```
from PIL import Image
import numpy as np
import numpy.linalg as la
import math
import visual_map
import watershed
import variance_map
import os
from numba import jit


"""
# creates mask that blocks the centre and extremities of fourier transform
# leaving only city block scale signal
def fourier_mask(shape):

    # read dimensions of image to create mask for
    x_pix = int(shape[0])
    y_pix = int(shape[1])

    print('input image has dimensions of x = %d and y = %d') % (x_pix, y_pix)

    # set size of mask (circular shape)
    mask_inner_radius = x_pix / 13
    mask_outer_radius = x_pix / 4
    x_cent = x_pix / 2
    y_cent = y_pix / 2

    # initialising zeroed array to hold mask
    f_mask = np.zeros((x_pix, y_pix))

    # cycling over all pixels in the zeroed array
    for x in xrange(0, x_pix):
        for y in xrange(0, y_pix):
            r = math.sqrt((x - x_cent) ** 2 + (y - y_cent) ** 2)

            # setting circular region larger than mask_inner_radius and less than
            # mask_outer_radius to 1, all other regions of array left at 0
            if ((r > mask_inner_radius) & (r < mask_outer_radius)):
                f_mask[x][y] = 1

    # mask saved as image for debugging purposes, maybe remove?
    img = Image.fromarray(np.uint8(f_mask))
    img.save('./images/mask.png')
    print('mask saved to disk')

    return f_mask


def threshold_tweak(ftrans, max_peak, peaks):

    thresh_step = 0.0001

    # setting threshold values to iterate over
    thresh_iter = np.arange(0.001, 0.2, thresh_step)


    for thresh in thresh_iter:
        ftrans_temp = ftrans
        ftrans_temp[ftrans_temp < (max_peak * thresh)] = 0

        # uncomment to make function verbose
        #print('%d non zero pixels detected at threshold of %f %% of peak value') \
            % (np.count_nonzero(ftrans_temp), thresh * 100)

        if(np.count_nonzero(ftrans_temp) == peaks):
            print('%d peaks found when the threshhold = %f %% of the max peak \
                intensity') % (peaks, thresh * 100)
            return ftrans_temp

        if(np.count_nonzero(ftrans_temp) < peaks):
            print('threshold iteration has skipped over %d peak values, try again \
                with a finer threshold step') % peaks

            return 0

    print('no good threshold found, sorry...')
```

```
    return 0


# extracts the angle of inclination of the chip from a filtered 2dfft
def find_angle(clean_fft, peaks):

    args = np.zeros((peaks, 2))

    # fills array args up with the indicies of non zero pixels
    for peak in xrange(0, peaks):
        args[peak]  = np.unravel_index(np.argmax(clean_fft), np.shape(clean_fft))

        print('peak %d has intensity of %f') % (peak + 1, np.amax(clean_fft))
        print('and position of [%d, %d]') % (args[peak][0], args[peak][1])

        clean_fft[int(args[peak][0])][int(args[peak][1])] = 0

    # claculate vector between two identified pixels
    vector = np.zeros(2)
    vector[0] = args[0][0] - args[1][0]
    vector[1] = args[0][1] - args[1][1]

    #finding magnitude of vector
    vec_mag = math.sqrt(vector[0] ** 2 + vector[1] ** 2)

    print('vec_mag = %f') % vec_mag

    # setting reference vertical vector
    vert_vect = np.array([0, 1])

    #computing angle between calculated and reference vector
    angle = vector_angle(vector, vert_vect)

    deg_angle = angle * 360 / (2 * math.pi)
    print('angle = %f degrees before quadrant correction') % deg_angle

    # finding quadrant in which the calculated angle is closest to reference values of
    # 0, pi/2, pi and (3 * pi) / 4
    quadrant_angles = np.array([0.0, math.pi / 2.0, math.pi, - math.pi / 2.0])
    quadrant_delta = np.array([0.0, 0.0, 0.0, 0.0])

    for quadrant in xrange(0,4):
      quadrant_delta[quadrant] = angle - quadrant_angles[quadrant]

    quadrant = int(np.argmin(np.fabs(quadrant_delta)))

    angle = quadrant_delta[quadrant]

