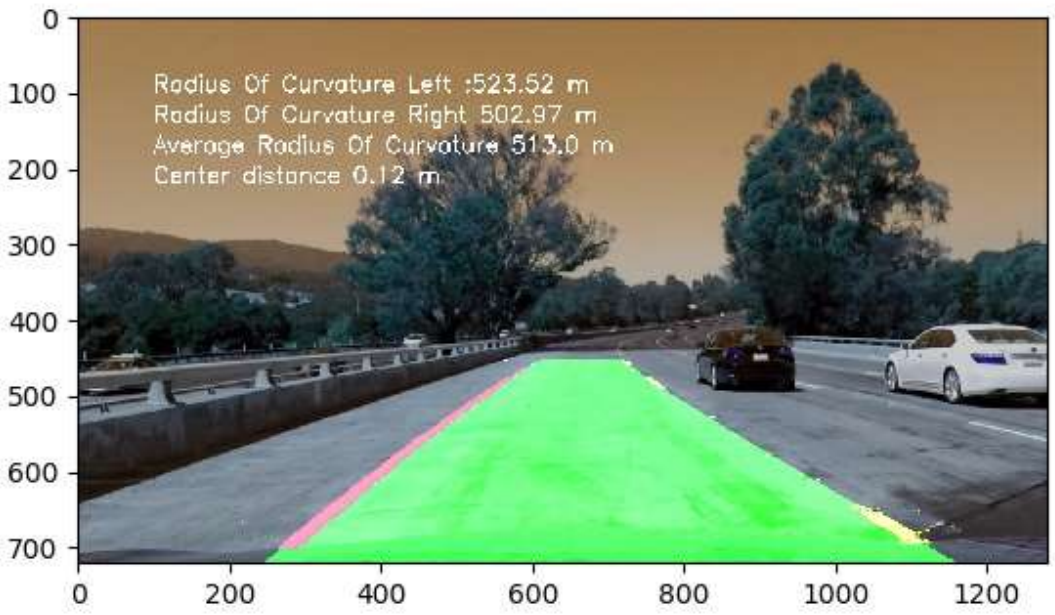
Advanced Lane Detection



Steps Involved :

1. Camera Calibration (to remove camera aberrations for accurate detection and measurements) , in this step we compute the transform matrix and distortion coefficients using some uncalibrated images.
2. Undistort the images by transformation matrix and distortion coefficient previously computed.
3. Apply Color, Gradient , Direction of gradient, Magnitude of gradient thresholding to filter lane lines from the image .

4. Apply “Perspective transform” over a ROI(region of interest) that can capture most of the lane turning and generalize well for all other images.
5. Apply “Peak Histogram Search” or “Local Search” for detecting lane pixels, apply coloring and fit a polynomial using regression analysis(np.polyfit), also calculating the Radius Of Curvature, offset of car position from the lane’s center.
6. Apply “Inverse Perspective Transform” of Perspective image with lanes detected.
7. Blend in the Inverse Perspective Transform image with the original image, this makes an overlay of the lane over the original image.

Camera Calibration

(The code for this is included in the project)

I defined a utility class that contains most of the constants that i use throughout the notebook, i defined constants CALIB_MTX(transform matrix) and DIST_COEFF(Distortion Coefficients) that will be populated by an instance of this class. I also defined a utility function outside of this class for finding parameters required for camera calibration(Transform matrix and Distortion coefficients).

Calibration requires some destination points in undistorted images(Object ponts) and points that need to be mapped to(Image points).

Since we are dealing with a chess that has a well defined pattern(boxes and corresponding corners), we can generate a numpy multidimensional array(grid points) of size (9*6,3)(9 inner corners in x direction and 6 inner corners in y direction).

