Project Writeup Advanced Lane Finding

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.

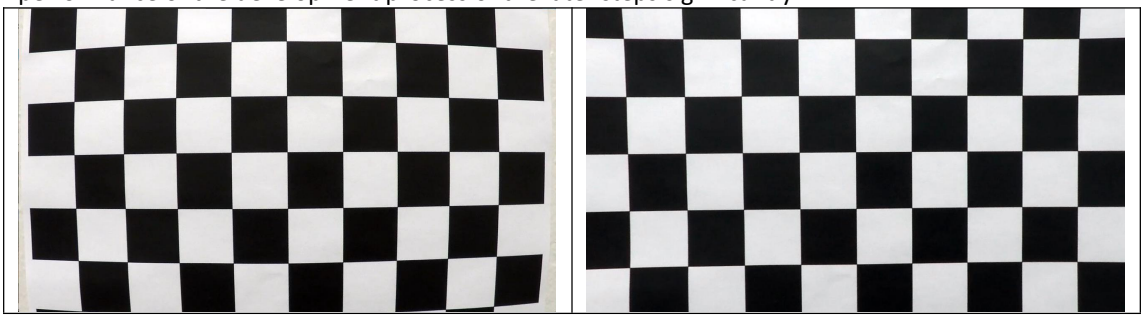
General approach / File structure

To maintain modularity and ease of programming, the code has been structured into different Python modules, each with its on sets of unit tests. These are:

test_calibrate.py	
test_distorter.py	
test_histogram.py	
test_perspective.py	
test_threshold.py	
test_videoprocess_ch1.py	
test_dap.py	
test_gradient.py	
test_lanefinder.py	
test_pipeline.py	
test_videoprocess.py	

Camera Calibration

Code for the calibration of the camera is in calibrate.py. It includes code that calls the openCV functions to find corners in a chessboard, and code to calculate the calibration based on openCV. Main code is *calibrateFromDir,* which processes files from a directory and calculates calibration based on that.
Everything is called from test_calibrate.py, which also uses pickle to save the resulting transformation matrix. This is done, because the calculation is quite slow and separating that step out increases performance of the development process of the later steps significantly.



Processing Pipeline

Distortion

The pipeline is implemented in pipeline.py and takes an img as a parameter. As a first step, the image is distorted. The effect is clearly visible on this image of a brigde:

Perspective Transformation

Next step is to „warp“ the image, i.e. transform it into a top down view. This is done in perspective.py . The perpsective transformation parameters are

```
DEFAULT_SRC = np.float32(
[[120, 720],
[550, 470],
[730, 470],
[1160, 720]])

DEFAULT_DST = np.float32(
[[200,720],
[200,0],
[1080,0],
[1080,720]])
```

For convenience/debugging, several of the images of the pipeline are drawn into the final video. This is a screen shot:

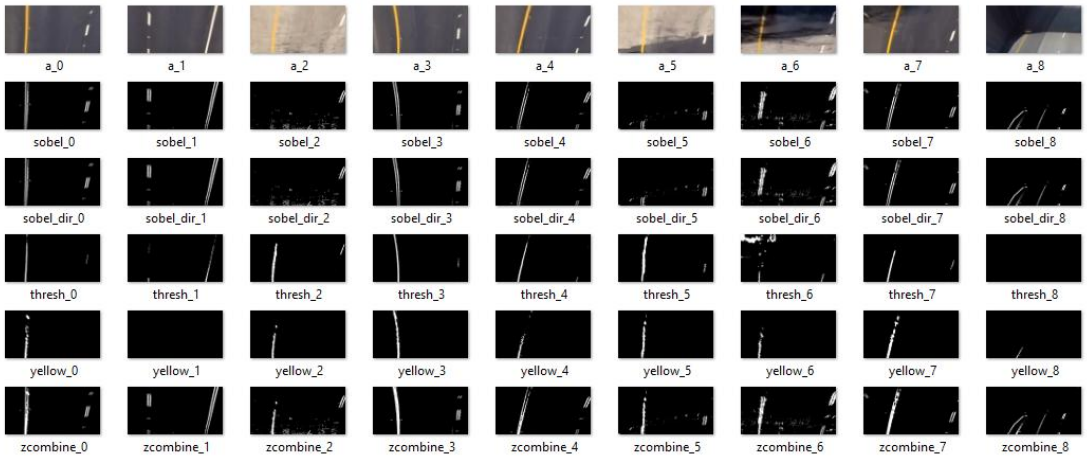
Top Down-View is on the top left.

Sobel Filtering

To find the lines, gradients are detected and calculated with a simple sobel filter in gradient.py. The image is first converted to gray for that. The result can be seen at the 2nd image on the top in the screenshot above.

Yellow Filtering

During testing, it turned out that in road segments with low contrast (light tarmac), the yellow line did not provide enough contrast to be recognized. The idea was to use additional information from another color space (HLS). However, for our case this turned out unsufficient. Doing some testing with several approaches, it turned out that, in addition to Sobel (2nd line in image below), filtering for high yellow values seemed promising (5th line) and that a combination (adding of information, shown in last line) seemd promising.



It turned out that this approach was successful. The result of the operation can be seen in the 2nd top image:

Lane Finding

Code for lanefinding is in lanefinder.py, which (as other modules) has been implemented in an object-oriented fashion. This encapsulates nicely the data from the previous run. Code for this has been basically taken from the Udacity course and does not contain any specific information. Since The code also provides information about the involved pixels, there are displayed in the 3rd image.

„Augmented Reality“

In the next step, the polygon information is used in lanefinder.py draw() to calculate the overlay image containing the lane boundaries. It is a simple drawing, which is then retransformed to the perspective of the original image and added with add_weighted.

Curvature and Distance from center of lane

Both curvature and distance from the center of lane are calculated in curvature.py. The code for the curvature is taken from the Udacity course and for the center of lane calculation, the points at the lower bottom of the image are calculated for left lane and right lane and the position is then simply:

```
caroff = (leftl+(rightl-leftl)/2-640)*xm_per_pix
```

All information is then drawn into the final image with openCV functions.

Final video

Video generation is done through test_videoprocess.py and test_videprocess_ch1.py (challenge video). Invocation is through unit tests, since it makes it easier to also always generate short subclips for problematic video sections (such as light tarmac, as discussed above).

Video file is in Lproject_video.mp4

Discussion

I also tried the code on the challenge video. Performance is worse here. Analysis of the problematic sections give the following result:

- Shadows of the lane wall (left) and repaired segments of tarmac (middle of line) are showing up in the sobel image similar to white and yellow lines. Solution could be to some additional image color analysis and filter out / map to black everything greyish.
- Videos contain darker sections (such as under the bridge). Color normalization comes to mind to make sure that lanes are as bright as possible and contrast is high enough.