Finding Lane Lines on the Road Writeup

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

The reference rubric can be found here

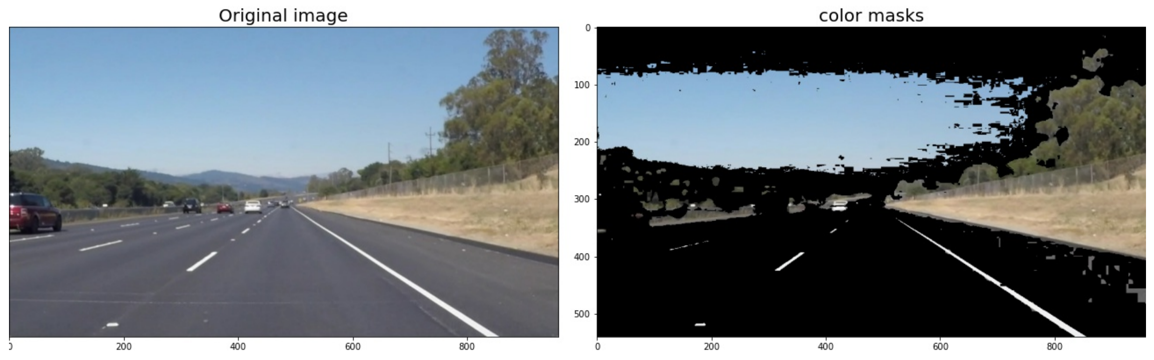
Lane finding pipeline

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

My pipeline consisted of 7 steps.

1. I use the inRange function to apply yellow and white masks to the input image

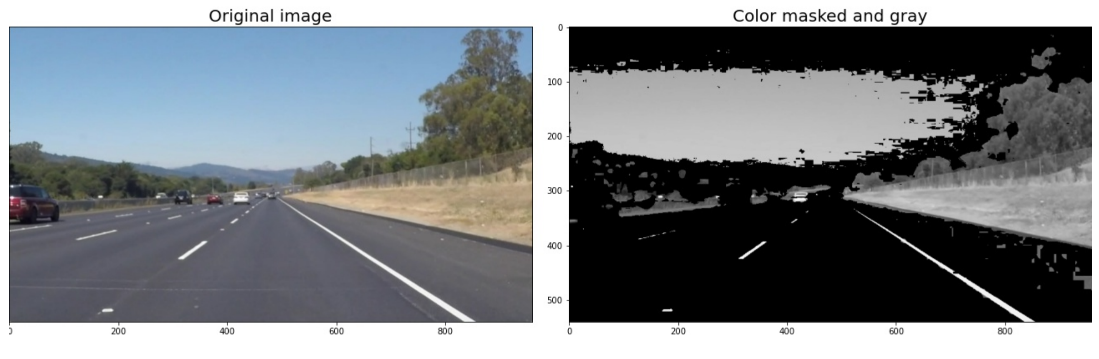
Since the lines on the road are either yellow or white, I applied a mask to filter out other colors. I used the inRange function from open cv to do that. The result of the pipeline is quite sensitive to these parameters, and I spent quite some time on finding good parameter set for the images before moving on to the videos. The figure below shows the result of the color masking for the test image "solidWhiteRight". All images are submitted too (in the test_images_output folder). The figure shows that I did not crop the image, yet and the color filter keep large portion of the sky



and the meadow next to the road.

2. I converted the masked image to grayscale

This is a straightforward process as shown in the class. The figure below shows the same image as above after applying a grey filter.

