Autonomous Path-Following with the Texas Instruments RSLK

ECE 3 S20 Final Report

Kyle Leong (123456789)

Introduction and Background

The objective of the ECE 3 final project was to program a Texas Instruments RSLK robot to autonomously follow arbitrary paths [1]. This robot should be able to correct its deviations from the path without requiring the student to manually "hard-code" the path in. The paths that the robot should follow are given by a black line of various shades, with the darkest black shade in the middle of the track. We were supplied two paths with which to test our robot, and these paths were to be used in the final examination of the robot. One path was a straight line with flanking lines that begin to converge as the robot travels further down the path. It is meant to measure the stability of the robot, i.e. if the robot can detect and stay on the path given an imperfect, off-the-line starting position. It is shown in Figure 2. The second path is called the ribbon. It is shown in Figure 1. Not visible in the figure are the boxes which outline the starting positions of the car. None of these starting positions are centered on the path, providing an additional means by which to test the stability of the car. Additionally, in the ribbon path, the robot must turn around when it reaches the solid black bar and the end of the track and traverse the track in reverse, stopping once it reaches the other solid black bar. Again, despite knowing the paths that we will be tested on during race day, the objective is to have the robot work with arbitrary paths, not just the ones provided.

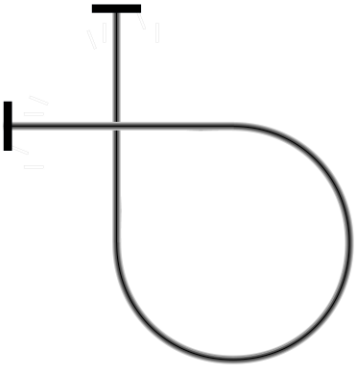

Figure 1: Scaled Ribbon Path
Provided by Dr. Briggs

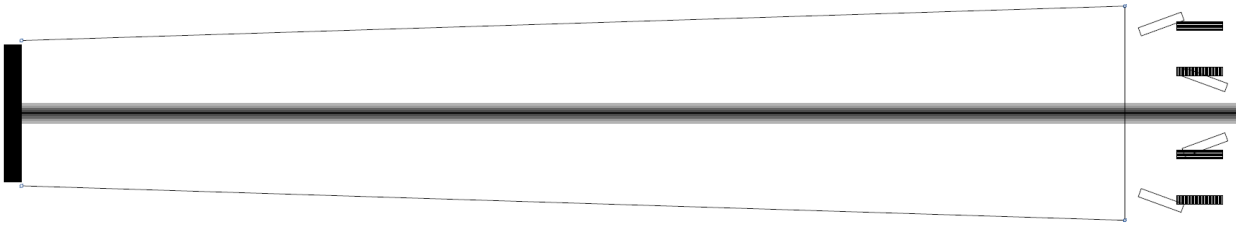

Figure 2: Scaled Straight Line Path
Provided by Dr. Briggs

The RSLK obtains information about its current position via an array of eight infrared reflectance sensors. These are positioned perpendicular to the path so that an algorithm to map infrared sensor values to a distance from the center of the track can be applied while the robot is in motion. A phototransistor in each sensor is the primary component that allows the robot to detect the reflectance in the path. The operation of a phototransistor can be likened to that of a

variable resistor, where the resistance depends on the intensity of the reflected light. A more reflective path will result in a lesser effective resistance. The circuit that is used to compute a value of the reflectance can be seen below.

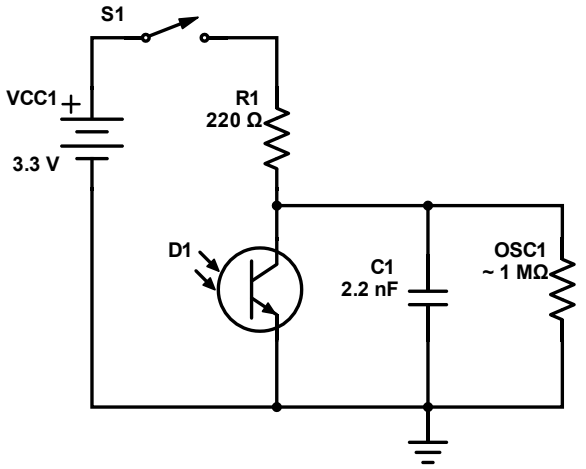

Figure 3: Schematic of Internals of Reflectance Sensor
Original Schematic Provided by Dr. Briggs
Schematic recreated with Scheme-It

In the schematic of the reflectance sensor, VCC1 represents the voltage coming from the RSLK's power supply, S1 is the sensor's internal switch, R1 is a resistor used to limit current, D1 is the phototransistor, C1 is the capacitor, and OSC1 represents a device that can take voltage readings across the capacitor [2]. A value for the reflectance of a sensor is calculated as follows. The switch is closed for a time long enough for the voltage across the capacitor to be equal to the voltage source $V_{CC1} = V_{C1} = 3.3V$.

The switch is opened at time $t = 0$. At this point, the circuit becomes a RC circuit with two resistors in parallel. A rule for parallel resistors is that the resistor with the smaller value will dominate. It is given that the resistance of the phototransistor is much smaller than that of the load provided by the oscilloscope, so the resistance of the phototransistor will dominate in this scenario. Furthermore, any basic circuits textbook will prove that the voltage across the capacitor as a function of time in an RC circuit can be given by [3]:

$$V_{C1}(t) = V_{CC1}(1 - exp(1/D_1C_1))$$

The value $\tau \equiv D_1C_1$ is known as the RC time constant. After five RC time constants pass, the voltage across the capacitor is less than a percent of its original value, so we can consider the capacitor to be fully discharged. Depending on the reflectance of the path, it will take anywhere from microseconds to a few milliseconds to fully discharge the capacitor.

