
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

Camera Calibration

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

The camera calibration adjusts the camera image to account for the adverse effects of distortion. Distortion can alter an object in an image making it appear different than it actually is. The camera calibration code takes an object whose dimensions are known then determine a mapping from distorted to undistorted image.

The code for this camera falls primarily on two steps.

1. Obtain the image points and object points
2. OpenCV functions cv2.calibrateCamera() and cv2.undistort()

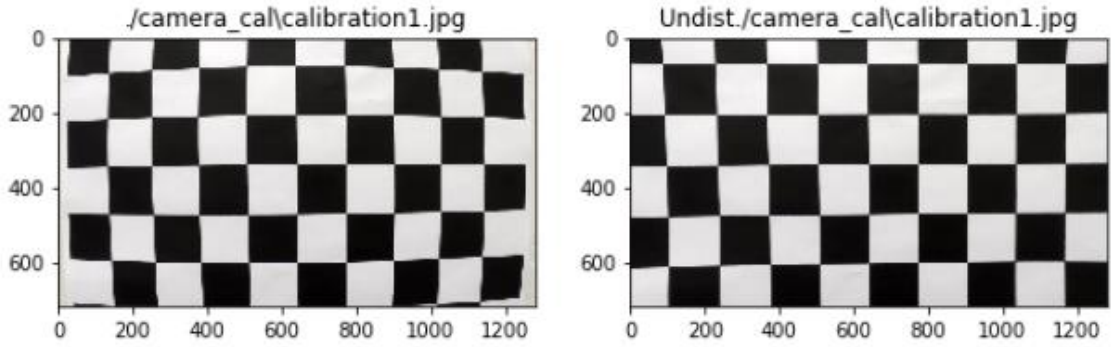
To obtain the object and image points you conduct the following:

- First determine the number of inside corners of the chessboard (nx, ny).

- Convert the image to grayscale
- Use OpenCV function cv2.findChessboardCorners which returns corners if found
- The corners found fall into image points
- The object points assumes a fixed chessboard

To calibrate camera and undistort the image:

- Run cv2.calibrateCamera using object and image points
- This returns the calibration matrix and distortion coefficients
- Funnel these values into cv2.undistort to get an undistorted image



Pipeline (single images)

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

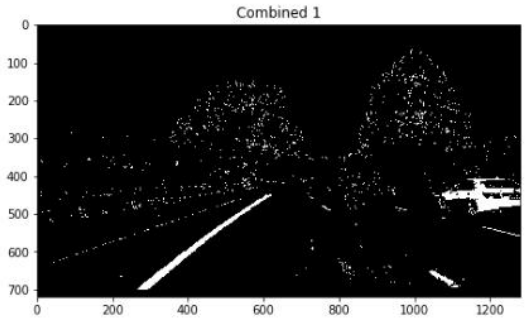
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this

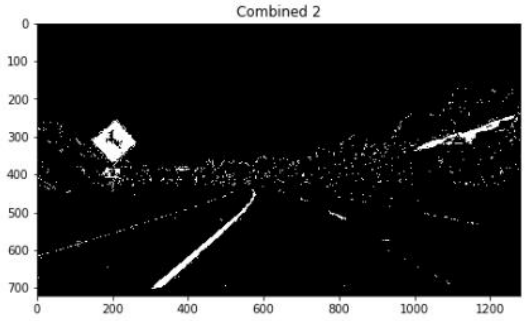
one:

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.

I tested gray, red, green, blue, hue, saturation, lightness, sobelx, sobely, magnitude, and directional threshold methods. I tested these thresholds on six test images. Based on the effectiveness of those thresholds I decided to utilize a combination of red, saturation, lightness, sobelx, magnitude and directional. I applied a binary AND between red, saturation, and lightness as a group. The second group was a binary AND between sobel x, magnitude, and directional. These two groups were OR together for the final binary image.

Here are examples of my binary image threshold





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

The perspective transform takes the existing camera image and transforms it into the birds eye perspective.

The code for this resides in a function called `perspective_trans()`. The function conducts two things:

1. defines the source and destination points
2. Runs cv2.getPerspectiveTransform

This returns a conversion map from source to destination.

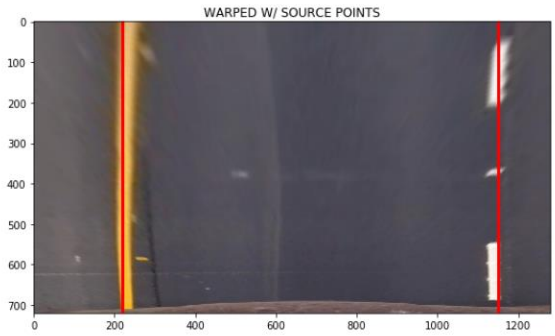
I chose the hardcoded source and destination points

```
# Trapezoid
left_bottom = [220, ysize]
right_bottom = [1150, ysize]
left_top = [580, 475]
right_top = [755, 475]

# Offsets
left_top_offset = 545
right_top_offset = 770

# Define src & dst
src = np.float32([[left_top_offset, left_top[1]], [right_top_offset, right_top[1]], [xsize,ysize], [0, ysize]])
dst = np.float32([[0,0],[img_size[0],0],[img_size[0], img_size[1]],[0, img_size[1]]])
```

This resulted in the following source and destination points:

Source	Destination
545, 475	0, 0
770, 475	1280, 0
1280, 720	1280, 720
0, 720	0, 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.



