Udacity CarND Project 4

Advanced Lane Detection

→Steps followed were:-

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 colour transforms, gradients, etc., to create a threshold 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 centre.
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.

→Flowchart:-

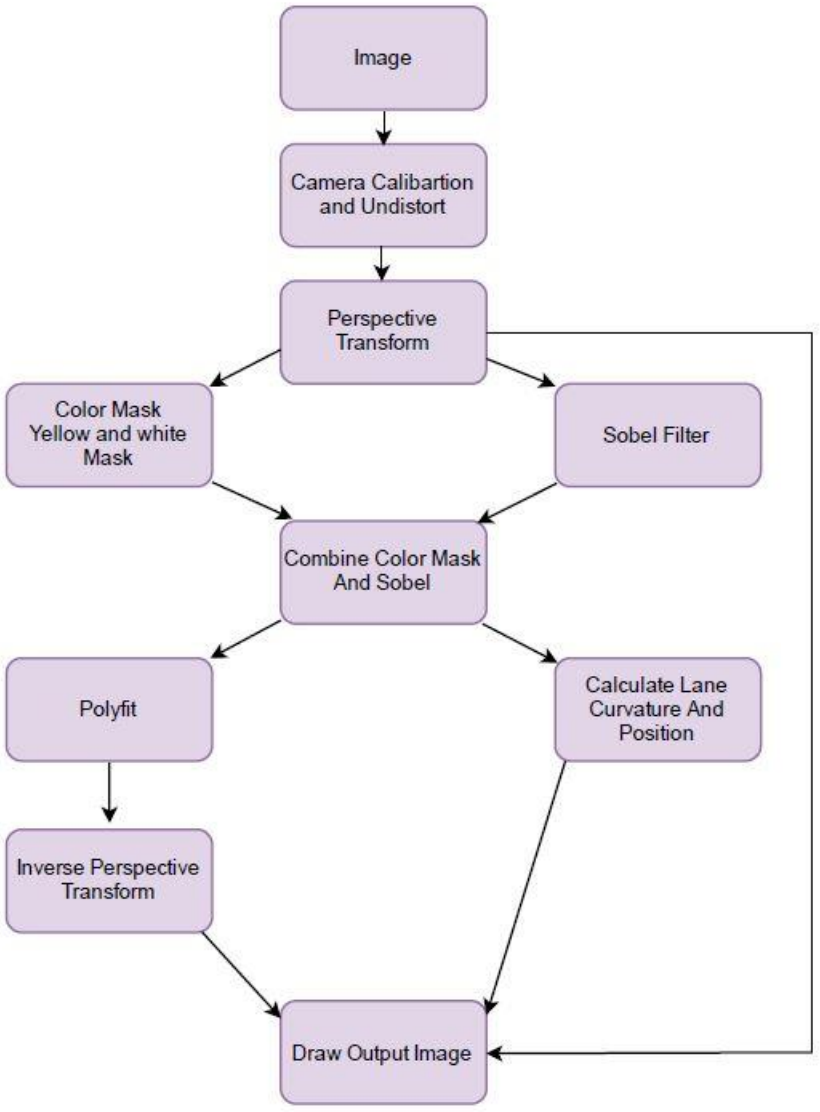


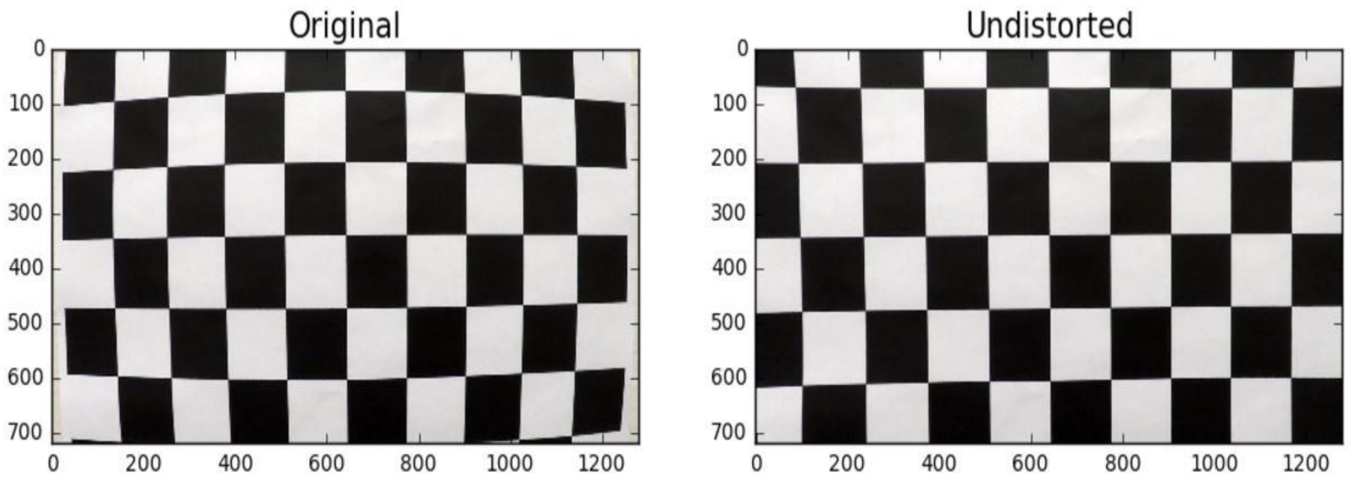
Image
Camera Calibartion and Undistort
Perspective Transform
Color Mask Yellow and white Mask
Sobel Filter
Combine Color Mask And Sobel
Polyfit
Calculate Lane Curvature And Position
Inverse Perspective Transform
Draw Output Image

