Advanced Lane Finding

Project Writeup

Tevis Gehr

Monday, May 29 2017

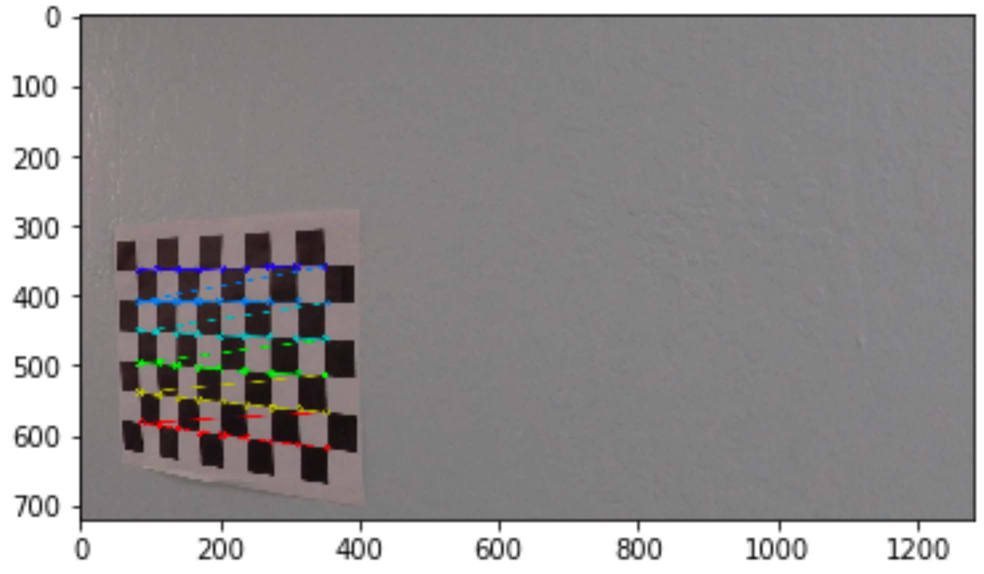
Introduction

In this project, I used video from a camera that was mounted on a car driving down the highway. Using OpenCV, I calibrated the camera and processed each frame in order to detect lane lines. I also calculated lane curvature and vehicle offset from the lane center. The detected lane lines were then drawn onto each frame and the images were put together to form a new video identical to the original with overlays of the detected lane lines, the lane curvature, and the vehicle offset. Below is a detailed description of the lane detection process in the order that it is implemented in the Jupyter Notebook file.

Camera Calibration

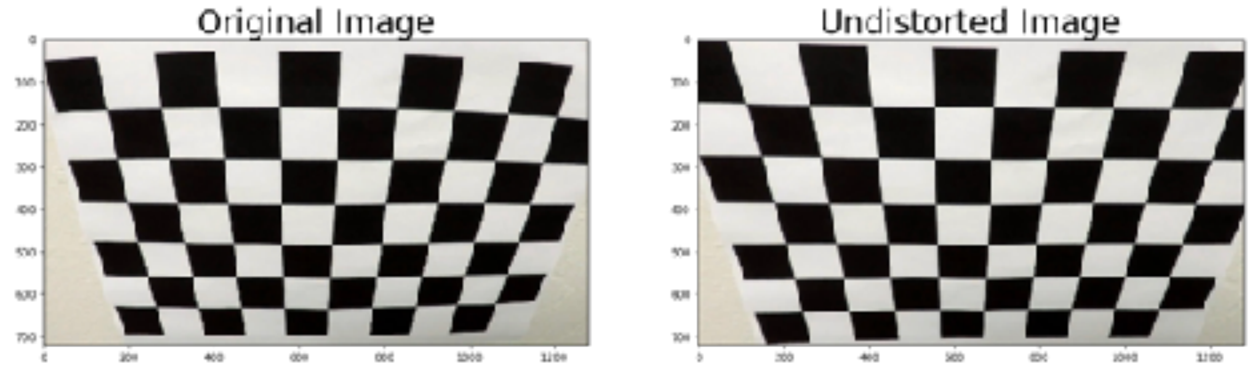
1. **Find Corners**

The 9x6 internal corners of a chessboard pattern were found on several images in various locations and orientations. This was done using OpenCV's cv2.findChessboardCorners method. An example result is shown below. The points from each of the images were compiled into two arrays to fully describe the distortion of the camera.



