Advanced Lane Finding

Udacity Self-Driving Car Nanodegree Program

Sulabh Matele - May 2, 2017

Advanced Lane Finding Project

The goals / steps of this project are the following:

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

Submission contains following different files and folders:
- **advancedLaneDetect_Pipeline.py** - This file contains the main implementation of project pipeline and also contains all the data structure and methods used.
- **project_result.mp4** - This is the final output and processed video of the detected lanes.
- **visualize_results.ipynb** - This file visualizes the images at different stages of the pipeline.
- **AdvancedLaneFinding_Writeup.pdf** - This document
- **Processed_Images** - This folder contains sample processed images at different stages of project pipeline.

Rubric Points

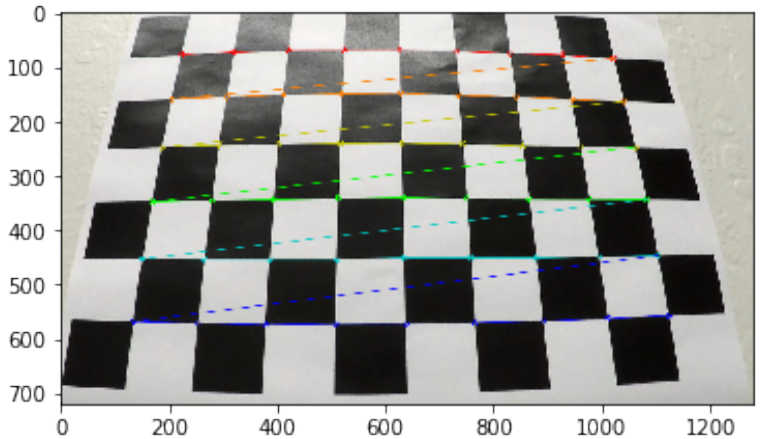
Camera Calibration and distortion removal from Image

The code for this step is contained in file "***advancedLaneDetect_Pipeline.py***" at line number 109 and line number 143 with methods "calibrateCamera()" and "undistortImage(img)".

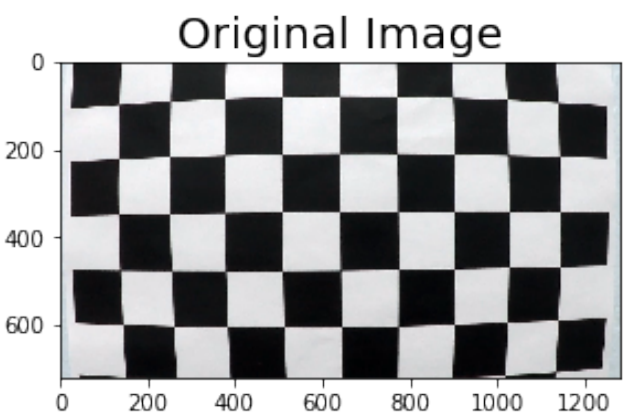
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image.

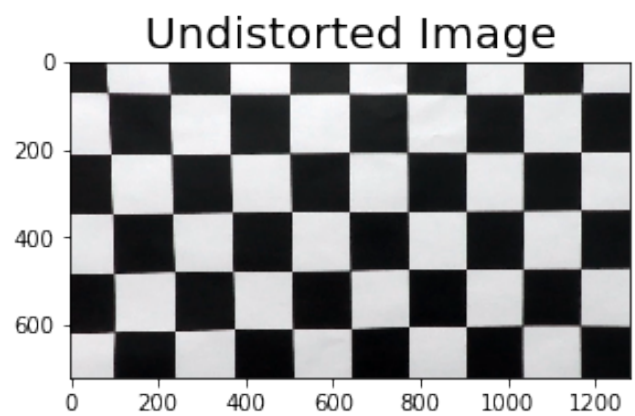
`imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

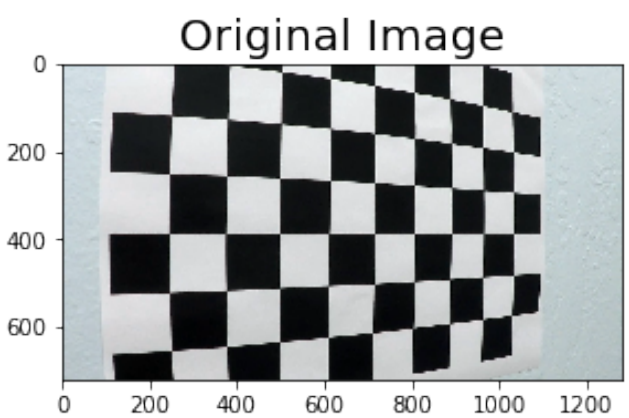
Following is the result of corner detection:

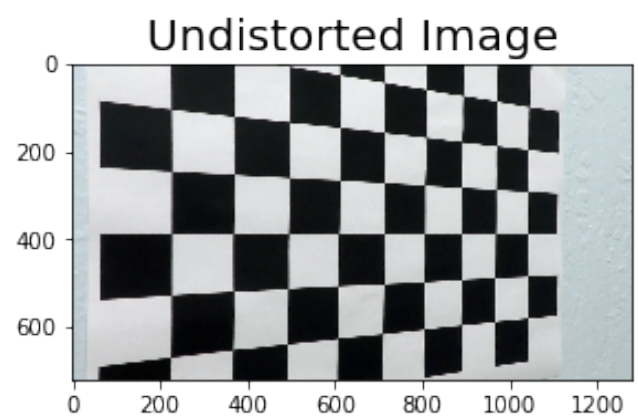


Then I used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

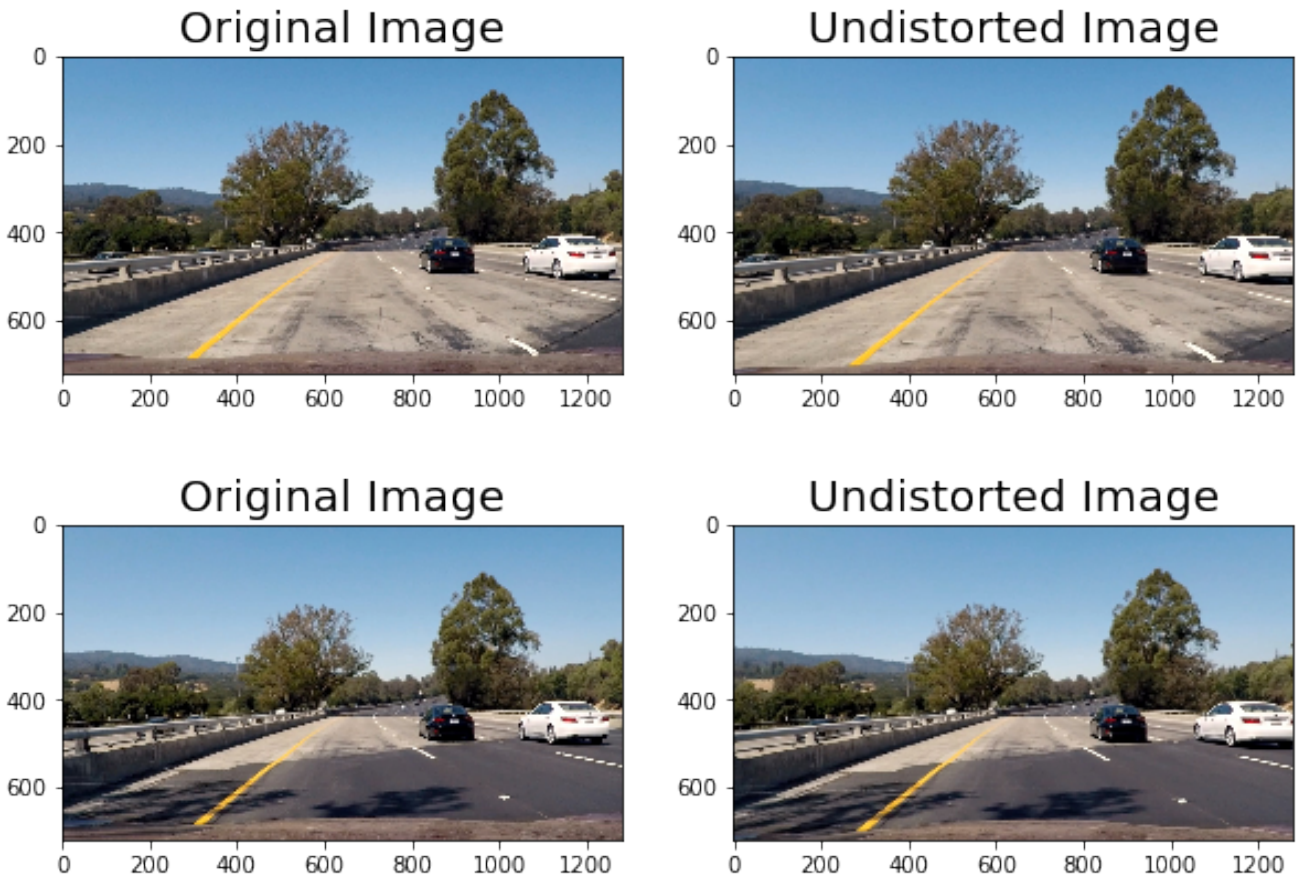








To demonstrate that the , distortion correction works on test images also, I applied the same on provided test image and following is the result:



More results and the associated code can be visualized easily in "**visualize_results.ipynb**".

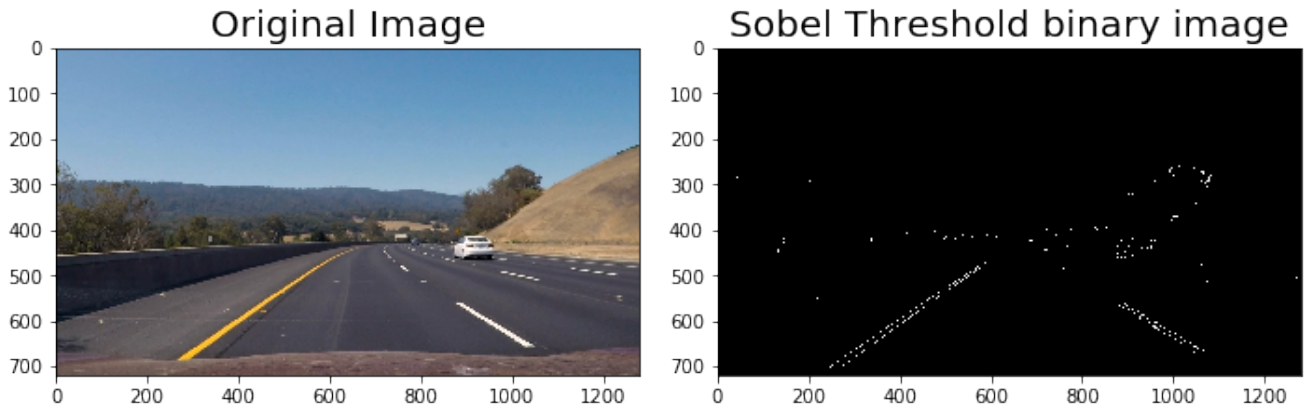
Color transform, Gradient and Binary image creation:

Now when I have undistorted image, I need to find lane lines accurately, but also I need to remove the undesired information from the image as tree, road color, vehicles, sky etc. and detect the lane lines.

I tried different techniques of color transformation and gradient calculation to visualize the results and find the best technique or combination of techniques, which can detect the lane lines accurately, and converted them to binary image to filter the undesired information.

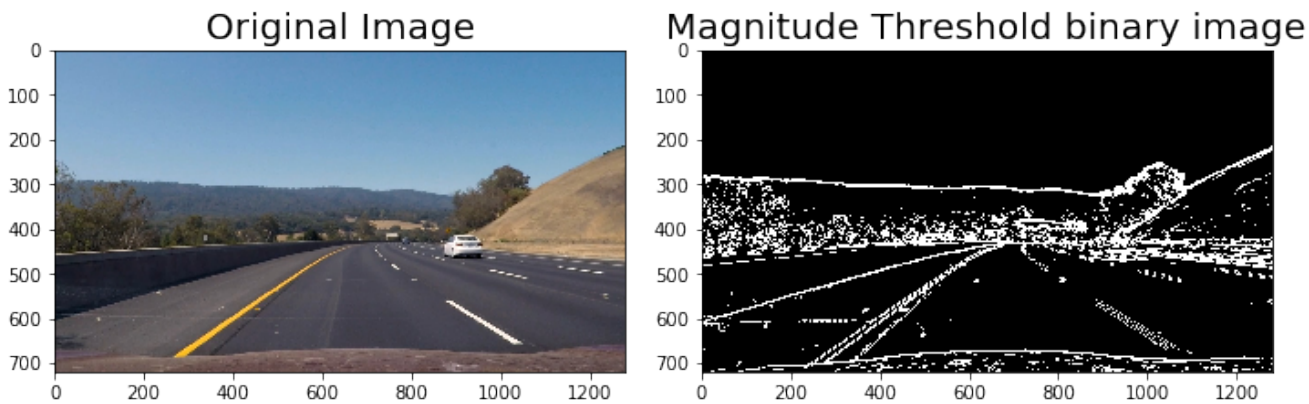
First I tried the Sobel transformation in X and Y orientation and also thresholding the same to preserve only the desired information on the image.