→Camera Calibration:-

In order to compute the camera matrix and distortion coefficients, CameraCalibration uses OpenCV's **cv2.findChessboardCorners** function on grayscale version of each input image in turn, to generate a set of image points. I then use numpy.mgrid in order to create a matrix of (x, y, z) object points.

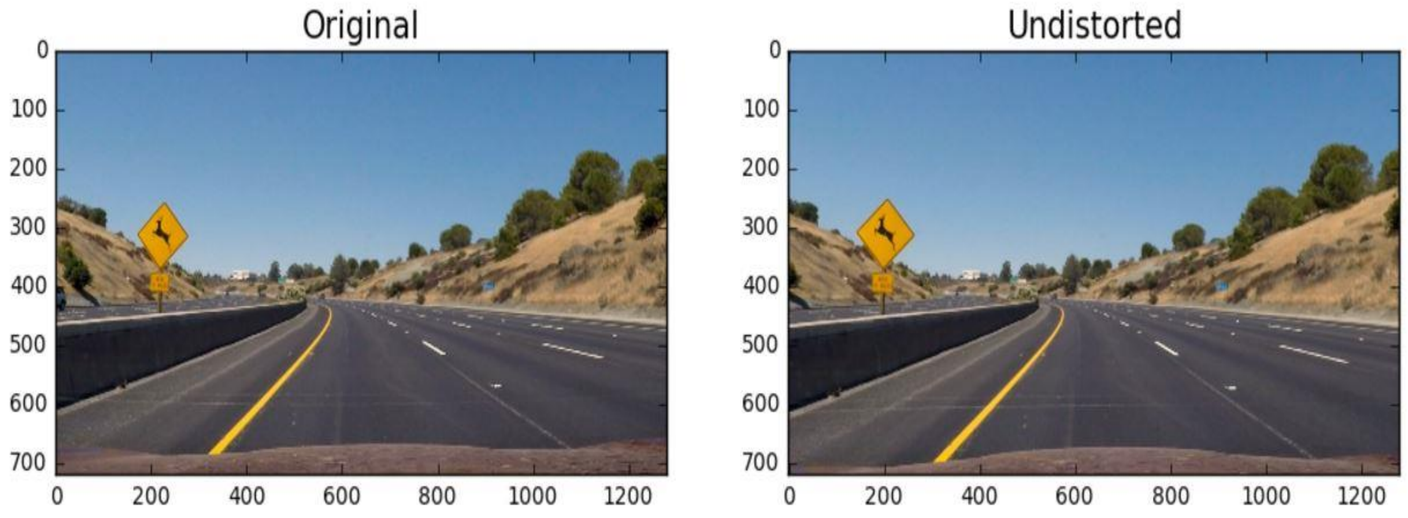
Once I have the image points and the matching object points for the entire calibration dataset, I then use **cv2.calibrateCamera** to create the camera matrix and distortion coefficients. Then **cv2.undistort**, returns an undistorted version of the user's input image.

Code is implemented in **camera_Calibraton** and **undistort** functions.



→ Example of a distortion-corrected image:-

After calculating mtx and dist, we use them to undistort our testimages using **undistort** function.