A circuit equivalent to the one in the figure above was created and the voltage across the capacitor was taken by an oscilloscope as the switch was opened. The figure below shows the differences in discharge time for the same circuit but with varying brightness/reflectance. It is shown that a darker path will take longer to discharge the capacitor, about an order of magnitude longer than a bright path. When mounted to the RSLK, the sensor array outputs the raw voltages of each of the sensors. The provided ECE3 library includes a procedure that functions as the oscilloscope did in circuit above [4]. It then returns an integer value ranging from 0 to 2500, inclusive. This data was massaged and fed into a sensor fusion algorithm which results with the estimated distance (in millimeters) of the robot from the center of the path. The specifics and design of this algorithm will be described in greater detail in a later part of this report.

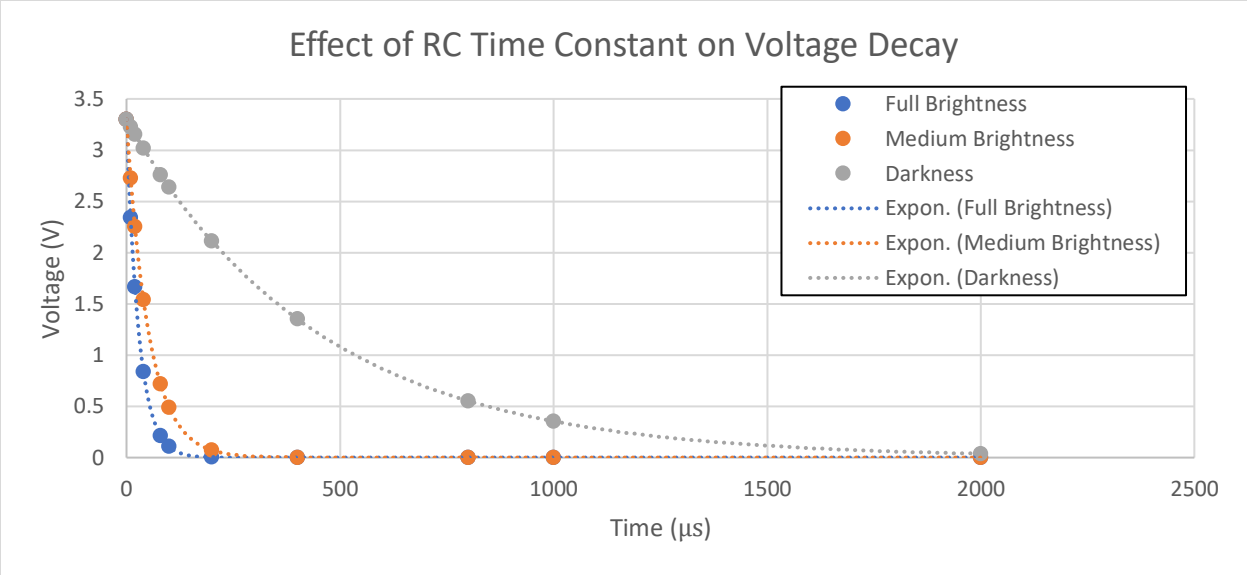

Figure 4: Effect of Path Reflectance on Voltage Decay Rate

Testing Methodology

In the previous section, a brief overview of the way a sensor detects the reflectance of the path underneath it was given. The RSLK's complete path sensing functionality comes in the form of eight of these sensors arranged in a line such that they are perpendicular to the "forward" and "backward" directions of the robot. However, only the raw values produced by voltage decay time of the sensors were exposed via the RSLK's API. To be able to determine whether the robot is too far to the left, too far to the right, or dead center on the path required that I design and implement a sensor fusion algorithm. Preliminary tests were necessary to ensure correct operation of the motor drive and path sensing subsystems.

Test Setup and Conduction

Before I could implement the main part of the software, the path sensing algorithm, it was imperative to ensure that the robot itself worked correctly with trivial programs. This would ensure that any error in the robot's functionality was due to my implementation of the path sensing algorithm and not something with the RSLK's hardware. Programming the RSLK involves setting various pins on the launchpad to HIGH, LOW, or some analog value between 0 and 255, and I was given a skeleton program by the TA to modify for my needs [5]. To ensure correct operation of the motor drive system, I performed two tests, one to find the ideal integer value to send to the left and right motors that sends the robot in a straight line, and another to ensure that the RSLK could perform a donut. The pin chart required to program the RSLK is shown in the figure below.

