1. Load libraries/functions

In [1]:
```python
import os
import scipy #
import hyperspy.api as hs
import numpy as np
#import copy

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from mpl_toolkits.mplot3d import Axes3D
import mpl_toolkits.mplot3d.axes3d as p3
from matplotlib.colors import to_rgb, LinearSegmentedColormap
import matplotlib.patches as patches

import pandas as pd

import sklearn as skl
from sklearn import preprocessing
from sklearn.decomposition import PCA

import skimage as ski
import skimage.io as io
from skimage import measure
from skimage.filters import threshold_otsu
from skimage.draw import disk
import skimage.morphology as mph

import hdbscan
import seaborn as sns
```

In [3]:
```python
os.chdir('Your:/File/Path')
```

In [3]:
```python
def poisson_noise_norm(signal):
    """normalises hyperspy style signal for poissonian noise. based on [Keenan2004]_.
    Parameters
    ----------'
    signal - a hyperspy data stack
    Returns
    -------
    normalised Signal in a vector format
        The decomposition loadings, as a Signal with same dimension as the original navi
    data_factor_signals : tuple of Signals
    """
    # retreive original data shape
    y, x, e = signal.data.shape
    print('inital mean=', signal.data.mean(),' inital max =', signal.data.max(),' inital
    with signal.unfolded():
            # The rest of the code assumes that the first data axis
            # is the navigation axis. We transpose the data if that
            # is not the case
            #navigation_shapes = np.asarray(signal.axes_manager.navigation_shape).squeez
            #signal_shape = signal.axes_manager.signal_shape # value equal to number of

            if signal.axes_manager[0].index_in_array == 0:

                dc = signal.data.copy()
            else:
                dc = signal.data.T.copy()
```

```
        # make sure dc is correct data type for scaling
        dc = dc.astype('float64')

        aG = dc[:, :].sum(1).squeeze()
        bH = dc[:, :].sum(0).squeeze()
        #print(aG,bH)
        root_aG = np.sqrt(aG)[:, np.newaxis]
        root_bH = np.sqrt(bH)[np.newaxis, :]
        # We ignore numpy's warning when the result of an
        # operation produces nans - instead we set 0/0 = 0
        # Faisal: this is the key line of code that may have been causing problems o
        with np.errstate(divide="ignore", invalid="ignore"):
            # this is quation 8 of (Keenan & Kotula, 2004)
            dc[:, :] /= root_aG * root_bH
            dc[:, :] = np.nan_to_num(dc[:, :])
    print(dc.shape)
    print('scaled mean=',dc.mean(),' scaled max =',dc.max(),' scaled min=',dc.min())

    # convert dc array shape ((y*x), energy_channels) into d_norm (y, x, energy_channels
    d_norm = dc.reshape(y, x, e)

    # convert d_norm numpy array into hyperspy EDSSEMSpectrum
    s_norm = hs.signals.EDSSEMSpectrum(d_norm)
    # copy metadata and axes_manager from original signal
    s_norm.metadata = signal.metadata
    s_norm.axes_manager = signal.axes_manager

    return s_norm, d_norm, dc   # return the hyperspy object s_norm, numpy array d_norm
                                # and vectorised numpy array dc (y*x,energy)
```

In [4]:

```
def flatten_masked_array(im, mask):
    """Flatten an image array containing NaN values, or excluding False values from mask

    Parameters
    ----------
    im - an np array that requires masking (shape = (y, x, ...))
    mask - a binary boolean array (shape = (y, x)), True = data to be included in vect
                      False = NaN values excluded from vect
        Returns:
    --------
    vect - a flattened array from im, excluding NaN values (shape = ((y*x)-(number of Na
    """
    # for 2D images
    if len(im.shape)==2:
        vect = np.empty([])
        vect = np.vstack(im[mask==1])
    # for EDS spectral images
    elif len(im.shape)==3:
        vect = np.empty([])
        vect = np.vstack(im[mask==1, :])
    return vect
```

In [5]:

```
def reconstruct_masked_image(arr, mask, im_shape):
    """Reconstruct an image from a flattened array to contain masked NaN values.
    Parameters
    ----------
    arr - a flattened array, excluding NaN values from mask (shape = ((y*x)-(number of N
    mask - a binary boolean array (shape = (y, x)), True = data belonging to arr, False
    im_shape - tuple of desired image shape (e.g. (y, x, e))
    Returns:
    --------
    im - an image array (shape = (y, x, ...)), containing NaN values where (mask == Fals
    """
    # for 2D images
```

```
    if len(im_shape)==2:
        # find desired image shape
        y_pix = im_shape[0]
        x_pix = im_shape[1]
        # create empty NaN array
        im = np.zeros((y_pix, x_pix))
        im[:] = np.nan
        # replace True values on mask array with the data from the flattened array
        index = 0
        for i in range(0, y_pix):
            for j in range(0, x_pix):
                if mask[i,j]==1:
                    im[i,j] = arr[index]
                    index+=1
    # for EDS spectral images
    elif len(im_shape)==3:
        # find desired image shape
        y_pix = im_shape[0]
        x_pix = im_shape[1]
        e_len = im_shape[2]
        # create empty NaN array
        im = np.zeros((y_pix, x_pix, e_len))
        im[:] = np.nan
        # replace True values on mask array with the data from the flattened array
        index = 0
        for i in range(0, y_pix):
            for j in range(0, x_pix):
                if mask[i,j]==1:
                    im[i,j,:] = arr[index]
                    index+=1
    return im
```

In [6]:

```
def poisson_scale_mask(data):
    """normalises numpy array signal for poissonian noise. based on [Keenan2004]_.
    Parameters
    ----------'
    data - a numpy array
    Returns
    -------
    normalised Signal in a vector format
        The decomposition loadings, as a Signal with same dimension as the original navi
    data_factor_signals : tuple of Signals
    """
    # retreive original data shape
    n, e = data.shape
    print('inital mean=', data.mean(),' inital max =', data.max(),' inital min=', data.m
    dc = np.copy(data)
    # make sure dc is correct data type for scaling
    dc = dc.astype('float64')
    aG = dc[:, :].sum(1)
    bH = dc[:, :].sum(0)
    #print(aG,bH)
    root_aG = np.sqrt(aG)[:, np.newaxis]
    root_bH = np.sqrt(bH)[np.newaxis, :]
    # We ignore numpy's warning when the result of an
    # operation produces nans - instead we set 0/0 = 0
    with np.errstate(divide="ignore", invalid="ignore"):
        # this is quation 8 of (Keenan & Kotula, 2004)
        dc[:, :] /= root_aG * root_bH
        dc[:, :] = np.nan_to_num(dc[:, :])
    print(dc.shape)
    print('scaled mean=',dc.mean(),' scaled max =',dc.max(),' scaled min=',dc.min())

    return dc   # return the hyperspy object s_norm, numpy array d_norm (y,x,energy)
                            # and vectorised numpy array dc (y*x,energy)
```

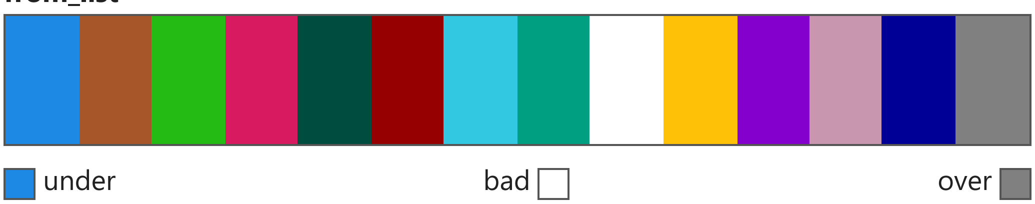
Creating a more accessible color map

These colors are based on pallete shown on http://mkweb.bcgsc.ca/colorblind/palettes.mhtml

In [7]:
```
colors_cbf = np.array((
np.array([30/255, 136/255, 229/255, 1]),
np.array([166/255, 86/255, 40/255, 1]),
np.array([35/255, 187/255, 20/255, 1]),
np.array([216/255, 27/255, 96/255, 1]),
np.array([0/255, 77/255, 64/255, 1]),
np.array([150/255, 0/255, 0/255, 1]),
np.array([50/255, 200/255, 225/255, 1]),
np.array([0/255, 159/255, 129/255, 1]),
np.array([255/255, 255/255, 255/255, 1]),
np.array([255/255, 193/255, 7/255, 1]),
np.array([132/255, 0/255, 205/255, 1]),
np.array([200/255, 150/255, 175/255, 1]),
np.array([0/255, 0/255, 150/255, 1]),
np.array([128/255, 128/255, 128/255, 1]),
))
cbf=matplotlib.colors.ListedColormap(colors_cbf)
matplotlib.colormaps.register(cmap=cbf,name='CBF', force=True)
cbf=matplotlib.colors.ListedColormap(colors_cbf)
cbf
```

Out[7]:

from_list



Change working directory

In [9]:
```
elements = ['C', 'Si', 'O', 'Mg', 'Al', 'Ca', 'Cl', 'Fe', 'Ti', 'K', 'Mn', 'S','V','Cr'
e_colors = ['red', 'wheat', 'green', 'lime', 'fuchsia', 'gold', 'gray', 'magenta', 'fir
```

Load & calibrate raw data

This step is to pre-process the raw data that were exported from Aztec (in .raw and .rpl formats) so that they can be analyzed using Hyperspy

In [16]:
```
s = hs.load('EDS Data.rpl', signal_type='EDS_SEM').T # .T to transpose the data such th
display(s.metadata) # check metadata
display(s.axes_manager) # check axes manager
```

