Advanced Lane Finding Project

Ximing Chen

I. Goals and Steps

The goals / steps of this project are the following:

- 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.

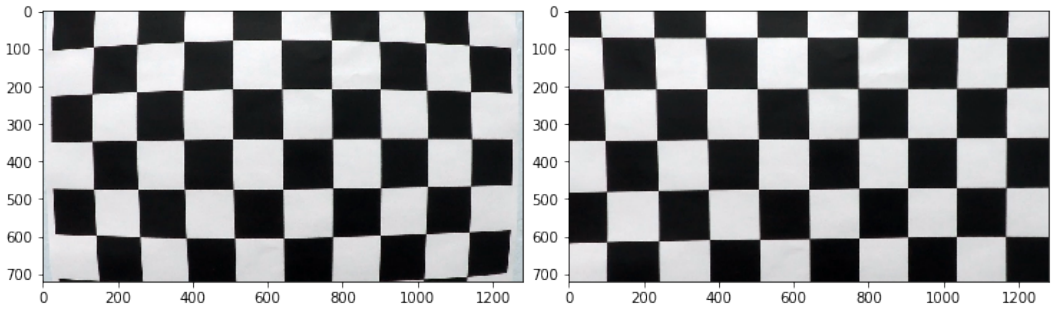
Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

II. Camera Calibration

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, object points 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.

In my experiments, I have used a mesh-grid of $n_x = 9$ and $n_y = 6$ to create 54 object points. The image points are obtained from using the 'cv2.findChessboardCorners()' function. Thus, it finds 54 points on the image. Next, I used the output 'objpoints' and 'imgpoints' to

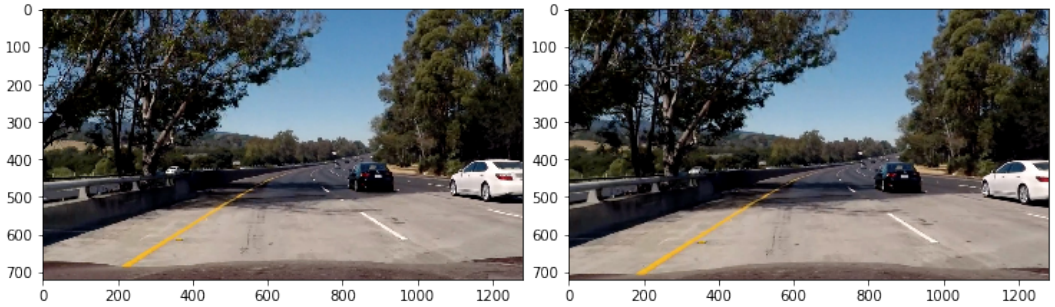
compute the camera calibration and distortion coefficients using the ‘cv2.calibrateCamera()’ function. I applied this distortion correction to the test image using the ‘cv2.undistort()‘ function and obtained the following result:



(a) (b)

Figure 1: In (a), we show the original image. In (b) we show the undistorted image.

I apply the distortion correction to one of the test images:



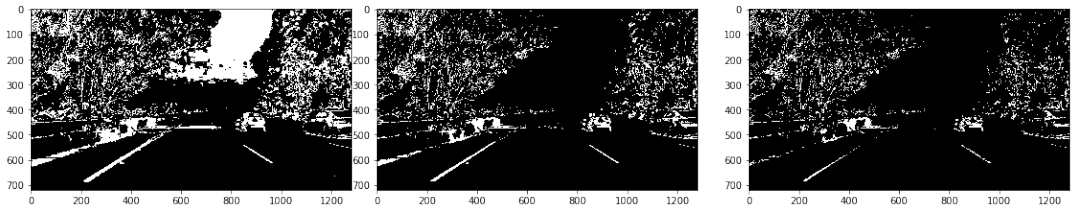
(a) (b)

Figure 2: In (a), we show the original test image. In (b) we show the undistorted test image.

III. Thresholding Color Images

After the distortion correction process, I begin to apply filters to the color images for better detection of the lanes. I used a combination of color and gradient thresholds to generate a binary image. To get which filter combinations work the best, I use one of the test images, as depicted in Figure 2-(a).

First, I transform the image from RGB domain to HLS domain. The reason for this is that the lanes usually have high satuation values despite they may have different colors. Thus, HLS domains are more robust hence more suitable for our goal. We use a threshold for the S-channel. More specificially, only the pixel values within the range $[threshlow, threshhigh]$ are taken, otherwise they will be set to 0. The S-channel is shown as follows: We observed

