

# Advanced Lane Finding Project

Jakrin Juangbhanich

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.

## Introduction

This project was created using PyCharm IDE, with all the code logic contained in python files in the script folder. Inspired by the previous project, and how the DNN architecture worked, I first decided to create a pipeline class, and a layer class for this project. This is so I can see what each step of the pipeline does easily, and so I can adjust the logic in each layer independently.

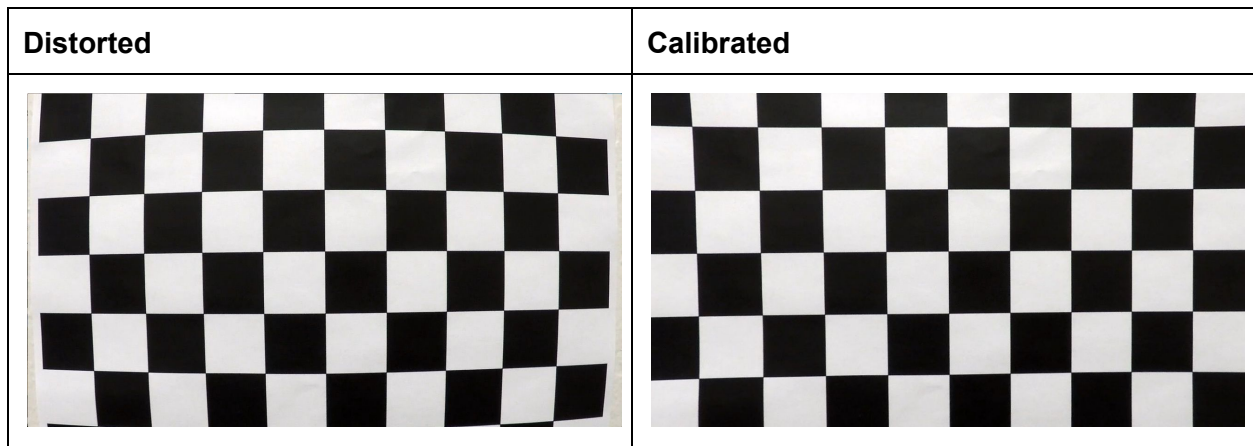
The pipeline contains a definition of layers (the architecture). It takes in an image, and each layer will process that image. Each layer returns the image for the next layer as input. Each layer also has the option of saving its output image into a folder, so I can have full visibility on what is being processed each layer.

There is a 'model' class also being passed down the layers, which contains persistent data about the road so far, such as the confidence in its lane predictions, and the previous lane polyfits.

## Rubric Points

### Camera Calibration

The camera is calibrated at the start of the program. The calibration script also outputs samples of the undistorted chessboards. The code for this is in `calibrate.py` using `cv2's` **`calibrateCamera`** and **`undistort`** methods.

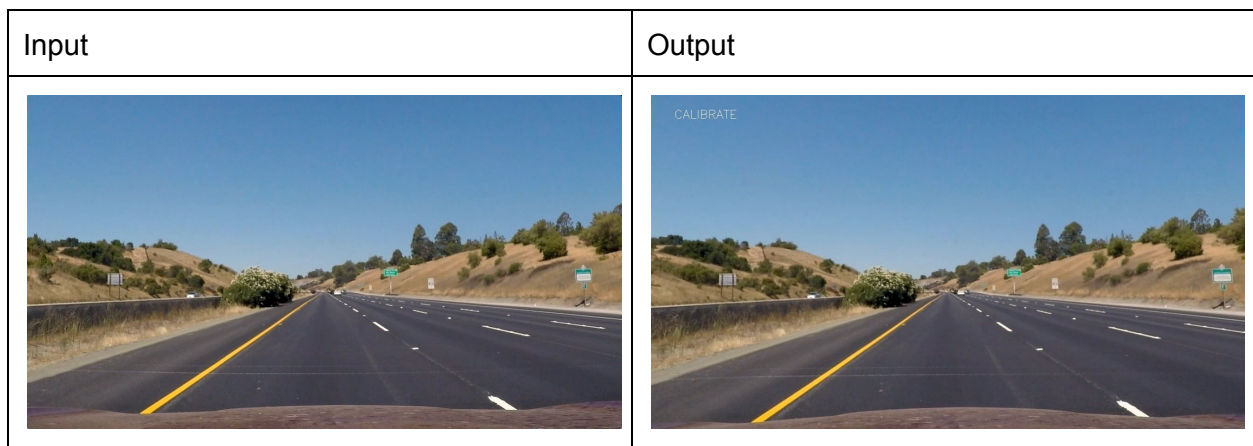


## Image Processing Pipeline

Here is a description of the full pipeline used for both images and videos, as well as sample outputs from each.