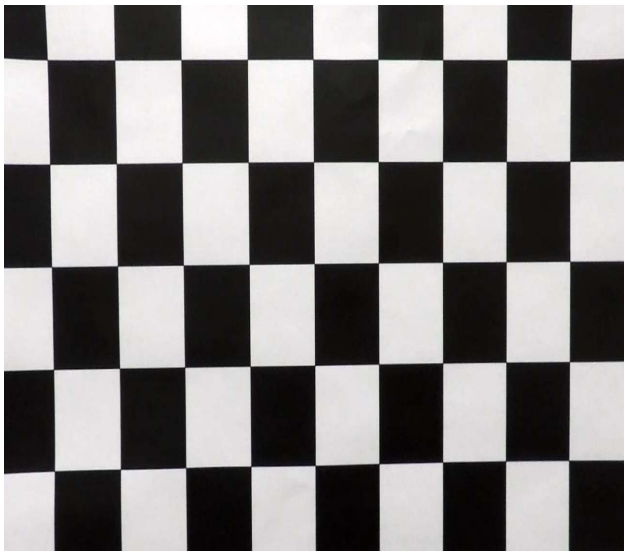
Inner corner positions inside the chess board can be considered as Image points . We can detect corners inside the chess board using OpenCv’s cv2.findChessboardCorners function.

Then iterate over calibration images present inside the “camera_cal” directory ,done using “glob” API(python library for iterating over files), collects image points and corresponding object points for every image.

We then pass these collected Object points and Image points to OpenCv's cv2.calibrateCamera() for computing transform matrix and distortion coefficients

, then populates the constants CALIB_MTX and DIST_COEFF.

Original Image

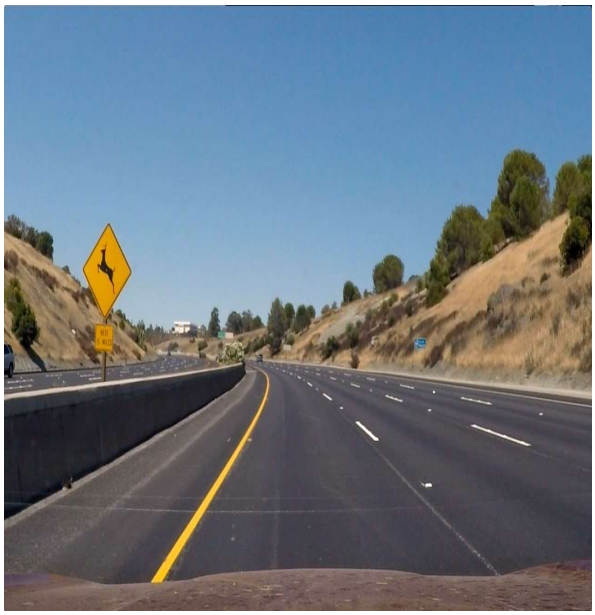


Undistorted Image

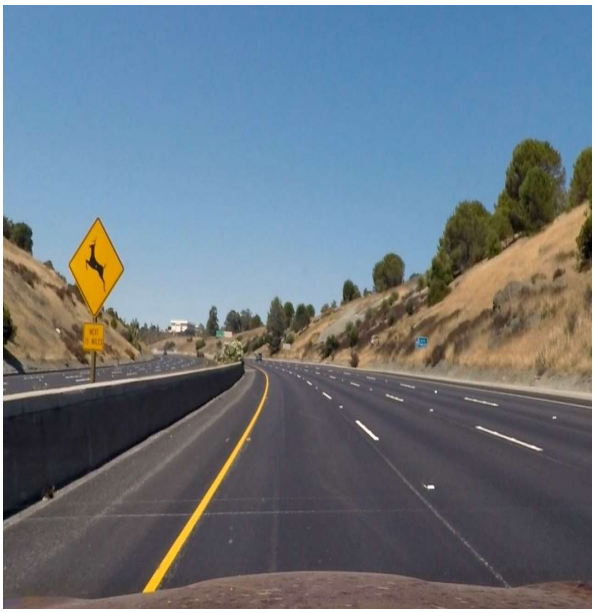
Software Pipeline

Undistorting The Images:

For Undistorting image i declared a separate utility function that takes an image as an argument and un-distort the image using OpenCV's cv2.undistortImage() , this function makes use of the transform matrix computed and distortion coefficients computed before to undistort the image.



