**Project 4 : Advanced Lane Finding Project**

**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**](https://review.udacity.com/#!/rubrics/571/view)**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**](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 first code cell of the IPython notebook located in the main directory.

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.

Then, in the second code cell, I used the output objpoints and imgpoints to compute the camera calibration and distortion coefficients using the cv2.calibrateCamera() function.

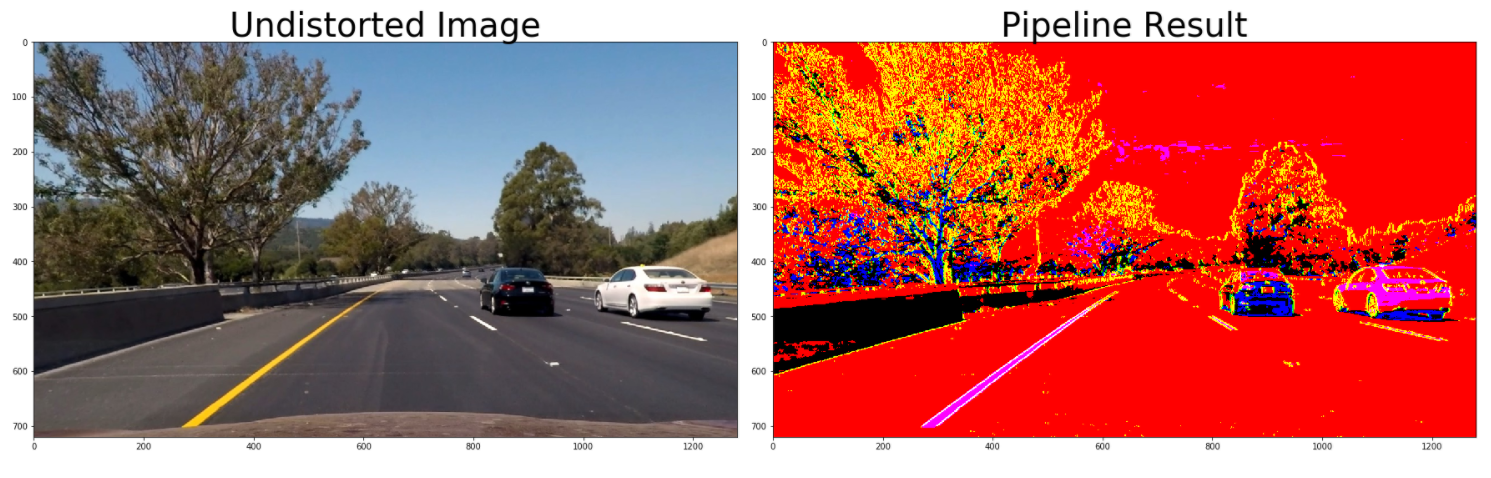
**Pipeline (single images)**

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

I applied the distortion correction to the test image using the cv2.undistort() function using the distortion coefficients from the cv2.calibrateCamera() function and obtained this result:  


**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 in the third cell of the IPython notebook ). 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 corners\_unwarp(), which appears in lines 2 through 12 in the 4th code cell of the IPython notebook.

I also created a function called corners\_rewarp() and draw\_corners in the same code cell of the IPython notebook.

The first function does the same job as corners\_unwarp() but inverses source (src) and destination (dst) points to go back from the perspective transform to the original shape.

The second function draws the region formed by the source points in the undistorted image.

The called corners\_unwarp()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([[225,700],[590,450],[690,450],[1100,700]])

new\_top\_left=np.array([src[0,0],0])

new\_top\_right=np.array([src[3,0],0])

offset=[150,0]

dst = np.float32([src[0]+offset,new\_top\_left+offset,new\_top\_right-offset ,src[3]-offset

This resulted in the following source and destination points:

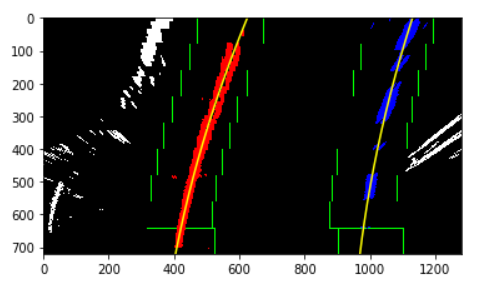
| **Source** | **Destination** |
| --- | --- |
| 225, 700 | 375, 700 |
| 590, 450 | 375, 0 |
| 690, 450 | 540, 0 |
| 1100, 700 | 950, 700 |

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 (see image below).



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

Then I created a function called find\_lines() that uses the code given in the lecture to find the lane lines using the histogram and fit them with a 2nd order polynomial. The code is located in the 5th code cell of the IPython notebook and this is the output image of the cell:



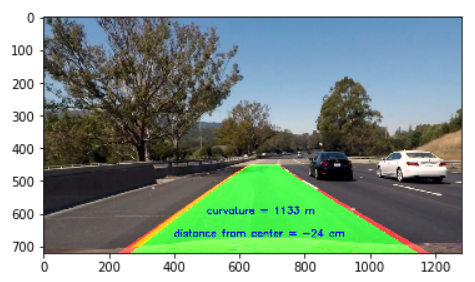
Then I created another function called find\_next\_lines() (located in the 6th code cell of the IPython notebook) to find the lane lines in the image taking into account the position of the lines from the previous image.

**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.**

For the curvature, I used the code from the lecture, while for the distance from center I computed the difference in pixels between the center of the image and the center of the lane lines then multiplied it by the ratio meter/pixel in the x dimension. The code is located in the 7th code cell of the IPython notebook.

**6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.**

I created a function called draw\_lines() which does the calculations for the curvature and distance from center and also convert the transform perspective to real shape using the function corners\_rewarp(). 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 called output\_project\_video.mp4 can be found in the main folder

**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?**

At first I used the technique of the sliding windows to find the lines but it did not work well on the video. That is why I used the other method.

My pipeline failed in the challenge\_video and the harder\_challenge\_video. One thig we could do for the harder challenge is to fit the lines with a 3rd order polynomial.