### Calibration Layer: `calibrate.py`

The first layer of the pipeline will simply undistort the image using the calibration matrix computed prior.

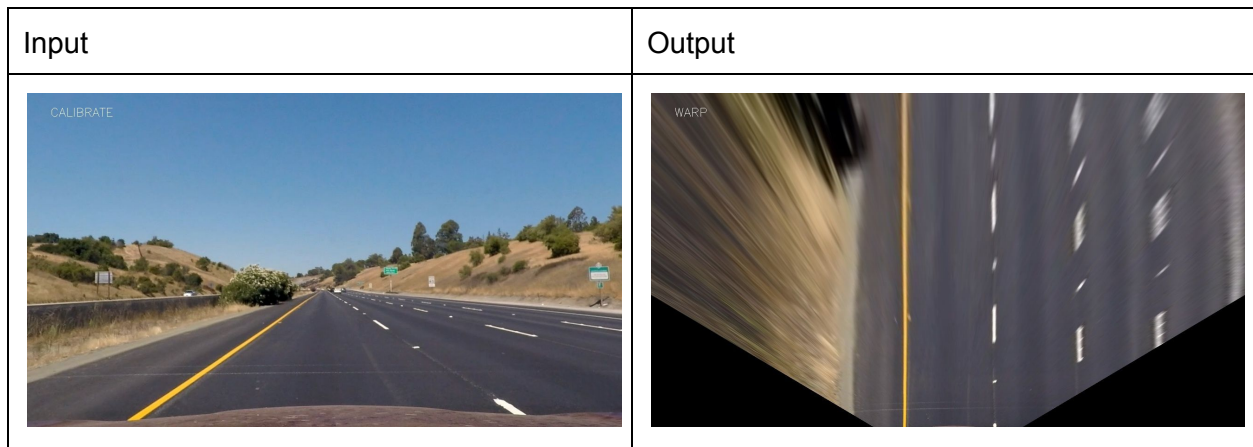


### Warp Layer: `warp.py`

The second layer warps the image to get a bird's eye perspective. I picked 4 points based on one of the images with a straight road, which gave me the factors from x and y to use as source points for the warp. I decided to warp before doing color thresholding because I think it is easier to threshold an image where the lines are already somewhat more obvious.

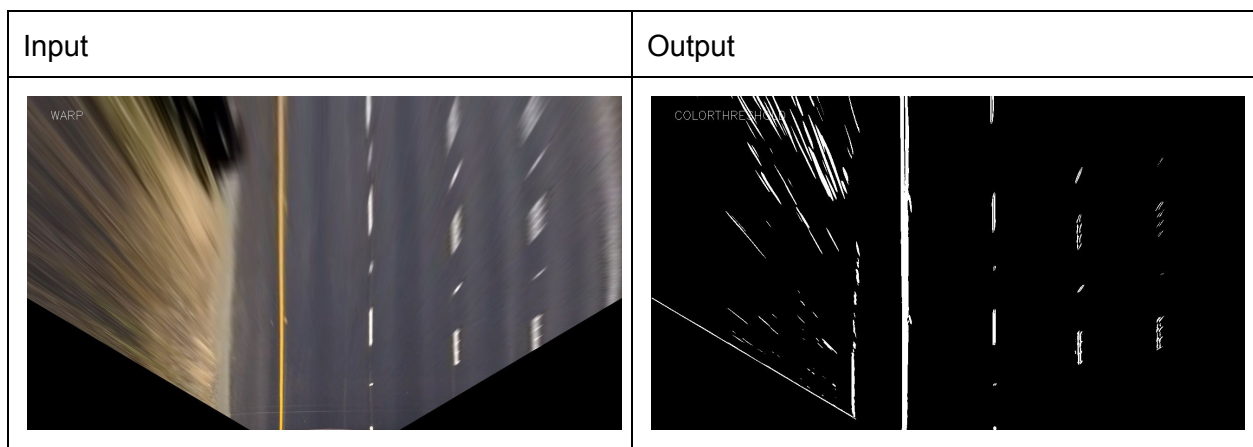
I also wanted a wider view of the road, so by shrinking my destination region to 0.25x, I'm

actually able to 'zoom out' more. I thought this will be useful if I wanted to be concerned with other lanes in the future.



