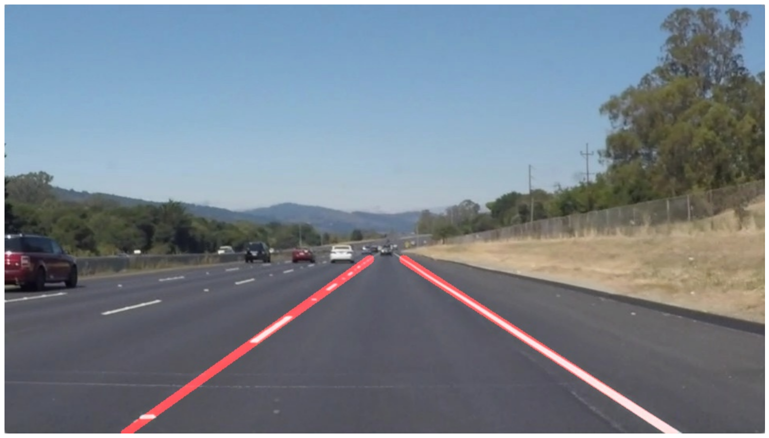
Finding Lane Lines on the Road

One of the most basic aspects of driving a car (safely) is to make sure the car is within a lane. A good human driver does it almost involuntarily by constantly correcting to make sure the vehicle doesn't veer off into another vehicles path. What we humans do visually can be replicated in an autonomous vehicle using a series of Computer Vision techniques. The objective of this implementation and the process pipeline required to achieve this is explained below.

Objective

The objective of this project is to implement a set of functions which combine together to identify lane lines on a road. When presented with a video stream from a front view camera, this implementation should identify the left and right lane markings and highlight them as a thick red line extending from the front of the vehicle into the plane of the image away from it as shown below.



Workspace

For this nanodegree, while working on python based projects, I've decided to use Google Colab as my workspace. The advantages of this approach for me are many

1. Most python libraries required for this nano degree are readily available on Colab and adding new libraries is very easy as well
2. Gives me flexibility of carrying my workspace whereever I want
3. Can easily fire up GPU and TPU supported instances and speed up some of the training process
4. Can mount my google Drive on Colab and have a copy of my files on the cloud
5. Can commit my work to github directly from colab

Lane Detection Pipeline

The input for the lane detection task , as mentioned earlier, is a video stream from the front view camera. The video stream in reality is only a continuous stream of individual image frames. So, our lane detection effectively starts by extracting each frame from a video. Once we have the individual frame, the processing pipeline follows

1. Convert the color image to gray scale image and apply Gaussian Blur

2. Detect edges in the image using Canny Edge Detector
3. Mask and retain only the region of interest from the entire image
4. Apply Hough Transform on the masked edge features to identify lines
5. Identify the left and right lane lines from their slopes
6. From all the detected left and right lanes, get the average left line's slope and intercept and the right line's slope and intercept
7. Use the average slope to draw an extrapolated line in the region of interest
8. Overlay the extrapolated line on the original image

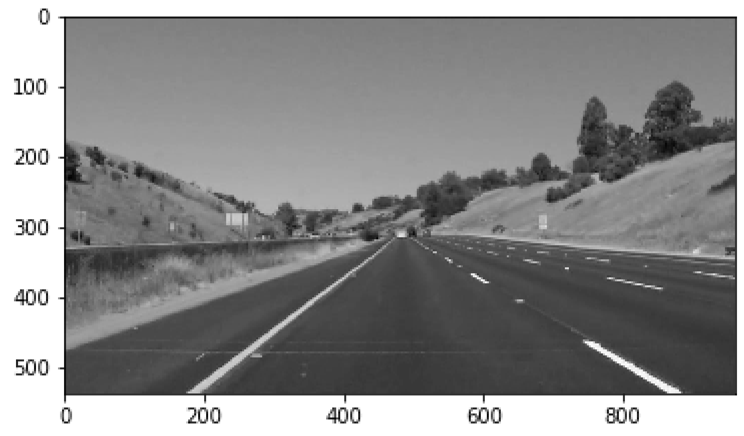
Grayscale and Blurring

The basic idea behind lane line detection is the identification of the edges that constitute the lane. Although edge detection can be done on a RGB image, the effective edge is determined by the channel with the maximum gradient. So, for efficient edge detection, the first task is to convert to **gray** scale image.