    deg_angle = angle * 360 / (2 * math.pi)
    print('chip is rotated  %f degrees counter clockwise') % deg_angle

    return angle, vec_mag

# returns the angle in radians between vectors v1 and v2
def vector_angle(v1, v2):

    cosang = np.dot(v1, v2)
    sinang = la.norm(np.cross(v1, v2))

    return np.arctan2(sinang, cosang)


# attempts to determine the orientation of a chip image
# (clockwise rotation in radians)
def orient(filename):

    img_array = np.asarray(Image.open(filename).convert('L'))
    print('image loaded')

    shape = np.shape(img_array)

    # performing fourier transform
    ftrans = np.fft.fft2(img_array)

    # gets mask for fourier transform
    f_mask = fourier_mask(shape)
```

```
    # sets peak intensity to be at the centre of the image
    ftrans = np.fft.fftshift(ftrans)

    # determine peak of fourier transform
    max_peak = np.max(np.abs(ftrans))

    # convolve mask with fourier data
    masked_ftrans = ftrans * f_mask

    # image of mask loaded into image and saved
    img = Image.fromarray(np.uint8(masked_ftrans))
    img.save('./images/masked_ftrans.png')
    print('masked ftrans saved')

    # number of peaks that the threshold will be tweaked to find (2 by default),
    # different angle determination method required
    # with more than 2 peaks
    peaks = 2

    masked_ftrans = threshold_tweak(masked_ftrans, max_peak, peaks)

    # log scale data
    abs_data = 1 + np.abs(masked_ftrans)
    c = 255.0 / np.log(1 + max_peak)
    log_data = c * np.log(abs_data)

    # array loaded into image and saved
    img = Image.fromarray(np.uint8(log_data))
    img.save('./images/orient.png')
    print('image saved to disk')

    theta, vec_mag = find_angle(log_data, peaks)

    return theta, vec_mag

"""

@jit(nopython=True)
def rotate (x, y, rot_matrix):
    col_vec = np.zeros((2))
    col_vec[0] = x
    col_vec[1] = y
    col_vec = np.dot(rot_matrix, col_vec)

    return col_vec[0], col_vec[1]


# generates a mask from the theoretical layout of a chip
def chip_mask_gen (x_pix_max, y_pix_max, cell_real_size,
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

    chip_mask = chip_mask_crunch(x_pix_max, y_pix_max, cell_real_size,
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z)

    chip_mask = np.flipud(chip_mask)
    chip_mask = np.rot90(chip_mask, 3)

    return chip_mask


@jit(nopython=True)
def chip_mask_crunch (x_pix_max, y_pix_max, cell_real_size,
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

    # final cell index
    i_max = 11664

    i_half = 5832

    # initializing mask array
    mask = np.zeros((x_pix_max, y_pix_max))

    cell_pix_size_float = cell_real_size * pix_scale # width of cell in pixels
    cell_pix_size_int = int(math.ceil(cell_pix_size_float))

    X_pix = np.zeros(i_max)
```

```
    Y_pix = np.zeros(i_max)

    # location of cells in terms of image pixels
    for i in xrange(0,11664):

        X_pix[i] = round((X[i] + x_real_offset) * pix_scale)
        Y_pix[i] = round((Y[i] + y_real_offset) * pix_scale)

    x_mid = X_pix[i_half]
    y_mid = Y_pix[i_half]

    rot_matrix = np.zeros((2, 2))
    rot_matrix[0][0] = math.cos(theta)
    rot_matrix[1][1] = math.cos(theta)
    rot_matrix[1][0] = - math.sin(theta)
    rot_matrix[0][1] = math.sin(theta)

    # accounting for x, y shift of the middle of mask so rotation only takes
    # place around centre of chip
    x, y = rotate(x_mid, y_mid, rot_matrix)
    delt_x = x - x_mid
    delt_y = y - y_mid

    for i in xrange(0, i_max):
        for r_x in xrange(-cell_pix_size_int, cell_pix_size_int):
            for r_y in xrange(-cell_pix_size_int, cell_pix_size_int):

                if (math.sqrt(r_x ** 2 + r_y ** 2) <= cell_pix_size_float):

                    x = int(round(X_pix[i] + r_x)) # temporary x
                    y = int(round(Y_pix[i] + r_y)) # temporary y

                    x, y = rotate(x, y, rot_matrix)

                    # correcting for non centred rotation
