Laser Tracker Technical Manual

Developed By: Alex Oh, Ryan Taylor, Arianna Santiago

Table of Contents

Section I - Overview of Laser Follower **3**
A. Github Repository 3
B. UML Class Diagram 3
C. Runner Class 3
D. Context Diagram 4
E. Data Flow Diagram 5
F. Threading and Managing Shared Memory 5

Section II - Tracker **7**
A. Camera, Hardware, and Format 7
B. Capturing and Initializing the Image 7
C. Applying Appropriate Filters 7
D. Terminating the Sequence 8

Section III - Distance Finder **9**
A. Approach and Motivation 9
B. Modeling Forward Distance from Laser 9
C. Modeling Horizontal Distance from Laser 10
D. Further Potential Improvements 14

Section IV - Motor Control **15**
A. Conceptual Program Flow 15
B. Modeling Base Speed 15
C. Modeling the Left and Right Motor Speed 16

Section V - Motor Logger **18**
A. Purpose 18
B. Implementation 18
C. Observations and Optimizations 18

Section VI - Retracer **19**
A. Initial Design 19
B. Log Compression 19
C. Recalling the Log 19
D. Future Improvements 19

Section I - Overview of Laser Follower

A. Github Repository

A public GitHub repository with code is available at the following link. The repository can be cloned through git.

https://github.com/basicallyAlexOh/LaserTracker.git

B. UML Class Diagram

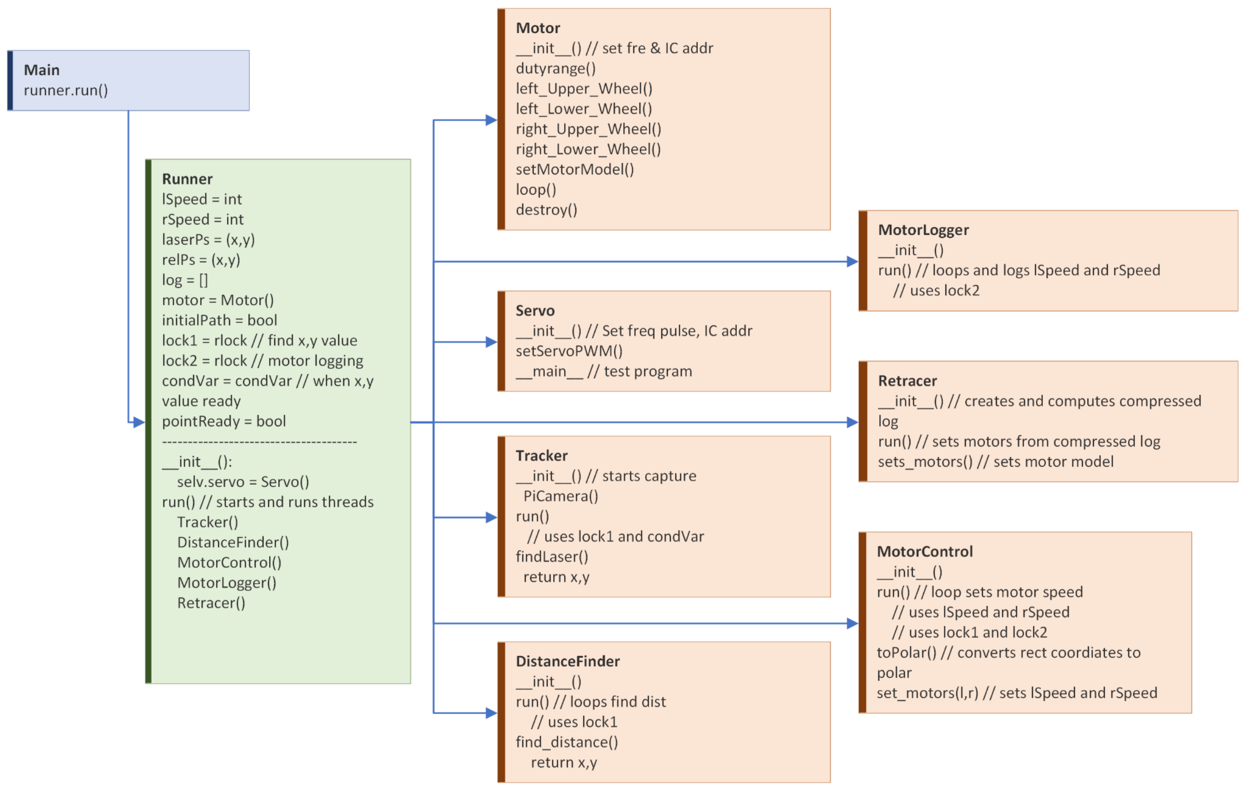

Figure 1-1: Class Diagram

C. Runner Class

The Runner Class is the main core of the program responsible for calling, managing, and threads and “static” variables available to other classes. The Runner also includes two locks and a single condition variable.

lock1 is responsible for protecting the Laser Coordinates (laserPos) and the Relative Distance (relPos) variables. The condition variable is used to correctly notify and the DistanceFinder to run after the Tracker finds the laser. lock2 is responsible for protecting the motor speeds (lSpeed, rSpeed).

Other static variables include:

- lSpeed, rSpeed: the left and right speeds of the motors respectively
- laserPos: the (x,y) position of the laser on the camera image *(See Tracker for more detail)*
- relPos: the position of the car relative to the laser in the format of (forward/back, left/right)
- initialPath: holds a boolean value that runs while the laser is visible, indicating that the first path is being drawn
- pointReady: holds a boolean value that works with the condition variable for checking if the point is ready to be processed in DistanceFinder.

The runner has two main responsibilities. The first is to initialize the Camera, Servo, and Motor objects and set them to their default values. The servo should position the camera head to the correct position. The second responsibility is to manage threads and call the threads. There are four threads that are started simultaneously: Tracker, DistanceFinder, MotorControl, and MotorLogger. The runner is responsible for joining the threads after the laser is turned off, then starting a Retracer thread after 10 seconds.

