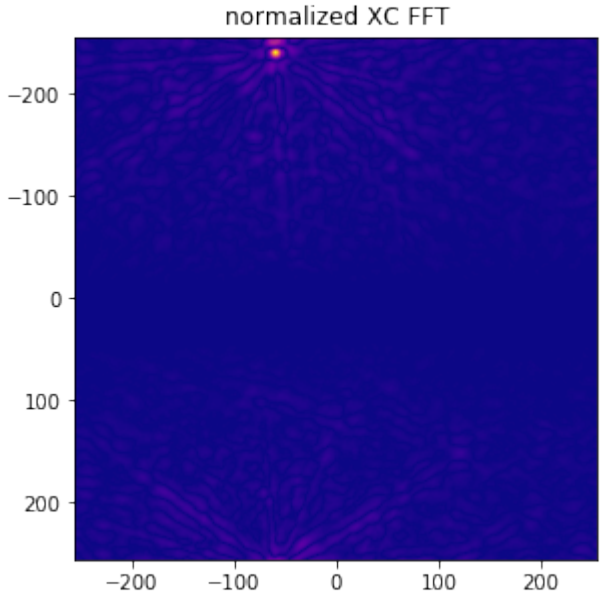


[64]:
```
autocorrelation_1 = np.fft.fftshift(np.abs(np.fft.ifft2(np.fft.fftn(image1)*np.fft.
↪fft2(image1).conjugate())))
autocorrelation_2 = np.fft.fftshift(np.abs(np.fft.ifft2(np.fft.fftn(image2)*np.fft.
↪fft2(image2).conjugate())))
plot_image(np.log(autocorrelation_1+0.1), title='log(autocorrelation) image 1', cmap=
↪'viridis')
plot_image(np.log(autocorrelation_2+0.1), title='log(autocorrelation) image 2', cmap=
↪'viridis')
```





[65]:
```
intensity_1 = np.max(autocorrelation_1)
intensity_2 = np.max(autocorrelation_2)
xc_denom = np.sqrt(intensity_1)* np.sqrt(intensity_2)

xc_normalized = xc_fft/xc_denom

f, ax = plt.subplots(figsize=(4.8, 4.8))
# set extent to have axes with shift value (0,0) at center and shifting corresponding␣
↪visually to images
# (only ++ quadrant is used for the known shift)
ax.imshow(np.flipud(np.fliplr(xc_normalized)), cmap='plasma',
          extent=(-N // 2, N // 2, M // 2, -M // 2))
ax.set_title('normalized XC FFT');
```



We can now find the position of the maximum and the corresponding value of xc_normalied, which should correspond to the cross correlation coefficient reference value r_ncc as calculated above:

[66]:
```
# indices of maxmimum in 2D
max_rc = np.unravel_index(np.argmax(xc_normalized, axis=None), xc_normalized.shape)
# maximum value shoould be at theoretical shift vector
```

(continues on next page)

(continued from previous page)

```
r_fft = np.max(xc_normalized)


print('Compare results (should be equal):\n')
print('known shift vector: ', shift_row, shift_col)

print('position of found r_fft maximum: ', (256 - max_rc[0]) % 512 , (256 - max_rc[1]) %␣
↪512)
print('r_fft: ', r_fft)
print('r_ncc: ', r_ncc)

# throw error if not equal up to 6 decimals
np.testing.assert_almost_equal(r_fft, r_ncc, 6)
```

```
Compare results (should be equal):

known shift vector:  271 454
position of found r_fft maximum:  271 454
r_fft:  0.54014856
r_ncc:  0.5401485802133111
```

We have thus shown how to obtain the same cross-correlation coefficient as r_ncc by (a) normalization (mean=0, stddev=1.0) of the input images and then padding by zeroes inside a common 2D array size, and (b) the suitable scaling of the FFT by the maximum of the autocorrelations of both images (i.e. the energy in both images). This demonstrates the intrinsic similarity of r_fft (determined by FFT) and r_ncc (determined by pixel-wise formula for the normalized cross-correlation coeffcient).

Subpixel resolution

http://scikit-image.org/docs/dev/auto_examples/transform/plot_register_translation.html

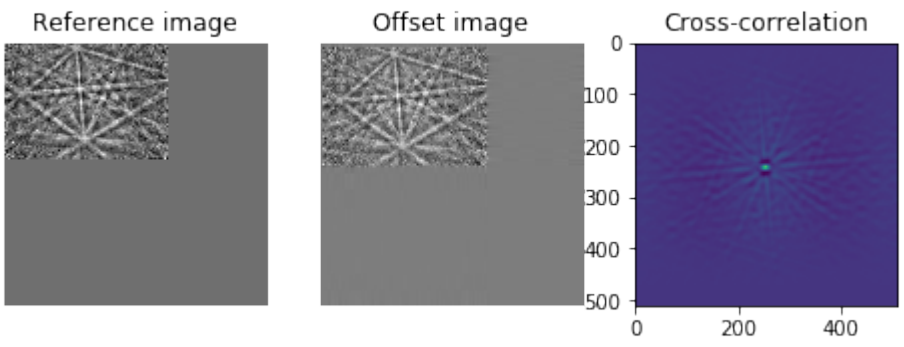
[67]:
```
from skimage import data
from skimage.feature import register_translation
from skimage.feature.register_translation import _upsampled_dft
from scipy.ndimage import fourier_shift

image = image1
shift = (12.4, 3.32)
# The shift corresponds to the pixel offset relative to the reference image
offset_image = fourier_shift(np.fft.fft2(image), shift)
offset_image = np.fft.ifft2(offset_image)
print("Known offset (y, x): {}".format(shift))

# pixel precision first
shift, error, diffphase = register_translation(image, offset_image)
print("Detected pixel offset (y, x): {}".format(shift))

fig = plt.figure(figsize=(8, 3))
ax1 = plt.subplot(1, 3, 1)
ax2 = plt.subplot(1, 3, 2, sharex=ax1, sharey=ax1)
ax3 = plt.subplot(1, 3, 3)

ax1.imshow(image, cmap='gray')
ax1.set_axis_off()
ax1.set_title('Reference image')

ax2.imshow(offset_image.real, cmap='gray')
```

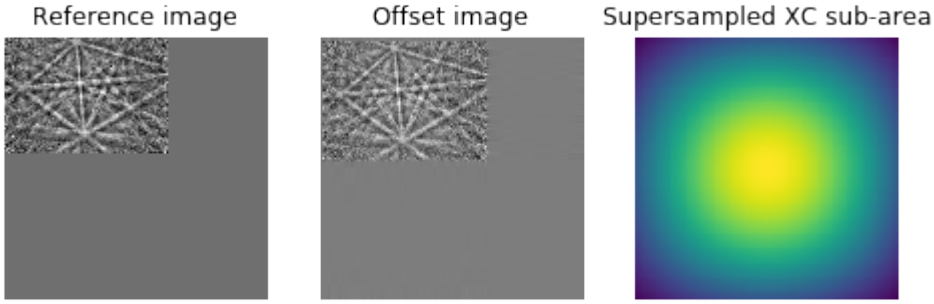
(continues on next page)

(continued from previous page)

```
ax2.set_axis_off()
ax2.set_title('Offset image')

# Show the output of a cross-correlation to show what the algorithm is
# doing behind the scenes
image_product = np.fft.fft2(image) * np.fft.fft2(offset_image).conj()
cc_image = np.fft.fftshift(np.fft.ifft2(image_product))
ax3.imshow(cc_image.real)
#ax3.set_axis_off()
ax3.set_title("Cross-correlation")

plt.show()


# subpixel precision
shift, error, diffphase = register_translation(image, offset_image, 200)

fig = plt.figure(figsize=(8, 3))
ax1 = plt.subplot(1, 3, 1)
ax2 = plt.subplot(1, 3, 2, sharex=ax1, sharey=ax1)
ax3 = plt.subplot(1, 3, 3)

ax1.imshow(image, cmap='gray')
ax1.set_axis_off()
ax1.set_title('Reference image')

ax2.imshow(offset_image.real, cmap='gray')
ax2.set_axis_off()
ax2.set_title('Offset image')

# Calculate the upsampled DFT, again to show what the algorithm is doing
# behind the scenes.  Constants correspond to calculated values in routine.
# See source code for details.
cc_image = _upsampled_dft(image_product, 150, 100, (shift*100)+75).conj()
ax3.imshow(cc_image.real)
ax3.set_axis_off()
ax3.set_title("Supersampled XC sub-area")


plt.show()

print("Detected subpixel offset (y, x): {}".format(shift))
```

```
Known offset (y, x): (12.4, 3.32)
Detected pixel offset (y, x): [-12.  -3.]
```






```
Detected subpixel offset (y, x): [-12.4   -3.32]
```

Appendix

Numpy corrcoef

We can also use numpy directly to get the NCCs for all combinations of several data sets In the returned matrix of numpy.corrcoef[19], the i-j element gives the NCC of dataset i with dataset j.

For an example, see also: https://stackoverflow.com/questions/3425439/why-does-corrcoef-return-a-matrix

[68]:
```
datasets=[U_true,U_exp,V_true,V_exp]
ncc_matrix=np.corrcoef(datasets)
print(ncc_matrix)
```

```
[[1.         0.5954721  1.         0.91582973]
 [0.5954721  1.         0.5954721  0.58435245]
 [1.         0.5954721  1.         0.91582973]
 [0.91582973 0.58435245 0.91582973 1.        ]]
```

Cholesky decomposition for correlated data simulation

https://math.stackexchange.com/questions/163470/generating-correlated-random-numbers-why-does-cholesky-dec

https://stats.stackexchange.com/questions/160054/how-to-use-the-cholesky-decomposition-or-an-alternative-for-co

[69]:
```
import numpy as np
np.random.seed(1234)

no_obs = 10000             # Number of observations
means = [1, 2, 3]          # Mean values of each column
no_cols = 3                # Number of columns

sds = [1, 2, 3]            # SD of each column
sd = np.diag(sds)          # SD in a diagonal matrix for later operations

observations = np.random.normal(0, 1, (no_cols, no_obs)) # Rd draws N(0,1) in [3 x 1,000]

cor_matrix = np.array([[1.0, 0.6, 0.9],
                       [0.6, 1.0, 0.5],
                       [0.9, 0.5, 1.0]])          # The correlation matrix [3 x 3]
```

(continues on next page)

[19] https://docs.scipy.org/doc/numpy/reference/generated/numpy.corrcoef.html

(continued from previous page)

```
cov_matrix = np.dot(sd, np.dot(cor_matrix, sd))   # The covariance matrix

Chol = np.linalg.cholesky(cov_matrix)             # Cholesky decomposition

#array([[ 1.        ,  0.        ,  0.        ],
#       [ 1.2       ,  1.6       ,  0.        ],
#       [ 2.7       , -0.15      ,  1.29903811]])

sam_eq_mean = Chol.dot(observations)             # Generating random MVN (0, cov_matrix)

s = sam_eq_mean.transpose() + means               # Adding the means column wise
samples = s.transpose()                           # Transposing back

print(np.corrcoef(samples))                       # Checking correlation consistency.

# reference output (random, use seed 1234)
#[[ 1.          0.59714623  0.89991841]
# [ 0.59714623  1.          0.49739338]
# [ 0.89991841  0.49739338  1.        ]]
```

```
[[1.         0.59714623 0.89991841]
 [0.59714623 1.         0.49739338]
 [0.89991841 0.49739338 1.        ]]
```

6 Crystallography

6.1 Tools for Crystal Symmetry Analysis

Crystallographic symmetry analysis which can be necessary to interpret EBSD data can be carried out using the pymatgen[20] ("Python Materials Genomics") library [1].

The crystallographic analysis of the structure data is provided in pymatgen via the powerful spglib[21] library, which also defines the crystallographic conventions[22] used in space group analysis.

Pymatgen includes routines to access crystal structure data in various input formats, including data import directly from the online Crystallography Open Database[23].

