

## 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 distortion correction to raw images.
- Use color transforms, gradients, etc. to create a threshold binary image.
- Apply a perspective transform to rectify binary image (“Bird’s-eye View”)
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and the 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.

Now we will explain each point in details with some supporting images for every functionality:

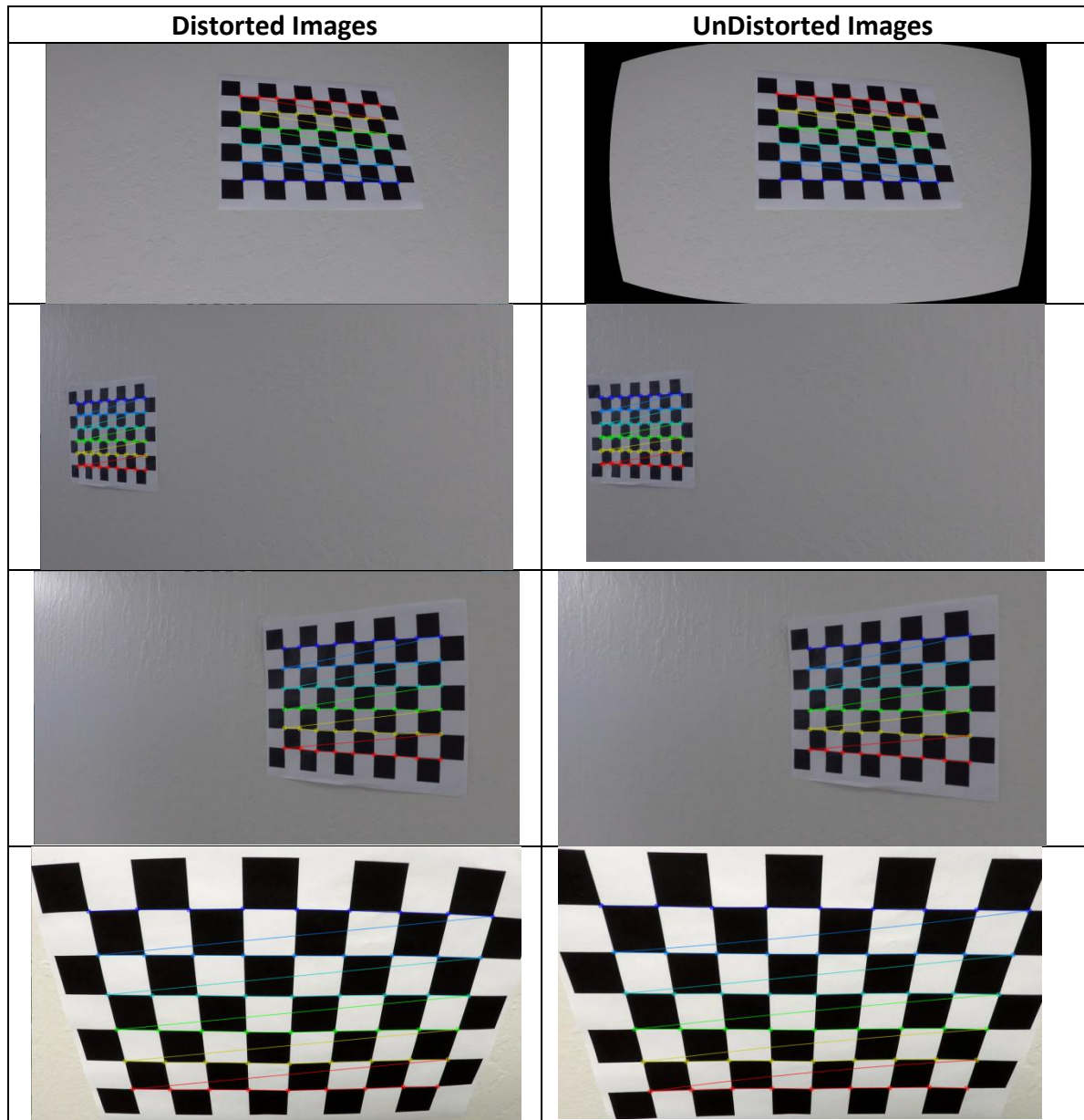
### Camera Calibration

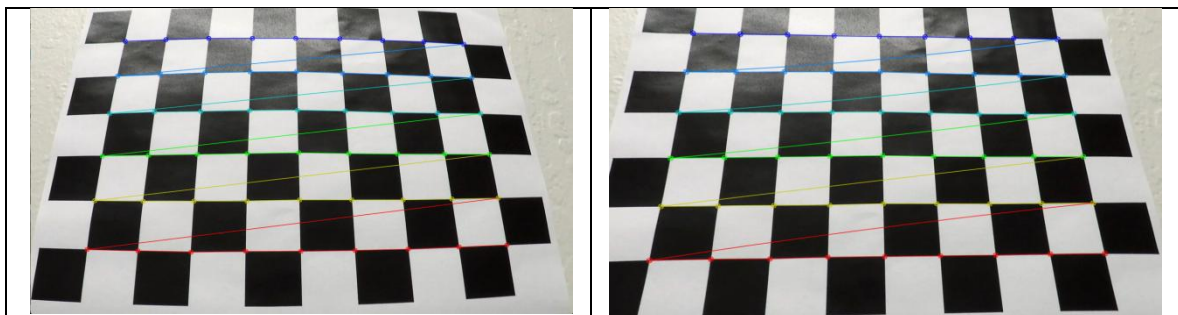
1. **Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?**

The code for this step is contained in (Camera Calibration using pickle to save both mtx, and dist out of the camera calibration) cell in the IPython notebook (Advanced\_Lane\_Finding). It actually depends on (Camera Calibration and undistortion function) cell which contains Camera Calibration function that is used in calculating the Camera matrix in addition to removing the distortion factor.

