Report of Advanced Finding lane

1. 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.

2. 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.

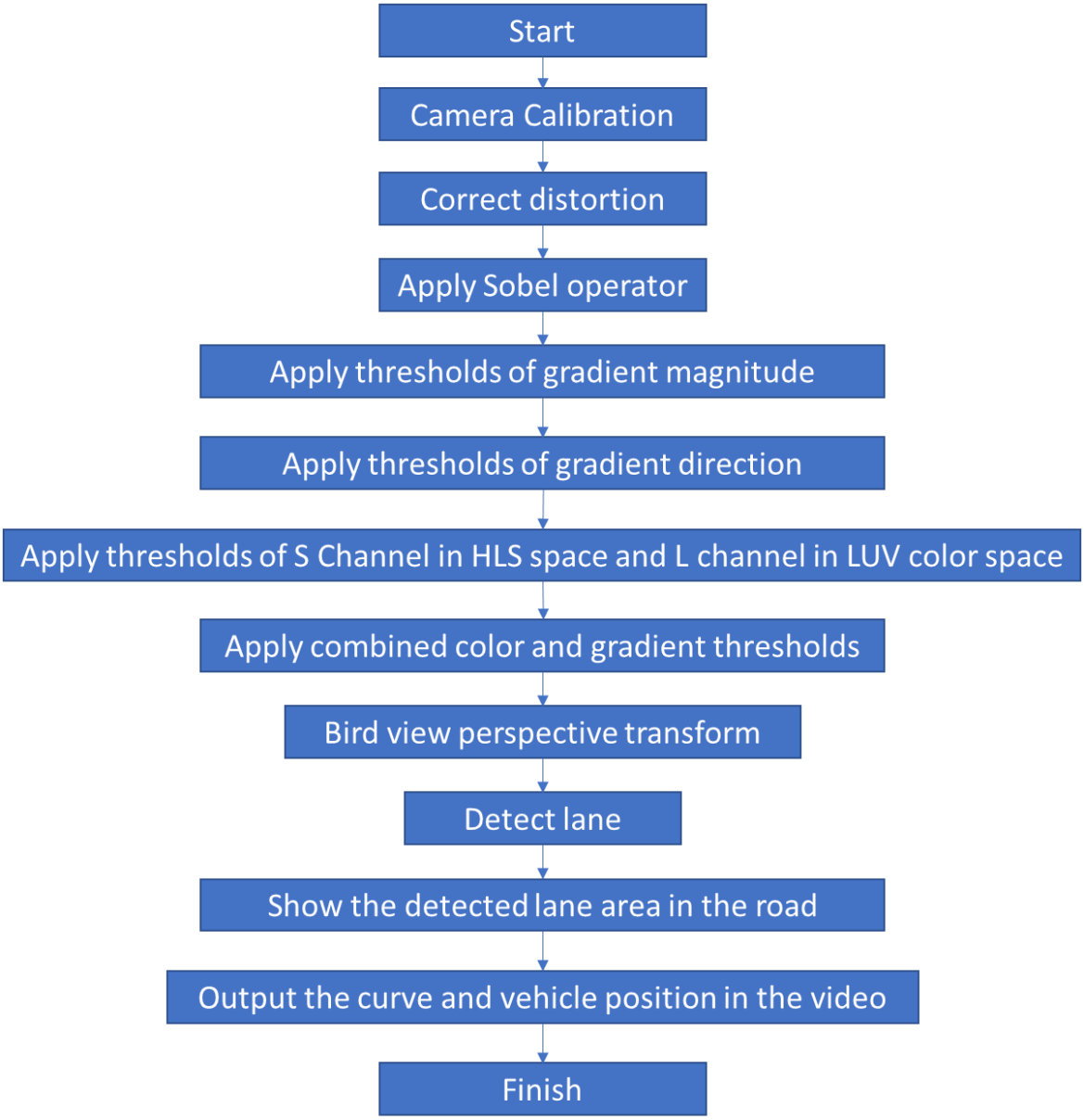

Figure 1High level flow chart of pipeline

- 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.

