Writeup

Advanced Lane Finding Project

Advanced Lane Finding Project

The goals / steps of this project are the following:

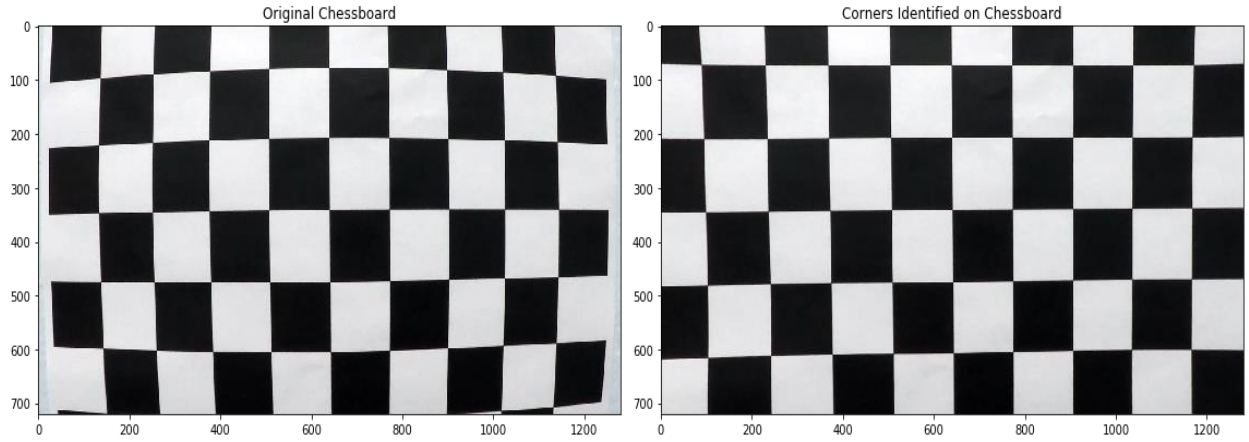
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- 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.

Camera Calibration

The code for this particular step is contained in the second code cell of the IPython notebook. I began by preparing both 3d coordinates in world space and 2d coordinates of the image that correspond to the imaging plane. I then went on to find the chessboard corners from a few calibration images using `cv2.findChessboardCorners()` so as to populate my 2d image coordinates.

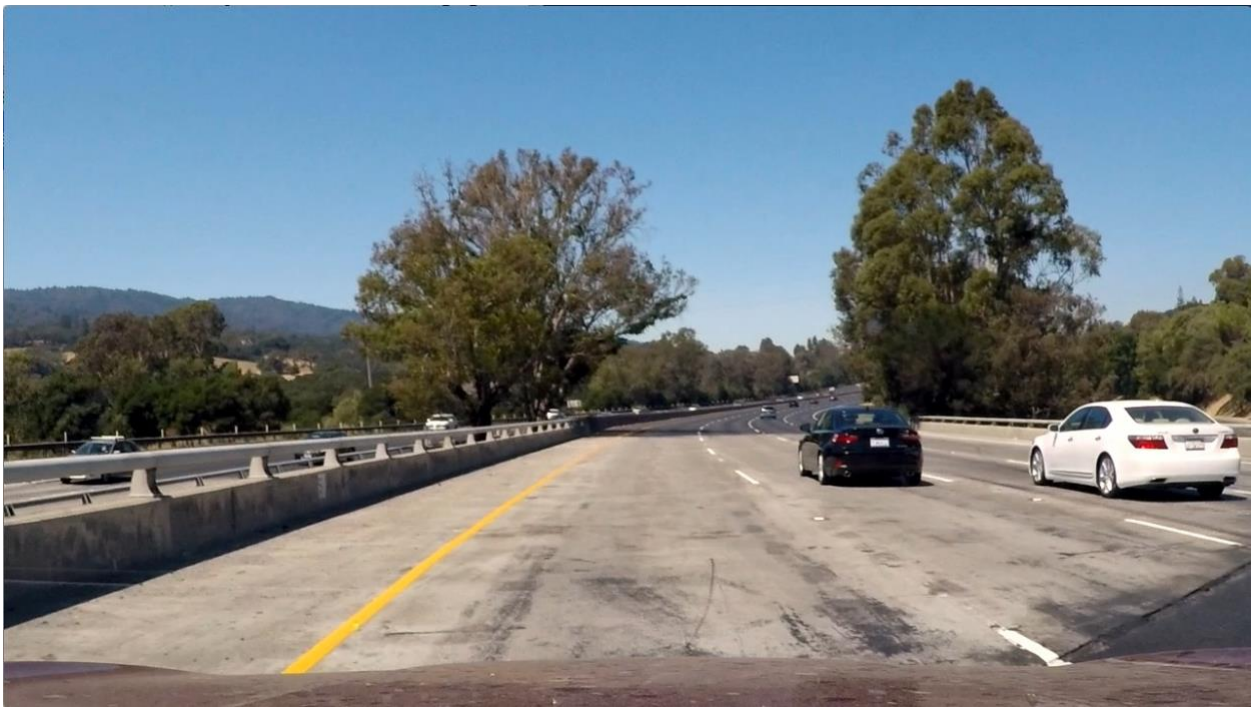
I then applied `cv2.calibrateCamera()` to get the camera matrix and distortion coefficients that were crucial in my application of `cv2.undistort()` to get the final undistorted image.

