Final Report

EEE3099S

Group 8:

Friso Vijverberg (VJVFRI001)

Liam Breytenbach (BRYLIA002)

Julian Dower (DWRJUL001)

Table of Contents:

Table of Figures:

List of Tables:

Introduction:

The rapid progress in autonomous systems has led to remarkable innovations that have redefined the boundaries of what machines can achieve. This report goes into the design of an autonomous mobile robot engineered to navigate through unknown mazes using line tracking, object detection, distance computation, and path optimization. By combining components such as line sensors, ultrasonic sensors, and Arduino chips, this report demonstrates the connection between rule-based artificial intelligence, sensors, electronics, and robotics.

The primary objective of this project is to demonstrate how a robot can intelligently manoeuvre through a maze, accurately detecting objects along its path, calculating their distances, identifying the nearest object once all the objects have been located and ultimately navigate back to the closest object that was detected. This report illustrates the remarkable potential of robotic systems in accomplishing complex challenges and shows how to integrate the skill sets acquired over previous semesters into solving this design problem that resonates deeply with the field of Mechatronics.

This report details the entire robot design form a hardware and software perspective. This includes the electrical design and microcontroller resource allocation, in addition to the design, implementation and results of the software created for the project. This is split into sub-sections, including motion control, line tracking and hunt solving.

System Level Design Overview:

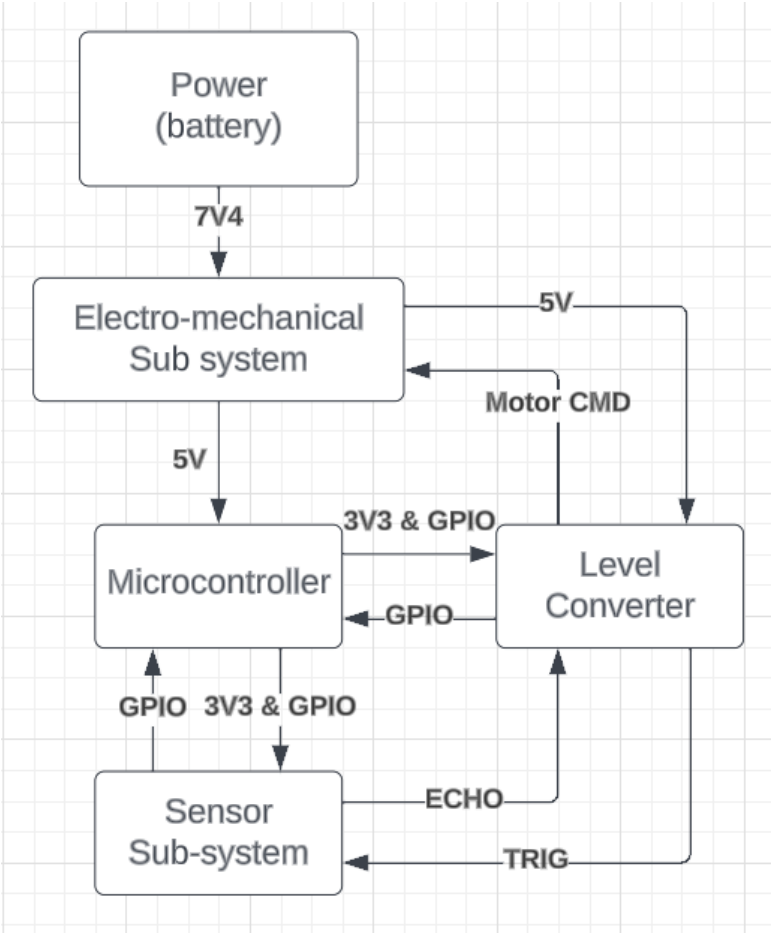

Figure 1: Electrical System Overview

Electrical and Veroboard Design

Our physical design was mostly prebuilt and set by the outline given to us in the project brief, from which our only deviation was the exclusion of the motor encoder. We found, through initial testing that the onboard encoders were unreliable and so we turned to another, more robust, method for tracking 90 and 180 degree turns. This method is detailed in the Motion Control section of the report.

The Veroboard design was initially drawn and planned out on KiCad, then sketched on a piece of lined paper which helped us easily visualise and plan the exact locations of each cut and bridge. The bridging wires were cut to an exact length to keep a low profile and reduce the rat's nest. Electrical tape was also used to neaten the wiring and improve overall presentation. We mostly avoided the use of breadboard jumper wires as these are prone to come loose and break contact, something we unfortunately experienced on the day of our demonstration. Although jumpers are useful in the early phases of breadboard testing and troubleshooting, it would be best to avoid them entirely in a future redesign by directly soldering wires.

