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.

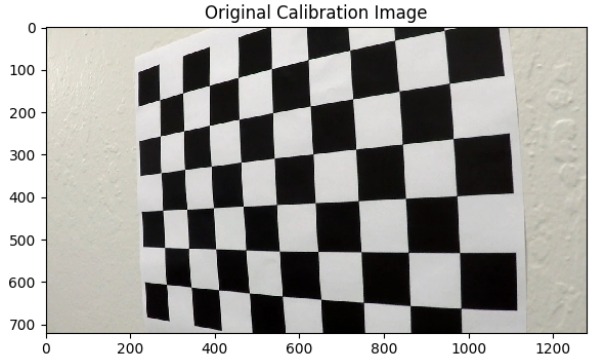
To meet the specifications and pass this project successfully, all Rubric Points have to be addressed.

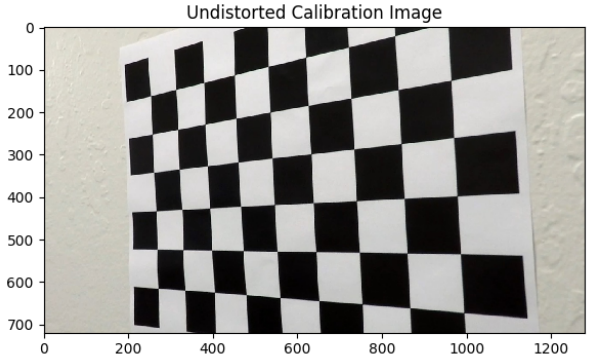
Writeup

Camera Calibration

The camera calibration is handled by a class called *CameraCalibration* which is defined in the advanced_lane_finder/calibration.py file.

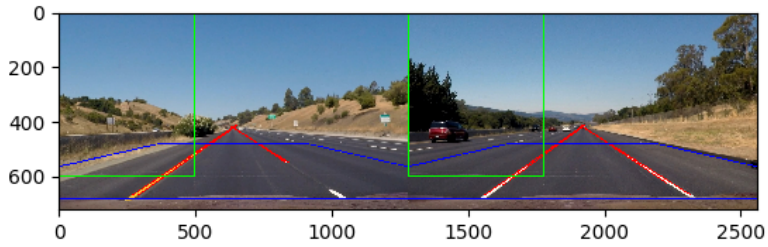
Calibration is similar done as during the lesson. First an array for the object points was generated with shape ((9*6), 3) to store the 3-dimensional object points and filled it with `np.mgrid` function from (0,0,0) to (8,5,0). To find the chessboard corners each calibration picture was loaded and the points retrieved by `cv2.findChessboardCorners()` function. More accurate positions were achieved by refining found points with `cv2.cornerSubPix` method before adding them to the *image_points* array. For each list of corners the previously generated *obj_pt_tempalate* list was added to a list. By passing object and image points to `cv2.calibrateCamera()` a calibration matrix `mtx` and distortion coefficients `dist` get computed. Those values can be applied with `cv2.undistort()` to any picture taken with this camera.



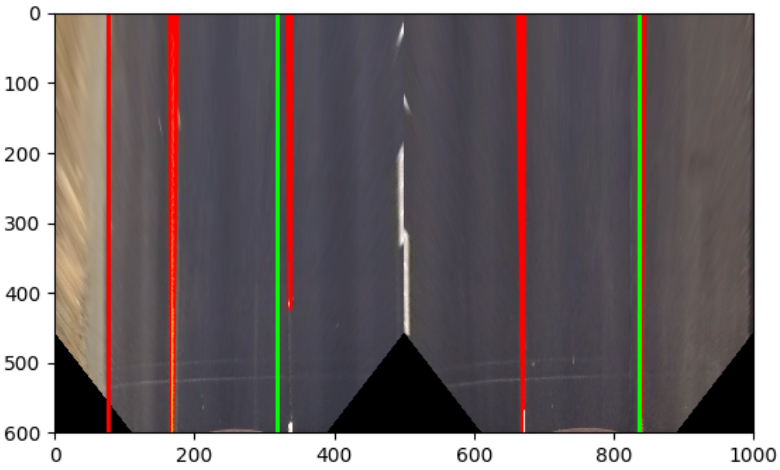


Perspective Transformation

Class *ImageTransformation*, defined in advanced_lane_finder/transformation.py, finds a proper perspective transformation matrix and calculates the pixel to meter factor. To determine good source points for the `cv2.getPerspectiveTransform()` method, the interception point of both lane lines is needed. Two frames with straight lane lines are loaded and `cv2.Canny()` and `cv2.HoughLinesP()` are applied to each one of them. Those lines are intercepted and the point where all lines cross is the reference point for the source trapezoid. The rest of the values are determined arbitrarily to fit the trapezoid nicely.
In the picture below are the trapezoid in blue, the destination rectangle in green and the detected lines for the interception point in red.