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

In order to identify the lane-line pixels, I created a function called lane_detect that takes in a binary warped image. The image represents a birds eye view with '1s' representing a pixel identified as a lane. The lane_detect function returns the location of left and right lanes.

The algorithm is segmented into three parts:

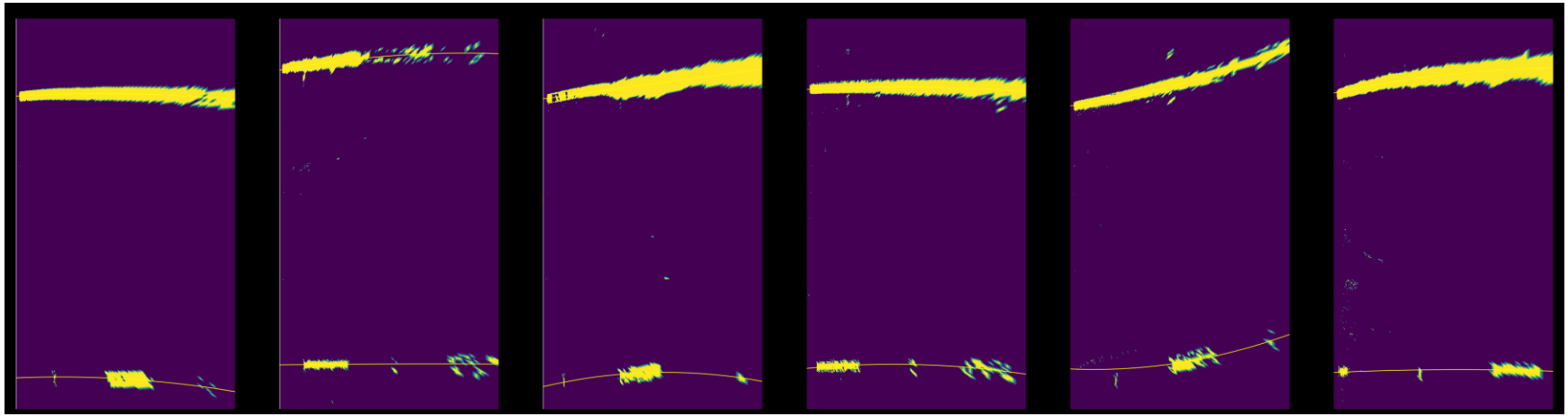
1. Initialization of window and starting point
2. Step through windows one by one
3. Return the lane pixel points

The algorithm identifies the starting point of the lane by creating a histogram of half of the image through summation. The historgram conveys the concentration of pixels. The left and right starting points are selected based on the max concentrations of the lift and right side of the image. This gives a decent estimate of the starting point assuming the thresholds are accurate. For future redesign, I would reduce the image to maybe a quarter due to turns.

For the window, the number of windows and the dimensions of the window are defined. I defined the number of windows to 9 and the width as 100. The height is the total vertical space of the image divided by the number of windows.

The algorithm walks through each window recording and identifying nonzero pixels within the window. If the number of nonzero pixels exceeds the minimum pixel count then the window is re-centered around the mean of the nonzero pixels.

The nonzero pixels locations are extracted into left and right x and y positions. These values serve as inputs to the np.polyfit function to create a polynomial line. Below are example images of the polynomial 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.

In the Curvature Calculation section of my code, I calculate the curvature of the lane and position of the vehicle with respect to center. To start, I took the predefined meters to pixel conversion from the lecture.

As an aside, at my work we use a micrometer (NIST calibrated) then put it under the camera and take an image. Then I use imageJ a NIH S/W tool to measure the known micrometer distance usually 1mm. Then I can establish a pixel to 1 mm conversion.

The actual algorithm is taken from the Measuring Curvature lecture.

Radius of Curvature

The radius of curvature (awesome tutorial here) at any point x of the function $x = f(y)$ is given as follows:

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

So, our equation for radius of curvature becomes:

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

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

In the Draw Lane Boundaries section of code, I provide images of where the lane area is identified. Example images are shown below.

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!).

The video can be found in the folder provided with this template.

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?

I encountered a challenging bug in my code. It was challenging because I separated the pipeline code with the individual parts of the pipeline. The individual parts of the pipeline worked but together in the pipeline code I introduced a bug. The bug ultimately was utilizing the full RGB image instead of only one color set. The problem didn't manifest itself in a straight forward manner.

I wanted to try the challenges more. I did try the first challenge and my filter was not strong enough to handle the variability of the road. I could tell the sobel filters had trouble with the gradient of the middle crack in the road. If I have time I would like to refine the filter.

The other aspect I would have liked to explore would be to make the thresholding more dynamic. For example, It could take an average and assume that is daylight then construe an offset for shadow and darkness.

In the department of fitting to a lane, I think more filtering could be done to better remove outlying line configurations that seem impossible. For example, the next lane should not deviate from the original too much as it is not feasible. That could have been built into the code as an alarm .