Programming

The approach for the programming and logic was with an emphasis on robustness and modularity in mind, leading to a final design with self-correcting behaviour that resulted in our robot never losing track of the line. The line tracking is controlled by a purely proportional controller (P = 0.87) which outputs a constant angular velocity in the direction opposite to the line sensor which first senses the environment. Throughout line tracking there is a constant velocity (V = 0.14 m/s). This helped increase the speed and fluency of the robot as it traversed through the maze.

The decision making and maze solving logic of the robot is contained within a Stateflow chart. This Stateflow chart has 2 sections: Exploration and Hunting. The ‘Exploration’ section contains all the states to explore the entirety of the maze and measure the closest object. The ‘Hunting’ section contains logic to return to the closest object that was measured. The condition out of the 'Exploration’ super state is the detection of the finish line. Note that the two sections have similar logic, and mostly differ in objective.

Within these sections there are substates for left and right turns, junctions, dead ends and detect and finish lines. All the preprocessing required to recognize these states is calculated through logic gates. While early iterations used MATLAB and Simulink embedded function blocks to achieve the same task, these functions were replaced as they added unwanted delay and complexity to our design.

Testing and Debugging

With respect to sub-system testing, a segmented design approach was undertaken. Each sub-system and sub-sub-system were individually simulated and tested to fully understand the individual sections of the model. This improved the speed of debugging and allowed for an efficient workflow.

Debugging was the most time-consuming part of programming the robot. The ‘Test and Debug’ feature in the Arduino toolbox allowed us to run our program with the microcontroller still connected over USB. This helped in quickly identifying issues that would otherwise take very long to find. It allowed for viewing of variables in real time as well as the current state within the Stateflow chart.

Electrical Design

Requirements

- Level shifter between GPIO pin (3V3) and electronics operating voltage (5V)
- Motor supply must be greater than 7V

Specifications

- Battery voltage: 7.4V (nominal), 8.4V (maximum)
- 5V voltage regulator (78M05) on H-bridge with a maximum of 500mA driving current.
- 3V3 internal voltage regulator on Arduino with a maximum of 150mA driving current.
- GPIO input/output voltage: 3V3
- Level shifter (HV: 5V to LV: 3V3)

Power Calculations

Table 1: Maximum Power Consumption for Major Components

Component	**Quantity**	**Power Consumption (W)**
Line Sensor	4	0.033 x 4 = 0.132
Ultrasonic Sensor (HC-SR04)	1	0.075
H-Bridge (L298N) *	1	5.52
Level Shifter (using BSS138)	2	1.44 x 2 = 2.88
Microcontroller (SAMD21G18)	1	0.099
Total	-	8.706

*Includes power output to motor

Power Supply Design

The power sub-system was designed to accommodate 3 voltage levels:

- 7V4 (nominal) to drive the motors act as Vss for the H-Bridge.
- 5V to power the Arduino, in addition to the ultrasonic sensor and the 6 pins between the H-bridge and the motors (EN and IN).
- 3V3 for the line sensors and Arduino GPIO pins.

As a result, two voltage regulators are required. To obtain a steady 5V line, the 78M05 onboard voltage regulator on the H-Bridge was used. This regulator had a maximum driving current of 500mA, which is sufficient for the car. The major advantage of using this regulator was that it had a maximum voltage of 46V. Thus, the entire board was extremely well protected from high voltages.

Additionally, the internal voltage regulator on the Arduino was used to obtain a steady 3V3 line. While this regulator had lower driving current of 150mA, it was calculated to be sufficient to drive the line sensors and all the required GPIO pins.

Through the use of these onboard regulators, no additional protection circuitry was required, simplifying the design. Fig. 2 and 3 demonstrates the required connections to successfully implement these voltage regulators.

Other than over-voltage protection, reverse polarity and under-voltage protection circuitry was considered. However, due to the increased complexity and power drawn by these devices, it was elected not to include them. As such, extreme care was taken when putting the batteries in, and voltage levels were checked regularly.

Microcontroller Resource Allocation

Table 2: Arduino GPIO Allocation

Arduino Pin	Use	Input/Output
A0	Left Outer Line Sensor	Input
A1	Left Inner Line Sensor	Input
A2	Right Inner Line Sensor	Input
A3	Right Outer Line Sensor	Input
D2	Ultrasonic Trigger	Output
D3	Enable B (Right Wheel)	Output (PWM)
D5	Enable A (Left Wheel)	Output (PWM)
D6	IN4 (Right Wheel)	Output
D7	IN2 (Left Wheel)	Output
D9	IN1 (Left Wheel)	Output
D11	IN3 (Right Wheel)	Output
D12	Ultrasonic Echo	Input

- Memory Usage:
    1. Program: 26.24 kB (10%)
    2. Data: 6.90 kB (21%)

