Advanced Lane Finding Project

Camera Calibration

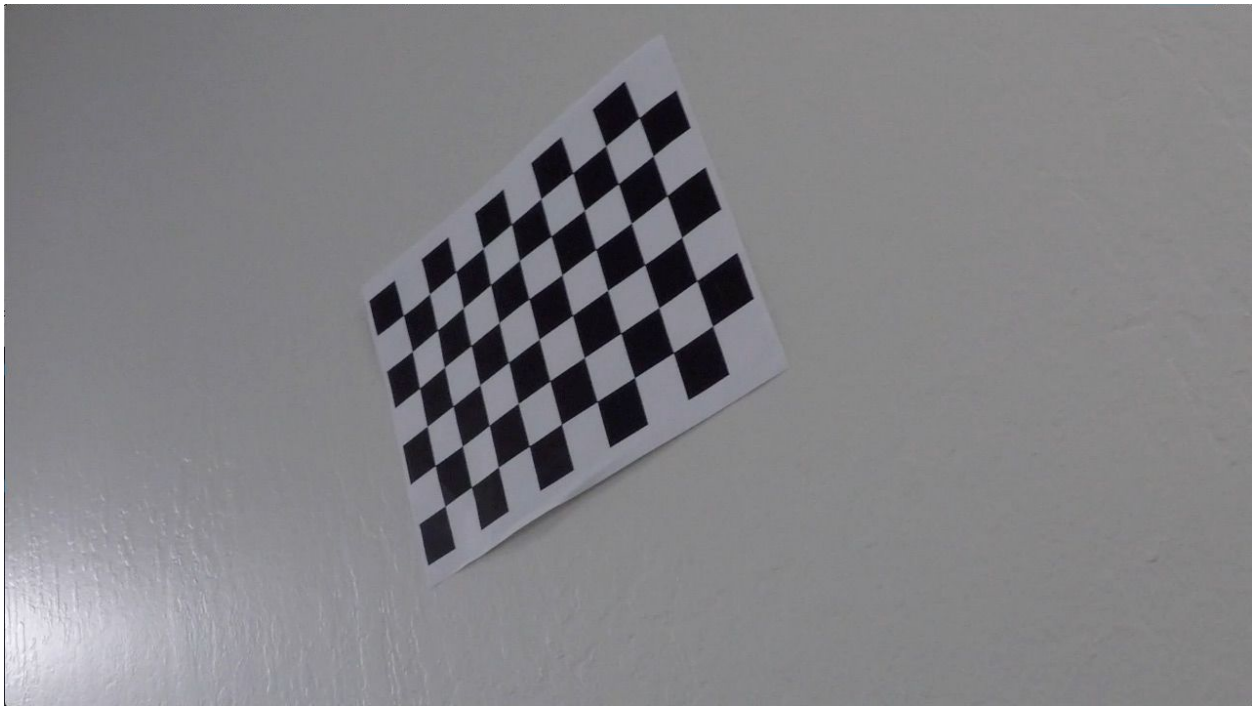
Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in “source/calibrate camera.py”

