ADAVANCED LANE FINDING PROJECT

Faisal Waris

Udacity
Self-Driving Nano Degree

Introduction

The goal of this project is to perform several image processing steps to identify lanes lines and measure the lane curvature.

The basic steps are as follows:

1. Calibrate camera to remove distortion
2. Threshold images to surface the salient features for lane detection
3. Apply perspective transform to obtain a bird's-eye-view of the road for lane detection and curvature estimation
4. Identify lane lines and measure curvature
5. Project the detected lane boundaries onto the original image for validation and viewing

The steps outline above are explained in detail in the subsequent sections of this report. The final section is the discussion of the results.

The links to source code and other artifacts relevant to this report are given below:

GitHub repository:	https://github.com/fwaris/CarND-Advanced-Lane-Lines
Processed video file:	https://github.com/fwaris/CarND-Advanced-Lane-Lines/blob/master/output_videos/project_video_lanes.mp4
Sample output images folder:	https://github.com/fwaris/CarND-Advanced-Lane-Lines/tree/master/output_images

The pipeline processing code is organized into the following modules (files):

Module / file name	Function	Link to source in repo
BaseTypes.fs	Contains a structure to encapsulate hyperparameters such as thresholds window sizes and other constants for automated tuning of hyperparameters if need be. This structure is passed to most functions in the code and each function uses the hyperparameters from the structure as needed	link
CalibrateCamera.fs	Calibrate camera	link
ImageProc.fs	Processing single images for thresholding, perspective transform, etc.	
LaneFind.fs	Find lanes by performing processing over pixels of the 'thresholded' and transformed images	link
LineFitting.fs	Fit hyperbola curves over the identified lane line pixels	link
VideoProcessing.fs	Contains the processing that puts everything together to process a video file	link

Camera calibration

The camera calibration was performed by using the give set of calibration images in the project repo (calibration 1-20). I followed the approach given in the OpenCV tutorial. Essentially the provided set of calibration images were used to collect 'object points' and matching 'image points' from OpenCV FindChessboardCorners function. Figure 1 is an example of visualizing the found chessboard corners.

Figure 1: Chessboard corners visualized

The object points are a static collection of points roughly corresponding to the number of board squares (link to code to generate static points). The code to generate the collection of image points from the calibration images is given here. The collection of points was used to derive a 3x3 camera matrix to remove distortion, using the OpenCV CalibrateCamera function.

The derived camera matrix was then used to undistort the original camera image as show in Figure 2.

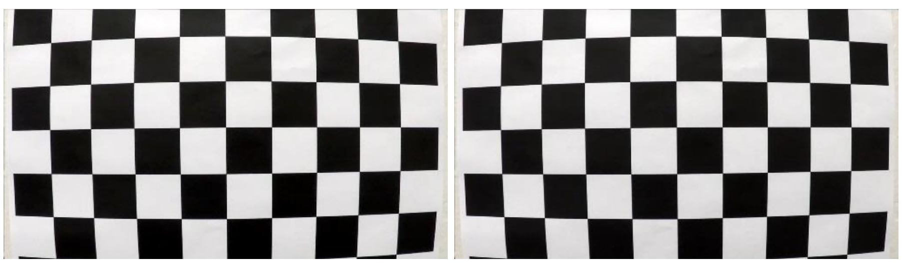

Figure 2: Calibration original and undistorted image

Unfortunately the undistorted image does not look much different from the original image. I am not sure where the issue lies, even though the process follows seems to be correct. For this reason, the camera calibration was not used for pipeline processing. I did check that the camera matrix has has non-zero values. It may be the API call to convert 2-dimensional float array to an InputArray OpenCV object is not functioning correctly.

Color and Gradient Transforms

The input image was processed with 3 types of thresholds:

Threshold	Comments	Code Link
Yellow color isolation	Used HSV color format to isolate yellow color in images such as those for solid lane lines	link

	OpenCV methods CvtColor and InRange were utilized	
White color isolation	Same as above for white color. The yellow and white color masks were bitwise ORed using OpenCV BitwiseOr method	[link]()
Gradient threshold	The OpenCV Sobel function was used to find gradients in the x direction and then a mask was created that contained pixels where gradient is in a specified range	[link]()
Combination	The results of the yellow and white color masks was combined with the gradient mask using OpenCV BitwiseOr function	[link]()

Figure 3: Original and 'thresholded' images

An example of the orginal and thresholded version of the same images are shown in Figure 3.

Perspective Transform

To create a birds-eye-view perspective transform, 4 points where selected on the camera image and then their transformed points where determined using human judgment.