Design and Schematics

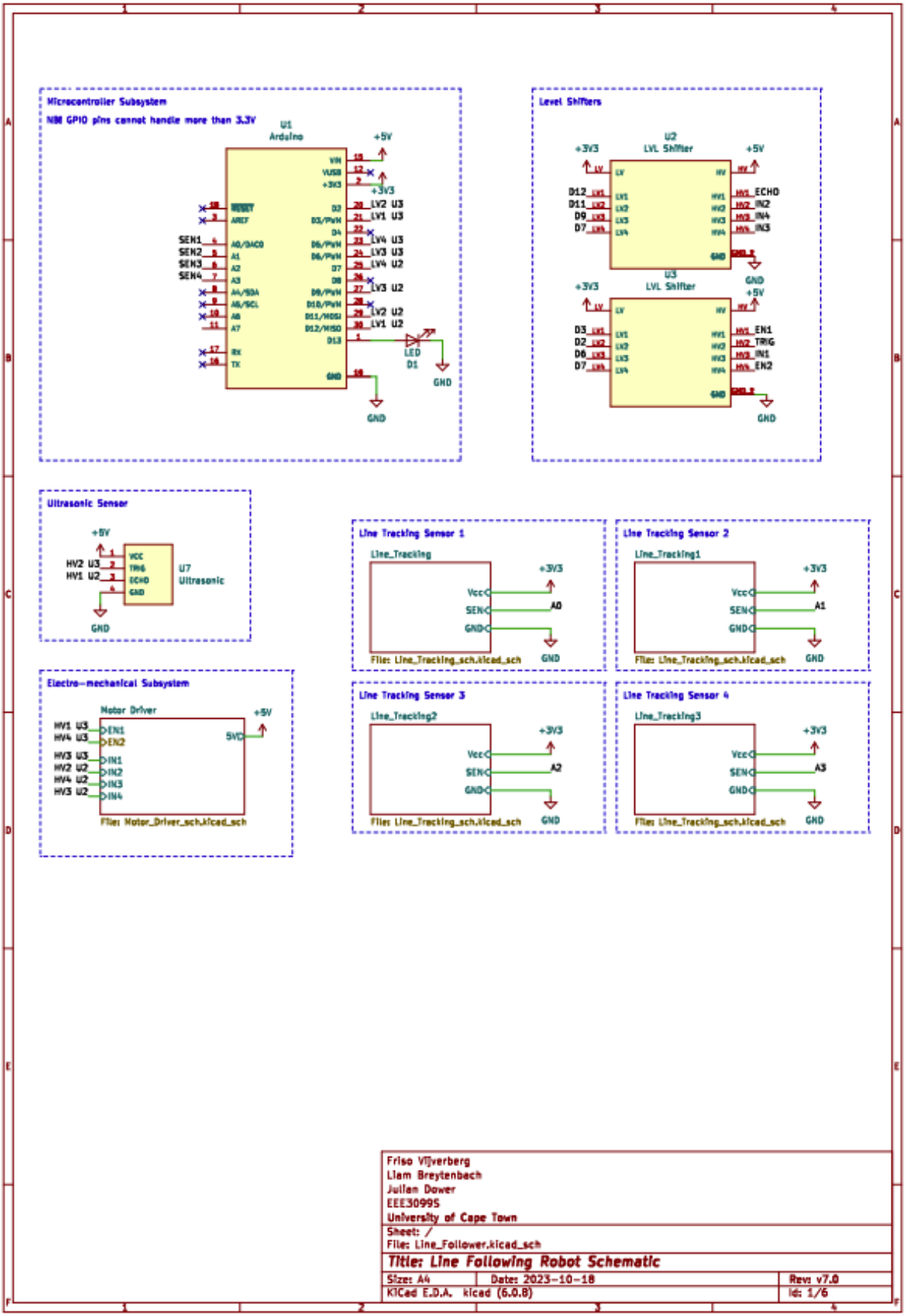

Figure 2: KiCad Schematic demonstrating Veroboard Connections

