Spectral_image class guide

isabelpostmes

January 2021

1 Guide

1.1 Loading data

Lets talk through the Spectral_image class. We start by loading a spectral image, saved in a .dm3 or .dm4 file through:

```
>>> im = Spectral_image.load_data('path/to/dmfile.dm4') 1
```

This calls on an alternative constructor, in which the data from the dm-file is loaded, and plugged into the regular constructor. In this function, the loading package `ncempy.io.dm` is used, more info here.

```
81      @classmethod
82      def load_data(cls, path_to_dmfile):
83          """
84          INPUT:
85              path_to_dmfile: str, path to spectral image file (.
                    dm3 or .dm4 extension)
86          OUTPUT:
87              image -- Spectral_image, object of Spectral_image
                    class containing the data of the dm-file
88          """
89          dmfile = dm.fileDM(path_to_dmfile).getDataset(0)
90          data = np.swapaxes(np.swapaxes(dmfile['data'], 0,1), 1,
                2)
91          ddeltaE = dmfile['pixelSize'][0]
92          pixelsize = np.array(dmfile['pixelSize'][1:])
93          energyUnit = dmfile['pixelUnit'][0]
94          ddeltaE *= cls.get_prefix(energyUnit, 'eV')
95          pixelUnit = dmfile['pixelUnit'][1]
96          pixelsize *= cls.get_prefix(pixelUnit, 'm')
97          image = cls(data, ddeltaE, pixelsize = pixelsize)
98          return image
```

Furthermore, we see the `cls.get_prefix()`, which is a small function which recognises the prefix in a unit and transfers it to a numerical value (e.g. 1E3 for k), see lines 870-916 in the complete code. Furthermore, the general constructor is called upon with `cls(data, ddeltaE, pixelsize = pixelsize)`.

The spectral image class starts by defining some constant variables, both class related and physical, which you can find in the complete code. The class constructor takes in at least the data of the spectral image, `data`, and the broadness of the energy loss bins, `deltadeltaE`. Other metadata can be given if known.

Also, the delta_E axis, that is the energy-loss axis is determined by `self.determine_deltaE()`, based upon the broadness of the energy-loss bins and the index at which the average of all spectra in the image has it maximum. The definition of `determine_deltaE()` can be found at line 101 in the complete code. The image axes are determined by `self.calc_axes()`, and output either index arrays, or, if the pixel size is defined, the spacings array in meters. The definition of `calc_axes()` can be found at line 116 in the complete code.

The definitions of some other properties, such as `im.l`, `im.image_shape`, and `im.shape` can be found in the complete code from line 69 onwards.

Furthermore, there are some retrieving functions, such as `im.get_data()`, `im.get_deltaE()`, `im.get_metadata`, and `get_pixel_signal`, which should be quite self-explainatory, but whose definitions can be found in the complete code from line 124 onwards.

```
41  class Spectral_image():

53      def __init__(self, data, deltadeltaE, pixelsize = None,
            beam_energy = None, collection_angle = None, name = None
            ):
54          self.data = data
55          self.ddeltaE = deltadeltaE
56          self.determine_deltaE()
57          if pixelsize is not None:
58              self.pixelsize = pixelsize
59          self.calc_axes()
60          if beam_energy is not None:
61              self.beam_energy = beam_energy
62          if collection_angle is not None:
63              self.collection_angle = collection_angle
64          if name is not None:
65              self.name = name
```

1.2 Preparing data

Now that we have loaded the data, we can perform some operations on the data of the image before starting any calculations, if wished. For example, if we wish to cut the image to a rectangle ranging from pixel a trough pixel b in

width and from pixel c trough pixel d in height, you can simply run (added the +1's to emphasise the excluding nature of the function, for definition of function `cut_image`, see line 154):

```
>>> im.cut_image([a,b+1], [c,d+1])
```

Also, one can, in the future, cut to a certain energy range from `E1` to `E2`, by running (for definition of function `cut`, see line 150):

```
>>> im.cut(E1, E2)
```

Also, one can decide to smooth the spectra. The default smoothing is done by convoluting a length 10 Hanning window, but this can be altered by adding arguments `window_len=` , and `window=` respectively to the call function. Also it should be noted that by defualt, the orginal spectra are disregarded and overwritten by the smoothed signal, to save memory. If you do not want this, add `keep_original=True` to your call-statement. Please note that in this case to call upon your smoothed data, you should call `im.data_smooth` in stead of simply `im.data`. The definition of function `smooth` can be found in the complete code from line 164. For smoothing your data by a moving average over 50 values for example, run:

```
>>> im.smooth(window_len= 50, window = 'flat')
```

1.3 Calculations on image

Now that you have altered your image to your wishes, we can start the calculations on the spectra.

One of the first things you probably want to do (otherwise why are you using this class instead of hyperspy), is loading in the trained ZLPs for the image. This can be done by running (for the definition of function `calc_ZLPs_gen2`, see complete code from line 446):

```
>>> im.calc_ZLPs_gen2()
```

Due to memory considerations, it is not adviced to calculate the ZLPs for each pixel at once, but calculate them per pixel as they are needed. This can be done for pixel at coordinate [i,j] by (for the definition of function `calc_ZLPs`, see complete code from line 446):

```
>>> ZLPs_pixel_i_j = im.calc_ZLPs(i,j)
```

NB: the two functions mentioned above are at the time mere near copies from the code of Laurien, they might/will change significantly when the training of the neural network for ZLPs is incorperated into this class.

