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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

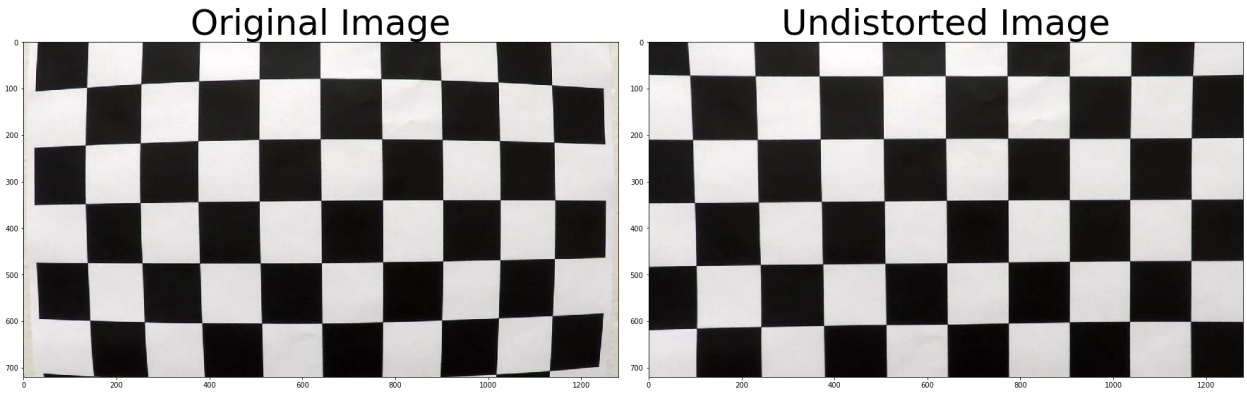
Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

First I used the OpenCV functions findChessboardCorners() and drawChessboardCorners() to automatically find and draw corners in an image of a chessboard pattern. These are located inside the “camera_calibration” function in my IPython notebook, In [2].

The I used chessboard images to obtain image points and object points, and then used the OpenCV functions cv2.calibrateCamera() and cv2.undistort() to compute the calibration and undistortion.

I defined “obejct points” and “imgpoints”. "object points" 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 and obtained this result:



Pipeline (single images)

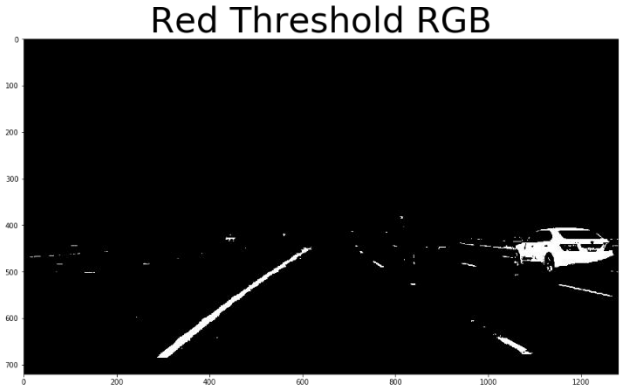
1. Provide an example of a distortion-corrected image.

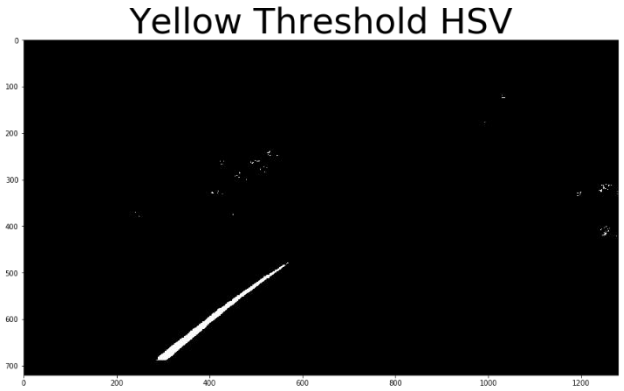
I used the OpenCV function cv2.undistort to correct the distortion of the images. The camera calibration matrix and distortion coefficients obtained from camera calibration are used for undistorting the images. This code is present in, In [4,5]. The image below shows a typical output.