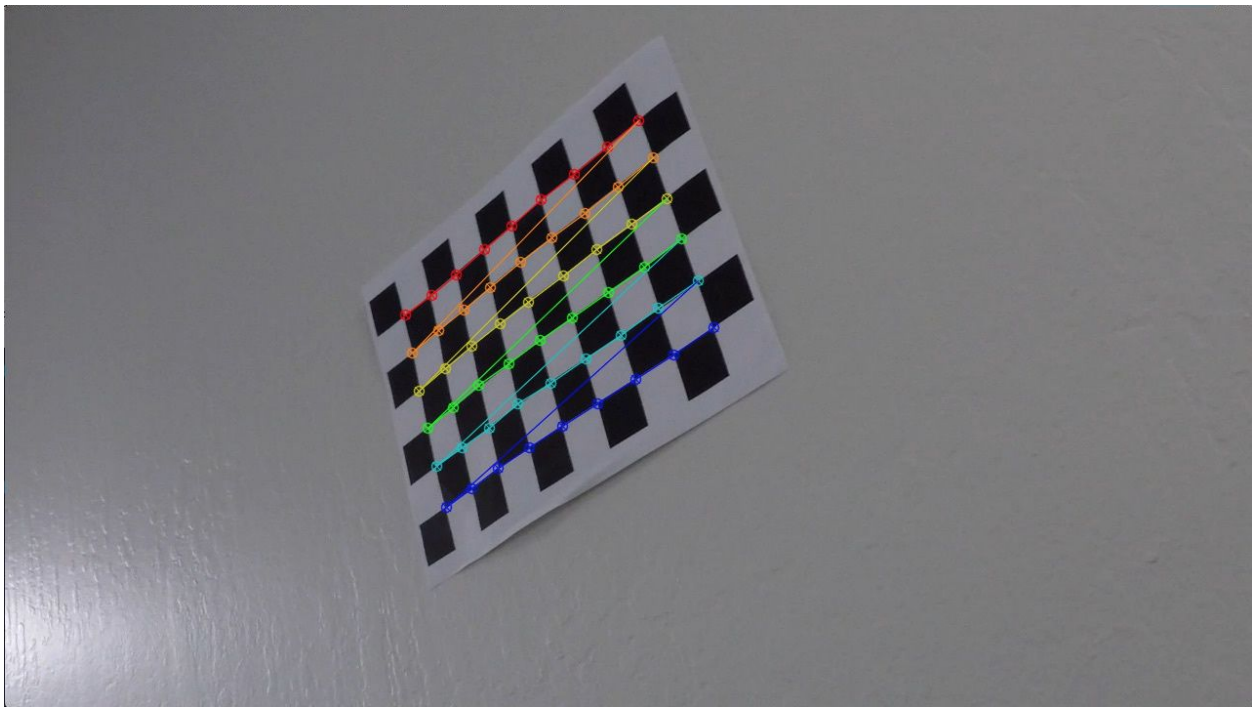
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, object points is just a replicated array of coordinates, and object points will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. Image points 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 object points and Image points 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:

Original image:



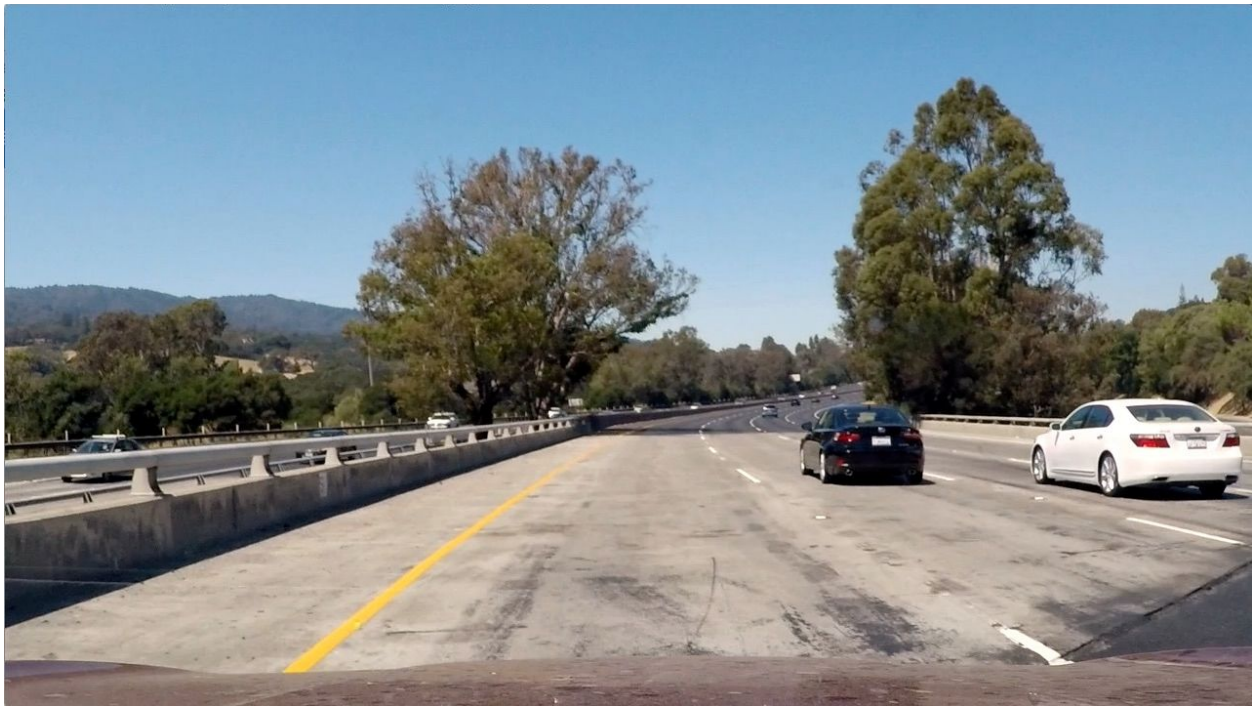
With camera calibration:



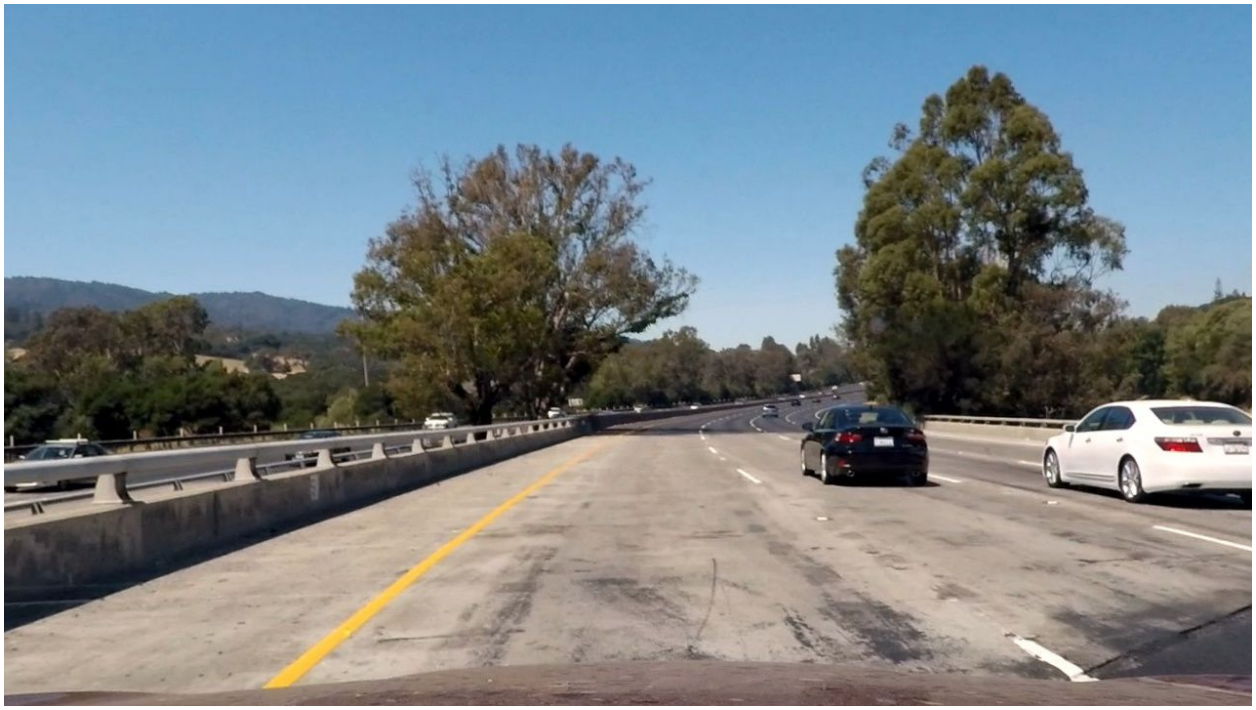
Pipeline (single images)

Provide an example of a distortion-corrected image.

Original image:



With distortion correction:



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.

