University of Maryland

ENPM 673: Perception for Autonomous Robots

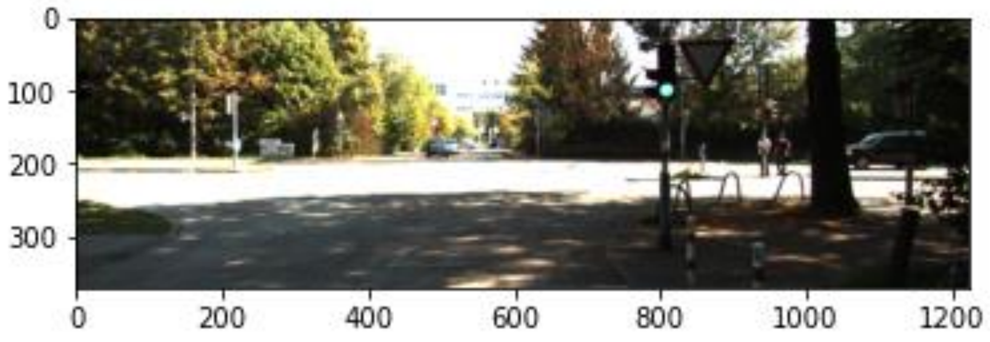
Project 2

Jai Sharma

UID: 118160016

Question 1: Histogram Equalization

We are given a set of 25 frames . We enhance the contrast of each image and improve its visual appearance using Histogram Equalization methods. I have carried out standard histogram equalization and Adaptive Histogram Equalization on the given frames without using any inbuilt functions. A sample image is shown below:

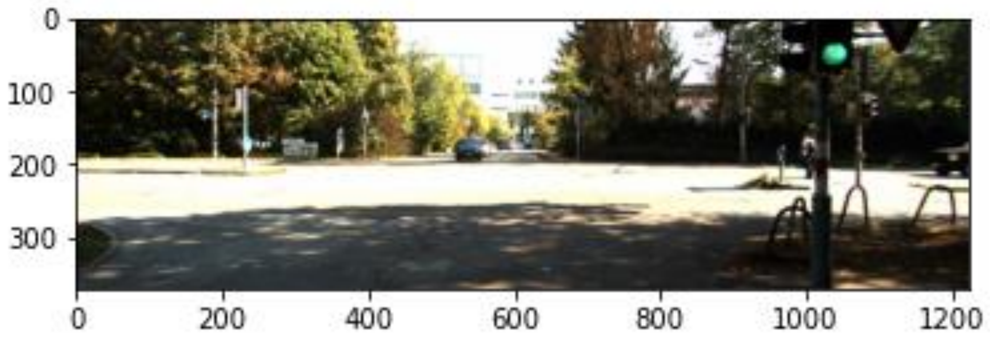


Each frame can be represented using a histogram that plots the pixel count against the intensity. The goal of histogram equalization is to distribute the histogram of the image to use the entire dynamic range. We use cumulative distribution function to get the pixels for which the intensity is less than a set threshold value.

The processing of histogram equalization relies on the use of the cumulative distribution function (cdf). It achieves so by calculating the fraction of pixels with an intensity equal to or less than a specific value.

$$cdf(x) = \sum_{k=-\infty}^{x} P(k)$$

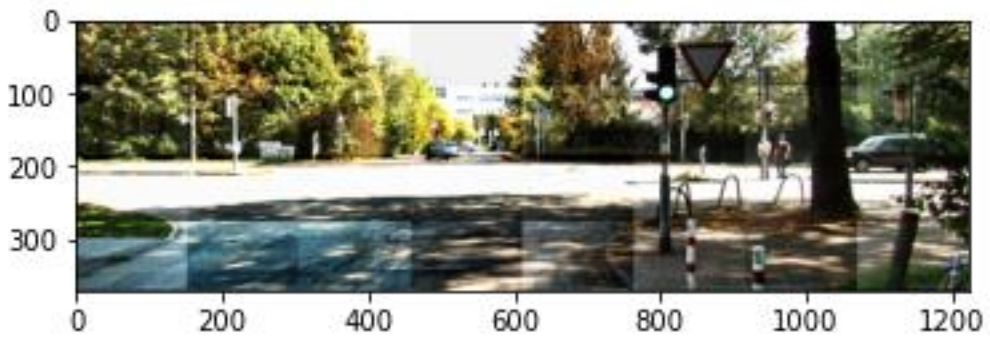
The image below is a result of standard Histogram Equalization. You can notice that the brighter shades are slightly darkened. You can also see parts of the building right in the middle that you cannot see in the original image.



