SEM EDS performance parameters

June 1, 2023

1 1. Import the packages

[ ]:
```python
import hyperspy.api as hs
import numpy as np
import pandas as pd

from helper_functions import elementlines, nearestlines, theoretical_energy
```

2 3. Select spectra, an specify settings like i_b, E_0, ICR, and PT

[ ]:
```python
##### SEM Apreo March 2023 #####
path = '../../Masteroppgave/2023-03-08_EDS-Apreo/exports/'
# All data files are available in the repository at:
# https://github.com/brynjarmorka/eds-sem-bulk-corrections/tree/main/data

##### GaAs #####
elements = ['Ga', 'As']
lines_of_interest = ['Ga_Ka', 'As_Ka', 'Ga_La', 'As_La', 'Ga_Kb', 'As_Kb']
fiori_peaks = ['Ga_La', 'As_La', 'Ga_Ka']
line_ratio_list = [['As_Ka', 'As_La'], ['Ga_Ka', 'Ga_La'], ['Ga_La', 'As_La']]

# setting file name, beam current, input count rate and process time
file_current_ICR_PT = [
    ['GaAs_05kV_25pA.emsa', 25, 880, 6], # 0
    ['GaAs_10kV_25pA.emsa', 25, 1750, 6], # 1
    ['GaAs_15kV_25pA.emsa', 25, 3300, 6], # 2
    ['GaAs_30kV_25pA.emsa', 25, 8000, 6], # 3
    ['GaAs_30kV_50pA.emsa', 50, 16400, 6], # 4
]

##### GaSb #####
# Sb_Ka has to high energy
# elements = ['Ga', 'Sb']
# lines_of_interest = ['Ga_Ka', 'Ga_Kb', 'Ga_La', 'Sb_La', 'Sb_Lb1']
# fiori_peaks = ['Ga_La', 'Sb_La', 'Ga_Ka', 'Sb_Lb1']
# line_ratio_list = [['Ga_Ka', 'Ga_La'], ['Sb_La', 'Sb_Lb1'], ['Ga_La',␣
↪'Sb_La']]

# file_current_ICR_PT = [
#     ['GaSb_05kV_50pA.emsa', 50, 1080, 6], # 0
#     ['GaSb_10kV_50pA.emsa', 50, 2300, 6], # 1
#     ['GaSb_15kV_50pA.emsa', 50, 5700, 6], # 2
#     ['GaSb_15kV_200pA.emsa', 200, 22000, 6], # 3
#     ['GaSb_15kV_400pA.emsa', 400, 42000, 6], # 4
#     ['GaSb_30kV_50pA.emsa', 50, 17000, 6], # 5
#     ['GaSb_30kV_50pA_noPPUC.emsa', 50, 17000, 6], # 6
#     ['GaSb_30kV_50pA_processTime1.emsa', 50, 17000, 1], # 7
#     ['GaSb_30kV_50pA_processTime2.emsa', 50, 17000, 2], # 8
#     ['GaSb_30kV_50pA_processTime4.emsa', 50, 17000, 4], # 9
#     ['GaSb_30kV_400pA_processTime1.emsa', 400, 160000, 1], # 10
# ]

# change `number` to change file
number = 4

file = file_current_ICR_PT[number][0]
beam_current = file_current_ICR_PT[number][1]
input_count_rate = file_current_ICR_PT[number][2]
process_time = file_current_ICR_PT[number][3]
```

[ ]:
```
# common settings
make_info_on_all_lines = True
zero_peak_end_index = 40
model_background_order = 6
```

3 3. Import the data with HyperSpy, set the elements in the spectrum, and slice off the noise peak

[ ]:
```
def load_data(path, file, elements, zero_peak_end_index, plot_s=False):
    """Loading data with HS and adding the elements, eventually removing the␣
↪zero peak"""
    s = hs.load(path + '/' + file, signal_type='EDS_SEM')
    s.add_elements(elements)
    if zero_peak_end_index is not None:
        s = s.isig[zero_peak_end_index:]

    Vacc = s.metadata.Acquisition_instrument.SEM.beam_energy
    x = s.axes_manager[0].axis  # x-axis in keV
    name = f'{file[:-5]}'
    if plot_s:
        s.plot(xray_lines=True)
    return s, Vacc, x, name
```

[ ]:
```
s, Vacc, x, name = load_data(path, file, elements, zero_peak_end_index)
```

