

# ADVANCED LANE FINDING

CHRISTIAN SCHÄFER

## CONTENTS

1	Introduction	2
2	Source file overview	2
3	Camera Calibration	3
4	Pipeline	3
4.1	Image transformations . . . . .	4
4.2	Lane Detection . . . . .	4
4.3	Car position and lane line curvature . . . . .	8
5	Results and Discussion	8

## LIST OF FIGURES

Figure 1	Example video images . . . . .	2
Figure 2	Source file structure . . . . .	2
Figure 3	An example of original and undistorted images . . . . .	3
Figure 4	Image processing pipeline . . . . .	3
Figure 5	Undistorted and transformed images . . . . .	4
Figure 6	Binary image calculation process . . . . .	5
Figure 7	Intermediate results of binary imaging process . . . . .	5
Figure 8	Binary image result . . . . .	6
Figure 9	Histogram of warped edge image . . . . .	6
Figure 10	Fitted line . . . . .	7
Figure 11	Detected lane and unwarped lane image . . . . .	7
Figure 12	Histogram of warped edge image . . . . .	8

## LIST OF TABLES

Table 1	Corner co-ordinates . . . . .	4
Table 2	Color channel thresholds . . . . .	6
Table 3	Coefficients limits . . . . .	7

## 1 INTRODUCTION

The Goal of the 3rd project in Udacity's Self Driving Car Nanodegree Program is to detect lane lines on a project video and mark them in the video file using advanced computer vision techniques as well as calculating curvature and vehicle position. Besides the project video Udacity provides two more different video files:

1. **Project video:** The vehicle is driving on a motorway with light curves, yellow and white lane lines. The asphalt is smooth and mostly dark grey. Video only contains a few difficulties like a shadow of a single tree.
2. **Challenge video:** The vehicle is driving on a more curvy motorway with yellow and white lane lines. The asphalt is repaired and partly dark or bright grey. Gaps are filled with bitumen resulting in black lines. Additional signs are mounted on the lane. The vehicle transits a shady bridge.
3. **Harder challenge video:** The vehicle is driving on a narrow, curvy and wavy street in a forest with yellow and white lane lines. The asphalt is smooth and dark grey. Video contains a lot difficulties like shadows or reflections in the front window.

Figure 1 shows a single example image for all videos.



Figure 1: Example image from project, challenge and harder challenge video.

## 2 SOURCE FILE OVERVIEW

All source files are located in the `/src` folder of the project. Below Figure 2 shows an overview of the source files. While classes are coloured white, grey boxes represent files containing functions only.

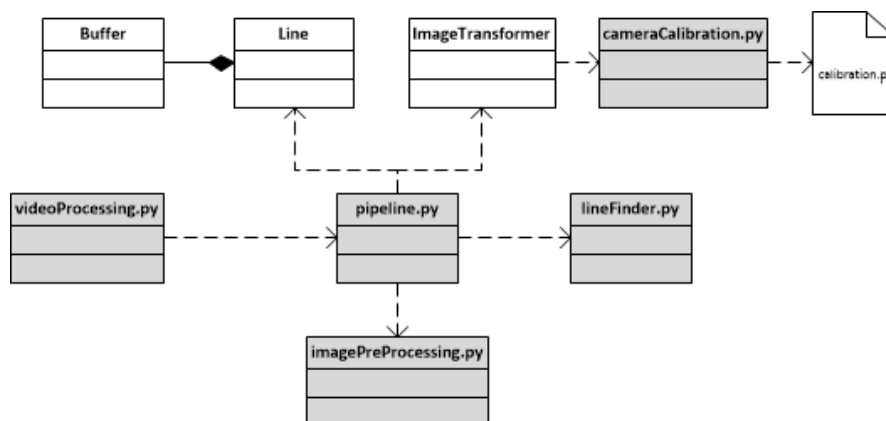


Figure 2: Structure of the source files

### 3 CAMERA CALIBRATION

The camera mounted on the vehicles front window need to be calibrated to be able to create undistorted images. Udacity provides a set of 20 calibration images of chessboards for this project.

