Vatche Donikian

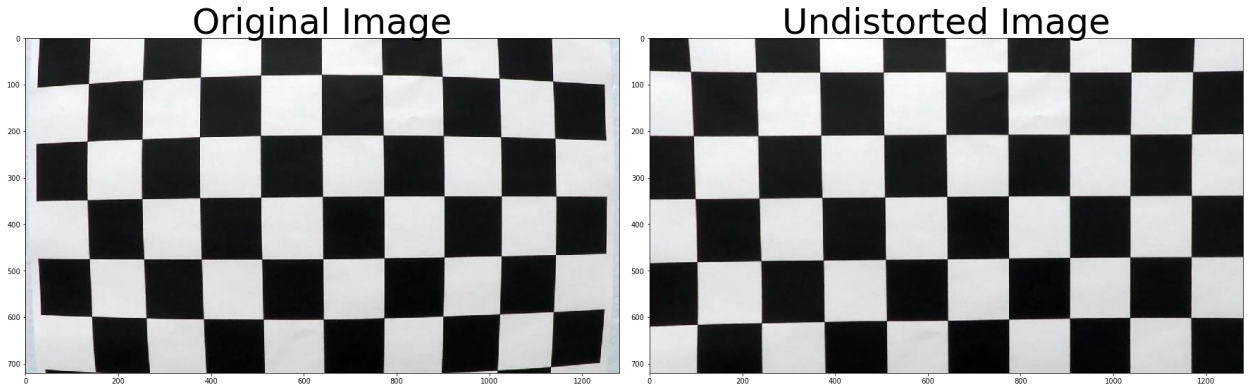
Project 4: Advanced Lane Lines

Camera Calibration

The first step for camera calibration was to prepare the object points. I used the chess board images for the calibration, thus I needed a 9x6 grid of cross points. Then, the next thing to do is to import all the images. This was done using the glob command, which helped in importing all the calibration images at once as so:

```
images = glob.glob('camera_cal/calibration*.jpg')
```

Next, for each image, I converted to grayscale, then found the chessboard corners and saved them into an array. These corners were used to find the camera matrix and distortion coefficients of the camera used in this project. This was done with the OpenCV command ‘calibrateCamera’ From here, I was able to undistort the chessboard images; an example is shown below:



Pipeline (single images)

1. Distortion Correction

With the camera matrix and distortion coefficients calculated, you can undistort any image taken from this camera. I did this using the OpenCV function undistort as shown:

```
udt = cv2.undistort(image, mtx, dist, None, mtx)
```

Using one of the test images, below is the original and corrected image:

2. Threshold Binary Image

I used both color transforms and gradients to create the threshold binary image. The first thing I did was convert the image from RGB to the HLS color space (. HLS stands for Hue, Lightness, and Saturation. Color Thresholding with this color space does a better job with shadows and changes in light or lane color. After this, I applied the Sobel operator, which is taking the gradient, in the x direction on the L channel. From this I created the binary image of the x gradient. Then, I used the color channels to create the color transform binary. I did this with a combination of H, L, and S channels. At first I was using just H and S, but then I was getting unwanted white areas in my results due to shadows. The L channel changes with lighting, so I used it to remove data wherever the shadow was picked up along with the S channel. This helped improve my results whenever there were tree shadows on the road. I then combined both the gradient and color transform binary images into one. This was all done in the 4th cell of the Jupyter notebook, which is well commented. The result is below:

3. Perspective Transform

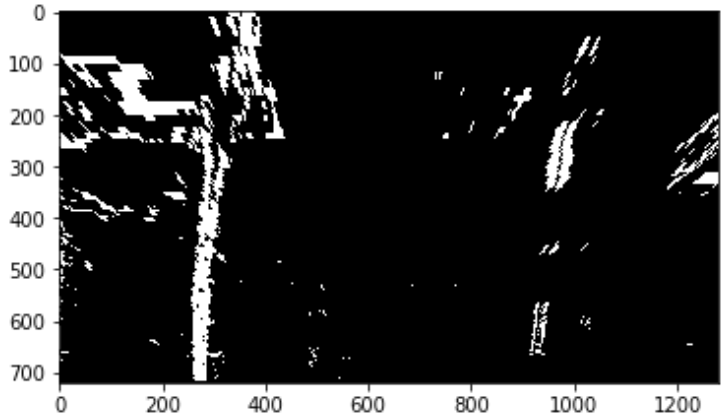
The next step was to perform a perspective transform to create a birds eye view image in order to calculate the lane lines. Using one of the test images of the straight lines, I chose my four source points which corresponded to the corners of the rectangle making the lane:

src = np.float32([[583,460],[700,460],[1105,720],[208,720]])

