Advanced Lane Lines

Udacity Student

Udacity
Self-driving car Nanodegree program
Project 4

Contents

1 Introduction

This document serves as the project writeup for the Udacity Self-driving car Nanodegree Program, term 1, project 4, advanced lane lines. The project code, as well as the present document can be retrieved at https://github.com/kacl780tr/carnd_project_04.

The goal of this project is to develop a processing pipeline to identify and visually indicate highway lanes from on-board camera images.

2 Project structure

The project code is organized into several files:

lane_lines_adv.py the main file including processing pipeline setup, input and output of image and video files

calibration.py code related to camera calibration

components.py functions and objects related to region identification

display.py support functions for image display

pipeline.py the main processing pipeline objects

tools_image.py functions related to image manipulation

tools_perspective.py functions and objects related to perspective transformations

tools_region.py functions related to identification and manipulation of image regions

tools_search.py functions and objects related to identifying highway lanes on prepared images

The overall code design is primarily object-oriented.

3 Processing pipeline

The processing pipeline devised here is composed of the following stages:

1. correction for camera distortion
2. identification of appropriate regions to serve as endpoints of the perspective transformation
3. creation of an image within which the lane boundary lines can be identified
4. identification and mapping of the lane boundaries
5. drawing of the identified lane and auxiliary information onto the corrected image for output

We now discuss each step in further detail.

3.1 Main pipeline

As noted in section 2, the main processing pipeline is instantiated and configured in the file **lane_lines_adv.py** The objects which constitute the main processing pipeline are found in **pipeline.py**. Upon instantiation, the *Pipeline* object carries out the following operations:

1. loading of the required camera calibration *Camera* object
2. instantiating the *RegionBuilder* object, which is responsible for region identification
3. instantiating the *PathFunctionBuilder* object, responsible for locating lane boundaries
4. instantiating the *BinaryImage* object, which creates images suitable for lane location

The pipeline object can be configured to trace its activity, by writing images produced by the steps of the process into a list object. The pipeline will then call a *TraceHandler* object, if such is configured. Two TraceHandler subclasses are currently available: *TraceDisplay* will display the image sequence to the user, while *TraceDump* will dump the trace images to disk. This facility is useful for debugging.

3.2 Camera calibration

Code related to the camera calibration is found in **calibration.py** The main pipeline object retrieves the *Camera* object upon construction, and fails if it is unable to do so. The function **calibration_retrieve()** can either compute the calibration from a series of suitable images on the fly, or it can be loaded from disk, if already computed. Such is the case here.

An example of a raw camera image is shown in figure 1, while the same image, corrected by the camera object, is shown in figure 2.

Appendix A shows a further test of the camera calibration.

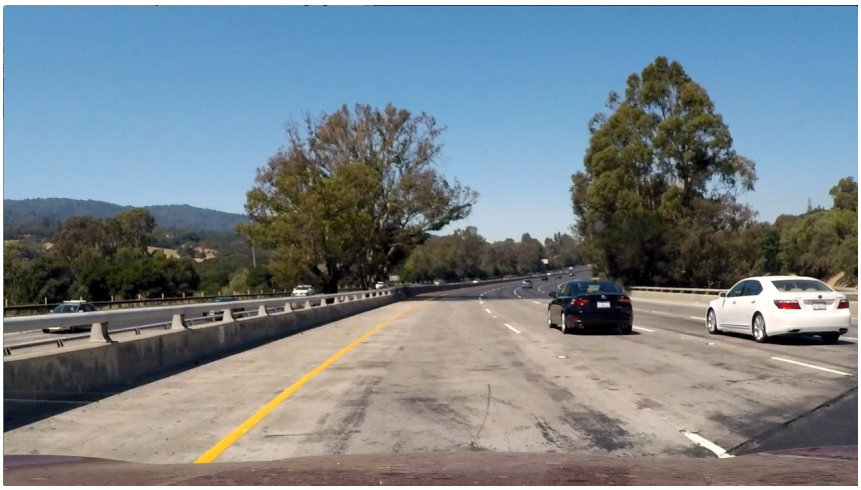

Figure 1: Example of raw camera image

Figure 2: Corrected version of image in figure 1

3.3 Region identification

It was observed during development that perspective transformations defined by points which accurately reflected the position of the lines in the image resulted in more regular lines after the transformation had been applied. Assuming that more regular post-perspective lines would be easier to locate, the decision was made to attempt to locate the lines in each frame.

The *perspective region* identification is carried out via a process adapted from SDC Project 1, consisting of

1. converting the image to grayscale
2. applying a Gaussian blur
3. Canny edge detection
4. Focus-masking on a region of interest
5. Hough line identification
6. processing of line information to obtain a **source** region for the perspective transformation
7. computing a **target** region for the perspective transformation

These steps are executed by the *RegionBuilder* object (**components.py**), which produces a *Region* object, which encapsulates the source and target regions, as well as the resulting *Perspective* object (**tools_perspective.py**), which is constructed from the identified source and target regions.

