ADVANCED LANE DETECTION

TABLE OF CONTENTS

OBJECTIVE

This project requires the developer to develop code that will detect lane boundaries from car-mounted camera feeds of highway driving in real-time.

GOALS

The goals of this project were the following:

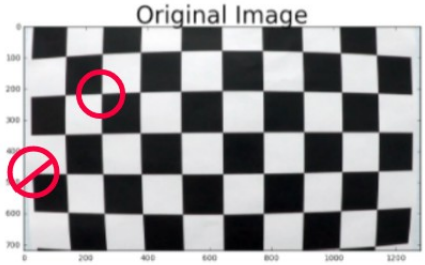
1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images for a specific camera.
2. Apply a distortion correction to raw images taken from that camera
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform 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.

Note: All of the code developed is in the file p495v.py, and references from here onward will refer to it thus, [1].

1. CAMERA CALIBRATION

After reading in the provided (chessboard) images for camera calibration (found in the folder: camera_cal in the Udacity-provided respository), each image was read one by one and processed ([1], file: p495v.py, lines 390-394):

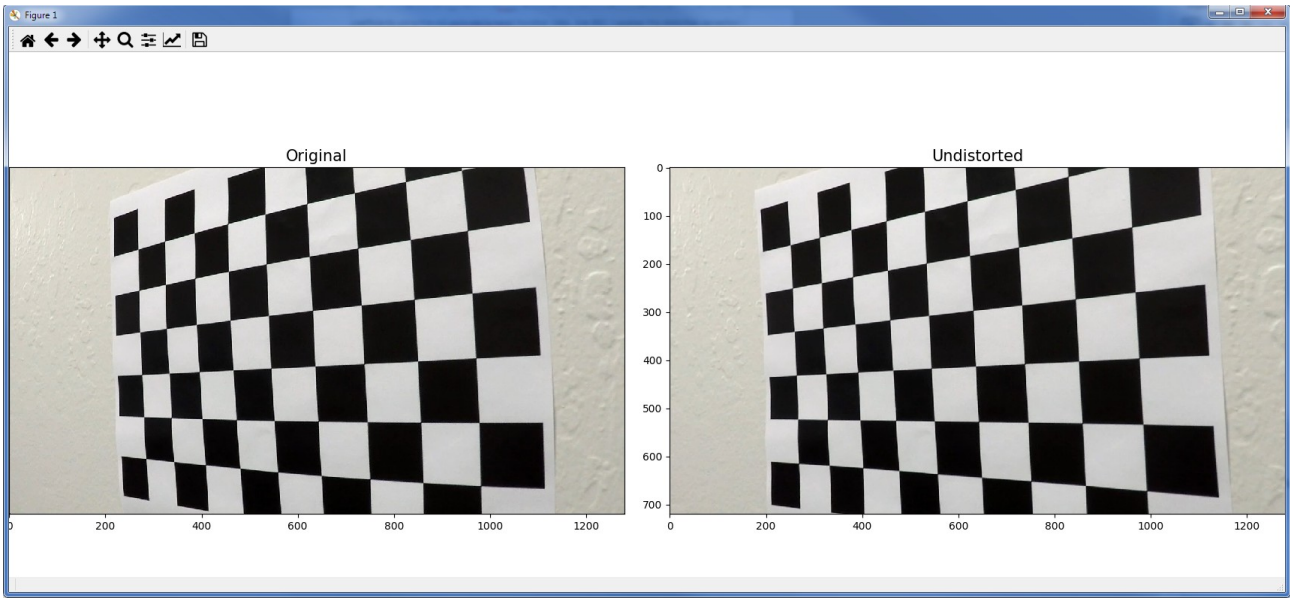
The process begins by preparing "object points" (obj_points) and “image points” (img_points), two numpy arrays. Object points are the (x, y, z) coordinates of the chessboard corners in worldspace. Here we assume the chessboard to be fixed on the (x, y) plane at z=0, such that these object points are the same for each calibration image. `objp` (an array defined inside the function FindDrawChessboardCorners()), is a replicated array of coordinates, and `obj_points` will be appended with a copy of `objp` every time all chessboard corners in a test image are successfully detected. Image points are a collection of the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection. A corner is defined as a four-way intersection, i.e., where two black and white squares meet. See image to the right.