I then chose my four destination points, where I wanted these points to end up. Naturally I picked the top and bottom of the image:

dest = np.float32([[300,0],[900,0],[900,image_size[1]],[300,image_size[1]]])

Using these sets, of points, I was able to warp the image into an overhead view using two OpenCV commands, 'getPerspectiveTransform' and 'warpPerspective'. This was done in the 5th cell of the Jupyter notebook. An example of this transform, from part 2 to birds eye view, is shown below:

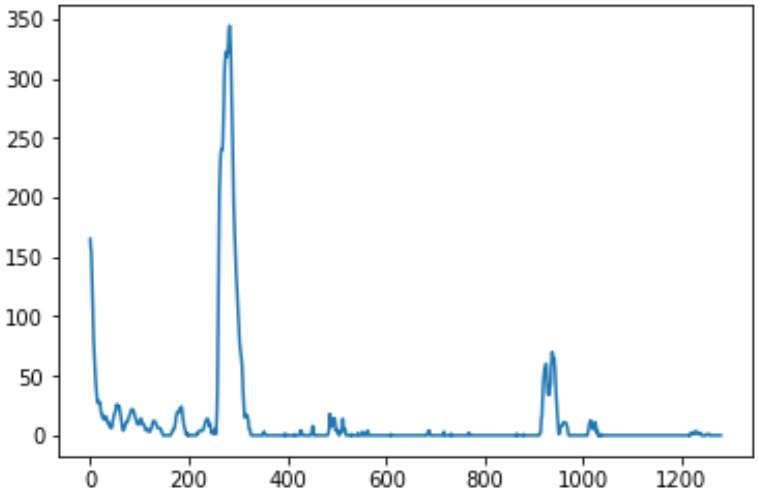


4. Lane Line Polynomial Fitting

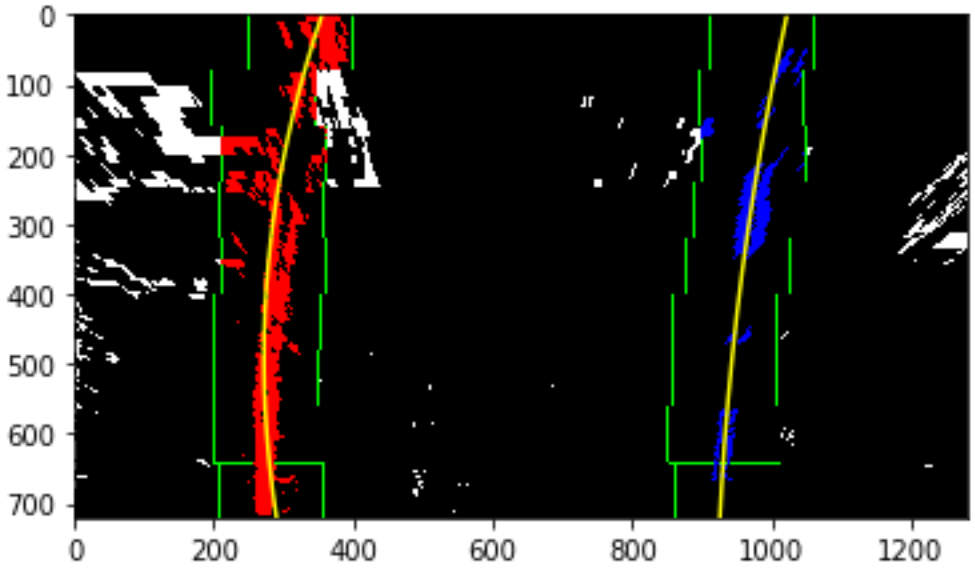
Now with the overhead image, I moved on to detecting the lines. The first step was to calculate the histogram of the bottom half of the image. This would tell us where to start the search for the white pixels. This was done with the line below:

histogram = np.sum(image[image.shape[0]/2:,:], axis=0)

For the same image used above, the histogram looks like this:



The two highest peaks on each side of the graph represent the place with the most white pixels, meaning the place where the lane lines will reside. The search is started here. The method I used to find the lines was the sliding window method. I specified the number of sliding windows, and the width of the windows. Then, using these windows, I searched through the transformed image to find the coordinates of the pixels that would correspond to the left and right lane lines. This can be seen in the 6th code block of the notebook. After this search was completed, the points found were used to fit a second order polynomial to the lane lines. The numpy library has a valuable command called 'polyfit' which performs this very well. The image below shows the detected lane pixels (red for left lane, blue for right lane), and the corresponding polynomial (yellow lines):