- ▼ Acquisition_instrument
  - ▶ SEM
- ▼ General
  - ▶ FileIO
  - date =
  - original_filename = EDS Data.rpl
  - time =
  - title =

▼ Signal

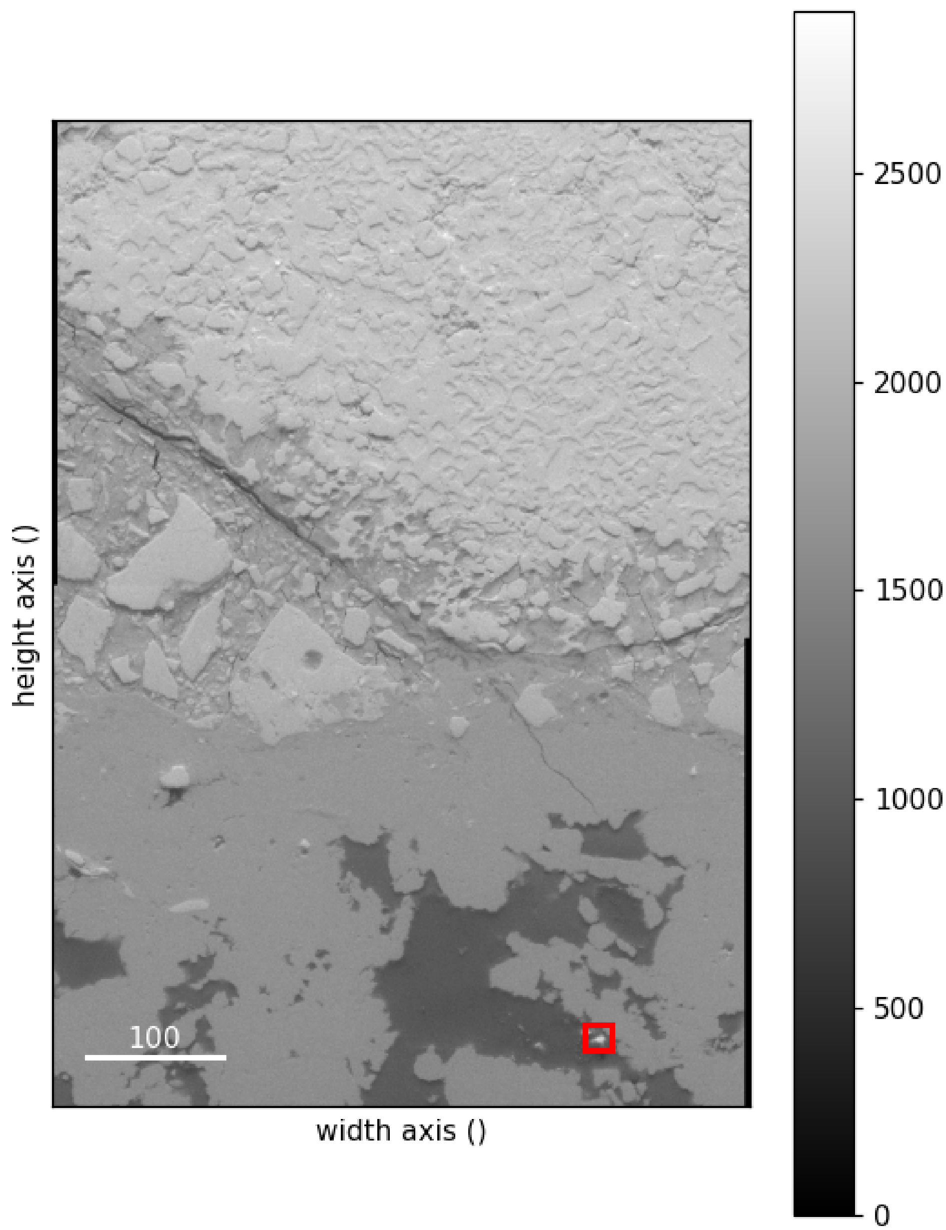
- signal_type = EDS_SEM

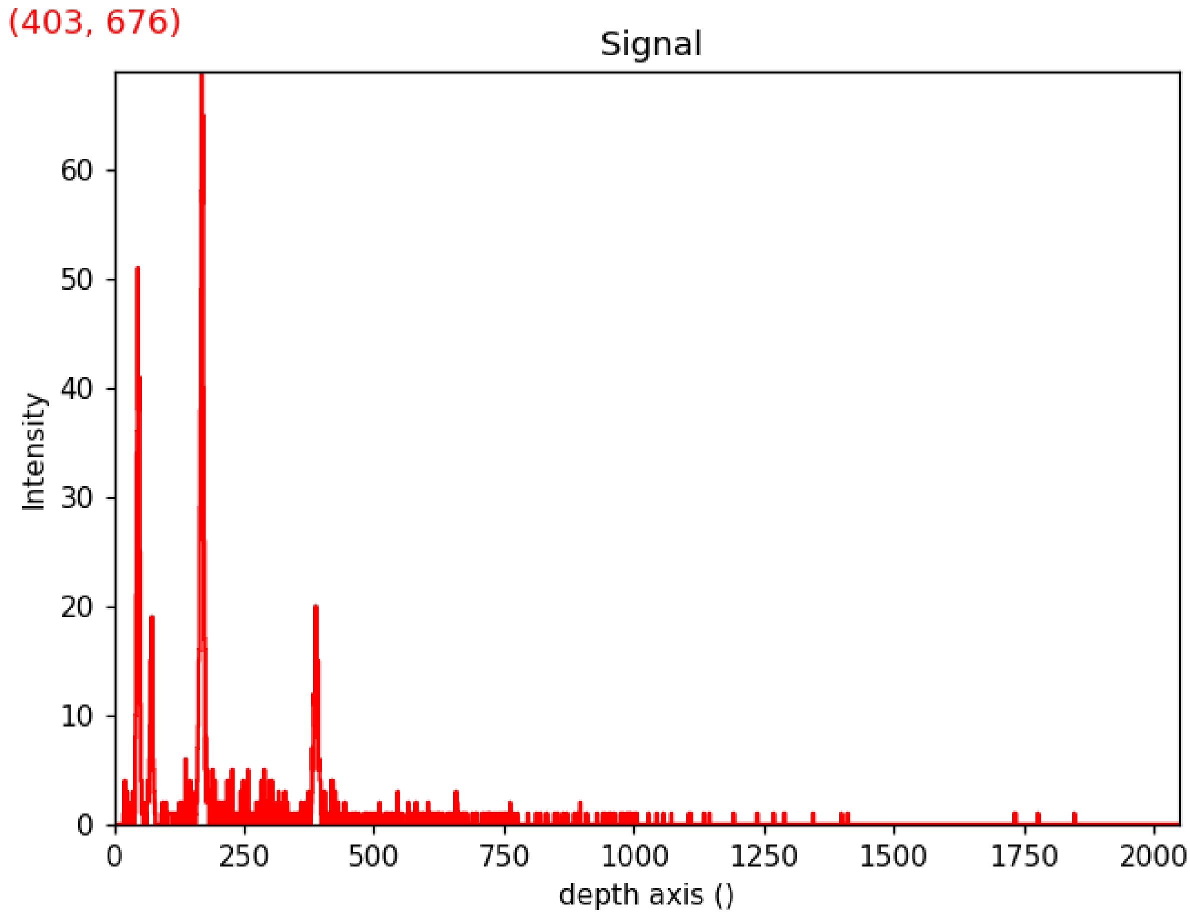
< Axes manager, axes: (516, 728|2048) >

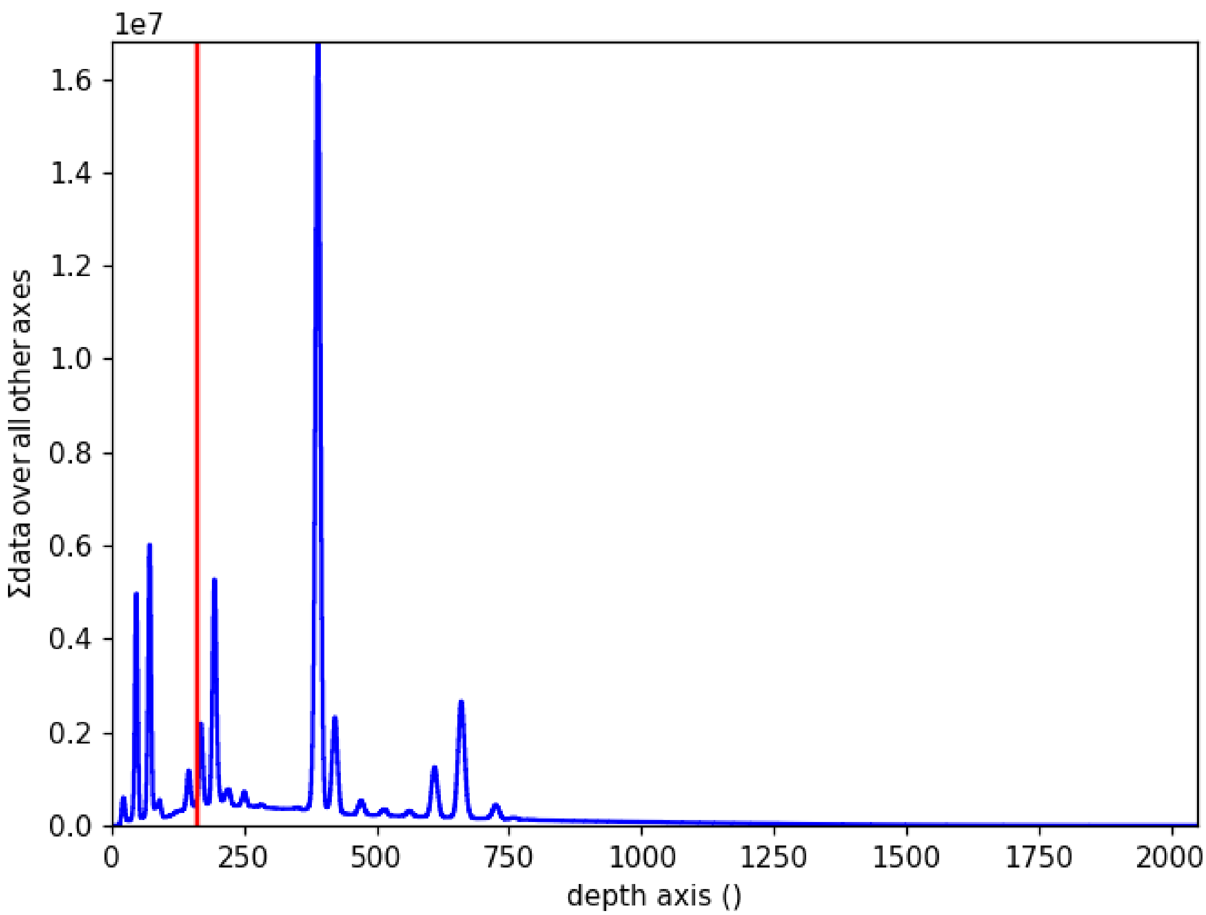
Navigation axis name	size	index	offset	scale	units
width	516	0	0.0	1.0	
height	728	0	0.0	1.0	

