Advanced Lane Finding Project

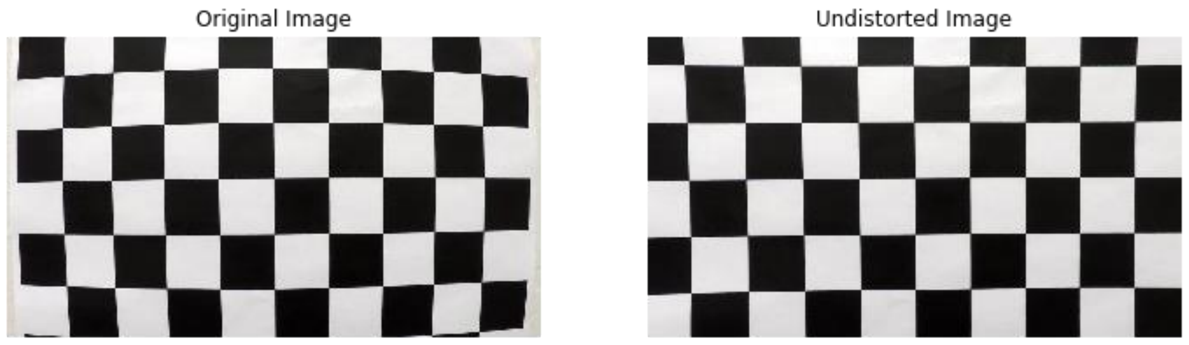
The goals / steps of this project are the following:

- 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 step is contained in the Section 1 of the IPython notebook named "Advanced Lane Detection". I started 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 a test image using the cv2.undistort() function and obtained this result:



Pipeline (single images)

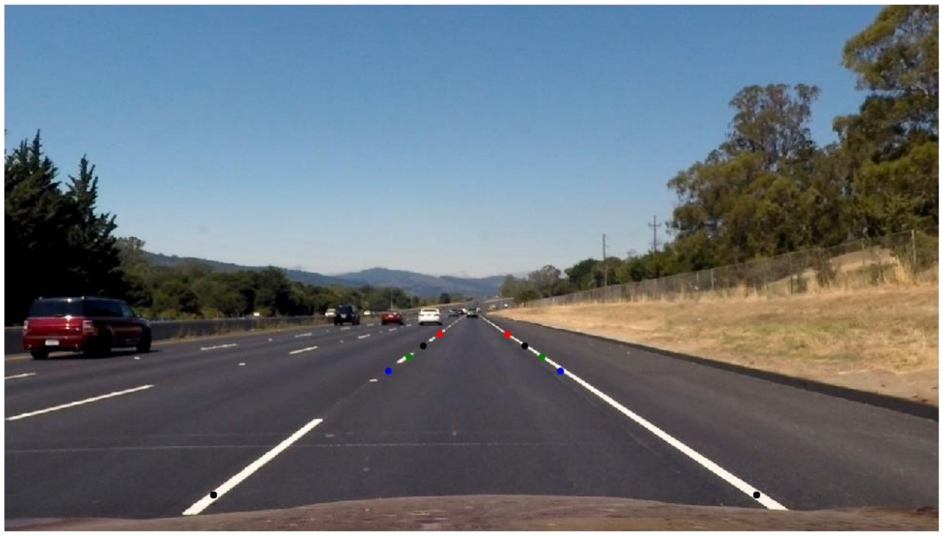
The following steps describe the pipeline I developed to process each image in the video.

1- Distortion Correction

Each road image is corrected for distortion using the procedure described above.

