Submitted by Kyle Martin

Project 4
Advanced Lane Finding

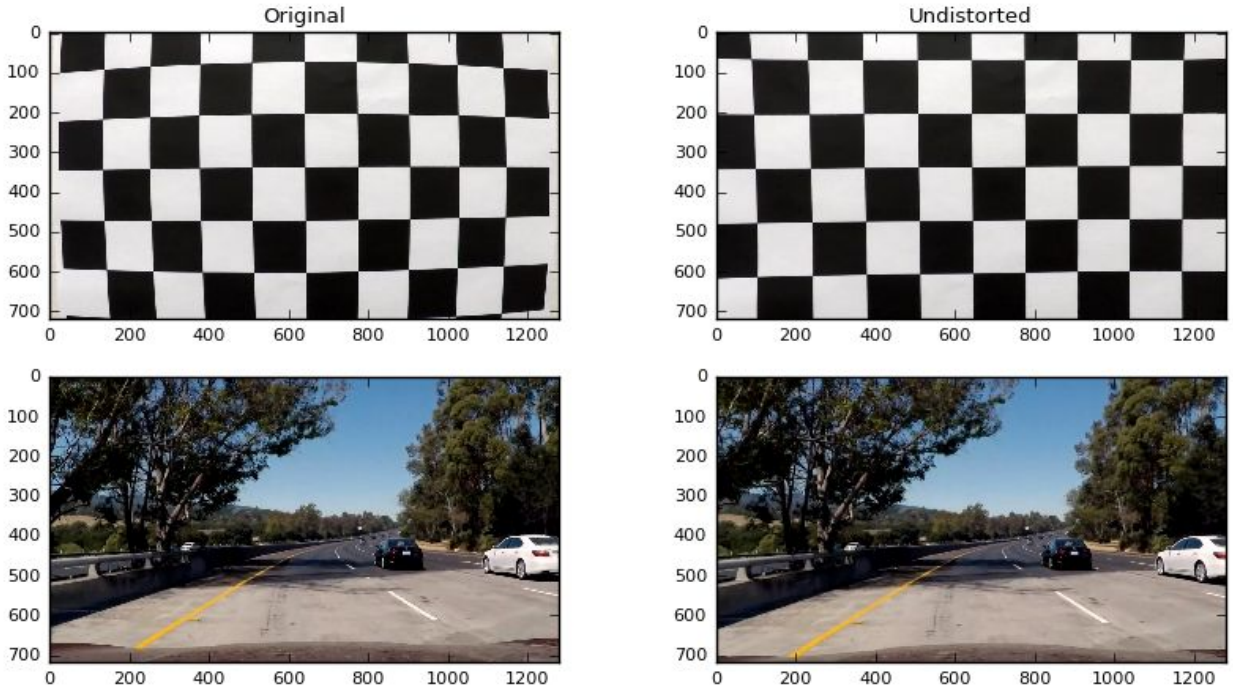
The goal of this project was to detect lane lines in a video stream using modern computer vision methods. These methods include color space thresholding, perspective transformation, polynomial fitting, and noise reduction.

Camera Calibration

All camera lenses introduce distortion into images due to their convex shape. This distortion can be a big problem when trying to determine the geometry of objects, accurately, in 3D space. Especially when your self-driving algorithm is trying to evaluate your distance to other cars!

To correct for this, I used the provided chessboard calibration images and OpenCV's calibrateCamera() function to create a distortion matrix. This matrix can be used to undistort any image taken with this camera.

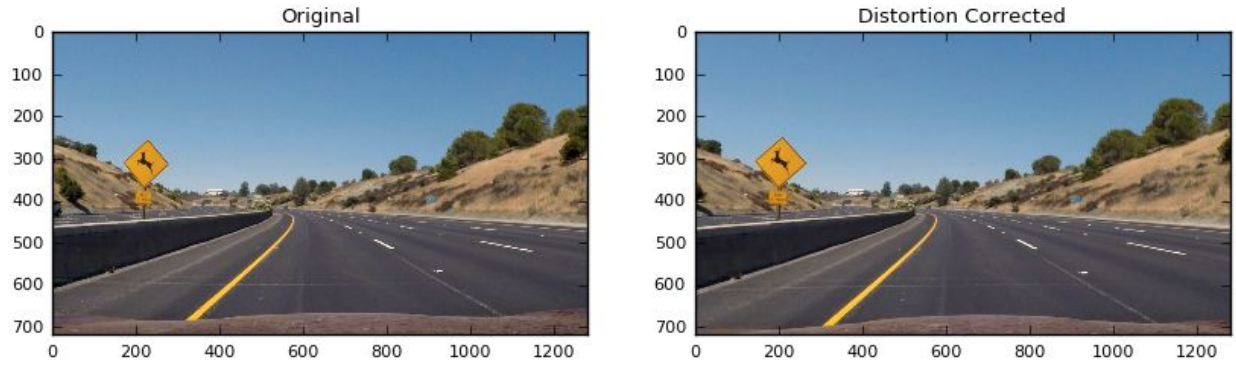
Code for for calibration is contained within Camera.py around line 56.



