Camera Calibration

Camera calibration was accomplished using images from the raw camera feed of a 9x6 chessboard at different angles and distances. The original chessboard images were processed by finding the points where two black squares meet two white squares, identified as the *img_points*, using the OpenCV library. Next, the desired vertices were defined as the *obj_points*, and these vertices result in the squares being completely square, i.e. the vertices are equidistant from one another. With both sets of points defined, the OpenCV library was again used to provide the transform parameters, a camera matrix parameter, *mtx,* and a distortion parameter, *dist*. These parameters can be then used to undistort future images.

The camera calibration is carried out in the `get_camera_calibration_coefficients` function in my program. This function first checks to see if the camera calibration coefficients have previously been saved to a pickle file. In the case that no previous coefficients have been saved, the calibration is carried out and they are then saved to a pickle file for future use. Incidentally, this function also returns the warping parameter *M*, which is used to transform the perspective of a head-one image to a top-view perspective. Seen below is an example of the raw camera feed image of the chessboard and the resulting undistorted image.

Raw camera image

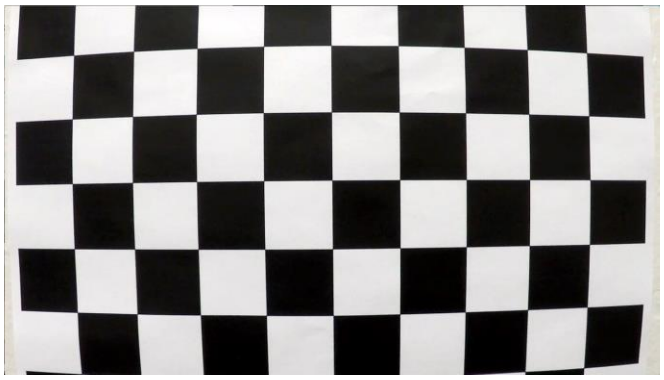
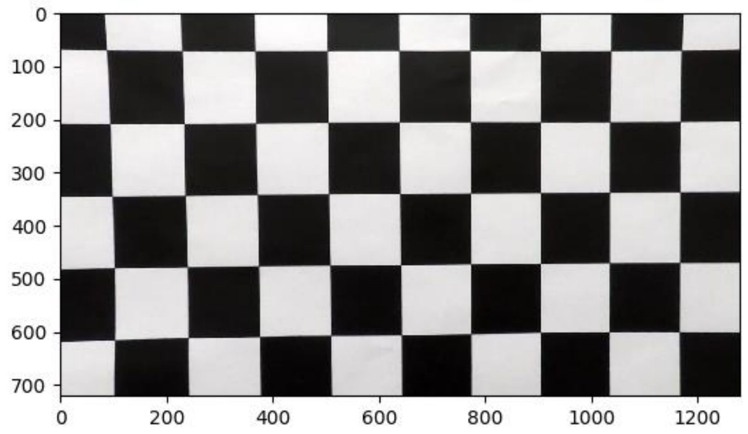


Fig1. Resulting undistorted camera image



Pipeline

The pipeline for processing the image is all carried out in the `process_image` function. The steps are outlined below:

1) Threshold the raw image

2) Undistort and warp the image to top-view perspective

3) Identify the pixels of the left and right lanes, and get the best polynomial fit for each lane

4) Check polynomial fits for usability

5) Find the radii of curvature and offset of vehicle from the center of the lane

6) If fits are usable, do a sanity check on the separation between the left and right lanes.

