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.

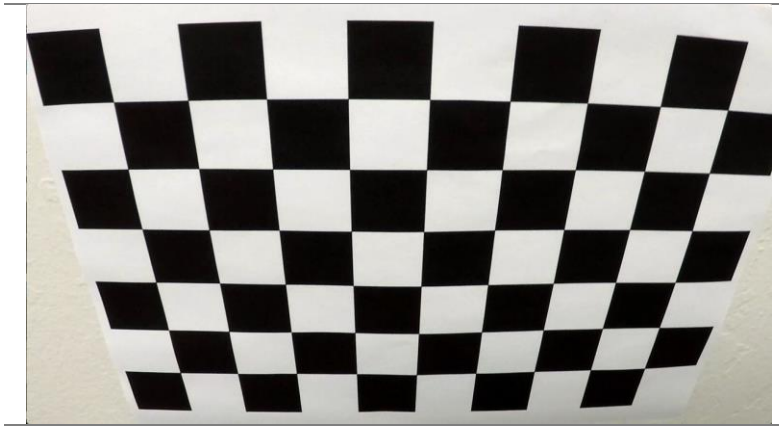
1. Camera Calibration

a. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

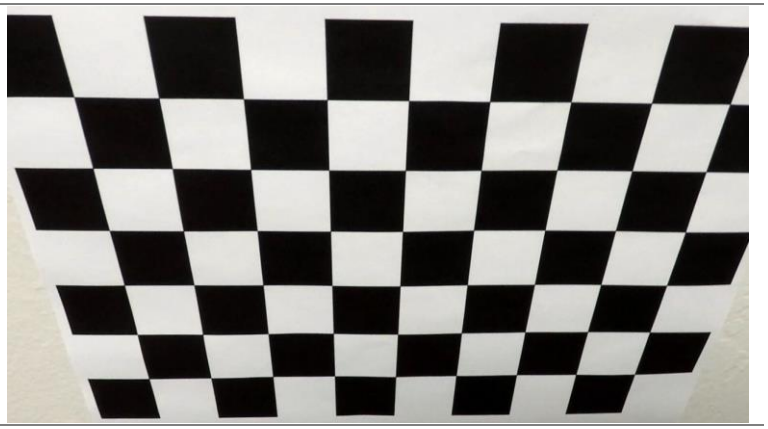
The code for this step is contained in the fourth code cell of the IPython notebook “code1.ipynb”.

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:



Original Image



Undistorted image

2. Pipeline (single images)

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

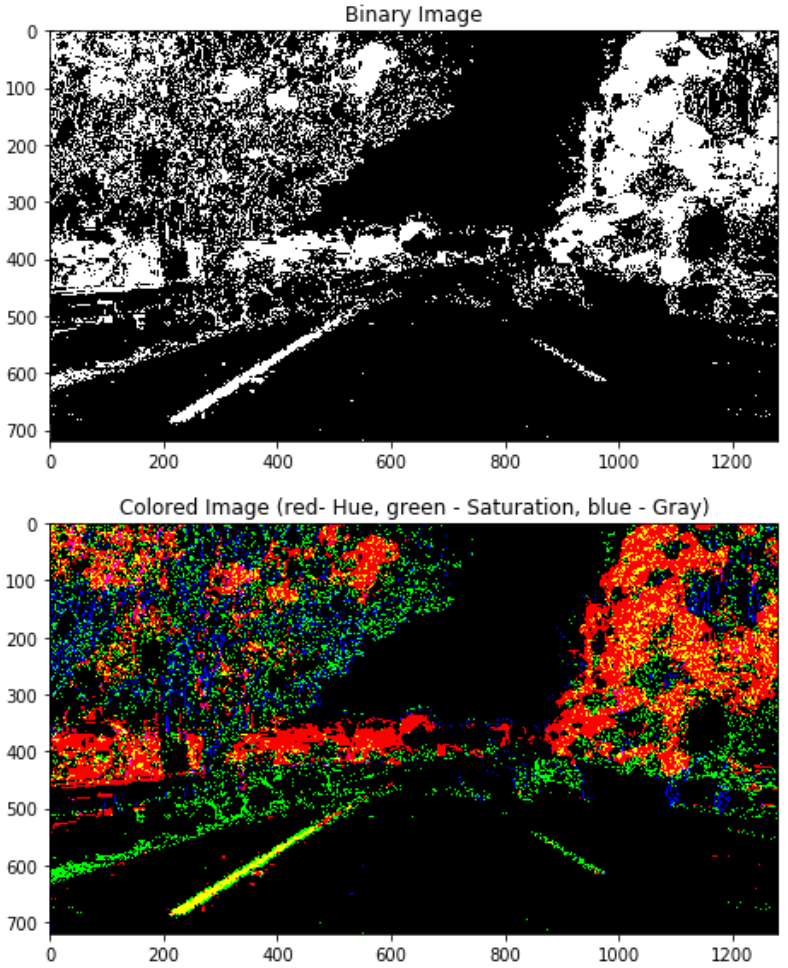
Original Image

Undistorted image

b. 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 code blocks 8 and 9 in *code1.ipynb*). Here's an example of my output for this step. (note: this is not actually from one of the test images)

Original Image



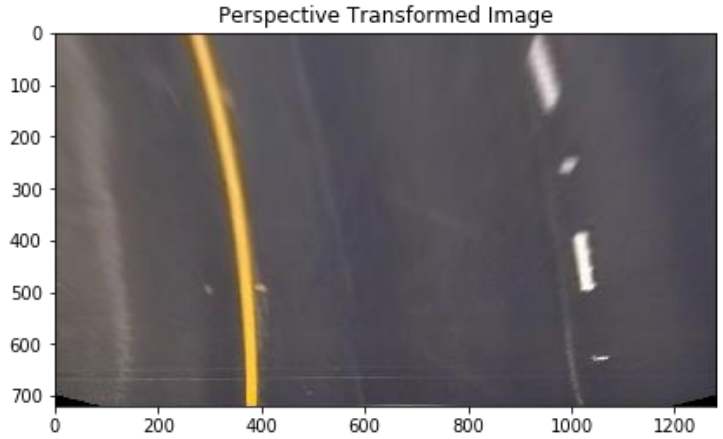
Binary Images

c. 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 *apply_perspective()*, which appears in the 6th code cell of the IPython notebook. The *apply_perspective()* function takes as inputs an image (img). I chose the hardcode the source and destination points in the following manner:

	SOURCE (X,Y)	DESTINATION (X,Y)
TOP LEFT	572,460	280, 1
TOP RIGHT	708,460	1000, 1
BOTTOM LEFT	220,680	280,720
BOTTOM RIGHT	1060,680	1000, 720

It returns the perspective transformation matrix, inverse-perspective transform matrix and the warped image

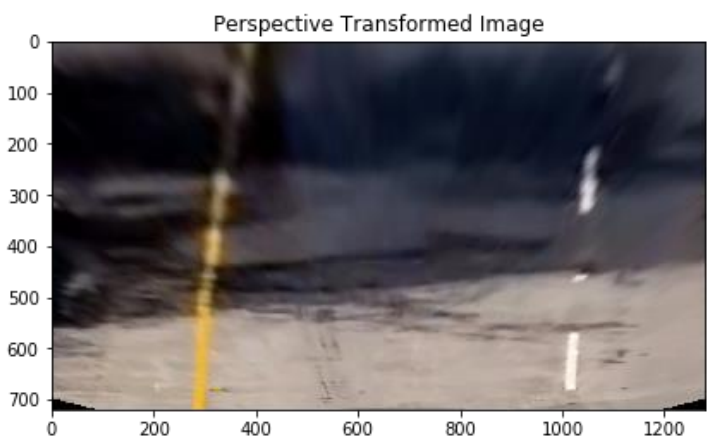


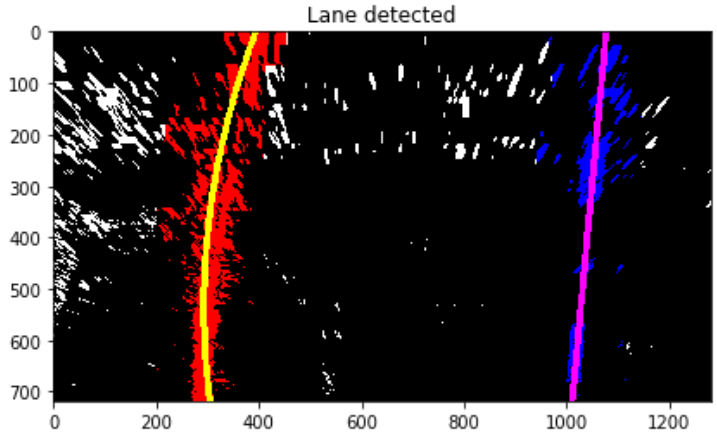
d. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