[1] Shyue Ping Ong, William Davidson Richards, Anubhav Jain, Geoffroy Hautier, Michael Kocher, Shreyas Cholia, Dan Gunter, Vincent Chevrier, Kristin A. Persson, Gerbrand Ceder. *Python Materials Genomics (pymatgen) : A Robust, Open-Source Python Library for Materials Analysis.* Computational Materials Science, 2013, 68, 314–319. doi:10.1016/j.commatsci.2012.10.028[24]

[1]:
```
import numpy as np
import pymatgen as mg
from pymatgen.io.cif import CifParser
from pymatgen.ext.cod import COD
from pymatgen.symmetry.analyzer import SpacegroupAnalyzer
```

[20] http://pymatgen.org/
[21] https://atztogo.github.io/spglib/python-spglib.html#python-spglib
[22] https://atztogo.github.io/spglib/definition.html
[23] http://www.crystallography.net/cod/
[24] http://dx.doi.org/10.1016/j.commatsci.2012.10.028

Importing Structure Data

[2]:
```
#this gets the primitive cell
parser = CifParser("Cu-Copper.cif")
prim_structure = parser.get_structures()[0]
print(prim_structure)
```

```
Full Formula (Cu1)
Reduced Formula: Cu
abc   :   2.556163   2.556163   2.556163
angles:  60.000000  60.000000  60.000000
Sites (1)
  #  SP      a    b    c
---  ----  ---  ---  ---
  0  Cu      0    0    0
```

[3]:
```
cod = COD()
cod_structure = cod.get_structure_by_id(1010064)
print(cod_structure)
```

```
Full Formula (Li8 O4)
Reduced Formula: Li2O
abc   :   4.610000   4.610000   4.610000
angles:  90.000000  90.000000  90.000000
Sites (12)
  #  SP       a     b     c
---  ----  ----  ----  ----
  0  Li+   0.25  0.25  0.25
  1  Li+   0.25  0.75  0.75
  2  Li+   0.75  0.25  0.75
  3  Li+   0.75  0.75  0.25
  4  Li+   0.75  0.75  0.75
  5  Li+   0.75  0.25  0.25
  6  Li+   0.25  0.75  0.25
  7  Li+   0.25  0.25  0.75
  8  O2-   0     0     0
  9  O2-   0     0.5   0.5
 10  O2-   0.5   0     0.5
 11  O2-   0.5   0.5   0
```

[4]:
```
# Reading a structure from CIF
#structure = mg.Structure.from_str(open("CsCl.cif").read(), fmt="cif")
structure = mg.Structure.from_file("Cu-Copper.cif")
print(structure)
```

```
Full Formula (Cu4)
Reduced Formula: Cu
abc   :   3.614960   3.614960   3.614960
angles:  90.000000  90.000000  90.000000
Sites (4)
  #  SP      a    b    c
---  ----  ---  ---  ---
  0  Cu    0    0    0
  1  Cu    0    0.5  0.5
  2  Cu    0.5  0    0.5
  3  Cu    0.5  0.5  0
```

Space Group Assignment

[5]:
```
finder = SpacegroupAnalyzer(structure)
print("The space group symbol is {}".format(finder.get_space_group_symbol()))
print("The point group symbol is {}".format(finder.get_point_group_symbol()))

#structure=finder.get_symmetrized_structure()
#structure =finder.get_conventional_standard_structure()
#print(structure)
```

```
The space group symbol is Fm-3m
The point group symbol is m-3m
```

We can output a dictionary of all the symmetry data that has been determined by SpacegroupAnalyzer via spglib:

[6]:
```
symdata=finder.get_symmetry_dataset()
print(symdata)
```

```
{'number': 225, 'hall_number': 523, 'international': 'Fm-3m', 'hall': '-F 4 2 3', 'choice':
 ↪ '', 'transformation_matrix': array([[  1.00000000e+00,   3.79646512e-17,   0.
↪00000000e+00],
       [  1.91466924e-19,   1.91466924e-19,  -1.00000000e+00],
       [  0.00000000e+00,   1.00000000e+00,   0.00000000e+00]]), 'origin_shift': array([ 1.
↪,  1.,  1.]), 'rotations': array([[[ 1,  0,  0],
        [ 0,  1,  0],
        [ 0,  0,  1]],

       [[-1,  0,  0],
        [ 0, -1,  0],
        [ 0,  0, -1]],

       [[ 0,  0,  1],
        [ 0,  1,  0],
        [-1,  0,  0]],

       ...,
       [[ 0,  0,  1],
        [ 1,  0,  0],
        [ 0, -1,  0]],

       [[ 1,  0,  0],
        [ 0,  0, -1],
        [ 0,  1,  0]],

       [[-1,  0,  0],
        [ 0,  0,  1],
        [ 0, -1,  0]]], dtype=int32), 'translations': array([[  0.00000000e+00,   0.
↪00000000e+00,   0.00000000e+00],
       [  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
       [  1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
       [  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
       [  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
       [  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
       [  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
       [  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
       [  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
       [  1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
       [  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
```

(continues on next page)

(continued from previous page)

```
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
```

(continues on next page)

(continued from previous page)

```
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   0.00000000e+00],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  0.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
```

(continues on next page)

(continued from previous page)

```
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   1.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   0.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   0.00000000e+00,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
[  5.00000000e-01,   1.00000000e+00,   5.00000000e-01],
```

(continues on next page)

(continued from previous page)

```
       [  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
       [  0.00000000e+00,   5.00000000e-01,   5.00000000e-01],
       [  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
       [  1.00000000e+00,   5.00000000e-01,   5.00000000e-01],
       [  5.00000000e-01,   5.00000000e-01,   4.44089210e-16],
       [  1.00000000e+00,   5.00000000e-01,   5.00000000e-01]]), 'wyckoffs': ['a', 'a', 'a
↪', 'a'], 'equivalent_atoms': array([0, 0, 0, 0], dtype=int32), 'mapping_to_primitive':␣
↪array([0, 0, 0, 0], dtype=int32), 'std_lattice': array([[ 3.61496,  0.     ,  0.     ],
       [ 0.     ,  3.61496,  0.     ],
       [ 0.     ,  0.     ,  3.61496]]), 'std_types': array([1, 1, 1, 1], dtype=int32),
↪'std_positions': array([[ 0. ,  0. ,  0. ],
       [ 0. ,  0.5,  0.5],
       [ 0.5,  0. ,  0.5],
       [ 0.5,  0.5,  0. ]]), 'std_mapping_to_primitive': array([0, 0, 0, 0], dtype=int32),
↪'pointgroup': 'm-3m'}
```

Getting the unique point group operators:

[7]:
```
pg = np.array([ [[ 1.,  0.,  0.],
                 [ 0.,  1.,  0.],
                 [ 0.,  0.,  1.]] ] )
for idx,op in enumerate(symdata['rotations']):
    is_new_op=True
    for pg_op in pg:
        if np.array_equal(pg_op,op):
            is_new_op=False
    if is_new_op:
        pg=np.append(pg,[op],axis=0)
#print(pg)
print(pg.shape)
```

```
(48, 3, 3)
```

[8]:
```
hkl=np.array([1,0,0])
new_hkl=np.matmul(hkl,pg)
print(new_hkl.shape)
print(new_hkl)
```

```
(48, 3)
[[ 1.  0.  0.]
 [-1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]
 [-1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0. -1.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [-1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]
 [-1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0. -1.]
 [ 0.  0.  1.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]
```

(continues on next page)

(continued from previous page)

```
 [ 0.  1.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]
 [ 0.  0. -1.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [-1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]
 [-1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]
 [-1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0. -1.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [-1.  0.  0.]]
```

[9]:
```
# sort out the unique hkls of the family
hkl_family=np.array([new_hkl[0]])
for idx,hkl in enumerate(new_hkl):
    is_new_hkl=True
    for f in hkl_family:
        if np.array_equal(hkl,f):
            is_new_hkl=False
    if is_new_hkl:
        hkl_family=np.append(hkl_family,[hkl],axis=0)

print(hkl_family.shape)
print(hkl_family)
```

```
(6, 3)
[[ 1.  0.  0.]
 [-1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]
 [ 0.  1.  0.]
 [ 0. -1.  0.]]
```

[10]:
```
# alternative approach

# returns a list of SymmOp objects
pg_ops=finder.get_point_group_operations()

hkl=np.array([0,0,1])
family=np.array([hkl])
```

(continues on next page)

(continued from previous page)

```
for idx,op in enumerate(pg_ops):
    new_hkl=pg_ops[idx].apply_rotation_only(hkl)
    hkl_is_new=True
    for fvec in family:
        if np.array_equal(new_hkl,fvec):
            hkl_is_new=False
    if hkl_is_new:
        family=np.append(family,[new_hkl],axis=0)
print(hkl, " family:")
print(family)
```

```
[0 0 1]  family:
[[ 0.  0.  1.]
 [ 0.  0. -1.]
 [ 1.  0.  0.]
 [-1.  0.  0.]
 [ 0. -1.  0.]
 [ 0.  1.  0.]]
```

spglib Examples

[11]:
```
import sys
import spglib
import numpy as np

#######################################
# Uncomment to use Atoms class in ASE #
#######################################
# from ase import Atoms

#############################################################################
# Using the local Atoms-like class (BSD license) where a small set of #
# ASE Atoms features is comaptible but enough for this example.       #
#############################################################################
from atoms import Atoms
```

[12]:
```
def show_symmetry(symmetry):
    for i in range(symmetry['rotations'].shape[0]):
        print("  --------------- %4d ---------------" % (i + 1))
        rot = symmetry['rotations'][i]
        trans = symmetry['translations'][i]
        print("  rotation:")
        for x in rot:
            print("     [%2d %2d %2d]" % (x[0], x[1], x[2]))
        print("  translation:")
        print("     (%8.5f %8.5f %8.5f)" % (trans[0], trans[1], trans[2]))

def show_lattice(lattice):
    print("Basis vectors:")
    for vec, axis in zip(lattice, ("a", "b", "c")):
        print("%s %10.5f %10.5f %10.5f" % (tuple(axis,) + tuple(vec)))

def show_cell(lattice, positions, numbers):
    show_lattice(lattice)
    print("Atomic points:")
    for p, s in zip(positions, numbers):
        print("%2d %10.5f %10.5f %10.5f" % ((s,) + tuple(p)))
```

[13]:
```
silicon_ase = Atoms(symbols=['Si'] * 8,
                    cell=[(4, 0, 0),
                          (0, 4, 0),
                          (0, 0, 4)],
                    scaled_positions=[(0, 0, 0),
                                      (0, 0.5, 0.5),
                                      (0.5, 0, 0.5),
                                      (0.5, 0.5, 0),
                                      (0.25, 0.25, 0.25),
                                      (0.25, 0.75, 0.75),
                                      (0.75, 0.25, 0.75),
                                      (0.75, 0.75, 0.25)],
                    pbc=True)

silicon = ([(4, 0, 0),
            (0, 4, 0),
            (0, 0, 4)],
           [(0, 0, 0),
            (0, 0.5, 0.5),
            (0.5, 0, 0.5),
            (0.5, 0.5, 0),
            (0.25, 0.25, 0.25),
            (0.25, 0.75, 0.75),
            (0.75, 0.25, 0.75),
            (0.75, 0.75, 0.25)],
           [14,] * 8)

silicon_dist = ([(4.01, 0, 0),
                 (0, 4, 0),
                 (0, 0, 3.99)],
                [(0.001, 0, 0),
                 (0, 0.5, 0.5),
                 (0.5, 0, 0.5),
                 (0.5, 0.5, 0),
                 (0.25, 0.25, 0.251),
                 (0.25, 0.75, 0.75),
                 (0.75, 0.25, 0.75),
                 (0.75, 0.75, 0.25)],
                [14,] * 8)

silicon_prim = ([(0, 2, 2),
                 (2, 0, 2),
                 (2, 2, 0)],
                [(0, 0, 0),
                 (0.25, 0.25, 0.25)],
                [14, 14])

rutile = ([(4, 0, 0),
           (0, 4, 0),
           (0, 0, 3)],
          [(0, 0, 0),
           (0.5, 0.5, 0.5),
           (0.3, 0.3, 0.0),
           (0.7, 0.7, 0.0),
           (0.2, 0.8, 0.5),
           (0.8, 0.2, 0.5)],
          [14, 14, 8, 8, 8, 8])

rutile_dist = ([(3.97, 0, 0),
```

(continues on next page)

(continued from previous page)

```
                (0, 4.03, 0),
                (0, 0, 3.0)],
               [(0, 0, 0),
                (0.5001, 0.5, 0.5),
                (0.3, 0.3, 0.0),
                (0.7, 0.7, 0.002),
                (0.2, 0.8, 0.5),
                (0.8, 0.2, 0.5)],
               [14, 14, 8, 8, 8, 8])

a = 3.07
c = 3.52
MgB2 = ([(a, 0, 0),
         (-a/2, a/2*np.sqrt(3), 0),
         (0, 0, c)],
        [(0, 0, 0),
         (1.0/3, 2.0/3, 0.5),
         (2.0/3, 1.0/3, 0.5)],
        [12, 5, 5])

a = [3., 0., 0.]
b = [-3.66666667, 3.68178701, 0.]
c = [-0.66666667, -1.3429469, 1.32364995]
niggli_lattice = np.array([a, b, c])
```

