

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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.

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.

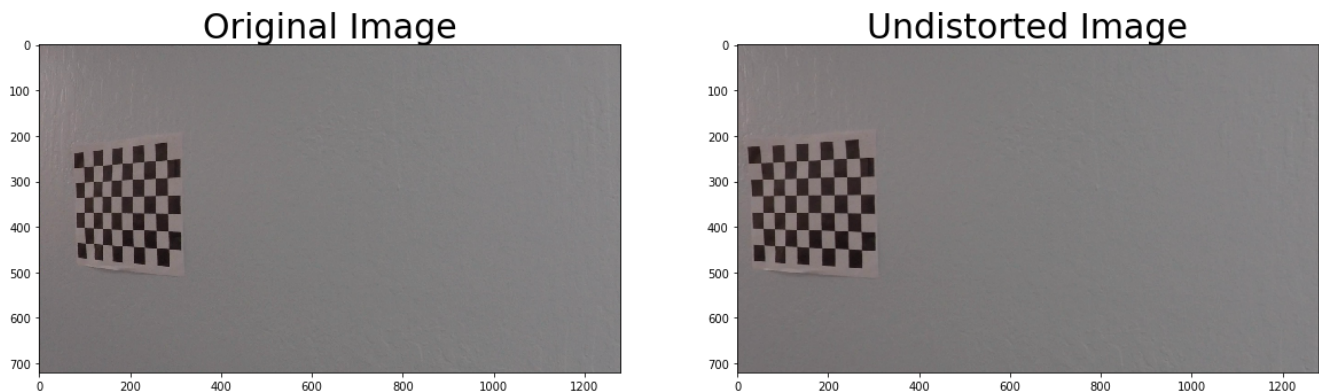
Camera Calibration

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 in lines 22 through 60 of the file called `calibrate.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:



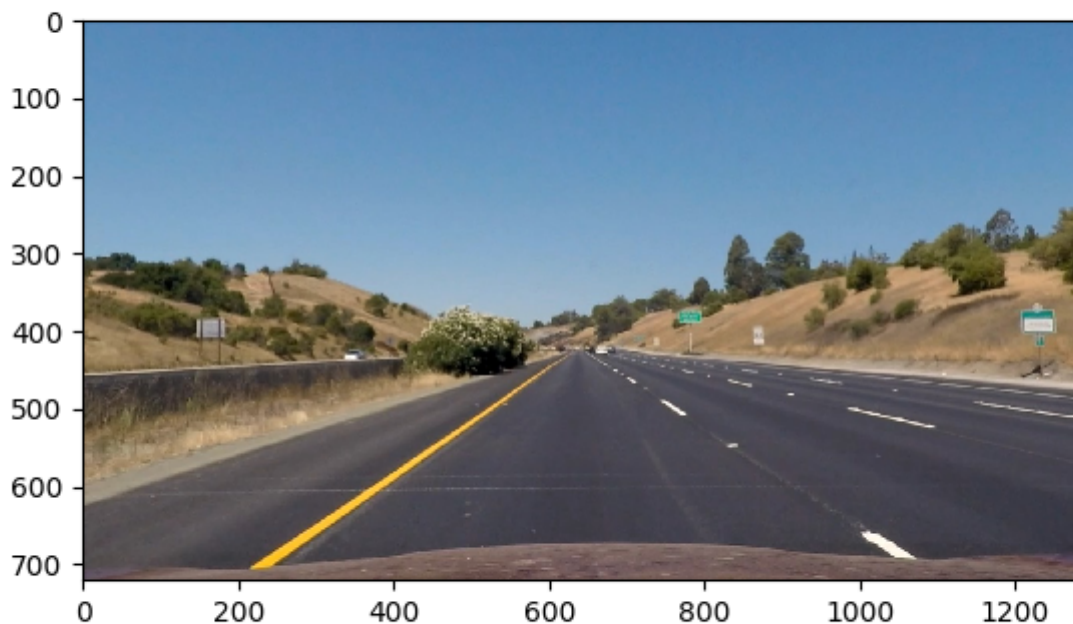
Here is a sample working on the road images.