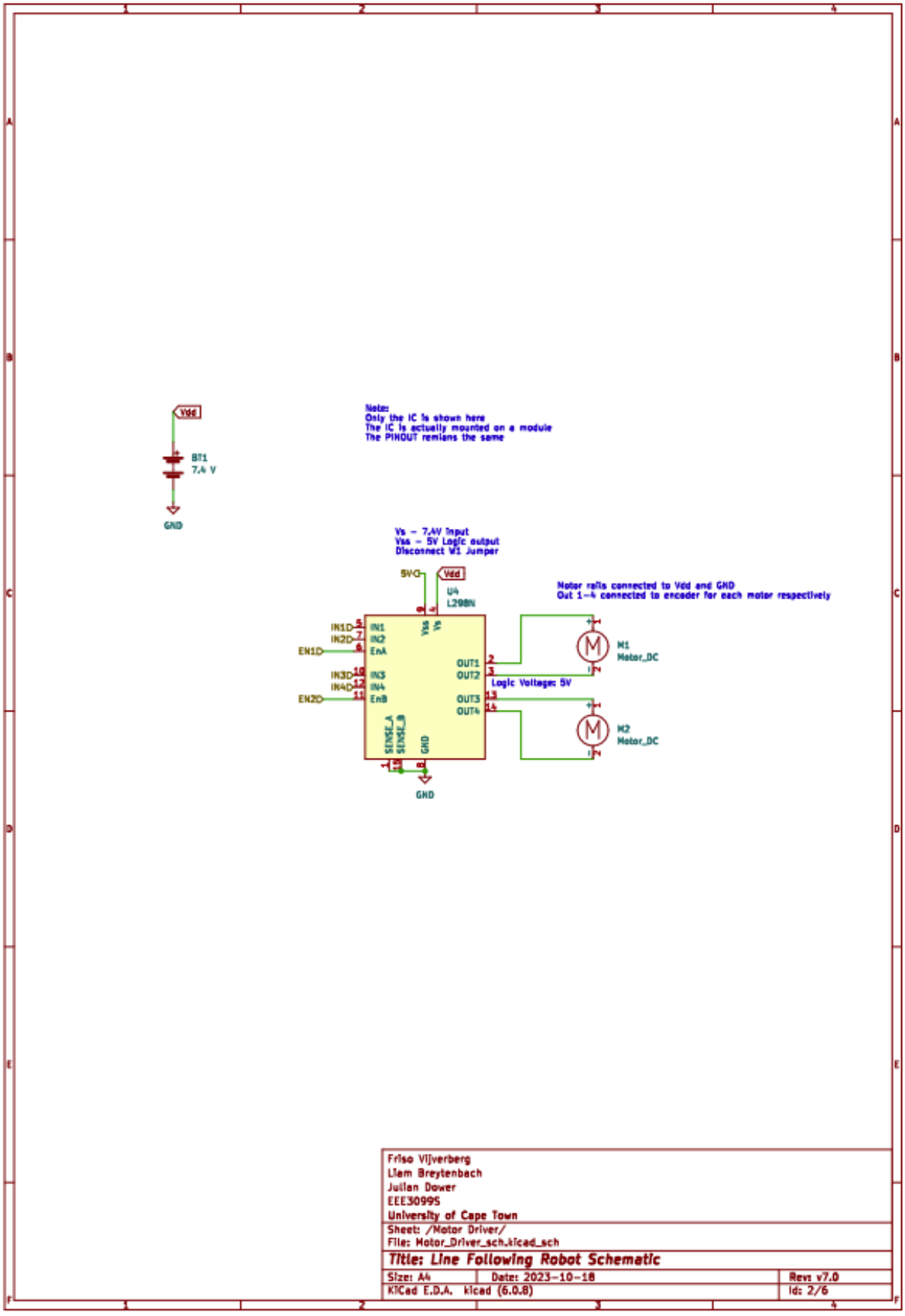

Figure *3: Motor Driver Schematic*

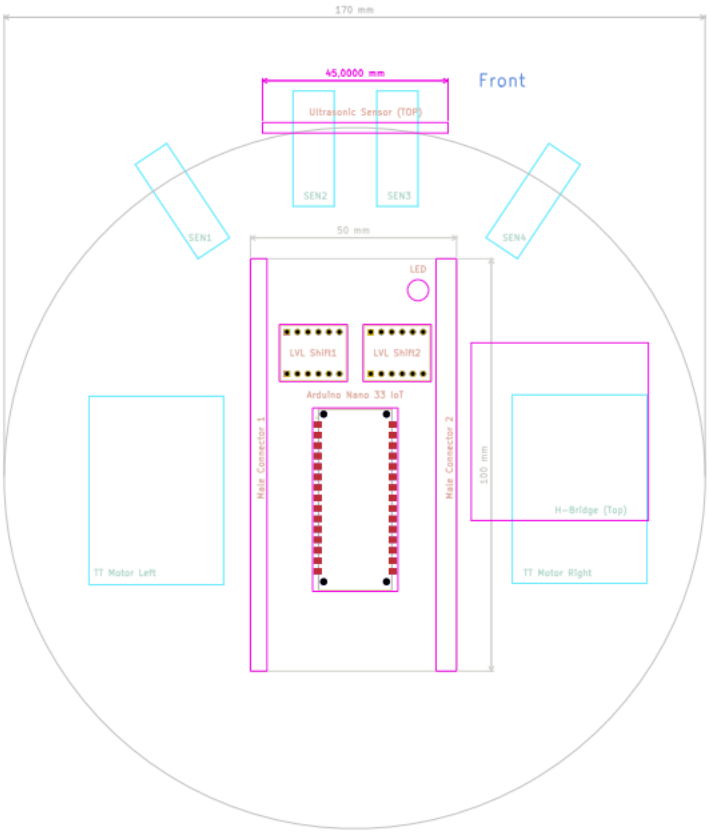

Figure 4: Top View of Physical Layout (Ideal)

