## \*\*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.

**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!

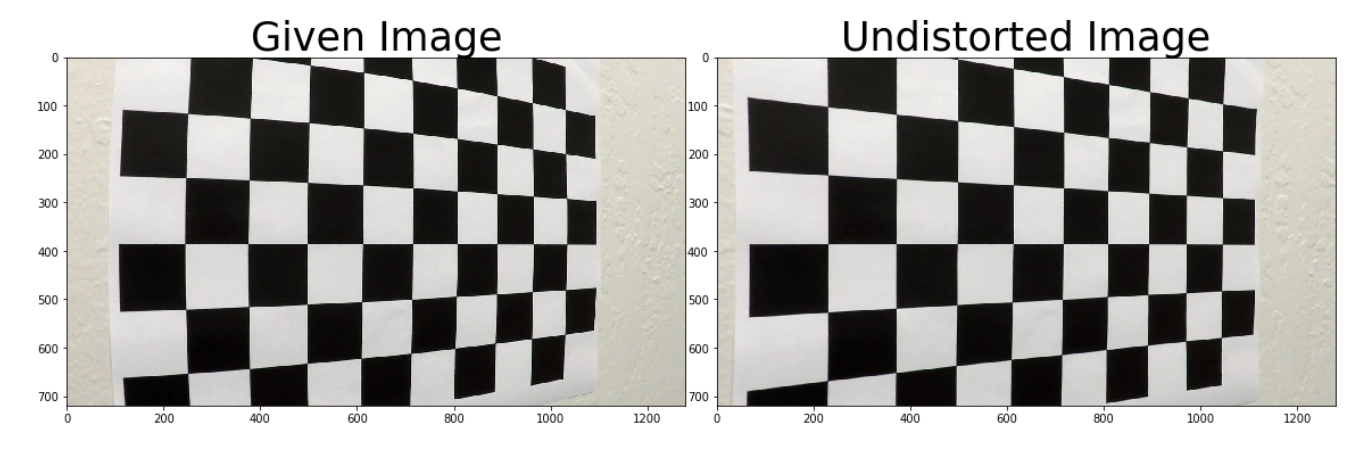
**Camera Calibration**

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 Section-1 of the IPython notebook located in "./examples/example.ipynb"

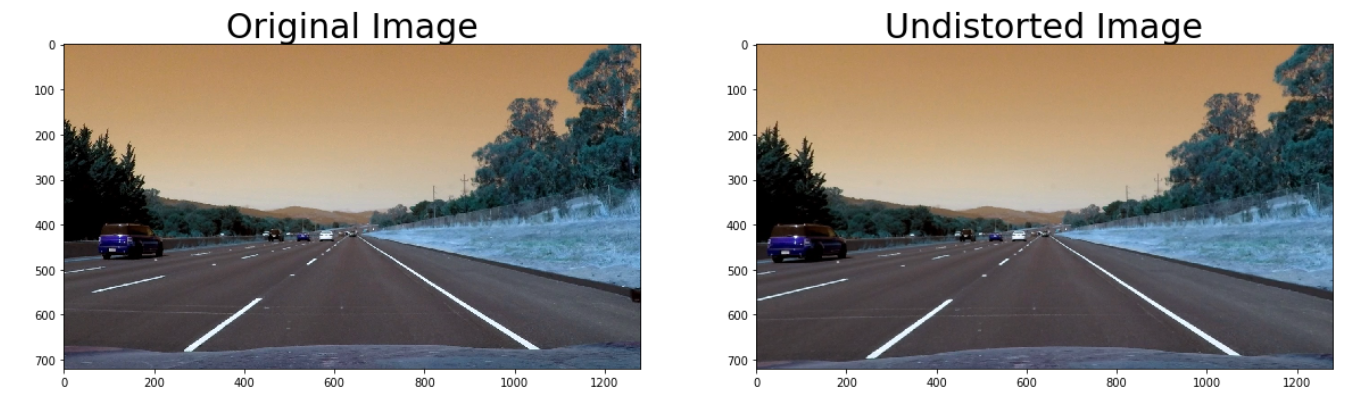
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of 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:

->Sample undistorted image of the chessboard.

