Advanced Lane Finding Project

Report by Cristian Cucchiella

Objective

The goals / steps of this project are the following:

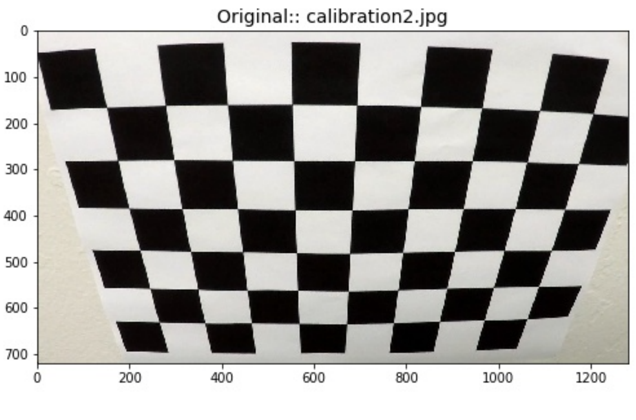
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- 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.

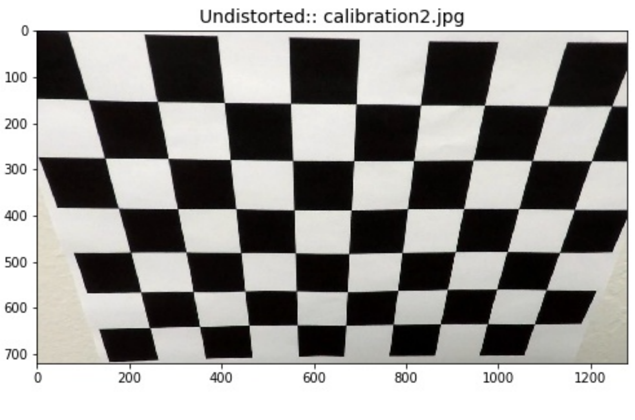
The code for this project is contained in the IPython notebook named P2.ipynb.

Camera calibration

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

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:

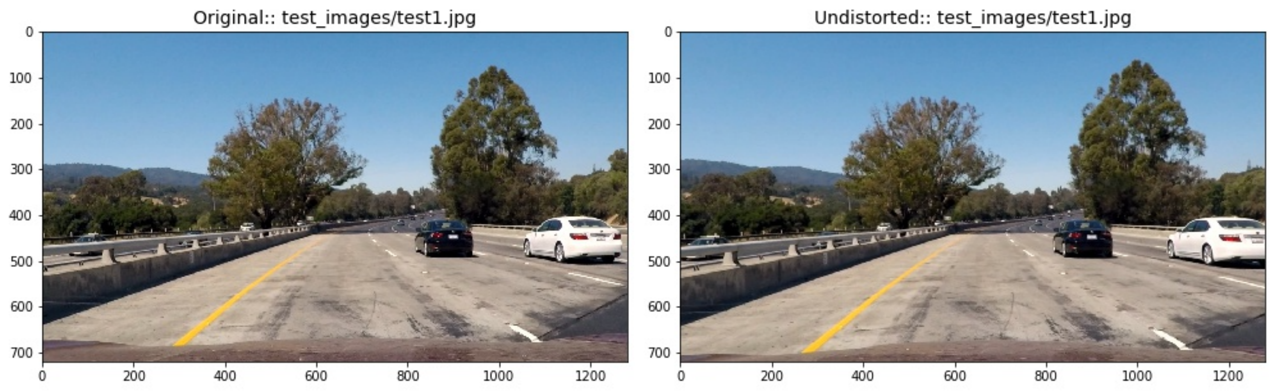




Pipeline (single images)

