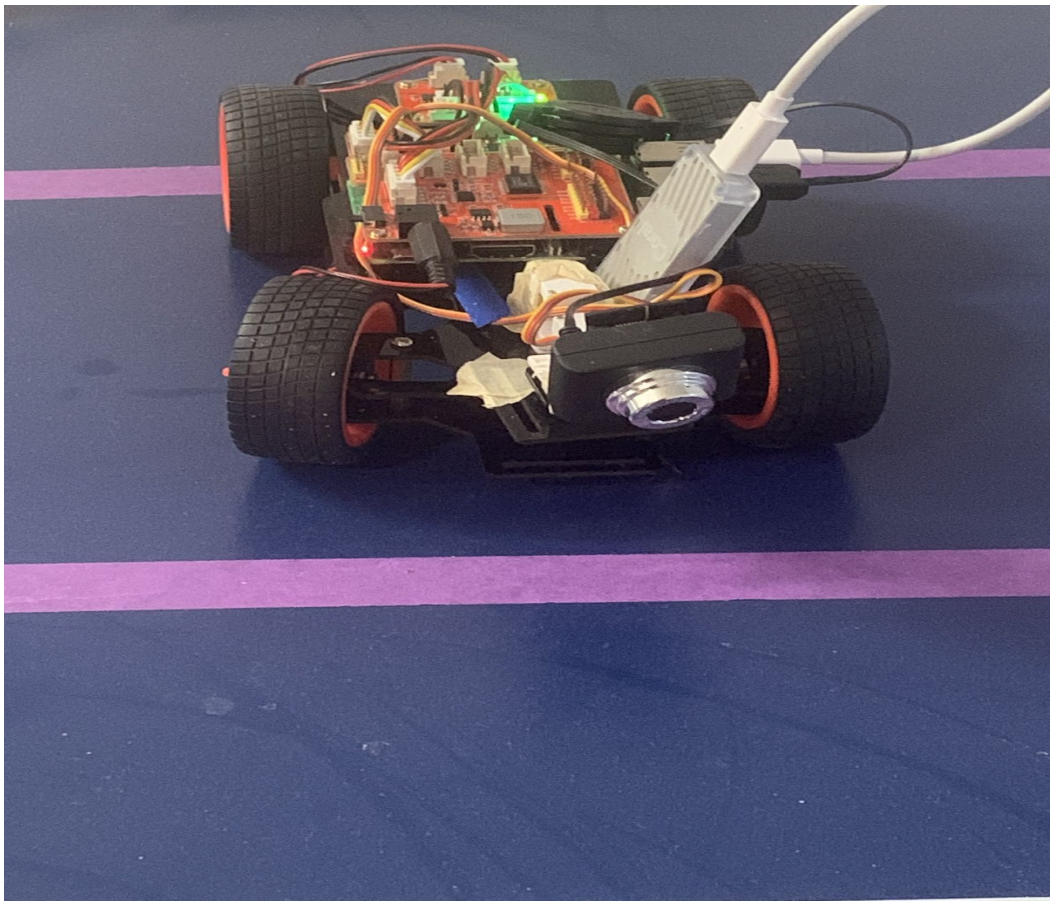
Introduction:

Hi! In this document, I hope to explain the process and algorithmic concepts I used in my implementation for my Raspberry Pi Car Project. In particular, for computing a steering angle based on images received from the car's central camera, I made heavy use of concepts relating to computer vision and image processing. The primary goal of my project was to develop a small-scaled autonomous vehicle that could travel alongside the sidewalks of my neighborhood within a certain radius. The information in this document contains the transformations applied to one of the thousands of frames in a video captured by our car. By applying these necessary transformations, a steering angle for what direction a car should take can be computed.

Here is a picture of my raspberry pi car:



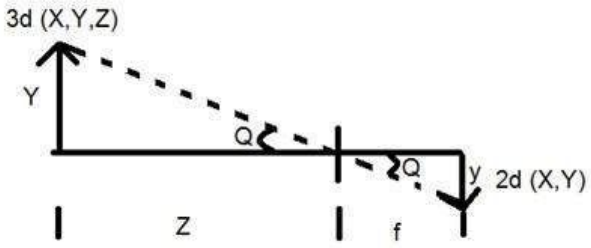
It contains a TPU, a processing unit that speeds up the process of deep learning, a raspberry pi system, a device containing all necessary code that controls the car, circuit mappings, four plastic wheels, and a small central camera that will be used to capture images and record videos; these pieces of information will be essential in transforming our car to an autonomous one.

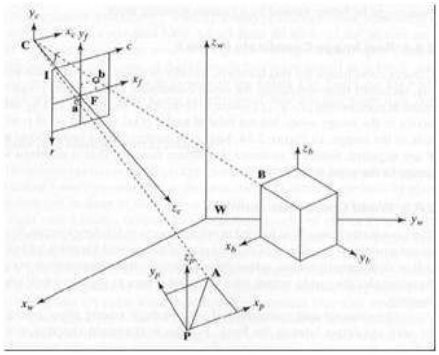
Applying Perspective Transform

The main goal of this project is to ensure that the car can travel alongside the sidewalks of my neighborhood. Consequently, we must first find the boundaries of the sidewalk, or the areas where grass exists. By doing this, our car can determine what is safe passing and what is an obstruction or what is considered off road areas.