Pipeline (single images)

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

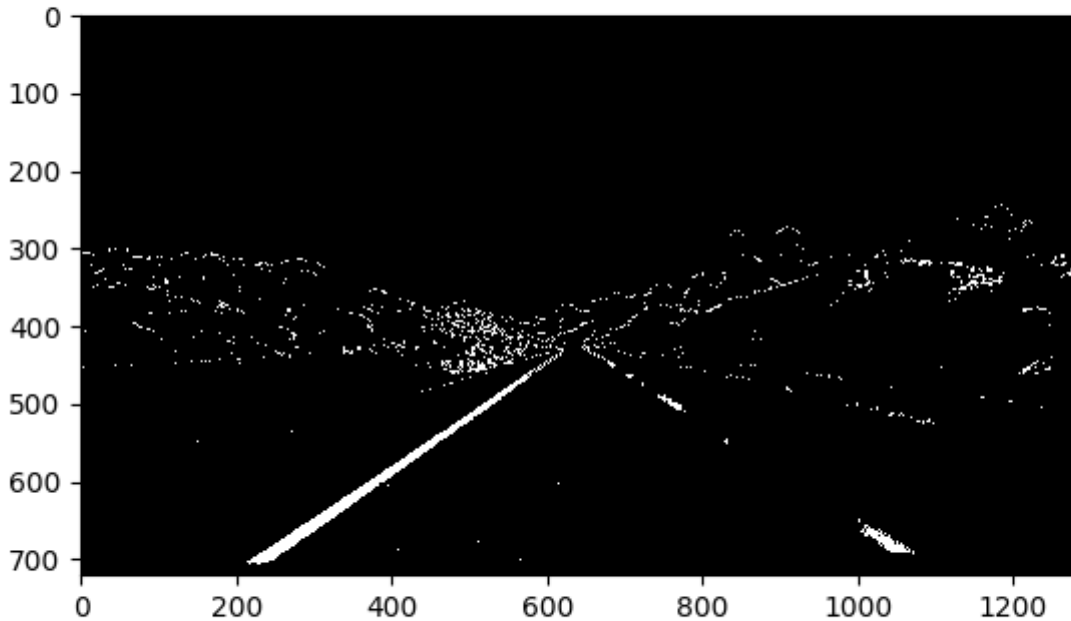
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:



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 and direction thresholds to generate a binary image (thresholding steps at lines 102 through 111 in `helper.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 includes a function called `warp_transform()` , which appears in lines 36 through 60 in the file `helper.py` . The `warp_transform()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. It also accepts a default parameter named `reverse` which when true performs a reverse transformation. I chose to hardcode the source and destination points in the following manner:

```
src = np.float32([
    [(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
    [((img_size[0] / 6) - 10), img_size[1]],
```

```

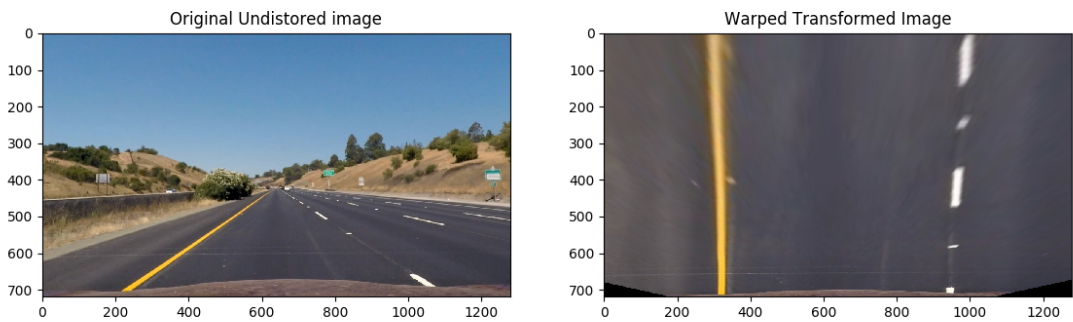
        [(img_size[0] * 5 / 6) + 60, img_size[1]],
        [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
    [(img_size[0] / 4), 0],
    [(img_size[0] / 4), img_size[1]],
    [(img_size[0] * 3 / 4), img_size[1]],
    [(img_size[0] * 3 / 4), 0]])