Pipeline

Distortion Correction

Each video frame is first undistorted using Camera.undistort() inside of Camera.py (line 32). This assumes that the camera distortion has already been previously calculated.



Lane Mask Generation

After distortion correction, it's time to extract the pixels of lane lines. I used many techniques to create a robust daytime extraction pipeline. This was by far the most time consuming part of the project because it took a long time to find a combination of methods that behaved robustly under various environmental conditions like bad exposure and shadows.

The code related this extraction method technique is located in mask_generator.py.

Below is the lane extraction pipeline steps:

1. Vertical Cropping
Reduce the amount of search area by only looking at the bottom half of the image. I found through trial and error that cropping all pixels below 400px worked well.

This code can be found at line 142 in mask_generator.py.

2. Colorspace Thresholding
Through experimentation, I found that the Y and U channels of the YUV color space highlighted edges rather well. I also found that the S channel of the HLS color space also highlighted lane lines rather well too.

Instead of logically ORing these channels to form a mask, I created a fake image that consisted of these three channels (Y, U, S) and averaged it into a grayscale image. This technique boosts agreements between the mask and discounts differences.

This code can be found at line 144-148 in mask_generator.py.

3. Sobel Operations
Next up is to extract more geometrical information by studying the gradient of the thresholded mask. By decomposing the gradient of the image into magnitude and direction components we further filter pixels that don't belong.

First, pixels that don't have a strong magnitude component are removed.

Second, pixels that don't have a mostly vertical direction component are removed. This is done because we're assuming that the mask has a bird's eye view of the road and therefore lane lines should be fairly vertical.

All together this step significantly helps removes noise and filter for shapes that are mostly vertical.

This code can be found at line 153-154 and 161-166 in mask_generator.py.

4. Yellow Thresholding
The above steps do a great job extracting white lane lines however not yellow ones. In order to compensate for this I threshold a specific range of yellow in the HSV colorspace.

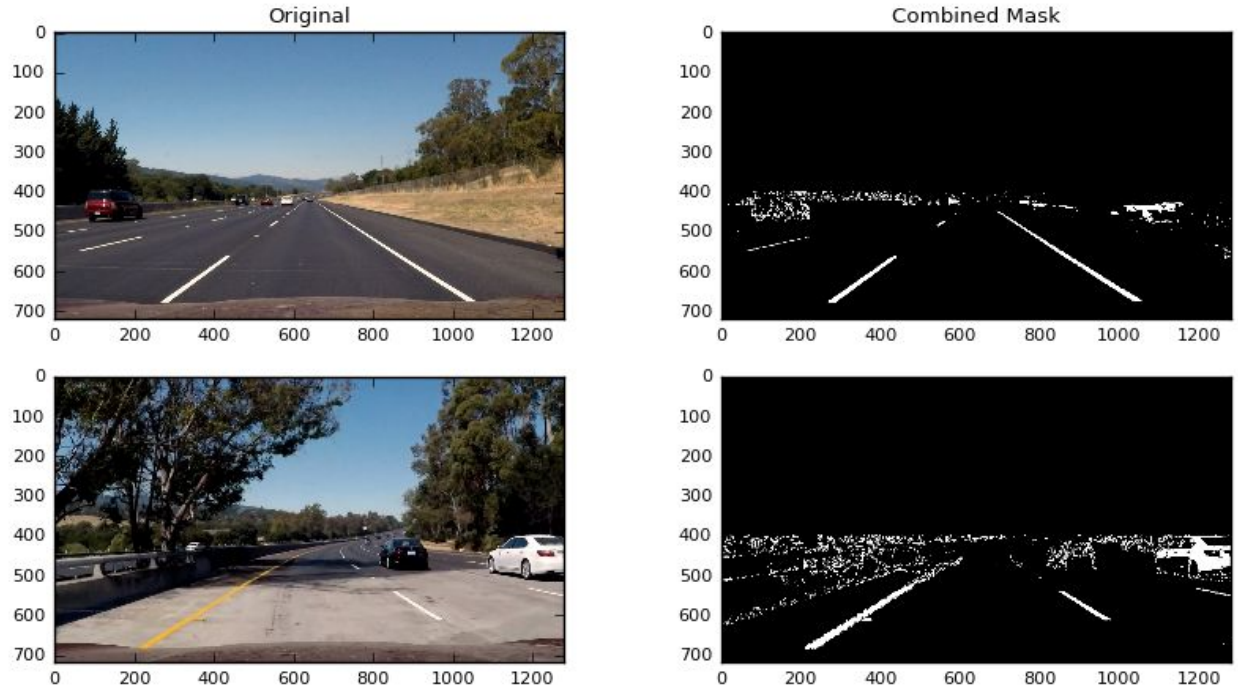
This code can be found at line 69-80 in mask_generator.py.

