Advanced Lane Lines Project Writeup

Charlie Wartnaby

Applus IDIADA

charlie.wartnaby@idiada.com

Udacity Self-Driving Car Engineer Nanodegree

Term 1 Computer Vision and Deep Learning

5 Sep 2017

Abstract

A Python program was successfully written to process single images or a complete video file to identify road lane line markings, including the following features:

- Correction of calibration distortions
- Image processing through colour, gradient and threshold transformations to highlight lane markings in preference to other objects
- Transformation of the image to a birdseye perspective
- Fitting of polynomial curves to the lane line pixels using an initial 'sliding convolution' approach, and then an 'iterative least squares fit'
- Extraction of real-world parameters such as road curvature and camera offset position
- Visualisation of the fitting process and extracted data by augmentation of the original video

Some limitations of the methods used are finally discussed, and possible enhanced approaches discussed.

Write-Up

This document is the project write-up, meeting the rubric points specified. The headings correspond to the structure of the rubric. The output images can be found in **`output_images`**, including the processed video **`output_images/project_video.mp4`**.

Accompanying Code

All of the code can be found in the accompanying file **`find_lane_lines.py`**, which is commented extensively; fragments are reproduced in this write-up, but please see the source file for complete implementation.

The entry point is, being Python, at the bottom of the file:

```
if __name__ == "__main__":
    # Validate command-line arguments and run appropriate action(s)
...
```

Command-line options have been included to produce output to meet the various rubric points as follows. Validation was included to ensure that required options were all provided together where appropriate. The program could be called to perform camera calibration, to work on single images, or to process a video as required.

Command-line option	Description
--cam_cal_path_pattern	Path pattern to match camera calibration images
--cam_cal_nx	Expected number of calibration image x-direction corners
--cam_cal_ny	Expected number of calibration image y-direction corners
--cam_cal_eg_src	Example image to process to demonstrate camera calibration
--cam_cal_eg_dst	Output image to demonstrate camera calibration
--img_path_pattern	Process single JPEG images matching this path pattern
--img_out_dir	Output directory for processed single binary images
--video_in_file	File path for input video to process
--video_out_dir	Output directory for processed video

Camera Calibration

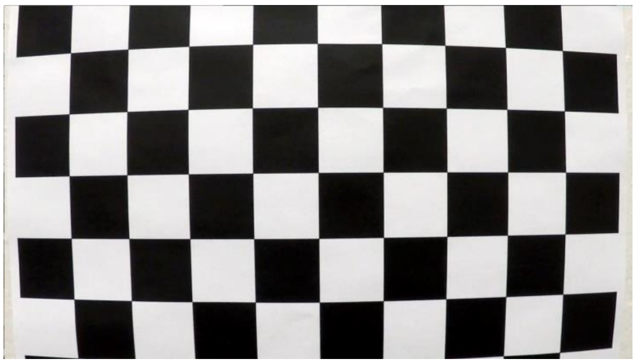
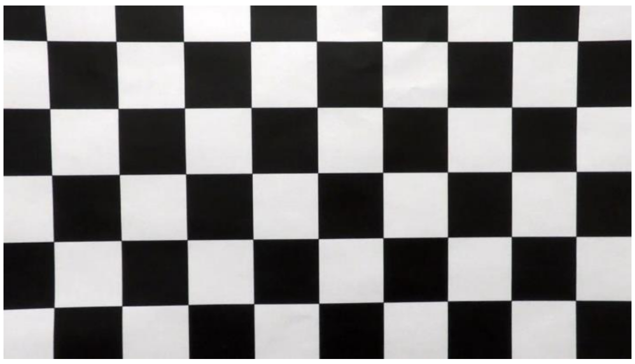
The function **`calibrate_camera_from_images()`** implements camera calibration, using OpenCV functions **`findChessboardCorners()`** to identify all grid intersections in each calibration image, and **`calibrateCamera()`** to compute the required camera matrix and distortion coefficients. As a set of images shot from different positions were used simultaneously to obtain the best fit, the rotation and translation vectors were meaningless and were discarded.