Following is the result:

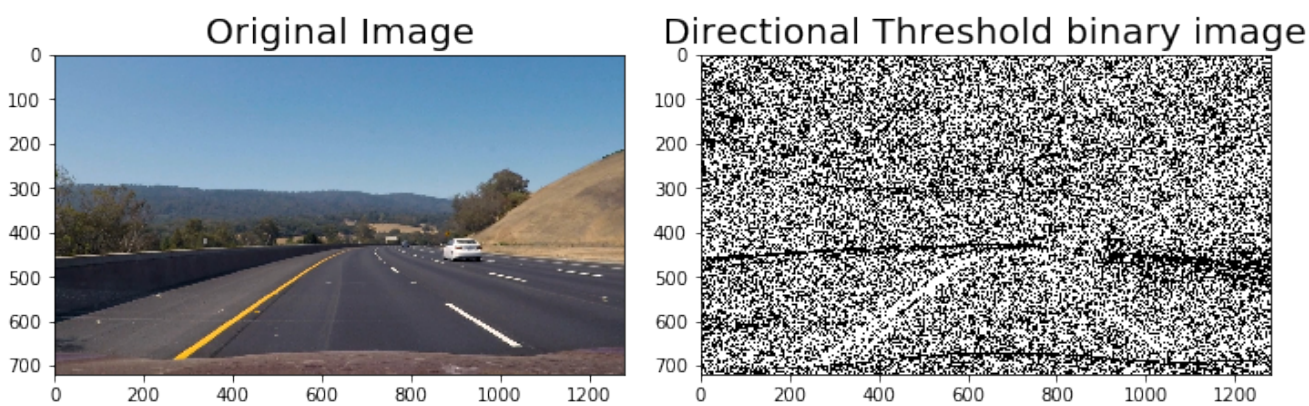


Then different techniques results:

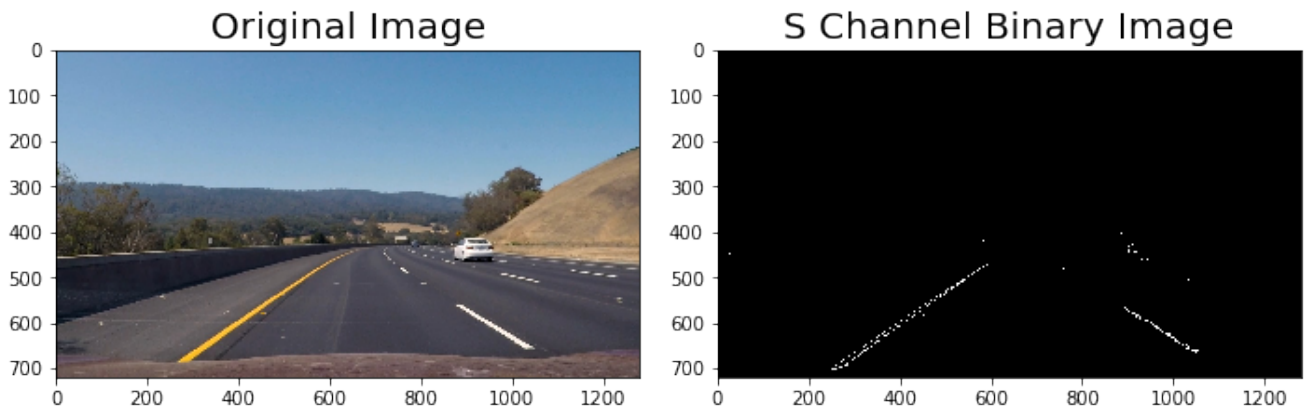
Gradient magnitude mean with thresholding:



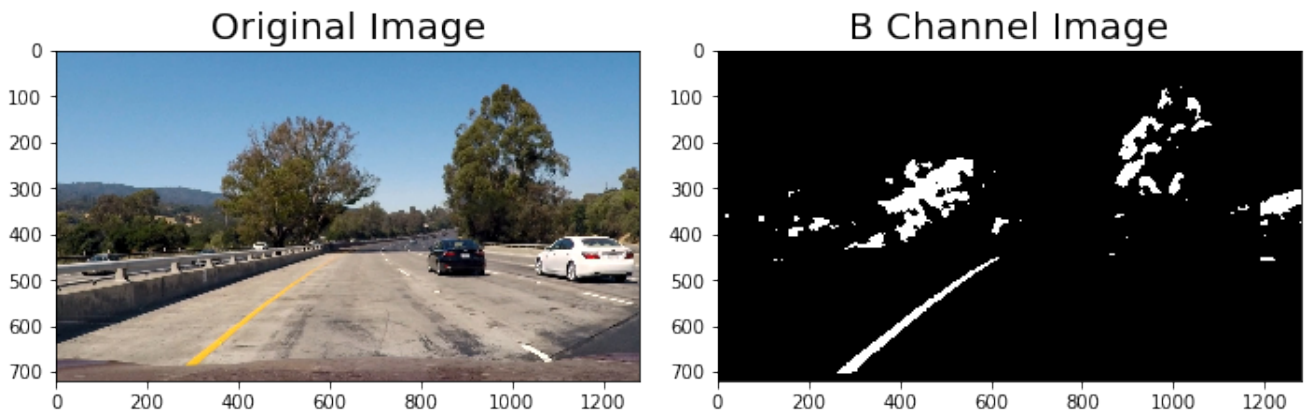
Directional Sobel and thresholding:

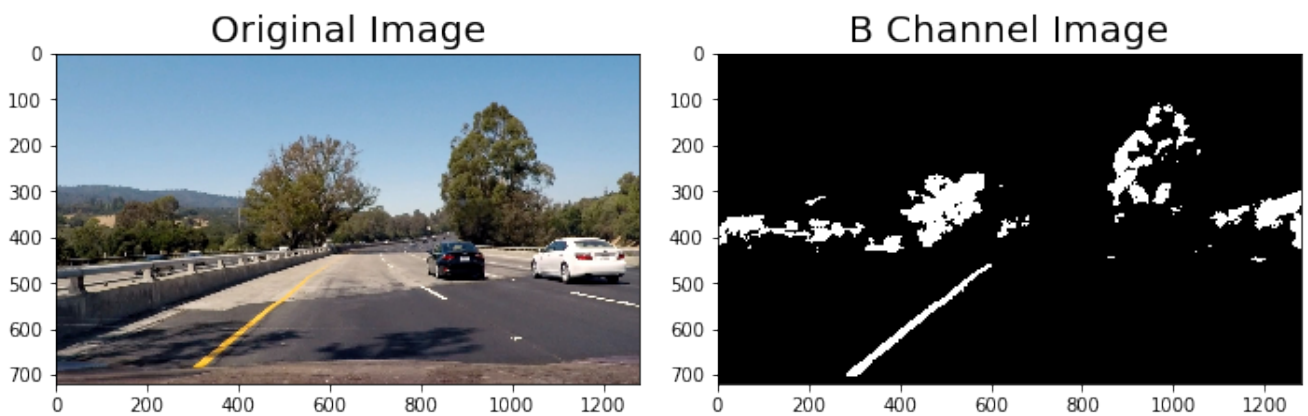


S Channel image transformation from HLS color space and thresholding:

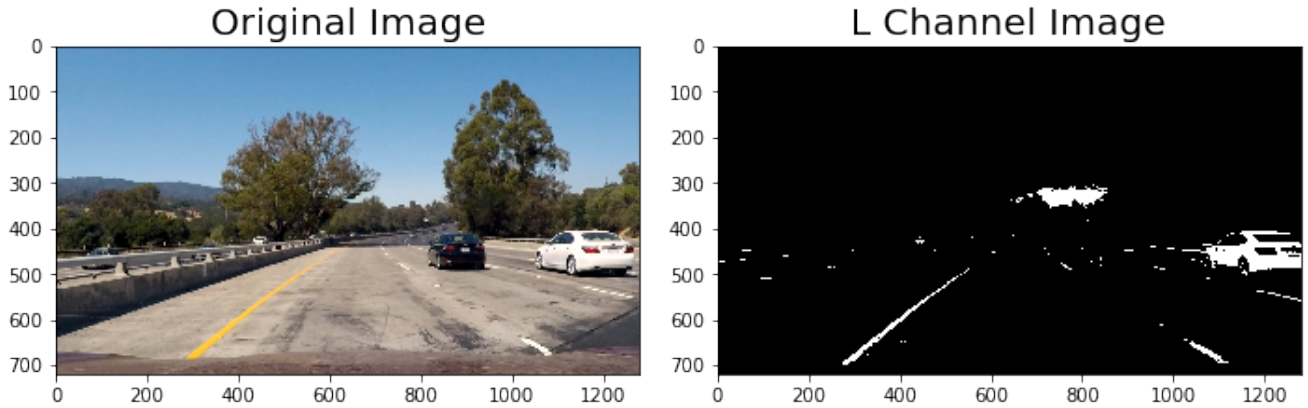


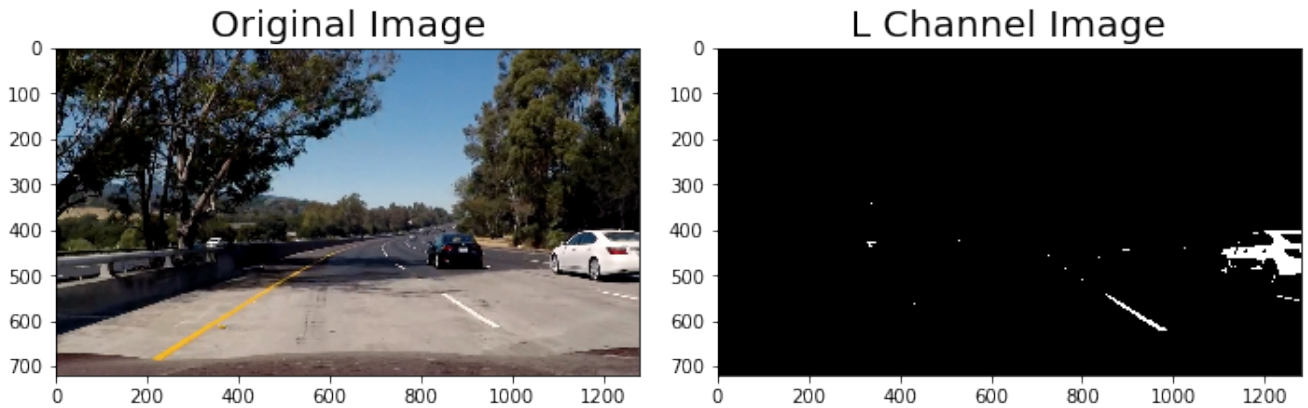
B Channel thresholded image from LAB color space:





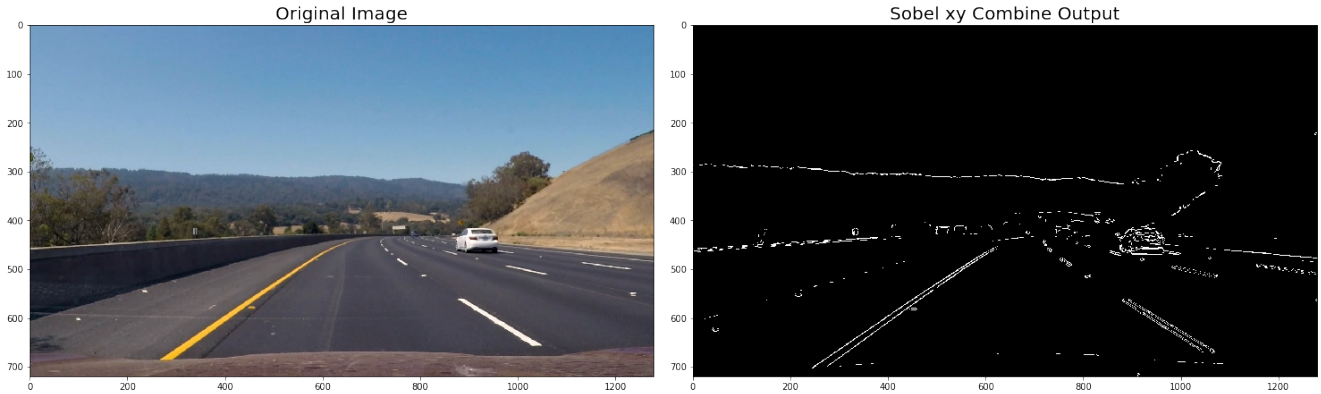
L Channel thresholded image from LUV color space:



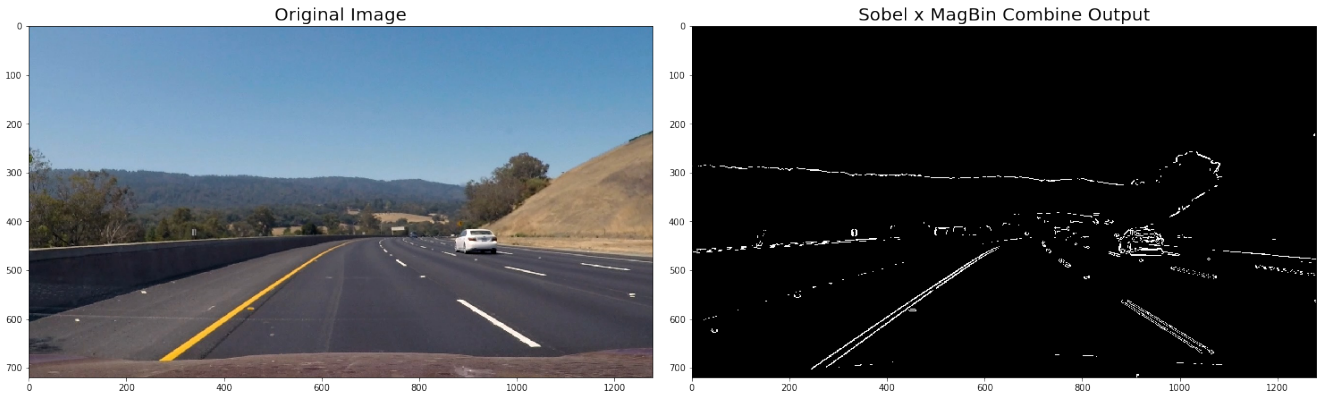


No single technique was able to find the lane lines very accurate for different images, although some looks just ok to be used.

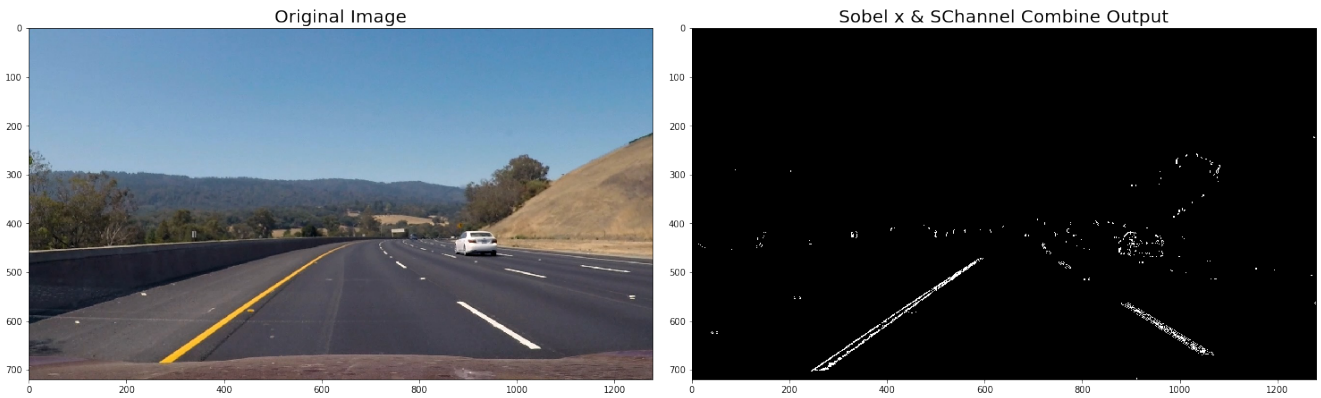
It was noticeable that L channel binary image from LUV and B channel binary from LAB color spaces were able to detect yellow and white quite clearly. So I started using the combination of techniques to get the better results and following are some outputs:
Combine Sobel X and Y results:



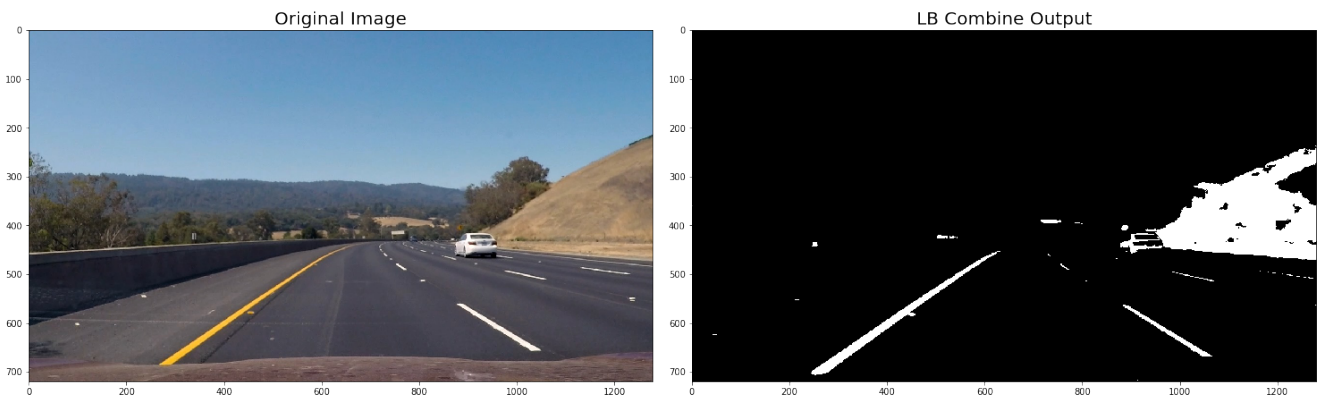
Combine Sobel X and magnitude gradient:



Combine Sobel X and S channel transformed:



Combine L and B channel transformed:



By the observation on different test images and trying different combinations, I selected to use combination of LB channel combine image results which worked better than other combinations to detect Yellow and White lane lines accurately over different test images.

