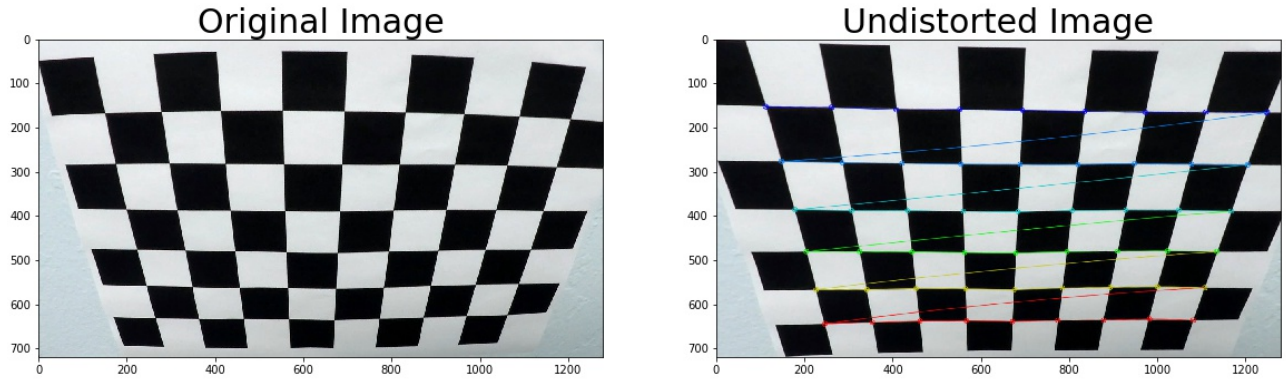
Advanced Lane Finding Project

The headings in this document are the same as in the provided notebook. So a navigation between code and writeup should be as easy as possible.

Camera Calibration and Distortion correction

The first step is the correct the bias of the camera lense. This step is done via the **calibration with a chessboard**. As a start, I define “object points”, which represent (x, y, z) coordinates of the chessboard corners in the world. These “object points” are fixed and describe a planar space (x, y) with z = 0, which represents the corners of the chessboard on the wall. This steps creates an artificial grid, which are fitted to the detected corners of the chessboard image. To detect the corners of the chessboard I use the opencv2 function cv2.findChessboardCorners(). Those “image points” contains the pixel location on the distorted image.

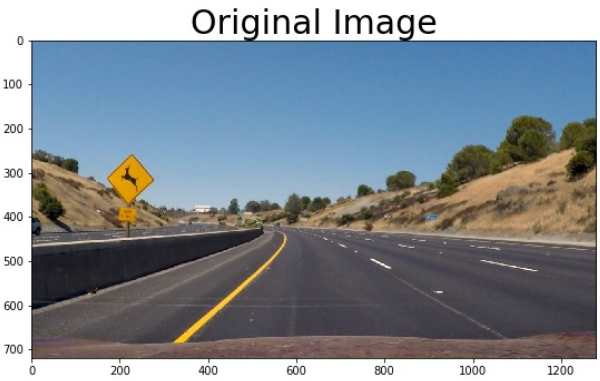
With the known defined planar coordinates of the object points and the measured coordinates of the “image points”, I can **correct the distortion of the camera lense**. The distortion correction can be computed by the cv2 function cv2.undistort(), which uses the object points and image points for the image correction. The result for the chessboard looks like this image (for better checking, I kept the detected chessboard corners in the undistorted image):

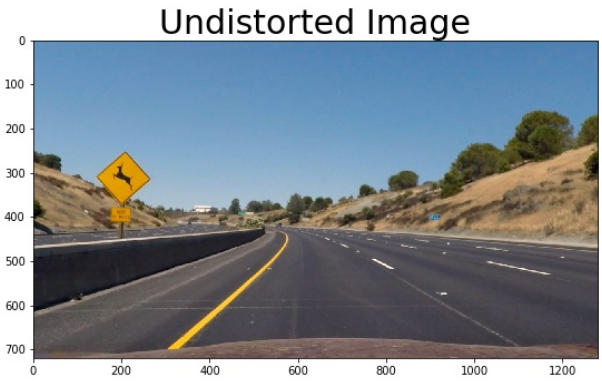


Pipeline (single images)