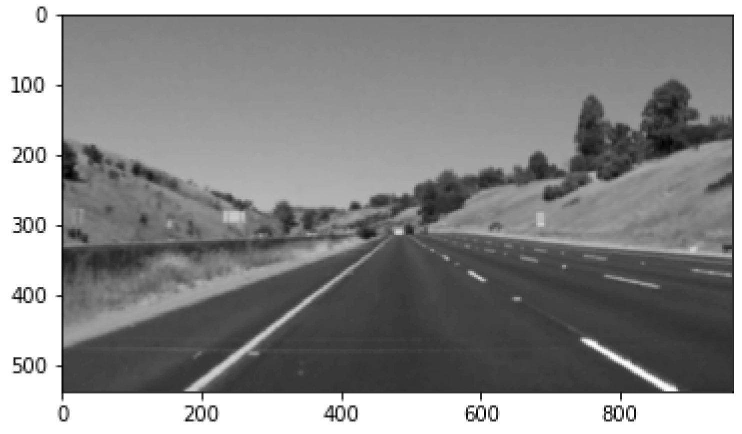
Blurring is another important technique which helps reduce high frequency content in the image and hence reduces noise and helps in better edge extraction. There are several blurring methods available in openCV. The most commonly used method is the Gaussian Blur. In this method, a **M x N** kernel is convolved with the input image and the intensity of each pixel is modified to the weighted average of the intensities of the pixels in the kernel. The Gaussian function used is shown below

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}}$$

here x and y are the distances from the kernel center and σ is the standard deviation of the pixels in the Gaussian kernal



Raw Image



Blurred Image

Canny Edge Detection

Canny edge detection is a technique used to identify gradients of any orientation in an image that helps us extract the structural information in an image. The algorithm internally performs the following four steps

1. Gaussian blurring to smooth the input image
2. Finding intensity gradients G_x and G_y

$$EdgeMagnitude = \sqrt{G_x^2 + G_y^2}$$

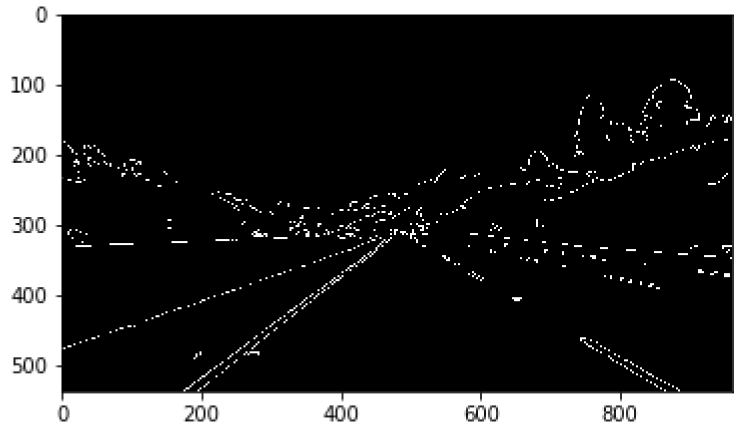
$$EdgeOrientation(\theta) = tan^{-1}(\frac{G_y}{G_x})$$

3. Non Maxima suppression

> The entire image is scanned to check for any unwanted pixels that might not constitute an edge. Every pixel is compared with the pixels in a 3x3 neighborhood. If the magnitude of the central pixel is greater than the pixels in the gradient direction the central pixel is retained or else dropped.

4. Hysteresis Thresholding

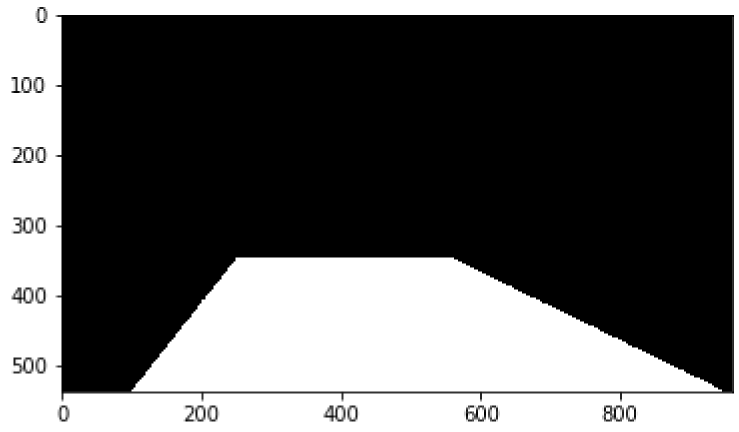
> To further consider a pixel as an edge or not, a threshold is set by a min and a max value. If the pixel lies below the minimum threshold, its ignored. If a pixel intensity is more than the max threshold then it is a sure edge. If a pixel lies between the max and the min value but is connected to the pixel above the threshold then it is considered as a part of the edge else ignored.



