Project 1 - Detect Lane Lines

Anurag Atmakuri

The idea of lane detection is a complex problem due to the number of parameters taken into consideration. For example, image brightness, weather, lane markings, and lane consistency to name a few.

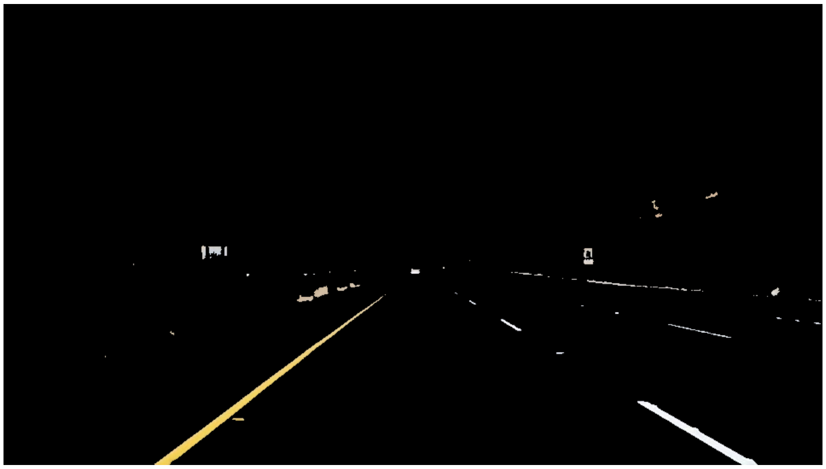
As we learnt in class, I have started out with the basic framework of steps needed to successfully detect basic lane lines as part of Project 1.

The function which is the core of the lane detection algorithm is lanelines(). I have incorporated a slightly different approach here. Assuming that the camera is mounted and has the same field of view at all times. I have worked backwards considering the fact that I know rough quadrilateral which is the area of concern and I have tried to iron out the rest of the image. I will walk through the pipeline briefly below

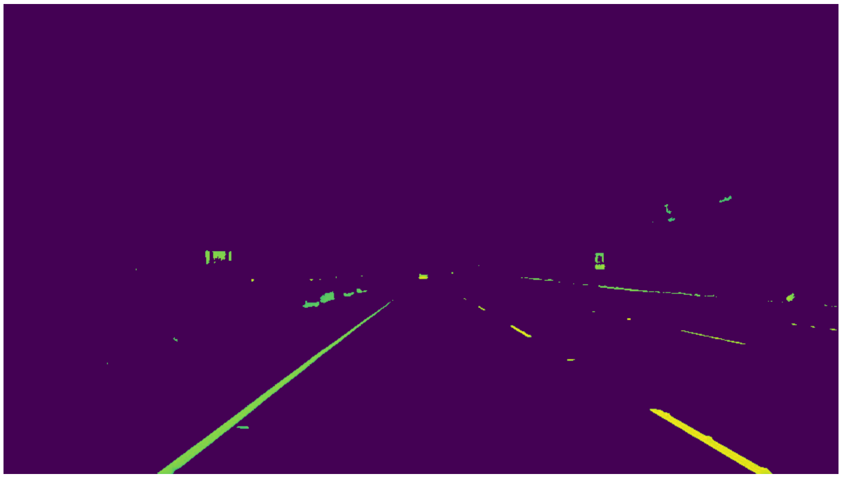
- I have considered the whiteCarLaneSwitch.jpg from the test images for now. This image was selected because it has yellow and white lines.

- First I obtained the yellow and white regions in the image. To do this, I tried two approaches. First approach was to convert the Image from RGB to HSV to have a better understanding of the image in terms of Color intensity. White color was easier to obtain in terms of HSV values by tuning. But yellow ranges over a wide span of HSV and it was time consuming to accurately be able to detect the yellow lines. I tried the second approach by using the RGB image and masking the non-white and non-yellow regions in the image.

```
white_thresholds = (image[:,:,0] < 200) | (image[:,:,1] < 200) | (image[:,:,2] < 200)
thresholds = (image[:,:,0] < 200) | (image[:,:,1] < 170) | (image[:,:,2] < 0)
image[~(~thresholds | ~ white_thresholds)] = [0,0,0]
dst2= cv2.bitwise_and(img_wip,image)
```