D. Context Diagram

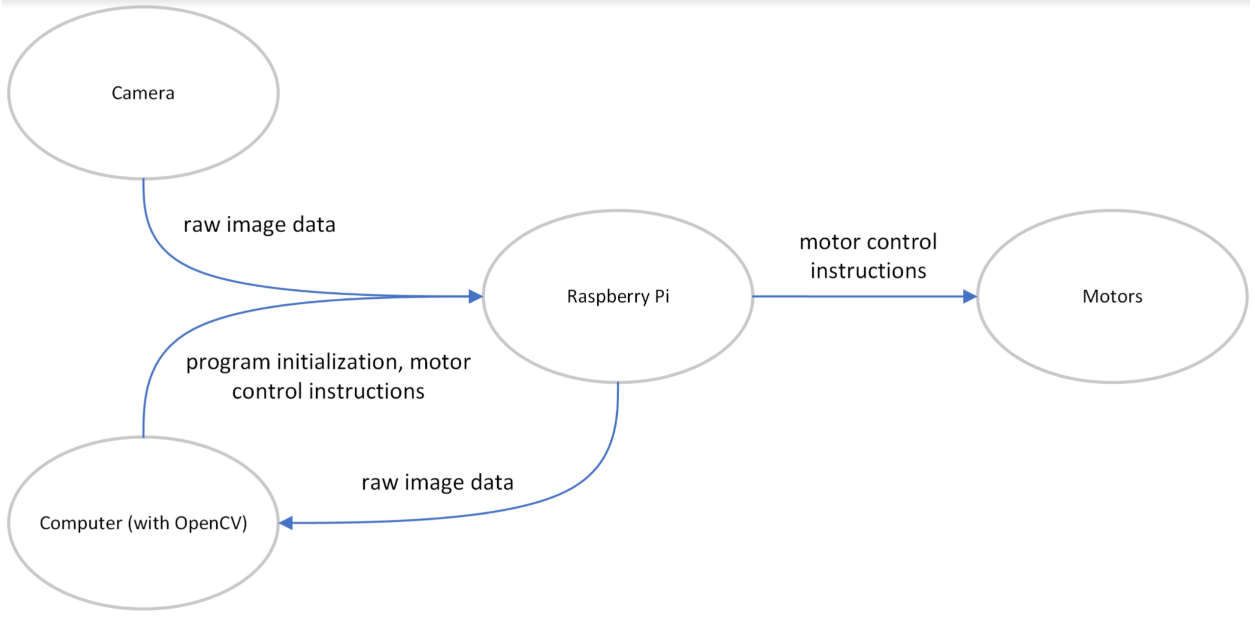
Camera
raw image data
Raspberry Pi
motor control instructions
Motors
program initialization, motor control instructions
raw image data
Computer (with OpenCV)