[14]:
```
print("[get_spacegroup]")
print("  Spacegroup of Silicon is %s." % spglib.get_spacegroup(silicon))
print('')

print("[get_spacegroup]")
print("  Spacegroup of Silicon (ASE Atoms-like format) is %s." %
      spglib.get_spacegroup(silicon_ase))
print('')
print("[get_spacegroup]")
print("  Spacegroup of Rutile is %s." % spglib.get_spacegroup(rutile))
print('')
print("[get_spacegroup]")
print("  Spacegroup of MgB2 is %s." % spglib.get_spacegroup(MgB2))
print('')
print("[get_symmetry]")
print("  Symmetry operations of Rutile unitcell are:")
print('')
symmetry = spglib.get_symmetry(rutile)
show_symmetry(symmetry)
print('')
print("[get_symmetry]")
print("  Symmetry operations of MgB2 are:")
print('')
symmetry = spglib.get_symmetry(MgB2)
show_symmetry(symmetry)
print('')
print("[get_pointgroup]")
print("  Pointgroup of Rutile is %s." %
      spglib.get_pointgroup(symmetry['rotations'])[0])
print('')

dataset = spglib.get_symmetry_dataset( rutile )
print("[get_symmetry_dataset] ['international']")
```

(continues on next page)

(continued from previous page)

```
print("  Spacegroup of Rutile is %s (%d)." % (dataset['international'],
                                               dataset['number']))
print('')
print("[get_symmetry_dataset] ['pointgroup']")
print("  Pointgroup of Rutile is %s." % (dataset['pointgroup']))
print('')
print("[get_symmetry_dataset] ['hall']")
print("  Hall symbol of Rutile is %s (%d)." % (dataset['hall'],
                                               dataset['hall_number']))
print('')
print("[get_symmetry_dataset] ['wyckoffs']")
alphabet = "abcdefghijklmnopqrstuvwxyz"
print("  Wyckoff letters of Rutile are: ", dataset['wyckoffs'])
print('')
print("[get_symmetry_dataset] ['equivalent_atoms']")
print("  Mapping to equivalent atoms of Rutile are: ")
for i, x in enumerate(dataset['equivalent_atoms']):
  print("  %d -> %d" % (i + 1, x + 1))
print('')
print("[get_symmetry_dataset] ['rotations'], ['translations']")
print("  Symmetry operations of Rutile unitcell are:")
for i, (rot,trans) in enumerate(zip(dataset['rotations'],
                                    dataset['translations'])):
    print("  --------------- %4d ---------------" % (i + 1))
    print("  rotation:")
    for x in rot:
        print("     [%2d %2d %2d]" % (x[0], x[1], x[2]))
    print("  translation:")
    print("     (%8.5f %8.5f %8.5f)" % (trans[0], trans[1], trans[2]))
print('')

print("[refine_cell]")
print(" Refine distorted rutile structure")
lattice, positions, numbers = spglib.refine_cell(rutile_dist, symprec=1e-1)
show_cell(lattice, positions, numbers)
print('')

print("[find_primitive]")
print(" Fine primitive distorted silicon structure")
lattice, positions, numbers = spglib.find_primitive(silicon_dist, symprec=1e-1)
show_cell(lattice, positions, numbers)
print('')

print("[standardize_cell]")
print(" Standardize distorted rutile structure:")
print(" (to_primitive=0 and no_idealize=0)")
lattice, positions, numbers = spglib.standardize_cell(rutile_dist,
                                                      to_primitive=0,
                                                      no_idealize=0,
                                                      symprec=1e-1)
show_cell(lattice, positions, numbers)
print('')

print("[standardize_cell]")
print(" Standardize distorted rutile structure:")
print(" (to_primitive=0 and no_idealize=1)")
lattice, positions, numbers = spglib.standardize_cell(rutile_dist,
                                                      to_primitive=0,
```

(continues on next page)

(continued from previous page)

```
                                                    no_idealize=1,
                                                    symprec=1e-1)
show_cell(lattice, positions, numbers)
print('')

print("[standardize_cell]")
print(" Standardize distorted silicon structure:")
print(" (to_primitive=1 and no_idealize=0)")
lattice, positions, numbers = spglib.standardize_cell(silicon_dist,
                                                    to_primitive=1,
                                                    no_idealize=0,
                                                    symprec=1e-1)
show_cell(lattice, positions, numbers)
print('')

print("[standardize_cell]")
print(" Standardize distorted silicon structure:")
print(" (to_primitive=1 and no_idealize=1)")
lattice, positions, numbers = spglib.standardize_cell(silicon_dist,
                                                    to_primitive=1,
                                                    no_idealize=1,
                                                    symprec=1e-1)
show_cell(lattice, positions, numbers)
print('')

symmetry = spglib.get_symmetry(silicon)
print("[get_symmetry]")
print("  Number of symmetry operations of silicon conventional")
print("  unit cell is %d (192)." % len(symmetry['rotations']))
show_symmetry(symmetry)
print('')

symmetry = spglib.get_symmetry_from_database(525)
print("[get_symmetry_from_database]")
print("  Number of symmetry operations of silicon conventional")
print("  unit cell is %d (192)." % len(symmetry['rotations']))
show_symmetry(symmetry)
print('')

reduced_lattice = spglib.niggli_reduce(niggli_lattice)
print("[niggli_reduce]")
print("  Original lattice")
show_lattice(niggli_lattice)
print("  Reduced lattice")
show_lattice(reduced_lattice)
print('')


mapping, grid = spglib.get_ir_reciprocal_mesh([11, 11, 11],
                                              silicon_prim,
                                              is_shift=[0, 0, 0])
num_ir_kpt = len(np.unique(mapping))
print("[get_ir_reciprocal_mesh]")
print("  Number of irreducible k-points of primitive silicon with")
print("  11x11x11 Monkhorst-Pack mesh is %d (56)." % num_ir_kpt)
print('')

mapping, grid = spglib.get_ir_reciprocal_mesh([8, 8, 8],
```

(continues on next page)

(continued from previous page)

```
                                            rutile,
                                            is_shift=[1, 1, 1])
num_ir_kpt = len(np.unique(mapping))
print("[get_ir_reciprocal_mesh]")
print("  Number of irreducible k-points of Rutile with")
print("  8x8x8 Monkhorst-Pack mesh is %d (40)." % num_ir_kpt)
print('')

mapping, grid = spglib.get_ir_reciprocal_mesh([9, 9, 8],
                                            MgB2,
                                            is_shift=[0, 0, 1])
num_ir_kpt = len(np.unique(mapping))
print("[get_ir_reciprocal_mesh]")
print("  Number of irreducible k-points of MgB2 with")
print("  9x9x8 Monkhorst-Pack mesh is %s (48)." % num_ir_kpt)
print('')
```

```
[get_spacegroup]
  Spacegroup of Silicon is Fd-3m (227).

[get_spacegroup]
  Spacegroup of Silicon (ASE Atoms-like format) is Fd-3m (227).

[get_spacegroup]
  Spacegroup of Rutile is P4_2/mnm (136).

[get_spacegroup]
  Spacegroup of MgB2 is P6/mmm (191).

[get_symmetry]
  Symmetry operations of Rutile unitcell are:

  ---------------    1 ---------------
  rotation:
     [ 1  0  0]
     [ 0  1  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    2 ---------------
  rotation:
     [-1  0  0]
     [ 0 -1  0]
     [ 0  0 -1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    3 ---------------
  rotation:
     [ 0 -1  0]
     [ 1  0  0]
     [ 0  0  1]
  translation:
     ( 0.50000  0.50000  0.50000)
  ---------------    4 ---------------
  rotation:
     [ 0  1  0]
     [-1  0  0]
     [ 0  0 -1]
  translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.50000  0.50000  0.50000)
---------------    5 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    6 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    7 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------    8 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------    9 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------   10 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------   11 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   12 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   13 ---------------
```

(continues on next page)

(continued from previous page)

```
  rotation:
     [-1  0  0]
     [ 0  1  0]
     [ 0  0 -1]
  translation:
     ( 0.50000  0.50000  0.50000)
  ---------------   14 ---------------
  rotation:
     [ 1  0  0]
     [ 0 -1  0]
     [ 0  0  1]
  translation:
     ( 0.50000  0.50000  0.50000)
  ---------------   15 ---------------
  rotation:
     [ 0  1  0]
     [ 1  0  0]
     [ 0  0 -1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------   16 ---------------
  rotation:
     [ 0 -1  0]
     [-1  0  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)

[get_symmetry]
  Symmetry operations of MgB2 are:

  ---------------    1 ---------------
  rotation:
     [ 1  0  0]
     [ 0  1  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    2 ---------------
  rotation:
     [-1  0  0]
     [ 0 -1  0]
     [ 0  0 -1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    3 ---------------
  rotation:
     [ 1 -1  0]
     [ 1  0  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    4 ---------------
  rotation:
     [-1  1  0]
     [-1  0  0]
     [ 0  0 -1]
  translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.00000  0.00000  0.00000)
---------------    5 ---------------
rotation:
   [ 0 -1  0]
   [ 1 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    6 ---------------
rotation:
   [ 0  1  0]
   [-1  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    7 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    8 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    9 ---------------
rotation:
   [-1  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   10 ---------------
rotation:
   [ 1 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   11 ---------------
rotation:
   [ 0  1  0]
   [-1  1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   12 ---------------
rotation:
   [ 0 -1  0]
   [ 1 -1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   13 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   14 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   15 ---------------
rotation:
   [-1  0  0]
   [-1  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   16 ---------------
rotation:
   [ 1  0  0]
   [ 1 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   17 ---------------
rotation:
   [-1  1  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   18 ---------------
rotation:
   [ 1 -1  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   19 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   20 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   21 ---------------
rotation:
   [ 1  0  0]
```

(continues on next page)

(continued from previous page)

```
    [ 1 -1  0]
    [ 0  0 -1]
  translation:
    ( 0.00000  0.00000  0.00000)
  ---------------   22 ---------------
  rotation:
    [-1  0  0]
    [-1  1  0]
    [ 0  0  1]
  translation:
    ( 0.00000  0.00000  0.00000)
  ---------------   23 ---------------
  rotation:
    [ 1 -1  0]
    [ 0 -1  0]
    [ 0  0 -1]
  translation:
    ( 0.00000  0.00000  0.00000)
  ---------------   24 ---------------
  rotation:
    [-1  1  0]
    [ 0  1  0]
    [ 0  0  1]
  translation:
    ( 0.00000  0.00000  0.00000)

[get_pointgroup]
  Pointgroup of Rutile is 6/mmm.

[get_symmetry_dataset] ['international']
  Spacegroup of Rutile is P4_2/mnm (136).

[get_symmetry_dataset] ['pointgroup']
  Pointgroup of Rutile is 4/mmm.

[get_symmetry_dataset] ['hall']
  Hall symbol of Rutile is -P 4n 2n (419).

[get_symmetry_dataset] ['wyckoffs']
  Wyckoff letters of Rutile are:  ['a', 'a', 'f', 'f', 'f', 'f']

[get_symmetry_dataset] ['equivalent_atoms']
  Mapping to equivalent atoms of Rutile are:
  1 -> 1
  2 -> 1
  3 -> 3
  4 -> 3
  5 -> 3
  6 -> 3

[get_symmetry_dataset] ['rotations'], ['translations']
  Symmetry operations of Rutile unitcell are:
  ---------------    1 ---------------
  rotation:
    [ 1  0  0]
    [ 0  1  0]
    [ 0  0  1]
  translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.00000  0.00000  0.00000)
---------------    2 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    3 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------    4 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------    5 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    6 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    7 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------    8 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------    9 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.50000)
---------------   10 ---------------
```

(continues on next page)

(continued from previous page)

```
  rotation:
     [-1  0  0]
     [ 0  1  0]
     [ 0  0  1]
  translation:
     ( 0.50000  0.50000  0.50000)
  ---------------   11 ---------------
  rotation:
     [ 0 -1  0]
     [-1  0  0]
     [ 0  0 -1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------   12 ---------------
  rotation:
     [ 0  1  0]
     [ 1  0  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------   13 ---------------
  rotation:
     [-1  0  0]
     [ 0  1  0]
     [ 0  0 -1]
  translation:
     ( 0.50000  0.50000  0.50000)
  ---------------   14 ---------------
  rotation:
     [ 1  0  0]
     [ 0 -1  0]
     [ 0  0  1]
  translation:
     ( 0.50000  0.50000  0.50000)
  ---------------   15 ---------------
  rotation:
     [ 0  1  0]
     [ 1  0  0]
     [ 0  0 -1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------   16 ---------------
  rotation:
     [ 0 -1  0]
     [-1  0  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)

[refine_cell]
 Refine distorted rutile structure
Basis vectors:
a    4.00000    0.00000    0.00000
b    0.00000    4.00000    0.00000
c    0.00000    0.00000    3.00000
Atomic points:
14    0.00000    0.00000    0.00000
14    0.50000    0.50000    0.50000
```

(continues on next page)