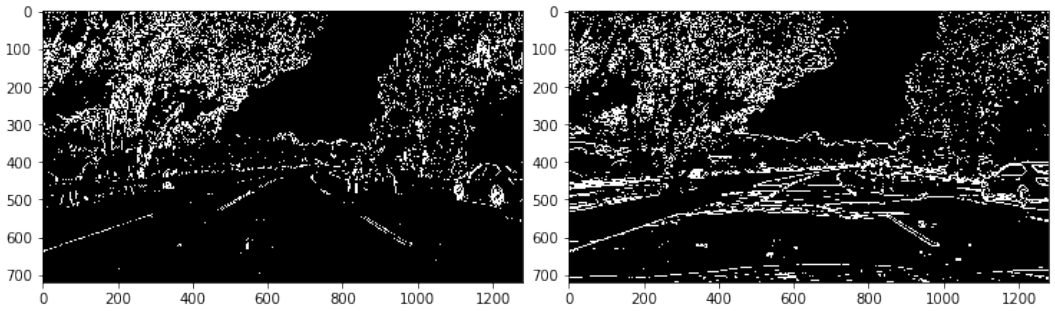


(a) (b) (c)

Figure 3: In (a), the thresh is set to be $[100, 255]$. In (b), the threshold is $[150, 255]$. In (c), the threshold is $[200, 255]$.

that the best threshold is around 150. In this case, we preserve most of the details on the lanes while removing unnecessary information, e.g., the sky as in Figure 3-(a).

Nonetheless, using only the S channel is not enough to provide good quality of the lanes. Thus, we consider the following three additional steps. First, we apply a sobel filter – a filter typically used to detect the edges – on the x and y-direction of the image, respectively. The performance is shown as follows: Notice that sobel-x performs much better than sobel-y.



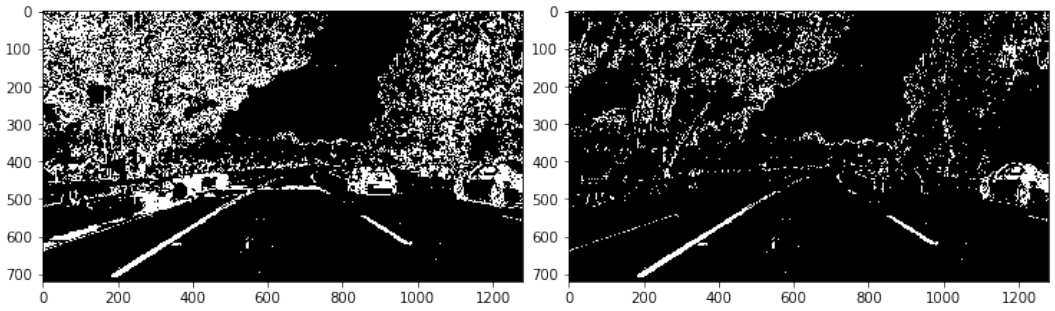
(a) (b)

Figure 4: In (a), we show the sobel x filtered image with threshold $(20, 255)$. In (b), we show the sobel y filtered image with threshold $(20, 255)$.

This is due to most of the lanes are perpendicular to the bottom-most boundary of the image.

Consequently, the lane colors have high value changes on the x direction. The threshold are obtained using trail-and-error.

We then combine the S-channel information with the binary image obtained using sobel-x filter. However, this combination is unable to remove the shadows on the road, no matter how the thresholds are tuned, as shown in Figure 8-(a). To address this issue, we use the L-channel. The intuition behind this is that the lanes are usually of higher light intensity than the shadows. Thus, we aim to select the 'light' and 'highly saturated' areas.



(a) (b)

Figure 5: In (a), we show the figure obtained using HLS and sobel-x. In (b), we show the sobel y filtered image using S-channel, L-channel and sobel-x.

IV. Perspective Transform

The code for my perspective transform includes a function called 'perspective transform()', in the 10-th code cell of the IPython notebook. The function takes an image, as well as source ('src') and destination ('dst') points. I chose the hardcode the source and destination points in the following manner:

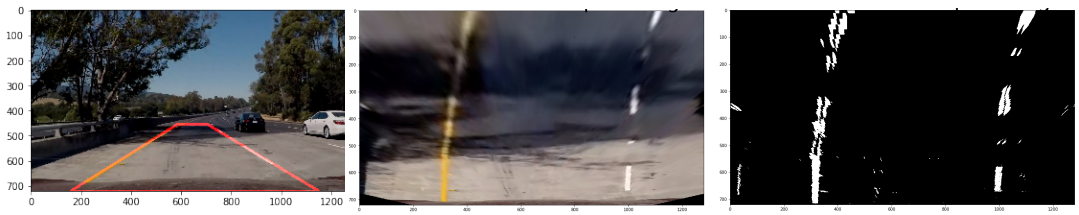
```
offset = 150
left_bottom_x = 160
right_bottom_x = 1150
left_top_x = 585
right_top_x = 705
src = np.float32([[left_top_x, 455], [right_top_x, 455],
[right_bottom_x, 720], [left_bottom_x, 720]])
```

```
dst = np.float32([[left_bottom_x + offset, 0],
[right_bottom_x-offset, 0],
[right_bottom_x-offset, img_size[0]],
[left_bottom_x + offset, img_size[0]]])
```

This results in the following transformation:

Source	*Destination*
(585, 455)	(310, 0)
(705, 455)	(1000, 0)
(1150, 720)	(1000, 720)
(160, 720)	(310, 720)

Table I: This table shows the source and destination point of perspective transform.



