Finding Lane Lines on the Road

Project Goal:

The goals / steps of this project are the following:

- Making a pipeline that finds lane lines on the road
- Reflection on my work in a written report

REFLECTION

1.DESCRIPTION:

Introduction:

This project is to find the lane lines on the road using Python3 and OpenCV. The following techniques are used effectively to detect the lane lines:

1) Grayscale conversion
2) Gaussian blur filter
3) Canny Edge detection
4) Region of Interest selection
5) Hough Transform to draw lines

The following Python libraries are mainly used in this project:

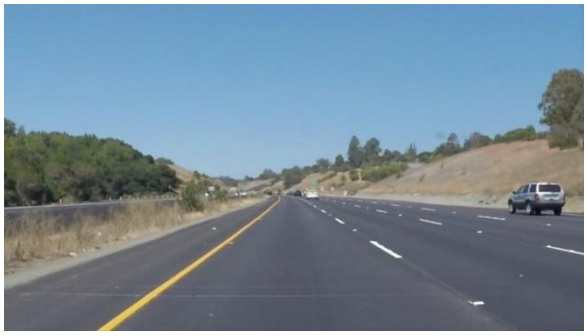
1) cv2 – OpenCV for Python
2) matplotlib.plot – Plotting the images
3) matplotlib.image – Read/Write images
4) numpy – for math operations

Test input:

The test input is both RGB colour image and RGB colour video. For explanation of the algorithm, I am using these test images. The lane lines are solid yellow, solid white and dotted white lines.

The following are the test images:






Pipeline:

My pipeline has 7 steps. The following gives clear details about each step.

Step 1:

The first step is to convert the colour images to grayscale so that it can be used to detect edges.

The following is the code snippet for this step,

cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
where
image is the original coloured image,
return is the grayscale image of the original coloured image

Once this step is executed, the input images looks like below: (Left - input image, Right -output image)











Step 2:

Next step is to blur the grayscale image to reduce noise in the image. This is implemented by using Gaussian blur technique.

The following is the code snippet for applying Gaussian blur
cv2.GaussianBlur(image, (kernel_size, kernel_size),0)
where,
image is the grayscale image
kernel_size is the size of blur filter
return is the required image

Once this step is executed, the input images looks like below: (Left - input image, Right - output image)









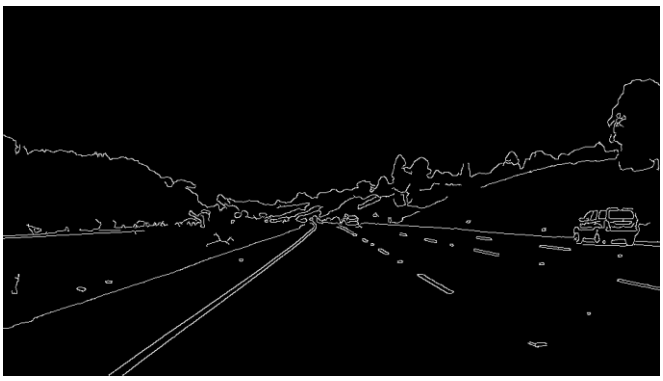


Step 3:

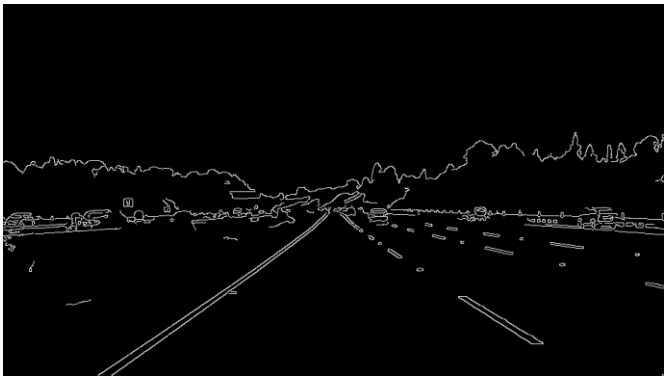
Next step is to find the edges in the noise reduced grayscale image (blurred grayscale image) using Canny Edge Detection technique.

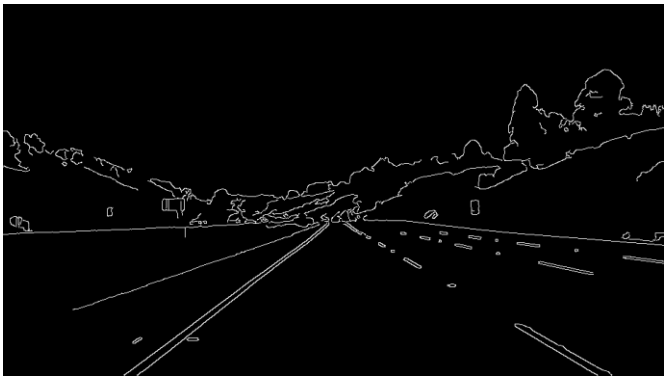
The code snippet is
cv2.Canny(image, low_threshold, high_threshold)
where,
image is the noise reduced grayscale image
high_threshold is the gradient value above which will be considered as strong edges
low_threshold is the gradient value below which will be rejected as weak edges
return is the required image

Once this step is executed, the input images looks like below: (Left - input image, Right - output image)







Step 4:

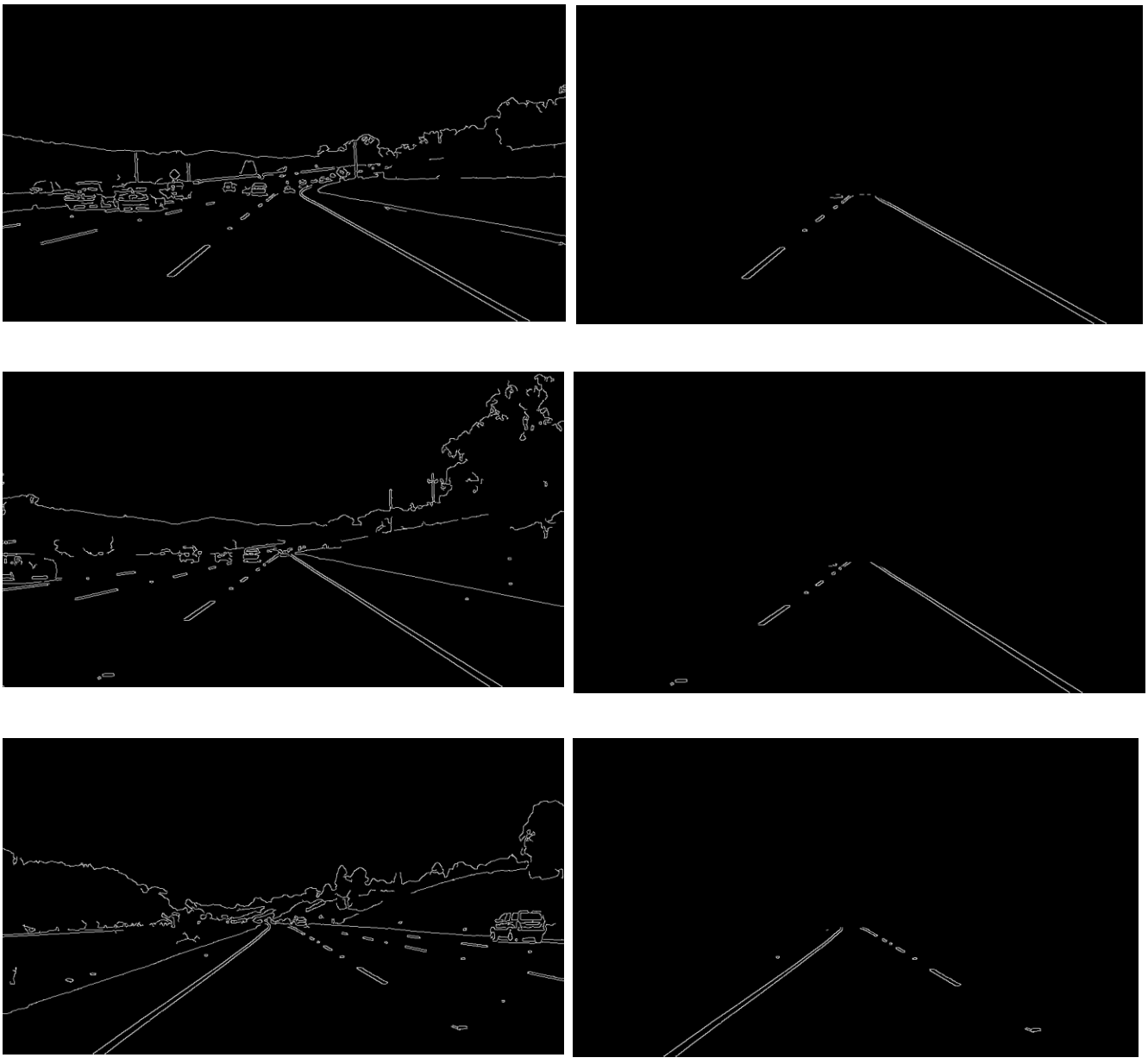
The previous step detects edges throughout the image. Since the camera orientation and position is fixed, it is known that the lanes required lie in certain region of the image. So, it is worth to process the required region than to process the whole image. Hence to filter the required region, I used a trapezoidal mask which looks like below: (white is the required region and black is the region to be removed)

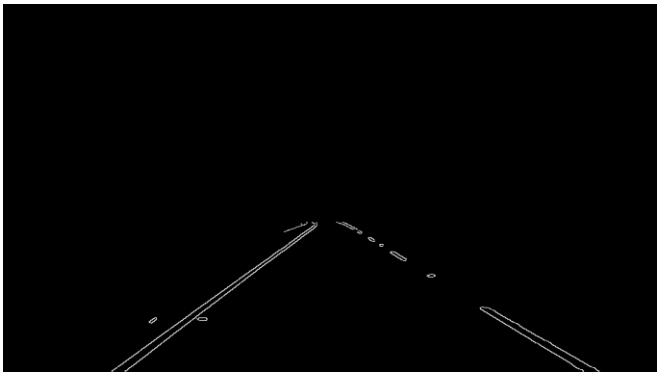
To create this mask, I used the following code:
cv2.fillPoly(image, vertices, color)
where,
image is the image on which the trapezoid should be drawn. Here it is a black image.
vertices are the vertices of the polygon.
color is the colour of the polygon
return is the required image

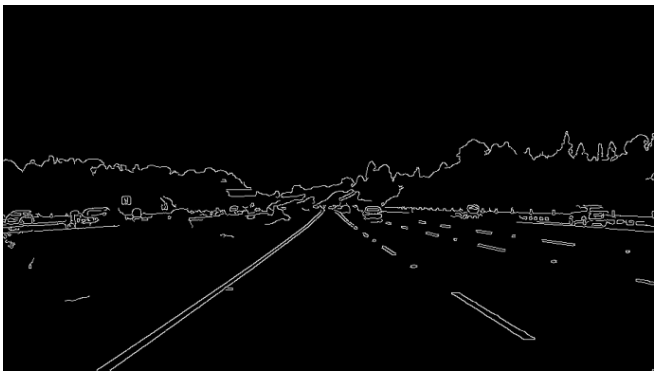
Once the above mask is applied, the edges detected in Step 3 will get filtered to the region.