                    x = int(round(x - delt_x))
                    y = int(round(y - delt_y))

                    if (x >= 0 and x < x_pix_max and y >= 0 and y < y_pix_max):
                            mask[int(x)][int(y)] = Z[int(i)]

    return mask


@jit(nopython=True)
def rect_cent_mask_gen (x_pix_max, y_pix_max, pix_scale, real_circ_rad, good_rect_log, cent_cords):

    pix_circ_rad = int(round(real_circ_rad * pix_scale))

    mask = np.zeros((x_pix_max, y_pix_max))

    length = good_rect_log.size

    for count in xrange(1, length):
        for x in xrange(-pix_circ_rad, pix_circ_rad):
            for y in xrange(-pix_circ_rad, pix_circ_rad):

                # only masking for rectangles with adjacent neighbours (horizontal and vertical)
                if (good_rect_log[count] == 1):
                    if (math.sqrt(x ** 2 + y ** 2) <= pix_circ_rad):

                        if ((cent_cords[count][0] + x >= 0) and (cent_cords[count][0] + x < x_pix_max) \
                            and (cent_cords[count][1] + y >= 0) and (cent_cords[count][1] + y < y_pix_max)):

                            mask[cent_cords[count][0] + x][cent_cords[count][1] + y] = 1.0

    return mask


# returns the index of an iterable given its value
@jit(nopython=True)
def index (value, min, step):
    I = int(round((value - min) / step) - 1)

    return I
```

```python
# returns the value of an iterable for a given index
@jit(nopython=True)
def de_index (I, min, step):
    value = ((I + 1) * step) + min

    return value


@jit(nopython=True)
def smart_img_clean (rect_mask, img_array, x_pix_max, y_pix_max):

    for x in xrange(0, y_pix_max):
        for y in xrange(0, x_pix_max):
            if ((rect_mask[x][y] <= 0.5) & (img_array[x][y] >= 255)):
                img_array[x][y] = 0

    return img_array


# Search mask generation parameter space to find optimal fitting
@jit
def sweep(filename, x_r_off_min, x_r_off_max, y_r_off_min, y_r_off_max, \
    real_trans_step_x, real_trans_step_y, cell_real_size, \
    X, Y, Z, cent_cords, good_rect_log, img_array, rect_mask,
    pix_scale, theta, std_dev_map, sweep_type=0):

    print ('in sweep' )

    # stops searching of space outside of this zone
    x_hard_min = 8.0
    x_hard_max = 16
    y_hard_min = 0
    y_hard_max = 5.2

    # size of image
    #x_pix_max = 1292
    #y_pix_max = 964

    print ('before shape' )

    img_shape = np.shape(img_array)

    x_pix_max = img_shape[0]
    y_pix_max = img_shape[1]

    print ('after shape' )

    real_circ_rad = 1.05 # radius of circle around identified rectangles to be masked (mm)

    #img = Image.fromarray(np.uint8(img_array))
    #img.save('./images/smart_clean_test.png')
    #print('smart_clean_test saved')

    # values to iterate x_real_offset and y_real_offset over
    x_r_off_iter = np.arange(x_r_off_min, x_r_off_max, real_trans_step_x)
    y_r_off_iter = np.arange(y_r_off_min, y_r_off_max, real_trans_step_y)

    print ('iterables created' )

    # calculating number of solutions
    ind_x_max = 1 + index(x_r_off_max, x_r_off_min, real_trans_step_x)
    ind_y_max = 1 + index(y_r_off_max, y_r_off_min, real_trans_step_y)

    print ('calculating array dimension sizes' )

    sums = np.zeros((ind_x_max, ind_y_max))
    sums_omni = np.zeros((ind_x_max, ind_y_max))
    conv_std_dev_img_array = np.zeros((x_pix_max, y_pix_max))

    print ('about to start loop' )


    # iterating over x and y offset values for chip mask
    for x_real_offset in x_r_off_iter:
        for y_real_offset in y_r_off_iter:

            print ('x offset = %f' ) % x_real_offset
```

```
            print ('y offset = %f') % y_real_offset

            sum_current = 0
            sum_current_rect = 0
            non_zero_pixels = 0
            sum_current_std_dev = 0

            if ((x_real_offset > x_hard_min) & (x_real_offset < x_hard_max) & \
                (y_real_offset > y_hard_min) & (y_real_offset < y_hard_max)):