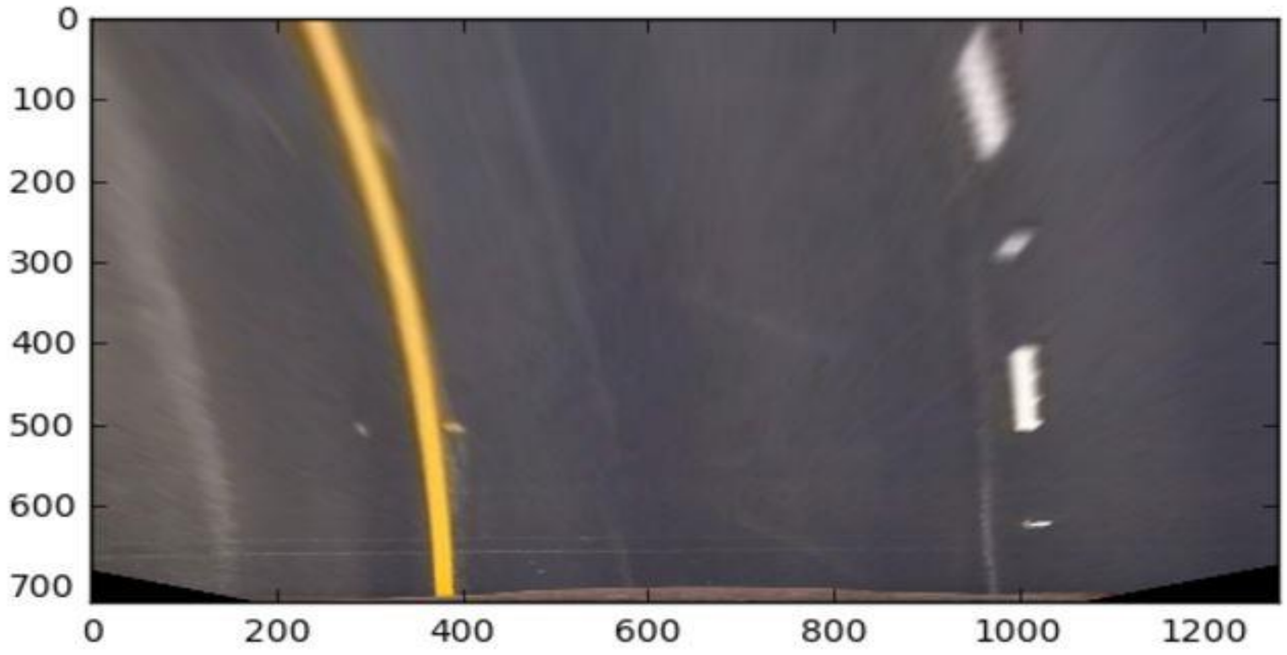


→Perspective Transform:-

Perspective transformation gives us bird's eye view of the road, this makes further processing easier as any irrelevant information about background is removed from the warped image.
Code for this can be found in **transform_perspective** function. Here I am defining source and destination points.

$$
\begin{aligned}
src \;=\; np.array([&[585./1280.*\,img\_size[1], 455./720.*\,img\_size[0]],\\
&[705./1280.*\,img\_size[1], 455./720.*\,img\_size[0]],\\
&[1130./1280.*\,img\_size[1], 720./720.*\,img\_size[0]],\\
&[190./1280.*\,img\_size[1], 720./720.*\,img\_size[0]]], np.float32)\\
dst \;=\; np.array([&[300./1280.*\,img\_size[1], 100./720.*\,img\_size[0]],\\
&[1000./1280.*\,img\_size[1], 100./720.*\,img\_size[0]],\\
&[1000./1280.*\,img\_size[1], 720./720.*\,img\_size[0]],\\
&[300./1280.*\,img\_size[1], 720./720.*\,img\_size[0]]], np.float32)
\end{aligned}
$$

