RecSegImage-LA: Reconstruction, Segmentation of LA-ICP Imaging Data

In [10]:
```
import sys
sys.path.append("../")

from recsegimage import *

import numpy as np
import matplotlib.pyplot as plt
import glob
import re
%matplotlib inline
```

Reconstruction of images for all the analyzed metals

The following lines of code perform image reconstruction of LA-ICP-MS data, save the data in the results folder and generate plots of the reconstructed images. The final images are in order of acquisition in the raw data (Metal1, Metal2, Metal3, ..., MetalN)

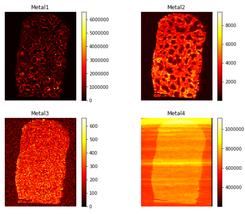
- foldername = string, name of the folder that includes the raw data files and the ipython script RecSegImage-LA.ipynb
- data_name = string, name given to the data (no blank spaces allowed in the name)
- spot_size = integer, spot size in microns of the laser used to acquire the data
- scan_rate = integer, scan rate of the laser in microns/second
- nmetals = integer, number of metals analyzed, when performing the images
- ldiscard = integer, number of columns on the far left side of the image to be eliminated in case there is sample carryover. Default value is 0

In [11]:
```
foldername = "data/"
data_name = "Example"
spot_size = 50
scan_rate = 15
nmetals = 4
ldiscard = 0

# Image reconstruction of the LA-ICP-MS raw data
final_matrices,sumdata = image_reconstruction(foldername,ldiscard,"*.xl",nmetals,spot_size,scan_rate
)

# Save the analysis in .csv files in the folder RegSegImage-LA/results folder
write_data_analysis(final_matrices,ldiscard,sumdata,nmetals)

# Generate the image plots of all the analyzed metals
generate_plot_all_metals(final_matrices, nmetals)
```



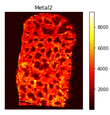
Reconstruction of the image of a single metal

Image reconstruction of one of all the analyzed metals. The metal index of the metal needs to be specified. The metal index corresponds to the order in which the metal is analyzed by the ICP-MS. For this particular example, the ICP-MS performs the readings of the metals in the following order: Bi, Fe, Zn, S. This means that the associated indexes are: Bi (Metal1, metal_index=1), Fe (Metal2, metal_index=2), Zn (Metal3, metal_index=3) and S (Metal4, metal_index=4)

- metal_index = integer, index of the metal that we want to plot. For example, for Metal 2 (Fe), the index is 2.

In [12]:
```
metal_index = 2

# Functions to generate image of one metal plot
generate_metal_plot(final_matrices,metal_index,nmetals)
```



Background Subtraction

Use the Zn image (or other metal that marks the tissue boundary) to differentiate tissue from background

- background_index = integer, index of the image used for background subtraction. In this case is the Zn image
- line = integer, row or column from which the standard deviation will be calculated. By default, the value is 1, which corresponds to the first row and column. The script will calculate the smallest standard deviation among the selected rows and columns
- std_threshold = integer, how many standard deviations will be tolerated to set the threshold of what is considered to be tissue and background

In [13]:
```
background_index = 3
line = 1
std_threshold = 4

background_mask = remove_background(final_matrices,background_index,line,std_threshold)
background_plot = generate_background_plot(background_mask)
```

Normalization with background subtraction

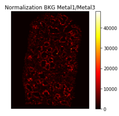
Normalization of the image with background subtraction. Background subtraction should be done first to obtain the background mask. The normalization corresponds to a pixel/pixel division of the images, so metal_numerator/metal_denominator should be specified in the following parameters:

- metal_numerator = integer, index of the metal that will correspond to the metal numerator in the division operation
- metal_denominator = integer, index of the metal that will correspond to the metal denominator in the division operation

The normalized image with background subtraction is saved as a text file in the results folder. The name of the file is: "Normalization Background Metal_numerator / Metal_denominator"

In [14]:
```
metal_numerator = 1
metal_denominator = 3

normalization_with_background(final_matrices,background_mask,metal_numerator,metal_denominator)
```



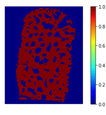
k-means segmentation

Segmentation of the images using k-means clustering. This is without spatial awareness.