                # creating mask of chip
                chip_mask = chip_mask_gen(y_pix_max, x_pix_max, cell_real_size, x_real_offset, \
                    y_real_offset, pix_scale, theta, X, Y, Z)

                # convolving chip mask with image
                chip_conv_img_array = img_array * chip_mask
                sum_current = np.sum(chip_conv_img_array)

                # sweep using a larger cell size and against the good rect mask to
                # ensure a good city block match
                if (sweep_type == 1):

                    # convolving chip mask convolved image with known city block mask
                    conv_rect_img_array = chip_conv_img_array * rect_mask

                    non_zero_pixels = np.count_nonzero(conv_rect_img_array)
                    print ('%d non zero pixels') % non_zero_pixels

                    # loading array into image and saving
                    #img = Image.fromarray(np.uint8(conv_rect_img_array))
                    #imname = ('./images/%f_%f.png') % (round(x_real_offset, 4), round(y_real_offset, 4))
                    #img.save(imname)

                    sum_current_rect = np.sum(conv_rect_img_array)

                    # convolving std_dev_map with chip_mask
                    conv_std_dev_img_array = std_dev_map * chip_mask

                    # convert to non zero pix count!!
                    sum_current_std_dev = np.count_nonzero(conv_std_dev_img_array)

                    print ('%d non zero std dev pixels') % sum_current_std_dev


            ind_x = index(x_real_offset, x_r_off_min, real_trans_step_x)
            ind_y = index(y_real_offset, y_r_off_min, real_trans_step_y)

            sums[ind_x, ind_y] = sum_current

            # edit for different weighting!
            #sums_omni[ind_x, ind_y] = non_zero_pixels + (1.0 / 15000.0) * sum_current + (1.0 / 200) * sum_cu
rrent_std_dev
            sums_omni[ind_x, ind_y] = non_zero_pixels + sum_current_std_dev

            print ('sum current = %f') % sums[ind_x, ind_y]
            print ('sum_omni = %f') % sums_omni[ind_x, ind_y]

    if (sweep_type == 1):

        # returns the indicies of the sums element with the highest value
        i, j = np.unravel_index(sums_omni.argmax(), sums_omni.shape)
        print (i, j)
        x_real_offset = de_index(i, x_r_off_min, real_trans_step_x)
        y_real_offset = de_index(j, y_r_off_min, real_trans_step_y)

        print ('%f pixels per mm, x offset of %f mm and y offset of %f mm') % \
            (pix_scale, x_real_offset, y_real_offset)

        # calculating and displaying the convolution of the best fit mask with the input image
        chip_mask = chip_mask_gen(y_pix_max, x_pix_max, cell_real_size, x_real_offset, \
            y_real_offset, pix_scale, theta, X, Y, Z)

        img_array = img_array * chip_mask * rect_mask

        # loading array into image and saving
        img = Image.fromarray(np.uint8(img_array))
        img.save('./images/type1_fit.png')
```

```
    elif (sweep_type == 0):

        print ('sweep_type = %d' ) % sweep_type

        print ('determining best fit from sweep type 0' )

        # returns the indicies of the sums element with the highest value
        i, j = np.unravel_index(sums.argmax(), sums.shape)

        x_real_offset = de_index(i, x_r_off_min, real_trans_step_x)
        y_real_offset = de_index(j, y_r_off_min, real_trans_step_y)

        print ('%f pixels per mm, x offset of %f mm and y offset of %f mm' ) % \
            (pix_scale, x_real_offset, y_real_offset)

        # calculating and displaying the convolution of the best fit mask with the input image

        chip_mask = chip_mask_gen(y_pix_max, x_pix_max, cell_real_size, x_real_offset, \
            y_real_offset, pix_scale, theta, X, Y, Z)

        img_array = img_array * chip_mask

        img = Image.fromarray(np.uint8(img_array))
        img.save('./images/type0_fit.png' )