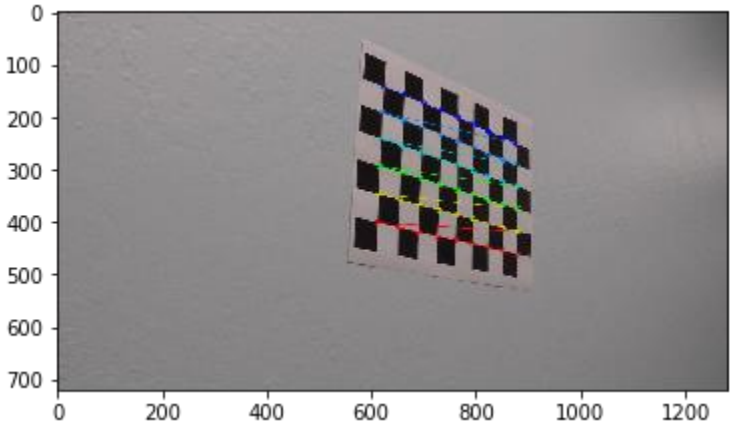

Figure 2 corners founded in the chessboard

- 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.

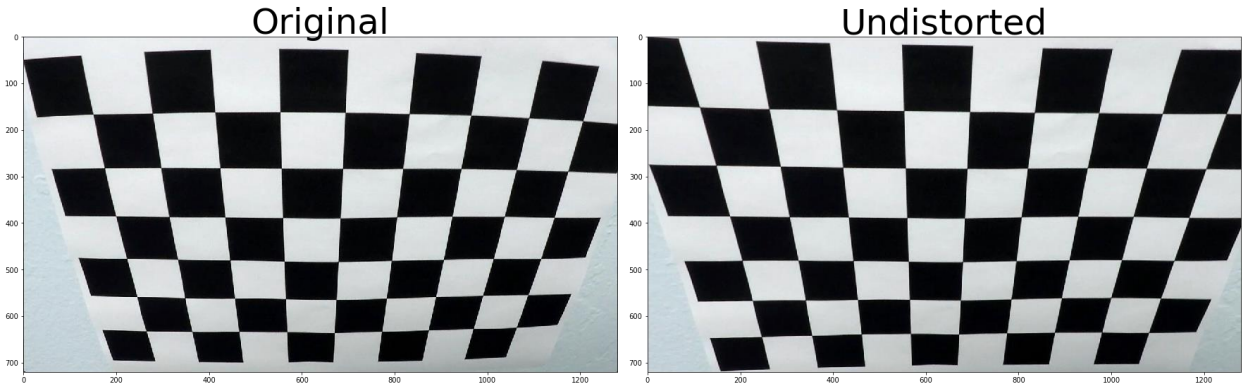

Figure 3 Original vs Undistorted

- 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.