Figure 1-2: Context Diagram

E. Data Flow Diagram

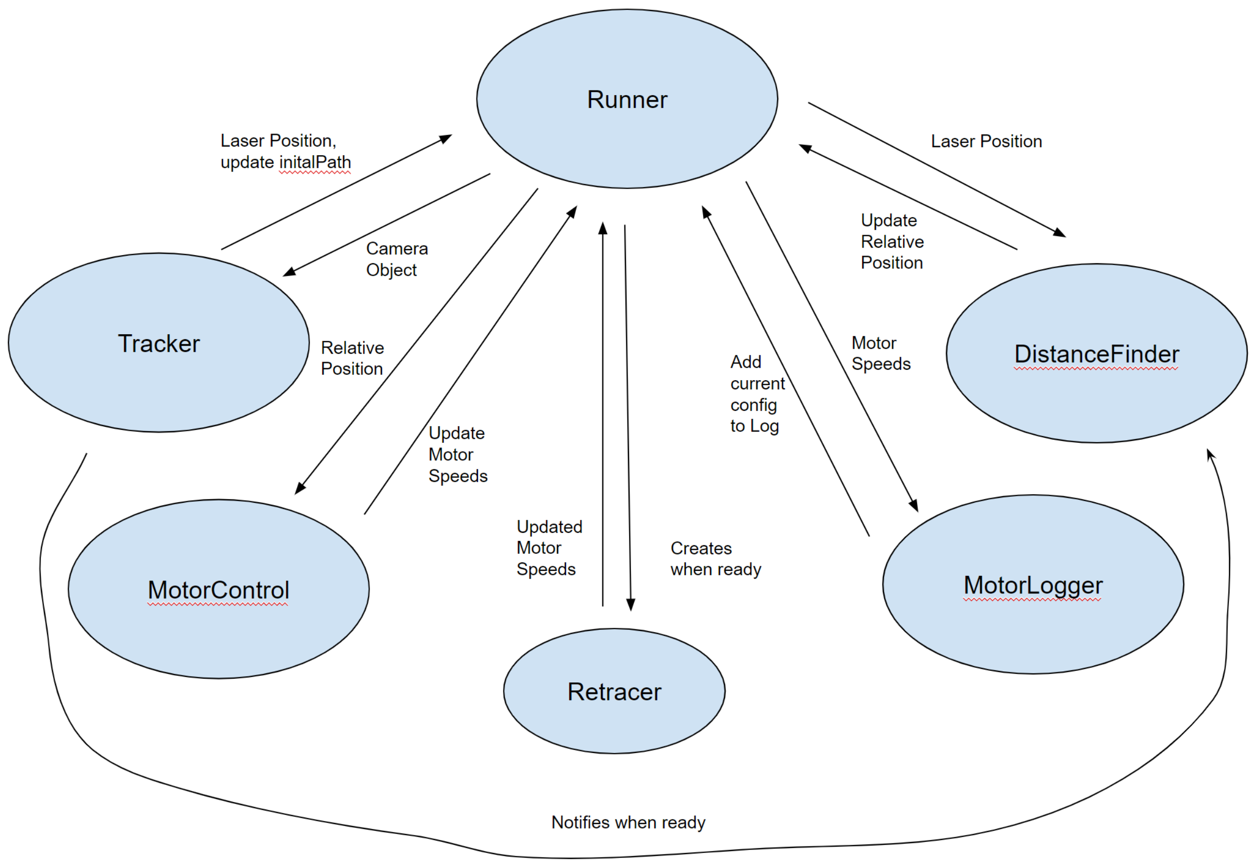

Figure 1-3: Data Flow Diagram

F. Threading and Managing Shared Memory

A simple venn diagram below maps out the shared data that must be managed using locks in Python (the equivalent of Mutexs in C). See Figure 1-4.

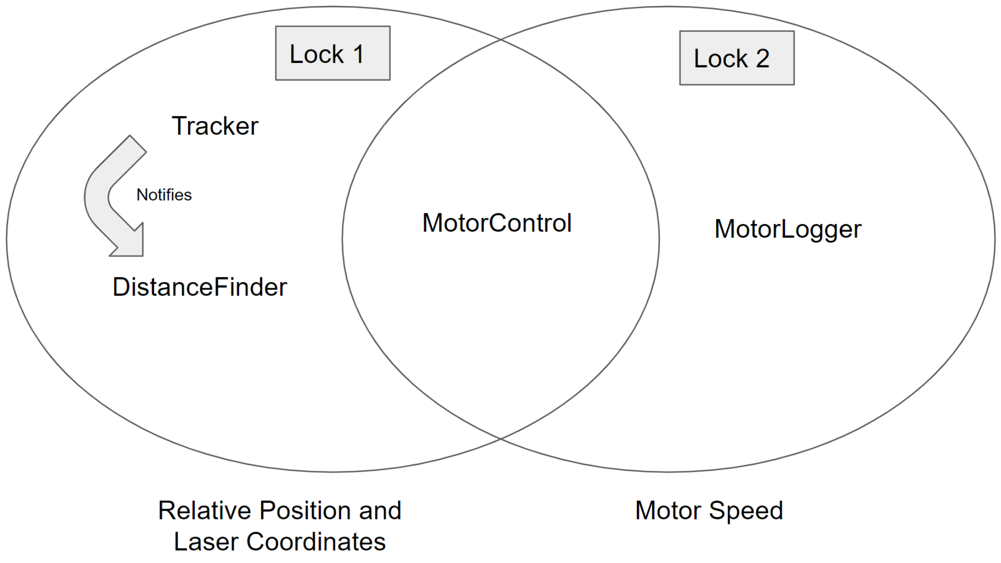

Figure 1-4: Shared Data and Threading Structure

With this, the locking order is not relevant, since there is only one thread that requires both lock1 and lock2. The order of locking/unlocking for MotorControl are as follows: lock1 → lock2 → lock2 → lock1, where the first two lock, and the last two unlock.

Section II - Tracker

A. Camera, Hardware, and Format

The camera included on the Freenove Car for Raspberry Pi kit is a relatively standard 640 x 480 pixel camera. Some images from this camera may be distorted or simply not capture color correctly, which will be discussed later.

The Tracker reads in an image from the camera using the Picamera2 Python package. This thread will take a capture of the image and save it to a file called "tempRedLaser" which is a legacy name as the initial design used a red laser.

Using OpenCV, this file is read in as an array of pixel values in BGR format where each pixel is a 3 element array with the blue, green, and red values between 0-255, representing the full 8-bit color space.

B. Capturing and Initializing the Image

As mentioned previously, the image is captured into the file called "tempRedLaser.jpg." The Raspberry Pi will override this file every time that it captures the image. The image is captured and now can be read using OpenCV. The library includes a function called `imread` which takes a path to the image, and returns the image as an array of pixels.

C. Applying Appropriate Filters

OpenCV provides a large variety of filtering options such that the filter can easily be changed without affecting the rest of the program. This program currently has a filter to find blue light, however the initial design used red light (as red lasers were at first). Some documentation details may include figures for red light.

From here, the image is filtered only for blue light, and less for red and green light. This process is relatively tedious and is quite variable considering that the robot is subject to various different lighting conditions. As each pixel is an array of 3 values (B, G, R), we can simply find a reasonable range for which the laser will appear on the image. After experimentation, our conclusions yield that filtering Blue to be within a range of [240,255] and Green and Red within [0,230] yielded relatively consistent results. Figure 2-1 below shows the capturing of an image and the bounding rectangle of the largest contour found.

Notice in the figure that the middle is filtered out. After looking further into this issue, it was found that the camera is not able to perceive very intense light properly and often will perceive it as white light, meaning that the green and red values are exceeding the maximum of 230. A theoretically simple solution would be to simply change that number to 255 and filter out nothing for red and green. However, this yielded more issues of false detections. This appeared to be the most consistent method, yet is still quite variable to ambient lighting.

After finding all contours, the bounding rectangle of the largest contour is returned. We approximated that the center of the laser will be approximately in the center of the bounding rectangle which is quite reasonable. This is returned as an (x,y) coordinate and placed into the static variable `Runner.laserPos`. The `DistanceFinder` thread now is notified through the condition variable.

Many of the files used for testing and debugging can be found in the "Debugging" directory of the GitHub repository for this project.