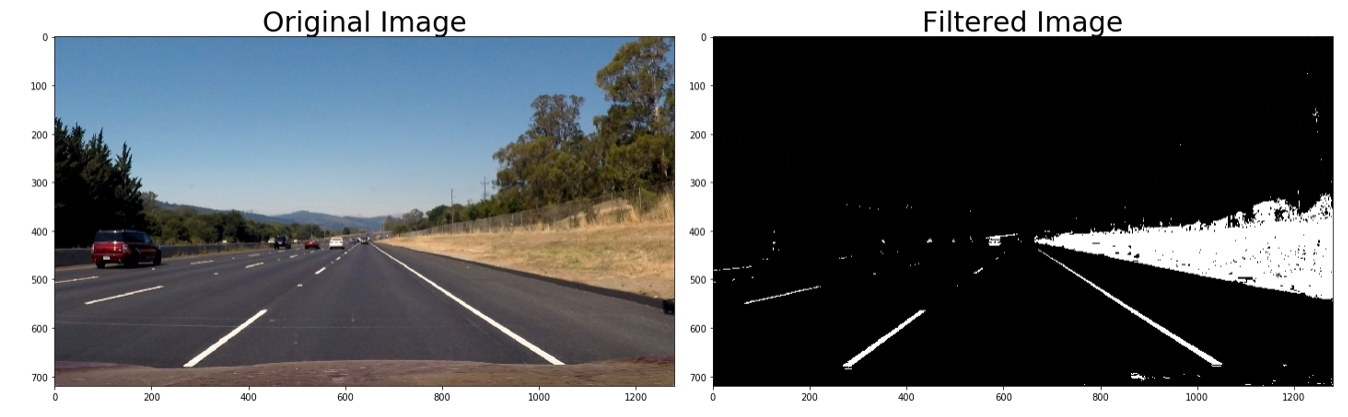
**Pipeline (single images)**

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: This was done in Section-2 of the notebook

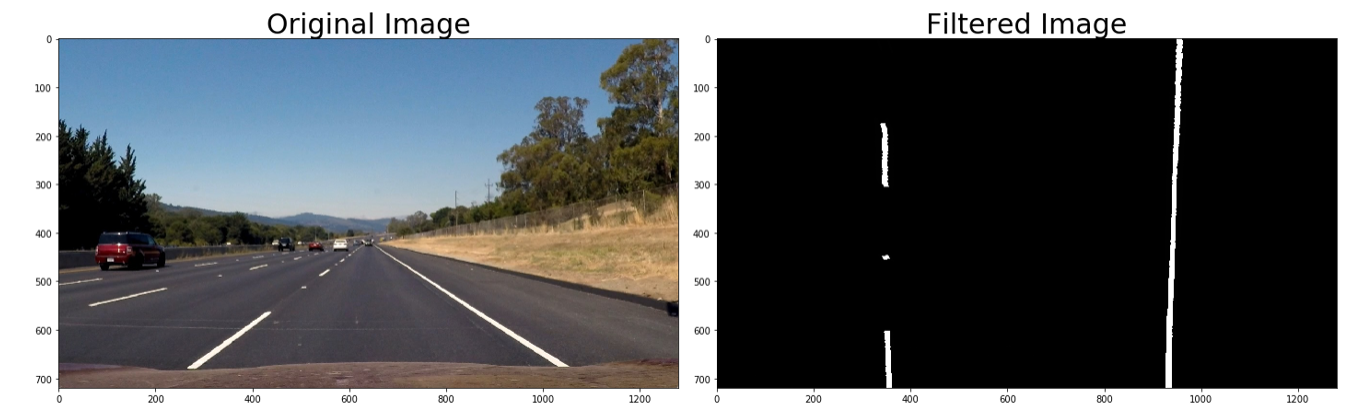
Once the object and image points were generated and the calibrateCamera() function used, we had access to the calibration and undistortion matrices which helped us to cv2.undistort() function to undistort any image given to us for the rest of the project. **We can see that the colors are slightly incorrect in the above image due to BGR and RGB discrepancy**