First an image is converted to gray scale. Afterwards the chessboard corners are searched with OpenCV method `cv2.findChessboardCorners()` and stored within an array. Together with the known objects points for a 9x6 chessboard the located corners are passed to the `cv2.calibrateCamera()` method that calculates the distortion coefficients and the camera matrix. The calculated distortion coefficients and camera matrix are then stored to a pickle file within the calibration folder. The `src/cameraCalibration.py` file contains the code for the camera calibration.

Figure 3 shows an example of two images from the calibration set and there undistorted images.

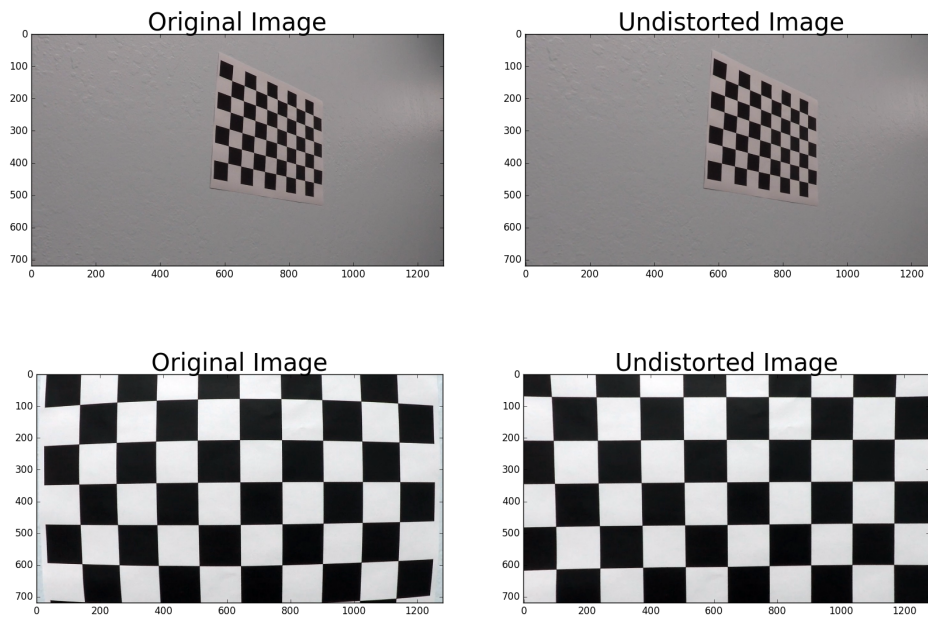


Figure 3: An example of original and undistorted images from the calibration set.

### 4 PIPELINE

Figure 4 shows the complete pipeline every single image is passed through. While the calibration is only done once and described in the chapter before, the single actions of the pipeline are explained in detail in the following sections.

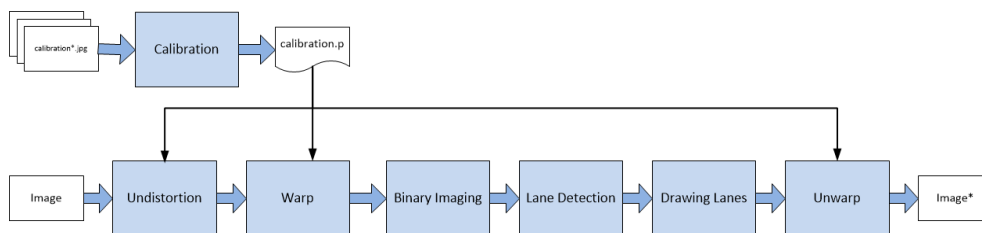


Figure 4: The image processing pipeline.

## 4.1 Image transformations

For all image transformation the *ImageTransformer* class in **src/ImageTransformer.py** is used. The *ImageTransformer* gets the image dimension on construction. The constructor also loads the determined camera and distortion matrices from the pickle file **camera\_cal/calibration.p** from former camera calibration step.

### 4.1.1 Undistortion

With the former calculated camera matrix every single picture from the video is first transformed to an undistorted image using the OpenCV method `cv2.undistort()`.

