Advanced Lane Detection Project – Sughosh Rao

GOALS

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.

RUBRIC POINTS

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

WRITEUP / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

CAMERA CALIBRATION

1. 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 lines 17 through 48 of the file called `lane_detect_final.py`.

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

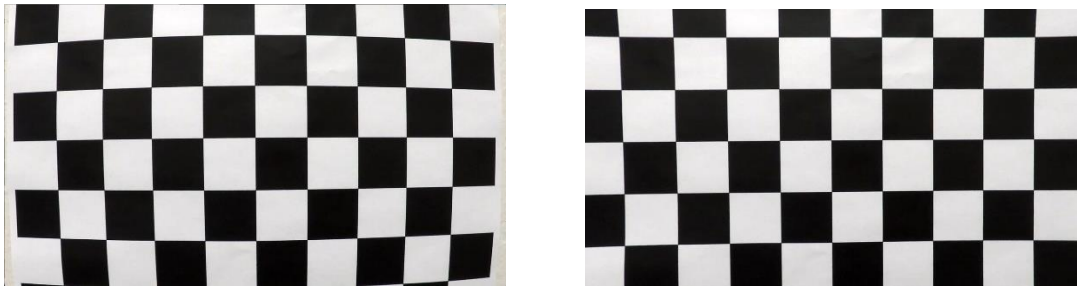


Figure 1: Original Test Image and Undistorted Image

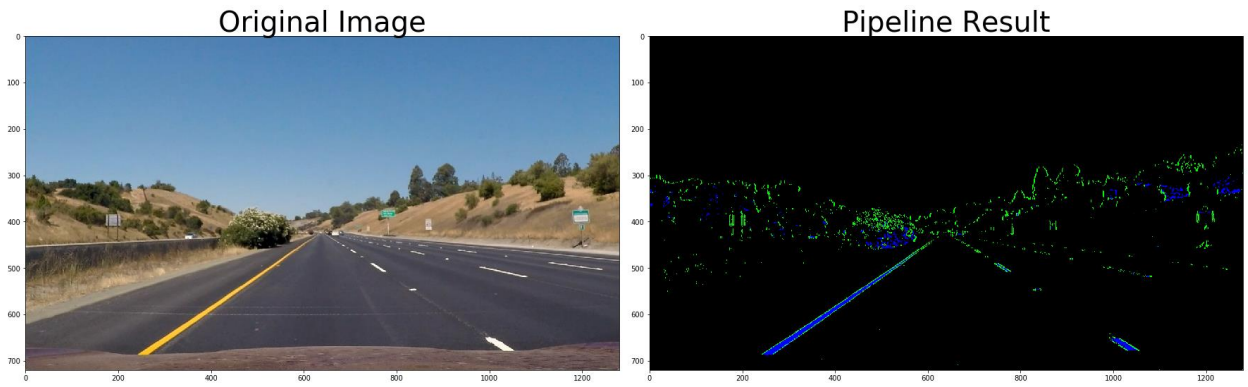
PIPELINE (SINGLE IMAGES)

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

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images. The results are shown above in Figure 1.

2. 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 used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines 210 through 220 in `lane_detect_final.py`). Here's an example of my output for this step. (note: this is not actually from one of the test images)



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

The code for my perspective transform appears in lines 50 through 60 in the file `lane_detect_final.py` . Since the perspective transform only needs to be calculated once, a separate function is not created. I used as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points to calculate the transform matrix. I chose to hardcode the source and destination points in the following manner:

```
src = np.float32([[562,470],[720,470],[1105,685],[279,685]])

dst = np.float32([[250,0], [img_size[0]-250, 0],
                  [img_size[0]-250, img_size[1]], [250, img_size[1]]])
```