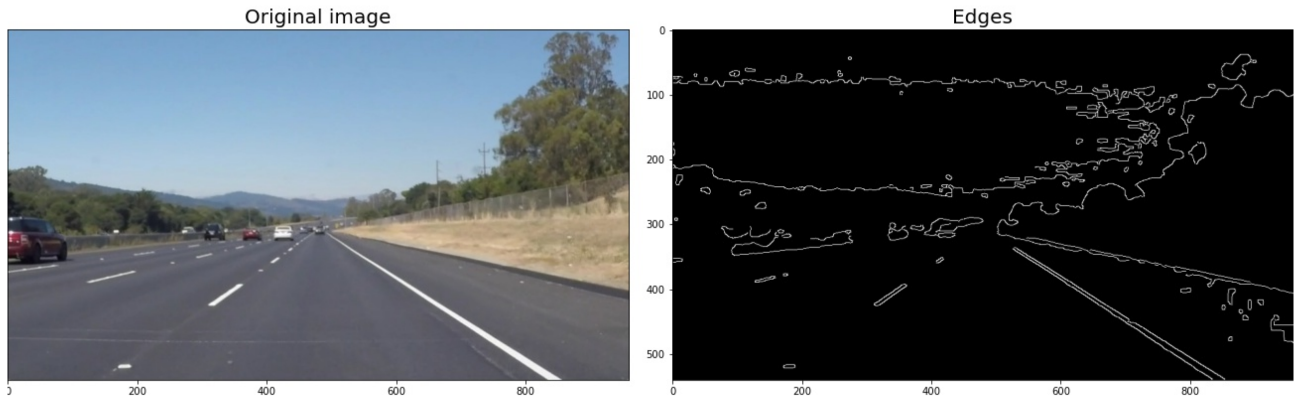


3. I applied gaussian blur

I used different values for the kernel size. I used the helper tool from one of the recommended articles (https://github.com/maunesh/opencv-gui-helper-tool) and chose my preferred values. I used kernel size 9 at the end.

4. I detected the image edges with the Canny detector

I tried different threshold values, low=90 and high=200 seem good enough (also visually suggested by the tool mentioned above). The result is shown in the picture below.

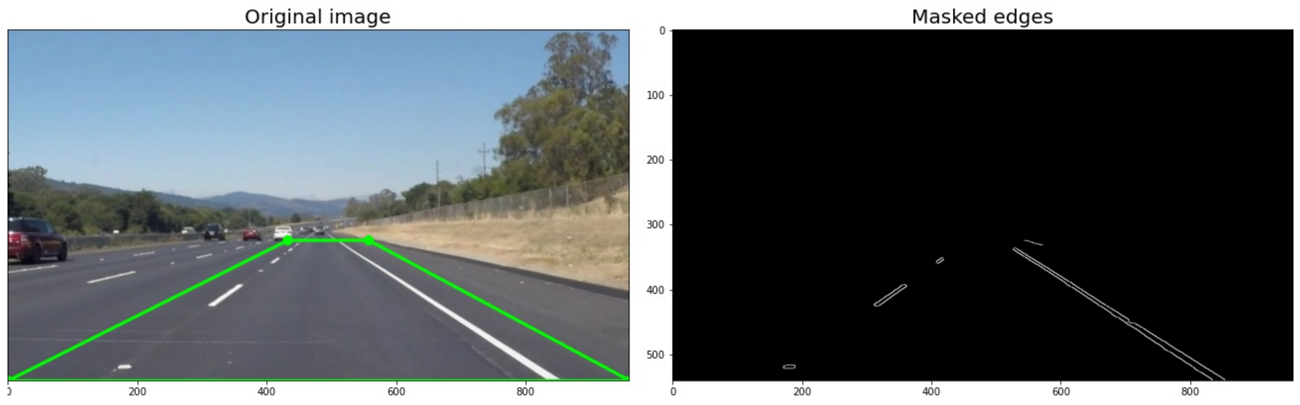


5. I defined the region of interest (RoI) for our usecase (a trapezius)

I decided to define a trapezoid polygon. I defined the 4 vertices as P1, to P4 and used them later in the code. I tried different values as proportion to the image size. I adapted the position of P1 to P4 as a proportion of the image size so that the RoI logic also works with the optional video. The figure below shows the trapezius shape (on the left hand side) and the result of cropping the image with the detected edges.

6. I detected lines within RoI (using the Hough transform) & drew detected lines

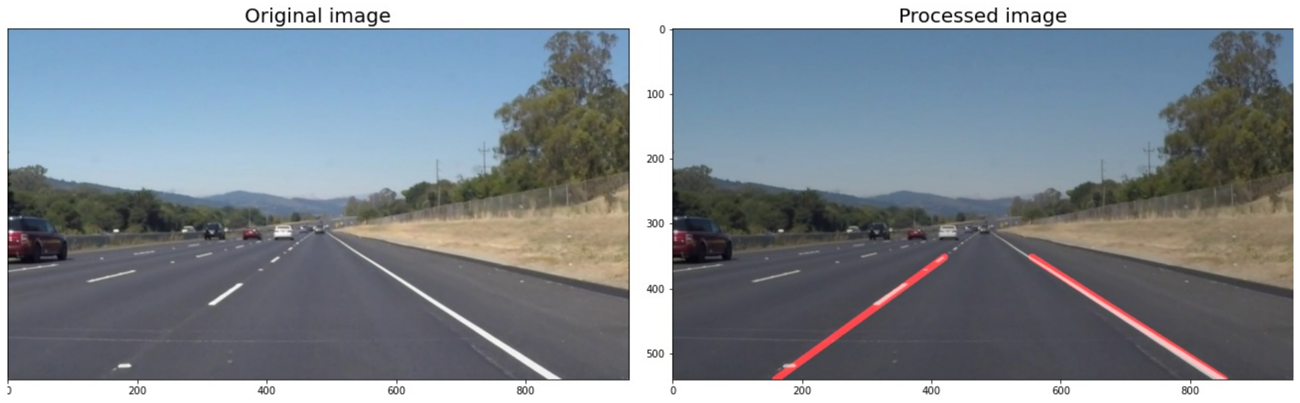
I tried a few different parameters especially resolution (distance and angular) but also the minimum number of votes for a line to be detected. The final parameters works ok in my opinion but I have hard time with dashed lines. In fact a much higher min vote value results in more robust solid lane detection but poor dashed line detection.