Once these points are collected for all of the calibration images, they (`obj_points` and `img_points`) are used to compute the camera calibration matrix and distortion coefficients using the `cv2.calibrateCamera()` function ([1], line 392).

With the matrix and the coefficients thus returned by the above function, now, one can apply them to the test image (found in the folder: test_images in the Udacity-provided respository) using the `cv2.undistort()` function to obtain this result ([1], line 393):

Notice the disappearance of the curviness from the undistorted image (below right).



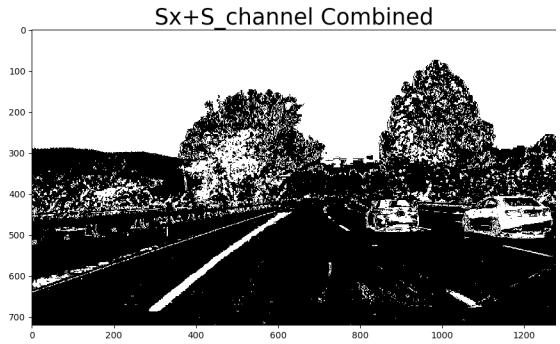
2. EXAMPLES OF DISTORTION CORRECTION TO RAW TEST IMAGES TAKEN FROM THE SAME CAMERA

This function, `cv2.undistort()`, is called at line 447 in [1], with the matrix and distortion coefficients returned above.

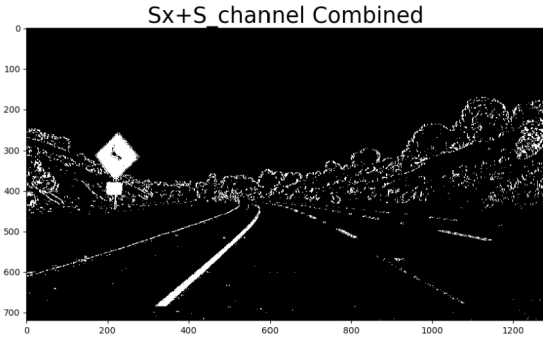
3. COLOR TRANSFORMS AND GRADIENTS TO CREATE A THRESHOLDED BINARY IMAGE

After much experimentation, a combination of Sobel X (horizontal) and the S-Channel of the image in the HLS color space produced a consistently clean result (to create a binary-thresholded image). Calculating the gradient magnitude nor its cousin, the gradient direction, produced better results. The routines `ApplySobel ()` and `ColorThreshold ()` contain the code used to achieve this ([1], line 72, line 104). A saturation threshold range (170-255) on the S-channel (combined with Sobel X) removed more noise than the range (90, 255).

90-255



170-255



The images shown above are examples of the (combined) output from these two thresholding operations (Sobel X and Color Thresholding on the S-Channel). The range on the right was retained in the code.