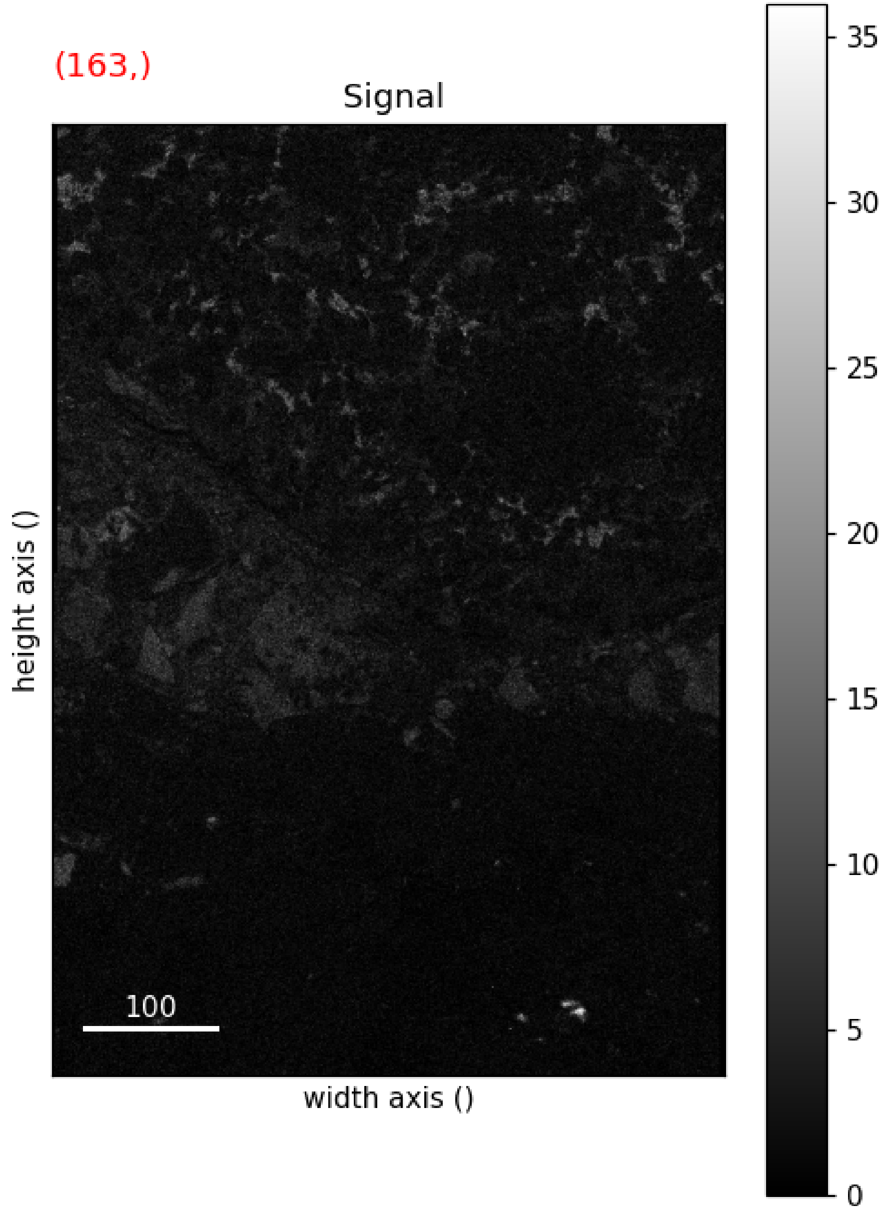
Signal axis name	size	offset	scale	units
depth	2048	0.0	1.0	

In [17]:
```
%matplotlib notebook
s.plot()
s.T.plot()
```




(403, 676)
Signal
Intensity
depth axis ()


1e7
Σdata over all other axes
depth axis ()



From the axes_manager and metadata we can see that none of the calibration has been exported with the raw data. We will need to calibrate the data ourselves. As a result, when we plot the data, none of the energy channels are calibrated to show their corresponding energy in kV.

In [18]:
```
# load the metadata text file(can be obtained from Aztec), create dictionary of values
met_dic = {} # create empty dictionary
with open('meta_data.txt', mode='r') as metd: # open the text file from directory
    lines = metd.readlines()
    for l in lines: # for each line of text file
        (key, val) = l.split(sep=':\t', maxsplit=1) # seperate values from keys based c
        met_dic[key] = val[:-1]

    metd.close() # close text file

for key in met_dic:
    print(key, ':', met_dic[key])
```

```
Label : EDS Montaged Map Data
Collected : 14/04/2023 09:44:54
Resolution (Width) : 516 pixels
Resolution (Height) : 728 pixels
Map Width : 734Î¼m
Map Height : 1040Î¼m
```

```
Accelerating Voltage : 20.0kV
Working Distance : 8.0mm
Number of Completed Frames : 250
Energy Range (keV) : 20 keV
Number Of Channels : 2048
Process Time : 4
Live Time : 18627s
Total Counts in Smart Map : 673337271
Primary Detector :
Primary Detector Serial Number : UVA7677
```

In [19]:

```
# rename the axes and identify the units
s.axes_manager[0].name = 'X'
s.axes_manager['X'].units = 'um'
s.axes_manager['X'].scale = (int(met_dic['Map Width'][:-3]))/int(met_dic['Resolution (W
s.axes_manager[1].name = 'Y'
s.axes_manager['Y'].units = 'um'
s.axes_manager['Y'].scale = s.axes_manager['X'].scale # set y pixel scale equal to x sc
s.axes_manager[2].name = 'E'
s.axes_manager
```

Out[19]: **< Axes manager, axes: (516, 728|2048) >**

Navigation axis name	size	index	offset	scale	units
X	516	403	0.0	1.4224806201550388	um
Y	728	676	0.0	1.4224806201550388	um

Signal axis name	size		offset	scale	units
E	2048		0.0	1.0	

Now the axis manager reflects the true pixel scale, units, and x/y axis length.

Next, we need to calibrate the scale and offset for the energuy axis.

In [20]:

```
spec = s.sum(axis=(0,1))
spec.axes_manager
```

Out[20]: **< Axes manager, axes: (|2048) >**

Signal axis name	size		offset	scale	units
E	2048		0.0	1.0	

In [24]:

```
spec.axes_manager[0].name = 'E'
spec.axes_manager['E'].units = 'keV'
```

In [42]:

```
spec.set_elements(elements)  # add expected bulk elements
spec.add_lines()
```

In [25]:

```
#Find the peaks of the spectra
spec_peaks = spec.find_peaks1D_ohaver(medfilt_radius=3,maxpeakn=25, slope_thresh=10,pea
print(len(spec_peaks[0]))
spec_peaks
```

```
[                                        ] | 0% Completed |  0.0s
D:\Faisal\lib\site-packages\scipy\signal\signaltools.py:1531: UserWarning: kernel_size
exceeds volume extent: the volume will be zero-padded.
  warnings.warn('kernel_size exceeds volume extent: the volume will be '
[########################################] | 100% Completed |  0.5s
```

```
25
```

Out[25]:
```
array([array([( 22.51164436,  331.31269515,  10.23773266),
              ( 46.60453518,  800.91770159,   9.83336707),
              ( 71.82072713,  868.63308137,  10.57114428),
              ( 89.69874538,  305.11076514,  18.77273629),
              (145.48035337,  429.58214529,  15.58706326),
              (168.78535845,  561.85333004,  14.64875249),
              (194.13205246,  828.45258158,  13.48715628),
              (219.68976141,  361.45311795,  30.29385575),
              (250.69657604,  347.13881554,  25.14692417),
              (262.63743202,  273.70680682, 162.98845523),
              (282.20117199,  283.23891719,  44.91693295),
              (298.53994391,  263.59349506,          nan),
              (337.24915128,  256.8054582 ,          nan),
              (350.74092227,  266.95163447,  52.75059048),
              (389.42293001, 1375.35655666,  16.50558867),
              (421.57386838,  579.97760448,  18.89908944),
              (470.92214525,  307.38445467,  25.40074194),
              (514.42859732,  254.69744467,  32.6993603 ),
              (536.48537301,  204.71583328, 115.70901082),
              (542.98480793,  204.75254147, 362.16515576),
              (561.73822003,  242.41251949,  34.36429775),
              (609.58829734,  444.05830751,  21.68255924),
              (660.09155176,  616.55522025,  21.49355483),
              (725.23911443,  281.95214712,  26.93319622),
              (758.89691957,  185.25704738,  44.06222594)],
             dtype=[('position', '<f8'), ('height', '<f8'), ('width', '<f8')])],
      dtype=object)
```

