Advanced Lane Finding

The goals / steps of this project are the following:

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

The project can be divided into two main parts: image pre-processing and lane detection and display. The first step is composed of camera calibration, distortion correction, color and gradient binary, and perspective transform; the second steps is lane lines detection, their visual display and curvature and vehicle position determination.

For the explanation purpose, the document will be written based on the file ***AdvancedLaneFinding_Figure.py,*** which is used for the image (fixed frame video) testing process. The video verification can be found in the file ***AdvancedLaneFinding_video.ipynb***.

Step 1: Image Pre-processing

Camera Calibration

The camera calibration was conducted by using the provided chessboard photos taken with different angles. I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at $z = 0$, such that the object points are the same for each calibration image. Thus, ***objp*** is just a replicated array of coordinates, and ***objpoints*** will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. ***imgpoints*** will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output ***objpoints*** and ***imgpoints*** to compute the camera calibration and distortion coefficients using the ***cv2.calibrateCamera()*** function. I applied this distortion correction to the test image using the ***cv2.undistort()*** function and obtained the result shown in Figure 1.

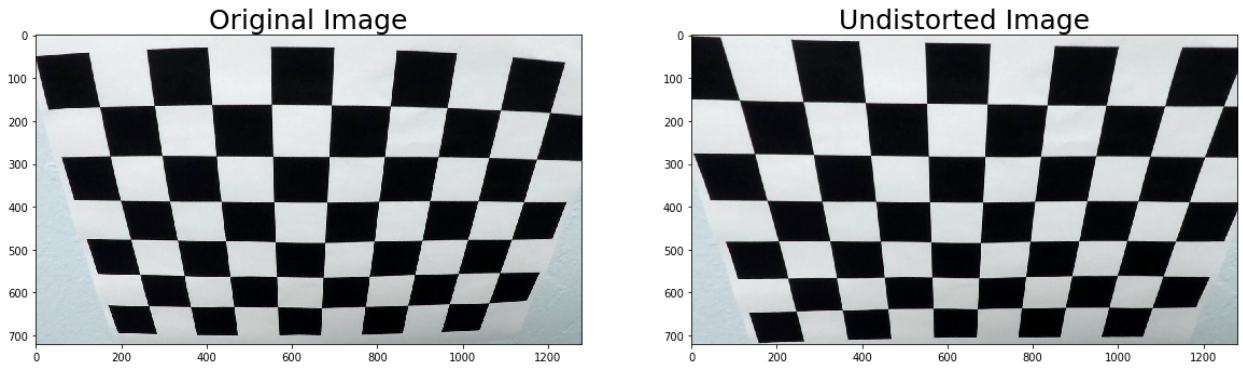

Figure 1 Camera Calibration

Distortion Correction

The calculated ***objpoints*** and ***imgpoints*** were used to apply the distortion correction to the provided testing images. The ***cal_undistort()*** function was used during the process. The result of distortion correction to the ***straight_lines2.jpg*** is shown in Figure 2.

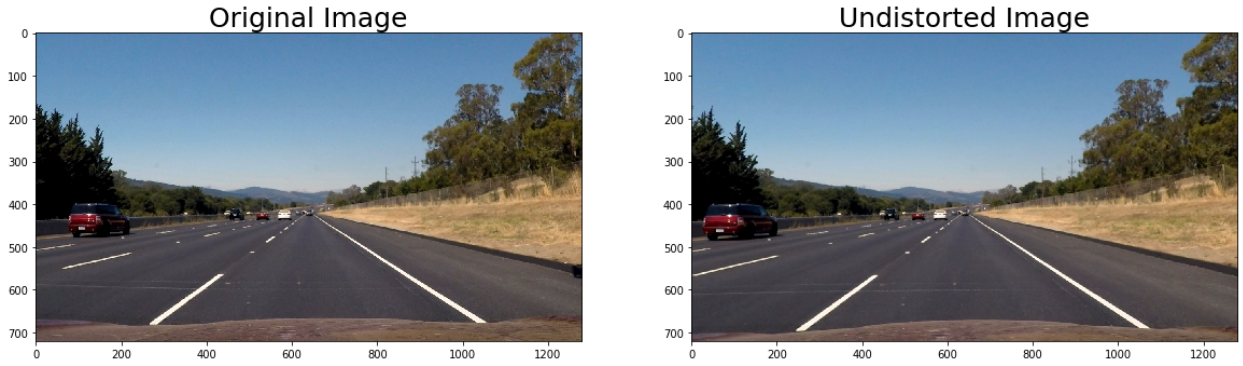

Figure 2 Distortion Correction

Color and Gradient Threshold

One HLS color thresholding binary ***hls_white_yellow_binary()*** and one combined gradient thresholding binary was applied to process the images individually, then they were combined into one binary mask.