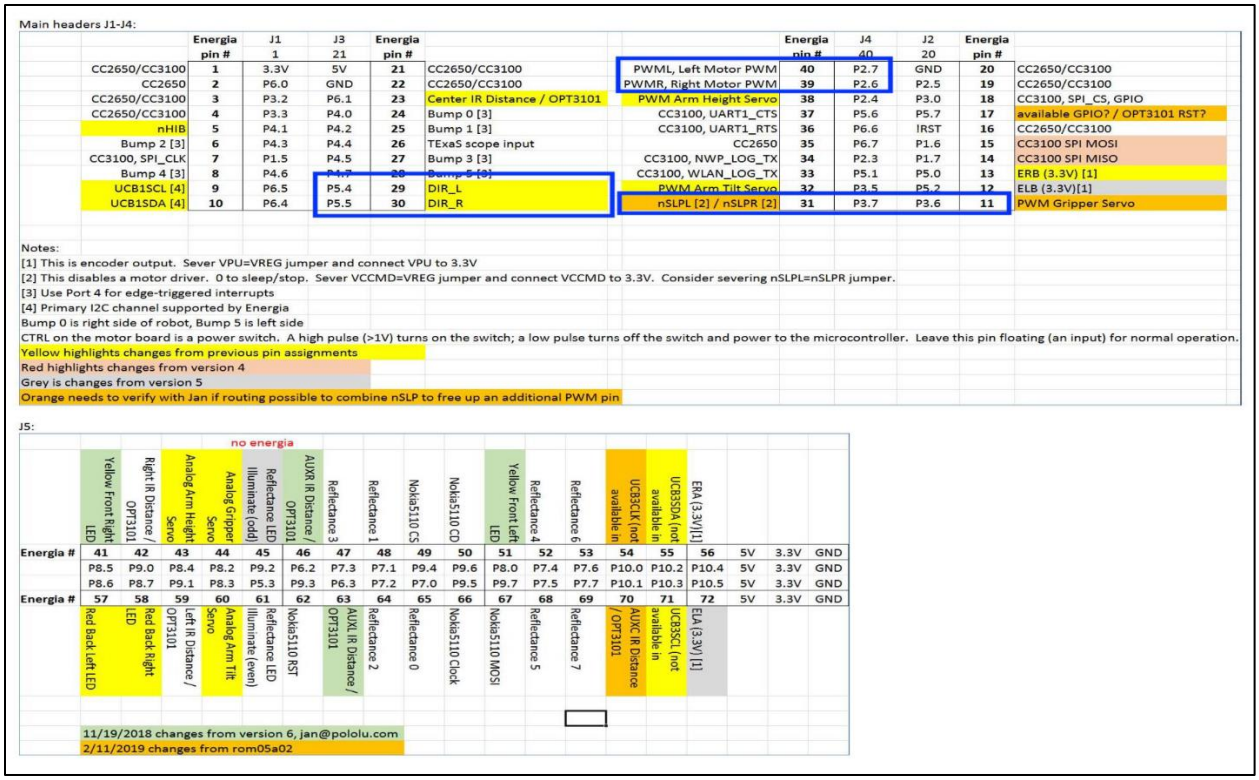

Figure 5: RSLK Pin Chart
Provided by Dr. Briggs

To find the values that would send the robot in a straight line, I first chose an arbitrary value, 70, and wrote a simple program that would tell the RSLK to move forward at that constant speed. I used the tiles in my living room as a reference for a straight line. The RSLK was initially placed so that the right wheel would align with the edge of a tile. I would then take note of whether the robot veered to the left or the right, and any other non-straight behavior. With these results, I then adjusted the intensity that was sent to the motor until the RLSK could run in a sufficiently straight line for a few feet before veering off the tile's edge. A record of the speeds that were tested and their behavior is shown in the table below. Reading the table from top to bottom is representative of the order in which I tried these values. In the final version of my program, I went with a value of 30 for both motors.

Left	Right	Comments
70	70	70 was the initial value provided by TA's sample code. Veers slightly left.
70	68	Continues in a straight line for an adequate distance.
30	29	Slowed down to accommodate for lack of derivative controller.
30	30	Fixes problems with robot not turning left fast enough.

Table 1: Motor Speeds Tested for Straight Line

Additionally, I wrote a program to ensure that the robot could perform a 180-degree donut where the axis of rotation was the center of the RSLK rather than one of the wheels. This

was necessary because both race-day tracks contain solid black lines that, when sensed by the RSLK, mean that the car must perform a "U-turn" and continue travelling in the opposite direction. I first made the robot perform an infinite donut around its center by taking the code for the straight line in the previous paragraph and changing the direction on one of the wheels (chosen arbitrarily) to spin in the direction opposite it was for the straight line. I then experimentally recorded the time it took for the robot to perform a turn and adjusted the time so that after I finished, the robot would perfectly turn 180 degrees. To aid in measuring the angle that the robot rotated, I aligned a floor tile in my house with one of the wheels of the RSLK. The values I tested in order to determine what amount of time the robot takes to perform a 180-degree turn is shown in the table below. The ordering of the rows top to bottom show the order in which the times were tested and their results. The final iteration of my program uses a 750-millisecond donut.

Time in Donut (ms)	**Results**
1000	Overshot the 180-degree turn significantly.
800	Slightly overshot the 180-degree turn.
700	Slightly undershot the 180-degree turn.
750	Nearly perfect 180-degree turn.

Table 2: Experimental Time to Perform 180-degree Turn

I now had created and verified code that would allow the robot to move in a straight line, and I had implemented code that tells the robot to turn 180 degrees, giving me confidence that the motor subsystem was indeed working correctly. However, the correct operation of the path sensing subsystem needed to be verified. Recall that the ECE3 library provided by Professor Briggs contains a function to read the values of each of the eight sensors into an array, and after the function call, each element in the array will contain a value from 0 to 2500, inclusive. To verify that this subsystem was indeed working, I then wrote a simple program that obtained the values of each sensor via the ECE3 library function and then dumped those values to the serial console. Professor Briggs provided a short section of the straight path that was to be used in the RSLK's calibration. I slid this calibration paper underneath the robot and moved it side to side so that the dark path would pass underneath different sensors as it moved. In the serial monitor on my computer, I could see that the values recognized by the sensors were changing as I moved the paper, and thus it was verified that the path sensing hardware was operating correctly. It was important to add a two second delay when the RSLK was initializing and a delay greater than ten milliseconds in between successive loops of the program, otherwise the serial monitor would output garbage values, despite the baud rate being set correctly.