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 ( $z=0$ ), such that the object points are the same for each calibration image. Thus, **objp** is just a replicated array coordinates, and **obj\_points** will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. **Img\_points** 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 `obj_points` and `img_points` to compute the camera calibration and distortion coefficients using `cv2.calibrateCamera()` function. I applied this distortion correction to the test images using the `cv2.undistort()` function and obtained thus result:





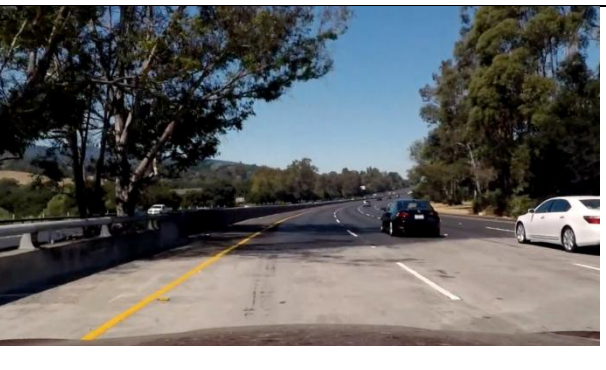
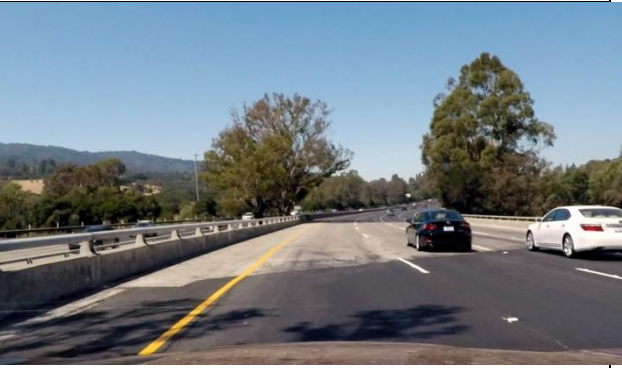
## Pipeline (Single images)

### 1. Has the distortion correction been correctly applied to each image?

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















2. Has a binary Image been created using color transform, gradients or other methods?

I will show now the testing images:

Undistorted images	Binary undistorted images
	
	
	





