writeup_template

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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

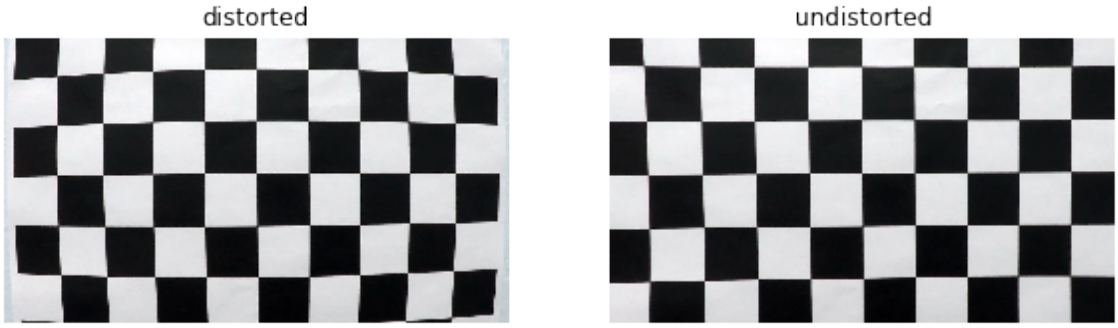
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the third code cell of the IPython notebook located in "./examples/Submission.ipynb". The code cell has title "Distortion correction".

I start by preparing "object points", which 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.

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one (colors in BGR):

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 code for this is in the threshold() function in the section "Gradient and Threshold" of Submission.ipynb.

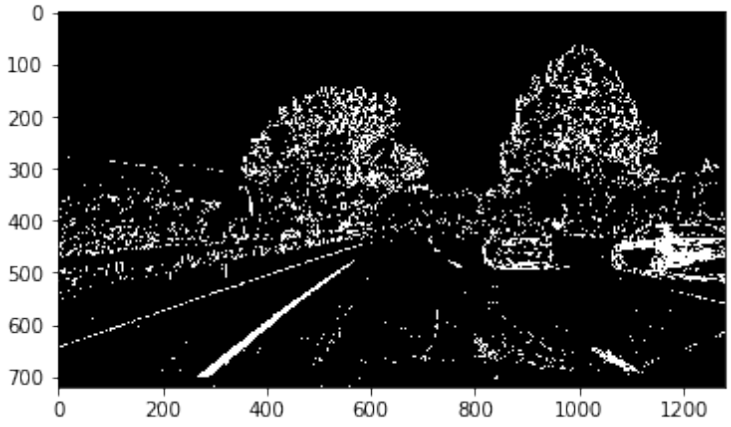
I used a combination of two thresholds:

1. Sobel gradient derivative along X coordinate. This is useful in detecting lanes since they are perpendicular to X axis.
2. S channel of HLS color space. I found that S provides a useful gradient compared to other channels - like R,G,B,H and L. This is perhaps because the colors of lanes have high saturation with bright whites and

yellows.

I chose threshold range of (170,255) for S channel and (20,100) for the Sobel X gradient.

Combining these two together, here is a sample output:



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 perspective transform is in the sub-section "Warp/transform image" under section "Perspective transform". It involves a function warp() that accepts an image, source points and destination points.

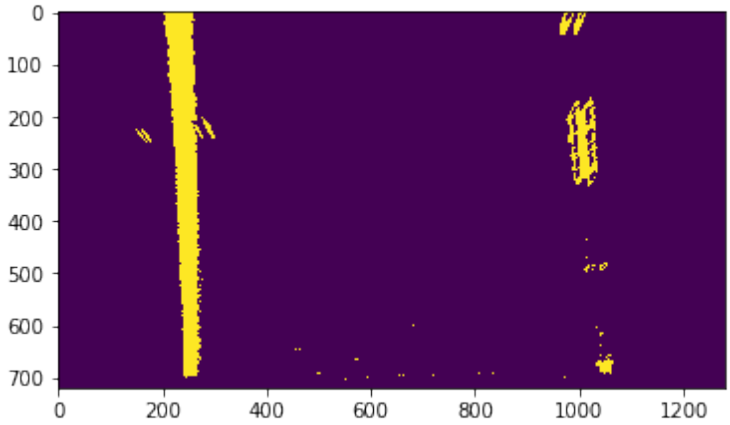
As shows in the lecture videos, I hardcoded the source and destination points. They were set to values which "looked good" across multiple sample images.

This resulted in the following source and destination points:

Source	Destination
762, 275	1160, 10
1160, 686	1160, 686
267, 686	267, 686
568, 475	267, 10

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.

