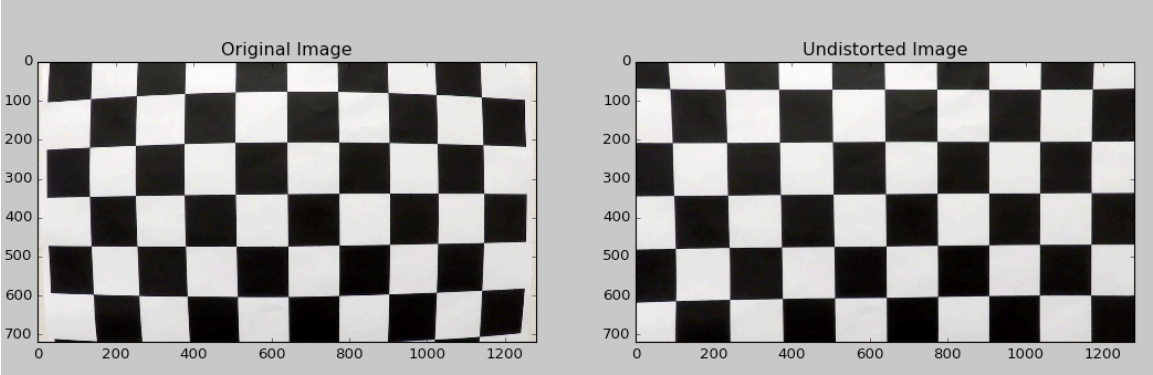
Self-driving car Nanodegree P4

Note: The rubric is copied and pasted in this document, followed by descriptions of my implementation details. The processing methods in processing pipeline and parameters are call coded in object *Filter* in filters.py. The image processing pipeline is packed into member function called pipline(self, img).

Also note that I have so far only succeeded in processing project_video.mp3. For challenge and harder video I will leave them for later work as I have no time left (still need to revise P2 and P3 submission).

Camera Calibration

The file that performs camera calibration is called "calibrate.py". The methods used in my implementation are exactly the same as described in class videos. All calibration images are used, their object points and image points (image points being the points where two black boxes touch each other) stored in a list. Then cv2.calibrateCamera is called to compute camera matrix and distortion coefficients. However, cv2.findChessboardCorners returns failure for certain images. These images are discarded during calibration. Here is a sample of calibration result:



Pipeline (test images)

Distortion correction

For distortion correction in the pipeline, simply use the camera matrix and distortion coefficients found in calibration step, and opencv function cv2.undistort(img, self.mtx, self.dist). Here is a sample output:

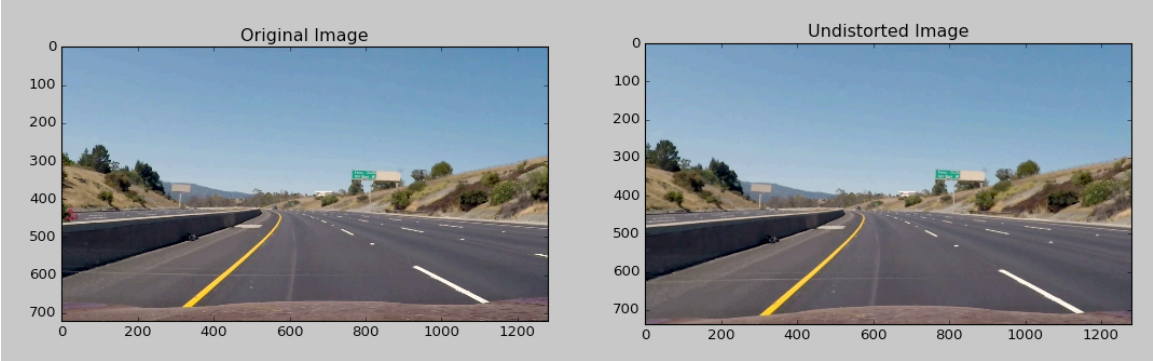


Image Thresholding

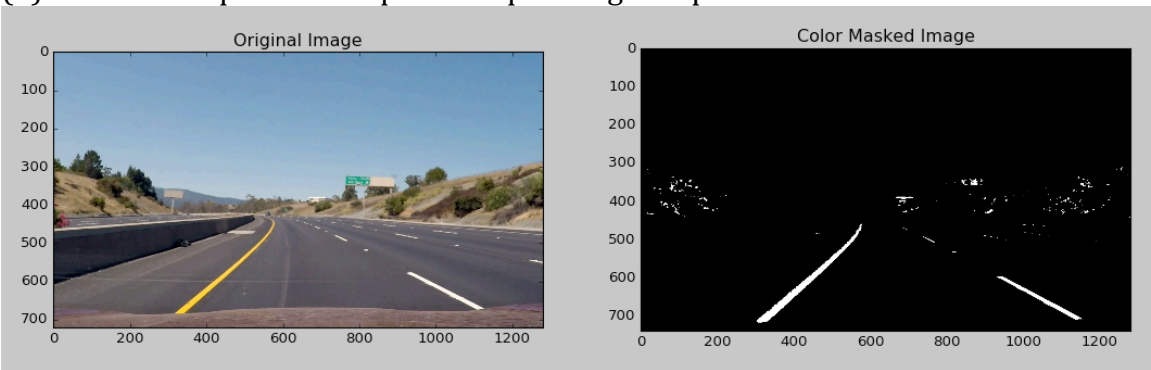
A explicit and implicit method are used to perform thresholding on the image. The function for thresholding is thresholding(self, img). Although methods for finding edges using sobel filters are implemented, they are not used in actions for this implantation, as I find that color threshoding is sufficient for project_video.mp3.

First the explicit method. The method name is color_mask(self, img). Only color thresholding is used here. The color thresholding consists of 3 steps.

(1)The original image is first transformed to HLS color space. The S channel needs to be normalized and multiplied by 255 for succeeding thresholding. Then the hls_binary is generated by:
hls_binary[((H <= 24) & (H >= 18)) & (S >= 100)] = 1

(2)Step one work well for yellow pixels. To select white pixels, first the original images are converted to grayscale, then the gray_binary is generated by:
gray_binary[gray >= 200] = 1

(3) Combine step#1 and step#2. Sample image output:



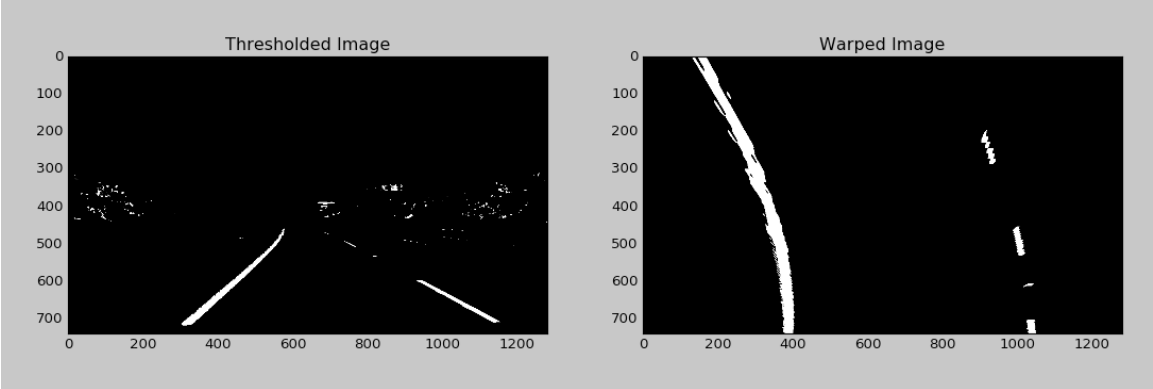
The implicit thresholding is the perspective transform, which will be covered next.

Perspective transform

warp(self, img, dbgcode=0) performs the perspective transformation(dbgcode is just for my debugging use). Four points are selected manually from the original image so that the resulting perspective transformed image contains lane lines that are nearly parallel.