(continued from previous page)

```
 8    0.29995    0.29995    0.00000
 8    0.70005    0.70005    0.00000
 8    0.20005    0.79995    0.50000
 8    0.79995    0.20005    0.50000

[find_primitive]
 Fine primitive distorted silicon structure
Basis vectors:
a    0.00000    2.00000    2.00000
b    2.00000    0.00000    2.00000
c    2.00000    2.00000    0.00000
Atomic points:
14    0.25000    0.25000    0.25000
14    0.00000    0.00000    0.00000

[standardize_cell]
 Standardize distorted rutile structure:
 (to_primitive=0 and no_idealize=0)
Basis vectors:
a    4.00000    0.00000    0.00000
b    0.00000    4.00000    0.00000
c    0.00000    0.00000    3.00000
Atomic points:
14    0.00000    0.00000    0.00000
14    0.50000    0.50000    0.50000
 8    0.29995    0.29995    0.00000
 8    0.70005    0.70005    0.00000
 8    0.20005    0.79995    0.50000
 8    0.79995    0.20005    0.50000

[standardize_cell]
 Standardize distorted rutile structure:
 (to_primitive=0 and no_idealize=1)
Basis vectors:
a    3.97000    0.00000    0.00000
b    0.00000    4.03000    0.00000
c    0.00000    0.00000    3.00000
Atomic points:
14    0.00000    0.00000    0.00000
14    0.50010    0.50000    0.50000
 8    0.30000    0.30000    0.00000
 8    0.70000    0.70000    0.00200
 8    0.20000    0.80000    0.50000
 8    0.80000    0.20000    0.50000

[standardize_cell]
 Standardize distorted silicon structure:
 (to_primitive=1 and no_idealize=0)
Basis vectors:
a    0.00000    2.00000    2.00000
b    2.00000    0.00000    2.00000
c    2.00000    2.00000    0.00000
Atomic points:
14    0.25000    0.25000    0.25000
14    0.00000    0.00000    0.00000

[standardize_cell]
 Standardize distorted silicon structure:
```

(continues on next page)

(continued from previous page)

```
 (to_primitive=1 and no_idealize=1)
Basis vectors:
a    0.00000    2.00000   -1.99500
b    2.00500    2.00000    0.00000
c    2.00500    0.00000   -1.99500
Atomic points:
14    0.99975    0.00025    0.00025
14    0.74975    0.75025    0.74975

[get_symmetry]
  Number of symmetry operations of silicon conventional
  unit cell is 192 (192).
  ---------------    1 ---------------
  rotation:
     [ 1  0  0]
     [ 0  1  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    2 ---------------
  rotation:
     [ 0  0  1]
     [ 0  1  0]
     [-1  0  0]
  translation:
     ( 0.25000  0.75000  0.75000)
  ---------------    3 ---------------
  rotation:
     [-1  0  0]
     [ 0  1  0]
     [ 0  0 -1]
  translation:
     ( 0.50000  0.00000  0.50000)
  ---------------    4 ---------------
  rotation:
     [ 0  0 -1]
     [ 0  1  0]
     [ 1  0  0]
  translation:
     ( 0.25000  0.75000  0.75000)
  ---------------    5 ---------------
  rotation:
     [ 1  0  0]
     [ 0 -1  0]
     [ 0  0 -1]
  translation:
     ( 0.00000  0.50000  0.50000)
  ---------------    6 ---------------
  rotation:
     [ 0  0  1]
     [ 0 -1  0]
     [ 1  0  0]
  translation:
     ( 0.25000  0.75000  0.75000)
  ---------------    7 ---------------
  rotation:
     [-1  0  0]
     [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------    8 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------    9 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   10 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   11 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   12 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   13 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   14 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   15 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.50000  0.00000  0.50000)
---------------   16 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   17 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   18 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   19 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   20 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   21 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   22 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   23 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   24 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   25 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   26 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   27 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   28 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   29 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   30 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   31 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   32 ---------------
rotation:
   [ 0  0  1]
```

(continues on next page)

(continued from previous page)

```
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   33 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   34 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   35 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   36 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   37 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   38 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   39 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   40 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
```

(continues on next page)

(continued from previous page)

```
translation:
   ( 0.00000  0.50000  0.50000)
---------------   41 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   42 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   43 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   44 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   45 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   46 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   47 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   48 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
```

(continues on next page)

(continued from previous page)

```
---------------   49 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   50 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   51 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   52 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   53 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   54 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   55 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   56 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   57 ---------------
rotation:
```

(continues on next page)

(continued from previous page)

```
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   58 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   59 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   60 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   61 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   62 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   63 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   64 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   65 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   66 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   67 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   68 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   69 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   70 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   71 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   72 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   73 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.75000  0.25000  0.75000)
---------------   74 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   75 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   76 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   77 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   78 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   79 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   80 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   81 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   82 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   83 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   84 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   85 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   86 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   87 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   88 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   89 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   90 ---------------
rotation:
   [-1  0  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   91 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   92 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   93 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   94 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   95 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   96 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   97 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   98 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
```

(continues on next page)

(continued from previous page)

```
translation:
   ( 0.75000  0.75000  0.25000)
---------------   99 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  100 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  101 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  102 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  103 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  104 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  105 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  106 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
```

(continues on next page)

(continued from previous page)

```
---------------  107 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  108 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  109 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  110 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  111 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  112 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  113 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  114 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  115 ---------------
rotation:
```

(continues on next page)

(continued from previous page)

```
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  116 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  117 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  118 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  119 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  120 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  121 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  122 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  123 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  124 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  125 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  126 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  127 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  128 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  129 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  130 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  131 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.75000  0.75000  0.25000)
---------------  132 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  133 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  134 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  135 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  136 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  137 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  138 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  139 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  140 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  141 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  142 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  143 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  144 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  145 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  146 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  147 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  148 ---------------
rotation:
   [ 0  0 -1]
```

(continues on next page)

(continued from previous page)

```
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  149 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  150 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  151 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  152 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  153 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  154 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  155 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  156 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
```

(continues on next page)

(continued from previous page)

```
translation:
   ( 0.25000  0.25000  0.25000)
---------------  157 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  158 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  159 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  160 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  161 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  162 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  163 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  164 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.25000  0.25000)
```

(continues on next page)

(continued from previous page)

```
---------------  165 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  166 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  167 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  168 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  169 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  170 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  171 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  172 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  173 ---------------
rotation:
```

(continues on next page)

(continued from previous page)

```
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  174 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  175 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  176 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  177 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  178 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  179 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  180 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  181 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
```

(continues on next page)

(continued from previous page)

```
   [ 1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  182 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  183 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  184 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  185 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  186 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  187 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  188 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  189 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
```

(continues on next page)

(continued from previous page)

```
    ( 0.25000  0.25000  0.25000)
  ---------------  190 ---------------
  rotation:
     [ 1  0  0]
     [ 0  0 -1]
     [ 0 -1  0]
  translation:
     ( 0.00000  0.50000  0.50000)
  ---------------  191 ---------------
  rotation:
     [ 0  0  1]
     [ 1  0  0]
     [ 0 -1  0]
  translation:
     ( 0.25000  0.25000  0.25000)
  ---------------  192 ---------------
  rotation:
     [-1  0  0]
     [ 0  0  1]
     [ 0 -1  0]
  translation:
     ( 0.50000  0.00000  0.50000)

[get_symmetry_from_database]
  Number of symmetry operations of silicon conventional
  unit cell is 192 (192).
  ---------------    1 ---------------
  rotation:
     [ 1  0  0]
     [ 0  1  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------    2 ---------------
  rotation:
     [ 0 -1  0]
     [ 1  0  0]
     [ 0  0  1]
  translation:
     ( 0.25000  0.25000  0.25000)
  ---------------    3 ---------------
  rotation:
     [-1  0  0]
     [ 0 -1  0]
     [ 0  0  1]
  translation:
     ( 0.00000  0.50000  0.50000)
  ---------------    4 ---------------
  rotation:
     [ 0  1  0]
     [-1  0  0]
     [ 0  0  1]
  translation:
     ( 0.75000  0.25000  0.75000)
  ---------------    5 ---------------
  rotation:
     [ 1  0  0]
     [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------    6 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------    7 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------    8 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------    9 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   10 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   11 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   12 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   13 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.00000  0.00000  0.00000)
---------------   14 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   15 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   16 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   17 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   18 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   19 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   20 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   21 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   22 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   23 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   24 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   25 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   26 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   27 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   28 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   29 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   30 ---------------
rotation:
   [ 0  1  0]
```

(continues on next page)

(continued from previous page)

```
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   31 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   32 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   33 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   34 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   35 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   36 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   37 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   38 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
```

(continues on next page)

(continued from previous page)

```
translation:
   ( 0.50000  0.00000  0.50000)
---------------   39 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   40 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   41 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   42 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   43 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   44 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   45 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   46 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
```

(continues on next page)

(continued from previous page)

```
---------------   47 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   48 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   49 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   50 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   51 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   52 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   53 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   54 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   55 ---------------
rotation:
```

(continues on next page)

(continued from previous page)

```
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   56 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   57 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   58 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   59 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   60 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   61 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   62 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   63 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   64 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   65 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   66 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   67 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   68 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   69 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   70 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   71 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.50000  0.00000  0.50000)
---------------   72 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   73 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   74 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   75 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   76 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   77 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   78 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   79 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------   80 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   81 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   82 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   83 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   84 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   85 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   86 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------   87 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------   88 ---------------
rotation:
   [ 0  0  1]
```

(continues on next page)

(continued from previous page)

```
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   89 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   90 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------   91 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   92 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------   93 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------   94 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   95 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   96 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
translation:
   ( 0.50000  0.50000  0.00000)
---------------   97 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------   98 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------   99 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  100 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  101 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  102 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  103 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  104 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.75000  0.75000)
```

(continues on next page)

(continued from previous page)

```
---------------  105 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  106 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  107 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  108 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  109 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  110 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  111 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  112 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  113 ---------------
rotation:
```

(continues on next page)

(continued from previous page)

```
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  114 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  115 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  116 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  117 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  118 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  119 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  120 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  121 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  122 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  123 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  124 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  125 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  126 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  127 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  128 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  129 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
```

(continues on next page)

(continued from previous page)

```
   ( 0.75000  0.25000  0.75000)
---------------  130 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  131 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  132 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  133 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  134 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  135 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  136 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  137 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  138 ---------------
```

(continues on next page)

(continued from previous page)

```
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  139 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  140 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  141 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  142 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  143 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  144 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  145 ---------------
rotation:
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  146 ---------------
rotation:
   [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  147 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  148 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  149 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  150 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  151 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  152 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  153 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  154 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [ 1  0  0]
```

(continues on next page)

(continued from previous page)

```
translation:
   ( 0.75000  0.75000  0.25000)
---------------  155 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  156 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  157 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  158 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  159 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  160 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  161 ---------------
rotation:
   [ 0  1  0]
   [ 0  0  1]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  162 ---------------
rotation:
   [ 1  0  0]
   [ 0  0  1]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
```

(continues on next page)

(continued from previous page)

```
---------------  163 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0  1]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  164 ---------------
rotation:
   [-1  0  0]
   [ 0  0  1]
   [ 0  1  0]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  165 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  166 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0 -1  0]
translation:
   ( 0.25000  0.25000  0.25000)
---------------  167 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  168 ---------------
rotation:
   [ 1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  169 ---------------
rotation:
   [-1  0  0]
   [ 0 -1  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  170 ---------------
rotation:
   [ 0  1  0]
   [-1  0  0]
   [ 0  0 -1]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  171 ---------------
rotation:
```

(continues on next page)

(continued from previous page)

```
   [ 1  0  0]
   [ 0  1  0]
   [ 0  0 -1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  172 ---------------
rotation:
   [ 0 -1  0]
   [ 1  0  0]
   [ 0  0 -1]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  173 ---------------
rotation:
   [-1  0  0]
   [ 0  1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  174 ---------------
rotation:
   [ 0  1  0]
   [ 1  0  0]
   [ 0  0  1]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  175 ---------------
rotation:
   [ 1  0  0]
   [ 0 -1  0]
   [ 0  0  1]
translation:
   ( 0.75000  0.25000  0.75000)
---------------  176 ---------------
rotation:
   [ 0 -1  0]
   [-1  0  0]
   [ 0  0  1]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  177 ---------------
rotation:
   [ 0  0 -1]
   [-1  0  0]
   [ 0 -1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  178 ---------------
rotation:
   [ 0  0 -1]
   [ 0  1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  179 ---------------
rotation:
   [ 0  0 -1]
   [ 1  0  0]
```

(continues on next page)

