

Advanced Lane Finding

CRITERIA:

Details as described in the ReadMe File.



MEETS SPECIFICATIONS

The pipeline for this project comprises the following steps:

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images
2. Apply a distortion correction to raw images
3. Use color transforms, gradients, etc., to create a thresholded binary image
4. Apply a perspective transform to rectify binary image ("birds-eye view")
5. Detect lane pixels and fit to find the lane boundary
6. Determine the curvature of the lane and vehicle position with respect to center
7. Warp the detected lane boundaries back onto the original image
8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Each step is discussed in more detail below, including references to various Python classes and functions implemented for each step.

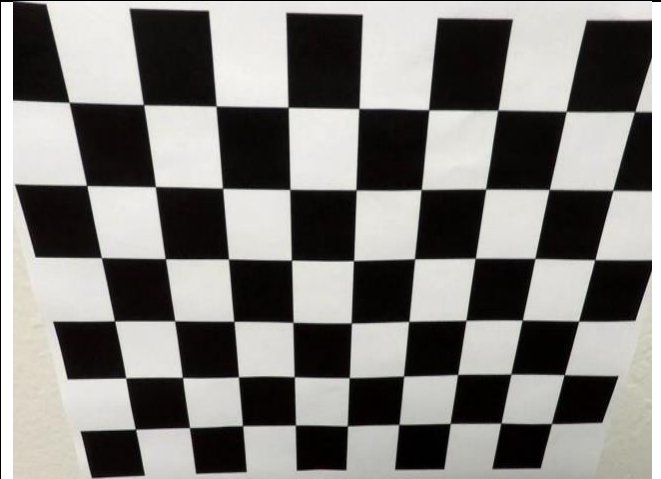
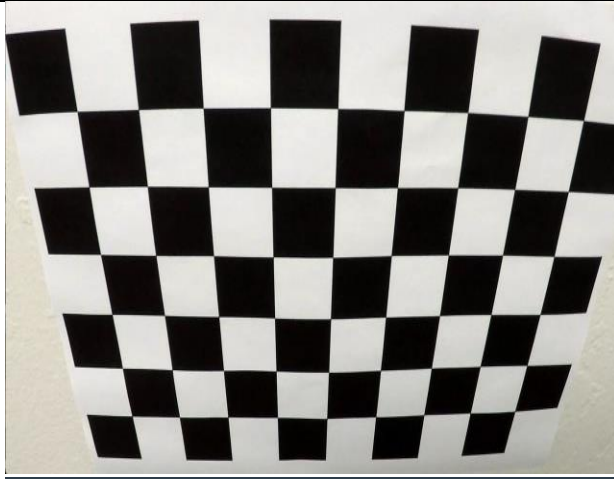
All code for this project is implemented in the following .py files & UML structure is illustrated in the UML image files as below.

Alf_Pipeline.py	Main script of the project that ties all other .py scripts
CalibrateCamera.py	Calibrates Camera
TransformPerspective.py	Calculates perspective transform
FindLane.py	Contains lane detection code
ThresholdBinary.py	Contains all code to create the binary threshold image
Line.py	Class that encapsulates code for a single Lane line (be it left or right lane)
Package Structure UML Diagram	 package-structure.jpg
Class Structure UML Diagram	 class-structure.jpg

To run the project execute "Alf_Pipeline.py". It is setup to produce all images included in the output_images folder and to process the project video project_video.mp4.

Steps 1 & 2 Camera Calibration & Distortion correction

- Checkboard images as provided with the project document were used to calculate calibration matrix and distortion coefficients. Required code is encapsulated in CaribrateCamera class. Function __calibrate performs the actual calibration. Funtion undistort returns undistorted version of the image.
- After calibration completes the calibration matrix and distortion coefficients are saved into storage media. Subsequent runs of the pipeline read the calibration matrix and distortion coefficients from storage media, to expedite performing the calibration again each time.
- Checkboard images provided with this project do not contain the same number of detectable corners. However, this was resolved by manually determining the number of detectable corners for each image. To show calibration works one of the checkerboard images is shown below.
- The left image (as below) is the original image, the right image (as below) is the image obtained after undistorting it using the calibration matrix and distortion coefficients.



