```
import numpy as np
import cv2
from PIL import Image
import math
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import random
from numba import jit

@jit#(nopython=True)
def find_neighbours (cent_cords, count):

        # calculating the distance from every good rect centre to every other
        # rect centre in x, y and vector magnitude
        p_to_p_dist = np.zeros((count, count, 3))

        # recording the closest to fourth closest neighbours delt_x and delt_y
        # and vector mag values
        closest_neighbours = np.zeros((count, 4, 3))

        closest_neighbours = closest_neighbours + 3000

        # looping over start points
        for s_p in xrange(0, count):
                # looping over end points
                for e_p in xrange(0, count):

                        if (s_p != e_p):
                                p_to_p_dist[s_p][e_p][0] = cent_cords[s_p][0] - cent_cords[e_p][0]
                                p_to_p_dist[s_p][e_p][1] = cent_cords[s_p][1] - cent_cords[e_p][1]
                                p_to_p_dist[s_p][e_p][2] = math.sqrt((p_to_p_dist[s_p][e_p][0] ** 2) \
                                         + (p_to_p_dist[s_p][e_p][1] ** 2))

                                if (closest_neighbours[s_p][3][2] > p_to_p_dist[s_p][e_p][2]):

                                        if (closest_neighbours[s_p][0][2] > p_to_p_dist[s_p][e_p][2]):
                                                closest_neighbours[s_p][0][0] = p_to_p_dist[s_p][e_p][0]
                                                closest_neighbours[s_p][0][1] = p_to_p_dist[s_p][e_p][1]
                                                closest_neighbours[s_p][0][2] = p_to_p_dist[s_p][e_p][2]

                                        elif (closest_neighbours[s_p][1][2] > p_to_p_dist[s_p][e_p][2]):
                                                closest_neighbours[s_p][1][0] = p_to_p_dist[s_p][e_p][0]
                                                closest_neighbours[s_p][1][1] = p_to_p_dist[s_p][e_p][1]
                                                closest_neighbours[s_p][1][2] = p_to_p_dist[s_p][e_p][2]

                                        elif (closest_neighbours[s_p][2][2] > p_to_p_dist[s_p][e_p][2]):
                                                closest_neighbours[s_p][2][0] = p_to_p_dist[s_p][e_p][0]
                                                closest_neighbours[s_p][2][1] = p_to_p_dist[s_p][e_p][1]
                                                closest_neighbours[s_p][2][2] = p_to_p_dist[s_p][e_p][2]
                                        else :
                                                closest_neighbours[s_p][3][0] = p_to_p_dist[s_p][e_p][0]
                                                closest_neighbours[s_p][3][1] = p_to_p_dist[s_p][e_p][1]
                                                closest_neighbours[s_p][3][2] = p_to_p_dist[s_p][e_p][2]

        return closest_neighbours


def output (pix_scale_omni, angle_omni, hor_neigh_count, \
         ver_neigh_count):
        pix_scale_std_err = np.std(pix_scale_omni) / math.sqrt(pix_scale_omni.size)

        pix_scale_mean = np.mean(pix_scale_omni)

        angle_std_err = np.std(angle_omni) / math.sqrt(angle_omni.size)

        angle_mean = np.mean(angle_omni)

        return angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err

def error_minimize (filename, mode=0):
        # iterates over values of C to find the best angle and pix_scale disagreement
        max_C = 15
        length = 2 * max_C
        min_good_rects = 10

        # angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords
        # good_rect_log
```

```
        data_out = np.zeros((5, length))

        for C in xrange(-max_C, max_C):
                data_out[0][C + max_C], data_out[1][C + max_C], data_out[2][C + max_C], \
                        data_out[3][C + max_C], data_out[4][C + max_C], cent_cords, good_rect_log = main(file
name, C)

                if (np.count_nonzero(good_rect_log) < min_good_rects):
                        data_out[1][C + max_C] = float('nan' )
                        data_out[3][C + max_C] = float('nan' )

        angle_std_err = data_out[1,:]
        pix_scale_std_err = data_out[3,:]
        rects = data_out[4,:]

        ang_weight = 1 / np.nanmean(angle_std_err)
        pix_weight = 1 / np.nanmean(pix_scale_std_err)

        avrg_err = (angle_std_err * ang_weight + pix_scale_std_err * pix_weight) / \
                (ang_weight + pix_weight)

        min_angle_err_C = np.nanargmin(angle_std_err) - max_C
        min_pix_err_C = np.nanargmin(pix_scale_std_err) - max_C
        min_avrg_err_c = np.nanargmin(avrg_err) - max_C
        max_rects_C = np.nanargmax(rects) - max_C

        if (mode == 0):
                C = min_avrg_err_c
        elif (mode == 1):
                C = max_rects_C
        else :
                print ('error, incorrect mode, must be either 0 or 1'          )
                return 0

        data = data_out[:,C + max_C]

        data_out[0][C + max_C], data_out[1][C + max_C], data_out[2][C + max_C], \
                data_out[3][C + max_C], data_out[4][C + max_C], cent_cords, good_rect_log = main(filename, C)


        print ('best C = '   )
        print (min_avrg_err_c)

        return data, cent_cords, good_rect_log


# when flag == '-p' image threshold and fitted rectangles will be displayed
def main(filename, C, flag='-n' ):

        # thresholding parameters, iterate over C perhaps, strong effect on angle disagreement
        #C = -10 # background subtraction when thresholding, 2 reccomended value
        kern_dims = 21 # size of gaussian blurring kernal, default 21
        adapt_box_size = 40 # size of adaptive threshold box, good value of 40

        # rectangle sifting parameters
        min_area = 2500.0 # 2500 - 2700 default
        max_area = 3700.0 # 3700 default
        max_side_ratio = 1.1 # max accepted value of longer side divided by shorter side

        # spacing of city blocks in mm
        horizontal_spacing = 2.2
        vertical_spacing = 2.5

        # creating random colours for circles representing rectangle points
        R = random.random()
        G = random.random()
        B = random.random()

        #print('R = %f, G = %f, B = %f') % (R, G, B)

        input_filename = filename

        pil_image = Image.open(input_filename).convert('RGB')

        open_cv_image = np.array(pil_image)

        gray = cv2.cvtColor(open_cv_image,cv2.COLOR_BGR2GRAY)
```

```
        # gaussian blur input image
        gray = cv2.GaussianBlur(gray, (kern_dims, kern_dims), 0)

        if ((adapt_box_size % 2) == 0):
                adapt_box_size += 1

        # performs adaptive thresholding of image, C = 3 by default
        thresh = cv2.adaptiveThreshold(gray.astype(np.uint8), 255, \
                cv2.ADAPTIVE_THRESH_MEAN_C,  cv2.THRESH_BINARY, adapt_box_size, C)

        # finds OTSU binarisation of image
        #ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

        #thresh = np.invert(thresh)

        # saving hard copy of thresh to avoid damage from findcontours
        thresh_perm = np.copy(thresh)

        # find array of contours
        (cnts, _) = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

        shape = np.shape(cnts)

        length = int(shape[0])

        areas = np.zeros(length)
        side_lengths = np.zeros((length, 4))

        cent_cords = [0, 0]

        #cent_cords = np.zeros((length, 2))
        side_ratio = np.zeros(length)

        # reduced array containing only coordinates of

        count = 0

        # cycle over contours
        for  i  in  xrange(0,length):

                rect = cv2.minAreaRect(cnts[i])
                box = cv2.cv.BoxPoints(rect)
                box = np.int0(box)

                for  side  in  xrange(0,4):
                        if (side == 3):
                                side_plus = 0
                        else :
                                side_plus = side + 1

                        side_lengths[i][side] = math.sqrt((box[side][0] - box[side_plus][0]) \
                                ** 2 + (box[side][1] - box[side_plus][1]) ** 2)

                areas[i] = side_lengths[i][0] * side_lengths[i][1]

                # ensure no zero area rectangles are further processed
                if ((side_lengths[i][0] != 0) & (side_lengths[i][1] != 0)):

                        side_ratio[i] = side_lengths[i][0] / side_lengths[i][1]

                        if (side_ratio[i] < 1):
                                side_ratio[i] = 1.0 / side_ratio[i]

                        if (areas[i] > 5000):
                                areas[i] = 0

                        if ((areas[i] > min_area) & (areas[i] < max_area) & (side_ratio[i] \
                                < max_side_ratio)):

                                count += 1

                                delt_x_1 = box[0][0] - box[1][0]
                                delt_y_1 = box[0][1] - box[1][1]

                                delt_x_2 = box[1][0] - box[2][0]
                                delt_y_2 = box[1][1] - box[2][1]
```

```
                        delt_x = (delt_x_1 / 2) + (delt_x_2 / 2)
                        delt_y = (delt_y_1 / 2) + (delt_y_2 / 2)

                        cent_cords_x = box[0][0] - delt_x
                        cent_cords_y = box[0][1] - delt_y

                        cent_cords_vec = [cent_cords_x, cent_cords_y]

                        cent_cords = np.vstack((cent_cords, cent_cords_vec))

                        cv2.circle(open_cv_image, (int(cent_cords[count][0]), \
                                int(cent_cords[count][1])), 40, (B * 255, G * 255, R * 255), 2)

                        #cv2.drawContours(open_cv_image,[box],0,(0,0,255),2)

        closest_neighbours = find_neighbours(cent_cords, count)

        # number of horizontal neighbours
        hor_neigh_count = 0

        # number of vertical neighbours
        ver_neigh_count = 0

        # attributes of horizontal neighbours
        hor_neigh = []

        # attributes of vertical neighbours
        ver_neigh = []

        # recording the pix scale implied by the horizontal and vertical spacing of city blocks
        pix_scale_hor = []
        pix_scale_ver = []

        # recording the angles of connections between neighbouring points
        angle_hor = []
        angle_ver = []

        good_rect_log = np.zeros(count)

        # iterating over starting point
        for  s_p  in  xrange(1,count):
                # iterating over both first to fourth closest neighbour
                for  neighbour  in  xrange(0,4):
                        if ((closest_neighbours[s_p,neighbour,2] > 75) & \
                                (closest_neighbours[s_p,neighbour,2] < 84)):

                                good_rect_log[s_p] = 1

                                pix_scale_hor.append(closest_neighbours[s_p,neighbour,2] \
                                        / horizontal_spacing)

                                angle = math.atan(closest_neighbours[s_p,neighbour,1] \
                                        / closest_neighbours[s_p,neighbour,0])

                                angle = math.degrees(angle)
                                angle_hor.append(angle)

                                hor_neigh_count += 1

                        if ((closest_neighbours[s_p,neighbour,2] > 88) & \
                                (closest_neighbours[s_p,neighbour,2] < 94)):

                                good_rect_log[s_p] = 1

                                pix_scale_ver.append(closest_neighbours[s_p,neighbour,2] / \
                                 vertical_spacing)

                                angle = math.atan(closest_neighbours[s_p,neighbour,0] / \
                                        closest_neighbours[s_p,neighbour,1])

                                angle = math.degrees(angle)
                                angle_ver.append(-angle)

                                ver_neigh_count += 1
```

```
        pix_scale_omni = np.asarray(pix_scale_hor + pix_scale_ver)
        angle_omni = np.asarray(angle_hor + angle_ver)

        angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err = \
                output(pix_scale_omni, angle_omni, \
                hor_neigh_count, ver_neigh_count)

        # if -p is given as a flag output is printed
        if (flag == '-p' ):

                print (cent_cords)

                print ('%d rectangles drawn'    ) % count

                #filename = ('./vision/adaptive_test_kern_%d.png') % kern_dims

                #image = Image.fromarray(open_cv_image)
                #image.save(filename)
                #print('image saved to disk')

                cv2.imshow('city block centres'     , thresh_perm)
                #cv2.imshow('city block centres', open_cv_image)

                # saving threshold binarisation
                image_out = Image.fromarray(np.uint8(thresh_perm))
                image_out.save('./images/adapt_thresh.png'     )

                # saving fitted rectangles
                image_out = Image.fromarray(np.uint8(open_cv_image))
                image_out.save('./images/rect_fit.png'      )

                if cv2.waitKey(0) & 0xff == 27:
                    cv2.destroyAllWindows()

        # angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords
        # good_rect_log
        #data_out = [angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords, good_re
ct_log]

        return  angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords, good_rect_log
```