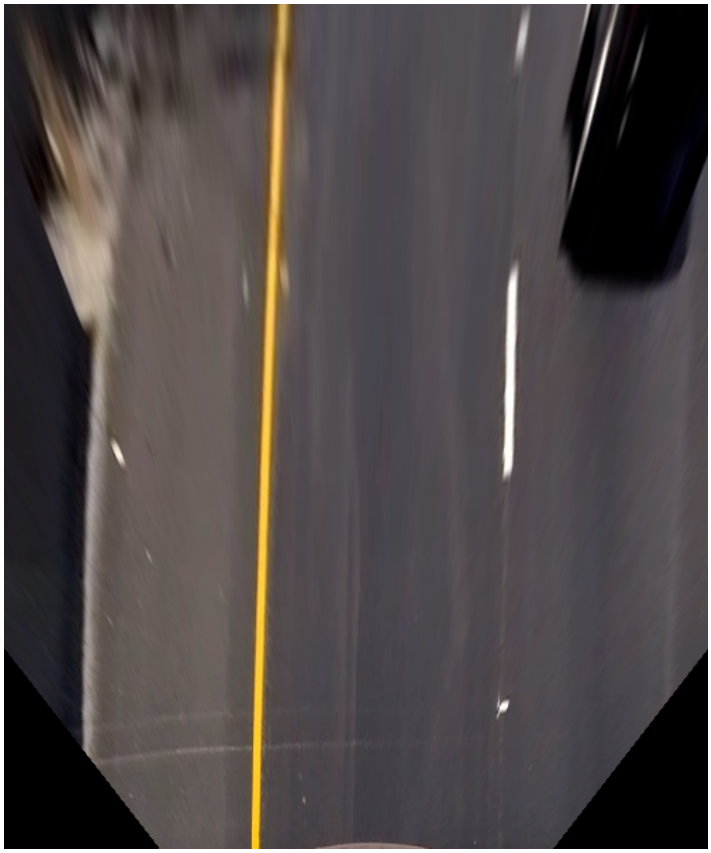
The *ImageTransformation* class also contains a function to calculate the suitable pixel to meter factor. First a picture with almost parallel and straight lane lines has to be undistorted and warped to the previous determined destination format. Then the image gets converted to HLS space and converted to a binary representation by threshold the H-channel on higher than 128. To remove lines of the curb and the middle of the road, 50 pixel on each side were set to zero. By applying the `cv2.moments()` method to the left and right hand side of them image separately the x values for the lines got calculated. The distance between the lane lines is the distance between these x values and therefore 3.7 meters. With the inverse of the multiplied transformation and calibration matrix, the factor for the y direction can be calculated.