2- Perspective Transform

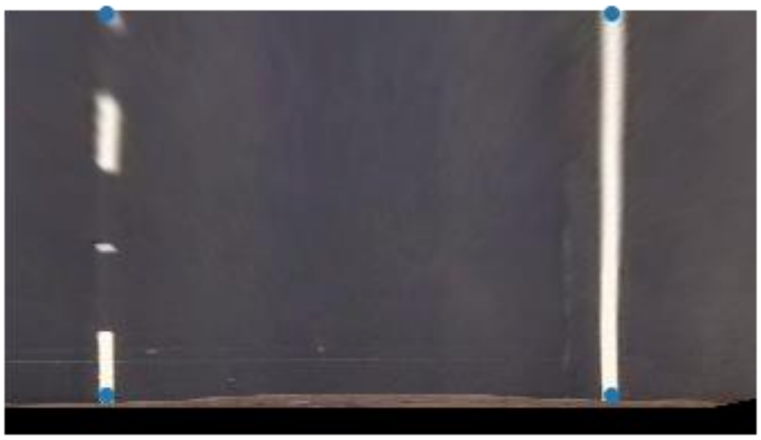
To detect the road lines and calculate the radius of the curvature, it’s better to transform the images to the bird-eye view. In order to obtain the transform coefficients, we need to pick four sample source points and four destination points on an image. I chose one of the road images with straight lines and picked four points on the left and right lane lines as the source points. In order to obtain the optimum four source points, I moved the higher two points (the two points farther from the car), placed them at several locations along the lane lines (shown with different colors) and evaluated the result of transformation. The best performance was obtained for points shown with black color.



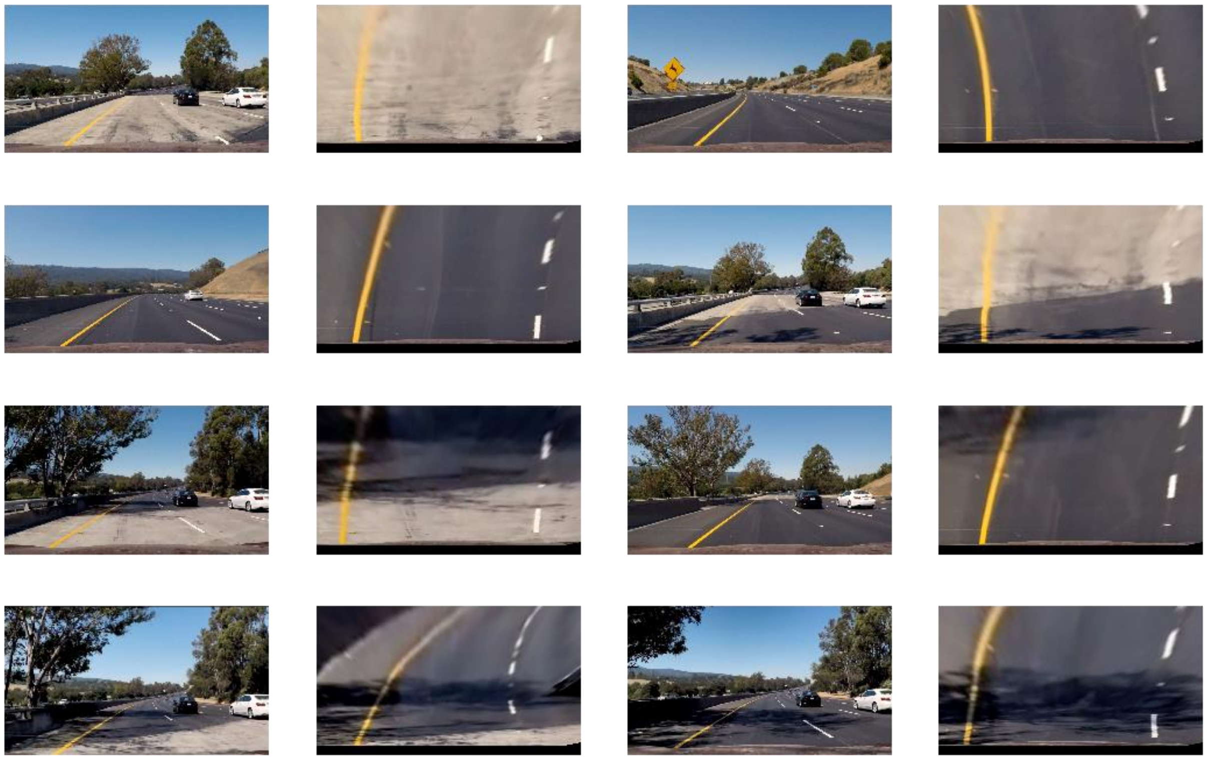
The (x,y) coordinates of these four source points shown with the black color are:

(x,y) = (572,465), (712,465), (285,670), and (1030,670)

I then chose four points (170,0), (1030,0), (170,650), and (1030,650) as my destination points and calculated the transform matrix M based on that. The image below shows the same image after the transformation with the fours destination point on it.