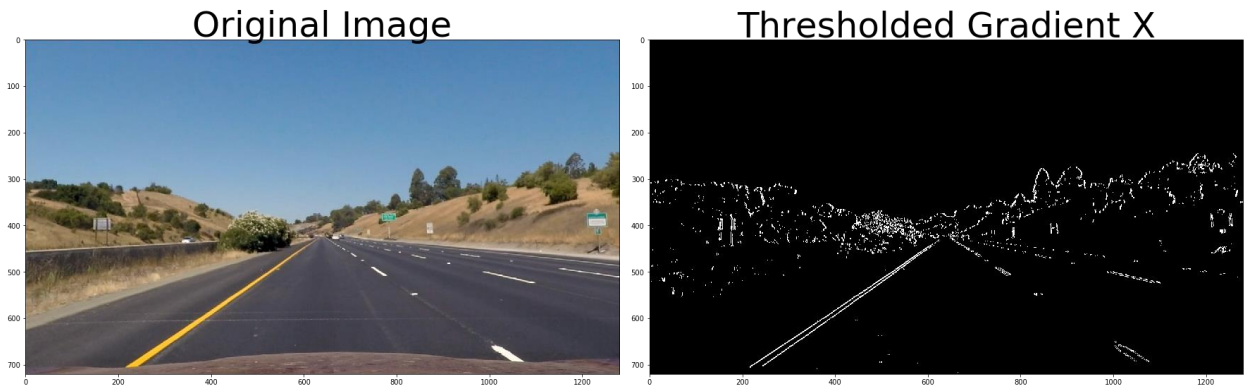

Figure 4 Thresholded Gradient in X direction

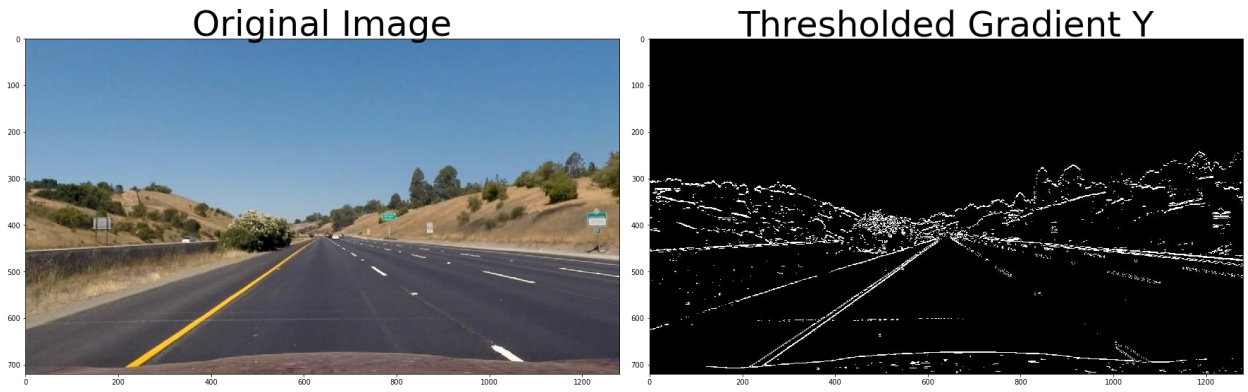

Figure 5 Thresholded Gradient in Y direction

- 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.

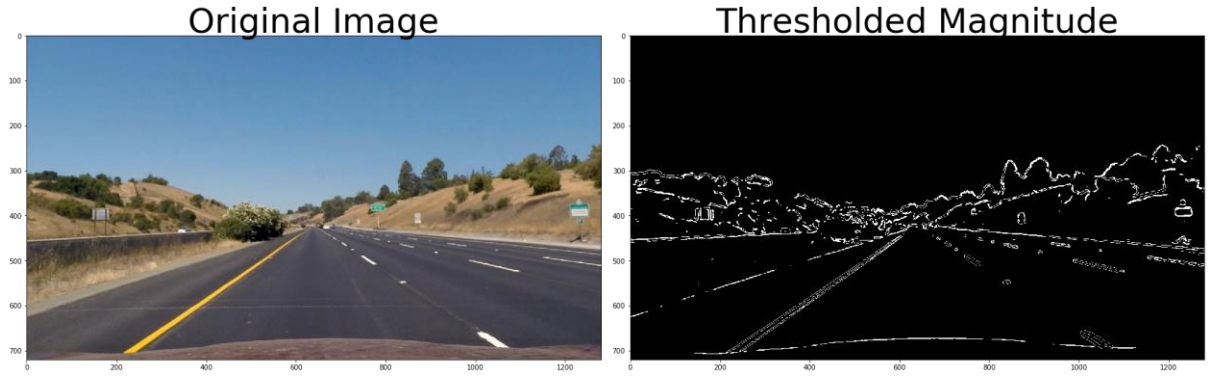

Figure 6 thresholded Magnitude

- 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.