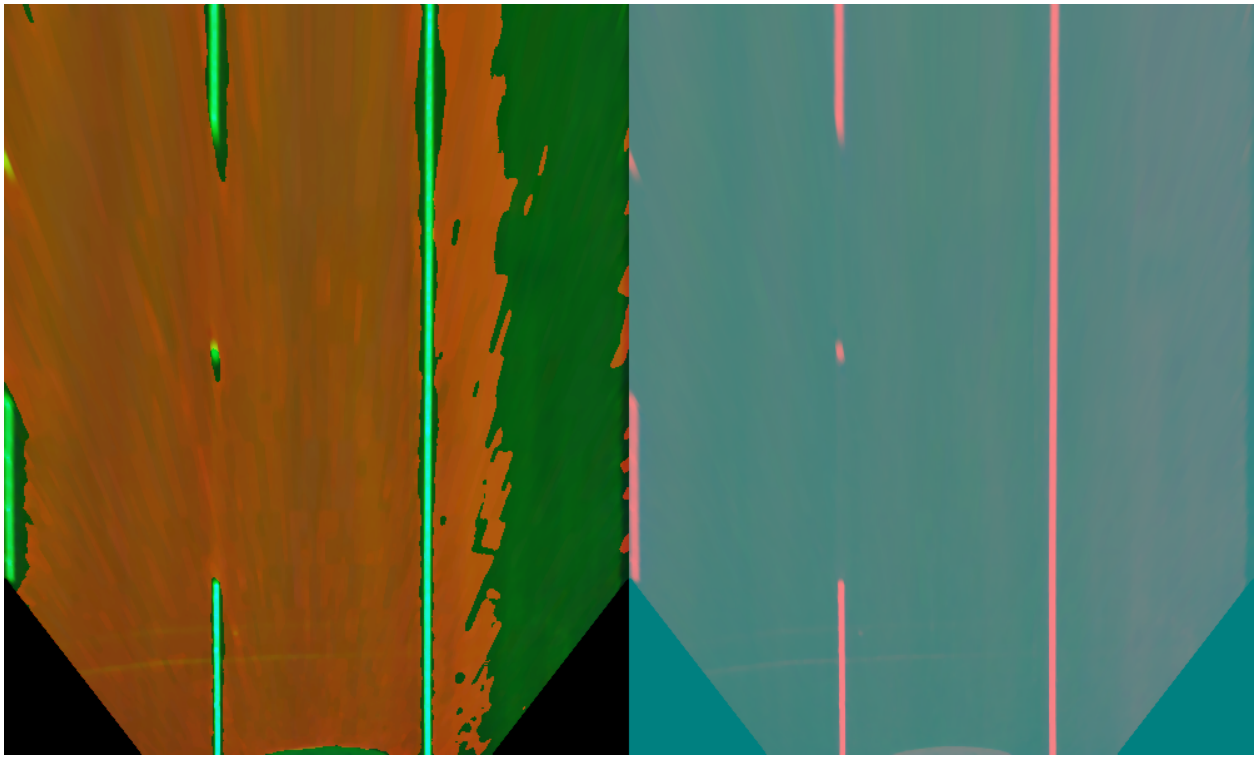
Pipeline

The pipeline for single images is the same as for video frames, since they technically are the same, and is defined in the *Frame* class advanced_lane_finder/frame.py in line 172. First the frame gets corrected with `cv2.undistort()` . This is not or only hardly visible in this image, since the errors are very small.

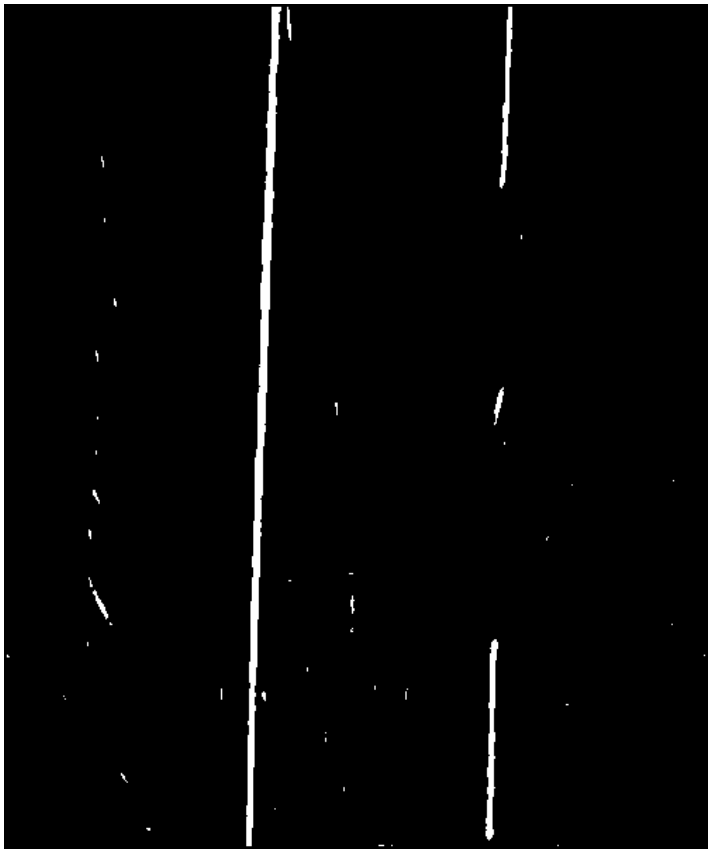
To reduce computational time and reduce false positives, the frame is transformed before further processing.



Converting the section to HLS and LAB colorspace makes it easier to extract yellow lines and lines in different lightning conditions.

