Lane lines on the road

Author: Andreas Scharf September 17th 2017

Contents

1 Reflection

1.1 Description of pipeline

In the following subsections, I briefly discuss the development of my pipeline and outline the different implementation steps of my code located in `./../MySource/LandFinder.py.`

1.1.1 Initial pipeline

My initial pipeline consisted of the 6 steps (since I am used to work in *PyDev-Eclispe* I overlooked the notebook file 'P1.ipynb' until the end) introduced in the lessons of this project.

1. Converting the colored image to gray colored image.
2. Applying a smoothing of the picture to reduce noise effects.
3. Apply an edge detection algorithm, here the Canny algorithm.
4. Defining a region of interest, where lines forming road lanes should be identified.
5. Using the Hough transform to identify lines in the image
6. Draw the resulting lines in the picture (draw one line for each lane line).

This approach worked quite well for the test-images while the results for the videos were rather bad.

1.1.2 Color transformations

Due to e.g. tree shadows a lot of irrelevant lines were identified and due to low contrast, the lane lines were partly not visible (see Figures 1 and 2).

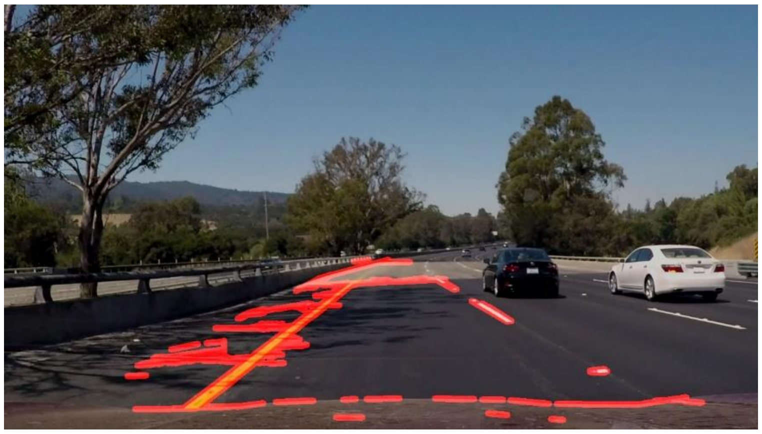

Figure 1: Noise lines from shadows and wind shield.

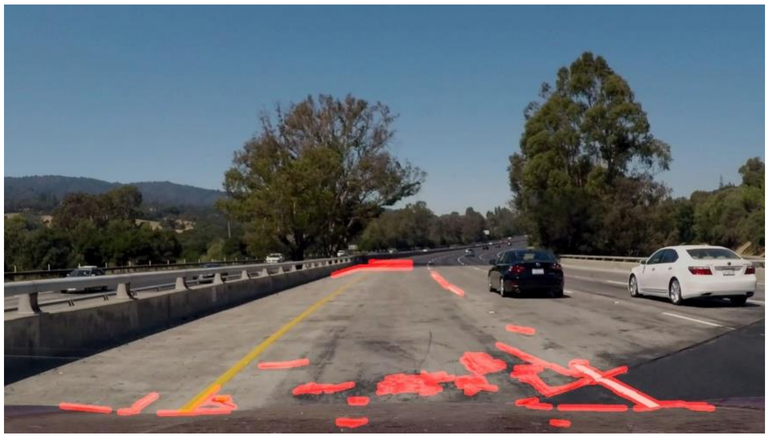

Figure 2: Left lane line is not recognized due to low contrast.

Therefor I implemented

a) A conversion to the HSV color space to increase contrast.
b) Color masks to identify white and yellow objects in the images.

before step 1. (converting to gray colored image). For this purpose, I introduced a (static-like) class **`ColorTransforms`**.

With these measures, the identification of the relevant lines in all videos improved significantly (see Figures 3 and 4 and compare with Figures 1 and 2).

