Finding Lane Lines on the Road

Reflection

Description of the Pipeline

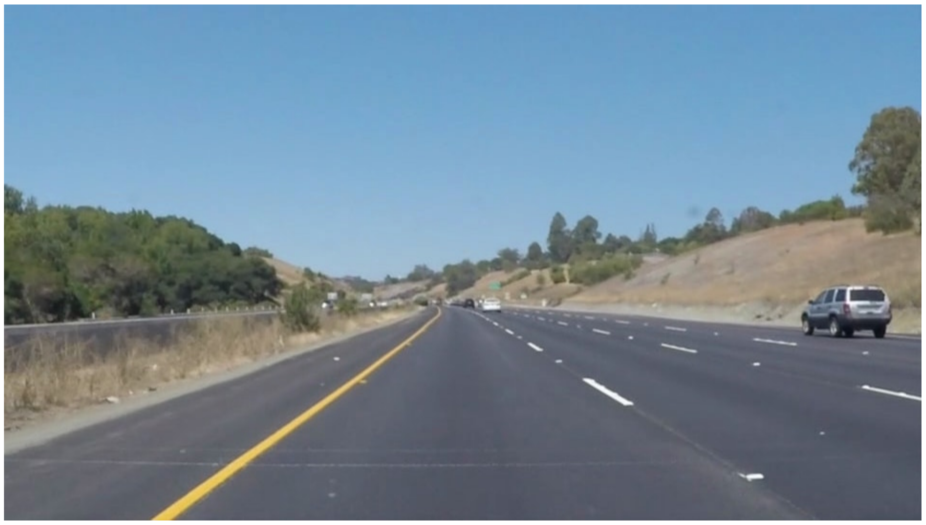


Figure 1: Original image

My first approach for defining the image processing pipeline was to make use of/play around with the functions already implemented in the jupyter notebook. Since the final result should display the original image with two lines on top of the lane markings I realized I can get rid of a lot of image information on the way of extracting these two lines. First I reduced the given image to the region of interest which will always lie in a trapeze form in the lower part of the image. With the coordinate system starting at the left upper corner (x from left to right and y from up to down) in mind I needed to play a bit with the numbers to get this result.

I further reduced the image by the color information since color is not important for drawing the lines. A simple grayscale function did this job.

When plotting and saving this first grayscale image vs. the original image I found out that opencv and matplotlib are using the exact opposite order of saving red green and blue values.

Note: When displaying an opencv image with matplotlib functions one needs to switch the order of rgb values with v.cvtColor(img, cv.COLOR_RGB2BGR)

My next step for extracting the two lines overlaying the lane markings was to make use of an edge detection. Since I already heard about the canny edge detection I just added this function to see what it would do. The function takes

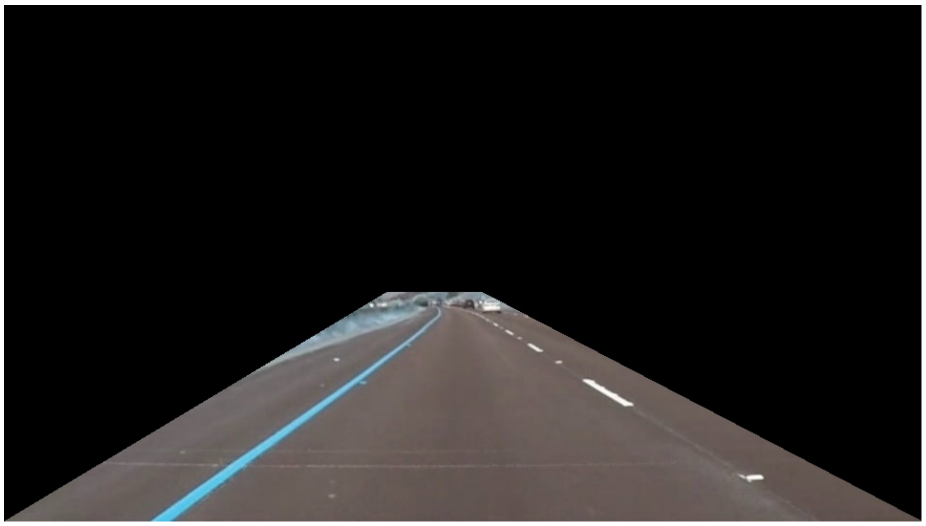
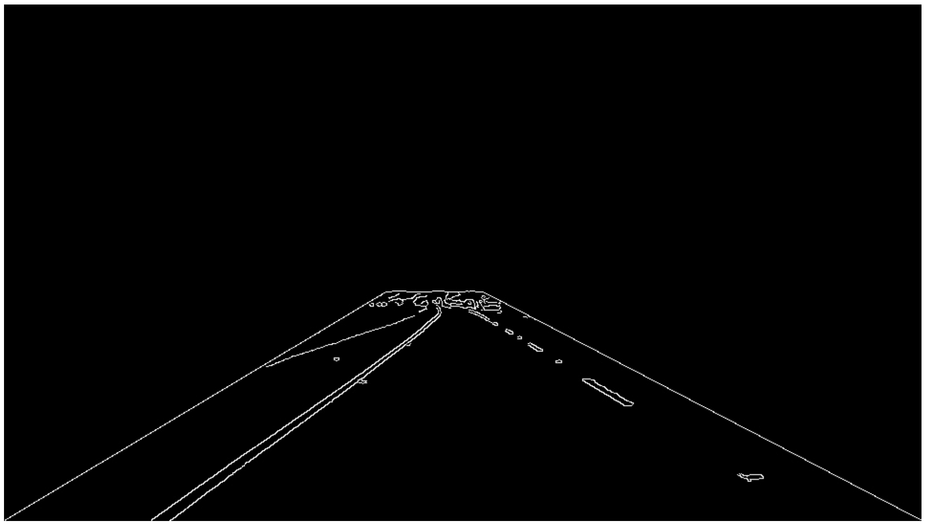