7) If data is good, add to averaging array. Averaging array contains n_samples with first-in-first-out structure

8) If fits are not usable, continue using the previous average values

9) Apply a mask of the lane identification over the original video feed

1) Threshold the raw image

The thresholding is completed in the `return_thresholded_image` function. To threshold the raw image, it is first converted from BGR space to HSV and HLS spaces. The hue, saturation, and value channels are then separated from the HSV and HLS images. For the saturation channel, a Sobel gradient is applied in the x-direction, and then a threshold is applied afterwards. For the hue and value channels, only a threshold is applied such that binary images for each channel are created. The final gray binary is generated by checking for any non-zero pixel from either of these three images, i.e. either the Sobel gradient in x-direction of the s channel is sufficient, or the hue channel is sufficient, or the value channel is sufficient:

```
gray_binary[(v_binary==1) | (sxbinary==1) | (h_binary==1)] = 1
```

For example, here is an original image and the resulting image when the threshold is applied:

Original image

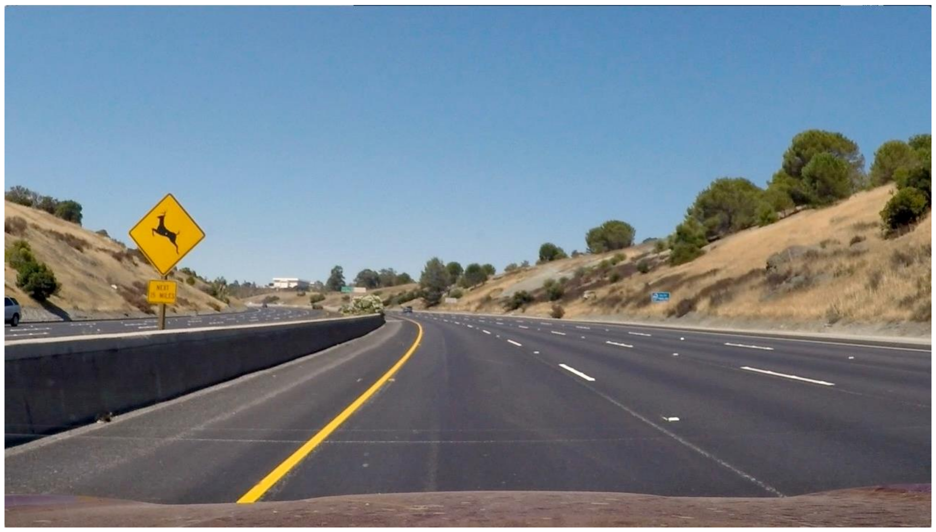
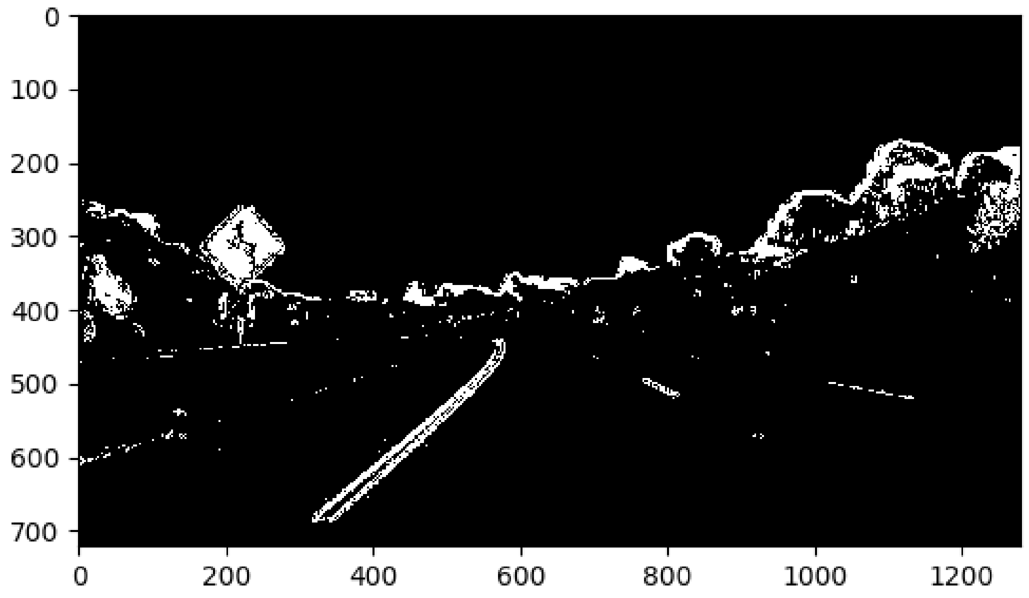