Distortion correction

The same step above for the camera undistortion will be applied to the test road images. I use therefor the parameters obtained from the camera calibration. The difference is especially recognisable for the human eye at the tree and bushes. The results look like:



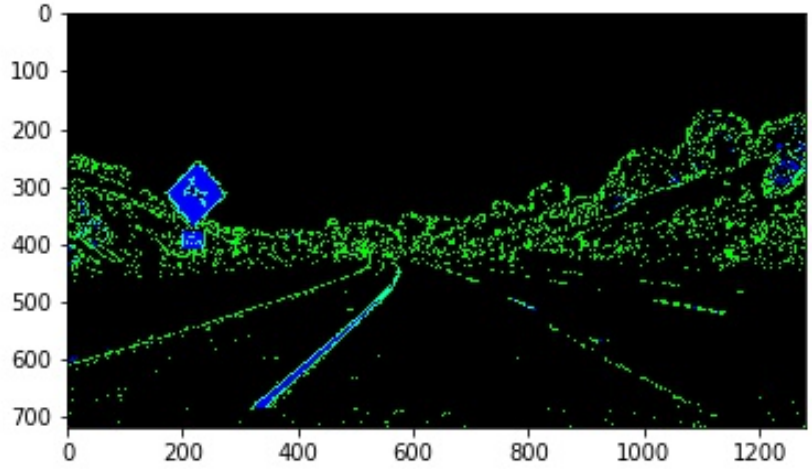


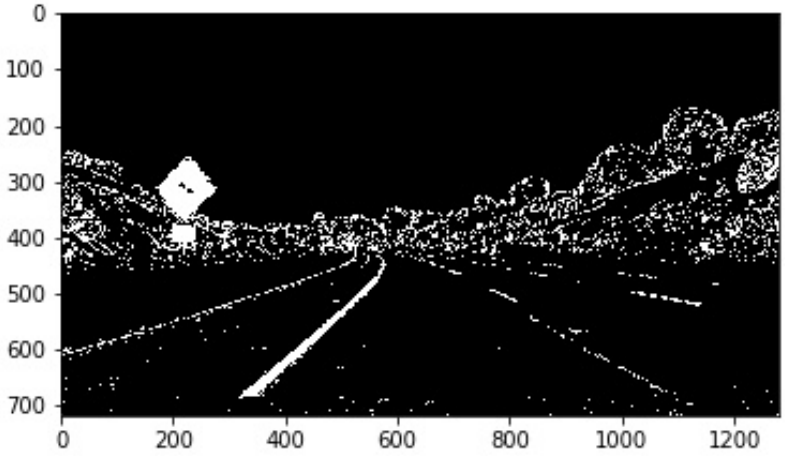
Color transfer and binary image creation

To make it a bit easier for the computer to detect lanes, we **transform the colour space** from RGB to HLS (hue, lightness, saturation). For further processing I keep the lightness channel and the saturation channel. The lightness channel represents the brightness of colors and the saturation channel describes the colourfulness of an image. I use the opencv function cv2.cvtColor(image, cv2.COLOR_RGB2HLS)

Based on the lightness channel I **apply a sobel x filter** (cv2.Sobel()). The sobel x filters horizontal lines out of the image. So I want to find lane lines in the image. After I applied the filter, the matrix values will be scaled to the range of 0 - 255. With the sobel x threshold a binary matrix will be returned for values between the sobel x threshold of 20 - 100.