When the ZLPs are calculated, one can deconvolute the EEL spectra, to obtain the single scattering distributions. This is done as Egerton explains in his book, and the definition of function `calc_ZLPs` can be found in the complete code from line 202 on. This is also done per pixel and per ZLP, once again for memory considerations.

```
>>> S_E_ijk = im.deconvolute(i,j,ZLP_k)
```

Both the calculation of the ZLPs and the deconvolution of the signal are needed for the evaluation with the Kramers Kronig analysis. The Kramers Kronigs analysis inplemented in our code is the version Egerton explained in his book, with the hyperspy adaptation for tail correction. The definition of function `kramers_kronig_hs` can be found in the complete code from line 481 on. For a single scattering distribution `S_E_ijk` and integrated ZLP_k intensity `N_ZLP_k`, you can call get the dielectric function, thickness, and an approximation of the surface scattering contribution in the single scattering distribution by running:

```
>>> df, t, SS_E = im.kramers_kronig_hs(S_E_ijk, N_ZLP_k)
```

If one wishes to calculate for each pixel the average dielectric function and thickness, you can call upon the `im_dielectric_function` function, whose definition can be found at line 681. It subseqently calculates for each pixel [i,j], for each predicited ZLP_k, the dielectric function and thickness, and saves as an attribute for each pixel the average dielectric function and average thickness, and the standard deviation in both.

```
>>> im.im_dielectric_function()
```

Now that the dielectric function is calculated, we can evaluate it, for example by considering the crossings of the real part of the dielectric function from negative to positive. If the function `crossings_im` (line 756) is called upon, there are two attributes created: `im.crossings_E` and `im.crossings_n`, where in the first all the energy values at which the average dielectric function of each pixel crosses are saved, and in the latter the number of crossings at each pixel.

```
>>> im.crossings_im()
```

1.4 Plotting functions

At this point, two plotting functions are implemented: `plot_sum` (line 804) and `plot_all` (line 835). The first plots the integrated intensity at each pixel.

```
>>> im.plot_sum()
```

With the latter, you can plot all spectra in the image, either in a single plot (default), or in a single image each (set `same_image =False`). Furthermore you can choose a range of pixels for the spectra you want to plot, the range of energy, and change from the IEELS default, to plotting other functions per pixel, such as the average dielectric function. Other functionels can be found in the definition (line 835 complete code).

```
>>> im.plot_all(self, range_x = [10,20], range_y = [0,80],
    range_E = [0.8,5], signal = "dielectric_function")
```

2 Complete code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 18 17:38:57 2020

@author: isabel
"""
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 17 23:12:19 2020

@author: isabel
"""
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 10 01:05:56 2020

@author: isabel
"""

import pandas as pd
import glob
import matplotlib.pyplot as plt
import seaborn as sns
import natsort
import tensorflow.compat.v1 as tf
import seaborn as sns
import numpy as np
#from lmfit import Model
from scipy.fftpack import next_fast_len
import logging
from ncempy.io import dm;

tf.get_logger().setLevel('ERROR')

_logger = logging.getLogger(__name__)


class Spectral_image():
    DIELECTRIC_FUNCTION_NAMES = ['dielectric_function', 'dielectricfunction', '
        dielec_func', 'die_fun', 'df', 'epsilon']
```

```
43      EELS_NAMES = ['electron_energy_loss_spectrum','electron_energy_loss','EELS', 'EEL',
            'energy_loss', 'data']
44      IEELS_NAMES = ['inelastic_scattering_energy_loss_spectrum', '
            inelastic_scattering_energy_loss', 'inelastic_scattering', 'IEELS', 'IES']
45      ZLP_NAMES = ['zeros_loss_peak', 'zero_loss', 'ZLP', 'ZLPs']
46
47      m_0 = 511.06 #eV, electron rest mass
48      a_0 = 5.29E-11 #m, Bohr radius
49      h_bar = 6.582119569E-16 #eV/s
50      c = 2.99792458E8 #m/s
51
52
53      def __init__(self, data, deltadeltaE, pixelsize = None, beam_energy = None,
            collection_angle = None, name = None):
54          self.data = data
55          self.ddeltaE = deltadeltaE
56          self.determine_deltaE()
57          if pixelsize is not None:
58              self.pixelsize = pixelsize
59          self.calc_axes()
60          if beam_energy is not None:
61              self.beam_energy = beam_energy
62          if collection_angle is not None:
63              self.collection_angle = collection_angle
64          if name is not None:
65              self.name = name
66
67
68      #PROPERTIES
69      @property
70      def l(self):
71          return self.data.shape[2]
72      @property
73      def image_shape(self):
74          return self.data.shape[:2]
75
76      @property
77      def shape(self):
78          self.shape = self.data.shape
79
80
81      @classmethod
82      def load_data(cls, path_to_dmfile):
83          """
84          INPUT:
85              path_to_dmfile: str, path to spectral image file (.dm3 or .dm4 extension)
```

```
86          OUTPUT:
87              image -- Spectral_image, object of Spectral_image class containing the data
                    of the dm-file
88          """
89          dmfile = dm.fileDM(path_to_dmfile).getDataset(0)
90          data = np.swapaxes(np.swapaxes(dmfile['data'], 0,1), 1,2)
91          ddeltaE = dmfile['pixelSize'][0]
92          pixelsize = np.array(dmfile['pixelSize'][1:])
93          energyUnit = dmfile['pixelUnit'][0]
94          ddeltaE *= cls.get_prefix(energyUnit, 'eV')
95          pixelUnit = dmfile['pixelUnit'][1]
96          pixelsize *= cls.get_prefix(pixelUnit, 'm')
97          image = cls(data, ddeltaE, pixelsize = pixelsize)
98          return image
99
100
101     def determine_deltaE(self):
102         """
103         INPUT:
104             self
105
106         Determines the delta energies of the spectral image, based on the delta delta
                energie,
107         and the index on which the spectral image has on average the highest intesity,
                this
108         is taken as the zero point for the delta energy.
109         """
110         data_avg = np.average(self.data, axis = (0,1))
111         ind_max = np.argmax(data_avg)
112         self.deltaE = np.linspace(-ind_max * self.ddeltaE, (self.l-ind_max-1)*self.
                ddeltaE, self.l)
113         #return deltaE
114
115
116     def calc_axes(self):
117         self.y_axis = np.linspace(0, self.image_shape[0]-1, self.image_shape[0])
118         self.x_axis = np.linspace(0, self.image_shape[1]-1, self.image_shape[1])
119         if hasattr(self, 'pixelsize'):
120             self.y_axis *= self.pixelsize[0]
121             self.x_axis *= self.pixelsize[1]
122
123     #RETRIEVING FUNCTIONS
124     def get_data(self):
125         return self.data
126
127     def get_deltaE(self):
```

