Project 2: Advanced Lane Lines Detection
Single Image

Introduction

In this project, we will apply computer vision techniques to detect lane lines on the road. We will use Opencv, and python as tools. The final goal is to be able to detect complex lanes such as curve lanes. We will follow the following steps:

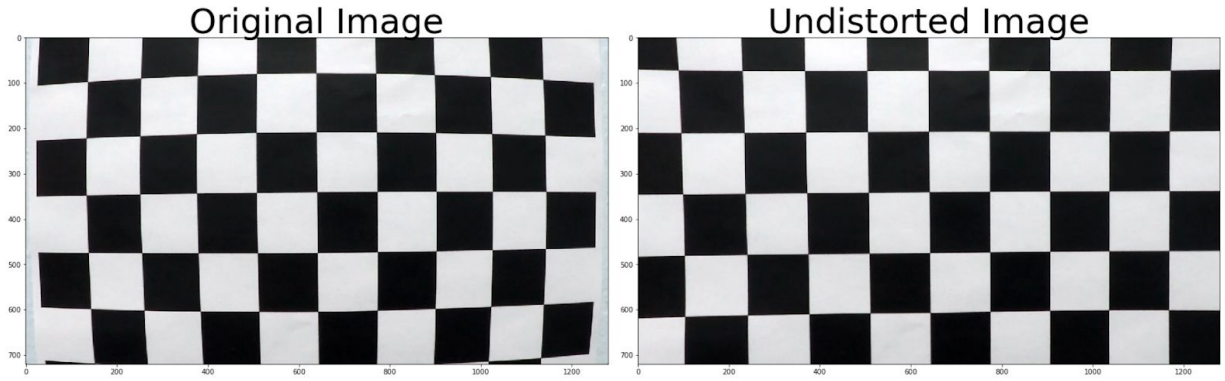
1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
2. Apply the distortion correction to raw images.
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform to rectify binary image.
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Warp the detected lane boundaries back onto the original image.
8. Output visual display of the boundaries and numerical estimation of lane curvature and position.

1. Camera Calibration Matrix and Distortion Coefficient

We will take images with a camera, and today's cameras introduce a lot of distortion to images. There are two major distortions that can appear in images: radial distortion and tangential distortion. In radial distortion, straight lines will appear curved as we move away from the center. In tangential distortion, some area in the image may look nearer than expected.
The code for the steps - 1 and 2 - is contained in the second code cell of the IPython notebook located in 'lanedetection.ipynb'. I start by calling the function 'get_img_obj_points' contained in the file 'codes/preprocessing.py'. This function takes as input a list of chessboard images, the number of corner pattern in the x-axis, and in the y-axis. It returns a list of object points and a list of image points. Object points is a list of (x, y, z) coordinates of the chessboard corners in the world, after successful chessboard detection. 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. Image points is a list of (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection. I then use the previous output to compute the camera calibration matrix and the distortion coefficients using '*calibrate_camera*' function in '*code/preprocessing.py*'.

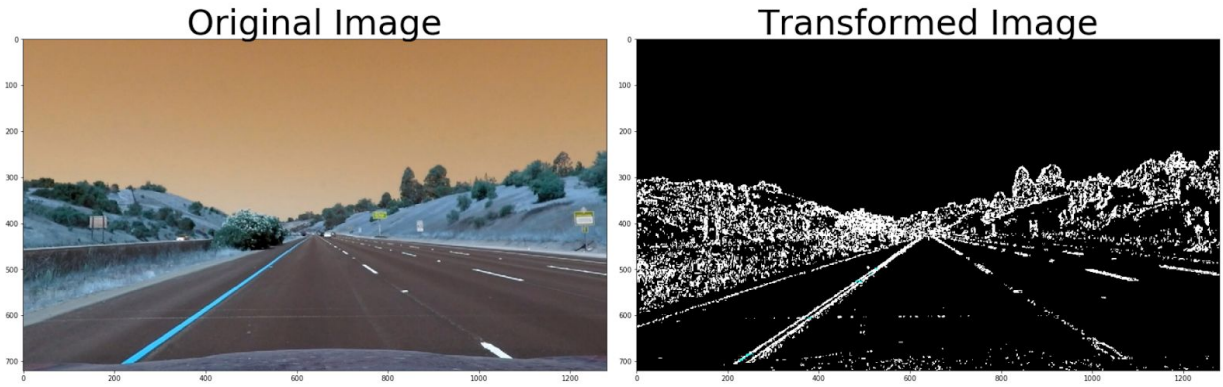
2. Distortion Correction

Having the camera calibration matrix and the distortion coefficients, I apply them to raw images using the function '*undistort_image*' in '*code/preprocessing.py'.* The final result looks like in the figure below.