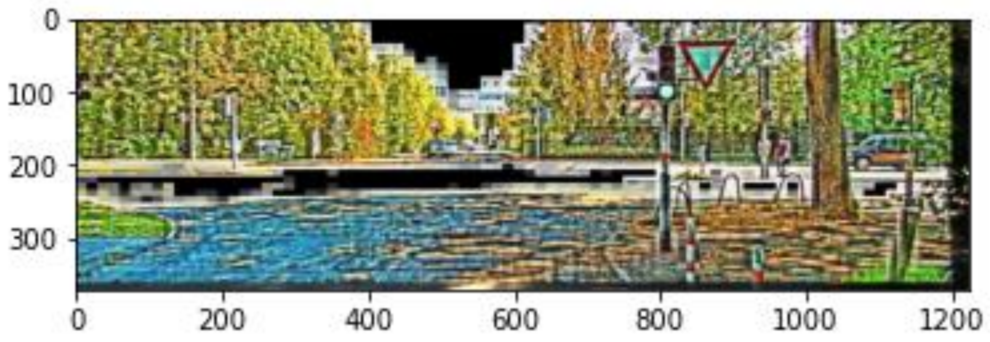
Overall Adaptive Histogram Equalization does a much better job at improving the image quality than the standard method. But it is important to pick the right parameters. I tried the Adaptive Histogram

Equalization method with several grid sizes. Some outputs with smaller grid sizes gives an extremely processed image. On the other end, larger grid sizes give under processed image. Both outputs are shown below:

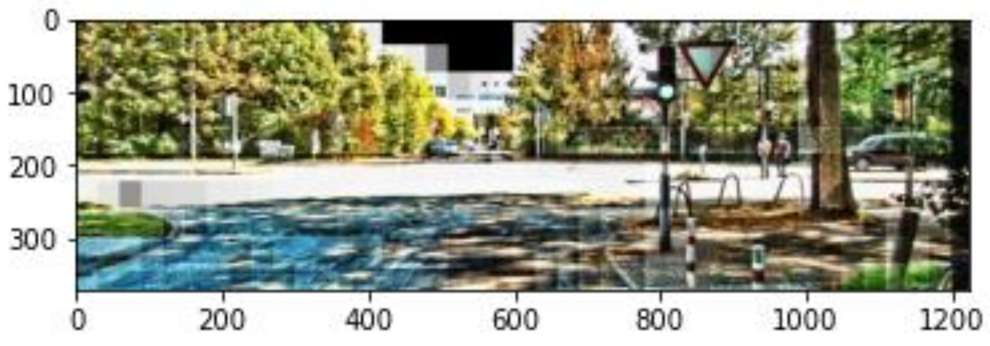
4 rows x 8 columns:



40 rows x 80 columns:



The best result came for the grid size of 10 rows and 40 columns. The method seems to have uniformly improved all areas of the image. While the frame does become pixelated and has a few pixel gaps, the rest of the image is enhanced with a reasonable amount of detail.

