PROJECT 2: ADVANCED LANE DETECTION

README of Project2.ipynb

The project focuses on two main things. One a pipeline to test project video and next a detailed readme file explaining the codes.

The project mainly focuses on 8 goals:

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
2. Apply a distortion correction to raw images.
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform to rectify binary image ("birds-eye view").
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Warp the detected lane boundaries back onto the original image.
8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

This Readme file explains the codes used for implementation of above mentioned goals which is saved as Project2.ipynb

GOAL 1: Compute the camera calibration matrix and distortion coefficients given a set of chessboard images

This step is useful for the calibration of the camera

Chess board corners are found using **findChessboardCorners()** command. Corners are draws to the image using **drawChessboardCorners()** command. The chess board corners are used to form image points which are required for camera calibration.

An output image with corners marked is stored in **output_images** folder with name **ChessCorners.jpg,** which is shown in figure below:

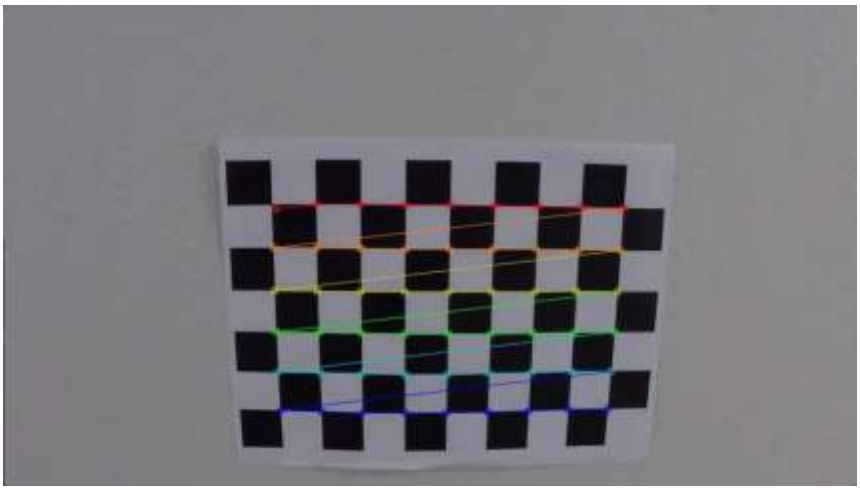

Figure 1: Chessboard corners

Code details:

Glob function is used to read all the chessboard images and for all the images corners are obtained, we append it as image points. The image points along with the generated object points are used for obtaining camera matrix and distortion coefficients using **cv2.calibrateCamera()** function. These values are in turn used by **cv2.undistort()** command for obtaining the undistorted image.

The undistorted chess board image obtained is stored in the **output_images** folder with name **test_undist_result.jpg**. Using subplot the distorted image and undistorted image displayed in one row and two columns as shown in figure 2.