The grayscale image used by the *RegionBuilder* is produced by a subclass of the *ScaledImage* object (**components.py**) and is shown in figure 3. The currently utilized version of *ScaledImage* is that of *ScaledImageSMinusH*. It should be noted that this is *not* a standard grayscale image, but rather the S-channel of the HLS colorspace, with the H-channel subtracted to reduce clutter.

Figure 3: "Grayscale" image used for perspective region identification

Applying the Gaussian blur, with kernel size $= 5$ results in the image shown in figure 4.

Figure 4: Gaussian blur of image in figure 3

Next we apply the Canny edge detection algorithm, shown in figure 5, while applying the focus mask provides us with our Hough line detection target, as figure 6.

Using figure 6, the Hough line algorithm is run, and the resulting lines are processed to produce the source region of the perspective transformation. The source region detected from figure 6 is shown drawn over the corrected image (figure 2) in figure 7.

The target region for the perspective transformation is then simply the source region, with the left- and right-hand edges made vertical, and adjusted to result in a convenient magnification.

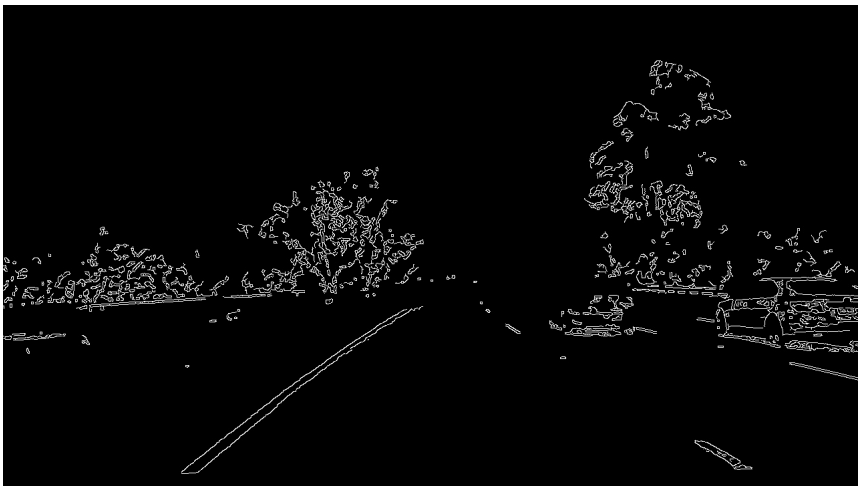

Figure 5: Result of Canny edge detection on blurred image in figure 4

Figure 6: Final target for Hough line detection

3.4 Search image for lane boundaries

The search image used for lane boundary searches is produced by a particular subclass of the *BinaryImage* object (**components.py**). The currently used implementation is found in the *BinaryImage-SobelSMHChannel* object. This implementation uses the *ScaledImageSMinusH* object to produce a single-channel image, then applies the Sobel operator in the x-direction, followed by binary thresholding. Binary thresholding is then applied directly to the single-channel image, and the resulting binary image is 1 wherever either of the preceding two binary images are themselves $= 1$, zero elsewhere.

This procedure, applied to the corrected image in figure 2, results in the image shown in 8. One can see that the lane boundaries stand out well, although there remains a considerable amount of clutter.

Applying the perspective transform as produced by the region identification outlined in section 3.3, gives us, in figure 9, the image to be used for lane boundary location. A nice side effect of the perspective transformation is the reduction in clutter

Figure 7: Source region for the perspective transformation as identified by the *RegionBuilder* object

Figure 8: Binary image used for lane boundary identification

3.5 Lane boundary path mapping

The procedure used here for locating and mapping the lane boundaries makes use either of the lane boundary from the previous frame, if available, or the left and right points of the perspective target region. That is, assuming that our perspective transformation started from correctly-identified lines, then these lines will be located, at the bottom of the image, precisely at the x-coordinates of the perspective target region. This can be seen by comparing figures 8 and 9 above.

Otherwise, if we have the *PathFunction* object from the previous frame available, then we use this as a guide for extracting the new boundary path. These values, either from the perspective region, or the previous path function, are identified in the code as *anchor* points.