- metal_segmentation_index = integer, index of the metal used for segmentation. In this example we use Fe and the index of Fe in the reconstructed data corresponds to 2
- clusters = integer, number of clusters to perform k-means segmentation. For this particular common example we had determined that the ideal number of clusters is 2

In [15]:
```
metal_segmentation_index = 2
segmentation_clusters = 2

label_image,segmented_image = segmentation(final_matrices,background_mask,metal_segmentation_index,s
egmentation_clusters)
```



k-means multimetal segmentation with neighboring pixel evaluation

Application of neighboring pixel evaluation using average filtering. The k-means segmentation part of the code should be run first before performing neighboring pixel evaluation. The number of clusters and metal segmentation index are the ones specified in the k-means segmentation part of the code. If the user wants to change these parameters, this can be done in the k-means segmentation part of the code. No inputs are required here by the user

Multimetal segmented images correspond to the segmented images using Fe and the background mask (Zn) for segmentation. A weighted image corresponds to the image after filtering to determine the tissue boundaries.

In [16]:
```
weighted_pixels = neighbouring_average(label_image,background_mask)
```

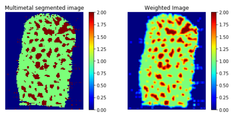


Image masks of the segmented areas

It is possible to set up to four diferent areas determined by segmentation and neighboring pixel evaluation. It is necessary to set up the cutoffs of the areas in relation to the weighted image (0 to 18 scale). For this particular example we set up the cutoff values for:

Area 1 = Background (values between 0 and 5) Area 2 = Red Pulp (values between 6 and 10) Area 3 = Marginal zone (values between 11 and 14) Area 4 = White pulp (values between 15 and 18)

The variables shown should specify the low and high cutoff of a particular area:

- low_A1 = integer, low cutoff of Area 1
- high_A1 = integer, high cutoff of Area 1
- low_A2 = integer, low cutoff of Area 2
- high_A2 = integer, high cutoff of Area 2
- low_A3 = integer, low cutoff of Area 3
- high_A3 = integer, high cutoff of Area 3
- low_A4 = integer, low cutoff of Area 4
- high_A4 = integer, high cutoff of Area 4

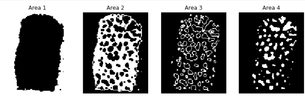
In [17]:
```
low_A1 = 0
high_A1 = 0.59

low_A2 = 0.60
high_A2 = 1.19

low_A3 = 1.20
high_A3 = 1.49

low_A4 = 1.50
high_A4 = 2.0

area1,area2,area3,area4 = image_masks(weighted_pixels,low_A1,high_A1,low_A2,high_A2,low_A3,high_A3,l
ow_A4,high_A4)
```



Quantitation in different segments

Quantitation of pixels in the different segments determined by the image masks. Four different areas of a tissue were determined after segmentation. The segmented areas can be use to get the number of pixels in each of the particular areas, find the average and error of any of the metals in each of the segmented areas. It is necessary to choose the metal that we desire to quantify in each of the areas as the (quantitation_index). In this particular example, we want to quantify the Bismuth (quantitation_index=1) so the index needs to be set to the Bi index (Bismuth index is 1). It is also possible to quantify the average signal of any of the other metals, for example if we want to quantify the Fe in each of the segmented areas we should set (quantitation_index=2) as the Fe corresponds to the metal with the index=2.

- quantitation_index = int, index of the metal that we want to quantify in each of the segments