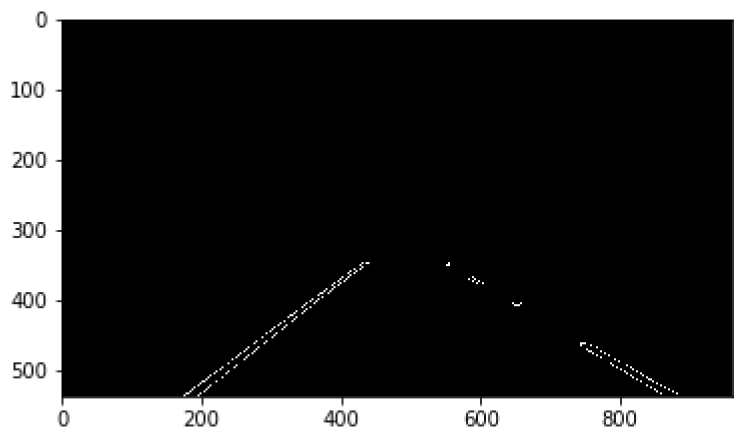
Sample Canny Edge Detection output

Region of Interest

Since we are only interested in the lane lines, the region of the image that is not immediately in front of the vehicle, like the cars in the adjascent lane, the barriers at the extreme left and right can all be neglected. This is achieved by masking the blurred image with a polygon that covers only the region of interest. This polygon is defined by four vertices [bottom_left, top_left, top_right, bottom_right]. The mask used is as shown below



Region of Interest Mask



Masked Edge image

Two different polygons were used for the two different sets of videos in the project. The solidWhiteRight and the solidYellowLeft videos have the same mask where as the challenge video has a different mask.

Hough Transform

Given a set of edge points in a binary image, Hough transformation is used to find a line that passes through all these points. A line in a cartesian space is represented by its slope m and intercept b as $y = mx + b$

The same line when transformed in the $m - b$ space will be a point as shown in the image below

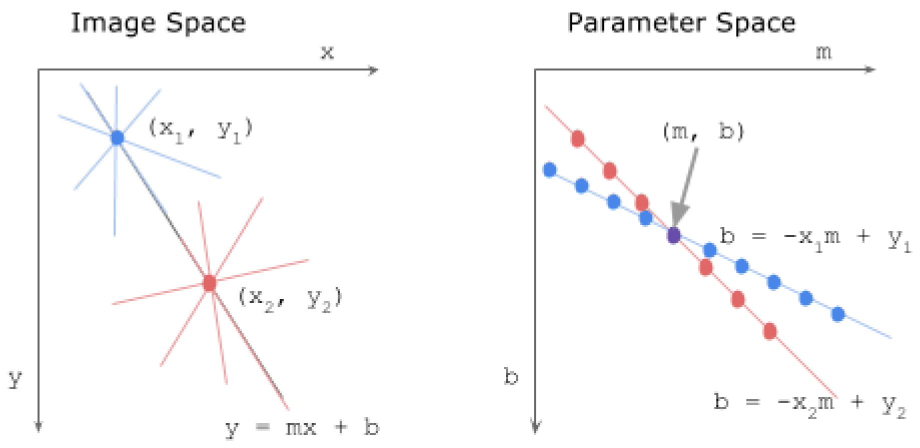


Image from Understanding Hough Transform with Python by Alyssa Quek

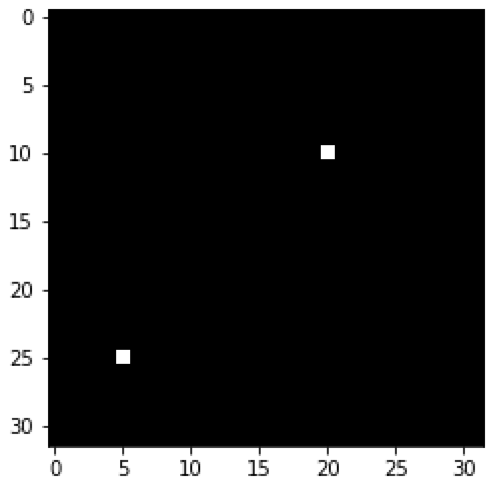
Where this representation fails is when the line is vertical with an infinite slope. To over come this, the $\rho - \theta$ space is used. The same line in $\rho - \theta$ space can be represented as

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

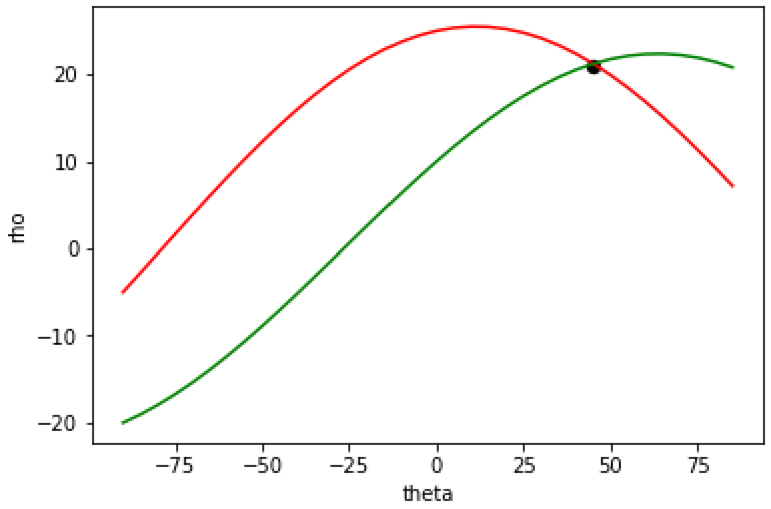
In the image below we have two points (5,25) and (20,10) represented as edge points. If we plot all the ρ values corresponding to θ varying from $[-90^0 to 90^0]$, we see that the resulting $\rho - \theta$ curves for these two points intersect at $\rho = 21$ and $\theta = 45^0$

