Finding Lane Lines on the Road

Project Overview

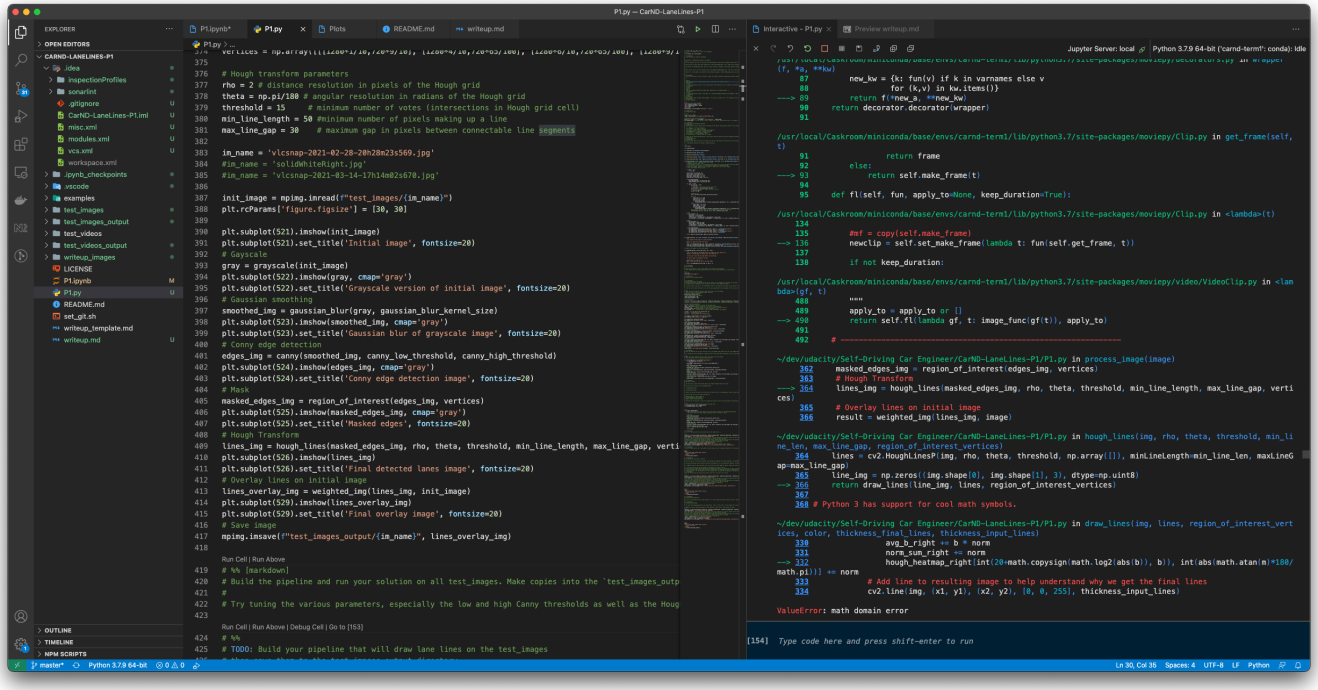
Finding Lane Lines on the Road

The goals / steps of this project are the following:

- Make a pipeline that finds lane lines on the road
- Reflect on your work in a written report

Tooling

To bring this project to an end, I installed the Conda environment locally. I used the online FAQ to fix the dependency resolution issues. I decided to use Visual Studio Code to import the project and converted the notebook to a python file to allow python debugging of my functions. This was very useful since I was getting mathematical errors during my testing and I needed to find where the issues were.



Once the project was finished, I just had to reconvert the python file to Jupyter notebook.