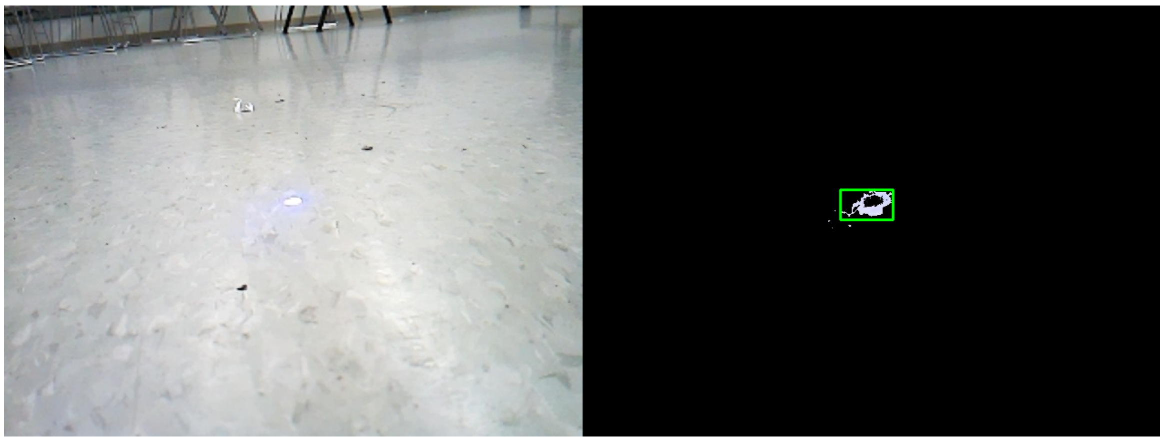
Figure 2-1: Laser Recognition

D. Terminating the Sequence

The Tracker class is responsible for finding out if the laser is no longer active, and will update the value of `initialPath` accordingly. The initial design was to exit simply once the laser was no longer detected. This led to many instances in which the laser was still on, but the camera simply did not pick it up. Therefore, the current implementation is to take 5 different captures, with approximately 50 ms in between each capture. If it is not found after 5 captures, then set `initialPath` to false which will allow all threads to run to completion, then join together in the main thread.

Section III - Distance Finder

A. Approach and Motivation

The purpose of the distance finder is quite interesting as there are various implementations of this project that do not require the distance to be known. Instead some may choose to keep the laser in the center of the display. This approach is functional, however, there are many undesirable side effects of taking such an approach. It may be hard to calibrate properly since servo direction severely affects the tracking distance. The non-linear nature of the image will also make it difficult to fine tune motor curves.

Therefore, the solution to this issue is to have an intermediary step that computes the distance between the car and the laser using the given (x,y) position of the laser from `Runner.laserPos`. This would allow the logic to be separated and the following distance to be adjusted and fine tuned much easier at the cost of added complexity.

The nonlinear nature of such images are often hard to manage, as the actual relative position is a non-trivial multivariate function of both the (x,y) position assuming that the position of the servo stays constant. Note Figure 3-1 below which gives a general intuition on why the pixels on an image do not scale linearly with the distance away. As it can be seen, the distance to the camera is much longer for points further away from the robot. Therefore, they will appear closer on the camera image than points closer to it.

Note: Forward/Backwards Distance will also be referred to as parallel distance, since it is distance parallel to the direction of the car, while left/right distance may be referred to as horizontal distance or perpendicular distance.

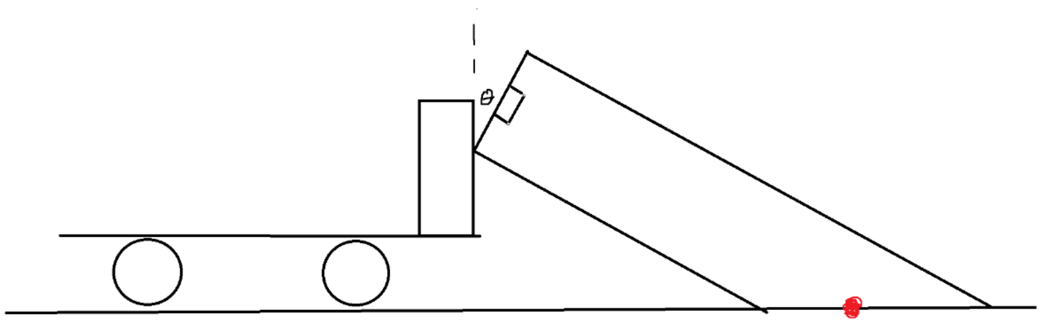

Figure 3-1: Simple Diagram of the Nonlinear Nature of Images

B. Modeling Forward Distance from Laser

The first challenge is to model the forward distance parallel to the direction of the car. Initially, an assumption is made that the vertical distance on the image is the only value that affects the distance away parallel to the car. As it will be seen in the next section, this assumption is validated through data taken from multiple horizontal (left/right) distances.

The first step to computing a model to find the distance away from the car is to take data points from priorly measured images. A flat board with duct tape was set up as a testing apparatus that had multiple markings for linear distances away from the car. The laser pointer was directed at 10, 20, 30, 40, 50, and 60 cm directly in the middle of the tape. The (x,y) position of the laser was appropriately recorded. The plot below shows the distance away as a function of the y pixel, and shows the curve that is used to model the distance.

