Assignment 2

Importing Libraries

In [167]:

```
import cv2
import numpy as np
import time
import math as m
import random
```

Gaussian function for getting gaussian values for a particular (x,y) point

In [168]:

```
def evaluator(x, y, sigma):
    return (1/(2 * (m.pi) * ((sigma) ** 2))*(1/(m.exp(((x**2) + (y**2))/(2
* (sigma ** 2))))))

x = np.array([[1, 2, 1],
              [0, 0, 0],
              [-1, -2, -1]])
x = x - 1
print(x)
```

```
[[ 0  1  0]
 [-1 -1 -1]
 [-2 -3 -2]]
```

Function to get the gaussian filter

In [169]:

```
def gauss_filter(size, sigma):

    gs_filter = np.zeros((size,size))

    for x in range(size):
        for y in range(size):
            gs_filter[x,y] = evaluator(x - (size//2),y - (size//2),sigma)

    return gs_filter
```

Function to apply a filter to an image

In [170]:

```
def filter_the_img(image,filt,size):

    image_row,image_col = image.shape # rows and columns in our image
    res = np.zeros(image.shape) # we'll save results in this array

    val = size//2 # limit so that these values aren't affected in the outpu
t array

    for row in range(image_row):
        for col in range(image_col):
            if(row < val or row > image_row - val - 1 or col < val or col > 
image_col - val - 1):
                res[row,col] = image[row,col] # cant apply filter to these
 values
            else:
                res[row, col] = np.sum(filt * image[row-val:row+val+1, col-
val:col+val+1])
    return res
```

Function to get horizontally and vertically sobel filtered images

In [171]:

```
def sobel_h(image):
    x = np.array([[1, 2, 1],
                  [0, 0, 0],
                  [-1, -2, -1]])
    res = filter_the_img(image,x,3)
    return res

def sobel_v(image):
    y = np.array([[1, 0, -1],
                  [2, 0, -2],
                  [1, 0, -1]])
    res = filter_the_img(image,y,3)
    return res
```

Function that applies non-max suppression to given input

In [172]:

```
def nms(image):
    res = image
    r,c = image.shape

    for i in range(1, r - 1):
        for j in range(1, c - 1):
            #check if current element is the maximum of surrounding element
s, if no, assign it to 0 else, set neighbours to 0
            if image[i,j] != max(image[i-1,j-1],image[i-1,j],image[i-1,j+1]
#first row elements
                                ,image[i,j-1],image[i,j],image[i,j+1] #seco
nd row elements
                                ,image[i+1,j-1],image[i+1,j],image[i+1,j+1
]): #thrid row elements
                res[i,j] = 0
            else:
                #setting all neighbouring elements to zero
                temp = image[i,j]
                res[i-1:i+2,j-1:j+2] = 0
                res[i][j] = temp
    return res
```

Function to calculating Hessian and thresholding it after that

In [173]:

```
def hes(image):

    r,c = image.shape
    res = np.zeros((r,c)) #result array


    I_xx = sobel_h(sobel_h(image)) #finding Ixx
    I_xy = sobel_h(sobel_v(image)) #finding Ixy
    I_yy = sobel_v(sobel_v(image)) #finding Iyy


    #getting the determinant of hessian matrix
    h_det = np.zeros(image.shape)
    for i in range(r):
        for j in range(c):
            h_det[i,j] = I_xx[i,j]*I_yy[i,j] - I_xy[i,j]*I_xy[i,j]

    #finding min and max values for normalizing the image
    #new min = 0, new max = 255
    mini = np.min(h_det)
    maxi = np.max(h_det)
    new_range = maxi - mini

    #noramlizing the image
    h_det = h_det - mini
    h_det = h_det * 255
    h_det = h_det / new_range

    #thresholding the determinant
    thresh = 140
    for i in range(r):
        for j in range(c):
            #if value < threshold, set it 0, else set it to max
            if h_det[i][j] < thresh:
                h_det[i][j] = 0
            else:
                h_det[i][j] = 255

    #applying non max supression to the normalized and thresholded hessian
 determinant
    res = nms(h_det)

    return res
```

Function that returns all elements that pass the threshold

In [174]:

```
def get_valid_pixels(image, threshold):
    #(x,y) pairs are stores as tuples
    pixels = []

    #for every pixel check if it is above the threshold
    for p in range(image.shape[0]):
        for q in range(image.shape[1]):
            if image[p,q] > threshold:
                pixels.append((q, p))

    return pixels
```

