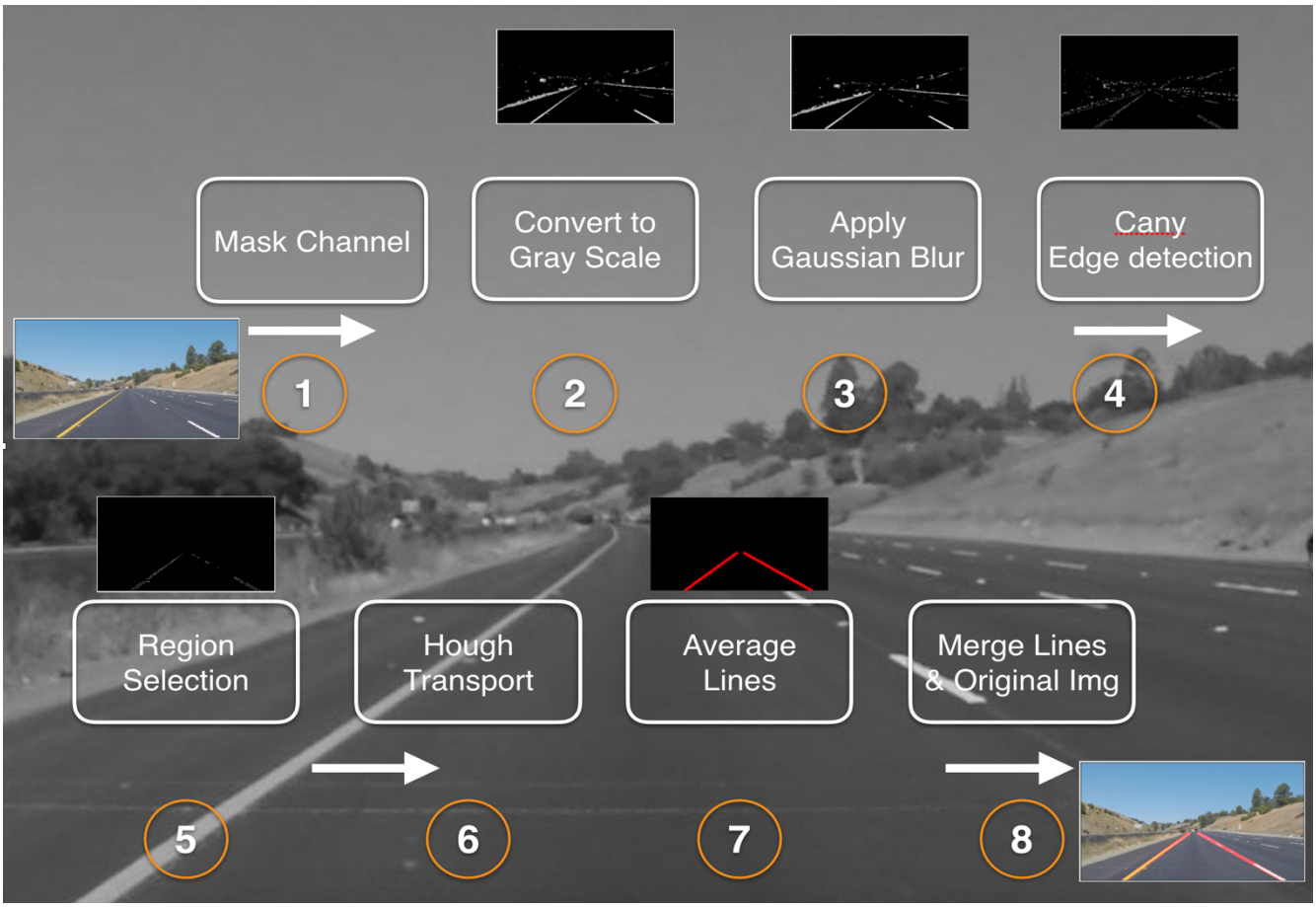
Lane Detection using Python & OpenCV

The goals / steps of this project are the following:

Make a pipeline that finds lane lines on the road

Apply the pipeline to find lanes in the videos

Pipeline architecture:



1. Apply Color Selection- Mask Channel
2. Covert the image to gray scale image
3. Apply Gaussian smoothing.
4. Perform Canny edge detection.
5. Determine the region of interest.
6. Apply Hough transform.
7. Average and extrapolating the lane lines.
8. Merge Line & Original images
9. Repeat step 1 to 8 for each frame on the video streams.

Prework:

I wrote a function called `display_img_grid()` that shows all the test images we're working on using matplotlib as a 3 rows 2 columns grid.

```
def  display_img_grid(imgs, cols = 2, rows =3 , cmap=None):
    """
    Display a list of images in a single figure with matplotlib.
        Parameters:
            imgs: List of np.arrays compatible with plt.imshow.
            cols (Default = 2): Number of columns in the figure.
            rows (Default = 3): Number of rows in the figure.
            cmap (Default = None): Used to display gray images.
    """
    plt.figure(figsize=(10, 11))
    for i, img in enumerate(imgs):
        plt.subplot(rows, cols, i+1)
        plt.imshow(img, cmap = cmap)
        plt.xticks([])
        plt.yticks([])
    plt.tight_layout(pad=0, h_pad=0, w_pad=0)
    plt.show()
```

---

1. Mask Channels(color selection)

Lane lines in the test images are in white and yellow. We need to choose the most suitable color space, that clearly highlights the lane lines. I applied color selection to the original RGB images.

