Finding Lane Lines on the Road

The goals / steps of this project are the following:
* Make a pipeline that finds lane lines on the road
* Reflect on your work in a written report

Reflection

1. Describe your pipeline. As part of the description, explain how you modified the draw_lines() function.

My pipeline of processing the image is based on how it was done in the lecture:

After making a working copy of the image, it is converted into grayscale. In the grayscale function, that was given, I added following lines:

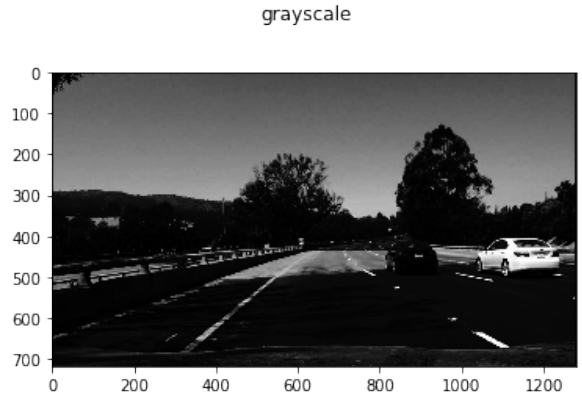
```
pavement_color=np.median(img[int(imshape[0]/1.6):int(imshape[0]/1.1),int(imshap
e[1]/2.4):int(imshape[1]/1.7143),:], axis=(0,1))+[20,40,255]
    for c in range(0, 2):
        img[:,:,c]=cv2.subtract(img[:,:,c],pavement_color[c],dtype=-
1)+pavement_color[c]
        img[:,:,c]=(img[:,:,c]-pavement_color[c])/(255-pavement_color[c])*255
```

In the first line a rectangular area just in front of the car is taken from the image and a median value is calculated for each color-channel. This can be seen as the color of the road right in front of the car. Then `[20,40,255]` are added to the channels respectively.

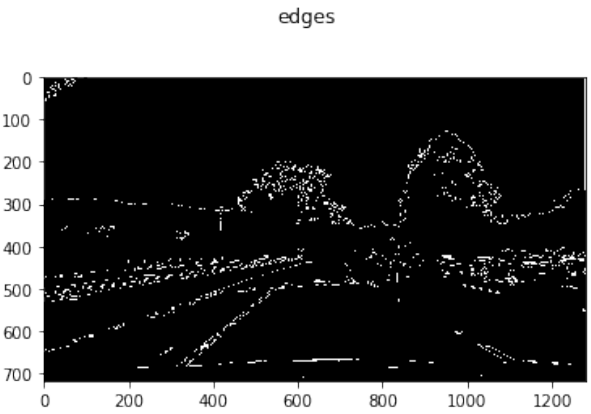
The resulting values are subtracted from the color image using the cv2.subtract function, so that negative pixel-values are automatically set to zero. Then the same values are added again, resulting in an image without anything darker than the road surface color.

In the final step the remaining color range is spread over the whole 8bit-range

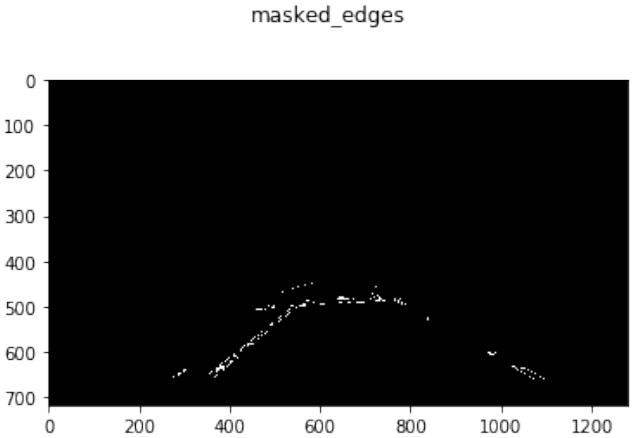
By using `[20,40,255]` as an addition to the road median color, I ensured that cracks and irregularities in the pavement are “swallowed” by this process. By choosing 255 for the blue channel it is omitted in the calculation of the gray image later on, as it is making the yellow (=not blue) line darker on the resulting gray image and is particularly prone to shadows on the darker pavement in the “challenge”-video.



In the next step the the resulting gray image is smoothed using the gaussian blur with kernel_size=5. The smoothed image is then fed to the canny edge-detector using low_threshold = 50 and high_threshold = 150.



Corresponding to the lecture, I defined a trapezoid mask that includes the road in front of the car but omits everything else when it is bit-wise multiplied with the image of the edges. As the “challenge”-video has a different resolution that the other videos, I defined the area relative to the image size.