An example of the result is displayed below.



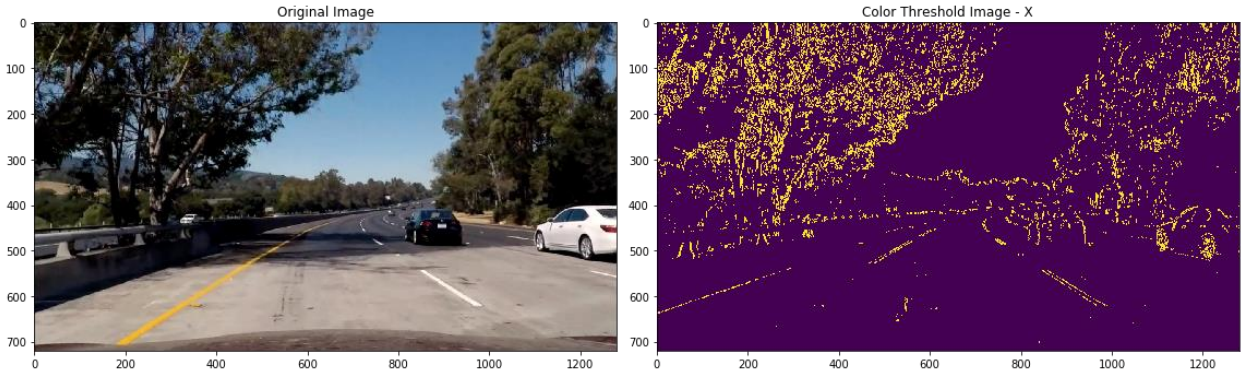
Pipeline (single images)

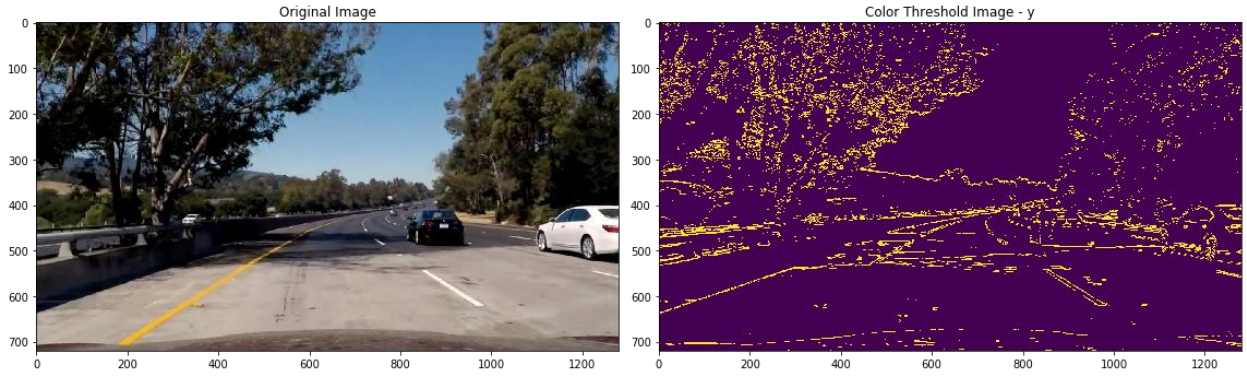
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:



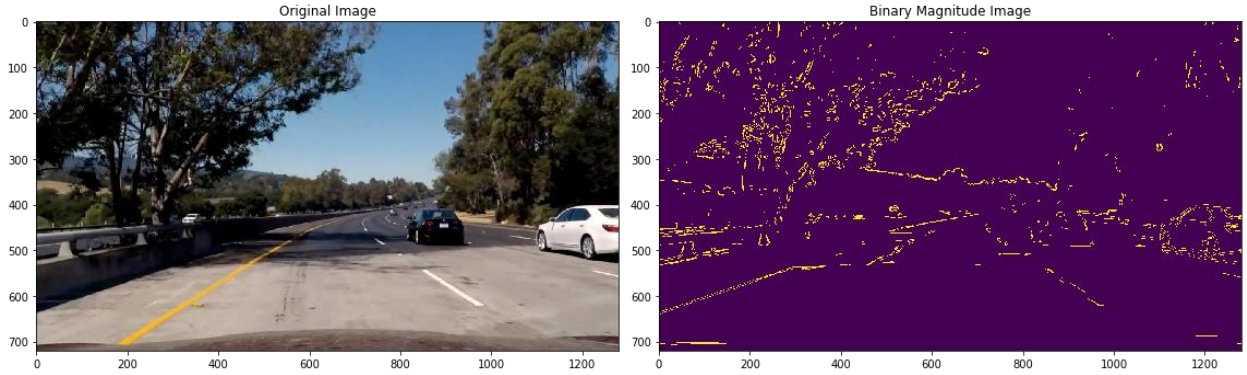
I used a variety of gradient and color thresholding techniques so as to get a binary image where the lane lines are easily identifiable.

I began by calculating the directional gradient (code cell 5) by applying `cv2.Sobel()` on both the x and y coordinate directions. I then scaled the resultant image for computational purposes before applying a threshold to ensure that only pixels within range are visible. An example of this for both x and y directions is displayed below.

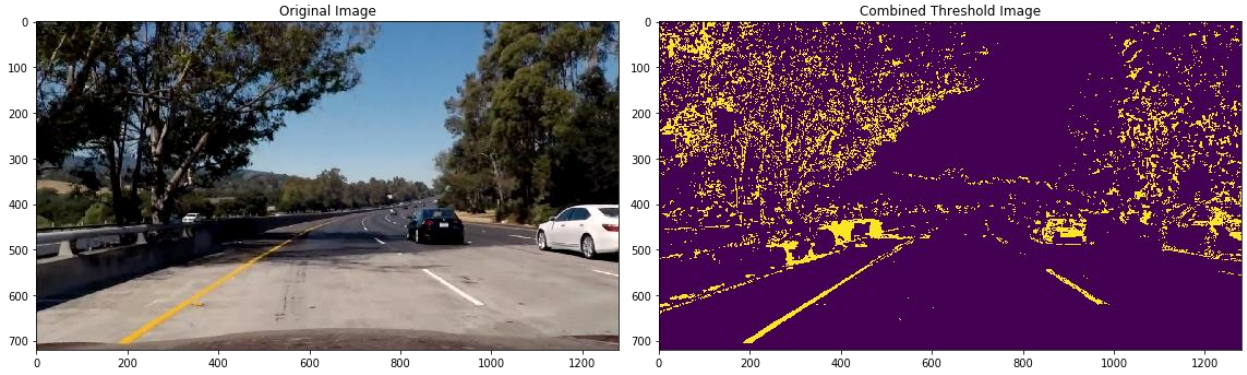




After calculating the directional gradient, I went further to calculate the gradient magnitude [code cell 8] by taking the square root of both `cv2.Sobel()` values on the x and y directions. This allows for the representation of both direction gradients on one image. An example of the above computation is displayed below.

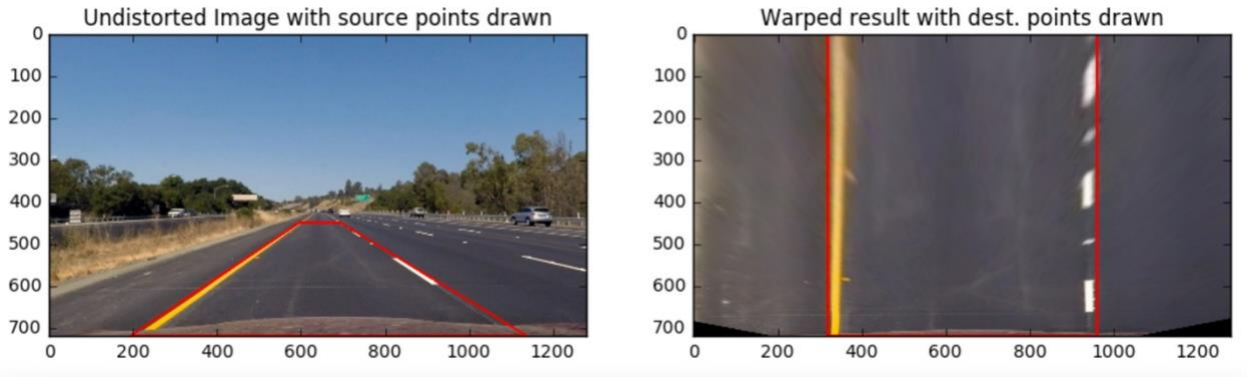


To compute the final combined color threshold, I first convert the image to `HLS` color format and isolate the `S` channel. I then apply thresholding to both the S color channel and the final direction magnitude image [code cell 11] to get the final color threshold lane image as displayed below.