The following figure shows the result of perspective transformation for eight sample test images.

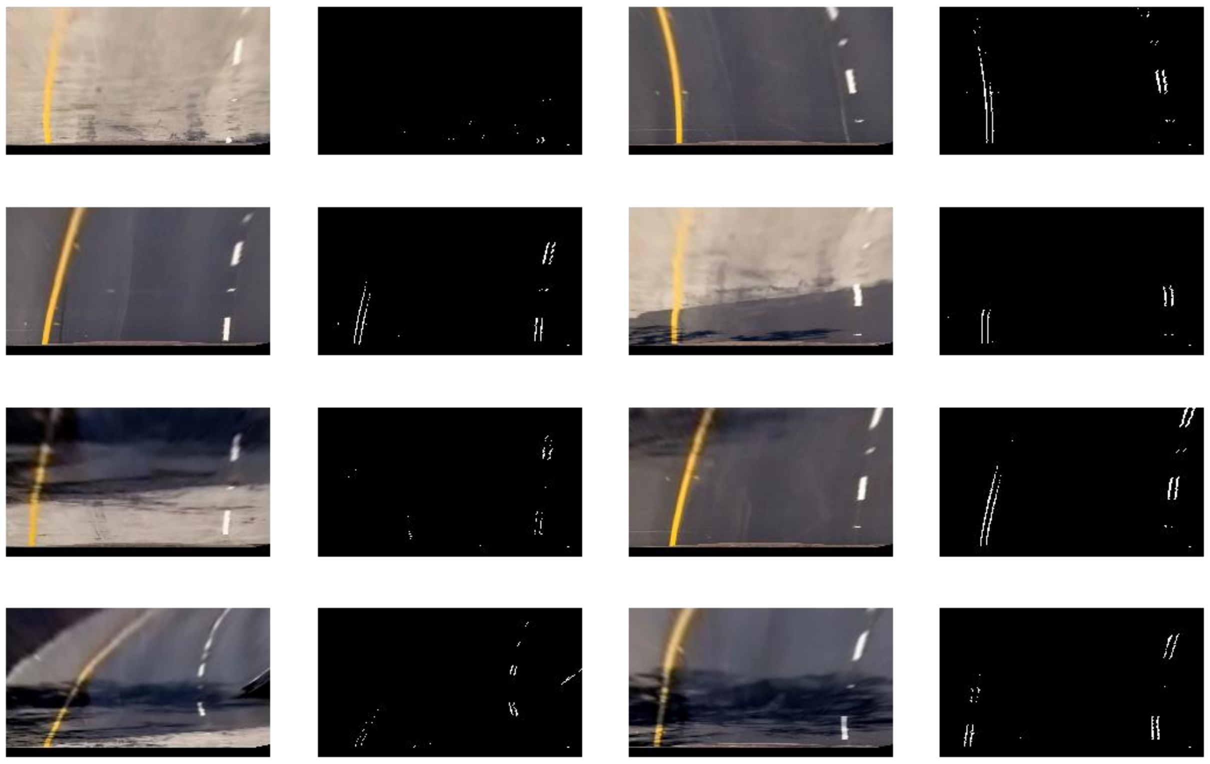


3- Lane lines Detection

Road images contain redundant objects such as trees and sky. In order to detect the lane lines properly, we need to remove these objects from the image first. One way to remove these objects is using masks like what we did in the first project. However, as can be seen above, warping the images using perspective transform also takes care of this. Here, I decided to warp the images first and perform the lane detection after that, instead of using a mask. One advantage of doing the transformation first is that since the lane lines tend to be vertical in the bird-eye view, we can only rely on the X orientation when thresholding the gradient and ignore the other directions.