In [26]:
```
#Find the locations of the peaks
poss_lines = []
lines_dict = {}
for i in range(len(spec_peaks[0])):
    poss_lines.append([spec_peaks[0][i][0]])
    lines_dict[i] = poss_lines
    poss_lines=[]

df = pd.DataFrame.from_dict(lines_dict, orient='index')
print(df.shape)
df
```

```
(25, 1)
```

Out[26]:

	0
0	[22.51164435907148]
1	[46.60453517760035]
2	[71.82072712937354]
3	[89.69874537502818]
4	[145.48035337252588]
5	[168.78535845318072]
6	[194.13205245513342]
7	[219.68976140522588]
8	[250.69657604343067]
9	[262.6374320226018]
10	[282.2011719898437]
11	[298.53994390901136]
12	[337.2491512845072]

13	[350.74092226805743]
14	[389.4229300797633]
15	[421.57386837638603]
16	[470.9221452457856]
17	[514.428597318152]
18	[536.4853730132145]
19	[542.9848079329144]
20	[561.7382200343169]
21	[609.588297343958]
22	[660.0915517603544]
23	[725.2391144255047]
24	[758.8969195700224]

In [39]:

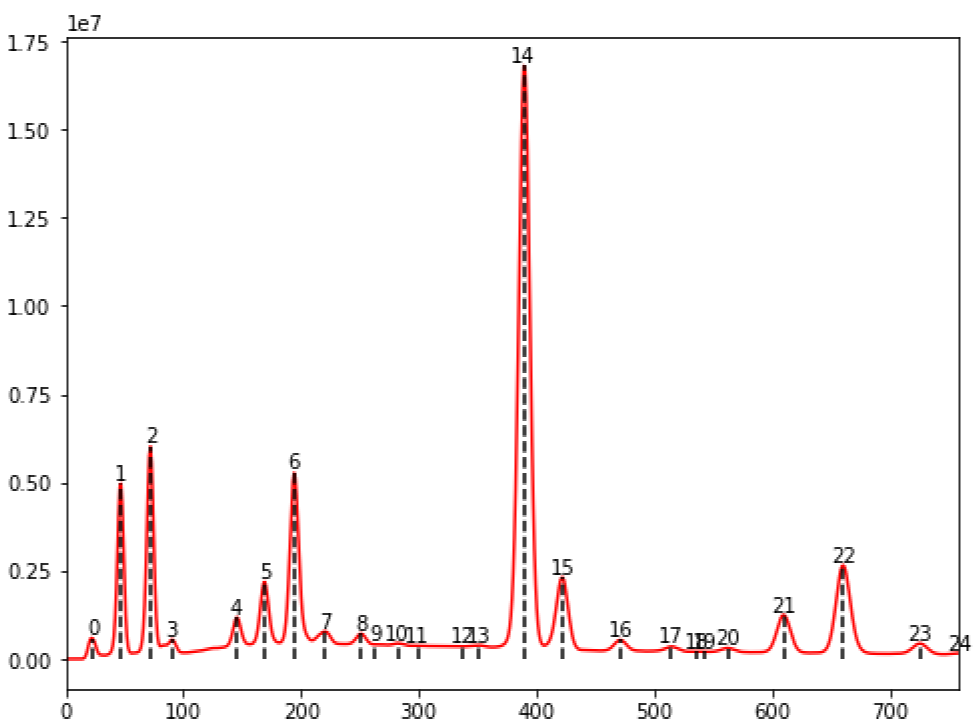
```
```

Out[39]:

```
758
```

In [40]:

```python
# Show these peaks on the spectra to confirm that the peaks identification is accurate
%matplotlib inline
plt.figure(figsize = (8,6))
plt.plot(spec, color='red')
plt.xlim(0, int(df.max()[0][0]))

idx=0
for i in lines_dict:
    xval = round(lines_dict[i][0][0])
    plt.vlines(xval, ymin = 0, ymax = spec.data[xval], linestyle = 'dashed', color = 'b
    plt.text(x=xval, y=spec.data[xval], s=str(idx), ha='center', va='bottom')
    idx+=1
```



In [43]:

```python
ele_lut=hs.material.elements.as_dictionary()
```

```
ele_list=[]
for i in np.arange(0,len(elements)):
    ele_list.append([elements[i],ele_lut[elements[i]]['Atomic_properties']['Xray_lines'
from operator import itemgetter
ele_list.sort(key=itemgetter(1))
print(ele_list)
```

```
[['C', 0.2774], ['O', 0.5249], ['Mg', 1.2536], ['Al', 1.4865], ['Si', 1.7397], ['S', 2.
3072], ['Cl', 2.6224], ['K', 3.3138], ['Ca', 3.6917], ['Ti', 4.5109], ['V', 4.9522],
['Cr', 5.4147], ['Mn', 5.8987], ['Fe', 6.4039]]
```

In [44]:

```
#selected energies/elements lines that we are fitting to
Ener=[ele_list[0][1], ele_list[1][1],ele_list[2][1],ele_list[3][1],ele_list[4][1],
      ele_list[5][1] ,ele_list[8][1],ele_list[9][1],ele_list[10][1],ele_list[11][1],ele
#now pair with the relevant lines
pix=[lines_dict[1][0][0],lines_dict[2][0][0],lines_dict[4][0][0],lines_dict[5][0][0],
     lines_dict[6][0][0],lines_dict[8][0][0],lines_dict[14][0][0], lines_dict[16][0][0]
```

In [45]:

```
#Fit the energy level (which were identified from the peaks identification step) with t
m,b = np.polyfit(pix, Ener , 1)
print(m)
print(b)
print(20/2048)
scipy.stats.linregress(pix, Ener)
```

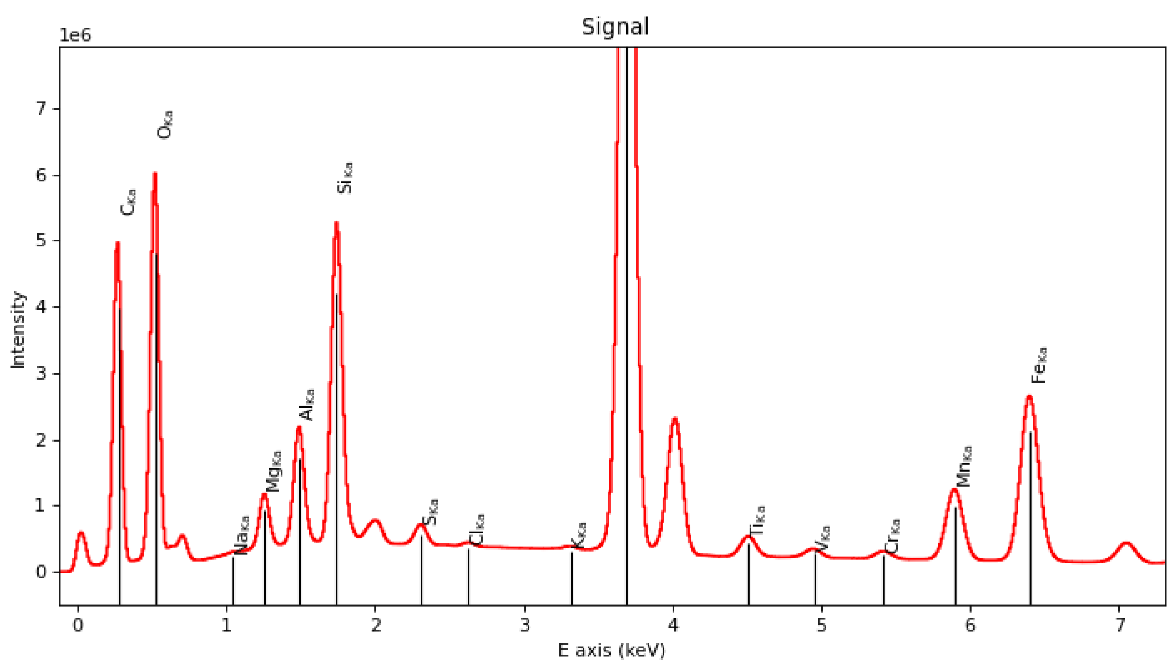
```
0.009996487253191 23
-0.19678069995209044
0.009765625
```

Out[45]:

```
LinregressResult(slope=0.009996487253191228, intercept=-0.1967806999520909, rvalue=0.99
99979190902778, pvalue=3.072665276614814e-28, stderr=6.4489604245970015e-06, intercept_
stderr=0.0025785376815004204)
```

In [47]:

```
# Use the slope and the intercept as scale and offset, respectively.
%matplotlib notebook
spec.axes_manager['E'].scale = m
spec.axes_manager['E'].offset = b
spec.plot(xray_lines=True)
```



In [48]:

```
#add the scale and the offset to the actual data to complete the calibration step
s.axes_manager['E'].scale = m
s.axes_manager['E'].offset = b
```

```
s.axes_manager['E'].units = 'keV'
s.add_elements(elements)
s.add_lines()
s.axes_manager
```

