Introduction of the Lane Line Detection Project

In any driving scenario, lane lines are an essential component of indicating traffic flow and where a vehicle should drive. It's also a good starting point when developing a self-driving car! In this project, we will build a machine learning project to detect lane lines in real-time. We will do this using the concepts of computer vision using OpenCV library in Python. Here's the structure of my lane detection project:

- Reading Images
- Color Filtering in HLS
- Region of Interest
- Canny Edge Detection
- Hough Line Detection
- Line Filtering and Averaging
- Overlay detected lane
- Applying to Video

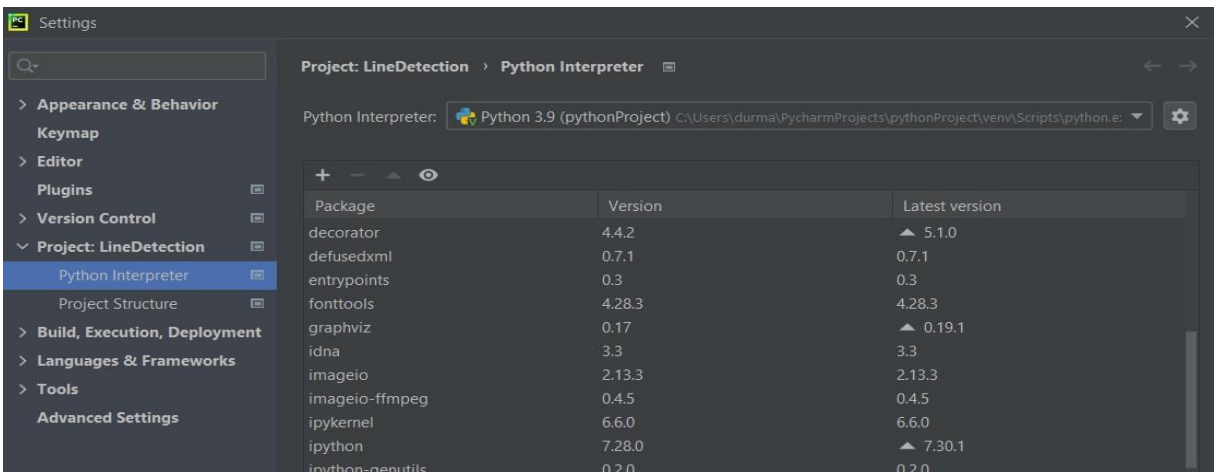
OpenCV

OpenCV is a very popular and well-documented library for computer vision. It's a great tool for any machine vision or deep learning.

Before we start;

We have to add some library to our sources

```
import os
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
from moviepy.editor import VideoFileClip
```



1. Reading Images

When developing an image processing pipeline, we need to read some example inputs that you can test your pipeline on. You can read single images with the following method in OpenCV:

We'll be reading a test images in a folder named *Images/.* Here's our code to read the test images in that directory:

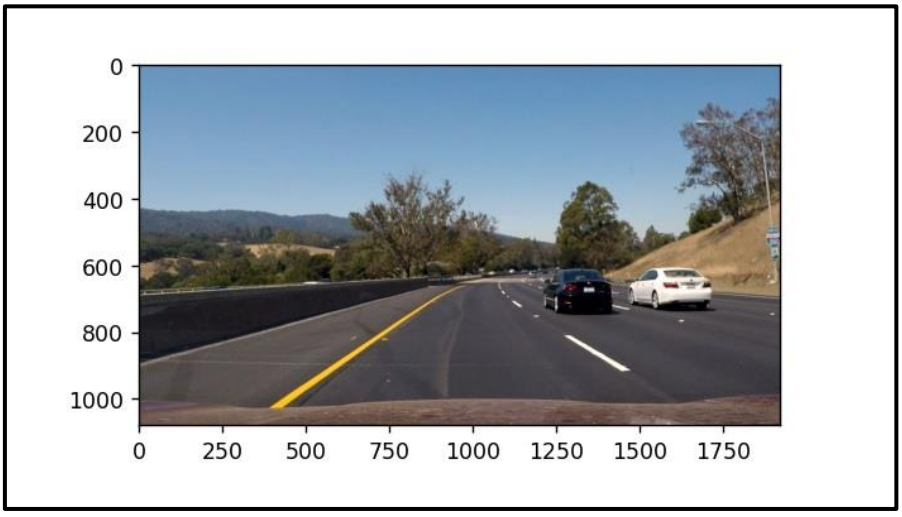
```
imageDir = 'Images/.'
imageFiles = os.listdir(imageDir)
imageList = [] #this list will contain all the test image
for i in range(0, len(imageFiles)):
    imageList.append(mpimg.imread(imageDir + imageFiles[i]))
```

and the following function to display our test image:

```
def display_images(images, cmap=None):
    plt.figure(figsize=(40,40))
    for i, image in enumerate(images):
        plt.subplot(3,2,i+1)
        plt.imshow(image, cmap)
        plt.autoscale(tight=True)
    plt.show()
```

