Project Writeup

For this writeup, I will use solidWhiteRight.jpg image as input. This project consists of five steps.

#1 GrayScale Conversion

Input images in RGB format are converted into grayscale images in this step. In grayscale representation, value of each pixel is a single sample, that is, it carries only intensity information. Intensity of pixels would be used in later steps for edges detection.

Figure 2:Input Image

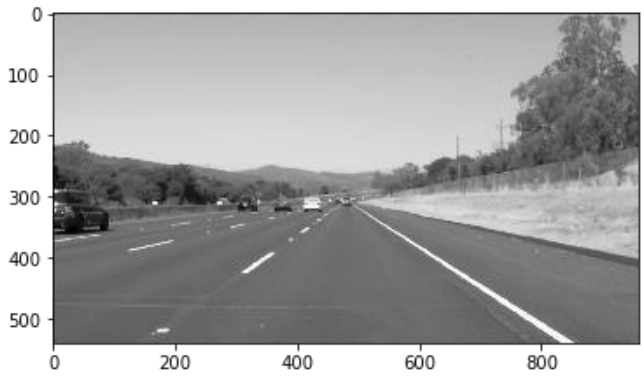

Figure 1: Grayscale Image

#2 Gaussian Blur and Canny Edge detection algorithm

Looking at above grayscale image, Pixel intensity changes rapidly across an edge. This property will be used for lane detection in canny edge algorithm. First, to eliminate image noise and to smoothen edges, Gaussian Blur function is applied on the image. This function takes **kernel_size** as input. A larger value of **kernel_size** parameter would imply higher degree of blurring. **kernel_size** of 5 is used in Gaussian Blur function.

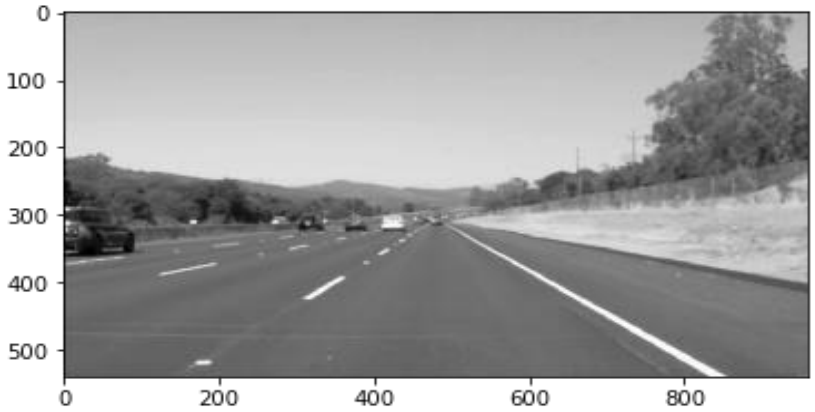

Figure 3 Output of Gaussian Blur function

Next, Canny Edge algorithm is applied to detect the lanes. It is implemented via OpenCV routine (cv2.Canny). cv2.Canny takes two threshold values (upper threshold and lower threshold).

From, OpenCV documentation, the double thresholds are used as follows:

a) If a pixel gradient is higher than the upper threshold, the pixel is accepted as an edge

b) If a pixel gradient value is below the lower threshold, then it is rejected.

c) If the pixel gradient is between the two thresholds, then it will be accepted only if it is connected to a pixel that is above the upper threshold.

d) Canny recommended a upper: lower ratio between 2:1 and 3:1.

After few rounds of trial and error, (low threshold = 60, high_threshold = 150) yielded good results.

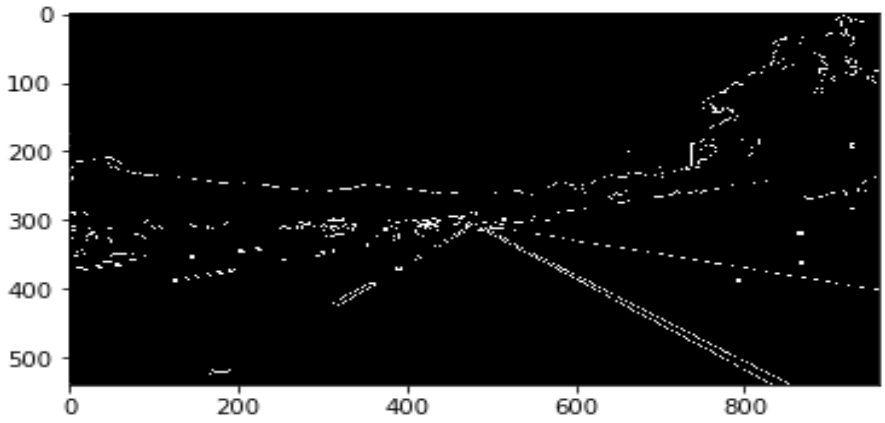

Figure 4: Output of Canny Edge Detection

#3 Region of interest selection

In above image, it can be observed that lot of edges are detected by Canny Edge algorithm. Our algorithm is only interested in detecting edges related to lanes and remaining unwanted edges are masked out in the image. This is done via helper function **region_of_interest (img, vertices).** A four sided polygon region with coordinates ((150,imshape[0]),(420,310), (490,310), (imshape[1],imshape[0])) is defined as region of interest. Below is the output of **region_of_interest()** function