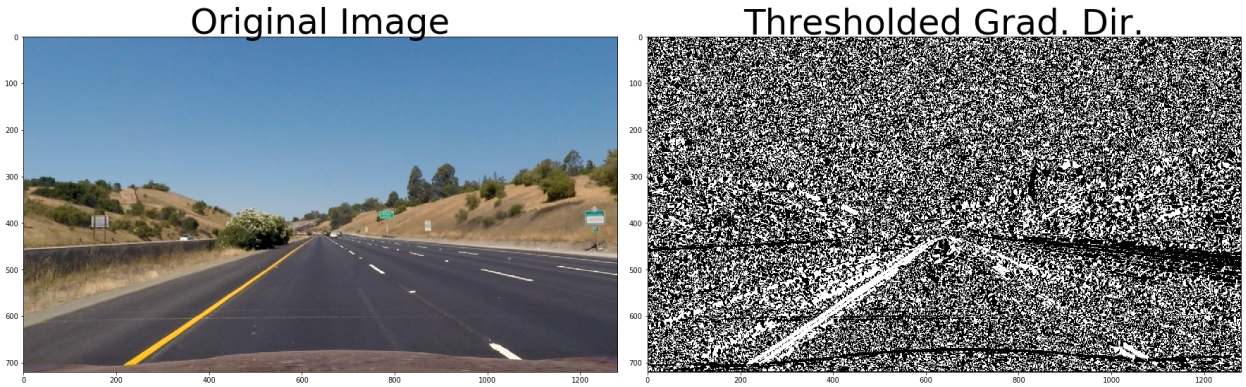

Figure 7 Thresholded gradient direction

- 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

Since the S channel thresholding in the HLS space is not good enough to detect the right lane. I also tired the L channel thresholding in the LUV color space. Figure 9 shows the process image after the thresholding. It gives a more clear detection of the right lane than the S channel thresholding in the HLS space. In my implantation, I set the minimum threshold as 225 and maximum threshold as 255. Please refer to luv_select in my code for the implementation details.

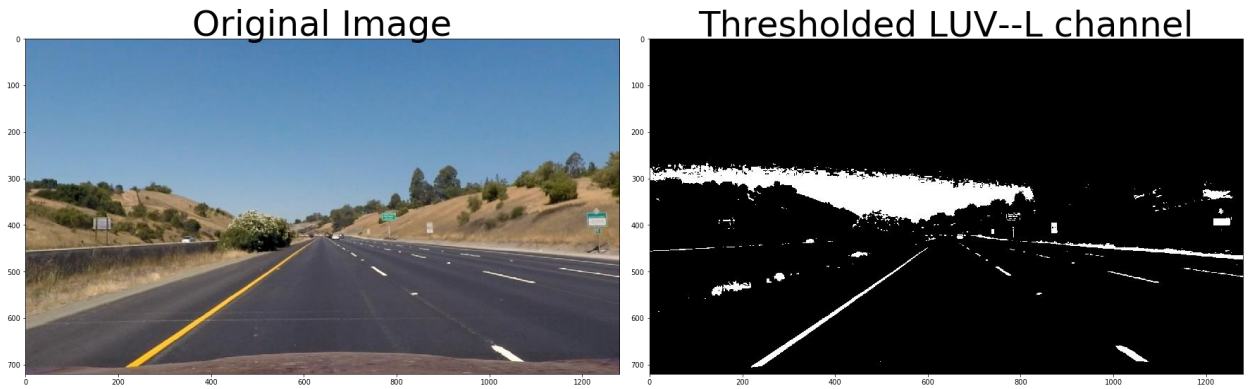

Figure 9 Thresholded L channle in LUV space

- Apply combined color and gradient thresholds

From the last 4 steps, I applied each thresholding separately. Now in this step, I combined all the thresholds and apply all of them together to reach a better processing performance. I firtly set the logic of all the threshold as follows: ((grad_binary_x == 1) & (grad_binary_y == 1)) & ((dir_binary== 1) & (mag_binary == 1)) & (hls_binary)==1, but the performance was not good enough. After some trials (also I realized that the color thresholding itself already gives good detection), I decide use the following combination: ((grad_binary_x == 1) & (grad_binary_y == 1)) & ((dir_binary== 1) & (mag_binary == 1)) | (hls_binary ==1) | (luv_binary ==1)== 1. The process result is shown in Figure 10. Please refer to Combine_all_thresholds(grad_binary_x, grad_binary_y, mag_binary, dir_binary, hls_binary) in my code for implementation details.

