Project: Finding Lane Lines on the Road

Udacity CarND

1. Overview

Finding Lane Lines on the Road

The goal of this project is to make a pipeline that finds lane lines on the road. It follows the following steps:

- Load image
- Set color threshold for R, G and B
- Define the region of interest
- Canny transfer
- Draw Hough lines

2. Project Introduction

The project is using the image processing technology to detect the lane on the road. The mainly used package is `cv2` provided by **OpenCV**. The process is firstly tested on the images and then finds the lanes on a movie.

3. Project Pipeline

The workflow of the project is followed by the pipeline below:

1. *Define the R, G and B color threshold and interest region to extract the write color from the image*
2. *Canny transfer by detecting high gradient of color change*
3. *Use Hough lines to draw detected lane lines*

3.1 Define color threshold

The images are taken from `test_images/.jpg` by calling the function `visualize_images(images, num_images)` .

```
#printing out some stats and plotting
def visualize_images(images, num_images, plot_size_a, plot_size_b):
    rows = 2
    columns = math.ceil(num_images/rows)
    fig, axeslist = plt.subplots(rows, columns)
    fig.set_size_inches(plot_size_a, plot_size_b)
    for ind, f in zip(range(num_images),images):
        axeslist.ravel()[ind].imshow(f)
        axeslist.ravel()[ind].set_axis_off()
    plt.tight_layout()
```

All the test images are shown as follows:

```
images = [mpimg.imread(f) for f in glob.glob('test_images/*.jpg')]
num_images = np.shape(images)[0]
visualize_images(images, num_images, 18.5, 10.5)
```

The color threshold and the interest region make the lane lines clearer and easy for the next step Canny Transfer.

The color threshold is used to detect white component and yellow component. The expected result is that after filtering by the threshold, the images will become black background and selected color component inside the interest region.

```
red_threshold = 200
green_threshold = 200
blue_threshold = 0
rgb_threshold = [red_threshold, green_threshold, blue_threshold]
thresholds = (image[:,:,0] < rgb_threshold[0])    \
            | (image[:,:,1] < rgb_threshold[1])   \
            | (image[:,:,2] < rgb_threshold[2])
color_select[thresholds] = [0,0,0]
```

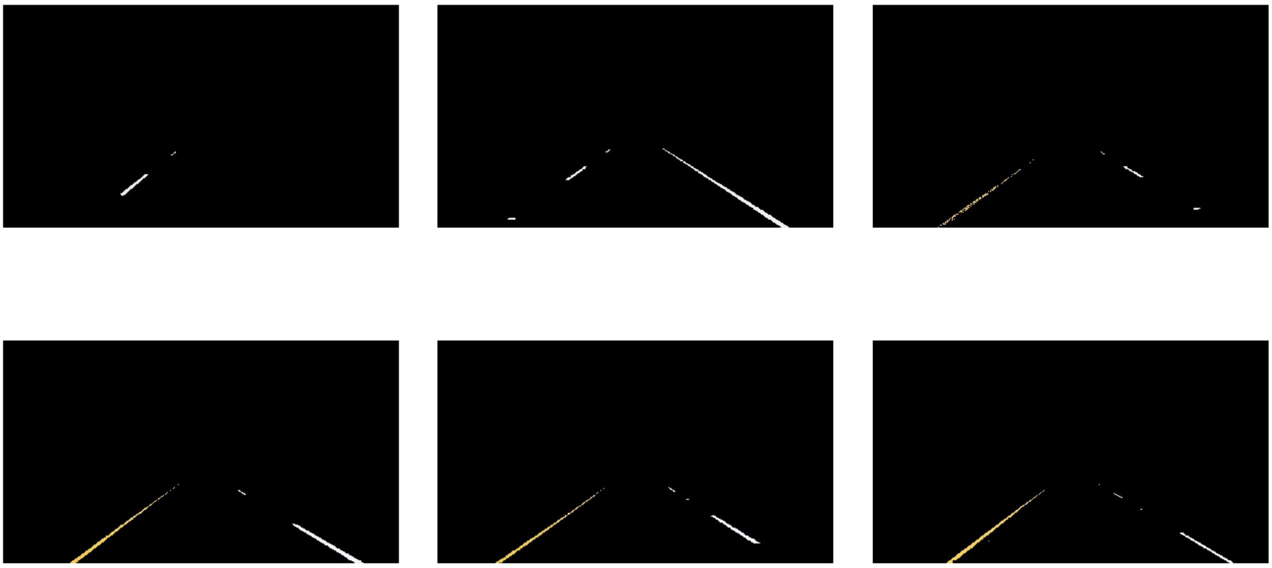
The interest region is to pick mostly middle-buttom region, which has higher posibility including the left lane line and the right lane line.

```
left_bottom = [100, 539]
right_bottom = [875, 539]
apex_left = [400, 350]
apex_right = [550, 350]
vertices = np.array([left_bottom, right_bottom, apex_right, apex_left], dtype = np.int32)
image_select = region_of_interest(color_select, [vertices])
```

