SDCND Advanced Lane Finding

Matthias Schinacher matthias.schinacher@googlemail.com

2019-03-11

Contents

1 Intro

The project is a homework assignment for Udacity's **Self Driving Car Nano Degree**. The project is the second one called *Advanced Lane Finding*.

1.1 Implementation

I choose to implement the necessary code as a series of python scripts invoked from the command line, rather than implementing a notebook. I did use however the example `example.ipynb` as a starting point for the calibration.
I also used my versions of the solutions to the various quizzes in the course as a code base for my scripts.
The scripts implement each a specific part of the required task for the project. Those scripts that produce results used in later stages, use the standard python *pickle* module to save data as python objects.
The scripts have positional command-line parameters.

2 Calibration

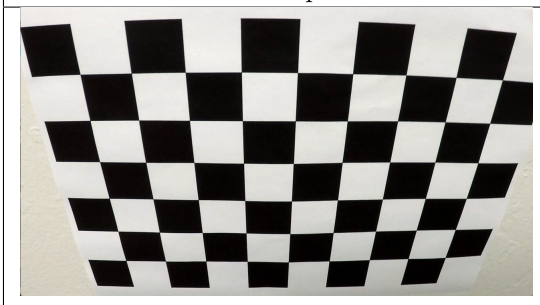
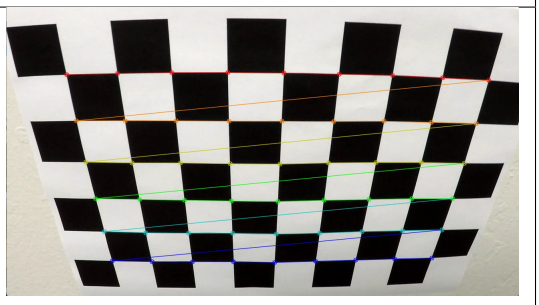
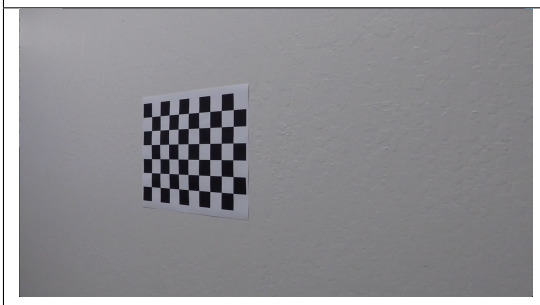
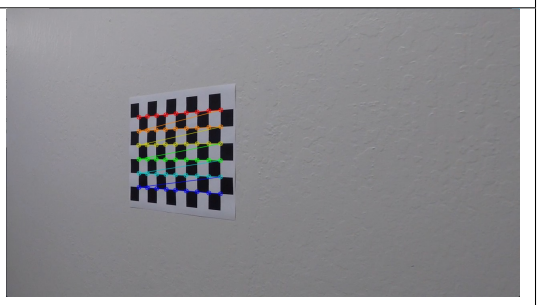
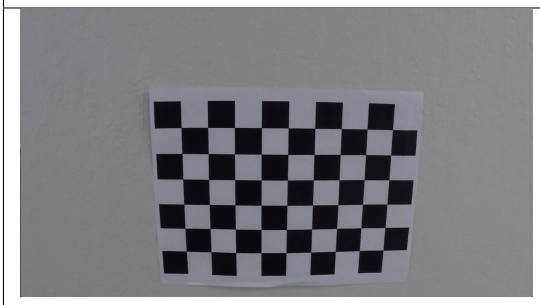
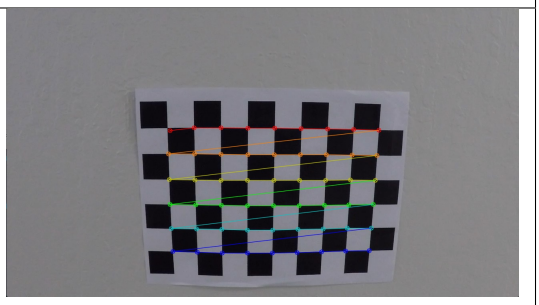
I calibrated the "camera" with the script textttcalibrate.py. It uses the `cv2` method `calibrateCamera(...)` to do the actual calibration and `findChessboardCorners(...)` to find the image points required within the given chessboard calibration pictures.
The calibration matrix etc. computed is written to a pickle file and the scripts allows to save and/or show modified calibration images, onto which the actual chess board corners (`findChessboardCorners(...)`) are drawn.