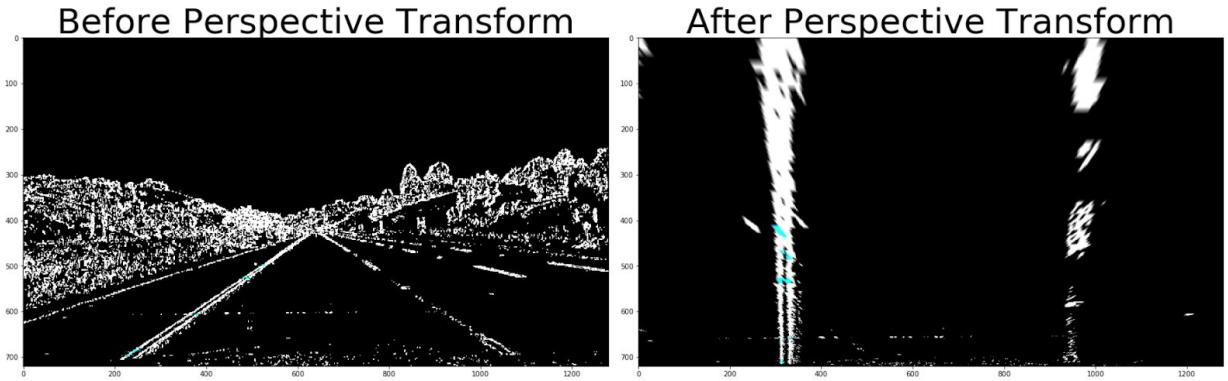
3. Color transform

The step in this section is described in the notebook cell number 5. In this section, we call the '*detect_edges*' function in '*codes/preprocessing.py*'.
Once we have an undistorted image, we need a robust way to locate lines in our image. To do so we will apply some transformations to our image. We first convert our image from RGB (Red, Green, Blue) color space to HLS(Hue, Lightness, and Saturation) color space. HLS is a more robust way to represent pixels of an image. Then, we filter the lightness and saturation channels, in which we apply our transformations. We apply the Sobel operator on the x-direction of our lightness channel to filter vertical edges, and color threshold on our saturation channel to filter pixel value on a specific range. Finally we stack the two channels togheter with an all black channel. The final result look like the figure below.



4. Perspective Transform

The steps in this section are described in the notebook cell number 8. In this section, we call 'perspective_transform' function in ‘codes/preprocessing.py’ to compute the perspective transform, and we call 'warp_image' in ‘codes/preprocessing.py’ to apply the transformation to our image.
Once we have successfully detected edges, we would like to have a bird's-eye view of our image. To achieve this, we will calculate the perspective transform and apply it to our image. The final result looks like the figure below.