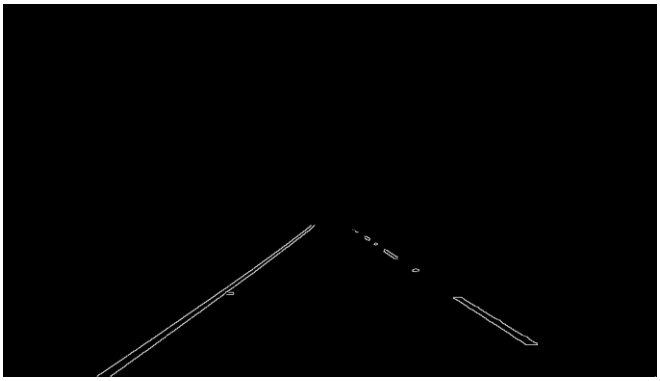
The code snippet is
cv2.bitwise_and(edge_image, ROI_image)
where edge_image is the image obtained in Step 3
ROI_image is the trapezoidal mask
return is the required image

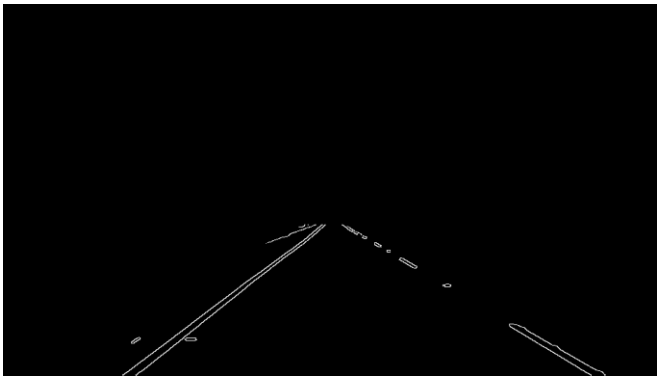
Once this step is executed, the input images looks like below: (Left - input image, Right - output image)











Step 5:

The Canny Edge detection technique does not use continuous lines to highlight edges. It is a series of closely located points. Hence to draw a continuous line, I used Hough Transform Line Detection technique.

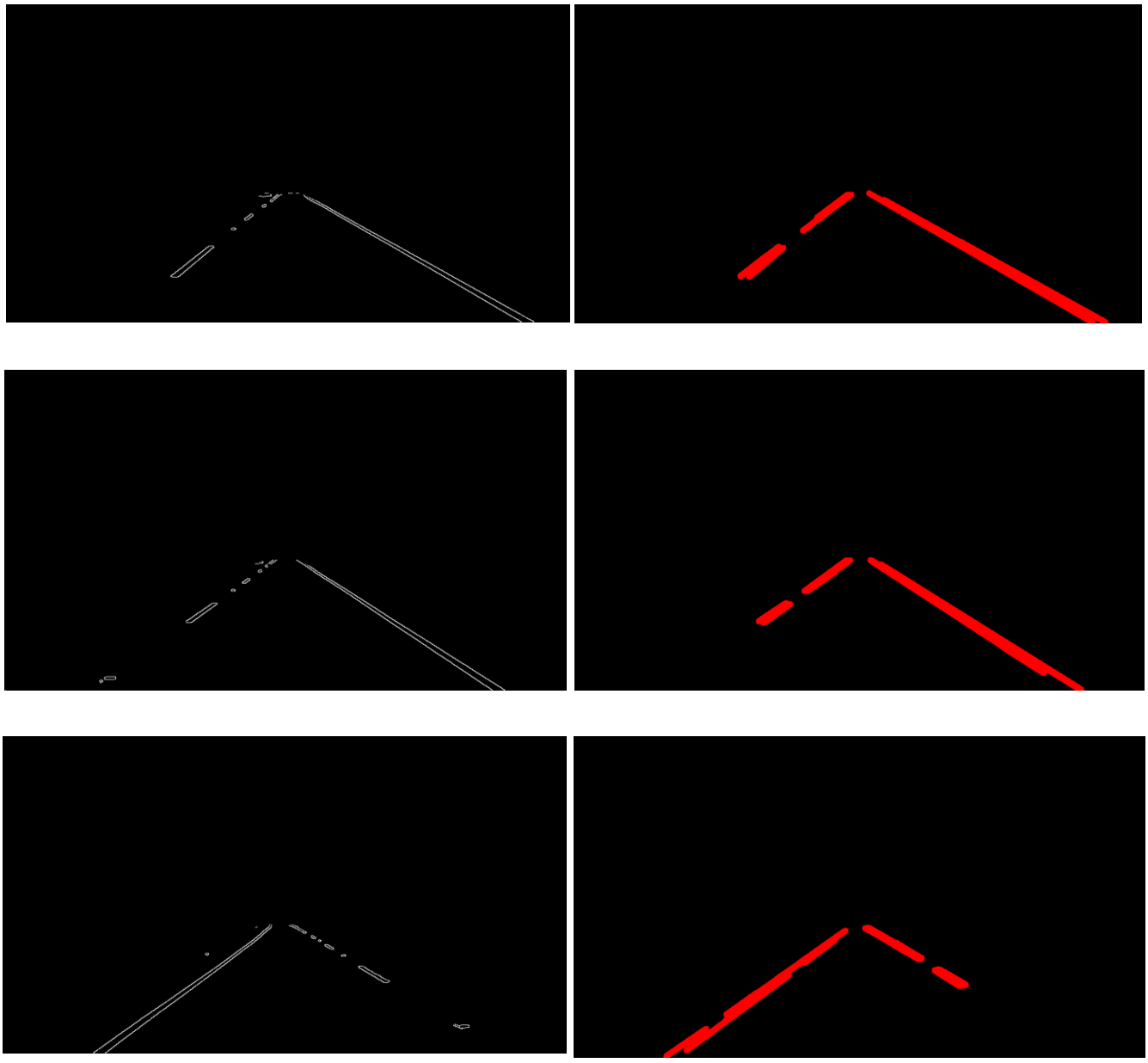
The code snippet is,
cv2.HoughLinesP(image, rho, theta, threshold, np.array([]),min_line_length, max_line_gap)
where,
image is the input image on which lines to be drawn. Here it is the edge image obtained in Step 4
rho is distance resolution in pixels of the Hough grid
theta is angular resolution in radians of the Hough grid
threshold is minimum number of votes (intersections in Hough grid cell)
min_line_length is minimum number of pixels making up a line
max_line_gap is maximum gap in pixels between connectable line segments
return is a list with each element containing two points forming a line on the image