where the function `region_of_interest(img, vertices)` is taken over from the class.

```
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.
    """
    #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
```

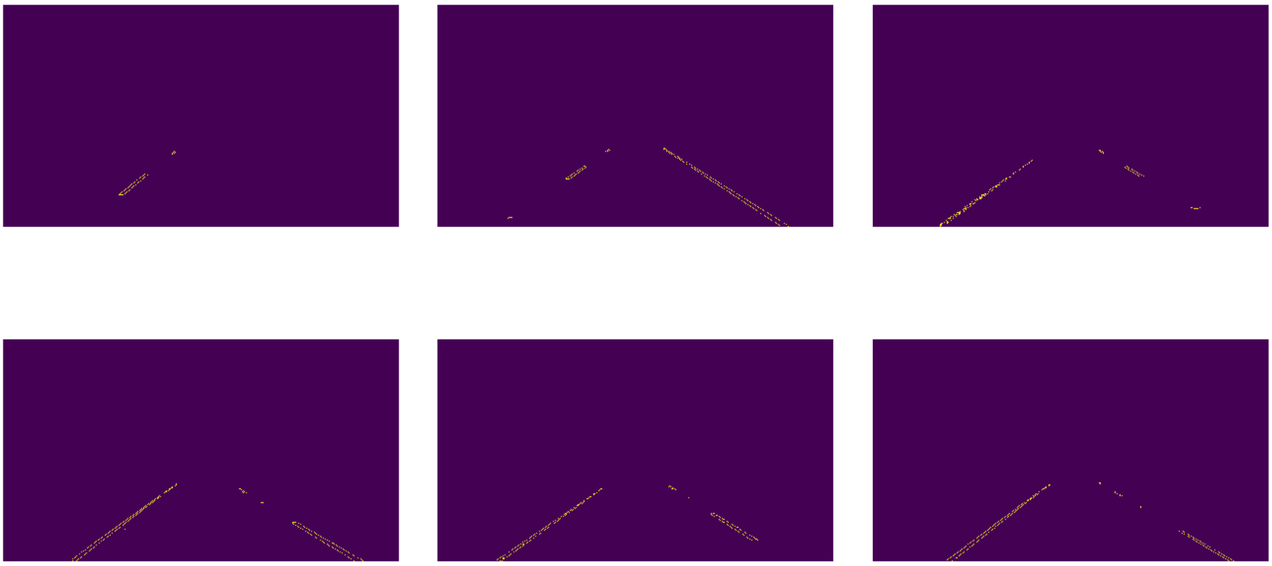
```
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image
```