Original Image



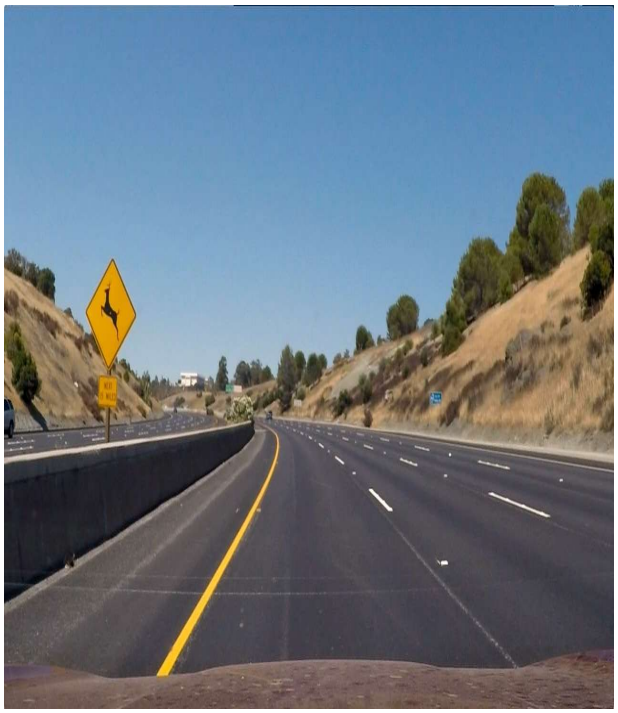
Undistorted Image

Color, Gradient, Magnitude Of Gradient, Direction Of Gradient thresholding :

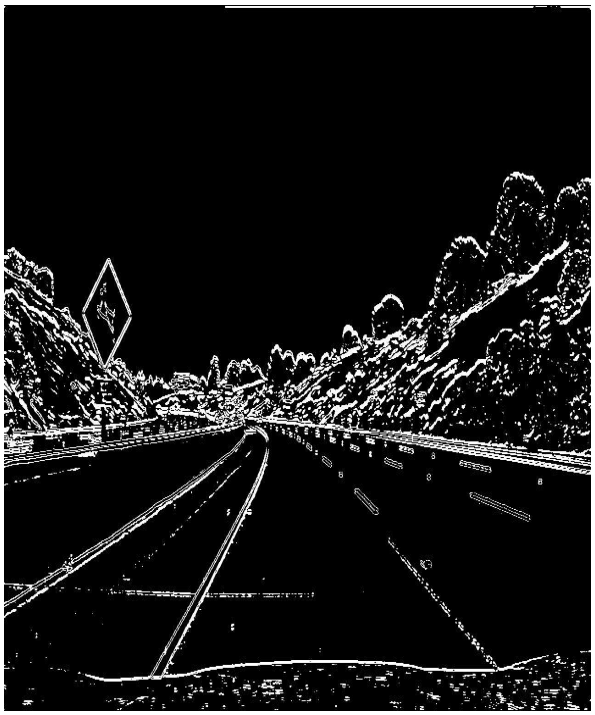
I defined separate utility functions for performing color , gradient , magnitude of gradient , direction of gradient thresholding and returning binary images respectively.

These thresholds are made globally accessible by the “Constants” class that maintains constants.