[ ]:
```
# these are temporary arrays used to show the effect of the calibrations
scale_list = [s.axes_manager[0].scale]
offset_list = [s.axes_manager[0].offset]
energy_res_list = [s.metadata.Acquisition_instrument.SEM.Detector.EDS.
 ↪energy_resolution_MnKa]
```

4 4. Calculate the Duane-Hunt limit, and slice the spectrum to the limit

[ ]:
```
# # Duane-Hunt method to find the real E_0
def calculate_duane_hunt(s=s, buffer_start=2, buffer_end=0.1,␣
 ↪xaxis_plot_buffer=0.5, dh_plot=False):
    x_max = s.axes_manager[0].high_value  # highest x-axis value in keV, used␣
 ↪in Duane-Hunt
    if Vacc > x_max:
        print(f'Vacc={Vacc} > x_max={x_max}, Duane-Hunt not possible')
        return np.nan
    else:
        s_dh = s.deepcopy()
        # making the lin fit of the background right before Vacc
        dh_start = Vacc-buffer_start
        dh_end = Vacc-buffer_end
        s_end = s_dh.isig[dh_start:dh_end] # slice with keV
        m_end = s_end.create_model(auto_background=False)
        m_end.add_polynomial_background(order=1)
        m_end.fit()

        x_s_end = s_dh.isig[dh_start-xaxis_plot_buffer:
 ↪dh_end+xaxis_plot_buffer].axes_manager[0].axis
        dh_bg_zero_index = np.argmin(np.abs(m_end[-1].function(x_s_end) * s_dh.
 ↪axes_manager[0].scale))

        dh_limit = x_s_end[dh_bg_zero_index]
        print(f'Duane-Hunt limit: {dh_limit:.3f} keV')

        return dh_limit
```

[ ]:
```
dh_limit = calculate_duane_hunt(dh_plot=False)
```

```
Vacc=30.0 > x_max=20.27, Duane-Hunt not possible
```

```
[ ]: # Using Duane-Hunt to slice the spectrum
     def use_dh_to_slice_spectrum(dh_limit=dh_limit, s=s, plot=False):
         if np.isnan(dh_limit):
             print('No Duane-Hunt limit found, not slicing the spectrum')
         else:
             s = s.isig[:dh_limit]
             print(f'Spectrum sliced at {dh_limit:.2f} keV')
             if plot:
                 s.plot(xray_lines=True)
         return s
```

```
[ ]: s = use_dh_to_slice_spectrum(dh_limit, plot=False)
     x = s.axes_manager[0].axis  # x-axis in keV, after slicing
```

```
No Duane-Hunt limit found, not slicing the spectrum
```

5 5. Make a model of the spectrum, and fit it to the data

```
[ ]: # creating a model and fitting it
     def make_model(s=s, model_background_order=model_background_order,␣
      ↪plot_m=False):
         m = s.create_model(auto_background=False)
         m.add_polynomial_background(order=model_background_order)
         m.fit_background()
         m.fit(bounded=True)
         if plot_m:
             m.plot(plot_components=True)
         return m
```

```
[ ]: m = make_model(s=s, model_background_order=model_background_order, plot_m=False)
```

```
[ ]: def remove_lines_not_in_model(m=m, lines_of_interest=lines_of_interest,␣
      ↪line_ratio_list=line_ratio_list, fwtm_to_fwhm_lines=fwtm_to_fwhm_lines):
         lines_in_model = [c.name for c in m][:-1] # not including the background

         lines_to_remove = []
         for line in lines_of_interest:
             if line not in lines_in_model:
                 # print(f'NB!: {line} not in model')
                 lines_to_remove.append(line)

         fwtm_to_fwhm_to_remove = []
         for line in fwtm_to_fwhm_lines:
             if line not in lines_in_model:
                 # print(f'NB!: {line} not in model, fwtm/fwhm not possible')
                 fwtm_to_fwhm_to_remove.append(line)
```

```
    line_ratio_to_remove = []
    for linepair in line_ratio_list:
        if linepair[0] not in lines_in_model or linepair[1] not in␣
↪lines_in_model:
            # print(f'NB!: {linepair} not in model, ratio not possible')
            line_ratio_to_remove.append(linepair)

    # remove the lines from the lists
    for line in lines_to_remove:
        lines_of_interest.remove(line)
    for linepair in line_ratio_to_remove:
        line_ratio_list.remove(linepair)
```