```
128         return self.deltaE
129
130     def get_metadata(self):
131         meta_data = {}
132         if self.beam_energy is not None:
133             meta_data['beam_energy'] = self.beam_energy
134         if self.collection_angle is not None:
135             meta_data['collection_angl'] = self.collection_angle
136         return meta_data
137
138     def get_pixel_signal(self, i,j, signal = 'EELS'):
139         #TODO: add alternative signals + names
140         if signal == 'EELS':
141             return np.copy(self.data[ i, j, :])
142         elif signal == 'df_avg':
143             return np.copy(self.dielectric_function_im_avg[ i, j, :])
144         else:
145             return np.copy(self.data[ i, j, :])
146
147
148     #METHODS ON SIGNAL
149
150     def cut(self, E1, E2):
151         #TODO
152         pass
153
154     def cut_image(self, range_width, range_height):
155         #TODO: add floats for cutting to meter sizes?
156         self.data = self.data[range_height[0]:range_height[1], range_width[0]:
                range_width[1]]
157         self.y_axis = self.y_axis[range_height[0]:range_height[1]]
158         self.x_axis = self.x_axis[range_width[0]:range_width[1]]
159
160     #TODO
161     def samenvoegen(self):
162         pass
163
164     def smooth(self, window_len=10,window='hanning', keep_original = False):
165         """smooth the data using a window with requested size.
166
167         This method is based on the convolution of a scaled window with the signal.
168         The signal is prepared by introducing reflected copies of the signal
169         (with the window size) in both ends so that transient parts are minimized
170         in the begining and end part of the output signal.
171
172         input:
```

```
173             x: the input signal
174             window_len: the dimension of the smoothing window; should be an odd integer
175             window: the type of window from 'flat', 'hanning', 'hamming', 'bartlett', '
                    blackman'
176                 flat window will produce a moving average smoothing.
177 
178         output:
179             the smoothed signal
180 
181         """
182         #TODO: add comnparison
183         window_len += (window_len+1)%2
184         s=np.r_['-1', self.data[:,:,window_len-1:0:-1],self.data,self.data[:,:,-2:-
                window_len-1:-1]]
185 
186         if window == 'flat': #moving average
187             w=np.ones(window_len,'d')
188         else:
189             w=eval('np.'+window+'(window_len)')
190 
191         #y=np.convolve(w/w.sum(),s,mode='valid')
192         surplus_data = int((window_len-1)*0.5)
193         if keep_original:
194             self.data_smooth = np.apply_along_axis(lambda m: np.convolve(m, w/w.sum(),
                    mode='valid'), axis=2, arr=s)[:,:,surplus_data:-surplus_data]
195         else:
196             self.data = np.apply_along_axis(lambda m: np.convolve(m, w/w.sum(), mode='
                    valid'), axis=2, arr=s)[:,:,surplus_data:-surplus_data]
197 
198 
199         return #y[(window_len-1):-(window_len)]
200 
201 
202     def deconvolute(self, i,j, ZLP):
203 
204         y = self.get_pixel_signal(i,j)
205         r = 3 #Drude model, can also use estimation from exp. data
206         A = y[-1]
207         n_times_extra = 2
208         sem_inf = next_fast_len(n_times_extra*self.l)
209 
210 
211 
212         y_extrp = np.zeros(sem_inf)
213         y_ZLP_extrp = np.zeros(sem_inf)
```

```
214         x_extrp = np.linspace(self.deltaE[0]- self.l*self.ddeltaE, sem_inf*self.ddeltaE
                +self.deltaE[0]- self.l*self.ddeltaE, sem_inf)
215
216         x_extrp = np.linspace(self.deltaE[0], sem_inf*self.ddeltaE+self.deltaE[0],
                sem_inf)
217
218         y_ZLP_extrp[:self.l] = ZLP
219         y_extrp[:self.l] = y
220         x_extrp[:self.l] = self.deltaE[-self.l:]
221
222         y_extrp[self.l:] = A*np.power(1+x_extrp[self.l:]-x_extrp[self.l],-r)
223
224         x = x_extrp
225         y = y_extrp
226         y_ZLP = y_ZLP_extrp
227
228         z_nu = CFT(x,y_ZLP)
229         i_nu = CFT(x,y)
230         abs_i_nu = np.absolute(i_nu)
231         N_ZLP = 1#scipy.integrate.cumtrapz(y_ZLP, x, initial=0)[-1]#1 #arbitrary units
                ??? np.sum(EELZLP)
232
233         s_nu = N_ZLP*np.log(i_nu/z_nu)
234         j1_nu = z_nu*s_nu/N_ZLP
235         S_E = np.real(iCFT(x,s_nu))
236         s_nu_nc = s_nu
237         s_nu_nc[500:-500] = 0
238         S_E_nc = np.real(iCFT( x,s_nu_nc))
239         J1_E = np.real(iCFT(x,j1_nu))
240
241         return J1_E[:self.l]
242
243     #METHODS ON ZLP
244     #CALCULATING ZLPs FROM PRETRAINDED MODELS
245     def calculate_general_ZLPs(self, path_to_models):
246         tf.reset_default_graph()
247         #TODO: redifine paths based upon new fitter saving modes
248         #TODO: rewrite to have models as atributes?
249
250         d_string = '07.09.2020'
251         path_to_data = 'Data_oud/Results/%(date)s/'% {"date": d_string}
252
253         path_predict = r'Predictions_*.csv'
254         path_cost = r'Cost_*.csv'
255
256         all_files = glob.glob(path_to_data + path_predict)
```

```
257
258         li = []
259         for filename in all_files:
260             df = pd.read_csv(filename, delimiter=",",  header=0, usecols=[0,1,2], names
                    =['x', 'y', 'pred'])
261             li.append(df)
262
263         training_data = pd.concat(li, axis=0, ignore_index=True)
264
265         self.dE1 = np.round(max(training_data['x'][(training_data['x']< 3)]),2)
266         self.dE2 = np.round(min(training_data['x'][(training_data['x']> 3)]),1)
267         self.dE0 = np.round(self.dE1 - .5, 2)
268
269         all_files_cost = glob.glob(path_to_data + path_cost)
270         all_files_cost_sorted = natsort.natsorted(all_files_cost)
271
272         chi2_array = []
273         chi2_index = []
274
275         for filename in all_files_cost_sorted:
276             df = pd.read_csv(filename, delimiter=",", header=0, usecols=[0,1], names=['
                    train', 'test'])
277             best_try = np.argmin(df['test'])
278             chi2_array.append(df.iloc[best_try,0])
279             chi2_index.append(best_try)
280
281         chi_data  = pd.DataFrame()
282         chi_data['Best chi2 value'] = chi2_array
283         chi_data['Epoch'] = chi2_index
284
285         good_files = []
286         count = 0
287         threshold = 3
288
289         for i,j in enumerate(chi2_array):
290             if j < threshold:
291                 good_files.append(1)
292                 count +=1
293             else:
294                 good_files.append(0)
295
296         tf.get_default_graph()
297         tf.disable_eager_execution()
298         #config = tf.ConfigProto()
299         #config.gpu_options.allow_growth = True
300
```