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
image = np.zeros((32,32),dtype='uint8')
image[25,5] = 255
image[10,20] = 255

plt.imshow(image,cmap='gray')
```



```
from math import cos,sin,pi
color=['r','g']
count=0
for x,y in [(25,5),(10,20)]:
  rho = []
  for theta in range(-90,90,5):
    rho.append(x*cos(theta*pi/180.)+y*sin(theta*pi/180.))
  plt.plot(range(-90,90,5),rho,color=color[count])
  count+=1
plt.xlabel("theta")
plt.ylabel("rho")
plt.scatter(45,21,s=40,color='k')
plt.show()
```



Curves generated by collinear points in the image space intersect at common (ρ, θ) in the Hough transform space. The more curves intersect at a point, the more vote the line gets. Thus by using a **threshold** number of votes per line, we can select prominent lines in an image.

Identify the left and right line and get the average slope

Once the lines are obtained from the Hough transform, we can pass the line end points to a first order polynomial fit function in numpy to get the slope and y-intercept of the fitted line. We can identify the left and right line by checking their slopes. The line with a negative slope is the left line and the one with the positive slope is the right line. This is due to the fact that for an image the (0,0) is at the top left corner of the image.

The average slope of these lines making up the left and right lines can be obtained by taking a weighted average of all the individual line slopes. The slopes are weighted based on the length of the line and hence the average slope is influenced more by the longer lines than the shorter ones.

Draw and overlay the extrapolated lane lines on the image

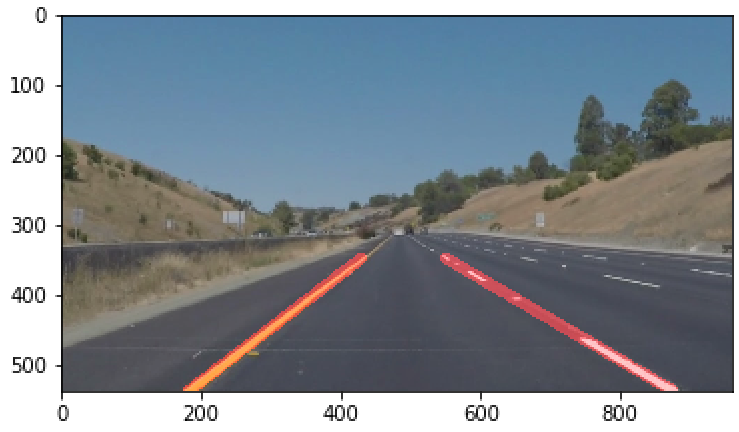
Using the averaged line slopes and intercepts , we can find the x co-ordinate of the start and end of the line

$$leftStartX = (leftStartY - leftAvgIntercept)/leftAvgSlope$$
$$leftEndX = (leftEndY - leftAvgIntercept)/leftAvgSlope$$

$$rightStartX = (rightStartY - rightAvgIntercept)/rightAvgSlope$$
$$rightEndX = (rightEndY - rightAvgIntercept)/rightAvgSlope$$

As was the case with the masks, since the views are different between the two sets of videos, the challenge video uses a different EndY value than the other two videos.

These lines are drawn on the original image with a set transparency to highlight the identified lane lines as shown below



Limitations

There are several limitations to this implementation. To list a few

1. Even with the simpler videos, the lane line flickers when there is some noise in the image. This stems from the fact that the average slope of the lane line is computed fresh for every frame and if edges with different slopes are detected between consecutive frames, the lane line is redrawn and appears as a flicker. This can be eliminated by keeping a history of the slopes of the previous few frames and taking an average of that would result in a more stable lane line.
2. In the challenge video, when the roads switches from asphalt to concrete, the intersection is identified as an edge. To avoid this, in the current implementation every edge is checked for its angle and if the edge is less than 25^0 it is not drawn. As a result the portion of the video that switches from asphalt to concrete does not have the lane marking. Even after thresholding the grayscale image, the problem persisted.
3. All lane lines are drawn as straignt lines. Hence the curvature of the road is not captured. A polynomial curve fitting needs to be implemented to be able to draw curved lane marking.

```
1
```