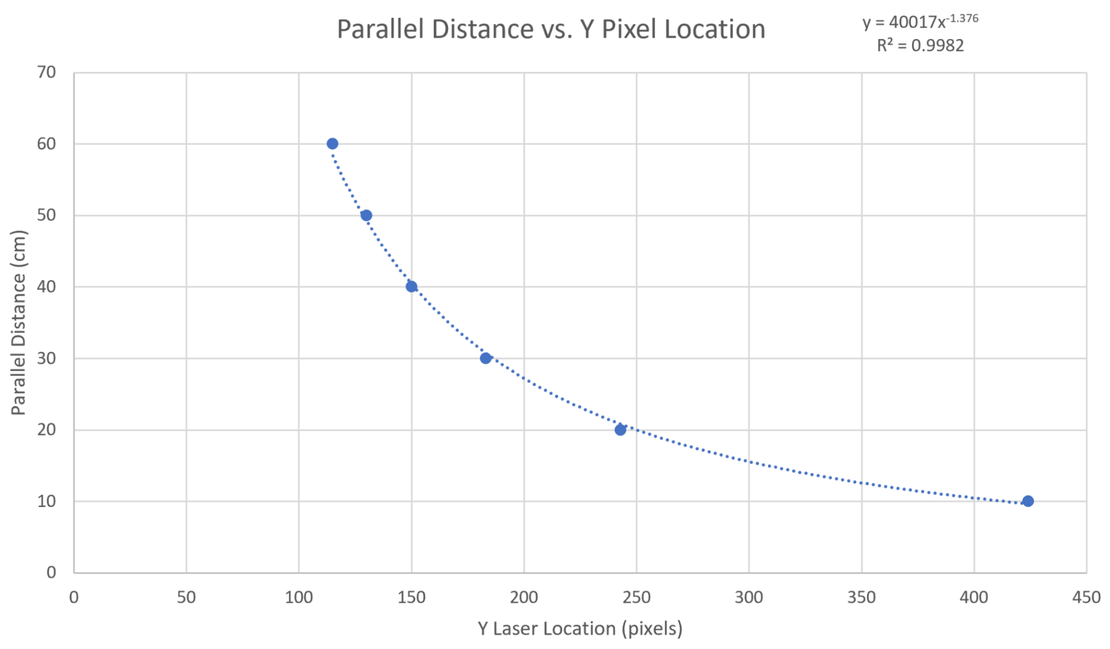

Figure 3-2: Modeling Forward Distance as a Function of Laser Location

The following Power model matches the captured data very well, however sufficient research has not been conducted to find the true relationship between the location and the true distance away. This model is accurate enough within a reasonable range of realistic values, therefore it is more than sufficient for this application.

C. Modeling Horizontal Distance from Laser

The vertical distance was relatively simple when compared to the horizontal distance. There were numerous findings that created many complications when modeling the distance to the left or to the right accurately. The initial guess to find the left/right distance was that the left/right position does not depend on the forward/backwards distance. However, it became very apparent that it would be heavily dependent on the forwards/backwards distance. Therefore, a creative solution is required, as multivariable regressions are quite complicated and beyond the scope of this project. See Figure 3-3 for a plot of points. For each y-value (forward/backwards distance), the 5 points from left to right are the X pixel values at which the laser was pointed 10 cm to the left of the center, 5 cm left, center, 5 cm right, then 10 cm right.

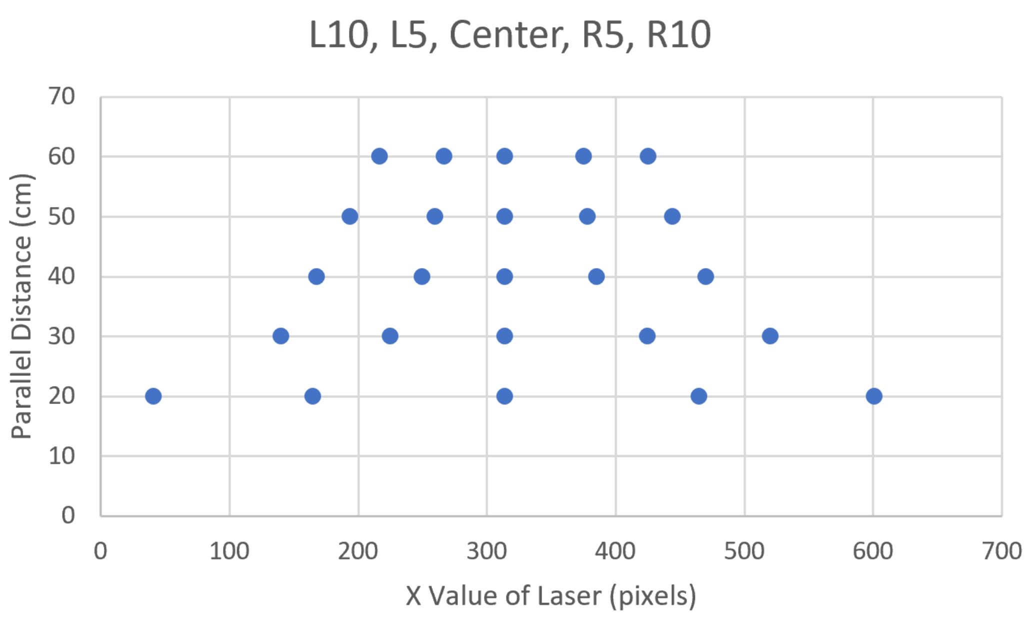

Figure 3-3: X Value of Laser at Various Forward Distances

The solution for this problem is to use the concepts of level curves. We will find functions for each of the curves corresponding to left 10, left 5, right 5, and right 10. The center is not used since it is trivially a single value. Each of these functions will find the X value of the pixel (output) that would correspond to a forward/backwards distance (input).

Each of these level curves can be computed individually. Figure 3-4 shows the 4 different level curves that were computed. A quadratic model was used as the curve will most likely best on a curve and is relatively accurate for a reasonable range of values.

For each of the following curves, the value of the pixel was found for 5 different forward/backwards distances as seen below. They were found at 20, 30, 40, 50, and 60 cm away. It is clearly a very non-linear curve that will work to compute the curves. The images for each of the following 25 data points taken here can easily be found in the repository under the `Debugging/CalibrationCaptures` folder.

In the code, the lambda functions are used to implement each of the functions. The center value is not shown here, but can be seen in the code as 314. The lambda functions are named `fL10, fL5, fCenter, fR5, fR10` and each take one parameter which is the forward/backwards distance.