```
301
302         def make_model(inputs, n_outputs):
303             hidden_layer_1 = tf.layers.dense(inputs, 10, activation=tf.nn.sigmoid)
304             hidden_layer_2 = tf.layers.dense(hidden_layer_1, 15, activation=tf.nn.
                    sigmoid)
305             hidden_layer_3 = tf.layers.dense(hidden_layer_2, 5, activation=tf.nn.relu)
306             output = tf.layers.dense(hidden_layer_3, n_outputs, name='outputs', reuse=
                    tf.AUTO_REUSE)
307             return output
308
309         x = tf.placeholder("float", [None, 1], name="x")
310         predictions = make_model(x, 1)
311
312
313         prediction_file = pd.DataFrame()
314         len_data = self.l
315         predict_x = np.linspace(-0.5, 20, 1000).reshape(1000,1)
316         predict_x = self.deltaE.reshape(self.l,1)
317
318         self.ZLPs_gen = np.zeros((count, len_data))
319         with tf.Session() as sess: #TODO: gives warning
320             sess.run(tf.global_variables_initializer())
321
322             for i in range(0,len(good_files)):
323                 if good_files[i] == 1:
324                     best_model = 'Models_oud/Best_models/%(s)s/best_model_%(i)s'% {'s':
                            d_string, 'i': i}
325                     saver = tf.train.Saver(max_to_keep=1000)
326                     saver.restore(sess, best_model)
327
328                     extrapolation = sess.run(predictions, #TODO: RESTARTS KERNEL!!!!!
329                                              feed_dict={
330                                              x: predict_x
331                                              })
332                     prediction_file['prediction_%(i)s' % {"i": i}] = extrapolation.
                            reshape(len_data,)
333                     self.ZLPs_gen[i, :] = np.exp(extrapolation)#.reshape(len_data,)
334
335
336     @staticmethod
337     def make_model(inputs, n_outputs):
338         hidden_layer_1 = tf.layers.dense(inputs, 10, activation=tf.nn.sigmoid)
339         hidden_layer_2 = tf.layers.dense(hidden_layer_1, 15, activation=tf.nn.sigmoid)
340         hidden_layer_3 = tf.layers.dense(hidden_layer_2, 5, activation=tf.nn.relu)
341         output = tf.layers.dense(hidden_layer_3, n_outputs, name='outputs', reuse=tf.
                AUTO_REUSE)
```

```
342         return output
343
344     def calc_ZLPs_gen2(self,  specimen = 4):
345         tf.reset_default_graph()
346         if specimen == 3:
347             d_string = '06.12.2020'
348             path_to_data = 'Data_oud/Results/sp3/%(date)s/'% {"date": d_string}
349         else:
350             d_string = '07.09.2020'
351             path_to_data = 'Data_oud/Results/%(date)s/'% {"date": d_string}
352
353         path_predict = r'Predictions_*.csv'
354         path_cost = r'Cost_*.csv'
355
356         all_files = glob.glob(path_to_data + path_predict)
357
358         li = []
359         for filename in all_files:
360             df = pd.read_csv(filename, delimiter=",",  header=0, usecols=[0,1,2], names
                    =['x', 'y', 'pred'])
361             li.append(df)
362
363
364         training_data = pd.concat(li, axis=0, ignore_index=True)
365
366
367         all_files_cost = glob.glob(path_to_data + path_cost)
368
369
370         import natsort
371
372         all_files_cost_sorted = natsort.natsorted(all_files_cost)
373
374         chi2_array = []
375         chi2_index = []
376
377         for filename in all_files_cost_sorted:
378             df = pd.read_csv(filename, delimiter=",", header=0, usecols=[0,1], names=['
                    train', 'test'])
379             best_try = np.argmin(df['test'])
380             chi2_array.append(df.iloc[best_try,0])
381             chi2_index.append(best_try)
382
383         chi_data  = pd.DataFrame()
384         chi_data['Best chi2 value'] = chi2_array
385         chi_data['Epoch'] = chi2_index
```

```
386
387
388
389         good_files = []
390         count = 0
391         threshold = 3
392
393         for i,j in enumerate(chi2_array):
394             if j < threshold:
395                 good_files.append(1)
396                 count +=1
397             else:
398                 good_files.append(0)
399
400
401
402
403         tf.get_default_graph
404         tf.disable_eager_execution()
405
406
407
408         x = tf.placeholder("float", [None, 1], name="x")
409         predictions = self.make_model(x, 1)
410
411
412         prediction_file = pd.DataFrame()
413         len_data = self.l
414         predict_x = np.linspace(-0.5, 20, 1000).reshape(1000,1)
415         predict_x = self.deltaE.reshape(len_data,1)
416
417         self.ZLPs_gen = np.zeros((count, len_data))
418         j=0
419         with tf.Session() as sess:
420             sess.run(tf.global_variables_initializer())
421
422             for i in range(0,len(good_files)):
423                 if good_files[i] == 1:
424                     if specimen ==3:
425                         best_model = 'Models_oud/Best_models/sp3/%(s)s/best_model_%(i)s
                                '% {'s': d_string, 'i': i}
426                     else:
427                         best_model = 'Models_oud/Best_models/%(s)s/best_model_%(i)s'% {
                                's': d_string, 'i': i}
428                     saver = tf.train.Saver(max_to_keep=1000)
429                     saver.restore(sess, best_model)
```