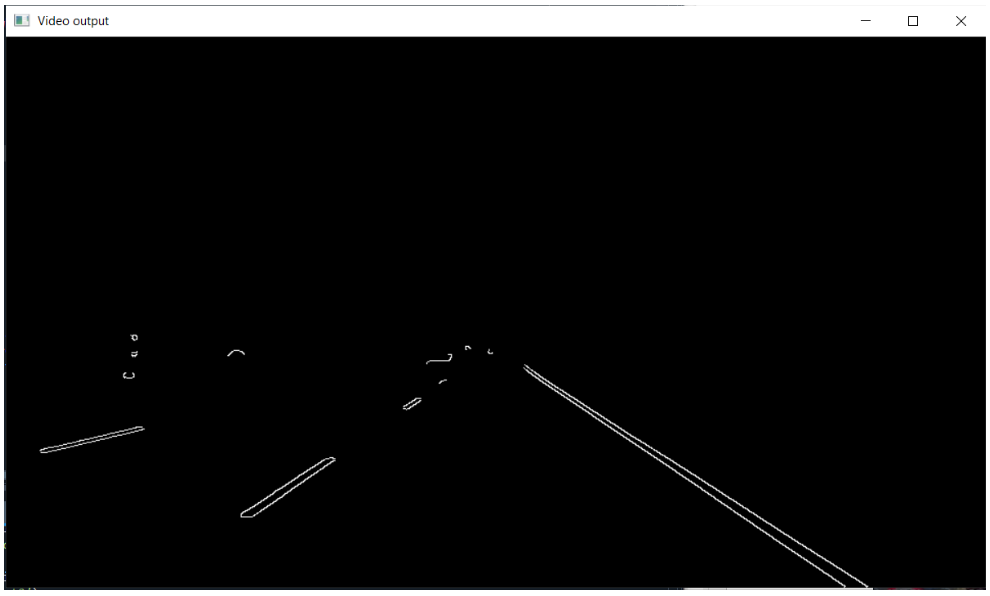


Question 2: Straight Line Detection

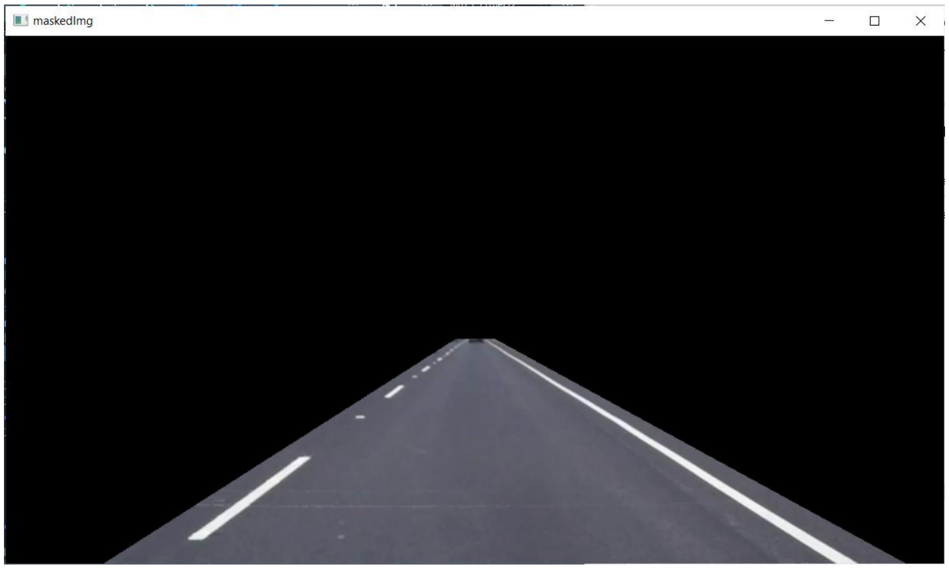
In this problem, we are tasked with detecting driving lanes for a given video sequence. I have segregated the solid lines from the dashed lines using a slope-based logic. This logic is used to ultimately classify dashed lines as Red and straight lines as Green. The pipeline for accomplishing this task is shown below.