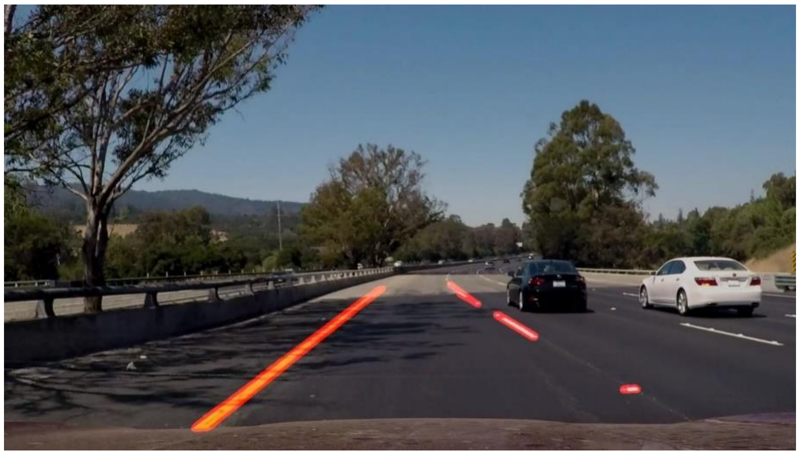

Figure 3: Same as Figure 1. Noise significantly reduced.

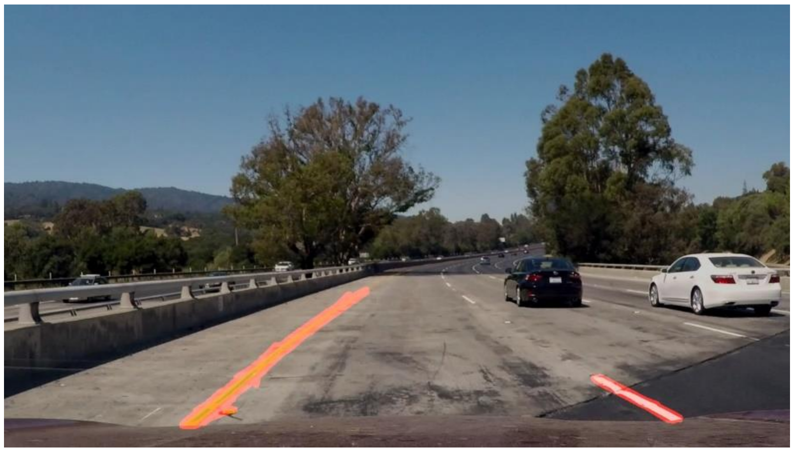

Figure 4: Same as Figure 2. Left lane line is identified

1.1.3 Draw single line for each lane line

To draw a single line on each lane line I assumed first that the lane lines can be represented by straight lines. In respect to the image coordinate system

- the left lane line corresponds to a straight line with negative slope
- the right lane line corresponds to a straight line with positive slope

Therefor I sorted the lines in each image according to their slope and filled arrays with the corresponding slopes and intercepts. From the slope- and intercept- arrays I calculated the mean values and used these mean values to construct the final lane lines. The corresponding method is **`IdentifyLanes.`** For test-images this approach produces satisfying results. For the video files “solidWhiteRight.mp4” and “solidYellowLeft.mp4” we find large variations in quality (see Figure 5 for some bas quality)

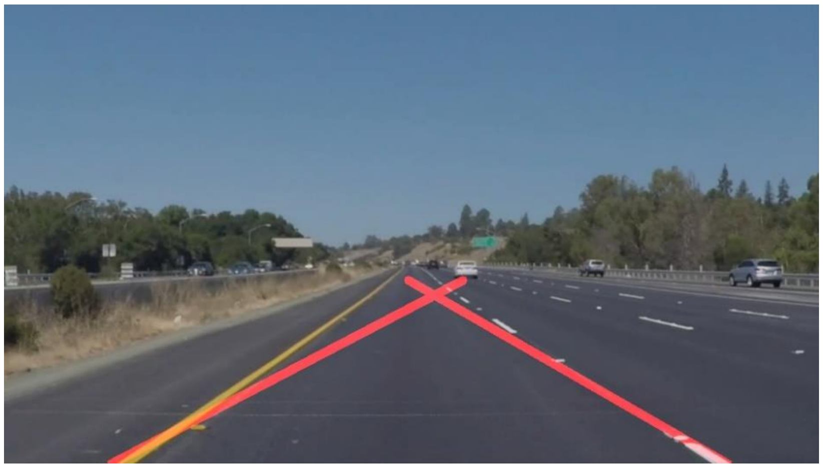
Figure 5: Deviations between real lane lines and identified lane lines

To reduce these deviations, I implemented an angle filtering in the method **`ApplyAngleFiltering`**. Under the assumption that the left/right lane line slopes correspond to the angles -45/+45 degrees I eliminated all lines which do not fulfill the condition:

$$30° < |\theta| < 60°$$

with θ being the angle of the line. The improvement on the projected lane lines is shown in Figure 6.

