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.**

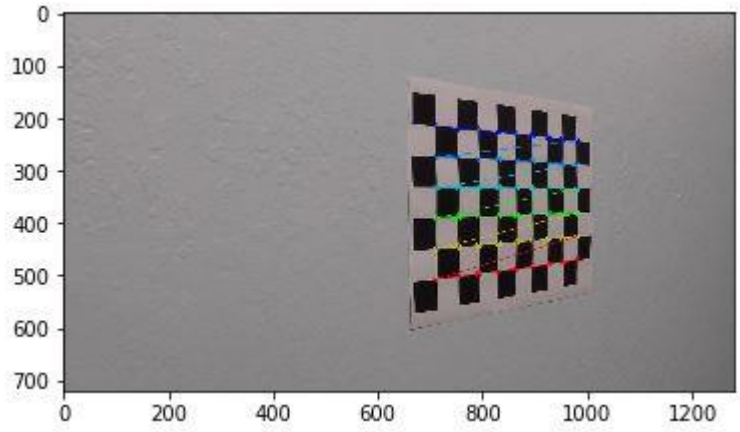
Now I will discuss each of the steps of the project in detail

Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.

The code for this step is contained in the first code cell of the IPython notebook called Advanced Lane Finding.

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.

By using the above strategy, I was able to successfully detect the chessboard corners perfectly, the output of which I am providing below:

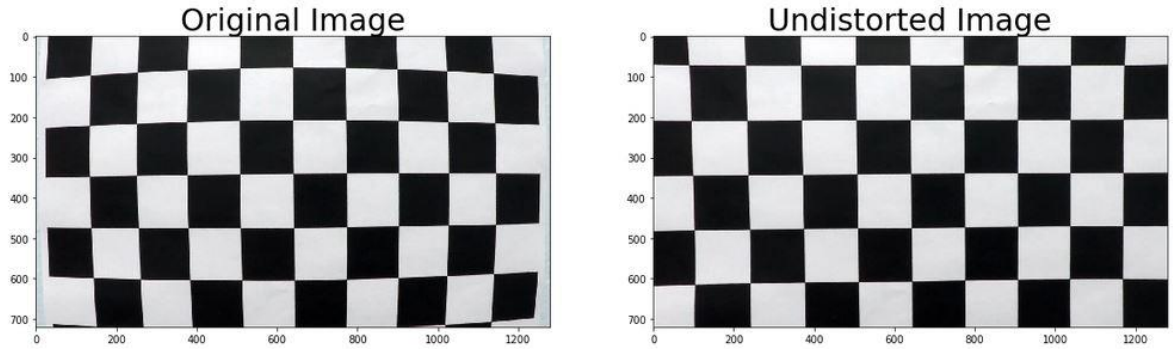


In this way I was able to determine the corners of all the images provided in the camera_cal folder.

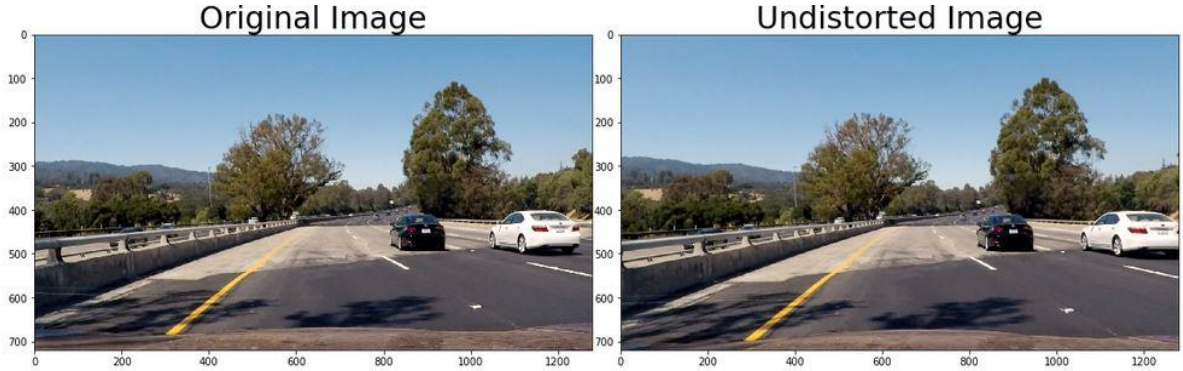
Apply a distortion correction to raw images.