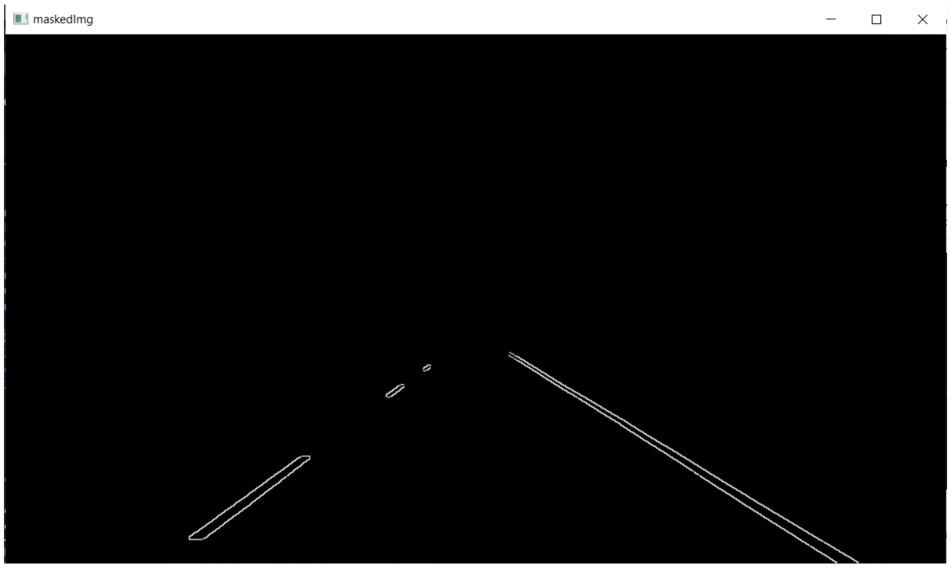
CannyEdge Image → Region of Interest Mask → Hough Transform→ lane segregating logic

To get a Canny Edge image, I preprocessed the still frame by converting it to greyscale, blurring it, and then applying the cv2.Canny function. The threshold values of 250 and 350 worked well for all frames. One of the resultant still frame is attached below:



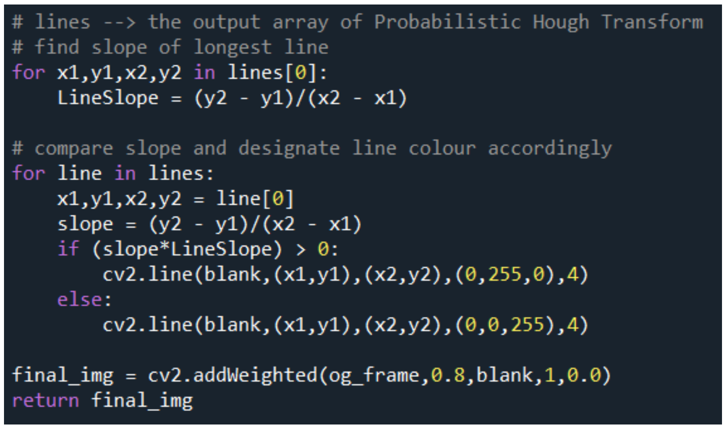
On the above image, I added a mask to only perform Hough Transform on the area of interest. Since the size of the image is known, the Vertices of Interest can be estimated. Next, cv2.fillPoly function can be used to create the mask. The mask I created looks as follows:





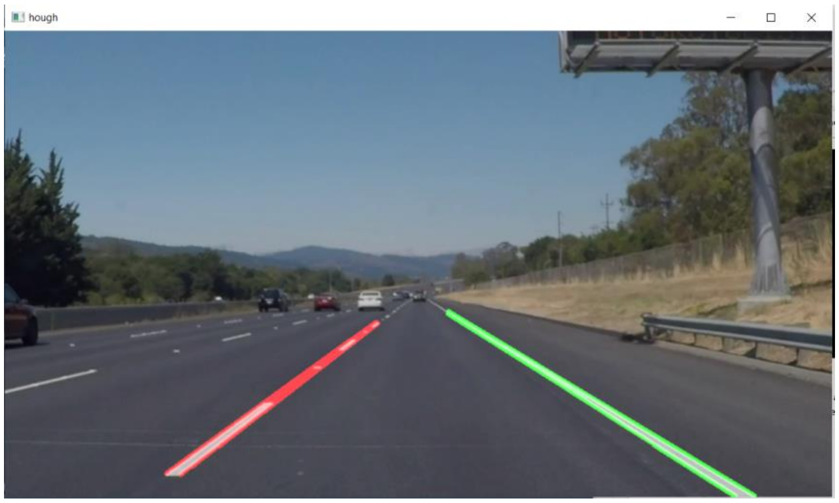
On this image, we perform Hough's Transform and retrieve all lines on a single frame of the video. Within the getHough function itself, I have integrated the logic that segregates dashed lines from straight lines.