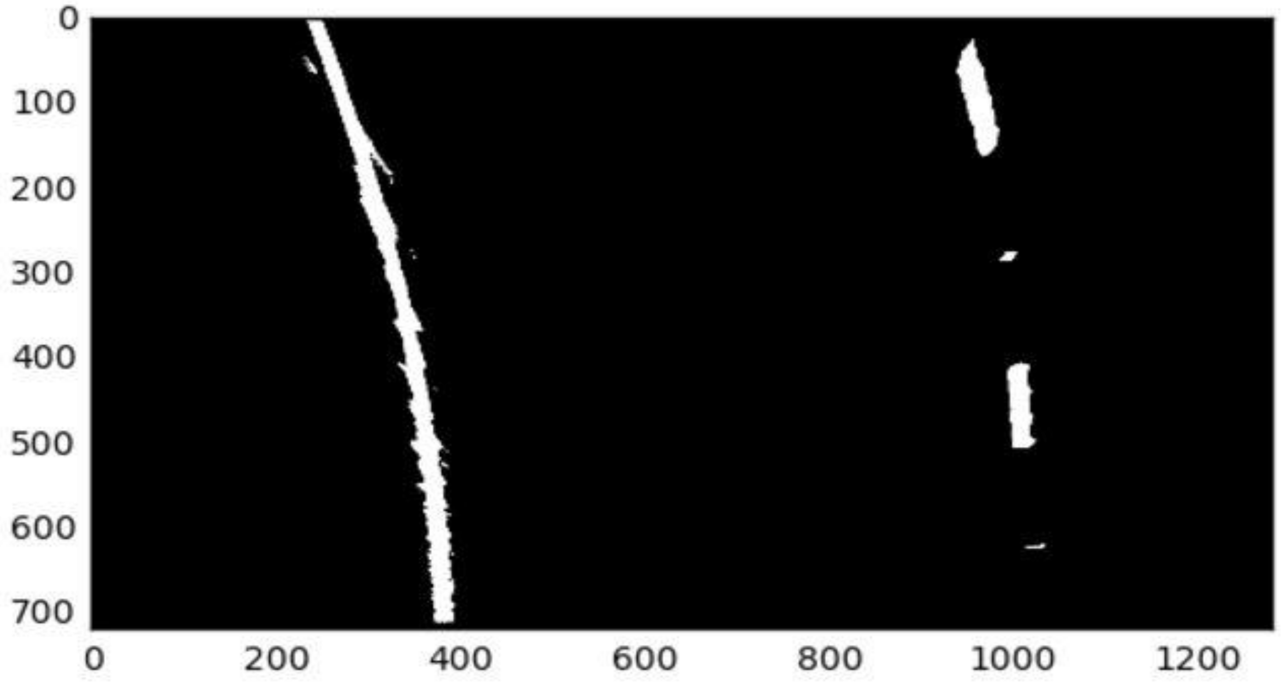
So here I am using **cv2.getPerspectiveTransform** and **cv2.warpPerspective** functions from OpenCV.



So in the above image we can see the bird’s eye view of the earlier used test image.