Here is an example of a picture taken from the perspective of my car:

Detecting the boundaries from this perspective proved to be challenging, so I looked to change the perspective of the images collected through my car's camera. One way to do this is through a method called **perspective transformation.** The essential idea of perspective transformation is to convert a 3d image into a 2d bird's eye perspective one.





- A basic image deriving the essential components of perspective transformation.
- Light is captured at an infinitesimally small point (where an imaginary normal plane intersects with surface of the image, at distance z)
- At this point, light will deflect off the surface of the image.
    - An inverted bird's eye perspective image can be formed.

Applying perspective transformation from the previous image, we obtain:

With a 2-D bird's eye view, we are able to now effectively find distinctive boundaries between the sidewalk and grass.

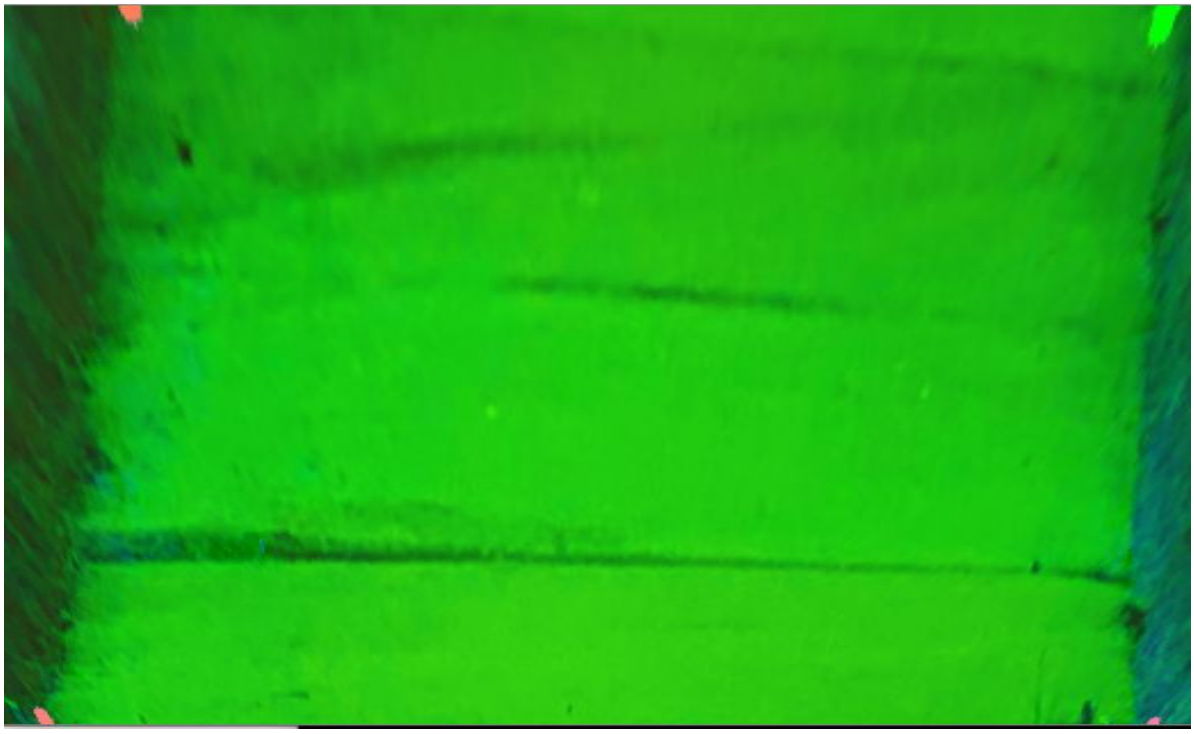
Applying HSL Filter

Although we ourselves are able to more easily determine the boundaries between a sidewalk and the grass, our autonomous car does not have that capability. Consequently, we can apply the gradient and color thresholds. By doing this, we can segment our image based on varying degrees in colors. This is, to again, ensure that our car understands that any area containing sidewalks is safe passing.

These processes are applicable because each pixel in an image represents an integer value that determines the gradient of color at that point in the image. Thus, the program for our car can effectively determine if a certain pixel falls within a desired integer value.

We must first apply a color threshold, the principle of segmenting an image based on different color spaces. To meet this end, the image must be converted to an HSL (Hue, Saturation, Lightness) from RGB. The motivation for this is because HSL proved to be more effective in masking areas that were not the color of the sidewalk when comparing the two color scales.

Applying an HSL filter to the image we obtain:

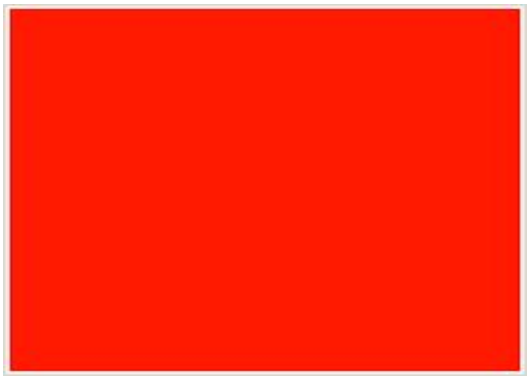


At first, this filter doesn't seem to do much; however, it proves to be especially useful in the next portion of this image processing process.