The program identified images which did not contain all of the expected corners and discarded them.

The program was invoked as follows to produce an example undistorted chessboard image:

```
(C:\udacity\Miniconda3\envs\carnd-term1) C:\svn\courses\udacity\self_driving_car\CarND-Advanced-Lane-Lines>python find_lane_lines.py --cam_cal_path_pattern camera_cal/calibration*.jpg --cam_cal_nx 9 --cam_cal_ny 6 --cam_cal_eg_src camera_cal/calibration1.jpg --cam_cal_eg_dst output_images/undistorted_calibration1.jpg
Calibrating camera expecting 9 by 6 corners in images matching 'camera_cal/calibration*.jpg'
Warning: expected corners not found in 'camera_cal\calibration1.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration4.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration5.jpg', skipping that file
17 image files had usable chessboard corner points
Camera calibration complete
Undistorted example camera_cal/calibration1.jpg output as
output_images/undistorted_calibration1.jpg
```

Here is the example image before and after undistortion:

Pipeline (test images)

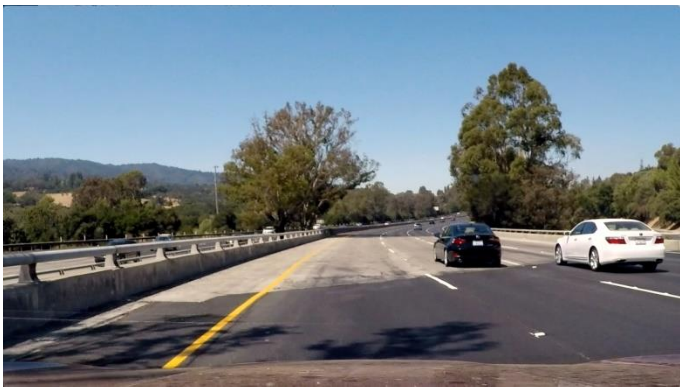
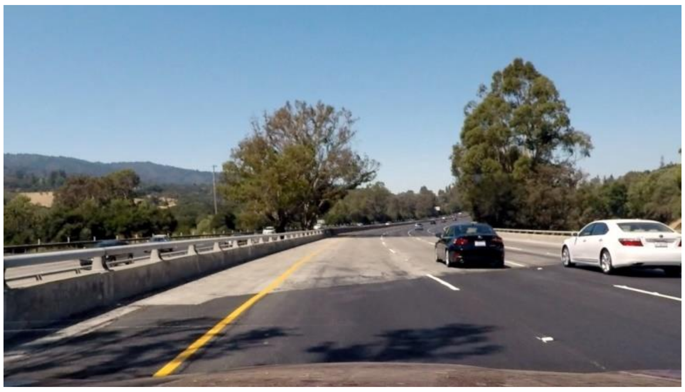
Distortion correction

The distortion matrix obtained by camera calibration (see above) was applied to each input image or frame to correct for camera distortion. See function **`undistort_one_image()`** in the code, which merely wraps the OpenCV function **`undistort()`**.

The program was invoked as follows to undistort the supplied **`test4.jpg`** as an example:

```
(C:\udacity\Miniconda3\envs\carnd-term1) C:\svn\courses\udacity\self_driving_car\CarND-Advanced-
Lane-Lines>python find_lane_lines.py --cam_cal_path_pattern camera_cal/calibration*.jpg --
cam_cal_nx 9 --cam_cal_ny 6 --cam_cal_eg_src test_images/test4.jpg --cam_cal_eg_dst
output_images/undistorted_test4.jpg
Calibrating camera expecting 9 by 6 corners in images matching 'camera_cal/calibration*.jpg'
Warning: expected corners not found in 'camera_cal\calibration1.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration4.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration5.jpg', skipping that file
17 image files had usable chessboard corner points
Camera calibration complete
Undistorted example test_images/test4.jpg output as output_images/undistorted_test4.jpg
```

Here is the image before and after undistortion. The effect is quite subtle, but it can be seen (for example) that the back of the white car is entirely in-frame before undistortion, but truncated on the right edge of the undistorted output:

Thresholded binary image

Several steps were involved in processing each image to highlight lane lines, rejecting other content in the picture.