(a) (b) (c)

Figure 6: (a) selected region. In (b), the bird-eye-view of the road is shown. In (c), the undistorted warped lanes are shown.

I verified that my perspective transform was working as expected by drawing the 'src' and 'dst' points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image, see Figure 6 for more details.

V. Lane Pixel Detection and Fitting

To detect lane pixels, I basically follow the following steps:

1) Plot the histogram of the number of pixels across different x-positions using only bottom half of the images;
2) Identify the peaks of the histogram iteratively using sliding windows, in this project, we have used 9 windows in total.

To fit the polynomial, I used the 'np.polyfit()' function in which the arguments are the identified pixel locations corresponding to the lanes in y-direction and in x-direction respecitvely. The degree of the polynomial considered equals to 2. The identified polynomial is shown below:

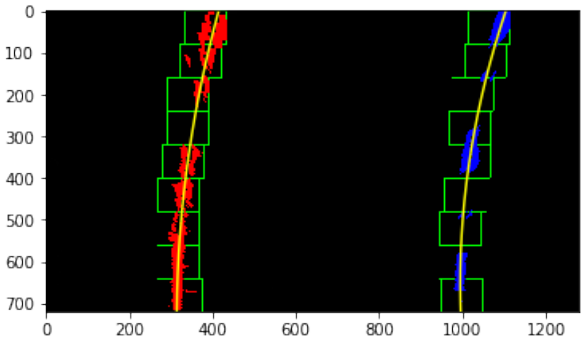


Figure 7: The sliding windows and the fitted polynomial

VI. Calculations

We calculated two related information: the curvature and the difference with respect to the center of the lanes.

The curvature is calculated as follows:

$$R = \frac{(1 + (2Ay + B)^2)^{\frac{3}{2}}}{|2A|}, \tag{1}$$

where A, B are the coefficients of the fitted polynomial $f(x) = Ax^2 + Bx + C$. The value y is chosen to be equal to the size of the image, as we are interested in the curvature of the lane at the front of the car. However, the calculated curvature is in the pixel-unit, instead of in meters. Thus, we consider doing the following remaping and recalculation.

First, we re-estimate the coefficients of polynomials. This is achieved by scaling the pixel locations by the following factors: $x_f = \frac{3.7}{700}$ and $y_f = \frac{30}{720}$, i.e., x_f represents 3.7 meters per 700 pixels. Then, we can calculate the curvature using the above formula, where A, B are the coefficients of polynomial fitted using scaled units and y equals to the size of the image times the scaling factor y_f.

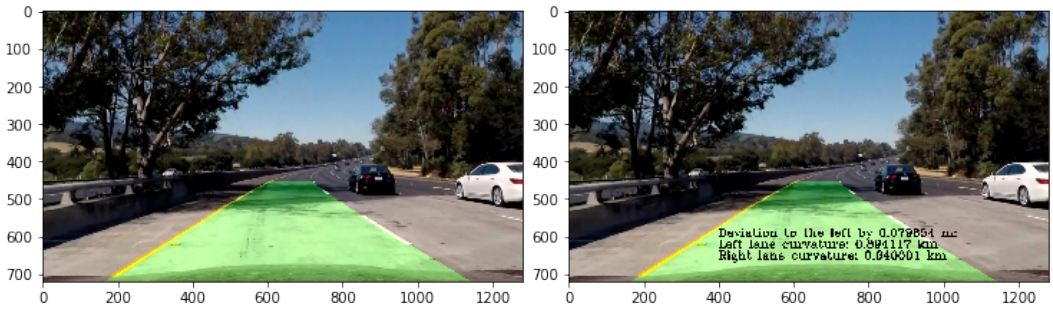
Similarly, to get the difference with respect to the center of the lanes. We first identify the location of two lanes – in meters unit. Then, the center of the lane equals to the average of

these two values, whereas the center of the car equals to $640 * x_f$ (meters), since the width of the image is 1280 pixels. Then, the difference is the mid position minus the center of the car.

I coded these logics in the functions: 'get bias()' and 'set curvature' in the Line class.

VII. Remapped Image

Finally, I mapped the identified polynomials into the original image using inverse-perspective transformation and annotate the image, as



(a) (b)

Figure 8

VIII. Video

Please see the zip file for video.

IX. Discussion

The performance of the lane tracking mostly relies on how good the filters are. When the filter cannot efficiently distinguish what are the pixels corresponding to the lanes, then all following work may be useless. In this case, simply using HLS and sobel filter may not be enough. There are a few instance when this fails: (i) lack of samples, which occurs when another car in front is changing lanes into ours, resulting in blocking the lanes, and (ii) poor samples, the lanes are not colored well. We may consider using prediction from the previously identified polynomials since the lanes are almost continuous, thus, they must follow relatively good geometric shape.

Also, if the driver changes a lane, then our method may fail to work since we have assumed that there are only left, and right lane on the road. To make it more robust, it is necessary to consider using a non-static method to add lanes into the image, for example, we plot all the lanes detected into the figure.