```
430
431                     extrapolation = sess.run(predictions,
432                                              feed_dict={
433                                              x: predict_x
434                                              })
435                     #prediction_file['prediction_%(i)s' % {"i": i}] = extrapolation.
                            reshape(1000,)
436                     self.ZLPs_gen[j,:] = np.exp(extrapolation.reshape(len_data,))
437                     prediction_file['prediction_%(i)s' % {"i": i}] = extrapolation.
                            reshape(len_data,)
438                     j += 1
439
440         self.dE1 = np.round(max(training_data['x'][(training_data['x']< 3)]),2)
441         self.dE2 = np.round(min(training_data['x'][(training_data['x']> 3)]),1)
442         self.dE0 = np.round(self.dE1 - .5, 2)
443
444         #return ZLPs_gen, dE0, dE1, dE2
445
446     def calc_ZLPs(self, i,j):
447         ### Definition for the matching procedure
448         signal = self.get_pixel_signal(i,j)
449
450         if not hasattr(self, 'ZLPs_gen'):
451             self.calc_ZLPs_gen2("iets")
452
453         def matching( signal, ind_ZLP):
454             gen_i_ZLP = self.ZLPs_gen[ind_ZLP, :]*np.max(signal)/np.max(self.ZLPs_gen[
                    ind_ZLP,:]) #TODO!!!!, normalize?
455             delta = np.divide((self.dE1 - self.dE0), 3)
456
457             factor_NN = np.exp(- np.divide((self.deltaE[(self.deltaE<self.dE1) & (self.
                    deltaE >= self.dE0)] - self.dE1)**2, delta**2))
458             factor_dm = 1 - factor_NN
459
460             range_0 = signal[self.deltaE < self.dE0]
461             range_1 = gen_i_ZLP[(self.deltaE < self.dE1) & (self.deltaE >= self.dE0)] *
                     factor_NN + signal[(self.deltaE < self.dE1) & (self.deltaE >= self.dE0
                    )] * factor_dm
462             range_2 = gen_i_ZLP[(self.deltaE >= self.dE1) & (self.deltaE < 3 * self.dE2
                    )]
463             range_3 = gen_i_ZLP[(self.deltaE >= 3 * self.dE2)] * 0
464             totalfile = np.concatenate((range_0, range_1, range_2, range_3), axis=0)
465             #TODO: now hardcoding no negative values!!!! CHECKKKK
466             totalfile = np.minimum(totalfile, signal)
467             return totalfile
468
```

```
        count = self.ZLPs_gen.shape[0]
        ZLPs = np.zeros(self.ZLPs_gen.shape) #np.zeros((count, len_data))


        for k in range(count):
            ZLPs[k,:] = matching(signal, k)#matching(energies, np.exp(mean_k), data)

        return ZLPs


    #METHODS ON DIELECTRIC FUNCTIONS

    def kramers_kronig_hs(self, I_EELS,
                          N_ZLP=None,
                          iterations=1,
                          n=None,
                          t=None,
                          delta=0.5, correct_S_s = False):
        r"""Calculate the complex
        dielectric function from a single scattering distribution (SSD) using
        the Kramers–Kronig relations.

        It uses the FFT method as in [1]_.  The SSD is an
        EELSSpectrum instance containing SSD low–loss EELS with no zero–loss
        peak. The internal loop is devised to approximately subtract the
        surface plasmon contribution supposing an unoxidized planar surface and
        neglecting coupling between the surfaces. This method does not account
        for retardation effects, instrumental broading and surface plasmon
        excitation in particles.

        Note that either refractive index or thickness are required.
        If both are None or if both are provided an exception is raised.

        Parameters
        ----------
        zlp: {None, number, Signal1D}
            ZLP intensity. It is optional (can be None) if 't' is None and 'n'
            is not None and the thickness estimation is not required. If 't'
            is not None, the ZLP is required to perform the normalization and
            if 't' is not None, the ZLP is required to calculate the thickness.
            If the ZLP is the same for all spectra, the integral of the ZLP
            can be provided as a number. Otherwise, if the ZLP intensity is not
            the same for all spectra, it can be provided as i) a Signal1D
            of the same dimensions as the current signal containing the ZLP
            spectra for each location ii) a BaseSignal of signal dimension 0
            and navigation_dimension equal to the current signal containing the
```

```
            integrated ZLP intensity.
        iterations: int
            Number of the iterations for the internal loop to remove the
            surface plasmon contribution. If 1 the surface plasmon contribution
            is not estimated and subtracted (the default is 1).
        n: {None, float}
            The medium refractive index. Used for normalization of the
            SSD to obtain the energy loss function. If given the thickness
            is estimated and returned. It is only required when `t` is None.
        t: {None, number, Signal1D}
            The sample thickness in nm. Used for normalization of the
             to obtain the energy loss function. It is only required when
            `n` is None. If the thickness is the same for all spectra it can be
            given by a number. Otherwise, it can be provided as a BaseSignal
            with signal dimension 0 and navigation_dimension equal to the
            current signal.
        delta : float
            A small number (0.1-0.5 eV) added to the energy axis in
            specific steps of the calculation the surface loss correction to
            improve stability.
        full_output : bool
            If True, return a dictionary that contains the estimated
            thickness if `t` is None and the estimated surface plasmon
            excitation and the spectrum corrected from surface plasmon
            excitations if `iterations` > 1.

        Returns
        -------
        eps: DielectricFunction instance
            The complex dielectric function results,

                .. math::
                    \epsilon = \epsilon_1 + i*\epsilon_2,

            contained in an DielectricFunction instance.
        output: Dictionary (optional)
            A dictionary of optional outputs with the following keys:

            ``thickness``
                The estimated  thickness in nm calculated by normalization of
                the SSD (only when `t` is None)

            ``surface plasmon estimation``
               The estimated surface plasmon excitation (only if
               `iterations` > 1.)
```

```
561         Raises
562         ------
563         ValuerError
564             If both `n` and `t` are undefined (None).
565         AttribureError
566             If the beam_energy or the collection semi-angle are not defined in
567             metadata.
568
569         Notes
570         -----
571         This method is based in Egerton's Matlab code [1]_ with some
572         minor differences:
573
574         * The wrap-around problem when computing the ffts is workarounded by
575           padding the signal instead of substracting the reflected tail.
576
577         .. [1] Ray Egerton, "Electron Energy-Loss Spectroscopy in the Electron
578            Microscope", Springer-Verlag, 2011.
579
580         """
581         output = {}
582         # Constants and units
583         me = 511.06
584
585         e0 = 200 #  keV
586         beta =30 #mrad
587
588         eaxis = self.deltaE[self.deltaE>0] #axis.axis.copy()
589         S_E = I_EELS[self.deltaE>0]
590         y = I_EELS[self.deltaE>0]
591         l = len(eaxis)
592         i0 = N_ZLP
593
594         # Kinetic definitions
595         ke = e0 * (1 + e0 / 2. / me) / (1 + e0 / me) ** 2
596         tgt = e0 * (2 * me + e0) / (me + e0)
597         rk0 = 2590 * (1 + e0 / me) * np.sqrt(2 * ke / me)
598
599         for io in range(iterations):
600             # Calculation of the ELF by normalization of the SSD
601             # We start by the "angular corrections"
602             Im = y / (np.log(1 + (beta * tgt / eaxis) ** 2)) / self.ddeltaE#axis.scale
603             if n is None and t is None:
604                 raise ValueError("The thickness and the refractive index are "
605                                  "not defined. Please provide one of them.")
606             elif n is not None and t is not None:
```