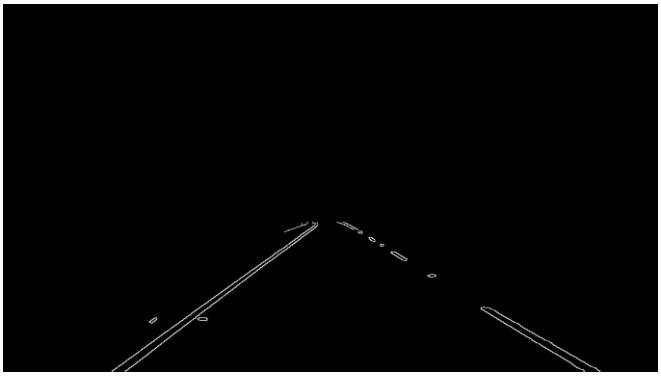
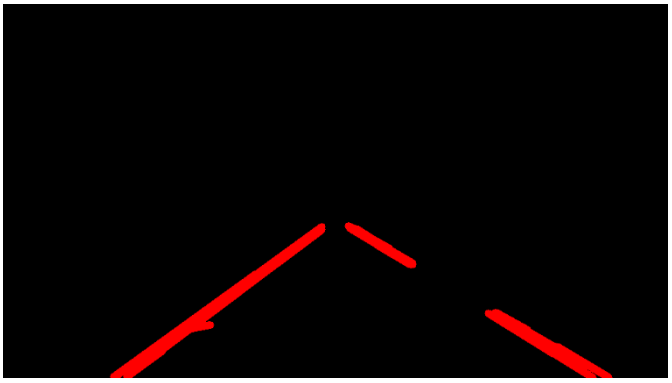
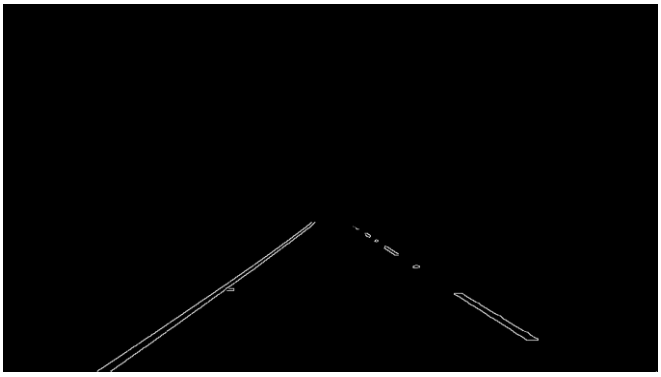
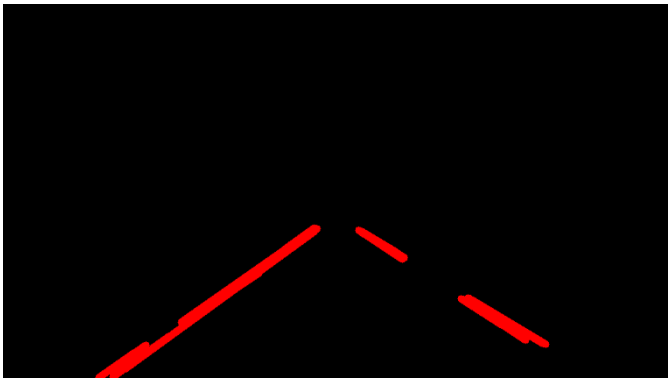
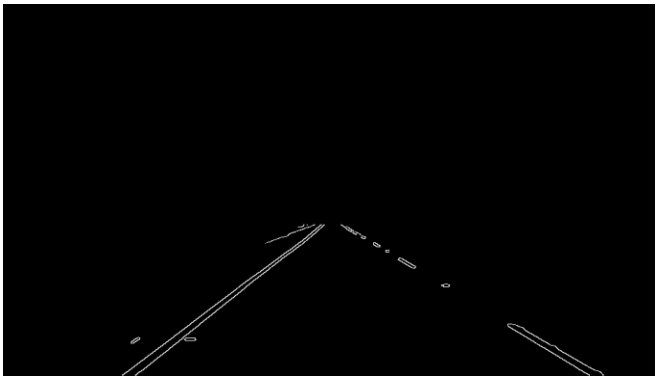
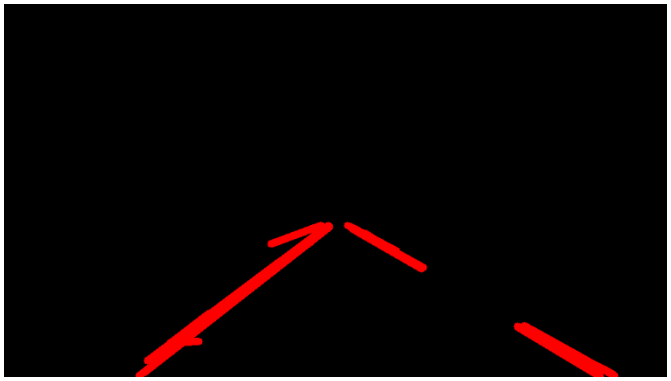
Step 6:

The above obtained list of line points is connected and drawn on the image.

The code snippet is,
```
for line in lines:
    for x1, y1, x2, y2 in line:
        cv2.line(image, (x1, y1), (x2, y2), color, thickness)
    return(image)
```
where,
image is the image on which lines to be drawn. Here it is a black image of appropriate dimensions.
(x1, y1) is point 1 of the line to be drawn
(x2, y2) is point 2 of the line to be drawn
color is the colour of the line to be drawn
thickness is the thickness of the line to be drawn
return is the image on which lines are drawn

Once this step is executed, the input images looks like below: (Left - input image, Right - output image)



Step 7:

Once we obtain the lines drawn on an empty image, I added that to the original image.

The code snippet is,
cv2.addWeighted(image1, α, image2, β, γ)
where,
image 1 and image 2 are the input images
α and β are the scaling factors
γ is the offset value
return is image = (image1 * α) + (image2 * β) + γ

Once this step is executed, the test images looks like below: (Left - original image, Right - output image)











Improvement:

It is clearly seen from above steps that the output line is not a single connected line showing the lane line. It is a series of lines showing the lane line. In order to draw a single line to show lane line, the following step should be replaced for Step 6

Step 6(Improved):

The goal is to find two lines for showing left and right side of the lane line. This turns out to determine four points, two for each line.

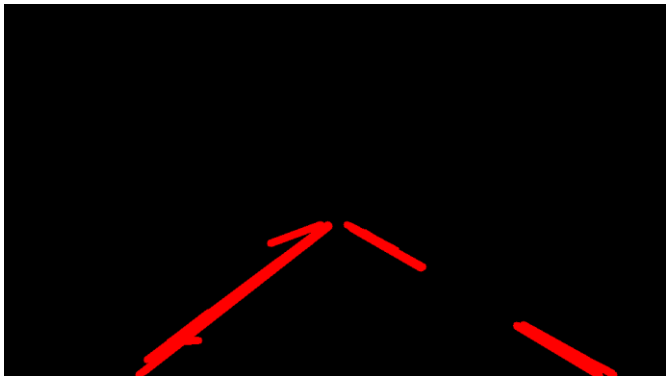
i) First, I separated the left and right lines by finding the slopes of the lines. It is clear that a slope which is negative is a left line and positive is a right line. The slope and the constant are appended to the lists correspondingly.
Once all the slopes and constants are separated, the average is found for each side.

The code snippet is,

```
#To detect left and right line's slope and constant
    for line in lines:
        for x1, y1, x2, y2 in line:
            m = (y2-y1)/(x2-x1)
            c = (y1 - (m*x1))
            if m < 0:
                left_const_list.append(c)
                left_slope_list.append(m)
            else:
                right_const_list.append(c)
                right_slope_list.append(m)

#Calculating slopes and constants
        left_slope = np.sum(left_slope_list)/len(left_slope_list)
        left_constant = np.sum(left_const_list)/len(left_const_list)
        right_slope = np.sum(right_slope_list)/len(right_slope_list)
        right_constant = np.sum(right_const_list)/len(right_const_list)
```

ii) I observed that due to some noisy unwanted lines in the list, the slope and constant gets deviated from the actual. An example image looks like this, (Check the left lane line, there is an additional line)



When I implement the logic for these kinds of lines, the slope and intercepts get deviated and the final output looks like below, (Check the ends of the left lane line, it goes out of the path)

Hence to remove these noisy unwanted lines, I can change the region of interest mask. But this may not work for all the images. So, I used average values as the base point and removed the slopes and intercepts which are far away from the average values correspondingly and recalculated the slopes and intercepts.

