Abdullah Samarkandi

We first used gaussian kernel

```
gaussian_kernel = (1.0/57) * np.array(
        [[0, 1, 2, 1, 0],
        [1, 3, 5, 3, 1],
        [2, 5, 9, 5, 2],
        [1, 3, 5, 3, 1],
```

with gradient algorithm, then soblel for x and y.

```
kernelx = np.array([[-1, 0, 1],
           [-2, 0, 2],
           [-1, 0, 1]])
kernely = np.array([[-1, -2, -1],
           [0, 0, 0],
           [1, 2, 1]])
```

Then hessian =Ixx * Iyy - Ixy ** 2 and used threshold of 100000

Then non maximum suppression

```
    if(op_im[i][j] != max(op_im[i-1][j-1], op_im[i-1][j], op_im[i-1][j+1], op_im[i][j-1],
op_im[i][j], op_im[i][j+1], op_im[i+1][j-1], op_im[i+1][j], op_im[i+1][j+1])):
                    E_nms[i-1][j-1] = 0
                else:
                    E_nms[i][j] = op_im[i][j]
```

Then ransac:
Pic two random numbers from the matrix of points we detected, then use them to solve the equation of the line to get m and c. Then we find the distance between the line and all other edges we detected, then keep picking these two random points depend on the iteration we pick. Best line in plotted.