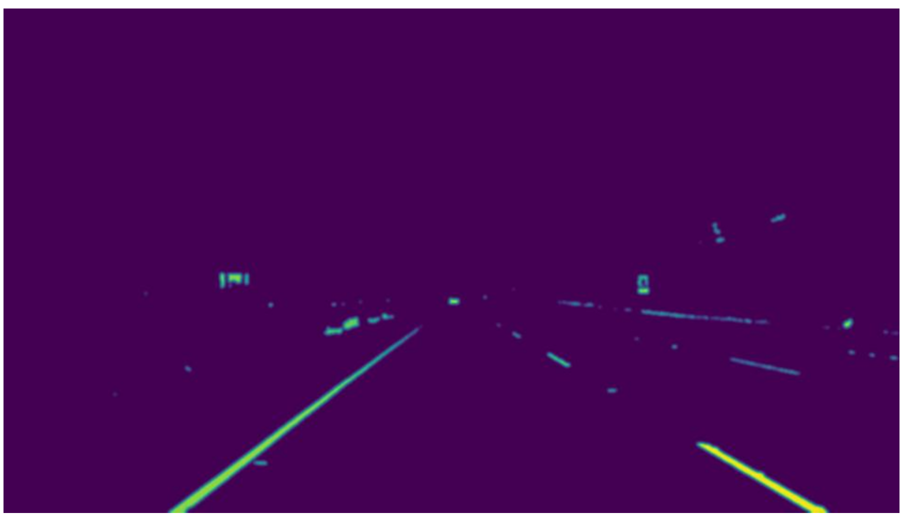


- The approach here is a little different to what one might do traditionally. Since I have highlighted the yellow and white regions of the image and I really am not concerned about the rest of the image, I went ahead and set the rest of the image to RGB [0,0,0]. The below image is RGB2GRAY of the image above.



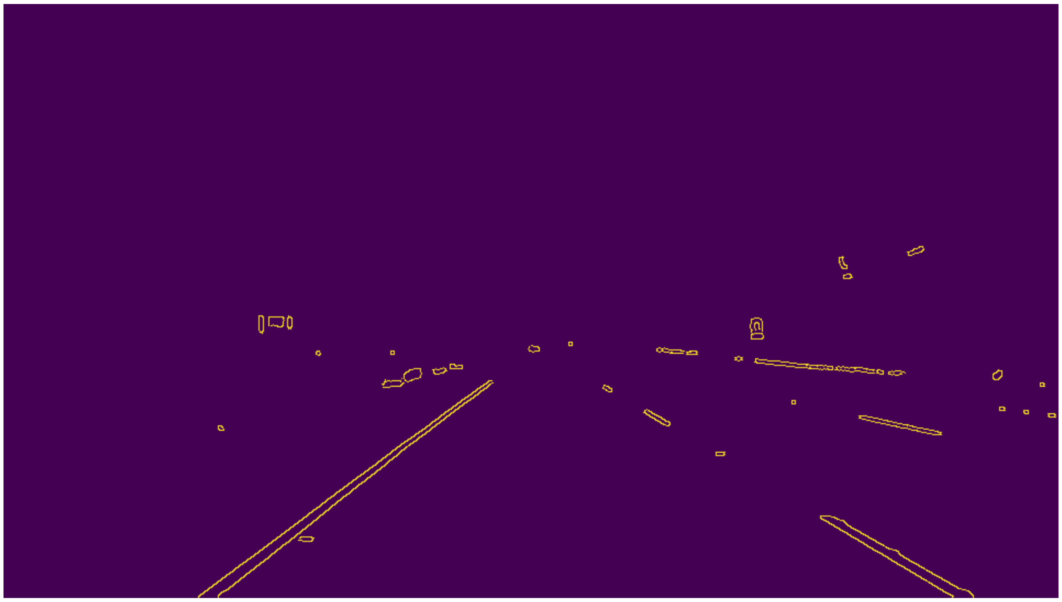
- If I had the full image, I would have gone ahead and used a kernel size 3 or 5 to gradually smoothen the image. But since I wanted to blur our all the noise in the image, I went ahead and used kernel size of 7 and the image is below.