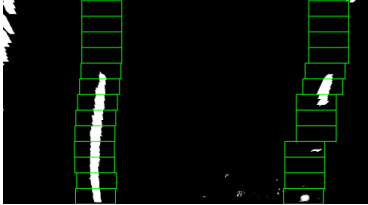


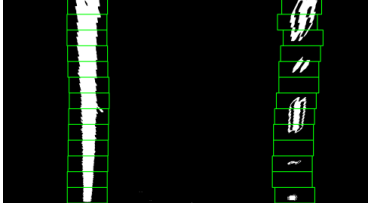


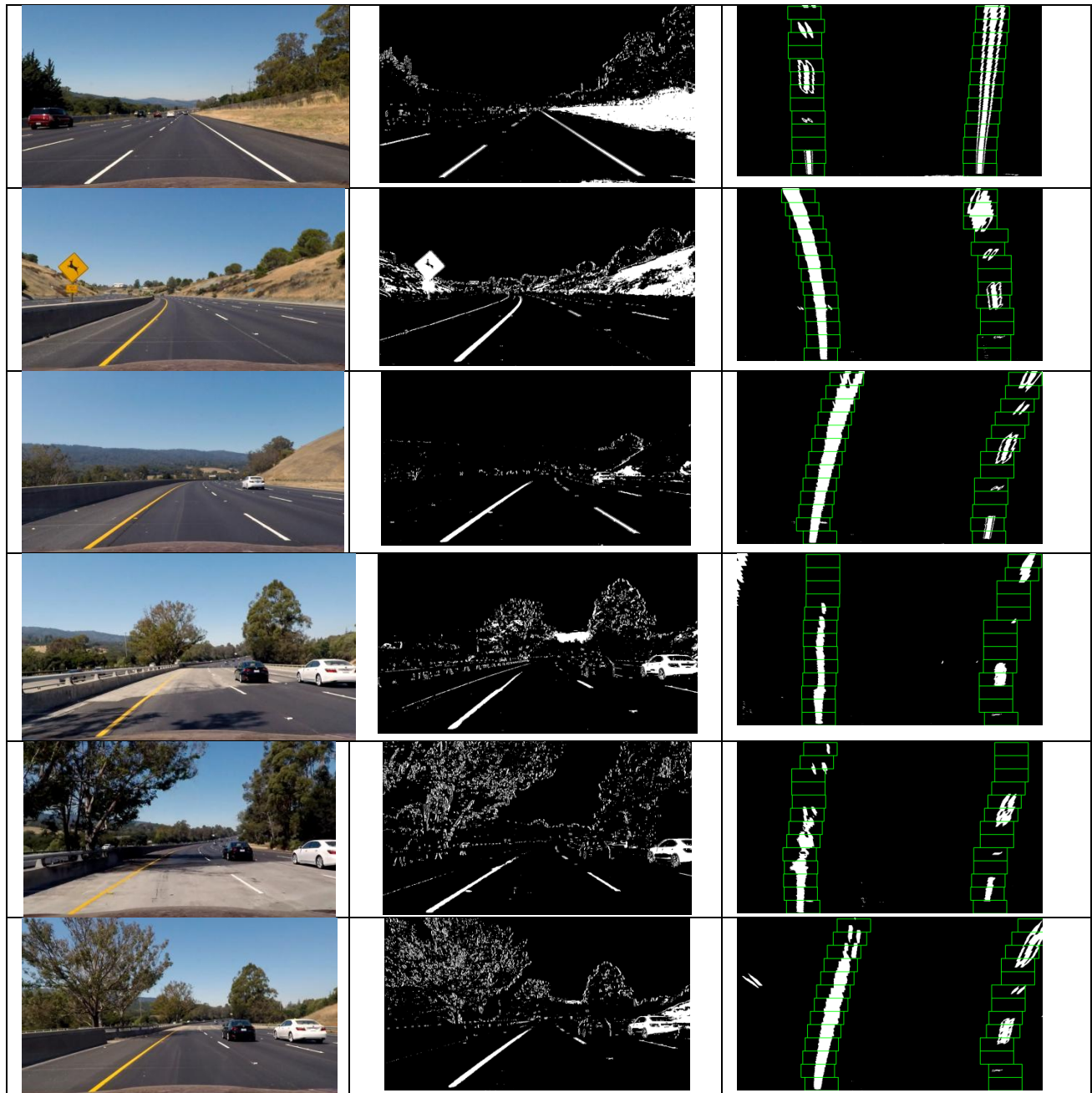
### 3. Has a perspective transform been applied to rectify the image?

