Project 4 - Advanced Lane Finding

Udacity Self-Driving Car Nanodegree

February 13, 2017

Robert Crowe

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

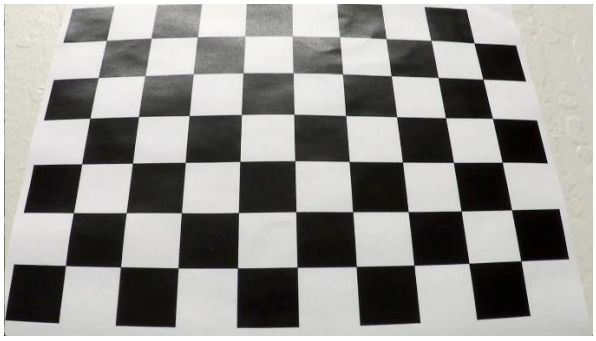
Camera Calibration

1. **Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?**

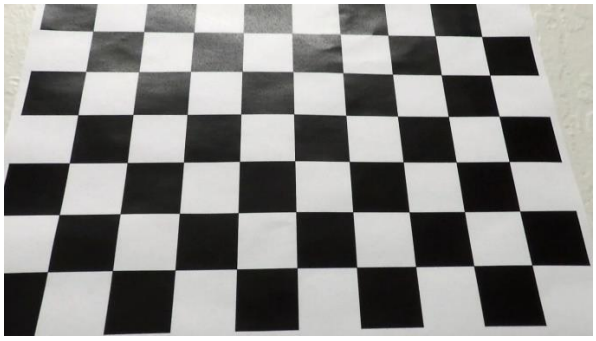
I calibrate the camera matrix and distortion coefficients in calibrate_camera.py and save the result to a pickle file. The function collect_points() uses cv2.findChessboardCorners() to collect all the object and image points from all calibration images. I then run a test using the matrix and coefficients with cv2.undistort() and save the result, then reload the pickle file and run a second test, comparing the results to make sure that the pickle is good. Mmmm, pickley.

The correction is subtle, but notice that the camera lens introduces curves in the lines, which should be straight. The correction corrects that.

Original



Camera Corrected



Pipeline (Single Images)

1. **Has the distortion correction been correctly applied to each image?**

My pipeline runs in process_frame(), which is in main.py and uses correct_image() from enhancer.py to load the pickle and correct the image:

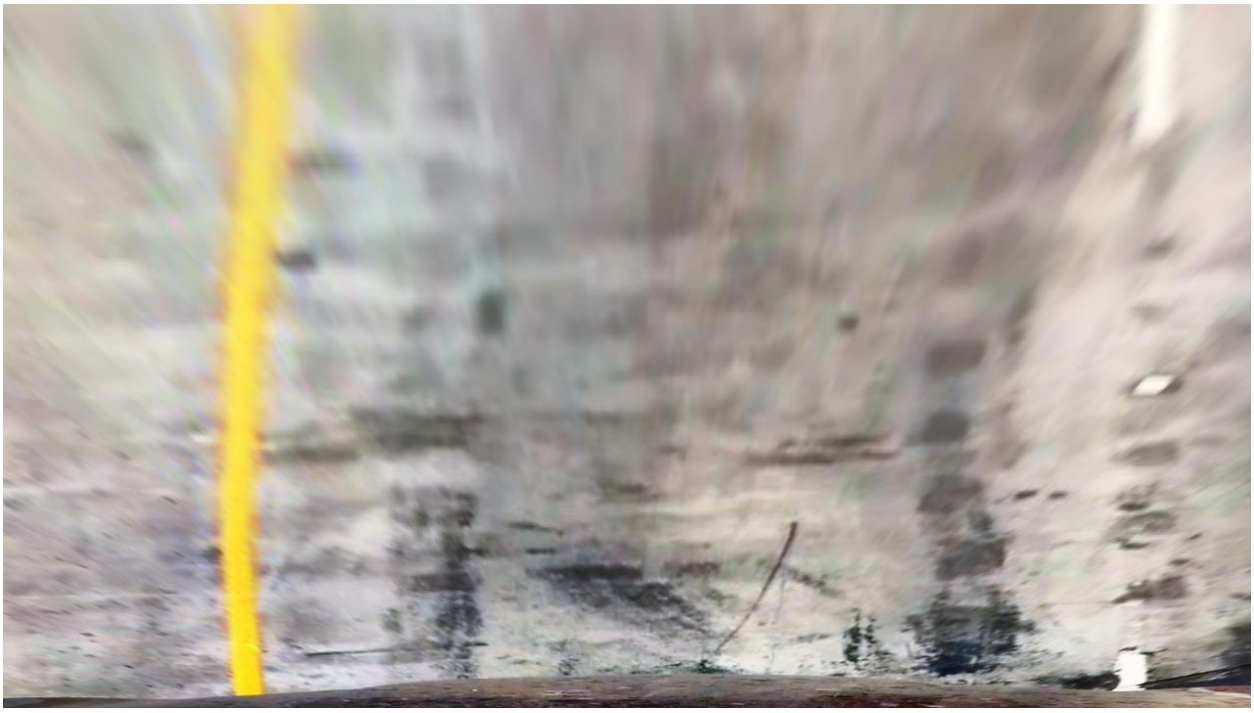
2. **Has a binary image been created using color transforms, gradients or other methods?**