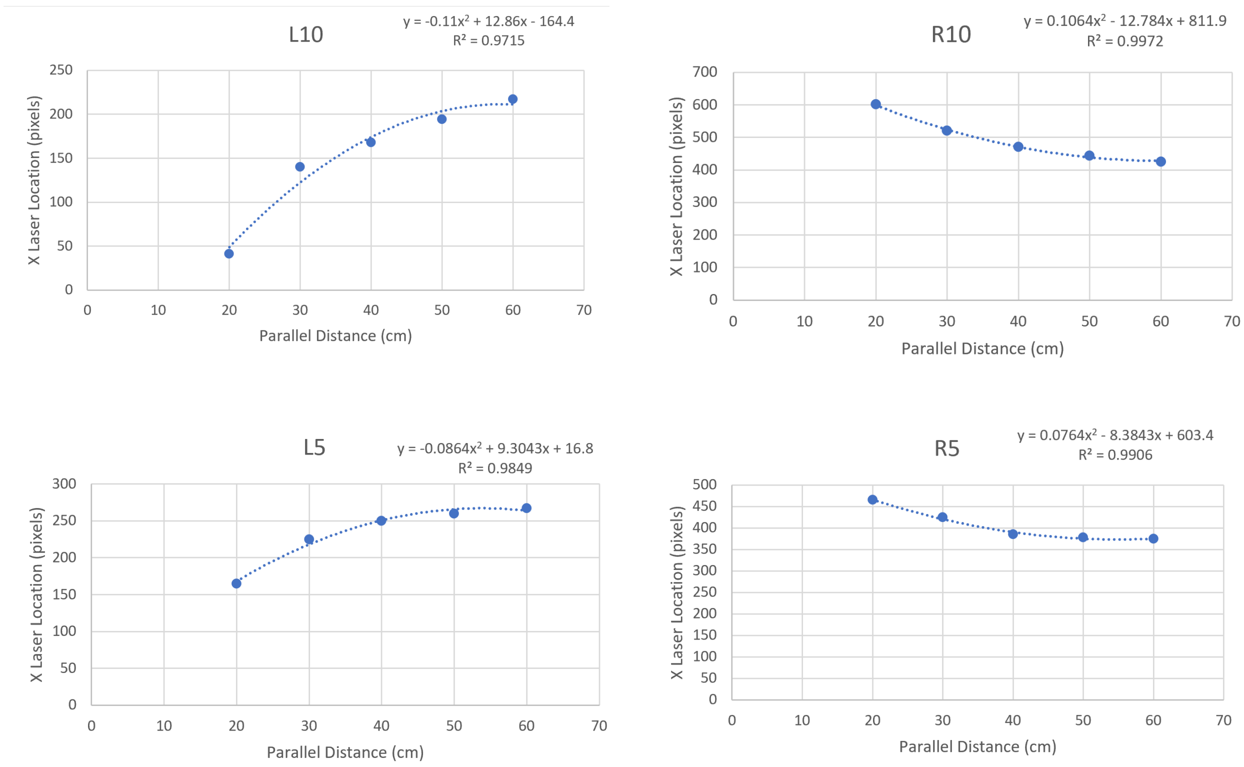

Figure 3-4: Modeling of Level Curves for

From here the four level curves and the center have been computed. Up to this point, given the forward/backwards distance from the laser, we can compute that pixel that relates to left 10, left 5, right 5, and right 10 centimeters at that given distance. With this, we still do not have a way of going from the pixel value to the actual distance (the inverse of what we just found).

The solution to this comes with the idea of taking a weighted average of the left/right distances based on how close the pixel is closed to each level curve. This is illustrated in an example below. Figure 3-5 also explains this visually

```
Ex. Laser found at Pixel Value (500, 200)
      (1) Find the forward (parallel) distance to the laser
            $d = 40017 * (200)^{-1.376} = 27.3$
      (2) Compute L10, L5, Center, R5, and R10 at 27.3 cm.
            $L10 = -0.11(27.3)^{2} + 12.86(27.3) - 164.4 = 104.7$
            $L5 = -0.0864(27.3)^{2} + 9.3043(27.3) + 16.8 = 206.4$
            center = 314
            $R5 = 0.0764(27.3)^{2} - 8.3843(27.3) + 603.4 = 431.5$
            $R10 = 0.1064(27.3)^{2} - 12.784(27.3) + 811.9 = 542.2$
      (3) Find the interval that the pixel is between.
```

```
        Note: If the pixel is outside of L10 and R10, simply
        return the -10 or +10 depending on which side it is
        on.

        500 is between R5 and R10.

        Let distL be the distance to the left boundary, and
        distR be the distance to the right boundary. Let
        totalDist be the total distance between the two
        boundaries.

        distL = 500 - 431.5 = 68.5
        distR = 542.2 - 500 = 42.2
        totalDist = distL + distR = 110.7

(4)   Compute a weighted average of the distance to bounds

        horizontalDist
              = (5)(68.5 / 110.7) + (10)(42.2 / 110.7) =
        6.906
(5)  Final Answer
        (27.3, 6.906)
```