2. **Calibrate**

Using the object and image points found at the corners of the calibration images, the camera calibration matrices were found using OpenCV's cv2.calibrateCamera method. The calibration returned a matrix transformation that was then applied to all images before processing. An example of a calibrated chessboard image is shown below.

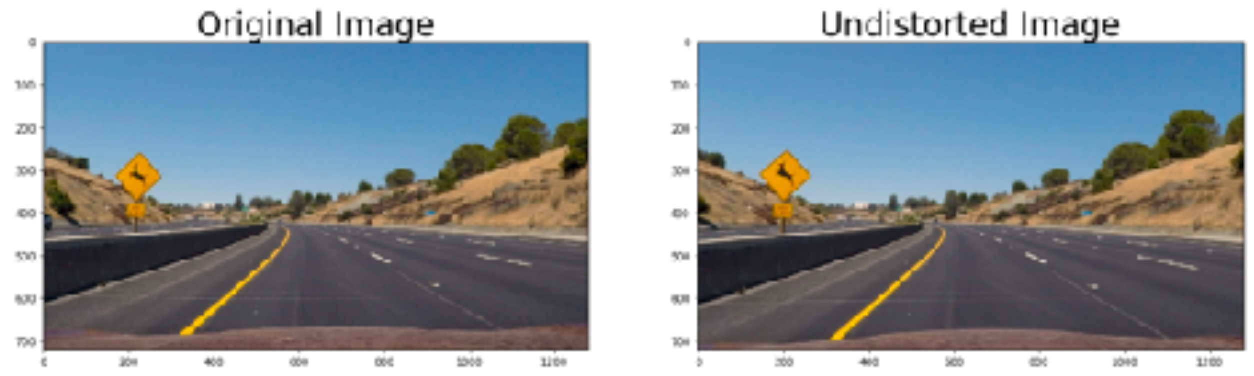


Pipeline

The following steps were developed on a set of test images and later applied to every image in the project video.

1. **Distortion Correction**

Using the camera calibration transformation, as described above, each the image was first undistorted using OpenCV's cv2.undistort method. Looking at the images below, it is difficult to see the exact results of the undistortion, but it can be seen that a transformation has been applied by looking closely at the hood of the car.

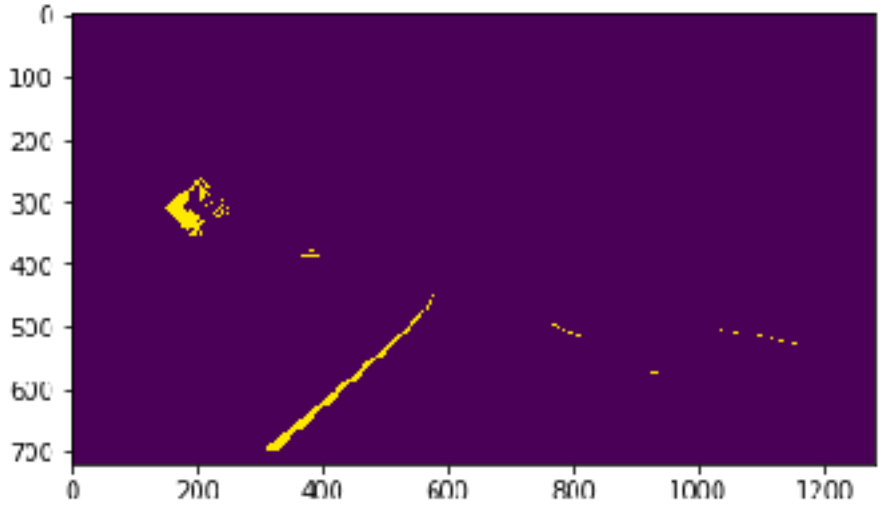


2. Thresholding

The thresholding steps were vital for pulling only the lane lines out of an otherwise noisy image. Much tuning and experimentation yielded an approach that uses two filtering techniques, as described below.