Through experimentation I evolved my sequence to do this a little differently. First I apply a CLAHE (Contrast Limited Adaptive Histogram Equalization) correction to adjust contrast and brightness:

Original:

CLAHE Corrected:

Then I transform the corrected image to a birds eye view (details below):



I then use the birds eye view to run the transform to a thresholded binary image:

The transform to a binary thresholded image is in enhance_lines() in enhancer.py. After converting from RGB to HSV I apply color thresholds, determined through experimentation, to locate both yellow and white lines. I found that for white I needed at least two sets of thresholds, because white was so often present in the background, resulting in a very noisy image:

Noisy Whites:

Corrected Whites:

I use the sum of the pixels in the color thresholded image to detect when I have a noisy image. Lines should only set a limited maximum number of pixels, so if I have too many set then the sum of the image pixels will be too high, indicating that there are pixels that are not part of the lines.

Originally I experimented with thresholding in HLS space, and with the Sobel transform and gradients from the code exercises (see sobel.py), but I found that I was able to achieve good results with color thresholding alone and reduce the processing requirements. Since I was hoping to process frames in near real time, as a car will have to, I wanted to minimize processing.

It's very interesting to save the binary thresholded images to the video, rather than the original view, since it gives you a look at how well the image enhancement is going through the video. See enhanced.mp4 for an example.

3. **Has a perspective transform been applied to rectify the image?**

I created a class Perspective in perspective.py to handle the transformations to and from the birds eye view (see images above). I chose the calibration points largely through experimentation, knowing that the points should form a rectangle in real space on the road, and that straight lines on the road should result in straight lines when viewed from above. I also wanted some margin on the left a right to allow for curved lines, which otherwise exit the image before reaching the top.

4. **Have lane line pixels been identified in the rectified image and fit with a polynomial?**

I first find the lines closest to the car by using find_line_starts() in line_start.py when I don't have them already from a previous frame. If I already have the lines from a previous frame, and the data looks good, I use those to avoid the additional processing of finding them again.

find_line_starts() does a walk down the image, starting with the full image and taking the bottom half at each iteration until it reaches a minimum size. By starting with the whole image we account for gaps in dashed lines, but as we walk down we account for line curvature towards the top of the image. If we can find the lines in the last slice closest to the car, they are likely to be the most accurate, so those pixels are included in all iterations.

I then use sliding windows to walk up the left and right lines, locating points along the lines as I go. See walk_lines() in slider.py. Both walk_lines() and find_line_starts() use a moving average of the pixels and look for peaks in the signal using scipy.signal.find_peaks_cwt(). When I can't find a peak, for example

during the gaps in dashed lines, I use the coefficients from the previous good data to calculate where the lines should be, or use a mean of the current line direction when I don't have coefficients.

Using numpy.polyfit() I fit a 2nd order polynomial to the points that are found with the sliding window, and then use the coefficients to generate a new, smoother line.