[ ]:
```
remove_lines_not_in_model()
```

6 6. Calibrate the offset and scale

[ ]:
```
def sort_lines(lines_of_interest):
    """Sort lines_of_interest by area, taking area from m[line].A.value
    Used because the calibrate_energy_axis(calibrate='resolution') use the␣
↪first line,
    and using the strongest line gives a good reference energy for the function␣
↪by
    Newbury and Fiori (1978), documented in Goldstein (2018), Eq. ??.
    """
    lines_of_interest = sorted(lines_of_interest, key=lambda x: m[x].A.value,␣
↪reverse=True)
    # for l in lines_of_interest:
        # print(f'{l}: {m[l].A.value:.2f}')
    return lines_of_interest
```

[ ]:
```
lines_of_interest = sort_lines(lines_of_interest)
all_lines = [l.name for l in m if not l.isbackground]
all_lines = sort_lines(all_lines)
```

[ ]:
```
def calibrate_axis(m=m, s=s, rounds=2, xray_lines=lines_of_interest):
    """Calibrating the scale and offset of the energy axis."""
    print('Calibrating energy axis (with many elements it can take multiple␣
↪minutes)')

    for i in range(rounds):
        print(f'Calibrating offset and scale, round {i+1} of {rounds}')
        m.calibrate_energy_axis(calibrate='offset', xray_lines=xray_lines)
        offset_list.append(s.axes_manager[0].offset)
        m.calibrate_energy_axis(calibrate='scale', xray_lines=xray_lines)
        scale_list.append(s.axes_manager[0].scale)
    print(f'Scale: {scale_list[-1]:.6f} eV/px \nOffset: {offset_list[-1]:.6f}␣
↪keV')

    return scale_list[-1], offset_list[-1]
```

[ ]:
```
scale, offset = calibrate_axis(rounds=2, xray_lines=lines_of_interest)
# using all lines gives basically the same result
# scale2, offset2 = calibrate_axis(rounds=1, xray_lines=all_lines)
```

```
Calibrating energy axis (with many elements it can take multiple minutes)
Calibrating offset and scale, round 1 of 2
Calibrating offset and scale, round 2 of 2
Scale: 0.010046 eV/px
Offset: 0.193846 keV
```

7 7. Calibrate the energy resolution

[ ]:
```
def calibrate_resolution(m=m, s=s, rounds=2, xray_lines=lines_of_interest[:3]):
    """
    Calibrating the energy resolution, i.e. the energy resolution at the Mn Ka␣
↪line.
    NB! The reference line is the first line in xray_lines, thus the first line␣
↪should be well defined.

    Using the three strongest lines to first calibrate the width, then estimate␣
↪the energy resolution,
    which is done with only the first line in lines_of_interest
    """
    for i in range(rounds):
        print(f'Calibrating energy resolution, round {i+1} of {rounds}')
        m.calibrate_energy_axis(calibrate='resolution', xray_lines=xray_lines)
        energy_res_list.append(s.metadata.Acquisition_instrument.SEM.Detector.
↪EDS.energy_resolution_MnKa)

    print(f'Calibrated energy resolution: {energy_res_list[-1]:.3f} eV')
    return energy_res_list[-1]
```

[ ]:
```
energy_resolution = calibrate_resolution()
```

```
Calibrating energy resolution, round 1 of 2
Calibrating energy resolution, round 2 of 2
Calibrated energy resolution: 128.676 eV
```

[ ]:
```
def print_calibration_info(scls=scale_list, offs=offset_list,␣
↪eres=energy_res_list):
    # make pretty print of calibration info
    infos = [' ', 'Scale [eV/channel]', 'Offset [keV]', 'E-res [eV]']
    row1 = ['Current', f'{scls[-1]:.6f}', f'{offs[-1]:.6f}', f'{eres[-1]:.3f}']
    row2 = ['Original', f'{scls[0]:.6f}', f'{offs[0]:.6f}', f'{eres[0]:.3f}']
    row3 = ['Δ original', f'{(scls[-1] - scls[0])/scls[-2]*100:.3f} %',
    f'{(offs[-1] - offs[0])/offs[-2]*100:.3f} %', f'{(eres[-1] - eres[0])/
↪eres[0]*100:.3f} %']
    row4 = ['Δ last step', f'{(scls[-1] - scls[-2])/scls[-2]*100:.3f} %',
    f'{(offs[-1] - offs[-2])/offs[-2]*100:.3f} %', f'{(eres[-1] - eres[-2])/
↪eres[-2]*100:.3f} %']

    for i in range(len(infos)):
        print(f'{infos[i]:<20}{row1[i]:<15}{row2[i]:<15}{row3[i]:<15}{row4[i]:
↪<15}')
```