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 a distorted chessboard image for verification using the `cv2.undistort()` function and obtained this result:



The, I applied my cv2.undistort() function to a test image and got the following result :



The differences in the above two images might not be visible immediately but if carefully observe the white car on the extreme right, we will be able to identify that the original image has indeed been undistorted by the cv2.undistort() function.

Apply a perspective transform to our test image ("birds-eye view").

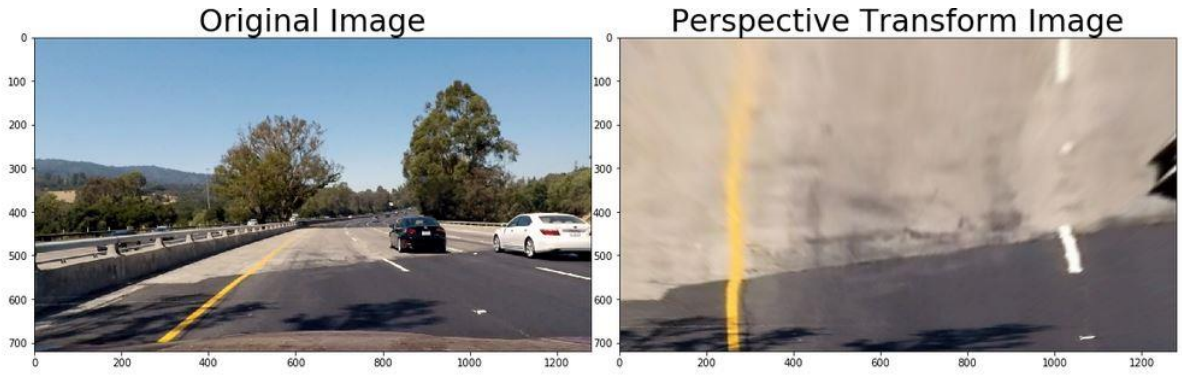
The code for my perspective transform includes a function called `perspective_transformr()`in the 5th code cell of the IPython notebook Advanced lane Finding. The function takes as inputs an image. I chose the hardcode the source and destination points in the following manner:

```
src = np.float32(
    [
        [580.0, 460.0],
        [740.0, 460.0],      # Defining the source coordinates
        [1100.0, 670.0],
        [270.0, 670.0],
    ]
)
dst = np.float32(
    [
        [200.0, 0],
        [width - 200.0, 0], # Defining the destination coordinates for the perspective image
        [width - 200.0, height],
        [200.0, height],
    ]
)
```

This resulted in the following source and destination points:

Source	Destination
585, 460	200, 0
740, 460	1080, 0
1100, 670	1080, 720
270, 670	200, 720

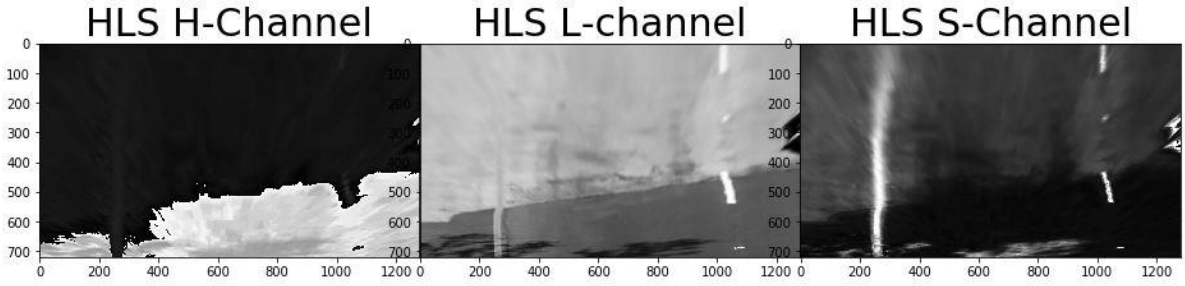
After applying the perspective transform to my test image, I got the result as follows:



As we can clearly observe the bird's eye view of our lane lines which will be immensely beneficial for us to detect our lane lines accurately and keep track of its curves and movements.

Use colour transforms, gradients, etc., to create a thresholded binary image.