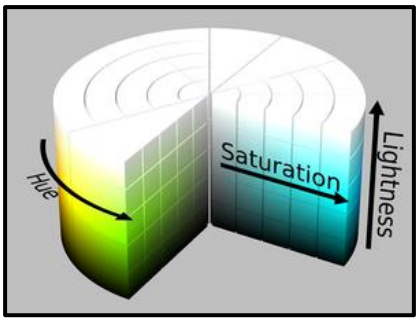
when we call the function *display_images* with the input *imageList*, you should see this in your output cell:

```
display_images(imageList)
```



Display_images(imageList)

2. Color Filtering



HLS Colorspace

Color filtering in RGB or BGR colorspaces is unnecessarily difficult, however. We can use different colorspaces, or **representations of color**, to easily filter out extraneous colors. One such colorspace that makes color filtering really easy is the **HLS colorspace**, which stands for **Hue, Lightness, and Saturation**. Each pixel dimension's value is between **0-255**.

We can convert images in the `BGR` colorspace to `HLS` like this:

```
hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
```

To get the yellow lane lines, we'll be getting rid of any pixels with a Hue value outside of **10** and **50** and a high **Saturation** value. To get the white lane lines, we'll be getting rid of any pixels that have a **Lightness** value that's less than **190.**

I add the filtered yellow and white lane lines into a single image.

Here's the function to filter out the lane lines and output them:

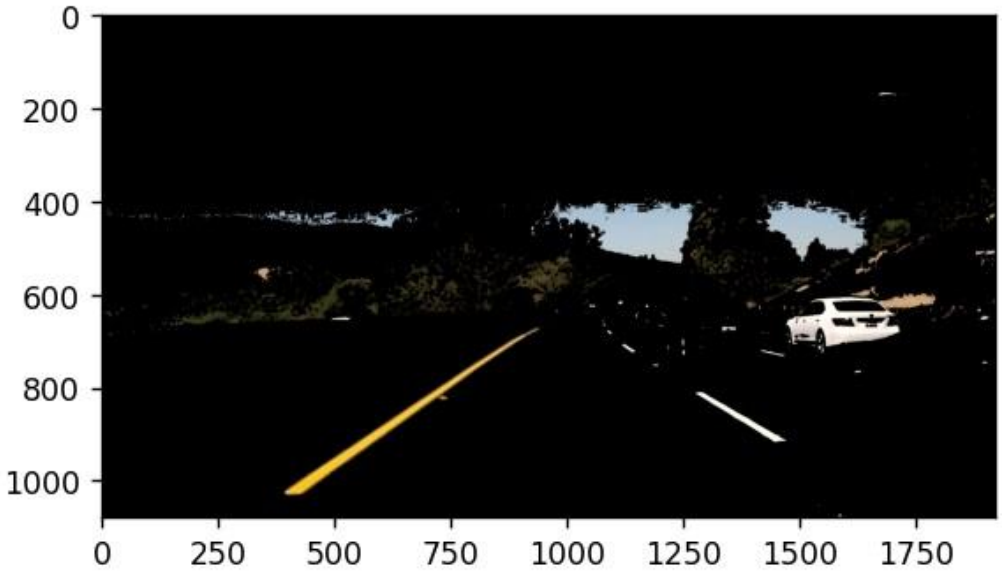
```
def color_filter(image):
    #convert to HLS to mask based on HLS
    hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
    lower = np.array([0,190,0])
    upper = np.array([255,255,255])
    yellower = np.array([10,0,90])
    yelupper = np.array([50,255,255])
    yellowmask = cv2.inRange(hls, yellower, yelupper)
    whitemask = cv2.inRange(hls, lower, upper)
    mask = cv2.bitwise_or(yellowmask, whitemask)
    masked = cv2.bitwise_and(image, image, mask = mask)
    return masked
```

to apply this to a list of images, we can use the `map()` function:

```
filtered_img = list(map(color_filter, imageList))
```