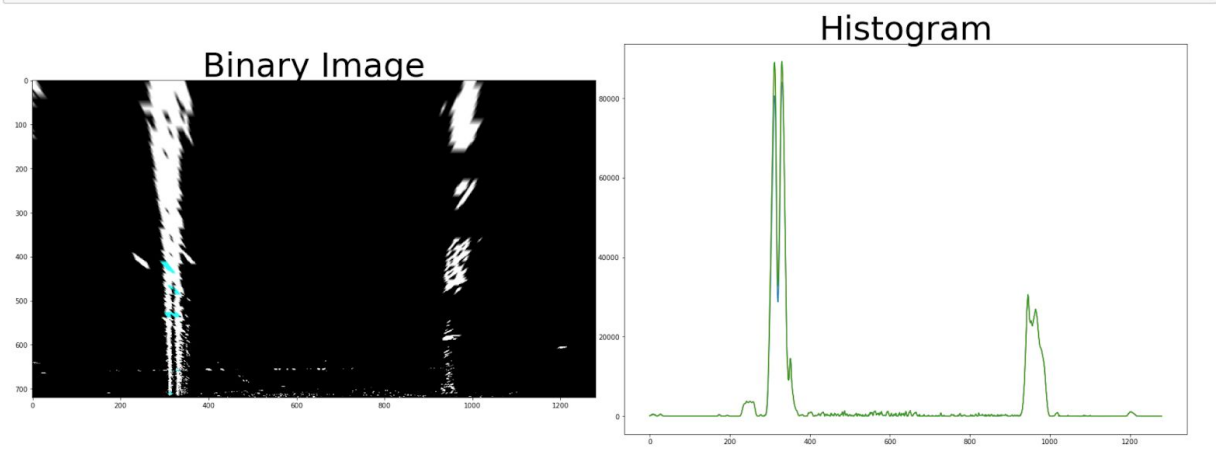


5. Detect lane pixels and fit to find the lane boundary

5-1. Detect the starting pixel lane

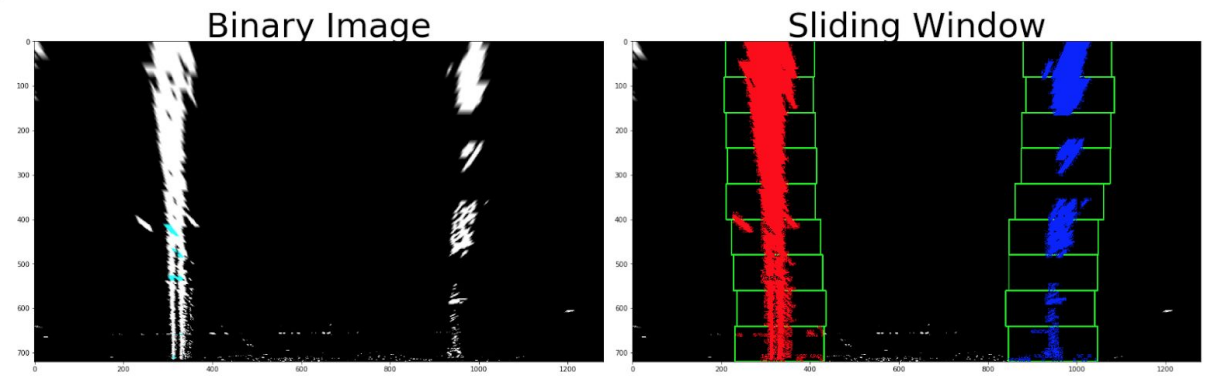
The step in this section is described in the notebook cell number 11. In this section, we call the 'hist' function in ‘codes/preprocessing.py’.

After the previous preprocessing step done earlier, we still need to decide which pixel belong to the left lane and which one belong to the right lane. To start searching for the lane, we use the histogram of pixels. Since we have a binary image - with value 0 or 1-, the histogram count the number of pixels along each column on the image. The column with the highest number of pixel values on the far left of our histogram will be the starting point (x-axis) of our left lane, the column on the far right of our histogram with the highest number of pixel values on the far right will be our right lane. The function 'hist' creates the counts that we will plot on a histogram. The end result will look like in the figure below