1. Colour transformation

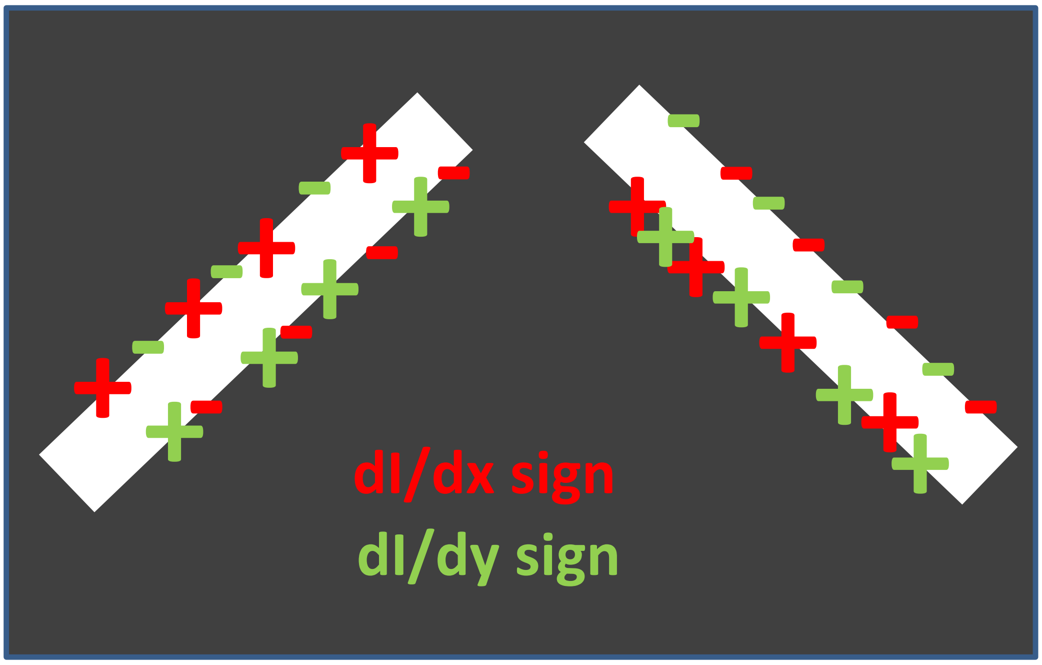
See function colour_trans_one_image(). This transformed the image to the HLS colour space and then kept just the S(aturation) channel as monochrome output, because this channel highlighted both yellow and white well – the lane line colours we need to detect.

2. Gradient analysis

See function **`find_useful_gradient()`**.

A Sobel filter was used to obtain the intensity gradient in the x and y directions (i.e. dI/dx and dI/dy), with a fairly generous smoothing kernel size of 15 pixels.

It was noted that lane lines are typically sloping at roughly 45 degrees to the horizontal when the vehicle is going straight ahead, though this becomes less true when the road is curving sharply. This tendency was exploited however to emphasize sloping lines somewhat more favourably than horizontal or vertical ones. Consider the sign of the x- and y-direction Sobel derivatives at the edges of bright left and right lane lines:



For the left lane line, dI/dx and dI/dy have opposing signs on each edge of the line. Therefore the **difference** dI/dx – dI/dy will have strong magnitude for sloping left lines.

By contrast, the right lane line has the same sign for dI/dx and dI/dy on each edge. Therefore the **sum** dI/dx + dI/dy will have strong magnitude for sloping right lines.

The sum and difference were therefore computed **before** taking an absolute magnitude, one to emphasize left lines and the other right lines. The result was multiplied by a factor including just **abs(dI/dx)** and **abs(dI/dy)**, because that is finite for sloping lines but zero for horizontal or vertical lines, so sloping lines are preferred by this filter. However, because lines in the distance can become vertical in curves, a tolerance factor was included to weaken this selection.