Out[48]: **< Axes manager, axes: (516, 728|2048) >**

Navigation axis name	size	index	offset	scale	units
X	516	403	0.0	1.4224806201550388	um
Y	728	676	0.0	1.4224806201550388	um

Signal axis name	size		offset	scale	units
E	2048		-0.19678069995209044	0.00999648725319123	keV

In [49]:
```
#crop the data so that blank areas are removed
s.crop(axis = 'X', start = 10, end = 500)
```

In [ ]:
```
s.save('s_calib')
```

In [50]:
```
#Upon investigation of the energy levels, it seems that there are no elements that appe
#Therefore, we will crop the data to 1024 since a smaller range requires smaller
s_crop = s.deepcopy()
s_crop.crop(axis = 'E', start = 0, end = 1024)
```

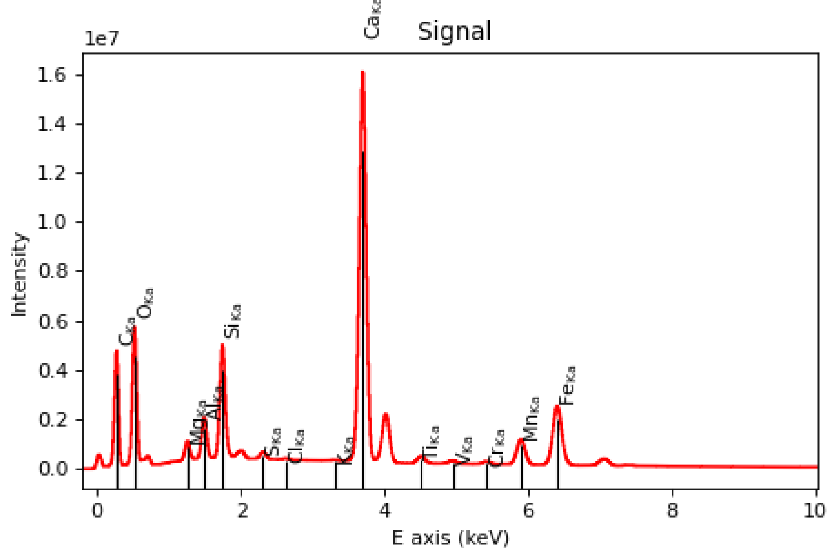
In [51]:
```
s_crop.axes_manager
```

Out[51]: **< Axes manager, axes: (490, 728|1024) >**

Navigation axis name	size	index	offset	scale	units
X	490	0	14.224806201550388	1.4224806201550388	um
Y	728	0	0.0	1.4224806201550388	um

Signal axis name	size		offset	scale	units
E	1024		-0.19678069995209044	0.00999648725319123	keV

In [52]:
```
s_crop.sum(axis=(0,1)).plot(True)
```



In [ ]:
```
s_crop.save('s_crop.hspy')
```

Load pre-calibrated data

In [10]:
```
# # calibrated hs
s_calib = hs.load('s_calib.hspy')
# cropped energy axis
s_crop = hs.load('s_crop.hspy')
# # poisson normalised vector
# d_vect = np.load('DH_poisson_vect.npy')
# pore mask
mask = np.load('PB_mask.npy')
# # poisson scaled masked data
d_msk_norm = np.load('PB_poisson_vect_pore_mask.npy')
labels=np.load('PB_hdbscan_10clus_labels.npy')
bse = io.imread('bse.tif')
print(bse.shape)
bse=bse[0:1456,20:1000]
bse_ds = ski.measure.block_reduce(bse, block_size = (2,2))
y, x, e = s_crop.data.shape
```
```
(1456, 1032)
```

In [11]:
```
bse = io.imread('bse.tif')
print(bse.shape)
bse=bse[0:1456,20:1000]
bse_ds = ski.measure.block_reduce(bse, block_size = (2,2))
```
```
(1456, 1032)
```

In [12]:
```
y, x, e = s_crop.data.shape
```

Poissonian noise scaling

This step is meant to reduce the Poissonian noise in the data

In [ ]:
```
s_crop = hs.load('s_crop.hspy')
```

In [ ]:
```
y, x, e = s_crop.data.shape
```

In [ ]:
```
#This calculation is performed to have an idea about the counts/pixel in the data
```

```
counts = s_crop.data.sum(axis = (0,1,2))
print(f'total map counts: {counts}')
cpp = counts/(y*x)
print(f'counts per pixel: {cpp}')
```

In [ ]:
```
s_crop.sum(axis=(0,1)).plot(True)
```

Mask surface pores by using thresholded BSE image

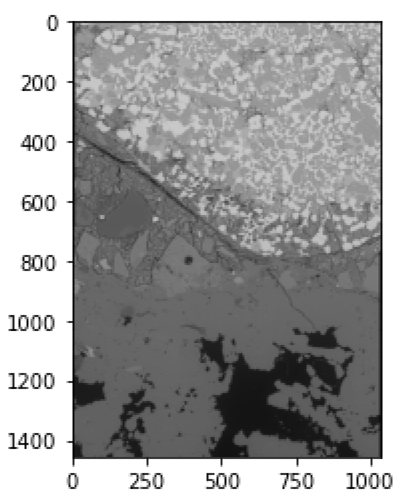
In [55]:
```
%matplotlib inline
```

In [56]:
```
bse = io.imread('bse.tif')
print(bse.shape)
```

```
(1456, 1032)
```

In [57]:
```
plt.imshow(bse, cmap = 'gray')
```

Out[57]:
```
<matplotlib.image.AxesImage at 0x1c098df4790>
```



In [58]:
```
bse=bse[0:1456,20:1000]
```

In [59]:
```
bse_ds = ski.measure.block_reduce(bse, block_size = (2,2)) ##### BLOCK REDUCE REFORMATS
print(bse_ds.shape)
print(s_crop.data.shape)
```

```
(728, 490)
(728, 490, 1024)
```

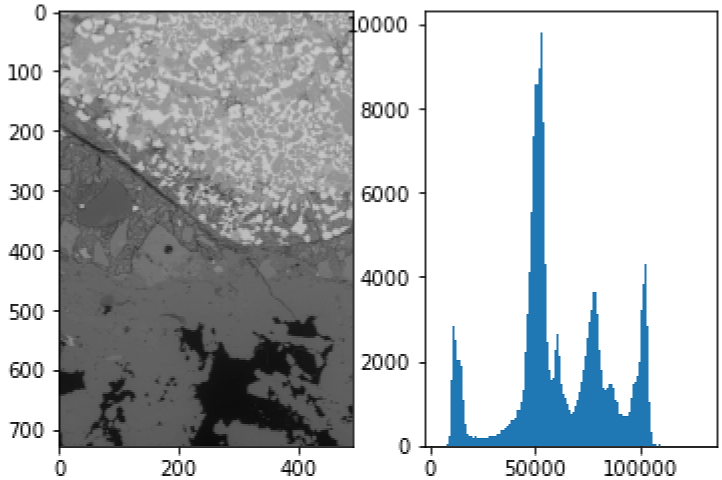
In [65]:
```
bse_ds.dtype
```

Out[65]:
```
dtype('uint32')
```

In [64]:
```
bse.max()
```

Out[64]:
```
32767
```

In [61]:
```
fig, (ax1,ax2) = plt.subplots(1,2)
ax1.imshow(bse_ds, cmap = 'gray')
ax2.hist(bse_ds.reshape(bse_ds.shape[0]*bse_ds.shape[1]), bins = 2**8)
ax2.set_box_aspect(bse_ds.shape[0]/bse_ds.shape[1])

#ax2.set_ylim((0, 8000))
#ax2.set_xlim((0, 2**16))
```

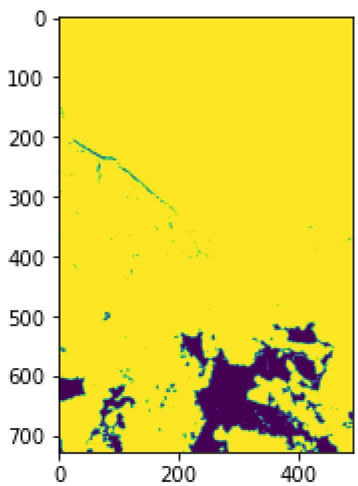


In [67]:
```
#thresh_value = ski.filters.threshold_otsu(bse_ds)
thresh_value =20000
print(thresh_value)
```

```
20000
```

In [68]:
```
mask = np.ones((728,490))
mask[bse_ds<=thresh_value] = 0
plt.figure()
plt.imshow(mask)
```

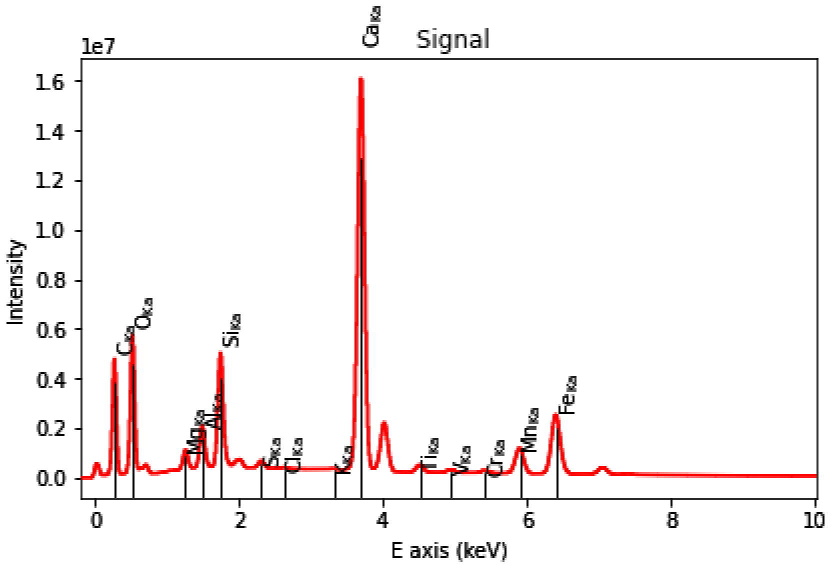
Out[68]:
```
<matplotlib.image.AxesImage at 0x1c09abaa730>
```