The object responsible for executing this part of the pipeline is the *PathFunctionBuilder* object, found in **tools_search.py**. This object carries out the following steps:

1. based on the information in either the *PathFunction* from the preceding frame, or the x-coordinates of the target region, a vertical split point is calculated as being midway between the two lane boundaries, and the image is split into two vertically
2. each vertical split is then sliced into horizontal slices, resulting in 10 1D arrays of sums over the slices. The upper and lower y-coordinates of the slices are also tracked

Figure 9: Transformed binary image used for lane boundary location

3. each set of slices is then processed, from the bottom of the image to the top, according to[1]
   (a) a 1D convolution C is carried out, using a unit kernel of width 50
   (b) the convolution array is normalized

$$\bar{C} = \frac{C}{\sum_i C_i} \tag{1}$$

   (c) a support region is computed as

$$X_{\pm} = A(y) \pm (\frac{1}{2}W + M) \tag{2}$$

   where $X_{\pm}$ are the support region x coordinates,
   $A(y)$ is the current anchor value at y,
   W is the convolution kernel width, and
   M is a configurable margin value (currently 100 pixels)

   (d) the expected values $E[x]$ and $E[x^2]$ are calculated as

$$E[x] = \sum_{X_-}^{X_+} \bar{C}_i i \tag{3}$$

$$E[x^2] = \sum_{X_-}^{X_+} \bar{C}_i i^2 \tag{4}$$

   (e) the variance is calculated as

$$\mathrm{var}[x] = E[x^2] - E[x]^2 \tag{5}$$

   (f) a decay rate τ is calculated as

$$\tau = \tau_0 \frac{\sqrt{\mathrm{var}[x]}}{E[x]} \tag{6}$$

   (g) a mixing ratio β is then calculated as

$$\beta = e^{-\tau |A(y) - E[x]|} \tag{7}$$

[1] The implementation of this search algorithm is found in the functions **make_linepath_active()**, and **make_linepath_static()** in the file **tools_search.py**.

(h) and finally the new estimate for the lane boundary location is computed as

$$\hat{x} = (1 - \beta)A(y) + \beta E[x] \tag{8}$$

If the lane boundary location is being carried out from a fixed anchor ($A(y) = a$), then we additionally update the anchor value $A(y) = \hat{x}$ at each step to allow $A(y)$ to "follow" the new path. The logic behind equations 6 and 7 is twofold: one, we are less inclined to believe new estimates $E[x]$ that depart widely from our present estimate, and two, we are less inclined to believe new estimates for which the variance is high, indicating that there may be a lot of noise in the search image. In either of these cases, the new estimate $\hat{x}$ is biased more strongly toward the previous estimate.

At the conclusion of this procedure, we obtain a set of coordinates x, y to which we fit a second-degree polynomial. The *PathFunction* object then contains one polynomial for each of the left-hand and right-hand lane boundaries, and contains code to allow them to be drawn onto an empty image. The region between the lane boundaries extracted from the binary image in figure 9 is shown in green in figure 10. It shows that the lane boundaries in the transformed binary image have been located reasonably well.

Figure 10: Area between located lane boundaries in figure 9

3.6 Final image creation

In order to create the final output image, three steps are necessary:

1. invert the perspective transformation that produced figure 9 from figure 8 in order to create a version of figure 10 that can be combined with the corrected input frame, figure 2.
2. compute of an estimate of lane curvature
3. compute an estimate of vehicle deviation from lane centre
4. overlay the transformed path, and computed information on the corrected image

The result of applying the inverse perspective transform to figure 10 is shown in figure 11. It is seen to have the shape one would expect.

The *PathFunction* object includes the ability to return a copy of itself, with its units converted to meters[2], The conversion to meters is done as

[2]Member function *PathFunction.realspace_path()*.

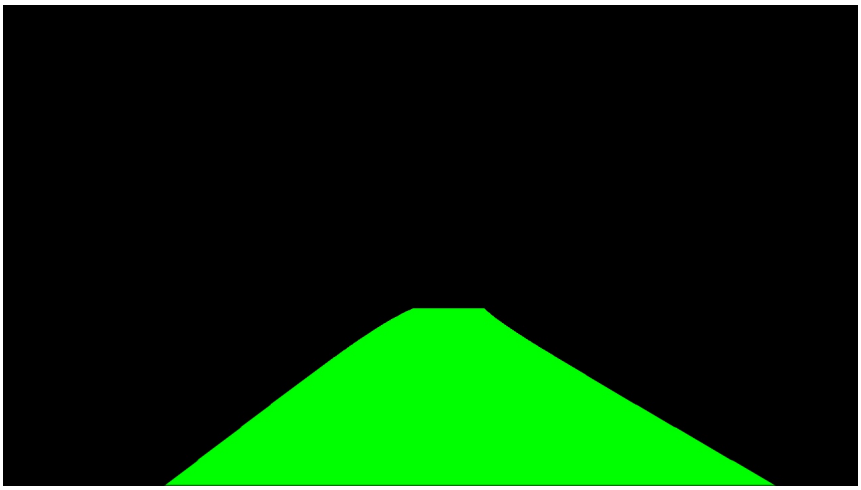

Figure 11: Result of inverse perspective transformation applied to figure 10

- for the x-direction $\alpha_x = \frac{3.7}{(x_r - x_\ell)} \frac{\text{m}}{\text{pixel}}$, where x_r, and x_ℓ are the pixel locations of the right- and left-hand lane boundaries, respectively, measured at the bottom of the image, and
- for the y-direction $\alpha_y = \frac{30}{h} \frac{\text{m}}{\text{pixel}}$, where h is the image height in pixels.