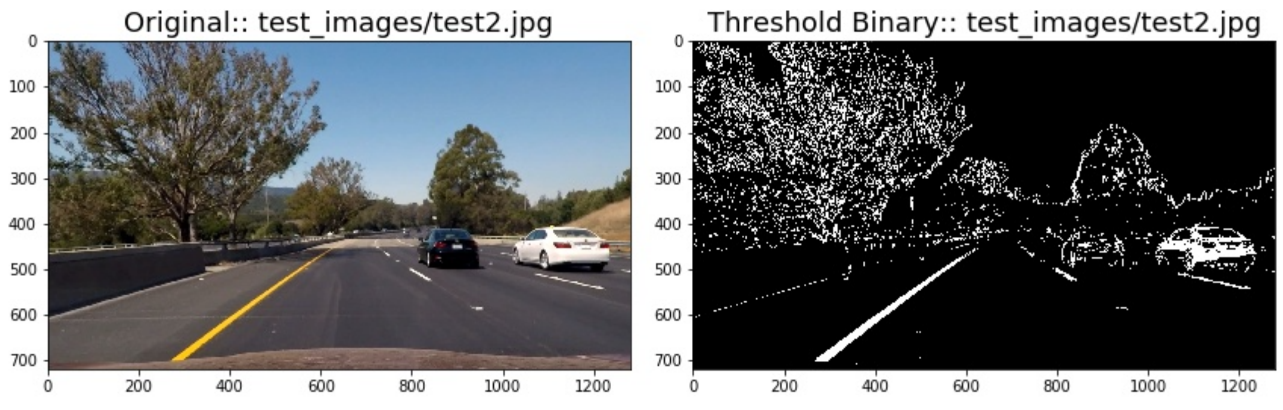
1. **Provide an example of a distortion-corrected image.**

There are two main steps to this process: use chessboard images to obtain image points and object points, and then use the OpenCV functions cv2.calibrateCamera() and cv2.undistort() to compute the calibration and undistortion.
See the meteor “undistort” in the Distortion Correction section. Here is the result:


Original:: test_images/test1.jpg
Undistorted:: test_images/test1.jpg

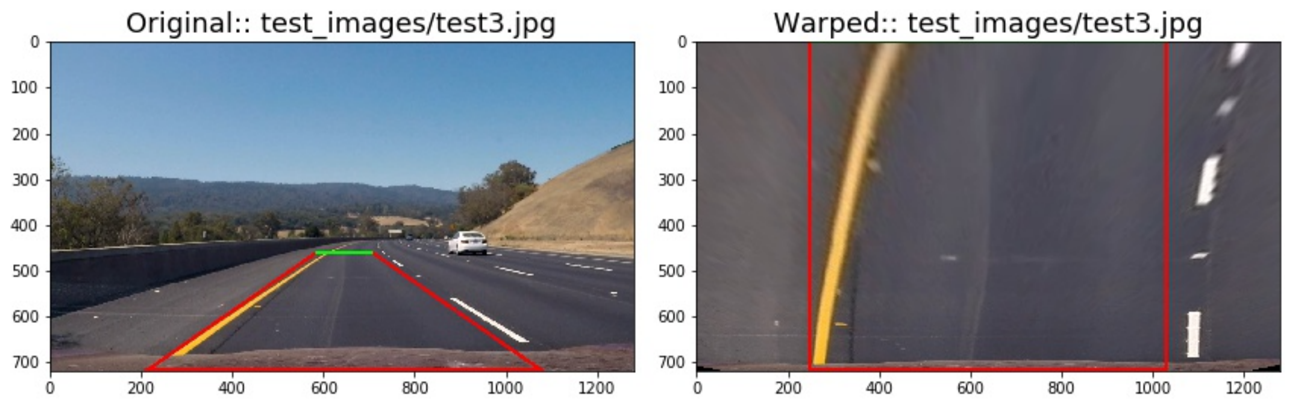
2. **Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.**

We know ahead of time that lane lines are mainly vertical lines. In order to detect steep edges in a smarty way we use the Sobel Operator. Using Sobel we can take the gradient in x or y and set thresholds to identify pixels within a certain gradient range. If you play around with the thresholds a bit, you'll find the x-gradient does a cleaner job of picking up the lane lines, but you can see the lines in the y-gradient as well. The magnitude of the gradient is just the square root of the squares of the individual x and y gradients. For a gradient in both the x and y directions, the magnitude is the square root of the sum of the squares.
It's also worth considering the size of the region in the image over which you'll be taking the gradient. Taking the gradient over larger regions can smooth over noisy intensity fluctuations on small scales. The default Sobel kernel size is 3.
In the case of lane lines, we're interested only in edges of a particular orientation. So we will explore the direction, or orientation, of the gradient. The direction of the gradient is simply the inverse tangent (arctangent) of the y gradient divided by the x gradient: arctan(sobely/sobelx).
In Gradient and color transform section I use sobel filter in both x and y direction to get gradient change in both axes to generate binary threshhold image.
In order to get color transformed binary threshold image I use color space HLS. Then, I used a combination of color and gradient thresholds to generate a binary image.
Here's an example of my output for this step:

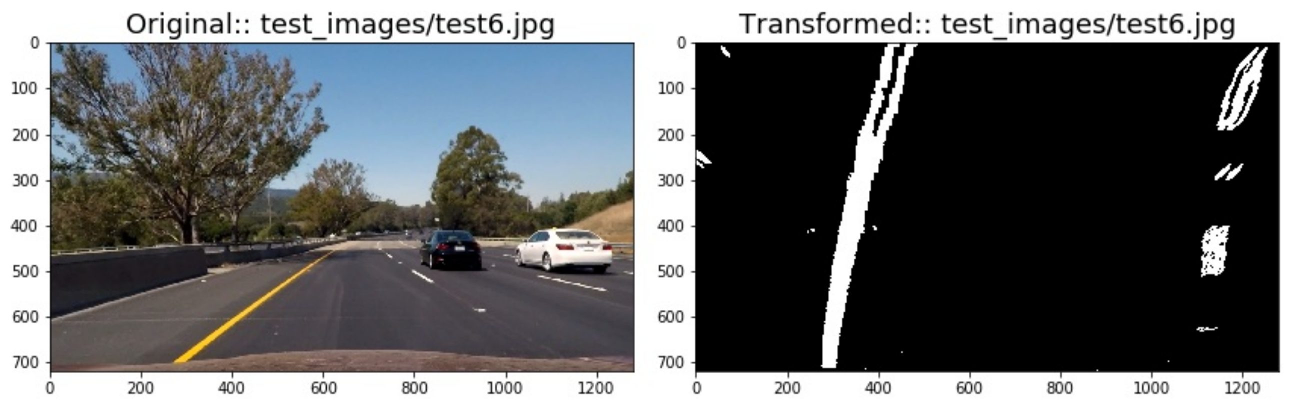

Original:: test_images/test2.jpg
Threshold Binary:: test_images/test2.jpg

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

A perspective transform maps the points in a given image to different, desired, image points with a new perspective.
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.


Original:: test_images/test3.jpg
Warped:: test_images/test3.jpg

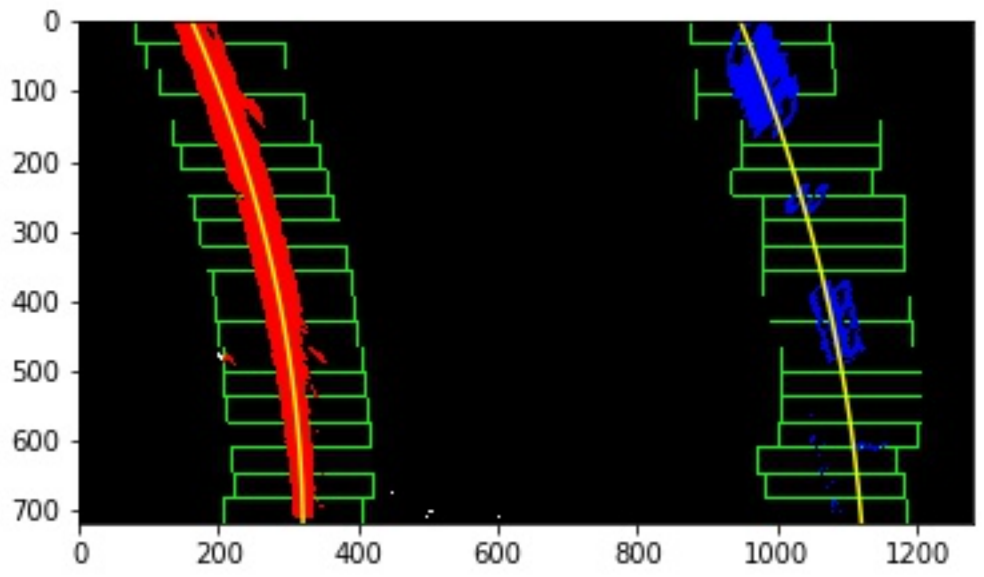
Here is the result of the perspective transform against combined binary:


Original:: test_images/test6.jpg
Transformed:: test_images/test6.jpg

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

After applying calibration, thresholding, and a perspective transform to a road image, we should have a binary image where the lane lines stand out clearly. However, we still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.
Plotting a histogram of where the binary activations occur across the image is one potential solution for this. With this histogram we are adding up the pixel values along each column in the image. In our thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. We can use that as a starting point for where to search for the lines. From that point, we can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

See “Lane line pixel detection and polynomial fitting” section in the code. Here is the result:



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 order to located the lane line pixels, we used their x and y pixel positions to fit a second order polynomial curve:

$f(y) = Ay^2 + By + C$

We are fitting fo f(y), rather than f(x) because the lane lines in the warped image are near vertical and may have the same x value for more than one y value.
The radius of curvature at any point x of the function x = f(y) is given as follows:

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

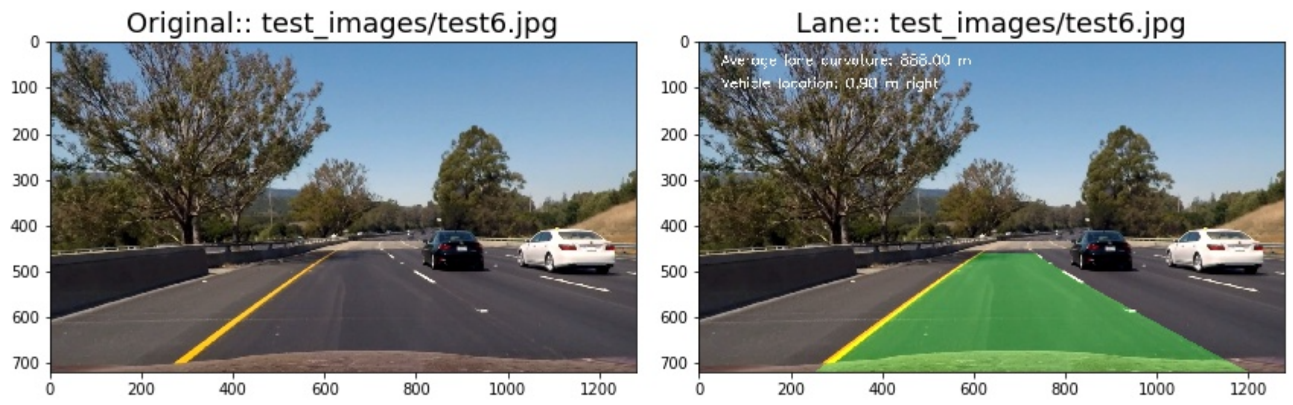
Where:

$f'(y) = dx/dy = 2Ay + B$
$f''(y) = d^2x/dy^2 = A$

See method “radius_curvature” in the “Radius of curvature” section in the code.

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

See method “show_curvatures” in the “Radius of curvature” section in the code. Here is the result:


Original:: test_images/test6.jpg
Lane:: test_images/test6.jpg
Average lane curvature: 888.00 m
Vehicle location: 0.90 m right
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

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 my video result:
test_videos_output/challenge_video.mp4

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?

The lanes lines in the challenge and harder challenge videos were extremely difficult to detect. They change in color, shape and direction. Moreover he challenge video has a section where the car goes underneath a tunnel and no lanes are detected. IMay be it is necessary to experiment on values of R, G channels and L channel thresholds.

The averaging of lane works well to smoothen the polynomial output. Harder challenge also poses a problem with very steep curves too. May be we need to fit higher polynomial to these steep curves.

Resources

https://www.intmath.com/applications-differentiation/8-radius-curvature.php