Advanced Lane Finding

Advanced lane finding project goals:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use colour 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 centre.
- 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

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 the 2nd and 3rd code cell of the Jupiter Notebook located in folder.

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 and saving the result as a pickled file. I applied this distortion correction to the test image using the distortion_correction() function which contains cv2.undistort() function and obtained this result:

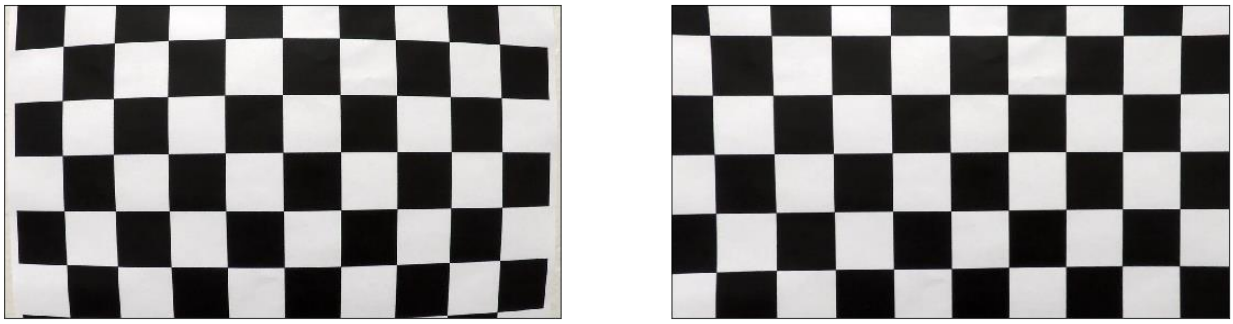

Figure 1. Original image on the left and corrected image on the right

Pipeline (test images)

Provide an example of a distortion-corrected image.

A distortion corrected image with the same calibration matrix can be seen here:

Figure 2. Original image on the left and corrected image on the right

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 to create thresholded binary image can be found under the “Use color transforms, gradients, etc, to create a threshold binary image”.

There are functions for absolute sobel thresholding, magnitude of the gradient thresholding, direction of the gradient thresholding and colour spaces combinations for thresholding. The get the satisfactory results, I went through all of them one by one and played around the parameters to get a feel in what range they should be and what features each of them brings out.

Finally, I used saturation and sobel x binary thresholded images and combined them with “or” logic to get a required result. The result can be seen here:

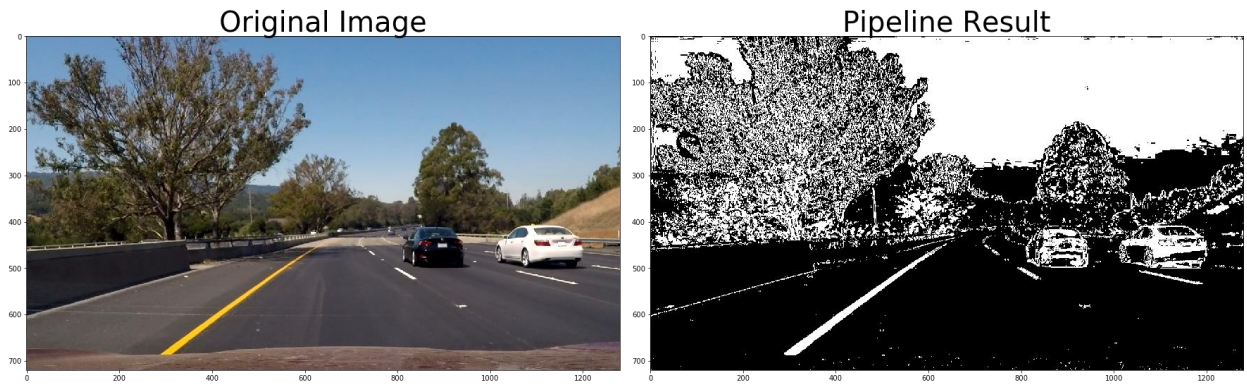

Figure 3. Original image on the left and binary image on the right

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

The perspective transform logic can be found under the “Apply a perspective transform to rectify binary image ("birds-eye view").” section in the Jupyter notebook. I chose the source points from a test image “straight_lines1.jpg” and assumed that the road if flat. I took trapezoid points to contain the lane lines roughly up to middle of picture and then converted those points to a rectangle.