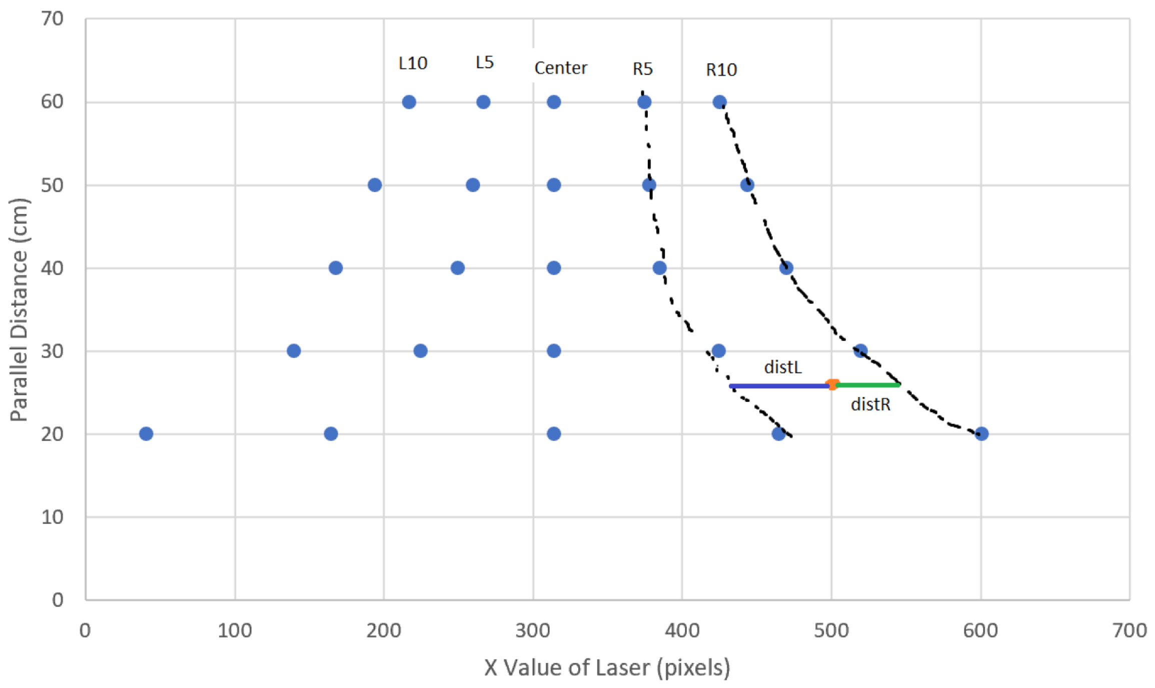

Figure 3-5: Visual Representation of Horizontal Algorithm

D. Further Potential Improvements

There are plenty of improvements that can be made to this section. One of the major improvements would be interpolation for distances to the left and right of 10 cm. Currently, the code will return values between -10 and 10 since we do not have a good method of interpolation. A linearization of the region to the left of L10 and right of R10 can be used, however could easily cause many issues with points significantly far away, and result in major issues.

Another area of improvement is in the model. We used an interesting method of creating level curves and finding a weighted average. Since the overall logic is not too sensitive to small inaccuracies, this method works sufficiently. However, in order to find a more accurate model, a machine learning algorithm could be used, however would require significantly more work to implement and train. Inherently training a neural network would require many more data points. This would be very tedious and time consuming to find the actual measured points on the image.

A compromise between this would be to create more level curves for increased accuracy. This could include a curve at potentially 15 cm in both directions and also another level curve between each of the current ones.

Section IV - Motor Control

A. Conceptual Program Flow

This component thread is responsible for taking the relative position of the car and adjusting the motors into the correct speed to move the car towards the laser. This can occur at any time that the locks are unlocked (i.e. none of the other threads are currently holding either of the locks). This thread requires both locks.

The position between the car and the laser is first converted to polar for calculating each of the speeds that will be seen in the next section. It is much easier to find a suitable function through a function of distance and angle, rather than two dimensional points.

B. Modeling Base Speed

The base speed should be modeled first as a piecewise function. The general process for designing this function is to consider the following distance of the car. The program is designed to keep the car at a distance of 10 cm away from the laser, and will try to catch up to that distance if it is further away. Thus, the car should be stopped if at a distance of less than 10 cm.

In order to limit the speed and constrain the function, we should consider making a distance at which the motor speed is maxed. To prevent the burning out of the motor and the overall speed of movement, the motors are limited to a speed of 1000, rather than the maximum speed of 4095.

For the region in between, the simple solution is to make the progression linear, which is the approach that was taken. The total piecewise function for the base speed is as follows by the constraints given.

$$\mathrm{BaseSpeed}(r) = \begin{cases} 0 & \text{if } r \leq 10 \\ 50 * (r - 10) & \text{if } 10 < r < 30 \\ 1000 & \text{if } 30 \leq r \end{cases}$$

See Figure 4-1 for the plot of the curve.

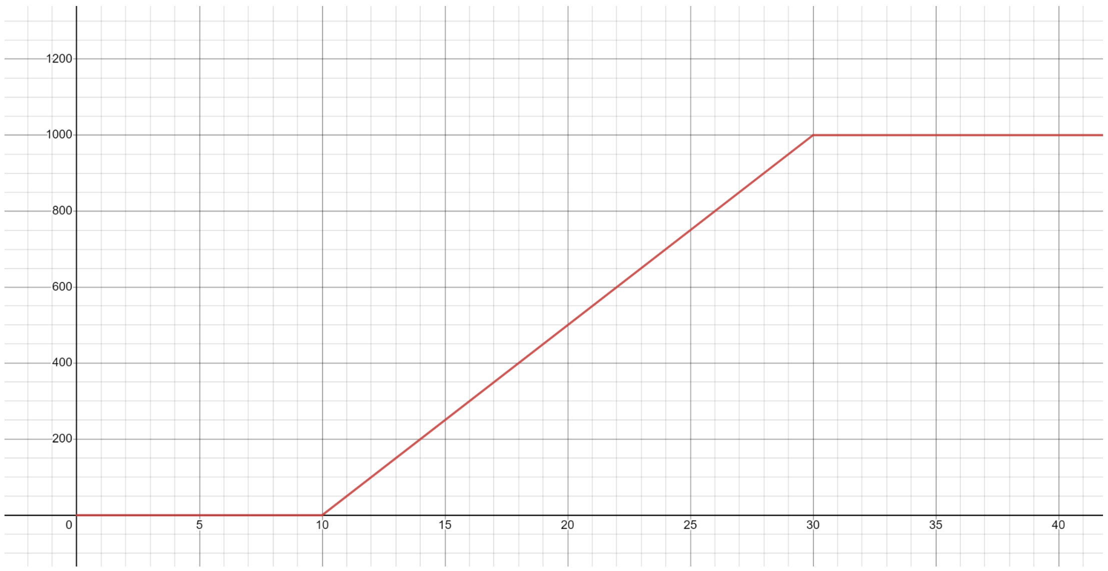

Figure 4-1 : Plot of Base Speed vs. Distance

C. Modeling the Left and Right Motor Speed

The approach to modeling the left and right speed from the base speed is relatively simple. First we can calculate an offset based off the angle of direction. In general, we want a function that will be negative for negative angles of theta, and positive for positive values of theta. Then, we can add the offset to the right motor and subtract from the left. In general, we would like a similar shape to the previous base speed plot, yet we want the slope at theta = 0 to be non-zero and the curve to level off at higher values of theta.