Function that returns two random elements from a list

In [175]:

```
def get_random(pixel_list):
    first,second = 0,0

    #while loop to ensure that both points are different
    while first == second:
        first = random.randint(0,len(pixel_list) - 1)
        second = random.randint(0,len(pixel_list) - 1)

    return pixel_list[first], pixel_list[second]
```

Function that returns slope and intercept of line passing through two points

In [176]:

```
def get_m_and_c(p1, p2):
    #getting x and y coordinates
    x1,y1 = p1
    x2,y2 = p2

    #if the line is vertical
    if(x1 == x2):
        slope = m.inf
    else:
        slope = (y2 - y1)/(x2 - x1)

    #y = slope*x + c satisfies p1, so c = -slope*x1 + y1
    c = -slope*x1 + y1
    return slope, c
```

Function that finds the perpendicular distance of a point from a line

In [177]:

```
def get_min_dist(slope,c,x,y):
    # min distance of (x1,y1) from line ax + by + c = 0 is abs(ax1 + by1 + c)/sqrt(a**2 + b**2)
    top = abs(y - slope*x - c)
    bottom = m.sqrt(slope*slope + 1)
    return top/bottom
```

RANSAC Algorithm to determine 4 best lines

In [178]:

```
def ransac(image, normal_image, num_of_lines, num_of_points):

    #getting valid pixel points
    points = get_valid_pixels(image,0)
    lines_found = 0

    while(lines_found < num_of_lines):
        #get 2 random points
        p1, p2 = get_random(points)

        #get a model line between these 2 points
        slope, c = get_m_and_c(p1, p2)
        #to keep a track of points that are close enough
        inliers = []

        # Variable for the largest distance a line can have based on its in
liers
        max_line_size = 0

        #Check each point if it's an inlier or not
        for p in points:
            # Get distance between line and point
            min_dist = get_min_dist(slope, c, p[0], p[1])

            #check if it's close enough
            if min_dist < 3:
                inliers.append(p)

        #check if numbers of inliers is greater than number of points neede
d to call it a valid line
        if(len(inliers) > num_of_points):
            lines_found += 1

            #remove inliers from original points so that they are not reuse
d
            for p1 in inliers:
                points.remove(p1)

                #plotting the inliers as 3x3 squares
                for i in range(0, 3):
                    for j in range(0, 3):
                        #checking if the point is out of bound
                        if ((p1[0] + i - 1) > image.shape[0]) or ((p1[1] +
j - 1) > image.shape[1]):
                            continue
                        else:
                            image[p1[1] + j - 1,p1[0] + i - 1] = 255

                #looping through inliers to find two farthest points
                for p2 in inliers:
                    #distance between two points
```

```
                    dist = m.sqrt(((p2[0] - p1[0])**2) + ((p2[1] - p1[1])**
2))

                    #check if it is greater than the max distance
                    if dist > max_line_size:
                        max_line_size = dist
                        farthest = (p1,p2)

            #plot line between two farthest points on the image with points
            cv2.line(image, farthest[0], farthest[1], (255, 255, 255), thic
kness=1)

            #plot line between two farthest points on the normal image
            cv2.line(normal_image, farthest[0], farthest[1], (0, 0, 0), thi
ckness=2)

        # Once the four strongest lines have been found show them on the im
age
        if lines_found == 4:
            cv2.imshow("RANSAC Point image", image)
            cv2.imwrite("RANSACPointImage.jpg", image)
            cv2.waitKey(0)

            cv2.imshow("RANSAC Normal image", normal_image)
            cv2.imwrite("RANSACNormalImage.jpg", normal_image)
            cv2.waitKey(0)
```

