# Advanced Lane Finding Project Writeup

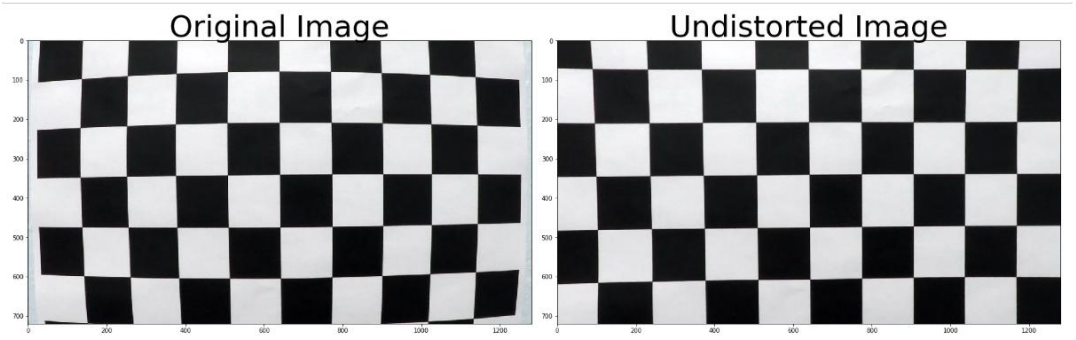
**The goals / steps of this project are the following:**
* Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
*Apply a distortion correction to raw images.
*Use color transforms, gradients, etc., to create a thresholded binary image.
*Apply a perspective 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.

A jupyter/iPython notebook was used and can be found in the examples folder labeled “Advanced Lane Finding.ipynb.”

**Rubric Points**
Here I will consider the [rubric points](https://review.udacity.com/#!/rubrics/1966/view) individually and describe how I addressed each point in my implementation.

**Camera Calibration:**

I started out by calibrating the camera to remove inherent distortions that can affect its perception of the world. Once calibrated, I wrote the function, cal_undistort, that undistorts any image passed through it.
The code for this step is contained in the third code cell of the IPython notebook. 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 coeffificients using the cv2.calibrateCamera() function. I applied this distortion correction to the test image using the cv2.undistort() function and obtained this result:




**Pipeline (single images):**

**Distortion corrected Image**

I applied the distortion correction to one of the test images like this one:


**Threshold binary images**

After that, I masked the image by only looking at the pixels inside of a region of interest. I, then, wrote pipeline, which combines color and gradient thresholds for lane detection. The result is a binary image, which accentuates the “lane lines.”