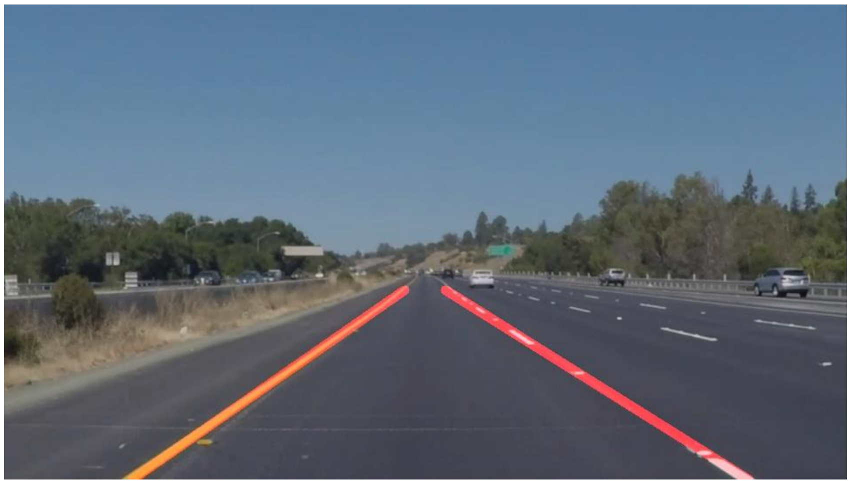
Figure 6: Same as Figure 5 but with angle filtering

1.1.4 Non-linear fit

For the challenge-video the linear fit is still not suitable, as can be seen in Figure 7.

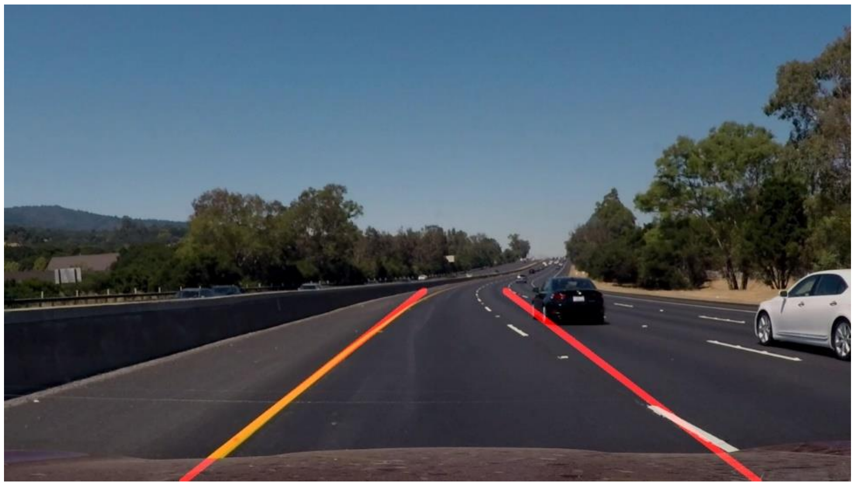

Figure 7: Linear fit result on challenge video

Instead of a linear fit I tried to use numpy's polynomial fit routine and used the (x,y) value pairs of all identified lines. Again, I distinguished between left and right lane lines, this time by just splitting each image into a left and a right part using the size of the image in x direction. The implementation of this approach can be found in the method **`IdentifyLinesWithPolynom`**. Using numpy's (linear) polynomial fit could also be used instead of the method **`IdentifyLanes`**. After generating the fit function, I used the original x values of the lines to obtain the fitted y_{fit} values. The result is shown in Figure 8.