Figure 2: Trapez view

two parameters that I do not want to think about/optimize now but rather have the rough order of functions in my pipeline with parameters that are roughly in the right range first. I read up on the opencv python example about canny edge detection in the official documentation and voila I had two parameters to start with.



As can be seen the edge detection detected not only the lane markings but also the edges of the trapeze which made me change the order of my pipeline. I moved the trapeze function to run after the edge detector, which gave me more of what I wanted.

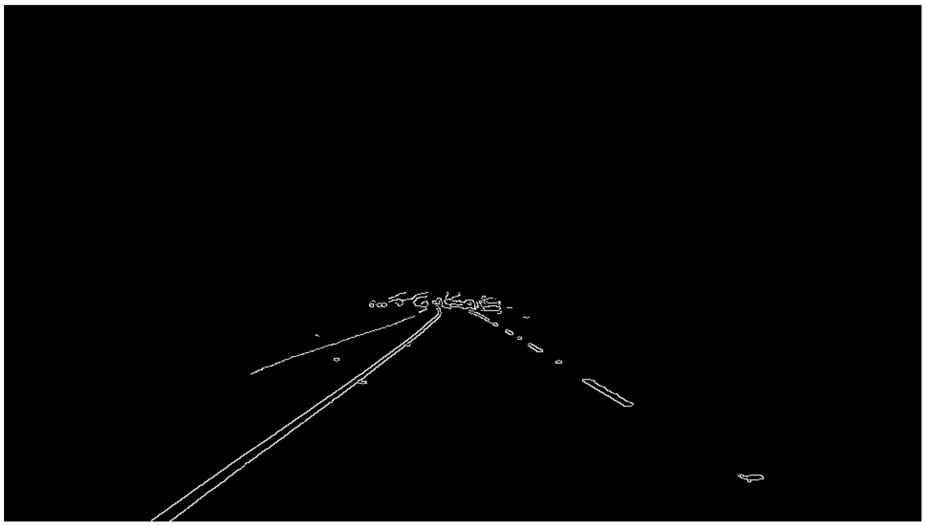


Figure 3: Canny edge detection - optimized

The result looked actually quite ok, but still it had a lot of points rather than long lines, which is what I want to have in the end. Using a smoothening of this rather pixely output was the logical next step. Since gaussian was implemented in the jupyter notebook I went for this smoothening function. In the opencv doc I found a parameter for the kernel size to start with. I got this result, which looks a lot more like real lines.

My next and last step of my image processing pipeline was to convert these pixel lines into mathematical lines, defined by the two parameters m and t (y = mx+t) so I can draw them myself on any picture or video.

Therefore I applied the hough_lines function which is doing exactly that, converting pixel lines into mathematical lines. I took the function's parameters again from the opencv documentation. With the output of the hough transformation I can draw the lines myself now and overlay them on top of the original image.

