Kyungpook National University

Program Assignment#1:

IMPLEMENT LANE DETECTION ALGORITHM

Student name and ID: Pham Quy Duc Thinh – ID: 2019220764

Course: *Topics in Computer Vision (COME824)* – Professor: **Heechul Jung**

Prerequisite

• Programming Language: I recommend the Python language, but you can use any kind of programming languages. e.g. Matlab, C, C++, JAVA ....

• Do not use any computer vision libraries to implement convolution, RANSAC, edge detection, or thresholding. e.g. OpenCV, Matlab Image Processing Toolkit ....

• Submit your code in a ".zip" file and submit your report (".pdf" file) to explain kindly your implementation and to discuss your results. (Due: 11:59pm, 07-June-2020)

• Putting comments in your code

The library using in the assignment are: numpy, random, matplotlib, and the function cv2.imread, and cv2.cvtColor to read, and transfer image to gray scale. All the implementation is made by using numpy.

Task 0. Implement a function that convolves image with a given kernel. (20 pt)

The convolution function consisted of 2 input which are the *image*, and *kernel*

+ Image should be converted to the gray scale by using the function in the openCV library

cv2.cvtColor

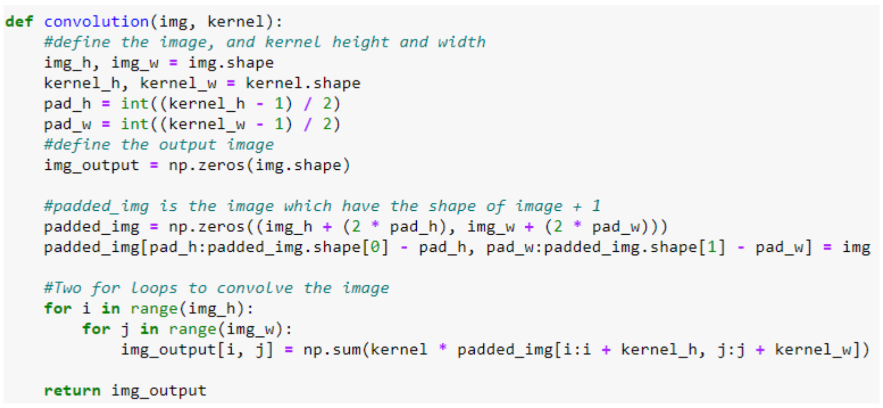
+ Kernel is a 3x3 kernel size

+ The algorithm for convolves image is described below:

- Define 7 variables which are: *img_h, img_w, kernel_h, kernel_w, pad_h, pad_w, and img_output*. They denoted the image height, image width, kernel height, kernel width, pad height, pad width, and the output, respectively
- Create two for loops that looping along the height (img_h), and the width(img_w) with the following function inside the last for loop

$$output = \sum_j \sum_i x[i,j] \cdot h[1-i, 0-j]$$

- The completed code is shown below:



```python
def convolution(img, kernel):
    #define the image, and kernel height and width
    img_h, img_w = img.shape
    kernel_h, kernel_w = kernel.shape
    pad_h = int((kernel_h - 1) / 2)
    pad_w = int((kernel_w - 1) / 2)
    #define the output image
    img_output = np.zeros(img.shape)

    #padded_img is the image which have the shape of image + 1
    padded_img = np.zeros((img_h + (2 * pad_h), img_w + (2 * pad_w)))
    padded_img[pad_h:padded_img.shape[0] - pad_h, pad_w:padded_img.shape[1] - pad_w] = img

    #Two for loops to convolve the image
    for i in range(img_h):
        for j in range(img_w):
            img_output[i, j] = np.sum(kernel * padded_img[i:i + kernel_h, j:j + kernel_w])

    return img_output
```

Task 1. Detect Edges from the given image (*lane.png*) using the convolution function you implemented in Task 0 (20 pt)

In my assignment, I used the Sobel edge detectors to detect the edge. The Sobel edge detectors function consisted of input which are the image (gray scale). Generally, the algorithm is shown below:

- Define the kernel filter X, Y which are:

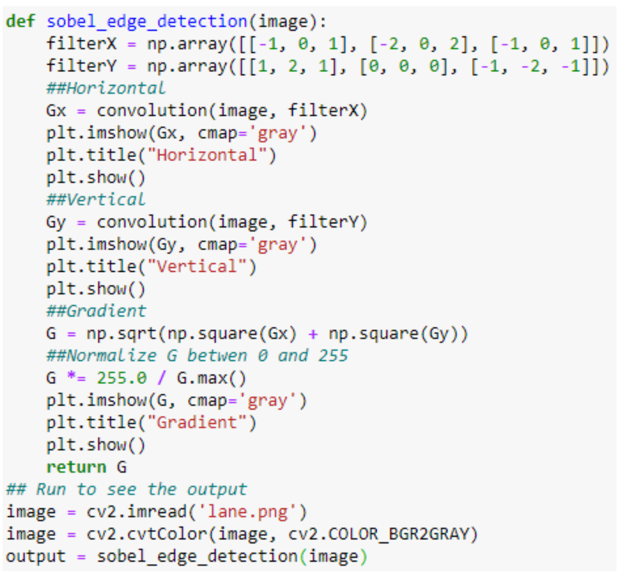
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- Since the X-coor is defined as increasing in the right-dir, and the y-coor is defined as increasing in the down-dir. Thus, the resulting gradient approximations can be expressed below:

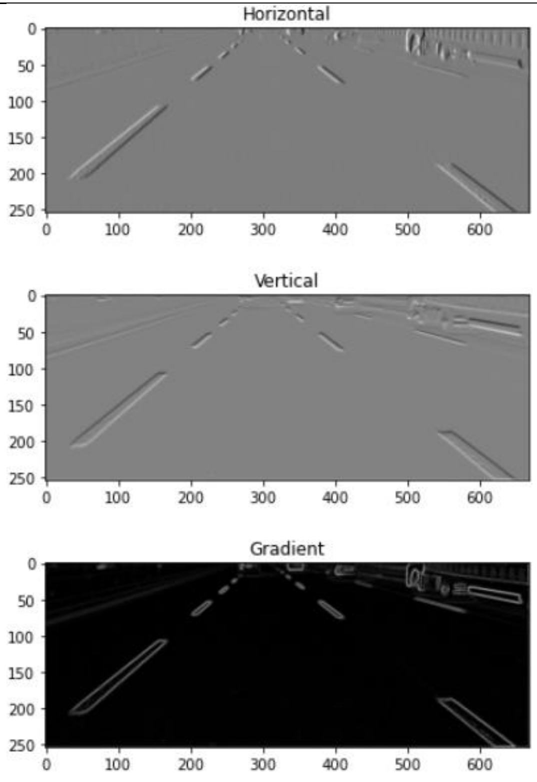
$$G = \sqrt{{G_x}^2 + {G_x}^2}$$

- Normalized the output to be between 0 and 255 for gray scale picture
- The completed code and output are shown below:

Code	Output



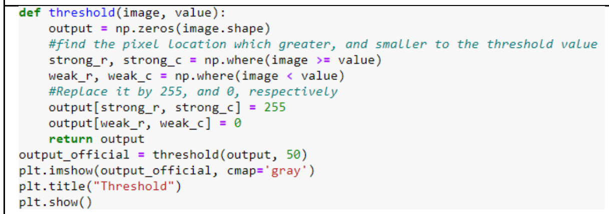
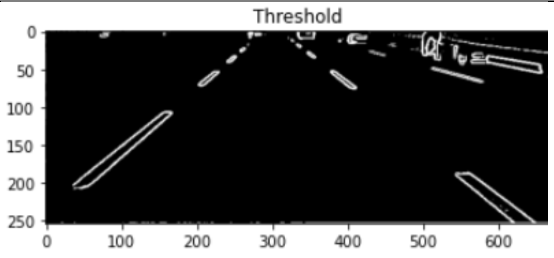
```python
def sobel_edge_detection(image):
    filterX = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    filterY = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    ##Horizontal
    Gx = convolution(image, filterX)
    plt.imshow(Gx, cmap='gray')
    plt.title("Horizontal")
    plt.show()
    ##Vertical
    Gy = convolution(image, filterY)
    plt.imshow(Gy, cmap='gray')
    plt.title("Vertical")
    plt.show()
    ##Gradient
    G = np.sqrt(np.square(Gx) + np.square(Gy))
    ##Normalize G betwen 0 and 255
    G *= 255.0 / G.max()
    plt.imshow(G, cmap='gray')
    plt.title("Gradient")
    plt.show()
    return G
## Run to see the output
image = cv2.imread('lane.png')
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
output = sobel_edge_detection(image)
```



Task 2. Thresholding the edge images. (10 pt)

A simple threshold function is implemented in the Code to get the important edge i.e., lane edge. It consisted of 2 input that is image, and the threshold number we can defined. Step by step for implemented the threshold function is described below:

- Find the pixel location which have the pixel value greater, and smaller than the threshold value we put in the function
- Replace the pixel location that greater than threshold value by 255 (white), and the smaller one is 0 (black)
- Code and output for **threshold value = 50** are:

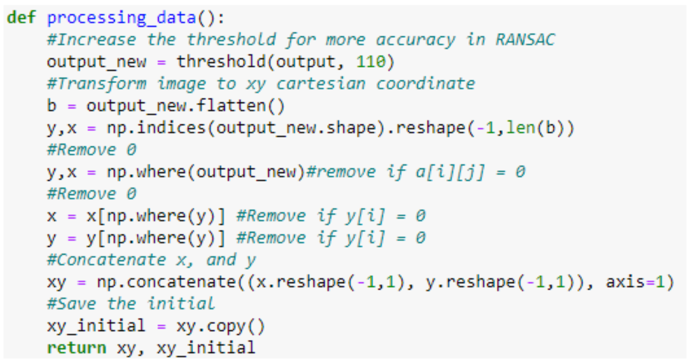
Code	Output
	

```python
def threshold(image, value):
    output = np.zeros(image.shape)
    #find the pixel location which greater, and smaller to the threshold value
    strong_r, strong_c = np.where(image >= value)
    weak_r, weak_c = np.where(image < value)
    #Replace it by 255, and 0, respectively
    output[strong_r, strong_c] = 255
    output[weak_r, weak_c] = 0
    return output
output_official = threshold(output, 50)
plt.imshow(output_official, cmap='gray')
plt.title("Threshold")
plt.show()
```

Task 3. Perform a RANSAC algorithm to detect lines which constitutes the lane on the Edge Image and Draw the lines on the image. All the examples below are good results for this task. (50 pt)

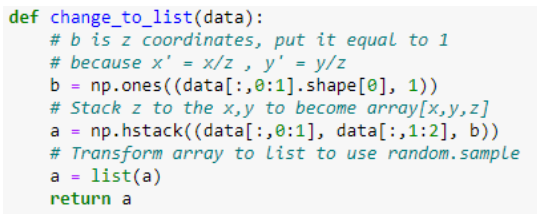
RANSAC (Random sample consensus) is an algorithm to estimate parameters of a model by random sampling of observed data. Thus, it was used to find the optimal fitting line to the lane picture even the picture had a lot of outlier. In my assignment, to implement the RANSAC algorithm, 7 function has to be developed. The main RANSAC function is the last function. The explanation of these function and the algorithm behind the RANSAC method will be discussed in detail below:

- First is the processing data function (*def processing_data()*): Instead of reading the x, y coordinates point in the image, we can transform the image into the list of array x, and y. This work can be done by flatten the image after thresholding and remove the 0 pixel to keep the non-zero pixel. And finally, concatenating the x, y to become an $m \times 2$ arrays. Since I used the Sobel edge detector. Thus, it is necessary to increase the threshold to get more accuracy at the RANSAC algorithm. That is why I used threshold equal to 110 in the processing data function. Below is the completed code of the function



```
def processing_data():
    #Increase the threshold for more accuracy in RANSAC
    output_new = threshold(output, 110)
    #Transform image to xy cartesian coordinate
    b = output_new.flatten()
    y,x = np.indices(output_new.shape).reshape(-1,len(b))
    #Remove 0
    y,x = np.where(output_new)#remove if a[i][j] = 0
    #Remove 0
    x = x[np.where(y)] #Remove if y[i] = 0
    y = y[np.where(y)] #Remove if y[i] = 0
    #Concatenate x, and y
    xy = np.concatenate((x.reshape(-1,1), y.reshape(-1,1)), axis=1)
    #Save the initial
    xy_initial = xy.copy()
    return xy, xy_initial
```

- Next is the *change_to_list* function with one input is the data after *processing_data* function. This function is to transform the x, y cartesian coordinates into the homogenous coordinates with the Z coordinate equal to 1 for all the x, and y because x' = x/z, and y' = y/z. This implementation is a must to perform the cross product to find the line between 2 points. Final output of the change_to_list function is to transfer the array of x,y,z to list in order to use the random.sample to randomly pick 2 points (random.sample only can work with list).



