Project 1: Find Lane Lines on the road

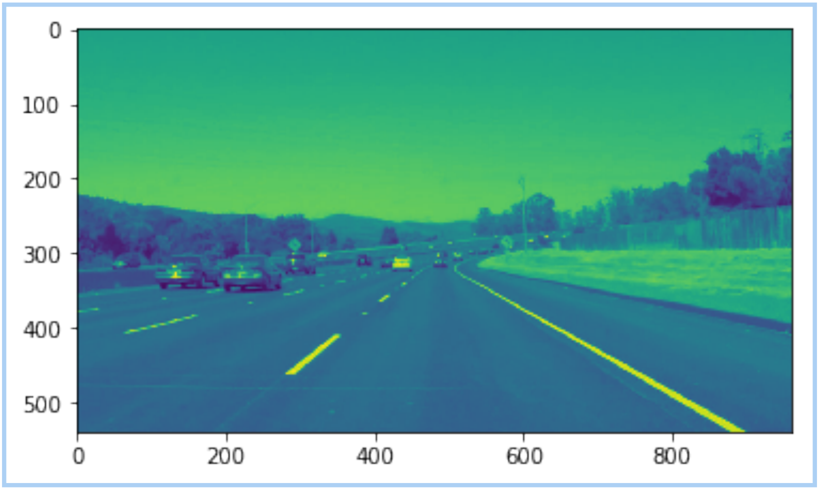
Introduction

This project is aiming to find lane lines on the road. We will have simple images and video clips as input, and try to detect and

Pipeline build procedure

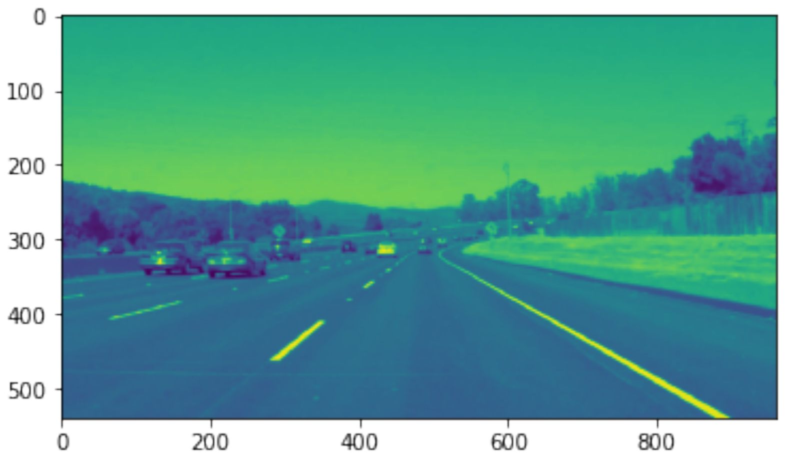
Read in and grayscale the image

- By applying grayscale(img) function



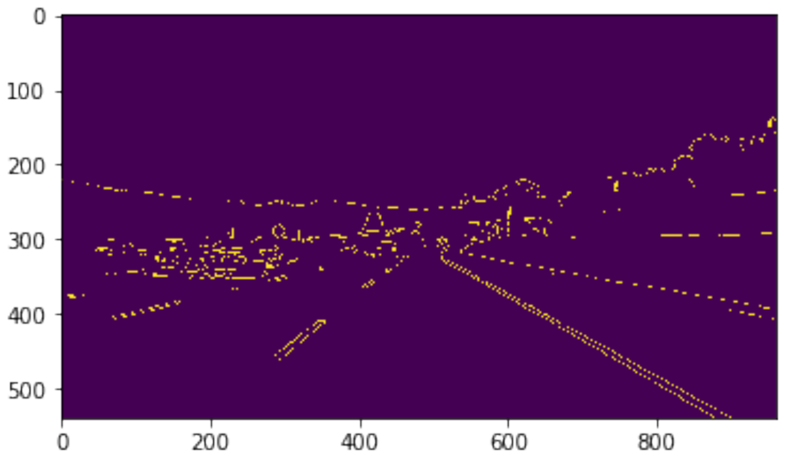
Smooth the image

- Define kernel_size and call gaussian_blur() to make the grey scale image more smooth