Looking at my result I needed to do the following optimization steps: - optimize the region of interest, because the end of the road on the left is detected from the edge detection and/or - understand and optimize the parameters of the pipeline functions to get a complete line on the left

Reading up on the parameters of canny edge detection, gaussian blur and hough lines, I got a very rough image of what to change, even though this does not make the trial and error process redundant. Looking further, I found a tool with some sliders for the parameters kernel size of the gaussian blur and the two thresholds of the canny edge detection. The image output was shown immediately to the screen. This made the trial and error process quite fast. I found out that the standard parameters of kernel_size=5, lower_threshold=100,

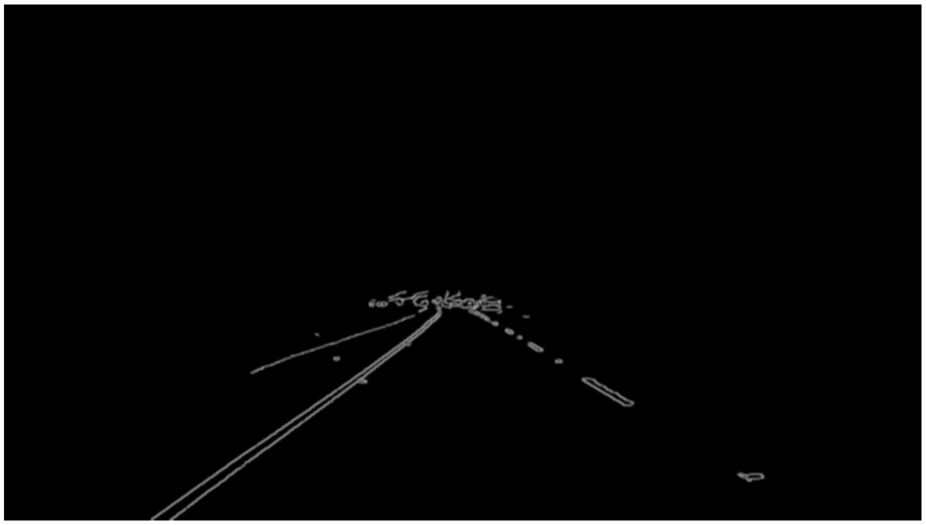


Figure 4: Gaussian blur

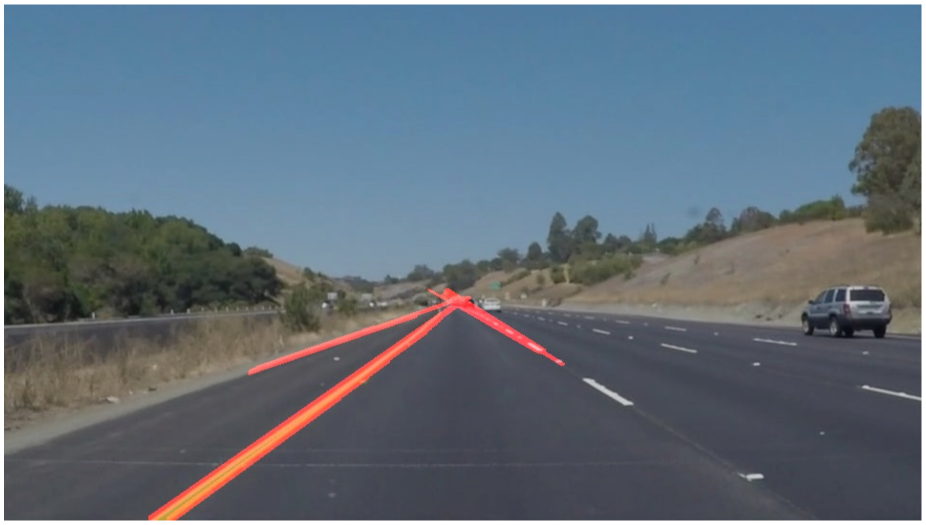


Figure 5: Hough transformation

upper_threshold=200 were not too bad after all.