I realized that the output of cv2.HoughLinesP is an array of all lines in the image and, that the lines are sorted in order of length. Thus, the first [x1 y1 x2 y2] element of the array relates to the longest line detected, i.e. always the solid lane. Another observation I made was that the two lanes are converging, the signs of their slopes are different. Hence, I identify the sign of the longest slope and compare the signs of the slopes of all lines in the image. In the above frame, as per the cv2 grid, the straight line has a positive slope and all the lines in that trajectory must have positive slope. Hence, they are all drawn as Green. The code that exercises this logic is shown below.



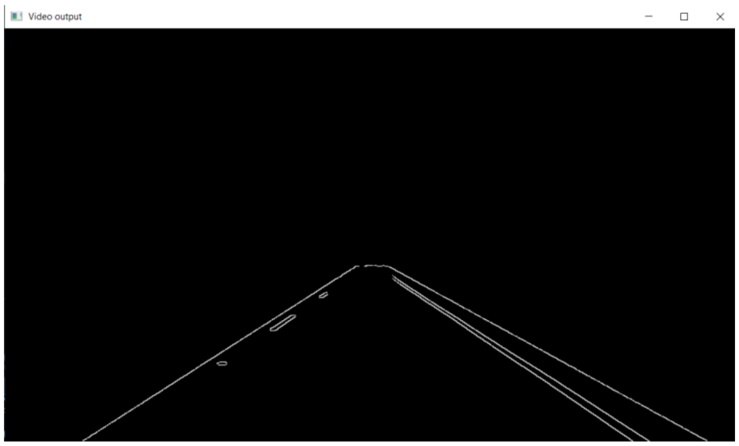
```python
# lines --> the output array of Probabilistic Hough Transform
# find slope of longest line
for x1,y1,x2,y2 in lines[0]:
    LineSlope = (y2 - y1)/(x2 - x1)

# compare slope and designate line colour accordingly
for line in lines:
    x1,y1,x2,y2 = line[0]
    slope = (y2 - y1)/(x2 - x1)
    if (slope*LineSlope) > 0:
        cv2.line(blank,(x1,y1),(x2,y2),(0,255,0),4)
    else:
        cv2.line(blank,(x1,y1),(x2,y2),(0,0,255),4)

final_img = cv2.addWeighted(og_frame,0.8,blank,1,0.0)
return final_img
```

One of the frames displaying the final outcome is shown below:

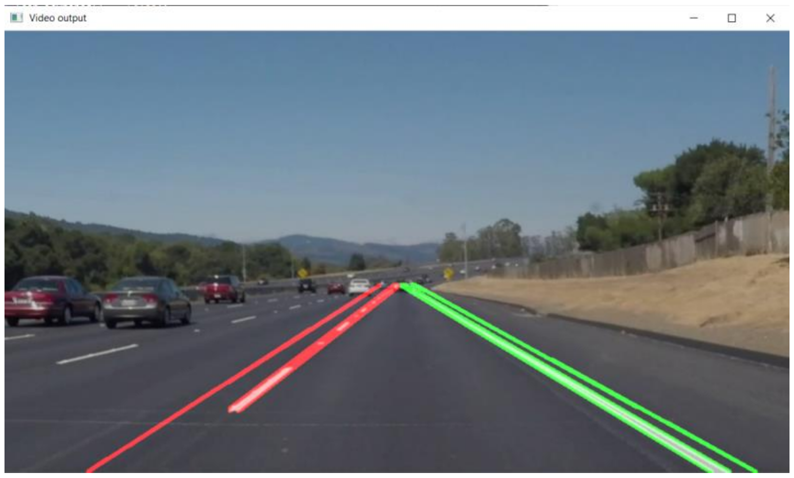


The video link to the resultant video is attached at the bottom of the report.

I realized that the sequence of integrating the Mask matters. In my final solution, I got the CannyEdge image and then I added a mask on it. If I do the opposite, the resultant frame looks like this:



Although the lanes are still getting detected, the borders of the mask also get detected as edges. This leads to problems as the Hough function also captures those lines and displays them as lanes. But as can be seen below, no lanes exist there.