In [ ]:
```
np.save('PB_mask.npy', mask)
```

Apply poissonian scaling

In [72]:
```
s_crop.sum(axis=(0,1)).plot(True)
```




```
In [73]: y, x, e= s_crop.data.shape
         print(y,x,e)
```

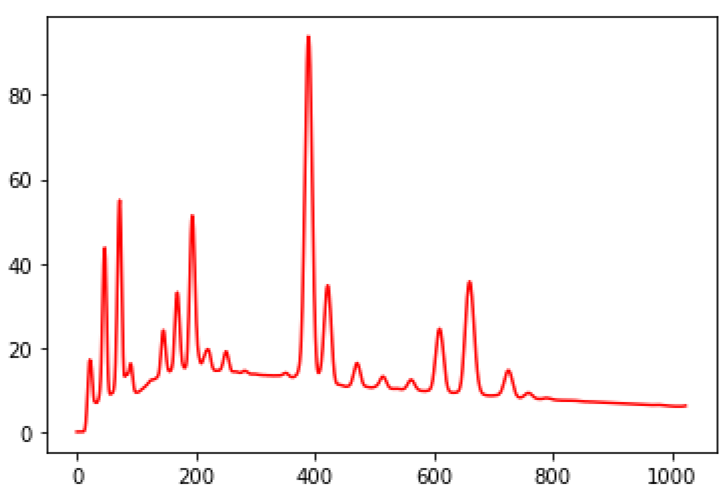
```
728 490 1024
```

```
In [74]: d_msk = flatten_masked_array(s_crop.data, mask)
         d_im = reconstruct_masked_image(d_msk, mask, (y,x,e))
         d_msk_norm = poisson_scale_mask(d_msk)
```

```
inital mean= 1.7683148356347136  inital max = 153  inital min= 0
(327892, 1024)
scaled mean= 4.2529486384219886e-05  scaled max = 0.01395363930464869  scaled min= 0.0
```

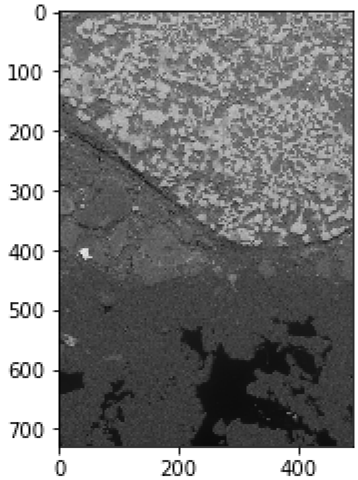
```
In [76]: plt.plot(d_msk_norm.sum(0), color = 'r', label = 'Normalized, pore-masked data')
```

```
Out[76]: [<matplotlib.lines.Line2D at 0x1c0ee5a1c70>]
```



```
In [86]: im_norm = reconstruct_masked_image(d_msk_norm, mask, (y,x,e))

         plt.figure()
         plt.imshow(bse_ds, cmap='gray')
         plt.pcolormesh(im_norm.sum(2), cmap='gray')
```

```
Out[86]: <matplotlib.collections.QuadMesh at 0x1c09d778910>
```



In [ ]:
```
# save masked poisson scaled data
np.save('PB_poisson_vect_pore_mask.npy', d_msk_norm)
```

In [32]:
```
d_msk_norm = np.load('PB_poisson_vect_pore_mask.npy')
```

Dimensionality Reduction

PCA analysis

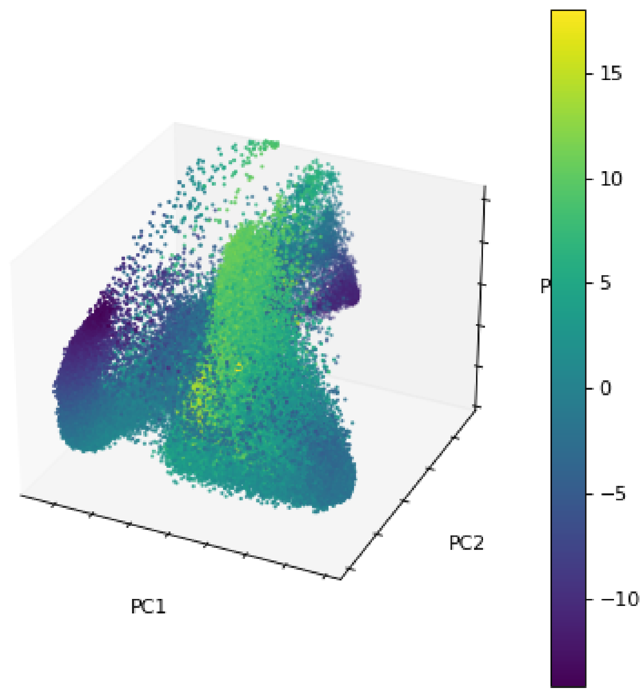
In [34]:
```
#Run the PCA analysis with 100 components first to explore the data
%matplotlib qt5
forpca=d_msk_norm
data_sc=skl.preprocessing.scale(forpca,axis=1)
pca = skl.decomposition.PCA(n_components=100)
pca.fit(data_sc)
f=plt.figure(figsize=(7,5))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```

In [89]:
```
# save the cummulative explained variance for reference
a=np.cumsum(pca.explained_variance_ratio_)
#np.savetxt("PB_PCA.csv", a, delimiter=",")
```

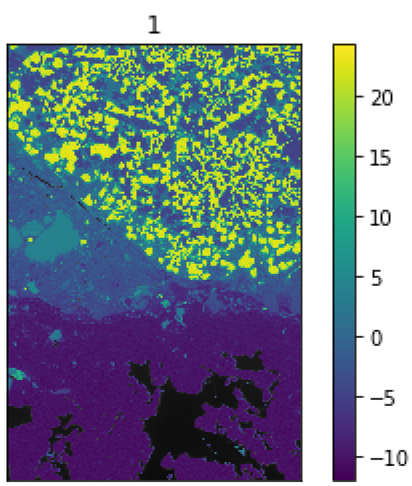
In [35]:
```
# It seems that 8 componenets describes the data well. Re-run the PCA with 8 componenet
pca = skl.decomposition.PCA(n_components=8)
pca.fit(data_sc)
f=plt.figure(figsize=(7,5))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
compz = pca.transform(data_sc)
```

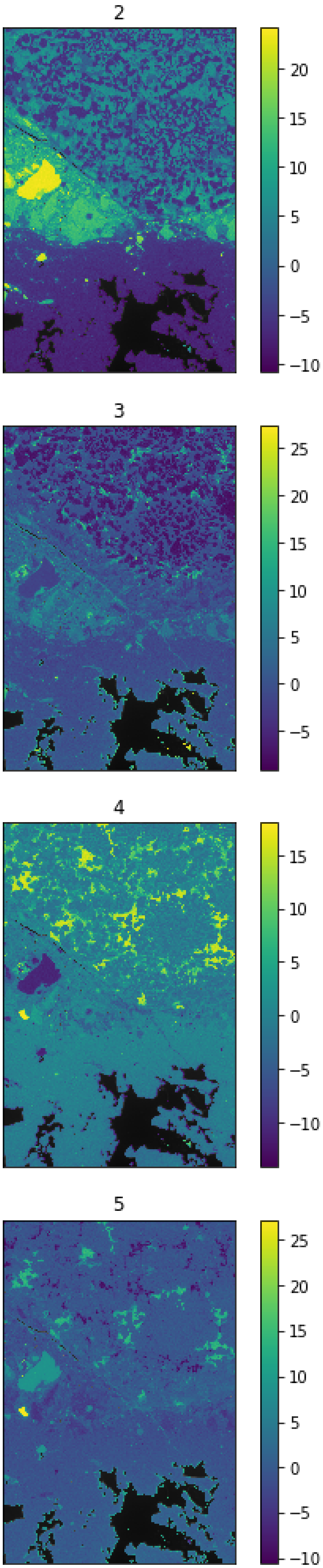
In [92]:
```
#plot the data as a function of PC1,PC2, PC3 and PC4
fig = plt.figure(figsize=(5,5),tight_layout=True)
ax = fig.add_subplot(projection='3d',elev=32, azim=-50)
plot = ax.scatter(compz[:,0],compz[:,1],compz[:,2],c= compz[:,3],cmap='viridis', s= 1)
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
ax.set_ybound(lower=-13,upper=20)
ax.set_ybound(lower=-11,upper=14)
ax.set_zbound(lower=-5,upper=21)
```

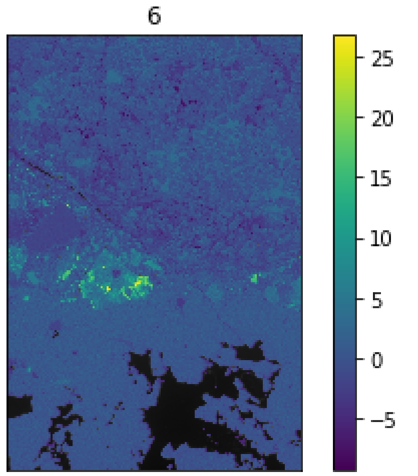
```
cbar = plt.colorbar(mappable = plot, ax = ax)
ax.tick_params(axis='both', top=False, bottom=False, left=False, right=False, labelleft
ax.grid(visible=False)
plt.show()
```

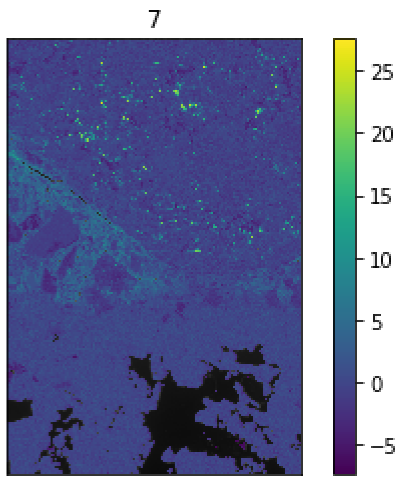


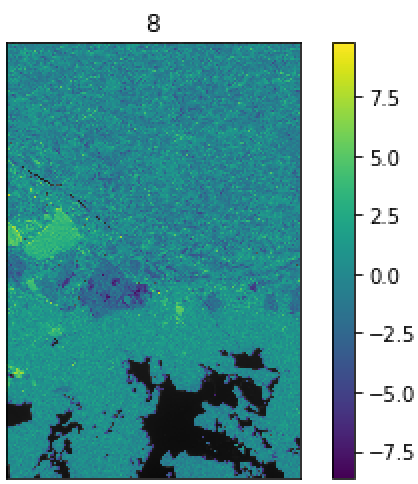
In [94]:
```
#plot the PCs on the images
%matplotlib inline
compz_im = reconstruct_masked_image(compz, mask, (y, x, 8))
for i in range(0,8):
    plt.figure()
    plt.title(i+1)
    plt.imshow(bse_ds, cmap = 'gray')
    plt.pcolormesh(compz_im[:,:,i], cmap=cm.viridis)
    plt.tick_params(top=False, bottom=False, left=False, right=False, labelleft=False,
    cbar = plt.colorbar()
```




2
20
15
10
5
0
−5
−10
3
25
20
15
10
5
0
−5
4
15
10
5
0
−5
−10
5
25
20
15
10
5
0
−5
−10



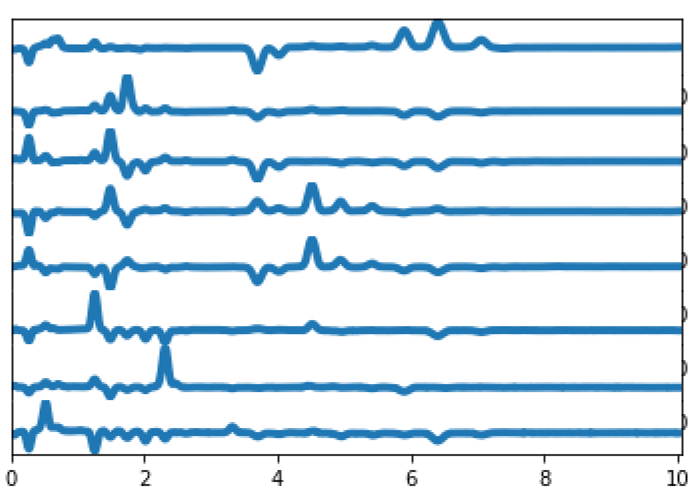




In [36]:
```
pca_loads = pca.components_
```

In [96]:
```
# plot the loadings
scale = s_crop.axes_manager[2].scale
offset = s_crop.axes_manager[2].offset
ofs = offset/scale