### 4.1.2 Perspective transform

The undistorted image is then transformed into a bird view. For that a trapezoid is defined that get transformed to a rectangle in the destination image. Table 1 shows the used co-ordinates within the 1280x720 pictures.

Table 1: Corner co-ordinates

corner	source	destination
left top	[595,450]	[300,0]
left bottom	[205,720]	[300,720]
right top	[685,450]	[980,0]
right bottom	[1122,720]	[980,720]

The values for source and destination are defined within the constructor of the *ImageTransformer* class. During construction the transformation matrices to warp and unwarp the images to and from bird view are calculated with OpenCV method `cv2.getPerspectiveTransform(src, dst)`. Figure 5 shows a undistorted image of lane lines with source trapezoid together with the output for that image (without trapezoid) from `ImageTransformer.warp()` method.

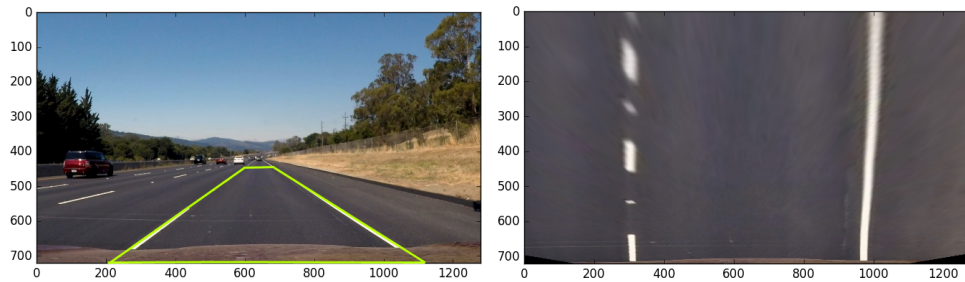


Figure 5: Undistorted image of lane lines with source trapezoid and transformed image in bird view.

## 4.2 Lane Detection

### 4.2.1 Binary images

To ease the lane detection images are first processed to emphasize the lane lines. An ideal processing would result in a binary image containing only lane lines. Through disturbance like shadows, lightning and different lane quality this process is one of the most challenging steps in lane detection. To reduce noise several techniques like color-thresholding and edge detection were combined. Figure 6 on the following page shows the complete process as a block diagram.

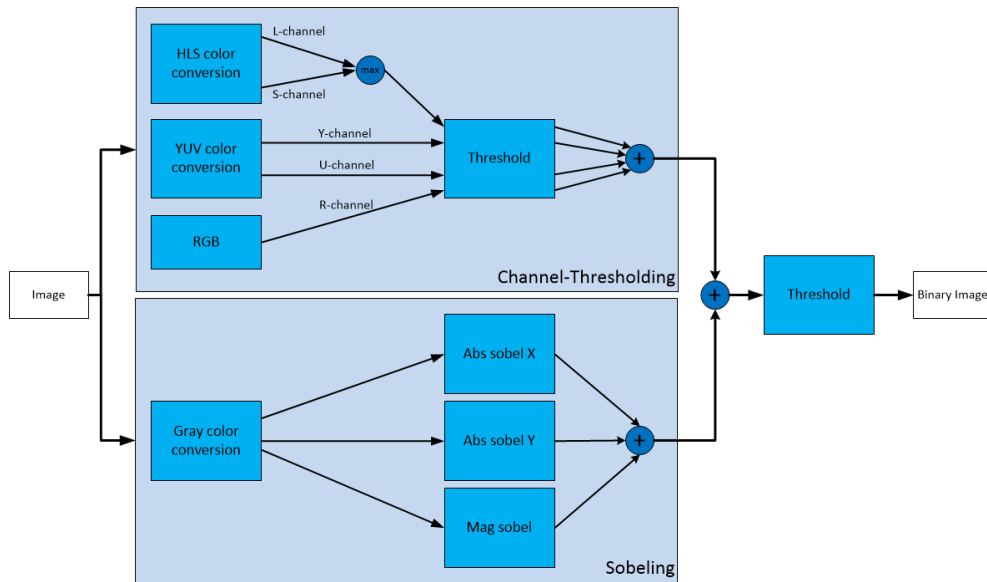


Figure 6: Binary image calculation process.