5. **Having identified the lane lines, has the radius of curvature of the road been estimated? And the position of the vehicle with respect to center in the lane?**

I use the X positions of the lines and the car center (which is also the image center) to determine the position of the car in the lane. I estimate the meters per pixel in real space and use numpy.polyfit() again to calculate the curvature of the lines in real space, since the previous numpy.polyfit() was using pixels in birds eye space. I add the car position and curvatures along with the lines and lane overlay to the image before returning it.

Pipeline (video)

1. **Does the pipeline established with the test images work to process the video?**

It works well on project_video.mp4, see project_video_result.mp4

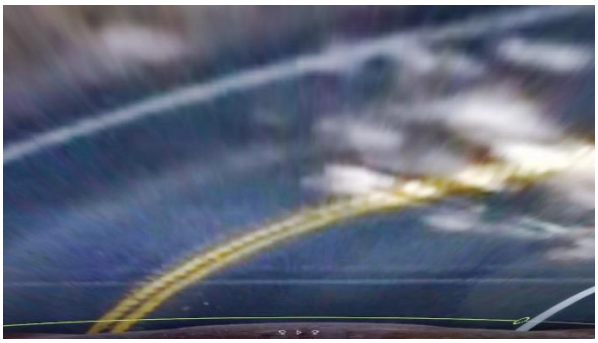
It has issues on both challenge_video.mp4 and harder_challenge_video.mp4. I have an update in progress to improve these results, and to get a sneak peek at check out challenge_video_result.mp4

Discussion

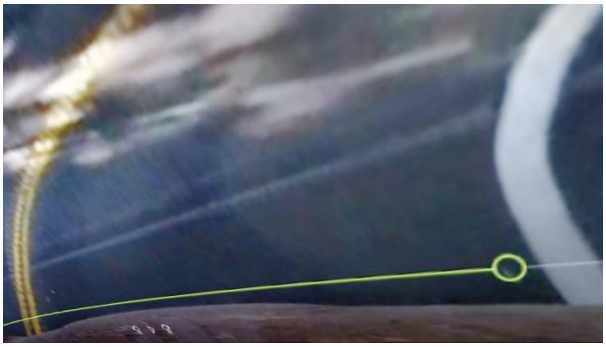
The approach that I took was a slight departure from the standard pipeline which evolved through experimentation. One of the things that I noticed was that using a single set of registration points for the transform from the real view to a birds eye view will probably be inadequate in a real car, because significant curves in the road cause the lines to exit the birds eye view and prevent finding the lines after they exit.

Original:

Exiting Lines (Birdseye):



Alternate Transform:



I haven't really spent time on it, but I think it's possible to find a transform that straightens these lines. The transform itself will then become a measure of the curvature of the road. I can imagine a relationship between the steering angle and the transform which could be used to dynamically modify the transform to fit the road.

One of the assumptions that I started from was that the car in this project was staying in the lane. I might propose that there are at least two lane finding modes:

1. When we want to remain in a lane
2. When we are transitioning between lanes

This project focuses only on mode 1, where we want to remain in the lane. Transitioning between lanes means that the lane positions relative to the car will vary more widely, which especially impacts the initial finding of the start of the lanes.

The Real Deal

I can see from this project that there are many road conditions that make this difficult for anything but the ideal conditions of project_video.mp4. At night in the rain for example, lines will be very hard to see and reflections from wet pavement will add a lot of noise to the camera image. Often the only visible lane markers are the reflectors, which are much smaller and more widely spaced than dashed lines. There will also need to be efficient enough code and enough processing power to perform lane finding in real time.

I can't help but wonder if there isn't a machine learning approach that might be more robust than this CV style approach. Training a RNN, perhaps an LSTM, to take a time series of images and predict line position would be an interesting approach to try. By training in a wide variety of road conditions it may very well be more adaptable.

Another very interesting approach that I'd love to try is to train a Generative Adversarial Network (GAN):

SAD-GAN: Synthetic Autonomous Driving using Generative Adversarial Networks

https://arxiv.org/pdf/1611.08788.pdf

Learning from Simulated and Unsupervised Images through Adversarial Training

https://arxiv.org/pdf/1612.07828v1.pdf