5. Lighting Invariance
Since object pixel values can vary based on lighting conditions, I implemented a simple color boosting step that filters colors that aren't above a percentile of intensity value.

This code can be found at line 97-108 in mask_generator.py.

6. Noise Reduction
The final step is to apply a 2D convolutional filter to reduce stray pixels by considering how many neighboring pixels each pixel has. If a pixel doesn't have at least 5 neighboring pixels then it's removed.

This code can be found at line 111-130 in mask_generator.py.



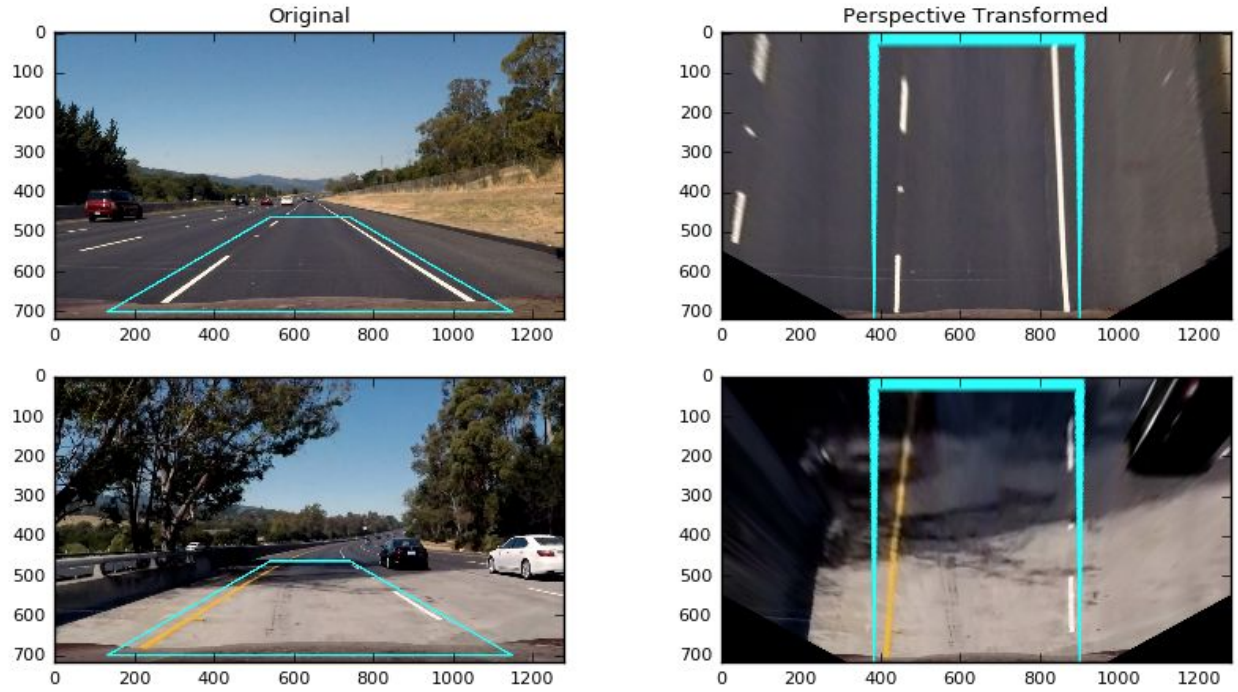
Perspective Transformation

Before we further analyze the frame for lane geometry, it's helpful to isolate the region ahead of the car. Not only that but we'd like to transform the image to appear as if we're looking top down on it. To do this I used OpenCV's getPerspectiveTransform() and warpPerspective() functions.