Next, one of the test images provided with this project is undistorted. The two images below show the original (left) and undistorted (right) version of test image test5.jpg respectively.



Step 3: Threshold binary image

- For detecting lane lines in an image a binary threshold image is used. Any white image pixels are likely to be part of one of the two lane lines.
- All code required to create the binary threshold image is contained in function `threshold_image` in class `ThresholdBinary`.
- Function `threshold_image`- used to create a binary threshold image and apply it to `self.image` which uses following Functions,

`cv2.cvtColor` - To Convert YUV and HLS color space

`np.stack, np.mean` - To use the U and V (YUV) and S (HLS) channels and convert to grayscale

`abs_sobel` - To Apply Sobel-x and Sobel-y operator

`dir_gradients & abs_gradients` - To apply direction and magnitude of the gradient

`extract_yellow` - To Extract yellow color of the left lane.

`extract_highlights` - To extract pixels that may be covered by shadow.

Mask created to combine everything into one mask

`region_of_interest` - To Ignore anything outside the region where lines are expected to be (trapezoidal shape)

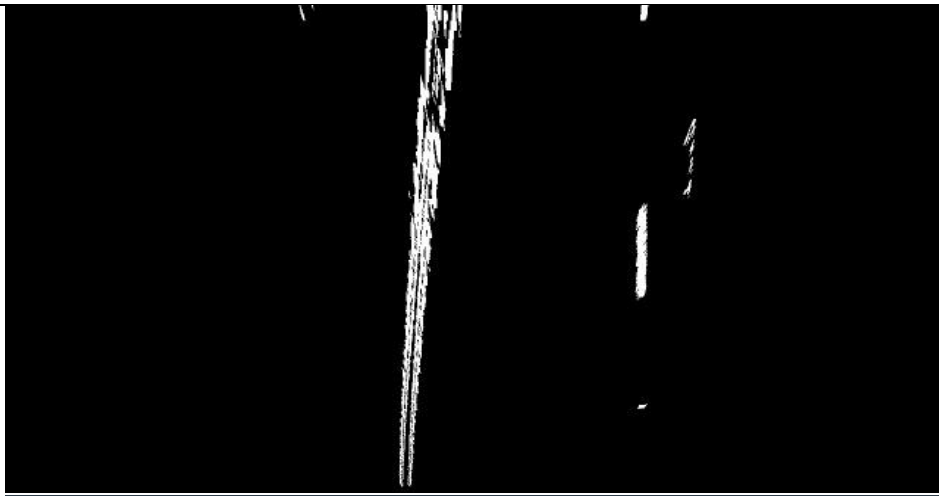
- The various thresholds were determined using trial-and-error in combination, the test images provided with this project and the diagnostics view (refer below). Once a proper binary threshold image is found, any detected pixels that are not inside a trapezoidal region of interest directly in front of the car are excluded to avoid false positives.

The binary threshold image for test image `test5.jpg` is shown below.



Step 4: Perspective transform

- Transform is applied to the binary threshold image in order to view the lane from above a bird's-eye view
- All code required to create the bird's-eye view transform is contained in the `__init__` function of `TransformPerspective` class.
- `cv2.getPerspectiveTransform` (of Open CV) function used (with correct source and destination points required were determined using the `straight_lines1.jpg` image as provided)
- The points were manually picked in such a way that the two lane lines appear as two straight and parallel lines in the bird's eye view image.
- The perspective transform is only calculated once rather than for each image/video frame because for this project the assumption that the road is always flat is valid.
- The image below shows the bird's-eye view of the binary threshold image shown under above section.