3.1. X Sobel Thresholding

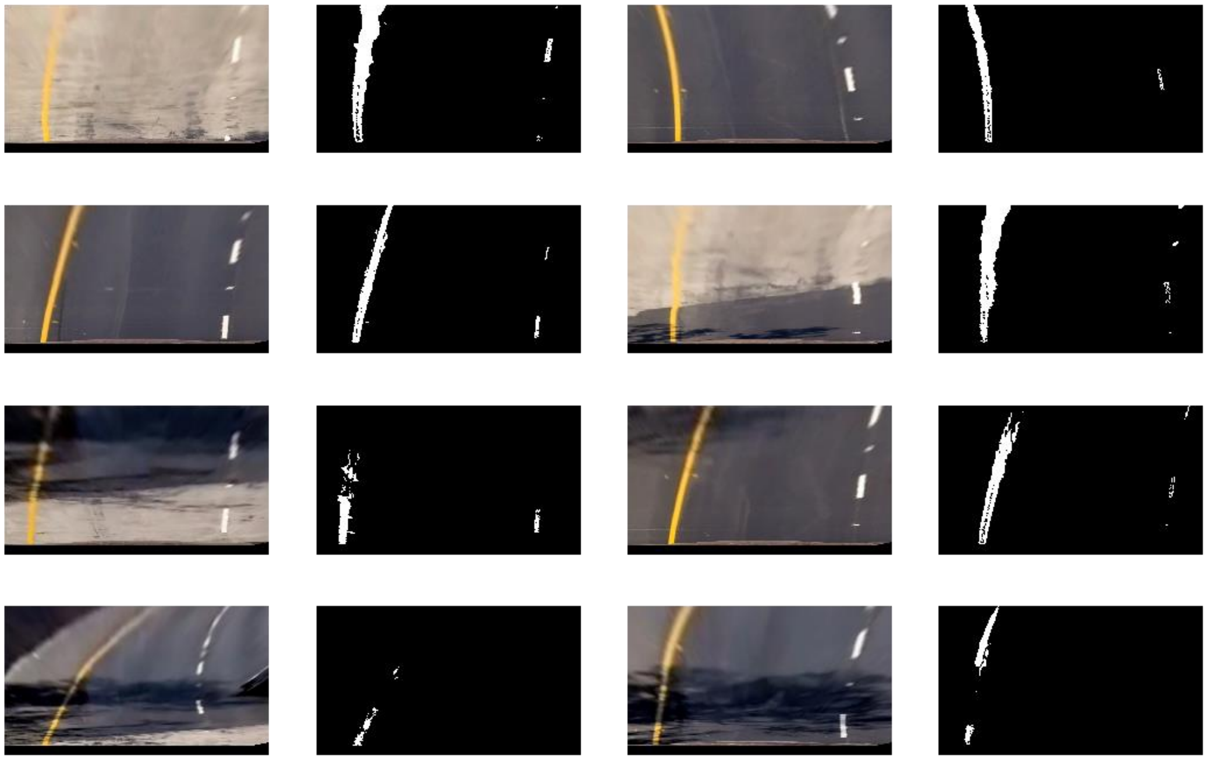
After transforming the images, we can now use the sobel transform in the X orientation to detect the vertical lines. Using threshold values of (50,255), the result of thresholding on the on test images would look like the following.