In [18]:
```
quantitation_index = 2

quantitation_segments(final_matrices,area1,area2,area3,area4,quantitation_index)
```

```
Area 1 Quantitation:
Area 1 pixels are: 3308
Area 1 average is: 518.7432395417972
Area 1 error is: 7.172345137306754

Area 2 Quantitation:
Area 2 pixels are: 4822
Area 2 average is: 3890.348333680314
Area 2 error is: 15.287243249383316

Area 3 Quantitation:
Area 3 pixels are: 1545
Area 3 average is: 2777.185256709202 3
Area 3 error is: 23.262159826047085

Area 4 Quantitation:
```

```
import numpy as np
import matplotlib.pyplot as plt
import glob
import re
import warnings; warnings.simplefilter('ignore')
from sklearn.cluster import KMeans

global ldiscard
ldiscard=0

def atoi(text):
    '''
        Natural Sorting of data: Functions used to organize the data in terms of
          type. Import the package re. Allow the use of
        backslashes to indicate special forms without evoking the special meaning.

        atoi function:
            input = str, text
            output = str and int, text
    '''
    return int(text) if text.isdigit() else text


def natural_keys(text):
    '''
        alist.sort(key=natural_keys) sorts in human order
        http://nedbatchelder.com/blog/200712/human_sorting.html
        (See Toothy's implementation in the comments)
    '''
    return [ atoi(c) for c in re.split('(\d+)', text) ]


def processfile(filename,ldiscard,spot_size,scan_rate):
    '''
        Function processfile used to load and read the data in a single file

        Input:
            filename = str, name of the folder where the data files are stored
            ldiscard = integer, number of columns on the far left side of the image
               to be eliminated. Default value is 0
            spot_size = integer, spot size in microns of the laser used to acquire
              the data
            scan_rate = integer, scan rate of the laser in microns/second

        Ouput:
            new_matrix = nd array, reduced data summed by sumdata amounts
            sumdata = int, number of data points that make one pixel (depends on
              laser spot size and scan rate)
    '''
```

```
    data_matrix = np.loadtxt(filename, delimiter=",", skiprows=(2+ldiscard))
    pixel_time = spot_size/scan_rate
    data_point_time = data_matrix[6,0]-data_matrix[5,0]
    sumdata = int(round(pixel_time/data_point_time))
    nrows = data_matrix.shape[0]//sumdata
    ncols = data_matrix.shape[1]-1
    new_matrix = np.zeros((nrows+1,ncols))
    for n in range(nrows):
        new_matrix[n,:] = np.sum(data_matrix[n*sumdata:(n+1)*sumdata,1:],axis=0)

    new_matrix[n+1,:] = np.sum(data_matrix[(n+1)*sumdata:,1:],axis=0)
    return new_matrix, sumdata


def image_reconstruction
  (foldername,ldiscard,extension,ncolumns,spot_size,scan_rate):
    '''
        Function image_reconstruction used for load files in a directory

        Input:
            foldername = str, folder in which the data files and script are located
            ldiscard = integer, number of columns on the far left side of the image
               to be eliminated. Default value is 0
            extension = str, extension of the data files
            ncolumns = int, number of columns in the reduced matrix (equal to the
              number of metals)
            spot_size = integer, spot size in microns of the laser used to acquire
              the data
            scan_rate = integer, scan rate of the laser in microns/second

            Output:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data of different analyzed metals
            sumdata = int, number of data points that make one pixel (depends on
              laser spot size and scan rate)
    '''
    files = glob.glob1(foldername,extension)
    files.sort(key=natural_keys)
    nfiles = len(files)
    dic_data = {}
    for n in range(ncolumns):
        dic_data[n]=[]
    for file in files:
        processed_matrix,sumdata = processfile(foldername
          +'/'+file,ldiscard,spot_size,scan_rate)
        for col in range(ncolumns):
            dic_data[col].append(processed_matrix[:,col])
    final_matrices={}
    for n in range(ncolumns):
```

```
        final_matrices[n]=np.array(dic_data[n])
    return final_matrices, sumdata


def write_data_analysis
  (final_matrices,ldiscard,sumdata,nmetals,foldername='results'):
    '''
        Function write_data_analysis use to write the processed data into separate
          csv files

        Input:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data of different analyzed metals
            ldiscard = int, number of datapoints discarded in each of the files, if
               needed
            sumdata = int, number of data points that make one pixel (depends on
              laser spot size and scan rate)
            nmetals = int, number of metals analyzed, when performing the images

        Output:
            files for each of the analyzed metals writen in .csv inside the /
              results directory
    '''
    dic_of_metals = {}
    keys = range(nmetals)
    for i in keys:
        dic_of_metals[i+1] = "Metal" + str(i+1)
    for metal in final_matrices:
        filename = foldername + "/Reconstruction-%s.xl" % (dic_of_metals[metal+1])
        np.savetxt(filename, final_matrices[metal], delimiter=',', newline='\n')


def generate_plot_all_metals(final_matrices,nmetals):
    '''
        Function genetate_plot_all_metals to plot all the metal images in one plot

        Input:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data matrices of different analyzed metals
            nmetals = int, number of metals analyzed, when performing the images

        Output:
            matplotlib image of the analyzed metals in one image (in the Jupyter
              notebook)
    '''
    dic_of_metals = {}
    keys = range(nmetals)
    for i in keys:
        dic_of_metals[i+1] = "Metal" + str(i+1)
```

```
    fig = plt.figure(figsize=[10,8])
    for n in range(1,nmetals+1):
        ax = fig.add_subplot(2,2,n)
        plt.imshow(final_matrices[n-1], interpolation='None', cmap=plt.cm.hot)
        plt.title("%s" % dic_of_metals[n])
        plt.axis('off')
        plt.colorbar()
    plt.show()


def generate_metal_plot(final_matrices,metal_index,nmetals):
    '''
        Function generate_metal_plot used to generate a plot of one metal

        Input:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data matrices of different analyzed metals
            metal_index = int, index of the specified metal in the dictionary (1 to
               nmetals)
            nmetals = int, number of metals analyzed, when performing the images

        Output:
            matplotlib image of a particular metal inline
    '''
    dic_of_metals = {}
    keys = range(nmetals)
    for i in keys:
        dic_of_metals[i+1] = "Metal" + str(i+1)
    fig = plt.figure(figsize=[5,4])
    ax = fig.add_subplot(1,1,1)
    plt.imshow(final_matrices[metal_index-1], interpolation='None',
      cmap=plt.cm.hot)
    plt.title("%s" % dic_of_metals[metal_index])
    plt.axis('off')
    plt.colorbar()
    plt.show()


def populate_border(matrix):
    '''
        Function populate_border used to fine tune, delimitate border of the tissue
           sample, based on any metal content. This function is
        concatenated with the remove_background function

        Input:
            matrix = np array, correspond to the matrix index_threshold. This is
              the matrix that have the applied condition
            matrix < threshold, this matrix correspond to a boolean matrix which
              have defined True and False values.
```

```
        Output:
            border = np array
    '''
    border = np.ones(matrix.shape)
    for n in range(matrix.shape[0]):
        for m in range(matrix.shape[1]):
            if matrix[n, m] == True:
                border[n, m] = 0
            elif matrix[n, m] == False:
                break
    for n in range(matrix.shape[0]):
        for m in range(matrix.shape[1]):
            if matrix[n, matrix.shape[1] - m - 1] == True:
                border[n, matrix.shape[1] - m - 1] = 0
            elif matrix[n, matrix.shape[1] - m - 1] == False:
                break
    for m in range(matrix.shape[1]):
        for n in range(matrix.shape[0]):
            if matrix[n, m] == True:
                border[n, m] = 0
            elif matrix[n, m] == False:
                break
    for m in range(matrix.shape[1]):
        for n in range(matrix.shape[0]):
            if matrix[matrix.shape[0] - n - 1, m] == True:
                border[matrix.shape[0] - n - 1, m] = 0
            elif matrix[matrix.shape[0] - n - 1, m] == False:
                break
    return border


def remove_background(final_matrices,background_index,line,std_threshold):
    '''
        Function remove_background used to calculate the average and std of the
          background and set the theshold values

        Input:
            matrix = np array, data matrix with the Zn data final_matrices
              [Zn_index]
            line = int, index of the line that will be used to perform the
              background calculation, usually 0
            tolerance_std = int, tolerance of the std, usually is 3

        Output:
            background_mask = np array, background mask of the image data
    '''
    matrix = final_matrices[background_index-1]
    average_col = np.mean(matrix[:, line-1])
```

```
    std_col = np.std(matrix[:, line-1])
    average_row = np.mean(matrix[line-1, :])
    std_row = np.std(matrix[line-1, :])
    if std_col < std_row:
        average = average_col
        std = std_col
    else:
        average = average_row
        std = std_row
    threshold = average + std_threshold*std
    index_threshold = matrix < threshold