I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at “source/thresholds.py”). Here's an example of my output for this step:

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

The code for my perspective transform includes a function called transform(), which appears in source/transformer.py. The warper() function takes as inputs an image, as well as image processed by thresholds. I chose the hard code the source and destination points in the following manner:

```
source = numpy.float32([[image_size[0] * (0.5 - middle_width / 2), image_size[1] * height_percentage],
                        [image_size[0] * (0.5 + middle_width / 2), image_size[1] * height_percentage],
                        [image_size[0] * (0.5 + bottom_width / 2), image_size[1] * bottom_trim],
                        [image_size[0] * (0.5 - bottom_width / 2), image_size[1] * bottom_trim]])
destination = numpy.float32([[offset, 0],
                             [image_size[0] - offset, 0],
                             [image_size[0] - offset, image_size[1]],
                             [offset, image_size[1]]])
```

This resulted in the following source and destination points:

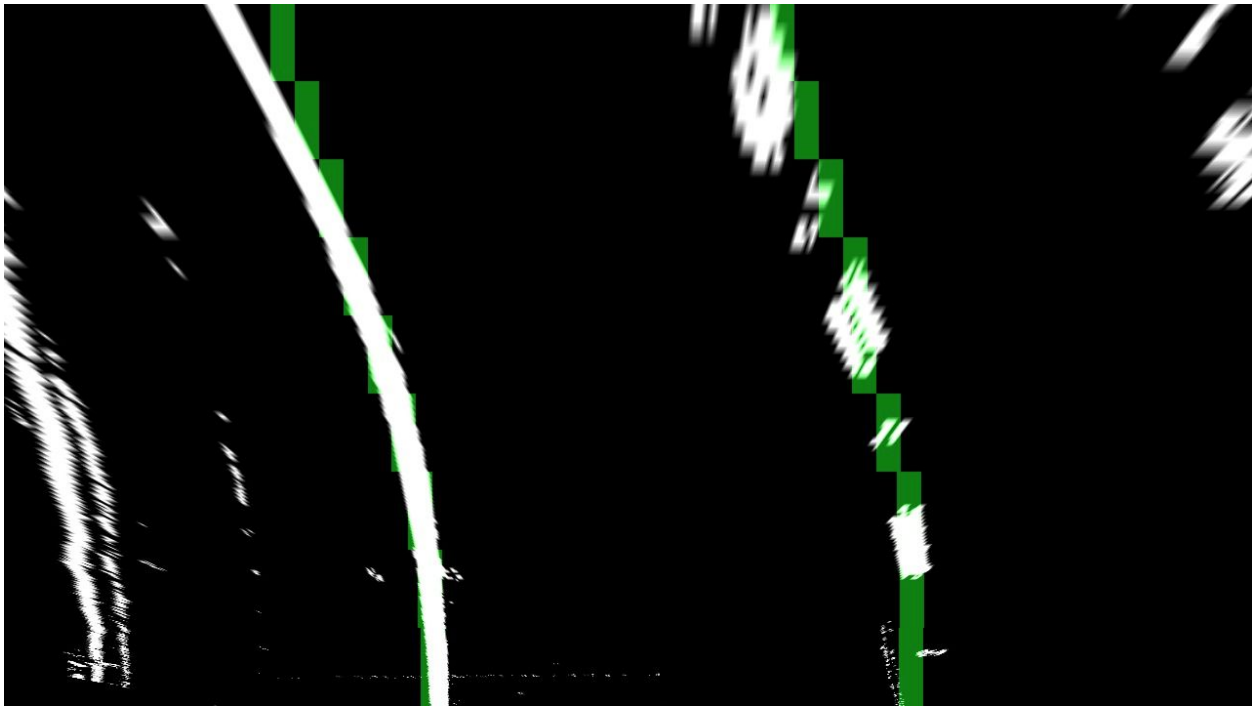
source
[[588.79998779 446.3999939]
 [691.20001221 446.3999939]
 [1126.40002441 673.20001221]
 [153.6000061   673.20001221]]

destination
[[320.   0.]
 [960.   0.]
 [960. 720.]
 [320. 720.]]