[ ]:
```
print_calibration_info()
```

```
                    Current        Original       Δ original     Δ last step
Scale [eV/channel]  0.010046       0.010000       0.461 %        0.282 %
Offset [keV]        0.193846       0.200000       -3.134 %       -1.293 %
E-res [eV]          128.676        130.000        -1.019 %       0.031 %
```

8 8. Calibrate the energy and width of the peaks

[ ]:
```
def calibrate_lines(m=m, rounds=2, xray_lines='all'):
    """Calibrating the energy and width of the specified lines in m."""
    for i in range(rounds):
        print(f'Calibrating peak positions, round {i+1} of {rounds}')
        m.calibrate_xray_lines(calibrate='energy', xray_lines=xray_lines,␣
↪kind='single') # use kind='multi' for better results? Dunno
        m.calibrate_xray_lines(calibrate='width', xray_lines=xray_lines,␣
↪kind='single')
```

[ ]:
```
calibrate_lines(rounds=2, xray_lines=lines_of_interest)
```

```
Calibrating peak positions, round 1 of 2
Calibrating peak positions, round 2 of 2
```

9 9. Calculate Fiori P/B, peak intensities, FWHMs, and peak deviations

[ ]:
```
def make_lines_info(m=m, all_lines=make_info_on_all_lines, sort_by='Area',␣
↪lines_of_interest=lines_of_interest):
    lines_info = {}
    for i in range(len(m) - 1): # last component is the background
        if (all_lines == True) or (m[i].name in lines_of_interest):
            lines_info[m[i].name] = {
                'Theoretical E [keV]': theoretical_energy(m[i].name),
                'Calibrated E [keV]': np.round(m[i].centre.value, 4),
                'Area': np.round(m[i].A.value, 1),
                'Fiori P/B': np.round(m[i].A.value / (m[-1].function(m[i].
↪centre.value) * scale), 1),
                'FWHM [eV]': np.round(m[i].fwhm * 1000, 3),
                'Sigma [keV]': np.round(m[i].sigma.value, 4),
                'Height': np.round(m[i].height * scale, 1),
                'FWHM(Mn Ka)' : np.round(np.sqrt(2.5*(5898.7 - m[i].centre.
↪value*1000) + (m[i].fwhm*1000)**2), 3),
            }
    lines_info = pd.DataFrame(lines_info).T
    lines_info['Delta E [eV]'] = np.round((lines_info['Calibrated E [keV]'] -␣
↪lines_info['Theoretical E [keV]'] ) * 1000, 2)
    lines_info = lines_info.sort_values(by=sort_by, ascending=False)
    return lines_info
```

[ ]:
```
lines_info = make_lines_info(all_lines=make_info_on_all_lines, sort_by='Area')
lines_info
```

[ ]:
```
        Theoretical E [keV]  Calibrated E [keV]      Area  Fiori P/B  \
Ga_La                1.0980              1.1000  332711.7      585.8
Ga_Ka                9.2517              9.2537  301654.4      761.5
As_Ka               10.5436             10.5456  183007.1      621.5
As_La                1.2819              1.2839  140489.7      236.2
Ga_Lb1               1.1249              1.1249   55576.2       97.2
Ga_Kb               10.2642             10.2662   38822.9      123.6
As_Kb               11.7262             11.7282   26698.9      117.4
As_Lb1               1.3174              1.3174   23467.4       39.1
Ga_Ll                0.9573              0.9573   18099.5       33.1
Ga_Lb3               1.1948              1.1948   15338.0       26.4
Ga_Ln                0.9842              0.9842    8347.7       15.2
As_Ll                1.1196              1.1196    6924.7       12.1
As_Lb3               1.3860              1.3860    6700.0       11.0
As_Ln                1.1551              1.1551    2710.0        4.7

        FWHM [eV]  Sigma [keV]   Height  FWHM(Mn Ka)  Delta E [eV]
Ga_La      66.866       0.0284  46960.0      128.327           2.0
Ga_Ka     162.634       0.0691  17505.2      134.396           2.0
As_Ka     172.548       0.0733  10009.8      134.743           2.0
As_La      72.009       0.0306  18412.9      129.315           2.0
Ga_Lb1     67.993       0.0289   7714.3      128.676           0.0
Ga_Kb     161.035       0.0684   2275.3      122.529           2.0
As_Kb     181.136       0.0769   1391.1      135.043           2.0
As_Lb1     71.444       0.0303   3100.0      128.676           0.0
Ga_Ll      64.838       0.0275   2634.5      128.676           0.0
Ga_Lb3     69.266       0.0294   2089.9      128.676           0.0
```

```
Ga_Ln      65.355       0.0278   1205.5      128.676          0.0
As_Ll      67.895       0.0288    962.6      128.676          0.0
As_Lb3     72.634       0.0308    870.6      128.676          0.0
As_Ln      68.546       0.0291    373.1      128.676          0.0
```