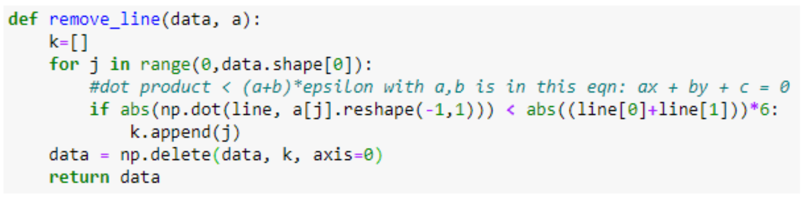
```
def change_to_list(data):
    # b is z coordinates, put it equal to 1
    # because x' = x/z , y' = y/z
    b = np.ones((data[:,0:1].shape[0], 1))
    # Stack z to the x,y to become array[x,y,z]
    a = np.hstack((data[:,0:1], data[:,1:2], b))
    # Transform array to list to use random.sample
    a = list(a)
    return a
```

- Next is the remove_line function. It had 2 input that is the output of the first and second function. This function helps to remove all the point that is in the lane line after we successfully detect that line to easily detect the line which near to it. The mathematic behind this remove_line flag is:

  Given A (x_1, y_1) and the line: $ax + by + c = 0$.

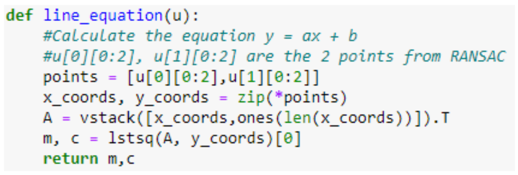
  If A is near or in the line

  ⇨ $ax_1 + by_1 + c < |a+b|\varepsilon$

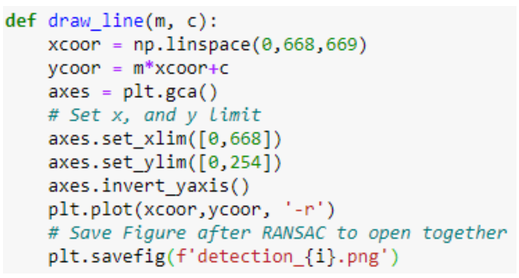


```
def remove_line(data, a):
    k=[]
    for j in range(0,data.shape[0]):
        #dot product < (a+b)*epsilon with a,b is in this eqn: ax + by + c = 0
        if abs(np.dot(line, a[j].reshape(-1,1))) < abs((line[0]+line[1]))*6:
            k.append(j)
    data = np.delete(data, k, axis=0)
    return data
```

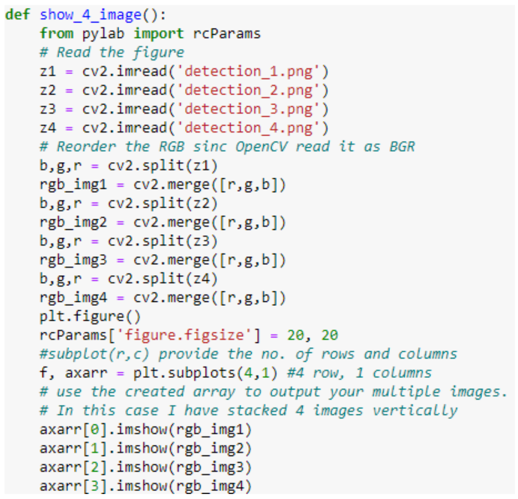
- Next is the 3 following function to calculate the line equation y = ax + b from 2 points from RANSAC algorithm, function to draw the lane line in the image, and the additional function to show the 4 red lane line together. The completed code is shown below:



```
def line_equation(u):
    #Calculate the equation y = ax + b
    #u[0][0:2], u[1][0:2] are the 2 points from RANSAC
    points = [u[0][0:2],u[1][0:2]]
    x_coords, y_coords = zip(*points)
    A = vstack([x_coords,ones(len(x_coords))]).T
    m, c = lstsq(A, y_coords)[0]
    return m,c
```



```
def draw_line(m, c):
    xcoor = np.linspace(0,668,669)
    ycoor = m*xcoor+c
    axes = plt.gca()
    # Set x, and y limit
    axes.set_xlim([0,668])
    axes.set_ylim([0,254])
    axes.invert_yaxis()
    plt.plot(xcoor,ycoor, '-r')
    # Save Figure after RANSAC to open together
    plt.savefig(f'detection_{i}.png')
```



```
def show_4_image():
    from pylab import rcParams
    # Read the figure
    z1 = cv2.imread('detection_1.png')
    z2 = cv2.imread('detection_2.png')
    z3 = cv2.imread('detection_3.png')
    z4 = cv2.imread('detection_4.png')
    # Reorder the RGB sinc OpenCV read it as BGR
    b,g,r = cv2.split(z1)
    rgb_img1 = cv2.merge([r,g,b])
    b,g,r = cv2.split(z2)
    rgb_img2 = cv2.merge([r,g,b])
    b,g,r = cv2.split(z3)
    rgb_img3 = cv2.merge([r,g,b])
    b,g,r = cv2.split(z4)
    rgb_img4 = cv2.merge([r,g,b])
    plt.figure()
    rcParams['figure.figsize'] = 20, 20
    #subplot(r,c) provide the no. of rows and columns
    f, axarr = plt.subplots(4,1) #4 row, 1 columns
    # use the created array to output your multiple images.
    # In this case I have stacked 4 images vertically
    axarr[0].imshow(rgb_img1)
    axarr[1].imshow(rgb_img2)
    axarr[2].imshow(rgb_img3)
    axarr[3].imshow(rgb_img4)
```

