# Advanced Lane Finding Project

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.

Much of the code was borrowed from the Advance Lane Lines lesson so much of the code looks similar to what is found in the lesson. Much of the difference is in how I used the Line() Class in keeping a history of lane characteristics and then filtering the results of each lane in order to achieve a smooth lane identification for the project.mp4 video. The output of this code was able to achieve normal.mp4 which shows does the lane overlay, radius of curvature identification, and lane center position on the video. This pipeline was not able to process the two challenge videos as tuning of the binary image lane identification could not result in proper identification of the lanes

### Camera Calibration

**Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.**

The code for this step is contained in the 2nd and 3rd code cells of the IPython notebook located in CarND-Advanced\_Lane\_Lines.ipynb

In order to undistort the images, I created a function ‘undistort’ that takes in an image, a camera matrix, and distortion matrix and uses the OpenCV function ‘undistort’ to return an undistorted image. In order to get the camera and distortion matrices, I used OpenCV’s calibrateCamera function to compute them. Here is the result of undistorting a calibration image

A black and silver text on a tile floor

Description generated with high confidence

The calibrateCamera function takes in object points and image points. Object points are defined as a fixed grid whereas the image points are picked up from several calibration images. In this case there were 20 images used to pick up image points to do the calibration

## Pipeline (single images)

**Provide an example of a distortion-corrected image.**

A sign on the side of a road

Description generated with very high confidence

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like the pair below:

I created a rough pipeline in Cell 14 of the jupyter notebook in order to test the functions that I created to process the images. I basically used the calibration result from the calibration images and applied that result to one of the straight lane images to understand how well the functions are working

**Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.**

I defined several functions to perform gradients, gradient magnitude and direction, and color transformations. These functions are defined in cells 6 through 9 of the notebook. An example of the x-direction gradient binary image is shown on the left below. Ultimately, for the project video I settled on using just the result of an x-direction gradient and an s-layer from an hls color transform to get the combined image shown on the right below. I started trying the use other combinations to try the challenge video as well, but at the time of this write-up have not yet succeeded in finding a good combination.

A close up of a logo

Description generated with high confidence

**Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.**

I created a function called unwarp to do a perspective transform in the 10th cell of the notebook. In this function I transformed a trapezoidal area to a rectangle by starting from the centerline of the image and then defining the boundaries symmetrically at the bottom of the image up to a point close to where the lanes vanish into the horizon. I used the straight\_lines images to adjust the top part of the lane until I got an image that had parallel lines. The top of the lane marker was also adjusted to get as much of the lane as possible to do the line fit. Finally, I also ‘squished’ the image in the x direction so that I could account for the bend in the lines. An example of the perspective transform is shown below. Note that the bottom of the lane lines in the warped image (300,720) and (1000,720) don’t match the binary image.

The image on the left is the lane transformed to a rectangle, while the image on the right is the source image. Although hard coding values for the transformation was convenient, it is not well suited for videos such as the more challenging video in which the lane bends aggressively, and the lane does not extend into the image as far as the project video

A close up of a logo

Description generated with high confidence

**Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?**

There are two functions that I created in the code to identify lane-line pixels are in cells 11 and 13. Cell 11 is a function called poly\_lanes\_search and cell 13 contains poly\_lanes. Cell 11 was used for the rough pipeline (cell 14) to check individual test images while the pipeline for checking test images and for creating the video is in cell 16 and uses the poly\_lanes function.

I’ll now describe the poly\_lanes function since it contains the overall approach for achieving lane identification in the video output.

This function takes in a binary image of the lane that has been warped using the perspective transformation from the description above. It then make a judgment of using a sliding windows search to identifying lanes or using a window around previously detected lanes based on how many averaged windows (20) have been found. If less than 20, use sliding windows, otherwise use the lane results.

The result is identifying the x and y position of each of the left and right lanes. These positions are then used in two second order polynomial fits (one for the pixels and one for the pixels converted to meters) to identify the lane positions. The fit using the pixels is used for lane identification in subsequent video frames while the one converted to meters is used for calculating the actual radius of curvature.

Then finally, if the lane characteristics criteria had been met (average distance between lanes and curvature over a minimum threshold) the lane fit coefficients are passed to the leftLine and rightLine instances of the Line() Class.

The trick here was to use the averaged coefficients of successful line detections to do the search for the next video frame. The following code snippet shows how this was done:

#get the average coefficients

left\_fit\_avg = leftLine.best\_fit.tolist()

right\_fit\_avg = rightLine.best\_fit.tolist()

#use the avg coefficients to do the search

left\_lane\_inds = ((nonzerox > (left\_fit\_avg[0]\*(nonzeroy\*\*2) + left\_fit\_avg[1]\*nonzeroy +

left\_fit\_avg[2] - margin)) & (nonzerox < (left\_fit\_avg[0]\*(nonzeroy\*\*2) +

left\_fit\_avg[1]\*nonzeroy + left\_fit\_avg[2] + margin)))

right\_lane\_inds = ((nonzerox > (right\_fit\_avg[0]\*(nonzeroy\*\*2) + right\_fit\_avg[1]\*nonzeroy +

right\_fit\_avg[2] - margin)) & (nonzerox < (right\_fit\_avg[0]\*(nonzeroy\*\*2) +

right\_fit\_avg[1]\*nonzeroy + right\_fit\_avg[2] + margin)))

#### 5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in lines # through # in my code in `my\_other\_file.py`

#### 6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

I implemented this step in lines # through # in my code in `yet\_another\_file.py` in the function `map\_lane()`. Here is an example of my result on a test image:

![alt text][image6]

---

### Pipeline (video)

#### 1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a [link to my video result](./project\_video.mp4)

---

### Discussion

#### 1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.