The masked image of the edges is then fed into the hough_line-detector. I used following parameters:

```
rho = 2
theta = np.pi/180
threshold = 50
min_line_length = 10
max_line_gap = 50
```

Inside of the hough function the “draw_lines” function is called, to which the detected lines are given.
I added following code:

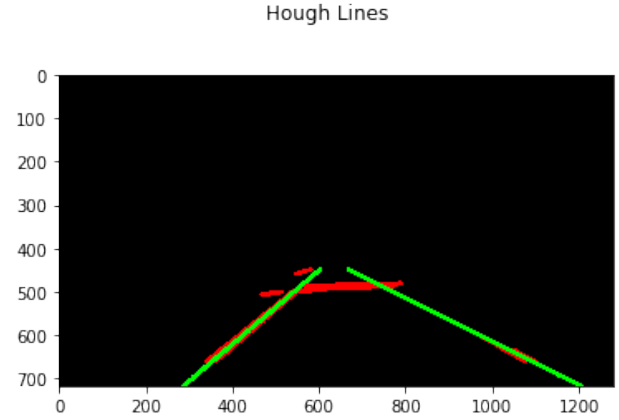
```
    horizontal_center=int(img.shape[1]/2)
    slope=0
    center_of_line=0
    right_line_x=[]
    right_line_y=[]
    left_line_x=[]
    left_line_y=[]

    for line in lines:
        for x1,y1,x2,y2 in line:
            with np.errstate(divide='ignore', invalid='ignore'):
                slope=(y2-y1)/float(x2-x1)
                center_of_line=(x1+x2)/2
            if (0.3 < slope < 1.5)and center_of_line>horizontal_center:
                right_line_x.append(x1)
                right_line_x.append(x2)
                right_line_y.append(y1)
                right_line_y.append(y2)
                cv2.line(img, (x1, y1), (x2, y2), color, thickness)
            elif (-0.3 > slope>-1.5) and center_of_line<horizontal_center:
                left_line_x.append(x1)
                left_line_x.append(x2)
                left_line_y.append(y1)
                left_line_y.append(y2)
                cv2.line(img, (x1, y1), (x2, y2), color, thickness)
            else:
                cv2.line(img, (x1, y1), (x2, y2), [0,0,255], thickness)



    if not(not(left_line_x) or not(right_line_x)):
        mr,br = np.polyfit(right_line_x, right_line_y, 1)
        ml,bl = np.polyfit(left_line_x, left_line_y, 1)

        cv2.line(img, (int((img.shape[0]-br)/mr), img.shape[0]),
(int((img.shape[0]/1.6-br)/mr),int(img.shape[0]/1.6)), [0, 255, 0], thickness)
        cv2.line(img, (int((img.shape[0]-bl)/ml), img.shape[0]),
(int((img.shape[0]/1.6-bl)/ml),int(img.shape[0]/1.6)), [0, 255, 0], thickness)
```

Here I created two “buckets” that a line falls into if it is part of either the left lane marking or the right lane marking. For this it had to fullfil two requirements:

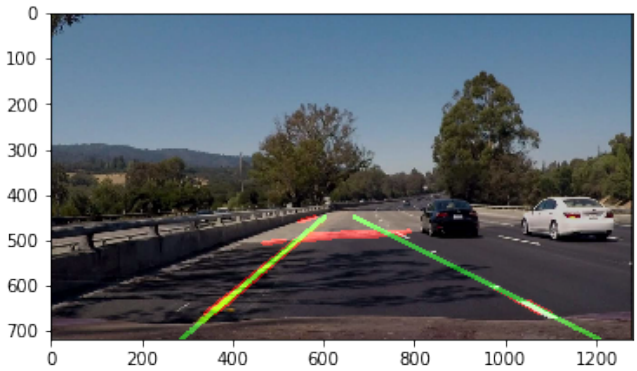
- the (horizontal) center of the line had to be on one side of the image, where...
- its slope had to be in a certain range.

I found by trial error that all lines that come from detected lane markings are within a small range of slopes (0.5 to 1.2).

Then the end-points of the lines are collected for each side separately and fed into a linear regression, in order to find the parameters of the final line on each side. Two lines with the final parameters are plotted in a lower part of the image.



Finally the plotted lines are blended with the input image using the “weighted_img function”



The images used here are from a screenshot of the “challenge” video. With shadows and change in pavement it represents the most difficult data this algorithm has to work with. While the line detection makes some errors, it accurately filters out the lines not belonging to the lane markings.

In the resulting raw-videos lines that neither belong to left nor right lane markings are shown blue.

2. Identify potential shortcomings with your current pipeline

- Using the median on each color channel of the image in the grayscale-function is quite computationally expensive as (to my knowledge) it includes a sorting algorithm.
- Unfortunatelly I was forced to use the np.polyfit function. First I just averaged the detected lines for each side (code is commented out in the file) but this resulted in a lot less robust overall line detection. The result was “wiggly” in the video

- There is at least one frame (at ca. 53% of the “challenge” video) where not a single line could be detected. While it would not be dramatic for a selfdriving car to be “blind” for 1/20s it shows the limit of the robustness of this algorithm.

3. Suggest possible improvements to your pipeline

- A possible improvement would be to use a custom kernel in the edge detection algorithm that puts more emphasis on vertical edges. This would make the following steps easier, as many of the horizontal edges (change in pavement, shadows…) are neglected from the beginning.
- Another potential improvement could be to take information from previous frames into account when calculating the new lane marking position. This would add to the robustness of the algorithm.
- Using the np.polyfit function it should be possible to do a higher order fit inorder to correctly detect the curvature of the lane. I am not sure about the robustness of the extrapolation into the distance, though.