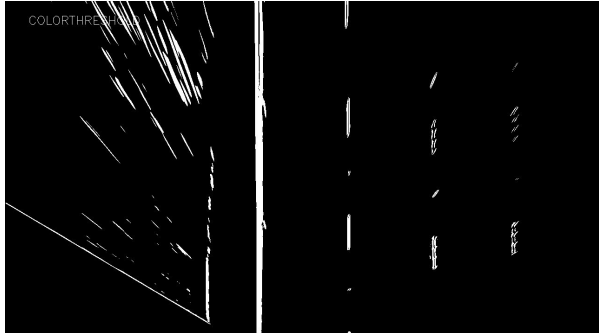
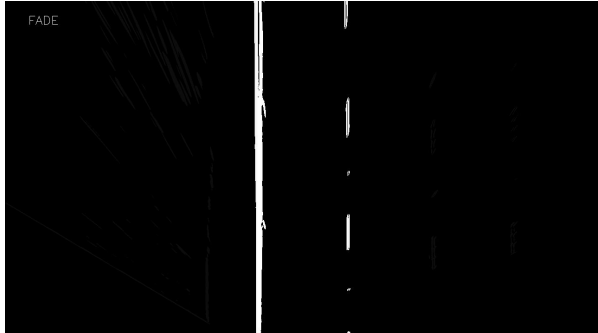
### Threshold Layer: `colorthreshold.py`

This layer applies all of the thresholding techniques I learnt to try and identify the lane pixels. There were a lot of ideas I wanted to play around with, for example, finding pixels that were brighter than average, or had a high R to B ratio (indicating a yellow lane). But in the end the most effective thing for me was to mix a saturation/lightness filter with a sobel-x filter. It gives me mostly what I want, but will still struggle with noise and with shadows. To the sobel-x filter, I also made it only work on pixels that were bright enough, to eliminate the noise such as dark or black lines in the road.



### Fade Layer: `fade.py`

I assumed that lines near the center were probably the most important, so I applied a fade layer to soften the values of the lines further away. So my image isn't actually a true binary image, since I actually use the strength of each pixel when adding to my histogram.

Input	Output
	

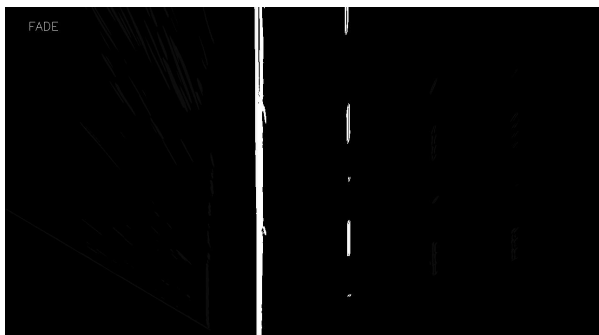
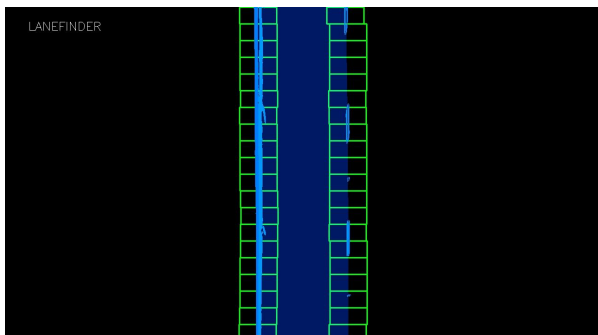
### Lane Finding Layer: lanefinder.py

This is the most complex layer and probably where 80% of the work of this algorithm is done. It first uses a histogram to try and predict the starting points of each lane, if no lane history exists. A 2nd order polynomial is then fit into the points it detects (using sliding windows).