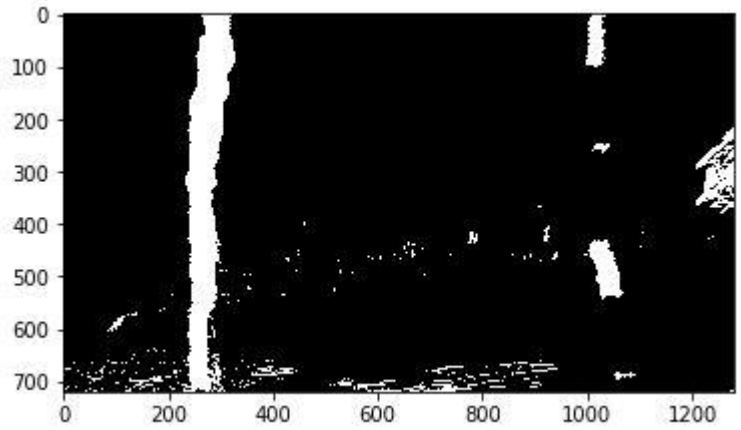
For the colour transformation, I have used the HLS (Hue, Lightness, Saturation) because I have clearly observed that two of the colour channels namely L & S gives much of the clear visual of the lane lines, the output of which I have shown below:



Now, since the L and S gives us the lane lines very clearly, so I decided to include both of them for calculating the gradient threshold, for which I used the Sobel operator which lies at the very heart of Canny edge detection as well. Sobel operator takes the gradient of the image in x and y direction both, so we have two different Sobel matrices to calculate our gradient separately in x and y directions.

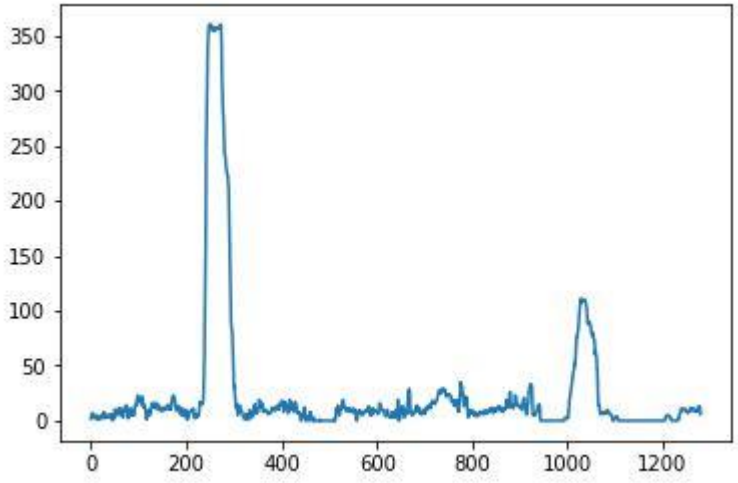
In the IPython notebook, the code cell 7th is having the function which is combining the effects of the L and S channels, code cells 8th , I have calculated the magnitude of gradient because both x and y give the lane boundaries very efficiently, in code cell 9th, I have calculated the orientation/direction because we are interested only in the edges of a particular orientation of the lanes. Finally, in code cell 10th, I have combined everything into a single function called combined_thresh().

By using the above combined_thresh() function, I was able to get the binary image as follows:



Detect lane pixels and fit to find the lane boundary.

First of all to visualize the region of the lane lines more prominently, I calculated the image histogram of the warped binary image obtained in the previous step, which is given as follows:



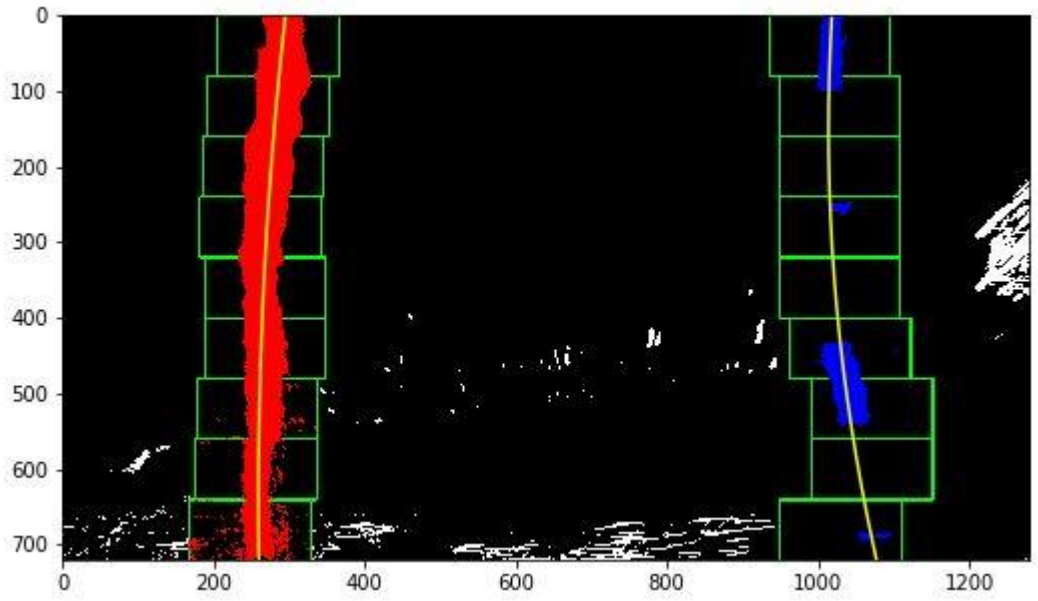
As expected, we can see two prominent peaks, one nearby 200 and other one nearby 1000, which actually is suggesting that we have two lane lines in our binary image.

After this I defined a function called sliding_window_poly2fit() in 13th code cell of IPython notebook which is essentially calculating the left lane and right lane pixels for our image by using the histogram we obtained above.

Now in this function sliding_window_poly2fit, First of all I calculate the mid-point of the histogram and divide it into two parts to properly separate the left lane from the right lane. Then after this I define some parameters of small rectangular windows such as its number, width (margin) and height. I also initialize the the x and y positions of the non zero (lane) pixels, empty lists each for left and right lanes to append those pixels which belongs to the lane line pixels and a threshold pixel value of 50 to recenter the rectangular sliding window. I then use a for loop over the number of windows to iterate through each window, collect its boundary coordinates x,y for both left and right lanes, then drawing the rectangles and collecting those non zero pixels and appending them to the lists which are definitely lane pixels according to

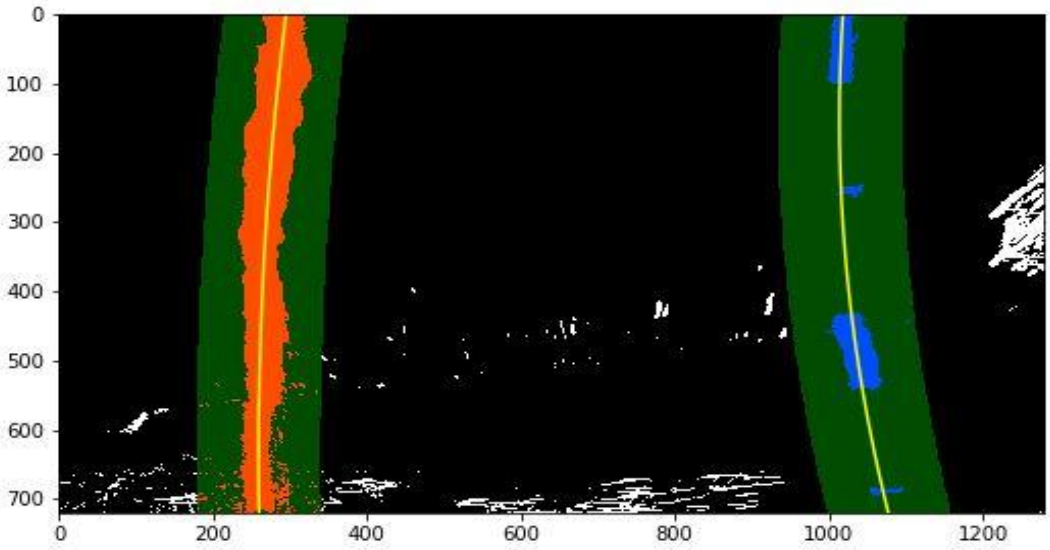
my threshold values provided. After this I extract the coordinates of the left and right lane pixel positions and fit a 2nd order polynomial through them.