Fig 2. Resulting binary image after threshold is applied



2) Undistort and warp the image to top-view perspective

Using the camera calibration parameters and the warping parameter, the image is then undistorted and warped to a top-view perspective. The warping parameter was generated in the *get_camera_calibration_coefficients* function using a test image of a road with straight lanes as a reference. The source points in the image were defined as the points where the lanes met the bottom of the image, and two points along each lane further along the road. The destination points were defined as the set of four points that would pull the left and right lanes straight to make them parallel to one another.

The warping parameter *M* provides the transformation to move the source points to the destination points, such that the head-on perspective is transformed to a top-view perspective. Here is the code for this process:

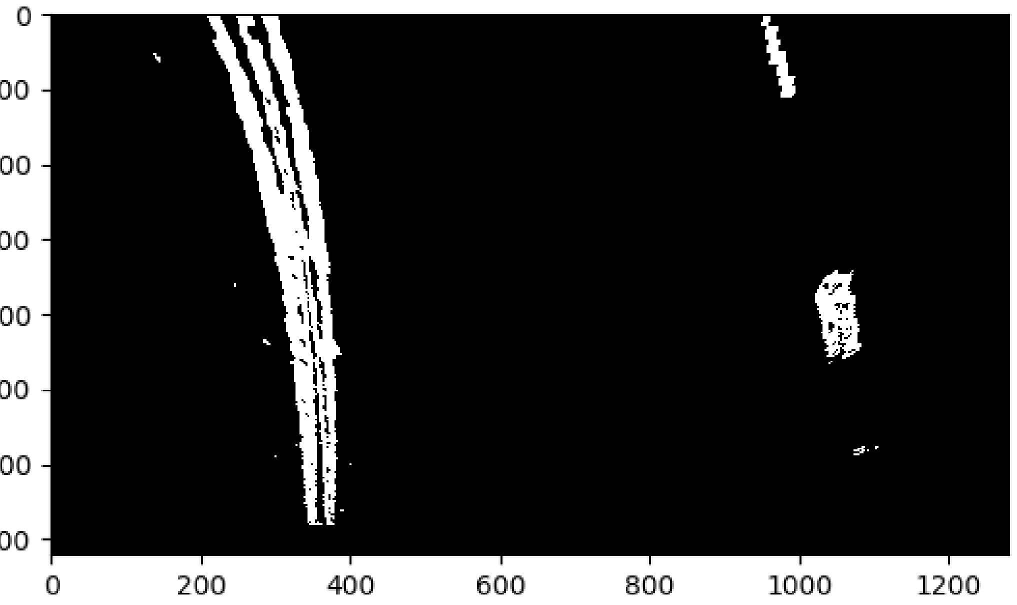
```
src = np.float32([[275,660],[540,484],[740,484],[1010,660]])

dst = np.float32([[275, 660],[275,275],[1010,275],[1010,660]])

M = cv2.getPerspectiveTransform(src, dst)
```