This data is then recorded to the model. On the next frame, however, things happen differently. If the model has two lane fits already found, and it is confident in them, it will apply a region of interest search instead of the histogram approach.

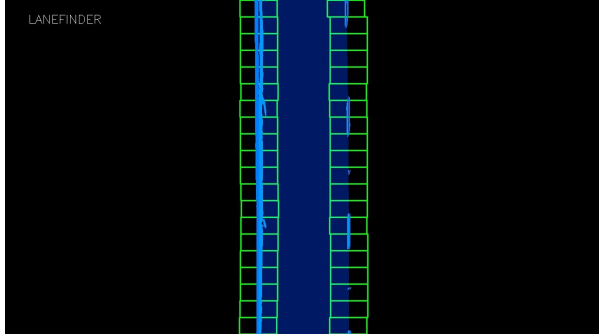
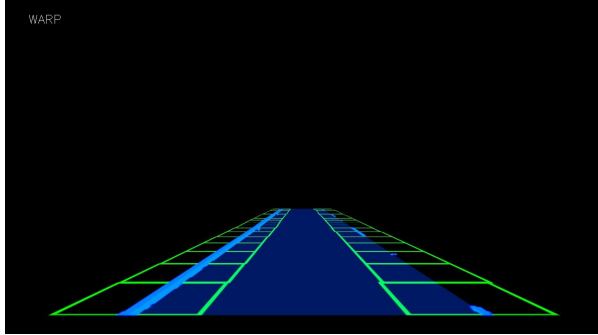
The confidence of the prediction increases if the polynomial has little average error, and if there are many points in the region. A fit will be marked as an outlier if the squared sum of the coefficient deltas are larger than a certain value. This fit will be ignored, but the lane's confidence will also be reduced. An outlier will only be detected if the lane's confidence is high. If it is too low, it will consider fitting to more points.

If the confidence is too low, the first technique it tries is to borrow data from the opposite lane (if that lane's confidence is high), since we know that lanes are roughly 3.7m apart and should be parallel. However, if this fails and the confidence still drops, the system reverts back to histogram and a sliding window search.

Input	Output
	

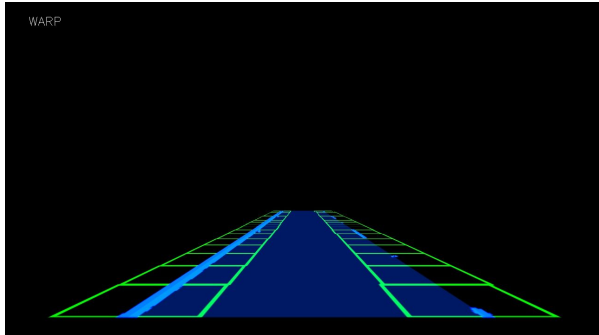
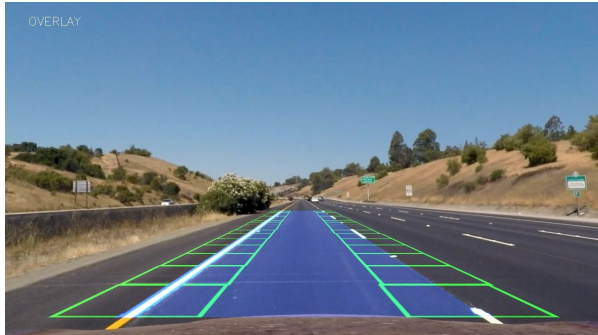
### Unwarp: warp.py

The next layer un-warps the image back into the normal perspective.