Once the *PathFunction* is converted to meters, then the curvature estimate is calculated as

$$\tilde{R} = \frac{1}{N} \sum_{i=1}^{N} \frac{\left(1 + \left(\frac{dx}{dy}(y_i)\right)^2\right)^{3/2}}{\left|\frac{d^2x}{dy^2}(y_i)\right|} \tag{9}$$

where the points y_i are taken over the entire image[3], and both left- and right-hand curvatures are averaged to produce a single number.

The estimate of deviation (in meters) of the vehicle centre from the centre of the lane is obtained as

$$\Delta_x = \frac{\alpha_x}{2}(w - (x_r + x_\ell)) \tag{10}$$

where w is the image width in pixels. This assumes that the camera is mounted dead center and correctly aligned with the vehicle.

The output of the pipeline applied to the raw input image, and following through the various steps, is shown in figure 12. This result shows that the processing seems to be broadly correct, although the curvature estimate is perhaps lower than one would have expected. The curvature, however, is sensitive to noise in the polynomial fits.

4 Results

As can be seen from the result in figure 12, the pipeline succeeds in finding the lanes, in at least this example. The use of perspective transformations for each frame was motivated by obtaining the most accurate transformations, but it may turn out to really only increase the noise in the results, despite it being able to provide anchoring hints to the lane search procedures.

[3] Converted from pixels to meters.

Figure 12: Final output of processing pipeline for raw image from figure 1

The use of variance and distance weighting in the updating of the path from frame to frame does not seem to have resulted in stable lane marking, or curvature numbers. However, not having run the pipeline with many other updating methods, I have no basis for comparison.

The object-oriented design of the pipeline, at least, makes such investigations easier to carry out, and the step tracing facility should facilitate this.

In addition to this document, and the code files described in section 2, the following files are available:

output_images the output of the pipeline trace for image test5.jpg

project_video.mp4 the project video as processed by the current pipeline

calibration.p the stored camera calibration

It is certainly possible that very tight turns or changes in lighting conditions may overwhelm the region identification, although in the case of videos it is hoped that the fallback to the previous region allows the pipeline to continue. The danger with the frame-by-frame region identification is that the new region is mis-identified, and is far from the previous region. An improvement might be to fall back to the previous region if the new region deviates too much.

The computation of the variance, as we have done above, might be put to better use as a quality score of the particular lane identification, rather than as a weighting adjustment as we have done here.

5 Conclusion

The main conclusion to be drawn from this project is that there are a basically limitless number of combinations of image processes, filters, transforms, etc., that may be applied to such a task. The main purpose to which such a pipeline should be aimed is the generation of labeled data sets which can then be used to train neural networks to perform the task in a more powerful and robust way.

A Camera calibration test

The camera calibration is further demonstrated here. Figure 13 shows the calibration test image before correction, while figure 14 shows the same image after correction. This serves to demonstrate, in addition to the result in figure 2, that the camera distortion correction is functioning as intended.

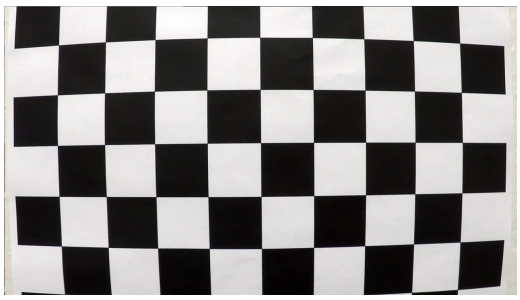

Figure 13: Camera calibration test input

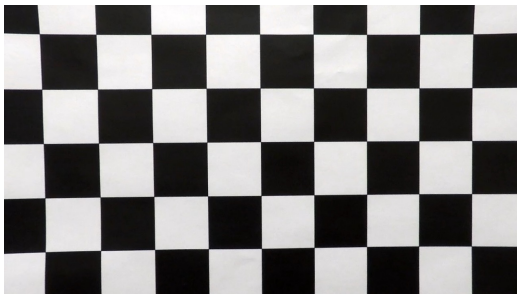

Figure 14: Camera calibration test output