- Final function is the main function to implement the RANSAC algorithm. The details mathematics behind the implementation is described below:
    - First, randomly pick two points from the image data point by using the (random.sample)
    - Then, if these points have the y-coor value different to each other, and both of them are greater than 80. Calculate the *cross product* of these point to compute the line between them (see **Figure 1**). The if function is an additional conditional statement in order to prevent the RANSAC algorithm choose the horizontal line instead of lane line, and also prevent it to pick the line near the Truck. This work can be optimized by create one more function to prevent it take the truck point.

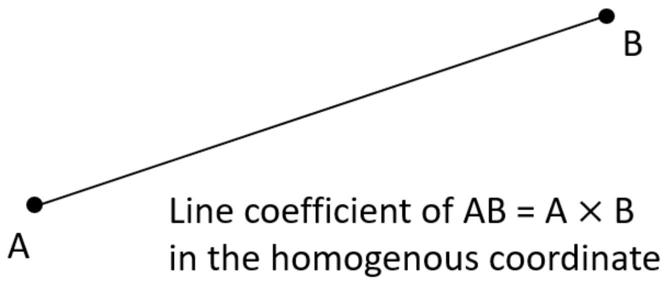
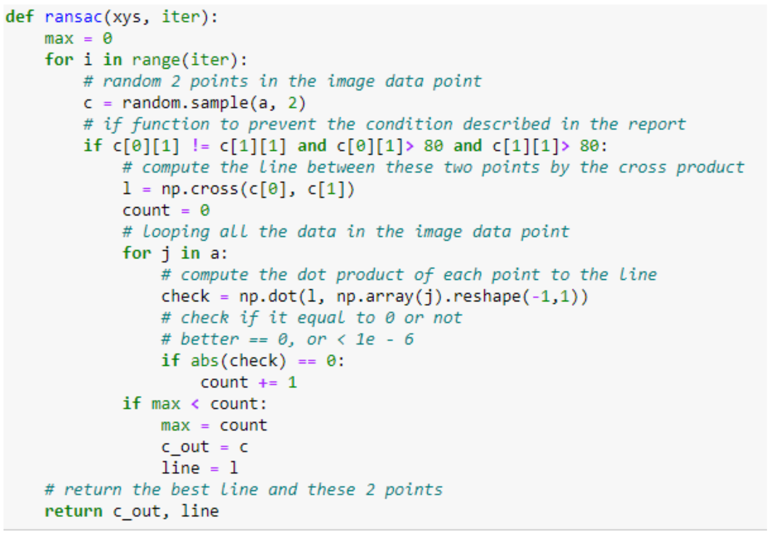


Figure 1. Cross product of 2 points in a homogenous coordinate

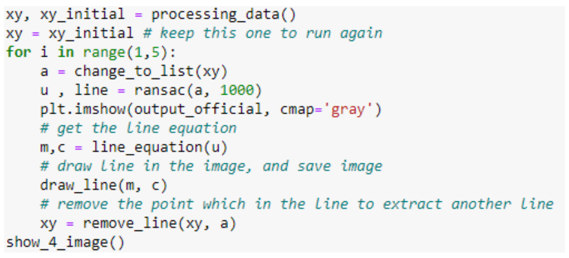
    - Then, looping all the point in the image data point to test if them belong to the point or not. This work can be done by compute the *dot product* between the point and the line coefficient computed above. If the dot product of them equal to nearly zero, then this point is belonging to the line, and vice versa.
    - Finally, return the line which have the highest point belong to it. Below is the completed code for the algorithm