- `gaussblur= gaussian_blur(gray_img,7)`



- Canny Edge Detection was used with a low_threshold to high_threshold ratio of 1:3 with the low_threshold being 50.

```
cannyimg= canny(gaussblur,50,150)
```



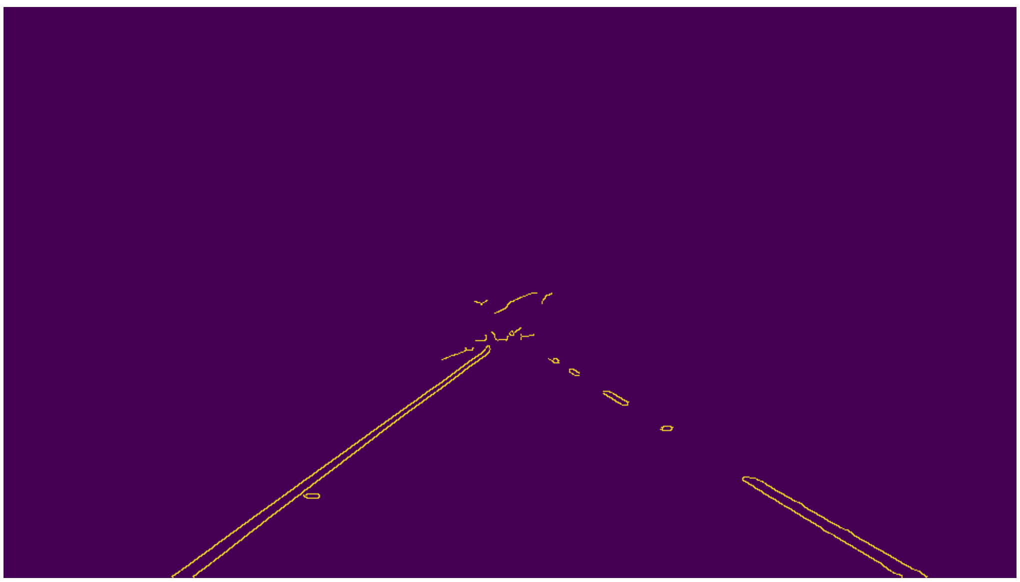
- The output image from Canny Edge Detector was then used in the function called interested_bbox() which masked out the image based on the x and y ratios. The interested bbox considered a default bounding box (polygon) of yratio, xratio_bottom and xratio_top.
- Yratio: It is the value between 0 and 1 and is used to determine the height of image from the image axis that is considered as the top of the quadrilateral.
- Xratio_bottom: Is the % of the bottom of the image that needs to be outside of the region under consideration. For example, if the size in X direction is 960 and xratio_bottom is 0.05, then 0.05*960 = 48.0 pixels from the left of the image and the right of the image (48 pixels and 912pixels are the x-coordinates of the bottom edges of the quadrilateral)
- Xratio_top: Is set to 0.45 as default and can be overridden during the function call.
- The output of the interested_box returns the vertices of the image under consideration.
- These vertices are input to the function called the region of interest, which outputs an image by masking out all the regions that fall outside of this quadrilateral

```
 vertices = interested_bbox(cannyimg,x_size,y_size,xratio_bottom="",xratio
_top=0.45,yratio=0.50)
regionimg= region_of_interest(cannyimg, vertices)


##Interested_bbox
def interested_bbox(img,xsize,ysize,xratio_bottom,xratio_top,yratio):
       """
       By default, Percentage of X at the bottom is 5 and top is 45 and th
ere is an option for user to override to tune.
```

```
        Percentage of Y is 55 from the top of the image. If image dimension
s ar 540 , then the default Y will be  297
        """
        if xratio_bottom =="":
                xratio_bottom = 2./100
        if xratio_top =="":
                xratio_top = 45./100
        if yratio =="":
                yratio= 55./100


        top_left = (xratio_top*xsize,yratio*ysize)
        top_right = (xsize*(1 - xratio_top),yratio*ysize)
        bottom_right= (xsize*(1 - xratio_bottom), ysize)
        bottom_left= (xratio_bottom*xsize ,ysize)
        vertices = np.array([[bottom_left, bottom_right,top_left,top_right]
],dtype=np.int32)
        return vertices
```