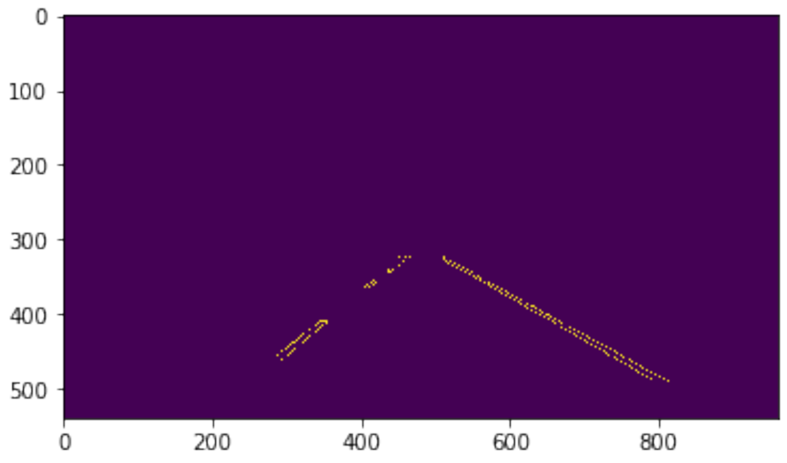
Define our parameters for Canny and apply

- Define low and high threshold for canny edge detection
- Call canny() function to get the edges



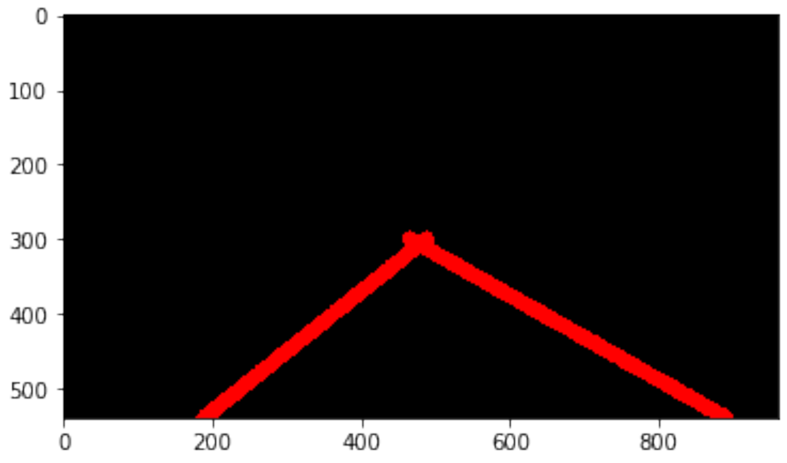
Define a mask of region that we are interested

- Define four verticals that form a shape we are interested in
- Call region_of_interest() function to effective area of edges



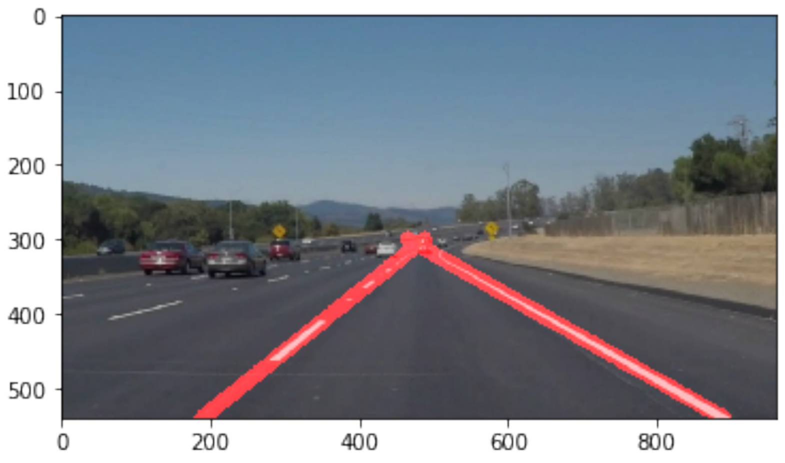
Define the Hough transform parameters

- Define $\rho, \theta,$ hough_threshold, min_line_len, max_line_gap
- Call the hough_lines() function to draw lines and apply hough transform