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

The main purpose here is to be able to identify the yellow and white lines under varying light conditions. RGB thresholding works best on white lane pixels and does not work on images that include varying light conditions or when lanes are different colors, like yellow. Therefore, it was interesting to explore different color spaces such as HSV and HLS.
To find the lane lines, I applied various individual color thresholds and combined them all at the end. Performing this method, I was able to identify both the yellow and white lanes pretty well and I did not need to use gradient threshold. I used the following color thresholds and the results of each method has been shown here. The final result which is the combination of all, is also shown for an example image.

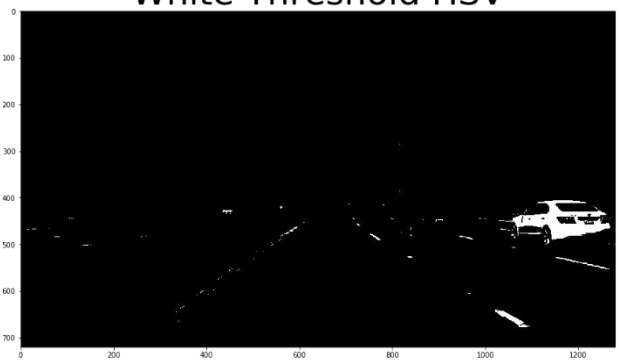
- RGB red channel threshold:
- HSV yellow threshold
- HSV white threshold
- HLS white threshold
- RGB white threshold
- White threshold combo
- Combination of all