I then applied **perspective transform** to the lane images [code cell 13] by defining source and destination coordinates before applying them to `cv2.getPerspectiveTransform()` to get the transform matrix. I then used the resultant matrix to warp the image using `cv2.warpPerspective()`.

To visualize this process, I drew the source and destination coordinates on a lane image as displayed below.



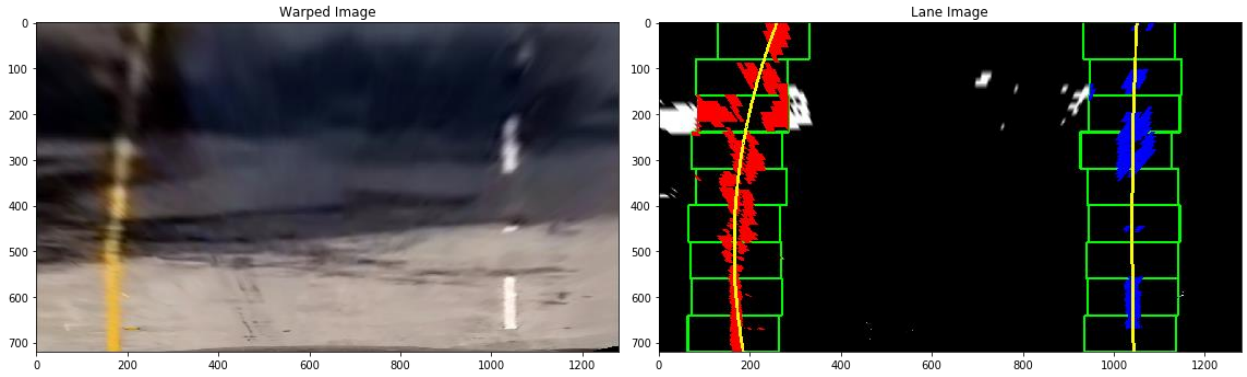
I went on to identify the lane lines in the image by plotting a histogram of the bottom half of the lane image [code cell 15], splitting it into half on the x direction and defining the beginning of both lane lines as the highest points on the histogram [code cell 16].

After that was deduced, I then used the sliding windows algorithm on incrementing y-coordinate regions on the image to find the position of both lane lines in frame [code cell 17].

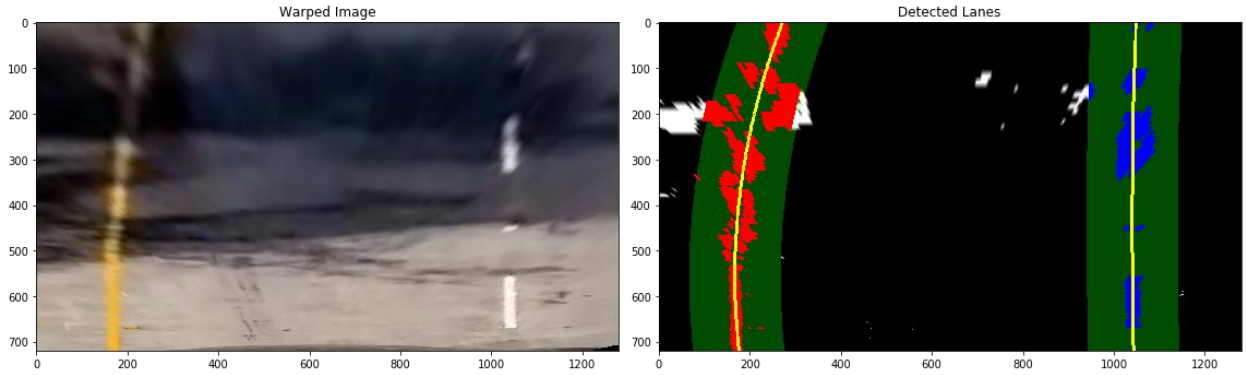
Although this algorithm worked as expected, it was not the most efficient as it searches for the position of the lane lines at every frame. To improve this, I implemented an intermediary step to my algorithm where for every efficient deduction of the lane lines position, I could use a margin on the image to find the next lane line position [code cell 18].