Weight the draw lines to original image

- Add the lines into the initial image

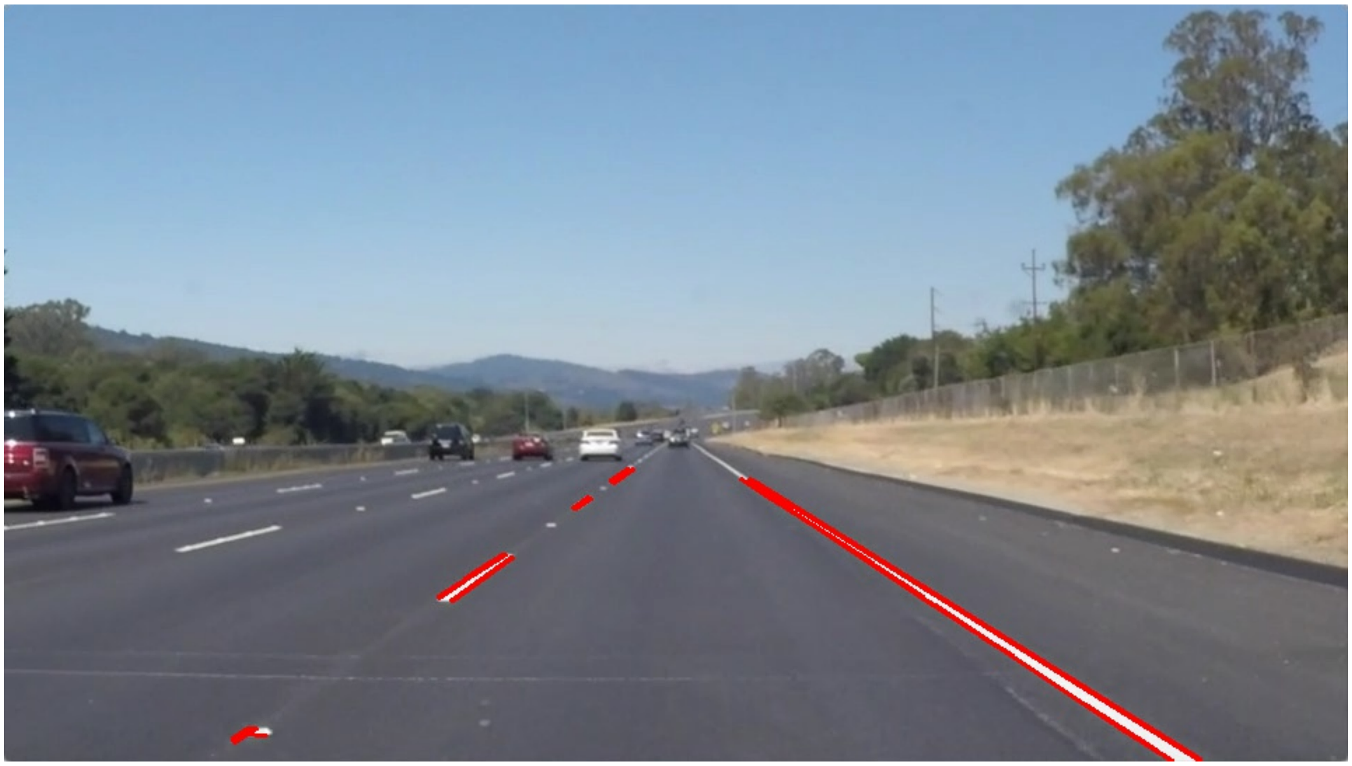


Improvement on line drawing function

The old version line drawing function simply iterates all the segments and draw the lines from the start point (x1, y1) to the end point (x2, y2) of current segment.