Here we see that perspective transform acts like area selection. In the sample below, only the lane pixels are included in the warped image output, whereas the noise from the background (such as trees, nearby cars) is excluded.

Here is a sample output.



Lane-line pixel identification and polynomial fit

Four steps are involved in this process.

(1) sliding(self, img, sum_th), where sum_th can be understood as a sensitivity parameter which is set to 0.15*self.box_height*self.box_width. This function will return a list of coordinates that corresponds to the locations of lane pixels. The idea is to use a box with size (self.box_height, self.box_width) sliding across and performing dot product with the warped image.

Before sliding process, the histogram of the image is computed as introduced in the class video, as follows:

histogram = np.sum(img[int(img.shape[0] / 2):, :], axis=0)

The peak on the left and right are found by taking x values that produce a maximum value for the above equation, as follows:

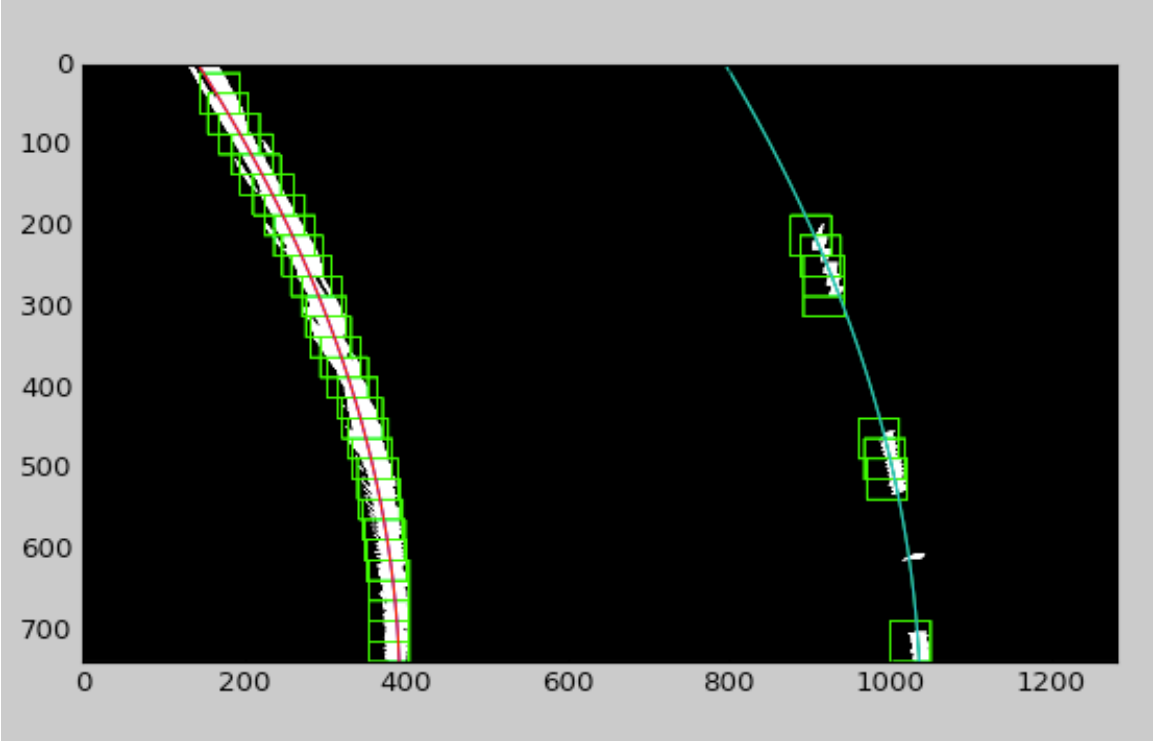
left_start = np.argmax(histogram[:int(img.shape[1] / 2)])
right_start = np.argmax(histogram[int(img.shape[1] / 2):]) + img.shape[1] / 2

We therefore have a starting point for the succeeding sliding process. As the np.sum function is relatively fast, I have not yet seen significant advantage of using historical data found in previous image. However, I do have a small threshold above which the maximum y values the histogram must meet, otherwise the left_start/right_start from previous image will be used. It's intended mainly to reduce noise.