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 Here's an example of my output for this step. This can be seen through Section-5 & 6 of my code

In the "image processing pipeline" in section-6 of my jupyter notebook , I used all three categories of the filter available including the sobel\_absolutex filter(thresh-(25,225),sobe\_absolutey(25,225) filter the sobel magnitude filter(14,230) and the sobel directional gradient filter(thresh-(0,0.7)).The independent functions and trial runs can be found in section-5-1 and section-5-2.

For the color thresholds we used both hls and hsv threshholds.HSV threshold([15,38,125] to [35,206,255]) were used for the yellow lines and hls threshold ([0,200,0] to [180,255,255]) were used to detect the white lines which are more prominent in the l space. Also an overall mask threshold was used above([60,60,60]. This color based threshold was later combined with the overall threshold.

Here we can see the pipeline working on an unwarped image of the road

**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 `UNWARP()`, which appears under the perspective tranform section in my notebook The `unwarp()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. This is my function

*h,w = img.shape[:2]*

*# use cv2.getPerspectiveTransform() to get the tranformation and reverse matrix*

*M = cv2.getPerspectiveTransform(src, dst)*

*Minv = cv2.getPerspectiveTransform(dst, src)*

*# use cv2.warpPerspective() to warp your image to the required perspective*

*warped = cv2.warpPerspective(img, M, (w,h), flags=cv2.INTER\_LINEAR)*

*return warped, M, Minv*

My source and destination points were carefully selected by looking at the lane lines and going through a trial and error process. My offset was also similarly chosen to be 350. **The trial run and the subsequent function can be seen in section-4 of my notebook**

This resulted in the following source and destination points:

| Source | Destination |

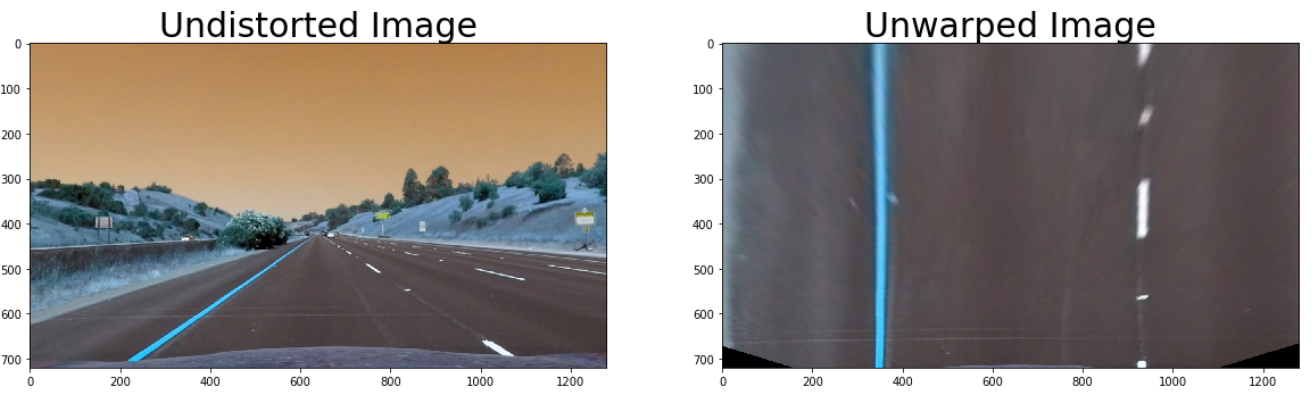
|:-------------:|:-------------:|

| 575, 465 | 350, 0 |

| 707, 465 | 930, 0 |

| 260, 685 | 350, 720 |

| 1050, 685 | 930, 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 relatively parallel in the warped image. I also return the inverse transform matrix for future uses.