I then visualized the result as follows :



Now in the 16th and 17th code cells, I have basically tried to extract the region of interest by drawing a slightly thick boundary lines over the lane lines detected by the sliding_window_poly2fit() function. This is done because in each video frame it is not good to run the algorithm as it will decrease our efficiency. So I search for the lane lines around a specified margin of 80 which gives. To do this I simply defined a function called shaded_boundary_line() which is taking the same margin of 80 as before, the binary warped image, left fit and right fit lines to determine the boundary lines on the basis of the previous frame result. In other words, we take help of the previous frame of the lane line video to detect and draw boundary lines in our current video frame which becomes much more efficient as the lines do not change drastically from frame to frame.

By doing all the work, I get the output as follows:



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

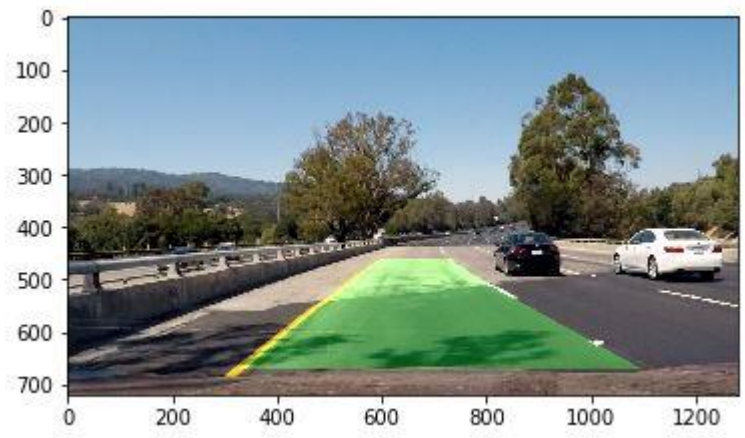
In the 19th code cell of my IPython notebook, I have tried to just convert the pixel world coordinates into the real world coordinates, means I have tried to determine meters per pixel in each x and y coordinates with the help of which I have been able to determine the curvature of our lane.

In the 20th code cell of my IPython notebook, I have calculated the vehicle position with respect to the lane center position, where I got the following values for the image which has been used above:

```
1433.1269663946957 m 838.8893329588586 m – Left curvature & Right curvature
-0.16487512022115608 m – Vehicle position w.r.t. Lane centre
```

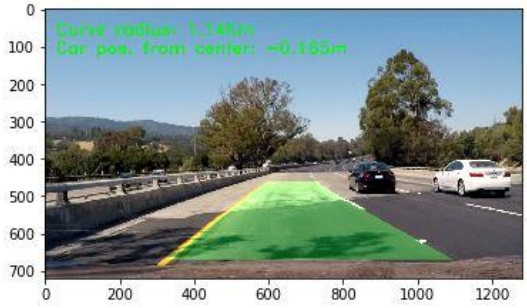
Warp the detected lane boundaries back onto the original image.

In the 22nd code cell of the IPython notebook, I have defined a function called warp_lanes_onto_original() in which I have taken the original image, binary image left fit, right fit and Minv values to warp the detected lane lines back to our original image, the output of which can be seen as below:



Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

In the 24th code cell, I have shown the original image with the curvature and vehicle position data shown by using a function, the output of which is given as follows:



Pipeline (video)

I have included the video along with this writeup.

Discussion

Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

I believe that my pipeline might be failing where the lighting conditions are not so good or the lighting conditions are fluctuating very frequently. To make the pipeline more robust we can use the other colour spaces such as HLS and LAB which does not get affected by the lighting conditions overall. We can combine the Colour channels where we find the that lighting conditions are not affecting our image frames and can work on that by combining them as I did for L & S channels here.

Another situation where my pipeline might fail is when there are sharp turns or sharp curves in the lanes or intersections. To improve that we can either change or play around with our threshold values and margin values to evaluate it under certain strict conditions.

Finally, to make the project more robust, it's better not to hard code the lane finding algorithms because hard coding makes it useful only for a handful of situations but we should rather let the machine decide what to do. In other words, to make it best, we should use deep learning and machine learning based approaches for lane lines detection which are more robust to situation changes and can adapt well by training them properly.