Reflection

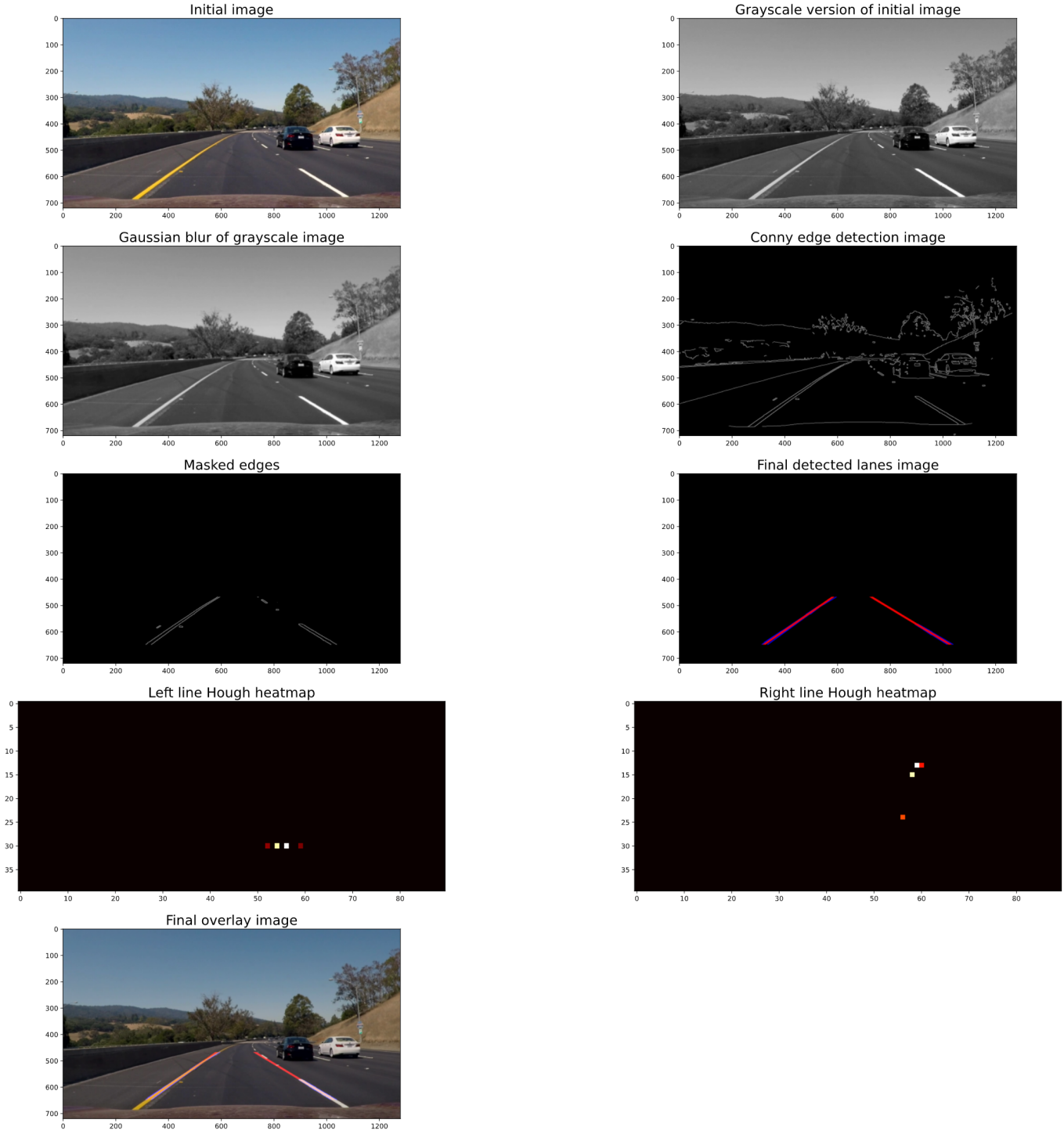
The Pipeline

My pipeline consisted of the following steps for each image:

1. Get the grayscale

2. Gaussian blur
3. Conny edge detection
4. Apply a mask to keep edges in the region of interest (tuned for each size of image/video)
5. Apply a Hough transform to find the lines
6. Compute the resulting 2 lines to overlay on the initial image

In order to visualize each step of the pipeline, I printed them in a figure. That way I was able to fully see the effect of each change of parameters on each step of the pipeline. See the figure below for the challenge video:



Once good parameters values were chosen for the initial set of images in the test_images folder and the 2 first videos, I modified the draw_lines function to draw 2 full lines on the left and right. This was the hardest

part of the project because of noise introduced by unwanted detected lines on the images. I tried 4 different approaches and decided to keep the last one:

Find average left and right lines with cartesian coordinates

See the draw_lines_orig() function.

While avoiding vertical lines, I performed the following steps for each list of Hough lines returned by the cv2.HoughLinesP() function:

> 1. Compute m & b in y = m*x+b formula of all lines
> 2. According to the m value, separate left and right lines
> 3. For left and right cases, take average of m and b (--> 2 average vectors of all left and right [m,b] vectors in Hough space)
> 4. After having iterated on all Hough lines, for left and right, compute x for y="bottom of image" and x for y="top of image"
> 5. Trace left and right resulting lines

This method yielded good results but I was getting errors due to divisions by zero when computing the resulting 2 lines since sometimes the average slope would be 0. I decided to use polar coordinates to circumvent the issue.