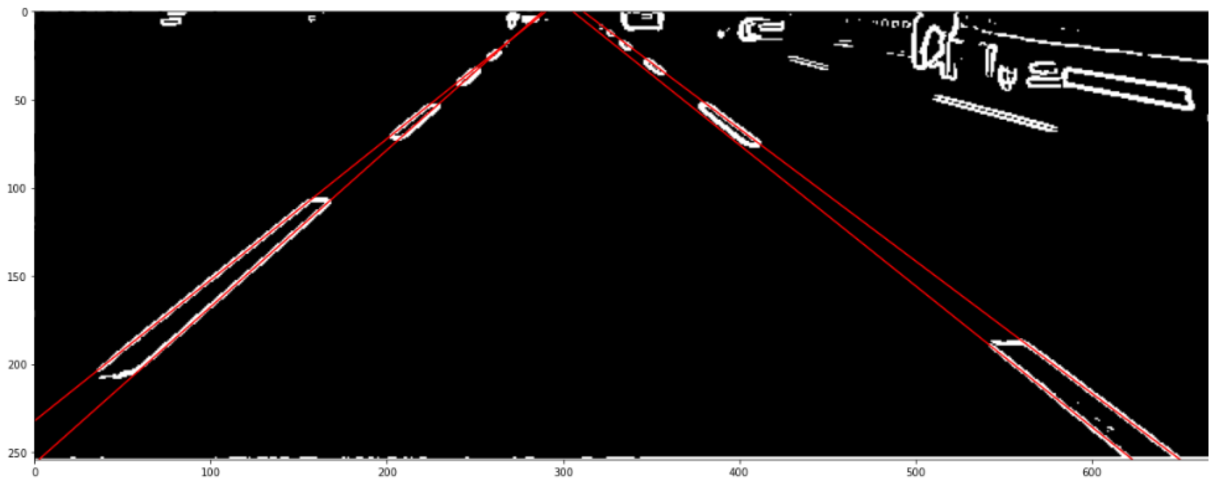
```python
def ransac(xys, iter):
    max = 0
    for i in range(iter):
        # random 2 points in the image data point
        c = random.sample(a, 2)
        # if function to prevent the condition described in the report
        if c[0][1] != c[1][1] and c[0][1]> 80 and c[1][1]> 80:
            # compute the line between these two points by the cross product
            l = np.cross(c[0], c[1])
            count = 0
            # looping all the data in the image data point
            for j in a:
                # compute the dot product of each point to the line
                check = np.dot(l, np.array(j).reshape(-1,1))
                # check if it equal to 0 or not
                # better == 0, or < 1e - 6
                if abs(check) == 0:
                    count += 1
            if max < count:
                max = count
                c_out = c
                line = l
    # return the best line and these 2 points
    return c_out, line
```

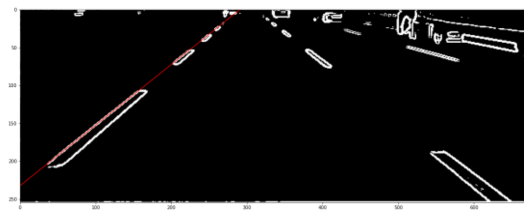
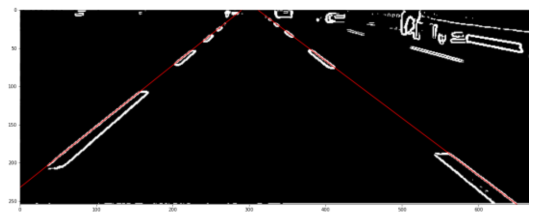
Finally, run the code and see the result. It was noticed that because the assignment uses the Sobel edge detection instead of Canny edge detection. So, the accuracy can not be equal to one using the Canny.

The accuracy to get 3 true line is 90% and the accuracy to get the all 4 line is 70%. Below is the main code to call these 7 functions and the output. It was run in 4 iteration in order to extract 4 lines.



```
xy, xy_initial = processing_data()
xy = xy_initial # keep this one to run again
for i in range(1,5):
    a = change_to_list(xy)
    u , line = ransac(a, 1000)
    plt.imshow(output_official, cmap='gray')
    # get the line equation
    m,c = line_equation(u)
    # draw line in the image, and save image
    draw_line(m, c)
    # remove the point which in the line to extract another line
    xy = remove_line(xy, a)
show_4_image()
```



Iteration 1	Iteration 2
	
Iteration 3	**Iteration 4**