```
607                 raise ValueError("Please provide the refractive index OR the "
608                                  "thickness information, not both")
609             elif n is not None:
610                 # normalize using the refractive index.
611                 K = np.sum(Im/eaxis)*self.ddeltaE
612                 K = (K / (np.pi / 2) / (1 - 1. / n ** 2))
613                 te = (332.5 * K * ke / i0)
614             elif t is not None:
615                 if N_ZLP is None:
616                     raise ValueError("The ZLP must be provided when the  "
617                                      "thickness is used for normalization.")
618                 # normalize using the thickness
619                 K = t * i0 / (332.5 * ke)
620                 te = t
621             Im = Im / K
622
623             # Kramers Kronig Transform:
624             # We calculate KKT(Im(-1/epsilon))=1+Re(1/epsilon) with FFT
625             # Follows: D W Johnson 1975 J. Phys. A: Math. Gen. 8 490
626             # Use an optimal FFT size to speed up the calculation, and
627             # make it double the closest upper value to workaround the
628             # wrap-around problem.
629             esize = next_fast_len(2*l) #2**math.floor(math.log2(l)+1)*4
630             q = -2 * np.fft.fft(Im, esize).imag / esize
631
632             q[:l] *= -1
633             q = np.fft.fft(q)
634             # Final touch, we have Re(1/eps)
635             Re = q[:l].real + 1
636             # Egerton does this to correct the wrap-around problem, but in our
637             # case this is not necessary because we compute the fft on an
638             # extended and padded spectrum to avoid this problem.
639             # Re=real(q)
640             # Tail correction
641             # vm=Re[axis.size-1]
642             # Re[:(axis.size-1)]=Re[:(axis.size-1)]+1-(0.5*vm*((axis.size-1) /
643             #  (axis.size*2-arange(0,axis.size-1)))**2)
644             # Re[axis.size:]=1+(0.5*vm*((axis.size-1) /
645             #  (axis.size+arange(0,axis.size)))**2)
646
647             # Epsilon appears:
648             #  We calculate the real and imaginary parts of the CDF
649             e1 = Re / (Re ** 2 + Im ** 2)
650             e2 = Im / (Re ** 2 + Im ** 2)
651
652             if iterations > 0 and N_ZLP is not None:
```

```
653                 # Surface losses correction:
654                 #  Calculates the surface ELF from a vaccumm border effect
655                 #  A simulated surface plasmon is subtracted from the ELF
656                 Srfelf = 4 * e2 / ((e1 + 1) ** 2 + e2 ** 2) - Im
657                 adep = (tgt / (eaxis + delta) *
658                         np.arctan(beta * tgt / eaxis) -
659                         beta / 1000. /
660                         (beta ** 2 + eaxis ** 2. / tgt ** 2))
661                 Srfint = 2000 * K * adep * Srfelf / rk0 / te * self.ddeltaE #axis.scale
662                 if correct_S_s == True:
663                     print("correcting S_s")
664                     Srfint[Srfint<0] = 0
665                     Srfint[Srfint>S_E] = S_E[Srfint>S_E]
666                 y = S_E - Srfint
667                 _logger.debug('Iteration number: %d / %d', io + 1, iterations)
668
669
670         eps = (e1 + e2 * 1j)
671         del y
672         del I_EELS
673         if 'thickness' in output:
674             # As above,prevent errors if the signal is a single spectrum
675             output['thickness'] = te
676
677         return eps, te, Srfint
678
679
680
681     def im_dielectric_function(self, track_process = False, plot = False):
682         """
683         INPUT:
684             self -- the image of which the dielectic functions are calculated
685             track_process -- boolean, default = False, if True: prints for each pixel
                    that program is busy with that pixel.
686             plot -- boolean, default = False, if True, plots all calculated dielectric
                    functions
687         OUTPUT:
688             self.dielectric_function_im_avg = average dielectric function for each
                    pixel
689             self.dielectric_function_im_std = standard deviation of the dielectric
                    function at each energy for each pixel
690             self.S_s_avg = average surface scattering distribution for each pixel
691             self.S_s_std = standard deviation of the surface scattering distribution at
                     each energy for each pixel
692             self.thickness_avg = average thickness for each pixel
693             self.thickness_std = standard deviation thickness for each pixel
```

```
694             self.IEELS_avg = average bulk scattering distribution for each pixel
695             self.IEELS_std = standard deviation of the bulk scattering distribution at
                    each energy for each pixel
696         """
697         #TODO
698         #data = self.data[self.deltaE>0, :,:]
699         #energies = self.deltaE[self.deltaE>0]
700         if not hasattr(self, 'ZLPs_gen'):
701             self.calc_ZLPs_gen2("iets")
702         self.dielectric_function_im_avg = (1+1j)*np.zeros(self.data[ :,:,self.deltaE>0
                ].shape)
703         self.dielectric_function_im_std = (1+1j)*np.zeros(self.data[ :,:,self.deltaE>0
                ].shape)
704         self.S_s_avg = (1+1j)*np.zeros(self.data[ :,:,self.deltaE>0].shape)
705         self.S_s_std = (1+1j)*np.zeros(self.data[ :,:,self.deltaE>0].shape)
706         self.thickness_avg = np.zeros(self.image_shape)
707         self.thickness_std = np.zeros(self.image_shape)
708         self.IEELS_avg = np.zeros(self.data.shape)
709         self.IEELS_std = np.zeros(self.data.shape)
710         N_ZLPs_calculated = hasattr(self, 'N_ZLPs')
711         #TODO: add N_ZLP saving
712         #if not N_ZLPs_calculated:
713         #    self.N_ZLPs = np.zeros(self.image_shape)
714         if plot:
715             fig1, ax1 = plt.subplots()
716             fig2, ax2 = plt.subplots()
717         for i in range(self.image_shape[0]):
718             for j in range(self.image_shape[1]):
719                 if track_process: print("calculating dielectric function for pixel " ,
                        i,j)
720                 data_ij = self.get_pixel_signal(i,j)#[self.deltaE>0]
721                 ZLPs = self.calc_ZLPs(i,j)#[:,self.deltaE>0]
722                 dielectric_functions = (1+1j)* np.zeros(ZLPs[:,self.deltaE>0].shape)
723                 S_ss = np.zeros(ZLPs[:,self.deltaE>0].shape)
724                 ts = np.zeros(ZLPs.shape[0])
725                 IEELSs = np.zeros(ZLPs.shape)
726                 for k in range(23,28):#ZLPs.shape[0]):
727                     ZLP_k = ZLPs[k,:]
728                     N_ZLP = np.sum(ZLP_k)
729                     IEELS = data_ij-ZLP_k
730                     IEELS = self.deconvolute(i, j, ZLP_k)
731                     IEELSs[k,:] = IEELS
732                     if plot:
733                         #ax1.plot(self.deltaE, IEELS)
734                         plt.figure()
735                         plt.plot(self.deltaE, IEELS)
```