Here is a source image and the warped image -



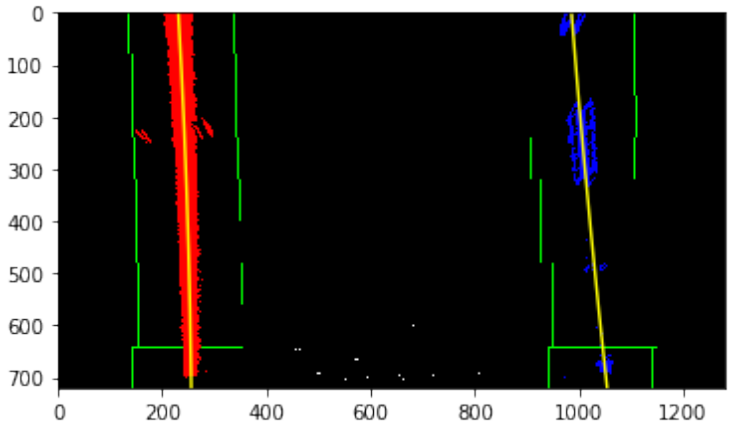
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

I applied the sliding window approach described in the course to identify lane pixels. Code for this can be found in the "Sliding Windows search" section.

Breaking it down, this approach is actually a series of steps:

1. Applying color and gradient thresholds (as described above). This results in a black and white image where the lanes are small areas of white.
2. Take a histogram of the bottom portion of the image.
3. Find the peaks of the histogram. The biggest peaks should be the 2 lanes.
4. Divide the image into vertical strips or windows (9 windows works fine)
5. Look in the vicinity of the peaks for non-zero x and y points. Append these points to a list.
6. Use np.polyfit() to fit a polynomial to discovered points

Here is the image from Q3 with the regression lines (thin yellow line = regression line)



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.

Check section "Radius of curvature and vehicle position" in Submission.ipynb for implementation.

To do this, we transform the points in x,y space to one where the points represent meters/pixel in x and y dimension. I used constants 3.7/700 and 30/720 for meters/pixel in X and Y spaces, respectively.

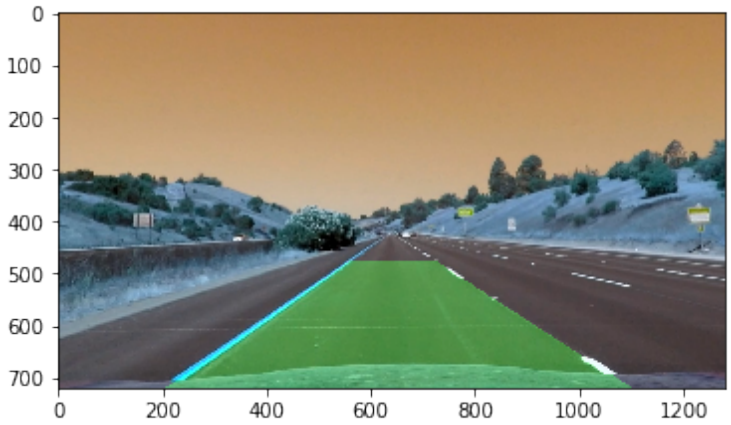
Once the points had been tranformed to the new space, I used np.polyfit to fit new regression lines that account for curves. Once this was done, radius of curvature and vehicle positions were calculated using the formula from the lecture videos.

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

I implemented this step in function "warp_back()" in the section "Image processing pipeline" of Submission.ipynb.

This was exactly the inverse operation of the perspective transform. I used the same SRC and DST points for the (un)warping to compute **inverse** perspective matrix.

Here is an example of my result on a test image:



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!).

Here's a link to my video result

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?

Drawbacks of this pipeline:

1. The lane detection algorithm naively searches the entire image which is computationally intensive.
2. No "memory". If a successive frame does not have well formed lanes, the algorithm will not "remember" the last known good lane.

Where will this fail:

1. Images without well defined lanes. For example, video of a road after heavy snowfall.
2. The color thresholding logic is not robust enough at this point to various light/shadow combinations. For example, on road where the color of the road changes frequently. This could be due to shadow or due to difference in materials used on the road.

Ways to make it more robust:

1. Add a "memory" state for last known good line. This should be used for a few frames if a lane is not available. After few frames, the lane can be recomputed.
2. Consider the case where there are no lane lines on the image. This could be solved by guessing the lane lines based on distance from the kerb/edges. Lane widths vary within a certain fixed range. Given image of an entire road, the algorithm should be able to safely predict a conservative estimate for where the lanes should be.