# **Report of Advanced Finding lane**

## **Goals of this project**

* 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.

## **High level flow chart of my pipeline**

I created a pipeline called *Advanced\_Finding\_Lanes* to process the video. Please see below the high-level flow chart of my pipeline in Figure 1. The completed process consists of 11 steps. I will explain each step in detail in the following paragraphs.

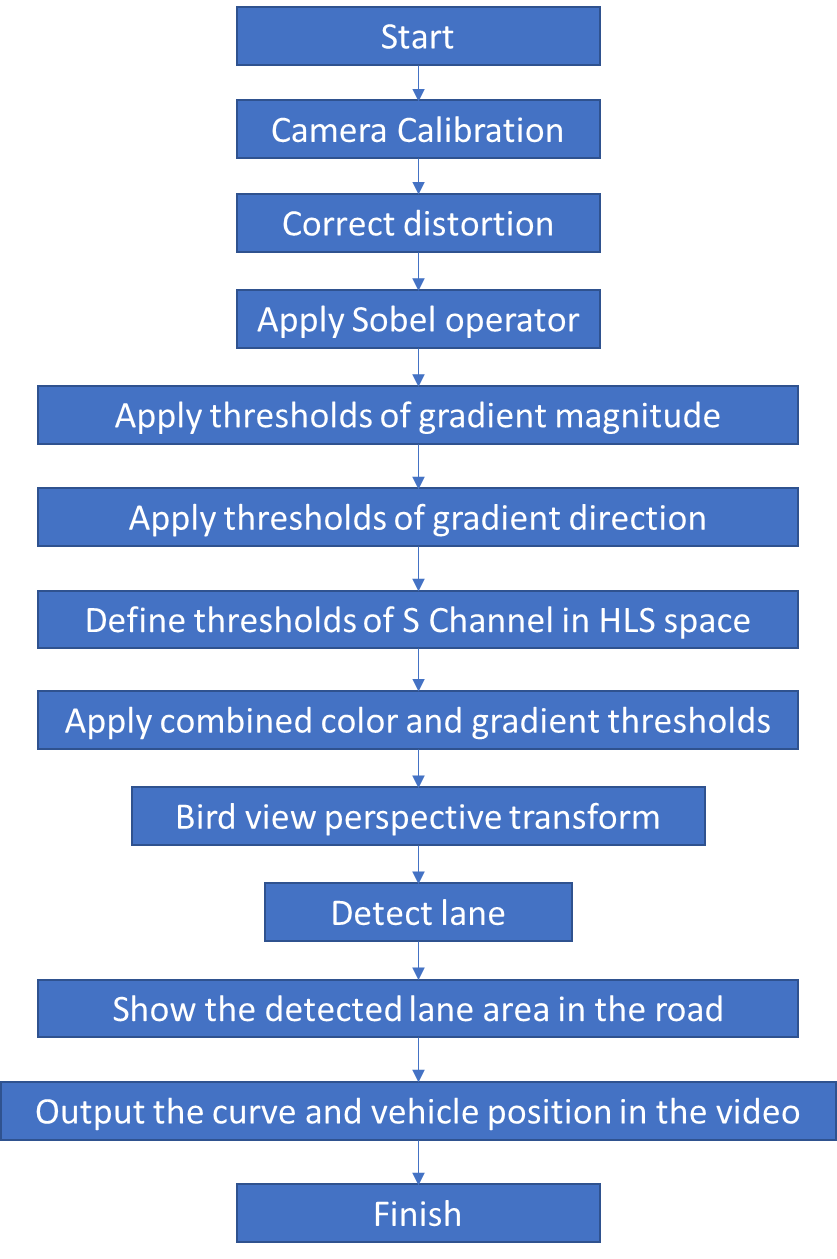


Figure 1High level flow chart of pipeline

### Camera calibration

As all the pictures taken by the cameras are distorted to some extent, the first thing we need to figure out is how much the camera is distorting the picture (this will help us to get a accurate lane curvature data in later steps). I used the chessboard pictures for my camera calibration function. There are 9 corners in the X direction and 6 corners in the Y direction. Firstly, I used *cv2.cvtColor* to grayscale the picture, and then used cv2.findChessboardCorners to look for corners. Figure 2 shows the corners that I found in one of the example picture. After finding all the corners, I started calibration using cv2.calibrateCamera. The output camera matrix and distortion coefficients will be later used for distortion correction. Please refer to Camera\_Calibration(images, nx, ny) in my code.

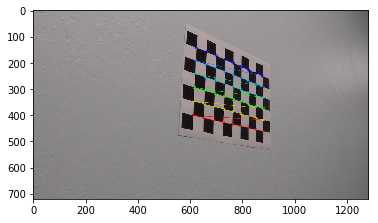


Figure 2 corners founded in the chessboard

### Correct distortion

With the camera matrix and distortion coefficient from last step, I use directly use the cv2.undistort function to correct all the distorted picture. Figure 3 is an example of the comparison of original and corrected images. Please refer to Distortion\_Correction(img,mtx,dist) in my code for implementation details.