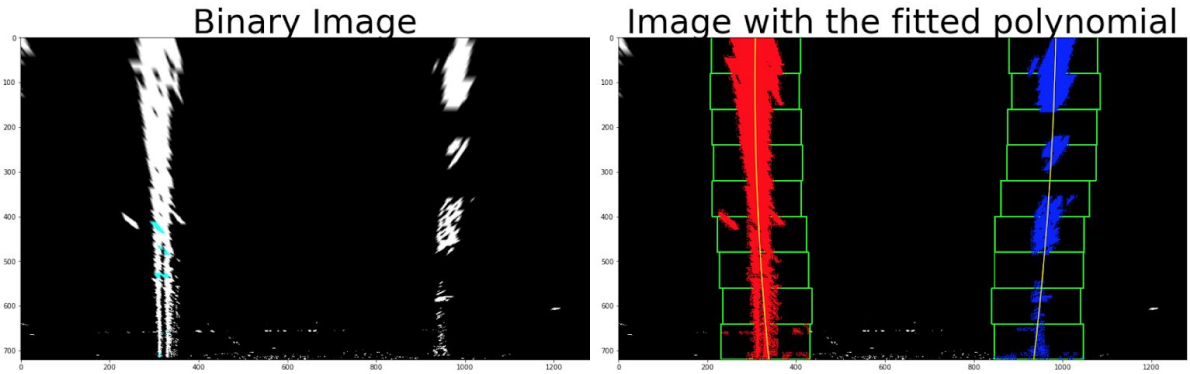
5.2 Sliding windows

The step in this section is described in the notebook cell number 12. In this section, we call the '*find_pixel_lane*' function, and the ' *fit_polynomial'* function in '*codes/preprocessing.py*'.
Once we have the starting point of each lane, we use the sliding windows algorithm to draw rectangles around the central pixel from that point until the end of the frame.
we start by defining the hyperparameters such as the number of rectangles we want to draw, the margin(the number of pixels on the left and on the right of the central pixel), the minimum number of non-zero pixel in each rectangle (minpix). We update the central pixel x-coordinate if the number of non-zero pixel in the rectangle is greater than minpix. The end result will look like in the figure below.



5.3 Fit the pixel lane

The step in this section is described in the notebook cell number 16. In this section, we call the '*plot_polynomial*' function in '*codes/preprocessing.py*'.
Once we have located the line pixel, we fit it. The end result will look like in the figure below.

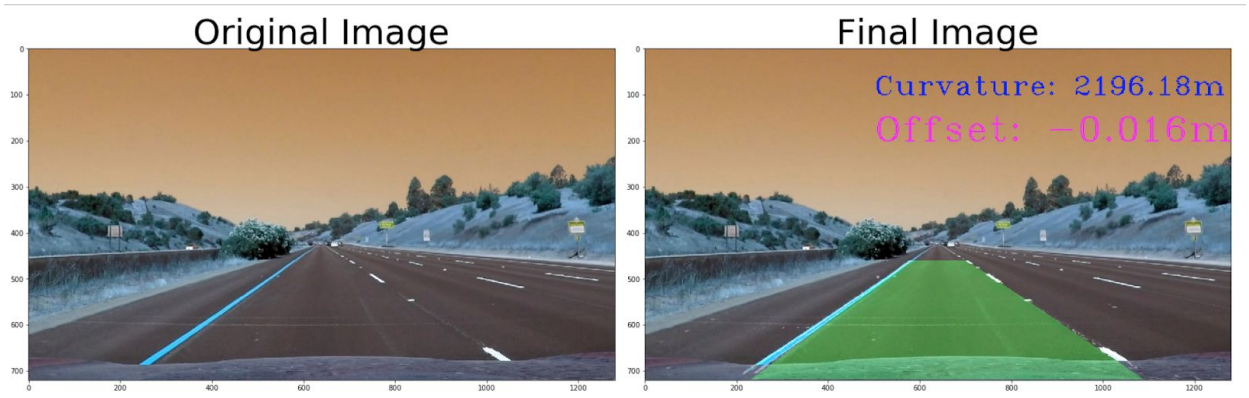


6. Determine the curvature of the lane and vehicle position with respect to the center

The step in this section is described in the notebook cell number 15. In this section, we call the '*calculate_curvature_offset*' function in '*codes/preprocessing.py*'.
Once we have detected the lanes, we can calculate the curvature of each lane. We will use the mean value ('mean_curvature')of lanes' curvature to represent our vehicle curvature. The 'mean_curvature' is the average between the left lane curvature and the right lane curvature.
To calculate the position of the vehicle with respect to the center(offset), we use the center lane minus the center of the image. Negative offset means our vehicle center is

on the left-hand side of the center of the image. And a positive offset means our vehicle center is on the right-hand side of our center lane. The final values are converted from pixel to meter.

7-8. Warp the detected lane boundaries back onto the original image and display
the steps in this section are described in the notebook cell number 16 located in './lane_detection'.
The final step in our lane detection is to warp the detected lane boundaries back onto the original image and display. The end result will look like in the figure below.



9- Pipeline Video

Here is the link to a final video

10- Discussion

Even though my code performs well in a sligly curved environment, it doesn't perform well on more complex curved lanes. Maybe I should consider use a more complex function for my fit, such as a third degree polynomial equation.