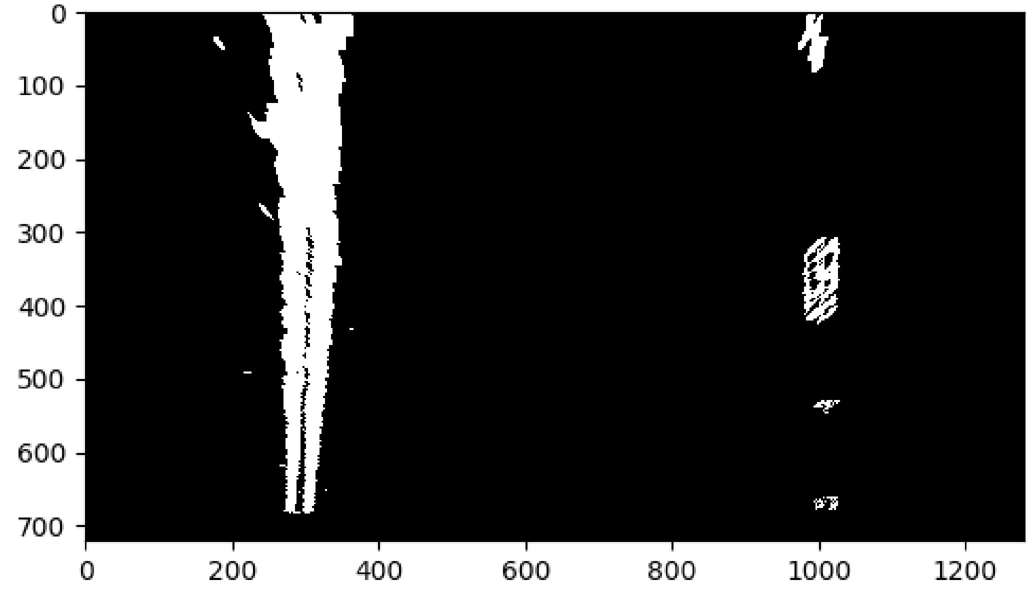
Below is the warped and undistorted image using the same image from the example above.

Fig3. Warped and undistorted image, curved lanes



And for verification, here is the warped and undistorted for an image with straight lanes:

Fig 4. Straight lane examlpe



3) Identify the pixels of the left and right lanes, and get the best polynomial fit for each lane

The left and right lanes are both identified and fit with a polynomial in the `return_polyfit` function.

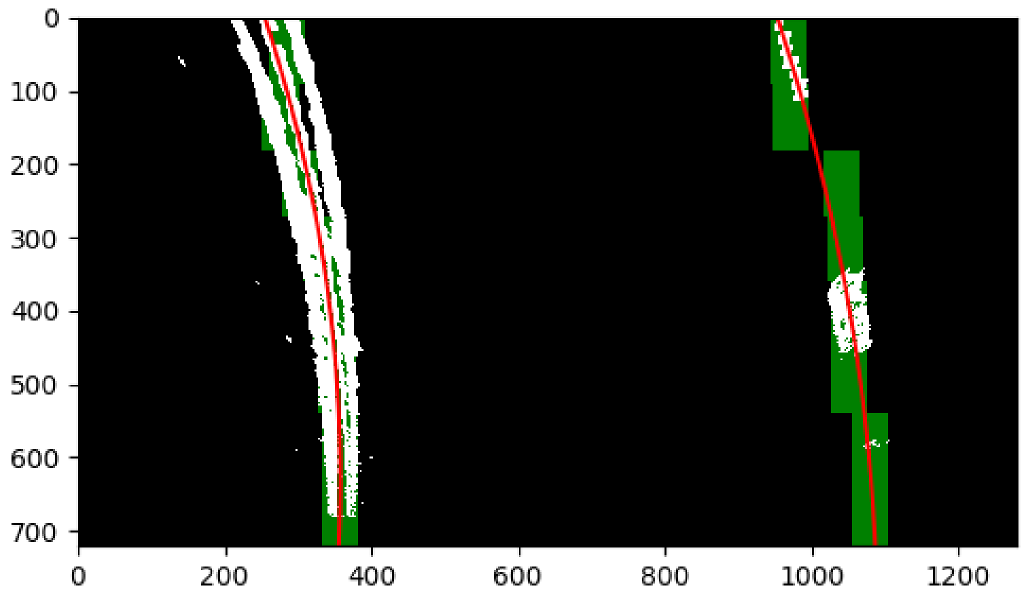
The base of the left and right lanes are identified by performing convolutions on the pixels in the lower left corner and lower right corner of the image. The array index belonging to the max value in each of these convolutions is identified as the base of the left lane and the base of the right lane. Once the base is identified, the image is split up into 8 horizontal slices and the same convolutions are performed for each slice. For efficiency, the subsequent convolutions are centered on the previously identified position of the left/right lane and extends to the left and right by a set margin instead of searching across the entire horizontal width of the image.