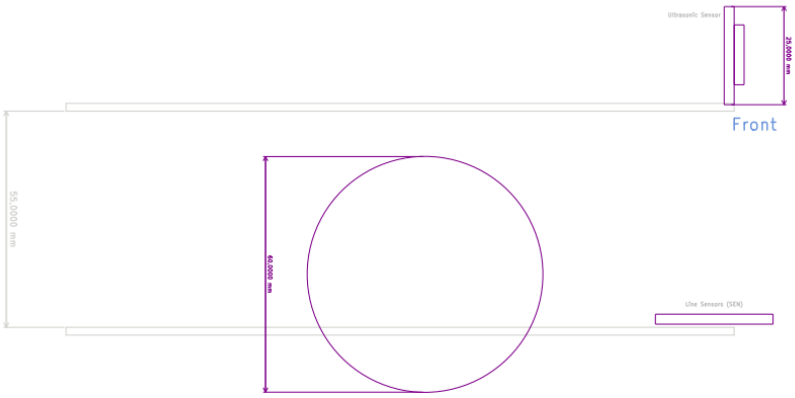

Figure 5: Right Hand View to Demonstrate Ultrasonic Sensor Placement

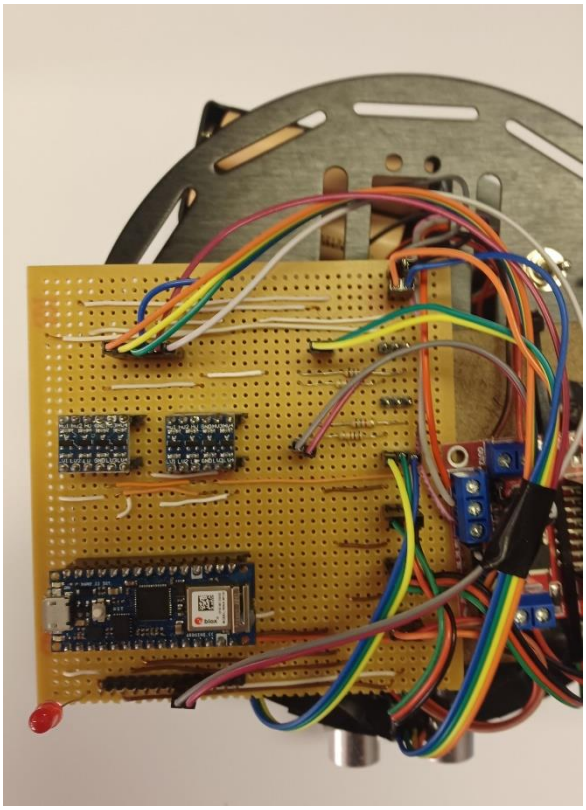

Figure 6: Photograph of Veroboard to Demonstrate Component Placement

Table 3: Stateflow Inputs and Outputs

Name	Description	Input/Output
Dead End	All Sensors on White	Input
Junction	Outer Sensors on Black	Input
Right Turn	Right Outer on Black, Left Outer on White	Input
Left Turn	Left Outer on Black, Right Outer on White	Input
Turn Direction	Direction of Last Inner Sensor to be on Black	Input
Inner Flag	Left or Right Inner Sensors on Black	Input
Inner Flag Trigger	Rising edge of Inner Flag	Input
vIn	Constant Linear Velocity	Input
wIn	Angular Velocity used for Line Tracking	Input
Distance	Measurement from Ultrasonic Sensor	Input
isObject	Boolean Depicting if Distance is Valid	Input
vOut	Linear Velocity of Car	Output
wOut	Angular Velocity of Car	Output
LED	State of the detect LED	Output

Motion Control

Design

The foundation of the motion control system is the algorithm that was used in simulation. This included using the motor encoder ticks to determine the orientation of the car, in addition to a PI controller that minimized the error between the desired rotation and the current rotation. While the approach was sound during simulation, the encoder ticks lagged in reality, and failed to provide accurate data. Furthermore, the PI controller resulted in car rotating extremely slowly when the error was small (while still present).

As a result, the encoders were not used as input data, and the circuit was removed from the Veroboard. Alternatively, the start and stop conditions for any form of motion control was determined by the line sensors. The logic for these conditions is described in the line sensing section. A second amendment to the original simulation algorithm was the removal of the PI controller used during rotation. This controller was replaced by a constant angular velocity during rotation. The turn would be considered completed once the line was detected again.

This method improved the robustness of the model, as the robot would never lose the line. Thus, while the robot may still make errors in decision, these would eventually be self-corrected.