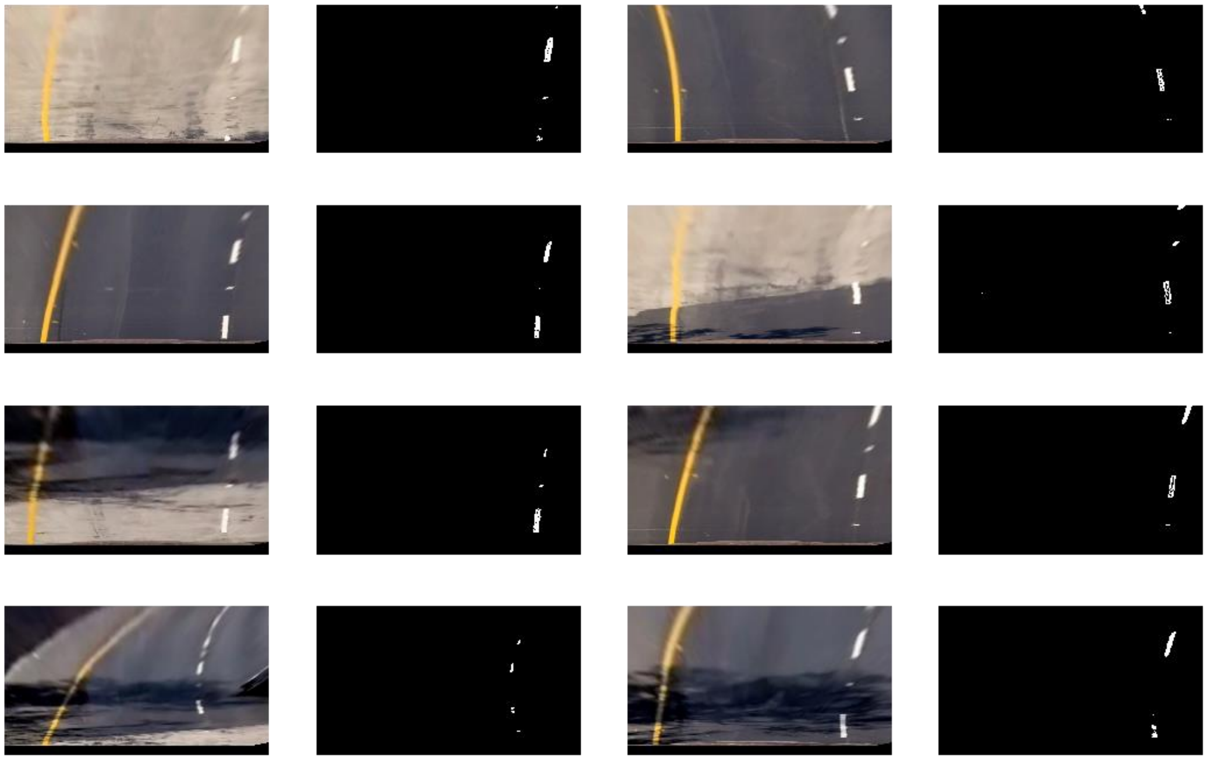


3.2. Color Thresholding

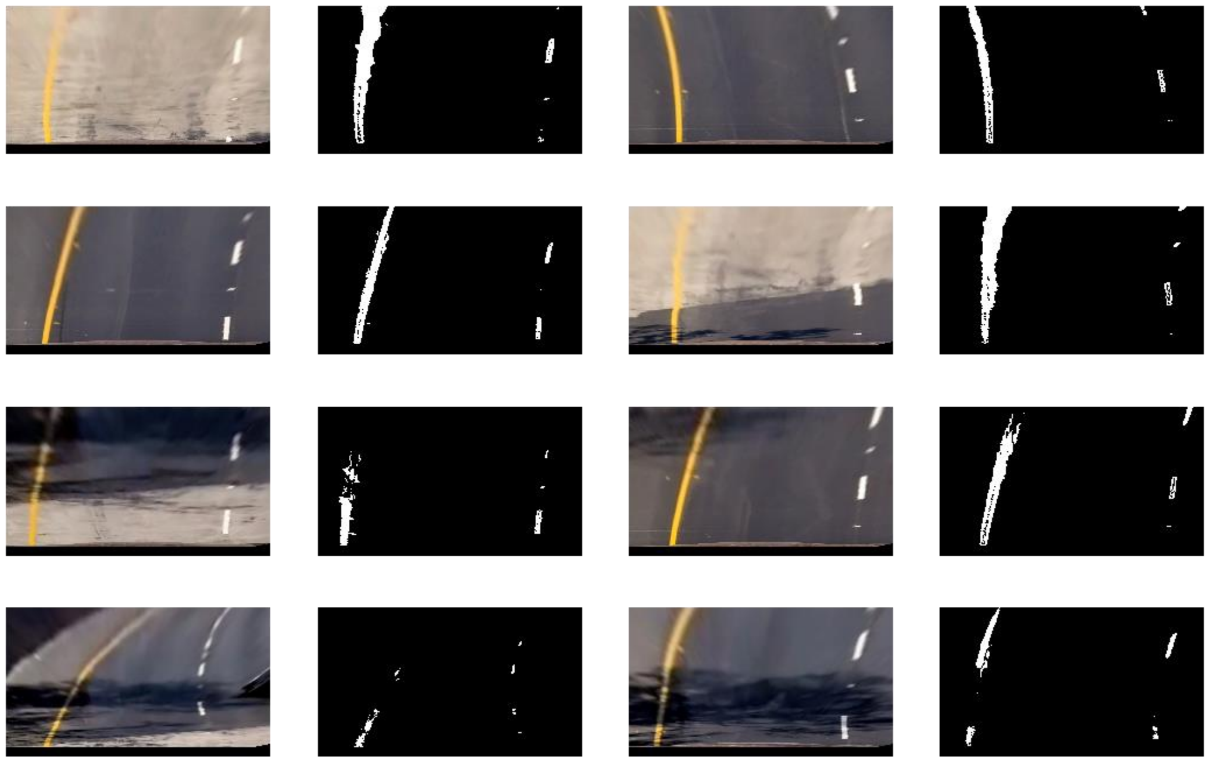
The lane lines can also be detected by applying thresholds to colors in the image. I initially tried to convert the color space to HLS, separate H, L, S channels and apply thresholding on each individual channel. As discussed in the course, the S channel shows a better performance in detecting and isolating the lines from other objects. However, I found that the S channel does not always perform well especially in the presence of shadows.

Since the left and right lines have either white or yellow color, I thought it is a good idea to isolate each one individually based on its color. Hence, I used the threshold values h= (10,30), l_thresh = (50,255), s_thresh = (100,255) to isolate the yellow color and h_thresh = (0,255), l_thresh = (210,255), s_thresh = (0,255) to isolate the white color lines in the image. The result of isolating yellow and white colors is shown in the following figures.


Isolating yellow lines in the image


Isolating white lines in the image

As can be seen, with applying two separate color thresholds, we can identify left and right lines separately. Now, if we combine these two binary images, we will have the following results.