Step 5: Detect lane lines

Lane detection code is contained in the `FindLane` class. The lane lines are detected in two steps:

`function __detect_lane:`

- Histogram is applied to the bottom half of the bird's-eye view of the binary threshold image.
- The Image is split into in half the points in both halves with the highest number of pixels are assumed to be the starting points of the left and right lane line respectively

`function __sliding_window:`

- A sliding window approach is used to detect peaks in small areas around each starting point falls within the two detected starting points for both lane lines.
- To detect the peaks, `scipy.signal's find_peaks_cwt` function is used.
- After using the sliding window to detect lane line points starting at the bottom until the top of the image, second degree polynomials are fitted to both
- lane lines using `numpy's np.polyfit` function.
- The two polynomials approximate the left and right lane lines respectively.

To avoid sudden jumps, any lane lines that are clearly not valid ones are rejected. Also, the lane lines detected for the current frame are smoothed using previous' frames lane lines.

Step 6: Determine curvature and vehicle position

- The curvature of a lane line is calculated in the `calc_curvature` function in the `Line` class.
- It uses the polynomial that approximates the lane line and converts detected lane pixels to meters first before calculating the curvature.
- The position of the car is calculated by subtracting the center of the image/video frame, which is assumed to be equal to the center of the car, from the lane's center point.
- The lane's center point is calculated by averaging the left and right lane's fitted polynomial. Calculating the position of the car is done in `FindLane's __plot_lane` function

Step 7 and 8: Plot detected lane and additional information

- Function `__plot_lane` in the `FindLane` class is used to warp the detected lane lines back onto the original image and plot the detected lane using a filled polygon. Additionally it plots information in the top left corner and at bottom of the image/video frame.
- Curvature and position of the car relative to the lane center is plotted in the top left corner
- The position of the car is indicated at the bottom with a long vertical white line. The detected lane center is indicated using a short white line. In the project video the car stays relatively close to the center of the lane, though it is mostly positioned somewhat to the left of the lane's center.
- The image below shows the result for test image `test5.jpg`:



Diagnostic View:

To simplify pipeline finetuning, a diagnostic view showing a number of images taken after various pipeline steps in a single image was used. This view is implemented in function `plot_diagnostics` in class `FindLane`.

An example is shown in the image below. It shows the detected lane and additional information in the top part of the image. Along the bottom it shows the original image, undistorted image, binary threshold image and bird's-eye view of the binary threshold image respectively.



(Results of 6 test images' process shown below)

Results for all six test images

The image below shows the detected lanes for all six test images in a single image. The pipeline is implemented in `FindLane` Class's `process_image` function.



Project Video

FindLane's `process_video_frame` function is used to implement this action.. That function is identical to `process_image`, except that it includes additional code to create and save a number of images used throughout the project outcome. (Refer attached video file : `project_video_output.MP4` (as part of this zip file)

Challenges: .

This was a challenging project. The 3 assumptions for this pipeline to work are that the lanes are,

- Having similar curvature
- Separated by approximately the right distance horizontally
- Roughly parallel

Computer vision solutions are sensitive to the chosen parameters, and if the parameters are not chosen correctly, they fail. I couldn't get the approach to work on the harder-challenge video, mainly because the lanes had large curvature, and as a result the lanes went outside the region of interest we chose for perspective transform and also, current parameters (different algorithm thresholds) not able to extract lane pixels for the challenge video correctly.

This project involved attempts of trial and error of parameters applied for right thresholds to enable the pipeline to work. Applied technique may work on yellow and white lanes, and may not probably do better in situations where lanes are different color. Improvements could be lane finding task for night situations or under adverse weather conditions like rain, snow etc. Also, just to add In India, it is not uncommon to see roads without any road lanes at all.

For complex road conditions such as Spiros (clothoid road models) perhaps a quadratic or cubic polynomial approximation might work.

It is better to apply perspective transform algorithms to programmatically detect four source points in an image based on edge or corner detection and analyzing attributes like color and surrounding pixels to more generalize the solution.

This project kindled my thought to appreciate Deep Learning approaches even more as Deep learning approaches avoid the need for fine-tuning these parameters, and are inherently more robust. I consider this approach is not robust for real world situations. Probably techniques such as Neural Network based approach might be more helpful.

I might need more efforts and time to do the improvements and I will try to do that.

.....