10 10. Calculate the relevant peak ratios

[ ]:
```
def peak_ratio(line1, line2, m=m):
    # give the K to L ratio of a line, e.g. 'Ga_Ka' to 'Ga_La'
    try:
        m[line1]
        m[line2]
    except ValueError:
        print('line not in model:', line1, line2)
        return np.nan
    return np.round(m[line1].A.value / m[line2].A.value, 3)


def calculate_all_line_ratios():
    line_ratios = {}
    for line_pair in line_ratio_list:
        pair_name = line_pair[0] + '/' + line_pair[1]
        line_ratios[pair_name] =  peak_ratio(line_pair[0], line_pair[1])
    line_ratios = pd.DataFrame(line_ratios, index=['Line ratio']).T
    return line_ratios
```

[ ]:
```
line_ratios = calculate_all_line_ratios()
print(line_ratios)
```

```
             Line ratio
As_Ka/As_La       1.303
Ga_Ka/Ga_La       0.907
Ga_La/As_La       2.368
```

11 11. Save the results in a DataFrame in a ".csv" file

[ ]:
```
def dead_time(s=s, m=m):
    real_time = s.metadata.Acquisition_instrument.SEM.Detector.EDS.real_time
    live_time = s.metadata.Acquisition_instrument.SEM.Detector.EDS.live_time
    dead_time_percent = (real_time - live_time)/real_time*100
    print(f'Dead time: {dead_time_percent:.1f}%')
    return round(dead_time_percent, 1)
```

[ ]:
```
dead_time_percent = dead_time(s=s, m=m)
```

```
Dead time: 43.7%
```

```
[ ]: def make_output():
         key_output = pd.DataFrame({
             'Name': [name],
             'Nominal beam energy [kV]': [Vacc],
             'Beam current [pA]': [beam_current],
             'Process time' : [process_time],
             'ICR' : [input_count_rate],
             'Dead time [%]': [dead_time_percent],
             'Live time [s]': [round(s.metadata.Acquisition_instrument.SEM.Detector.
       ↪EDS.live_time, 1)],
             'Duane-Hunt limit [kV]': [dh_limit],
             'Scale [keV]': [scale],
             'Offset [keV]': [offset],
             'Energy resolution [eV]': [energy_resolution],
         })
         for ratio_pair in line_ratios.T.columns:
             key_output[f'Ratio ({ratio_pair})'] = line_ratios.T[ratio_pair][0]

         for line in fiori_peaks:
             try:
                 key_output[f'Fiori PB ({line})'] = lines_info.loc[line, 'Fiori P/B']
             except KeyError:
                 key_output[f'Fiori PB ({line})'] = np.nan

         return key_output
```

```
[ ]: key_output = make_output()
     key_output.T
```

```
[ ]:                                    0
     Name                     GaAs_30kV_50pA
     Nominal beam energy [kV]          30.0
     Beam current [pA]                   50
     Process time                         6
     ICR                              16400
     Dead time [%]                     43.7
     Live time [s]                    120.0
     Duane-Hunt limit [kV]              NaN
     Scale [keV]                   0.010046
     Offset [keV]                  0.193846
     Energy resolution [eV]      128.675916
     Total counts                 1930388.0
     Background counts             757476.8
     FWTM/FWHM (Ga_Ka)                1.829
     FWTM/FWHM (Ga_La)                1.907
     Ratio (As_Ka/As_La)              1.303
     Ratio (Ga_Ka/Ga_La)              0.907
     Ratio (Ga_La/As_La)                2.368
     Fiori PB (Ga_La)                   585.8
     Fiori PB (As_La)                   236.2
     Fiori PB (Ga_Ka)                   761.5
```

[ ]:
```python
def save_output():
    # save the output-df to a csv file
    key_output.T.to_csv(f'results/{name}_output.csv')

    # saves the line info df to another csv file
    save_lines_info = True
    if save_lines_info:
        lines_info.to_csv(f'results/lines_info/{name}_lines_info.csv')
```

[ ]:
```python
save_output()
```

[ ]:
```python
# ###### snakkes.
# ![Image](https://folk.ntnu.no/brynjamm/marathon_dabz.gif)
```