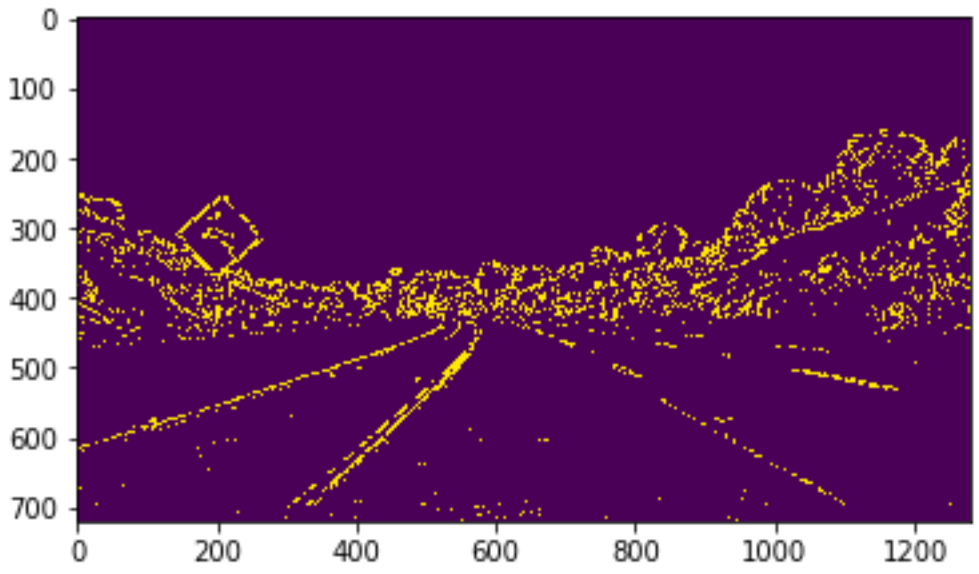
a. S-channel thresholding

After transforming the image into HSV colorspace, it was determined that the S (saturation) channel was the most useful for discerning the painted lane lines from other surfaces in the image. The S-channel was taken and filtered to a certain value range to yield images. All pixels that fell within a certain range were then converted to a binary representation. An example of the results of the S-channel thresholding is shown below.



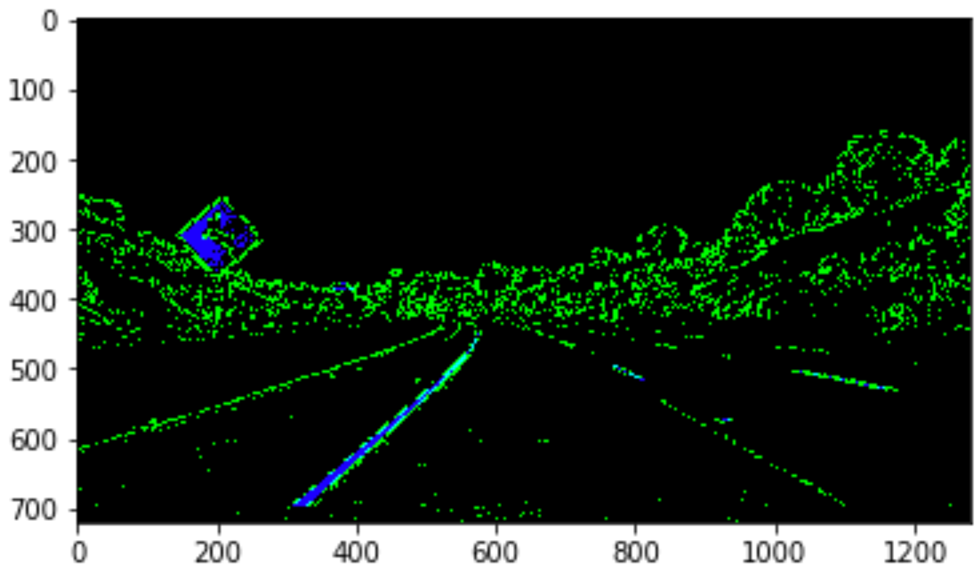
b. X-sobel thresholding

Although the S-channel thresholding worked well for nearly all video frames, it was necessary to include a second set of inputs in order to achieve good performance through the entire video. This was done using a x-directional Sobel filter, which detects edges from high pixel-value gradients in the x direction. An example result is shown below.