The code snippet is,

```
for i in range (0, len(left_slope_list)):
    slope = left_slope_list[i]
    const = left_const_list[i]
   #left_slope is the average of the slopes of left lines found in the previous step
    error = abs(left_slope – slope)
   #left_slope_error is the threshold above which the lines are considered to be noisy unwanted
```

```
            if(error <= left_slope_error):
                left_slope_filter.append(slope)
                left_constant_filter.append(const)

        for i in range(0,len(right_slope_list)):
            slope = right_slope_list[i]
            const = right_const_list[i]
           #right_slope is the average of the slopes of right lines found in the previous step
            error = abs(right_slope – slope)
           #right_slope_error is the threshold above which the lines are considered to be noisy un-
wanted
            if(error <= right_slope_error):
                right_slope_filter.append(slope)
                right_constant_filter.append(const)

#Recalculating the slopes and intercepts
        left_slope = np.sum(left_slope_filter)/(len(left_slope_filter))
        left_constant = np.sum(left_constant_filter)/(len(left_constant_filter))
        right_slope = np.sum(right_slope_filter)/(len(right_slope_filter))
        right_constant = np.sum(right_constant_filter)/(len(right_constant_filter))
```

iii) Once we find the slopes and intercepts of left and right side of the lane lines, then we need to find the points of the lines to draw.

It is obvious that the points should be somewhere on the border and around 60% of the border line. Hence the y-lines are obvious, one is ysize and the other is (0.6 of ysize) where ysize is the dimension of the image on y axis.

Using these y values and the slopes and intercepts from the previous step, x values can be found correspondingly.

The code snippet is,

```
left_y1 = int(ysize*0.6)
left_y2 = ysize

right_y1= int(ysize*0.6)
right_y2= ysize

left_x1  = ((left_y1 - left_constant)/(left_slope))
left_x1  = int(left_x1)

left_x2 = (ysize - left_constant)/(left_slope)
left_x2 = int(left_x2)

right_x1 = ((right_y1 - right_constant )/(right_slope))
right_x1 = int(right_x1)

right_x2 = (ysize - right_constant)/(right_slope)
right_x2= int(right_x2)
```