Color Binary

The HLS binary was used to identify yellow and white color due to its robustness. The yellow color is determined by the S channel ranging between $[90, 255]$. The while color is dominantly determined by the L channel only falling between $[200, 255]$. The HLS color binary result can be shown in Figure 3.

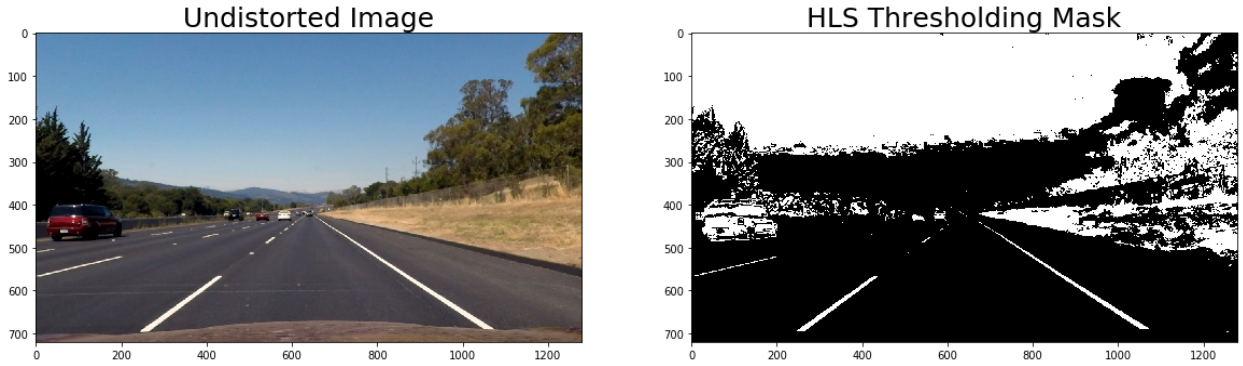

Figure 3 HLS Color Binary

Gradient Binary

The gradient binary is a combined binary comprised of the individual x and y gradients ***abs_sobel_thresh()***, both x and y gradients ***mag_thresh()***, and direction of gradient ***dir_threshold().*** The result can be shown in Figure 4.

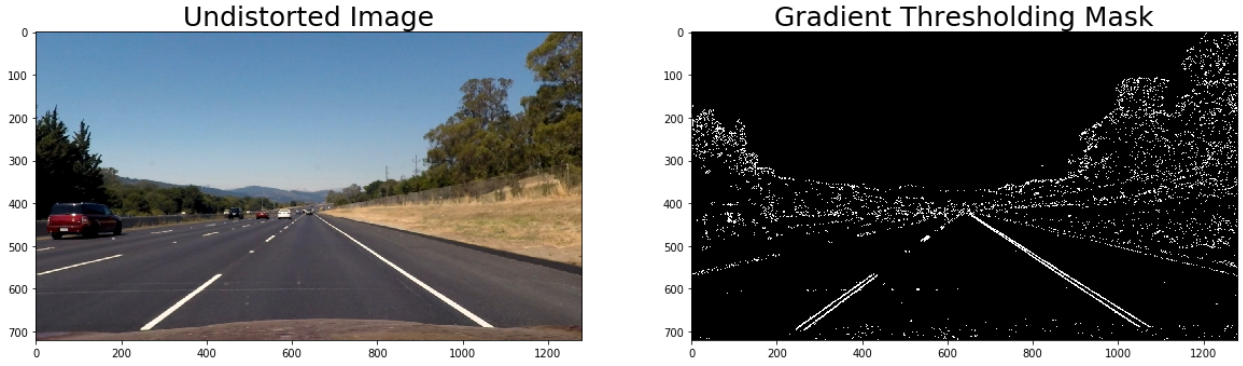

Figure 4 Gradient Binary

The final combined binary result is shown in Figure 5.

