ADVANCED LANE DETECTION

Files Submitted:

- Advanced Lane Detection Package:
    - ImageReader.py – Contains the code to read images
    - ImageTransformer.py – Contains the code to Calibrate Camera, Undistort Images and Applying Perspective Transform
    - ColorThresholding.py – Contains code to apply various color thresholds
    - SobelThresholding.py – Contains code to apply various Gradient thresholds
    - Thresholding.py – Contains the code to combine Color and Sobel thresholded images
    - LineMaintainer.py – Contains the class to maintain the line details
    - BlindLineFinder.py – Contains the code for Blind Line Searching
    - ConfidentLineFinder.py – Contains the code for skipping the blind line search and search around a margin from the previous fit
    - **LaneDetect.py** – The main file where the pipeline is constructed and run.
- output_images folders contain the results of running the pipeline on test images
- result.mp4 contains the project output video
- writeup_report.pdf contains the project report
- dist.npy, mtx.npy, imgpoints.npy, objpoints.npy contains the distortion coefficients, camera matrix, image and object points respectively.

Camera Calibration:

The code for this step is contained in the ImageTransformer.py

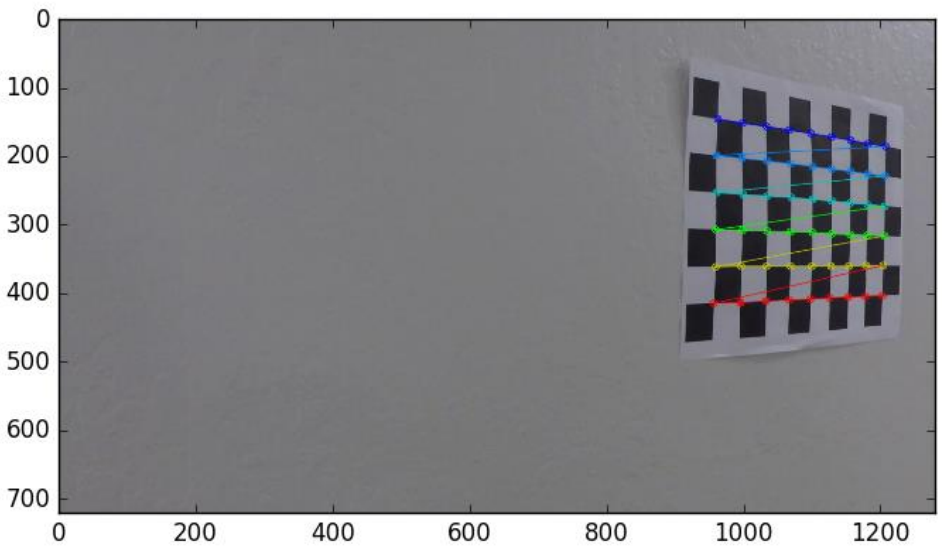
Lines 15 – 31 contain camera calibration code

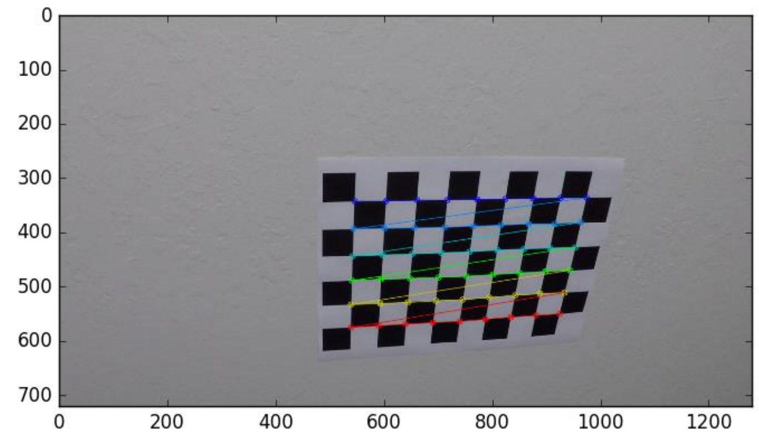
Lines 33 – 45 contain un-distortion code