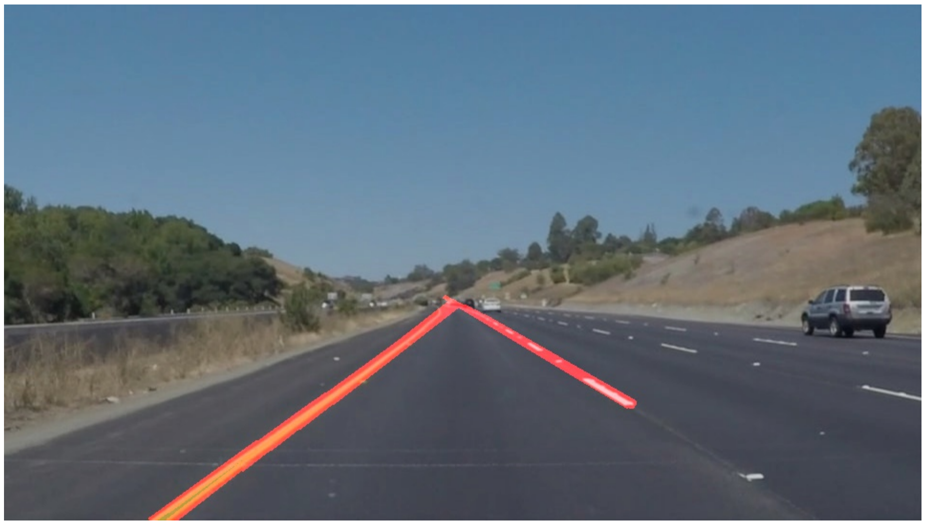


Figure 6: Optimized image

My optimization was already kind of fruitful but it is still not enough to show the right lane all the way to the bottom of the image.

I started thinking about extrapolating the right lane to the bottom. Since the probabilistic hough transform is giving me many lines instead of just one, I will take the mean of all the lines on the left as well as the mean of all the lines on the right. After some adjustments of the draw_lines() function (taking all gradients within a certain range that seems realistic –> get rid of all lines with gradient smaller than X and larger than Y) I got this as a result:

Looking at the other pictures I could see that the pipeline is quite robust for these.

2. Identification of possible shortcomings

The biggest shortcoming would probably be that all parameters are optimized on a couple of specific input images. As soon as the quality of the image or the type of environment changes results might be different.

When the brightness of the image changes significantly i.e. when in a tunnel, at night or when the camera is opposed to direct sunlight. In this case the lanes might not be detected. In general the pipeline will be sensitive to all additional noise in the input video stream, like rain, fog, direct sunlight, . . .

Another shortcoming could be if there are many additional lines within the region of the trapeze i.e. due to a car directly in front, or a double line on the