The code for my perspective transform includes a function called (`corners_unwrap`) which appears in cell (Prespective transform) in the IPython notebook (Advanced\_Lane\_Finding). This function takes as inputs an image as well as source and destination points. My src and destination points are in the following manner:

Src points	Dst points
(575,460)	(250,0)
(150, 720)	(250, 720)
(1150,720)	(1050,720)
(700, 460)	(1050,0)

I verified that my perspective transform was working well as expected by drawing the test images and its warped counterpart to verify that the lines appear parallel in the warped image.

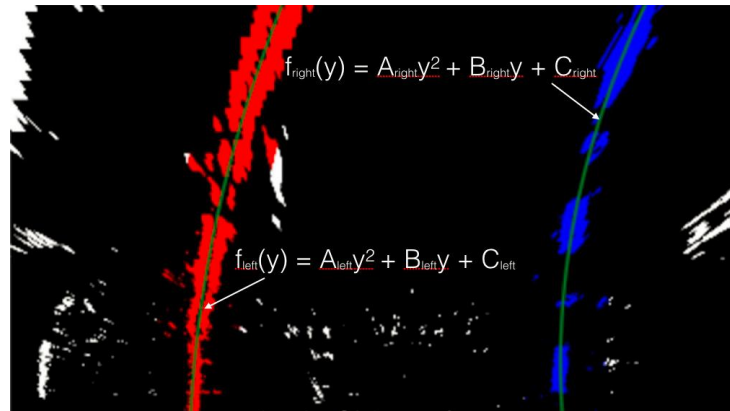
Undistorted images	Binary undistorted images	Warped Binary undistorted images
		
		