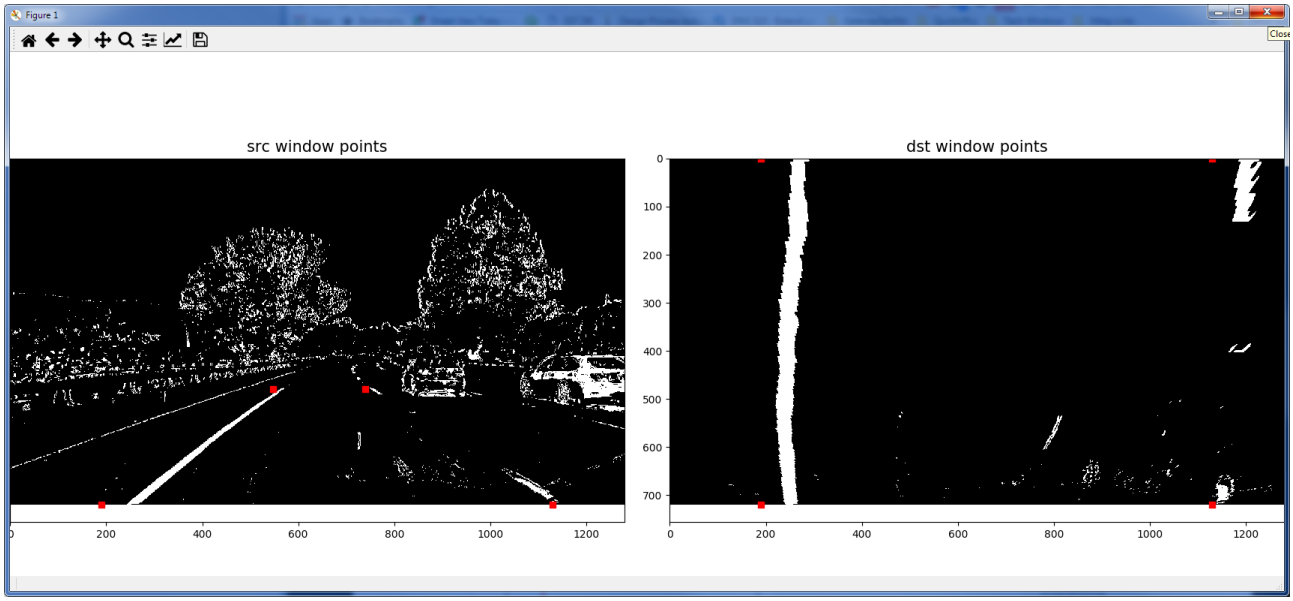
4. A PERSPECTIVE TRANSFORM TO RECTIFY BINARY IMAGE (A "BIRDS-EYE VIEW")

The code for perspective transform includes a function called `WarpPerspectiveXform()`, ([1], lines 112 through 133). This function takes as input a binary thresholded image.

The source and destination points for the transform were hardcoded in the following manner, after some experimentation, to work across all test images and for the entire duration of the project video.

```
Source:      src = np.float32([[200, 720], [480, 510], [780, 510], [1130, 720]])
Destination: dst = np.float32([[200, 720], [200, 100], [1130*, 100],[1130*, 720]])
* Also 1180 works well.
```

The image below provides confirmation of the fact that the chosen line points in the warped image are parallel to each other (look for the red squares).



5. LANE LINE PIXEL IDENTIFICATION AND POLYNOMIAL FIT

The lane line pixels were identified by drawing histograms across the bottom of the binary thresholded image window (see lines 166 through 173). See code in the function `DetectLanes_FitPoly()` ([1], lines 165 through 265). Looking for the maximum values of these pixels indicate the left edge and the right edge of the lane boundary. After this, the search window is moved up and the process repeated.

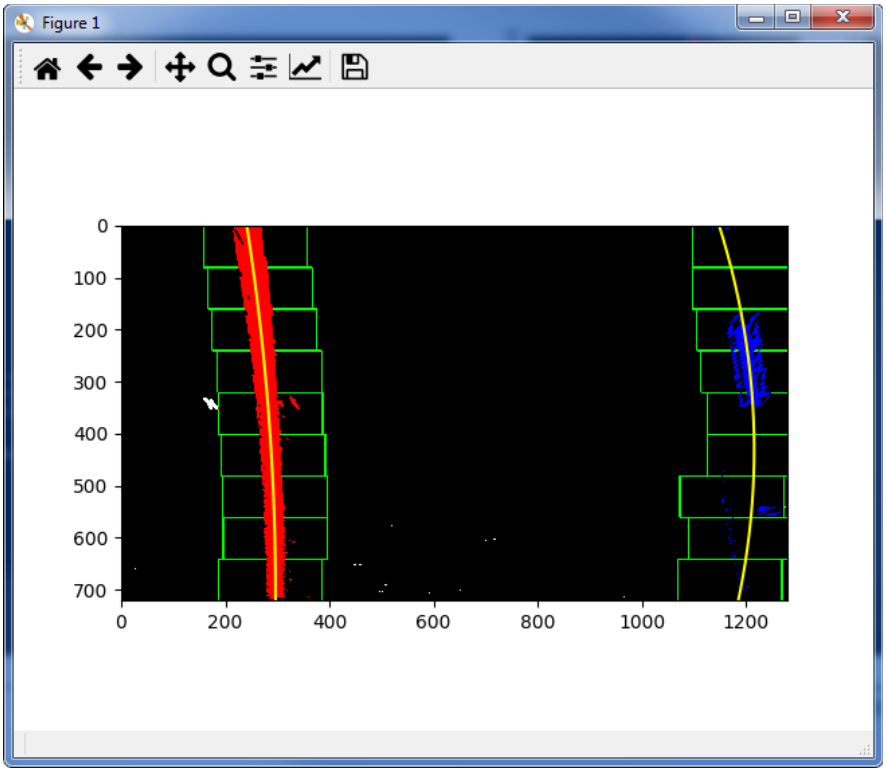
This function is used once when an image file is analyzed. However, when a video is analyzed, each subsequent frame beyond the first one is analyzed via a separate function `SearchSubsequentFrames()` to detect the (continuation of) lane lines from the previous frame. The technique is to search for lane line pixels in the neighborhood (by setting a margin) around the previously known lane lines. Once the two lane base points are identified, then the image height is divided into a number of sliding windows, and the process repeated to find the lane pixels.