Applying a Color Threshold

With our image in HSL color format, our car can more easily determine colors that are unique to the sidewalk and colors that are not. This is useful because we can more easily apply a mask to the image. A mask is a kernel used to filter an image and can remove areas of the image that do not contain a certain color. Consequently, we can ignore pixels containing colors in the image that are unwanted, removing noise from the image.

An example of a mask being applied to an image can be seen below:



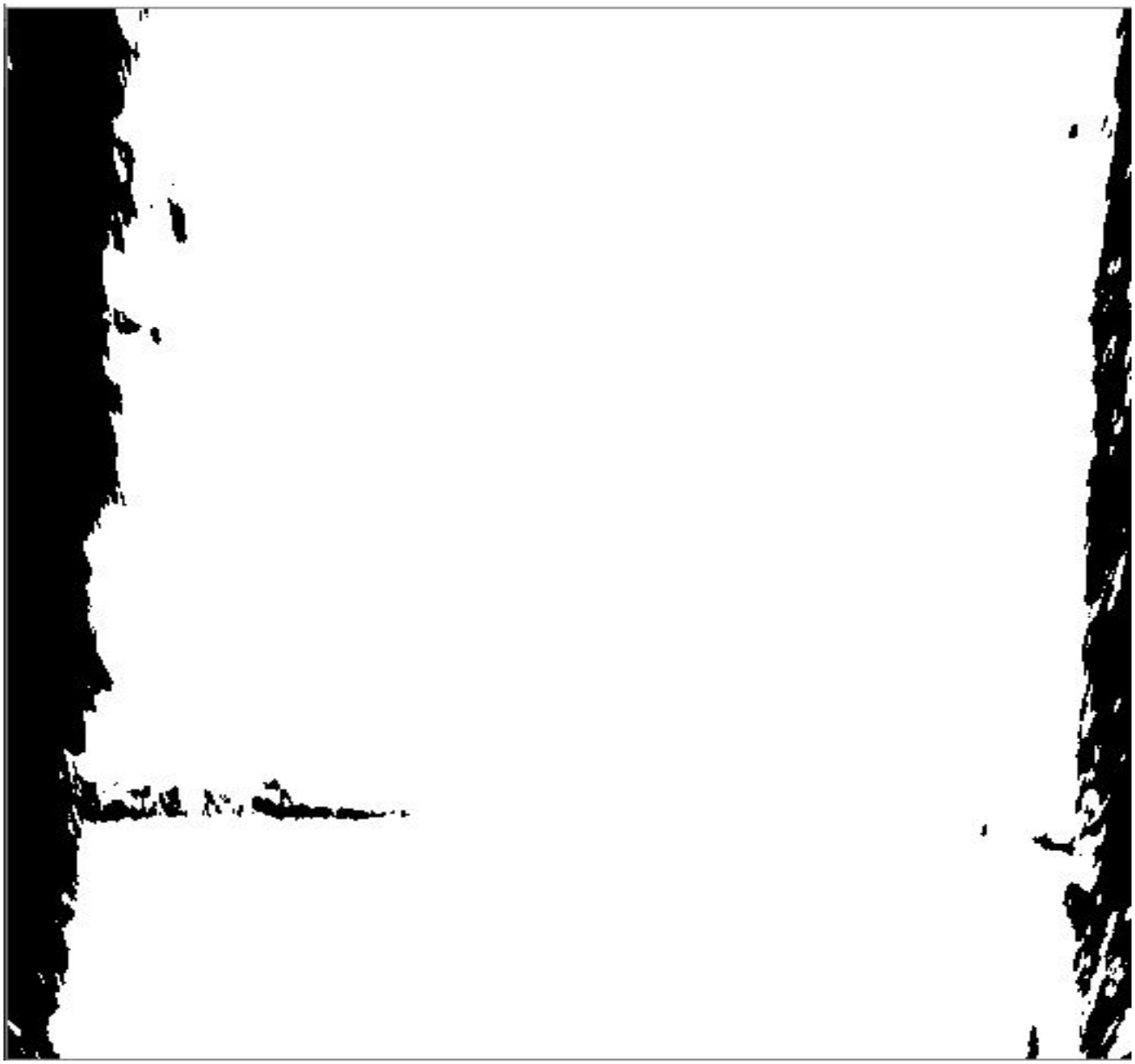
Before Mask

- Red has an RGB value of (0,0,225)
- If a mask that only includes includes pixels that contain rgb values ranging from (0,0,70) to (50,50,125) is applied to an image, all red areas from that image will be omitted

After mask is applied

- After the mask is applied, all pixels values that do not fall within the allotted mask range will be omitted from the picture, or turned black (RGB value of (0,0,0))
- Consequently, we obtain a binary image (an image containing only two colors)

By applying a mask ((I was able to determine pixel values for the mask through empirical testing) that ignores the colors (in the HSL filter) that are not unique to the sidewalk, we obtain:



The area highlighted in white represents the areas of sidewalk, while the areas in black represent areas that are not part of the sidewalk. By applying this mask based on a color threshold, we have more definable areas of sidewalk within our image, which again is useful for the next step in this image processing procedure. Moreover, we have transformed our image into a binary one, an image that contains only two colors - in this case, black and white.