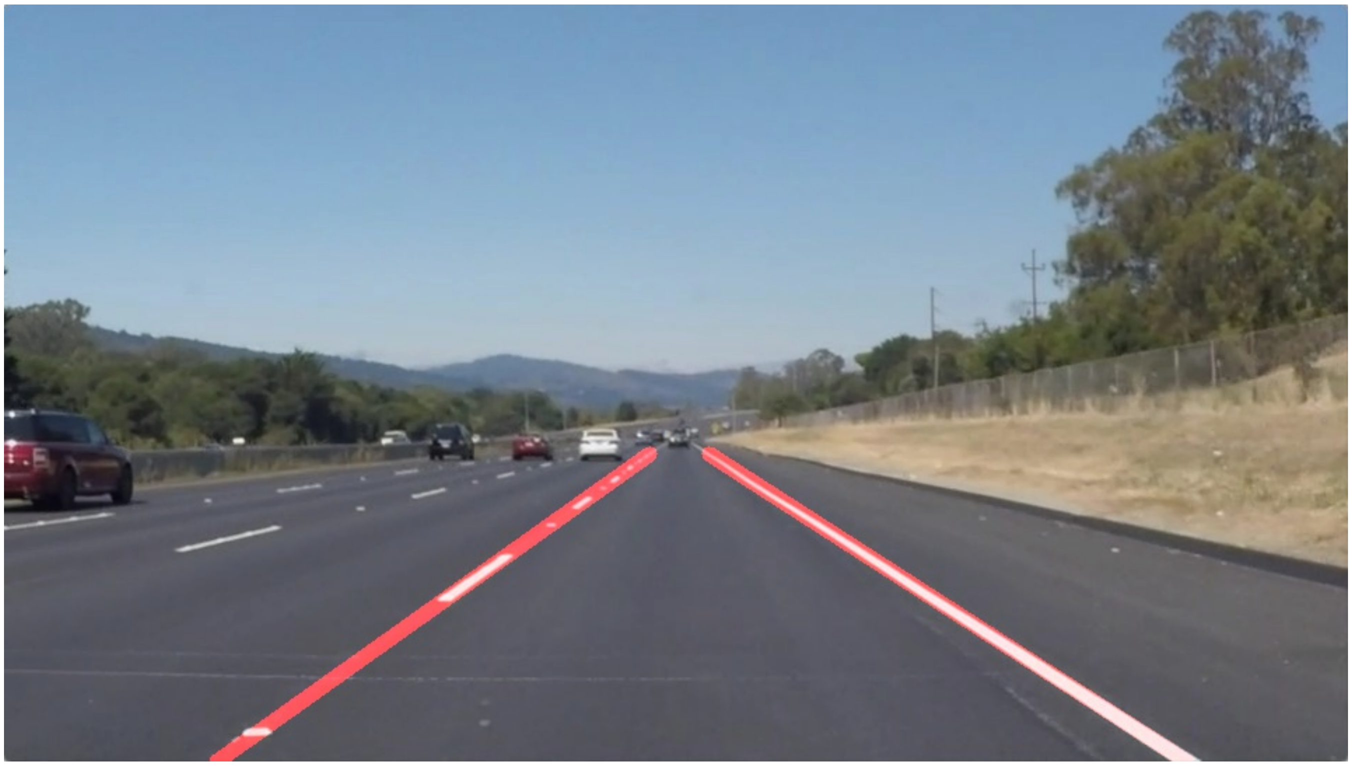
```
def draw_lines(img, lines, color=[255, 0, 0], thickness=2):
    """
    NOTE: this is the function you might want to use as a starting point once you want to
    average/extrapolate the line segments you detect to map out the full
    extent of the lane (going from the result shown in raw-lines-example.mp4
    to that shown in P1_example.mp4).

    Think about things like separating line segments by their
    slope ((y2-y1)/(x2-x1)) to decide which segments are part of the left
    line vs. the right line.  Then, you can average the position of each of
    the lines and extrapolate to the top and bottom of the lane.

    This function draws `lines` with `color` and `thickness`.
    Lines are drawn on the image inplace (mutates the image).
    If you want to make the lines semi-transparent, think about combining
    this function with the weighted_img() function below
    """
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)

```

From the video clips or sample images we can see, some lanes are solid and some are dash.



https://github.com/pineal/CarND-LaneLines-P1/blob/master/examples/line-segments-example.jpg?raw=true

We want improve the function to draw a line either the lane is solid or dash.



https://github.com/pineal/CarND-LaneLines-P1/blob/master/examples/laneLines_thirdPass.jpg?raw=true

The result will have two solid lines laying on the real lane lines. We use the slope to distinguish the left lane or right lane. If the lane line is formed of several segments. Geometrically, a straight line could be expressed as:

$$Y = kX + b$$

we get a straight line cover them well by calculating the average slope k and extrapolation b.