Find average left and right lines with polar coordinates

This method is the same as the above approach, but using polar coordinates. *See the draw_lines_polar() function.*

> 1. Compute theta & rho for all lines
> 2. Keep angle in range]0;180] since line(angle) == line(angle+180°)
> 3. Separate left and right lines
> 4. Take average of theta & rho (--> 2 average vectors of all left and right [theta,rho] vectors in Hough space)
> 5. Compute x for y="bottom of image" and x for y="top of image"
> 6. Trace line between those 2 points

I obtained wrong results with this approach, most likely due to wrong usage of trigonometry. I gave up using this approach and jumped on the third and final approach I considered.

Find weighted average left and right lines with cartesian coordinates

This method is the same as the previous cartesian approach, but using a weighted average method to give more importance to long Hough lines during the computation of the resulting left and right lines. I also switched x and y when finding the m and b values to compute the average lines end points coordinates more easily. *See the draw_lines() function.*

While avoiding horizontal lines, I performed the following steps for each list of Hough lines returned by the cv2.HoughLinesP() function:

> 1. Compute m & b in x = m*y+b formula of all lines
> 2. According to the m value, separate left and right lines

> 3. For left and right cases, take weighted average of m and b using the vector norm as weight (--> 2 average vectors of all left and right [m,b] vectors in Hough space)
> 4. After having iterated on all Hough lines, for left and right, compute x for y="bottom of image" and x for y="top of image"
> 5. Trace left and right resulting lines

This approach reduced a bit the noise brough by undesired detected Hough lines.

Find median left and right lines with cartesian coordinates

To remove the strong effect that the outliers had on my resulting lines, I switched to using a median instead of the average for m and b in the previous method. That way outliers were ignored. I obtained a smoother line overlay that reflected better the ground lines. *See the draw_lines_median() function.*

While avoiding horizontal lines, I performed the following steps for each list of Hough lines returned by the cv2.HoughLinesP() function:

> 1. Compute m & b in x = m*y+b formula of all lines
> 2. According to the m value, separate left and right lines and append [m,b] values to left or right array
> 3. After having iterated on all Hough lines, compute left and right median m and b values
> 4. Compute x for y="bottom of image" and x for y="top of image"
> 5. Trace left and right resulting lines

I decided to stop here since I assumed that the following lessons would provide more advanced tools to further reduce noise.

Potential Shortcomings with the Current Pipeline

- Right now the car is always aligned by the lanes on the ground. If the car finds itself unaligned at some point, the algorithm to detect left and right lanes might not work.
- The current algorithm would only apply to highways, not smaller roads or crossings etc..
- Worn out lanes on the ground would be an issue.
- The pipeline does not correctly recognize the lines on the ground in some video images.

Possible Improvements to the Pipeline

2 Neighbouring Points in Hough Space

The third draw_line method still sometimes gave me erroneous left and right lines. It could be possible to better isolate the lanes on the ground from the noise because after Hough transformation each lane results in 2 parallel lines. In Hough space, this means 2 points close to each other with approximately the same m/theta value. If we could somehow extract those 2 points for the left side and right side, then we could get more precise final left and right overlay lines.

With a snapshot from the challenge video, I plotted a quick heatmap of the generated left and right Hough lines in the pipeline to better visualize Hough lines in Hough space for a problematic image (see pipeline figure). The vertical axis is the b value in logarithmic scale, the horizontal axis is the absolute value of the

angle of the line in degrees. I did offset the b axis to be able to display on the image where the coordinates starts at pixel 0:

```
hough_heatmap_left[int(20+math.copysign(math.log2(abs(b)), b)),
int(abs(math.atan(m)*180/math.pi))] += norm
hough_heatmap_right[int(20+math.copysign(math.log2(abs(b)), b)),
int(abs(math.atan(m)*180/math.pi))] += norm
```

I didn't investigate this further as I think better techniques will be discussed later.

Use Previously Computed Lines

We could use the resulting lines in the previous video images to better compute the lines in the current image by excluding the input lines where the m coefficient is too different from the resulting lines in the previous pictures. That would also smoothen the rendering of the resulting lines on the video. However, since the resulting lines will be used as input to the steering wheel we might not want to smoothen too much to keep a good responsiveness of the vehicle.

Further Adjust Parameters based on Image Size

Some steps of the pipeline might beneficiate from adjusting the parameters depending of the size of the image input, like the gaussian blur.

Code Optimisation

E.g. extract the generation of the mask since no need to regenerate for every image.