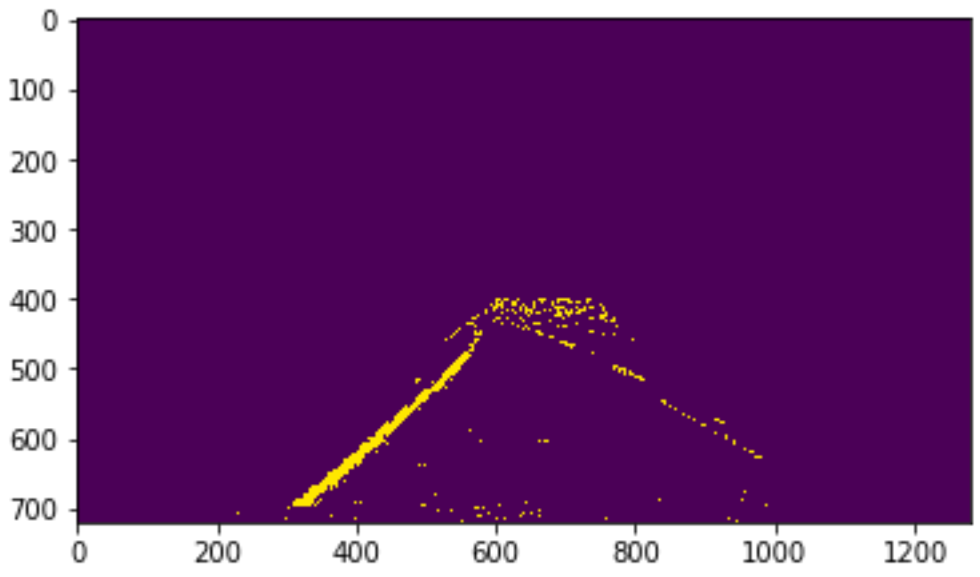
c. Combined thresholding

The binary images from the two thresholding methods were then combined into a single image for each frame, as shown below.



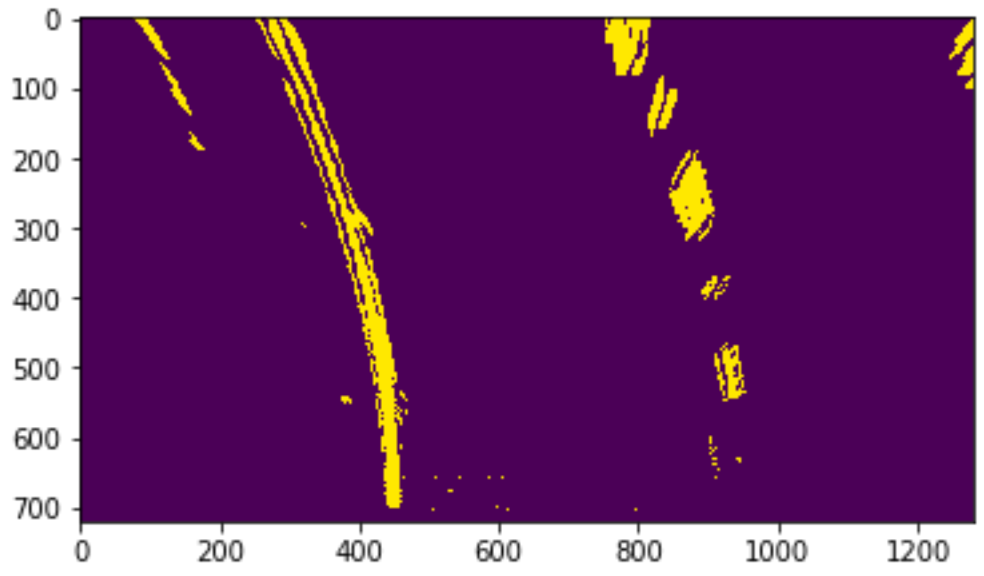
3. Cropping

Next, in order to remove noise and distracting edges that were not on the correct part of the road surface, the combined binary images were cropped to only included pixels from a trapezoidal area as illustrated below. This process required several iterations in order to determine an appropriate window.



4. **Perspective Transform**

Now that the images had been undistorted, filtered, and cropped, the road area was run through a perspective transform. The transform matrix was defined manually by iteratively selecting a trapezoid of points that would be transformed into a rectangle in ‘birds-eye-view’. However, once and acceptable transform was found on a single image, it could be successfully applied to an other images from the same calibrated camera.