        print ('image saved' )

    # returns best fit mask generation parameters and image data
    return img_array, pix_scale, x_real_offset, y_real_offset, theta



#
def meta_sweep(filename, filename_out):

    # opening image and converting to greyscale
    img_array = np.asarray(Image.open(filename).convert('L' ))
    img_array.flags.writeable = True # making array readable

    X, Y, Z = visual_map.main() # get real space position of cells on chip
    Z[Z == 2] = 0
    Z[Z == 7] = 0
    Z[Z == 4] = 1

    # get fitting information from feature recognition code
    # format of angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count
    # getting fit which gives lowest pixel scale and angle error
    data, cent_cords_null, good_rect_log_null = watershed.error_minimize(filename, 0)

    # getting fit which yields highest number of rectangles
    data_null, cent_cords, good_rect_log = watershed.error_minimize(filename, 1)

    theta = - math.radians(data[0])
    pix_scale = data[2]

    print ('theta = %f rads' ) % theta
    print ('pix scale = %f pixels per mm' ) % pix_scale

    std_dev_box_size = 2
    std_dev_map = variance_map.main(filename, std_dev_box_size)

    img_shape = np.shape(img_array)

    x_pix_max = img_shape[1]
    y_pix_max = img_shape[0]

    real_circ_rad = 1.05 # radius of circle around identified rectangles to be masked (mm)
    rect_mask = rect_cent_mask_gen(x_pix_max, y_pix_max, pix_scale, real_circ_rad, good_rect_log, cent_cords)

    # correcting for orientation
    rect_mask = np.flipud(rect_mask)
    rect_mask = np.rot90(rect_mask, 3)

    # loading array into image and saving
    rect_mask_img = Image.fromarray(np.uint8(rect_mask * 200))
    rect_mask_img.save('./images/rect_mask.png' )
```

```
    img_array_perm = smart_img_clean(rect_mask, img_array, x_pix_max, y_pix_max)

    # loading array into image and saving
    smart_clean_img = Image.fromarray(np.uint8(img_array_perm))
    smart_clean_img.save('./images/smart_clean.png')

    # default cell radius (mm)
    cell_real_size =  0.03

    cell_real_size_city_block = 0.08

    sweep_num_max = 8

    # setting x and y offset iteration parameters (mm)
    x_r_off_min = 10.5
    x_r_off_max = 13

    y_r_off_min = 1.5
    y_r_off_max = 4

    # loading sweep pattern
    sweep_data = sweep_pattern(sweep_num_max)

    for sweep_num in xrange(1, sweep_num_max):
        sweep_type = 0
        print ('sweep number = %d') % sweep_num
        cell_size = cell_real_size

        if (sweep_num == 5):
            print ('sweep type of 1')
            sweep_type = 1
            cell_size = cell_real_size_city_block

        print (x_r_off_min, x_r_off_max, y_r_off_min, y_r_off_max, \
            sweep_data[sweep_num - 1][1], sweep_data[sweep_num - 1][2])

        img_array = img_array_perm

        img_array, pix_scale, x_real_offset, y_real_offset, theta = \
            sweep(filename, x_r_off_min, x_r_off_max, y_r_off_min, y_r_off_max, \
            sweep_data[sweep_num - 1][1], sweep_data[sweep_num -1][2], cell_real_size, \
            X, Y, Z, cent_cords, good_rect_log, img_array, rect_mask,
            pix_scale, theta, std_dev_map, sweep_type)

        print ('after sweep of sweep_num %d of sweep_type %d') % (sweep_num, sweep_type)

        if (sweep_num <= sweep_num_max - 2):
            x_r_off_min = x_real_offset - sweep_data[sweep_num][0] * sweep_data[sweep_num][1]
            x_r_off_max = x_real_offset + sweep_data[sweep_num][0] * sweep_data[sweep_num][1]

            y_r_off_min = y_real_offset - sweep_data[sweep_num][0] * sweep_data[sweep_num][2]
            y_r_off_max = y_real_offset + sweep_data[sweep_num][0] * sweep_data[sweep_num][2]



    fit_params = (img_array_perm, cell_real_size, x_real_offset, y_real_offset,\
        pix_scale, theta, X, Y, Z)

    i_list = read_out(img_array_perm, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z)

    np.savetxt(filename_out, i_list)