Method to get average slope and extrapolation
We can always store all the inputs, sum up and divide by the size of samples to get the average, but it takes extra $O(N)$ space. Here I use a running average algorithm to calculate average values instead, which will save the extra space to constant. The drawback is it cost a little more on calculation especially in float numbers, although the time complexity keeps same.

$$Avg = \frac{Avg^{'} \times N + val}{N+1}$$

```
    l_x1_avg, l_x2_avg, l_y1_avg, l_y2_avg = 0.0, 0.0, 0.0, 0.0
    r_x1_avg, r_x2_avg, r_y1_avg, r_y2_avg = 0.0, 0.0, 0.0, 0.0

    # N: count of left lines, M: count of right lines
    N, M = 0, 0

    for line in lines:
        for x1, y1, x2, y2 in line:
            slope = (y2 - y1) / (x2 - x1)
            if slope > 0:
                r_x1_avg = float(r_x1_avg * M + x1) / float(M + 1)
                r_x2_avg = float(r_x2_avg * M + x2) / float(M + 1)
                r_y1_avg = float(r_y1_avg * M + y1) / float(M + 1)

                r_y2_avg = float(r_y2_avg * M + y2) / float(M + 1)
                M = M + 1
            else:
                l_x1_avg = float(l_x1_avg * N + x1) / float(N + 1)
                l_x2_avg = float(l_x2_avg * N + x2) / float(N + 1)
                l_y1_avg = float(l_y1_avg * N + y1) / float(N + 1)

                l_y2_avg = float(l_y2_avg * N + y2) / float(N + 1)
                N = N + 1
```

Then we can get slope:
$k = \frac{y_2 - y_1}{x_2 - x_1}$

and get extrapolation:
$b = y_0 - k \times x_0$

```

    # calculate params for the stright segment line
    l_k = (l_y2_avg - l_y1_avg) / (l_x2_avg - l_x1_avg) if (l_x2_avg - l_x
1_avg) else 0
    r_k = (r_y2_avg - r_y1_avg) / (r_x2_avg - r_x1_avg) if (r_x2_avg - r_x
1_avg) else 0
```

```

    l_b = l_y1_avg - l_k * l_x1_avg
    r_b = r_y2_avg - r_k * r_x2_avg

```

Finally, we pick a start point and draw the extended lines.

```
   y_min = 300

   # Calculate the extended lines
    left_y1 = y_min
    left_y2 = y
    left_x1 = int((left_y1 - l_b) / l_k) if l_k else 0
    left_x2 = int((left_y2 - l_b) / l_k) if l_k else 0
    right_y1 = y_min
    right_y2 = y
    right_x1 = int((right_y1 - r_b) / r_k) if r_k else 0
    right_x2 = int((right_y2 - r_b) / r_k) if r_k else 0

    # Draw the lines
    if l_k and r_k:
        cv2.line(img, (left_x1, left_y1), (left_x2, left_y2), color, thickness)
        cv2.line(img, (right_x1, right_y1), (right_x2, right_y2), color, thickness)

```

Sometimes we will get a invalid slope, which is the drawback of this method to representing a straight line, but this is a rare situation, so my choice is just skip this frame. A better solution could be record the last result and update every time, if the lines are invalid in this frame, just use the previous one.

Summary

This pipelines failed the challenge part. Obviously we should get a better result on edge detection, and restrict them in a certain area. In addition, we should observe the bias every time, if it changes a lot, it should be a noise result, and should apply the previous lines probably.