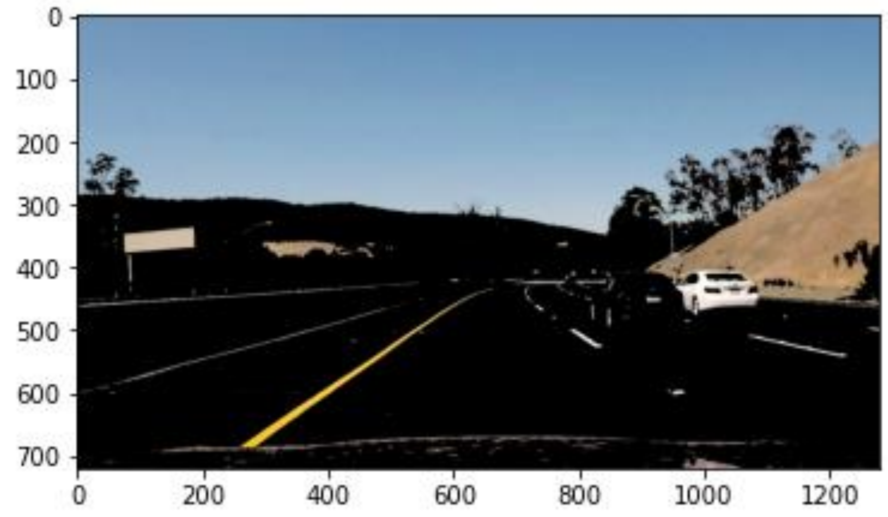


Question 3: Predict Turns

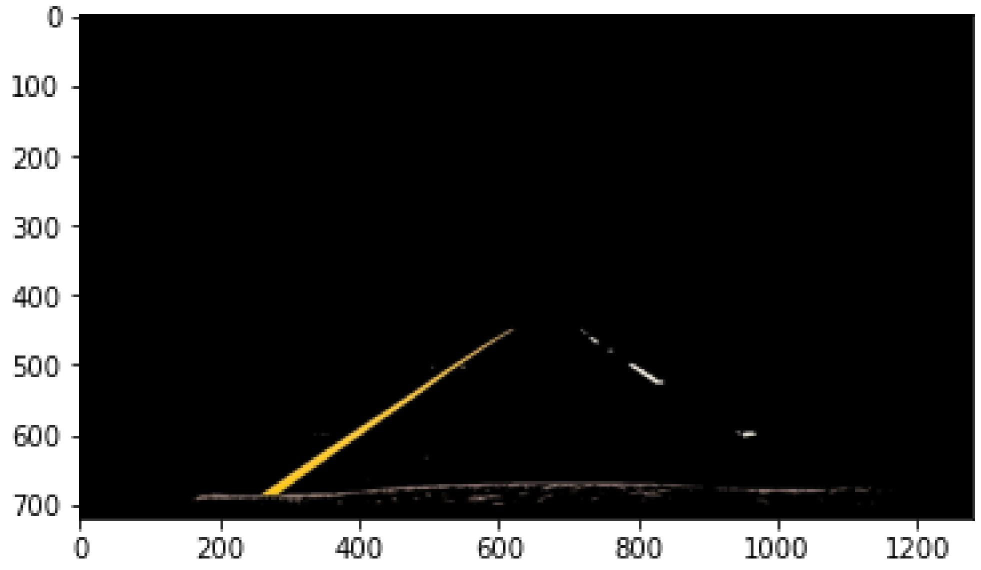
For a new video set, we are tasked with tracking the curvature of the road and predicting right turns and left turns. The initial pipeline is similar to what we used in Question 2. Functions like Canny Edge and masking. General Pipeline:

1. Preprocess the image with masks to locate the yellow and white mask.
2. Apply Canny Edge Detection method.
3. Use Hough Transform to detect all lines on the image.
4. Apply a perspective transform to attain a "birds-eye view".
5. Detect lane pixels on warped image and fit it to locate the lanes.
6. Us np.polyfit() to get equation of line of the two lanes.
7. Implement an algorithm to calculate radius of curvature.
8. Unwarp the image and display curvature radius and predicted direction on playing video.