When we display the images;

```
display_images(filtered_img)
```



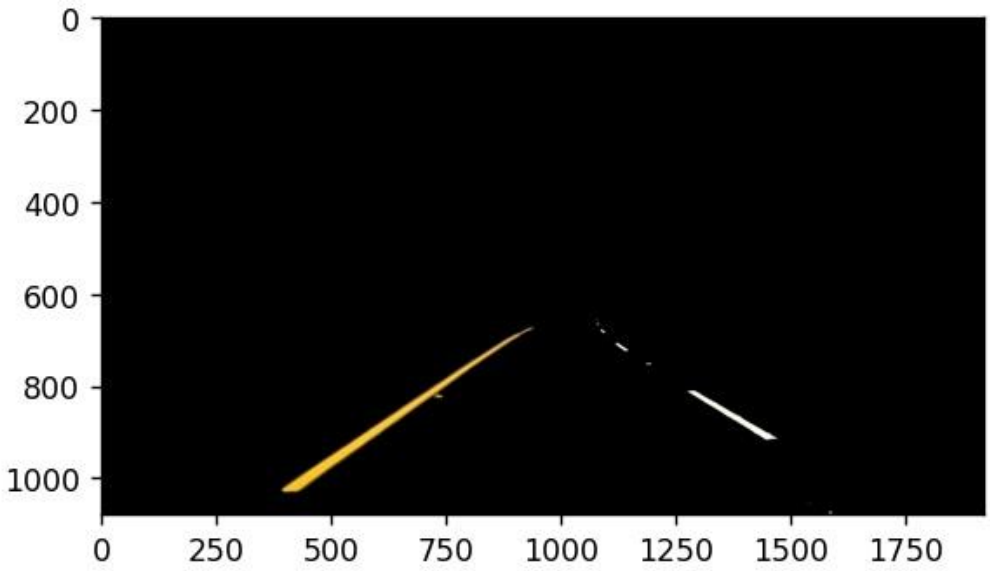
3. Region of Interest

There still some portions of the image that met the threshold, but weren't lane lines. We can narrow down where we're looking for lane lines by masking out a region of interest using the following function:

```
def roi(img):
    x = int(img.shape[1])
    y = int(img.shape[0])
    shape = np.array([[int(0), int(y)], [int(x), int(y)], [int(0.55*x), int(0.6*y)
], [int(0.45*x), int(0.6*y)]])
    #define a numpy array with the dimensions of img, but comprised of zeros
    mask = np.zeros_like(img)
    #Uses 3 channels or 1 channel for color depending on input image
    if len(img.shape) > 2:
        channel_count = img.shape[2]
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255
    #creates a polygon with the mask color
    cv2.fillPoly(mask, np.int32([shape]), ignore_mask_color)
    #returns the image only where the mask pixels are not zero
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image
```

We'll apply this to our test image using:

```
roi_img = list(map(roi, filtered_img))
```



4. Canny Edge Detection

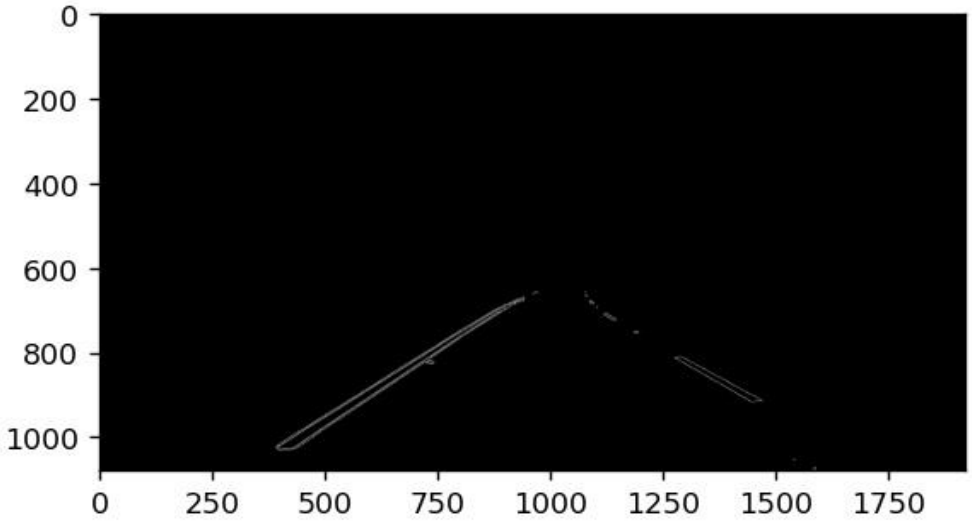
References *via https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html*