Now that it was verified that the drive and path sensing subsystems of the RSLK were operating normally, the next step is to implement the sensor fusion algorithm which will enable the programmer to determine whether the robot is to the left or the right of the path. Here, I reused the program used to verify the operation of the path sensing programs. For each sensor, the higher the number, the darker the path. Using the calibration path provided by Dr. Briggs, I

sketched out 2-millimeter increments, where the "zero point" is the location of the left hand side of the left wheel when the car is perfectly aligned over the center of the calibration path. Once the positions at which I was supposed to measure were sketched out on the calibration paper, I plugged the RSLK into the computer and looked at the output of the serial monitor for each position. Once the readings stabilized, the results were recorded in an Excel spreadsheet, and I moved onto the next position. This was repeated for all positions.

The result is table of sensor values for each distance from the "zero point". However, this is raw data, and in this form, it is not possible to logically deduce a distance from the center of the path. Sure, one can determine whether the robot is to the left or the right of the line, but it would be difficult to determine exactly how far away from the line it is. That information is necessary to correctly implement a proportional controller.

Data Analysis and Interpretation

To gain a better understanding of the raw data that I amassed in the previous paragraph, I plotted the reported reflectance for each sensor as a function of the distance from the "zero point". The results of that plot are shown in the figure below.

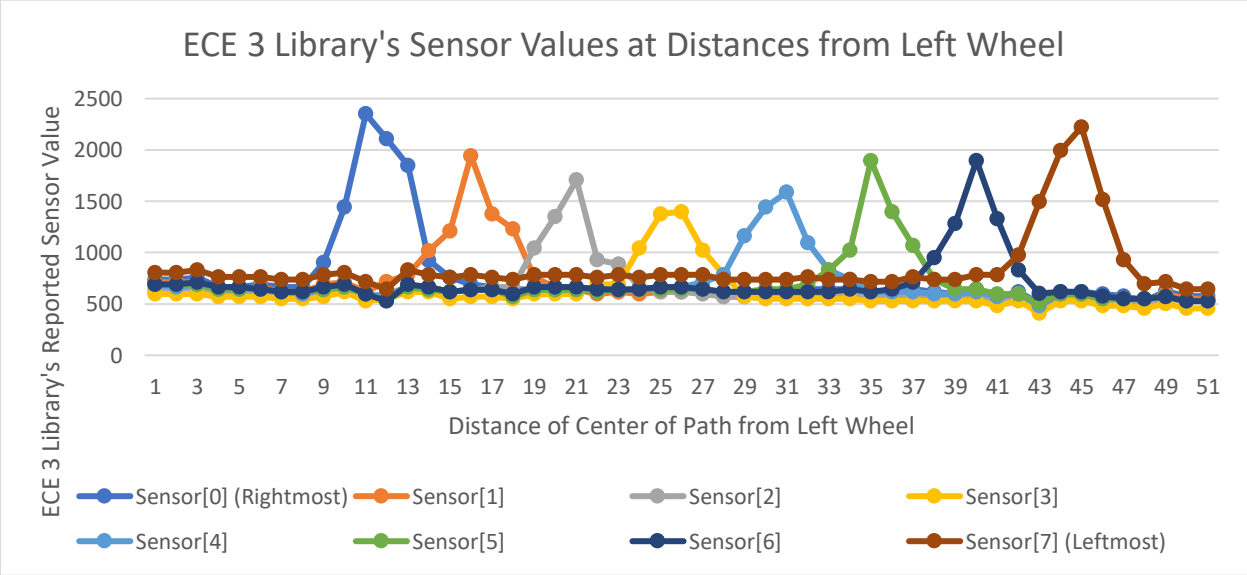

Figure 6: Reported Sensor Values at Distances from Left Side of Left Wheel

Note that the minimum and maximum values for each sensor differs greatly between each in the sensor array. It was then necessary to normalize the values for each sensor. In other words, the raw values of each sensor were interpolated by a function that has a minimum of 0 and maximum of 1000, inclusive. The interpolation function for a sensor can be given by

$$N(x) = \max\left\{0, \quad \left(\frac{1000}{M}\right) \cdot x\right\}$$

In the function above, $N(x)$ is the interpolated value, M is the maximum experimental value obtained by that sensor, and x is the raw value from the sensor obtained by the ECE3 library's function. After applying the normalization function, I graphed the normalized sensor readings as a function of the distance from the "zero point". The results of this normalization can be seen in the figure below.