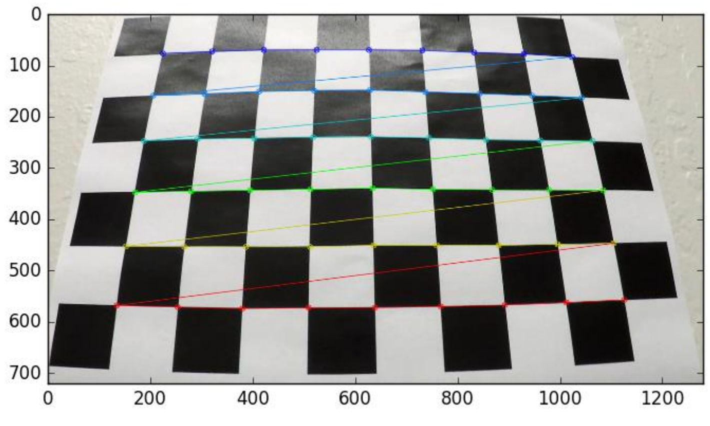
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, objp is just a replicated array of coordinates, 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.

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.

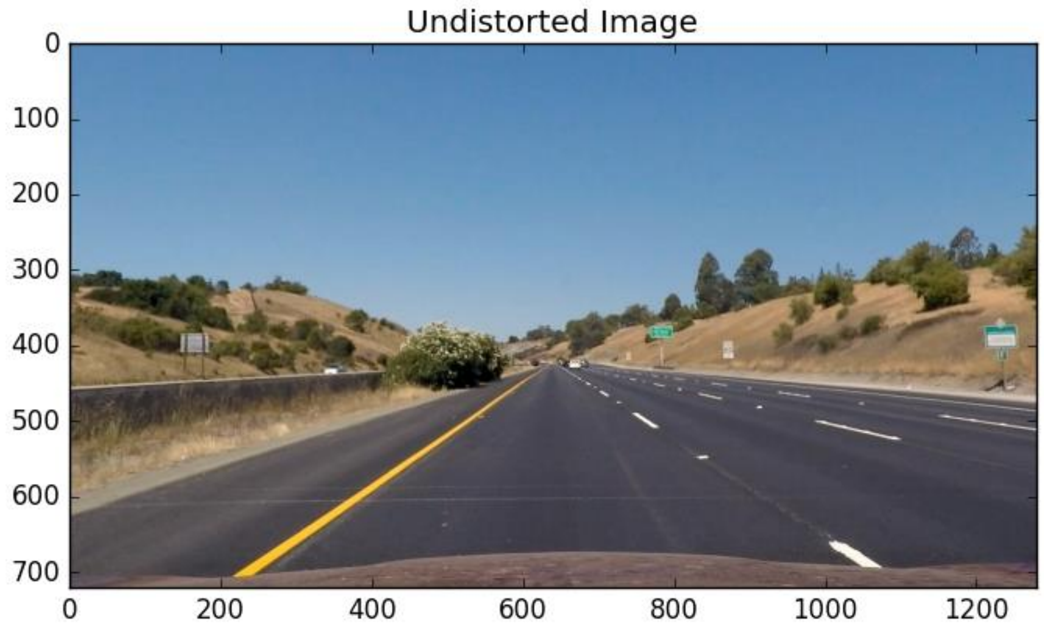
Here are some the of the chessboard images used for camera calibration with corners drawn:







Here is the result of undistorted test image:

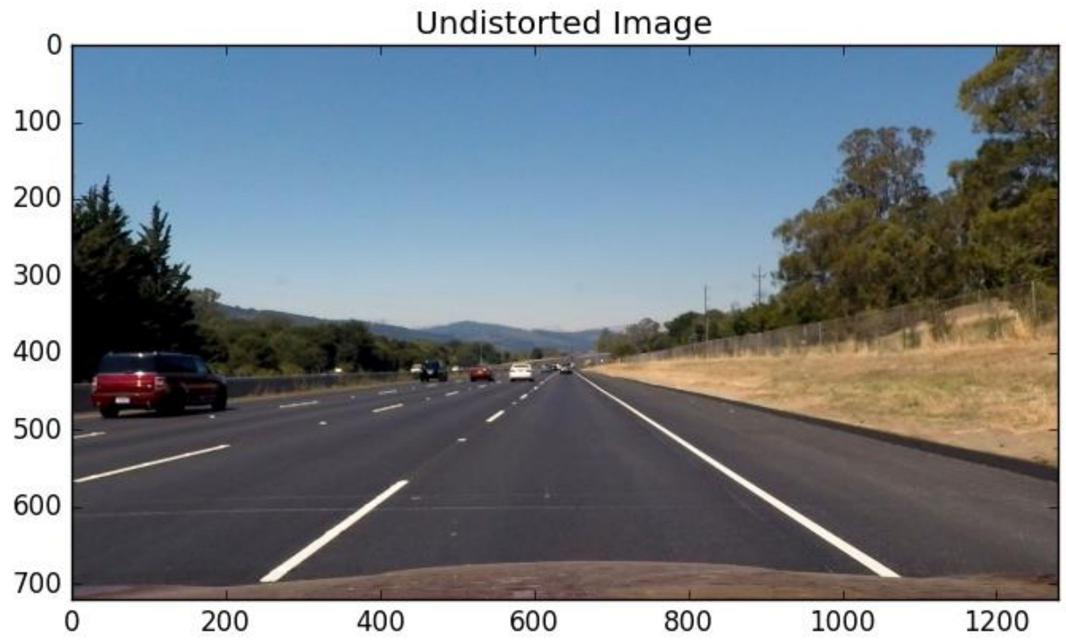


Pipeline:

- **Provide an example of a distortion-corrected image**

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

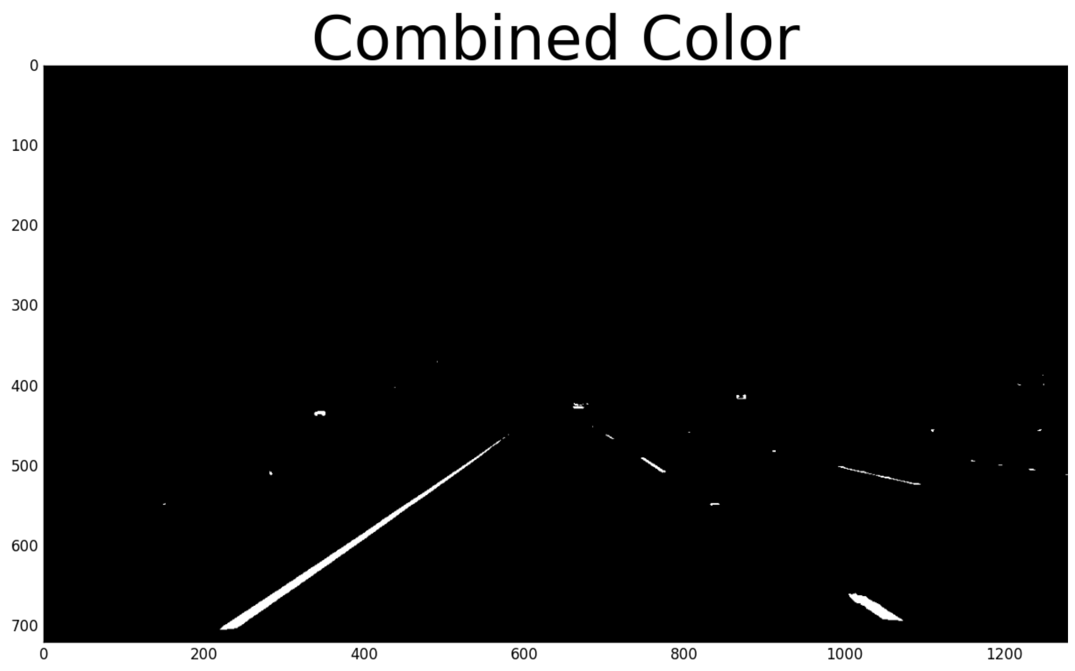
I use the camera matrix and distortion coefficients generated in the camera calibration step and use pass them to undistort function in ImageTransformer.py (Lines 33 – 45) to undistort the image and generate the following output:



- **Creating Threshold Image**

  I used a combination of color thresholds and Sobel thresholds to generate a binary image.
  The various threshold values used are in Thresholding.py – Lines 7 - 12
  Each of the steps are explained below:

  - **Color Thresholding:**
    The code for Color Thresholding is in ColorThresholding.py
    I used the HSV & LUV color spaces in the process. I picked out the V channel from HSV and L channel from LUV. Then I thresholded the images with corresponding values. I combined the two thresholded images and generated a color threshold image as shown below:

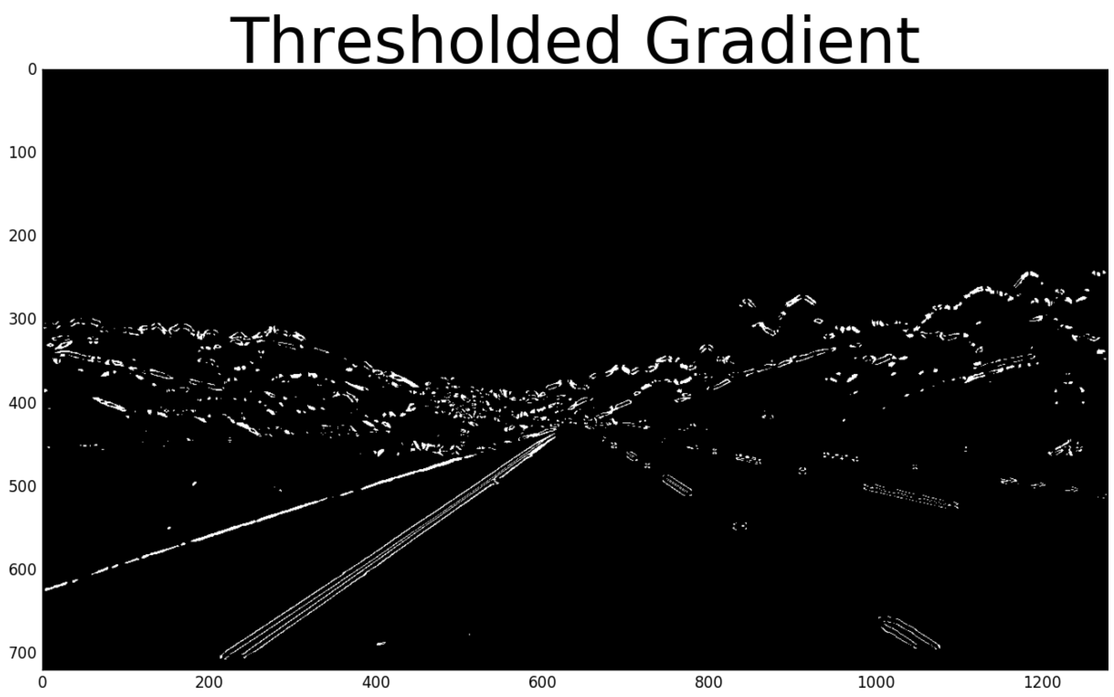


- **Sobel Thresholding:**