(Code: File: “advancedLaneDetect_Pipeline.py” line numbers 148, 158, 168 - methods: “getBChannelImage”, “getLChannelImage” and “combineBinaryImages”)

More results and images can be found in “visualize_results.ipynb” and “Processed_Images” folder.

Perspective transform

Now when I have filtered out the undesirable information, I need to also make sure that the information on each image for lane is correctly identified.

Perspective transform is a technique which changes the view angle of a particular area of the image. So we select the big enough polygon on our image to accommodate sufficient points for detecting lanes for right and left turns and then apply perspective transform to view it from top, also known as bird eye view.

Perspective transform takes the source and destination coordinates of the source and transformed image and performs the perspective transform.

(Code: File: “advancedLaneDetect_Pipeline.py” line number 178 - method: “perspectiveTransform”)

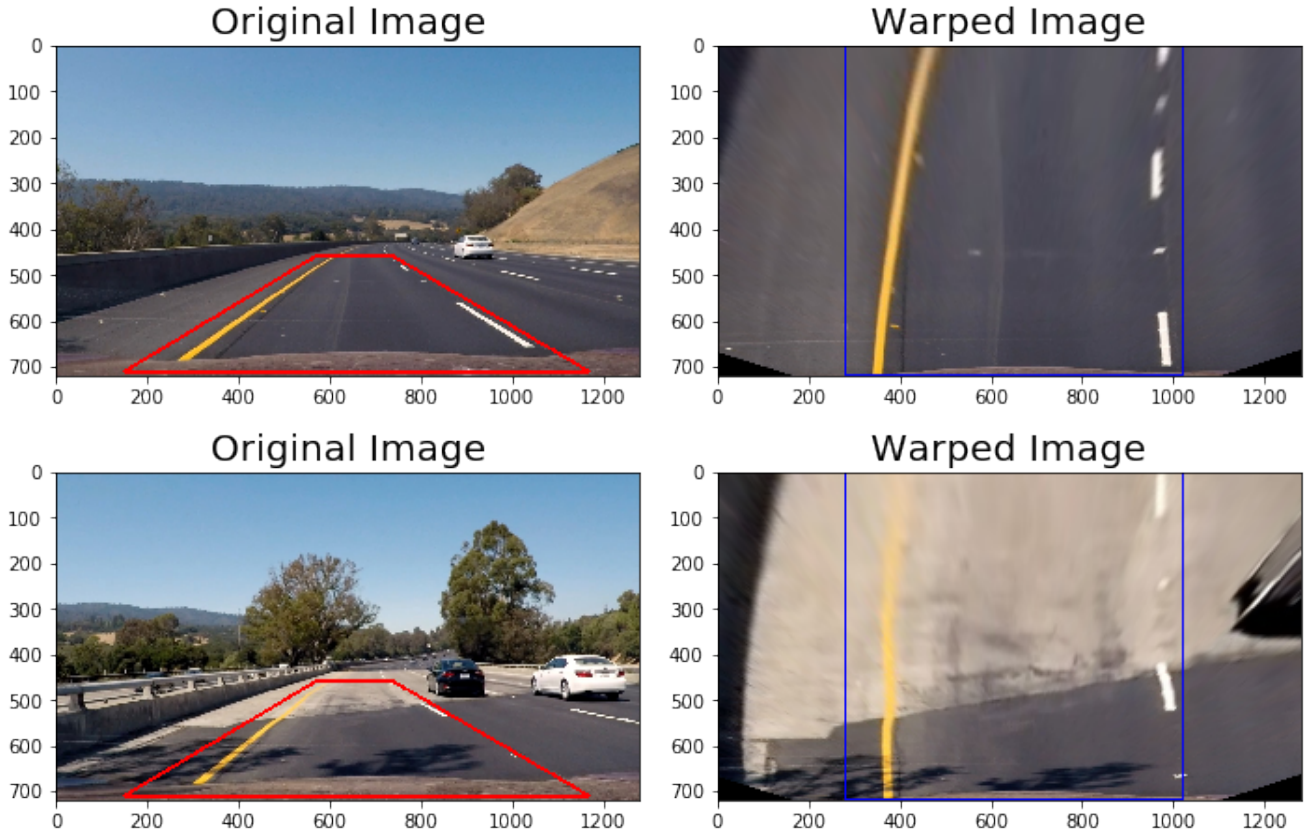
The method returns the transformed image which is also called as the warped image.

Following source and destination points I have used in my code:

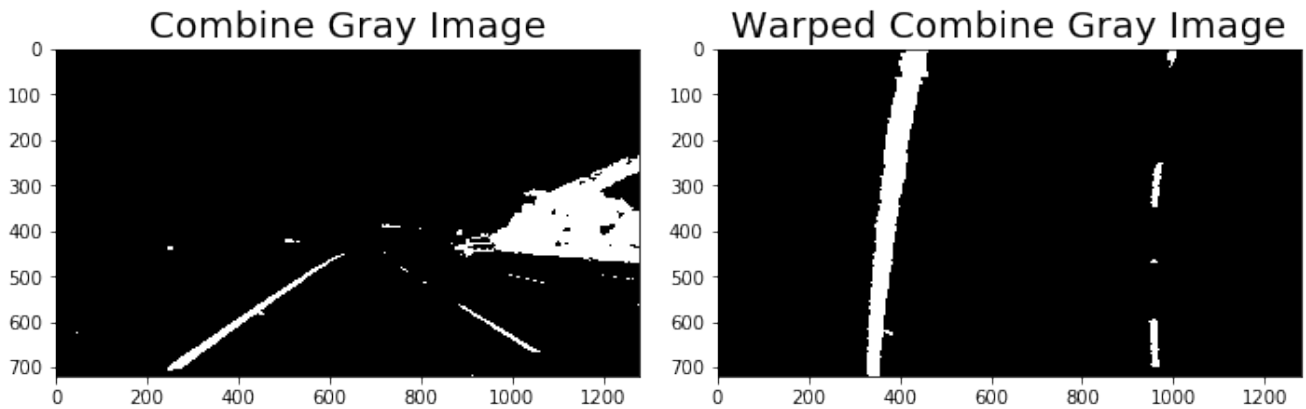
```
src = np.float32(
    [
        [1210, 710],  # Bottom right
        [160, 710],  # Bottom left
        [580, 460],  # Top left
        [735, 460]  # Top right
    ])

dst = np.float32(
    [
        [1020, 720],  # Bottom right
        [280, 720],  # Bottom left
        [280, 1],  # Top left
        [1020, 1],  # Top right
    ])
```

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.