    print ('i_list.txt saved')

    return img_array, pix_scale, x_real_offset, y_real_offset, theta


@jit
def sweep_pattern(sweep_num_max):

    # translational offset iteration step (mm), 0.125 is spacing between cells
    real_trans_step_1 = 0.25

    # size of search area in second sweep
```

```
    steps_2 = 5
    real_trans_step_2 = 0.0125

    # size of search area in third sweep
    steps_3 = 6
    real_trans_step_3 = 0.125

    # size of search area in fourth sweep
    steps_4 = 6
    real_trans_step_4 = 0.125

    # size of search area in fifth sweep by city block
    steps_5 = 3
    real_trans_step_x_5 = 2.2
    real_trans_step_y_5 = 2.5

    # size of search area in sixth sweep
    steps_6 = 5
    real_trans_step_x_6 = 0.125
    real_trans_step_y_6 = 0.125

    # size of search area in seventh sweep
    steps_7 = 5
    real_trans_step_x_7 = 0.0125
    real_trans_step_y_7 = 0.0125

    # stores the steps, real_trans_step_x, real_trans_step_y
    sweep_data = np.zeros((sweep_num_max - 1, 3))

    sweep_data = [[0, real_trans_step_1, real_trans_step_1],[steps_2, \
        real_trans_step_2, real_trans_step_2], [steps_3, real_trans_step_3, \
        real_trans_step_3],[steps_4, real_trans_step_4, \
        real_trans_step_4],[steps_5, real_trans_step_x_5, real_trans_step_y_5], \
        [steps_6, real_trans_step_x_6, real_trans_step_y_6], \
        [steps_7, real_trans_step_x_7, real_trans_step_y_7]]

    return sweep_data


# !!!!!!!!!!!!!! sometimes segfaults!
# reads out summed values for for each cell
def read_out (img_array, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

    # putting image array into correct frame
    img_array = np.rot90(img_array, 1) # maybe
    img_array = np.flipud(img_array)

    i_list = read_out_crunch(img_array, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z)

    return i_list


#@jit(nopython=True)
def read_out_crunch (img_array, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

    # final cell index
    i_max = 11664
    i_half = 5832

    x_pix_max = 1292
    y_pix_max = 964

    # [cell number][total intensity, number of pixels for cell]
    i_list = np.zeros((i_max, 2))

    cell_pix_size_float = cell_real_size * pix_scale # width of cell in pixels
    cell_pix_size_int = int(math.ceil(cell_pix_size_float))

    X_pix = np.zeros(i_max)
    Y_pix = np.zeros(i_max)

    # location of cells in terms of image pixels
    for i in xrange(0,11664):
```

```
        X_pix[i] = round((X[i] + x_real_offset) * pix_scale)
        Y_pix[i] = round((Y[i] + y_real_offset) * pix_scale)

    x_mid = X_pix[i_half]
    y_mid = Y_pix[i_half]

    rot_matrix = np.zeros((2, 2))
    rot_matrix[0][0] = math.cos(theta)
    rot_matrix[1][1] = math.cos(theta)
    rot_matrix[1][0] = - math.sin(theta)
    rot_matrix[0][1] = math.sin(theta)

    # accounting for x, y shift of the middle of mask so rotation only takes
    # place around centre of chip
    x, y = rotate(x_mid, y_mid, rot_matrix)
    delt_x = x - x_mid
    delt_y = y - y_mid

    for i in xrange(0, i_max):
        for r_x in xrange(-cell_pix_size_int, cell_pix_size_int):
            for r_y in xrange(-cell_pix_size_int, cell_pix_size_int):

                if (math.sqrt(r_x ** 2 + r_y ** 2) <= cell_pix_size_float):

                    x = int(round(X_pix[i] + r_x)) # temporary x
                    y = int(round(Y_pix[i] + r_y)) # temporary y

                    x, y = rotate(x, y, rot_matrix)

                    # correcting for non centred rotation
                    x = int(round(x - delt_x))
                    y = int(round(y - delt_y))

                    if (x >= 0 and x < x_pix_max and y >= 0 and y < y_pix_max):
                            i_list[int(i)][0] = i_list[int(i)][0] + img_array[int(x)][int(y)]
                            i_list[int(i)][1] += 1

                    else :
                         b = 2 # placeholder
    return i_list
```