Hough Transform to find 4 strongly supported lines

In [179]:

```
def hough_trans(image, normal_image, num_lines):
    r,c = image.shape

    #max and min value of rho
    max_rho = r + c
    min_rho = -c

    # since we cannot plot negative rho values we make it positive with thi
s and then subtract again to get original rho
    zero_maker = -min_rho

    rho_range = max_rho - min_rho

    #vote collector
    vc = np.zeros((rho_range, 181))

    #get feature points from the image
    points = get_valid_pixels(image, 0)

    #looping through all points
    for p in points:
        x,y = p

        #looping through all angles
        for deg in range(0, 181):
            rad = m.radians(deg)
            #rho = xcos(theta) + ysin(theta)
            rho = int(x*m.cos(rad) + y*m.sin(rad) + zero_maker)

            #vote for every rho angle pair
            vc[rho,deg] += 50 #more votes for better display

    #display image
    cv2.imshow("Vote Collector", vc/255)
    cv2.imwrite("VoteCollector.jpg", vc)
    cv2.waitKey(0)

    #current num of lines drawn
    lines_drawn = 0

    #highest value in hough transform
    maxim = 0

    #applying non max suppression to vote collector
    svc = nms(vc)
    cv2.imshow("Suppressed Vote Collector", svc/255)
    cv2.imwrite("SuppressedVoteCollector.jpg", svc)
    cv2.waitKey(0)

    while lines_drawn < num_lines:
        # Find the max value in the hough transform which should corelate t
```

```
o the parameters of the strongest line
        for i in range(svc.shape[0]):
            for j in range(svc.shape[1]):
                if svc[i,j] > maxim:
                    maxi_pair = (i,j)
                    maxim = svc[i,j]

        #setting maximum to zero for next iteration
        maxim = 0

        #setting neighbours to 0 so we don't find the same line again and a
gain
        svc[maxi_pair[0] - 10:maxi_pair[0] + 11, maxi_pair[1] - 10:maxi_pai
r[1] + 11] = 0

        theta = m.radians(maxi_pair[1])
        #getting original rho
        rho = maxi_pair[0] - zero_maker

        #getting line params
        p = m.cos(theta)
        q = m.sin(theta)
        x_temp = p*rho
        y_temp = q*rho

        #generating two points
        p1 = (int(x_temp + 10000*(-q)), int(y_temp + 10000*(p)))
        p2 = (int(x_temp - 10000*(-q)), int(y_temp - 10000*(p)))

        #plot line on the point image
        cv2.line(image, p1, p2, (255, 255, 255), thickness=1)

        #plot line on the normal image
        cv2.line(normal_image, p1, p2, (0, 0, 0), thickness=2)

        lines_drawn += 1

    cv2.imshow("Hough Lines", image)
    cv2.imwrite("HoughLines.jpg", image)
    cv2.waitKey(0)

    cv2.imshow("Hough Lines Original image", normal_image)
    cv2.imwrite("HoughLinesOriginalImage.jpg", normal_image)
    cv2.waitKey(0)
```

Main function

In [180]:

```
if __name__ == "__main__":

    image = cv2.imread("road.png")
    i_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    cv2.imshow("Input Image", i_gray)
    cv2.waitKey(0)

    gaus = filter_the_img(i_gray, gauss_filter(5,1), 5)

    #getting points using hessian
    points = hes(gaus)

    #showing the key points
    cv2.imshow("Suppressed key points", points.copy())
    cv2.imwrite("SuppressedKeyPoints.jpg", points.copy())
    cv2.waitKey(0)

    #RANSAC algo to find 4 lines with at least 35 inliers each
    ransac(points.copy(), i_gray.copy(), 4, 35)

    #hough transform to find 4 best lines
    hough_trans(points.copy(), i_gray.copy(), 4)
```

Gaussian filtering and how a filter is applied to an image has already been discussed. Shortly, you just take a filter and perform a dot product with corresponding similar sub matrix of pixels. Sobel filters are used to find derivatives with respect to x and y. Second derivatives required for the Hessian determinant are also obtained using the same Sobel filters. Then we calculate the Hessian determinant. Then we fit the pixel values between 0 and 255. After doing that we threshold around a particular pixel value so that we can obtain important pixels. Then we apply non maximum suppression to that matrix for the sake of reducing the number of pixels. These points are passed into RANSAC with two parameters namely number of lines and number of inliers required for a valid line model. Then we select two random points and form a line out of them and then find distances of remaining points from that line. If it is below a certain distance then the point is considered an inlier. After finding sufficient inliers we find two farthest inliers and draw a line between them. Now for Hough Transform, we define a vote collector for 0 to 180 degrees. Rho value can be negative so we defined a constant that makes rho 0 when added to minimum rho. After the process we subtract the constant to get original rho. Every point votes for all possible lines that can pass from there. After that we supply non maximum suppression to the Vote Collector to get more discrete points. Thereafter, we unless we find the required number of lines, we find the bin with maximum votes. Then after storing rho and theta we make that bin and its neighbouring bins equal to zero. So that we find a different maximum in the next iteration. Then we find two distant points lying on that line. Then plot the line on the image.