# PROJECT 2: ADVANCED LANE DETECTION

**OBJECTIVES:**

- Compute the camera calibration matrix and distortion coefficients using a set of chessboard images
- Calibrate raw images using the predetermined coefficients
- Create a thresholded binary image using a right combination of color and gradient thresholds
- Apply perspective transform to change to 'birds-eye-view'
- Detect lane pixels and find lane lines
- Determine lane curvature and vehicle position with respect to lane center
- Warp the detected lane boundaries back onto the original image
- Output visual display of the lane and estimates of lane curvature and vehicle position

**RUBRIC POINTS:**

The next section will describe in detail how I addressed each project criteria.

CRITERIA 1 – WRITEUP:

The first criteria of submitting a 'writeup' is fulfilled by this document.

CRITERIA 2 – CAMERA CALIBRATION:

The code pertaining to this section can be found in the second section of P2_Advanced_Lane_Finding.pynb file. The 3D coordinates of the chessboard corners in the real world are set as 'objpts', whose z coordinate is assumed zero with. After every successful corner detection on the test image, the 'objpts' are appended onto 'objp'. Similarly, the pixel coordinates of the corners will be appended onto 2D array 'imgpts' after each successful chessboard detection.

Using the OpenCV cv2.calibrateCamera() function, the camera calibration and distortion coefficients are computed feeding 'objpts' and 'imgpts' as inputs. The results are demonstrated below.

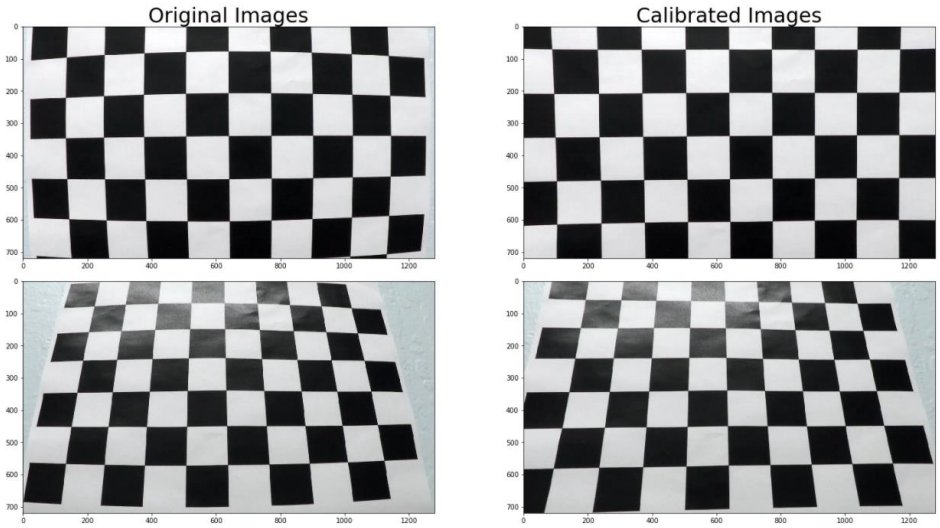


*Figure 1 Calibration output*

CRITERIA 3 – PIPELINE:

*Step 1: Raw image calibration*

The coefficients calculated from camera calibration are used within the user-defined 'calibrator' function to calibrate raw images, defined in 'Section 2: Function Definitions' of the code file. The results are demonstrated below.


*Figure 2 Camera Calibration: Original image vs Calibrated image*

*Step 2: Generating a thresholded binary image*

This was the most crucial part of the project as it determines how well lane lines are detected. There was a lot of experimenting with the thresholds of color, direction gradients, magnitude and combinations of all. The 'sthresh' function is defined in section 2 of code file. The best combination that seemed to detect lane lines in varying conditions is displayed below.