Implementation

Each motor has 3 input pins:

- An Enable pin which receives a PWM signal and controls the speed of the motor
- Two ‘IN’ pins which control the direction of rotation, as shown in the table below:

Table 4: Combination of IN pins for an individual motor

IN1	IN2	Direction
0	0	Forward
0	1	Brake
1	0	Reverse
1	1	Brake

Using a Simulink mask (To wlwr) the linear and angular velocity of the car was converted to an angular velocity for the left and right wheel, respectively. Subsequently, these angular velocities were sent through a look-up table (LUT) to determine the pulse width for the Enable pin. Since this pin requires a strictly positive input, the absolute value was taken. Additionally, the PWM signal was compared to 0, the output of which was sent to IN1. Finally, since braking was not required, the Boolean inverse of IN1 (i.e. !IN1) was sent to IN2.

This implementation is best described graphically, as shown Fig. 7 below.

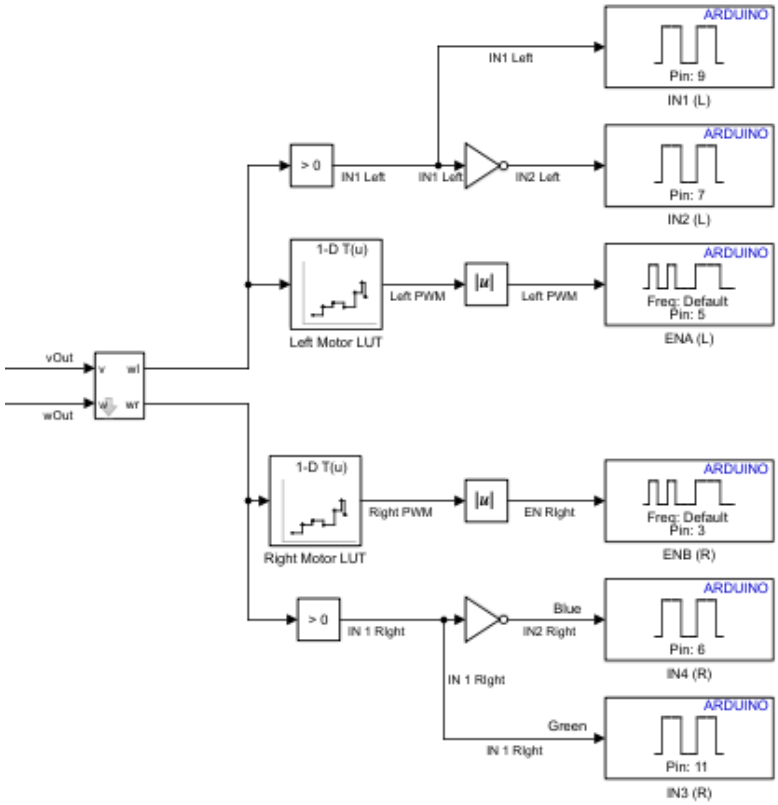

Figure 7: Motor Control Logic in Simulink Model

The linear and angular output velocities were determined by the state that the car was in. Please refer to the Line Tracking section for more information on this.

Results

The motion control algorithm proved mostly successful in the final demonstration. The constant linear velocity during line tracking ensured that the execution was extremely smooth. Moreover, the use of constant angular velocity during rotation, as opposed to a PI controller, decreased the time taken to complete a turn.

An improvement to the algorithm would have been a slightly lower linear velocity during line tracking. This would have allowed for more time for the car to correct itself upon approach at a junction, and hence a lower probability of an incorrect decision being made. However, since there was a competitive aspect to the assignment, there was a fine line between too slow and too fast.

Line Sensing:

Design

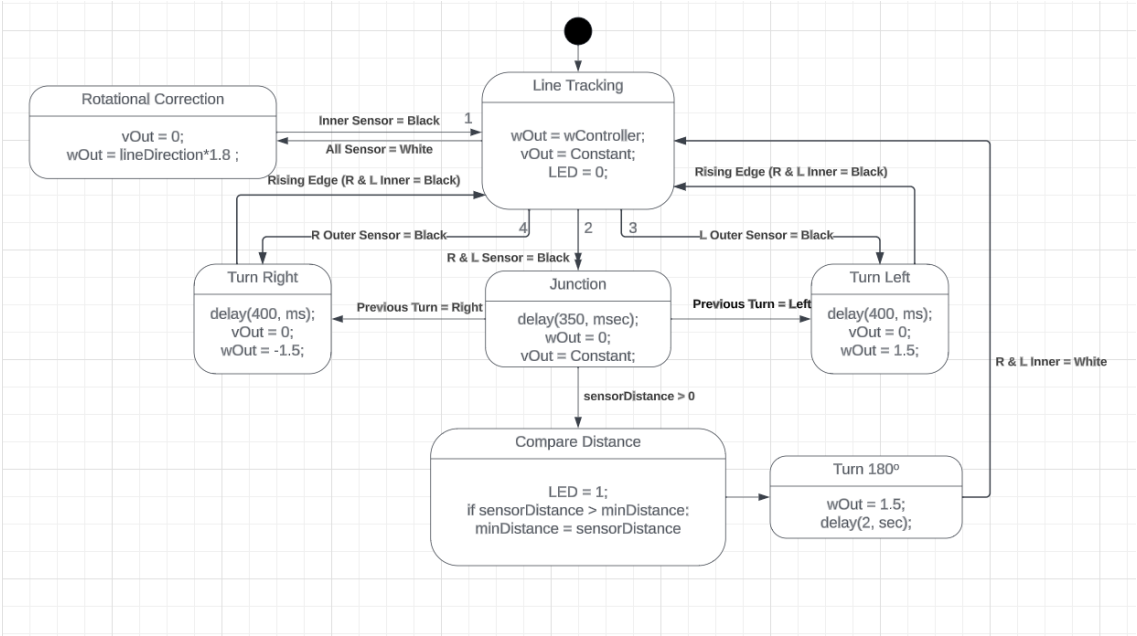

Figure 8: Line Sensing UML Diagram

The line sensing model predominantly relied on the inner sensors, with the outer sensors providing a fail-safe if the deviation from the line was large enough. Thus, examining the model can be split into two sections: An ideal case where only the inner sensors are required and the line is smooth, and the fall-back mechanism when there is a large deviation from the line or a sharp corner.

- Ideal:

The foundation of this design was the simulated model, which used a PD controller to minimize the error between the desired deviation (0) and the actual deviation. The actual deviation was calculated using a weighted sum, as shown in Eq. 1.

$$Deviation = \frac{\sum Sensor\ Readings \times Corresponding\ Weight}{Number\ of\ Sensors\ not\ Detecing\ Line} \quad (1)$$

The corresponding weight of each sensor is shown in the figure below.

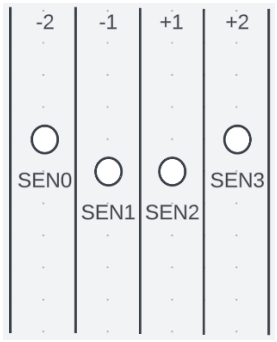

Figure 9: Line Senser Weighting (Initial Design)

However, this design was proven to be impractical in reality, as the outer sensors rarely contributed to the deviation. As a result, only the inner sensors were used in calculating the deviation. Consequently, the controller type was changed to a P controller, as the step change in deviation caused an undesired surge in the derivative part of the controller.

The robot had a constant linear velocity of 0.14 m/s during this state. This allowed for smooth line tracking and an overall increase in speed.

- Fall- back mechanism:

If all sensors where off the line, a 'correction' state would be activated. During this state, the linear velocity is set to 0, and the car rotates about its axis in the direction of the last inner sensor to have detected the line. I.e., if the left-inner sensor were the last to go off the line, the car would rotate in a counterclockwise direction. The opposite is true if the right-inner sensor was the last to go off the line. This state would be active until both inner sensors were back on the line.

The last sensor to have left the line was determined using an enabled subsystem block in Simulink. Pseudocode for the operation of this function is given below:

If (InnerL XOR InnerR == white): % Enable condition

If (innerL == white): % Only one sensor needs to be checked
turnDirection = clockwise % Right sensor was the last on the line.

Else

turnDirection = counterclockwise

end

end

If the subsystem is disabled, the last value is held. This is a key aspect to the successful operation of the block.

In addition to the correction state, a corner algorithm was created for the case where a sharp turn or severe deviation from the line was present. When a corner was detected, the car rotated at a constant angular velocity until a rising edge on the inner sensors was detected. The rising edge was used to accommodate for the case where the inner sensors were already on the line. In this situation, these sensors would need to move off the line and back on it for a turn to be considered as completed. This is demonstrated in the Fig. 10.

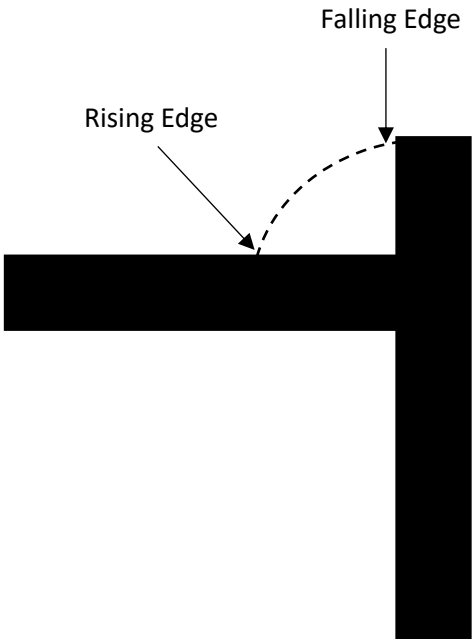

Figure 10: Edge Detection Diagram

Note that for this method to be effective, the car must first drive past the corner. This was achieved by adding a delay to the state, where any actions were undertaken only after the time delay was completed.