3. Region masking

In **`find_useful_gradient()`** a trapezoidal mask was applied to the image before intensity normalisation as follows, to reject bright areas of the image that are not of interest and which would otherwise diminish the final brightness of the desired lane line pixels:

```
# Before normalisation, multiply by region of interest mask so that we discard
# any accidental bright regions elsewhere in image that might dominate overall
# brightness otherwise
masked_sobel_sum = sobel_diagonal_filtered * interest_region_mask
```

4. Binary thresholding

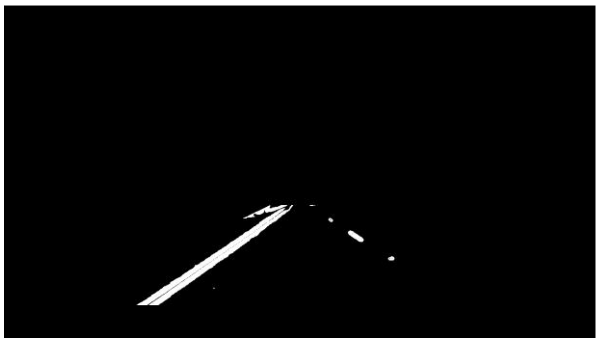
In **`apply_binary_threshold()`** the monochrome (greyscale) image thus far was subjected to a threshold filter to reject weak pixels. It was found that quite a low threshold was required to retain weaker lines.

The binary image for each frame is included as a picture-in-picture (PIP) segment in the final video, after birdseye distortion and the addition of visualisation graphics for line identification.

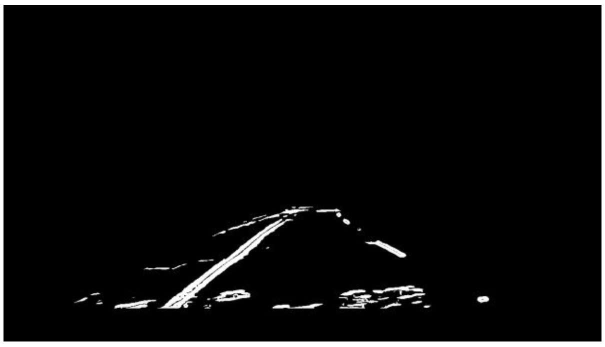
The project rubric requires examples of images after thresholding however. These were obtained by invoking the program as follows:

```
(carnd-term1) C:\charlie\udacity\CarND-Advanced-Lane-Lines>python find_lane_line
s.py --cam_cal_path_pattern camera_cal/calibration*.jpg --cam_cal_nx 9 --cam_cal
_ny 6 --img_path_pattern test_images/*.jpg --img_out_dir output_images
Calibrating camera expecting 9 by 6 corners in images matching 'camera_cal/calib
ration*.jpg'
Warning: expected corners not found in 'camera_cal\calibration1.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration4.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration5.jpg', skipping that file
17 image files had usable chessboard corner points
Camera calibration complete
Processing single images matching test_images/*.jpg
8 processed single binary images written with same filenames to directory output_images
```

Here are some examples of the output after binary thresholding. The first image (**`straight_lines1.jpg`**) produces very clean output:



test4.jpg is a much less clean example, with shadows on the road resulting in some unwanted artefacts. However, transient noise of this sort in the video is handled by filtering across time between frames:

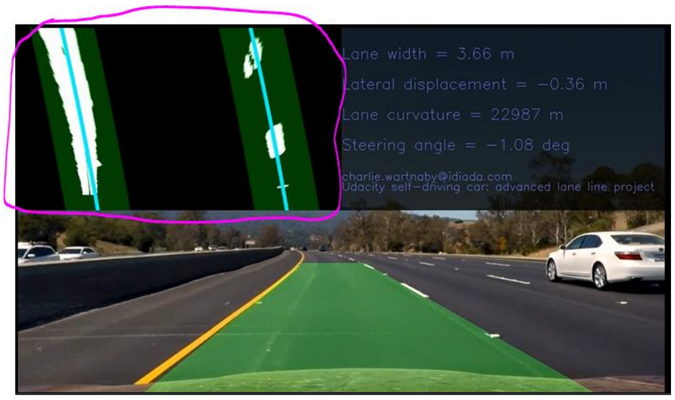


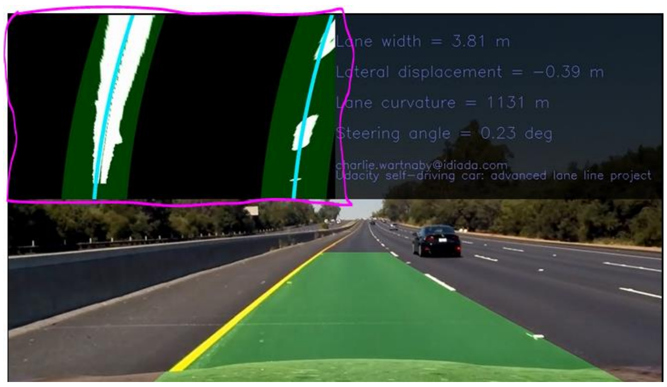
Perspective transform

A transformation matrix for birdseye distortion is computed (just once) in **`compute_warp_matrices()`**, with pixel numbers obtained by hand from **`straight_lines1.jpg`**.

The distortion of each image or video frame is then done using that matrix in **`warp_image()`**, which is little more than a wrapper for the OpenCV function **`warpPerspective()`**.

The perspective-transformed image, with added visualisation of lane line identification, is included in every frame of the output video in the top left corner. Here are examples, as required by the project rubric; the birdseye warped picture is highlighted:


Lane width = 3.66 m
Lateral displacement = −0.36 m
Lane curvature = 22987 m
Steering angle = −1.08 deg
charlie.wartnaby@idiada.com
Udacity self-driving car: advanced lane line project


Lane width = 3.81 m
Lateral displacement = −0.39 m
Lane curvature = 1131 m
Steering angle = 0.23 deg
charlie.wartnaby@idiada.com
Udacity self-driving car: advanced lane line project

Identification of lane-line pixels

Two methods were used to identify lane lines, both suggested in the tutorials, but with enhancements applied, as described below:

Convolution method

When starting from scratch, or if the previous fit was lost (see below), a convolution method was applied to identify the most significant peaks in each of 10 vertical "slices" of the birdseye-distorted image. See **`find_lines_by_sliced_convolution()`** in the code.

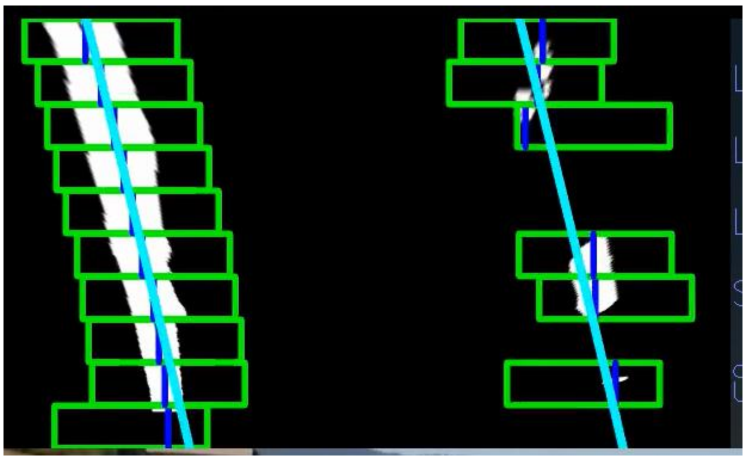
In the tutorial, the convolution was done with a step or 'top hat' function. However, that does not highlight the centre of a group of pixels particularly; it yields the same convolution value so long as the pixels are completely contained within the step. Instead, a Gaussian bell curve was used with its own gentle peak in the centre. That gives a convolution value that peaks when it slides over the centre of a group of pixels.

To reject small patches of noise, a lower threshold was applied below which no lane line was detected in each slice.