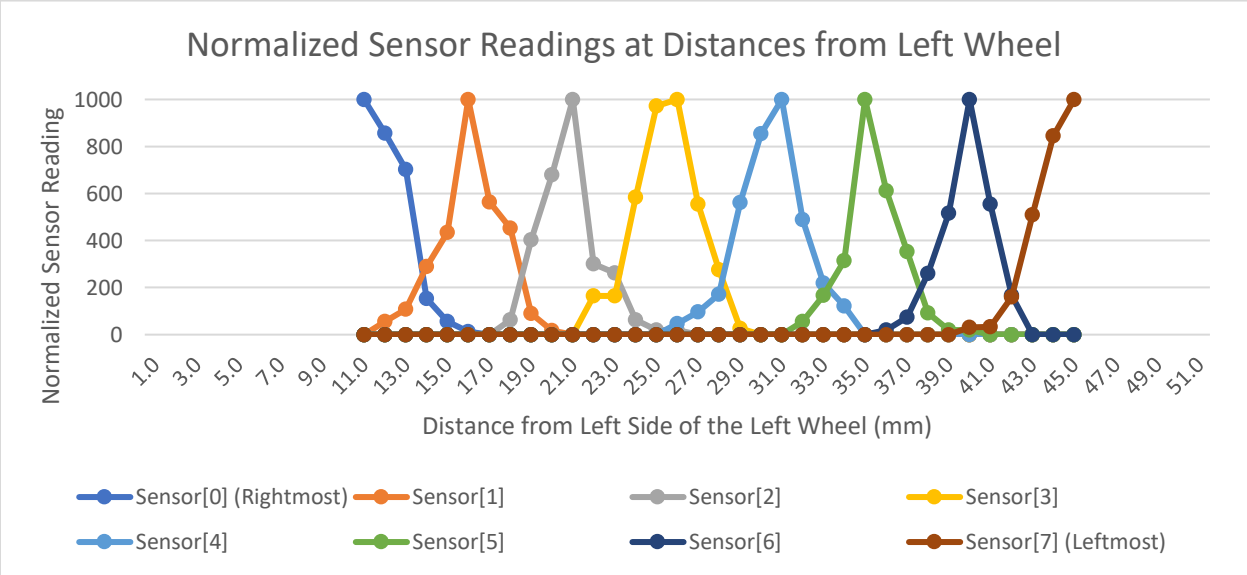

Figure 7: Normalized Sensor Readings at Distances from Left Side of Left Wheel

I needed to combine the eight normalized values into one value representing a distance from the center of the path. I did so by assigning weights to each of the sensors, with sensors on the left of the RSLK having negative weights, and positive for the others. Sensors that were to the extreme of one side were given a greater weight magnitude. Each sensor's normalized value was then multiplied by its weight, and the resulting products were summed together, yielding a single scalar. All that was left to do is map this scalar to a real distance from the center of the path, in millimeters. From there, to obtain a final mapping of the sum of products to real distance from the center of the line, a line of best fit was created. This value representing true distance from the center of the line in millimeters is hereafter referred to as the error. To arrive at the optimal weightings, I first created the linear regression line, and attempted to find the weighing scheme that would result in the strongest correlation. A summary of weighing schemes and their corresponding coefficients of determinations is shown below in Table 1. A higher coefficient of determination means that the experimental readings at a given distance from the center of the path more closely follow their predicted values. The function for computing a single scalar from representing the distance of the robot from the center of the path can be given as follows.

$$r = \sum_{s \in S} [W(s) \cdot V(s)]$$

In the above formula, r is the resulting scalar whose magnitude represents distance from the center of the path (a greater magnitude means the robot is farther from the center) and sign represents whether the RSLK is to the left or the right of the path (a negative number means that

the robot is to the left, and a positive signifies that the RSLK is to the right). Each $s \epsilon S$ is a sensor in the set of all eight sensors. The weight of a given sensor is $W(s)$. The normalized value reported by the sensor is given by $V(s)$. Various weighing scenes were tested, and their line of best fit and its corresponding coefficient of determination were compared.

Weighing Scheme (Left to Right)	**Coefficient of Determination (R^2)**
(-3, -2, -1, -.5, .5, 1, 2, 3)	0.8798
(-4, -3, -2, -1, 1, 2, 3, 4)	0.8956
(-8, -4, -2, -1, 1, 2, 4, 8)	0.8347

Table 3: Weighing Schemes and Their Coefficients of Determination

Since the second weighing scheme had the greatest coefficient of determination, it was the most accurate at predicting the true location of the car, and it was chosen for use in the actual program. A graph of the computed sum of products at each distance and the corresponding regression that comes from the selected weighing scheme is shown below in Figure 5.