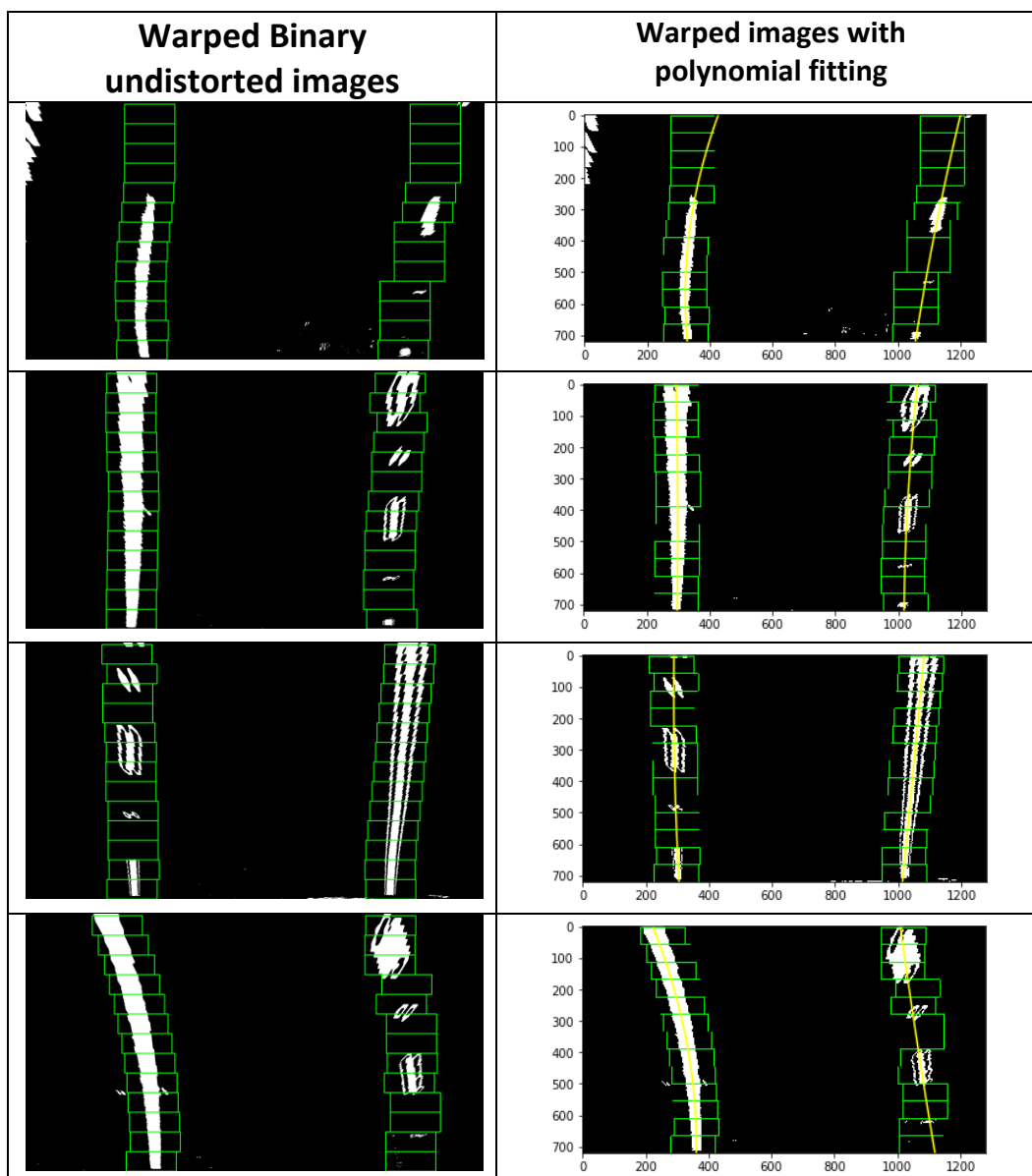
#### 4. Have lane line pixels been identified in the rectified image and fit with a polynomial?