Script parameters :

Position	Name	Description	Default
1	`show-flag`	show the modified calibration images?	`False`
2	`save-flag`	save the modified calibration images?	`True`
3	`pickle-file-name`	picke file name	`calibration.pickle`

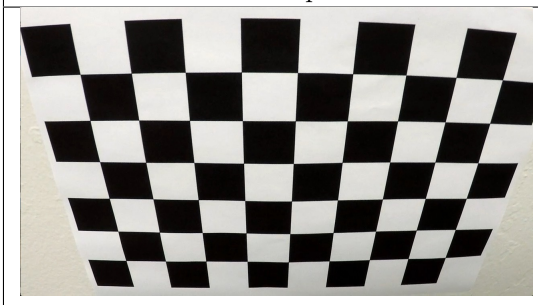
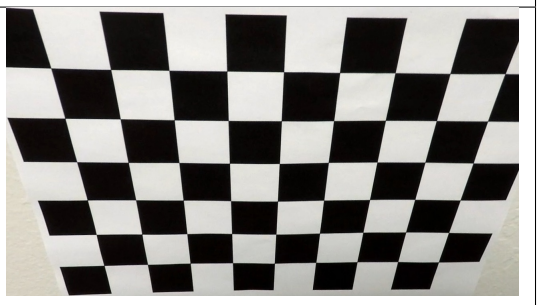
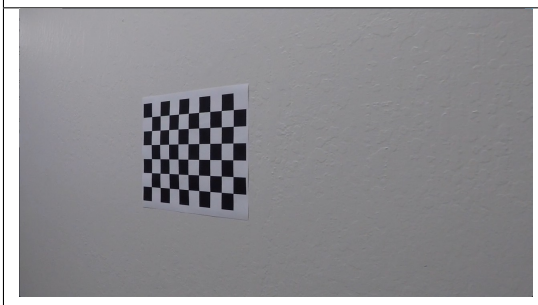
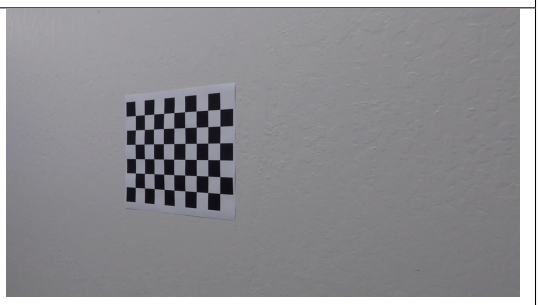
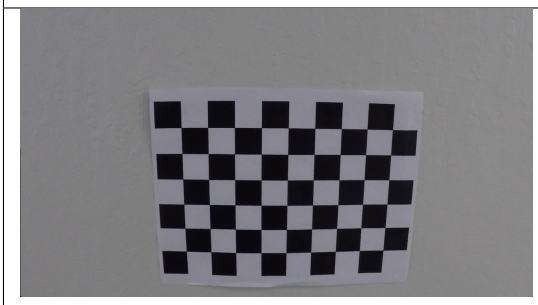
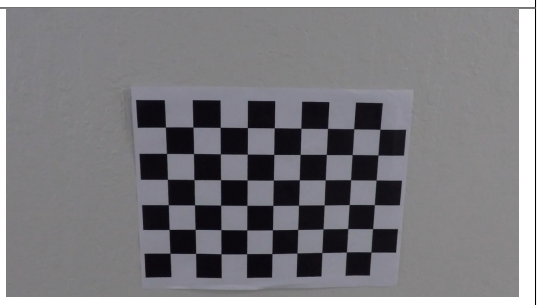
(the "save-flag" is only for the pictures, the pickle file will always be written)

Graphs *(some examples of the chessboard corners found in the calibration images)* :

Calibration picture	... with corners identified
	
	
	

Note: creation of actual undistorted images is done by the additional script `undistort.py`, which reads the calibration parameters from the pickle file (see source for required params).

(same examples with undistorted images) :

Calibration picture	... with corners identified
	
	
	

3 Perspective transform

To find a good perspective transform I implemented the script `transform.py`; It takes as positional input parameters the x and y coordinates for a source quadrilateral and a destination rectangle defining the desired transform.
Additional parameters control whether to show the test images and how they transform on the screen and potentially save transformed images and the transformation parameters (in a pickle file).
This allowed me to experiment with a number of different transformations quite rapidly. The "best" transform I came up with *(dubbed "MA")*, was derived from the first picture *with straight lines*.
To get the coordinates for the source quadrilateral I would usually open one of the *straight lines*- pictures with gimp to identify 4 corners that were not a rectangle but would obviously be close to one in the real. I would then experiment with a couple of target rectangle coordinates.

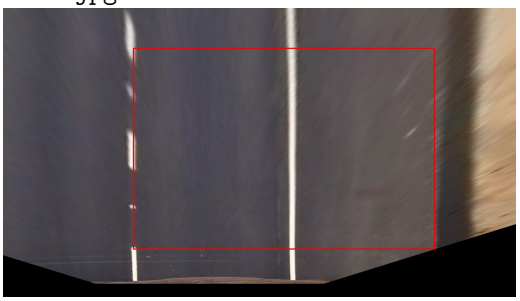
Script parameters :

Position	Name	Description	Default
1	`transformation-name`		(*mandatory*)
2	`cooordinate-string`		(*mandatory*)
3	`show-flag`	show images?	`False`
4	`save-flag`	save images?	`False`

The coordinates must be given as 16 integers within one string; that requires the string to be in quotes on the command line, like:

```
python transform.py MA '425 570 1276 570 806 458 586 458 325 600 1076 600 1076 100 325 100' 1 1
```