A function that meet this description is the inverse tangent function. Using the inverse tangent function provides each of the properties we need after some simple scaling of values. With this offset, we can add that as a multiplier for our base speed. For example, if our offset function returns a value of 0.2, then the left speed would have a multiplier of 0.8 and the right speed with a multiplier of 1.2. This is now multiplied by the base speed.

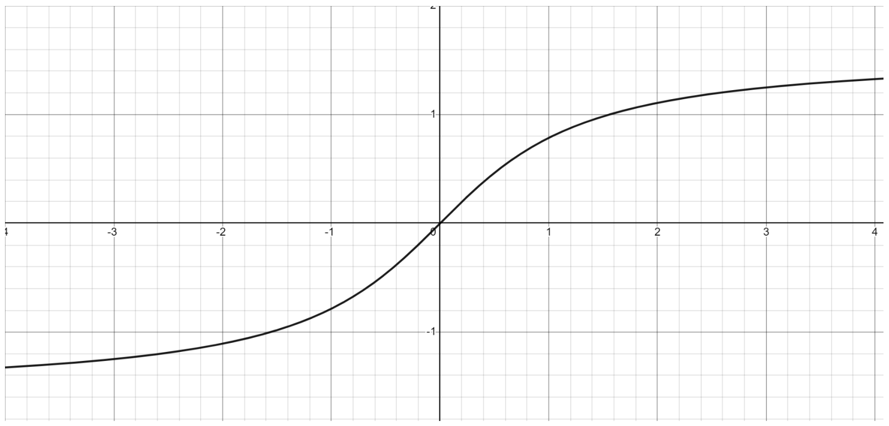

Figure 4-2: Plot of Offset Function

Depending on the surface, an extra multiplier was optionally included in the case that the car is not turning enough. A comfortable multiplier was found to be between 1.5 to 2 depending on the surface conditions of the floor.

Section V - Motor Logger

A. Purpose

The Motor Logger class is responsible for providing the retracer class with an array of time, lSpeed and rSpeed values that allow it to retraverse the path of the laser that it followed. Every time the thread's loop is executed, the lSpeed, rSpeed and time are placed in the `runner.Runner.log` static variable.

B. Implementation

The implementation of the class is relatively straightforward, as it contains a run loop that appends the time, lSpeed, and rSpeed values to the previously mentioned log. `lock2` protects the section of data that logs the motor speed. This is needed since it is possible to leave the thread then change the motor values or time after part of the log was captured. However, it is acceptable to find the laser again and find the distance, but just not update the motors.

C. Observations and Optimizations

This code was simple in implementation, however there are many issues and optimizations that can be made here. This is the core fundamental structure that controls the behavior of the retracing. One issue with this implementation is that the log will often push many of the same values onto the log. This is explained by the fact that capturing images and processing the new distance and motor speeds often takes much longer than simply logging one variable. This is solved later in the `Retracer` class, however the implementation would be much more memory efficient if implemented here.

Section VI - Retracer

A. Initial Design

The initial design of the motor log was to simply loop over all variables and sleep for the delta of the duration between two of the motor logged time. However, due to the inaccurate nature of `time.sleep` for smaller values, the total error added up and caused extreme inaccuracies in the retraced path.

This implementation did not work and did require some work around solutions. One solution would be to undersample, however this would cause problems in not being accurate in the path now. After observing the log when printing to the console, due to the fact that many values in the log are the same, compressing the log through combining the times of the same values.

B. Log Compression

Consider an array of values. There must be an algorithm to compress the array into each of the motor values. The motor values should not be repeated after the algorithm runs.

This algorithm works through a simple two pointers solution. Place two pointers (index value) on first value in the array, then move the right pointer down until the motor values are no longer the same as the previous. Now, subtract the time of the left pointer's value from the one of the right pointer's time. Now push the left pointer's speed, with the new time difference to the log. Place the left pointer where the right pointer currently is. Repeat until the end of the array is reached and put in the last value to stop the car.

C. Recalling the Log

Recalling the motor is a relatively trivial task as it simply involves calling the motors and tuning them to the correct speed and delaying for a given amount of time according to the log. The `time.sleep` method proved to work, however the program would sleep for slightly too long and wake up too late. Therefore, the program is written to sleep for a lower amount of time, so that the program will resume slightly faster. This proved to be much more accurate. It has been found that dividing the time by approximately 1.15 is an effective technique to fixing this issue. This is due to the nature of `time.sleep` using ticks rather than a hardware timer.

D. Future Improvements

An improvement that can be made in the future is the changing of versions of Python. Currently, the supported version of Python is 3.9. However, Python 3.11 does use the nanosleep function which uses a hardware timer rather than one that operates on software ticks. This would solve the previously mentioned problem, and sleep for closer to the correct amount of time.

Another alternative to this is to use `cffi` which is a python library that allows for C code to be introduced directly into the Python environment. Therefore, this could also be used to implement nanosleep, however comes with many dangers that are unwanted and could potentially crash the system, which is catastrophic even for a soft real time system.