The undistorted and warped images are passed to the `binaryImage(img)` method in the `/src/imagePreProcessing.py` Python file. Here the images are passed to `thresholding(img)` and `sobeling()`. The results from this two methods are then summed and thresholded to get a binary image.

For color channel thresholding Y, U, R and the maximum of L and S were chosen. While the Y-channel from YUV-colorspace is good for white colors, the U-channel is simply perfect for yellow colors. R from RGB colorspace can represent both, white and yellow colors. The max(L,S) combination from HLS colorspace shows also good results for both white and yellow lane lines.

To be more robust against lightning, noise edge detection algorithms were also used. For edge detection the OpenCV method `cv2.Sobel()` is used in X and Y direction as well the magnitude for both. Figure 7 shows an example image for every part of the binary image calculation.



Figure 7: From left to right and top to bottom: warped image, max(S,L) threshold, Y channel threshold, U channel threshold, R channel threshold, absolute Sobel x direction, absolute Sobel y direction, magnitude Sobel, summed image

The used threshold are shown in Tabel 2.

Table 2: Color channel thresholds

input	threshold
max(S,L)	(200,255)
Y	(200,255)
U	(50,120)
R	(200,255)
abs sobel x	(14,100)
abs sobel y	(15,110)
mag sobel	(30,200)
summed	$\geq 3$

The summed image is finally transformed in a single binary image by another threshold. Figure 8 shows the final result of the binary image calculation process.

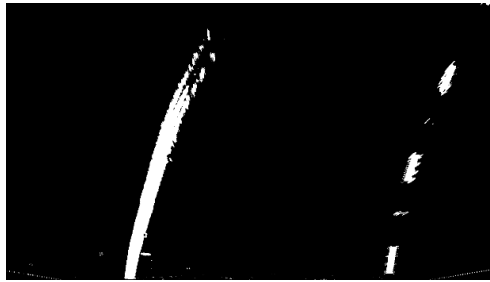


Figure 8: Resulting binary image after thresholding the summed image.

#### 4.2.2 Lane Detection Algorithm

The warped binary image is then passed to the `findLines()` method within the **LineFinder.py** file. If no line was detected before, a histogram of  $1/3$  of the image's underpart is generated (Figure 9). Otherwise the average start position of last  $N$  detected lines is used.

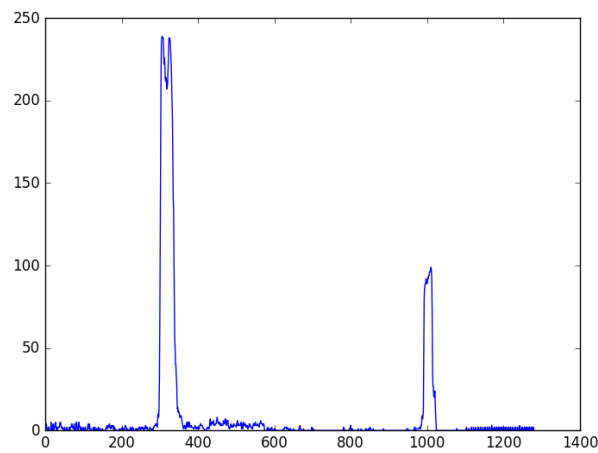


Figure 9: Histogram of a example binary image.

The maxima of left and right half of the histogram are used as the starting positions for left and right lane line detection.

To find the lane lines a search window of  $1/20$  of image height and 200 px is placed on the discovered lane line start positions. Every pixel of the binary images within this window is considered to be part of the lane line. If more than 20 pixels have been found the midpoint in x-direction of the next search window is re-centred to the mean of found pixels position. This is repeated till the top of image is reached.