In code block 14, the ***lane_polynomials()*** function, is main function which is responsible for lane lines detection from the first frame of the video. First it identifies the position of all non-zero pixels in the binary warped image. Then it divides the image vertically into 10 sections. It then initializes a left and a right window, on positions that are given by the arguments (*leftx_base, rightx_base*). Upon getting a minimum number of pixels, given by *minpix*, it changes the positions of the left and right windows and traverses to the top. The positions along **x** are changed based on 20% of previous value and 80% of the new mean value

of pixels. This is to prevent abrupt change in position of windows along x-axis. After getting all the pixels constituting the left and right lane, a second order polynomial is made using **numpy.polyfit()**. After this ***plot_lane_lines()*** is called.

plot_lane_lines() function plots pixels constituting the left lane in red color and the right lane in blue color. It also plots the second order polynomial curves formed by the pixels of left and right lanes. An example is:





e. 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.

The radius of curvature is calculated in function ***radius_of_curvature***. It takes two arguments namely, *(left_fit, right_fit).* These are left lane polynomial and right lane polynomial equations. It plots linearly spaced points using **numpy.linspace()** in these two second order polynomial equations and find the radius of curvatures using the formula given in classroom materials. It also normalizes the pixel values to corresponding value in **m**.

The position of the vehicle is calculated in ***vehicle_position*** function. It calculates the difference between the midpoint of the lane and midpoint of the lanes, and then return a tuple. This tuple contains two values, first one is the distance of the vehicle's midpoint from left lane, second values is the distance of the vehicle's midpoint from right lane. Both values are given in **m**.

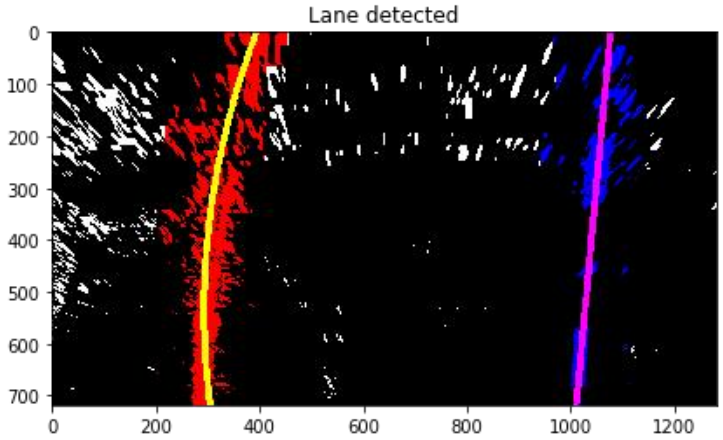
f. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

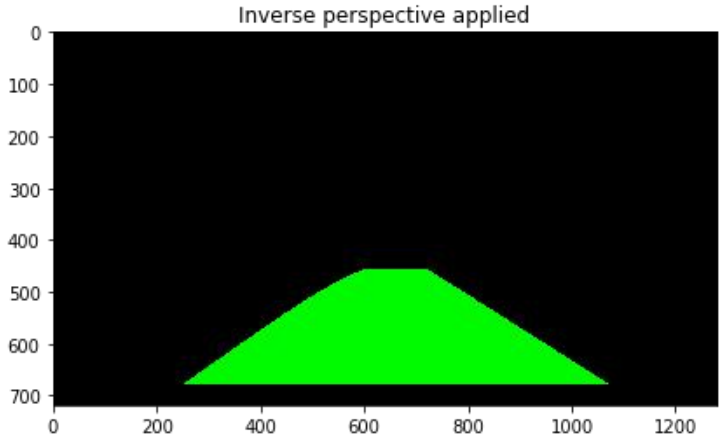
Here is an example of my result on a test image:

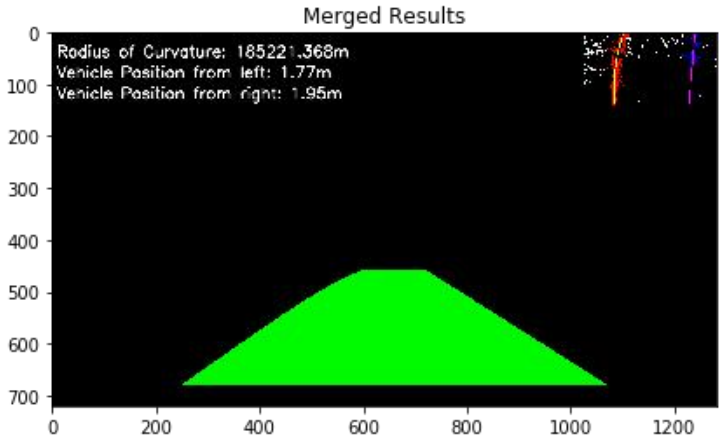
3. Pipeline (video)

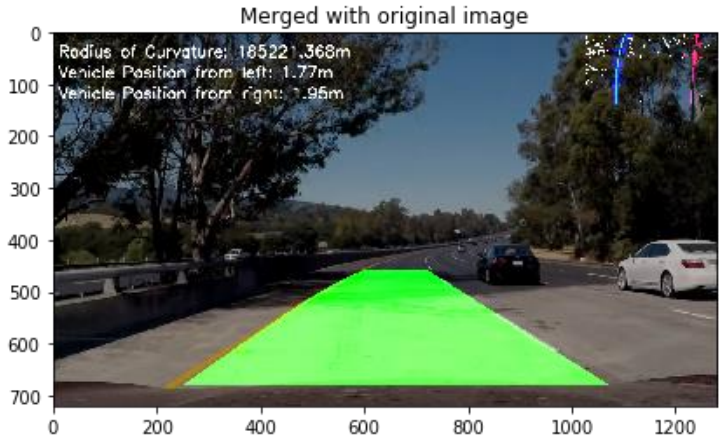
a. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

The video is located in */output_video/project_video.mp4.* However here are images from different stages in the pipeline.









4. Discussion

a. 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?

The approach that I undertook in the project is:

I. Calibrate the camera to get distortion coefficients and matrix
II. Read a frame, from the clip
III. Undistort the frame/image using the distortion coefficients and matrix obtained.
IV. Apply color/gradients thresholding
V. Perform perspective transformation on binary image
VI. Detect the left and the right lanes from the warped binary image and find a suitable second order polynomial equations for both lanes
VII. Fill the area between the two curves, and place it back on the road using inverse perspective transform.
VIII. Calculate the radius of curvature, and position of the vehicle and print it in above image
IX. Merge the above image obtained with the original frame/image
X. Place the merged frame back in the clip and go to step II

Some of the issues that I faced are:

- First of all, I was unable to detect the lane line pixels properly in the color/gradient thresholding. My initial approach was check the change in gradient along x-axis and also saturation channel between certain thresholds. In this approach, in images the lane lines which were in shadows weren't detected properly. I came across an article from https://navoshta.com/detecting-road-features/ where magnitude and direction of saturation channel was also used. Inspired from the post, I incorporated the changes and lane lines were better detected now. After that I also used the hue channel with thresholds (20,32) which represents yellow color in the image to detect the yellow lines. After these fine tunings I was able to detect the lane lines properly.

- Next challenge was to prevent the window which detected the lane pixels to abruptly change positions based on mean on the current 'non-zero' pixels. For this I used weighted decision such that the new position of the window will be based 0.2 of previous position and 0.8 of the new mean position.

- After this when I tried my pipeline on challenge video, I that after some time, the right lane translated to the left and overlapped the left lane. To prevent this, I kept a sanity check so that the left lane remains on left 40% of the image and the right lane remains on the right 40% of the image, and both do not cross the midpoint.

Some cases in which the pipeline might fail are:

- There are bumps in the road which change the perspective transformation
- There may be U-turns in the road where the right or left lanes go past the middle of the screen
- In places where lane lines have worn out