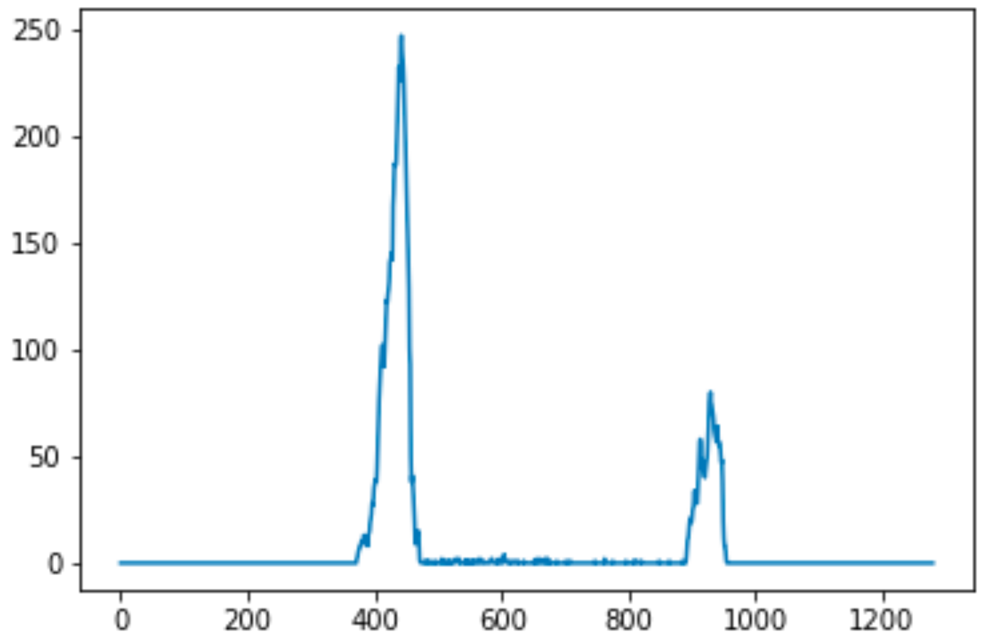
5. **Interface with Frame() Class**

In order to have good performance on an imperfect and variable road surface, it was necessary to have the pipeline remember certain information about previous video frames. This allowed the algorithm to search for lane lines only in a specified neighborhood of the lines that were detected in the previous image. It also allowed the pipeline to use averaging to ‘smooth’ the lines so that they do not jump around very much when one or several frames have yield poor detection performance. Lastly the fame number was tracked in order to only update the printed text once every 9 frames, so that the text could be more easily read by a human watching the video in real-time.

All of these attributes of the frames were held within a Frame() class that in initialized at the beginning of each new video.

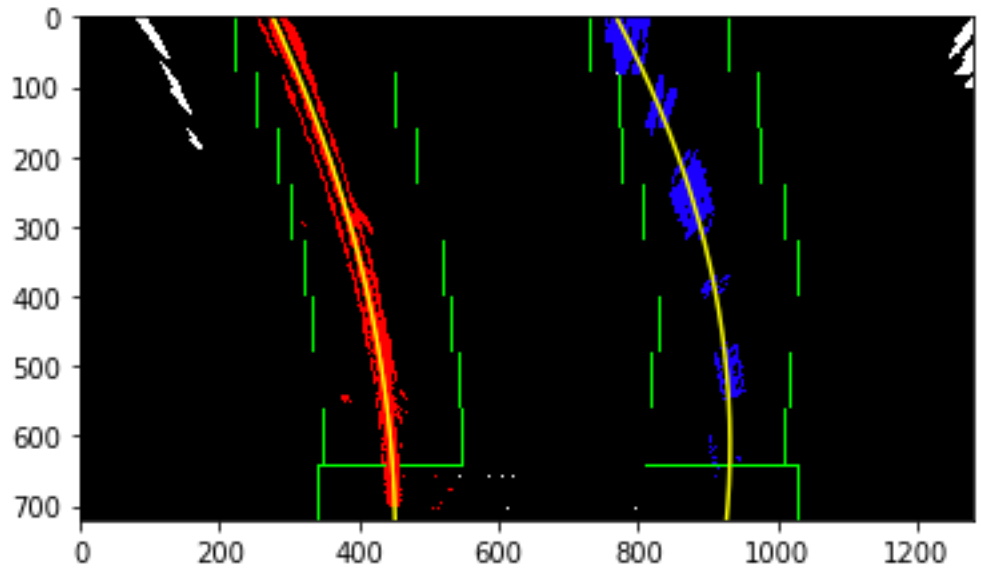
6. **Detect Lane Lines**

The lane lines were detected using a second-degree polynomial fit applied to the peaks of a sliding histogram of all activated pixels. After filtering, it was expected that the lane lines themselves should be the dominant features of any small y-window across the transformed image. The figure below shows an example of a histogram that has accurately identified exactly two distinct lane lines. Two methods were used to reach for lane lines across each transformed image.