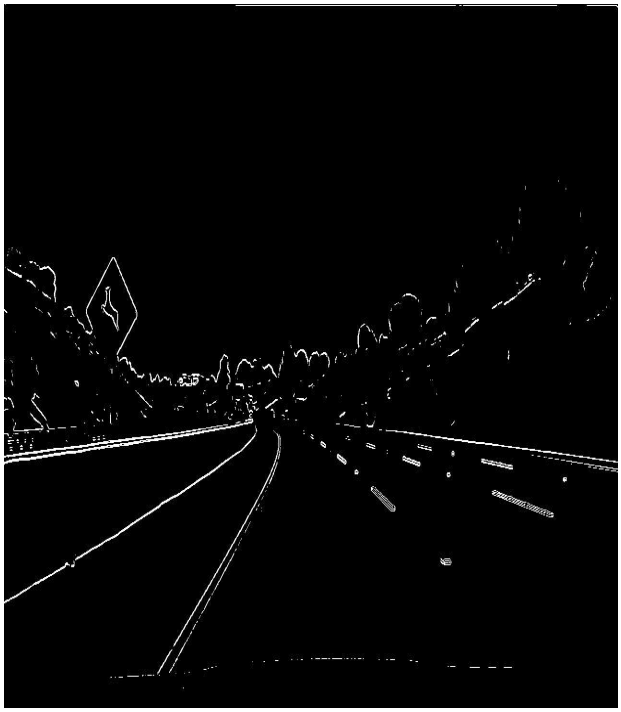
I then used these binary images returned by utility functions to combine into one image , obtained by performing some combination binary logical operations on these binary images , that have lane lines highlighted very clearly. This helps us to perform polynomial line fitting accurately without being much affected with the noise around lanes.



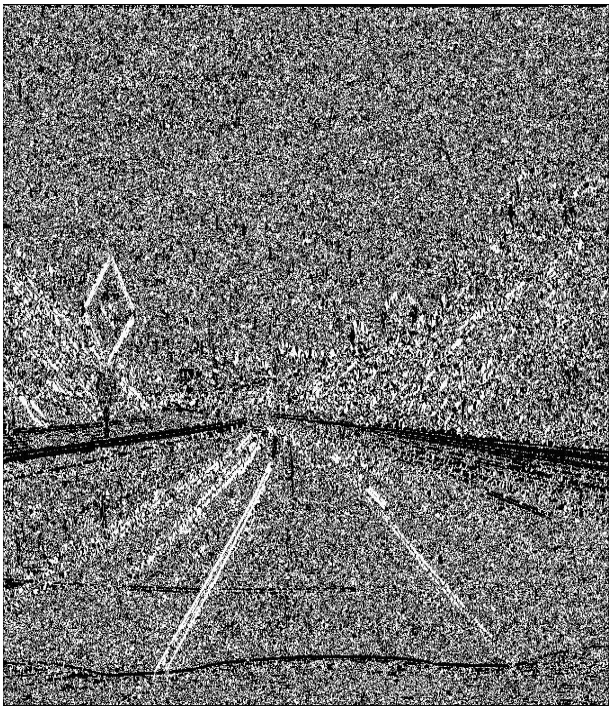
Original Image



Gradient thresholded Image(along x)



Magnitude Of gradient Threshold Image



Direction Of Gradient thresholded Image

Color Thresholded Image(S channel)

Final Combined Image

Perspective Transform of Combined Image:

Now i used “Combined image” for taking perspective transform of a region that contains lane lines.

This region is specified by some coordinates or pixel positions in the image called Source Points .

To specify what points to transform to we use destination points .

Then Perspective Transform is done by using a utility function “perspective_transform” that takes in combined image, source points , destination points.

This utility function uses OpenCv’s cv2.getPerspectiveTransform() function to calculate the transformation matrix by using source points and destination points, this transformation matrix is then used by OpenCv’s cv2.warpPerspective() function to take the perspective transform of the region specified by the source points and transforms to position specified by the destination points.

Source Points :

```
srcpoints = np.float32([left_top,right_top,right_bottom,left_bottom])
```

Destination Points:

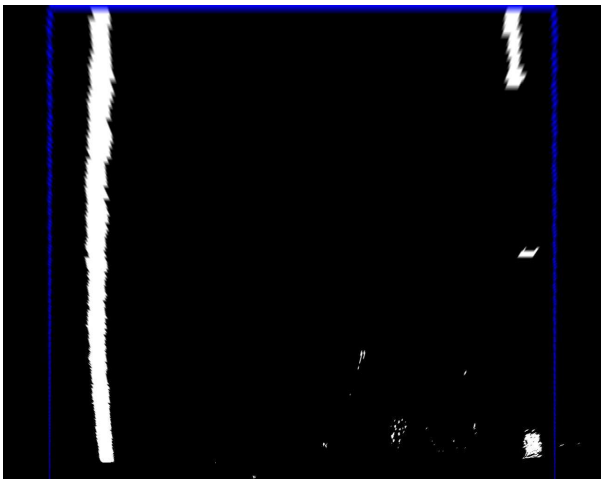
```
dstpoints = np.float32([[offset,0],[combined_image.shape[1] - offset,0],[combined_image.shape[1]-offset,combined_image.shape[0]],[offset,combined_image.shape[0]]])
```

Where:

```
xsize = img.shape[1]
ysize = img.shape[0]
left_bottom = [0.1*xsize,ysize]
right_bottom = [0.965*xsize,ysize]
left_top = [xsize//2.24,ysize//1.59]
right_top = [xsize//1.72,ysize//1.59]
```

Image With Region Indicated

(Lane Lines Appear to be converging)



Perspective Transformed Image

(Lines Are Parallel)

Polynomial Line Fit :

Now i use a utility function “polyline_fit” that takes in the perspective transformed image with lane lines to identify the activated pixels , color the lane pixels, and also fill in the lane portion using the polynomial x and y values, this function also calculates the radius of curvature of left and right lanes by using “radius_of_curvature()” utility function.

There are two methods to identify pixels corresponding to the lane and doing polynomial fitting, coloring. Second approach depends on the first Approach for functioning.

1) PEAK HISTOGRAM SEARCH:

- In this approach we search for the position of the pixel along the x-axis of the image where there is a high pixel density , then we define a window of a certain height and width to expand to a certain margin from the position of the pixel.
- Color pixels that are inside the window and also store the positions of the pixels present inside the window for polynomial fitting.
- We then take the average of the pixel positions for mounting the next window on top of it.
- We iteratively do the above steps for n windows. At the end based on the pixel positions collected we fit the polynomial .

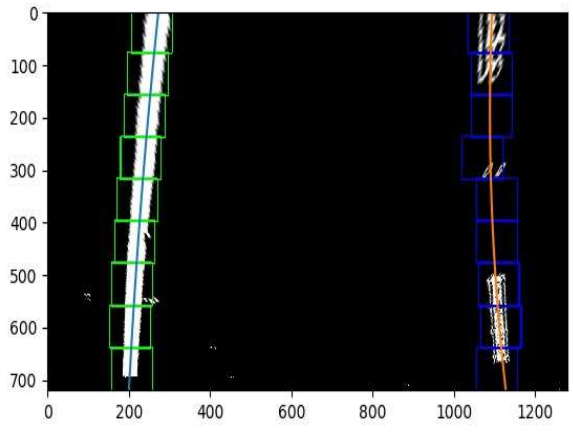
2) Local Search :

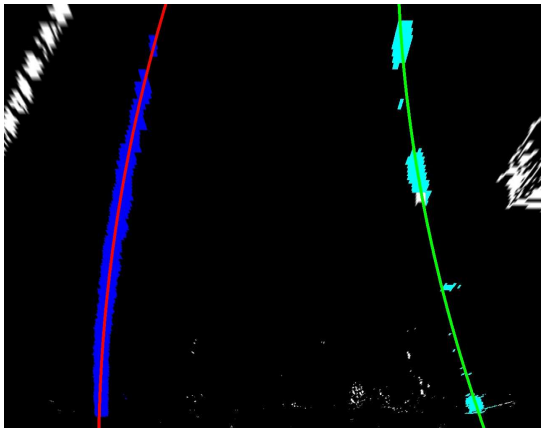
- Peak Histogram Search is expensive for processing video frames , because for each video frame we have to iterate all over windows again , which is not computationally efficient.
- Due to the fact that, for each frame there is no high change in the lane position , we can take advantage of this to do a highly targeted search.
- We take the parameters of the polynomial from the last frame of the video and search around it by some margin and then take the positions of active pixels inside this margin to fit the polynomial.
- This search was done from the second frame of the video , for the first video frame we have to fit a polynomial based on the Peak Histogram Search technique.