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

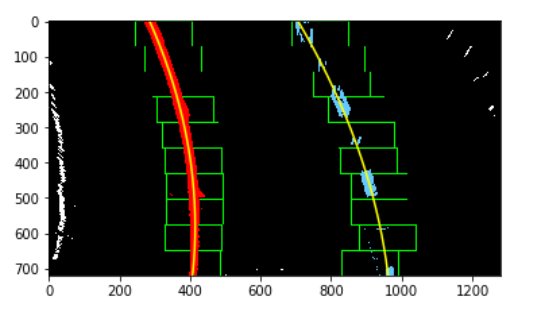
I have identified the lane pixels and fit them to a polynomial in the "sliding window search function" section where I have used the sliding window search where the image was divided into 10 windows and each of those windows were separately centered on the mean position of the bulk of the non zero pixels in that section. These non-zero pixels were then assigned to the window section and concatenated into two arrays for the left and right lanes. The left and right lane indices were then fit to 2nd order polynomial and the polynomial parameters were then returned as lane\_fit indices.

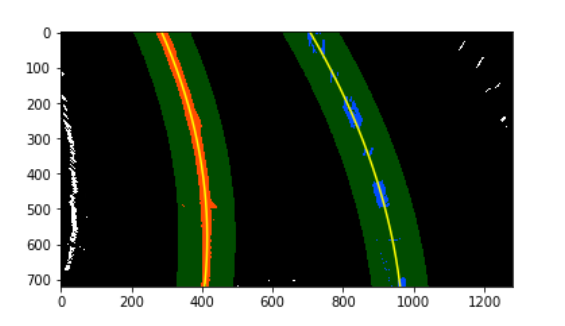
For later in the pipeline, to reduce the processing time, the polyfit\_using\_prev function was used where instead of sliding windows across the entire image, the windows were slid only within a margin are of the previous fit lines which was chosen as 80 pixels.

Later on in the function "process image" in the second last section of the notebook, a condition was set that if the lanes were not detected in the previous frame the function would use sliding window approach, otherwise the function would use the margin based approach.

Other conditions were also added to the same process\_image function including that if difference between the bottom pixel on the lanes is less than 350, the fit is invalidated. This was done so as to verify the authenticity of the fit

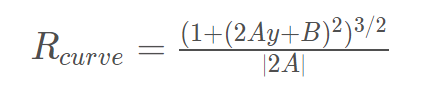
Result of the sliding window function:



Result of the polyfit\_using\_prev function

**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 my function curv\_and\_center\_dist() in the Finding lane curvature and finding deviation off center section using the given relation



where A and B are the polynomial coefficients of the fit generated on the lane lines.

**This was replicated in the section-9 of the code** where the final curvature was then found to be the average of the curvature of both the lines. This was done by choosing the following values for the pixel per meter parameters.

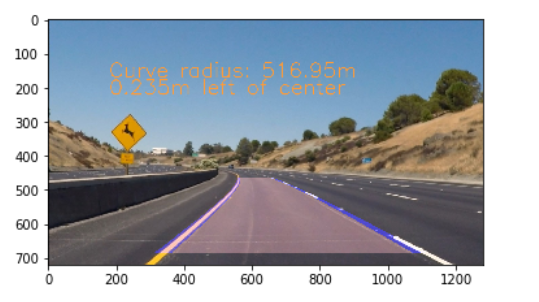
ym\_per\_pix = 3.048/80

xm\_per\_pix = 3.7/700

The ym criteria was based on the lane line and the xm criteria was based on the lane width

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 section-9 of my code and this was what the result looked like



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!).

It is attached with the given 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?

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. I think the image pipeline and the calibration was one of the biggest issue I faced. I think the calibration was the reason My project did not run on the challenge video. I would like to develop standardized procedures for doing the calibration of filters and the development of the pipeline. Another issue I faced is the smoothness of the filter and the measurement. If this is applied to an actual car, it would be jittery.

I think that we would have to apply something like a Kalman filter to smoothen the jitters and ensure roper control of the car. Also a debugging tool has to be integrated into the overall pipeline to ensure that we are able to pinpoint the issue whenever we face one.