Advanced Lane Line Detection

Nicolas Keck

1. Camera Calibration

To calibrate the camera used in the project, I ran cv2's chessboard calibration on the calibration images provided in order to determine the distortion on the camera. I then saved these distortion numbers to a numpy zip file so that my pipeline wouldn't have to re-calculate distortion everytime I wanted to run it. The code for this process can be found in **camera_calibrate.py**. I also tested the undistortion effect on **calibration1.jpg**, the resulting undistorted image can be found in **output_images/Undistortion_test.png**.

2. Pipeline (Single images)

Distortion correction:

To apply the distortion correction to each pipeline image, I first load in the distortion parameters I saved in the numpy zip I mentioned earlier. These numbers are loaded into 5 separate arrays and are used to define a new, optimal camera matrix for undistorting each pipeline image. You can tell that the undistortion is applied correctly because the reflection lines on the windshield of the car in the new images are completely straight. This can be seen clearly in **output_images/Undistorted_img.png**. Note that I do not actually crop out the region of interest from the rest of the image, mostly because I don't want the size or placement of pixels in the image to be altered. The code for this process can be found on lines 5 through 19 in **Pipeline_visualizer.py**.

Combined binary image:

The combined binary image I use for this project is made up of three distinct parts: a sobel x binary, a saturation binary, and a value binary. Each of these individual binaries are processed with parameters I personally tuned on the test images, and when combined provide a good overview of the lane lines on the road. Example single binary images can be found in **output_images**, the sobel binary is called **Sobel_binary.png**, the value binary is called **Value_binary.png**, and the saturation binary is called **Saturation_binary.png**. Finally, the combined binary is called **Combined_binary.png**.

Perspective transform:

In order to perform the perspective transform, I had to manually tweak and tune variables in order to ensure that the straight lines within the straight line images were still straight after the warp. To do this, I essentially just eyeballed some points on the line and then continually changed the points once warped until the lines were straight. In order to tune the values I used **Warp_tuner.py** and the output of warping the combined binary image can be found in **output_images/Warped_binary.png**. An example image comparing the original image lane lines and the warped lane lines can also be found at **output_images/Warp_test.png**. Please note that the colors are incorrect because I read the image in BGR, but displayed it as an RGB.

Start points	End points
(598, 452)	(355, 0)
(299, 654)	(350, 720)
(1020, 645)	(945, 720)
(715, 452)	(955, 0)

Lane Line Detection:

In order to detect the lane lines, I run my warped binary image through a sliding window to detect major spikes on either side of the image. This is done by taking a histogram of the combined binary image, and then detecting non zero pixels which could represent lane lines. The hyperparameters I found to be most effective were 20 windows with a margin of 50 and a minpix of 50. The code for this can be found in lines 63 through 133 in **Pipeline_Visualizer.py**. Note that in the actual pipeline, I've also implemented a method for searching around an already-fit polynomial. An image of the sliding window result can be found in **output_images/Sliding_window_result.png**. A polynomial is also fitted to the detected pixels on the left and right side, creating two lane line estimations. Two example polynomials can be seen in **output_images/Polynomials.png**. The code for fitting polynomials is in lines 135-140 of **Pipeline_visualizer.py**.

Radius of Curvature and Vehicle Position:

The radius of the curvature of the lane lines is calculated by using the formula seen in class. When outputted visually onto the video, the 5-frame mean is used instead of the actual curvature value, as it changes too quickly to see if a mean is not used. One issue I found with the radius of the curvature is that straighter lines tend to have absurdly large curvature radii due to their first term being very small and the denominator in the calculation fraction. Meanwhile, the distance from the center calculation is based upon the assumption that the camera is located in the middle of the car. Using this assumption, we can calculate the distance between the center of the lane and the center of the image, thus calculating the displacement. This is also displayed as a 5-frame in the video. An example image of these overlays can be seen in

output_images/Curvature_example.png. The code for both of these operations can be found on lines 161 through 187 and 216 through 226 of **Video_annotator.py**.

Warping Result onto Original Image:

I found that warping the result onto the original image in a satisfactory manner took a bit of effort. I found that cv2.addWeighted() wouldn't make my lane lines visible enough within the video, so I took a different approach using masking. When my left and right lane polynomials are found, I draw them on two separate images. One is the image used for dewarping, and contains a colored line; the other is used for masking, and contains a white line. I then dewarp both of these images, and use bitwise_and to create a hole in the original image the size of the lane lines. I also use the white line image as a mask on the colored lines image. This allows me to add both images together and overlay the colored line perfectly atop the original image. The example resulting line can also be seen in **output_images/Curvature_example.png** and the code can be found in lines 189 through 214 of **Video_annotator.py**.

3. Pipeline (Video)

The full pipeline code can be found in **Video_annotator.py**. As was stated earlier, the pipeline originally runs a sliding window in order to determine where lane lines are, but once a polynomial is found it searches around the polynomial to find lane lines instead. A sliding window can be re-performed if the lane line is found to be not detecting properly, which happens twice in the video. In these scenarios, the curvature dips way too low, and detecting this, the program re-draws the sliding window such that the car gets back on track again.

My completed video can be found at: **project_video_output.avi**

I also attempted the challenge video, but there is too much noise for the program to run effectively. In the future I will probably update this project with more noise detection or finer parameters in order to attempt the harder videos. My challenge video attempt can be found at **challenge_video_output.avi**.

4. Discussion

The processing is quite processor-intensive, and I do not think I would be able to run this on my laptop in real-time. Some of the videos are around 50 seconds long, but the processing takes much longer. Either a more efficient method of lane detection needs to be used to rectify this, or an autonomous vehicle will need to have a much more beefy processor.

Some other issues with the program include issues with quickly-curving lines, something which can be fixed with more care and effort in the future, and the aforementioned noise issue. Again, noise can probably be fixed with some finer parameters and a bit more time.

Finally, I think it would be interesting to see if I could get a neural network to perform all of this for me. While the training would take an exceptionally long time, once the NN is up and running, using its fine-tuned algorithms to detect the lane lines instead of the nonzero and histogram methods could likely be much more time-efficient.