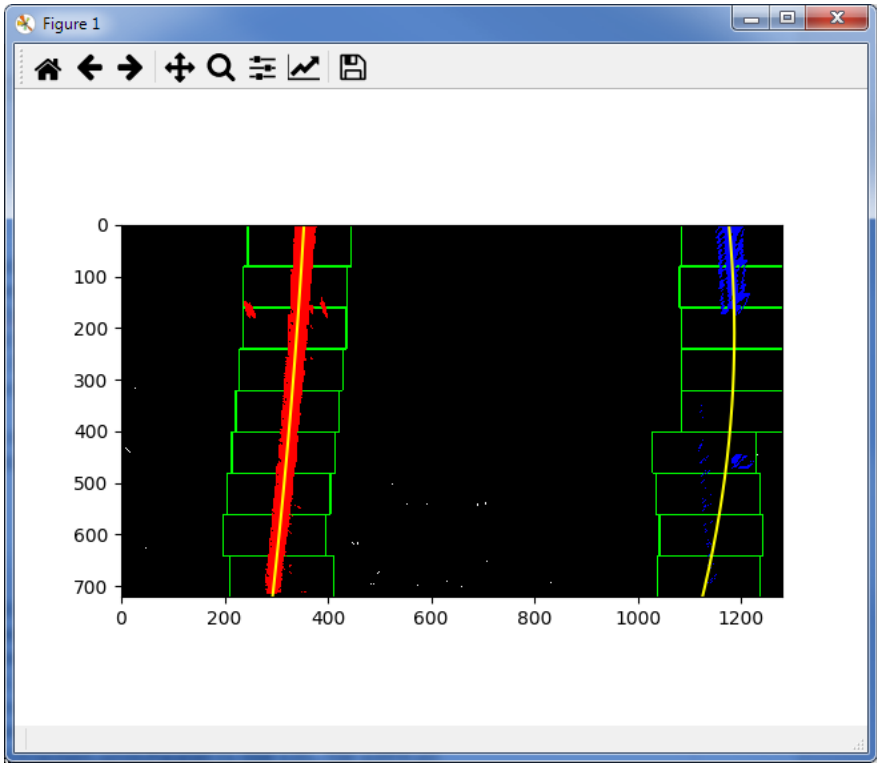
Once the two sides of the lane are identified via the above process, the `numpy.polyfit()` function is used to fit a quadratic curve through these points. This numpy function is used in both of the abovementioned functions.

The equation for this line / curve is a quadratic one of the form:

Equation 1: $f(y) = Ay^2 + By + C$

Note we are fitting f(y) and not the usual f(x) as the lane lines in the warped image are nearly vertical, and will have the same x value for different y values (as we trace the line up through the image).