Mask pixels out of the threshold ranges using cv2.inRange function

```
def color_selection(img):
    """
    Apply color selection to image to mask everything except for white and yellow lane lines.
        Parameters:
            img: An np.array.
    """
    #Define thresholds
    red_threshold = green_threshold = blue_threshold = 200
    upper_threshold = np.uint8([255, 255, 255])
    lower_threshold = np.uint8([red_threshold, green_threshold, blue_threshold])
    #White color mask
    white_mask = cv2.inRange(img, lower_threshold, upper_threshold)

    #Yellow color mask
    lower_threshold = np.uint8([170, 170, 0])
    yellow_mask = cv2.inRange(img, lower_threshold, upper_threshold)

    #Combine white and yellow mask images
    mask = cv2.bitwise_or(white_mask, yellow_mask)
    masked_image = cv2.bitwise_and(img, img, mask = mask)

    return masked_image
```

2. Grayscale Conversion

Convert the masked output images to Grayscale image this helps in

3. Canny Edge Detection

We need to detect edges in the images to be able to correctly detect lane lines. The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. The Canny algorithm involves the following steps:

- Gray scaling the images: The Canny edge detection algorithm measures the intensity gradients of each pixel. So, we need to convert the images into gray scale in order to detect edges.
- Gaussian smoothing: Since all edge detection results are easily affected by image noise, it is essential to filter out the noise to prevent false detection caused by noise. To smooth the image, a Gaussian filter is applied to convolve with the image. This step will slightly smooth the image to reduce the effects of obvious noise on the edge detector.
- Find the intensity gradients of the image.
- Apply non-maximum suppression to get rid of spurious response to edge detection.
- Apply double threshold to determine potential edges.
- Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges. If an edge pixel’s gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel’s gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it

will be suppressed. The two threshold values are empirically determined and their definition will depend on the content of a given input image.

4. Region of interest

We're interested in the area facing the camera, where the lane lines are found. So, we'll apply region masking to cut out everything else. Get image Shape & set vertices based on the first image as all the images are of same shape

```
def region_of_interest(img, vertices):
    """
    Applies an image mask.

    Only keeps the region of the image defined by the polygon
    formed from `vertices`. The rest of the image is set to black.
    `vertices` should be a numpy array of integer points.
    """
    #defining a blank mask to start with
    mask = np.zeros_like(img)

    #defining a 3 channel or 1 channel color to fill the mask with depending on the input image
    if len(img.shape) > 2:
        channel_count = img.shape[2]  # i.e. 3 or 4 depending on your image
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255

    #filling pixels inside the polygon defined by "vertices" with the fill color
    cv2.fillPoly(mask, vertices, ignore_mask_color)

    #returning the image only where mask pixels are nonzero
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image
```

5 Hough Transform

The Hough transform is a technique which can be used to isolate features of a particular shape within an image. I'll use it to detected the lane lines in `selected_region_images`.

```
def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
    """
    `img` should be the output of a Canny transform.

    Returns an image with hough lines drawn.
    """
    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, average_lane_lines(line_img,lines),[255, 0, 0],13)
    return line_img
```

6. Averaging and extrapolating the lane lines

We have multiple lines detected for each lane line. We need to average all these lines and draw a single line for each lane line. We also need to extrapolate the lane lines to cover the full lane line length.

The left lane should have a negative slope (given that `y` coordinate is reversed in the images), and the right lane should have a negative slope. Therefore, we'll collect positive slope lines and negative slope lines separately and take averages. After calculating the average slope and intercept of each lane line, we'll convert these values to pixel points, and then we'll be able to draw the full length lane line.