In order to switch between methods I have created a utility class called “Lines” that maintains previous frame’s polynomial line parameters , also stores the n previous frames polynomial line parameters for taking average and then using these as the polyline parameters for line fitting and searching around it by some margin in “Local search “ based polyline fitting algorithm.

Radius Of Curvature Calculation :

- Once I have the x positions of pixels that represent the lanes , I scaled the positions to meter.
- I then used numpy's "np.polyfit()" function on scaled positions to obtain the parameters of the lane lines.
- I used these computed parameters and also scaled the Y position that is nearest to the car's camera position to compute the radius of curvature.
- Lane midpoint is calculated using the left and right lane's polyline x position, Imagemidpoint is the frame's horizontal midpoint , then I took their difference , scaling this difference to meters gives the offset of the car with respect to the lane midpoint.





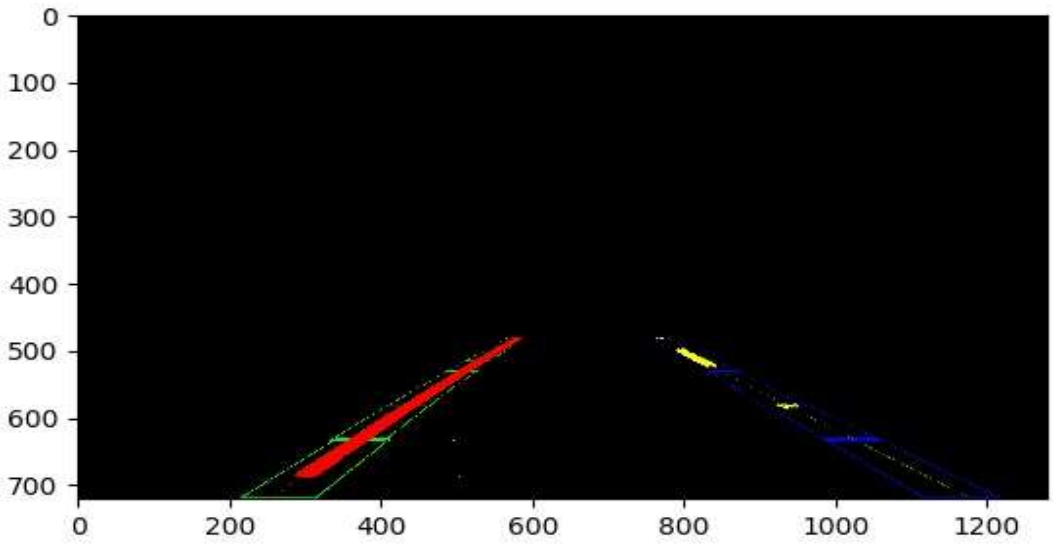
Peak Histogram Search Local Search

Inverse Transform Of Image with Lane Portion and Lane Lines Highlighted:

Now in order to overlay the perspective image with lane lines and lane portions identified onto the original image , first we need to take the inverse perspective transform of the perspective image inorder to match the spatial length and width of the lane present in the original image.

I used the same utility function that was used for computing perspective transformation of combined image but now the source and destination points are interchange because we are doing inverse.