```
736                     #TODO: FIX ZLP: now becomes very negative!!!!!!!
737                     #TODO: VERY IMPORTANT
738                     dielectric_functions[k,:], ts[k], S_ss[k] = self.kramers_kronig_hs(
                            IEELS, N_ZLP = N_ZLP, n =3)
739                     if plot:
740                         #plt.figure()
741                         plt.plot(self.deltaE[self.deltaE>0], dielectric_functions[k,:]*
                                2)
742                         plt.xlim(0,10)
743                         plt.ylim(-100, 400)
744
745                 #print(ts)
746                 self.dielectric_function_im_avg[i,j,:] = np.average(
                        dielectric_functions, axis = 0)
747                 self.dielectric_function_im_std[i,j,:] = np.std(dielectric_functions,
                        axis = 0)
748                 self.S_s_avg[i,j,:] = np.average(S_ss, axis = 0)
749                 self.S_s_std[i,j,:] = np.std(S_ss, axis = 0)
750                 self.thickness_avg[i,j] = np.average(ts)
751                 self.thickness_std[i,j] = np.std(ts)
752                 self.IEELS_avg[i,j,:] = np.average(IEELSs, axis = 0)
753                 self.IEELS_std[i,j,:] = np.std(IEELSs, axis = 0)
754         #return dielectric_function_im_avg, dielectric_function_im_std
755
756     def crossings_im(self):#,  delta = 50):
757         """
758         INPUT:
759             self
760         OUTPUT:
761             self.crossings_E = numpy array (image-shape, N_c), where N_c the maximimun
                    number of crossings of any pixel, 0 indicates no crossing
762             self.crossings_n = numpy array (image-shape), number of crossings per pixel
763         Calculates for each pixel the crossings of the real part of the dielectric
                function \
764             from negative to positive.
765         """
766         self.crossings_E = np.zeros((self.image_shape[0], self.image_shape[1],1))
767         self.crossings_n = np.zeros(self.image_shape)
768         n_max = 1
769         for i in range(self.image_shape[0]):
770             #print("cross", i)
771             for j in range(self.image_shape[1]):
772                 #print("cross", i, j)
773                 crossings_E_ij, n = self.crossings(i,j)#, delta)
774                 if n > n_max:
775                     #print("cross", i, j, n, n_max, crossings_E.shape)
```

```
776                     crossings_E_new = np.zeros((self.image_shape[0], self.image_shape[1
                            ],n))
777                     #print("cross", i, j, n, n_max, crossings_E.shape, crossings_E_new
                            [:,:,:n_max].shape)
778                     crossings_E_new[:,:,:n_max] = self.crossings_E
779                     self.crossings_E = crossings_E_new
780                     n_max = n
781                     del crossings_E_new
782                 self.crossings_E[i,j,:n] = crossings_E_ij
783                 self.crossings_n[i,j] = n
784 
785     def crossings(self, i, j):#, delta = 50):
786         #l = len(die_fun)
787         die_fun_avg = np.real(self.dielectric_function_im_avg[ i, j, :])
788         #die_fun_f = np.zeros(l-2*delta)
789         #TODO: use smooth?
790         """
791         for i in range(self.l-delta):
792             die_fun_avg[i] = np.average(self.dielectric_function_im_avg[i:i+delta])
793         """
794         crossing = np.concatenate((np.array([0]),(die_fun_avg[:-1]<0) * (die_fun_avg[1
                :] >=0)))
795         deltaE_n = self.deltaE[self.deltaE>0]
796         #deltaE_n = deltaE_n[50:-50]
797         crossing_E = deltaE_n[crossing.astype('bool')]
798         n = len(crossing_E)
799         return crossing_E, n
800 
801 
802 
803     #PLOTTING FUNCTIONS
804     def plot_sum(self, title = None, xlab = None, ylab = None):
805         """
806         INPUT:
807             self -- spectral image
808             title -- str, delfault = None, title of plot
809             xlab -- str, default = None, x-label
810             ylab -- str, default = None, y-label
811         OUTPUT:
812         Plots the summation over the intensity for each pixel in a heatmap.
813         """
814         #TODO: invert colours
815         if hasattr(self, 'name'):
816             name = self.name
817         else:
818             name = ''
```

```
819         plt.figure()
820         if title is not None:
821             plt.title("intgrated intensity spectrum " + name)
822         else:
823             plt.title(title)
824         ax = sns.heatmap(np.sum(self.data, axis = 2))
825         if not hasattr(self, 'pixelsize'):
826             plt.xlabel(self.pixelsize)
827             plt.ylabel(self.pixelsize)
828         if xlab is not None:
829             plt.xlabel(xlab)
830         if ylab is not None:
831             plt.ylabel = ylab
832         plt.show()
833 
834 
835     def plot_all(self, same_image = True, normalize = False, legend = False,
836                  range_x = None, range_y = None, range_E = None, signal = "EELS", log =
                         False):
837         #TODO: add titles and such
838         if range_x is None:
839             range_x = [0,self.image_shape[1]]
840         if range_y is None:
841             range_y = [0,self.image_shape[0]]
842         if same_image:
843             plt.figure()
844             plt.title("Spectrum image " + signal + " spectra")
845             plt.xlabel("[eV]")
846             if range_E is not None:
847                 plt.xlim(range_E)
848         for i in range(range_y[0], range_y[1]):
849             for j in range(range_x[0], range_x[1]):
850                 if not same_image:
851                     plt.figure()
852                     plt.title("Spectrum pixel: [" + str(j) +","+ str(i) + "]")
853                     plt.xlabel("[eV]")
854                     if range_E is not None:
855                         plt.xlim(range_E)
856                     if legend:
857                         plt.legend()
858                 signal_pixel = self.get_pixel_signal(i,j,signal)
859                 if normalize:
860                     signal_pixel /= np.max(np.absolute(signal_pixel))
861                 if log:
862                     signal_pixel = np.log(signal_pixel)
863                     plt.ylabel("log intensity")
```