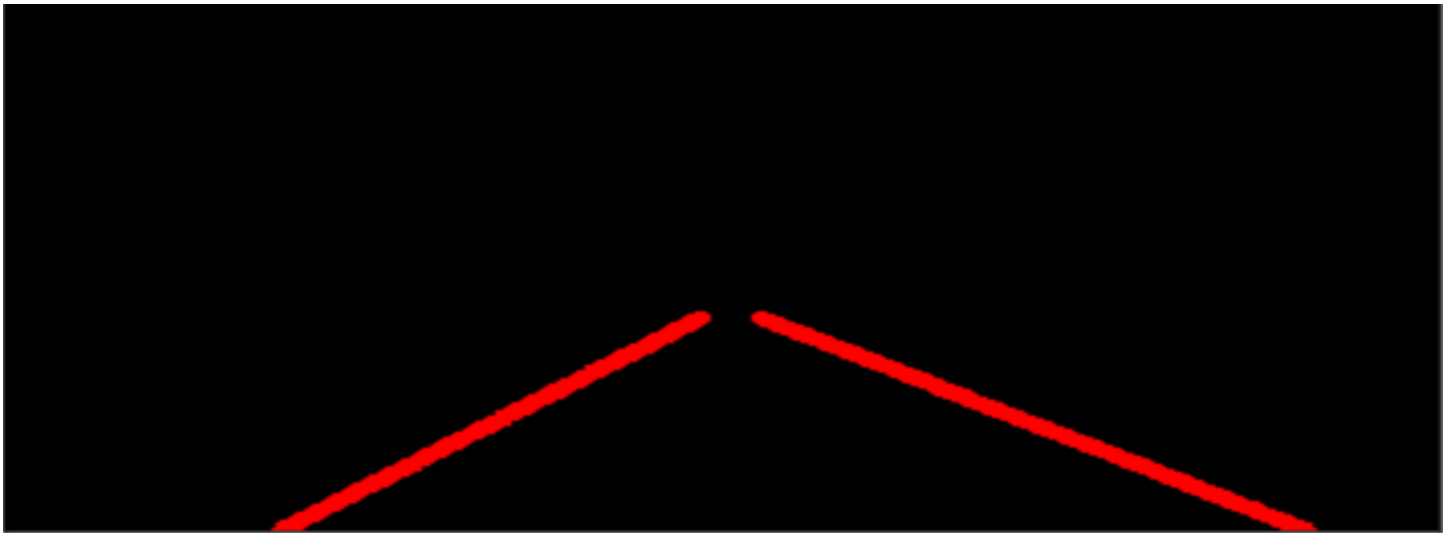
```
def extrapolate_slope_intercept(lines):
```

```
    """
    Find the slope and intercept of the left and right lanes of each image.
        Parameters:
            lines: The output lines from Hough Transform.
    """
    left_lines    = [] #(slope, intercept)
    left_weights  = [] #(length,)
    right_lines   = [] #(slope, intercept)
    right_weights = [] #(length,)

    for line in lines:
        for x1, y1, x2, y2 in line:
            if x1 == x2:
                continue
            slope = (y2 - y1) / (x2 - x1)
            intercept = y1 - (slope * x1)
            length = np.sqrt(((y2 - y1) ** 2) + ((x2 - x1) ** 2))
            if slope < 0:
                left_lines.append((slope, intercept))
                left_weights.append((length))
            else:
                right_lines.append((slope, intercept))
                right_weights.append((length))
    left_lane  = np.dot(left_weights,  left_lines) / np.sum(left_weights)  if len(left_weights) >
0 else None
    right_lane = np.dot(right_weights, right_lines) / np.sum(right_weights) if len(right_weights)
> 0 else None
    return left_lane, right_lane

def pixel_points(y1, y2, line):
    """
    Converts the slope and intercept of each line into pixel points.
        Parameters:
            y1: y-value of the line's starting point.
            y2: y-value of the line's end point.
            line: The slope and intercept of the line.
    """
    if line is None:
        return None
    slope, intercept = line
    x1 = int((y1 - intercept)/slope)
    x2 = int((y2 - intercept)/slope)
    y1 = int(y1)
    y2 = int(y2)
    return ((x1, y1), (x2, y2))

def  average_lane_lines(image, lines):
    """
    Create full lenght lines from pixel points.
        Parameters:
            image: The input test image.
            lines: The output lines from Hough Transform.
    """
    left_lane, right_lane = extrapolate_slope_intercept(lines)
    y1 = image.shape[0]
    y2 = y1 * 0.6
    left_line  = pixel_points(y1, y2, left_lane)
    right_line = pixel_points(y1, y2, right_lane)
    return left_line, right_line
```



7. Merge Lines & Original Images

Merge average Hough Transform line images with the Original images**.**

7. Apply on video streams

Now, we'll use the above functions to detect lane lines from a video stream. The video inputs are in test_videos folder. The video outputs are generated in output_videos folder.

```
def process_image(image):

    #Apply the below pipline steps for each from of the video
    masked_color_img = color_selection(image)
    gray_img = grayscale(masked_color_img)
    gaussian_img = gaussian_blur(gray_img,5)
    cany_img = canny(gaussian_img, 100,200)
    region_selected_img = region_of_interest(cany_img, vertices)
    hough_line_img = hough_lines(region_selected_img, rho, theta, threshold, min_line_length,
max_line_gap)
    return weighted_img(image,hough_line_img)

def process_video(test_video, output_video):
    """
    Read input video stream and produce a video file with detected lane lines.
        Parameters:
            test_video: Input video.
            output_video: A video file with detected lane lines.
    """
        white_output = 'test_videos_output/solidWhiteRight.mp4'
        #Apply the below pipline steps for each from of the video
        clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4")
        white_clip = clip1.fl_image(process_image) #NOTE: this function expects color images!!
%       time white_clip.write_videofile(white_output, audio=False)
```

Conclution

The project succeeded in detecting the lane lines clearly in the video streams. This project is intended to only detect (mostly) straight lines. A possible improvement would be to modify the above pipeline to detect curved lane line.