Applying a Gradient Threshold

Coupled with a color threshold, a gradient threshold is necessary in determining the boundaries, or edges that exist in an image. An edge can be defined (for our purposes) as a line that separates the white areas and black areas of an image.

The most widely used algorithm for determining edges is the Canny Algorithm. Most importantly, however, is the implementation of the sobel kernel within the Canny Algorithm. A kernel is essentially a small matrix of values, and here is an example of 3x3 sobel kernels.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

G_x kernel that measures horizontal change G_y kernel that measures vertical change

As previously mentioned, each pixel in an image highlights an integer value representing a certain color. By multiplying these two kernels across all 3x3 regions in the matrix of pixels in our image, we can determine areas of extreme differences between two pixels; in other words, areas where there is a sudden change in color between pixels.

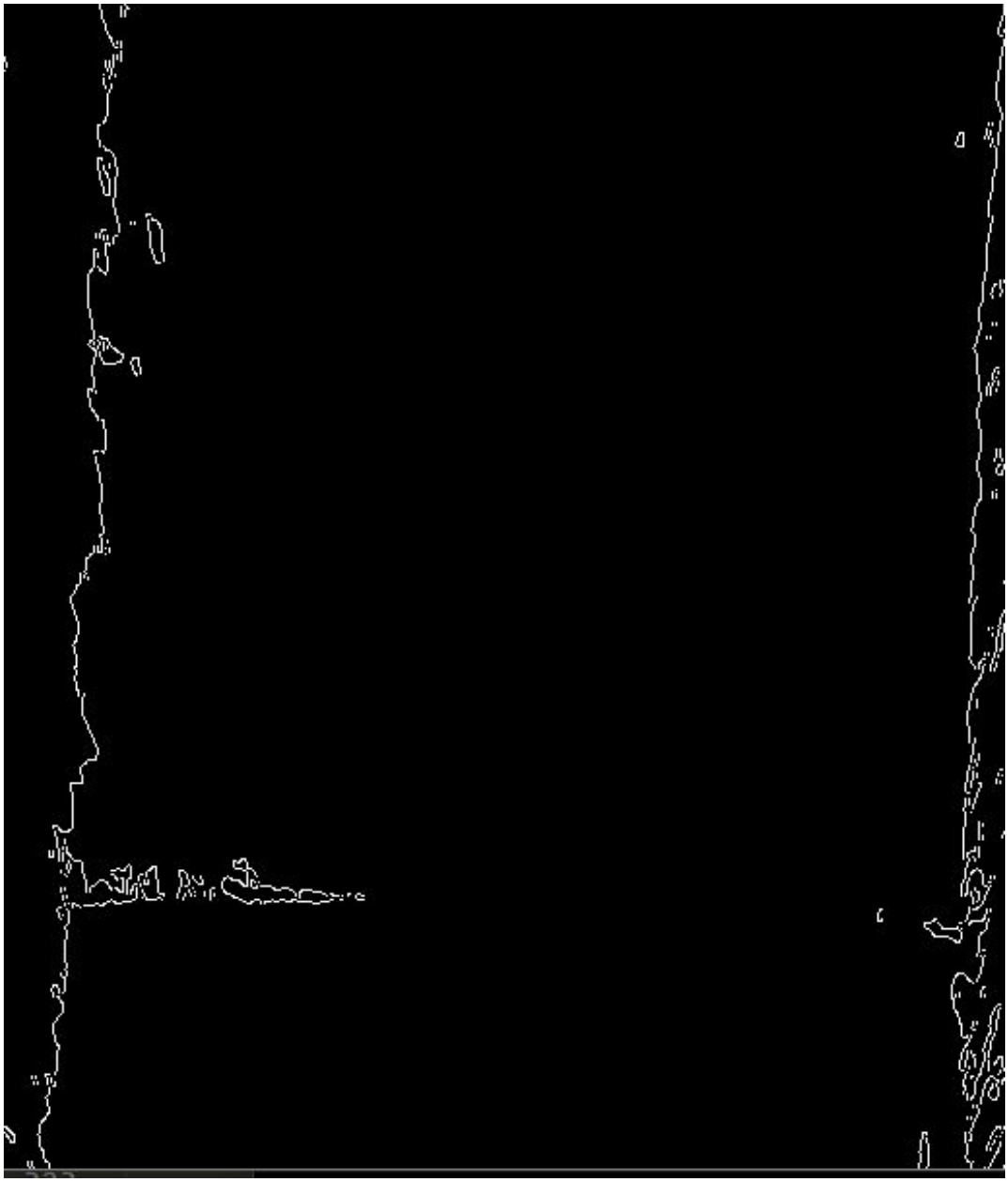
The total value of a gradient at a 3x3 region can be determined by applying a variant of the pythagorean theorem:

$$G = \sqrt{G_x^2 + G_y^2}$$

- Where G represents the gradient a 3x3 pixel region of an image

We can set an arbitrary threshold T, as a value that a gradient at a certain 3x3 region must exceed to be considered an area where an edge can be drawn. For instance, if G is set to 20, and T is set to 15, then the area G can be considered an area where an edge, in this case a boundary that exists between a sidewalk and grass. Likewise, if G is set to 10, then no edge can be drawn at this specified region. 3x3 Again, the optimal value for T was determined through empirical testing, and edges were subsequently drawn based on the image.