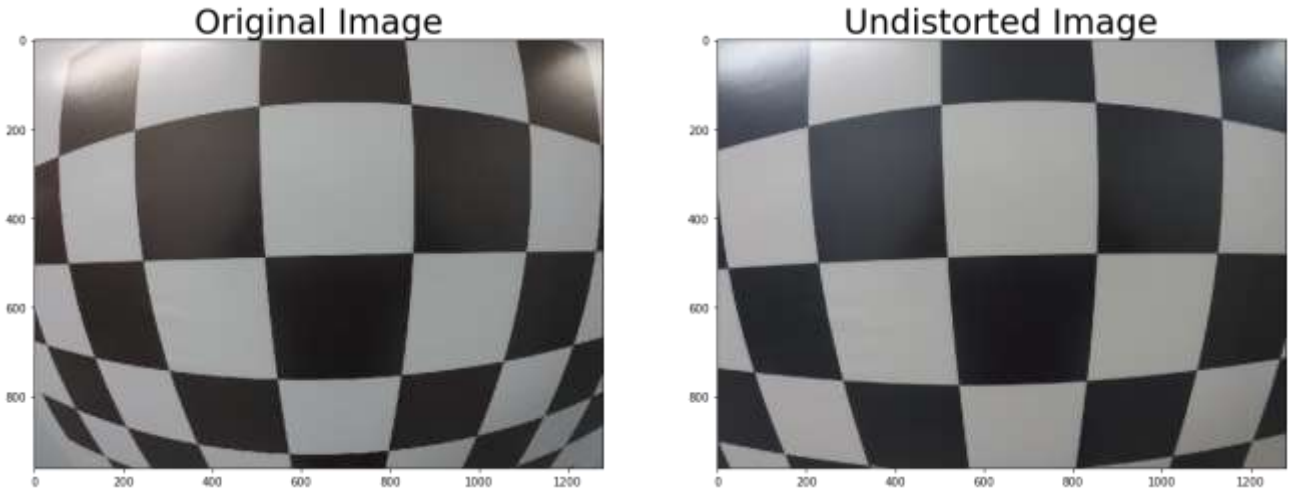

Figure 2

Save the image calibration results mtx and dist as pickle file

The camera matrix and distortion matrix will be useful to calibrate camera images in coming examples. Hence, it is stored in a pickle file for later use in the main folder with the name **wide_dist_pickle.p**.

GOAL 2: Apply a distortion correction to raw images

We have already used the **cv2.undistort()** function to obtain an undistorted image of chessboard. Now we will apply the same for testing how it works on our test images.

The above algorithm is tested on **test1.jpg** given in the **test_images** folder. The result is stored as **test1_undist_result.jpg** in **output_images** folder. The result can also be viewed in fugure 3.

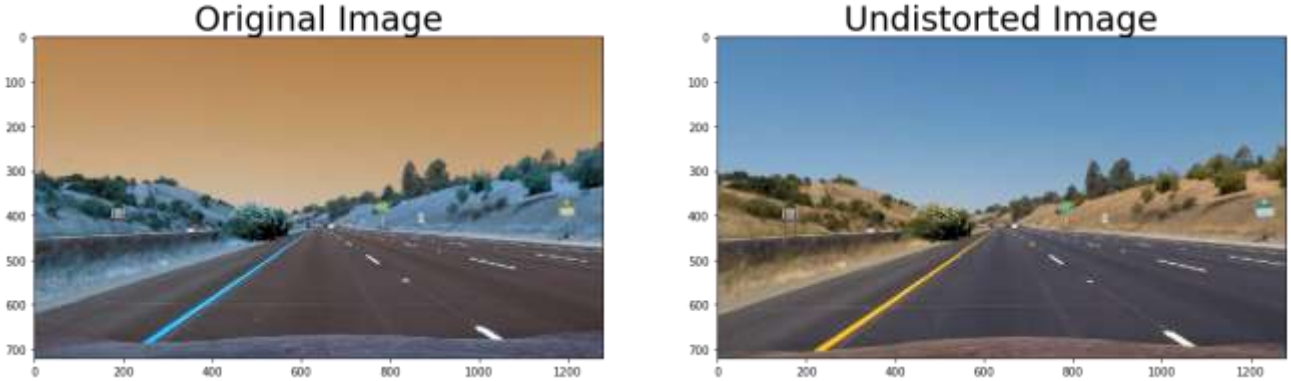

Figure 3

GOAL 3: Use color transforms, gradients, etc., to create a thresholded binary image

Steps:

1. Use the pickle file parameters to undistort the image aby applying cv2.undistort() function
2. Sobel_thresh() function finds both sobel transform in both x gradient and y gradient
3. Color_threshold(): Convert the image to HLS, LAB and HSV color spaces and S, B and V channels are extracted for thresholding
4. Both binary images are combined to form new binary image named combined_binary

The result of few test images is saved in folder **output_images/combined_binary**. The name of the output image is same as the test images. The output for **straight_lines1.jpg** is shown in figure 4.

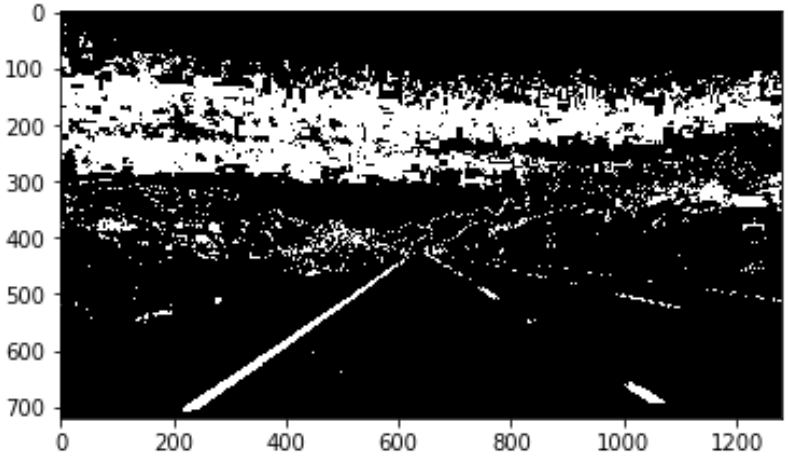

Figure 4

GOAL 4: Apply a perspective transform to rectify binary image ("birds-eye view")

Before applying perspective transform, region of interest is extracted using the functions used in project one. The masked image obtained in this stage is shown in figure 5, which is also saved in **output_images** folder with name **roi_straight_lines1**.

