Advanced Lane Finding Project

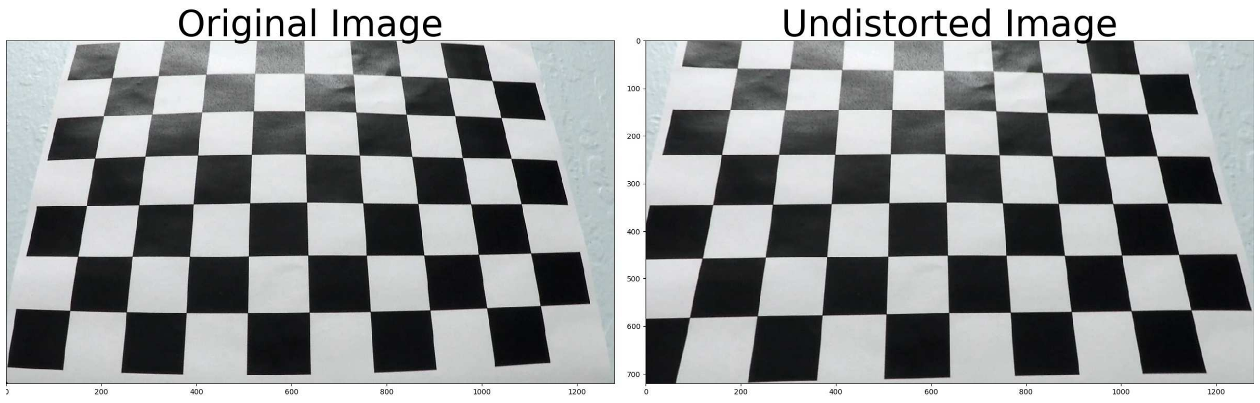
The goals / steps of this project are the following:

- 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.

Camera Calibration

The code for this step is contained in the first function "ImageCalibration". 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 saved this distortion correction to a file and applied this distortion correction to the test image using the `cv2.undistort()` function by reading distortion correction from saved file.

Applying distortion correction to chessboard images obtained this result:



All the resulted distorted & undistorted chessboard images saved in to “output_images/calibration*.jpg”

Pipeline (single images)

Pipeline performs following steps to Find lines on an image, this part of the code can be found in the function: ImagePipeline()

1. Apply Gaussian smoothing
2. Gradient and color threshold
3. Distortion correction and perspective transform
4. Extract region of interesting area in an image
5. Find lane Lines and connect points
6. Calculate radius of the curvature
7. Calculate position on the vehicle from the center of the lane
8. Plot lane finding result and original image

I will describe how I applied above steps to one of the test images like this one:

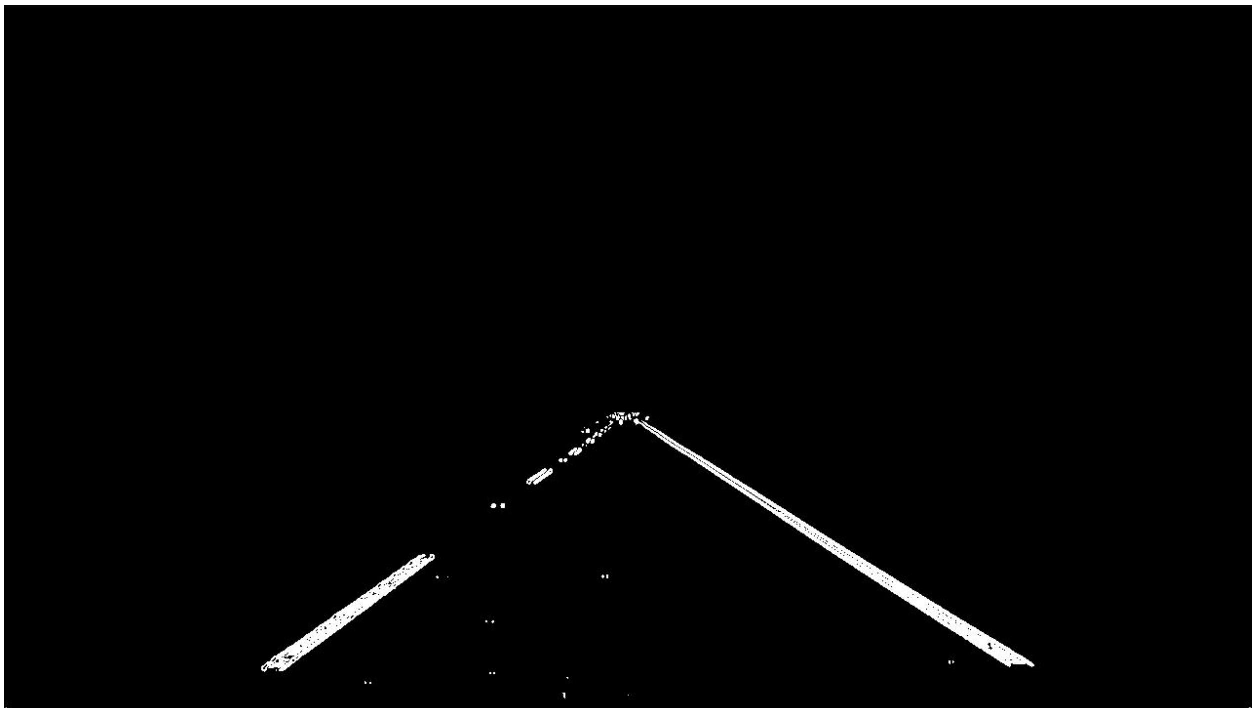
1. **Gaussian Filter**

Most useful filter (although not the fastest) for image smoothing or reducing noise. Gaussian filtering is done by convolving each point in the input array with a Gaussian kernel and then summing them all to produce the output array

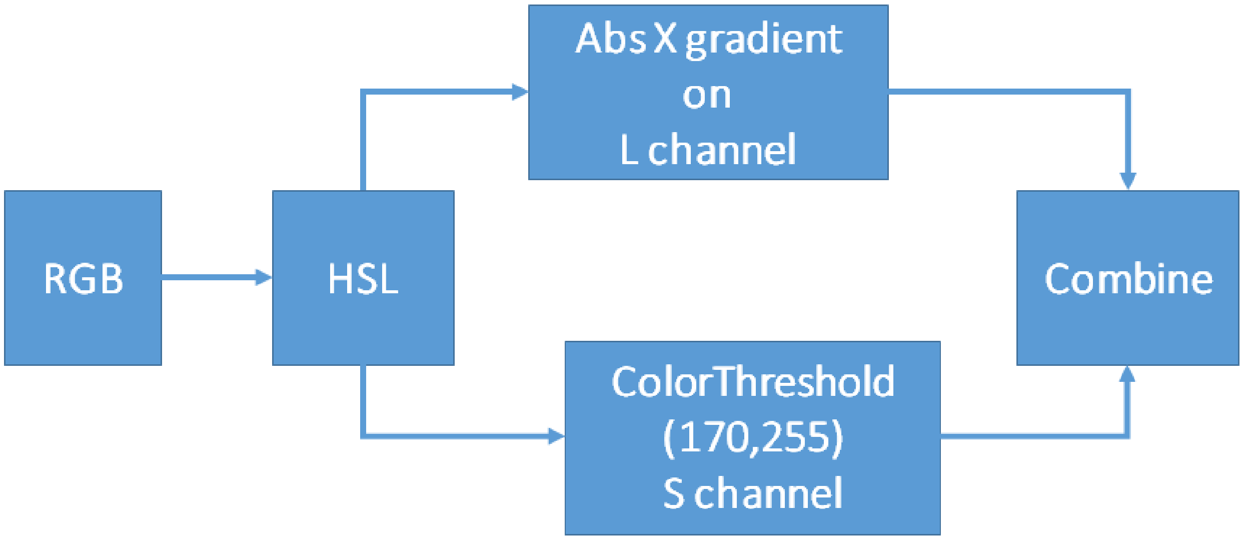
Used OpenCV API Gaussian_blur() with kernel size 5

2. **Threshold binary image(color and gradient)**

I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at function AppyColorAndGradient()). Here's an example of my output for this step. (note: this is not actually from one of the test images)



Below diagram describes the steps taken to achieve binary threshold image


RGB
HSL
Abs X gradient on L channel
ColorThreshold (170,255) S channel
Combine

3. Distortion correction and perspective transform

Firstly applied distortion correction to the image by retrieving saved distorted correction mtx and dist. Used open cv function cv2.undistort() function.

The code for perspective transform includes a function called corners_unwarp (), which appears in lines 145 through 163. The corners_unwarp `()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) transform points. I chose the hardcode the source and destination points in the following manner:

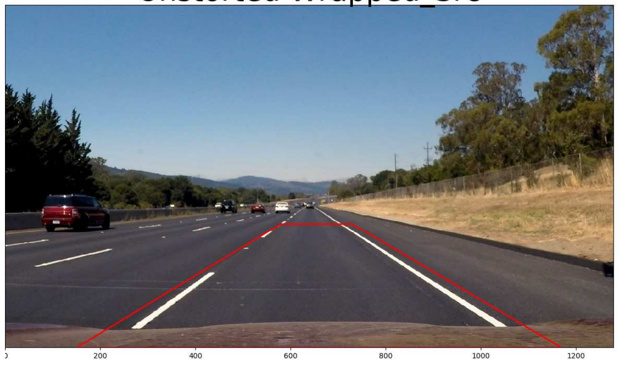
```
#points on source image
src = np.float32([[580,460],[730,460],[1170,img_size[1]],[150,img_size[1]]])

offset=200
#points on destination image
dst = np.float32([[offset, 0],[img_size[0]-offset, 0],[img_size[0]-offset,img_size[1]],[offset,
img_size[1]]])
```