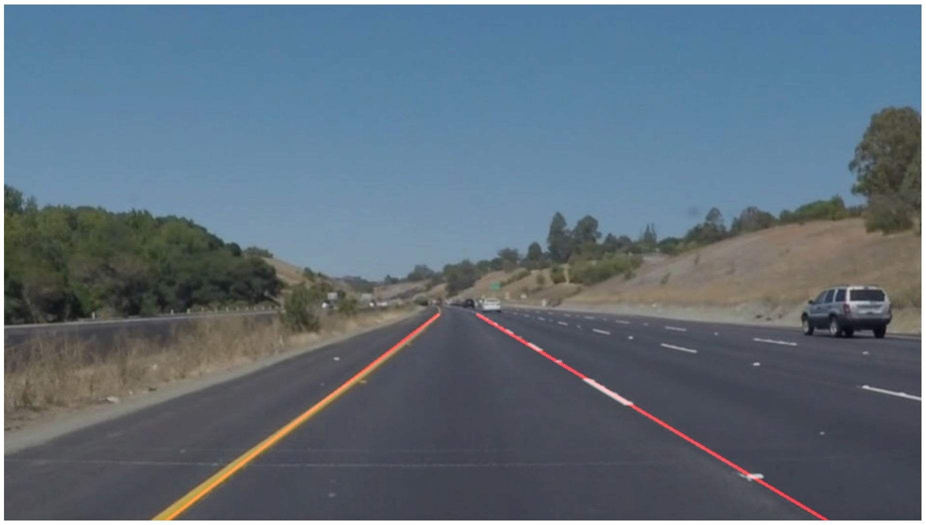


Figure 7: Final image - solid yellow

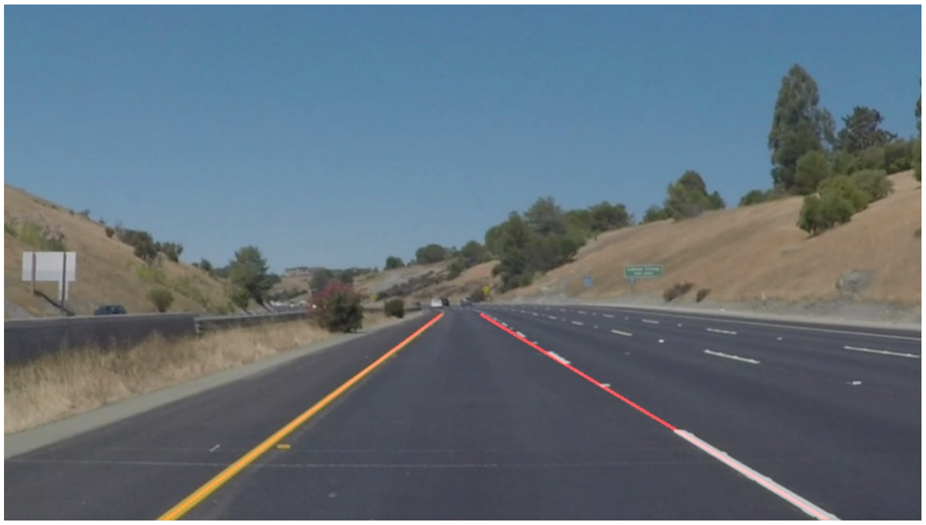


Figure 8: Final image - solid yellow 2

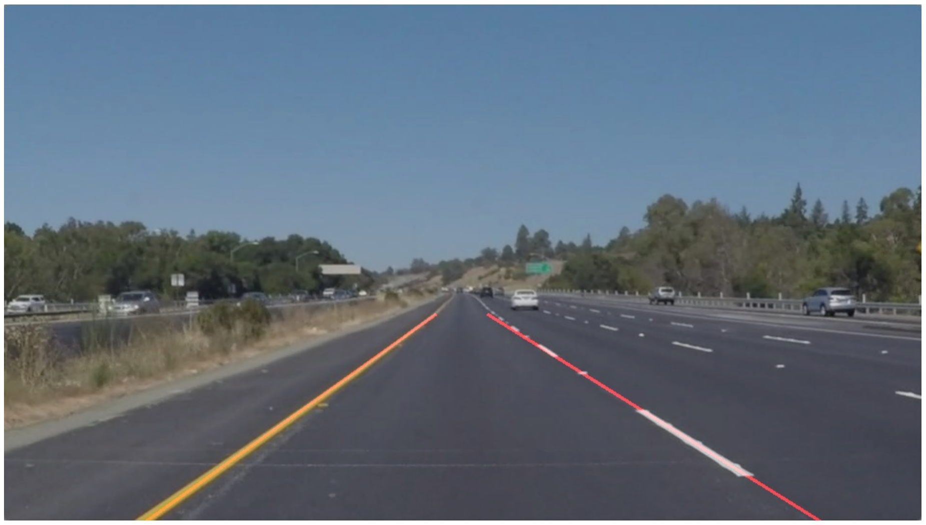


Figure 9: Final image - solid yellow 3

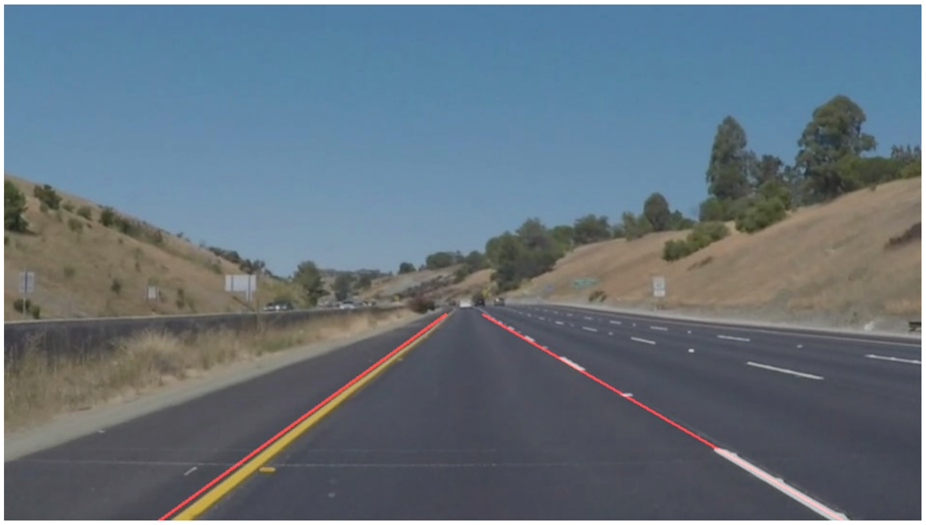


Figure 10: Final image - solid yellow 4

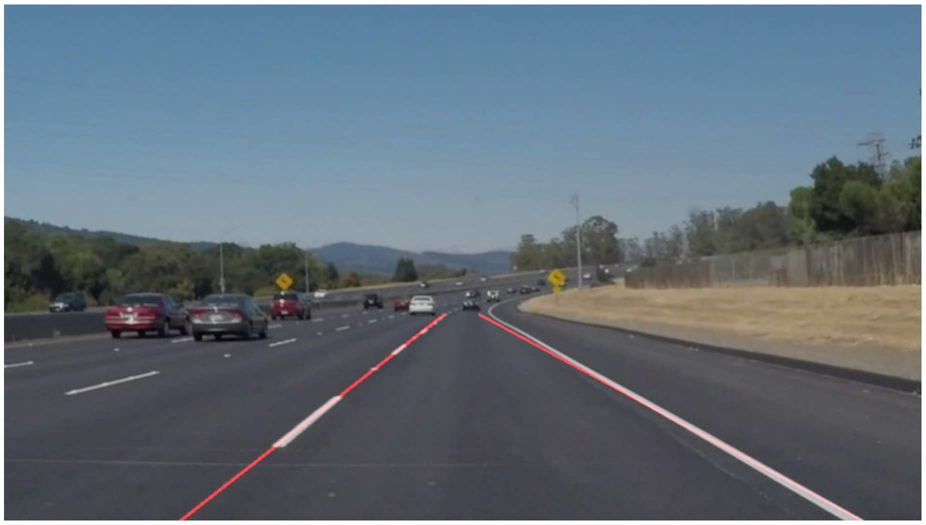


Figure 11: Final image - solid white

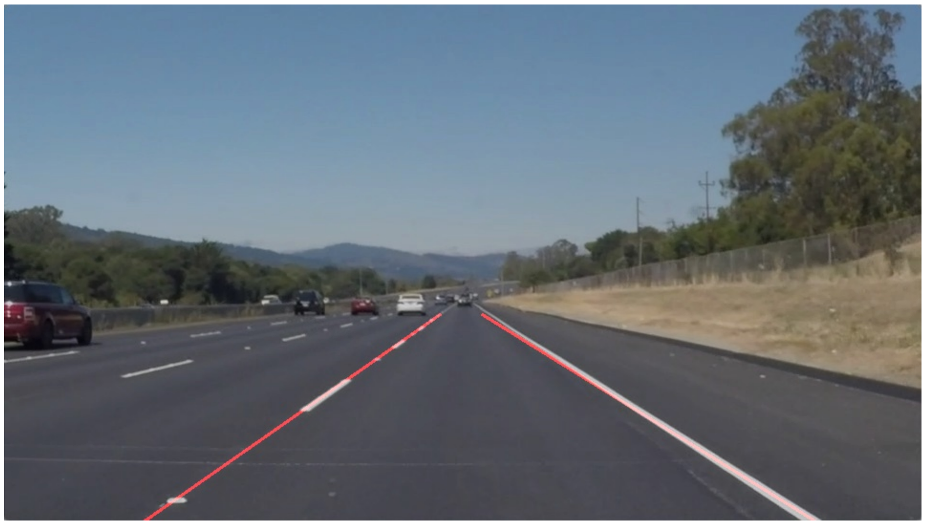


Figure 12: Final image - solid white 2

side. Sharp corners could also be difficult to detect. In general objects in front of the car will result in bad results.

3. Suggestions of possible improvements

A possible improvement would be to adjust the parameters of the image-processing functions (canny edge detection, hough transformation, gaussian blur, ...) if there is noise like rain, fog, direct sunlight, tunnel, night, ... i.e. if it is dark outside, the image brightness could be adjusted, etc.

Common objects in front of the car should be detected as such i.e. cars, trucks, roadworks, ... to avoid detecting false lines as lanes.

The draw_lines() function can be improved. As mentioned above I am taking the average of all the lines from the hough lines function. When there are no lines detected by the hough transformation then I will miss the input for the draw_lines function obviously. In this case it would be nice to take the calculated line from the previous image of the video. This could be done by defining a line object and only update the object values if they are valid.

One more possible improvement that I want to mention is to use polygens for the lanes instead of straight lines. This would probably improve the result on the challenge video.