Source	**Destination**
588, 450	328, 0
692, 450	950, 0
1120, 719	950, 719
158, 719	328, 719

After I had found the points I used the *cv2.getPerspectiveTransform()* function to find the perspective transform matrix. I calculated the matrix and its inverse matrix as well to unwarp the image later on in the pipeline. The result of a warped image can be seen here:

Figure 4. Original image on the left and warped image on the right

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

The polynomial was fitted in the “Detect lane pixels and fit to find the lane boundary.” section. At first I will use fit_polynomial() and after I have the initial reading I am using search_around_poly() function to have more targeted search based on previous results. For search_around_poly I cut down the margin size in order to have better results as the lane lines are quite close to each other in frames one after another.

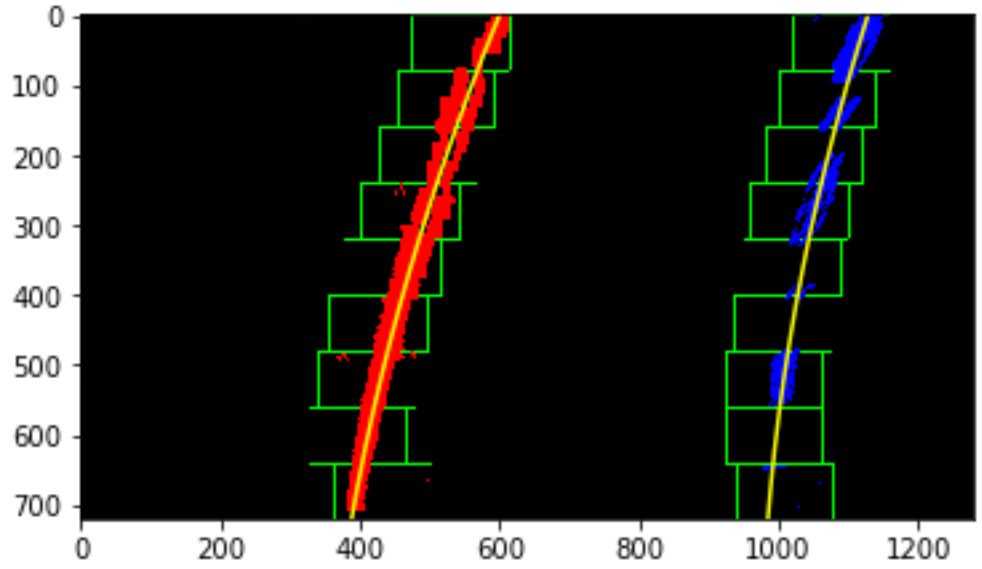

Figure 5. Image of a fitted polynomial

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.

The curvature radius and car’s position in regards to centre of the lane were calculated in the “Determine the curvature of the lane and vehicle position with respect to center.” section. The USA lane lines should be 3.7 m apart and from the above picture it can be seen that the lane lines are ~600 px apart in x direction. The dashed lines whould be 3 m long and the closest segment fits in one rectangle, and there are 9 rectangles in the image, so overall there should be 27 meters in the y direction of the image. This gives us calibration values for x 3.7 / 600 and for y we get 27 / 720. The result can be seen here:

Figure 6. Original image on the left and image with offset and radius image on the right

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

To get the fitted polynomial onto the original image I used the inverse matrix calculated earlier and filled all the pixels between the two polynomials green and blended it with the original image. The whole pipeline can be seen in "Picture pipeline coming together" section. The final result is here:

Figure 7. Original image on the left and image with offset and final image on the right

Pipeline (video)

The video can be found in the folder under the name "project_video_output.mp4"

Discussion

Right now, the pipeline works pretty well, although there is little fluctuation at some frames. In order to improve it further one could add checks to see how much the new reading differs from the previous reading and also compare the left and right polynomials and check if they are similar enough. Also the pipeline would work very well at night and to handle this situation an infrared camera might be used.