    background_mask = populate_border(index_threshold)
    return background_mask


def generate_background_plot(background_mask):
    '''
        Function generate_background_plot used to generate a plot of the background
           mask

        Input:
            background_mask = np array, background mask of the image data

        Output:
            matplotlib inline image of the background mask
    '''
    fig = plt.figure(figsize=[5,4])
    ax = fig.add_subplot(1,1,1)
    plt.imshow(background_mask, interpolation='None', cmap=plt.cm.hot)
    plt.title('Background mask')
    plt.axis('off')
    plt.show()


def normalization_with_background
  (final_matrices,background_mask,metal_numerator,metal_denominator,vmin=None,vmax=
  None,inter='None',
                                foldername='results'):
    '''
        Function normalization_with_background to divide two matrices (metal1/
          metal2), saved the data and plotted it inline

        Input:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data matrices of different analyzed metals
            background_mask = np array, background mask of the image data
            metal_numerator = integer, index of the metal that will correspond to
              the metal numerator in the division operation
            metal_denominator = integer, index of the metal that will correspond to
```

```
                the metal denominator in the division operation

        Output:
            file inside the results directory with the results of the
              metal_numerator/metal_denominator division
            matplotlib inline image of the metal_numerator/metal_denominator
              division
    '''
    old_err_state = np.seterr(divide='raise')
    ignored_states = np.seterr(**old_err_state)
    fig = plt.figure(figsize=[5,4])
    ax = fig.add_subplot(1,1,1)
    division_background = (final_matrices[metal_numerator-1]/final_matrices
      [metal_denominator-1])*background_mask
    division_background[np.isnan(division_background)] = 0
    np.savetxt(foldername+'/Normalization-Background-Metal'+str(metal_numerator)
      +'-'+'Metal'+str
      (metal_denominator),division_background,delimiter=',',newline='\n')
    plt.imshow
      (division_background,interpolation=inter,vmin=vmin,vmax=vmax,cmap=plt.cm.hot)
    plt.title('Normalization BKG '+'Metal'+str(metal_numerator)+'/Metal'+str
      (metal_denominator))
    plt.axis('off')
    plt.colorbar()
    plt.show()


def segmentation
  (final_matrices,background_mask,metal_segmentation_index,segmentation_clusters):
    '''
        Function segmentation for the segmentation of the images using k-means
          clustering without filtering

        Input:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data matrices of different analyzed metals
            background_mask = np array, background mask of the image data
            metal_segmentation_index = int, index of the metal used for
              segmentation.
            segmentation_clusters = int, number of clusters to perform k-means
              segmentation.

        Output:
            label_image = np array, segmented image with its labels
            segmented_image = np array, segmented image with its centroids
    '''
    metal_segmentation = final_matrices[metal_segmentation_index-1]*background_mask
    rows = metal_segmentation.shape[0]
    columns = metal_segmentation.shape[1]
```

```
    metal_segmentation_vector = metal_segmentation.reshape(rows*columns, 1)
    # specifies that kmeans will be applied with n-clusters
    kmeans = KMeans(segmentation_clusters)
    # Perform kmeans over metal_segmentation_vector
    kmeans.fit(metal_segmentation_vector)
    # Find cluster center associated with each data point
    segmented_vector = kmeans.cluster_centers_[kmeans.predict
      (metal_segmentation_vector)]
    # Find labels associated with each cluster
    centroids = np.sort(np.unique(segmented_vector))
    labels = np.zeros(segmented_vector.shape)
    for index, centroid in enumerate(centroids):
        labels[segmented_vector==centroid] = index
    label_image = labels.reshape(rows, columns)
    # Reshaped of the image, plotting and comparison with raw_data
    segmented_image = segmented_vector.reshape(rows, columns)
    # Image plot of the labels
    plt.imshow(label_image, cmap='jet')
    plt.colorbar()
    plt.axis('off')
    plt.show()
    return label_image,segmented_image


def neighbouring_average(label_image,background_mask):
    '''
        Function neighbouring_average for filtering the multimetal image

        Input:
            label_image = np array, segmented image with its labels
            background_mask = np array, background mask of the image data

        Output:
            weighted_pixels = np array, filtering of label image data
    '''
    # Re-asignation of zero values
    label_image[label_image == 0] = 2