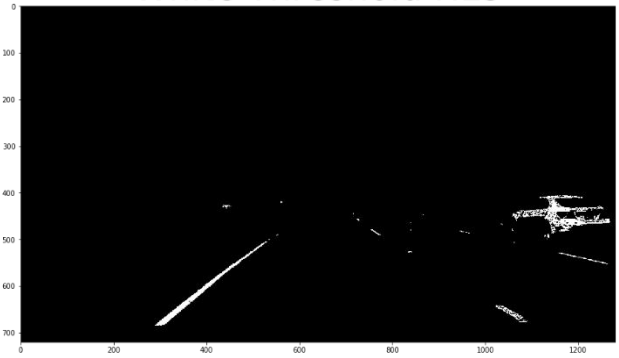
Original Image

White Threshold HSV



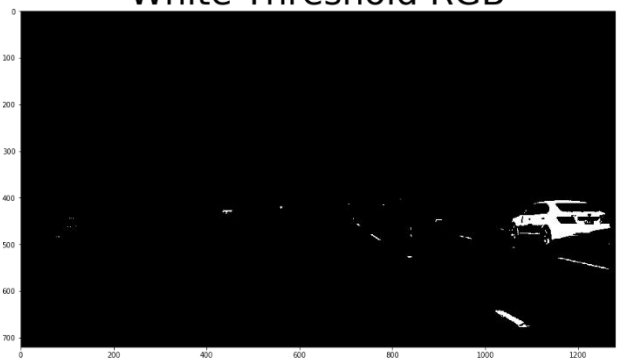
Original Image

White Threshold HLS



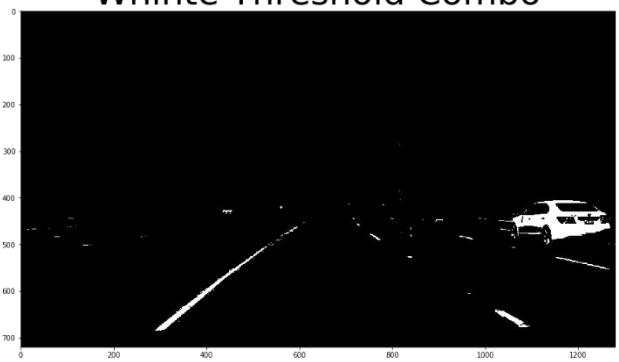
Original Image

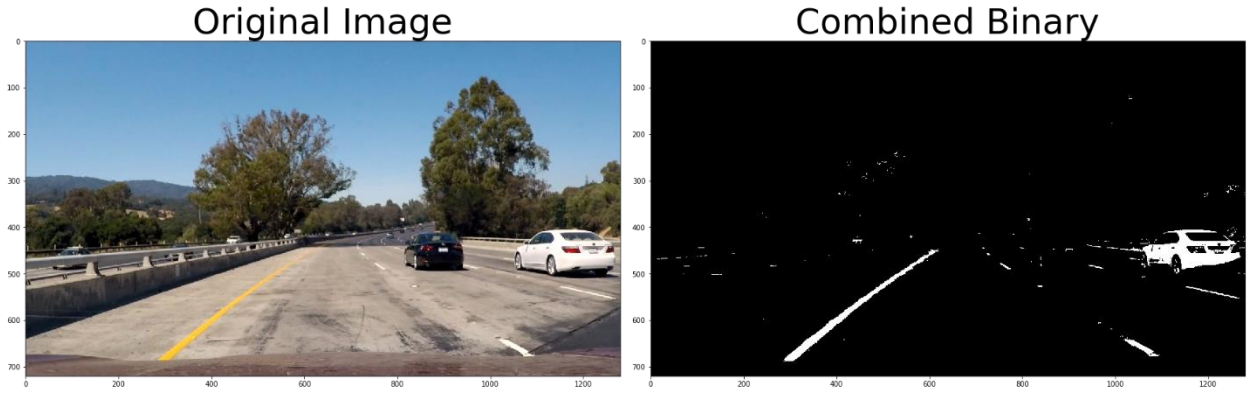
White Threshold RGB



Original Image

Whihte Threshold Combo





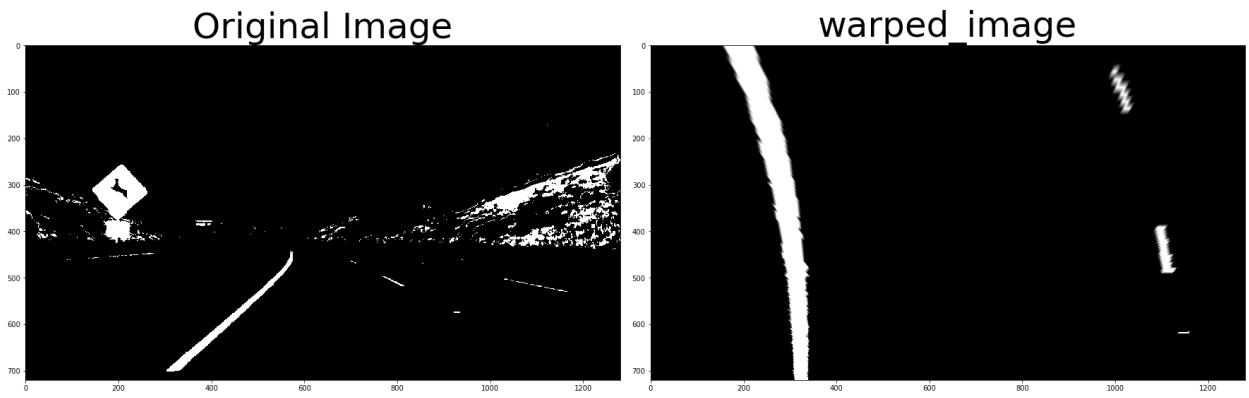
To get the threshold values, I used a method explained by Roy Huang in the following article. (https://medium.com/@royhuang_87663/how-to-find-threshold-f05f6b697a00).
By plotting the pixel intensity of one row of image on the original image, we can guess the threshold values for the yellow and white colors.
I used OpenCV, cv2.threshold and cv2.inRange functions to apply the color threshold and obtain the binary image. The code for color thresholding can be found at In [6].