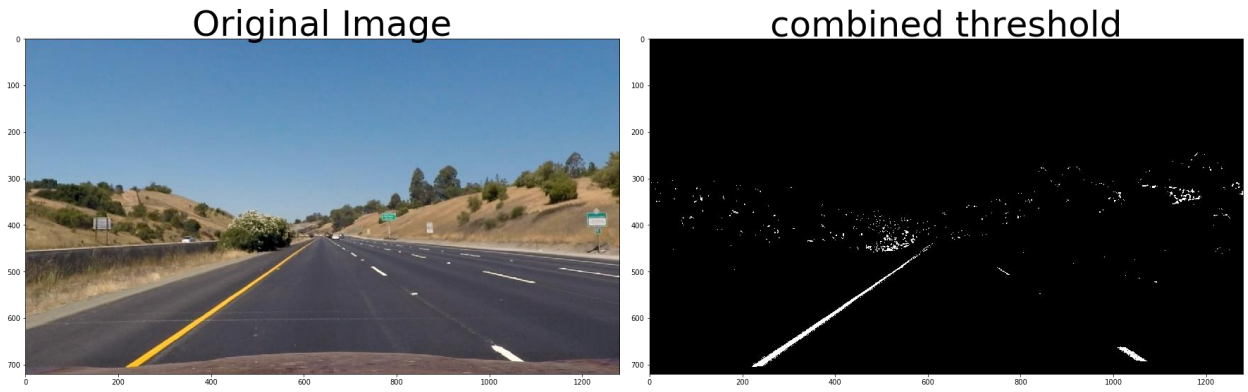

Figure 10 Combined thresholding

- Bird view perspective transform

In this step, we transform the perspective to a view that is most convenient for the processing, in this case, the bird view. This will be useful for calculating the lane curvature later on.

The solution for perspective transforming is quite straightforward. Firstly, I decided the source and destination points. In my case, I decide the source points (from bottom left to top left to top right to bottom right) as: src = np.float32([[290,695], [600,450], [730,450], [1100,695]]), and the destination points as: dst = np.float32([[260, 700], [260, 0], [1080, 0], [1080, 700]]). Then, I used cv2.getPerspectiveTransform(src, dst) to calculate the perspective transform matrix.

In addition, I also calculated the inverse perspective transform matrix in this step by swapping source and destination points. The inverse perspective transform matrix will be useful later on when drawing the lane area in the road.

Finally, I used cv2.warpPerspective to create the image with bird view.

Figure 11 presents the image after transforming to bird view perspective. Please refer to Bird_View_Transform(img) in my code fort the implementation details.

