Advanced Lane Finding Project

Student: Florian Wolf

The goals / steps of this project are the following:

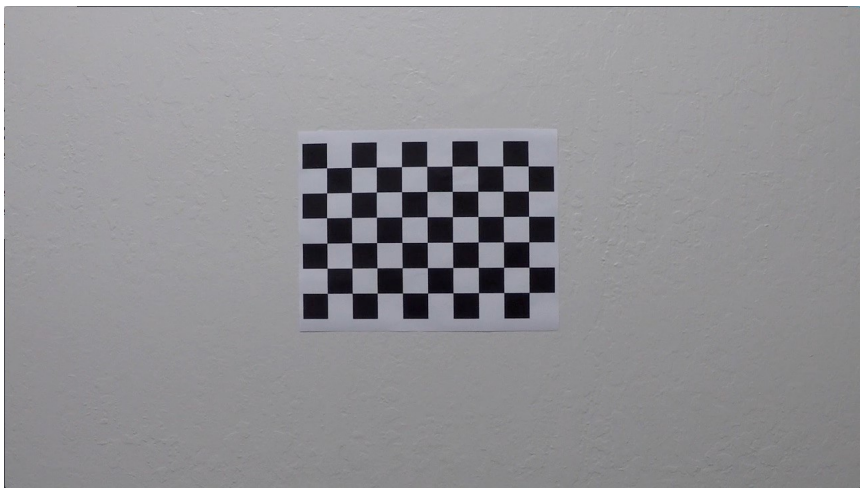
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Writeup

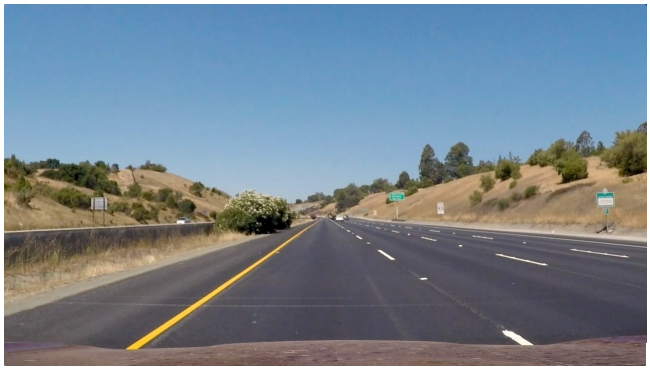
Here I will explain how I solved the "Advanced Lane Finding" project of the Self-Driving Car Engineer nanodegree by Udacity.

Camera Calibration

Every camera with a not-pinhole lens produces image distortion. There is radial and tangential distortion. Common distortion errors are stretched objects at the edges of the image, or the so called "fish eye" effect. To use an image as source for the steering angle calculation we need to get rid of this distortion errors. The open The open module provides useful tools for image distortion. cv2.findChessBoardCorners finds the corners of a chessboard in an image. The found corner points are saved as image points. Object points are created manually. This information is fed into the cv2.calibrateCamera function. With the resulting matrix the image can be undistorted by using cv2.undistord. For the camera calibration 20 different pictures of the same chessboard were used. Here is one example:



Here you can see an image of the front camera of the car, before and after undistortion:




Pipeline to process single images

1. Thresholding the image

The first step after loading the image into my pipeline is to identify pixels, that are likely to be part of the lane lines. Therefore several methods are used and combined. The Sobel operator in x and y direction is calculated and the image is thresholded for both. Only pixels that are in the threshold are passed as ones to a binary image. Then the same happens with the direction and the magnitude and the gradient. In the end the two binary images are combined. In addition the image is converted to gray-scale and a threshold is applied to filter the lighter areas of the image. To also catch the yellow lanes, the red channel of the image is isolated and a threshold applied. One last threshold is applied to the S (Saturation) channel of the HLS transformed image. All these binary outputted images are added up and the final output is flattened to be a binary image. Here is an example of how the output binary image looks like compared to the original image:


2. Correcting image distortion

The second step in the pipeline is to correct distortion of the image. To do so the procedure explained earlier is applied. The transformation matrix has been saved and is loaded only once to save time since calculating the transformation matrix is a memory expensive action.

3. Cropping the image to a region of interest

The binary image at this stage detects the lines pretty well but also a lot of other light or sharp edged objects. To focus on the lane lines we set all the pixels outside the chosen trapezoid to zero. The shape of the area of interest is:

(200,imshape[0]),(480, *ymia), (780, y*mid), (1200, imshape[0])

with

x_mid = imshape[1]/2

y_mid = 480

4. Perspective transform the image.

In order to retrieve the steering angle a perspective transformation of the image is required. The function cv2.getPerspectiveTransform takes in source and destination points and calculates the transformation matrix. The function cv2.wrapPerspective takes this matrix as input and transforms the image. To find the right source points a function was created, which transformed a test image of a straight line and calculated the angle. Once the lane lines on the transformed image were parallel and vertical, the source points were saved. The destination points were chosen as a rectangle. For the project video the following source and destination points resulted:

| Source | Destination |

|:----------:|:----------:|

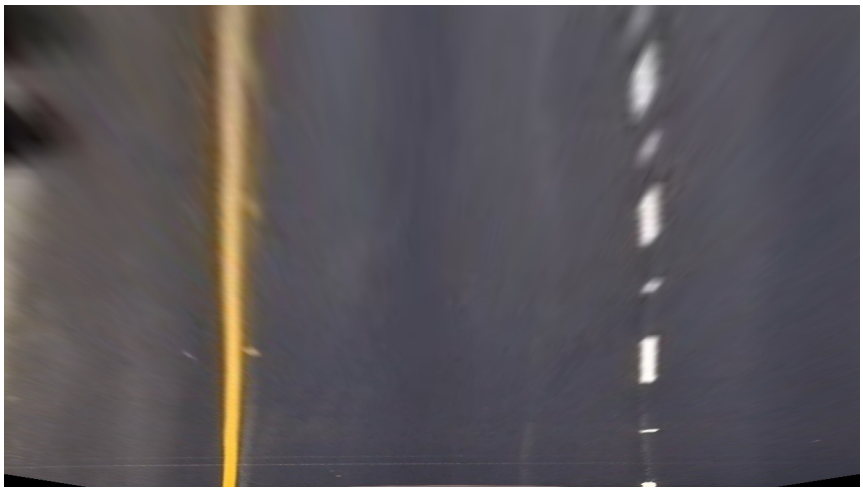
| 479 , 480 | 80, 450 |

| 799, 480 | 1200, 450 |

| 1200, 630 | 1200, 700 |

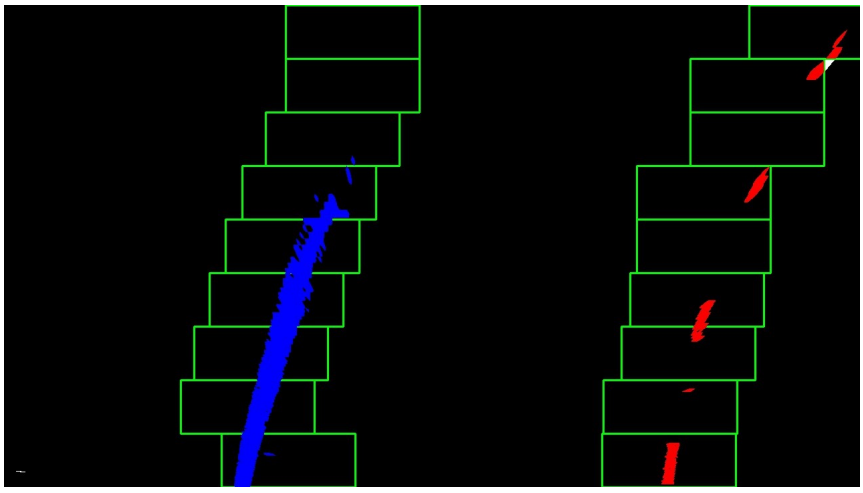
| 80, 630 | 80, 700 |

Here is an image after the transformation:



5. Identify lane line pixel and fit a second order poly line

To find the start of the lane lines, a histogram of the binary image is used. the two peaks indicate, where the most pixels accumulate. It is most likely that these two points are the starting points of the lane lines. The image is divided in two half to find the left and right line. The height of the images is divided by nine and each of the nine areas is searched for a maximum with rectangles of 100 pixels. For each of the nine sections the midpoints are connected to form a poly-line. The following image shows the procedure.



A class is created to store the previous results of the poly line. every new result is averages with the last four

results to avoid sudden changes. The area in between the poly-lines is plotted in green, transformed back to the original perspective and overlayed on the initial image. The steering angle is calculated with the following code:

> # Calculate the new radii of curvature left*curverad = ((1 + (2leftfitcr[0]*yeval*ymperpix + *leftfitcr[1])2)1.5) / np.absolute(2leftfitcr[0]) rightcurverad = ((1 + (2rightfitcr[0]*yeval*ymperpix + *rightfitcr[1])2)1.5) / np.absolute(2*right*fit*cr[0]) >> This relays on the following function:
>
> f(y)=sqr(Ay)+By+C

The radius of the left and the right lanes are printed on the image in red color at the upper left corner for every processed image. To calculate the position of the car compared to the center of the lane line, the center of the lane line had to be found. Therefore the the left and the right starting point of the lines were calculated using the polynomial functions of the lines and the maximum y value. Adding these two values and dividing them by 2 will result the position of the center of the line. The assumption is made, that the center of the car matches the center of the image. subtracting the center of the car by the center of the lane line and taking the absolute value results in the absolute distance between these two

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

A blank image is created on which the space between the left and the right poly-line is plotted in green color. This image is transformed back to the original image size by using the same same function used for the initial perspective transform but swapping source and destination points. then this image is overlayed on the original image. This is the final image which is the output of the pipeline.

Pipeline (video)

The pipeline explained above was applied to the project video by using the imageio and moviepy libraries.

Here's a ~~link to my video result~~

Discussion

The pipeline performed well on the project video, though it fails on more challenging videos. While it performs well on dark asphalt with bright lane lines, it seems to have problems with shadows on the road as well as with lighter asphalt. This could be improved to a certain point by adjusting the thresholds. A good way to improve the lane finding system would be to include a satellite vision of Google Maps and compare if the values match and avoid fatal steering failures due to computer vision. The code of the pipeline could be leaned up by restricting the search areas for line points for the following points