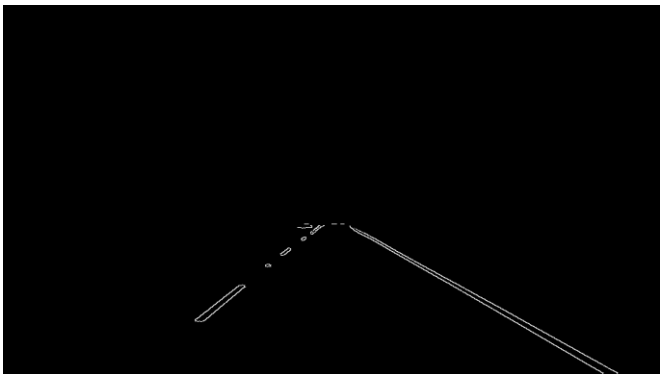
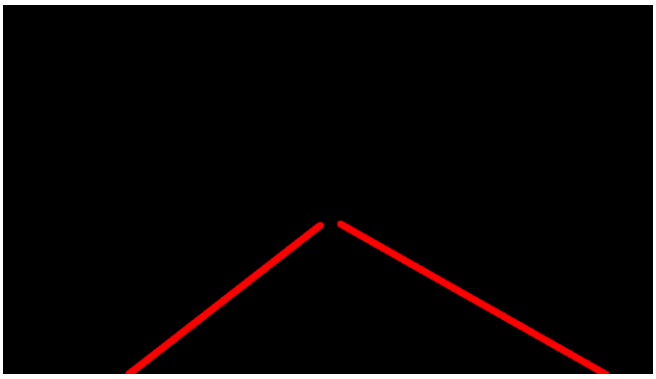
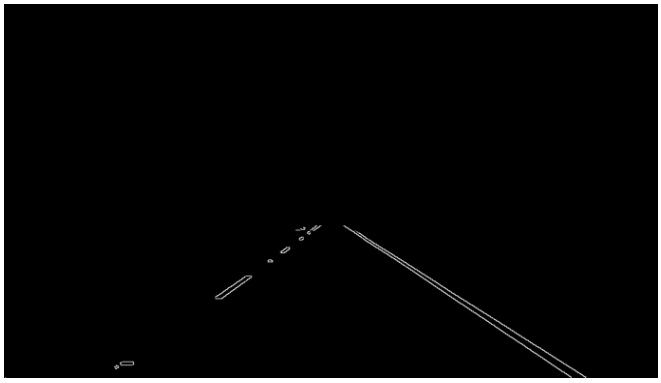
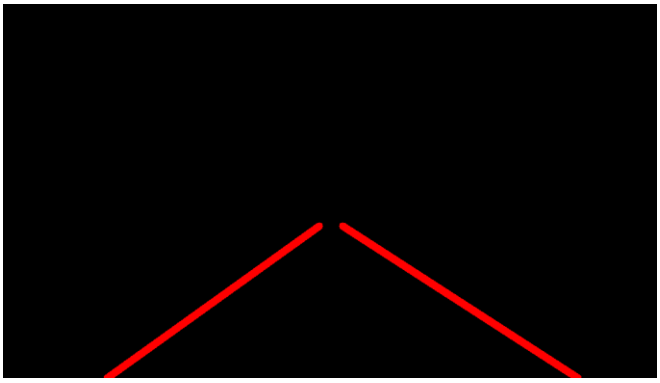
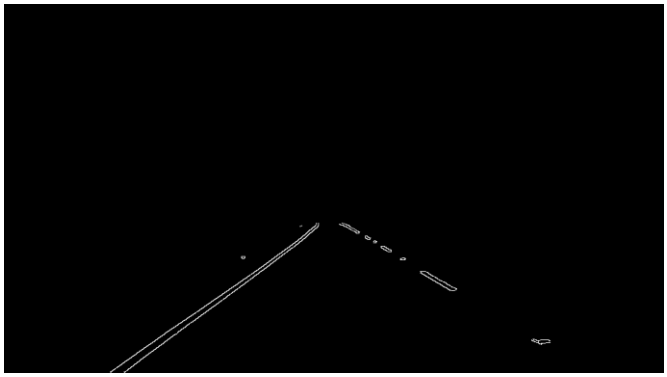
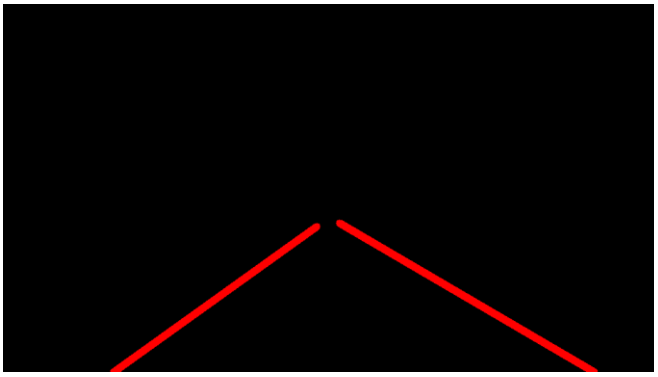
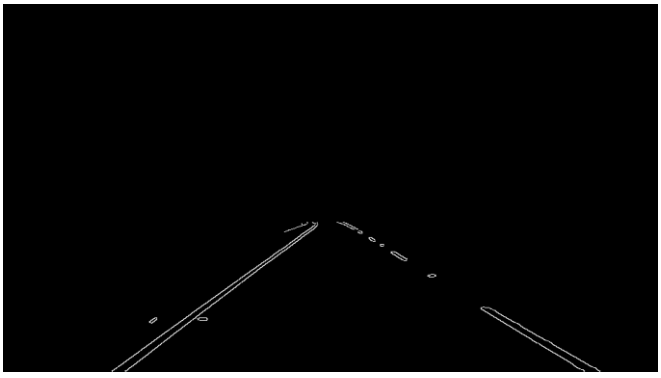
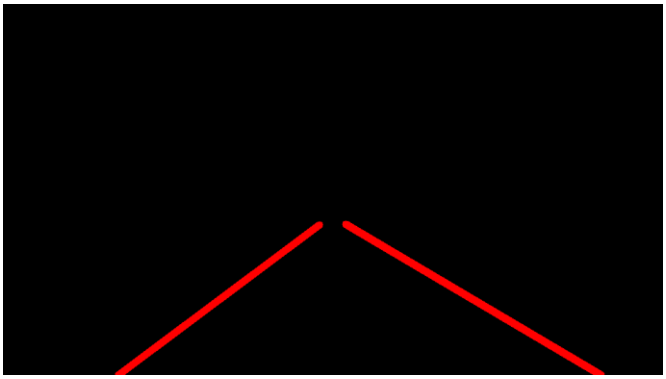
iv) Once points are obtained, lines are drawn on the image.