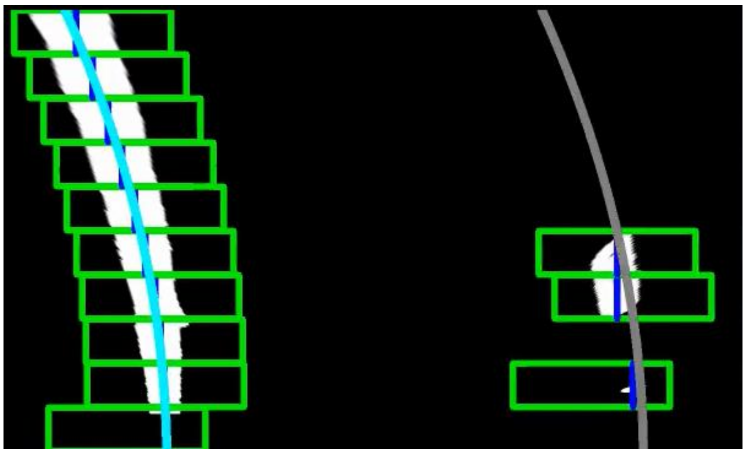
For each slice where a line was detected, a single lane point was then defined as the centre of the convolution peak at the centre height of that slice. Those points were then fitted with a polynomial, but only if a minimum number were available, to avoid a nonsense fit. Therefore it could be that no lines were found on the first frame, and the whole program was written to be robust to having no available fit.

Once a fit for each line was obtained, a direct least-squares polynomial fit was attempted on successive frames (see next). However, by temporarily setting **`self.reinit_count_on_good_detection = 0`**, the program could be forced to use the convolution method on every frame, which is then shown in the top left of the output video. Here are some examples:

Here, a fit has been obtained successfully on both sides. Each green box shows the area scanned by convolution, with a dark blue bar showing the peak of the convolution output value.



In this next image however, not enough slices yielded valid points on the right hand side, so the previous fit (in grey) has been left alone:



Polynomial fit method

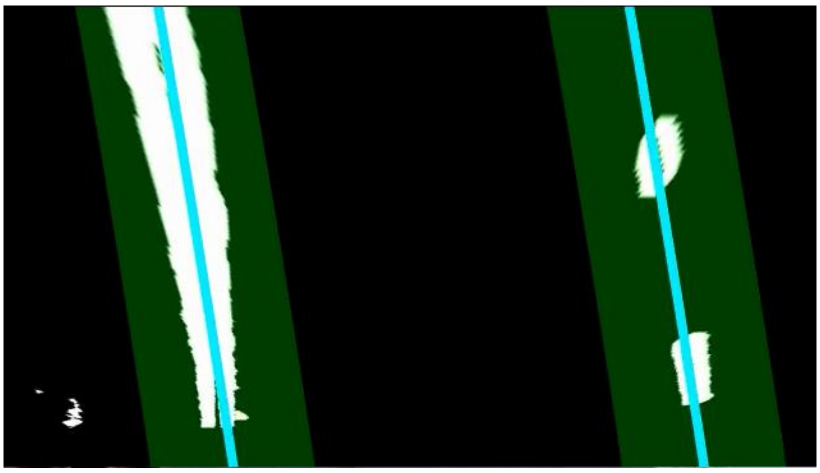
If a polynomial fit was already available from preceding frames, all bright points within a window about that fit in the x-direction were detected and a least-squares fit attempted; see **`find_lines_near_existing_fit()`** in the code.

To make this more robust, the fit was only accepted if a certain minimum number of points lay within the window to fit, and also if the proportion of points within the window exceeded a threshold proportion of the total points available in that half of the image; this was to avoid nonsense fitting of any bursts of noise.

The fit was made more robust by filtering in the polynomial coefficients, so no single frame had a dominating effect. The first two coefficients of the (second order) fit were averaged together between the left and right hand sides to ensure the lines were parallel; this helped fit the side with only intermittent markings, where a solid line was present on the other side. The weighting of left and right hand sides depended on the number of points on each side.