```
                plt.plot(self.deltaE, signal_pixel, label = "[" + str(j) +","+ str(i) +
                    "]")
            if legend:
                plt.legend()


    #STATIC METHODS
    @staticmethod
    def get_prefix(unit, SIunit = None, numeric = True):
        """
        INPUT:
            unit -- str, unit of which the prefix is wanted
            SIunit -- str, default = None, the SI unit of the unit of which the prefix
                is wanted \
                        (eg 'eV' for 'keV'), if None, first character of unit is
                            evaluated as prefix
            numeric -- bool, default = True, if numeric the prefix is translated to the
                 numeric value \
                        (e.g. 1E3 for 'k')
        OUTPUT:
            prefix -- str or int, the character of the prefix or the numeric value of
                the prefix
        """
        if SIunit is not None:
            lenSI = len(SIunit)
            if unit[-lenSI:] == SIunit:
                prefix = unit[:-lenSI]
                if len(prefix) == 0:
                    if numeric: return 1
                    else: return prefix
            else:
                print("provided unit not same as target unit: " + unit + ", and " +
                    SIunit)
                if numeric: return 1
                else: return prefix
        else:
            prefix = unit[0]
        if not numeric:
            return prefix

        if prefix == 'p':
            return 1E-12
        if prefix == 'n':
            return 1E-9
        if prefix == 'µ' or prefix == 'μ' or prefix == 'u':
            return 1E-6
```

```
904         if prefix == 'm':
905             return 1E-3
906         if prefix == 'k':
907             return 1E3
908         if prefix == 'M':
909             return 1E6
910         if prefix == 'G':
911             return 1E9
912         if prefix == 'T':
913             return 1E12
914         else:
915             print("either no or unknown prefix in unit: " + unit + ", found prefix " +
                    prefix + ", asuming no.")
916         return 1
917
918
919
920     #CLASS THINGIES
921     def __getitem__(self, key):
922         """ Determines behavior of `self[key]` """
923         return self.data[key]
924         #pass
925
926
927
928     def __str__(self):
929         if hasattr(self, 'name'):
930             name_str = ", name = " + self.name
931         else:
932             name_str = ""
933         return 'Spectral image: ' + name_str + ", image size:"+ str(self.data.shape[0])
                + 'x' + \
934                 str(self.data.shape[1]) + ', deltaE range: [' + str(round(self.
                    deltaE[0],3)) + ',' + \
935                     str(round(self.deltaE[-1],3)) + '], deltadeltaE: ' + str(round(
                        self.ddeltaE,3))
936
937     def __repr__(self):
938         data_str = "data * np.ones(" + str(self.shape) + ")"
939         if hasattr(self, 'name'):
940             name_str = ", name = " + self.name
941         else:
942             name_str = ""
943         return "Spectral_image(" + data_str +  ", deltadeltaE=" + str(round(self.
                ddeltaE, 3)) + name_str + ")"
944
```

```
945     def __len__(self):
946         return self.l
947
948
949
950
951 def CFT(x, y):
952     x_0 = np.min(x)
953     N_0 = np.argmin(np.absolute(x))
954     N = len(x)
955     x_max = np.max(x)
956     delta_x = (x_max-x_0)/N
957     k = np.linspace(0, N-1, N)
958     cont_factor = np.exp(2j*np.pi*N_0*k/N)*delta_x #np.exp(-1j*(x_0)*k*delta_omg)*
            delta_x
959     F_k = cont_factor * np.fft.fft(y)
960     return F_k
961
962 def iCFT(x, Y_k):
963     x_0 = np.min(x)
964     N_0 = np.argmin(np.absolute(x))
965     x_max = np.max(x)
966     N = len(x)
967     delta_x = (x_max-x_0)/N
968     k = np.linspace(0, N-1, N)
969     cont_factor = np.exp(-2j*np.pi*N_0*k/N)
970     f_n = np.fft.ifft(cont_factor*Y_k)/delta_x # 2*np.pi ##np.exp(-2j*np.pi*x_0*k)
971     return f_n
972
973
974
975
976
977
978
979
980 #%%
981
982
983 #data = np.load("area03-eels-SI-aligned.npy")
984 #energies = np.load("area03-eels-SI-aligned_energy.npy")
985
986
987 #dielectric_function_im_avg, dielectric_function_im_std = im_dielectric_function(data,
        energies)
988
```

```
989
990 #%%
991 #crossings_E, crossings_n =  crossings_im(dielectric_function_im_avg, energies)
992
993
994 #%%
995
996 #plt.figure()
997 #plt.imshow(crossings_n, cmap='hot', interpolation='nearest')
998 #plt.
999
1000
1001 #ax = sns.heatmap(crossings_n)
1002 #plt.show()
1003
1004 #%%
1005 #dmfile = dm.fileDM('area03-eels-SI-aligned.dm4')
1006 #data2 = dmfile.getDataset(0)
1007
1008 im = Spectral_image.load_data('area03-eels-SI-aligned.dm4')#('pyfiles/area03-eels-SI-
        aligned.dm4')
1009 im.cut_image([0,70], [95,100])
1010 #im.cut_image([40,41],[4,5])
1011 im.calc_ZLPs_gen2(specimen = 4)
1012 im.smooth(window_len=50)
1013 im.im_dielectric_function()
1014 im.crossings_im()
1015
1016 #%%
1017
1018 plt.figure()
1019 plt.title("number of crossings real part dielectric function")
1020 ax = sns.heatmap(im.crossings_n)
1021 plt.show()
1022
1023 plt.figure()
1024 plt.title("energy of first crossings real part dielectric function")
1025 ax = sns.heatmap(im.crossings_E[:,:,0])
1026 plt.show()
1027
1028
1029 plt.figure()
1030 plt.title("thickness of sample")
1031 ax = sns.heatmap(im.thickness_avg)
1032 plt.show()
```