5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

To calculate the radius of curvature of the lanes, The equation below was used:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{\frac{3}{2}}}{|2A|}$$

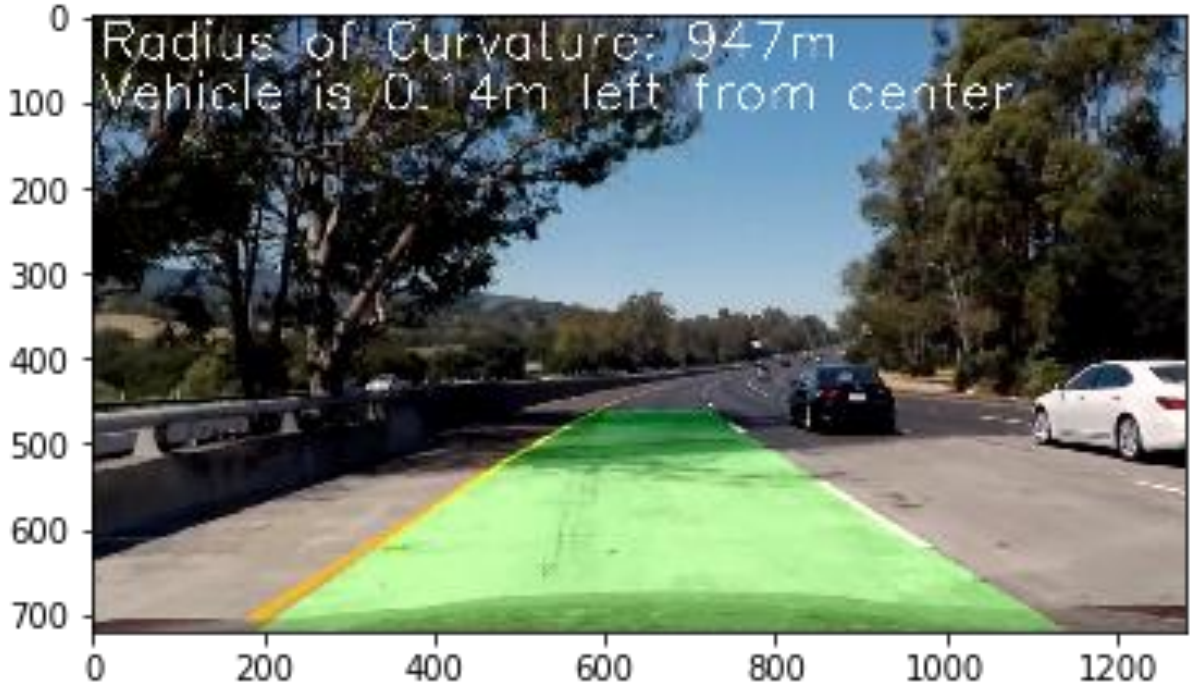
where A and B are coefficients of the polynomial curve fit. This was performed in code (7th cell) by using the results a new curve fit; this one was with the pixel values converted to meters using the conversions provided in the lesson.

To calculate the offset, I needed to know the vehicle center and the lane center. Considering that the camera is mounted on the center of the vehicle, I know that the vehicle center is the image center. The lane center was calculated by taking the midpoint between the left lane line and the right lane line at the very bottom of the image. This was performed in the 'print_vals' function in

the 8th block of code. I did this by finding the first non-zero and last non-zero pixel value for the bottom row of the filled lane image (this is described in section 6 below). With the two values calculated, a simple subtraction was performed to find the distance the car was from center of the lane.

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

The final step was to plot the calculated lane on the image and warp the image back to its original form. This was also done in the 8th cell of the Jupyter notebook, under the function 'warp_back'. OpenCV was once again used; this time it was to fill the region between the left and right lanes with green color. The image was then transformed back, and overlayed on the original image. Here I also printed the offset and radius of curvature on the image. The final result for the pipeline can be seen below:



Final Video Output

The link to video is provided here: https://youtu.be/-b4cTwGvujw

I have also included the video in the zip file for reference.

Discussion

I think my project runs pretty smooth. It did not take too much time to lane search, and the recognition is smooth enough that I did not need to smooth out over any steps. I was having a problem at the beginning where the image was losing sight of the lines when going over the lighter portions of the road and when shadows from the trees were interfering. However, when I removed shadow instances with the L channel of the HLS image, I was able to correct for this. The pipeline my fail when there are sharp curves, because the sliding window may not be able to capture the whole lane line. Also, it can fail when the lines are faded off, or in snow or rain when the line visibility is not much. To make it more robust, maybe I would need to fine other markers, not just the lanes, to keep the vehicle in the road in these adverse weather conditions. A very accurate GPS would help as well.