(continued from previous page)

```
   [ 0  1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  180 ---------------
rotation:
   [ 0  0 -1]
   [ 0 -1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.00000  0.00000)
---------------  181 ---------------
rotation:
   [ 0  0  1]
   [-1  0  0]
   [ 0  1  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  182 ---------------
rotation:
   [ 0  0  1]
   [ 0  1  0]
   [ 1  0  0]
translation:
   ( 0.00000  0.50000  0.50000)
---------------  183 ---------------
rotation:
   [ 0  0  1]
   [ 1  0  0]
   [ 0 -1  0]
translation:
   ( 0.25000  0.75000  0.75000)
---------------  184 ---------------
rotation:
   [ 0  0  1]
   [ 0 -1  0]
   [-1  0  0]
translation:
   ( 0.50000  0.00000  0.50000)
---------------  185 ---------------
rotation:
   [ 0 -1  0]
   [ 0  0 -1]
   [-1  0  0]
translation:
   ( 0.75000  0.75000  0.25000)
---------------  186 ---------------
rotation:
   [-1  0  0]
   [ 0  0 -1]
   [ 0  1  0]
translation:
   ( 0.50000  0.50000  0.00000)
---------------  187 ---------------
rotation:
   [ 0  1  0]
   [ 0  0 -1]
   [ 1  0  0]
translation:
```

(continues on next page)

(continued from previous page)

```
    ( 0.25000  0.25000  0.25000)
  ---------------  188 ---------------
  rotation:
     [ 1  0  0]
     [ 0  0 -1]
     [ 0 -1  0]
  translation:
     ( 0.50000  0.00000  0.50000)
  ---------------  189 ---------------
  rotation:
     [ 0  1  0]
     [ 0  0  1]
     [-1  0  0]
  translation:
     ( 0.75000  0.75000  0.25000)
  ---------------  190 ---------------
  rotation:
     [ 1  0  0]
     [ 0  0  1]
     [ 0  1  0]
  translation:
     ( 0.00000  0.00000  0.00000)
  ---------------  191 ---------------
  rotation:
     [ 0 -1  0]
     [ 0  0  1]
     [ 1  0  0]
  translation:
     ( 0.25000  0.25000  0.25000)
  ---------------  192 ---------------
  rotation:
     [-1  0  0]
     [ 0  0  1]
     [ 0 -1  0]
  translation:
     ( 0.00000  0.50000  0.50000)

[niggli_reduce]
  Original lattice
Basis vectors:
a    3.00000    0.00000    0.00000
b   -3.66667    3.68179    0.00000
c   -0.66667   -1.34295    1.32365
  Reduced lattice
Basis vectors:
a   -0.66667   -1.34295    1.32365
b    2.33333   -1.34295    1.32365
c    1.00000    0.99589    2.64730

[get_ir_reciprocal_mesh]
  Number of irreducible k-points of primitive silicon with
  11x11x11 Monkhorst-Pack mesh is 56 (56).

[get_ir_reciprocal_mesh]
  Number of irreducible k-points of Rutile with
  8x8x8 Monkhorst-Pack mesh is 40 (40).

[get_ir_reciprocal_mesh]
```

(continues on next page)

(continued from previous page)

```
Number of irreducible k-points of MgB2 with
9x9x8 Monkhorst-Pack mesh is 48 (48).
```

7 aloe package

The aloe package bundles the functionality for Kikuchi pattern analysis.

7.1 Subpackages

aloe.exp package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.exp.affine3D module

aloe.exp.affine3D.**do_fit_affine3D**()
test the affine transformation fit

aloe.exp.affine3D.**fit_affine3D**(*xpri=None*, *ysec=None*)
Solve the least squares problem xpri * A = ysec

aloe.exp.affine3D.**sem_fit_affine3D**()

aloe.exp.affine3D.**transform_affine**(*x*, *A*)
transform x (n x 3) by A (4x4) to y (n x 3)

aloe.exp.calibpc module

aloe.exp.calibpc.**PCstats**(*DataMatrixo*, *mean_dim*, *title*)
Statistics of pattern center positions

PCX, PCY, and DD should be constant along rows or colums respectively thus the stdDev along a column gives an error estimate

aloe.exp.calibpc.**brkr_to_gnom**(*pcx*, *pcy*, *dd*, *aspect*)
convert Bruker PCX,PCY,DD,Aspect to gnomonic

aloe.exp.calibpc.**brkr_to_pcxyz**(*pcbrkr*, *top_clip=0.0*)
convert Bruker pc coordinates PCX,PCY,DD into microns in detector system

aloe.exp.calibpc.**calibratePC**(*ScanPointList*, *bcf_filename*, *mapinfo*, *XTilt=-20.0*)
This function calibrates the Projection Center from the current PC fit values in ScanPointList assuming that the step size and the sample tilt is known. This makes it possible to extrapolate all experimentally determined PC values to a common reference point (0,0) assuming a regular x-y grid of steps tilted by around the detector X-axis. The extrapolated reference PC values are then averaged to result in the PC estimation.

Notes: * the SEM beam X-scan is exactly parallel to the X-Tilt detector axis * Bruker: XTilt=(SampleTilt-90)-DetectorTilt (deg) * i.e. SampleTilt=70 usually leads to NEGATIVE XTilts!!! * the method produces PC values in a tilted rectangular region * no trapezoidal/projective distortion is considered

aloe.exp.calibpc.**calibratePC_BRKR**(*pcdata*, *mapinfo*, *XTilt=-20.0*)
This function calibrates the Projection Center from the current PC fit values in ScanPointList assuming that the step size and the sample tilt is known. This makes it possible to extrapolate all experimentally determined PC values to a common reference point (0,0) assuming a regular x-y grid of steps tilted by around the detector X-axis. The extrapolated reference PC values are then averaged to result in the PC estimation.

Notes: * the SEM beam X-scan is exactly parallel to the X-Tilt detector axis * Bruker: XTilt=(SampleTilt-90)-DetectorTilt (deg) * i.e. SampleTilt=70 usually leads to NEGATIVE XTilts!!! * the method produces PC values in a tilted rectangular region * no trapezoidal/projective distortion is considered

aloe.exp.calibpc.**make_interpolated_PCdata**(*xc_beam_indices*, *xc_pc_coords*, *nwidth*, *nheight*, *outfile=''*)
interpolates between projection center values, input assumes column vectors

Radial Basis Functions for interpolation of scattered data: http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html

aloe.exp.calibpc.**make_map_indices**(*w*, *h*)
makes 3 1D arrays of x and y, z=1 indices of a map w*h for projective transformations

aloe.exp.calibpc.**make_projective_PCdata**(*A*, *MapWidth*, *MapHeight*, *outfile=None*)
apply projective transformation matrix to all map indices

TODO: 1. save projection center values for direct import into DynamicS

2. save PC values in HDF5

aloe.exp.calibpc.**normrange**(*Y*)
normalize the range of the Y values

aloe.exp.calibpc.**pcxyz_to_brkr**(*pc_xyz*, *top_clip=0.0*)
Convert detector pc coordinates into Bruker convention

aloe.exp.calibpc.**pcxyz_to_gnom**(*pc_xyz*, *top_clip=0.0*)
convert detector pc coordinates to gnomonic projection values

aloe.exp.calibpc.**plotPC**(*secondary*, *fit*, *pc3=None*, *units='Bruker'*, *plotdir=''*)
plots the pattern center data and shows parameters

aloe.exp.calibpc.**project_points**(*pts_src_3d*, *h*)
make 2D plane-projected points from homography matrix

aloe.exp.ebsdconst module

This module contains a number constants which are relevant for different EBSD systems.

Constants:

BRKR_WIDTH_MICRONS: The full width of Bruker EBSD detector in microns. This value is constant for all measured patterns, and is the only one which should be relied on when calculating with projection center values from a given image. If the aspect ratio of an image measured with the Bruker detector is not exactly 4:3, e.g. 400x300, 160x120, but e.g. 400x288 or 160x115, the image has been clipped by removing some of its top lines and the effective BRKR_HEIGHT_MICRONS (see below) is reduced from its ideal 4:3 value.

BRKR_HEIGHT_MICRONS: The full height of Bruker EBSD detector. Due to hardware limitations, the top part of the measured patterns can be removed by setting a "top clip" value in the measurement software. The top clip is usually 0.04 or 0.05, which leads to pattern dimensions like 160x115. The effective screen height is thus reduced. When calculating the absolute coordinates of the projection center in microns, the possible top clip value has to be considered because the PCX,PCY,DD are given relative to the pixel-image dimensions. The absolute physical image height then corresponds to BRKR_HEIGHT_MICRONS*(1.0-top_clip).

TIMEPIX_WIDTH_MICRONS, TIMEPIX_HEIGHT_MICRONS: Dimensions of the TIMEPIX detector.

aloe.exp.ebsp_detection_model module

aloe.exp.pcfitter module

aloe.ext package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.ext.ctfFile3 module

ctfFile – Reader for CTF files

class aloe.ext.ctfFile3.**Ctf**(*filepath*)
Bases: object

getAcceleratingVoltage()
Return the accelerating voltage [units=kV]

Return type float

getAcquisitionEuler1()
Return the acquisition euler angle (θ_1) [units=deg]

Return type float

getAcquisitionEuler2()
Return the acquisition euler angle (θ_2) [units=deg]

Return type float

getAcquisitionEuler3()
Return the acquisition euler angle (θ_3) [units=deg]

Return type float

getAcquisitionEulers()
Return the acquisition euler angles ($\theta_1, \theta_2, \theta_3$) [units=deg]

Return type float

getAuthor()
Return the author

Return type str

getCoverage()

`getDevice`()

`getHeight`()
Return the height of mapping (points in Y) [units=px]

Return type int

`getJobMode`()
Return the job mode

Return type str

`getMagnification`()
Return the magnification [units=X]

Return type float

`getNumberPhases`()
Return the number of phases

Return type int

`getNumberPixels`()
Return the number of pixels in the map

Return type int

`getPhaseLatticeAngleAlpha`(*id*)
Return the lattice angle α of a given phase *id* [units=deg]

Parameters **id** (*int*) – id of the phase (the first phase has `id == 1`)

Return type float

`getPhaseLatticeAngleBeta`(*id*)
Return the lattice angle β of a given phase *id* [units=deg]

Parameters **id** (*int*) – id of the phase (the first phase has `id == 1`)

Return type float

`getPhaseLatticeAngleGamma`(*id*)
Return the lattice angle γ of a given phase *id* [units=deg]

Parameters **id** (*int*) – id of the phase (the first phase has `id == 1`)

Return type float

`getPhaseLatticeAngles`(*id*)
Return the lattice angles of a given phase *id* [units=deg]

Parameters **id** (*int*) – id of the phase (the first phase has `id == 1`)

Return type (float(α), float(β), float(γ))

`getPhaseLatticeParameterA`(*id*)
Return the lattice parameter a of a given phase *id* [units=:math:*text{angstroms}*]

Parameters **id** (*int*) – id of the phase (the first phase has `id == 1`)

Return type float

`getPhaseLatticeParameterB`(*id*)

Return the lattice parameter b of a given phase *id* [units=:math:*text{angstroms}*]

Parameters **id** (*int*) – id of the phase (the first phase has `id == 1`)

Return type float

`getPhaseLatticeParameterC`(*id*)

Return the lattice parameter c of a given phase *id* [units=:math:*text{angstroms}*]

Parameters `id` (*int*) – id of the phase (the first phase has `id == 1`)

Return type float

`getPhaseLatticeParameters`(*id*)

Return the lattice parameters of a given phase *id* [units=:math:*text{angstroms}*]

Parameters `id` (*int*) – id of the phase (the first phase has `id == 1`)

Return type (float(a), float(b), float(c))

`getPhaseName`(*id*)

Return the phase of a given phase *id*

Parameters `id` (*int*) – id of the phase (the first phase has `id == 1`)

Return type str

`getPhaseSpaceGroupNo`(*id*)

Return the space group no. of a given phase

Parameters `id` (*int*) – id of the phase (the first phase has `id == 1`)

Return type int between [1, 230]

`getPhases`(*id=None*)

Return the dictionary of a given phase *id* or the whole phase dictionary if `id == None`

Parameters `id` (*int*) – id of the phase (the first phase has `id == 1`)

Return type dict

`getPhasesList`()

Return a list with all the phases name.

Return type list

`getPixArray`(*key='euler1'*, *noneValue=None*, ***conditions*)

Return the filtered list for a given column header *key* and a set of conditions. The pixels that don't respect the condition(s) have a value of *noneValue*.

Parameters: *key*: name of the columns. Refer to `getPixelResults()` for list of keys.

The conditions are given as a *tuple* where

- the first element is the operator (`'='`, `'>'`, `'>='`, `'<'`, `'<='`, `'!='`)
- the second element is the value

Note: The greater than and less than only works with float and decimal.