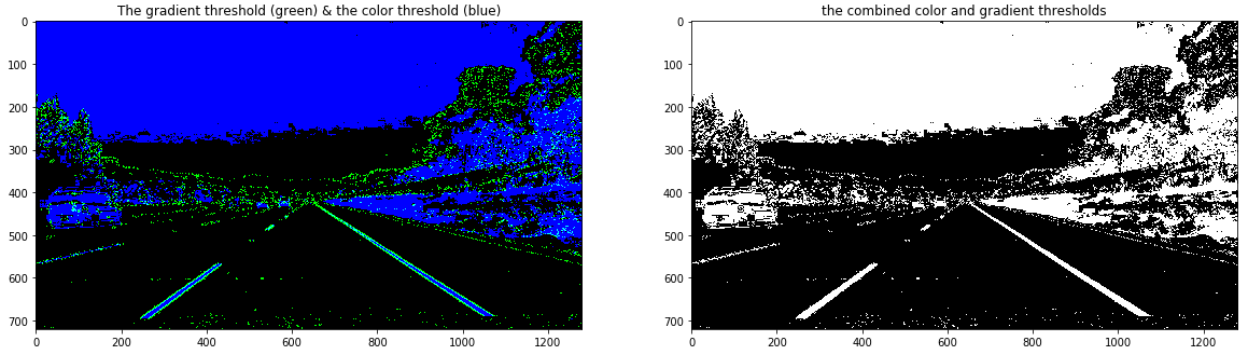

Figure 5 Final Combined Color and Gradient Binary

Perspective Transform

The purpose of the conducting perspective transform is to obtain a bird's-eye view transform that let us

view a lane from above; this will be useful for calculating the lane curvature later on. The code for my perspective transform is includes a function called ***warp()***.The ***warp()*** function takes as inputs an image (***img***) , source (***src***) and destination (***dst***) points as well as one flag. The flag sign was used to determine the perspective transform matrix (***M***) calculation and the image size. I chose the source and destination points as follows:

src_pts = np.array([[690, 450], [1110,img.shape[0]-2],[210,img.shape[0]-2], [595, 450]], np.int32)

dst_pts = np.array([[1000,0], [1000,img.shape[0]-2],[200,img.shape[0]-2], [200, 0]], np.int32)

The obtained result can be shown in Figure 6.

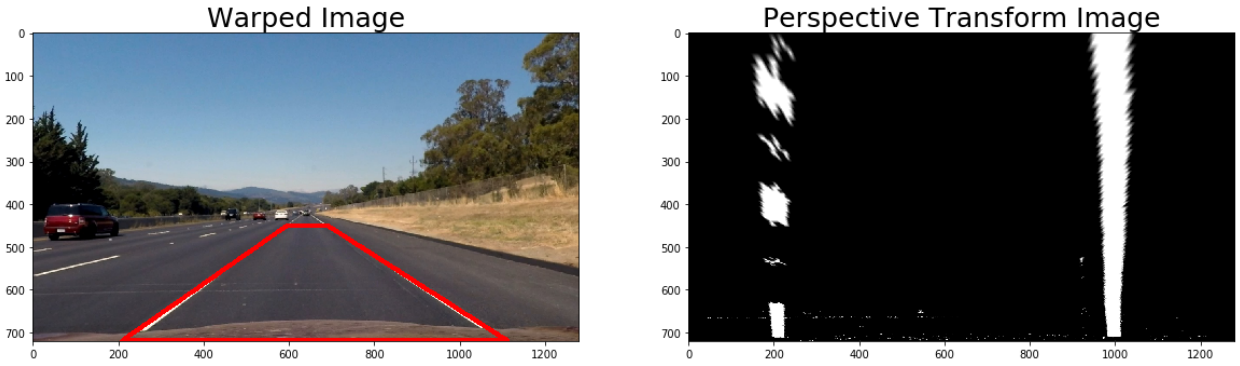

Figure 6 Perspective Transform

Step 2

After applying calibration, binary thresholding, and a perspective transform to a road image, we obtained a binary image where the lane lines stand out clearly. However, we still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line. We first took a histogram along all the columns in the lower half of the perspective transformed image shown in Figure 7.