Graphs *(some examples perspective transform; source to transformed picture with the areas from the transform definition marked.)* :

source	transformed
`straight_lines1.jpg`	
	
`straight_lines2.jpg`	
	
`test3.jpg`	
	

4 Pipeline

The pipeline to actually process the whole lane finding is mainly implemented in the python sources `sliding_window.py` (helper functions/ methods for the

sliding window approach) and `single_image_pipeline.py`.
The latter contains most of the functionality including the reading/ writing of a parameter file (which uses pythons `configparser`).
If called directly it allows for the processing of the test images or a single image (depending on positional parameter) by the pipeline, while the various steps resulting images are shown and/or saved to image files. This allowed for experimentation with different parameter sets through the use of parameter files (which also "recorded" the parameters used).
The script to create the videos (`clip_pipeline.py`) is mainly a wrapper around the pipeline for single images, only with the function/ method- arguments set in a way, so that no direct image showing or saving is attempted (as the images are not to be saved individually, but as part of a video stream).

4.1 Pipeline overview

The images undergo following steps:

- Distortion correction (using the values from the calibration)
- Conversion to HLS and using only the S channel after that
- Thresholding the S- channel
- Sobel processing with a combination of magnitude and directional thresholding
- Canny
- Masking with a polygon
- Hough lines
- Perspective transform
- Sliding window processing with second order polynomial fit
- Filling of the lane (green) in transformed coordinates given the fitting functions
- Retransformation of the picture and combining with original image

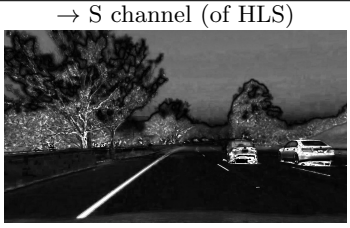
4.2 Example graphics

The pipeline visualized on test- image `test6.jpg` :

original image

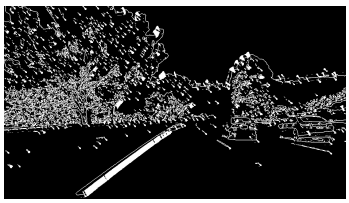
→ undistorted image

→ S channel (of HLS)

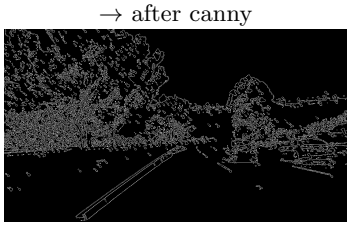


→ S- channel thresholded

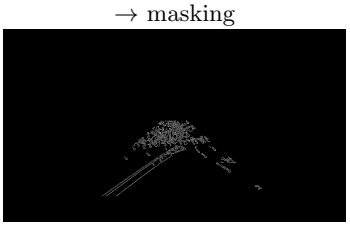
→ after sobel



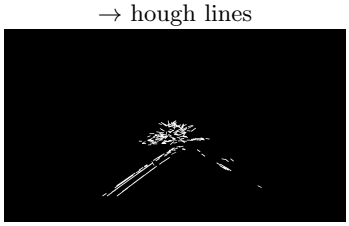
→ after canny



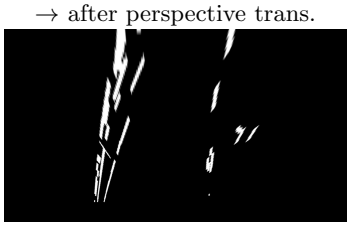
→ masking



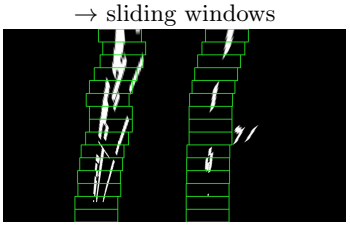
→ hough lines



→ after perspective trans.



→ sliding windows



→ final combined image

The parameters used are from `params_05.ini`.

4.3 Video

Up until now I could only half successfully convert the first 45 seconds of the `project_video.mp4`, see respective output folder.

5 Discussion

I was a bit frustrated, because I could not find a set of parameters (within my limited time), that would yield a pipeline robust enough for my taste.
While the implementation seems to work well enough on single images (see the graphics for `test6.jpg` above), the application to video yields a lane marking way too shaky. I tried to remedy this by remembering the starting positions for the sliding window processing from frame to frame (say: image to next image) and even implemented a scheme as to reuse the last polynomial fits with a certain fraction (30 percent) to smooth the computation; did not really help much.
The main problem seems to me, that the intermittent lane markings are not identified/ spotted well enough.

5.1 Possible improvements

I would make an educated guess, that extensive experimentation with different parameters for the various steps could be helpful. I think especially the "right" combination of parameters for the *canny- step*, the masking polygon and the hough-line processing might be promising (*I would actually need to implement the parameters for the masking polygon to be read from config - as the mask is hard coded as of now*).
To improve the lane detection within the context of the video, I would maybe experiment with something like "carrying over" a fraction of the sliding window pixels from the last frame randomly, but only those within the boxes; this might improve the stability of the polynomial fit (because more data points are available).