Examples::

```
# Return a array of band contrast for pixels of only the second phase
# and that have a band contrast higher than 50
pixArray = ctf.getPixelArray(key='bc', phase=('=', 2), bc=('>', 50)))
```

Return type list

`getPixelArray`(*key='euler1', noneValue=None, **conditions*)
Return the filtered list for a given column header *key* and a set of conditions. The pixels that don't respect the condition(s) have a value of *noneValue*.

Parameters: *key*: name of the columns. Refer to `getPixelResults()` for list of keys.

The conditions are given as a *tuple* where

- the first element is the operator (`'='`, `'>'`, `'>='`, `'<'`, `'<='`, `'!='`)
- the second element is the value

Note: The greater than and less than only works with float and decimal.

Examples::

```
# Return a array of band contrast for pixels of only the second phase
# and that have a band contrast higher than 50
pixArray = ctf.getPixelArray(key='bc', phase=('=', 2), bc=('>', 50)))
```

Return type list

`getPixelCoordinate`(*index*)
Return the coordinate of the pixel from its index. The pixel (0,0) has an index of 1.

Parameters `index` (*int*) – the index of the pixel

Return type a integer tuple (x, y)

`getPixelImageLabel`(***data*)
Return the filename of the pixel image (diffraction pattern).

Parameters

- `coord` (*tuple*) – a integer tuple (x, y) between
  - $0 < x <= (\text{XCells} - 1)$
  - $0 < y <= (\text{YCells} - 1)$
- `index` (*int*) – index of the pixel

Return type string

`getPixelIndex`(*coord*)
Return the index of the pixel which also represents the image number of that pixel. The pixel (0,0) has an index of 1.

Parameters `coord` (*tuple*) – a integer tuple (x, y) between

- $0 < x <= (\text{XCells} - 1)$
- $0 < y <= (\text{YCells} - 1)$

Return type int

`getPixelIndexLabel`(***data*)
Return the label of the index of the pixel which also represents the image number of that pixel. The pixel (0,0) has an index of 1. The number of zero before the index is asjusted accordingly to the maximum index.

Parameters `coord` (*tuple*) – a integer tuple (x, y) between

- $0 < x <= (\text{XCells} - 1)$
- $0 < y <= (\text{YCells} - 1)$

Return type str

`getPixelResults_coordinate`(*coord*)

Return a dictionary with the data for a given coordinate.

key	Description
phase	id of the phase (the first phase has `id == 1`)
x	x position of the pixel
y	y position of the pixel
bands	number of bands used in the indexation
errorcode	see below
error	explanation of the *errorcode*
euler1	first euler angle (Bunge convention)
euler2	second euler angle (Bunge convention)
euler3	third euler angle (Bunge convention)
mad	mean angular deviation
bc	band contrast
bs	band slope

errorcode	Description
0	Success
1	Low band contrast
2	Low band slope
3	No solution
4	High MAD
5	Not yet analysed (job cancelled before point!)
6	Unexpected error (excepts etc.)

Parameters `coord` (*tuple*) – a integer tuple (x, y) between

- $0 < x <= (\text{XCells} - 1)$
- $0 < y <= (\text{YCells} - 1)$

Return type dict

`getPixelResults_index`(*index*)

Return a dictionary with the data for a given index.

Parameters `index` (*int*) – the index of the pixel

Return type dict

`getProjectFolderName`()

Return the immediate folder where the project is located from the information stored in the ctf.

Return type str

`getProjectFolderPath`()

Return the full path of the folder where the project is located from the information stored in the ctf.

Return type str

`getProjectImagesFolderName`()

Return the folder name where the project images are located from the information stored in the ctf.

Return type str

`getProjectImagesFolderPath()`
Return the full path of the folder where the project images are located from the information stored in the ctf.

Return type str

`getProjectName()`
Return the name of the project (No .cpr or .ctf) from the information stored in the ctf.

Return type str

`getSize()`
Return the number of points in the mapping [units=px]

Return type int

`getTiltAngle()`
Return the tilt angle [units=deg]

Return type float

`getTiltAxis()`
Return the tilt axis

Return type int

`getWidth()`
Return the width of mapping (points in) [units=px]

Return type int

`getXCells()`
Return the number of points in the X direction [units=px]

Return type int

`getXStep()`
Return the size of the step in the X direction [units= μm]

Return type float

`getYCells()`
Return the number of points in the Y direction [units=px]

Return type int

`getYStep()`
Return the size of the step in the Y direction [units= μm]

Return type float

aloe.ext.transformations module

Homogeneous Transformation Matrices and Quaternions.

A library for calculating 4x4 matrices for translating, rotating, reflecting, scaling, shearing, projecting, orthogonalizing, and superimposing arrays of 3D homogeneous coordinates as well as for converting between rotation matrices, Euler angles, and quaternions. Also includes an Arcball control object and functions to decompose transformation matrices.

Author Christoph Gohlke[25]

Organization Laboratory for Fluorescence Dynamics, University of California, Irvine

Version 2017.02.17

[25] http://www.lfd.uci.edu/~gohlke/

Requirements

- CPython 2.7 or 3.5[26]
- Numpy 1.11[27]
- Transformations.c 2017.02.17[28] (recommended for speedup of some functions)

Notes

The API is not stable yet and is expected to change between revisions.

This Python code is not optimized for speed. Refer to the transformations.c module for a faster implementation of some functions.

Documentation in HTML format can be generated with epydoc.

Matrices (M) can be inverted using numpy.linalg.inv(M), be concatenated using numpy.dot(M0, M1), or transform homogeneous coordinate arrays (v) using numpy.dot(M, v) for shape (4, *) column vectors, respectively numpy.dot(v, M.T) for shape (*, 4) row vectors ("array of points").

This module follows the "column vectors on the right" and "row major storage" (C contiguous) conventions. The translation components are in the right column of the transformation matrix, i.e. M[:3, 3]. The transpose of the transformation matrices may have to be used to interface with other graphics systems, e.g. with OpenGL's glMultMatrixd(). See also [16].

Calculations are carried out with numpy.float64 precision.

Vector, point, quaternion, and matrix function arguments are expected to be "array like", i.e. tuple, list, or numpy arrays.

Return types are numpy arrays unless specified otherwise.

Angles are in radians unless specified otherwise.

Quaternions w+ix+jy+kz are represented as [w, x, y, z].

A triple of Euler angles can be applied/interpreted in 24 ways, which can be specified using a 4 character string or encoded 4-tuple:

Axes 4-string: e.g. 'sxyz' or 'ryxy'

- first character : rotations are applied to 's'tatic or 'r'otating frame
- remaining characters : successive rotation axis 'x', 'y', or 'z'

Axes 4-tuple: e.g. (0, 0, 0, 0) or (1, 1, 1, 1)

- inner axis: code of axis ('x':0, 'y':1, 'z':2) of rightmost matrix.
- parity : even (0) if inner axis 'x' is followed by 'y', 'y' is followed by 'z', or 'z' is followed by 'x'. Otherwise odd (1).
- repetition : first and last axis are same (1) or different (0).
- frame : rotations are applied to static (0) or rotating (1) frame.

Other Python packages and modules for 3D transformations and quaternions:

- **Transforms3d**[29] includes most code of this module.
- Blender.mathutils[30]

[26] http://www.python.org
[27] http://www.numpy.org
[28] http://www.lfd.uci.edu/~gohlke/
[29] https://pypi.python.org/pypi/transforms3d
[30] http://www.blender.org/api/blender_python_api

- numpy-dtypes[31]

References

(1) Matrices and transformations. Ronald Goldman. In "Graphics Gems I", pp 472-475. Morgan Kaufmann, 1990.

(2) More matrices and transformations: shear and pseudo-perspective. Ronald Goldman. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

(3) Decomposing a matrix into simple transformations. Spencer Thomas. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

(4) Recovering the data from the transformation matrix. Ronald Goldman. In "Graphics Gems II", pp 324-331. Morgan Kaufmann, 1991.

(5) Euler angle conversion. Ken Shoemake. In "Graphics Gems IV", pp 222-229. Morgan Kaufmann, 1994.

(6) Arcball rotation control. Ken Shoemake. In "Graphics Gems IV", pp 175-192. Morgan Kaufmann, 1994.

(7) Representing attitude: Euler angles, unit quaternions, and rotation vectors. James Diebel. 2006.

(8) A discussion of the solution for the best rotation to relate two sets of vectors. W Kabsch. Acta Cryst. 1978. A34, 827-828.

(9) Closed-form solution of absolute orientation using unit quaternions. BKP Horn. J Opt Soc Am A. 1987. 4(4):629-642.

(10) Quaternions. Ken Shoemake. http://www.sfu.ca/~jwa3/cmpt461/files/quatut.pdf

(11) From quaternion to matrix and back. JMP van Waveren. 2005. http://www.intel.com/cd/ids/developer/asmo-na/eng/293748.htm

(12) Uniform random rotations. Ken Shoemake. In "Graphics Gems III", pp 124-132. Morgan Kaufmann, 1992.

(13) Quaternion in molecular modeling. CFF Karney. J Mol Graph Mod, 25(5):595-604

(14) New method for extracting the quaternion from a rotation matrix. Itzhack Y Bar-Itzhack, J Guid Contr Dynam. 2000. 23(6): 1085-1087.

(15) Multiple View Geometry in Computer Vision. Hartley and Zissermann. Cambridge University Press; 2nd Ed. 2004. Chapter 4, Algorithm 4.7, p 130.

(16) Column Vectors vs. Row Vectors. http://steve.hollasch.net/cgindex/math/matrix/column-vec.html

Examples

```
>>> alpha, beta, gamma = 0.123, -1.234, 2.345
>>> origin, xaxis, yaxis, zaxis = [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]
>>> I = identity_matrix()
>>> Rx = rotation_matrix(alpha, xaxis)
>>> Ry = rotation_matrix(beta, yaxis)
>>> Rz = rotation_matrix(gamma, zaxis)
>>> R = concatenate_matrices(Rx, Ry, Rz)
>>> euler = euler_from_matrix(R, 'rxyz')
>>> numpy.allclose([alpha, beta, gamma], euler)
```

(continues on next page)

[31] https://github.com/numpy/numpy-dtypes

(continued from previous page)

```
True
>>> Re = euler_matrix(alpha, beta, gamma, 'rxyz')
>>> is_same_transform(R, Re)
True
>>> al, be, ga = euler_from_matrix(Re, 'rxyz')
>>> is_same_transform(Re, euler_matrix(al, be, ga, 'rxyz'))
True
>>> qx = quaternion_about_axis(alpha, xaxis)
>>> qy = quaternion_about_axis(beta, yaxis)
>>> qz = quaternion_about_axis(gamma, zaxis)
>>> q = quaternion_multiply(qx, qy)
>>> q = quaternion_multiply(q, qz)
>>> Rq = quaternion_matrix(q)
>>> is_same_transform(R, Rq)
True
>>> S = scale_matrix(1.23, origin)
>>> T = translation_matrix([1, 2, 3])
>>> Z = shear_matrix(beta, xaxis, origin, zaxis)
>>> R = random_rotation_matrix(numpy.random.rand(3))
>>> M = concatenate_matrices(T, R, Z, S)
>>> scale, shear, angles, trans, persp = decompose_matrix(M)
>>> numpy.allclose(scale, 1.23)
True
>>> numpy.allclose(trans, [1, 2, 3])
True
>>> numpy.allclose(shear, [0, math.tan(beta), 0])
True
>>> is_same_transform(R, euler_matrix(axes='sxyz', *angles))
True
>>> M1 = compose_matrix(scale, shear, angles, trans, persp)
>>> is_same_transform(M, M1)
True
>>> v0, v1 = random_vector(3), random_vector(3)
>>> M = rotation_matrix(angle_between_vectors(v0, v1), vector_product(v0, v1))
>>> v2 = numpy.dot(v0, M[:3,:3].T)
>>> numpy.allclose(unit_vector(v1), unit_vector(v2))
True
```

aloe.fit package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.fit.ebsp_detection_model module

aloe.image package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.image.arbse module

aloe.image.downsample module

Downsample a numpy array. Use for binning of images.

This code is (c) Adam Ginsburg (agpy)

Image Tools from: https://github.com/keflavich/image_tools

https://github.com/keflavich/image_tools/blob/master/image_tools/downsample.py

aloe.image.downsample.**downsample**(*myarr*, *factor*, *estimator=<function nanmean>*)
Downsample a 2D array by averaging over *factor* pixels in each axis. Crops upper edge if the shape is not a multiple of factor.

This code is pure np and should be fast.

keywords:

estimator - default to mean. You can downsample by summing or something else if you want a different estimator (e.g., downsampling error: you want to sum & divide by sqrt(n))

aloe.image.downsample.**downsample_1d**(*myarr*, *factor*, *estimator=<function nanmean>*)
Downsample a 1D array by averaging over *factor* pixels. Crops right side if the shape is not a multiple of factor.