Applying the Canny Edge Detection Algorithm to the image, we obtain:



Where the white lines represent edges that exist between the binary image obtained in the prior step of this image processing procedure. More importantly, they can be classified as edges that exist between the sidewalk captured in an image and the surrounding grass strands. Yet, there are a lot of edges that create extra noise, and some can barely be considered edges. Consequently, this requires the use of another algorithm to filter out these unwanted edges.

Hough Line Transform

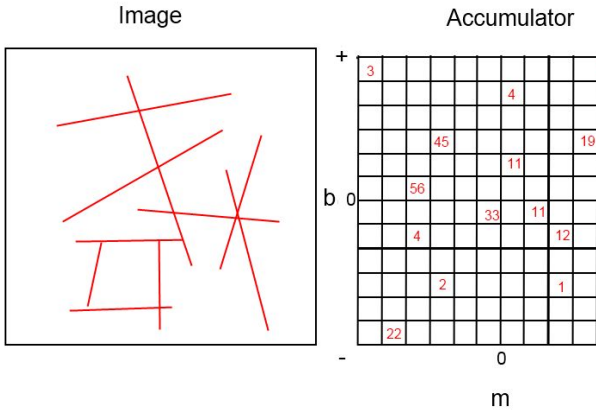
Although we were able to find the edges that exist in our image, which again serve as a way to provide boundaries to the car, some of these edges create unnecessary noise in the image. Fortunately, the Hough Line Transform algorithm serves as a remedy to this problem. Let's motivate this discussion by providing certain parameters to a Hough Line.

(Points can be defined as any pixel that is considered part of an edge).

- minVotes - Minimum number of points (or votes) that a line must intersect to be considered a line
- minDist - Minimum distance between any two points on a line to be deemed a line
- minLen - Minimum distance a line must be to be considered an edge
- maxGapPoints - Maximum distance that can be between two points before it is not an edge

With these parameters in mind, the Hough Line Transform is an algorithm that finds any line that exists within an image and passes through a point; the line is considered an edge if it passes the parameters described above. Key components of the algorithm are described below:

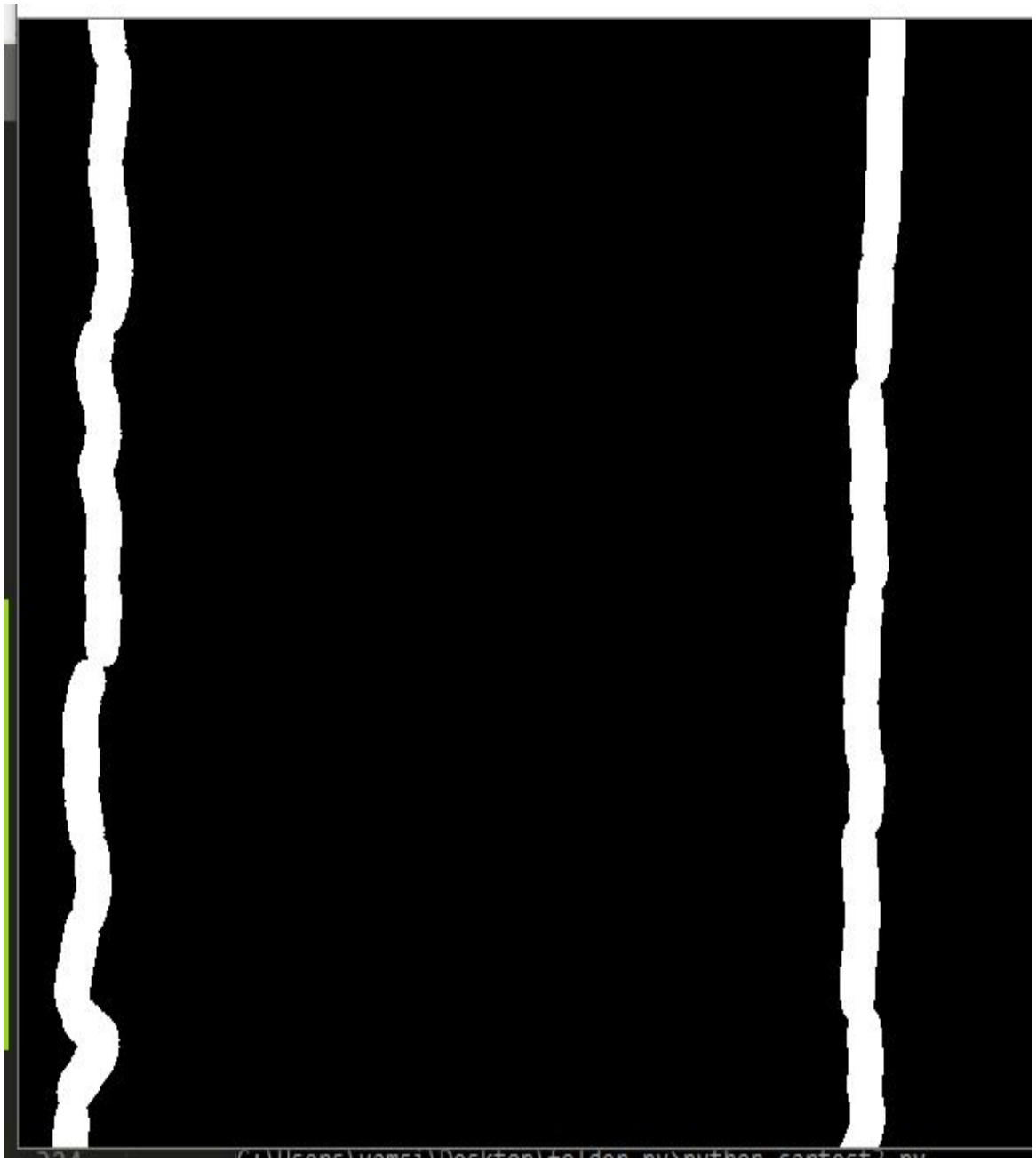
- Any linear equation can be expressed as y = mx +b
- Each (x,y) representing a pixel has a certain number of lines passing through it.
- The values of m and b that describe a line are stored in an accumulator array.
- For each m and b in an accumulator array, a vote is given to the line if it passes through a point.
- If a line meets all the parameters described in the first section, it is considered an edge, otherwise the edge that was previously drawn is removed from the image.



