**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.

### **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**](https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) **is a template writeup for this project you can use as a guide and a starting point.**

You're reading it!

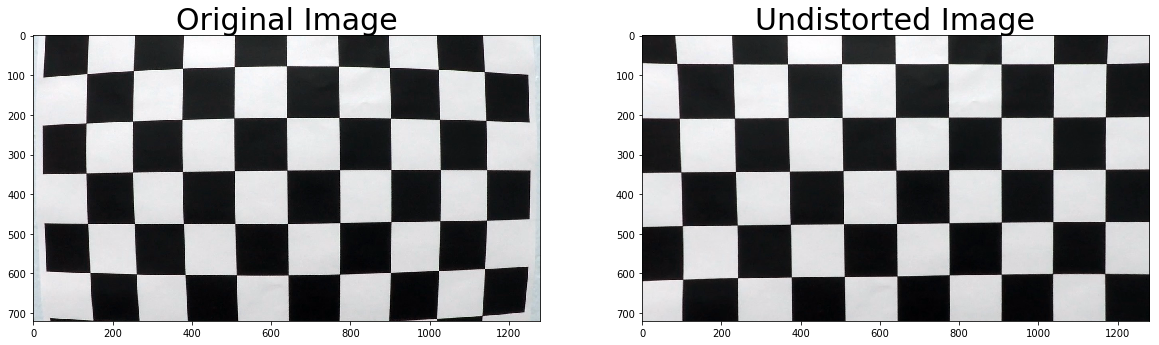
### **Camera Calibration**

#### **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 the file called camera\_calib.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:



### **Pipeline (single images)**

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

#### Here is the original image:

#### 

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

The code is in the Python file ‚pipeline\_functions.py‘ and the function is called undistort.

I used the cv2.undistort() function which has as input the image, camera calibration and distortion coefficients (resulting from the camera calibration). The output is then uncalibrated image which is shown below:



#### **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 lin the function binary\_image pipeline\_functions.py). Here's an example of my output for this step. (note: this is not actually from one of the test images)



I conducted the following steps:

* Convert the undistorted image to HLS color space and separate the S channel
* Convert image to grayscale
* Apply Sobel x
* Apply Threshold x gradient and threshold color channel
* Stack each channel to view their individual contributions in green and blue respectively
* Combine the two binary thresholds
* Return binary image

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

#### In comparison to the original image I got the following warped image:

#### 

The code for my perspective transform includes a function called corners\_unwarp, which appears in the file pipeline\_functions.py. The warper() function takes as inputs an image (img). I chose the hardcode the source and destination points in the following manner (where h,w are the image height and width)

h,w =img.shape[:2]

src = np.float32([[w,h-10], #rd

[540,470], #lu

[740,470], #ru

[0, h-10]]) #ld

dst= np.float32([[w,h], #rd

[0,0], #lu

[w,0], #ru

[0, h]]) #ld

This resulted in the following source and destination points:

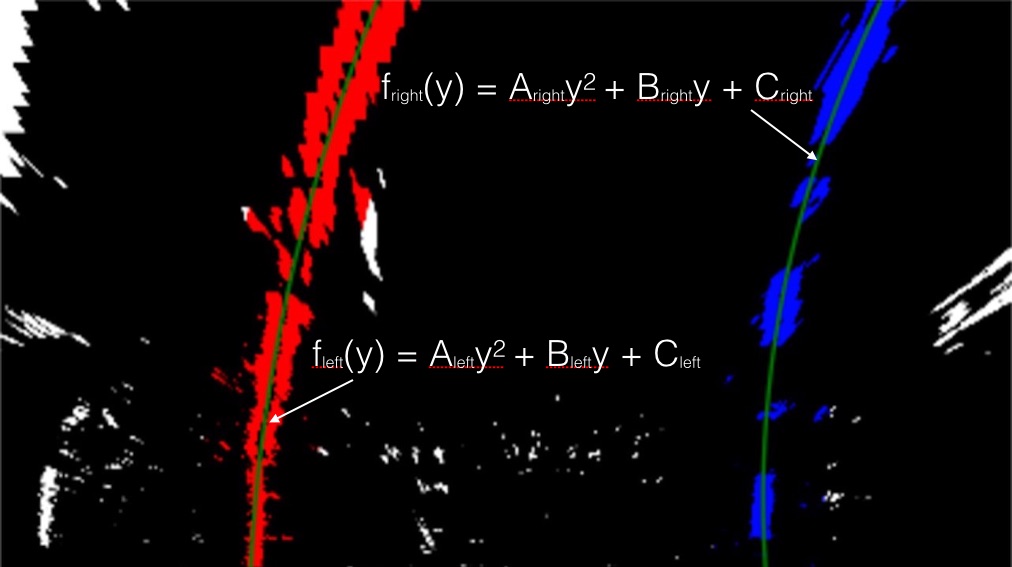
| **Source** | **Destination** |
| --- | --- |
| w, h-10 | w, h |
| 540, 470 | 0,0 |
| 740, 470 | w, 0 |
| 0, h-10 | 0, h |

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?**

Then I did some other stuff and fit my lane lines with a 2nd order polynomial kinda like this:



* I took as input the binary, warped image.
* Then I am calculating a histogram in order to find the peak of the left and right halves of the histogram. These will be the starting point for the left and right lines.
* Using the sliding windows technique the x and y values for each window are defined.
* When stacking the values into an array it is possible to calculate with numpy.polyfit the required polynomial which is the base for drawing the lines.