This code is pure np and should be fast.

keywords:

estimator - default to mean. You can downsample by summing or something else if you want a different estimator (e.g., downsampling error: you want to sum & divide by sqrt(n))

aloe.image.downsample.**downsample_axis**(*myarr*, *factor*, *axis*, *estimator=<function nanmean>*, *truncate=False*)
Downsample an ND array by averaging over *factor* pixels along an axis. Crops right side if the shape is not a multiple of factor.

This code is pure np and should be fast.

keywords:

estimator - default to mean. You can downsample by summing or something else if you want a different estimator (e.g., downsampling error: you want to sum & divide by sqrt(n))

aloe.image.downsample.**downsample_cube**(*myarr*, *factor*, *ignoredim=0*)
Downsample a 3D array by averaging over *factor* pixels on the last two axes.

aloe.image.filterfft module

2D image filter of numpy arrays, via FFT.

Connelly Barnes, public domain 2007.

aloe.image.filterfft.**filter**(*I*, *K*, *cache=None*)
Filter image I with kernel K.

Image color values outside I are set equal to the nearest border color on I.

To filter many images of the same size with the same kernel more efficiently, use:

```
>>> cache = []
>>> filter(I1, K, cache)
>>> filter(I2, K, cache)
...
```

An even width filter is aligned by centering the filter window around each given output pixel and then rounding down the window extents in the x and y directions.

aloe.image.filterfft.**gaussian**(*sigma=0.5*, *shape=None*)
Gaussian kernel numpy array with given sigma and shape.

The shape argument defaults to ceil(6*sigma).

aloe.image.filterfft.**test**()

aloe.image.kikufilter module

aloe.image.npa module

aloe.image.npa.**get_npa**(*x*, *y*, *patterns*, *indexmap*, *nn=0*)
get neighbor pattern average

use -nn..+nn neighbors, i.e. at nn=1, there will be 8 neighbors = 9 pattern average

patterns in assumed to be 1D array of 2D patterns index of pattern as function of x,y is in indexmap

aloe.image.nxcc module

aloe.image.utils module

aloe.io package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.io.mapexplorer module

aloe.io.patternproviders module

aloe.io.progress module

aloe.io.progress.**print_progress_line**(*tstart*, *i*, *imax*, *every=100*)
print simple progress info line

i and imax are the for-loop-values using a python range, starting from 0

aloe.math package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Subpackages

aloe.math.trans package

Submodules

aloe.math.trans.transformations module

Homogeneous Transformation Matrices and Quaternions.

A library for calculating 4x4 matrices for translating, rotating, reflecting, scaling, shearing, projecting, orthogonalizing, and superimposing arrays of 3D homogeneous coordinates as well as for converting between rotation matrices, Euler angles, and quaternions. Also includes an Arcball control object and functions to decompose transformation matrices.

Author Christoph Gohlke[32]

Organization Laboratory for Fluorescence Dynamics, University of California, Irvine

Version 2017.02.17

Requirements

- CPython 2.7 or 3.5[33]
- Numpy 1.11[34]
- Transformations.c 2017.02.17[35] (recommended for speedup of some functions)

Notes

The API is not stable yet and is expected to change between revisions.

This Python code is not optimized for speed. Refer to the transformations.c module for a faster implementation of some functions.

Documentation in HTML format can be generated with epydoc.

Matrices (M) can be inverted using numpy.linalg.inv(M), be concatenated using numpy.dot(M0, M1), or transform homogeneous coordinate arrays (v) using numpy.dot(M, v) for shape (4, *) column vectors, respectively numpy.dot(v, M.T) for shape (*, 4) row vectors ("array of points").

This module follows the "column vectors on the right" and "row major storage" (C contiguous) conventions. The translation components are in the right column of the transformation matrix, i.e. M[:3, 3]. The transpose of the transformation matrices may have to be used to interface with other graphics systems, e.g. with OpenGL's glMultMatrixd(). See also [16].

Calculations are carried out with numpy.float64 precision.

[32] http://www.lfd.uci.edu/~gohlke/
[33] http://www.python.org
[34] http://www.numpy.org
[35] http://www.lfd.uci.edu/~gohlke/

Vector, point, quaternion, and matrix function arguments are expected to be "array like", i.e. tuple, list, or numpy arrays.

Return types are numpy arrays unless specified otherwise.

Angles are in radians unless specified otherwise.

Quaternions w+ix+jy+kz are represented as [w, x, y, z].

A triple of Euler angles can be applied/interpreted in 24 ways, which can be specified using a 4 character string or encoded 4-tuple:

Axes 4-string: e.g. 'sxyz' or 'ryxy'

- first character : rotations are applied to 's'tatic or 'r'otating frame
- remaining characters : successive rotation axis 'x', 'y', or 'z'

Axes 4-tuple: e.g. (0, 0, 0, 0) or (1, 1, 1, 1)

- inner axis: code of axis ('x':0, 'y':1, 'z':2) of rightmost matrix.
- parity : even (0) if inner axis 'x' is followed by 'y', 'y' is followed by 'z', or 'z' is followed by 'x'. Otherwise odd (1).
- repetition : first and last axis are same (1) or different (0).
- frame : rotations are applied to static (0) or rotating (1) frame.

Other Python packages and modules for 3D transformations and quaternions:

- **Transforms3d**[36] includes most code of this module.
- Blender.mathutils[37]
- numpy-dtypes[38]

References

(1) Matrices and transformations. Ronald Goldman. In "Graphics Gems I", pp 472-475. Morgan Kaufmann, 1990.

(2) More matrices and transformations: shear and pseudo-perspective. Ronald Goldman. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

(3) Decomposing a matrix into simple transformations. Spencer Thomas. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

(4) Recovering the data from the transformation matrix. Ronald Goldman. In "Graphics Gems II", pp 324-331. Morgan Kaufmann, 1991.

(5) Euler angle conversion. Ken Shoemake. In "Graphics Gems IV", pp 222-229. Morgan Kaufmann, 1994.

(6) Arcball rotation control. Ken Shoemake. In "Graphics Gems IV", pp 175-192. Morgan Kaufmann, 1994.

(7) Representing attitude: Euler angles, unit quaternions, and rotation vectors. James Diebel. 2006.

(8) A discussion of the solution for the best rotation to relate two sets of vectors. W Kabsch. Acta Cryst. 1978. A34, 827-828.

(9) Closed-form solution of absolute orientation using unit quaternions. BKP Horn. J Opt Soc Am A. 1987. 4(4):629-642.

[36] https://pypi.python.org/pypi/transforms3d
[37] http://www.blender.org/api/blender_python_api
[38] https://github.com/numpy/numpy-dtypes

(10) Quaternions. Ken Shoemake. http://www.sfu.ca/~jwa3/cmpt461/files/quatut.pdf

(11) From quaternion to matrix and back. JMP van Waveren. 2005. http://www.intel.com/cd/ids/developer/asmo-na/eng/293748.htm

(12) Uniform random rotations. Ken Shoemake. In "Graphics Gems III", pp 124-132. Morgan Kaufmann, 1992.

(13) Quaternion in molecular modeling. CFF Karney. J Mol Graph Mod, 25(5):595-604

(14) New method for extracting the quaternion from a rotation matrix. Itzhack Y Bar-Itzhack, J Guid Contr Dynam. 2000. 23(6): 1085-1087.

(15) Multiple View Geometry in Computer Vision. Hartley and Zissermann. Cambridge University Press; 2nd Ed. 2004. Chapter 4, Algorithm 4.7, p 130.

(16) Column Vectors vs. Row Vectors. http://steve.hollasch.net/cgindex/math/matrix/column-vec.html

Examples

```
>>> alpha, beta, gamma = 0.123, -1.234, 2.345
>>> origin, xaxis, yaxis, zaxis = [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]
>>> I = identity_matrix()
>>> Rx = rotation_matrix(alpha, xaxis)
>>> Ry = rotation_matrix(beta, yaxis)
>>> Rz = rotation_matrix(gamma, zaxis)
>>> R = concatenate_matrices(Rx, Ry, Rz)
>>> euler = euler_from_matrix(R, 'rxyz')
>>> numpy.allclose([alpha, beta, gamma], euler)
True
>>> Re = euler_matrix(alpha, beta, gamma, 'rxyz')
>>> is_same_transform(R, Re)
True
>>> al, be, ga = euler_from_matrix(Re, 'rxyz')
>>> is_same_transform(Re, euler_matrix(al, be, ga, 'rxyz'))
True
>>> qx = quaternion_about_axis(alpha, xaxis)
>>> qy = quaternion_about_axis(beta, yaxis)
>>> qz = quaternion_about_axis(gamma, zaxis)
>>> q = quaternion_multiply(qx, qy)
>>> q = quaternion_multiply(q, qz)
>>> Rq = quaternion_matrix(q)
>>> is_same_transform(R, Rq)
True
>>> S = scale_matrix(1.23, origin)
>>> T = translation_matrix([1, 2, 3])
>>> Z = shear_matrix(beta, xaxis, origin, zaxis)
>>> R = random_rotation_matrix(numpy.random.rand(3))
>>> M = concatenate_matrices(T, R, Z, S)
>>> scale, shear, angles, trans, persp = decompose_matrix(M)
>>> numpy.allclose(scale, 1.23)
True
>>> numpy.allclose(trans, [1, 2, 3])
True
>>> numpy.allclose(shear, [0, math.tan(beta), 0])
True
>>> is_same_transform(R, euler_matrix(axes='sxyz', *angles))
True
>>> M1 = compose_matrix(scale, shear, angles, trans, persp)
```

(continues on next page)

(continued from previous page)

```
>>> is_same_transform(M, M1)
True
>>> v0, v1 = random_vector(3), random_vector(3)
>>> M = rotation_matrix(angle_between_vectors(v0, v1), vector_product(v0, v1))
>>> v2 = numpy.dot(v0, M[:3,:3].T)
>>> numpy.allclose(unit_vector(v1), unit_vector(v2))
True
```

Submodules

aloe.math.euler module

Rotation matrices from Euler Angles

see:

> J. B. Kuipers "Quaternions and Rotation Sequences", Princeton University Press, 1999

aloe.math.euler.**Rx**(*rotation_rad*)

Provide the rotation matrix around the X ($\vec{e_1}$) axis in a cartesian system, with the input rotation angle in radians.

Meaning of Rx acting from left on a COLUMN of VECTORS:

Transformation matrix U for VECTORS. This matrix rotates a set of "old" basis vectors $(\vec{e_1}, \vec{e_2}, \vec{e_3})^T$ (column) by +RotAngle (right hand rule) to a new set of basis vectors $(\vec{e_1}', \vec{e_2}', \vec{e_3}')^T$ (column)

Meaning of Rx acting from left a COLUMN of COORDINATE VALUES:

1. (N_P_O): coordinates of a fixed vector in a "New" basis that is rotated by +RotAngle (passive rotation)

2. (O_P_O): active rotation of vector coordinates in the same "Old" basis by -RotAngle

aloe.math.euler.**Ry**(*RotAngle*)

Provide the rotation matrix around the X ($\vec{e_1}$) axis in a cartesian system, with the input rotation angle in radians.

Meaning of Rx acting from left on a COLUMN of VECTORS:

Transformation matrix U for VECTORS. This matrix rotates a set of "old" basis vectors $(\vec{e_1}, \vec{e_2}, \vec{e_3})^T$ (column) by +RotAngle (right hand rule) to a new set of basis vectors $(\vec{e_1}', \vec{e_2}', \vec{e_3}')^T$ (column)

Meaning of Rx acting from left a COLUMN of COORDINATE VALUES:

1. (N_P_O): coordinates of a fixed vector in a "New" basis that is rotated by +RotAngle (passive rotation)

2. (O_P_O): active rotation of vector coordinates in the same "Old" basis by -RotAngle

aloe.math.euler.**Rz**(*rotation_rad*)

Provide the rotation matrix around the X ($\vec{e_1}$) axis in a cartesian system, with the input rotation angle in radians.