- To the left shows a graph of potential lines and an accumulator array that indicates how many votes, or points a line described by parameters m and b, pass through.
- If any line fails to meet the parameters established in the first section, then it is omitted from an image.

Through this filtration process, any unnecessarily small or rigid edges are removed from the image, allowing for a better understanding of where edges truly exist in an edge.

Applying the Hough Line Transform Algorithm to our image, we obtain: (Note: lines that were considered had their thickness increased for the purposes of the next algorithm in this process)

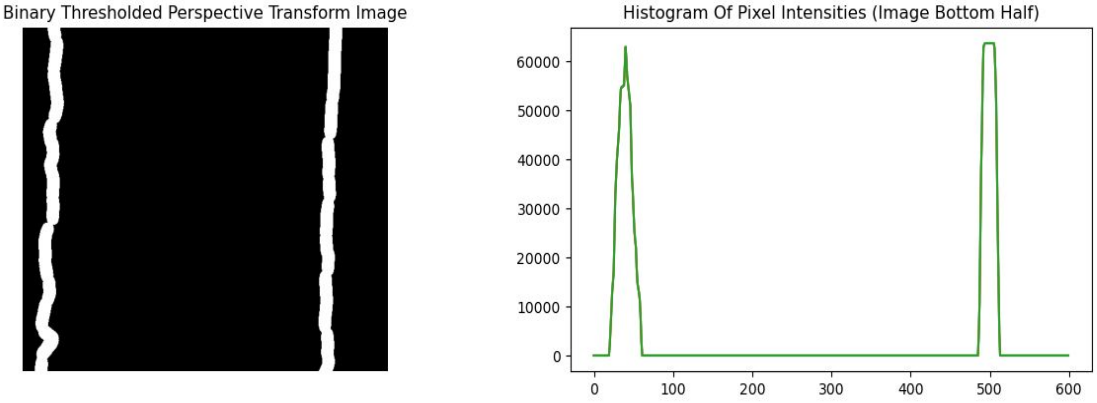


By removing all small, rigid edges that previously existed in our image, we are able to obtain two concrete lines that describe the boundary between the sidewalk and the surrounding grass. However, these lines don't represent anything tangible to the car; it doesn't form a barrier that tells the car to automatically avoid the two lines.

Creating a Histogram Describing Pixel Intensities

To combat the mentioned problem, we can highlight a general region in which the car has safe passing (areas of sidewalk).

By creating a histogram with an x-axis relating to the distance (in pixels) of the image, and the y axis representing pixel intensities, or the number of white pixels within a certain distance (x-axis) can be determined. The binary image should yield a bimodal histogram, with each of the peaks representing a large detected edge. Consequently, we can determine a concrete set of x-values as to where a part of a boundary may exist, represented by a set of white pixels.



- Above displays an image containing the binary image created from the previous step, and a histogram that displays the peaks for pixel intensities
- With this information, we now have a starting origin for where the lanes originate in regards to their x coordinate value on the binary origin.
- As expected, the bimodal graph shows where both boundaries exist between the sidewalk and grass.

