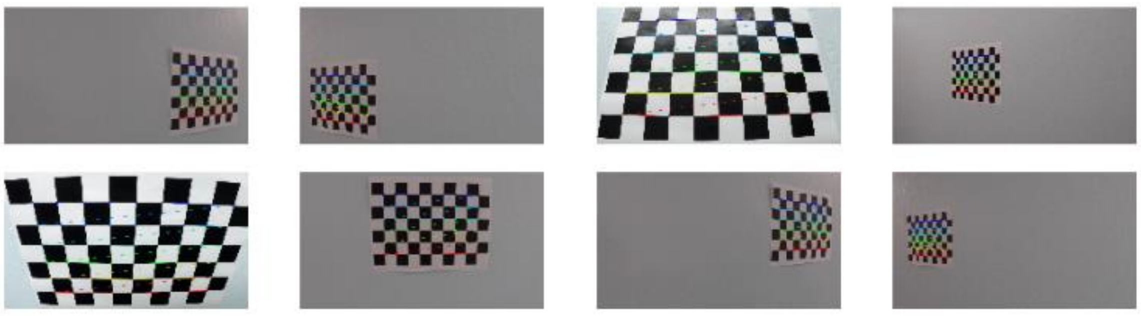
Advanced Lane Detection Project

1)Camera Calibration :

The code for this step is contained in the first code cell of the IPython notebook located in "./examples/advanced_lane_detection.ipynb"

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 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.

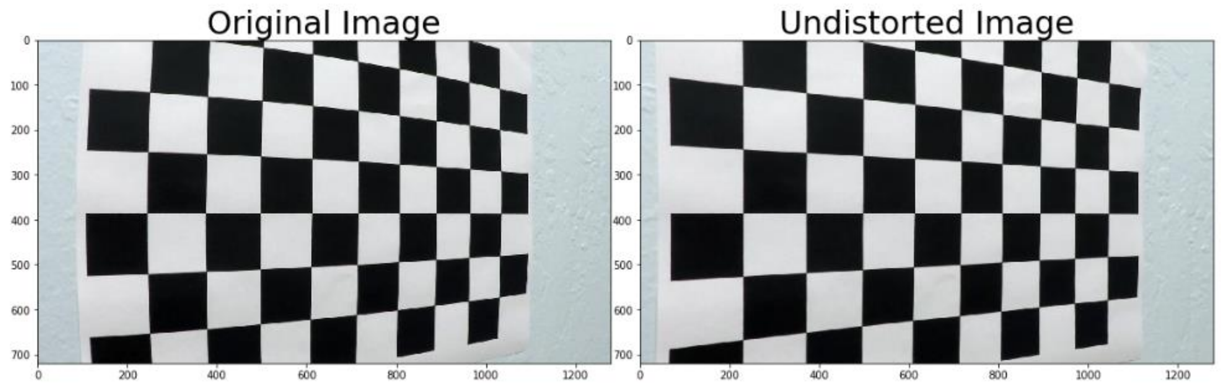
These are the photos of the detected corners drawn using drawChessboardCorners().