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform includes a function called `transform_prespective()`, which appears in the In [7] of the IPython notebook. The transform_prespective `()` function takes the image (`img`), as well as source (`src`) and destination (`dst`) points. I chose these points by trial and error and finding the best points which will give me the best output. Here is my src and dst points I used in my code.

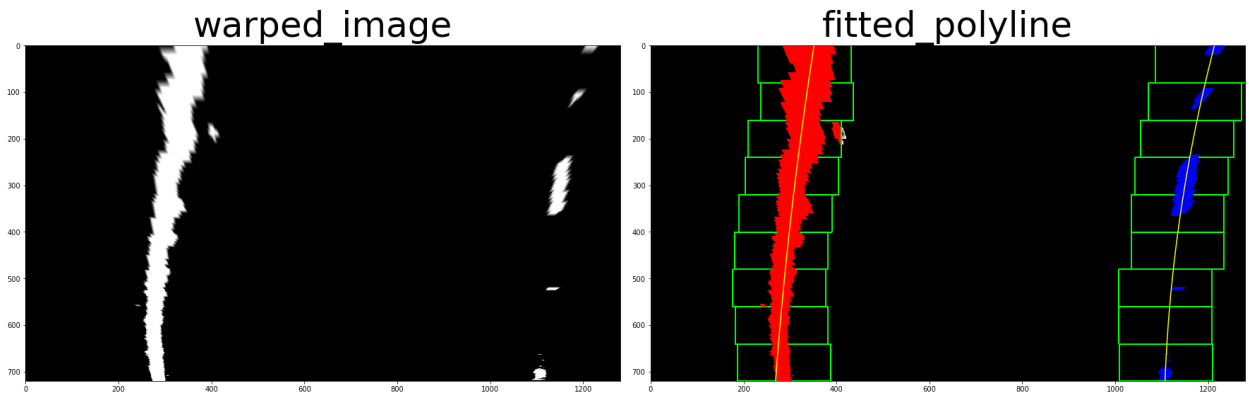
```
src = np.float32([[240,700],[580,460],[705,460],[1080,700]])

dst = np.float32([[250,720],[250,0],[1065,0],[1065,720]])
```

I verified that my perspective transform was working as expected by verifying that the lines appear parallel in the warped image. An output example is shown here.



4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The line detection code could be found at In [9-10] of the IPython notebook. Now that I have obtained the binary warped images, it was time to identify the pixels belonging to the lines and also separate the left and right lines. I plotted the histogram of the binary activations across the image. The maximum values in the histogram plot, gives us the starting x position of the left and right lines. Now that we have the start position of the lines, we can just search small windows starting from the bottom and all the way to the top until all the pixels in each line are identified. After identifying all the nonzero pixels, I used a polynomial fit (np.polyfit()) to find the best line fit for the points. The following picture shows the pixels found on each window, the windows and the fitted 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.

The code used to calculate this could be found at In [11] of my IPython notebook. I used the following formula to calculate the radius of curvature of the lane.