**Perspective transform**

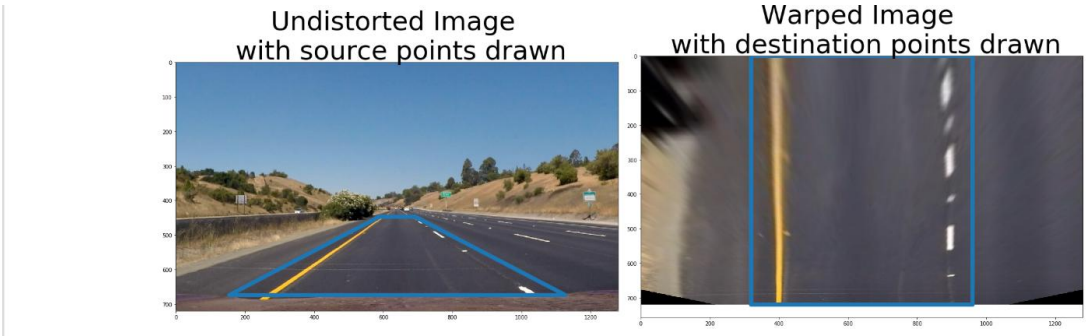
Next, I dealt with warping the image. Since parallel lines appear to get closer the farther they are from the camera, I applied a perspective transform in order to get a bird's-eye view of the lane lines. The code for my perspective transform is includes a function called warp() , which appears in the sixth code of the IPython notebook). The warp() function takes as inputs an image ( img ), as well as source ( src ) and destination ( dst ) points. I chose the hardcode the source and destination points in the following manner:

```
src = np.float32(
    [[0.46 * img.shape[1],0.62 * img.shape[0]],
     [0.54 * img.shape[1],0.62 * img.shape[0]],
     [0.88 *img.shape[1],.935*img.shape[0]],
     [0.12 * img.shape[1],.935* img.shape[0]]])

offset = img_size[0]*.25

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

This resulted in the following source and destination points:

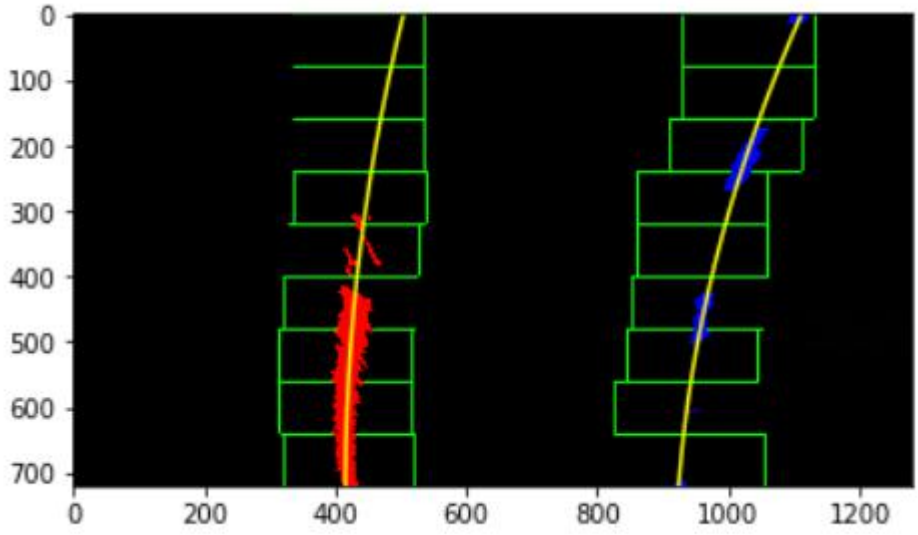
| Source | Destination |
|---|---|
| 589, 447 | 320, 0 |
| 692,447 | 960,0 |
| 1127,674 | 960,720 |
| 154,674 | 320,720 |

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.

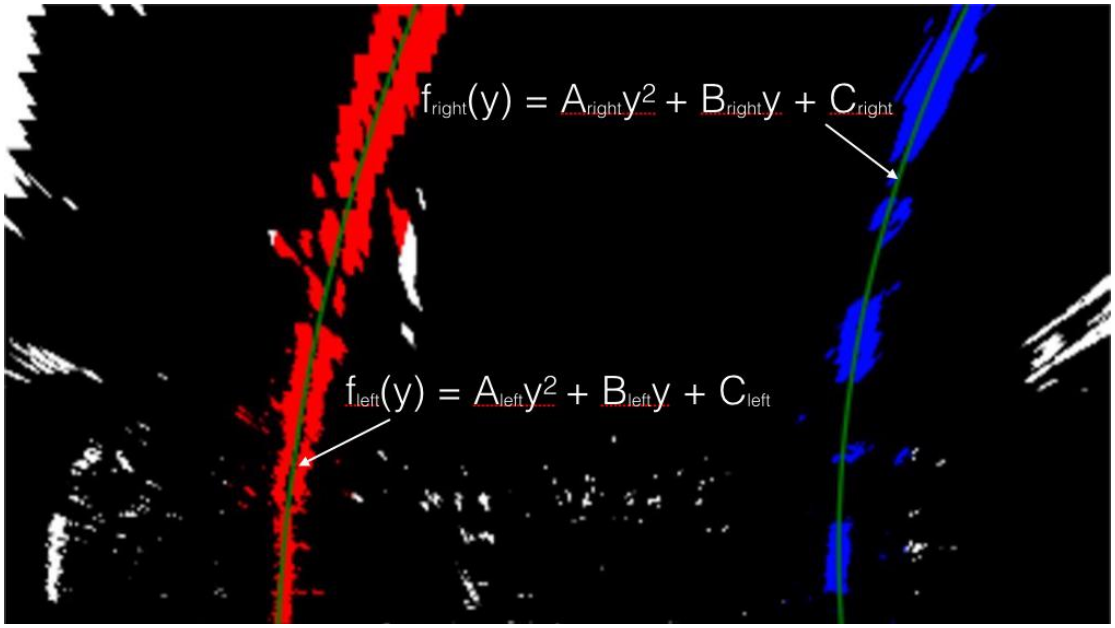



**Lane line pixel identification and polynomial fit**

Once warped, I used a histogram along all the columns in the lower half of the image to find peaks which could potentially be lane lines. After that, I used a sliding window technique to find and track the curvature of the lane lines.




Then I fit my lane lines with a 2nd order polynomial kinda like this:




Once I found lines, I skipped the sliding windows step because it is inefficient to start fresh on every frame since the lane lines don't move a lot from frame to frame. Instead, I just searched in a margin around the previous lane line position. So, once I knew where the lines were in one frame of the video, I was able to do a highly targeted search for them in the next frame. Then, I unwarped the image so that it could be from the view of the camera. Finally, I drew the lane lines onto the image.

**Radius of curvature calculation and vehicle center offset**

Having identified the lane lines, I estimated the radius of curvature of the road and the position of the vehicle with respect to the center of the lane. The code for this can be seen in code cells 13 and 14.
Here is an example of my result on a test image:


**Pipeline (video)**

Finally, I applied this process to the video.
The video are in the Jupyter Notebook and are labeled "project_video_output.mp4", "challenge_video_output.mp4," and"harder_challenge_video_output.mp4."

**Discussion:**

**Issues and Improvement**

The pipeline works well for the most part, but it is far from perfect. On the challenge video, I noticed the algorithm gets a little confused. I think that the center divider or its shadow is being confused for a lane line. There could be many lines close to vertical that could be confused for lane lines. One possible fix for this could be to choose the two lines closest to the center of the image as lane lines. We could also set some kind of a lane line width since lane lines should have a minimum width requirement. That would definitely prevent the algorithm from picking up two lane lines that are too close to one another.

The harder challenge video had its own set of issues. My algorithm was sometimes unable to detect the lane lines properly. I noticed that when there are issues when there are sharp turns. When there are turns, the lane lines do not appear trapezoidal as they do when the lines are parallel. They appear

curvier and more horizontal than vertical. That threw off my algorithm for different reasons. In my algorithm, I applied region masking using a trapezoid, which means anything out of the trapezoid were left out. With sharp turns, the lane lines to outside of that trapezoid, so it is no wonder they were not detected. Getting rid of the region masking or simply modifying it by making it wider near the top of the image would definitely help. Another possible improvement could be to use a sobel threshold in the y direction to help identify the horizontal lines better. In my algorithm, I only used a sobel threshold in the x direction. Finally, I could experiment with different color channels to see if they do a better job of detecting the lane lines. In my algorithm, I only used the S channel.