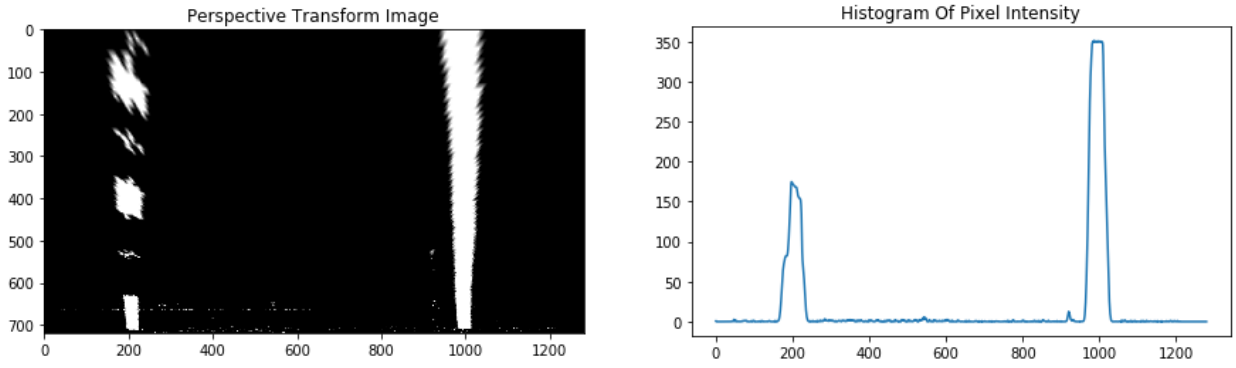

Figure 7 Histogram of Lower Half of the Perspective Transformed Image

We implemented the sliding windows technique to identify the lane line pixels, and then fitted the lines

with a polynomial as shown in Figure 8.

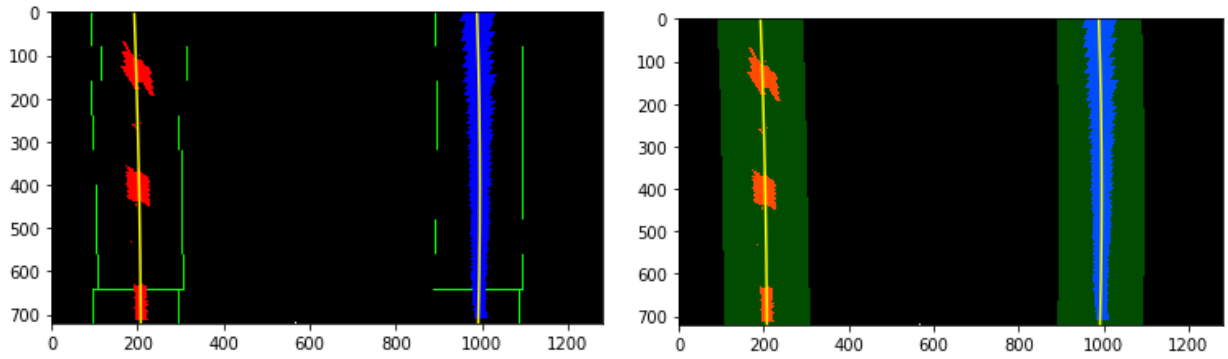

Figure 8 Lane Line Pixels Identification and Polynomial Fitting

The lane line radius curvature and the car offset was then calculated based on one image as shown in Figure 9.

Figure 9 Lane Radius Curvature and Car Offset Calculation

Reflections

Compared with the first project, it's more robust to identify the curved lane lines due to the second polynomial fitting technique used. However, as shown in in the challenge video test, the created pipeline doesn't perform well when the road lightness varies violently. The issue is due to the fact that the developed lane detection pipeline relies on color thresholding binary which is designed for yellow and while color specifically. More robust color masks can be developed to handle the issue.

In general, the computer vision projects are more straightforward and clear compared with the deep learning projects. We have full control of the fine-tuning process step by step, but they are designed for some specific applications; meanwhile, deep learning is more robust for wide application scenarios but with black box and frustrating development process.