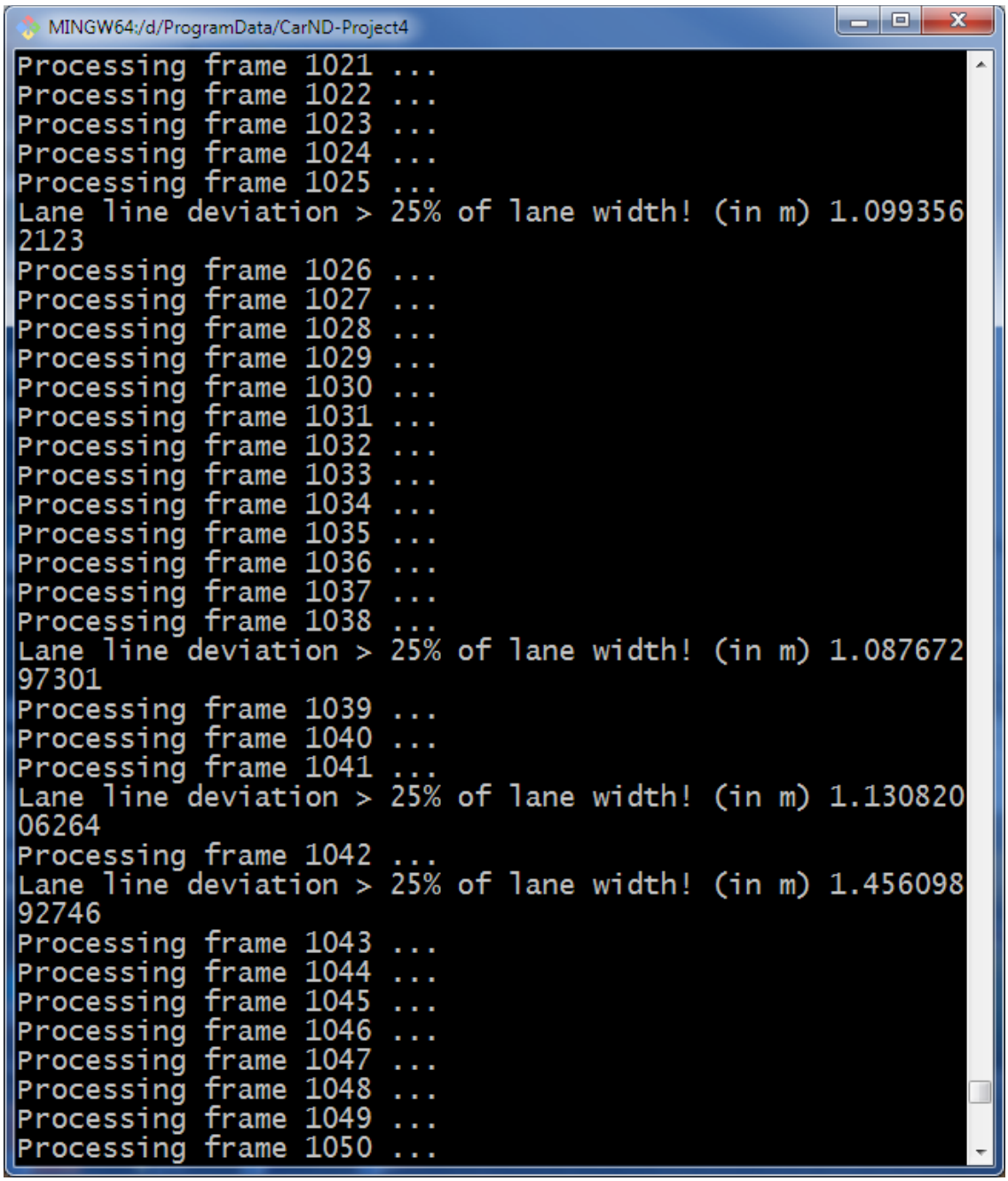
A SANITY CHECK

At this point, it is prudent to check whether the two quadratic lines are (approx.) parallel to each other. These parallel curves (or "offset" curves) are checked by measuring the distance between them in three locations along the height of the image. If any of these three differences exceeds 25% (entirely arbitrary) of the known lane width of 3.7m (according to [2], AASHTO's *A Policy on Design Standards: Interstate System)*, then a message is issued for that frame at that location by the algorithm and the process continued. See screenshot below for some example issuing of error messages. See the function, `CheckWhetherCurvesParallel ()`, ([1], line 135).

```
#Check differences, set tolerance to 25% of lane width (currently, 3.7m or 700
pix)
    tolerance = 0.25 * 700 #700 px make 3.7m, 1 lane width
    if   (np.abs(distances[0] - distances[1])) > tolerance:
        print ('Lane line deviation > 25% of lane width! (in m)',
np.abs(distances[0] - distances[1])*3.7/700)

    elif np.abs(distances[1] - distances[2]) > tolerance:
        print ('Lane line deviation > 25% of lane width! (in m)',
np.abs(distances[1] - distances[2])*3.7/700)

    elif np.abs(distances[0] - distances[2]) > tolerance:
        print ('Lane line deviation > 25% of lane width! (in m)',
np.abs(distances[0] - distances[2])*3.7/700)
```


MINGW64:/d/ProgramData/CarND-Project4
Processing frame 1021 ...
Processing frame 1022 ...
Processing frame 1023 ...
Processing frame 1024 ...
Processing frame 1025 ...