    # Differentiation of bacgkround using the background mask
    M = label_image*background_mask
    # Weighted pixel calculation
    weighted_pixels = np.zeros(M.shape)
    for n in range(1, M.shape[0]-1):
        for m in range(1, M.shape[1]-1):
            weighted_pixels[n, m]= (M[n-1,m-1] + M[n-1,m] + M[n-1,m+1] + M[n,m-1] +
               M[n,m] + M[n, m+1] + M[n+1, m-1] + M[n+1, m] + M[n+1, m+1])/9
    # Image generation of the multimetal segmentation
    plt.figure(figsize=(9, 4))
    ax1=plt.subplot(1, 2, 1)
    plt.imshow(M, cmap='jet')
```

```
    plt.colorbar()
    plt.axis('off')
    plt.title('Multimetal segmented image')
    # Image generation of the weighted image
    ax1=plt.subplot(1, 2, 2)
    plt.imshow(weighted_pixels, interpolation='none', cmap='jet')
    plt.colorbar()
    plt.axis('off')
    plt.title('Weighted Image')
    plt.show()
    return weighted_pixels

def image_masks(weighted_pixels, low_A1, high_A1, low_A2, high_A2, low_A3, high_A3, low_A4, high_A4):
    '''
        Function image_masks to obtain the masks images of the segmented areas

        Input:
            low_A1 = integer, low cutoff of Area 1
            high_A1 = integer, high cutoff of Area 1
            low_A2 = integer, low cutoff of Area 2
            high_A2 = integer, high cutoff of Area 2
            low_A3 = integer, low cutoff of Area 3
            high_A3 = integer, high cutoff of Area 3
            low_A4 = integer, low cutoff of Area 4
            high_A4 = integer, high cutoff of Area 4

        Output:
            area1 = image of image mask of area 1
            area2 = image of image mask of area 2
            area3 = image of image mask of area 3
            area4 = image of image mask of area 4
    '''
    area1 = (weighted_pixels <= high_A1)
    area2 = (weighted_pixels <= high_A2) ^ (weighted_pixels <= low_A2)
    area3 = (weighted_pixels <= high_A3) ^ (weighted_pixels <= low_A3)
    area4 = (weighted_pixels <= high_A4) ^ (weighted_pixels <= low_A4)
    # Image generation
    plt.figure(figsize=(12, 9))
    ax=plt.subplot(1, 4, 1)
    plt.imshow(area1, cmap='gray')
    plt.axis('off')
    plt.title('Area 1')
    ax=plt.subplot(1, 4, 2)
    plt.imshow(area2, interpolation='none', cmap='gray')
    plt.axis('off')
    plt.title('Area 2')
    ax=plt.subplot(1, 4, 3)
    plt.imshow(area3, interpolation='none', cmap='gray')
```

```
    plt.axis('off')
    plt.title('Area 3')
    ax=plt.subplot(1, 4, 4)
    plt.imshow(area4, interpolation='none', cmap='gray')
    plt.axis('off')
    plt.title('Area 4')
    plt.show()
    return area1, area2, area3, area4

def quantitation_segments
  (final_matrices,area1,area2,area3,area4,quantitation_index):
    '''
        Function quantitation_segments to obtain the masks images of the segmented
          areas

        Input:
            final_matrices = dic, dictionary composed of np.arrays of the final
              data matrices of different analyzed metals
            area1 = image of image mask of area 1
            area2 = image of image mask of area 2
            area3 = image of image mask of area 3
            area4 = image of image mask of area 4
            quantitation_index = int, index of the metal that we want to quantify
              in each of the segments

        Output:
            inline results of the averages, standard error and number of pixels of
              the segmented areas
    '''
    metal_quantitation = final_matrices[quantitation_index-1]
    # Area 1
    print ('Area 1 Quantitation:')
    pixels_A1 = len(metal_quantitation[area1])
    avg_A1 = np.mean(metal_quantitation[area1])
    error_A1 = np.std(metal_quantitation[area1])/np.sqrt(pixels_A1)
    print ('Area 1 pixels are:', pixels_A1)
    print ('Area 1 average is:', avg_A1)
    print ('Area 1 error is:', error_A1)
    # Area 2
    print (' ')
    print ('Area 2 Quantitation:')
    pixels_A2 = len(metal_quantitation[area2])
    avg_A2 = np.mean(metal_quantitation[area2])
    error_A2 = np.std(metal_quantitation[area2])/np.sqrt(pixels_A2)
    print ('Area 2 pixels are:', pixels_A2)
    print ('Area 2 average is:', avg_A2)
    print ('Area 2 error is:', error_A2)
    # Area 3
    print (' ')
```

```
    print ('Area 3 Quantitation:')
    pixels_A3 = len(metal_quantitation[area3])
    avg_A3 = np.mean(metal_quantitation[area3])
    error_A3 = np.std(metal_quantitation[area3])/np.sqrt(pixels_A3)
    print ('Area 3 pixels are:', pixels_A3)
    print ('Area 3 average is:', avg_A3)
    print ('Area 3 error is:', error_A3)
    # Area 4
    print (' ')
    print ('Area 4 Quantitation:')
    pixels_A4 = len(metal_quantitation[area4])
    avg_A4 = np.mean(metal_quantitation[area4])
    error_A4 = np.std(metal_quantitation[area4])/np.sqrt(pixels_A4)
    print ('Area 4 pixels are:', pixels_A4)
    print ('Area 4 average is:', avg_A4)
    print ('Area 4 error is:', error_A4)
```