The `return_polyfit` function also implements a look-ahead filter, where the location of the left and right lanes at the top of the previous frame are used as the known locations of the left and right lanes in the next frame. Using the look-ahead filter, the convolution don't need to be performed across the entire width of the image but instead only need to be performed around the known locations of the left and right lanes.

To create a best polynomial fit, the pixels belonging to the left and right lanes need to be identified. To do this, the function identifies the pixels within a small box that is centered on the center point for each lane in each of the 8 horizontal slices. Once the pixels for the left and right lanes are known, `np.polyfit` is used to return the coefficients for a 2nd degree polynomial.

Here is the resulting image the polynomial fit is applied:

Fig 5. Polynomial fit example



Next, the polynomial coefficients for each lane are returned to the `process_image` function. The points on the left and right lanes are generated across the entire vertical length of the image using the 2nd degree polynomial function, $f(y) = Ay^2 + By + C$.

4) Check polynomial fits for usability

The `return_polyfit` function checks to see if a good fit was generated by making sure that the left and right lanes actually contain pixels inside the rectangular margin around the fit lines. If no pixels are found for either fit, then `None` is returned for both sets of polynomial coefficients.

5) Find the radii of curvature and offset of vehicle from the center of the lane

The radii of curvature or the lanes is calculated between lines 302 and 306, and the offset of the vehicle's position from the center of the lane is calculated on line 269.

6) If fits are usable, do a sanity check on the separation between the left and right lanes.

Now that points for each lane have been generated, sanity checks can be performed to make sure the results are reasonable. A reliable result was defined as having both a reasonable separation between each lane, i.e. ~3.7 meters or between 675 and 800 pixels, and a curvature greater than 250 meters for both lanes.

7) If data is good, add to averaging array. Averaging array contains n_samples with first-in-first-out structure

When the results are satisfactory, they are added to an array of the previous 10 values, which are then averaged to generate the final left and right polynomial fits.

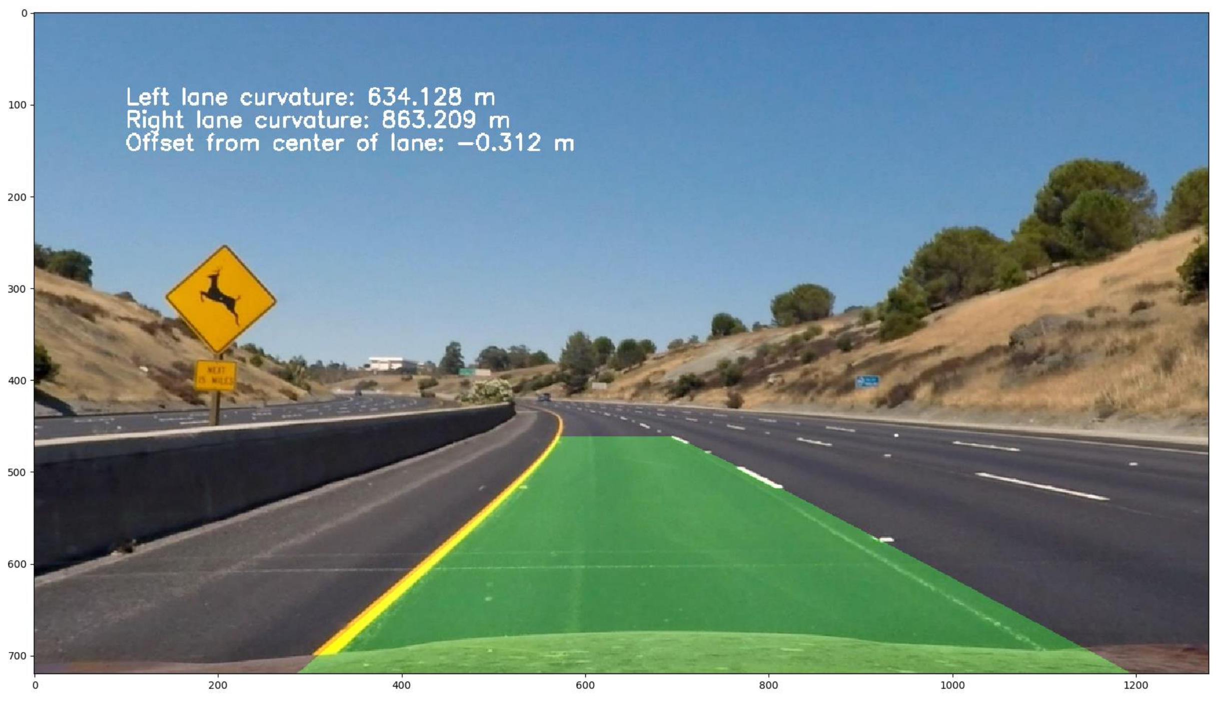
8) If fits are not usable, continue using the previous average values