Sliding Windows Algorithm

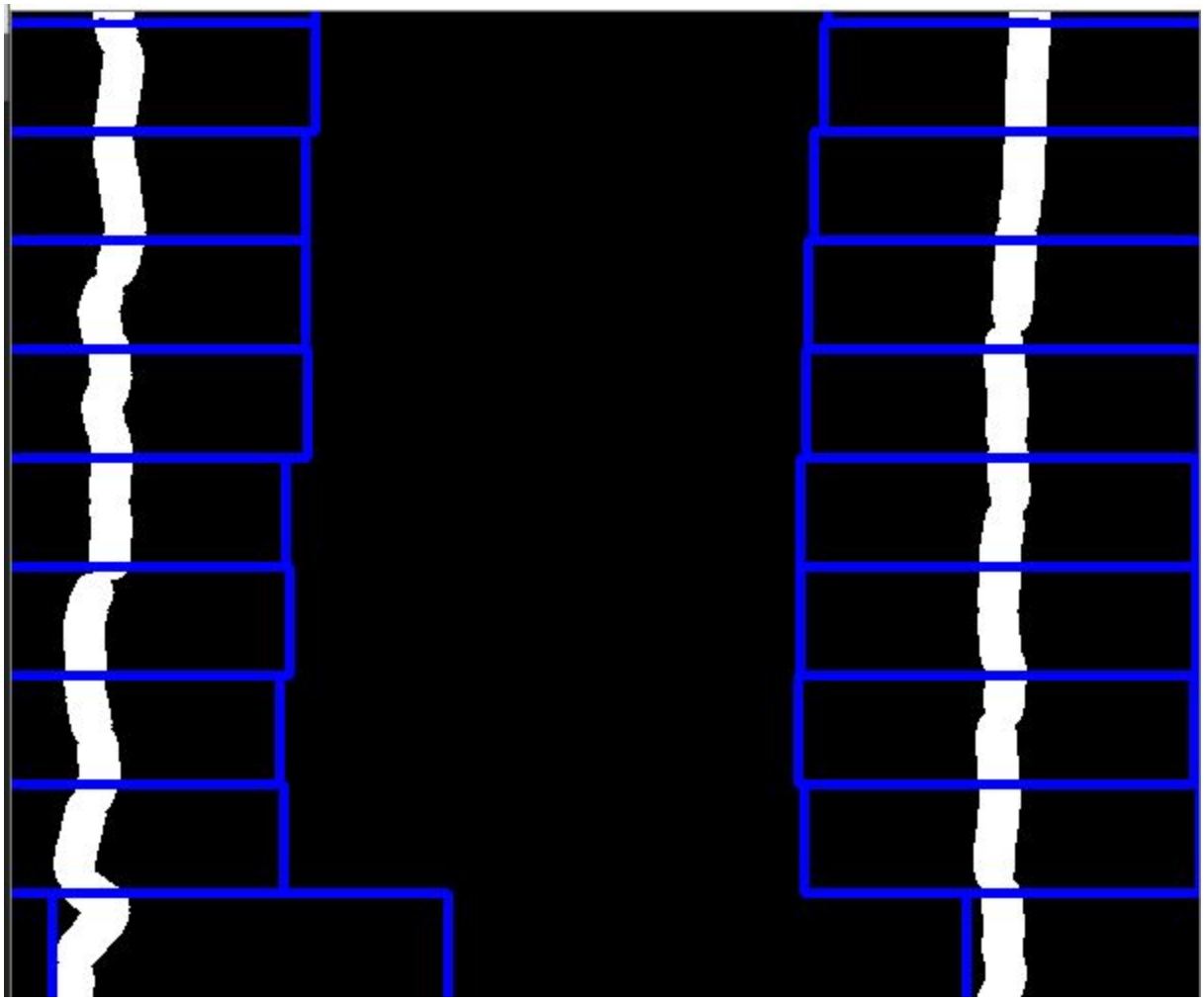
With a baseline starting value obtained for both the left and right edges from the histogram, we can begin to fit a polynomial of best fit for the boundaries. This will ultimately allow the car to understand a general region of area that is safe for passing, and will ultimately serve a purpose in determining a steering angle based on the shape, curvature and length of the fitted polynomial.

To fit the polynomial, however, we must find a set of points that can accommodate the polynomial and exist on the lane edges created from previous steps.

The Sliding Windows Algorithm serves as an effective medium in finding points to fit a polynomial to. Starting from the initial position computed from the pixel intensities histogram for both peaks, a blue window is placed. Subsequently, the window measures how many white pixels can be found within its rectangular boundaries. If the number of white pixels crosses an established threshold, the following box will shift its horizontal position to match the average location of the white pixels accommodated by the previous box; if they do not, then the new box will start in the same latitude position as the previous box.

Consequently, the boxes will be fitted in such a way that they will account for a majority of the white pixels. The centers of each of the windows are stored in an array and will be used later to fit a 2nd degree polynomial, creating a curve that will encompass a majority of the lane edges.