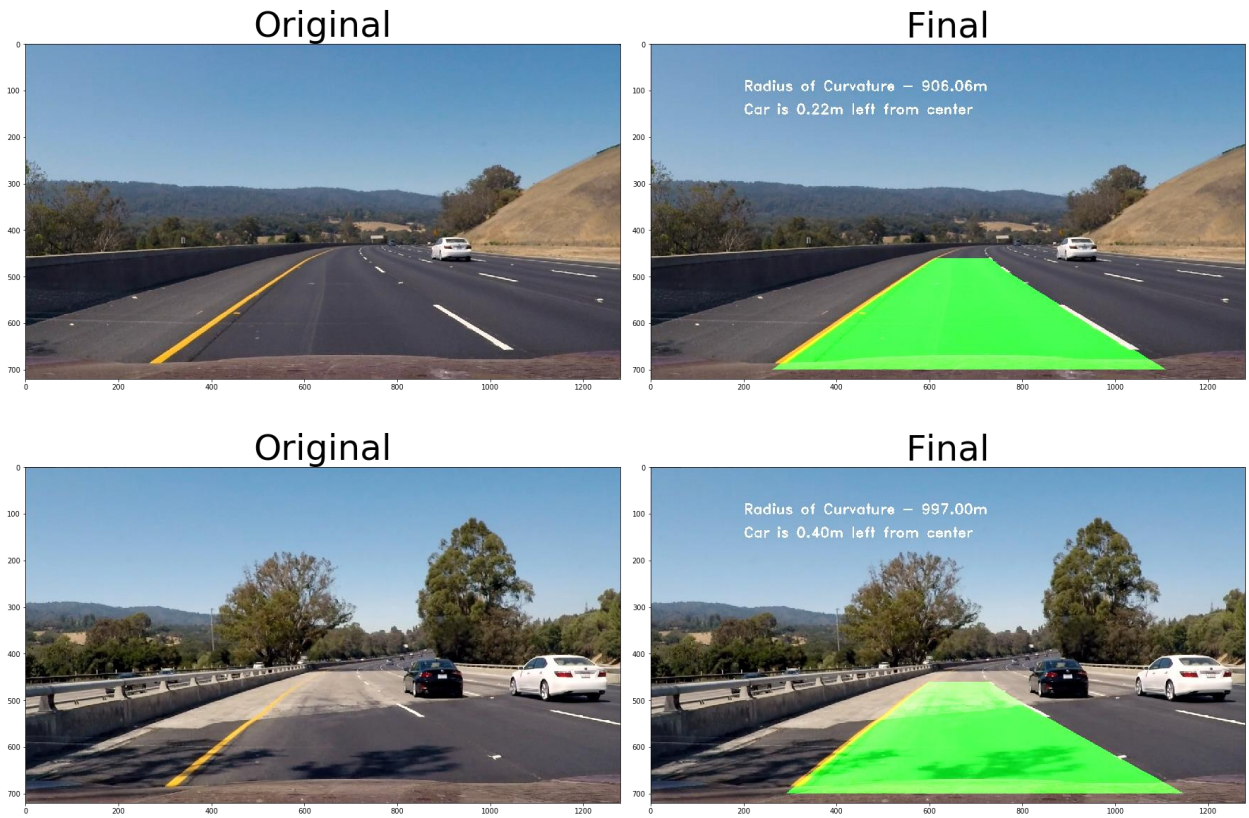
$$\text{Radius of curvature} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

To calculate the radius, first I converted the pixel values to meters and then returned the average value of the calculated radius of curvature for the left and right lane.

To find the vehicle position with respect to center, I assumed that the center of image is the center of car. Then I calculated the pixel positions of the left and right bottom most point and took the average. The difference between this value and the car center, is the car deviation from the center. Note that here too we need to convert the pixel values to meters.

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

The code used for this operation could be found on In [12-14], and the following images are examples of the final image mapping the lines into the image and plotting a polygon between the lines.



Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Project video output can be found inside the “test_videos_output” folder

---

Discussion

1. 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?

The most important part of the project was correct implementation of different thresholding techniques such as color space or gradient method. Based on my experience working with this

project, using a combination of color space thresholding worked well for the project video. However, it seems that this method is not working properly for challenge videos due to the presence of shadow, different lighting and patching conditions. To make this code also work for the challenge videos, we might need to take some pictures from the video, and improve the thresholding technique. This suggests that this method of thresholding is not very robust for different road conditions and some ways of generalized robust thresholding techniques needs to be developed.

Here I also chose my "source" and "destination" points manually to get the perspective view of the lane lines. This was purely based on the trial and error. The code can be improved to make choosing these points automatic and with high precision.

On challenge videos, I encounter errors such as TypeError("expected non-empty vector for x"). This means the line detection algorithm is failing on some frames of the video and needs to be investigated more in detail for that specific frames by taking the pictures of that frame and analyzing it.

To improve the robustness of the lane detection algorithm, the information from the previous frames can be used. For example, when a lane detection fails for a specific frame, the detected lines from the previous frames can be used.