- HoughLine function parameters after tuning recursively have been set at rho=1,theta=np.pi/180, threshold=20,min_line_length=15 and max_line_gap=40
- Houghlines function returns the coordinates of the line segments and these are input parameters to draw_lines() function.

```
rho_h = 1
theta_h = np.pi/180
threshold_h = 20     # minimum number of votes (intersections in Hou
gh grid cell) - 10
min_line_length_h = 15 #minimum number of pixels making up a line -
5
```

```
max_line_gap_h = 40    # maximum gap in pixels between connectable li
ne segments -40
#mpimg.imsave("test_imgregion.jpg",regionimg)
houghimg= hough_lines(regionimg, rho_h, theta_h, threshold_h, min_li
ne_length_h, max_line_gap_h)
```

- Draw_lines() Implementation:
  - ✓ The assumption is that the camera is mounted at a fixed point and the field of view is unaltered. Based on this assumption, each line in the lines from HoughLines will either have a positive slope or a negative slope.
  - ✓ Draw_lines function uses the slope equation (y2-y1)/(x2-x1) to calculate the slope of every line and append the X and Y coordinates of the line to the list of positive_slope_lines and negative_slope_lines. There could be noise that could cause lines to have different slopes to the lines under consideration. Since the test images and the videos have predominantly been straight lines, I have tuned the threshold of the slope with a positive slope of 0.40 and a negative slope of <-0.60 to filter out all the noise between this threshold. The values 0.40 and -0.60 are chosen because they are around 45degrees angles .

```
def draw_lines(img, lines, color=[255, 0, 0], thickness=10):

    if(len(lines)==0):
        return np.zeros ((2,2))

    pos_slope = []
    neg_slope = []
    "Pos_slope and neg_slop hold the slope values for future use to averag
e the slope and get a better line"

    pos_slope_lines_x1_array=[]
    pos_slope_lines_y1_array=[]
    neg_slope_lines_x1_array=[]
    neg_slope_lines_y1_array=[]
    "The *_slope_lines_x/y*_arrays are used to hold the x1,y1,x2,y2 pairs
for all the line segments from HoughLinesP"

    for line in lines:
        for x1,y1,x2,y2 in line:
                slope = ((y1-y2)/(x1-x2))
                if slope >= 0.40 :
                        pos_slope_lines_x1_array.append(x1)
                        pos_slope_lines_x1_array.append(x2)
                        pos_slope_lines_y1_array.append(y1)
                        pos_slope_lines_y1_array.append(y2)
                        pos_slope.append(slope)
                elif slope <= -0.60:
```

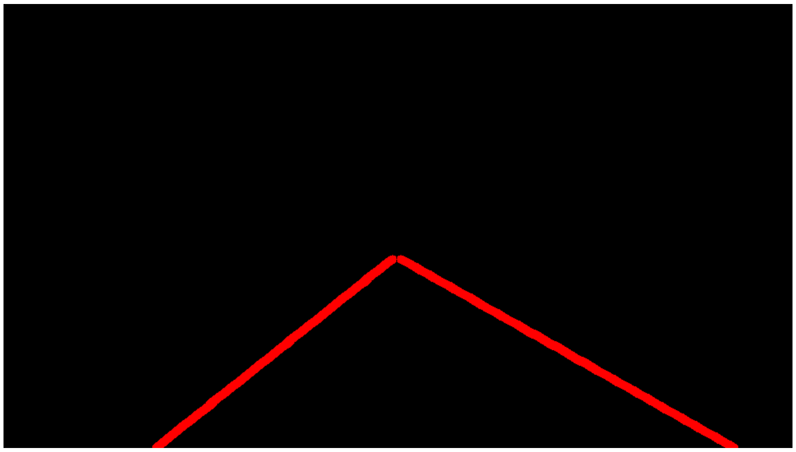
```
                        neg_slope_lines_x1_array.append(x1)
                        neg_slope_lines_x1_array.append(x2)
                        neg_slope_lines_y1_array.append(y1)
                        neg_slope_lines_y1_array.append(y2)
                        neg_slope.append(slope)
            #cv2.line(img, (x1, y1), (x2, y2), color, thickness)
    #print(pos_slope)
    #print(pos_slope_lines_x1_array)
    #print(pos_slope_lines_y1_array)
    [pos_x,pos_y] = extrapolate(pos_slope_lines_x1_array,pos_slope_lines_y
1_array)
    [neg_x,neg_y] = extrapolate(neg_slope_lines_x1_array,neg_slope_lines_y
1_array)
    pos_line = np.array(list(zip(pos_x,pos_y)),np.int32)
    neg_line = np.array(list(zip(neg_x,neg_y)),np.int32)
    for i in range(len(pos_line)-1):
        cv2.line(img,(pos_line[i][0],pos_line[i][1]),(pos_line[i+1][0],pos_
line[i+1][1]),color,thickness)
    for j in range(len(neg_line)-1):
        cv2.line(img,(neg_line[j][0],neg_line[j][1]),(neg_line[j+1][0],neg_
line[j+1][1]),color,thickness)
    #for line in lines:
    #    for x1,y1,x2,y2 in line:
    #        cv2.line(img, (x1, y1), (x2, y2), color, thickness)
```