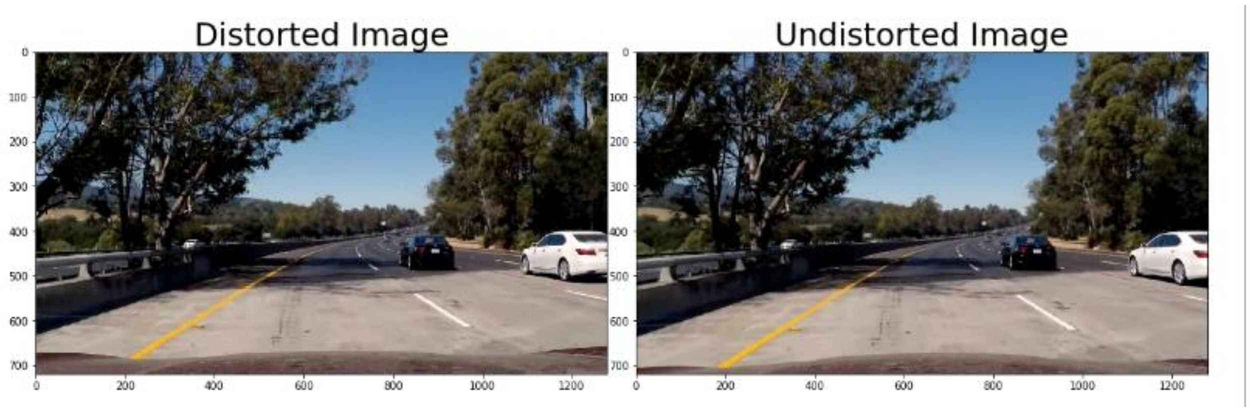


Pipeline:

1)Distortion Correction :

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 the test image using the `cv2.undistort()` function and obtained this result:





The above image shows how the algorithm uses cv2.calibrateCamera() function and provides mtx and dist for cv2.undistort() to undistort the image.

This is the code snippet used to undistort an image:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size,None,None)

undistort = cv2.undistort(image, mtx, dist, None, mtx)
```

2)Perpective Transform:

The code for my perspective transform includes a function called `img_warp ()`.

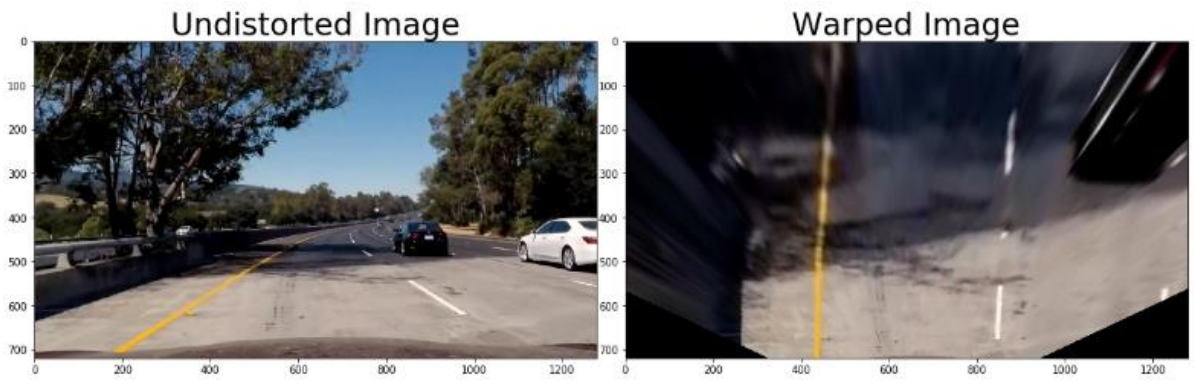
The `img_warp ()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

```python
src = np.float32([(575,464),
                  (707,464),
                  (258,682),
                  (1049,682)])
dst = np.float32([(450,0),
                  (w-450,0),
                  (450,h),
                  (w-450,h)])
```