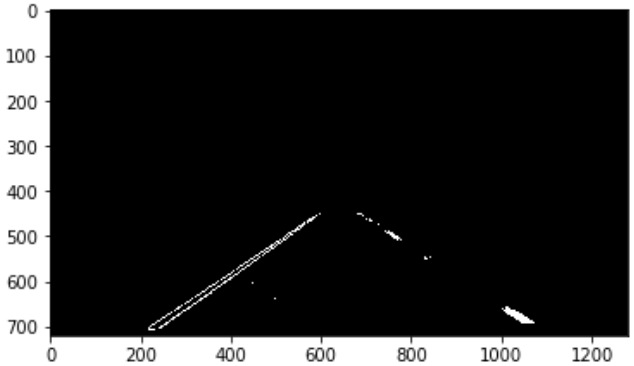

Figure 5: Region of interest of straight_lines1.jpg

Perspective transform is performed on the masked image to obtain a bird's eye view of the road. For this, points on the road image is mapped to desired locations on the image with the help of 2 functions **cv2.getPerspectiveTransform()** and **cv2.warpPerspective()**. First

function takes the source points and destination points to obtain a transformation coefficient, which is used by the later function to produce a warped image. I used histogram of the warped image to find better (hopefully) source points, as for the image considered, the warped image must have straight parallel lines which produces 2 steep peaks for histogram. The warped image obtained for **straight_lines1.jpg** is shown in figure 6. See **output_images/ warped_straight_lines1.png.**

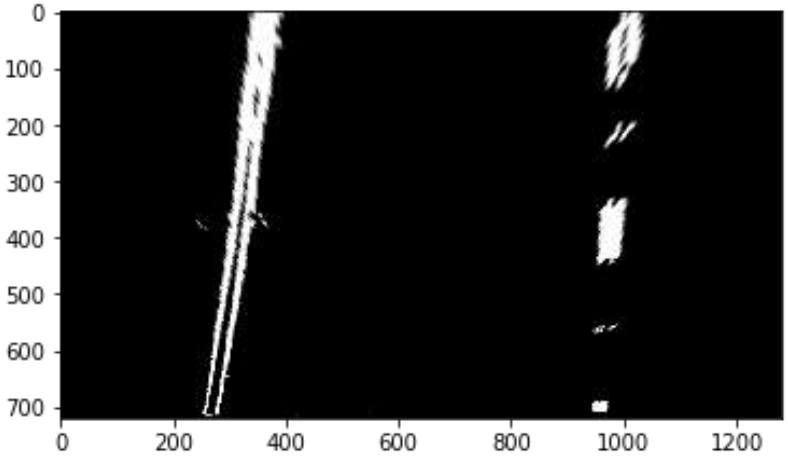

Figure 6: bird's eye view of straight_lines1.jpg

GOAL 5: Detect lane pixels and fit to find the lane boundary

Sliding Windows Method:

This level consists of many operations:

Finding mid-point: Only the below half of the image is considered for this step. Sum of all the white pixels are taken in the second half of the image using the following code: **np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0).** As we can qualitatively asses, the maximum peak of the histogram will be at the position where the lines are present. The mid-point, left base and right base of the lanes were found using the histogram info**.**

Finding lane pixels( ): Here we are trying to find the pixel coordinates of lanes using a sliding window. Window has height of y-axis/9 and width of 200 pixels. First we define; the window coordinates in pixel space using the following code (Here margin is 100):

win_y_low = binary_warped.shape[0] - (window+1)*window_height
win_y_high = binary_warped.shape[0] - window*window_height
win_xleft_low = leftx_current - margin
win_xleft_high = leftx_current + margin
win_xright_low = rightx_current - margin
win_xright_high = rightx_current + margin

Subsequently, we try to find the non-zero pixel indices inside the window by comparing the pixels inside the window and the non-zero pixel locations found along x and y axis. These indices are later used for finding the line. Also, the box orientation is changes if the number of non-zero pixels located inside the window is below a certain threshold value. In such cases, the window orientation is changed to the mean position. Now, inside the box we have left and right indices of non-zero pixels. We extract these to obtain left and right pixel lane pixel positions.

Fit Polynomial ( ): We use the x and y coordinate information of left and right non-zero pixels to fit a second order polynomial using **np.polyfit().** The left and right lane regions are given 2 colors for

visualization. The result of the image is shown in figure 7, which is also stored in **output_images** as **line_fit_straight_lines1.png**.

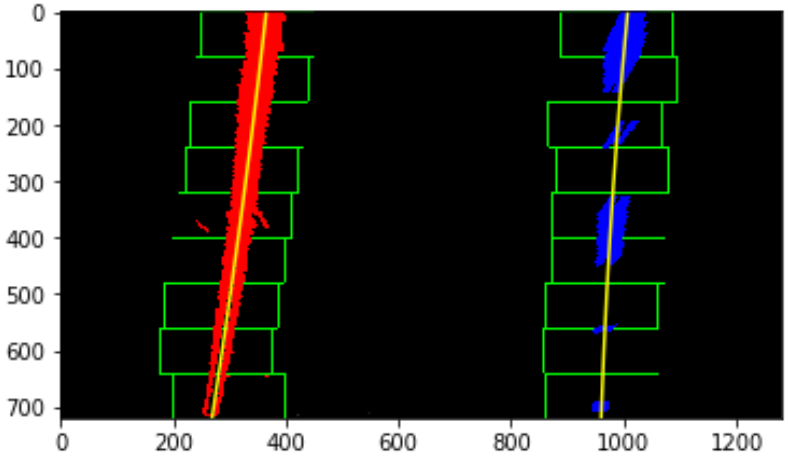

Figure 7

After this we make a blank canvas of the same size as binary_warped image and plot the left and right line on to it using **cv2.polylines( ).** We then use the **cv2.fillpoly()** command for filling the region between the lines with a color. Essentially, now we have an area corresponding to the road, between the lanes, filled with color and lines corresponding to the lane area marked by a line as shown in figure 8. See **output_images/lane_filled.png**.

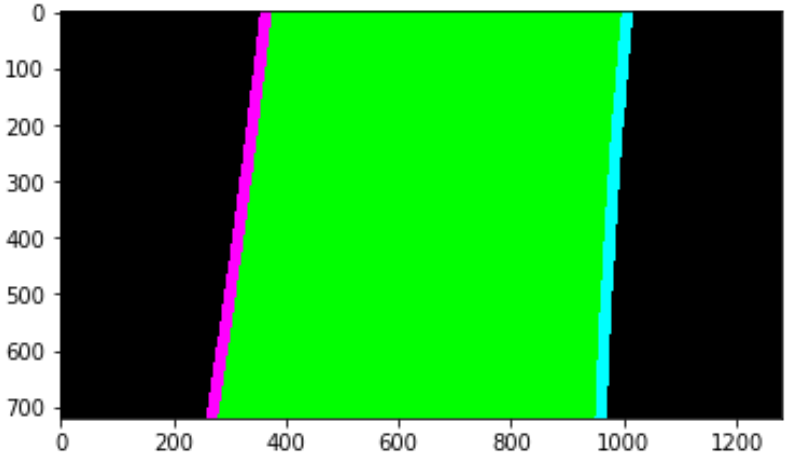

Figure 8

GOAL 7: Determine the curvature of the lane and vehicle position with respect to center

Making use of the polynomial fit data of left and right lines, curve radius of left and right lanes are calculates using the curve equation. We convert the fit data, which is represented by pixel position, to meters using the convention: y-axis=30/720 meters per pixel and x-axis: 3.7/700 meters per pixel. We substitute these values to curve equation to obtain the radius of curvature, whose average gives the radius of curvature of road.

We also need to find the offset of vehicle form the centre of the lane. For this we assume the image center is where the camera is placed. We can calculate the centre of the lane by using the equation **lane_center = ((right_line position – left_line position)/2) + left_line position.** The offset of car position is the difference of centre of image and the lane centre.

GOAL 7: Warp the detected lane boundaries back onto the original image

The result obtained in figure 8 is now warped back to the real perspective using the same commands **cv2.getPerspectiveTransform()** and **cv2.warpPerspective().** In this context we interchange the source and destination points to get the inverse transformation coefficient, which can be used to inverse-transform the image back to the real image perspective. The result is as shown in figure 9. See **output_images/ inverse_perspective.png**.