In the same code cell, I fit a second-degree polynomial to the pixel positions that the lane lines are found so as to identify with confidence where the lane lines in the frame are located.

A culmination of the above algorithms is shown below where lane line positions are found on individual positions on incrementing positions of y in the frame.



When smoothening is applied on the resultant lane line positions:

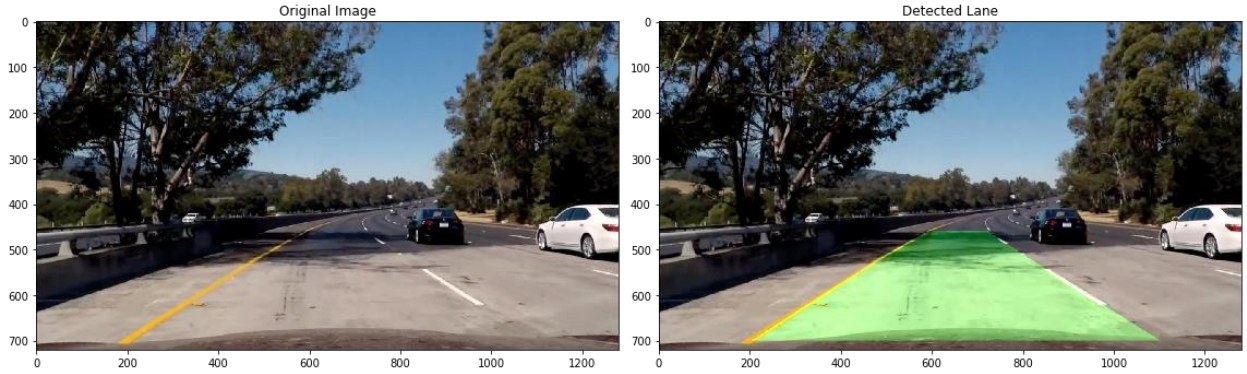


To calculate the radius of curvature [code cell 20], I fit a second order polynomial to each lane line in the image then convert them from pixels to meters.

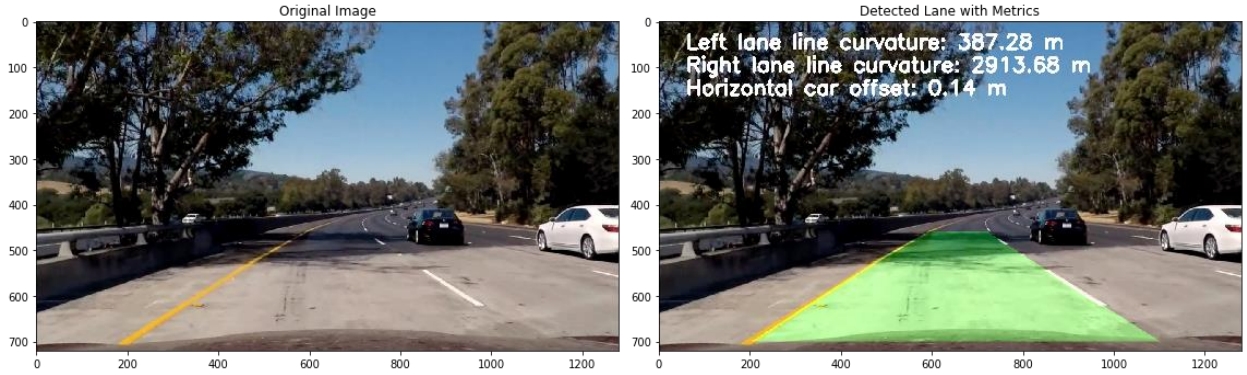
The conversions used were:

- $\boldsymbol{y\ meters\ per\ pixel = \frac{25}{720}}$
- $\boldsymbol{x\ meters\ per\ pixel = 3.\frac{7}{800}}$

When the source and destination points of perspective transform are switched, it allows for inverse perspective transform where the warped image is returned to normal perspective. When this is done on the warped lane image [code cell 24], it results as below.



Metrics are then added to this image and it results to:



Pipeline (video)

Here's a link to my video result

Discussion

The major issue that I faced while building this project is that when the lane lines are too steep or when the shadows are too dark, this pipeline fails.

One way to improve this pipeline in this regard would be to play around with the color thresholding to find the most optimum parameters.