If a good fit was not available, a counter was decremented and the coefficients left alone. The previous fit line was visualised in grey. If a fit was available, the counter was re-initialised to a 'good' value, and the updated fit shown in blue. If the counter eventually reached zero, corresponding to a significant period with no available fit, then the convolution method was used to re-initialise the fit as described above.

A typical image is shown here, where the white pixels are those obtained by previous gradient and threshold analysis, the dark green zones are where points are selected for polynomial fitting, and the light blue lines show that polynomial fit:



Such images are shown in every frame of the submitted processed project video in the top left corner.

Radius of curvature and lane position

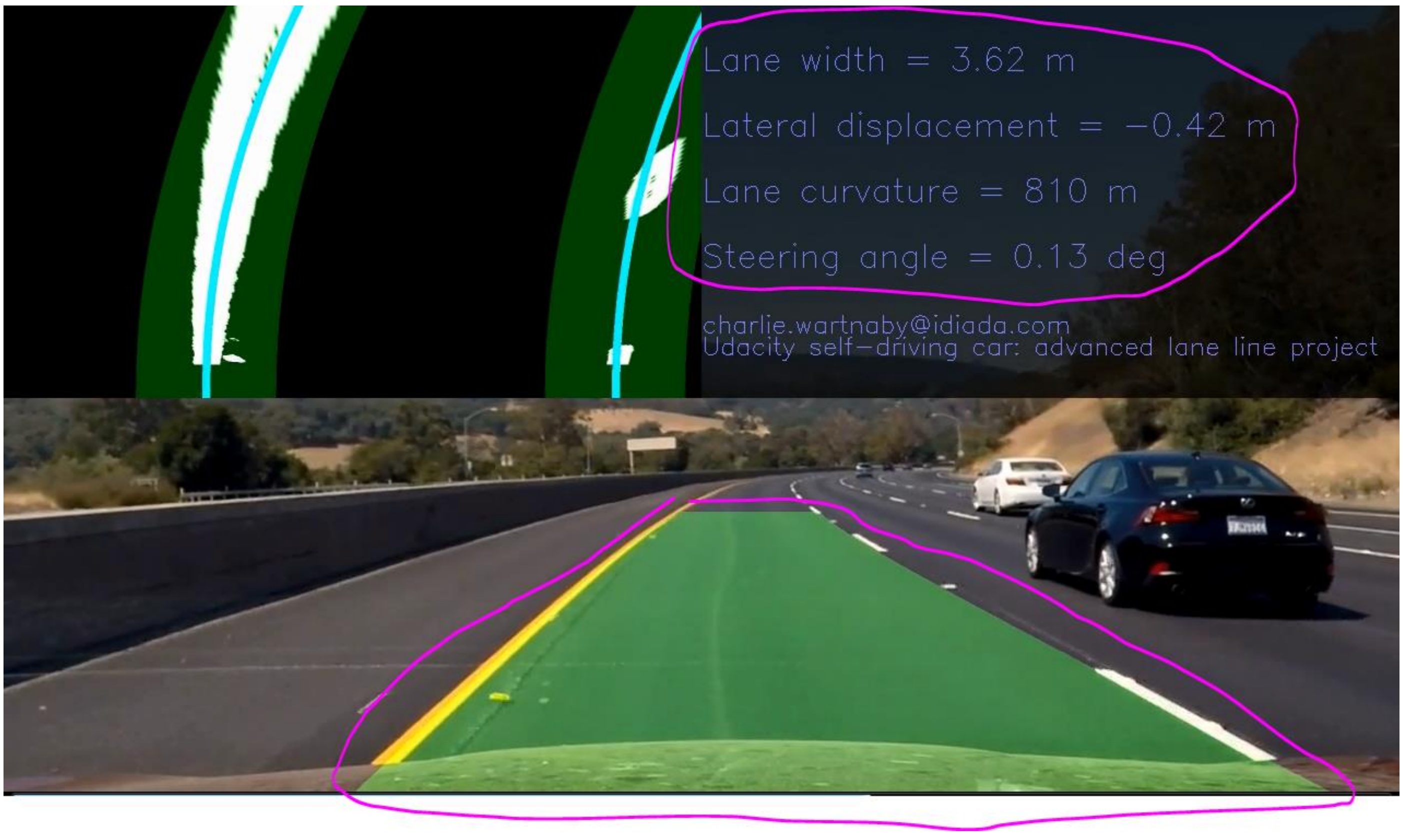
Methods from the tutorial material were used to:

1. Transform the lane line fit polynomials (expressed in pixels) to new polynomials (expressed in real-world metres). See **`convert_pixel_to_real_poly()`** in the code.
2. Obtain real-world parameters from those polynomials for output; see **`calc_real_world_data()`** in the code. That included:
    a. Radius of curvature (using tutorial code fragment)
    b. Offset of camera from centre of lane, from polynomial offset coefficients
    c. Lane width in metres (but assuming a 'known' width in the calibration image), from the difference in polynomial offset coefficients
    d. Steering angle (using tangent of fit polynomial)

These data were superimposed onto the top-right corner of each frame of the video. They were first-order filtered through time to give representative values that did not jitter too fast to be read.

Result superimposed back onto road

The region between the lane fit polynomials was warped back to real-world perspective and superimposed onto the 'original' (but camera-undistorted) road image using the same methods as the tutorial material. See **`superimpose_lane_fit()`** in the code for details. An example frame is shown below from the output video, where the ringed green shape between the lane lines is the highlighted inter-lane detection area, and the real-world values (required by the rubric) are also highlighted. Note that the camera here is visibly to the left of the lane centre, and the determined offset is indeed -0.42m (meaning to the left). The radius of curvature of 810m here is of the right order of magnitude for the 1km advised in the project briefing:


Lane width = 3.62 m
Lateral displacement = −0.42 m
Lane curvature = 810 m
Steering angle = 0.13 deg
charlie.wartnaby@idiada.com
Udacity self−driving car: advanced lane line project

Pipeline (video)

A video could be processed by invoking the program as follows:

```
(carnd-term1) C:\charlie\udacity\CarND-Advanced-Lane-Lines>python find_lane_line
s.py --cam_cal_path_pattern camera_cal/calibration*.jpg --cam_cal_nx 9 --cam_cal
_ny 6 --video_in_file project_video.mp4 --video_out_dir output_images
Calibrating camera expecting 9 by 6 corners in images matching 'camera_cal/calib
ration*.jpg'
Warning: expected corners not found in 'camera_cal\calibration1.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration4.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration5.jpg', skipping that file
17 image files had usable chessboard corner points
Camera calibration complete
Processing video file: project_video.mp4
[MoviePy] >>>> Building video output_images\project_video.mp4
[MoviePy] Writing video output_images\project_video.mp4
100%|#####################################9| 1260/1261 [04:59<00:00,  4.37it/s]
[MoviePy] Done.
[MoviePy] >>>> Video ready: output_images\project_video.mp4

Processed video written to: output_images\project_video.mp4
```

This used the image processing as detailed in the section above, taking advantage of the occurrence of successive frames to filter in the new lane fit and real-world values, making it more robust that the determination of positions or values from a single frame.

Please see **`output_images/project_video.mp4`** for the final output.

Discussion

Although the program as developed was successful for the required test video, it exhibits patchy performance on the 'challenge' videos provided, which have more rapidly shifting curves, lines obscured by a motorcyclist, deep shadows, etc. Lack of time precludes further development on this project, but it would not as it is be sufficient for a production system. As with neural network training, focusing on a single test video does run the obvious risk that the adjustable parameters in the program are 'over-fitted' for this specific example.

Also, no video taken at night, or in poor light, or in heavy rain or even snow was analysed. These would present greater challenges to the vision processing used here; possibly different, complimentary methods would have to be applied in parallel and then the most robust output selected. It would be interesting to attempt a neural network approach with highly varied training data, instead of the traditional algorithmic approach applied here. Training that network would of course require labelling of the correct lanes; that could be done in part using the algorithmic approach here, at least for cases where it works well.