We have image in which the lane lines have been isolated, we can compute the edges of the lane lines. This can easily be done using CannyEdgeDetection. The idea behind Canny Edge Detection is that pixels near edges generally have a high gradient, or rate of change in value. We can use to detect lane lines. ***First of all we have to convert colorpsace to grayscale.***

```
cv2.Canny(gray, 50, 120)
```

where 50 and 120 are thresholds for the hysteresis procedure. Here's the code I used to convert to grayscale and extract the edges:

```
def grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
def canny(img):
    return cv2.Canny(grayscale(img), 50, 120)
canny_img = list(map(canny, roi_img))
display_images(canny_img,cmap='gray')
```



Extracted Edges using the Canny Detector

5. Hough Line Detection

Refference :

https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html
https://www.geeksforgeeks.org/line-detection-python-opencv-houghline-method/
https://www.youtube.com/watch?v=VyLihutdsPk&t=1447s

This is the hardest part of the project. We need to determine the lane lines from the canny edges we previously detected.

How can we detect lines? We'll be computing the lines using `cv2.HoughLinesP()`, and then filtering out the lines. We'll start by removing lines that are outside a determined slope range, and location. Additionally, we will filter the lines into their own corresponding lanes by seeing whether or not the filtered lines have positive or negative slope

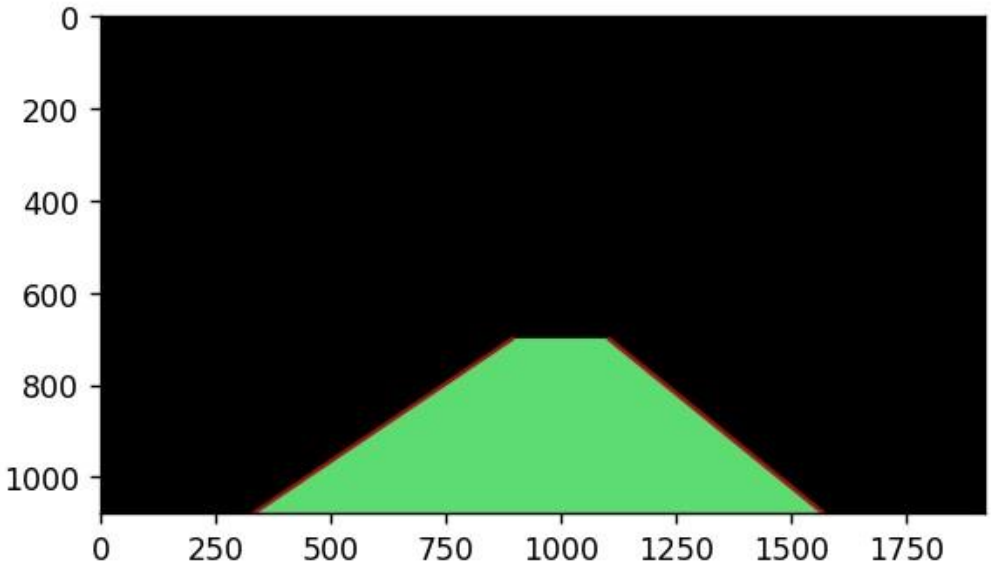
how do we combine them into a single line? One way to approach it is by using the **slope** and **intercept** of the lines. We'll be calculating the slope and intercept of each of the lines in a particular lane, then averaging the slope and intercepts to produce one line. I do this to both lanes.