Inverse Transform of Perspective Image with lane lines identified

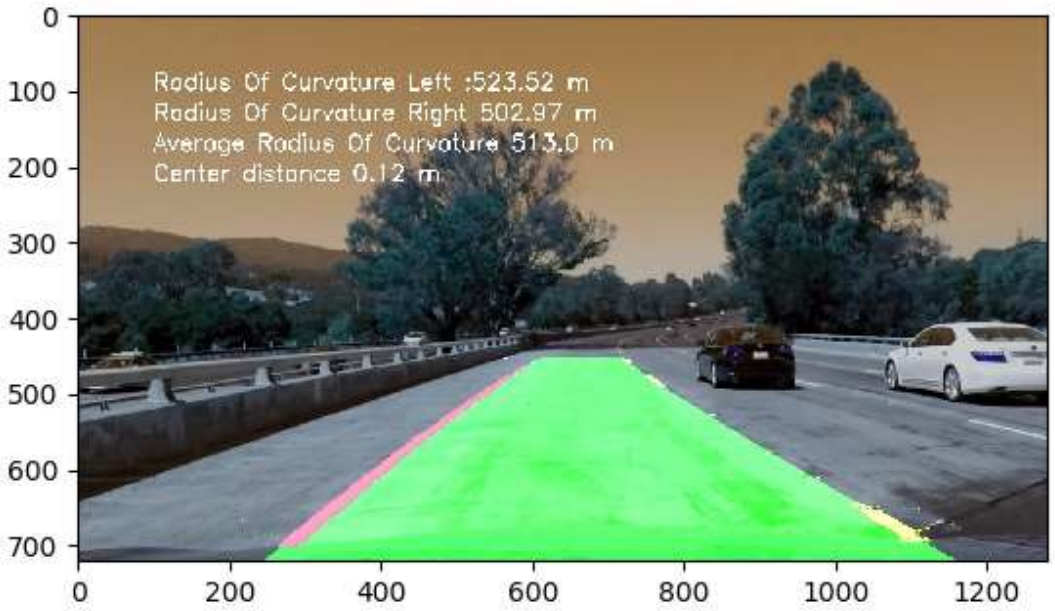


Blending Original Image with Inverse Transformed Image:

Inverse Transformed images have lane lines and lane portions identified and highlighted with colors . Now in order to have a visual feedback on how it is doing well we have to overlay this image onto the original image.

For overlaying images I used OpenCv's cv2.addWeighted() . For overlaying text indicating the radius of curvature and offset of the car with respect to the center of lane I have used OpenCv's cv2.putText() function.

Output of the entire pipeline applied to a single image:



Procedure followed for applying above pipeline for a video stream:

I have used MoviePy python library for loading the video clip and also their "fl_image" module for applying a function for each frame of the video .

I have defined a utility function that takes in each frame of video and calls the "software_pipeline" with argument beings current frame(image) and other being a boolean

value that specifies whether to use “Local search” or “Peak Histogram Search” based on the value of polyline parameters collected in “Lines” utility class.

Output Of The Software Pipeline when applied to a video stream:

Link to the final output of the software pipeline applied onto a video stream:

https://youtu.be/c3_PxmGNn-g

Shortcomings of this algorithm for lane line detection and lane portion segmenting:

- This is an image processing based lane annotation that does not have any sort of intelligence to identify the lane in highly stochastic real world scenarios like, when there are other objects on the lane such as other cars, etc.
- This approach just detects the lane marks , but this should also detect other marks on the road such as turnings, zebra crossing and segment them properly with some colors that make sense.
- Output of the software pipeline when fed with challenge video clearly shows that when there are more than two lane lines (common on road when it splits into two ways) our detection confuses to select one and fits the best for the two lanes.

Some of the ways to overcome above shortcomings:

- Our system should identify other objects and then intelligently overlay the lane region, this can be done using CNN's and other deep learning models based on CNN.
- Intelligent road sign marks detection can be done using Convolutional neural networks based image segmentation.
- When there are more than two lane lines our model confuses to select one, this can be mitigated by accurately selecting the region for taking perspective transform , this eliminates other lane lines.

Challenge Video Output.

As shown this software pipeline does not work much robustly in detection of lane lines due to much stochastic environment and requires a high time in tuning the thresholds to a finer precision for accurate detection of lane lines by eliminating most of the noise.

Link for output of the software pipeline applied on a challenge video stream:

https://youtu.be/Yr946y7LqdE

Milestones

This project is an extension to the previous lane detection that was only able to detect the straight lane lines ,but the algorithm now developed will be able to detect the curved lane lines and also able to detect the radius of curvature that will be used to steer proportionally using controllers.