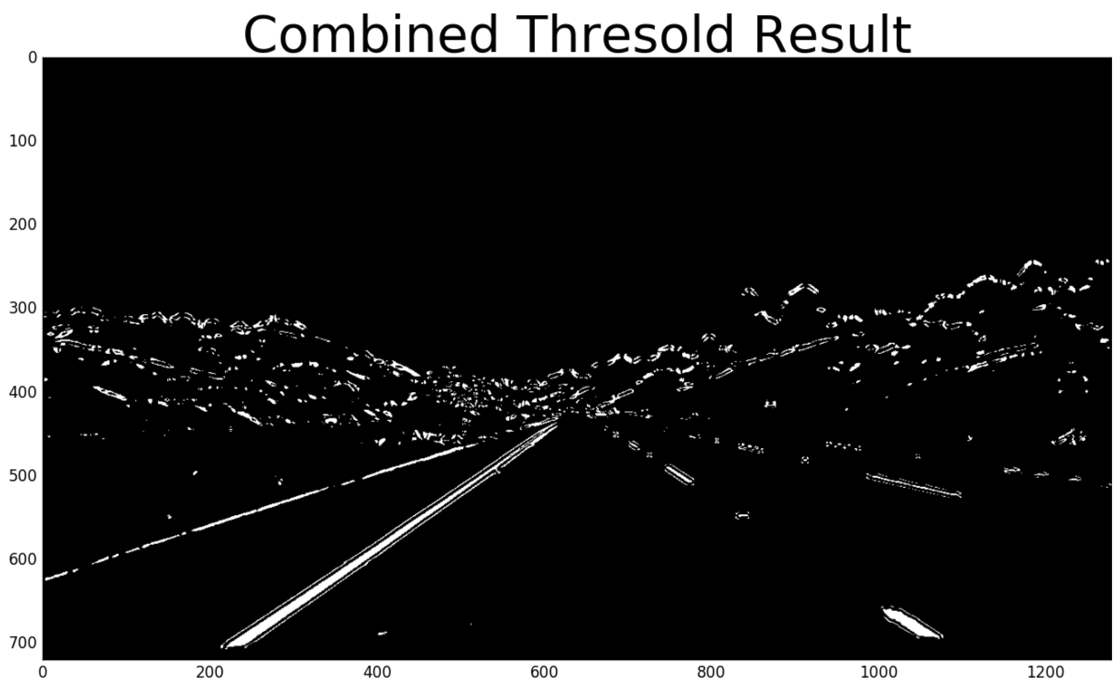
The code for Sobel Thresholding is in SobelThresholding.py

All the Sobel Thresholding has been done on the H channel after the image has been converted to HSV color space.

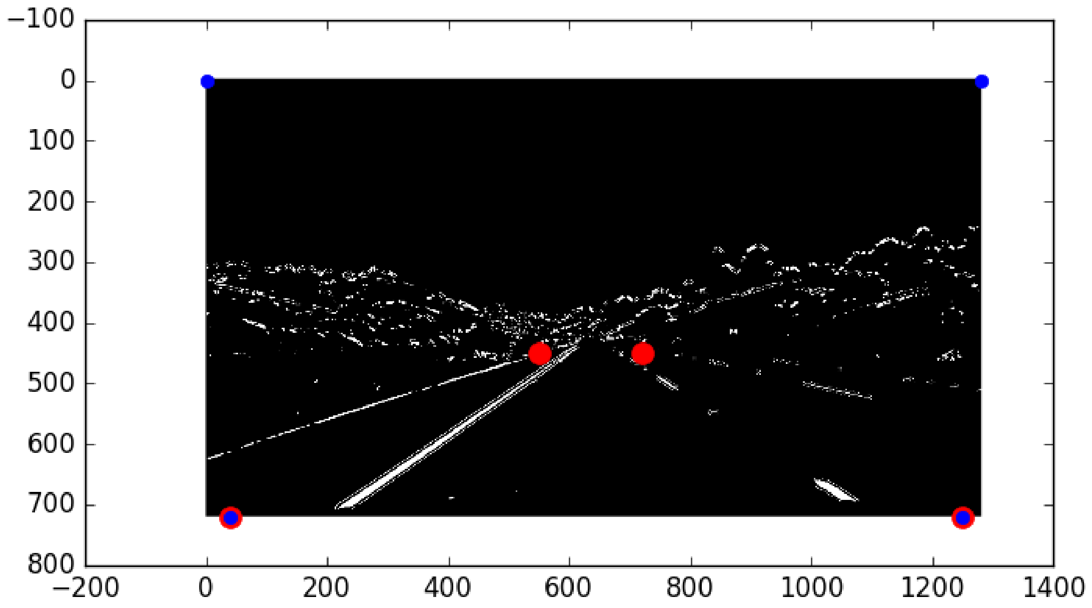
I have first computer the Sobel Threshold with respect to X and then with respect to Y. I have also computer a Magnitude of the gradient and also the direction of the gradient. All these have been thresholded correspondingly using the values in Thresholding.py – Lines 7 – 12. Finally, all of these gradients have been combined and the output is returned to generate an output like this:



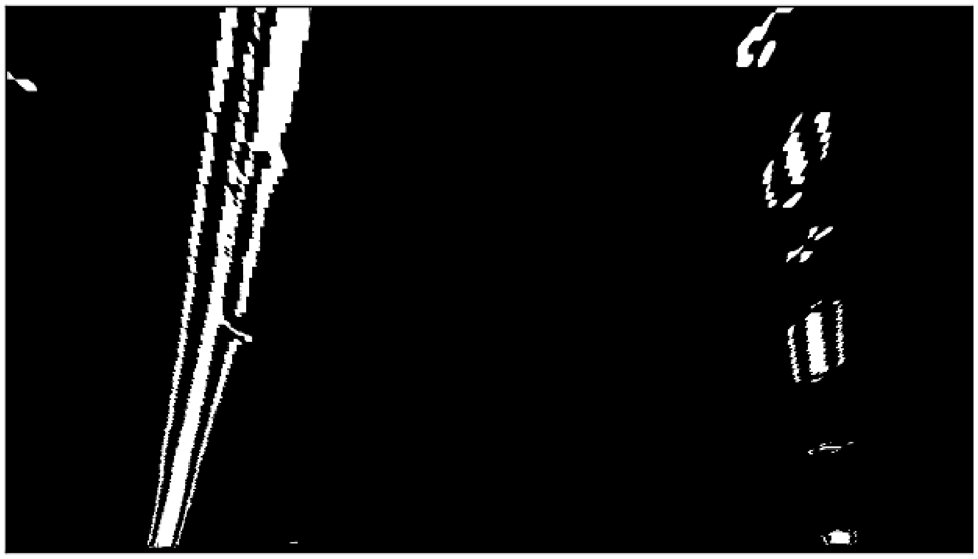
- **Combined Sobel and Color Thresholding:**
  The results for both the color and Sobel thresholds are then combined.
  The code for this is in Thresholding.py
  Once the thresholds are combined an output like this is generated:



- **Perspective Transform:**
  - The code for my perspective transform includes a function called apply_perspective_transform(), which appears in lines 48 through 79 in the file ImageTransformer.py. The apply_perspective_transform () function takes as inputs an image. I have then included a set of source and destinations points inside the function.
  - The persepective tranform is applied using cv2.getPerspectiveTransform() function which takes the source and destination points.
  - The source and destination points are shown plotted on the image below – The red points are the source points and the blue points are the destination points:



  - The result of the perspective transformation are shown in the image below:



- 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.

Finding Lane-Line Pixels & Fitting a Polynomial:

- This is the process I have followed for finding the lane line pixels:
- I followed two steps for finding the lane lines – Blind Search and Confident Search
- I use blind search initially and once I know where the lines are I shift to Confident search also when I lose confidence in a search I switch back to blind search.
- The code for blind search is in BlindLineFinder.py and the code for Confident Search is in ConfidentLineFinder.py
- I have created a class LineMaintainer which is present in LineMaintainer.py to maintain some line information. I have created two LineMaintainer objects, one for Left Line and one for Right Line.

<u>Blind Search:</u>

- First, I take a histogram along all the columns in the lower half of the image.
- Then I use a sliding window approach to blindly search the positions of the lines.
- With this histogram I added up the pixel values along each column in the image. In my thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. I used that as a starting point for where to search for the lines. From that point, I used a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.
- The code for this is in BlindLineFinder.py Lines 21 – 54 for the left line and lines 56 – 93 for the right line.
- I finally fit a polynomial to the lines using Numpy Polyfit function.