```

This resulted in the following source and destination points:

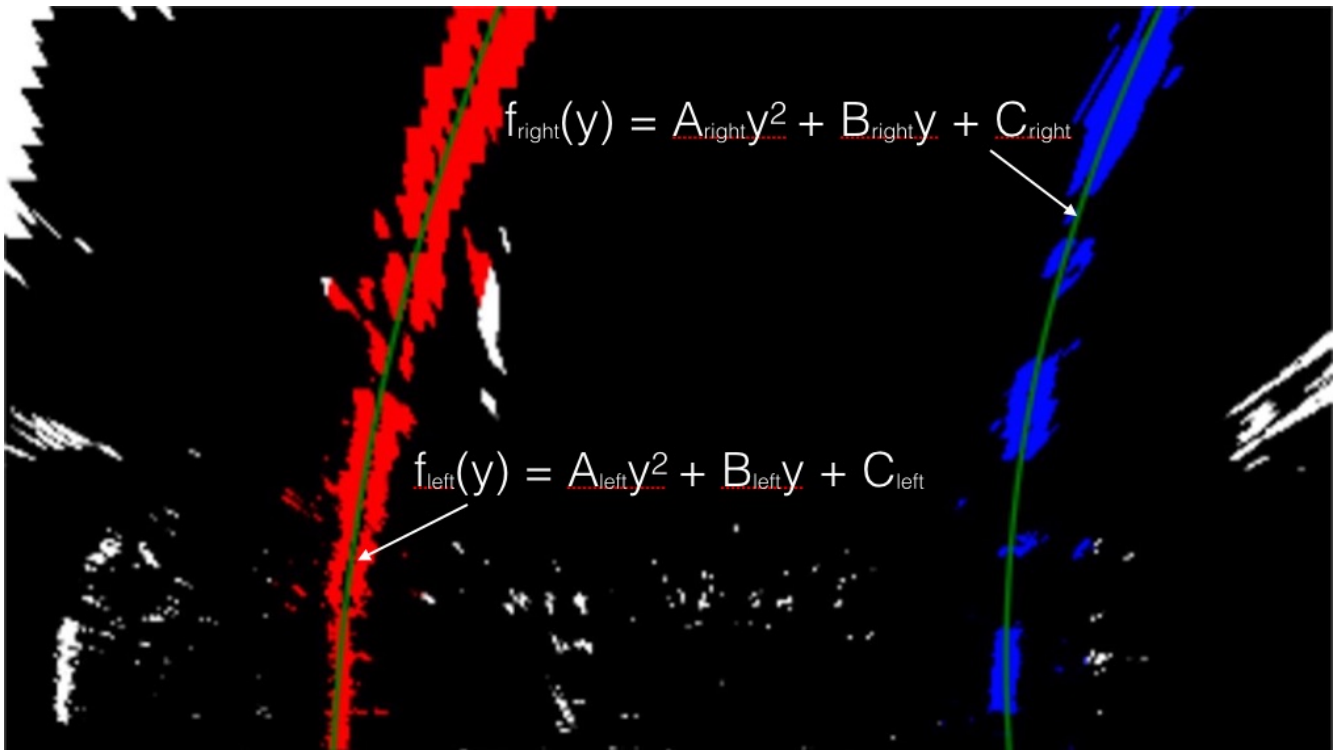
Source	Destination
585, 460	320, 0
203, 720	320, 720
1127, 720	960, 720
695, 460	960, 0

I verified that my perspective transform was working.



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

Lane line pixels are identified in line 42 to line 160 in `lane.py` .