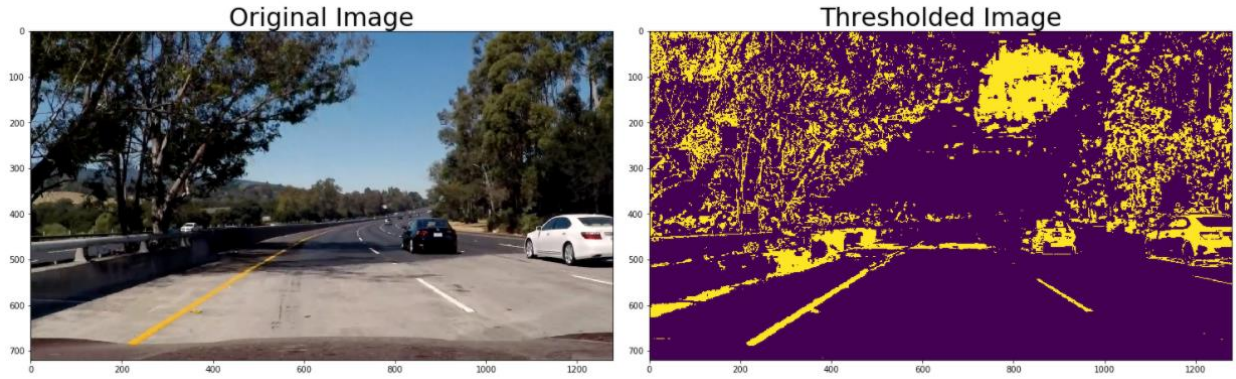


*Figure 3 Thresholded image detecting lane lines in varying road colors and shadows*

*Step 3: Applying a perspective transform*

The code for the 'persp_trans' function is within section 2 of code file. The function takes the thresholded image as input. The 'src' and 'dst' points were chosen manually based on the intuition gathered from perspective transform lesson and assignment.

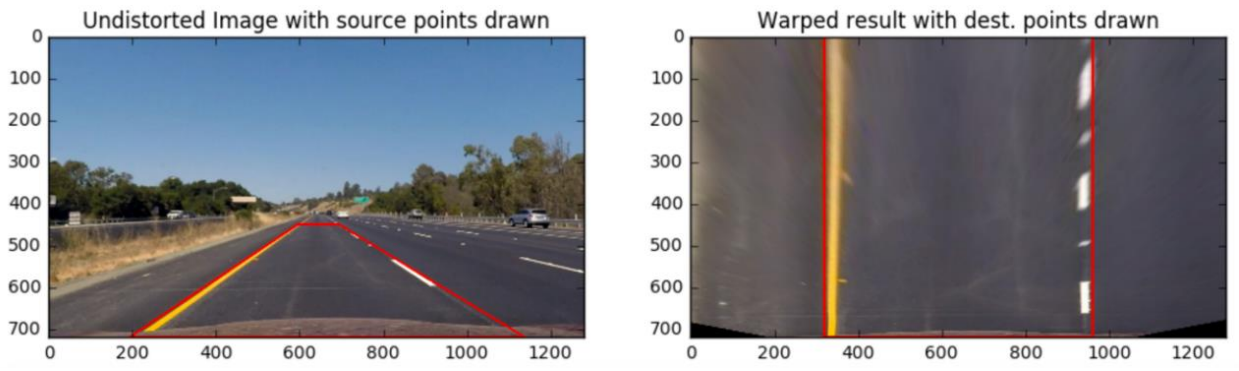


*Figure 4 How to select 'src' and 'dst' - Perspective Transform example from Udacity CarND course*

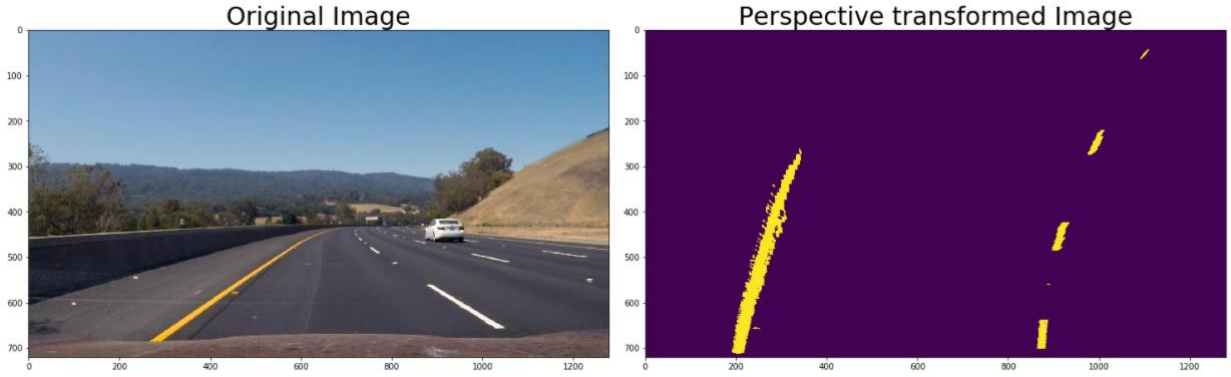


*Figure 5 Perspective transform output*

*Step 4: Lane line detection*

Lane lines are initially detected using the sliding window to detect lane pixels using the 'find_lane_pixels' function and later uses the information generated from previously detected pixels to simply interpolate and find lane pixels on the subsequent frames using 'targeted_lane_search' function, both taking perspective transformed binary thresholded image. This greatly reduces computation time and saves memory.