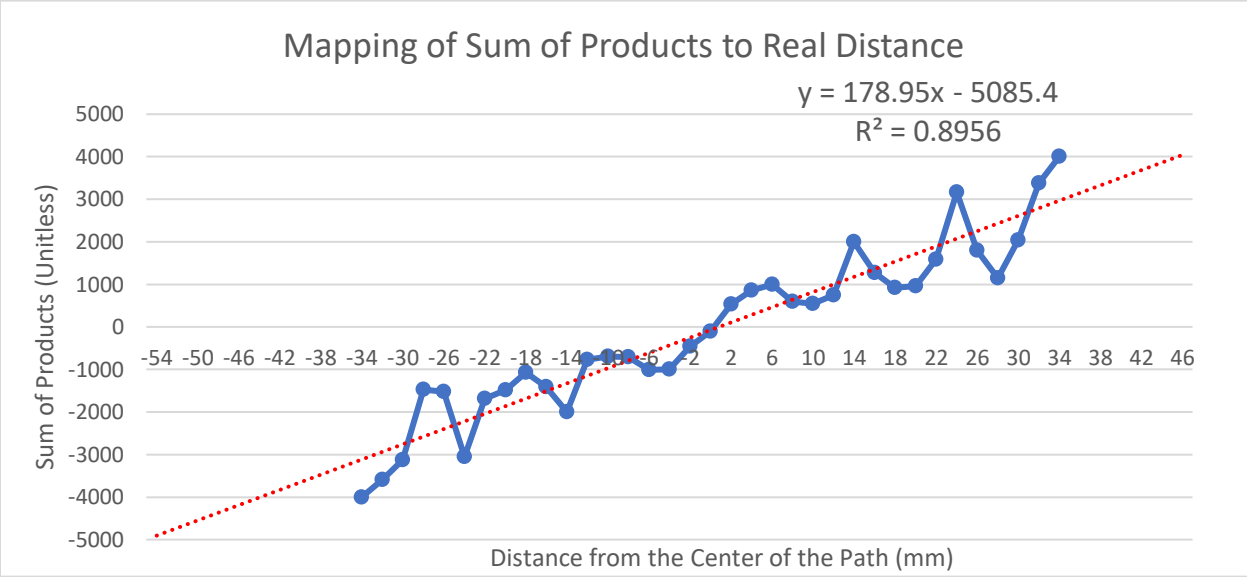

Figure 8: Equation for Final Line Distance Mapping

Clearly the graph is not linear. One can see peaks or troughs corresponding to the peaks seek in Figures 3 and 4. This is a result of the sensor array not detecting the path until it is almost directly underneath the sensor. Ideally, there should be a gradual increase in the reported number and corresponding gradual decrease. However, experimentally, there was a sharp increase and sharp decrease as the darkest part of the path passed underneath each sensor. The sharp increases and decreases make it so that no matter what the weighing, the graph will always be non-linear. Despite re-measuring the reported values, re-charging the batteries, and using another printout of the straight path used for calibration, I could not obtain a reading with the expected gradual increases and decreases.

After some further testing, I realized that the RSLK was not detecting the lighter shades of grey that are present on the racetracks — it only detects the darkest portion of the track. If the

RSLK was able to detect the various shades of gray and report an appropriate number for it, then the graph would likely have bene much more linear, and the coefficient of determination would have been much closer to a perfect value of 1. The failure to detect the rest of the path was not the fault of my printer — the solid black parts of the path that serve to signify to the RSLK that it should turn around were printed perfectly fine. It was also suggested to me that I should adjust the height of the sensor array from the ground. Since the sensor daughterboard was only held in place by the tension of the pins that connect to that main part of the RSLK, it was difficult to adjust the position. Thus, I concluded the reason for the final graph being starkly non-linear is due to an issue with the hardware, and not my testing environment.

Now that I had a model with which to accurately gauge the distance in millimeters of the RSLK from the center of the path, the next step was to implement a program that could utilize this distance data and adjust the speed sent to the left and the right motors appropriately. The class discussed the use of a PID controller, but I elected to only use the P (proportional) controller [6]. I chose not to use the I (integral) controller because it was not recommended by Dr. Briggs himself. I also chose not to use a D (derivative) controller. Since the action of the derivative controller depends on the rate of change of the error (i.e. the distance from the center of the path), and experimental results in the figure above show that the RSLK does not linearly predict where the RSLK is, it was presumed that the derivative controller would not wok correctly due to an imperfect fusion algorithm, even if the controller itself was implemented perfectly.

The two main parts of the proportional controller are the $error$ which we already have in the form of the distance from the center of the path, and the proportional term K_P which had yet to be determined. These two values are multiplied together to form the correction signal which I will designate u. Recall that the speed values that the RSLK's motors can take as input is the integers ranging from 0 to 255. Since the finalized RSLK base speed value was 30, I elected to aim for values of 30-50 as the speed that should be sent to each motor. A value of 50 was chosen for the maximum as to allow for fast turns without losing control or being able to update the RSLK's status quick enough. Thus, the constant of proportionality should be set so that the greatest error produced by the sensor fusion algorithm multiplied by the constant of proportionality should be no greater than 50.

To find an appropriate value of K_P, I reused the setup that I used to record the value of each sensor. However, this time I used my re-implemented program that produced a scalar signifying a distance from the line rather than the raw value from each sensor. Likely due to an error in my implementation of the sensor fusion algorithm, drawing the robot across the path recorded a maximum error value of ± 60000 rather than the expected value of ± 35. This is a non-issue because I can simply adjust K_P to obtain the desired range of ± 20. A table of experimental K_P values and their behavior are shown in the table below.

K_P Value	RSLK Behavior
0.00030	Found by dividing 20 by 60000. Movement extremely erratic and jumpy.
0.00005	Car turns fine on straight but sometimes fails to turn enough on donut track.
0.00015	Car may overshoot on straight track and looks wobbly from above.
0.00010	Car reliably correctly traverses both straight and donut tracks.

Table 4: Behavior of various Proportional Constant Values

The rows are listed in the order in which I tried the corresponding proportionality constant, so the final iteration of my program has $K_P = 0.0001$. The result of the applying the PID function to the error can be modeled as the following function, where u is an integer that the car should add to the base speed of one of the RSLK's motors to aid it changing its direction.

$$u = \lfloor K_P \cdot r \rfloor$$

In the function, r is the value of the sensor returned by my program as defined by the function in one of the previous pages. Depending on the sign of u, its magnitude would then be added to the base speed of 30 for the motor that needed to be made faster in order to turn in a given direction. The behavior of the program can be modeled as follows. In the function below, L is the speed of the left motor, R is the speed of the right motor, and u is calculated as shown above.

$$\begin{cases} Add\ |u|\ to\ left\ speed, set\ right\ speed\ to\ 30, & u < 0 \\ Add\ |u|\ to\ right\ speed, set\ left\ speed\ to\ 30, & u \geq 0 \end{cases}$$

Now, the RSLK can successfully follow both the straight-line and donut paths. However, the one missing bit of functionality is the ability to perform a donut when encountering the solid black line at the ends of each of the paths. The implementation of a modular piece of donut code as discussed in a previous section, so now the only remaining issue was to figure out how to detect the presence of the solid black line.