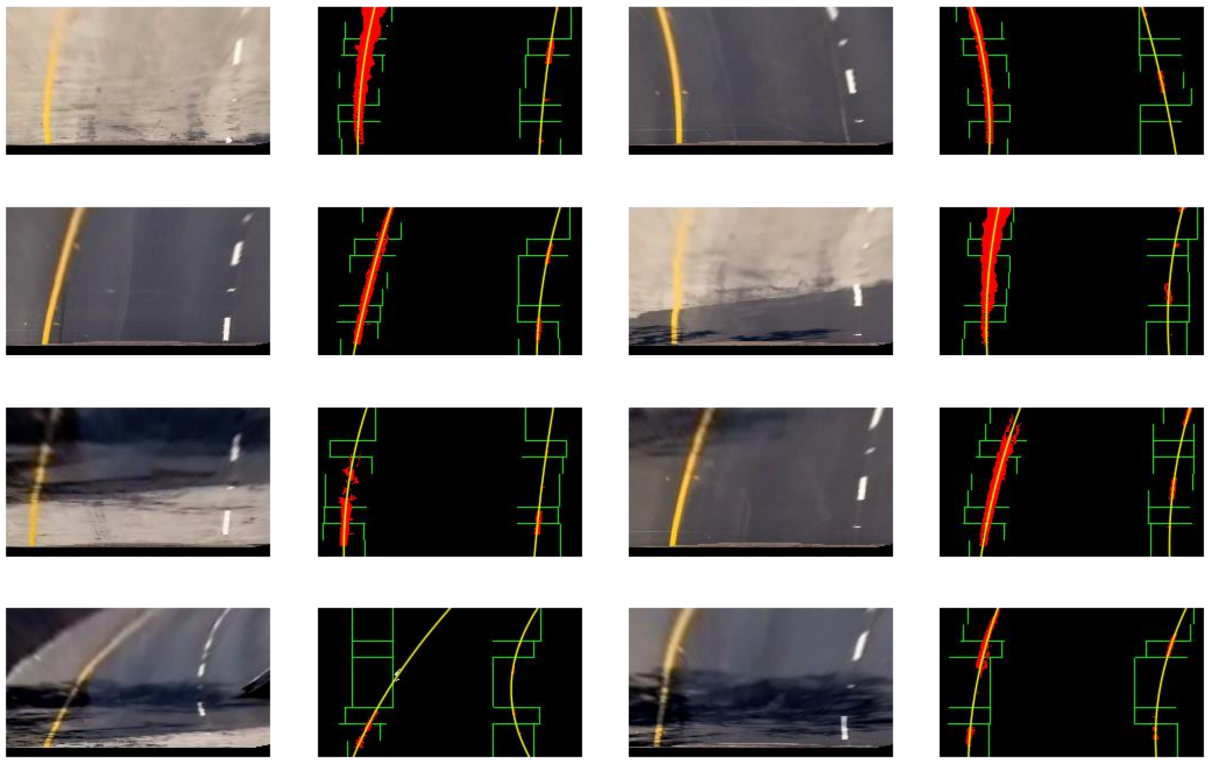
It can be seen that the lines are perfectly identified using this approach even in the presence of shadows and road light and texture variations.

4- Finding the Lines

After applying calibration, perspective transform, and thresholding to a road image, we should have a binary image where the lane lines stand out clearly. However, we still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

I used the code provided in the course and wrote a function `find_lines()` to do that. This function takes a perspective transformed binary image, identifies the left and right lines based on the density of points along the x axis using a sliding window, and fits a second order polynomial to the detected lines.

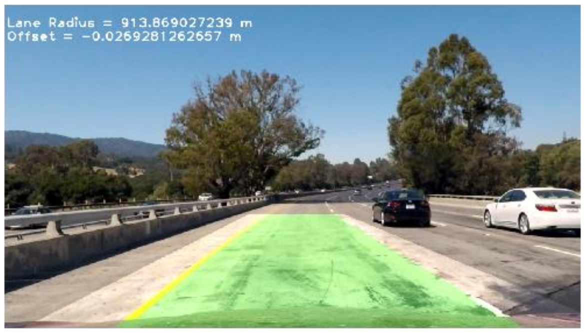
The figure below shows the result of applying this function on each test image. The detected line points are shown with red color and the polynomial fits to the points are shown with yellow color.



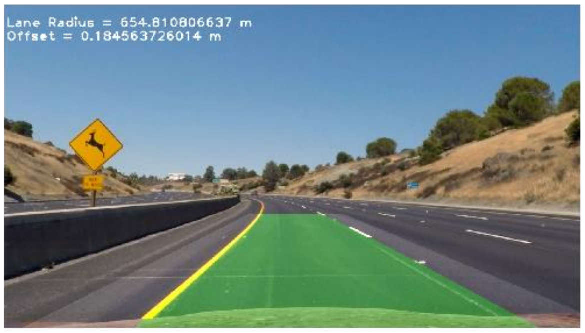
Now that we have identified the lines and fit the polynomial to them, we can transform the detected lines to the original view and overlay it on the original undistorted image. Using the coded provided in the course, I calculated the radius of the left and right lines in each image and printed the minimum of two radiuses on the final image as the radius of the lane.

I wrote a function named `process_image()` that performs the entire described pipeline procedure and outputs the image along with the detected lines in the original view. The result of applying this function on test images is shown below.