The sliding process will be performed horizontally and vertically. Horizontally, the sliding box will move from a negative offset to a positive offset with regard to the starting point (lef_start and righ_start). Then for each horizontal sliding, the maximum dot product found during sliding is considered as the center pixel of the lane lines for that particular horizontal stripe, if it is greater that specified sensitivity parameter(sum_th). If it is less than the sensitivity parameter, then that result will be discarded and not returned as a lane line pixels. In the sample image below, the centers of green boxes indicate the returned coordinates. As we see it identifies the lanes well.

(2) After locating the coordinates from step, bit-wise AND is performed between the warped image and these boxes(pixels inside boxes are all set to 1). The resulting image is called filtered_by_box_image_left and filtered_by_box_image_right for left lane lines and right lane lines.

(3) Filter.poly_fit(self, filtered_by_box_image). This function takes filtered_by_box_image_left and filtered_by_box_image_right and compute the polynomial fitting for left lane lines and right lines. In the sample image below, the red line and the blue-ish line is the resulting of polynomial fit.



Moving average of polyfit

The function is get_fitx_from_history(self,fitx_left,fitx_right). This function take the output from the polynomial fitting from last steps, store the x coordinates to a list of lists, and return the average of the last 10 lists. It will reduce the jittering of the final detection result of the pipeline.

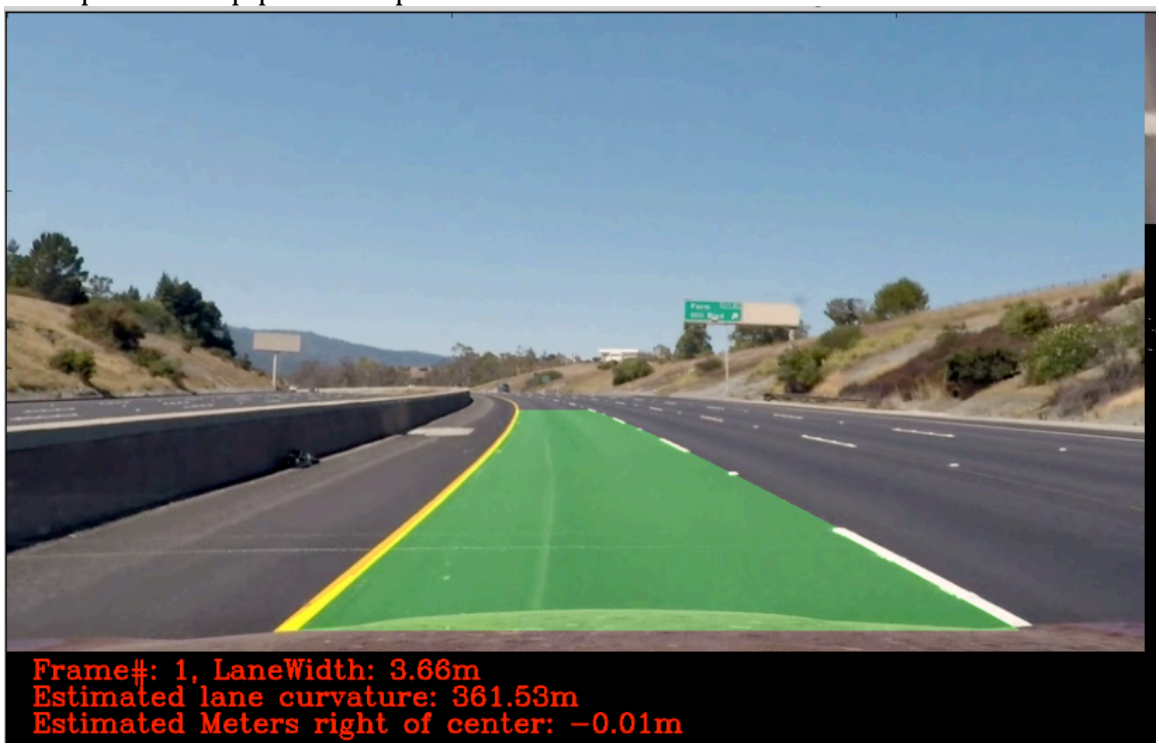
Radius of curvature & position of the vehicle

The function to calculate curvature is curvrad(y_left, fitx_left, y_right, fitx_right). It's exactly the same as described in class video.