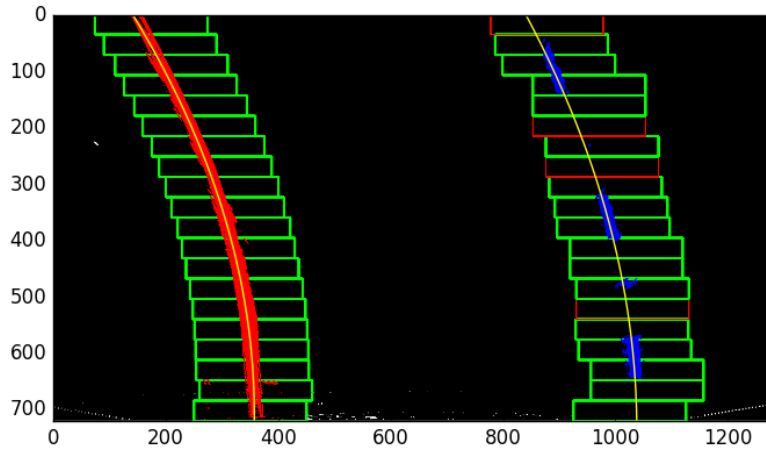


Figure 10: Search windows with located pixels of left and right lane as well as the discovered curves.

The identified lane pixels are then passed to the `Line` instance objects of the `Line` class within `src/Line.py` by `Line.setPixels()`. Afterwards the pixels are fitted to a polynomial function of degree 2 in `Line.fitLine()` with Numpy method `numpy.polyfit()`. Figure 10 shows all steps of the fitting process.

Finally the area between detected line in the image is filled and warped back into original image perspective with `ImageTransformer.unwarp()` (Figure 11).

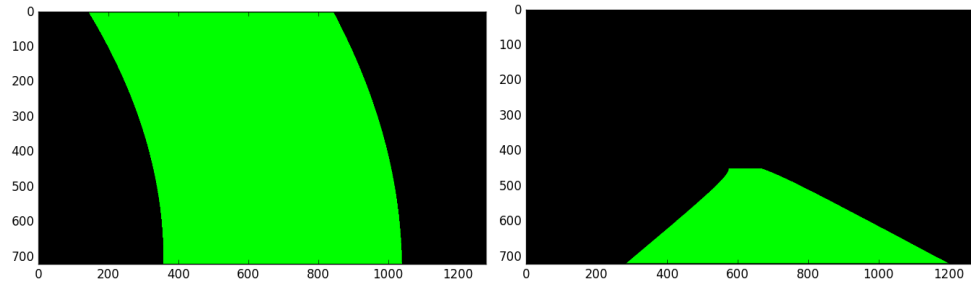


Figure 11: Detected lane and unwarped lane image

#### 4.2.3 Lane line evaluation

After fitting a `Line` is also evaluated to discard extreme outliers. Evaluation is done within `Line.evaluateLine()`. For evaluation the difference of the coefficients between the current and last fit are checked against the limits shown in Table 3.

Table 3: Coefficients limits

coeff_0	coeff_1	coeff_2
1.0e-3	1.0	2.5e2

Furthermore the lines are checked if they are more or less parallel. If a line is valid it is marked as detected and its coefficients are stored in a ring buffer. If it is not,

the line will be marked as undetected and not used for drawing. For drawing the average of the last  $N$  coefficients within the ring buffer are used.

### 4.3 Car position and lane line curvature



Figure 12: Histogram of a example binary image.

## 5 RESULTS AND DISCUSSION

The result video can be found in the **output\_video** folder within the project. While thresholding Y channel and U-V channel is enough to detected white and yellow lane lines with acceptable accuracy even without evaluating detected lane lines for the project video 1. thresholding most diffecult. Seeting threshold according to image brightness Performance. dynamic technique when less points found bumps in the road disturbs lane line parallelity parelletity according to image horizion

Another two interesting strategies were used by Nick Hortovanyi<sup>1</sup> and Mikel Bober-Irizar<sup>2</sup> within their projects. Nick Hortovanyi uses different binary thresholds for the top half and bottom half of warped images. Mikel Bober-Irizar uses a 720x1280 destination image instead a 1280x720 image for transformation. This reduces erosion of the lane line on top of the image and also would make the evaluation algorithm more robust against bumps in the road. Gauss filter interpolation techniques on warped images.

<sup>1</sup> ProjectNick Hortovanyi

<sup>2</sup> Project Mikel Bober-Irizar