Advanced Lane Finding Project Report

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

The rubric points will each be considered below.

Writeup

Camera Calibration

1. OpenCV functions were used to compute the camera matrix and distortion coefficients.

The OpenCV functions `cv2.findChessboardCorners` and `cv2.calibrateCamera` were used to compute the camera matrix and distortion coefficients, mtx and dist, respectively.

I used the 9x6 chessboard images provided in the "./camera_cal" directory, which required a small code modification. To test calibration success, I used the distortion matrix and distortion coefficients to undistort one of the calibration images in "./camera_cal". An example of a successfully undistorted calibration image is provided below.

The code for this step is in code cell 1 of the IPython notebook "./Advanced-Lane-Lines.ipynb".

To carry out calibration, start by preparing "object points", which will be the (x, y, z) coordinates of chessboard corners in a 3D space. Assume the chessboard is fixed on the (x, y) plane, so that z=0. Now `objp` is a a coordinate array and `objpoints` will be appended with a copy of it every time corners for a chessboard are successfully detected in an image. Similarily, each successful chessboard detection will result in `imgpoints` appended with the (x, y) pixel position of each of the corners in the image.

Next, `objpoints` and `imgpoints` are used to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. Applying this distortion correction pipeline to the test image from "./camera_cal" using the `cv2.undistort()` gives the result in figure "undistorted output."

Figure 1: undistorted output

Pipeline (single images)

1. An example of a distortion-corrected image from the video feed.

To demonstrate this step outlined above, I applied the distortion correction to one of the test images like the one in figure "test image."

2. Use of color transforms and gradients to create a thresholded binary image with example of a binary image result.

I used a combination of color and gradient thresholds to generate a binary image, see function `color\_gradient\_threshold` in code cell 4 of "./Advanced-Lane-Lines.ipynb". An example of output for this step is in figure "binary combo example".