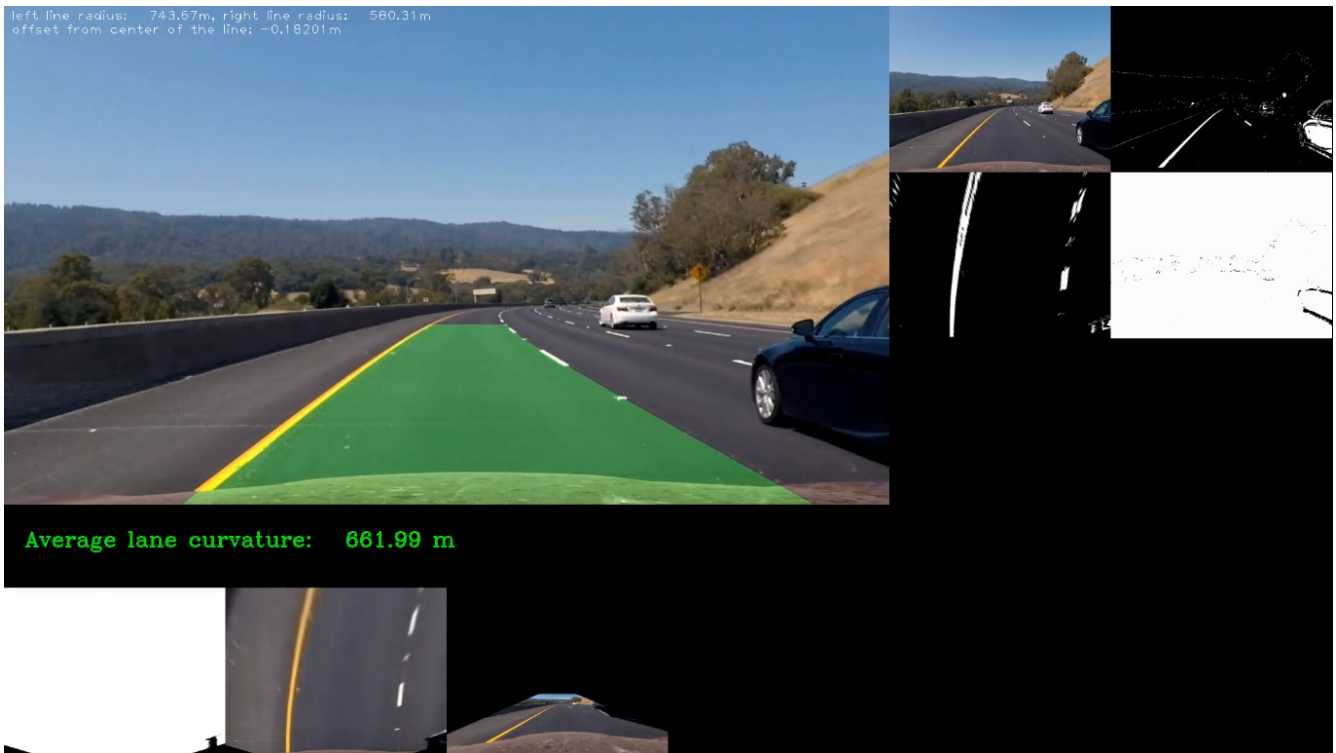


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in lines 22 through 40 in my code in `lane.py` in a function named `curvature.py`. We used second order polynomials to fit the curvy lines.

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 12 through 44 in my code in `lane_finder.py` in the function `draw_lane_line()` to draw the overlaid image (biggest image in the mosaic). I then used [Wonderful Pipeline by John Chen and Yu Shen](#) to create the big stitched image in function `pipeline()` from line 61 to 84. 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!).

Here's a [link to my video result within the repo](#) and here's the [Youtube Link](#)

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.

To successfully detect lines I did following

- Undistorted images
- Performed a perspective warp transform

- Applied a mix of color, magnitude and direction threshold to filter lane lines
- Use polynomial fitting and Sliding window approach to detect lane lines

Pipeline can and will fail if car were to switch lane lines or slightly change it's direction. Or distance between lanes changes or at an intersection. This pipeline is far from perfect and can be used to perform more efficiently if we train a model to automatically detect lanes by considering factors like fellow cars are moving besides each other at an average fixed distance (so there must be a lane), if possible we can cross verify it with satellite imagery to cross check our bird eye's view.