Input	Output
	

### Overlay: overlay.py

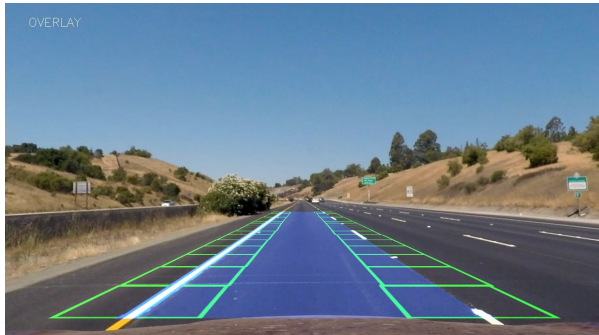
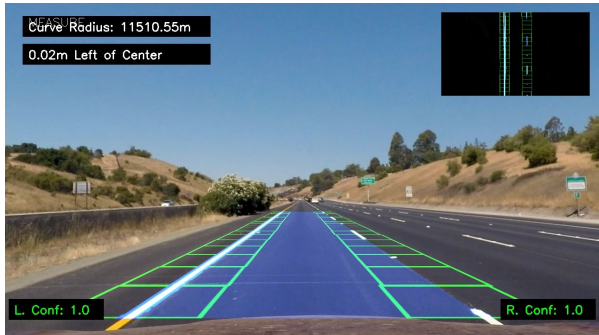
The next layer simply overlays the image back onto the original picture.

Input	Output
	

### Measurement: measure.py

The final layer is the one that measures the radius of the curve and the vehicle position. This is done using the poly-fit coefficients we generated earlier, combined with some approximated pixel : meter size for this camera. The measurements are then drawn onto the image:

- Curve radius of the road (m)
- Vehicle distance from the center of the lane (m)
- Left and Right lane confidence values.

Input	Output
	

## Video Processing Pipeline

The pipeline simply takes an image as input, and returns an image output. So it also works with a video, by feeding in a stream of images. The pipeline will continue to save each step in its respective output folder as an image, but in the interest of saving space, I made a parameter which determines how often it should do this (e.g. only save an image every 15 frames).

Here is a link to my output videos:

[https://github.com/Infrarift/AdvancedLaneLines/blob/master/output\\_videos/project\\_video.mp4](https://github.com/Infrarift/AdvancedLaneLines/blob/master/output_videos/project_video.mp4)

## Discussion

Going into this project, my strategy was to develop a system that would give me maximum visibility and control on every individual step of the pipeline. Since each step relies on the input of the previous, every single layer matters. This is why I developed the pipeline and layer structure, and I think it helped me a lot - without it this would have been probably impossible for me.

The first problem I faced was the thresholding. It was hard to find a set of filters that would pick up lane lines, but ignore dark marks on the road (like in the challenge video). I used a combination of region of interest masking, sobel filters, and color filters to get the result I wanted. But thresholding anything by absolute values is not robust, because the light conditions can change. This is why lanes in the shadows give my algorithm so much trouble.

A human can account for relative lighting of the surroundings, and can infer that a lane line is meant to be yellow, when to a computer it might actually be closer to blue or grey. So one step for improvement is to develop some kind of dynamic thresholding layer, that instead of using fixed values for its thresholds, will learn to look for values that a human would immediately notice. For example if the lightness is higher than the average of the road, or if the color ratio



(red:blue) is different.

The lane finder was probably where there could be the most improvement. My pipeline completely breaks down on the harder challenge video, which has extreme lighting conditions and road curvature. My intuition is that the memory system of the pipeline needs to be improved. In some instances, a human will know what lane he is in only because of the context. For example, if I'm in a city in traffic, the car in front of me will visually cover the lines completely. So no matter how good my thresholding is, my memory and context awareness is also critical.

I've made a naive foray into this by having a confidence score on each lane, and doing a ROI search. But to really improve this system, I'd like to be running my histogram search in parallel, and then consider both outputs to see what makes the most sense.

On that note, there is something to be said about running the operations in multiple passes in one frame. For example, if I find 4 peaks in my histogram search, rather than just naively picking the top peak on each side, I could find the pair that makes the most sense as a pair (i.e. they are a reasonable distance apart, and close to my previous lanes).

Also the knowledge that two lanes should be parallel and roughly X meters apart can surely be used somehow to help with predictions. I tried a bit of this in my lane finding layer, but it didn't work too well yet.

Finally, for the parts of my code that detects outliers or improves/reduces the confidence of a lane, right now it uses binary thresholds that I picked arbitrarily. I just wanted to see if the concept worked. But to be more robust, these values should probably be a gradient of some sort. So for example, the confidence reduction in a lane is proportional to how far from those limits it is.

I've also tried running this on my own video, but because I didn't calibrate the warp or distortion values properly for my camera, it was just a total disaster:

