README.md

Finding Lane Lines on the Road

Adam Tetelman

Project Introduction

The goal of this project was to create an image processing pipeline that is capable of taking images and/or video footage from a frontward facing car dash cam and accurately identify the lane lines.

The image processing pipeline used must be able to identify lane edges within the images. The pipeline must also be able to calculate a most likely location for the right and left lane line and overlay a solid line over the image indicating the lanes.

After completion of the coding portion a follow-up project summary will be written describing the technology used, the problem solving approach, the pipeline chosen, and reflections upon those design choices.

Image Processing For Identifying Lane Lines

Tools Summary

The tools used in the pipeline included the following:

- Contrast Increase: Cause brigher colors to appear brighter and darker colors to appear darker for more accurate analysis.
- Grayscale Converstion: Convert three layer RGB image to single layer grayscale image for noise reduction and easier/faster processing.
- Gaussian Blur: Uses a Gaussian Kernel to blur the image for noise reduction.
- Color Selection: Given a low and high color threshold will return an image containing only colors in the threshold.
- Polygon Region Selection: Given a set of vertices returns an image containing only values within the polygon. Used for analysing only areas of interest.
- Canny Edge Detection: OpenCV implementation of the Canny edge detection algorithm.
- Hough Transform (Line Detection): OpenCV implementation of the Hough Transform to identify likely lines in an image

Pipeline Summary

The Pipeline I implemented had 7 main steps. This was broken into two main parts, the edge detection segment and the lane detection segment.

Edge Detection

1. Convert to grayscale
2. Apply Gaussian blur
3. Color selection
4. Canny edge detection
5. Region of interest selection

Lane Detection

1. Draw lanes
2. Overlay lanes onto original image

In addition to those main steps I also played around with a few alternative image processing techniques and analysis steps. These were not included in the final pipeline but are detailed in the project notebook.

Edge Detection Pipeline

The edge detection segment is made up primarily of calls the external libraries. In this part of the pipeline we do two things. We first simplify the image and reduce noise (grayscale, blur, color selection). We then detect edges and output all detected lines (canny edge detection, region of interest).

This is the step with the most parameters to tune. As can be seen in the code example below, each step in the pipeline has several variables that can be used to modify the end result of the pipeline.

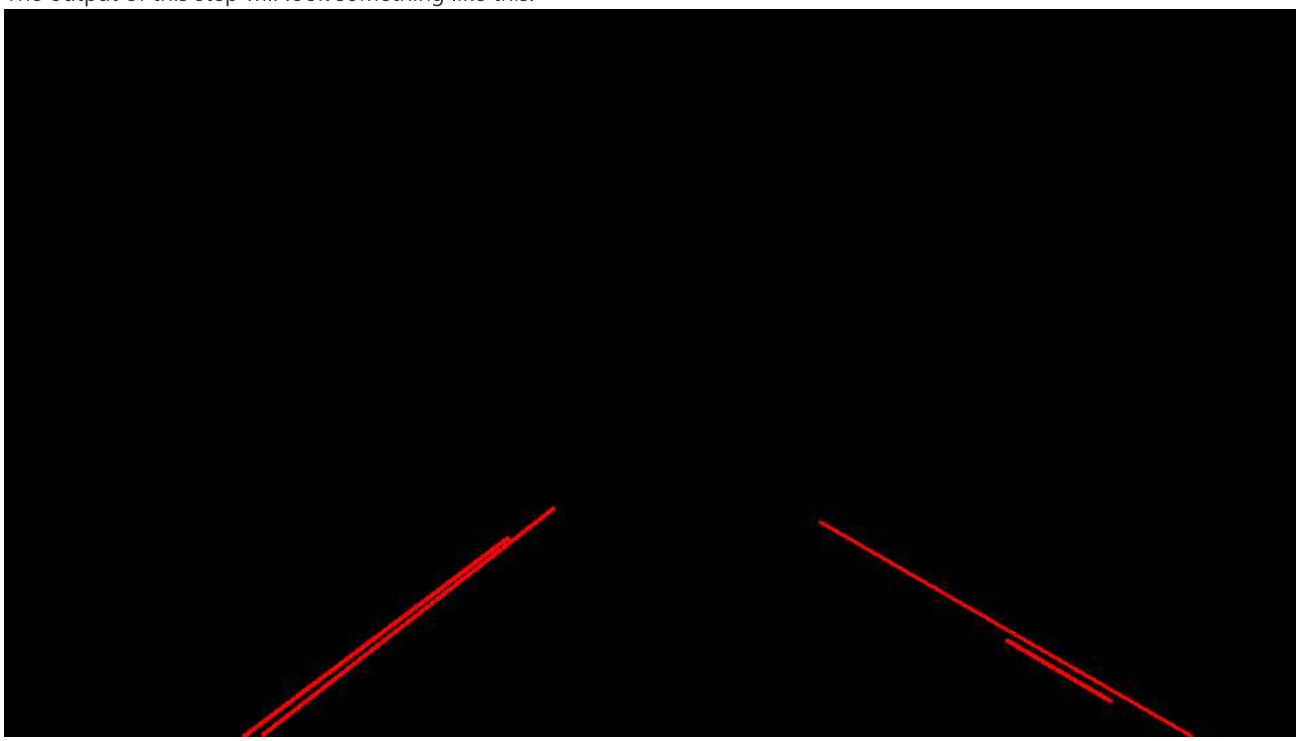
```
# Gaussian blur kernel
blur_kernel_size = 3

# Canny Edge detection
canny_ratio = 3
canny_low_threshold = 70
canny_high_threshold = canny_low_threshold * canny_ratio

# Color selection
color_low = 200
color_high = 255

# Region selection
poly_y = image.shape[0]
poly_x = image.shape[1]
polygon_vertices = ...
```

The output of this step will look something like this:



It is worth noting that the order of some of these steps can be changed. This could results in no changes, a more efficient pipeline, or slightly different end results.

Lane Detection

This segment of the pipeline uses the Hough transform along with much of my own algorithms.

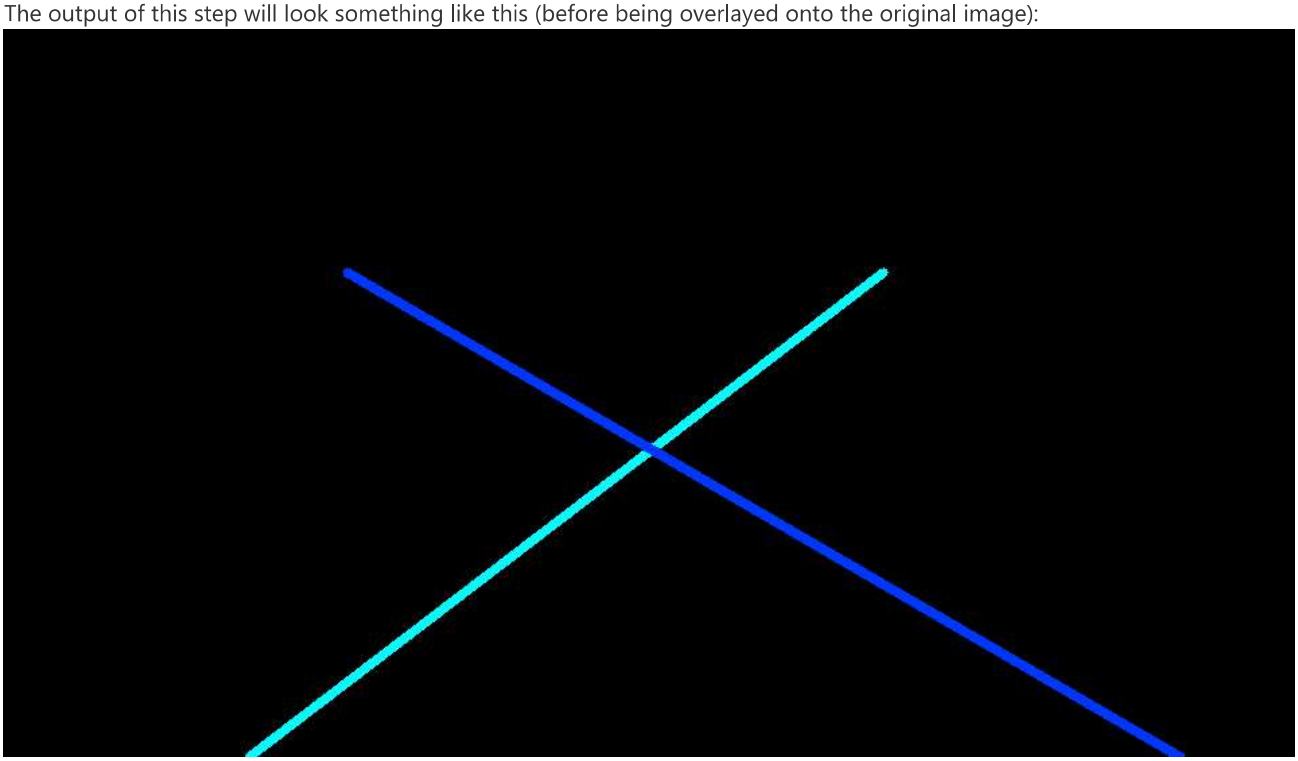
The previous steps of the pipeline results in an image that contains many points. I then take that image and feed it through the OpenCV HoughLinesP function. The output of this is a Numpy array of lines in [(x1,y2), (x2,y2)] format.

Given a list of lines, the next step is to determine which lines belong to the left and right lanes. I am able to accurately achieve this by calculating the slope of each lane.

After creating a list of left_lane_lines and right_lane_lines I am able to calculate the slope of each lane line and the center of each lane line.

The last step in drawing the lines is to use some math to calculate the (x,y) pairs. I manually define the y values to be the bottom of the image and a third from the top of the image. I am then able to use my calculated slope and calculated center to determine the other x values. The equestion used is Y1-Y2) = m(X1-X2).

The output of this step will look something like this (before being overlayed onto the original image):



The final step is simply to overlay this image with the original.

The tuning parameters were for the Hough transform and some slope validation:

```
# Define an invalid slope range
invalid_slope_low_max = .05
invalid_slope_low_min = -.05
invalid_slope_high_max = 100

# Hough Line detection
hough_rho = 2
hough_theta = np.pi/180
hough_threshold = 60
hough_min_line_len = 10
hough_max_line_gap = 100
```

There are several different ways this step could have been achieved and in the project notebook I have gone into a bit more depth on some of these options (linear regression).

Reflection

The pipeline I was able to put together runs very well. It is simply enough to explain in a few paragraphs, yet effective enough to highlight the lanes in the videos and images tested. I am happy with the tuning I was able to accomplish as well as the expandability of the pipeline itself.

That being said I would not want to ride in a car operating off of this algorithm. In many real world scenarios it would likely fail. Firstly, processing a 10 second video took 30 seconds. If this algorithm cannot effectively run in real time it is not useful, some optimizations and/or better hardware would be needed. Secondly, a car is often in situations with crowded roads, poor

weather, and hard to identify lanes; given a realistic situation in traffic I do not think this algorithm as it stands would be effective.

It would however do very well for a bulk of the driving I do, that being long hauls on minimaly populated highways, in good weather, on a sunny day. Given this sort of scenario I see no reason to think that the pipeline would do any worse than it did in the test videos.

Potential Shortcomings

Other potential shortcomings and bad scenarios:

- Dealing with accidents
- Dealing with inclement weather
- Dealing with construction
- Dealing with doubly painted or missing lines
- Dealing with dirt roads
- Dealing with merges
- Dealing with very sharp turns
- Dealing with non-standard lane colors
- Dealing with non-standard camera footage
- Dealing with heavy traffic
- Running in real time

Possible Improvements

There are numerous ways this could be improved. Here is a short list:

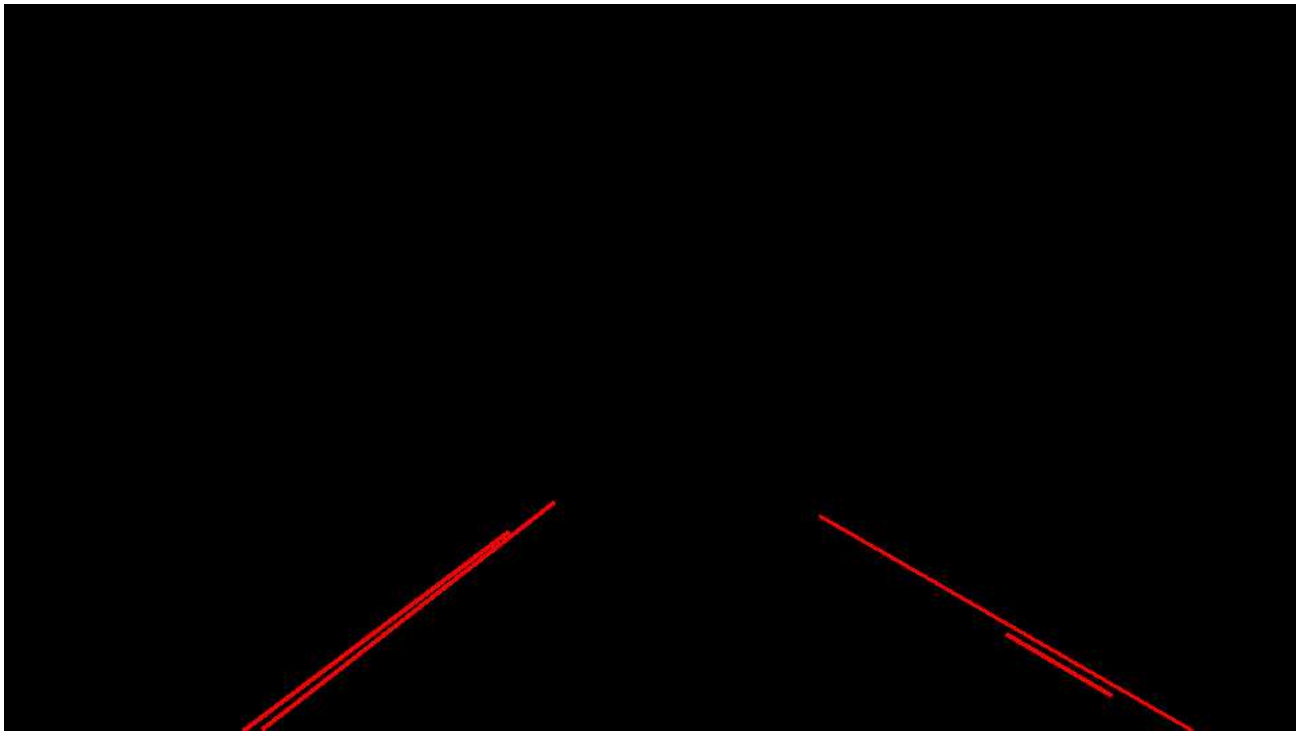
- Increase contrast for dealing with low light or high brightness scenarios
- Segment lanes using linear regression
- Segment lanes using a series of vectors rather than a single line
- Write code in something faster than python

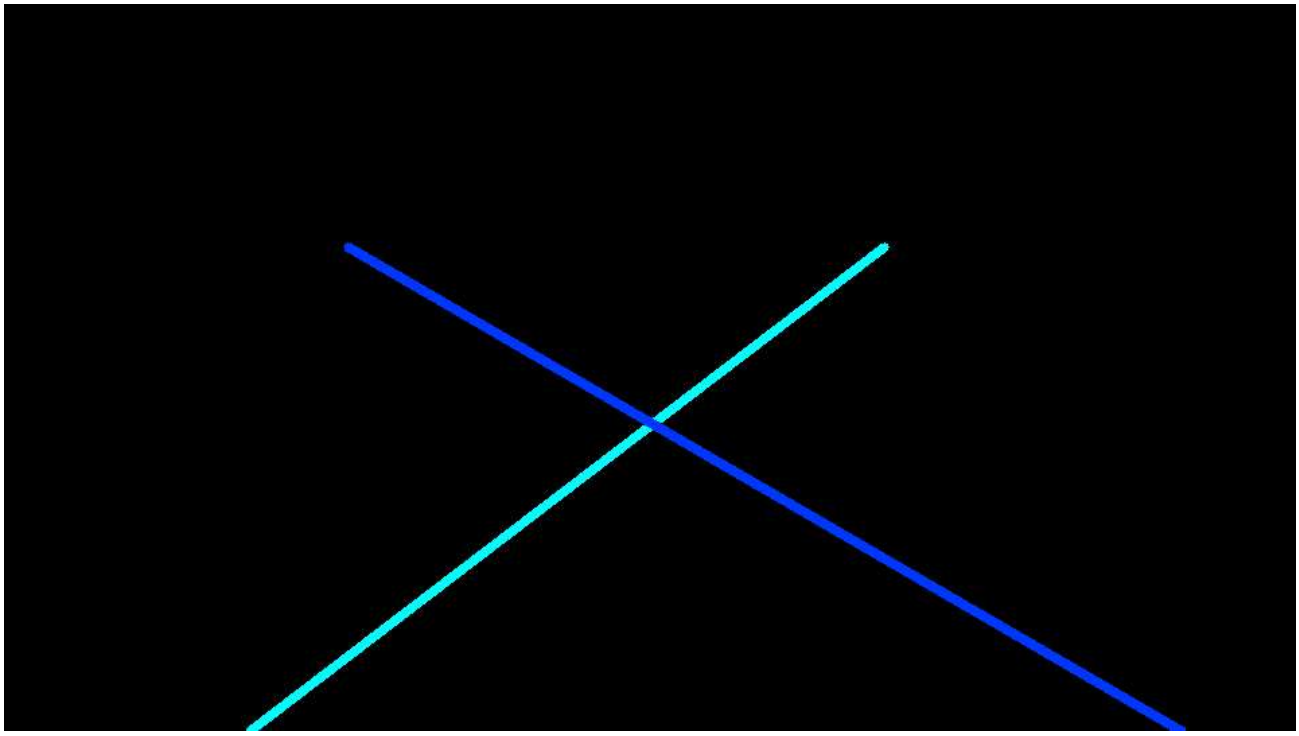
Results

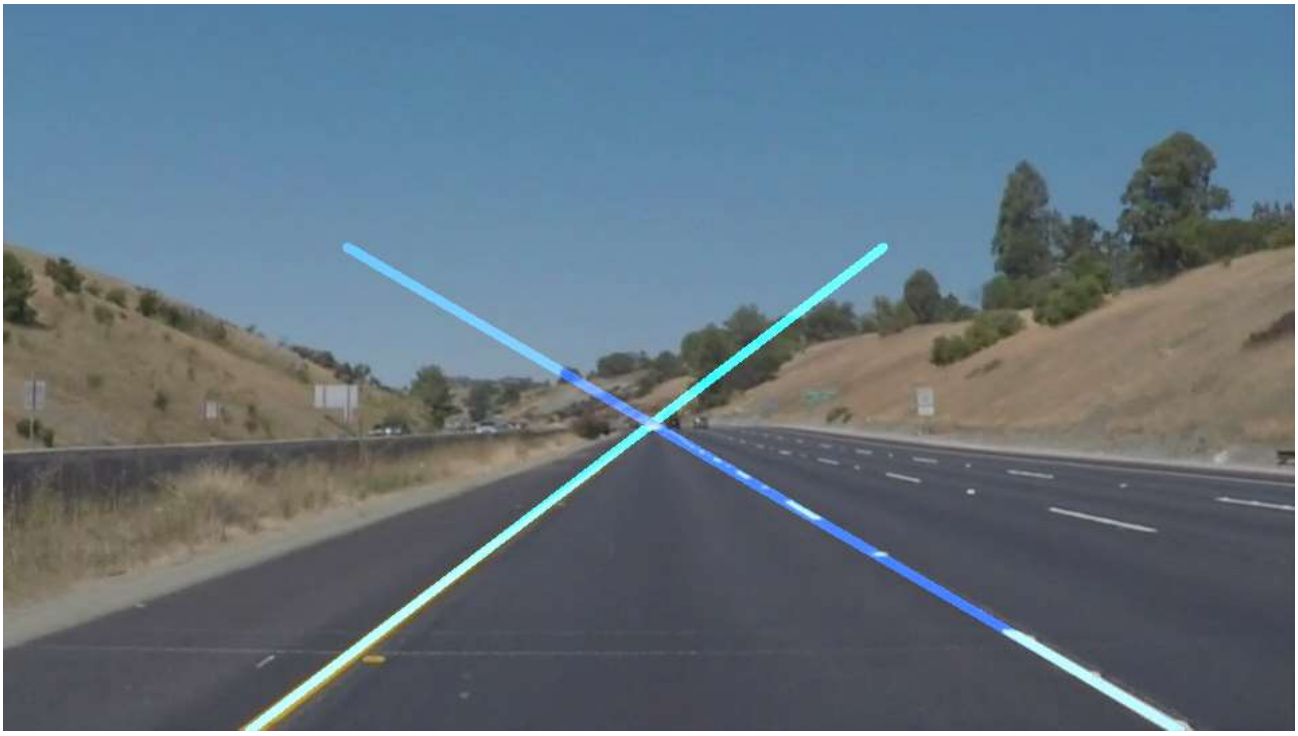
Final Videos

See results on Github.

Processing Steps







Raw Edge Detection Videos

See results on Github.

Test Images