Advanced Lane Finding in videos with Python

Goals

The goals / steps of this project are the following:

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images. Apply a distortion correction to raw images.
2. Use color transforms, gradients, etc., to create a thresholded binary image.
   Apply a perspective transform to rectify binary image ("birds-eye view").
3. Detect lane pixels and fit to find the lane boundary.
4. Determine the curvature of the lane and vehicle position with respect to center.
5. Warp the detected lane boundaries back onto the original image.
6. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Calibration

I have reused the tutorial for the camera calibration.
My solution contains two methods: calibrate and undistort. Calibrate uses the cv2.findChessboardCorners method to identify the image and the object points which are returned after execution. The resulting object and image points are passed to the undistort method that calculates the resulting destination image.

An example of a distorted and an undistorted chess board and of one test image shown below:

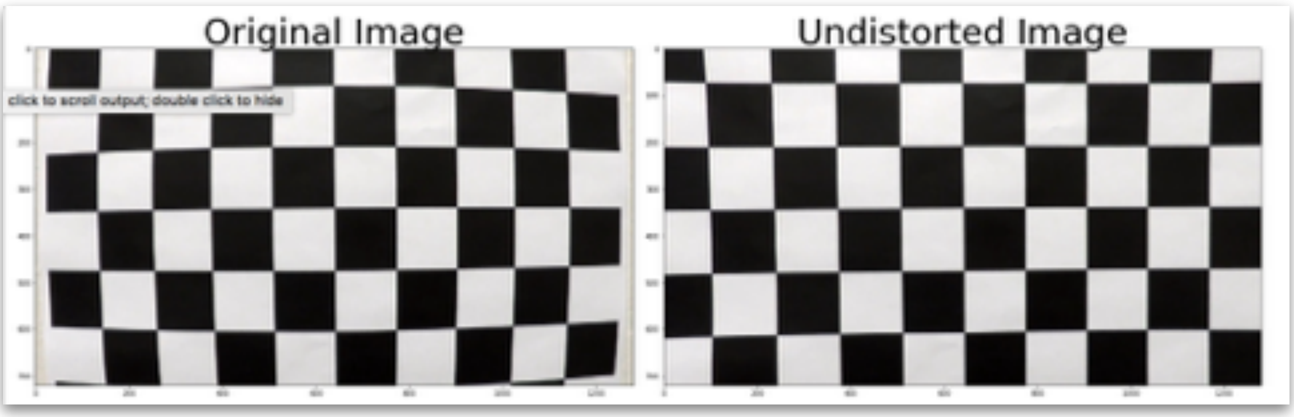

Figure 1: Original and undistorted chess board

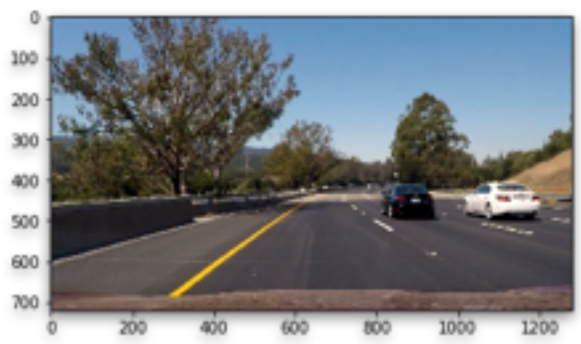

Figure 2: Original test image

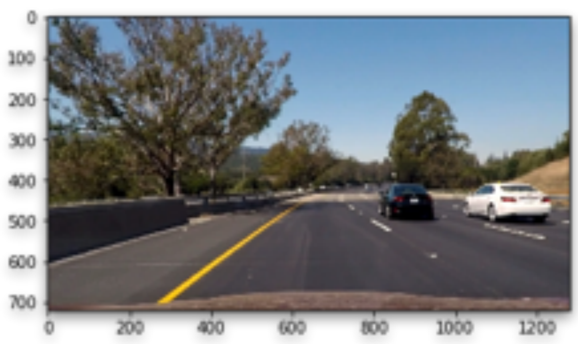

Figure 3: Undistorted test image

Note: figure 3 shows that the undistortion is working as the lower part of the image is reduced.

Preprocessing

After the successful calibration of the camera, I have followed up with the next major task: image preprocessing.
My approach was quite simple: trial and error of the methods provided in the tutorials:
1. gradient of x and y
2. magnitude
3. direction
4. exploration of color spaces

By combining the different methods, I have identified that the Saturation (in HLS colorspace) is best for the lane identification. Therefore, I have combined the x gradient and the value of the S channel. An example of the application is shown in figure 4. One can see that the lanes are displayed clearly.
The final method used for the creation of the binary image is: hls_binary(…) that returns a single channel image.