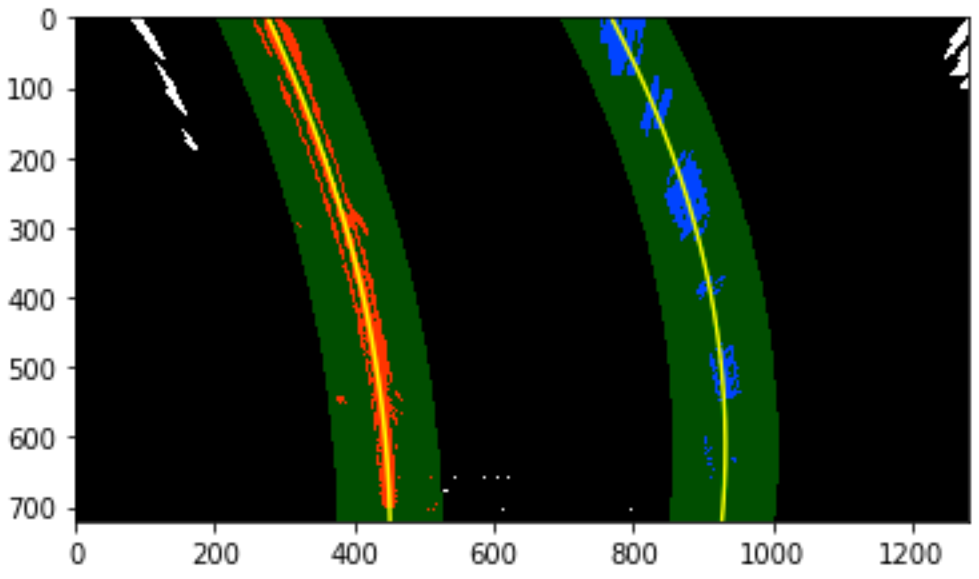
a. **Blind sliding window**

First a blind sliding window was used to search the entire image for lane lines. The image below shows the sliding window in green and the pixels that have been assigned to either histogram peak in red and blue. Pixels in white have not been assigned to either lane, and are being treated as noise.



b. Neighborhood search

Once a set of lines have been detected, the pipeline begins to search for subsequent lines only within a narrow neighborhood around the previous lines. The assumption is that an any normal driving situation, the lane lines should vary only slightly from moment to moment. The neighborhood search has two benefits. First, searching only a small area is faster and more efficient than searching the entire frame each time. Second, it allows any noise that it outside of the neighborhood to be immediately discarded, which results in better performance.



7. Smoothing

The Frame() class was used to average the polynomial fit from the last three line detections for each new frame. This helped the algorithm perform in the places where there were shadows and confounding edges.

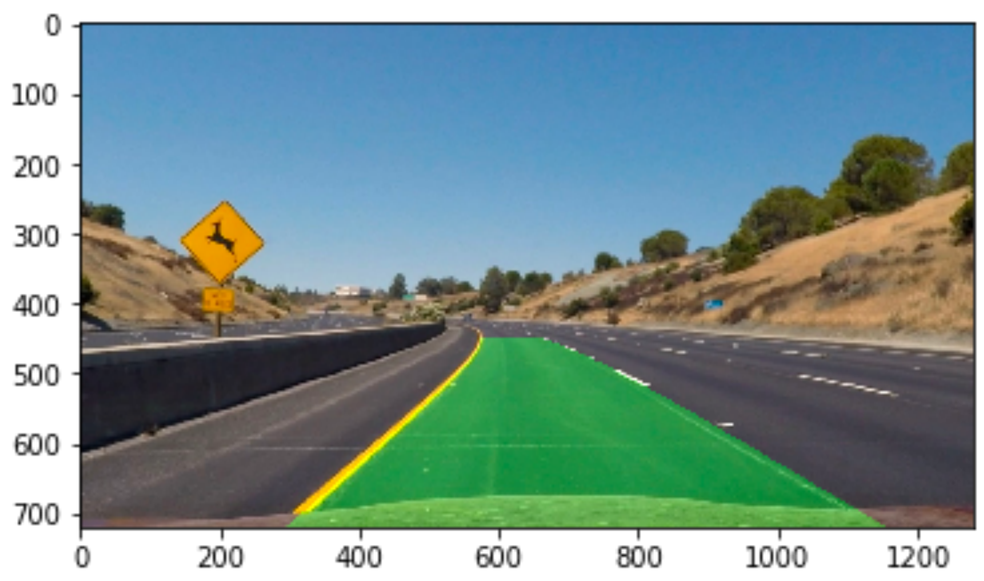
8. Determine Curvature and Lane-Center Offset