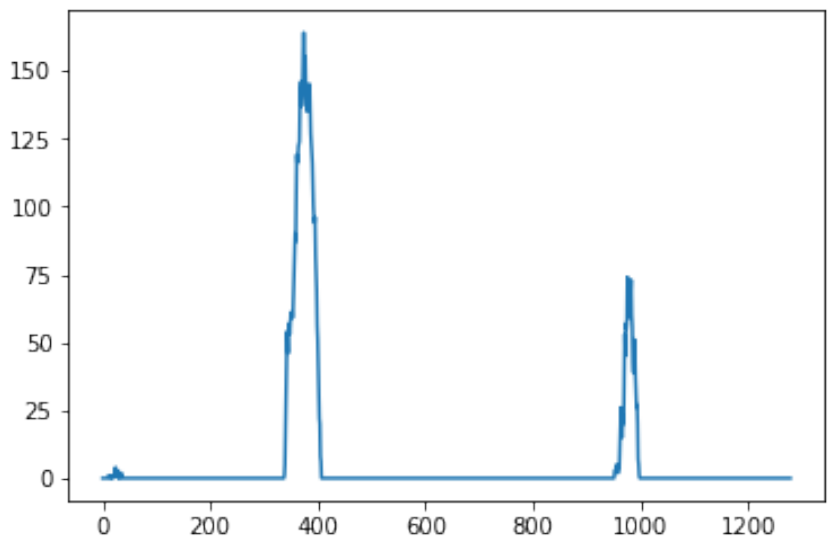


Once it was verified that the perspective transform works, I applied the same on the combined transformed binary image and following is the result:



Histogram:

Now when I have perspective transformed binary image so I can start searching for lane lines, I used the histogram of the image to find out where are the 2 peaks of the given image on lower right and left side:



Once we know where are the 2 peaks of the image, we can start the searching from those points and search unto the top of the image to map the lines on the image.

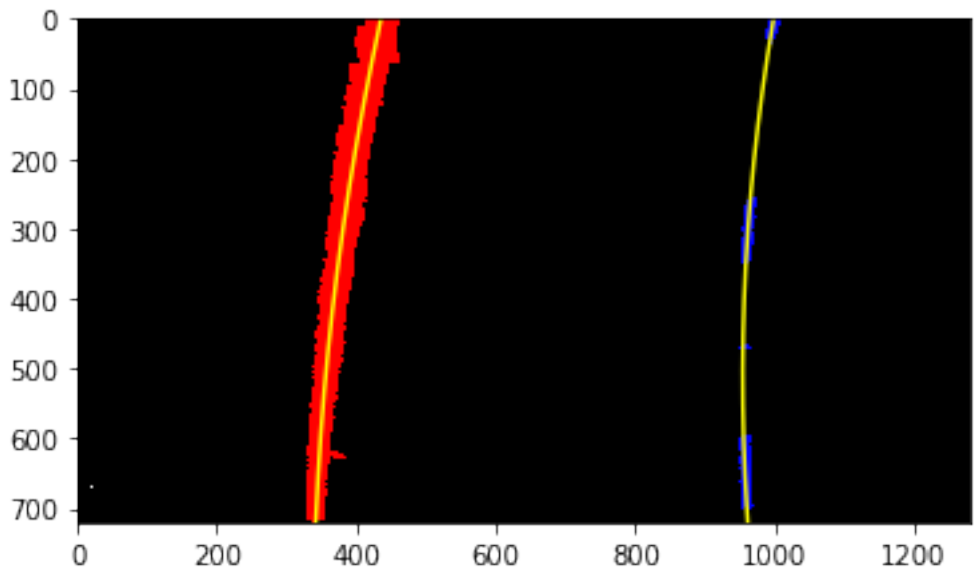
Main algorithm to detect and map lane lines

The main algorithm and function to detect lane lines is implemented as:

(Code: File: “advancedLaneDetect_Pipeline.py” line numbers 336, 201, 263 - methods: “findLaneLines”, “slidingWindowSearch” and “knownLaneDetection”)

The main concept used here to detect and trace the lane line is called as sliding window method. In sliding window, we divide our image in number of windows and for each window we try to find the nonzero X for corresponding nonzero Y. Once we have all the points on the windows we try to fit the second order polynomial and trace the same.

Histogram peaks are the starting point for our search on left and right side.
Following is a small visualization of polynomial fit:

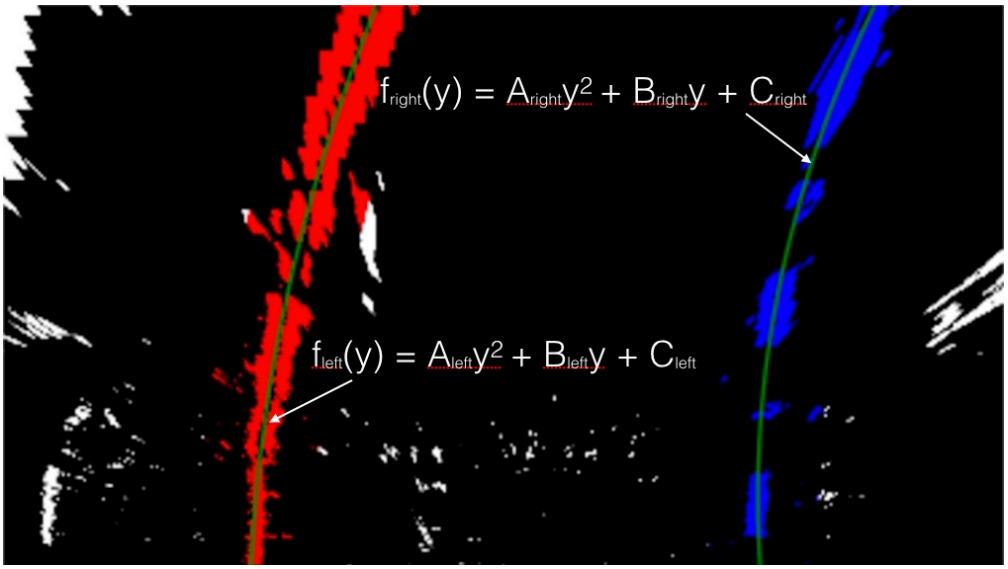


Once we detect the lane lines, from next frame onwards we know where to start searching and hence we perform the intense search around those points.