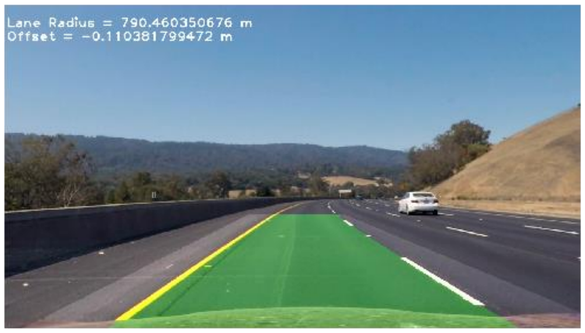
1



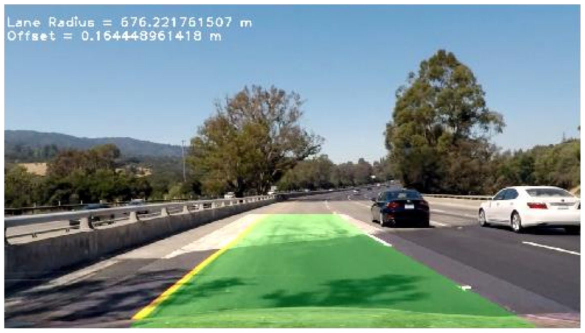
2



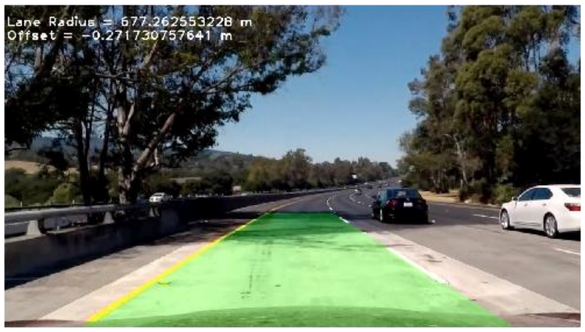
3



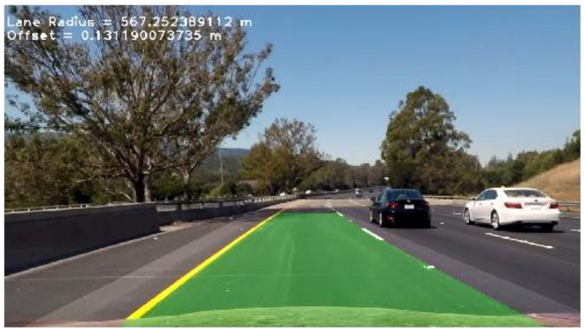
4



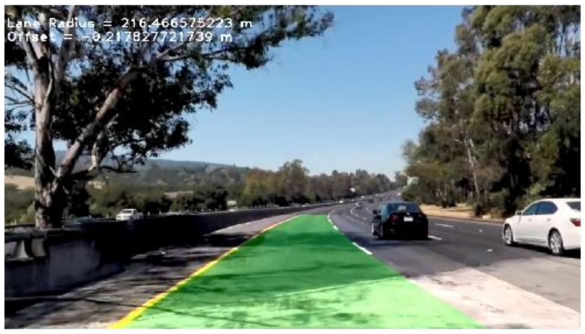
5



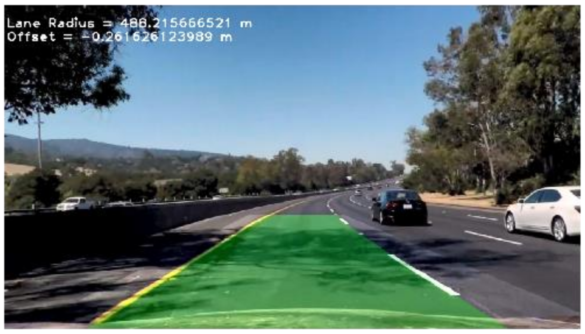
6



7



8



5- Create a Video

Finally, I used `process_image()` function to process each image frame in the video and create a video with detected lines as well as radius and offset information printed on it. The output video is named “project_video_output.mp4” and is available in the project folder.