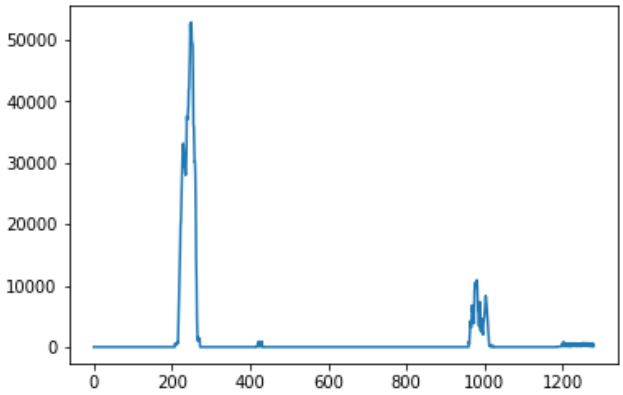
This resulted in the following source and destination points:

Source	Destination
562, 470	250, 0
720, 470	470, 0
1105, 685	470, 1280
279, 685	250, 1280

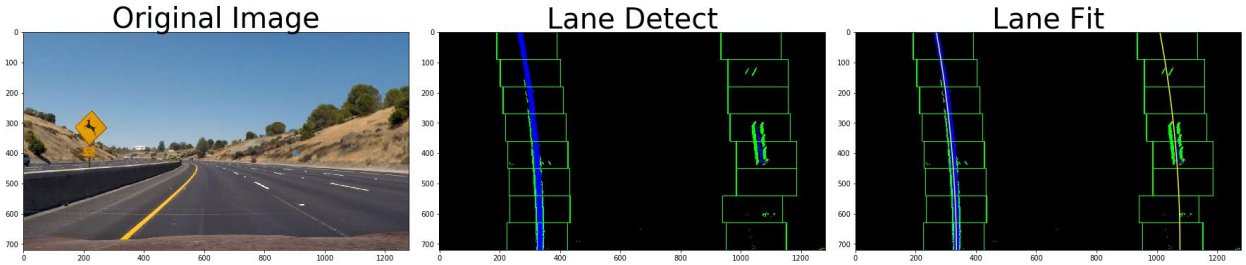
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

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

I detect the lane line pixels in the function ***find_lane_pixels*** that appears in lines 70 through 168 in the file *lane_detect_final.py*. I fit the polynomial to the lane lines in I used the histogram method to find the starting points of the lane lines to first find the beginning of the lane lines when no pervious lane lines have been detected. The peaks of the histogram show where the lane lines are. An example histogram generated is shown below:



Once the starting points of the lane lines were found, I used the sliding window method to find the first set of lane lines. I fit a second order polynomial to the lane line points that were detected. An example result is shown below:



Once the lane lines were found, I searched around the previous lane lines to find the new lane line.

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 using the function ***curvature*** in lines 170 through 179 in my code in `lane_detect_final.py. I used the same equations described in the project lectures to calculate the radius of curvature.`

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:

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!).

The video is saved in the repository as ***project_video_out.mp4***.

Here's a link to my video result

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.

My code initially had trouble tracking the lane lines on the 2 bridges in the ***project_video.mp4*** file. This is due to the fact that the road surface is a lighter color thus reducing the contrast with the lane lines. Moreover, there are shadows present on the bridges that cause further issues. To combat these issues, I employed a couple of techniques.

1. I used a smoothing technique to weight the current prediction with previous lane line coefficients.
2. I performed sanity checks to make sure the lanes detected were indeed nearly parallel.

The two techniques are described below:

Smoothing: I performed smoothing by performing a moving average with a forgetting factor. This method weights the newer measurements more than the older ones and the older ones drop out since their weights converge towards zero pretty quickly. I used a forgetting factor of 0.7. over the current fit with the previous fit. This is described in lines : 260 through 263 in the ***lane_detect_final.py*** file.

Sanity check: I checked if the two lines were parallel by checking if the value of 3rd term in the poly fit output, which is the constant term, made sense. If the two lines are parallel, the difference between the 3rd terms should be approximately equal to the lane width in pixels. I checked this using if statements in lines 252 through 269 in the ***lane_detect_final.py*** file.