the code,

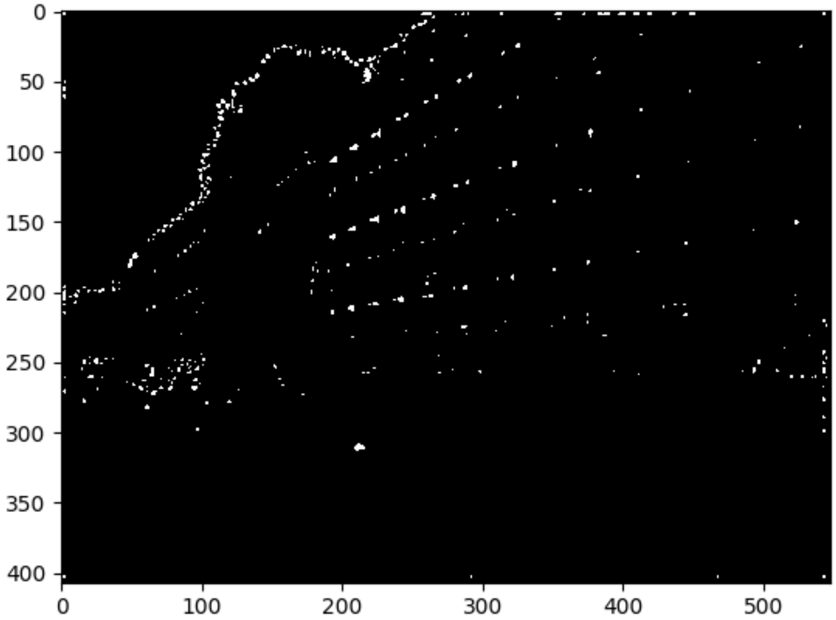
```
import numpy as np
import cv2
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import math
import PIL
from PIL import Image, ImageDraw
from r import *
import sys
#read image
input_image = cv2.imread("road.png", cv2.IMREAD_GRAYSCALE)

#image hight, width
height = input_image.shape[0]
width  = input_image.shape[1]

print ("height:",height, "width:", width)

gaussian_kernel = (1.0/57) * np.array(
        [[0, 1, 2, 1, 0],
        [1, 3, 5, 3, 1],
        [2, 5, 9, 5, 2],
        [1, 3, 5, 3, 1],
        [0, 1, 2, 1, 0]])

# Sobel kernels
kernelx = np.array([[-1, 0, 1],
           [-2, 0, 2],
           [-1, 0, 1]])
kernely = np.array([[-1, -2, -1],
           [0, 0, 0],
           [1, 2, 1]])

#function for kernels to use
def fun(img, kernel):
    img_out = np.zeros((img.shape[0], img.shape[1]), dtype=np.float32)
    offset = len(kernel) // 2
    for i in np.arange(offset, height-offset):
        for j in np.arange(offset, width-offset):
            sum = 0
            for k in np.arange(-offset, offset+1):
                for l in np.arange(-offset, offset+1):
                    a = img.item(i+k, j+l)
                    p = kernel[offset+k, offset+l]
                    sum = sum + (p *  a)
                    sum
            b = sum
            img_out.itemset((i,j), b)   #Insert scalar into an array (scalar is cast to array’s dtype,
                                        #if possible)
    return img_out

# gaussian process
gaussian =fun(input_image, gaussian_kernel)
```

```
Ix      =fun (gaussian, kernelx)
Ixx     = fun (Ix, kernelx)

Iy      =fun (gaussian, kernely)
Iyy     = fun (Iy, kernely)

Ixy     = fun (Ix,kernely)

hessian =Ixx * Iyy - Ixy ** 2

#threshold for gaussian
def threshold(hessian):
    for i in np.arange(0, height):
        for j in np.arange(0, width):
            a = hessian.item(i, j)
            if a > 100000:
                hessian[i][j]=255
            else:
                hessian[i][j]=0
    return hessian

hes=threshold(hessian)

#non maximum  that return a picture
def fun2(op_im):
    E_nms = np.zeros((height, width))
    for i in range(1, height-1):
            for j in range(1, width-1):
                if(op_im[i][j] != max(op_im[i-1][j-1], op_im[i-1][j], op_im[i-1][j+1], op_im[i][j-1],
op_im[i][j], op_im[i][j+1], op_im[i+1][j-1], op_im[i+1][j], op_im[i+1][j+1])):
                    E_nms[i-1][j-1] = 0
                else:
                    E_nms[i-1][j-1] = op_im[i][j]

    E_nms = E_nms.astype(np.uint8)
    f_im = Image.fromarray(E_nms)

    return f_im

maxi1 = fun2(hes)

#non maximum that return a matrix
def ran(op_im):
    count = 0
    E_nms = np.zeros((height, width))
    for i in range(1, height-1):
            for j in range(1, width-1):
                if(op_im[i][j] != max(op_im[i-1][j-1], op_im[i-1][j], op_im[i-1][j+1], op_im[i][j-1],
op_im[i][j], op_im[i][j+1], op_im[i+1][j-1], op_im[i+1][j], op_im[i+1][j+1])):
                    E_nms[i-1][j-1] = 0
                else:
                    E_nms[i-1][j-1] = op_im[i][j]
                    # count = count +1
                    # print("E_nms[i-1][j-1]",E_nms[i-1][j-1])

    E_nms = E_nms.astype(np.uint8)
    f_im = Image.fromarray(E_nms)

    return E_nms

maxi = ran(hes)

plt.imshow(maxi1)
plt.show()


#chose two random points from the maximum to use for ransac
def randomP(rans):
    line = []
    while True :
        x=np.random.randint(0, height)
        y=np.random.randint(0, width)

        if rans[x][y] == 255:
            if len(line) == 1:
                if not x == line[0][0] and not y == line[0][1]:
                    line.append((x,y))
            else:
                line.append((x,y))
        if len(line) == 2:
            break
    return line


h=maxi.shape[0]
w=maxi.shape[1]
print ("h,w",h,w)

# make line with two random points from the edge points
def find_line_model(points):
    """ find a line model for the given points
    :param points selected points for model fitting
    :return line model
    """
    # find a line model for these points
    m = (points[1][1] - points[0][1]) / (points[1][0] - points[0][0] + sys.float_info.epsilon)    #
slope (gradient) of the line
    c = points[1][1] - m * points[1][0]                                        # y-intercept of the line
    # print("Randommmmmmmm",points[1][1],points[0][1],points[1][1],points[1][0])
    return m, c
print(randomP(maxi))

print("m,c",find_line_model(randomP(maxi)))

#the distance between line and other points
def find_intercept_point(m, c, x0, y0):
    """ find an intercept point of the line model with
        a normal from point (x0,y0) to it
    :param m slope of the line model
    :param c y-intercept of the line model
    :param x0 point's x coordinate
    :param y0 point's y coordinate
    :return intercept point
    """

    # intersection point with the model
    x = (x0 + m*y0 - m*c)/(1 + m**2)
    y = (m*x0 + (m**2)*y0 - (m**2)*c)/(1 + m**2) + c
    return x, y

#ploting ransac
def ransac_plot(img,n, x, y, m, c, final=False, x_in=(), y_in=(), points=()):
    """ plot the current RANSAC step
    :param n      iteration
    :param points picked up points for modeling
    :param x      samples x
    :param y      samples y
    :param m      slope of the line model
    :param c      shift of the line model
    :param x_in   inliers x
    :param y_in   inliers y
    """

    xx= np.reshape(x,(-1,2))
    yy= np.reshape(y,(-1,2))

    f_im = Image.fromarray(img)

    # height, width= img.shape
    # height=img.shape

    implot = plt.imshow(f_im)
    # plt.scatter(xx[:,0], yy[:,0])
    # plt.plot(yy[:,0], xx[:,0], markersize=12, label='Input points', color='#00cc00',
linestyle='None', alpha=0.4)

    line_width = 1.
    line_color = '#0080ff'
    title = 'iteration ' + str(n)

    if final:
        fname = "output/final.png"
        line_width = 1.
        line_color = '#ff0000'
        title = 'final solution'

    # plt.figure("Ransac", figsize=(width, height))
    # plt.figure("Ransac")

    # grid for the plot

    grid = [np.min(xx) , np.max(xx) , np.min(yy) , np.max(yy)]
    plt.axis("tight")

    # put grid on the plot
    plt.grid(which='minor', color='0.1', linestyle='-')
    plt.xticks([i for i in range(np.min(yy) , np.max(yy) , 5)])
    plt.yticks([i for i in range(np.min(xx) , np.max(xx) , 10)])

    # plot input points

    plt.plot(yy[:,0], xx[:,0], marker='o', label='Input points', color='#00cc00', linestyle='None',
alpha=0.4)

    # draw the current model

    plt.plot(xx, m*xx + c, 'r', label='Line model', color=line_color, linewidth=0.2)

    # draw inliers
    if not final:
        plt.plot(x_in, y_in, marker='o', label='Inliers', linestyle='None', color='#ff0000', alpha=0.6)

    # # draw points picked up for the modeling
    #  plt.plot(points[:,0], points[:,1]
    if not final:
        plt.plot(points[0][0], points[0][1], marker='o', label='Picked points', color='#0000cc',
linestyle='None', alpha=0.6)
        plt.plot(points[1][0], points[1][1], marker='o', label='Picked points', color='#0000cc',
linestyle='None', alpha=0.6)

    plt.show()
    plt.title(title)
    plt.legend()
    plt.close()

ratio = 0.
model_m = 0.
model_c = 0.

# Ransac parameters
ransac_iterations = 20  # number of iterations
ransac_threshold = 12    # threshold
ransac_ratio = 0.3      # ratio of inliers required to assert
                        # that a model fits well to data
n_samples = np.count_nonzero(maxi)
print ("n_samples",n_samples)

x_list1=[]
y_list1=[]
numm = 0
for i in np.arange(width-2):
    for j in np.arange(height-2):
        a= maxi[j][i]
        if a == 255:
            x0=j
            y0=i
            # print ("x0,y0",maxi.item(j,i))
            # find an intercept point of the model with a normal from point (x0,y0)
            #distance from point to the model
            # print("dist",dist)
            # check whether it's an inlier or not
            x_list1.append(x0)
            y_list1.append(y0)
            numm += 1
x_all = np.array(x_list1)
y_all = np.array(y_list1)


# perform RANSAC iterations ransac_iterations
for it in range(ransac_iterations):

    # pick up two random points
    n = 2

    # find a line model for these points
    maybe_points = randomP(maxi)
    m, c = find_line_model(maybe_points)

    x_list = []
    y_list = []
    num = 0

    # find orthogonal lines to the model for all testing points
    for j in np.arange(height-3):
        for i in np.arange(width-3):
            a= maxi[j][i]
            if a == 255:
                x0=j
                y0=i
                # print ("x0,y0",maxi.item(j,i))
                # find an intercept point of the model with a normal from point (x0,y0)
                x1, y1 = find_intercept_point(m, c, x0, y0)
                # print ("x1,y1",x1,y1)
                #distance from point to the model
                dist = np.sqrt((x1 - x0)**2 + (y1 - y0)**2)
                # print("dist",dist)
                # check whether it's an inlier or not
                if dist < ransac_threshold:
                    x_list.append(x0)
                    y_list.append(y0)
                    num += 1

    x_inliers = np.array(x_list)
    y_inliers = np.array(y_list)
    # print ("x_inliers", x_inliers)
    # print ("y_inliers", y_inliers)

    # in case a new model is better - cache it
    if num/float(n_samples) > ratio:
        ratio = num/float(n_samples)
        model_m = m
        model_c = c

    print ('  inlier ratio = ', num/float(n_samples))
    print ('  model_m = ', model_m)
    print ('  model_c = ', model_c)

    # plot the current step
    ransac_plot(input_image, it, x_all, y_all, m, c, False, x_inliers, y_inliers, maybe_points)

    # we are done in case we have enough inliers
    if num > n_samples*ransac_ratio:
        print ('The model is found !')
        break

# plot the final model
# for i in 4
```

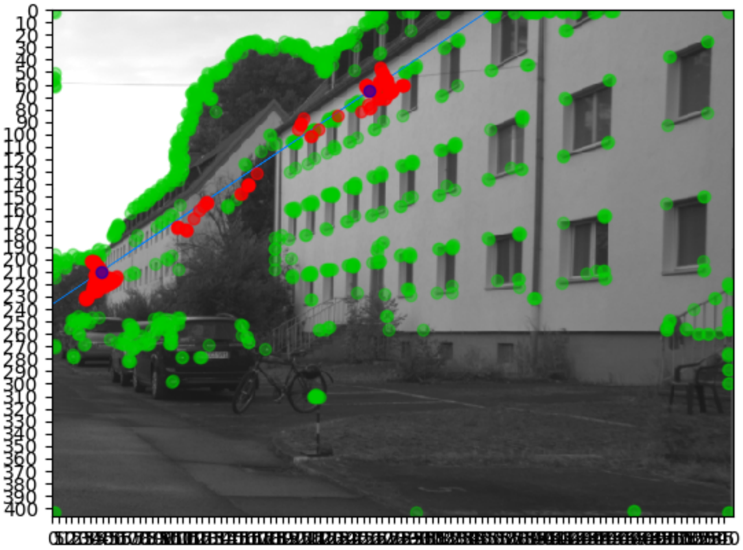
```
ransac_plot(input_image, 0, x_all,y_all, model_m, model_c, True)

print ('\nFinal model:\n')
print ('  ratio = ', ratio)
print ('  model_m = ', model_m)
print ('  model_c = ', model_c)

cv2.imshow('gaussian',gaussian)
cv2.imshow('Ix',Ix)
cv2.imshow('Ixx',Ixx)
cv2.imshow('Iy',Iy)
cv2.imshow('Ixx',Iyy)s
cv2.imshow('Ixy',Ixy)
cv2.imshow('hessian',hes)
cv2.imshow('maxi', maxi)


k=0
cv2.waitKey(0)
cv2.destroyAllWindows()
if k == 27:         # wait for ESC key to exit
    cv2.destroyAllWindows()

#plt.imshow(gaussian)
# plt.imshow(Ix)
# plt.imshow(Ixx)
# plt.imshow(Iy)
# plt.imshow(Iyy)
# plt.imshow(Ixy)
# plt.imshow(hes)
# plt.imshow(maxi)
# plt.show()
```