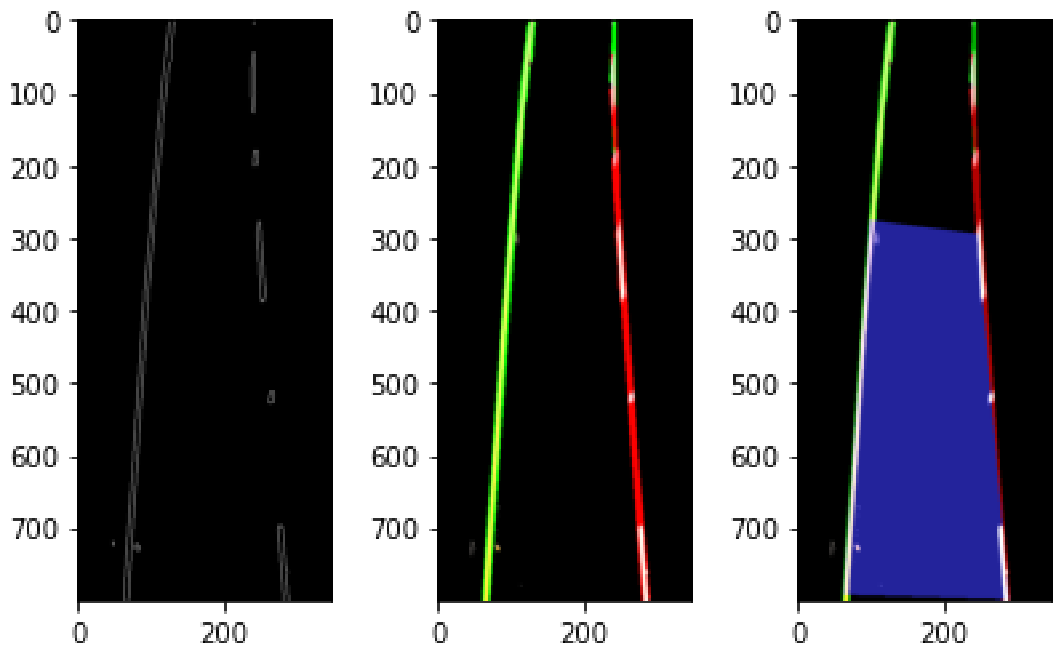
The biggest challenge for the given footage is to do with the changing brightness in the frame. There is a patch where, strong sunlight is hitting the road and detecting lanes becomes incredibly tough. To combat this, I used two different methods to isolate the two lanes. I made two masks, one for white, one for yellow and used the cv2.bitwise function to get a reasonably processed image. He output image is as follows:



After putting a mask to crop out the interested region, we get:



Next, I used Homography to warp the above image to give us a bird'eye view. Below, you can see the warped image after applying (a) Canny Edge, (b) Detected Lanes (c) Detected Patch of Road

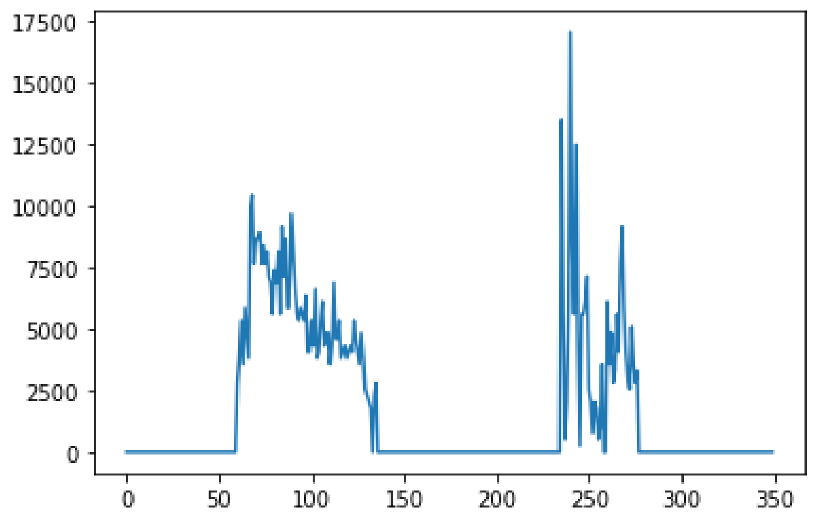


The lines are created using Hough Lines functions. I will briefly go over the theory behind Hough Transform. A line is represented as y = mx + c in the Cartesian Coordinate System. However, it can't represent vertical lines as the slope with go to Infinity. Instead, we use polar coordinate to express the same lines:

$$d = x * \cos(\theta) + y * \sin(\theta)$$
$$x = d * \cos(\theta)$$
$$y = d * \sin(\theta)$$