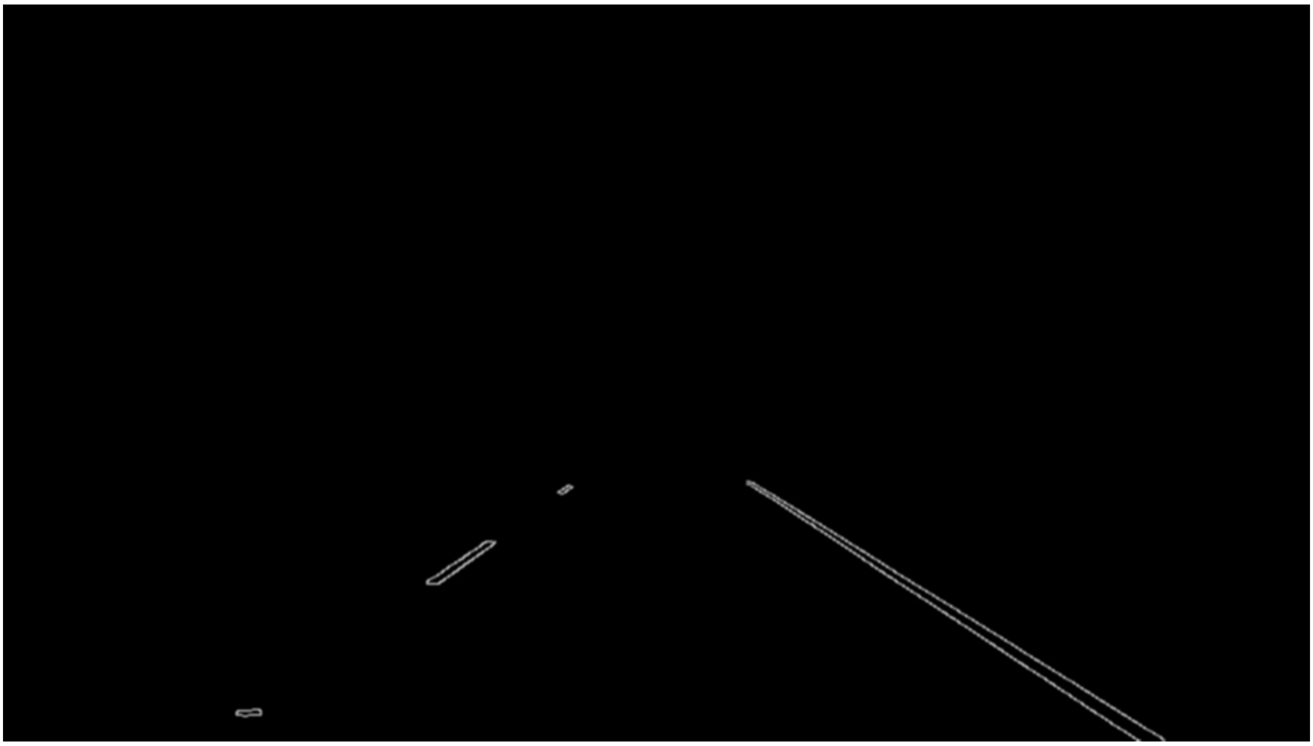
3.2 Canny Transfer

`cv2` provides 1-line code for *Canny Transfer*: `cv2.Canny(img, low_threshold, high_threshold)`.

`cv2.GaussianBlur` is added after the Canny Transfer. This will make the detected edges smoother.



Zoom one of these 6 test images.



3.3 Hough Lines

After the Canny Transfer, we get the single edge. To get the lane line detection, I need to connect them using lines. The hough lines is defined as follows:

```
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, lines)
    return line_img
```

The most important part of `hough_lines` is the function `draw_lines`. `draw_lines` reads each line segments from an array `line_img`. Basically, `draw_lines` calculates the slop of each line segments, seperates the left lane lines (usually the slop is negative value) and the right lane lines (usually the slop is positive value), filteres the slopes which are `0` or `inf`. I only need 1 line for left lane and 1 line for right lane, the final slope of each is the average value of each.

So here is the `draw_lines` step by step:

```
# filter the horizontal lines, for the rest, save to left lines array and right lines array
for line in lines:
    for x1,y1,x2,y2 in line:
```