Meaning of Rx acting from left on a COLUMN of VECTORS:

Transformation matrix U for VECTORS. This matrix rotates a set of "old" basis vectors $(\vec{e_1}, \vec{e_2}, \vec{e_3})^T$ (column) by +RotAngle (right hand rule) to a new set of basis vectors $(\vec{e_1}', \vec{e_2}', \vec{e_3}')^T$ (column)

Meaning of Rx acting from left a COLUMN of COORDINATE VALUES:

1. (N_P_O): coordinates of a fixed vector in a "New" basis that is rotated by +RotAngle (passive rotation)

2. (O_P_O): active rotation of vector coordinates in the same “Old” basis by -RotAngle

aloe.math.euler.**Rz2D**(*RotAngle*)
provides the Z axis (e3) rotation matrix in cartesian systems, input “RotAngle” in radians

Transformation matrix U for VECTORS. This matrix rotates a set of “old” basis vectors $(\vec{e_1}, \vec{e_2}, \vec{e_3})^T$ (column) by +RotAngle (right hand rule) to a new set of basis vectors $(\vec{e_1}', \vec{e_2}', \vec{e_3}')^T$ (column)

Meaning of Rx acting from left a COLUMN of COORDINATE VALUES:

1. (N_P_O): coordinates of a fixed vector in a “New” basis that is rotated by +RotAngle (passive rotation)

2. (O_P_O): active rotation of vector coordinates in the same “Old” basis by -RotAngle

aloe.math.euler.**euler_tsl2global**(*phi1_tsl*, *Phi_tsl*, *phi2_tsl*)
transform euler angles form edax-tsl software to global reference system

see: M. Jackson etal. Integrating Materials and Manufacturing Innovation 2014, 3:4 Page 8 of 12
http://www.immijournal.com/content/3/1/4

aloe.math.euler.**rand_rotation_matrix**(*deflection=1.0*, *randnums=None*)
Creates a random rotation matrix.

deflection: the magnitude of the rotation. For 0, no rotation; for 1, competely random rotation. Small deflection => small perturbation. randnums: 3 random numbers in the range [0, 1]. If *None*, they will be auto-generated.

aloe.math.gaussians module

functions for Gaussian peaks

aloe.math.gaussians.**gaussian_fwhm**(*x*, *xo*, *fwhm*)
Gaussian window peak defined by FWHM

aloe.physics package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.physics.reimer module

aloe.physics.reimer.**kikuprofile**(*theta*, *thetaBragg*, *K*, *sharpness=20.0*, *edgedarkness=0.6*, *rounding=100.0*, *asymmetry=1.0*)
model for Kikuchi band profile

aloe.physics.reimer.**reimer2bf**(*theta*, *thetaBragg*, *K*, *sgscale*, *p1*, *p2*)
full two-term two beam Kikuchi band profile as function of theta=angle from lattice plane (middle of Kikuchi band)

K : 1/wavelength gx: reciprocal space vector perpendicular to lattice plane zetag : extinction distance zetaos,zetags: absorption parameters

see: Reimer, SEM p. 351 eq. 9.45

aloe.physics.wave module

aloe.physics.wave.**XWavel**(*energy_keV*)
calculates photon wavelength

input: energy in keV output: wavelength in Angstroms

Notes: E=hc/lambda h*c is 1239.84 eV*nm

aloe.physics.wave.**bragg_angle**(*dhkl*, *wavel*)
Bragg law: lamda = 2*d*sin(theta)

aloe.physics.wave.**wavelength_e**(*Ekin*)
calculates relativistically correct electron wavelength

input: energy in keV output: wavelength in Angstroms

Notes:

based on formula for relativistic kinetic energy Ekin=sqrt((m*c**2)**2+(p*c)**2)-m*c**2 and lambda=h/p

aloe.plotting package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.plotting.make2Dmap module

aloe.plotting.make2Dmap.**make2Dmap**(*Data1D*, *Xidx*, *Yidx*, *NHeight*, *NWidth*)
make 2D map array from 1D Data1D list with index values in Xidx and Yidx final map array height and width are NHeight,NWidth

aloe.plotting.pcplotter module

class aloe.plotting.pcplotter.**PCPlotter**(*primary*, *secondary=None*, *plane_quad=None*, *grid2D=None*, *plane_PC=None*, *BI=None*)
Bases: object

plot(*plotdir='.'*, *show=False*, *plot3d=False*)
plot projection centers

aloe.plotting.pcplotter.**makeLabelList2D**(*bracket1*, *bracket2*, *IndexArray*)
returns the 2D indices formated with chosen bracket

aloe.sys package

crystal

CHECK :copyright: Copyright 2017 :license: BSD, see LICENSE for details.

Submodules

aloe.sys.aloe_logging module

logging configuration for aloe

aloe.sys.winfilever module

7.2 Submodules

aloe.jutils module

aloe.jutils.**show_source**(*obj*)

Display highlighted source code of function in Jupyter notebook.

Use to show source of imported functions etc.

aloe.plots module

8 Installation & Usage

8.1 Installation: aloe

The Jupyter notebooks used for making this documentation depend on the aloe package, which supplies some basic utilities for Kikuchi pattern analysis.

For normal installation of the aloe package, run the following command in the aloebsd directory:

```
python setup.py
```

Installation of the development version, i.e. python links to the code directly in the downloaded aloebsd directory:

```
python setup.py develop
```

Uninstall development version:

```
python setup.py develop --uninstall
```

8.2 Jupyter Notebooks Tips & Tricks

https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/

http://arogozhnikov.github.io/2015/11/29/using-fortran-from-python.html

We often use some standard initialization code in almost all files of this documentation. This includes matplotlib, numpy and image display.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np
```

(continues on next page)

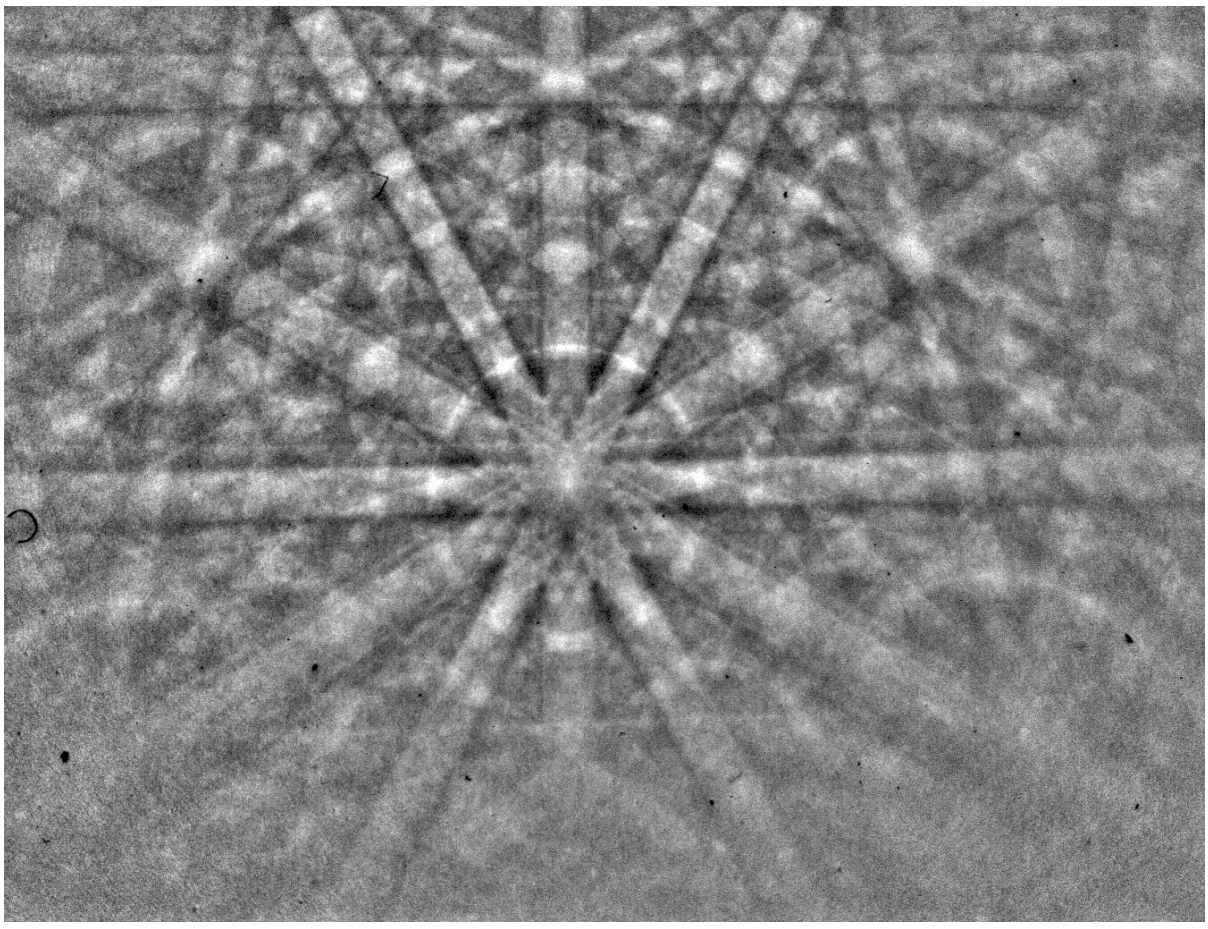
(continued from previous page)

```
# display images from www or local dir
from IPython.core.display import Image, display
```

[2]:
```
# display image from www https://xkcd.com/353/
display(Image(url='https://imgs.xkcd.com/comics/python.png', width=400))
```

```
<IPython.core.display.Image object>
```

[3]:
```
# example for image display from a local directory, make the image larger than screen
display(Image(filename='../data/patterns/GaN_HX_2.png', width=600, unconfined=True))
```



pythreejs Installation

```
git clone https://github.com/jovyan/pythreejs.git
cd pythreejs
pip install -e .
jupyter nbextension install --py --sys-prefix pythreejs
jupyter nbextension enable --py --sys-prefix pythreejs
```

matplotlib Tricks

http://blog.juliusschulz.de/blog/ultimate-ipython-notebook

https://matplotlib.org/examples/pylab_examples/subplots_demo.html

Numpy

Column Vectors vs. Row Vectors

Rememeber, in numpy the innermost array is the fastest changing. Written like below, we have a vector as a 3x1 matrix (column) or 1x3 matrix (row) with shapes (3,1) and (1,3), repsectively. In an n x m matrix, we have n rows of m columns, so n corresponds to "y" (vertical in an image) and m corresponds to "x" (horizontal in image).

https://docs.scipy.org/doc/numpy-1.13.0/reference/internals.html#multidimensional-array-indexing-order-issues

[4]:
```
column_vector = np.array([[1,2,3]]).T
print(column_vector.shape)
```

```
(3, 1)
```

[5]:
```
row_vector = np.array([[1,2,3]])
print(row_vector.shape)
```

```
(1, 3)
```

8.3 Python Documentation Guidelines

nbsphinx:

- https://nbsphinx.readthedocs.io

General Considerations:

- Python Style Guide at Khan Academy[39]

We are using the napoleon[40] extension to sphinx, which handles NumPy and Google style docstrings[41].

Specifications:

- NumPy[42]
- Google style[43]

Examples:

- Google style docstrings[44]
- Numpy style docstrings[45]

Setup tools

https://jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/

8.4 GitHub

Workflow: make private fork and contribute via pull requests:

https://stackoverflow.com/questions/10065526/github-how-to-make-a-fork-of-public-repository-private

[39] https://github.com/Khan/style-guides/blob/master/style/python.md#docstrings
[40] http://www.sphinx-doc.org/en/stable/ext/napoleon.html
[41] http://www.sphinx-doc.org/en/stable/ext/napoleon.html#google-vs-numpy
[42] https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard
[43] https://google.github.io/styleguide/pyguide.html#Comments
[44] http://www.sphinx-doc.org/en/stable/ext/example_google.html#example-google
[45] http://www.sphinx-doc.org/en/stable/ext/example_numpy.html#example-numpy

9 Miscellaneous

9.1 Reproducibility and Reliability of Research

- Nature Special: Challenges in irreproducible research[46]
- Symposium: Reproducibility and reliability of biomedical research: improving research practice (2015)[47]
- Towards Open Science in Acoustics: Foundations and Best Practices (2017)[48]

9.2 General Data Visualization

From the introduction to Ben Fry's Visualizing Data[49]:

> *Acquire* Obtain the data, whether from a file on a disk or a source over a network.
>
> *Parse* Provide some structure for the data's meaning, and order it into categories.
>
> *Filter* Remove all but the data of interest.
>
> *Mine* Apply methods from statistics or data mining as a way to discern patterns or place the data in mathematical context.
>
> *Represent* Choose a basic visual model, such as a bar graph, list, or tree.
>
> *Refine* Improve the basic representation to make it clearer and more visually engaging.
>
> *Interact* Add methods for manipulating the data or controlling what features are visible.

ACQUIRE - PARSE - FILTER - MINE - REPRESENT - REFINE - INTERACT

xcdskd[50] = *Cross-Correlation + Dynamical Simulations + Kikuchi Diffraction*

[46] https://www.nature.com/collections/prbfkwmwvz
[47] https://acmedsci.ac.uk/policy/policy-projects/reproducibility-and-reliability-of-biomedical-research
[48] https://github.com/spatialaudio/DAGA2017_towards_open_science_in_acoustics
[49] https://books.google.de/books?id=6jsVAiULQBgC&dq=Visualizing+Data&source=gbs_navlinks_s
[50] https://github.com/wiai/xcdskd

Python Module Index

a

c

Index

K

M

N

P

R

S

T

W

X