Lane line deviation > 25% of lane width! (in m) 1.099356
2123
Processing frame 1026 ...
Processing frame 1027 ...
Processing frame 1028 ...
Processing frame 1029 ...
Processing frame 1030 ...
Processing frame 1031 ...
Processing frame 1032 ...
Processing frame 1033 ...
Processing frame 1034 ...
Processing frame 1035 ...
Processing frame 1036 ...
Processing frame 1037 ...
Processing frame 1038 ...
Lane line deviation > 25% of lane width! (in m) 1.087672
97301
Processing frame 1039 ...
Processing frame 1040 ...
Processing frame 1041 ...
Lane line deviation > 25% of lane width! (in m) 1.130820
06264
Processing frame 1042 ...
Lane line deviation > 25% of lane width! (in m) 1.456098
92746
Processing frame 1043 ...
Processing frame 1044 ...
Processing frame 1045 ...
Processing frame 1046 ...
Processing frame 1047 ...
Processing frame 1048 ...
Processing frame 1049 ...
Processing frame 1050 ...

6. RADIUS OF CURVATURE (ROC), VEHICLE CENTER OFFSET

RADIUS OF CURVATURE

The function `CalcRadOfCurvature ()` ([1], line # 327) calculates the radius of curvature of the detected (fit) lane boundaries while also calculating the vehicle's center offset.

The formula for the radius of curvature at any point x for the curve y=f(x) is given by:

Equation 2:

$$\text{Radius of curvature} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

So, here the function, f(x) actually f(y), is equation 1 itself, shown before.

So, $f'(y) = dy/dx = 2A.y + B$: the first derivative

$f''(y) = d^2y/dx^2 = 2A$: the second derivative

So, equation 2 becomes:

$$\text{Radius of curvature} = \frac{(1 + (2Ay+B)^2)^{3/2}}{|2A|}$$

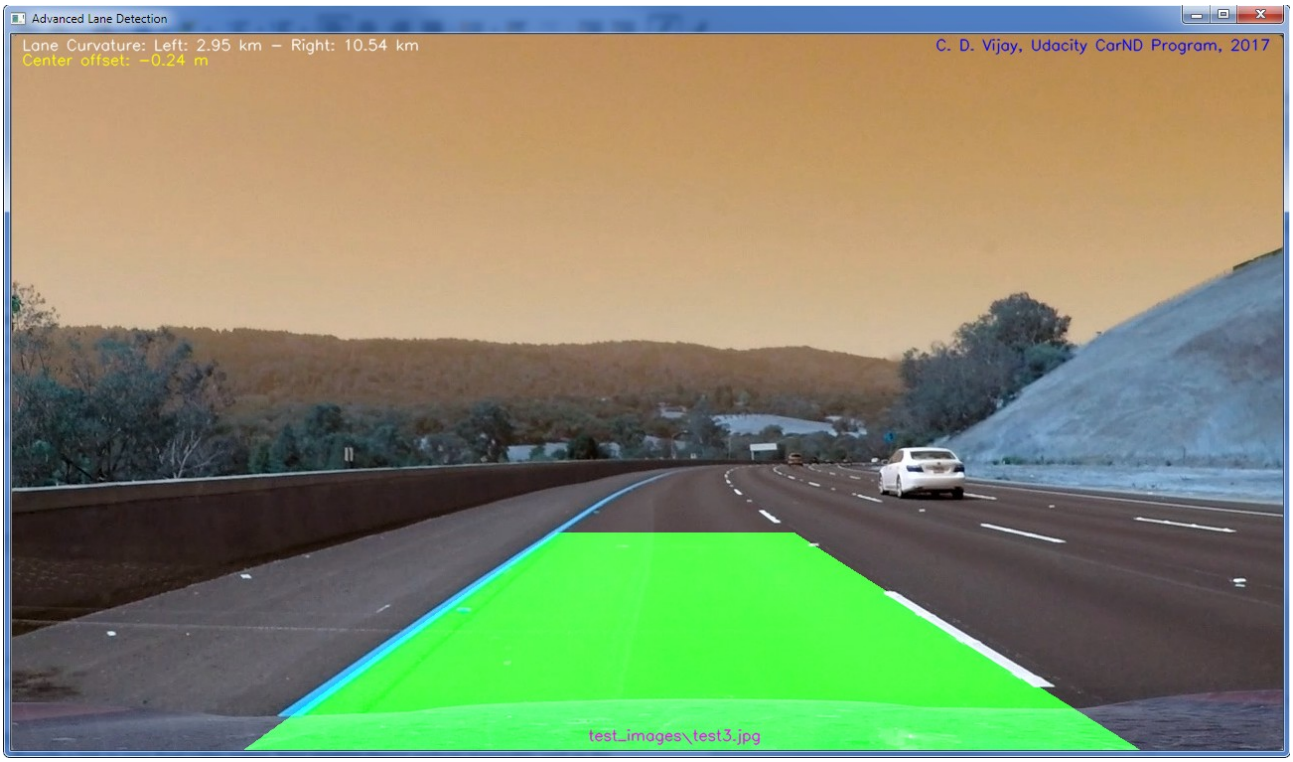
This is implemented in [1] at lines 338, 339.

CENTER OFFSET

The center offset is calculated from subtracting the center of the image from the center of the lane (as found from the bases of the lane boundaries). This number in pixels is scaled to world-space (3.7m for every 700 pixels). This is implemented at line 342 in [1]. This information along with ROC are displayed in the final output on the image.

7. FINAL RESULTANT IMAGE OVERLAID ON THE ORIGINAL

So, as can be seen, the lane area is clearly demarcated and overlaid on the actual lane, while the radius of curvature coming up ahead is printed too on the display.



Link to the output video on Youtube

8. DISCUSSION

A few points that may need revisiting:

1. While the code given here works well on the project_video.mp4 file, it is clear that this algorithm will require at least re-tuning, perhaps, rewriting certain sections of the code to generalize over any highway. This code runs about halfway on the challenge video, and about a third on the harder challenge video.

2. This code is not in any way optimized. There are some expensive operations that can be done away with.

3. More robust error detection and correction is called for. I have several ideas but not the time to implement them. For example, the R-channel seems like a more than acceptable alternative to the S-channel when thresholding. So, when the sanity check given above throws an error, I would like to switch to R-channel processing immediately.

4. Smoothing and averaging would be nice to have in this code.

5. Some way to capture the RankWarning issued by the numpy.polyfit() function would be better combined with R-channel switching.

6. I cannot help but contrast this code and the method of thinking as akin to a rules-based system, no different from a procedural system. Deep learning is definitely a model I would test as a potential solution.

REFERENCES

[1]. Source code file, p495v.py, submitted with this project

[2]. *A Policy on Design Standards: Interstate System,* American Association of State Highway and Transportation Officials (AASHTO)

[3] Udacity CarND Term1, Course material