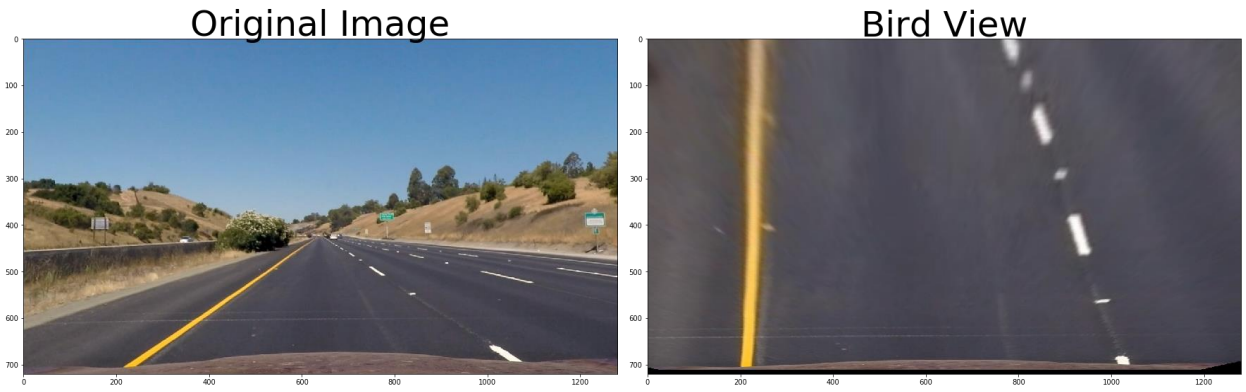

Figure 11 Bird view perspective transform

- Detect lane

This is the most complicated and important step for the whole pipeline. This step in my implementation consists of 2 functions: detect_lanes and Fine_lane_boundary.

The very first step is to calculate a histogram of where the binary activations occur across the image. The two peak are the positions of the left and right lanes. I took a histogram of the bottom half of the image. Then I tried to find the peak of the left and right halves of the histogram. These will be the starting point for the left and right lines.

After having the starting points for the two lanes, we take the sliding window approach to proceed with our process. Before using the sliding window, I need to configure the number, margin of width , and the height and the minimum pixel number to recenter. After the configuration, I loop through each window in nwindows. Then I try to find the boundaries of my current window. Now that I know the boundaries of my window, I later try to find out which activated pixels from nonzeroy and nonzerox above actually fall into the window.and I append these to our lists left_lane_inds and right_lane_inds. If the number of pixels I found in Step 4 are greater than your hyperparameter minpix, re-center my window. After I have found all pixels belonging to each line through the sliding window method, I fit a polynomial to the line. Figure 12 shows the result after fitting the polynomial. Please refer to det(img)ect_lanes for implementation details.

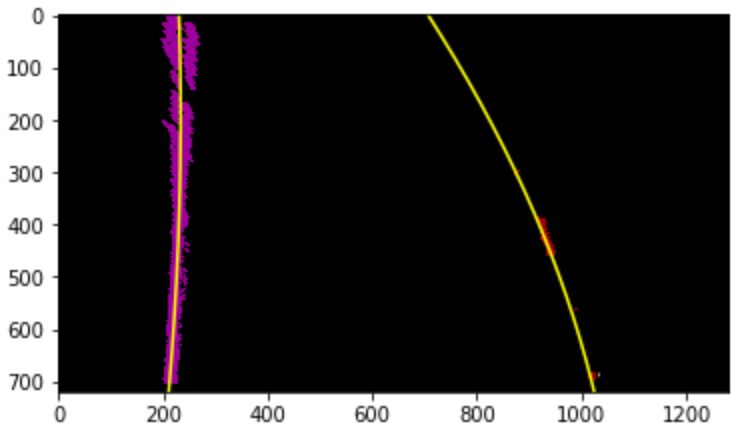

Figure 12 Detect lanes

As the lanes don't necessarily move a lot from frame to frame, in the next frame of video I don't need to do a blind search again, but instead I can just search in a margin around the previous lane line position. This can make the lane detection more efficient. The green area shown in Figure 13 is where I search for the lanes. Please refer to Fine_lane_boundary in my code for the implementation details

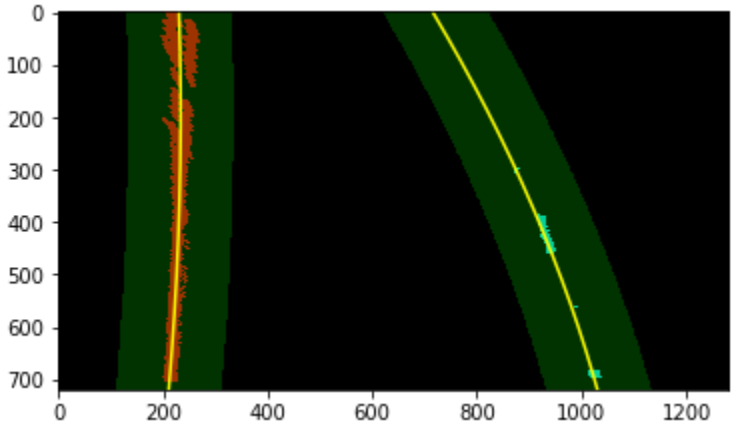

Figure 13 search_around_poly

- Show the detected lane area

In this step, I show the detected lane area in green on top of the original picture as shown in Figure 14. Please refer to drawing_lane in my code for the implementation details.