This resulted in the following source and destination points:

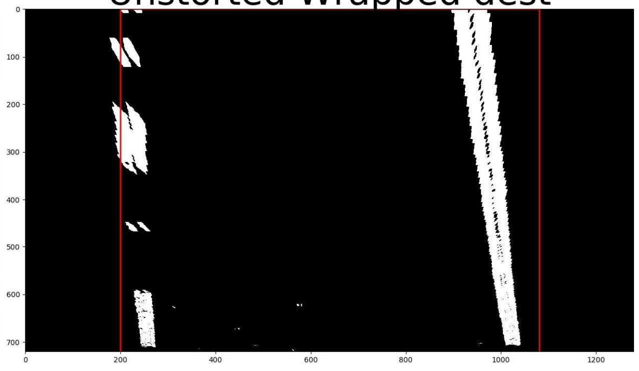
Source	Design
580,460	200,0
730,460	1080,0
1170,720	1080,720
150,720	200,720

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.

Unstorted Wrapped_Src



Unstorted Wrapped dest

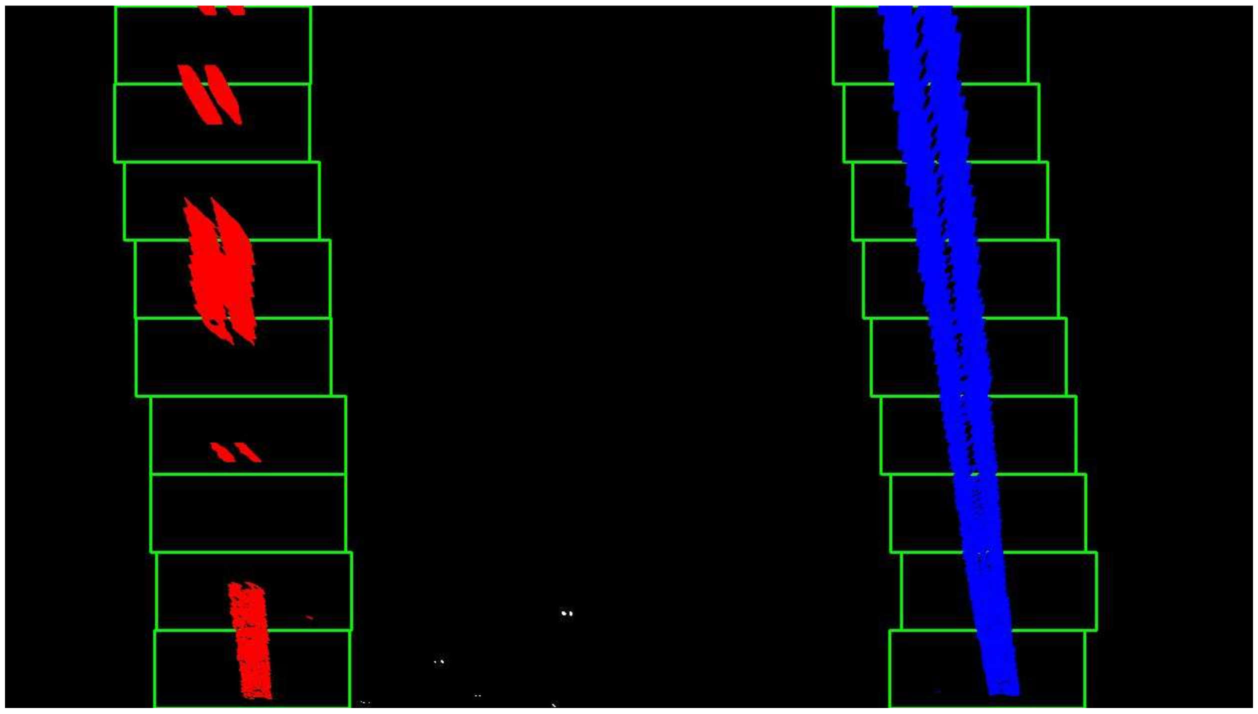


4. Identified lane-line pixels and fit their positions with a polynomial

Firstly taken the histogram of the bottom of the image and created 9 windows for sliding and each window of size 80(Image_height (720) / 9). Then I Step through the windows one by one to find left and right lane pixel indices.