```
M = cv2.getPerspectiveTransform(src, dst)
warped = cv2.warpPerspective(image, M, (w,h), flags=cv2.INTER_LINEAR)
```

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

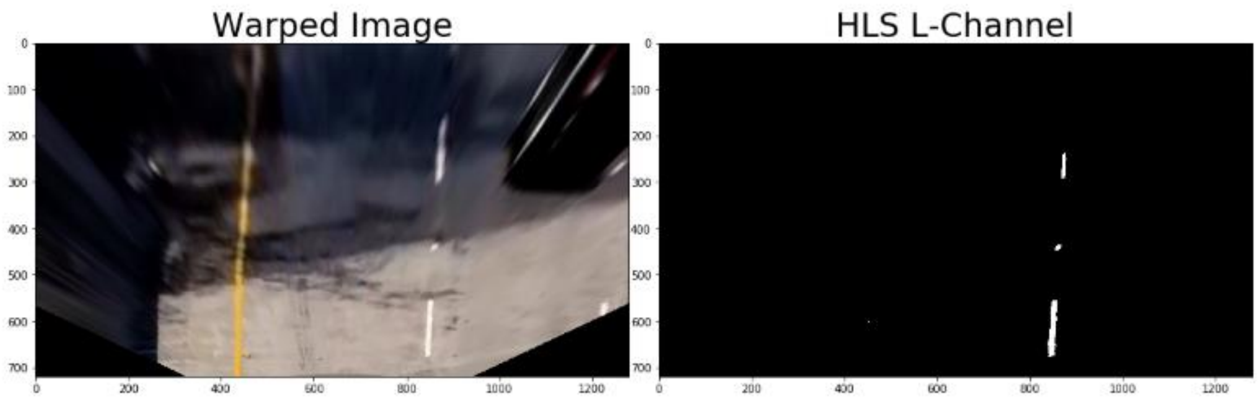
The following image shows the working of img_warp() function on a undistorted image.



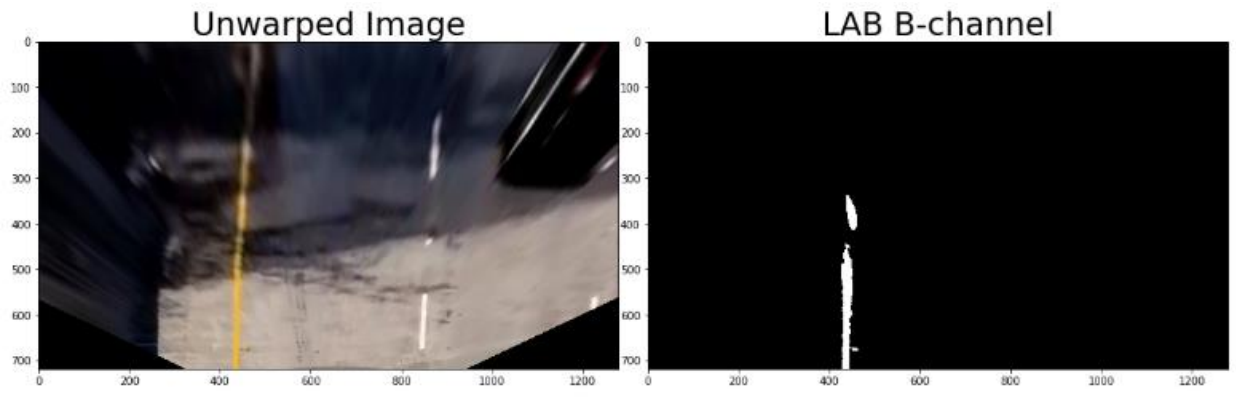
3)Threshold Binary Image:

For obtaining a binary image which I used 2 colour identifying function.

hls_lthresh() function is used to obtain white lines on the road. It identifies them by using lightness component in the HLS color space to pick up white lines.

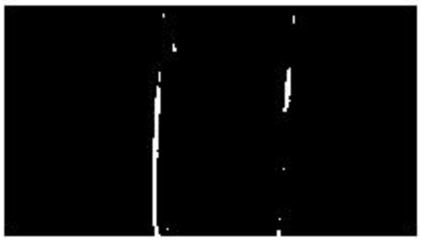


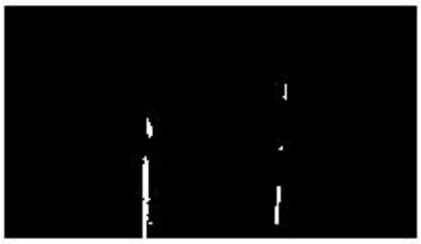