The decision algorithm is also developed to keep track that, we can rely on the results of the last frame, hence we should start the fresh search or keep searching near the old results.

The decision algorithm also takes care of storing the data on a instance of class to easily use for next frame. (method: “findLaneLines” implements the same algorithm), it also discards the bad frames.

Lane Curvature and distance from the center



When I had fit the polynomial, I used following ref. to calculate the radius of the curvature, when I received the left and right curvature, I averaged out he curvature to calculate the radius from the center of the lane.:

http://www.intmath.com/applications-differentiation/8-radius-curvature.php

Then I also calculated the distance from the center by assuming that the camera is mounted at the center of the car, and I know where is the lane center pixel in my image.

(Code: File: "advancedLaneDetect_Pipeline.py" line numbers 302, 325 - methods: "getCurvature", "handleVehPositionFromCenter")

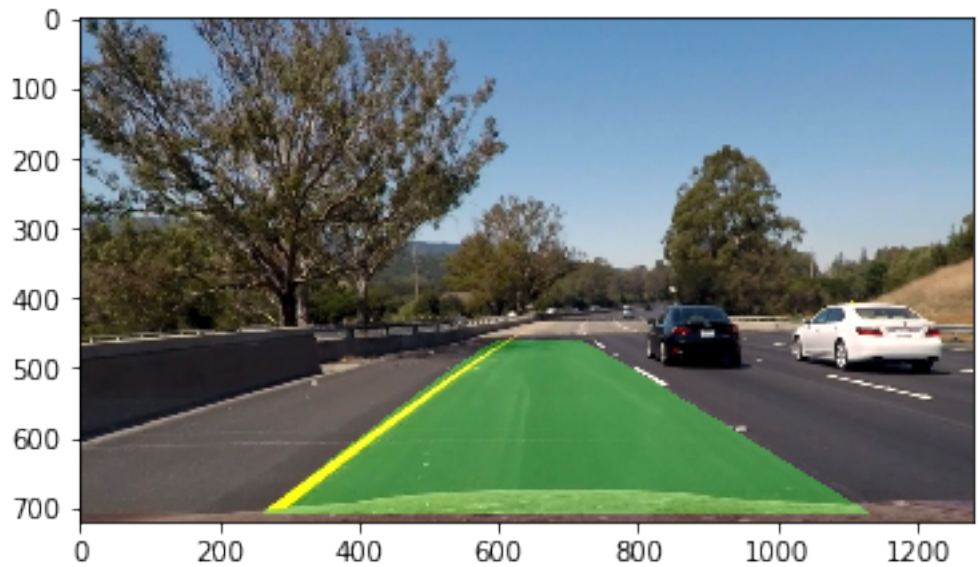
The results of lane curvature and distance from the center are stored in Class object to use further.

Final image preparation

Once I have the detected lane and nonzero points on second order polynomial, then I used them to draw the final image with the lane area high lighted.

(Code: File: "advancedLaneDetect_Pipeline.py" line numbers 457- methods: "fillFinalImage")

Following is the resulted image:

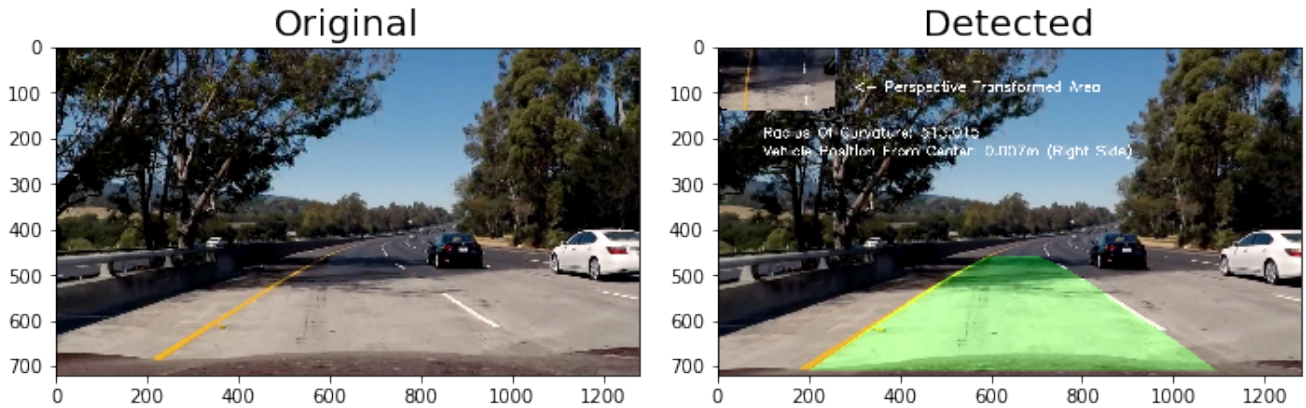


Pipeline processing

The above all methods and the sequence of calling the correct method is done and the final pipeline method takes the input image and results the output image.

(Code: File: “advancedLaneDetect_Pipeline.py” line numbers 501- methods: “findLaneLinesPipeline”)

It also uses method “processForDetails” line number 484, which writes the relevant information on the final output image. The result of pipelined image is :



Pipelined video

Pipelined output video is attached as submission. Following is a screenshot from the video:

Discussion

There were many different situations where the lane line was either not detected properly or I need to tweak the parameters to make it working.

Even though this pipeline works for most of the video for this project but it can further be improved by more exploration in the area for image transform and I can also improve the decision algorithm which I developed.

The pipeline can be tried to dynamically decide which transform to use and which works well for different situations. One transformation might not work for all kind of road and environment conditions.

References:

— Udacity class room pseudo code helped a lot in giving directions to complete this project in the given time frame.

— stackoverflow.com - for different image processing and modification techniques.