Applying the sliding window algorithm, we get:

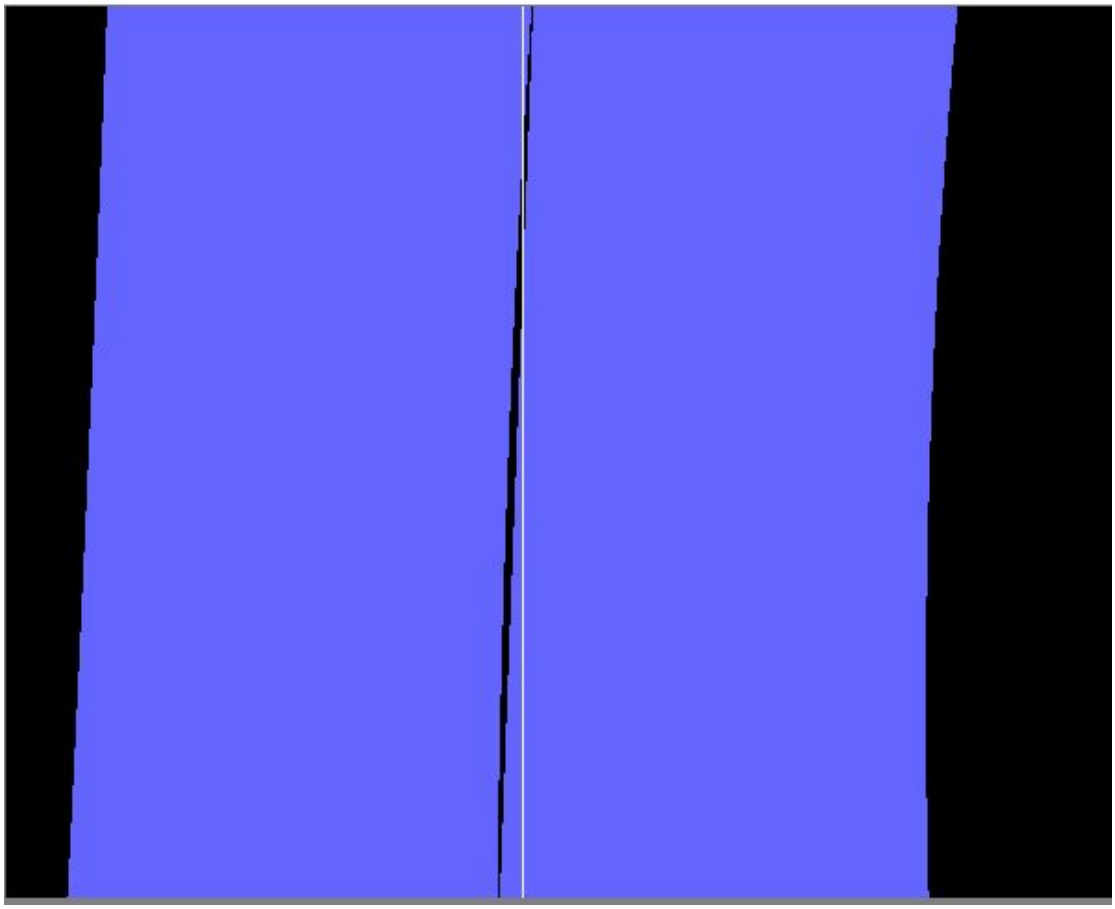


Fitting Polynomials

With these arrays of points found for each white pixelated edge, we can now fit a polynomial to accommodate for these points. 2nd degree polynomials seemed to fit the edges the most out of the sample of pictures collected, so using the numpy function np.polyfit(), we fit a 2nd degree polynomial for both edges.

With the fitted polynomials, we can fill in the area in between the fitted polynomials, along with a central curve that represents the average in terms of location and shape between the two fitted polynomials. Moreover, we can draw a line through the center of the image, indicating the car's relative position (more accurately the camera's relative position) within the sidewalk; this will be essential in computing the required steering angle.

Fitting the polynomials, filling in the area in between, and drawing a central curve and a vertical line through the center of the image, we obtain:



We can subsequently compute the distance in pixels between the bottom part of the black and white lines and apply a multiplier (determined from empirical testing)to find a steering angle; the greater the distance, the greater the turn the car must turn to stay in the center of the sidewalk.

Citations:

N. Arshad, K. Moon, S. Park, and J. Kim, "Lane Detection with Moving Vehicles Using Color Information," in World Congress on Engineering and Computer Science, 2011.

J. Canny, "A computational approach to edge detection," IEEE Transactions on pattern analysis and machine intelligence, pp. 679-698, 1986.

“Canny Edge Detection.” *OpenCV*, docs.opencv.org/master/da/d22/tutorial_py_canny.html.

V. Leavers, "Which Hough transform?", CVGIP: Image Understanding, vol. 58, pp. 250-264, 1993.

Narasimhan, Harini. “Image Processing with Opencv -Part 1.” *Medium*, Medium, 30 May 2019, medium.com/@harininarasimhan123/image-processing-with-opencv-part-1-8ff79e798a62.

Nguyen, Paul, and Michael Slutskiy. "Vision System for Autonomous Navigation."

D. Schreiber, B. Alefs, M. Clabian, "Single camera lane detection and tracking", Intelligent Transportation Systems 2005. Proceedings. 2005 IEEE, pp. 302-307, 2005.

Wan, Shuai & Yang, Fuzheng & He, Mingyi. (2008). Gradient-threshold Edge Detection based on Perceptually Adaptive Threshold Selection. 999 - 1002. 10.1109/ICIEA.2008.4582665.

B. Yu and A. K. Jain, "Lane boundary detection using a multiresolution hough transform," in Image Processing, 1997. Proceedings., International Conference on, 1997, pp. 748-751.