Here's the code to do this:

```
rightSlope, leftSlope, rightIntercept, leftIntercept = [],[],[],[]
def draw_lines(img, lines, thickness=5):
    global rightSlope, leftSlope, rightIntercept, leftIntercept
    rightColor=[162,19,17]    leftColor=[162,19,17]
    #this is used to filter out the outlying lines that can affect the average
    #We then use the slope we determined to find the y-intercept of the filtered l
ines by solving for b in y=mx+b
```

```
    for line in lines:
        for x1,y1,x2,y2 in line:
            slope = (y1-y2)/(x1-x2)
            if slope > 0.3:
                if x1 > 500 :
                    yintercept = y2 - (slope*x2)
                    rightSlope.append(slope)
                    rightIntercept.append(yintercept)
                else: None
            elif slope < -0.3:
                if x1 < 600:
                    yintercept = y2 - (slope*x2)
                    leftSlope.append(slope)
                    leftIntercept.append(yintercept)
    #We use slicing operators and np.mean() to find the averages of the 30 previou
s frames
    #This makes the lines more stable, and less likely to shift rapidly
    leftavgSlope = np.mean(leftSlope[-30:])
    leftavgIntercept = np.mean(leftIntercept[-30:])
    rightavgSlope = np.mean(rightSlope[-30:])
    rightavgIntercept = np.mean(rightIntercept[-30:])
    #Here we plot the lines and the shape of the lane using the average slope and
intercepts
    try:
        left_line_x1 = int((0.65*img.shape[0] - leftavgIntercept)/leftavgSlope)
        left_line_x2 = int((img.shape[0] - leftavgIntercept)/leftavgSlope)
        right_line_x1 = int((0.65*img.shape[0] - rightavgIntercept)/rightavgSlope)
        right_line_x2 = int((img.shape[0] - rightavgIntercept)/rightavgSlope)
        pts = np.array([[left_line_x1, int(0.65*img.shape[0])],[left_line_x2, int(
img.shape[0])],[right_line_x2, int(img.shape[0])],[right_line_x1, int(0.65*img.sha
pe[0])]], np.int32)
        pts = pts.reshape((-1,1,2))
        cv2.fillPoly(img,[pts],(93,220,114,100))
        cv2.line(img, (left_line_x1, int(0.65*img.shape[0])), (left_line_x2, int(i
mg.shape[0])), leftColor, 8)
        cv2.line(img, (right_line_x1, int(0.65*img.shape[0])), (right_line_x2, int
(img.shape[0])), rightColor, 8)
    except ValueError:
        pass
```

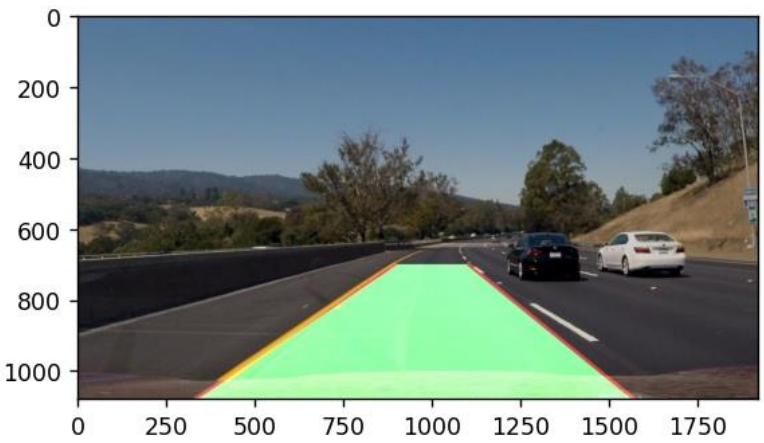
```
def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
    """
    `img` should be the output of a Canny transform.
    """
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines)
    return line_img
def linedetect(img):
    return hough_lines(img, 1, np.pi/180, 10, 20, 100)
hough_img = list(map(linedetect, canny_img))
display_images(hough_img)
```



6. Overlay Detected Lane:

I add the previously detected lanes to my original image using:

```
def weightSum(input_set):
    img = list(input_set)
    return cv2.addWeighted(img[0], 1, img[1], 0.8, 0)
result_img = list(map(weightSum, zip(hough_img, imageList)))
display_images(result_img)
```



7. Applying this to Video:

The **MoviePy** library provides some great tools for video processing.

The following function allows to apply the entire pipeline to video:

```
def processImage(image):
    interest = roi(image)
    filterimg = color_filter(interest)
    canny = cv2.Canny(grayscale(filterimg), 50, 120)
    myline = hough_lines(canny, 1, np.pi/180, 10, 20, 5)
    weighted_img = cv2.addWeighted(myline, 1, image, 0.8, 0)
    return weighted_img
```

```
output1 = 'test_videos_output/rendered-video.mp4'
clip1 = VideoFileClip("test_videos/video.mp4")
pclip1 = clip1.fl_image(processImage)
pclip1.write_videofile(output1, audio=False)
```