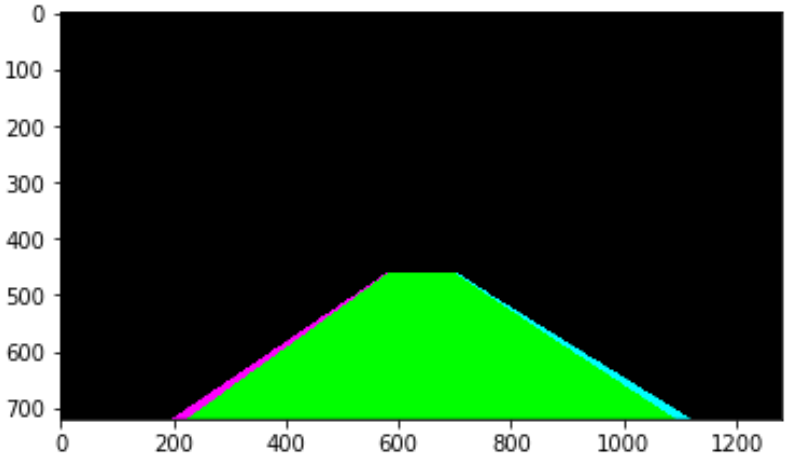

Figure 9

GOAL 8: Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position

In this section, we super impose the result in figure 9 with the original image using **cv2.addWeighted( )**. We make use of **cv2.putText()** to write the numerical estimation of lane curvature and vehicle position over the image. The resulting image for all the test images are shown below.