I verified that my perspective transform was working as expected by verifying that the lines appear parallel in the warped image:

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

The calculations for finding the lane line pixels and related logic are in source/tracker.py. I used numpy convolve to find the areas with the most amount of pixels, and then drew boxes at where the window centroids are. Here's the result for above image:

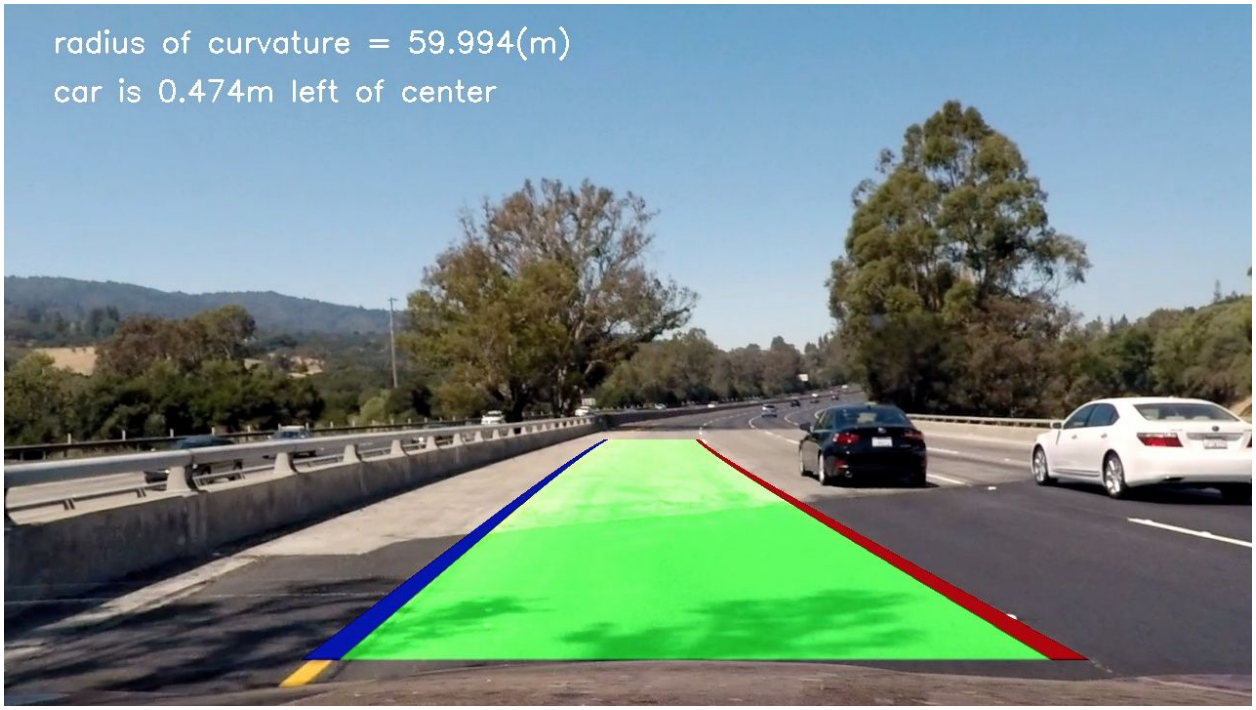


Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in source/curvature.py. I take the processed image, warped image, identified pixels, and curve centers, and use numpy polyfit to fit a curve the those pixels. The radius of the curve is calculated at line 45 and they are printed onto the images.

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 source/curvature.py in the function `draw_curve()`. Here is an example of my result on a test image:



Pipeline (video)

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

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 biggest issues in the implementation of this project are when I draw windows and curves. The logic behind it is complex and it's difficult to implement it in a way that's easy to read and reason about without having to look up other related documents. The pipeline will fail when there are noises in the image, like when there's a black car driving by and the white lines reflect off the black car, causing the algorithm to thinking that's where the line is. To fix this problem we could classify all the objects that's not the road, and exclude any lines in those objects.