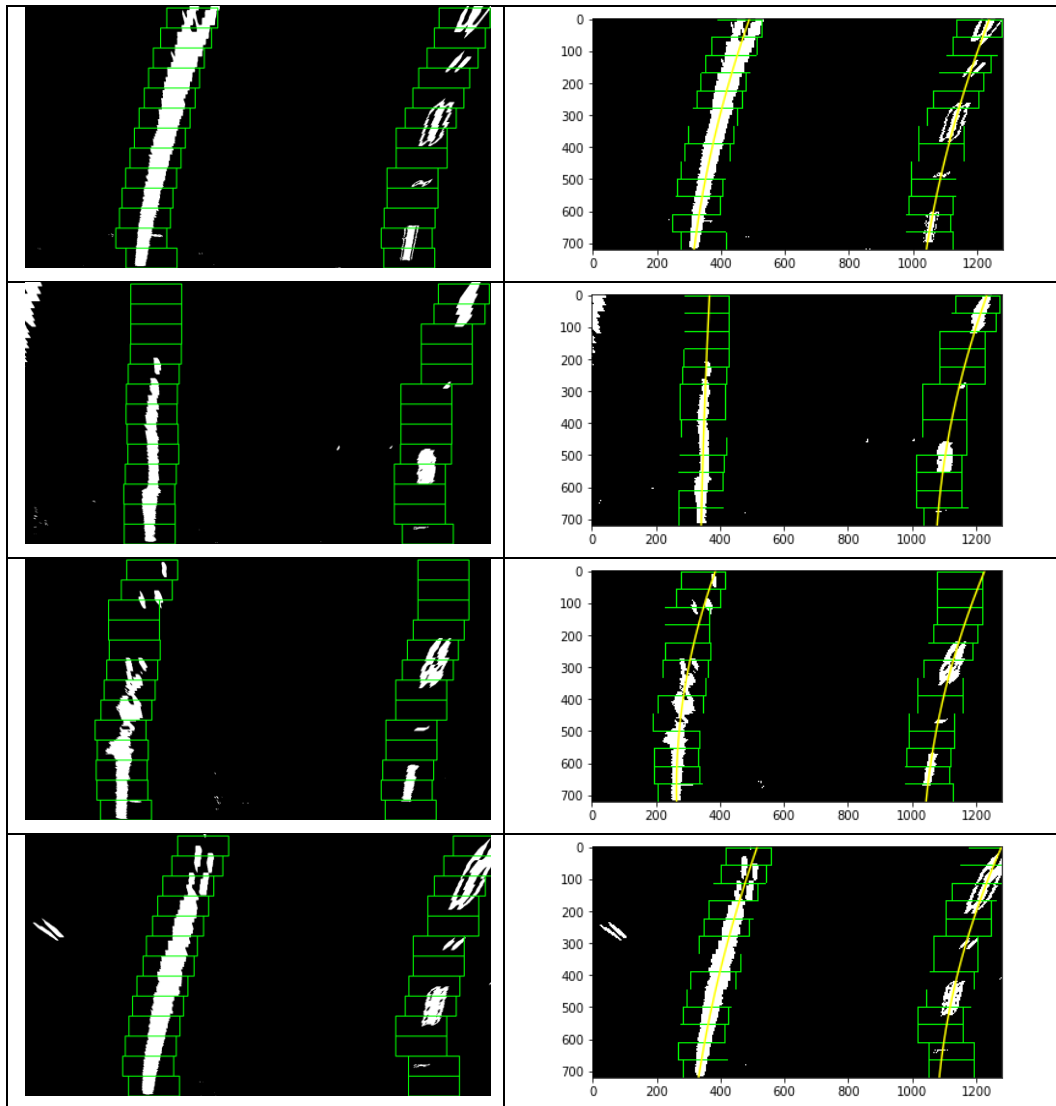
Then I did some other stuff and fit my lane lines with a 2<sup>nd</sup> order polynomial like this:





This stuff leads to have the following images:





**5. Having identified the lane lines, has the radius of curvature of the road been estimated? And the position of the vehicle with respect to center in the lane?**

Yes, Sure I calculated the radius of curvature of the left and the right lanes. In addition to that, I also calculated the position of the vehicle. This information was written over each image in the video attached.



## Pipeline (Video)

1. Does the pipeline established with the test images work to process the video?

Yes, Sure. The pipeline video is called (`project_output_video.mp4`).

## README

1. Has a README file been included that describe in details the steps taken to construct the pipeline, techniques used, areas where improvements could be made?

Yes, this document is considered as the README file.



## Discussion

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 have followed the same techniques described in the Lectures video like using Computer Vision (OpenCV) library for the processing done over the images extracted from the attached video.

The main challenges that faced me was:

- 1- Tuning of using Gradient and Color thresholds over the images. I have tried many thresholds. The challenges can be formulated in the bright and shadow parts in the video. Some applied thresholds makes the shadow of the trees and the cars to be in a white color in a gray scale image which causing many problems for extracting the lanes lines. I have solved these challenges using masks for Yellow, and White colors.

The **White mask** extracted from the RGB color space image within the range (200,200,200) to (255,255,255) which detects the white lanes perfectly.

The **Yellow mask** extracted from the HSV color space image within the range (0,100,100) to (80,255,255) which detects the yellow lanes perfectly as shown in the previous screenshots.

- 2- Applying the Sanity check if there is no lanes are detected, so every lane detected and fitted, I saved the parameters in order to skip using the sliding window technique.

Further improvements:

- 1- Although I used thresholds for the white and the yellow lanes, but still it is not the efficient way as I tried to use it in the challenge video, but it didn't succeed. This is due to the appearance of another bright lane beside the corrected lane we have in the video, which make a little bit confusion for the algorithm. This can be solved using more thresholds and need some tuning to ensure the robustness of the algorithm.
- 2- Applying more and more sanity checks could improve the tracking quality for the lanes especially for the wrong detected lanes.