The source and destination transformation points are given below:

Source Points List = [691,450; 1150,720; 250,720; 597,450]

Destination Points List = [1150,0; 1150,720; 250,720; 250,0]

A perspective transform matrix was created using OpenCV GetPerspectiveTransform function.

The original and perspective-transformed images are show in Figure 4. The transformed image was created with the OpenCV WrapPerspective function.

Figure 4: Original and perspective transformed image

For the purpose of performing lane detection however, the perspective tranform was applied to the thresholded image rather than the original image. Figure 5 is an example of such processing.

Figure 5: Perspective transform applied to thresholded image

Lane Detection

The input for detecting lane pixels for line fitting is the thresholded and warped image as show in Figure 5. The lane detection steps and code links are described in the following table:

Step	Description	Link to source
Find left and right lane start points	Use a frequency count of pixels in each column along with a sliding window to search for left and right lanes in the two vertical halves of the image, respectively	link1 and link2
Find left and right lane pixels	The sliding rectangle (window) search method was used to locate the pixels for the left and right lane pixels	link
Line fitting	A x and y dimensions of the pixel points are swapped fit parabola equations that fit will to the lane curvature. The ILNumerics FitPolynomial method was used to find the equation of the hyperbola	link
Lane marking	Using the two left and right curve equations, the lane section was drawn on an empty Mat using OpenCV polyfill	link

	function. This Mat was inverse perspective transformed and merged with the original image using several OpenCV functions. Figure 6 shows the result of a combination of processes to illustrate the identification of lanes on a warped and thresholded image. Figure 7 is an example of a fully processed frame.	

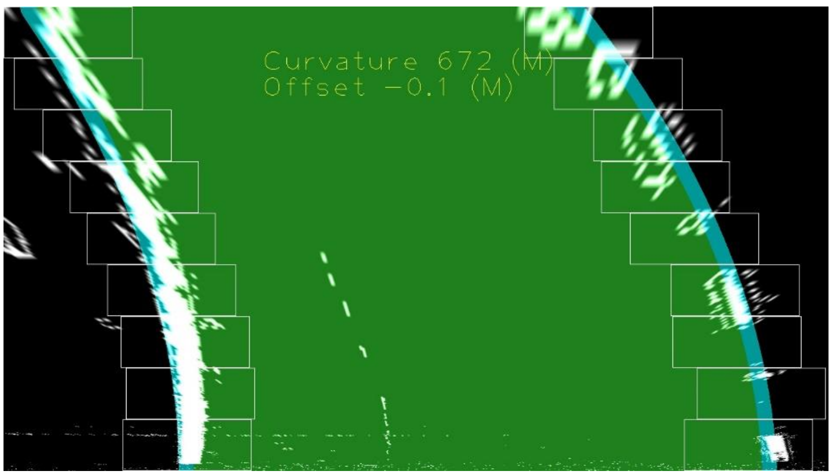

Figure 6: Lane curvature calculated and marked on warped image

Figure 7: Example fully processed frame

Curvature Measurement

To estimate the curvature of the road section in meters, the following processing was performed:

Step	Description	Link to source
Pixel to meters conversion	The estimated lane equations need to be re-estimated in world-space (in meters). The pixel-to-meters conversion factors, given in the lecture were used for this processing	link
Re-estimate lanes in world space	The lane equations were used to first express line points in pixel space. These line points were transformed into world space using the conversion factors described earlier. Note the X dimension (which really is the swapped Y dimension from the camera frame) was flipped so that vehicle is located as X=0 (instead of X=720). The word-space points were used to re-estimate lane equations in meters.	link
Measure curvature	Once the lanes were estimated, the curvature at the bottom of the camera frame was measured using the formula given in the lecture	link
Curvature smoothing	The left and right lane curvatures were averaged and smoothed over several frames before rendering as text on the output frame	link
Position offset	The offset of the vehicle was calculated as the difference between the center of the frame and the midpoint of the two lanes from their respective start points. The start points where determined earlier during the start of frame processing.	link

Discussion

During the development of the code, several alternative image pre-processing steps where tried. For example, instead of ORing the color and gradient thresholds, bitwise AND was considered. It did not perform as well.

Not being a computer vision expert, this was a difficult project for me as it required a deeper exploration of the OpenCV API as compared to previous projects.

I was not comfortable with my camera calibration results described in the initial part of this report. Therefore, camera calibration was not used as image pre-processing before finding lane lines.

The code performs reasonably well on the project video but does not perform well on the challenge videos. I feel that greater mastery of the OpenCV API is required for me to effectively start tackling the harder videos.