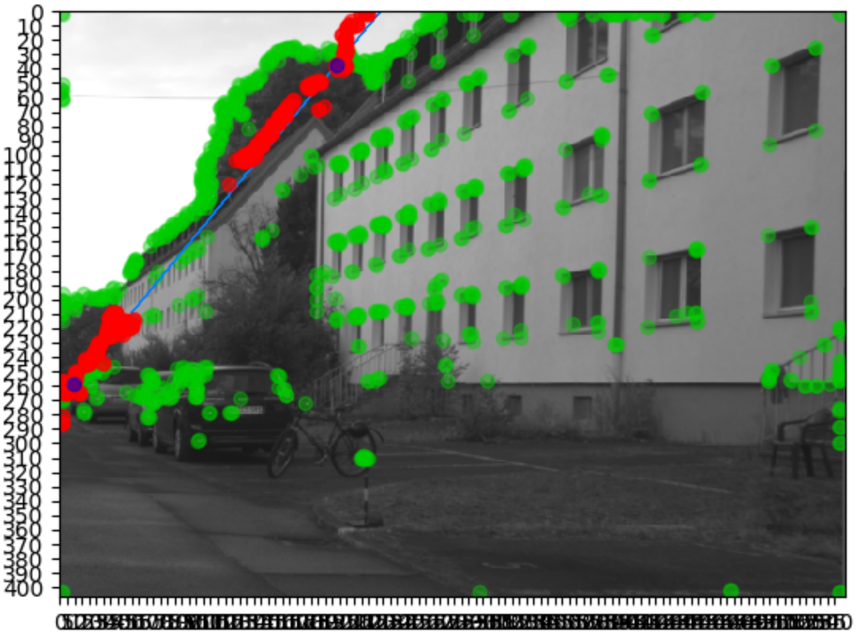
3) the output images.