Because all of the sensors would report solid black, their normalized values would all report 1000, and since the weighting on the left and right of the array are the same albeit with opposite signs, the RSLK would have detected that it was centered on the path, not at a turnaround. Instead of reading a scalar from the sensor fusion algorithm, I looked at the raw values reported from each sensor. Recall that there are eight sensors underneath the RSLK, and that each sensor can take on an integer value from 0 to 2500, inclusive. Thus, the maximum attainable value by the sum of all sensors is 20000. Such a large value would not be attainable under normal circumstances as the path should always be perpendicular to the sensor array, but turnarounds are parallel to the sensor array. I thus had to arbitrarily choose a threshold for the sum of all sensor values which, when reached, would signal to the car to go into donut mode and spin around 180 degrees. A table of tested threshold values is shown below.

Sensor Sum Threshold	Comments
16000	Seemed fine during testing, failed on dark flooring during race day.
18000	Adequately detects turnaround on dark floorboards.

Table 5: Experimental Turnaround Threshold Values

For most of the development lifecycle, my sensor sum threshold value was fixed at 16000. This value was obtained by looking at my maximum values for the raw sensor values. This value worked fine while testing the car, but on race day, I was in a different room with

darker flooring. This made it so that the RSLK would incorrectly detect the presence of a turnaround. Thus, I raised the threshold value to 18000 which alleviated any issues.

The completion of the donut functionality marked the completion of the robot's functionality. All the subsystems were tested and verified to be operational. I performed a couple of practice runs on every possible starting position and verified that the RSLK would run correctly. All that was left to do is wait until race day.

Results and Discussion

Test Discussion

In Table 1, one may notice that between the second and third tests, the base speed of the robot was reduced by over 50%. This is because I was debugging the robot's behavior by printing its current state (e.g. the direction it should move in, the values being sent to the left or right motors), and thus it had to be connected to the computer at all times. If the robot were moving with a speed of 70, then it would be difficult for me to keep the computer in my hand while following the robot, so the speed was reduced to 30. Given that we are not graded on the time it takes to complete the tests, this was an acceptable tradeoff and it was not worth the debugging work necessary to bring the speed back up. This table was useful as now I knew that the robot would move in a straight line given equal speeds sent to both motors.

In Table 2, the time for the robot to complete a 180-degree turn at speed 70 on both wheels was recorded. In the end, it was found that 750 milliseconds of "donut time" is close enough so that I feel confident that the robot will be able to resume its path sensing activities after it returns from the donut state to the normal state. Finding the timings necessary to perform a "U-turn" was helpful for the project because it meant I did not have to find the path again after entering the donut state. While I could have tried to time faster turning speeds, I felt this was not useful as we are not graded on completion time.

In Figure 6, two noticeable aspects of the graph are that it has sharp peaks corresponding to when the darkest part of the path passed underneath that sensor, and that the maximum value of each peak is different for each sensor. This figure was necessary for the project because I needed to create a linear regression model that would map the values of each of the eight sensors to the RSLK's distance from the center of the path. It was expected that the sensor's peak values would be different, so I did not spend any time refining my measurement techniques to obtain more uniform peaks.

In Figure 7, the normalized sensor readings are shown. In this graph, I also "trimmed" off the tailing edges of the rightmost and leftmost sensors to aid in the creation of the regression line. Had I left these edges in, the coefficient of determinations would have been much lower for every weighting scheme. This figure was useful because I could verify that normalization algorithm was functioning correctly. No further work was necessary.

In Table 3, there are various weighing schemes that serve to map the normalized values of the eight sensors into a single scalar value. It was useful because now we only must consider one value instead of eight. Given the immense number of permutations and weighing schemes, it

does not make sense to further investigate weighing schemes with a higher coefficient of determination.

In Figure 8, one may notice the plentiful peaks and troughs in the graph. This is a result of the sensor array not detecting the lighter gradients of the path and only detecting the darkest part of the path. Because of the sharp peaks in the raw and normalized values, no matter what weighing scheme was chosen, it would be impossible to create a nearly linear mapping, and the graph that is seen in the figure was implemented in the final project. Some solutions were proposed to me, and all of them failed. I tried re-printing the path used to calibrate the robot and ensured that the sensor array was about the same distance off the ground as it was in a reference image sent by Dr. Briggs. Due to time constraint and obligations from other classes, I eventually resigned to continuing with this flawed albeit working model. This model was perhaps one of the most important parts of the project as it enabled my robot to adapt to imperfections in its current direction. It was not worth investing additional time into creating a better model as the tests performed earlier lead me to believe this is a hardware issue rather than a software one.

Table 4 contains the proportional constants that were tried over different iterations of my software. This was useful in translating the value that the sensor fusion function spit out into a positive integer that would then be added to the motor speed in order to coax the RSLK to move in a direction. Given that I tried the final value of 0.0001 on all eight starting positions and succeeded in each of them, it is not worth investing more time into fine-tuning the value.

Table 5 contains the threshold value for the sum of each of the raw sensor values. If the sum of each raw sensor value exceeds the threshold value, then the robot will enter a donut state where it turns 180 degrees and then continues in the opposite direction. I conducted the race-day tests in a different room than when I was developing my program and the race-day room had a darker floor. Thus, the reported values of the path during race were higher than in the development room, and this meant that the robot would erroneously detect a turnaround where the path crosses over itself in the donut track. Increasing the threshold eliminated this false positive. Given this value worked for all the required tracks on race day, there is no point in further optimizing this property.

Race Day Discussion

On race day, each person was given five minutes to complete both tracks. Completing one track means completing two successful runs from starting positions chosen arbitrarily by Dr. Briggs. In total, one must complete four runs from different starting positions. If the person fails to complete these four runs in the allotted time, they can go again after everyone else has had a turn. The time it takes for a RSLK to complete a given run does not count for or against a person’s score, only completion matters.