If the results are not satisfactory, don't add them to the array of previous values, and instead use the averaged values to generate the left and right lane polynomial fits. If the results are not satisfactory for five frames in a row, then revert to doing a full window search to find the lane lines.

9) Apply a mask of the lane identification over the original video feed

A mask that outlines the identified lane is superimposed on the original video feed:

Fig 6. Final result


Left lane curvature: 634.128 m
Right lane curvature: 863.209 m
Offset from center of lane: −0.312 m
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Results

The final processed video 'project_video_annotated.mp4' can be found in the 'annotated_videos' folder, included in the zipped submission folder. The images above demonstrating each step in the `process_image` pipeline can be found in the 'output_images' folder.

Discussion

Problems / Issues

Choosing the correct thresholding parameters to create the binary image was fairly challenging. After finding a set of parameters that worked on one image, it was usually the case that the thresholding would fail on an image that had different lighting conditions or different lane marker colors. Choosing the most robust thresholding that applied to all the test images took a good amount of trial and error. Beyond that, even though the test image binaries looked effective, it turned out that these threshold values did not hold up in the more challenging areas of the project video and the parameters had to be tuned again.

Increasing the robustness of the image processing algorithm took a few quality gates. First, there was the possibility that the polynomial fit completely failed for one lane or the other, and in this case the numpy polyfit function gives a fatal error. This had to be correctly handled so that the process could be continued. Next, even if the fit was carried out, it had to be checked that the values were sensible and could be used. Finally, if the fit performance was poor for several frames in a row the algorithm had to divert back to a broader but less efficient search for lane markers. I wanted to make sure that even when the conditions were adverse and the fit completely failed, that the process continued in a safe and reasonable fashion.

Future Improvements

1) Handle polynomial fit failures for each lane separately

In the current implementation, when the polynomial fit fails for either lane then I decide that the entire fit has failed and the data should not be used. The fallback in this case is to use the average values from previous frames to create the fit lines, which uses the assumption that the lane markers in the next five frames are similar to the lane markers in the previous ten frames. In practice, this is effective for roads where the thresholding is successful, but in reality this can become a safety issue in areas where the thresholding is less successful.

To increase robustness, I would like to consider the quality of the fit for each lane separately. If the left lane fit was successful but the right lane fit was poor, then I would assume the right lane was shaped like the left lane. In most cases we expect both lanes to be shaped similarly. For lane-keeping applications it is better to base your positioning off of the lane that has been more confidently identified instead of just assuming the road ahead is similar to the road behind you.

2) Increase thresholding performance and implement region masking

The thresholding procedure is unsuccessful on the challenge videos. I’m sure the thresholding can be improved to perform better in difficult situations, and region masking will be effective at eliminating falsely detecting line shapes in the middle of lanes.