#### **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.**

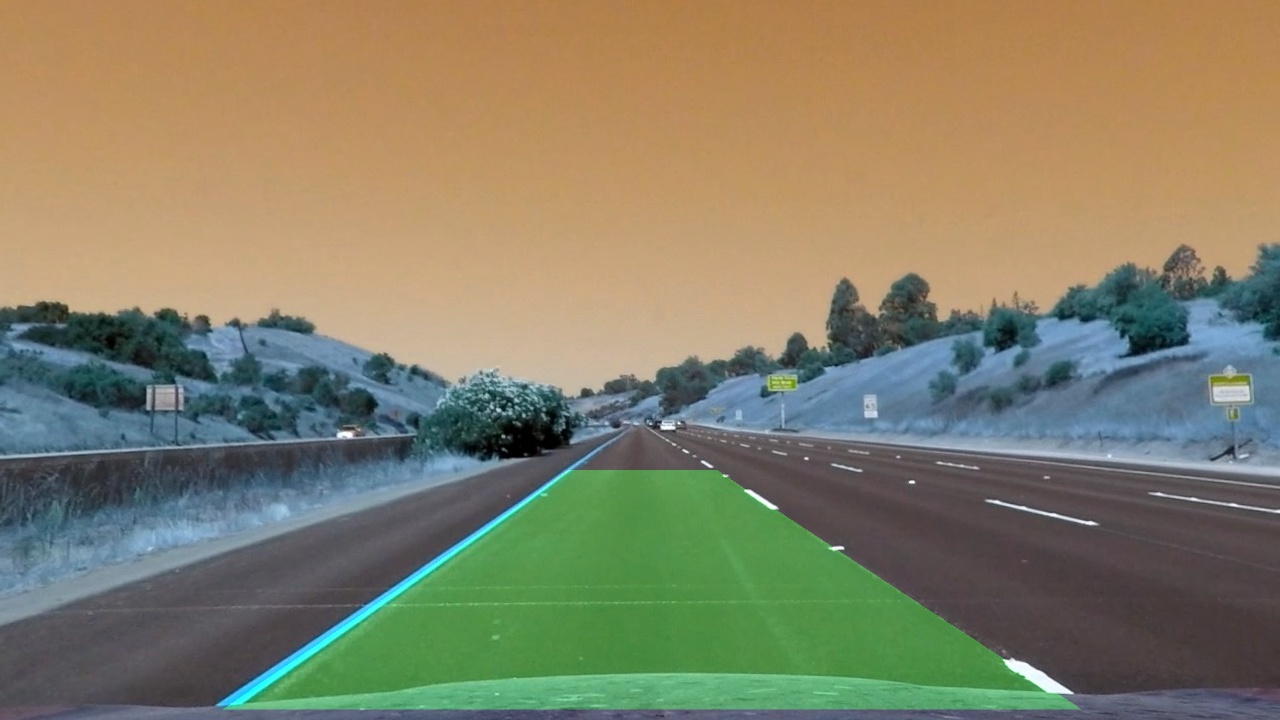
I did this in lines 109 through 127 in my code in run\_video.py. After calculating the polynomials I am checking:

* if the coefficients are larger than 1.3 in order to avoid implausible values. If yes, then the old values of the previous image are used

For defining the curvature I am evaluating the formula at a fix point which is the bottom of the image. If the curvature is not plausible then the old values of the previous image are used. Moreover, I am checking that the difference between two images is plausible by checking the distance between the x values.

#### **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 148 through 167 in my code in run\_video.py. 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:

<https://github.com/AGrosserHH/CarND-Advanced-Lane-Lines/blob/master/white.mp4>

which is also in the project folder with the name white.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?**

I had the following issues:

* I faced the issues that my coefficients went off a particular scale and hence I checked that the coefficients are within a certain range. Hence I am checking that the values are between -/+1.3.
* If some images did not find any line (due to the bad road conditions) then the coefficient went off as well. In order to ensure that the coefficients are within a certain range then I tried to ensure that if a position of an coefficient diverges from an image to another by 200 then I am using the coefficient of the „old“ image.

The image might fail if the sun is shining very bright.