Lab_bthresh() function is used to obtain yellow lines on the road. It identifies them by using the blue – yellow component in the **CIELAB** color space to pick up yellow lines.

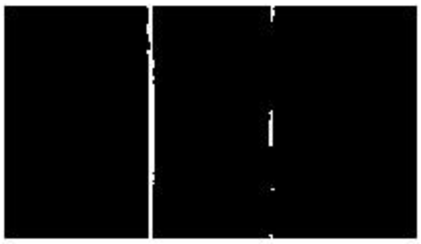


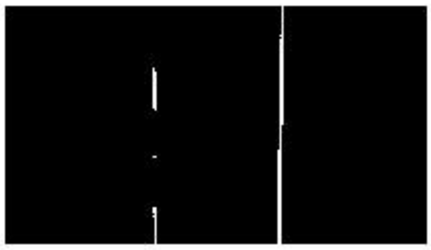
Apart from this I tried using Sobel operators for picking up lines and other color channels but I got the best results from the above two function which I combine to produce the final pipelined image.

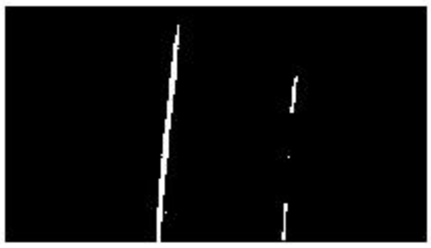
Here are some photos of the pipelined images using the above functions :

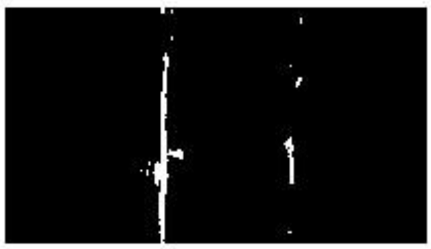








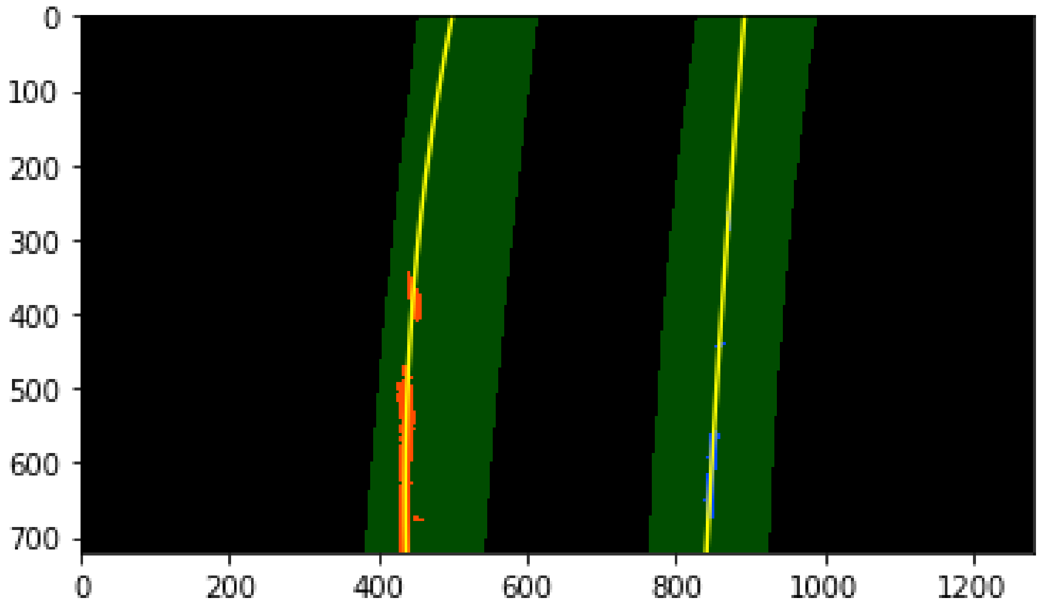




4) Identification of lane-line pixels and fitting their positions with a polynomial:

For identification of lane lines initially I used the function sliding_window_polyfit(image) which takes in an image and return left_fit, right_fit, left_lane_inds, right_lane_inds, visualization_data. This in turn in the next image frame is fed to polyfit_using_prev_fit()

Which takes in the previos left_fit and right_fit initially calculated to start finding lanes where they were previously found which increases efficiency.



The above image shows how the polynomial identifying function takes in the binary image and returns lanes highlighted with approximation of missing parts.

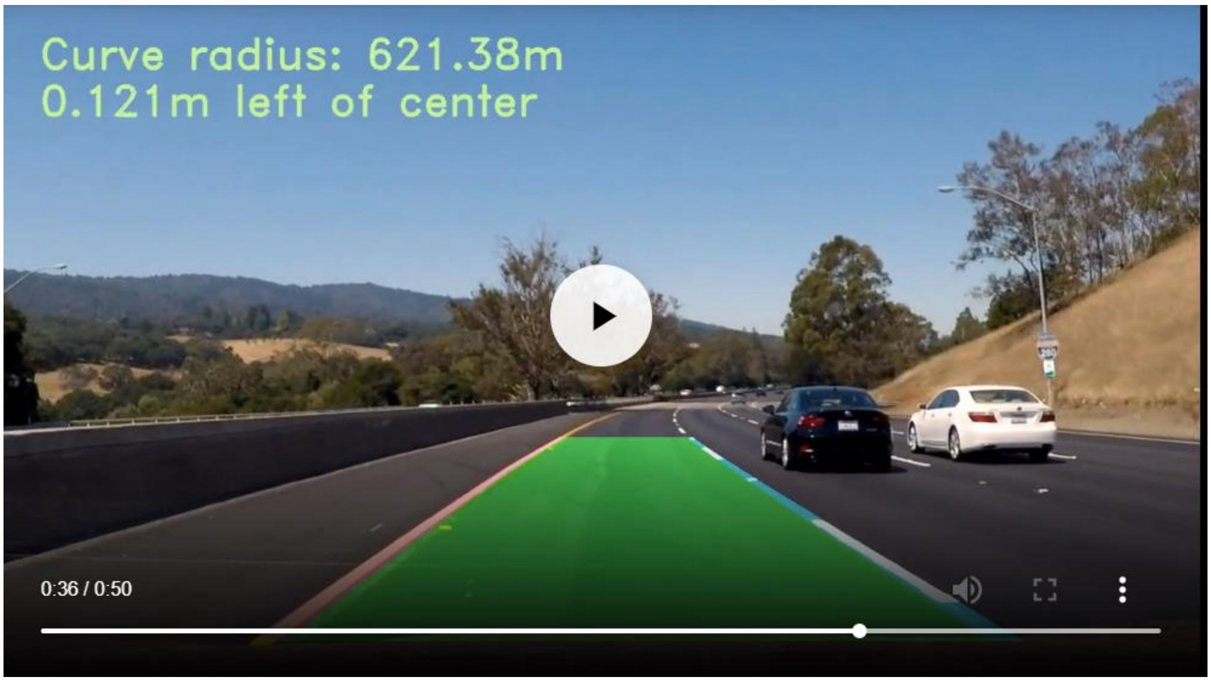
6) Calculation of the radius of curvature of the lane and the position of the vehicle with respect to center :

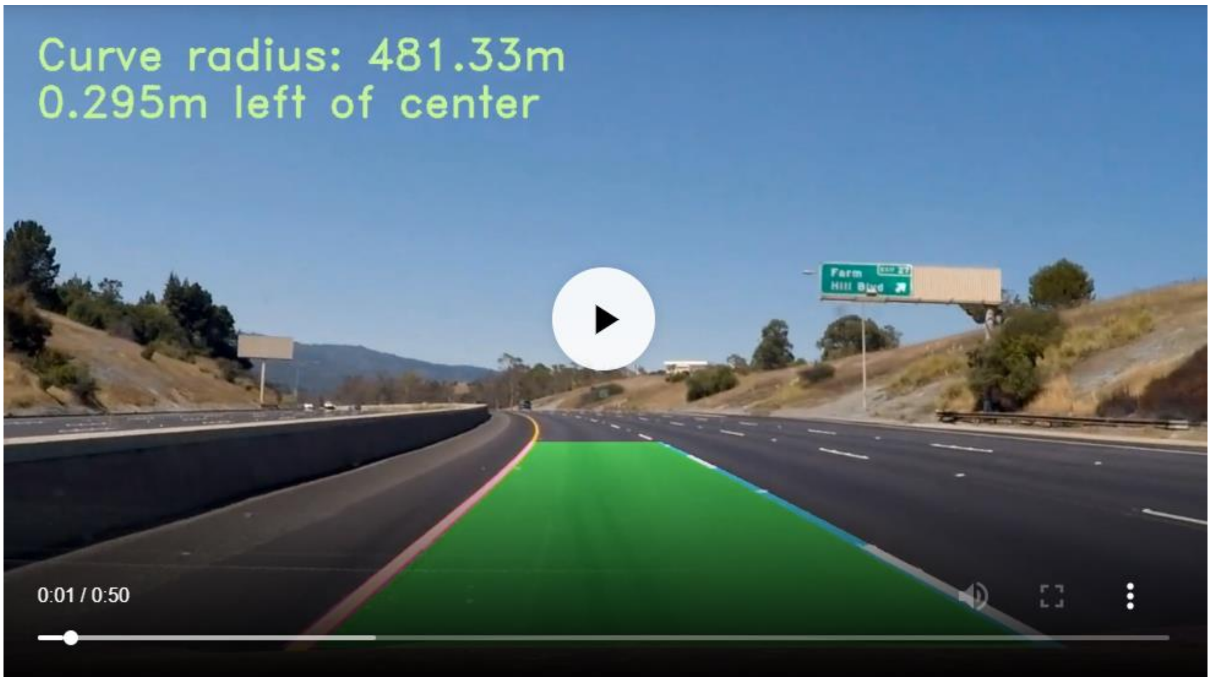