I created a class called PerspectiveTransformer that handles the transformation between the real-world perspective and bird's eye perspective.

To get a bird's eye perspective I use PerspectiveTransformer.transform() and to reverse this perspective to real-world I use PerspectiveTransformer.inverse_transform().

This code can be found at line 24-36 in perspective_transformer.py.



Lane Finding

After obtaining a thresholded, bird's eye view of the street it's time to find the left and right lanes in the image. Although a lot of filtering was done, there still could be a lot of noise so we need a way to find the pixels that just belong to the, now, vertical lane lines.

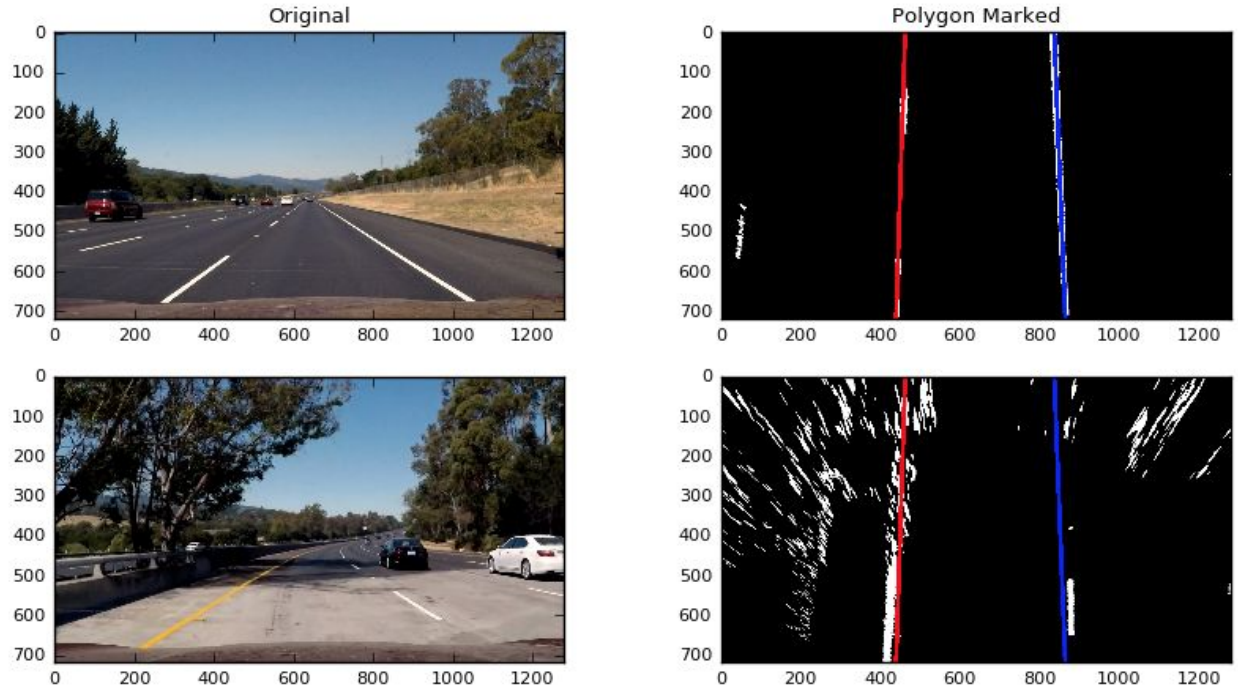
The approach I decided to take is two fold:

1. Slice the image in half vertically to search for the left and right lanes independently
2. Use a sliding histogram to find where along the x-axis there is a clustering of vertically connected pixels (we're assuming those will be lane lines)

Step 1's code can be found at line 7-49 in histogram_lane_detection.py.
Step 2's code can be found at line 49-76 in histogram_lane_detection.py.

Applying the sliding histogram across a whole frame is quite computationally expensive. So after the lane is found, I then only use the sliding histogram search around the

previously known location of the lane. If the lane is not found within this reduced search area, the whole frame is searched from scratch.



Polynomial Fitting

After the pixels belonging to each lane have been found we can try to find 2nd order polynomials to fit each lane's pixels. To stabilize the curve prediction and reduce visual jitter,I found using a simple moving average filter on each polynomial term really helped.

This code can be found at line 7-33 in lane_utils.py.

I also added a few sanity checks to determine if the predicted lane is similar to the previously known good lane position and curvature.