7. I overlay the detected lanes with the original image

This is a straightforward process as shown in the class.
I
n order to draw a single line on the left and right lanes, I modified the draw_lines() function mainly looking at the slope of each hough line, as suggested. All lines on the right of the image have positive slope, while the lines on the left have negative slope (that's because of the orientation of y being from the top to the bottom). Each image of the video has its own set of lines. So each image that goes through the pipeline results in a set of right lines and a set of left lines. The size of each set varies frame by frame as well as the deviation between slope values within each set. In fact, the color filter the the choice of the parameter perform in different ways depending on each frame.

After separating the left and right lines, I filter out some outliers from the slope with some hardcoded threshold values (I observed that most of the slopes have around +-0.6 or +-0.8. These hardcoded choices are probably a poor idea in the real world. The result of the pipeline for the example image is shown below.
After filtering out the outliers, I add all the coordinates of each set into some arrays so that I can build a first degree polynomial based on the the fitted lines. To build the first degree polynomial I use the bumpy function “polyfit” first and then I run the function “poly1d“ also from numpy. If the poly1d function returns a line (again, left and right lines are treated separately), I add it to a line buffer. The size of the buffer is set to 5 (it seems to be a good value for both first videos but it is



not enough for the challenging video, where I used 10). Until the buffer is not full, I draw the result of the poly1d and I append the values to the buffer, when the buffer is full, I average its values and plot the average/smoothed line. The result of the smoothing process is similar to the example video “P1_example.mp4”.

The resulting pipeline works ok in my opinion for both videos “solidWhiteRight.mp4” and “solidYellowLeft.mp4”. The pipeline also works for the challenging video but the performance is not so good because of the presence of shadows and different pavement colors. All videos are submitted and available in the folder “test_videos_output”

Reflection
2. Identify potential shortcomings with your current pipeline

I decided to enumerate the shortcomings so that I can refer to them in the improvement section.

Shortcoming 1-parameters: parameters tuning is probably the activity where I spent most of the time. In fact the choice of the parameters of each step of the pipeline has a cascade affect on the following steps. I think that the parameters that I used are ok, but there is room for improvements.

Even if I had found better parameters, the pipeline I suggested is not adaptive which means that new training images would require new manual parameter settings.

Shortcoming 2-shape: my solution assumes straight lines ahead and does not adapt the shape of the lane detection in case of sharp curves. Smooth curves are not an issue but the sharp curves are. Which is also shown in the challenge video.

Shortcoming 3-shadows and pavement color: sudden color changes on the the driving surface show the limitation of the color masks I applied at the beginning of the pipeline. Such shortcoming is visible in the challenging video where despite the smoothing, the detected lines are quite unstable.

Shortcoming 4-dashed/solid lanes: my pipeline does not really treat dashed lines and solid lines in a differ way.

Shortcoming 5-dirt: this algorithm is fully dependent on one single camera input, if the lens is obscured (e.g. direct light, dirt, fog, maybe strong rain etc.) the car might have hard time to detect the lane lines.

3. Suggest possible improvements to your pipeline

Possible to improvements

Shortcoming 1-parameters: parameter selection requires more experience and exploration. There are many resources online to understand the parameter space and the way they effect each step of the pipeline. Parameters are also depending on the lighting conditions of the original image, so more complex calibration rules and functions should be applied for example by detecting the brightness of the scene. Over night, for example, the parameter choice would be completely different.

Shortcoming 2-shape: I should change the way define the edges of the ROI. In a curve the shape is different. In the real world if I had access to a reliable positioning source (GNSS, RTK or cellular) and access to HD maps, I would know where I am on the road and predict the shape curves ahead.

Shortcoming3-shadows and color: the way I apply color masking is not robust enough. In fact even averaging the last 10 lines results in unstable line detections. A possible improvement of the pipeline is use better color filters to reduce the dependency on brightness changes and shadows.

Shortcoming 4-dashed/solid lanes: I observed that it should not be difficult to build two different extrapolators one for dashed lines and one for solid lines looking at their length.

Shortcoming 5-dirt: this is probably very hard to solve with software solution if not impossible to solve. I saw some solutions from Waymo to physically swipe the sensor surfaces every now and then.