<u>Confident Search & Averaging Fits:</u>

- In the cofident search, the obtain the previous fit and search with a small margin around the last known locations of the lines.
- I also take an average of the last 8 measurements which are maintained in a queue in order to smooth out the measurements. The code for this is present in LineMaintainer.py, Line 11 and ConfidentLineFinder.py, Lines 19-23. I finally then fit a polynomial using Numpy Polyfit function.

<u>Insanity Check</u>

- I also check for the radius of the curvature of the lines and if they are out of the ranges I have specified I switch back to blind search until I get a confident result again. The code for this is present in LineMaintainer.py, Lines 59 - 65.

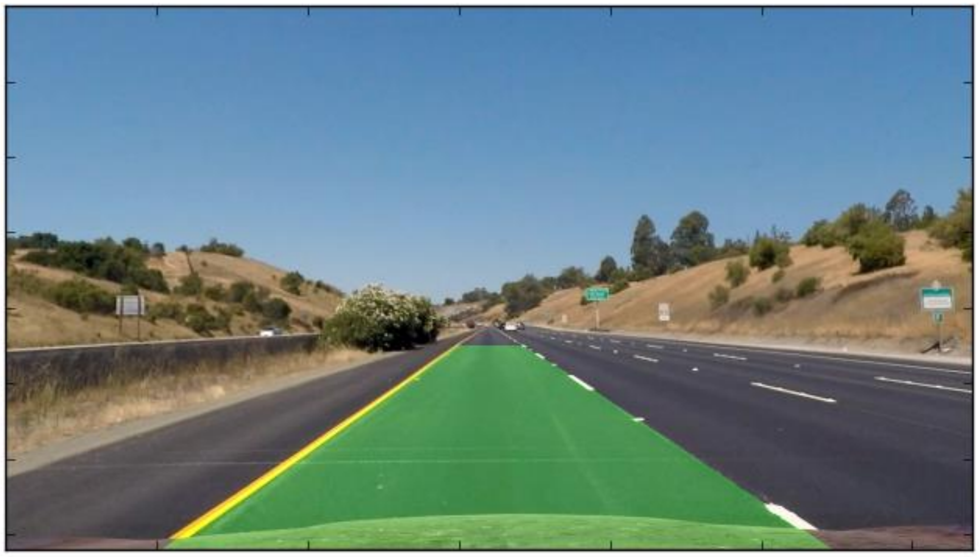
- I also check for the distance between two consecutive measurements and make sure they are not too absurd.
- The measurments which fail the sanity check are skipped and not appended to the queue

Radius & Center Calculation:

- The calculation for radius in BlineLineFinder.py, Line 92-101 and ConfidentLineFinder.py, Lines 78 – 86.
- For calculation of radius, the x and y values from converted to real world spaces using the conversion ym_per_pix = 30/720, xm_per_pix = 3.7/700. I then fit a polynomial after converting the x and y values to real world space.
- I then calculate the radius of curvature using the formula given in the lessons
- The calculation for the position of the vehicle with respect to the center is in LaneDetect.py, Lines 33-34
- I take the average of the left and right polynomials and get the midpoint and then I subtract it with the center of the image and the convert it to the real world space.

Example of Result on Test Image:

I implemented this step in lines 27 - 56 in my code in LaneDetect.py. Here is an example of my result on a test image



Pipeline Video:

Here is a link for my pipeline result on the project video - https://drive.google.com/open?id=0Bwh773at_cohYjJqUE5QbFZza28

Discussion

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

I approach I took was similar to how it was taught in the lessons. I first expirent a lot with various color spaces and graident thresholds and finally picked the ones which felt good enough.

I initially got sum bumpy results but then I implemented averaging out the last n fits in order to smooth out the results.

I think the pipeline might fail in other roads and videos and it has been strictly set to the project video. It was also difficult to make it work for the challenge videos.

I might improve the selection of source and destination points for Perspective Transform dynamically instead of settings them manually. I would also work improve the check for make sure lines are proper.