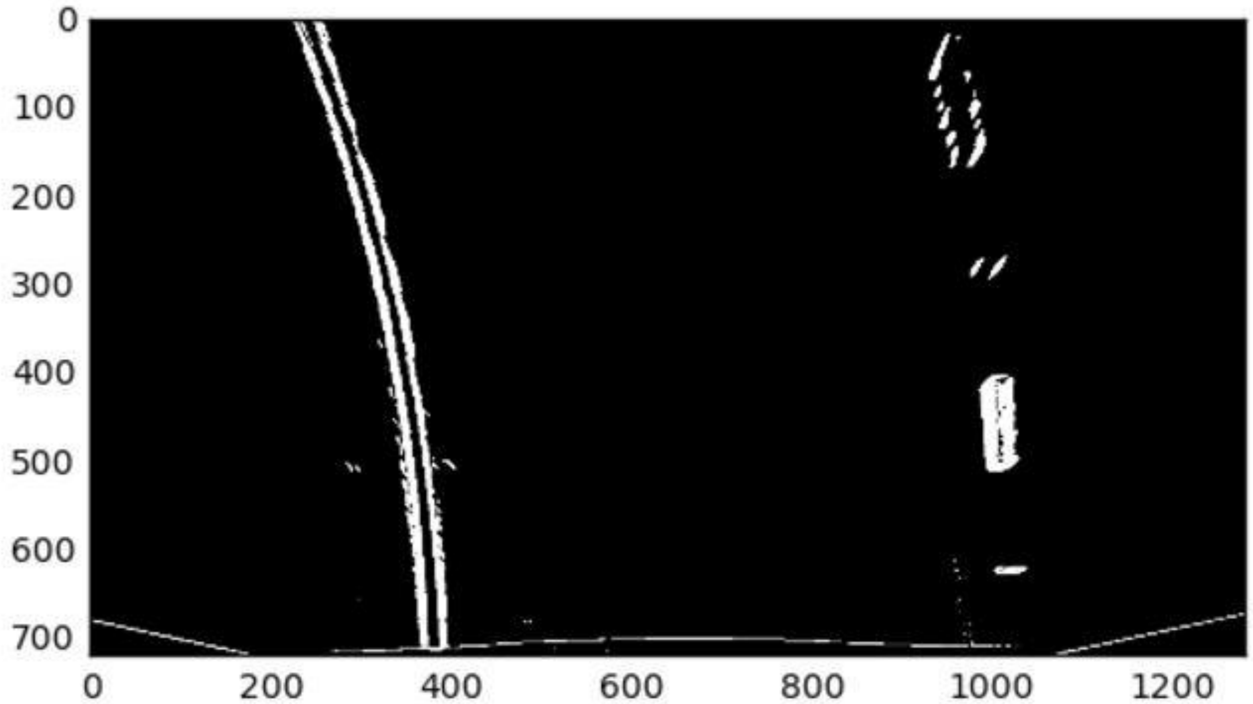
→Binary Images:-

Once we obtain the perspective transform, we next apply color masks to identify yellow and white pixels in the image. Color masks are applied after converting the image from RGB to HSV space. HSV space is more suited for identifying colors as it segements the colors into the color themselves (Hue), the ammount of color (Saturation) and brightness (Value). We identify yellow color as the pixels whose HSV-transformed intensities are between ([20, 100, 100]) and ([50, 255, 255]), and white color as the pixels with intensities between ([0, 0, 187]) and ([255, 255, 60]).

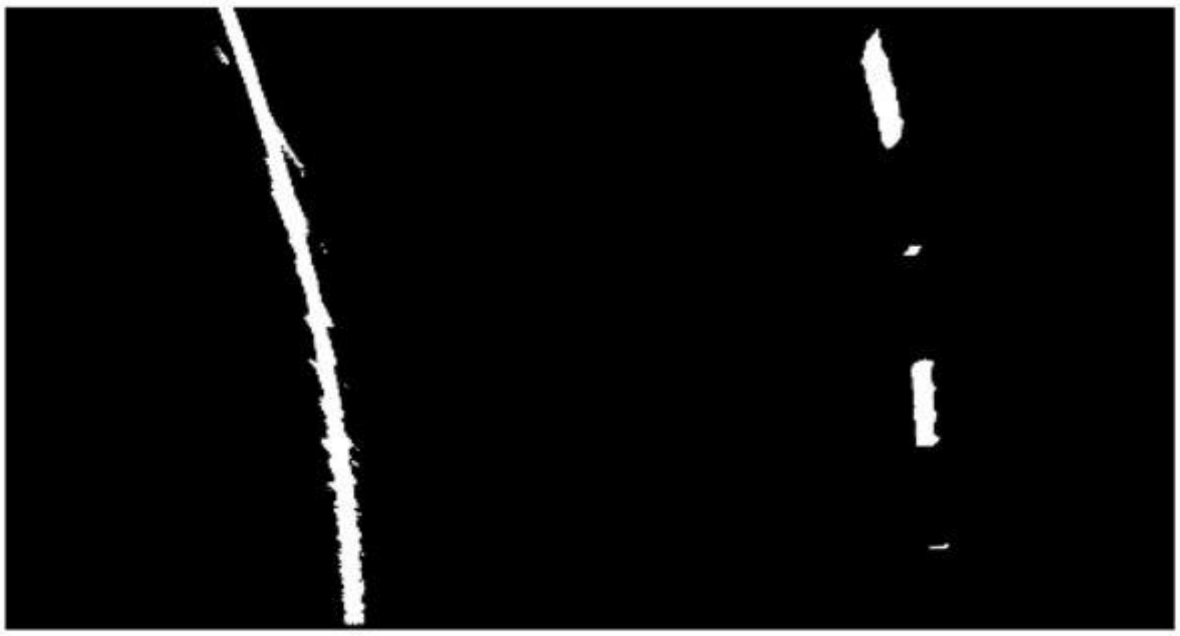


So above image is generated when we apply color mask and the code can be found in **color_mask** function.

In addition to the color masks, we apply sobel filters to detect edges. We apply sobel filters on L and S channels of image, as these were found to be robust to color and lighting variations. After trial and error, we decided to use the magnitude of gradient along x- and y- directions with thresholds of 50 to 200 to identify the edges.
Code for this can be found in **apply_sobel_filter** function.



Then we combine both the binaries and get this:-



Also I have used some Outlier removal methods like if the change in coefficient is above 0.005, the lanes are discarded, If any lane was found with less than 5 pixels, we use the previous line fit coefficients as the coefficients for the current one.
 All this was done by trial and error. Code can be seen in **pipeline** function.