Edge detection results

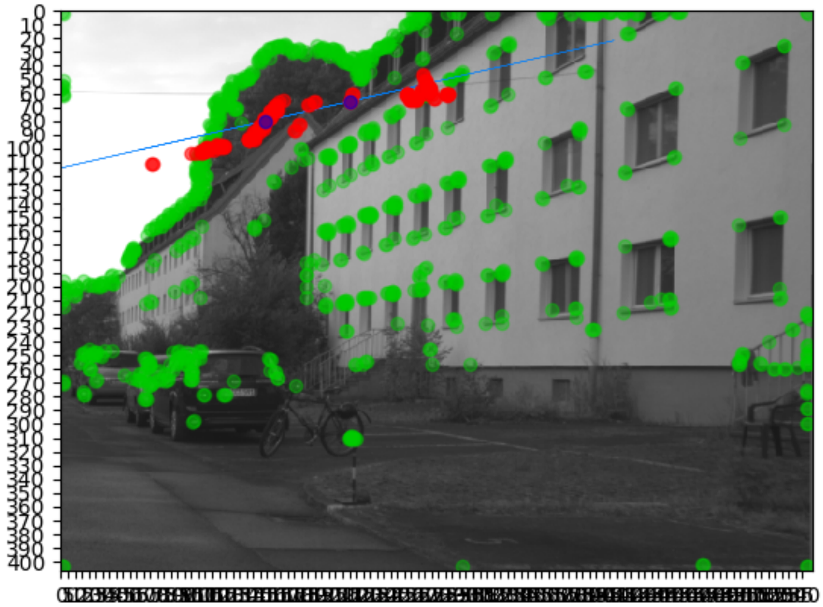


Ransac Result





0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
210
220
230
240
250
260
270
280
290
300
310
320
330
340
350
360
370
380
390
400



0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
210
220
230
240
250
260
270
280
290
300
310
320
330
340
350
360
370
380
390
400