This code can be found at line 58-74 in lane_utils.py.

Lane Curvature

If the left and right lanes are currently detected, we can calculate the curvature of the road by averaging their polynomial terms to find a curve representing the center of road. To find the curvature of a road I used the follow equation:

radius of curvature = [1+(dy/dx)^2]^3/2 / (d^2y/dx^2)

This code can be found at line 97-114 in lane_utils.py.

Center of Lane

Another interesting thing we can measure is how far the car has deviated from the center of the road. If we assume the camera is mounted in the center of the car we can use the center of the road polynomial to calculate the x-intercept and measure how far that is from the center of the image.

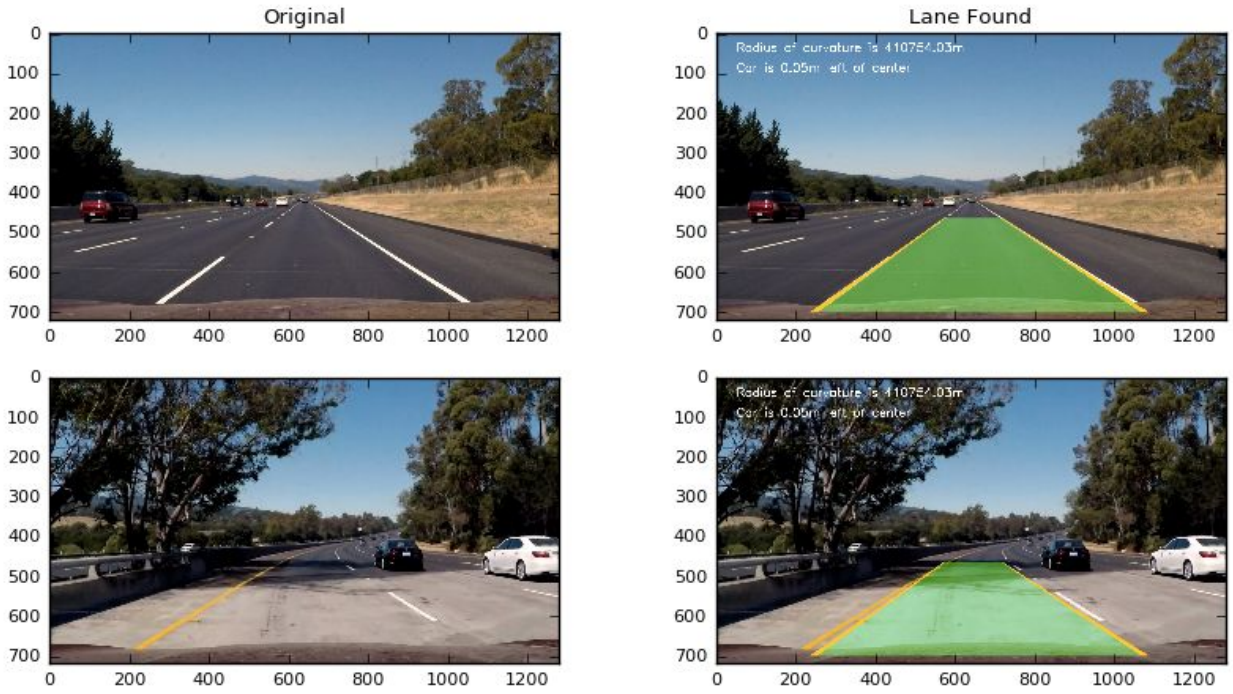
Heads Up Display (HUD) and Inverse Transform

The final step is to visualize the road geometry frame-by-frame, along with the deviation from the center of the lane and curvature of the road.

I used OpenCV's drawing functions to draw:
- The lane area polygon
- The yellow lane lines
- The detected lane ahead
- The deviation from the center of the lane
- The curvature of the road

After drawing on a blank image with the same shape as the warped image, I used the PerspectiveTransformer.inverse_transform() to project the drawings back on the the real-world frame. The final step was to overlay the HUD drawings on top of the original frame.

This code can be found at line 47-71 in lane_finder.py.



Conclusion

Overall this project was slightly challenging. It was a great introduction to more classical computer vision techniques. These techniques work well for highway drives but will not perform well in urban or rural environments with much more noisy environments. I can definitely see why sensor fusion and high-resolution maps are incredibly important in production self-driving cars.

I'd like to see how a neural network performs against more formal methods like this. I think a production level approach would make use of both neural and classical methods.