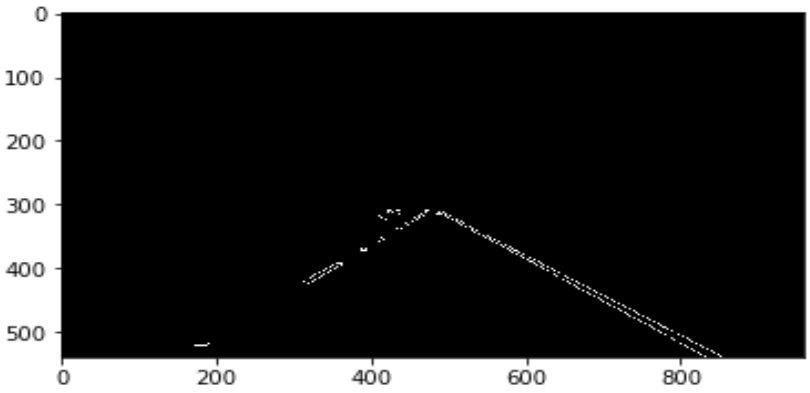

Figure 5: Output of region of interest function

#4 Hough Transform

Hough Transform is applied on above image to find the lane endpoints. **cv2.HoughLinesP** function is used. This function takes several parameters which must be tweaked to obtain best results. Below are the details of parameters along with their values used in this project.

```
 rho = 2 # distance resolution in pixels of the Hough grid\n",
theta = np.pi/180 # angular resolution in radians of the Hough grid\n",
threshold = 10     # minimum number of votes (intersections in Hough grid cell)\n",
min_line_len = 40 #minimum number of pixels making up a line\n",.
max_line_gap = 20    # maximum gap in pixels between connectable line segments\n",
```

This function returns endpoints on all the lanes detected. In next steps, these lanes would be categorized as left and right lines based on the slope.

#5 Right/left Lane categorization and complete lane generation.

Hough transform would return a list of lines with each line represented by a tuple of end points

In our image, top most left corner coordinates are chosen as (0,0). We would be using slope information to categorize the line into left and right lines. Slope of a line is defined as (y2-y1)/(x2-x1). A left lane would have negative slope and right lane would have positive slope. All lines returned by Hough transform would be assigned to either left list or right line list based on slope sign.

To plot the left and right line, we need to compute the average slope and intercept. These are calculated using **weighted average method. (calculate_wavg_slope_intercept).** In weighted average method, slope of longer length line would have higher weightage than slope of smaller length line.

```
def calculate_wavg_slope_intercept(lines):\n",

 # calculate slopes,intercepts and length for all lines.
 # Categorize the also categorize left and right line
 left_slopes    = []
 left_intercepts = []
 left_lengths  = []

 right_slopes    = []
 right_intercepts = []
 right_lengths  = []

 for line in lines:
     for x1, y1, x2, y2 in line:
         if x2==x1:
             continue # it is a vertical line
         slope = (y2-y1)/(x2-x1)
         intercept = y2 - slope*x2
         length = np.sqrt((y2-y1)**2+(x2-x1)**2)
         if slope < 0: # It is a left line
             left_slopes.append((slope))
             left_intercepts.append((intercept))
             left_lengths.append((length))
         else:
             right_slopes.append((slope))
             right_intercepts.append((intercept))
             right_lengths.append((length))
```

```
    # compute weighted average
    left_wavg_slope = np.dot(left_lengths,  left_slopes)/ /np.sum(left_lengths)
    left_wavg_intercept = np.dot(left_lengths,  left_intercepts) /np.sum(left_lengths)
    right_wavg_slope  = np.dot(right_lengths,  right_slopes) /np.sum(right_lengths)
    right_wavg_intercept=np.dot(right_lengths,right_intercepts) /np.sum(right_lengths)


    return left_wavg_slope,left_wavg_intercept,right_wavg_slope,right_wavg_intercept
```

After computing the average slope & intercepts, complete left and right lines are plotted using the helper functions. **(create_lane_lines and draw_lane ())** . In **create_lane_lines** function, y coordinates are chosen to be at **0, 0.6*(image.shape[0])**. Next, x coordinates are computed using slopes, intercepts and y co-ordinates. Finally, left and right lines are plotted on the input image using the function draw_lines().

```
def create_lane_lines(image, lines):

    # Get the average slope of left and rights lane
    left_slope,left_intercept,right_slope,right_intercept = calculate_wavg_slope_intercept(lines)

    # Choose the end points.
    y1 = image.shape[0] # bottom of the image
    y2 = int(y1*0.60)   # draw till 2/3 of the screen

    left_x1 = int((y1 - left_intercept)/left_slope)
    left_x2 = int((y2 - left_intercept)/left_slope)

    right_x1 = int((y1 - right_intercept)/right_slope)
    right_x2 = int((y2 - right_intercept)/right_slope)

    return ((left_x1, y1), (left_x2, y2)), ((right_x1, y1), (right_x2, y2))
```

Below is the final output where detected lanes are overlayed on top of the input image.

Figure 6: Output image with detected lanes

Reflections

This project provided a good introduction into computer vision algorithms and toolchain. I could try few variants of lane detection algorithm and evaluate the results.

Identify any shortcomings

a) current algorithm might fail for curved roads as slope of both right and left lanes will be of same sign.
b) Image masking is static and it might fail in certain scenarios such as uneven surface roads, bumpy roads etc.

Suggest possible improvements

a) Make image mask selection dynamic, so that it could work in different scenarios
b) Incorporate some machine learning algorithm into left lane/right lane detection routine so that it works for all kinds of roads in all conditions.