I implemented this step in lines 194 through 271 in my code in AdvanceLaneFinding.py in the function FindLanesSlidingWindow (). Here is an example of my result on a test image:

Then I fitted lane lines with a 2nd order polynomial, the result of looks like this:



To improve the performance of lane line finding used above explained sliding window mechanism for first frame of the video and for the rest of the frames used previous results to fit lane lines with a 2nd order polynomial. This part of the code implemented in function FindLanesMarginArround()

5. Radius of the curvature of the lane and the position of the vehicle with respect to center.

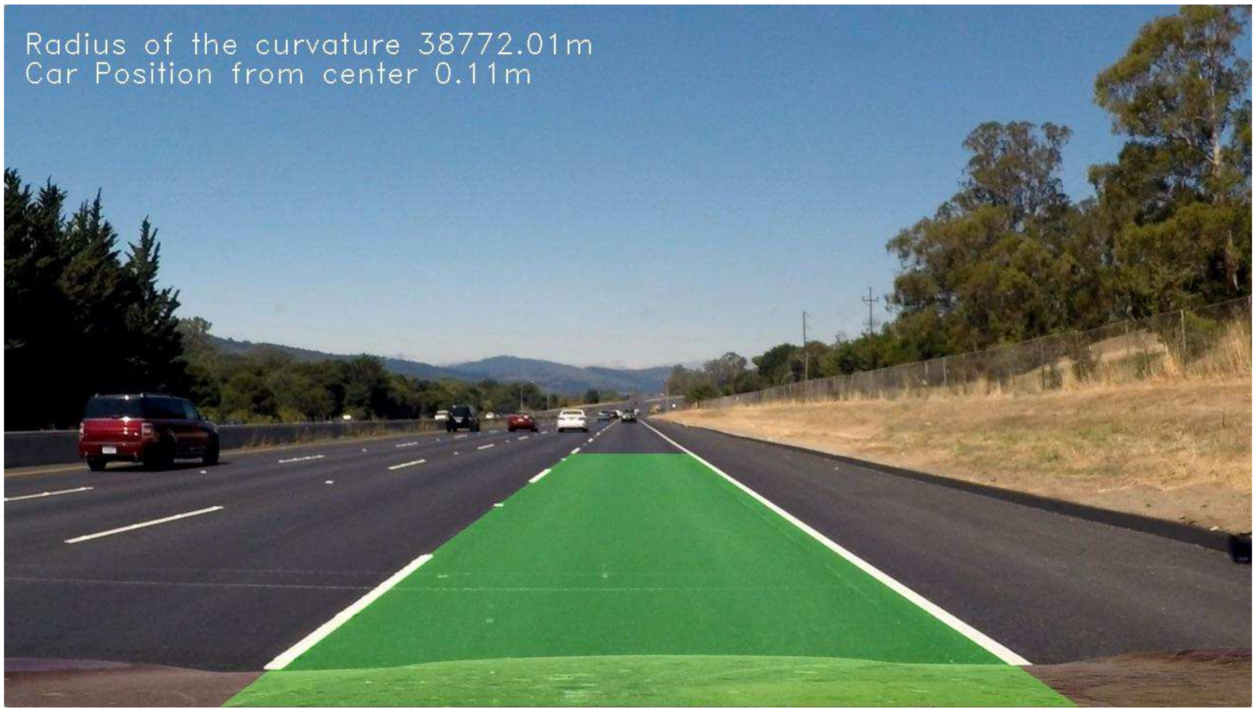
After located the lane line pixels, used their x and y pixel positions to fit a second order polynomial curve: formula: f(y) = Ay^2 + By + C, where A, B, and C are coefficients. The used radius of the curvature to find the radius of the curvature.

The positon of the vehicle is calculated by finding the center point in wrapped image and mapping pixel positon to real world space by assuming the lane is about 30 meters long and 3.7 meters wide.

I did this in lines 299 through 199 in functions FindLaneCurvature , FindVehiclePosition in my code in `AdvanceLaneFinding.py`

6. plotted back down onto the road such that the lane area is identified clearly.

I implemented this step in lines 382 through 399 in my code in `AdvanceLaneFinding.py` in the function ImagePipeline`()`. Here is an example of my result on a test image:


Radius of the curvature 38772.01m
Car Position from center 0.11m

Pipeline (video)

The output video file “white.mp4” also can be in the same hierarchy of this report and AdavceLaneFinding.py

https://github.com/anilhd2410/UdacityProjects/blob/master/CarND-Advanced-Lane-Lines/white.mp4

VideoOutput

Discussion

1. **Problems / issues you faced in your implementation of this project.**

   Finding optimal source and destination points for perspective transform, manage to find these points after few iterations and plotting these points on the test images.

2. My pipeline likely fail in

Finding Lane lines and fitting points of the lane lines, this I observed in challenging video, since I use previously found results to calculate lane line points if the first frame Lane line fit is bad then subsequent lane line findings will fail.

3. Suggestion to make it more robust?

- Sanity check
- Discard bad detection and use perfect detection
- Smoothen the result by taking average of the lane line detection points and use this result for subsequent lane line findings