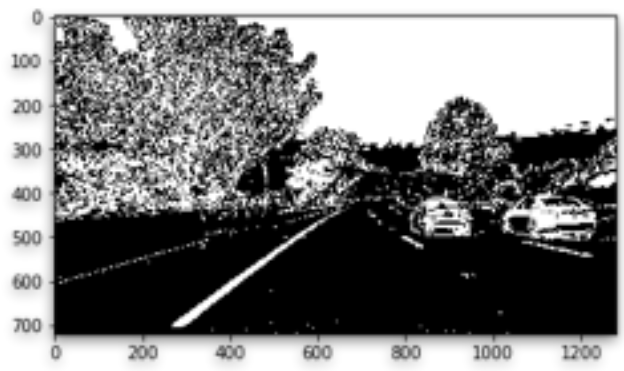

Figure 4: Applied sobel operator on the S channel of a test image

Perspective Transform

My perspectiveTransform(…) method defines the region on interest and returns the corresponding transformed image along with the transformation matrix.
An example of a transformed image before and after the preprocessing is shown in figure 6 and 7.

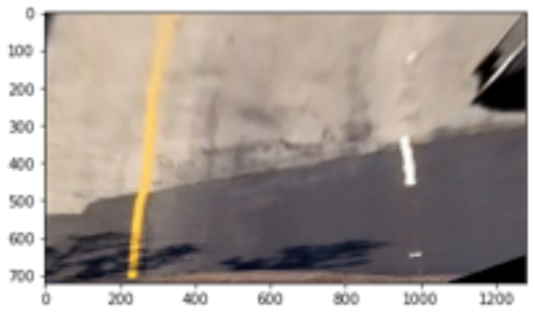
Figure 6: Perspective transformed (warped) image

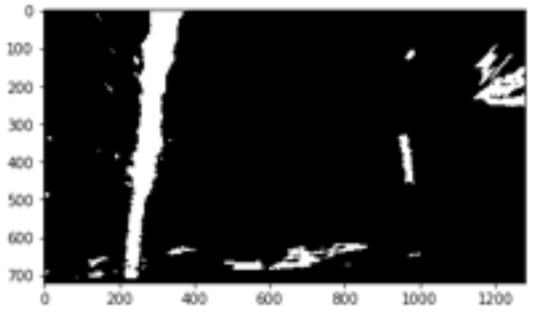
Figure 7: Binary warped image

For the source and destination points, I have reused the following selection. Illustration on a test image is shown in figure 8.

```
src=np.float32([[750, 470],
            [1200, 720],
            [230, 720],
            [550, 470],])

dst=np.float32([[1000, 0],
             [1000, 720],
             [200, 720],
             [200, 0],])
```

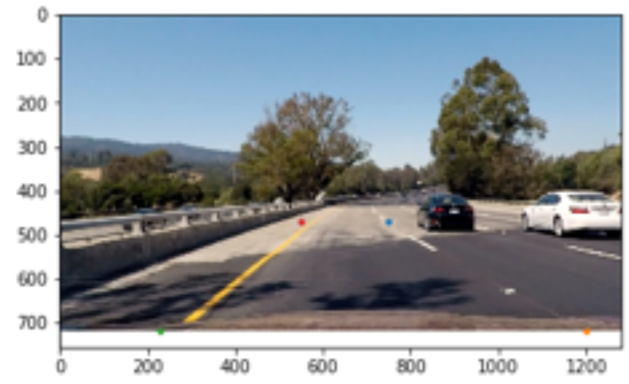
Figure 8: Region of interest

Finding Lanes

In order to identify the actual lanes, I have reused the code from the tutorial (sliding window) and applied the calculation to the warped binary image (example shown in figure 7). The corresponding method is findLines.
The result is shown in figure 8, where red lines were plotted onto the warped binary image.