On race day, I finished the two runs on the straight track without any issue. The robot managed to stay between the guidelines on its outgoing journey the entire time during both runs. However, I experienced multiple issues on the donut track. On the first run of the donut track, my robot erroneously turned around when it encountered the part of the path that crosses itself. This was fixed by increasing the turnaround threshold from 16000 to 18000. Another issue had to do with the printout of the donut track. Despite following instructions while printing the track,

my track had parts where edges that were supposed to connect with the track on an adjacent piece of paper were not connected properly.

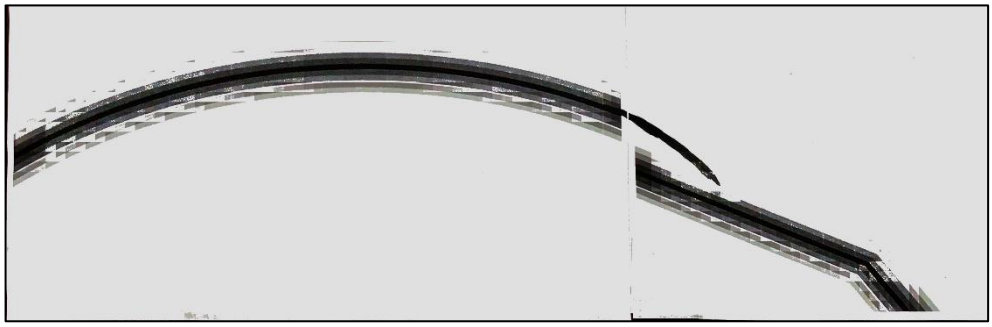

Figure 9: Example of the Glitched Track Printout and its Correction

This caused issues for the car where after transitioning to a disjoint piece of paper, the car would be so far off the track that it could not see any part of the track and thus would continue in a straight line off the track. By this time, I had run out of my initial 5 minute allocation, but I was given another chance near the end of the period Other students in the class had a similar problem which they solved by either printing the donut track out at FedEx or taping over the entirety of the track with electrical tape. The FedEx option was quoted around $7 which was too expensive for my taste and covering the entire track in electrical tape would use up almost all my supply. Thus, I elected to use a sharpie to connect disjoint parts of the path which solved the issue of my car getting lost. This marked the completion of the four required paths

One notable limitation of my code is its predicted inability to correctly function at higher base speeds. The base speed of the robot is the speed at which it moves in a straight line, and that speed is currently set to 30. Having a base speed faster than was predicted to not work because of the lack of the linearity in the sensor fusion algorithm. Because of the peaks and troughs, the robot would become extremely wobbly as it would adjust its speed sent to either of the motors based on the perceived distance from the center of the path. Additionally, these peaks and troughs prevented me from implementing a derivative controller which would aid the RSLK in traversing the path at high rates of speed. Additionally, if we were to test this car on tracks that have varying curvatures, it may not function correctly. Without the derivative controller, the RSLK would arguably not react fast enough to sharp turns. Given hardware that was more responsive to slight changes in reflectance, I could have created a better model with a more linear result. This in turn would have enabled me to implement a derivative controller which could better respond to arbitrary varying curves. Additionally, the car does not contain any path recovery functionality, meaning that if the RSLK completely loses track of the path, then it cannot “back up” and recover.

If I were to conduct this project again, I would first be adamant that I had functioning hardware and paths. I would print out the path at FedEx to ensure darker paths and a smoother transition between pieces of paper. Given that the hardware would detect gradients on the path correctly, I could implement a more linear model which could more accurately predict the true distance from the center of the path. With that, it would be possible to implement the integral and derivative parts of the PID controller. Adding the derivative controller would allow for the robot

to accurately traverse both sharp and gradual curves. Currently, I only have the proportional controller due to the inaccuracy of my model. Using a derivative controller on that model would result in undesirable movement as the rate of change of the distance from the center of the path would be incorrect. Furthermore, with the knowledge I have of the RSLK's internals, I could aim to push the normal operating speed of the robot, despite speed not being a graded category, it could be a good test of skills considering that I have completed the class.

Conclusion

I believe that with a properly functioning sensor array, many tasks such as the creation of the model and the proportional controller would have been much easier. While I am disappointed that I could not implement the derivative controller nor travel at a high rate of speed compare to my classmates, considering that the speed is only for bragging rights and does not affect the grade, I am fully content with the performance of my RSLK.

Time (and willpower) permitting, I would like to implement a derivative controller for the robot as well as have it be competitive with the times that my peers were obtaining. This can be tested by simply verifying that the robot successfully completes the two tracks. If there is no success, then I can simply repeat the procedure described in the testing methodology section albeit with these new values and parameters. I would also like to obtain a replacement sensor array to be able to determine whether the cause of the non-linear model is the hardware of the RSLK or the path that it was calibrated on.

References

[1] https://www.ti.com/tool/TIRSLK-EVM

[2] Stafsudd, O.M., EE3 Introduction to Electrical Engineering Laboratory Manual.

[3] https://courses.lumenlearning.com/boundless-physics/chapter/rc-circuits/

[4] https://ccle.ucla.edu/pluginfile.php/3349025/mod_resource/content/0/ECE3.zip

[5] https://ccle.ucla.edu/pluginfile.php/3403260/mod_resource/content/0/Basic_Code.ino

[6] https://www.csimn.com/CSI_pages/PIDforDummies.html