Using the polynomial fit for each frame, the lane curvature was calculated using basic calculus. The final curvature was taken as the average of the curvatures of each line.

The offset was calculated by determining the location of the midpoint between the two detected lines at the bottom of the frame. This location was compared to the midpoint of the frame and was scaled to covert from pixels to meters.

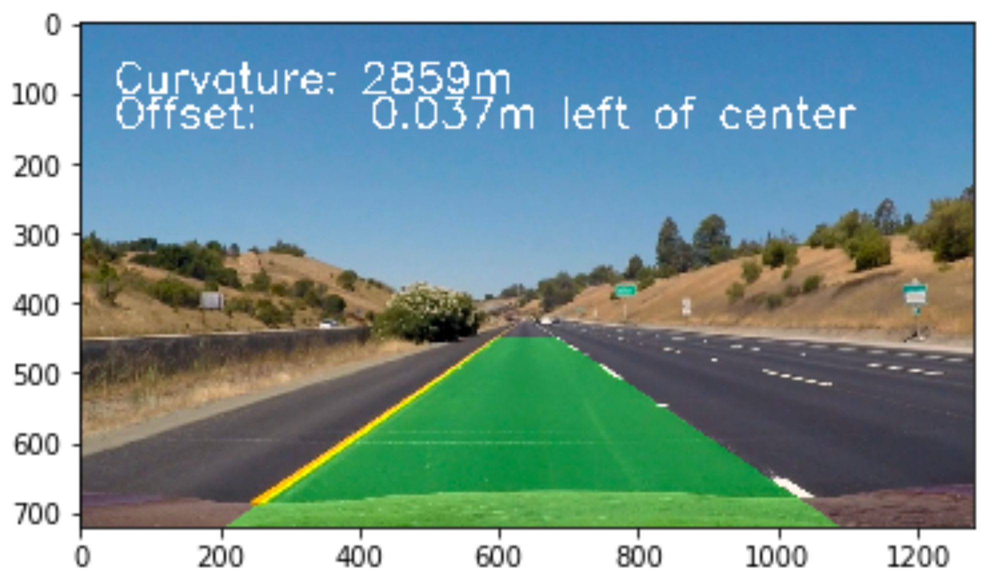
9. Draw Lane Lines

As shown below, untransformed lines were drawn on the original image, giving a green region to describe the driving area of the detected lane.



10. **Print Text on Image**

Lastly, the curvature and offset data was printed on each frame. This text was only updated once every 9 frames to improve readability.



Video and Discussion of Results

The final video was built by taking each image from the original video, processing it with the pipeline as describe above, and the compiling these images into a new video of the same format as the original. See the 'project video' file in the Output Videos folder.

The end result is quite good for this particular stretch of road. The pipeline performs very well on the black pavement sections. It has a little bit of trouble on the bridges where the pavement type changes and where there are shadows. However, the smoothing and the neighborhood search techniques help the pipeline stay on track during these tough spots, with only a little bit of visible fluctuations in the lane lines.

The results on the challenge videos were not as good. It seems that the pipeline has great difficulty when there are other strong line-like features on the road surface. A variety of techniques might be considered to address this issue. It might be possible to try additional variations of filtering in order to only pick up the lane lines and not other line-like features like cracks. However some attempts at this were made and the results were not promising.

A likely more productive approach would be to try to use more domain knowledge of the situation to disregard features that are not lane lines based on their relationships to previously detected lane lines. A first attempt at this was made by implementing the neighborhood search and the smoothing functionality. One could go further by putting constraints on lines such as that they must be roughly parallel or that they must be roughly a certain distance apart.

Another possible approach would be to meld this 'dumb' algorithm with some sort of machine learning technique so that the pipeline could be trained to determine the probability that a detected line is indeed the lane line that was sought.