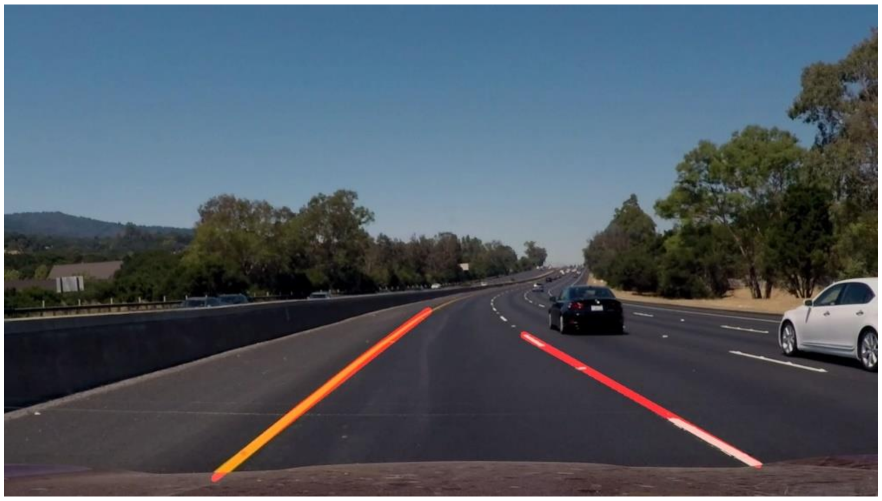

Figure 8: Same as Figure 7 using a polynomial of degree two.

I also tested the quadratic fit on the other videos and the test pictures and found very good results. My efforts to interpolate between the fitted line segments were not successful. Consequently, I cannot show two resulting lane lines for quadratic fits. In the method **`ProcessImage`** I implemented a switch between **`IdentifyLanes`** and **`IdentifyLinesWithPolynom`**. For test-pictures and the test-videos (solidYellowLeft.mp4' and 'solidWhiteRight.mp4) a linear fit (**`IdentifyLanes`**) is used resulting in two lane lines. For 'challenge.mp4' a quadratic fit (**`IdentifyLinesWithPolynom`**) is used resulting in several lane line segments.

1.1.5 Final pipeline

My final pipeline consists of the following steps:

1) Converting the image from BGR space to HSV space
2) Applying a yellow and a white color mask
3) Converting the HSV image to a gray colored image
4) Applying a smoothing of the picture to reduce noise effects.
5) Apply an edge detection algorithm, here the Canny algorithm.
6) Defining a region of interest, where lines forming road lanes should be identified.
7) Using the Hough transform to identify lines in the image
8) Using a linear/quadratic fit to identify the lane lines.
9) Draw the resulting lines in the picture
   a) Draw one line for each lane line only implemented for linear fit.
   b) Draw line segments for quadratic fit

1.1.6 Running the code

To run my code go to the end of the file LaneFinder.py. Please assign the variables 'DefaultPath' and 'DefaultImagePath' with the path where the test images and videos are located. Then run the code. First all test pictures will be shown for one second, then all videos will be played.

2 Identify potential shortcomings with your current pipeline

My current pipeline has the following shortcomings:

- Weather conditions like fog/rain/snow would probably result in a very bad performance of my pipeline. The same is true if lane lines are missing or covered (sand or snow).
- Changing lanes could result in bad performance e.g. because of the angle filtering
- Hilly environments with steep roads or parking cars might hide parts of the lane lines. No algorithm is implemented to compensate.
- A descent interpolating mechanism for non-linear fits is missing.
- The quadratic fit might not be useful in all use cases, e.g. changing lanes.

From this incomplete list, it becomes clear that a lane finder needs to be flexible and adaptive regarding many environmental factors. Using fixed parameter values as in my pipeline is only suitable for selected test cases.

3 Suggest possible improvements to your pipeline

The first improvement to my pipeline is to implement an interpolating mechanism for the quadratic fit. I would also check if the single steps can be improved in general and in particular cases. Color masks should be adjusted according to daytime, weather and environment (e.g. tunnels or forests) using GPS information. The performance of filters other than the Gaussian can be examined. The region of interest can be optimized e.g. depending on the status of the steering wheel. Furthermore, alternatives to Canny algorithm and Hough transformation can be studied. The performance of other fit functions must be examined. Last but not least all steps (including the missing ones) have to be optimized, so they can run in real time on a suitable (embedded?) device.