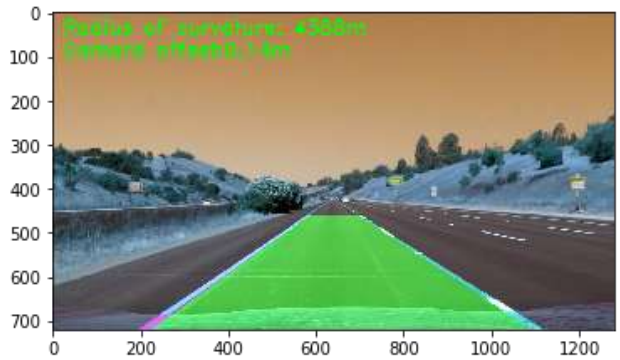

Figure 10: Result of straight_lines1.jpg

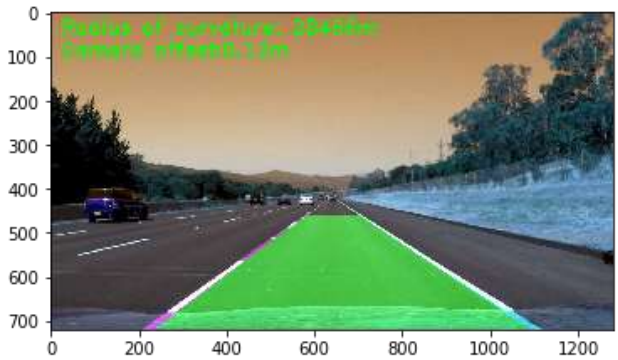

Figure 11: Result of straight_lines2.jpg

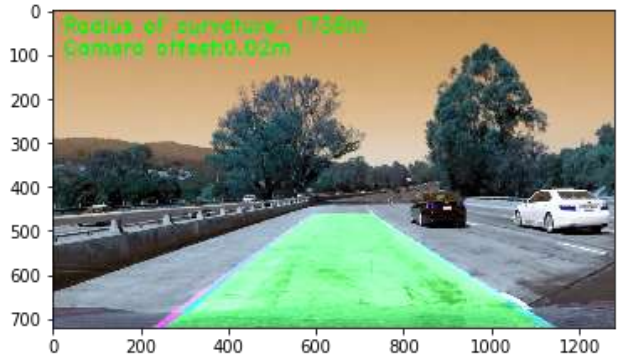

Figure 12: Result of test1.jpg

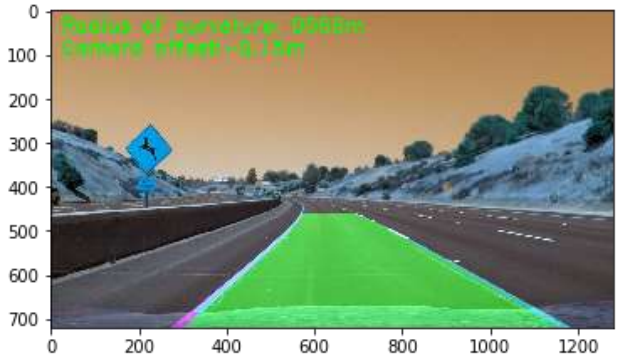

Figure 13: Result of test2.jpg

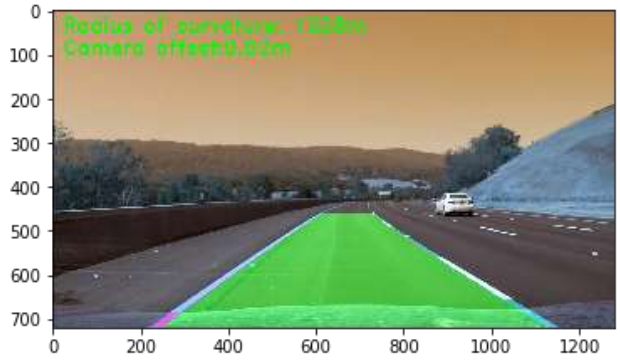

Figure 14: Result of test3.jpg

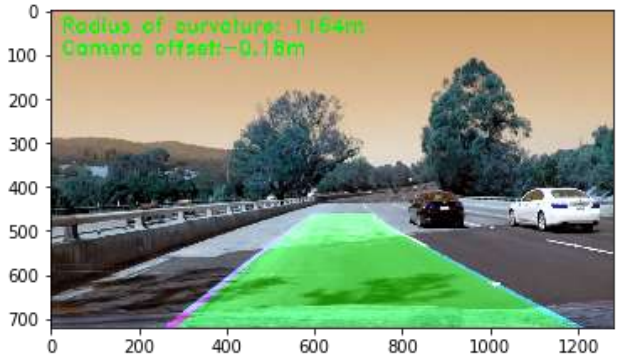

Figure 15: Result of test4.jpg

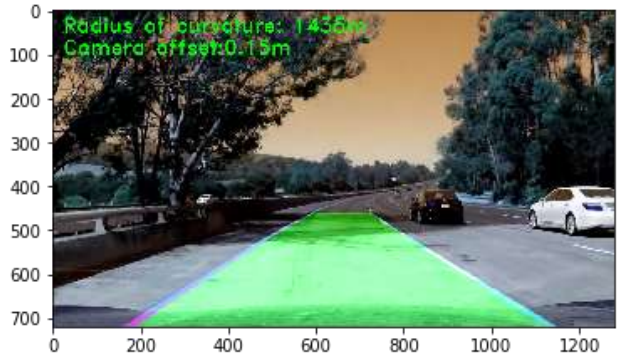

Figure 16: Result of test5.jpg

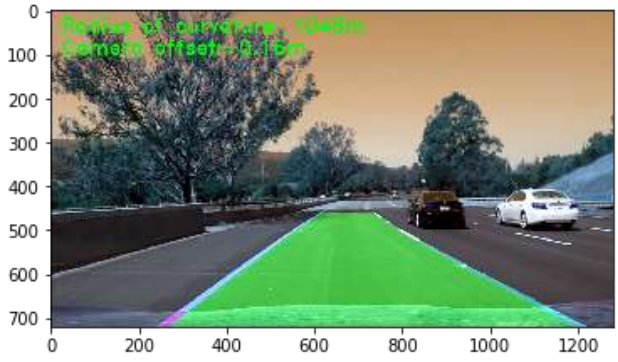

Figure 17: Result of test6.jpg

All the images are stored in **test_images/Final_result** folder. The name format of images stores are **<Sl.No.>_Result_<input image name>.png**.

Discussion (Resubmission)

The pipeline **(pipeline_advanced_lane)** was tested in all the test images and obtained a much better result. Moreover, the pipeline code was tested on the **project_video.mp4** and the obtained result is saved as **project_output_colour.mp4.** Lane lines are annotated with much more accuracy with the new pipeline. Apart from that, in last pipe line the challenge video couldn't be processed more than 2% due to poor extraction of lane points. However, the pipeline couldn't annotate the lanes accurately in challenge video. The harder challenge video was processed, the pipeline failed to annotate the lane lines correctly in almost all of the frames. The reasons I found are sudden change in lightings, reflections on the front glass of the car and curves.

In the resubmission, I focused mainly on changing the thresholding techniques as explained above in goal3 section. Also, one more area which needed my focus was the inverse warping of the processed image. In the last submission, I inverse warped the lane markings onto the distorted image instead of undistorted one, which is corrected in this submission.