- ✓ The output line X and Y coordinates are passed to the extrapolate function that generates a slope and intercept by extrapolating all the points on the projected line. The slope and intercept are then inputs to line_region function that generates a line based on this slope. However, since we don't want the lines to be extrapolated all through the image, the line_region function takes the Y range from 300 to 540 and generates an array of pixels within the range. The function then uses the line equation formed by the slope and intercept to obtain the X values that fall on this line. Line_region() outouts the X and Y coordinates that are part of the actual lane line and within the quadrilateral region of interest. These coordinates are then used to draw a line in red color and the output is the image below.

```
def extrapolate(slope_lines_x1_array,slope_lines_y1_array):
        """
        Extrapolate function takes the x and y coordinates from HoughLinesP
        Calculates the Slope and Intercept values by fitting a line.
        Extrapolates and returns all possible (X,Y) pairs on the line withi
n region of interest by calling lines_region

        Called by: draw_lines
        Calls: lines_region
```

```
    Returns: All possible (X,Y) pairs on the line within the region
    """
    [m,c]= np.polyfit(slope_lines_y1_array,slope_lines_x1_array,1)
    #print(m)
    #print(c)
    [x_coordinates,y_coordinates] = lines_region(m,c)
    return (x_coordinates,y_coordinates)
```



- Weighted Image function is then used to superimpose the extrapolated lines on to the main image as shown below.

Potential Shortcomings in the current pipeline:

- The pipeline assumed that Camera is mounted at a fixed point and the field of view remains more-or-less the same throughout. Which means that if the car is changing lanes, the slope of lanes changes with respect to the image coordinates and the draw_lines function needs refining as the range of slopes are hard coded.
- The draw lines function also uses a polyfit function to fit the given lines (from HoughLinesP) onto a straight line. Though this might work on straight roads, this will not work effectively on curvy roads.
- Bounding box vertices determination should be made robust for all possible values of Xratio_top, xratio_bottom and yratio as there are few areas we cannot neglect specially because the camera has a default field of view and we cannot choose to ignore. The code might fail if the ratios of X and Y for quadrilateral vertices are stretched beyond a certain range.

Improvements for future:

- Current algorithm uses the RGB values of White and Yellow to mask all other RGB values within the image. Though this works accurately for the given test images, realisitically, there could be different intensities of yellow and white under different conditions. HSV values for White and Yellow are more intensity related values and accurate compared to the RGB. One improvement could be to detect Yellow and White lines from HSV image.
- Polyfit function in the current algorithm returns a linear equation (straight line) that “tries” to satisfy all the lines coordinates from HoughLinesP. This works on straight lines (as shown in the test images and the videos submitted). However, when the road is curved line equations are not sufficient as the curved lane lines are non linear. The line_region function needs an update to take into consideration the curvy nature of lane lines. Once this change has been made, the extrapolate function also needs to be changed wrt the line_region function as we need more than slope and intercept.
- Stability: The lane lines prediction algorithm must be stabilized by keeping a buffer of most recent slopes for the left lane and the right lane lines and there by ironing out the unnecessary slope lines which are extremely off from the values in the buffer. This way we eliminate taking into consideration the spurious line values from HoughLineP