3. Perspective transform and example of a transformed image.

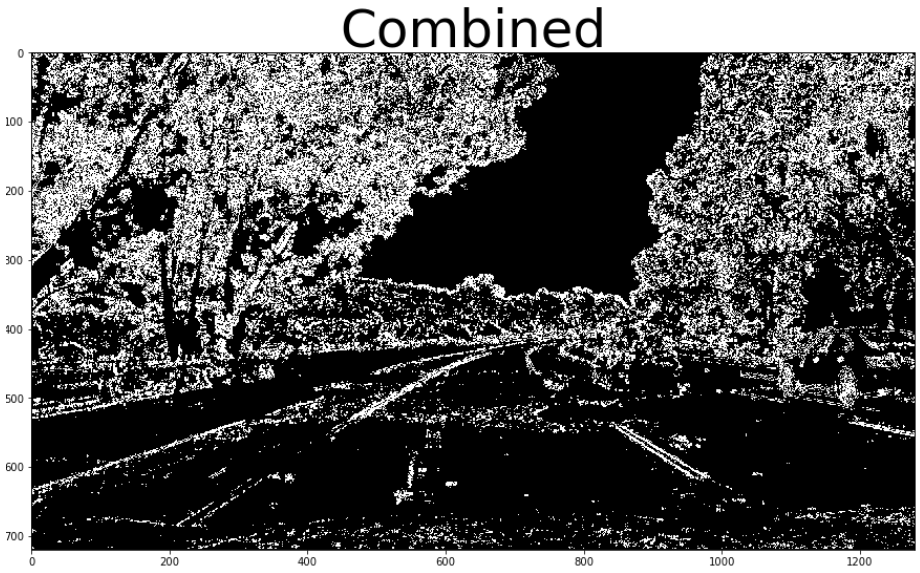


Figure 2: binary combo example

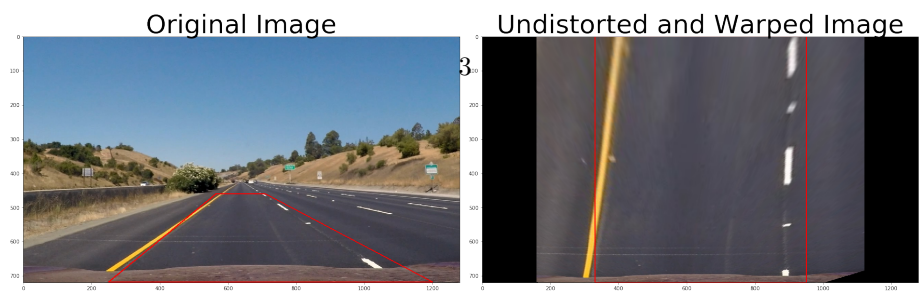
The code for my perspective transform is in code cell 5, in the `perspective_transform()` function. It takes an image (`img`) as input and uses source (`src`) and destination (`dst`) points to return a warped, i.e. a bird's eye view of the input image. The hardcoded source and destination points are set up as:

```
point_src = [[565,460],[710,460],[1200,718],[250,718]]
src = np.float32(points_src)
offsetx = 330
offsety = 0
points_dst = [[offsetx, offsety], [image_size[0]-offsetx, offsety], [image_size[0]-offse
dst = np.float32(points_dst)
```

This gives the following destination points:

Source	Destination
565, 460	330, 0
710, 460	950, 0
1200, 718	950, 720
250, 718	330, 720

This perspective transform worked well. Verifying the `src` and `dst` points superimposed over a test image and the warped image shows apparently parallel lines in the warped image; see the figure "warped straight lines."



where to search for the lines. With that as a starting guess, a sliding window around the line centers is used to follow the lines to the top of the image. Figure "color lines fit" shows the result of applying `find\_lanes()`.

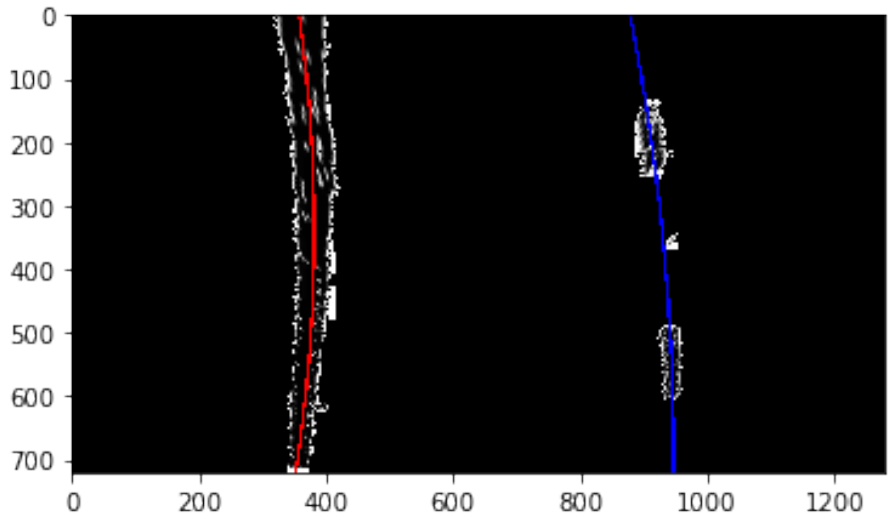


Figure 4: color lines fit

5. Calculation the radius of curvature of the lane.

This is done in cell 8 at the end of `find\_lanes()`.

6. Example image of result plotted back down onto the road identifying the lane area.

This is done in cell 8 at 25 lines before the end of `find\_lanes()`, using warpedPerspective again but with the inverse matrix Minv. An example of this result from a test image is in the "example output" figure.

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.

Figure 5: example output

Discussion

1. Problems or issues found during implementation of the project.

It's noticeable that when there is a bump in the road, the lanes are jerked then get back to normal. An improvement would be to sense that bump and correct for it. Finding appropriate `src` and `dst` seemed more heuristic than anticipated, so the method to determine these points could be made more efficient and precise.