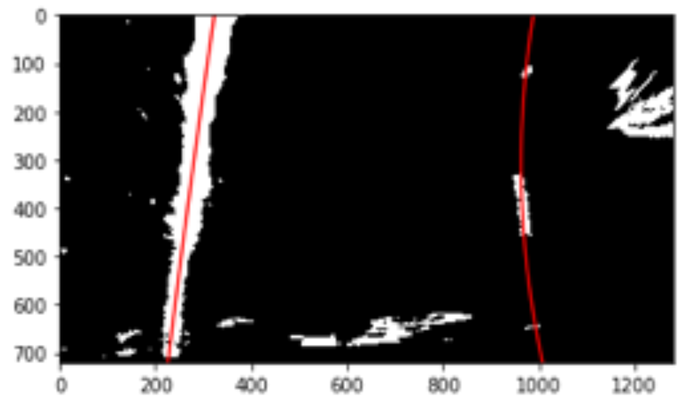

Figure 8: identified lines (based on 2nd degree polynomial)

Next, I have used transformed the lines onto the original image (as mentioned in the hints section). An example is shown in figure 9. In order to be able to cross check whether the perspective transform is working fine for every part of the video, I have displayed the birds eye view on the left upper corner as well as the curvature and the vehicle position estimation.

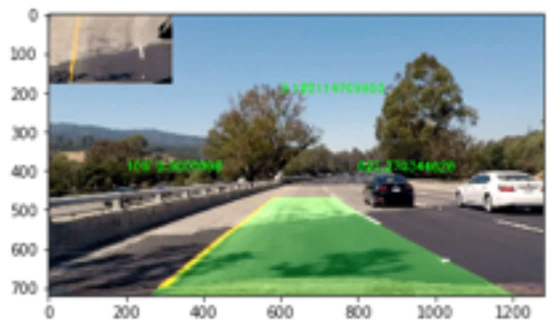

Figure 9: lines projected onto the original image.

Sanity Checks

Although most of the time the vehicle was able to identify the lane correctly (I assume that the selected preprocessing was a good choice) , in some cases failures appeared. One example is shown in figure 10, where the right line was identified on the left side.

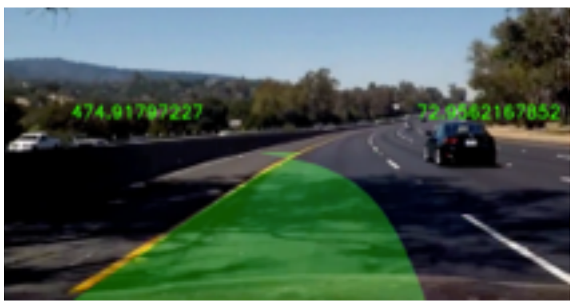

Figure 10: lines projected onto the original image.

In order to solve these issues, I have implemented 3 sanity checks.

1. Check if the current mean of the x values representing the line is not too different from the mean of the 100 x values
2. Check if the first derivative (representing the gradient) of the polynomial fit at position y=0, does not differ from the previous derivative at the same position. This derivative represents the slope of the polynomial line at the end of the warped image (y=0). Refer green markings in figure 11 for an illustration.

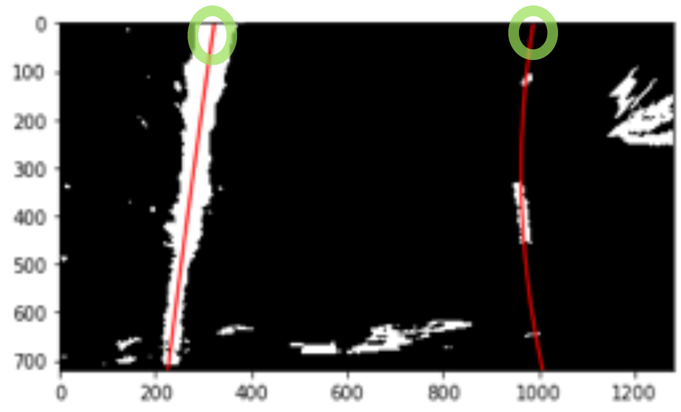

Figure 11: Gradient positions used for the 2nd sanity check

3. Check if the change in current curvature is does not exceed 1000

In case one of the checks failed, the previous lines have been used. I feel this is not the worst strategy as major changes in values are not expected between two images.

Processing Pipeline

My pipeline:
1. undistort image
2. perspective transform
3. find lines (includes sanity checkers)

The creation of the resulting video was straightforward as I was reusing the code from the first project.

Discussion

Although the lines are projected onto the video with no total failures for the project video, my implementation fails entirely for the challenging videos. The reason is most certainly my strategy on reusing previous lines in case one line does not pass a sanity check. Therefore, my solution is only suitable for videos with less dynamic environments because elevations, shadows and irregular lane markings will most likely lead to frozen projections.

Improvement: in order to react on rapid changes, my idea is to implement a smoothing technique that reuses the last n lanes for the calculation of a “substitute” lane in case a sanity check is not passed.
Another idea, as suggested within the project rubric is the reduction of the search space once a lane is found of course.