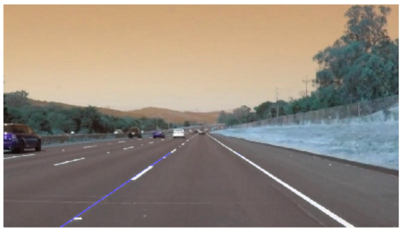
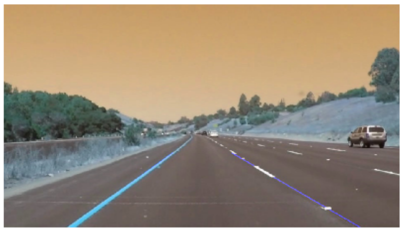
```
        # Skip lines which lead to slope of 0 or inf
        if (x1 == x2) or (y1 == y2):
            continue
        slope = ((y2-y1)/(x2-x1))
        # skip horizontal lines
        if (slope > -0.5 and slope < 0.5) or slope < -1 or slope > 1:
            continue
        if slope < 0:
            # Left Lane
            #cv2.line(img, (x1, y1), (x2, y2), color, 2)
            left_lines += [(x1, y1, x2, y2, slope)]
        else:
            # Right Lane
            #cv2.line(img, (x1, y1), (x2, y2), color, 2)
            right_lines += [(x1, y1, x2, y2, slope)]
```

```
# Start lanes from bottom of the image
# and extend to the top of ROI
imshape = img.shape
# x1, y1, x2, y2
left_lane = [0, imshape[0], 0, int(imshape[0]/2 + 90)]
right_lane = [0, imshape[0], 0, int(imshape[0]/2 + 90)]
```

```
# Calculate X co-ordinates using average slope and C intercepts
# y = mx + c; x = (y - c) / m
if len(left_lines):
    left_lines_avg = np.mean(left_lines, axis=0)
    # c = y1 - slope * x1
    left_c_x1 = left_lines_avg[1] - left_lines_avg[4] * left_lines_avg[0]
    left_c_x2 = left_lines_avg[3] - left_lines_avg[4] * left_lines_avg[2]
    # x1 = y1 - c / slope
    left_lane[0] = int((left_lane[1] - left_c_x1) / left_lines_avg[4])
    # x2 = y2 - c / slope
    left_lane[2] = int((left_lane[3] - left_c_x2) / left_lines_avg[4])
    left_lanes_history.append(left_lane)

if len(right_lines):
    right_lines_avg = np.mean(right_lines, axis=0)
    # c = y1 - slope * x1
    right_c_x1 = right_lines_avg[1] - right_lines_avg[4] * right_lines_avg[0]
    right_c_x2 = right_lines_avg[3] - right_lines_avg[4] * right_lines_avg[2]
    # x1 = y1 - c / slope
    right_lane[0] = int((right_lane[1] - right_c_x1) / right_lines_avg[4])
    # x2 = y2 - c / slope
    right_lane[2] = int((right_lane[3] - right_c_x2) / right_lines_avg[4])
    right_lanes_history.append(right_lane)
```

```
# Perform a moving average over the previously detected lane lines to
# smooth out the line and also to cover up for any missing lines
if len(left_lanes_history):
    moving_avg_left_lane = moving_average(left_lanes_history, 10)
    cv2.line(img, (moving_avg_left_lane[0], moving_avg_left_lane[1]),
                  (moving_avg_left_lane[2], moving_avg_left_lane[3]), color, thickness)

if len(right_lanes_history):
    moving_avg_right_lane = moving_average(right_lanes_history, 10)
    cv2.line(img, (moving_avg_right_lane[0], moving_avg_right_lane[1]),
                  (moving_avg_right_lane[2], moving_avg_right_lane[3]), color, thickness)
```

So put them together, I get the test images as follows.



3.4 Pipeline for single image

The pipeline is defined as follows:

1. read an image
2. set the color threshold for R, G and B
3. define the region of interest
4. `image_select` only containes the lane information
5. convert RGB image to gray
6. canny transfer
7. gaussian blur
8. hough lines
9. output image

3.5 Test on videos

I define the single image pipeline as a function `process_image(image)`. Simply using `moviepy` and `IPython.display` (for displaying on HTML) packages.

```
white_output = 'test_videos_output/solidWhiteRight.mp4'
## To speed up the testing process you may want to try your pipeline on a shorter subclip of
## the video
## To do so add .subclip(start_second,end_second) to the end of the line below
## Where start_second and end_second are integer values representing the start and end of the
## subclip
## You may also uncomment the following line for a subclip of the first 5 seconds
##clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4").subclip(0,5)
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)
```

4. Reflection

4.1 Identify potential shortcomings with current pipeline

Based the output videos, the lane line detection is not stable with slightly swing.

The other shortcoming is when the lane is in shadow (ChallengeVideo), the lane lines are not able to be detected.

4.2 Suggest possible improvements to pipeline

A possible improvement would be to use some color space such as HLS to get the lane lines clearer.

Another potential improvement could be that the function `draw_lines` can have better algorithm to get the lines.