For the **saturation channel** I **filter** all values between 40 and 140 and convert it to a binary matrix. In the last step, I combine both binary matrices and return a binary matrix, where 1 means, that this pixel is between the saturation threshold and the sober x threshold. The output images look like those images (colored by filter, and final binary output):





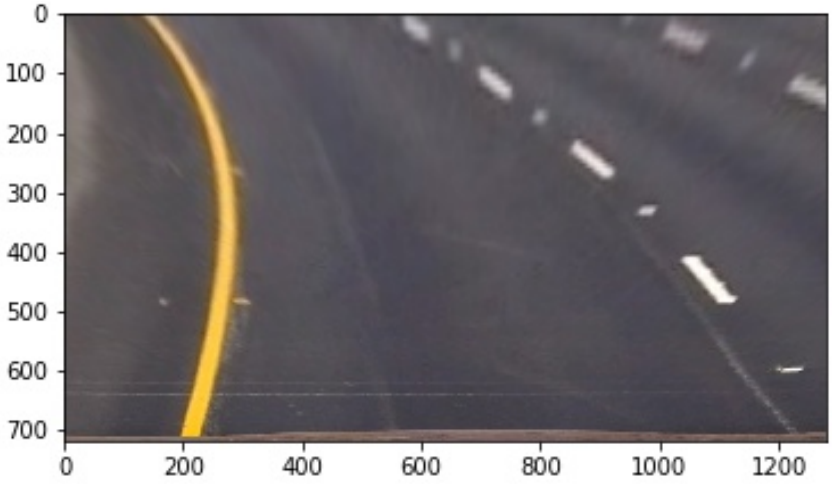
Rectify to "birds-eye view”

To make the lane detection a bit easier, we only look at a specific part of the image. Therefor we **define a polygon**, which loosely defines our area of interest. We use this area, transform and transform it to the the birds-eye view for our lane line detection. I transform the polygon area to the size of the whole previous image (width > high) to have a strong separation on the x-axis. In the first step I calculate the transformation matrix between those two shapes. Therefor I use cv2.getPerspectiveTransform() function from opencv2 and afterwards, I transform the polygon_image to the second image shape. Therefor I use the cv2.warpPerspective() function from opencv2.

The coordinates of the polygon and the rectangle:

Polygon = (1180, 700), (150, 700), (750, 440), (580, 440)
Rectamgle =(1280, 720), (0, 720), (1280, 0), (0, 0)

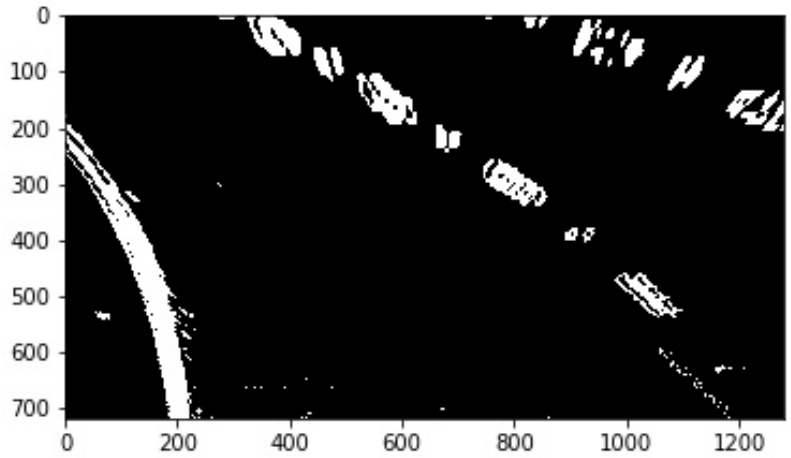
The outcome looks like this both images:

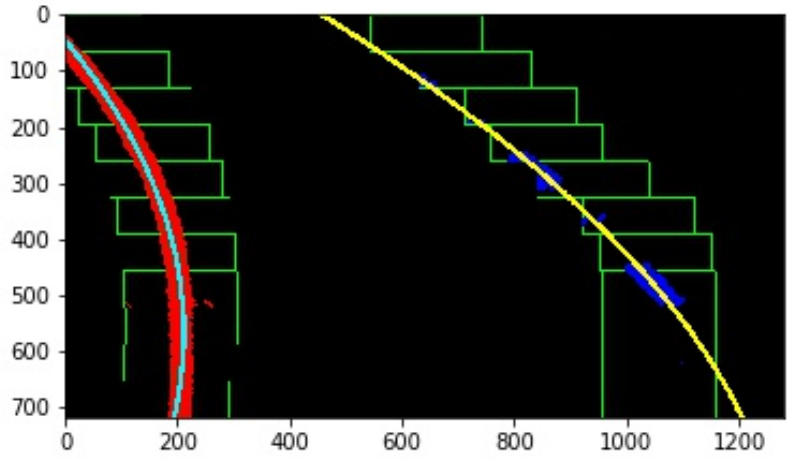


Find lane line

For lane line detection, the rectified “birds-eye view” has to be an binary image. The first step of lane fitting is the **detection of pixels in the binary image**. Since the lanes are mostly horizontal in the lower part of the image, I split the image in two parts and take the lower one for the calculation of the histogram along of the x-Axis. Afterwards I define the middle of the x-Axis a a midpoint and assume that one lane is left and right from this point. For a better lane finding, the image can be sliced in several parts, where for each window the followong lane detection technique will be applied. For each window, a left and a right peak will be defined. Around this peak a box will be created with some predefined size (like 100 pixel). If inside this box is a specific number of pixels, then the mean of all points is used for further lane fitting.

After this step, we got in the best case 11 coordinates for the left lane and 11 for the right lane. In the last step, we use a **polynomial fitting to fit a curve** through those points. Those curves represent the fitted lanes in the image. As shown here (image points aren’t shown in this example):





Real Lane Fitting

In the last step, the curve fitting on the rectified image is transformed to the original image. Therefor I calculate the inverse **transformation** matrix **between the birds-eye view and the original image**. I use cv2.getPerspectiveTransform() function from opencv2 and afterwards, I transform the birds-eye view to the original image shape with cv2.warpPerspective(). In the last step I combine both images with the cv2.addWeighted() function, so that I can see the fitted lanes on the original image.

Another task is the **calculation of the curvature** of the fitted lane. Therefor I use the equation for the radius of curvature, combine it with the derivation of the fitted polynomials and I can solve it quite easily. For the transformation from pixels to meters, there are some correction coefficients provided, which ends up to plausible values. E.g. in this example image, the curvature for the left lane is 215 m and for the right lane is 253 m. For the position of the car, I calculate first the average pixel of both lanes for the maximum y-value. Then I calculate the difference of the image mid, where the camera is located, to this average. In the last step I correct it with the pixel to meter coefficient. In the last image, the car is 0.36 m away from the mid.

Pipeline (video)

Here's a link to my video result:

https://github.com/SteffenHaeussler/Self-Driving_Car_Engineer/blob/master/P2/output_project_video.mp4

I think, it looks quite okayish. It’s a bit wobbly on the right lane in the beginning, but for the rest of the video the result looks good. Also sorry for the bad text font for the lane curvature and the relative location of the car to the mid of the lane.

Discussion

The Pipeline consists of a lot of hard coded parameters, which had to be adapted manually for different scenes. I failed the most of the time with adjusting the parameters for binary colour filtering and with the point detection for curve fitting (number of points vs fitting). In the end, I went back to the provided parameters, since they work okayish.

The easiest way to break the pipeline are images, which doesn’t fit to the parameters. I can think of sharp curves like in the harder challenge, rainy night images, snow images or images without any lane information. Since the parameters was manual fitted for highway roads, it will fail for all other scenarios. So one solution will be, that for each scenario (rain, city, night, …) own hard-coded parameters had to be estimated. This is not useful. Otherwise there are already deep learning solutions, which looks quite good. For example this recent paper:

https://arxiv.org/ftp/arxiv/papers/1809/1809.03994.pdf

So, for making it more robust, it will probably useful to try a different method, since object detection had a really huge impact in computer vision in the last years since Alexnet. Also it would apply more than one sobald filter with the CNN architecture.