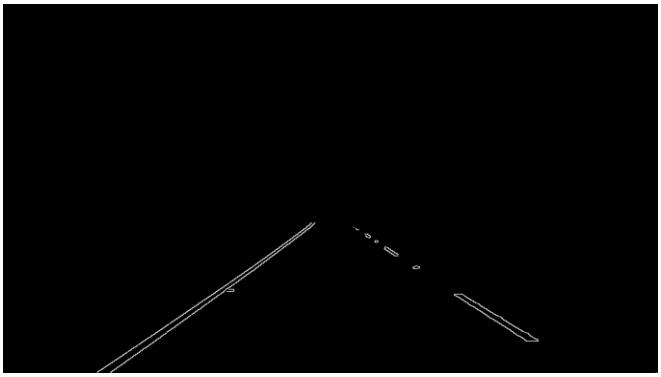
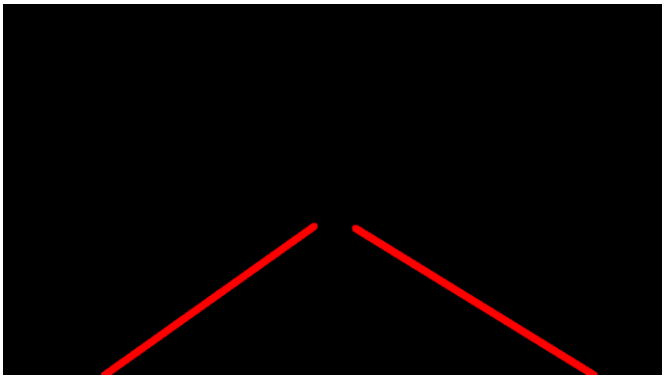
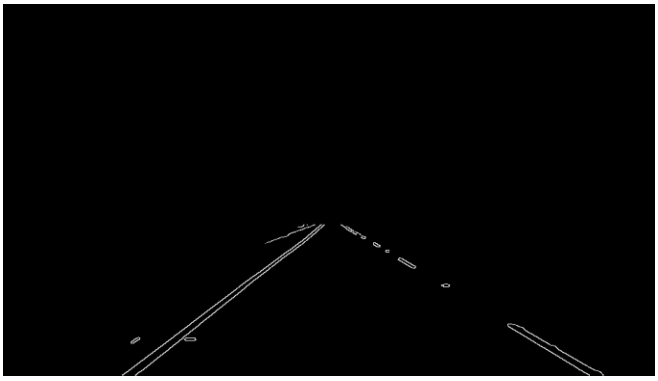
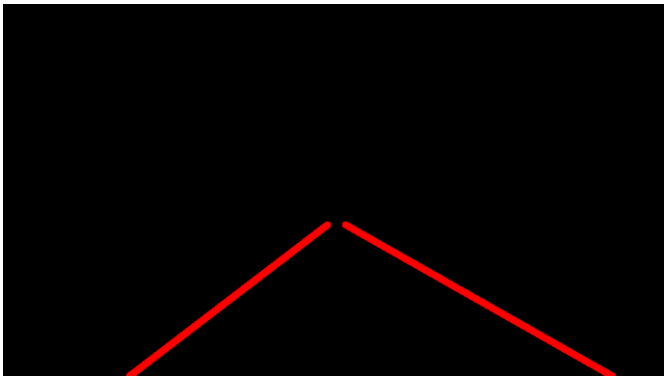
The code snippet is,

```
cv2.line(image,(left_x1,left_y1),(left_x2,left_y2),color,thickness)
cv2.line(image,(right_x1,right_y1),(right_x2,right_y2),color,thickness)
```

where,
image is the image on which lines to be drawn. Here it is a black image of appropriate dimensions.
(left_x1,left_y1) is point 1 of the left side lane line to be drawn
(left_x2,left_y2) is point 2 of the left side lane line to be drawn
(right_x1,right_y1) is point 1 of the left side lane line to be drawn
(right_x2,right_y2) is point 2 of the left side lane line to be drawn
color is the colour of the line to be drawn
thickness is the thickness of the line to be drawn
return is the image on which lines are drawn

Once this step is executed, the input images looks like below: (Left - input image, Right - output image)

After this improvement step the same previous Step 7 is applicable and the output looks like below after executing Step 7:










Conclusion:

With the above algorithm explained, I was able to identify the lane lines both from images and videos.

The raw and improvised output of the test images are available in this document and the improvised output of the test images are available in the folder *test_images_output*

The raw output video of the test input videos *solidWhiteRight.mp4* and *solidYellowLeft.mp4* are *solidWhiteRight_raw_line.mp4* and *solidYellowLeft_raw_line.mp4* respectively available in the folder *test_videos_output*

The extrapolated, averaged, filtered, improvised output video of the test input videos *solidWhiteRight.mp4* and *solidYellowLeft.mp4* are *solidWhiteRight.mp4* and *solidYellowLeft.mp4* respectively available in the folder *test_videos_output*

With the above discussed, the following is the details of the Project Rubrics:

CRITERIA	RESULT
Does the pipeline for line identification take road images from a video as input and return an annotated video stream as output?	Yes, it takes each frame of the video and does the processing.
Has a pipeline been implemented that uses the helper functions and / or other code to roughly identify the left and right lane lines with either line segments or solid lines?	Yes, a pipeline is implemented as discussed above and the output can be found at the location as discussed above.

CRITERIA	RESULT
Have detected line segments been filtered / averaged / extrapolated to map out the full extent of the left and right lane boundaries?	Yes, the line segments are improvised as discussed above and the output can be found at the location as discussed above.

There are two variables used in the project

1) RAW in function draw_lines() – Set this True if you need the lanes to be highlighted by line segments rather than a single line
2) CHALLENGE_VIDEO in function process_image() - Set this True if the challenge.mp4 needs to be processed. This will change the Region of Interest for challenge video. But the output is not as proper as the other videos. Tuning needs to be done. Making this True and testing the other videos might have slight disturbance in the output but not a huge change.

2. POTENTIAL SHORTCOMINGS:

The following are the shortcomings of this pipeline:

1) The pipeline can not detect horizontal and vertical lines. Unfortunately, if there are horizontal or vertical lane lines, the algorithm fails.
2) If there is some error in processing and unfortunately Hough transform does not return any list of lines, the algorithm fails.
3) The algorithm is very specific to region of interest. A proper region of interest leads to more accuracy. An example of an improper region of interest would produce an output like below,

4) If the algorithm fails anywhere it returns the same original image.

3. POSSIBLE IMPROVEMENTS:

The following are the possible improvements for the above mentioned shortcomings respectively,

1) The camera should be oriented in such a way, that the lane lines should be not either horizontal or vertical.
2) The canny edge, Hough transform parameters should be adaptively tuned if there are no lines detected for say 3 trials. Even in all the trials, the algorithm is not able to detect, then we can consider that there are no lines.
3) The region of interest should be adaptive than a fixed region.
4) We should store the latest results so that if the algorithm fails anywhere, if it is appropriate, we can use the stored latest results instead of sending the same original image.