To calculate the radius of the curvature it is necessary to calculate the real world conversions of pixels in terms of metric system i.e. meters. For this purpose the x and y values of the polynomials obtained are multiplied with pre-determined constants to give polynomial equation in metres. Radius of curvature is determined by the following :

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

The distance of the centre is determined by taking the value of midpoint in x direction of the image and substracting it with the central part of the left and right lane and multiplying them with x pixel to meter constant.

Pipeline Video Screenshots





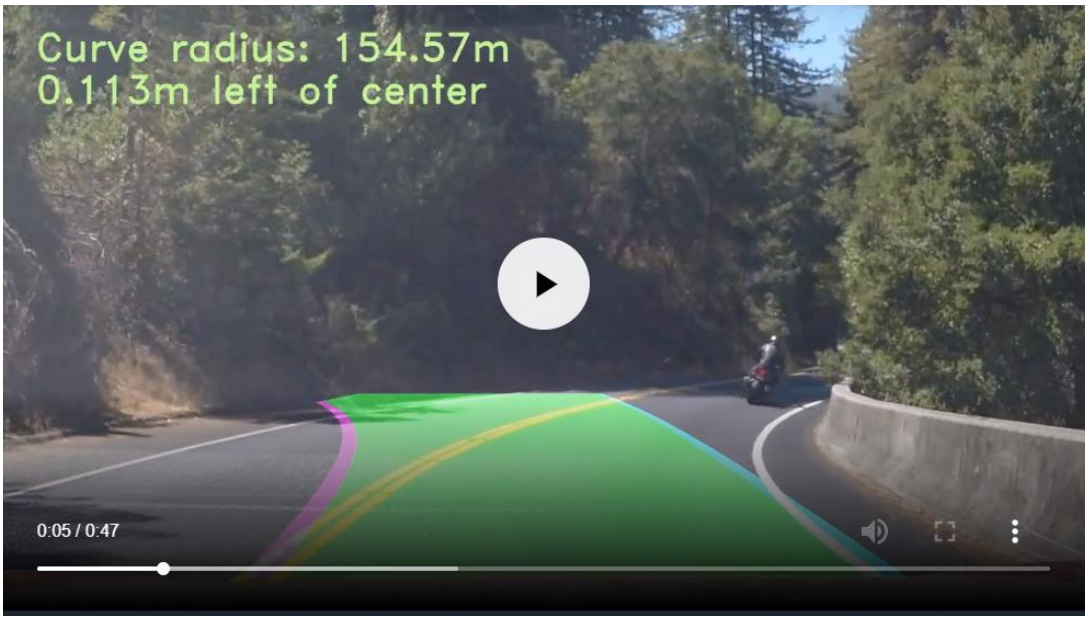
The complete video of the lane detection project is available in
'./project_video_output.mp4'

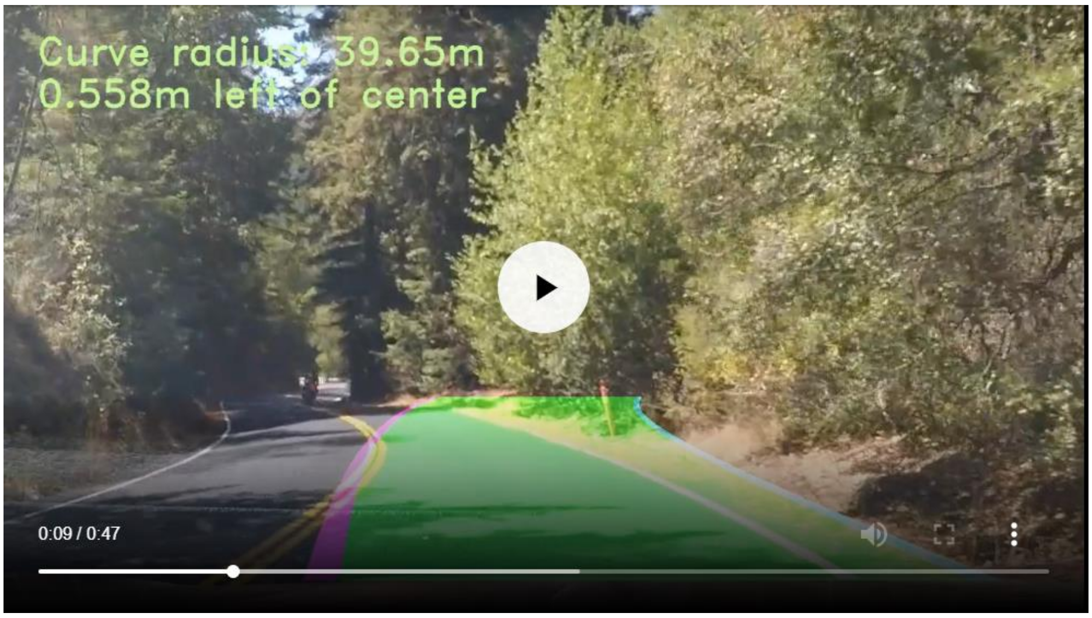
Project Difficulties and Shortcomings:

The project works fine on flat roads and at roads with fairly high radius of curvature.

However once there the radius of curvature becomes smaller and smaller and with more consecutive turns the pipeline fails to identify the roads. The problem I feel this is cause due to the warped image area taken into consideration while pipelining. Although reducing the warped area length may seem like the solution it gives rise to the problem where just a short length of the road ahead of the vehicle is detected which is not efficient for speed and vehicle dynamics planning.

Here are some images where the pipeline has failed to identify the lanes.


Curve radius: 154.57m
0.113m left of center
0:05 / 0:47


Curve radius: 39.65m
0.558m left of center
0:09 / 0:47