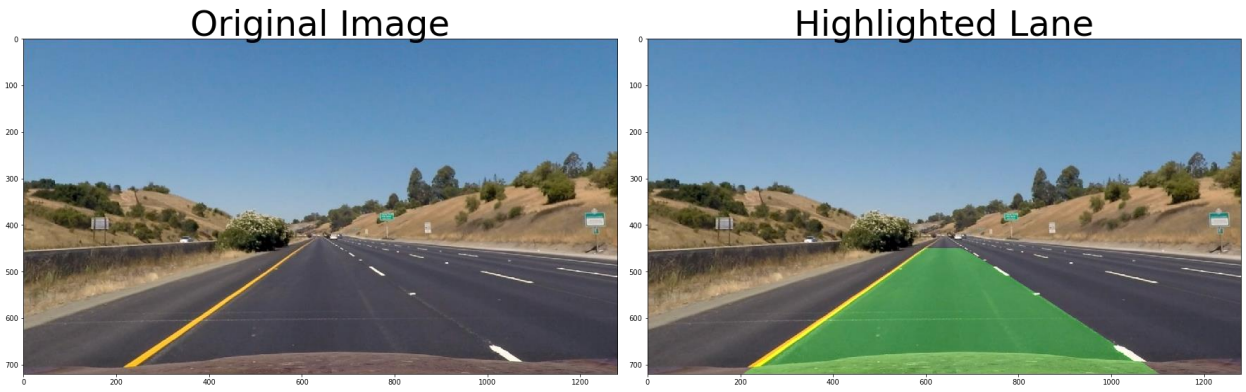

Figure 14 hIghlighted lane area in green

- Calculate and Output the curve and vehicle position

In this step, I calculate the curvature of both left and right lanes and the vehicle position with respect to the lane center. In Figure 15, you can see the final result with the curve and position data in the picture. Please refer to measure_real_curves and Vehicle_Position in my code for the calculation of curvature and vehicle position. Please refer to Output_numbers in my code to see how I show the calculated data in the picture.

Figure 15 Dsiplay curve and position

3. Discussion

3.1 Potential Problems of my solution

One problem that I already found is that my curvature is not accurate enough. As suggested by the tips and tricks for the project, the curvature is around 1km.Even though, the rough average of my calculated curvature is around 1 km, sometimes it can drop to 0.5 km while sometimes it can also jump to 2km.

My pipeline might fail if there are many cars around, especially white cars. As I can see in the last couple of seconds in the testing video, when there is another white car in front of me, my pipeline tends to become unsteady. Now, I took the advice from the reviewer to add the L thresholding in the LUV color space for the lane detection. The performance is much better. My lane detection does not got confused by the neighboring cars any more.

If the environment is more completed with bad lighting, bad weather, or when the drivers need to change lanes, my solution might also fail.

3.2 Future improvements

For achieving a more accurate curvature, one thing I can do is to add a sanity check function to see if the curvature calculated is right or wrong. Also, it is necessary to collect more real-world data about lane width and length in different countries and use machine learning to assist to calculate the accurate curvature.

Currently, I am only using the S channel thresholding in the color space. In real world situation, it will not be sufficient. A more complex combined thresholding needs to be explored, including also H and L channel in the HLS space, or even further, considering also RGB color space.

Due to time limits, I didn't get to try "Smoothing" suggested in the lecture to remove some outliers to have a cleaner result. Therefore, future improvement can be adding a feature to take the average of the most recently N measurements to obtain a more stable performance.

For the challenges of lanes in bad weather or lighting, and lane change scenarios, I guess the only best solution is deep learning to have a more intelligent sense. I look forward to the next lectures to give me more knowledge about machine learning.