→Lane Curvature:-

In order to calculate the lane curvature radius, I scaled the x and y coordinates of my lane pixels and then fit a new polynomial to the real-world-sized unit. Using this polynomial, I could then use the radius of curvature formula below to calculate the curve radius in metres at the base of the image:

$$R = \left| \frac{\left(1+\left(\frac{dy}{dx}\right)^2\right)^{3/2}}{\frac{d^2y}{dx^2}} \right|$$

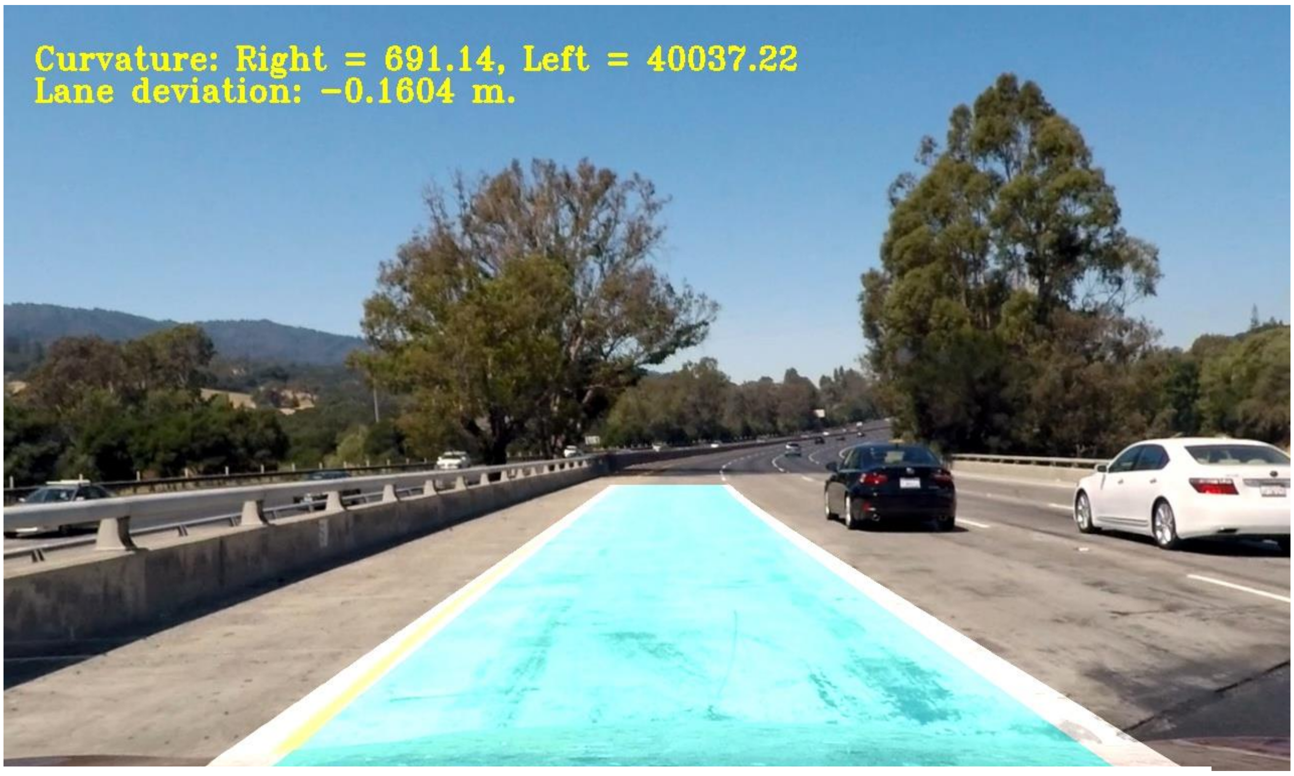
Code for this can be found in **get_curvature** function.

→Offset from lane center:-

To calculate the position of the car, I used the polyfit function to find the bottom coordinates of right and left lanes. Assuming the lane has width of 3.7 metres I calculated distance between centre of image and centre of lane. This is our offset.

Code for this can be found in **find_offset** function.

→Final output:-


Curvature: Right = 691.14, Left = 40037.22
Lane deviation: -0.1604 m.

All the processed images are saved in output_images folder.

Project Video output is there in repo as project_output.mp4.

You can check on https://youtu.be/r2Bv3DFVbBQ

→Discussion:-

I think the part where my pipeline is not robust is generating the mask. The pipeline works good for project output and fails for challenge videos. A lot of manual tuning is done in this project which is not good.

Also in order to make lane detection more good we can normalize the data.