#%matplotlib qt5
fig, axs = plt.subplots(8, 1)#, sharex=True)
fig.xlim=[0-ofs , 1024-ofs]
x_label = np.arange(0, 11, 2)
x_ticks = (x_label/scale) - ofs
fig.subplots_adjust(hspace=0)

axs[0].plot(pca_loads[0], linewidth=4)
axs[0].set_yticks([])
axs[0].spines['bottom'].set_visible(False)
```

```
axs[0].set_xlim(0-ofs,1024)

axs[1].plot(pca_loads[1], linewidth=4)
axs[1].set_yticks([])
axs[1].spines['top'].set_visible(False)
axs[1].spines['bottom'].set_visible(False)
axs[1].set_xlim(0-ofs,1024)

axs[2].plot(pca_loads[2], linewidth=4)
axs[2].set_yticks([])
axs[2].spines['top'].set_visible(False)
axs[2].spines['bottom'].set_visible(False)
axs[2].set_xlim(0-ofs,1024)

axs[3].plot(pca_loads[3], linewidth=4)
axs[3].set_yticks([])
axs[3].spines['top'].set_visible(False)
axs[3].spines['bottom'].set_visible(False)
axs[3].set_xlim(0-ofs,1024)

axs[4].plot(pca_loads[4], linewidth=4)
axs[4].set_yticks([])
axs[4].spines['top'].set_visible(False)
axs[4].spines['bottom'].set_visible(False)
axs[4].set_xlim(0-ofs,1024)

axs[5].plot(pca_loads[5],  linewidth=4)
axs[5].set_yticks([])
axs[5].spines['top'].set_visible(False)
axs[5].spines['bottom'].set_visible(False)
axs[5].set_xlim(0-ofs,1024)

axs[6].plot(pca_loads[6], linewidth=4)
axs[6].set_yticks([])
axs[6].spines['top'].set_visible(False)
axs[6].spines['bottom'].set_visible(False)
axs[6].set_xlim(0-ofs,1024)

axs[7].plot(pca_loads[7], linewidth=4)
axs[7].set_yticks([])
axs[7].spines['top'].set_visible(False)
axs[7].set_xlim(0-ofs,1024)
axs[7].set_xticks(x_ticks, x_label)
axs[7].set_xlim(0-ofs,1024)

plt.show()
```



In [37]:

```
scale = s_crop.axes_manager[2].scale
offset = s_crop.axes_manager[2].offset
```

```
ofs = offset/scale
scale, offset, ofs
```

Out[37]:

```
(0.009996487253191223, -0.19678069995209044, -19.684984831973964)
```

Clustering - HDBSCAN

In [38]:

```
clust = hdbscan.HDBSCAN(min_cluster_size=200,min_samples=200,prediction_data=True)
clust.fit(compz)
labels = clust.labels_
print(labels.shape)
# Hard cluster label for each data point, including outlier cluster of '-1'
n_cluster = len(set(labels))
print('Number of clusters:',str(n_cluster))
# Total number of clusters, inclusive of 'outliers' cluster
labels[np.where(labels==-1)[0]]= n_cluster-1
# Assign the largest cluster number as outlier cluster
# calculate % of cluster 4 assigned as outliers
n_outliers = 0
for i in range(0,int(compz.shape[0])):
    if labels[i]==n_cluster-1:
        n_outliers+=1
print('Percent outliers: '+str((n_outliers/compz.shape[0])*100))
```

```
(327892,)
Number of clusters: 14
Percent outliers: 33.28565503275468
```

In [39]:

```
# reconstruct hard cluster assignments
label_map = reconstruct_masked_image(arr = labels, mask = mask, im_shape = (y, x))
# create binary segmentations per cluster
labels_seg = []
for i in range(0, n_cluster):
    labels_seg.append(np.zeros((y, x)))
    labels_seg[i][label_map==i] = 1
labels_seg = np.asarray(labels_seg)
print(labels_seg.shape)
```

```
(14, 728, 490)
```

In [79]:

```
#plot the phase map
#%matplotlib qt5
plt.figure()
plt.imshow(bse_ds, cmap = 'gray')
plt.pcolormesh(label_map, cmap = cbf, alpha = 1)
plt.colorbar()
```

Out[79]:

```
<matplotlib.colorbar.Colorbar at 0x22a72d9fe20>
```

In [ ]:

```
#plot the dendrogram
#%matplotlib inline
plt.figure()
clust.condensed_tree_.plot(cmap='viridis', select_clusters=True, label_clusters = True,
                           selection_palette=sns.color_palette('Accent', n_cluster))
```

Save spectra for back-projection

In [41]:

```
np.save('PB_hdbscan_14clus_labels.npy', labels)
```

In [15]:

```
labels = np.load('PB_hdbscan_10clus_labels.npy')
n_cluster = len(set(labels))
```

In [42]:
```
# create summed spectra as % of total counts per cluster
clus_spec = []
for i in range(0,n_cluster):
    clus_spec.append((s_crop.data[labels_seg[i]==1,:]).sum(axis=0))
```

In [43]:
```
#plot each spectra alone for exploration
#%matplotlib inline
c_idx = 0
for im in labels_seg:
    plt.figure()
    plt.imshow(im)
    plt.title(f'Cluster {c_idx}')
    c_idx+=1
```

In [44]:
```
#plot the spectra of each cluster
#%matplotlib inline
c_idx = 0
for spec in clus_spec:
    plt.figure()
    plt.plot(spec, c = 'firebrick')
    plt.title(f'Cluster {c_idx}')

    c_idx+=1
```

In [45]:
```
#%matplotlib qt5
scale = s_crop.axes_manager[2].scale
offset = s_crop.axes_manager[2].offset
ofs = offset/scale
fig, axs = plt.subplots(7, 1)#, sharex=True)
# Remove vertical space between axes
fig.xlim=[0-ofs , 1024-ofs]
# define xlabels and tick location for 0 to 10 keV (with 2 keV spacing)
x_label = np.arange(0, 11, 2)
x_ticks = (x_label/scale) - ofs
fig.subplots_adjust(hspace=0)
axs[0].plot(clus_spec[12], c = 'mediumblue', linewidth=4)
axs[0].set_yticks([])
axs[0].spines['bottom'].set_visible(False)
axs[0].set_xlim(0-ofs,1024)
axs[1].plot(clus_spec[5], c = 'firebrick',linewidth=4)
axs[1].set_yticks([])
axs[1].spines['top'].set_visible(False)
axs[1].spines['bottom'].set_visible(False)
axs[1].set_xlim(0-ofs,1024)
axs[2].plot(clus_spec[10], c = 'blueviolet',linewidth=4)
axs[2].set_yticks([])
axs[2].spines['top'].set_visible(False)
axs[2].spines['bottom'].set_visible(False)
axs[2].set_xlim(0-ofs,1024)
axs[3].plot(clus_spec[7], c = 'teal',linewidth=4)
axs[3].set_yticks([])
axs[3].spines['top'].set_visible(False)
axs[3].spines['bottom'].set_visible(False)
axs[3].set_xlim(0-ofs,1024)
axs[4].plot(clus_spec[9], c = 'orange',linewidth=4)
axs[4].set_yticks([])
axs[4].spines['top'].set_visible(False)
axs[4].spines['bottom'].set_visible(False)
axs[4].set_xlim(0-ofs,1024)
axs[5].plot(clus_spec[2], c = 'lime',linewidth=4)
axs[5].set_yticks([])
axs[5].spines['top'].set_visible(False)
axs[5].spines['bottom'].set_visible(False)
axs[5].set_xlim(0-ofs,1024)
```

```python
axs[6].plot(clus_spec[13], c = 'dimgray',linewidth=4)
axs[6].set_yticks([])
axs[6].spines['top'].set_visible(False)
#axs[5].spines['bottom'].set_visible(False)
axs[6].set_xlim(0-ofs,1024)
#axs[7].plot(clus_spec[7], c = 'thistle',linewidth=3)
#axs[7].set_yticks([])
#axs[7].spines['bottom'].set_visible(False)
#axs[7].set_xlim(0-ofs,1024)
#axs[8].plot(clus_spec[8], c = 'slateblue',linewidth=3)
#axs[8].set_yticks([])
#axs[8].spines['bottom'].set_visible(False)
#axs[8].set_xlim(0-ofs,1024)
#axs[9].plot(clus_spec[9], c = 'gold',linewidth=3)
#axs[9].set_yticks([])
#axs[9].spines['bottom'].set_visible(False)
#axs[9].set_xlim(0-ofs,1024)
#axs[10].plot(clus_spec[10], c = 'peru',linewidth=3)
#axs[10].set_yticks([])
#axs[10].spines['bottom'].set_visible(False)
#axs[10].set_xlim(0-ofs,1024)
# set x label for energy (keV)
axs[6].set_xticks(x_ticks, x_label)

axs[6].set_xlim(0-ofs,1024)

plt.show()
```

Conversion from .msa to .spx file and exporting to Bruker Espirit format

This step is meant to create spectra files that can be read and analyzed using Bruker Espirit software. The user may need to study the spectra in a given zone, or the spectra in a given cluster.

Selecting a circular zone

In [ ]:
```python
#This step identify a circular region and then sums it up to report it spectra
```

In [ ]:
```python
s_calib.data.shape
```

In [49]:
```python
#%matplotlib qt5
im = s_calib
roi = hs.roi.CircleROI(cx=80,cy=180, r=15*s_calib.axes_manager['X'].scale)
im.plot()
crater = roi.interactive(im)
crater.plot()
```

In [50]:
```python
crater.data.shape
```

Out[50]:
```
(30, 30, 2048)
```

In [51]:
```python
shape = (crater.data.shape[1], crater.data.shape[1])
crater_mask1 = np.zeros(shape, dtype=np.uint8)
```

In [53]:
```python
#%matplotlib inline
plt.figure()
```

```
plt.imshow(crater_mask1)
```

Out[53]:
```
<matplotlib.image.AxesImage at 0x22a682cc1c0>
```

In [54]:
```
rr, cc = disk((15, 15), 14, shape=shape)
crater_mask1[rr, cc] = 1
```

In [55]:
```
#%matplotlib inline
plt.imshow(crater_mask1)
```

Out[55]:
```
<matplotlib.image.AxesImage at 0x22a682ec5e0>
```

In [56]:
```
crater_mask1.shape, crater.data.shape
```

Out[56]:
```
((30, 30), (30, 30, 2048))
```

In [58]:
```
display(crater.axes_manager)
crater.metadata
```

< Axes manager, axes: (30, 30|2048) >

Navigation axis name	size	index	offset	scale	units
X	30	0	59.74418604651163	1.4224806201550388	um
Y	30	0	159.31782945736435	1.4224806201550388	um

Signal axis name	size	offset	scale	units
E	2048	-0.1967806999520904	0.00999648725319123	keV

Out[58]:

- ▼ Acquisition_instrument
  - ▶ SEM
- ▼ General
  - ▶ FileIO
  - title =
- ▼ Signal
  - signal_type = EDS_SEM

In [67]:
```
A = flatten_masked_array(crater.data, crater_mask1)
```

In [68]:
```
plt.plot(A.sum(0))
```

Out[68]:
```
[<matplotlib.lines.Line2D at 0x22a74418d30>]
```

In [71]:
```
s_out = hs.signals.EDSSEMSpectrum(A.sum(0))
s_out.axes_manager[0].name = 'E'
s_out.axes_manager[0].offset = s_calib.axes_manager['E'].offset
s_out.axes_manager[0].scale = s_calib.axes_manager['E'].scale
s_out.axes_manager[0].units = s_calib.axes_manager['E'].units
s_out.add_elements(elements)
s_out.add_lines()
s_out.save(f'Cluster 13N.msa', format='XY')
s_out.plot(True)
```

In [19]:
```
s_calib.axes_manager
```

Out[19]: **< Axes manager, axes: (490, 728|2048) >**

Navigation axis name	size	index	offset	scale	units
X	490	0	14.224806201550388	1.4224806201550388	um
Y	728	0	0.0	1.4224806201550388	um

Signal axis name	size	offset	scale	units
E	2048	-0.1967806999520904	0.00999648725319123	keV

Selecting Data from an entire cluster

In [74]:

```
#This step sums the spectra for a given feature
cluster_of_interest=labels_seg[13]
COI = flatten_masked_array(s_calib.data, cluster_of_interest)
s_out = hs.signals.EDSSEMSpectrum(COI.sum(0))
s_out.axes_manager[0].name = 'E'
s_out.axes_manager[0].offset = s_crop.axes_manager['E'].offset
s_out.axes_manager[0].scale = s_crop.axes_manager['E'].scale
s_out.axes_manager[0].units = s_crop.axes_manager['E'].units
s_out.add_elements(elements)
s_out.add_lines()
s_out.save(f'Cluster 13N.msa', format='XY')
s_out.plot(True)
```

In [78]:

```
i = 0
for cluster in labels_seg:

    cluster_spec = flatten_masked_array(s_calib.data, cluster)
    s_out = hs.signals.EDSSEMSpectrum(cluster_spec.sum(0))
    s_out.axes_manager[0].name = 'E'
    s_out.axes_manager[0].offset = s_crop.axes_manager['E'].offset
    s_out.axes_manager[0].scale = s_crop.axes_manager['E'].scale
    s_out.axes_manager[0].units = s_crop.axes_manager['E'].units
    s_out.add_elements(elements)
    s_out.add_lines()
    s_out.plot(True)
    plt.title(str(i))
    i+=1
```

Image processing

This step is meant to analyze the properties of a given cluster

In [80]:

```
#selecting a cluster of interest for further image analysis
image=labels_seg[5]
image=image.astype('bool')
print(image.dtype)
```

```
bool
```

In [82]:

```
#Remove small objects as these resemble noise
image_cleaned=mph.remove_small_objects(image,2,1)
```

In [83]:

```
fig, axs = plt.subplots(1, 2)
axs[0].imshow(image)
```

```
axs[0].set_title('original')
axs[1].imshow(image_cleaned)
axs[1].set_title('cleaned')
plt.show
```

Out[83]: `<function matplotlib.pyplot.show(*, block=None)>`

In [87]:
```
plt.figure()
plt.imshow(bse_ds, cmap='gray')
#plt.imshow(image_cleaned, alpha=0.1)
```

Out[87]: `<matplotlib.image.AxesImage at 0x22a89614eb0>`

In [88]:
```
labels = measure.label(image_cleaned, connectivity=1)
props = measure.regionprops_table(labels, properties=['label','area', 'equivalent_diame
```

In [90]:
```
#Convert the measurement from pixel to micormeter. Pixel size is 1.4 um.
EqD_um=s_crop.axes_manager['X'].scale*props['equivalent_diameter']
```

In [95]:
```
len(EqD_um)
```

Out[95]: 192

In [92]:
```
#Calculate some features.
np.max(EqD_um), np.min(EqD_um), np.median(EqD_um)
```

Out[92]: (41.732534931881325, 2.2699506497259816, 3.589107114656585)

In [96]:
```
plt.figure()
plt.hist(EqD_um, bins=25)
plt.title('Particle EqD (um) distribution')
plt.xlabel('EqD(um)')
plt.ylabel('Frequency')
```

Out[96]: Text(0, 0.5, 'Frequency')