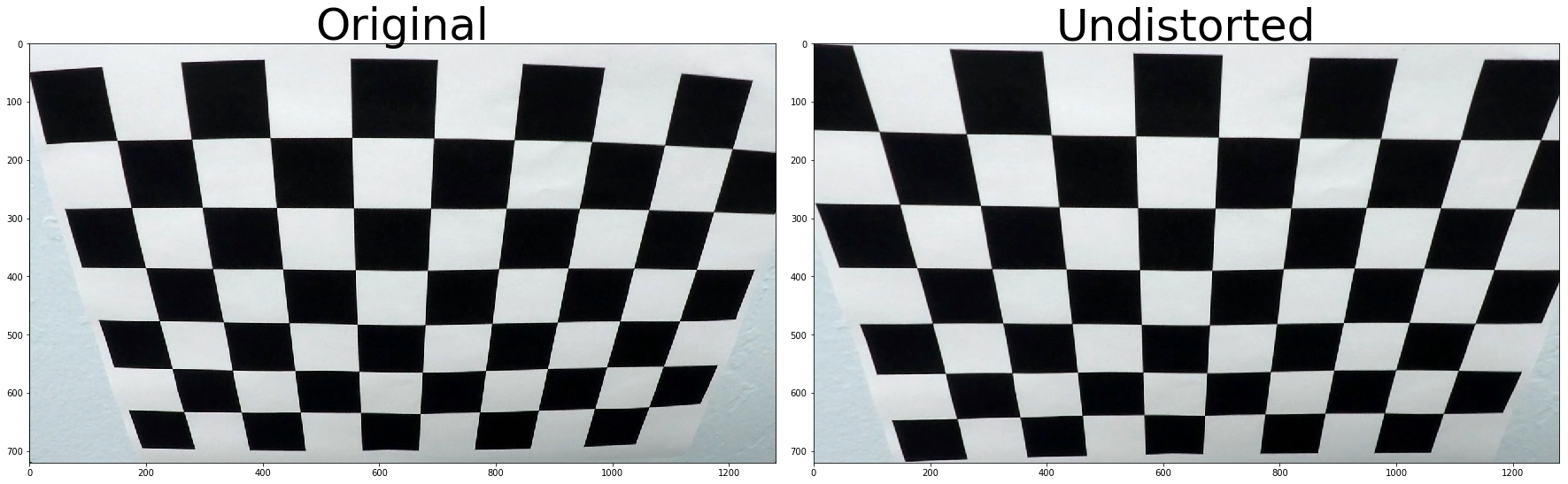


Figure 3 Original vs Undistorted

### Apply Sobel operator

From this step onwards, I start to working on the lane detection. The first step is to apply sobel operators to detect all the pixels that have big pixel value changes. Intuitively, the first step is still to grayscale the color image. Then, I apply the sobel operators for both X direction and Y direction, the gradients in X direction emphasize edges closer to vertical direction and the gradients in Y direction emphasize edges closer to horizontal direction. Please see Figure 4 for the gradients in X direction and Figure 5 for the gradients in Y direction. After some trials and what I learned from lectures, I set the minimum threshold as 30 and maximum threshold as 100. Please refer to abs\_sobel\_thresh(img, orient='x', thresh\_min=0, thresh\_max=255) in my code for implementation details.



Figure 4 Thresholded Gradient in X direction



Figure 5 Thresholded Gradient in Y direction

### Apply threshold of gradient magnitude

What I did in last step is just the very basic first step. As you can see from the images I showed, there are still a lot of noise that might might have negative effect on the lane detection. In order to clean up the processed images, we also apply the threshold of gradient magnitude to remove some noise. As you can see in Figure 6, after filtering out some noise by applying gradient magnitude thresholds, the process image is cleaner than before. In my implementation, I set the minimum threshold as 55, maximum threshold as 100 and kernel size as 15. Please refer to mag\_threshold(img, kernel=15, mag\_thresh=(55, 100) in my code for more details.



Figure 6 thresholded Magnitude

### Apply thresholds of gradient direction

The magnitude thresholds already helped me to clean some noise, but there are still some other non-lane things detected in the image. In the case of lane lines, we're interested only in edges of a particular orientation. Therefore, I decided to use also the direction of the gradient to help me find the lanes more accurately. The gradient is very easy to calculate by using np.arctan2 with the X, Y values from the sobel operator. In my implementation, I set the minimum threshold as 0.7, maximum threshold as 1.3and kernel size as 15. The processed image after direction thresholding is shown in Figure 7. Please refer to dir\_threshold(img, kernel=15, thresh=(0.7, 1.3)) in my code for more details.



Figure 7 Thresholded gradient direction

### Apply thresholds of color space

There are different kind of color spaces for image analysis, for example RGB. HSV and HLS. Accroding to the lecture, the S channel of the HLS spacs most likely guarantee the best performance of the lane detection. Therefore, I also applied S channel thresholding in my pipeline by calling hls[:,:,2]. In my implantation, I set the minimum threshold as 170 and maximum threshold as 255. Figure 8 shows how the processed image look like after applying the color thresholding. Please refer to hls\_select(img, thresh=(170, 255)) in my code for implementation details.



Figure 8 Thresholded S channel in HLS space

### Apply combined color and gradient thresholds

### Bird view perspective transform

### Detect lane

### Show the detected lane area

### Calculate and Output the curve and vehicle position