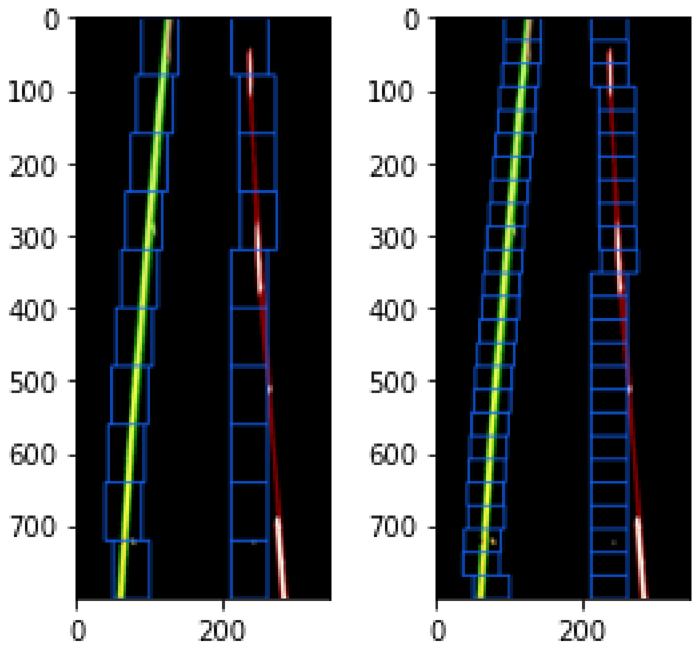
This essentially converts a line into a point in hough space, represented as (d, θ). So for each detected point on each detected edge on the image, and for θ varying from 0 to 180 degrees, we find a corresponding distance d and build the line equations. The output of hough transformation is a list of end coordinates set, each corresponding to single line. The set contains coordinates sin the form [x1, y1, x2, y2]

I have also created a histogram for the entire image. For the first frame, the histogram is shown below.



The idea behind the approach is that I can compare how the camera direction is changing with respect to the center of the lane. Based on threshold ranges, I can predict the lane to be turning left, right or center. I also make different lists for left and right points here by checking if they lie in the first half or the second half of the histogram picture.

I also employed sliding window search technique that starts from the bottom and iteratively scans all the way to the top of the image, adding detected pixels to a list. After I detected the pixels belonging to each lane line, I fit a polynomial through the points. This gave a good approximation of where the lane is.



There are two major problems with the algorithm that I could not resolve yet. The algorithm cannot detect lanes when the car enters a sunny patch in the video. I was able to prevent the code from crashing in such instances but better masking techniques, the lane detection technique can become more robust. The second issue is with the calculation of the radius of curvature. My approach was to use polynomial fit on the two lanes, get curve equations. The radius of curvature of the lane is then just the average of the two radii. Currently the value is extremely overestimating the real value. There must be a bug in that part of the code that I could potentially address and fix in the future.

For both Problem 2 and Problem 3, there are few limitations to the applied algorithm. The biggest factor is the lighting conditions in the video. The color thresholding that I apply to build masks works for the specific given video but it will need to be tuned for any new video with different lighting conditions.

Also, the mask I have made to single out the Area of Interest only works for the frames shown in the video. If the video perhaps shows lane changing maneuver or uses a camera of different frame size, the vertices defined for the mentioned task will not give the right output.

The algorithm is simply not robust enough to adapt to new variables or different variables. It is best to tune the code specifically for the use case in consideration.

Video Links

The solution videos are saved on the following Google Drive Link:

https://drive.google.com/drive/folders/1T96PgZn_4gbJAW50hzEo7YZnGQKx-IC9?usp=sharing