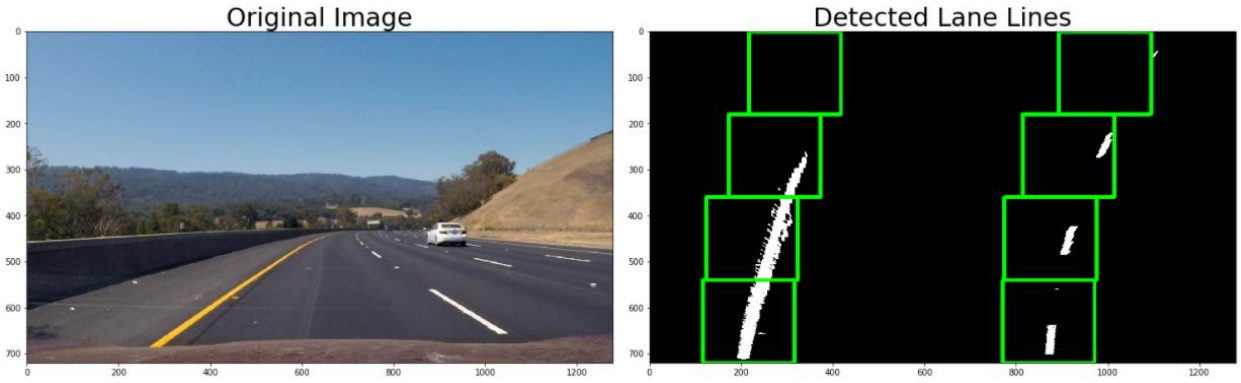


*Figure 6 Window Lane Line Detection*

Based on the algorithm used, either 'fit_poly' or 'fit_polynomial' functions fit a 2nd degree polynomial, described by the below image, that can be used to detect lane coordinates.

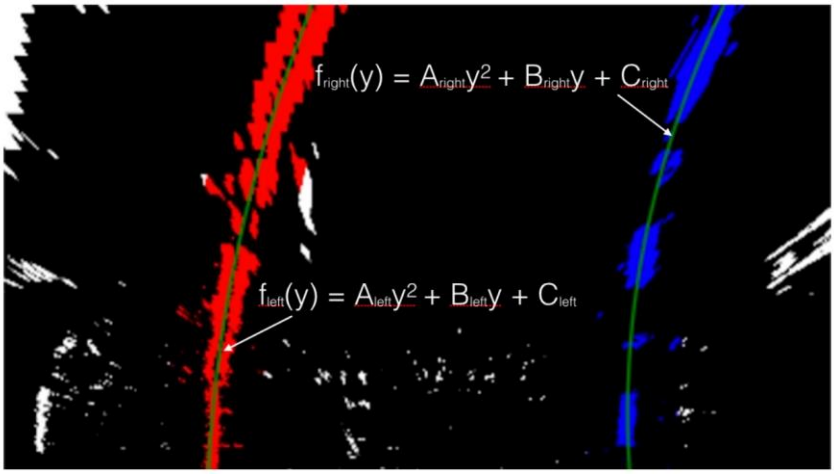


*Figure 7 Lane line fitting polynomial - Lane detection Udacity CarND*

The code pertaining to the above described functions is within the section 2 of code file.

*Step 5: Curvature and vehicle position determination*

The curvature and vehicle position in the lane are computed using 'curvature' function defined within section 2 of code file. This function takes the fit coefficients and 'y' coordinates generated by the previously described fit functions to calculate lane curvature and vehicle position within in the lane.

*Step 6: Displaying lane, lane curvature and vehicle position*

This is executed at the tail end of section 3 of the code file, under the visualization part. The lane is highlighted by using 'cv2.fillPoly' function that takes lane line coordinates previously generated as inputs and draws onto a blank image that is overlapped onto a inverse perspective transformed input raw image, using 'inv_persp_trans' function described in section 2 of code file, using 'cv2.addWeighted' function. The lane curvature and vehicle position are added as text using 'cv2.putText' function.

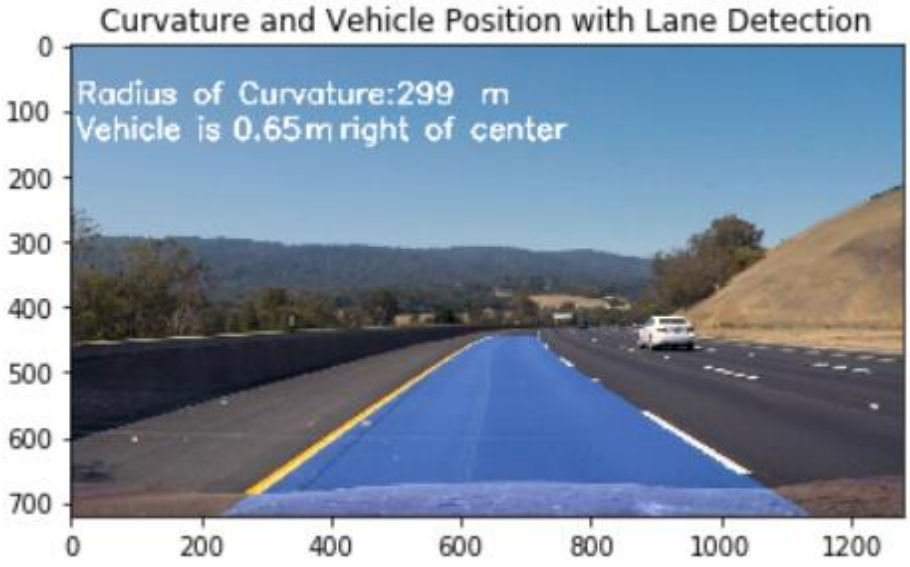


*Figure 8 Processed raw image*

CRITERIA 4 – PROJECT PIPELINE VIDEO:

Here is a link to my result video:

(https://github.com/ram-charan-m/Advanced_Lane_Detection/tree/master/Project_Output_Videos )

All the files named project_video_final*.mp4 are reasonable results with varying tuning parameters.

---

CRITERIA 5 - DISCUSSION:

***Common Issues and Fixes:***

Issue 1:

Missing lane lines – The targeted lane search sometimes does not detect lane lines.

*Fix: Adding a flag to check for this phenomenon in case of which revert to moving window search algorithm. This flag updates lane line class attribute 'detected' that partly controls which algorithm to use.*

Issue 2:

Incorrect curvature values – The curvature values are much larger than allowable value of 1km.

*Fix: Curvature values are a direct result of fit coefficients. Adding a sanity flag to check for poor curvature values and fit coefficients will avoid unreasonable values. The curvature and fit coefficients are updated as lane line class attributes. The sanity flag is updated as a sanity class attribute that partly controls which lane search algorithm to choose.*

Issue 3:

Frame to frame jumping of line positions – Even when the pipeline performs perfectly well on an image it might not translate to a video as depicted below.

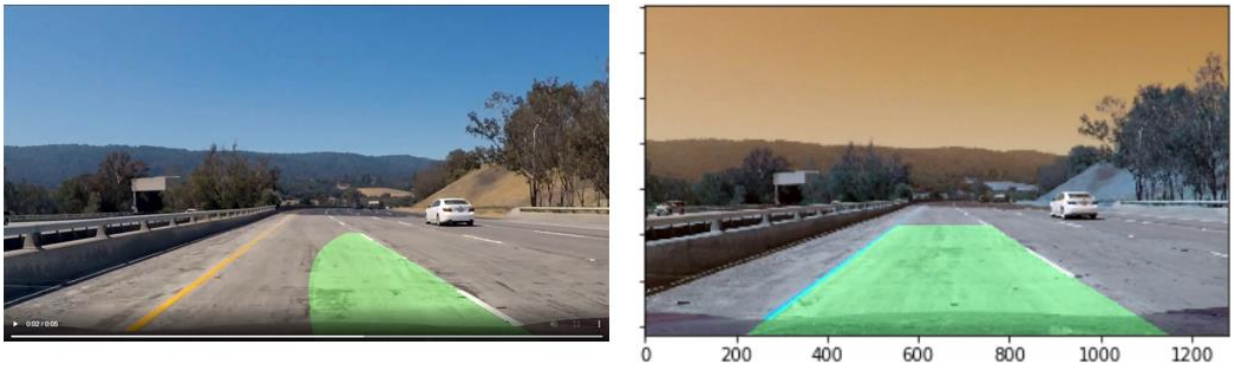


*Figure 9 Same frame processed gives different output on video and as image*

*Fix – Smoothening over the last 'n' frames (called history within the pipeline) will solve this issue. This includes appending 'good' fit coefficients to a lane line class attribute that holds the best fit coefficients and taking its average over 'n' frames as the current fit coefficients.*

***Potential Points of Failure and Possible Solutions***

- Large radius of curvature roads, i.e., hairpins and steep turns
    - This can be solved by modifying the sliding window algorithm to avoid stacking the sliding windows vertically against the side of the image, and likely leading to an imperfect polynomial fit
    - Deep learning techniques
- Adverse weather conditions
    - Using other sensor information through sensor fusion
- Varying road colors that can be detected as lane lines
    - Deep Learning