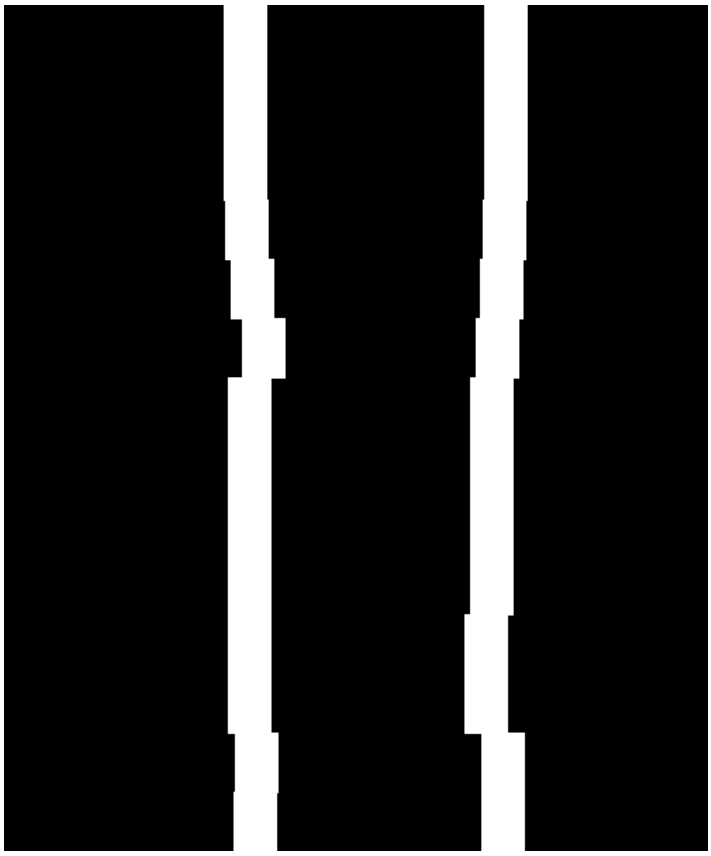


Further noise reduction was achieved by applying a light blur to each of the sections and removing cohere areas at the curb and towards the center of the road. A major improvement of the pipeline introduced the `cv2.morphologyEx()` method. This operation convolves a kernel over a binary image and, depending on the specified parameters, it erodes or dilates the pixels under the kernel. The result is an almost noise free binary section and easy to perform convolutional window search on.



Line Finding

The code for finding the lines and fitting the polynomial is in a separate class called *Line* defined in file advanced_lane_finder/line.py. This was separated to distinguish the lines easily by providing different offset values for the initial search window. The search algorithm in line 55 is an implementation of the convolution search and used to find the line for the first time. It runs over the binary image and convolves the pixel in a defined window. This produces a so called convolution signal which locates the point where the most pixels are build-up within this window. Marking these points with rectangles results in following graphic of two lane lines.



Function *fit* in line 108 uses this mask and fits a second order polynomial to the pixels when there are at least 150 pixels in y direction detected. For a steady highlighting the calculated coefficients are added to an array of old values which works as a history. The `np.mean` operation gets applied to this array to calculate the new values. To determine if the search and fit was successful, the deviation returned by `np.polyfit` , the count of y values and the radius have to be below or above a certain value, as well as the absolute position of the car should be below the center.

Merge the results

After processing and determine the points and polynomials of the lane lines, the area between them gets highlighted. In class *Frame* line 139, the *__highlight_lane()* method retrieves the points from each *Line* object and paints blue and red lines on the mask to mark the lane lines. Furthermore all points are concatenated to draw an area which is then colored green.

Radius and position are calculated in the *Line* class in line 172 and 177 respectively. They are called from the *fit* function to determine if its a good or bad line, or from the *___put_text()* method in class *Frame* to retrieve the values to put on the image which happens in line 150.

The section which were run through the pipeline gets then warped back to fit the original image size and in line 255 in class *Frame* added to the original, undistorted frame.

Video

As mentioned before, there is no special pipeline for videos.

The pipeline works well for the project video and is robust enough for the challenge video. In the harder challenge video the algorithm is not fast enough to keep up with the hard turns and has problems with missing lane lines.

Discussion

The implementation fails with very hard turns and one missing line. I tried to address this by mirroring the detected line but with rather moderate results, thus it requires some adjustments. Fast turns are a problem as well and could be addressed by an altering history length to speed up the adaption. Lastly there should be a check for analogousness of both lines to prevent them from drifting apart.