The function to calculate vehicle position is estimate_center_offset_and_lane_width(self, y_left, fitx_left, y_right, fitx_right, warped_shape). The center of the vehicle is assumed to be the middle pixel of the warped image, plus a fixed offset (which is determined manually from observing the video). The center of the road is calculated by taking the middle of the computed polynomial representation of the left and right lane lines. The lane width is also computer in the same process.

The sample image shown in the next steps will have the results for curvature, vehicle position, and lane width.

Example of final pipeline output:

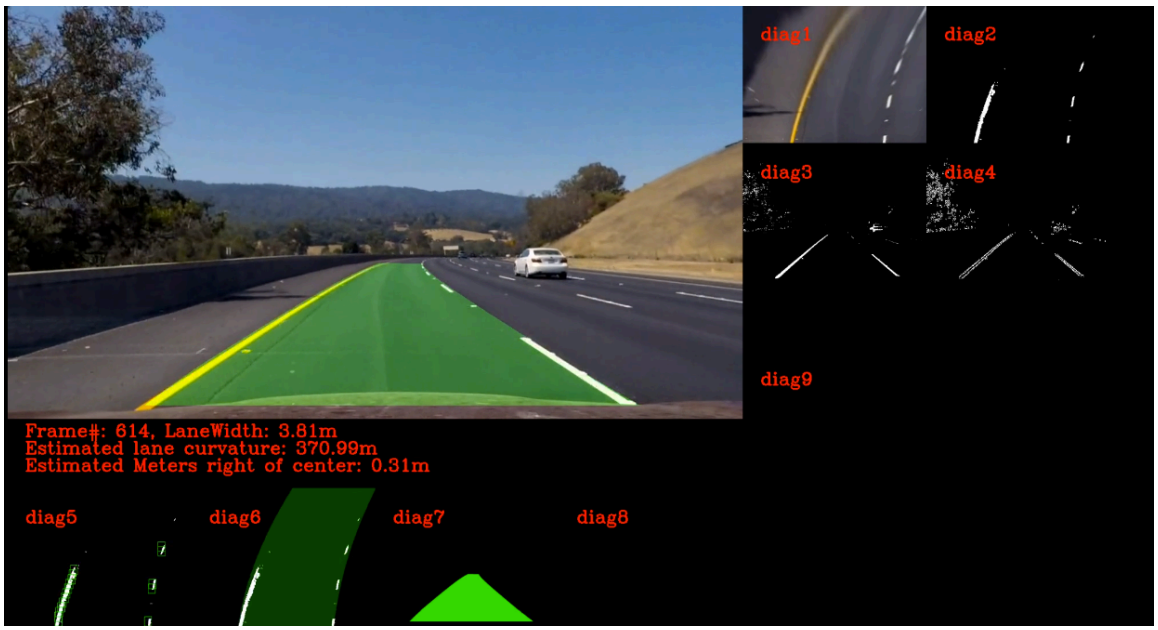


Pipeline (video)

Please find the video output in this folder named “project_video_out.mp3”.

Discussion

My lane detection pipeline depends heavily on color thresholding. It will need to be made more robust. For future improvement, first refer to the following screenshot:



“diag1” is the image after perspective transformation. “diag2” is the image after applying the color thresholding described in earlier session. The yellow line in the upper part of “diag1” is visually clear to a human being, but can’t be recogonized by the pipeline. I suspect this is one reason while my pipeline fails for the challenge video. But I have yet to verified this reasoning. Potential solution includes adding more thresholding such as the sobel filter. But I need to understand how a sobel filter and be combined with color thresholding to specifically solve this particular frame.