The combination of these fall-back mechanism ensured that the model was extremely robust, and the car would never lose the line.

Implementation

The line tracking algorithm relies heavily on the 4 line sensors placed symmetrically at the front of the car (see Fig. 4). These sensors were connected directly to analogue inputs on the microcontroller (pins A0 – A3) and were all supplied by the 3V3 power output line of the Arduino.

Since a Boolean output was required, a threshold on the analogue value was set at 1500 using a 'compare to constant' block in Simulink. Subsequently, the sensor data was sent through various logic gates to determine the state of the line that the car was situated on. A summary of the output of these logic gates is given in Table 3 and shown graphically in the Fig. 11 below. Note that the 'turnDirection' input is not given in the figure. Please refer to the Pseudocode for the operation of this function.

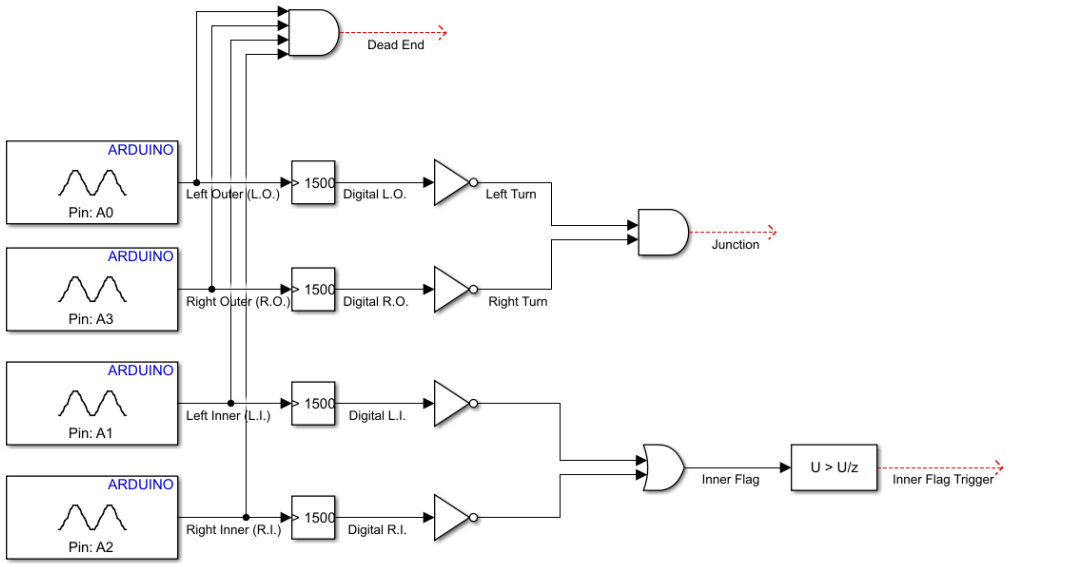

Figure 11: Line Sensor Processing Logic in Simulink Model

Initially, embedded MATLAB functions were used to extract information from the sensors. However, these were replaced with logic gates to reduce complexity and software overhead, thus increasing the speed of the software.

In addition to the information extracted using logic gates, the inner sensors were used to obtain the deviation from the line and ultimately used as error (relative to a desired deviation of 0) in a P controller. This is described in the design subsection.

Results

The line following algorithm was extremely robust – a primary goal of the model, as demonstrated by the car never losing the line, and correcting its own errors. This was a result of the focus on the various fail-safe mechanisms that were implemented.

However, the car did make decision errors at two corners. The root of this this error was a combination of the position of the sensors and an incorrect delay time in the Stateflow chart. More specifically, if a junction is crossed at an angle (where one sensor detects the junction well before the other), the program considers it a right or left turn, and not a junction. This could have been prevented by slowing down the linear velocity of the car during line tracking, which would have given it more time to correct itself upon approach to the junction.

Nevertheless, such an error is also dependant on the track and the starting position of the car. Thus, it would have been difficult to identify the issue on other tracks. Additionally, since changing the sensor position and delay time would have a cascaded effect on another element of the design. For instance, increasing the delay time for which a junction can be sensed (in other words, increasing the time in which the car proceeds to move forward after an outer sensor has been activated) would result in the car possibly leaving the line completely if there was no junction. Hence an emphasis was placed on ensuring the car does not lose the line, knowing the correct decision would be made eventually.

Hunt Solving:

Design

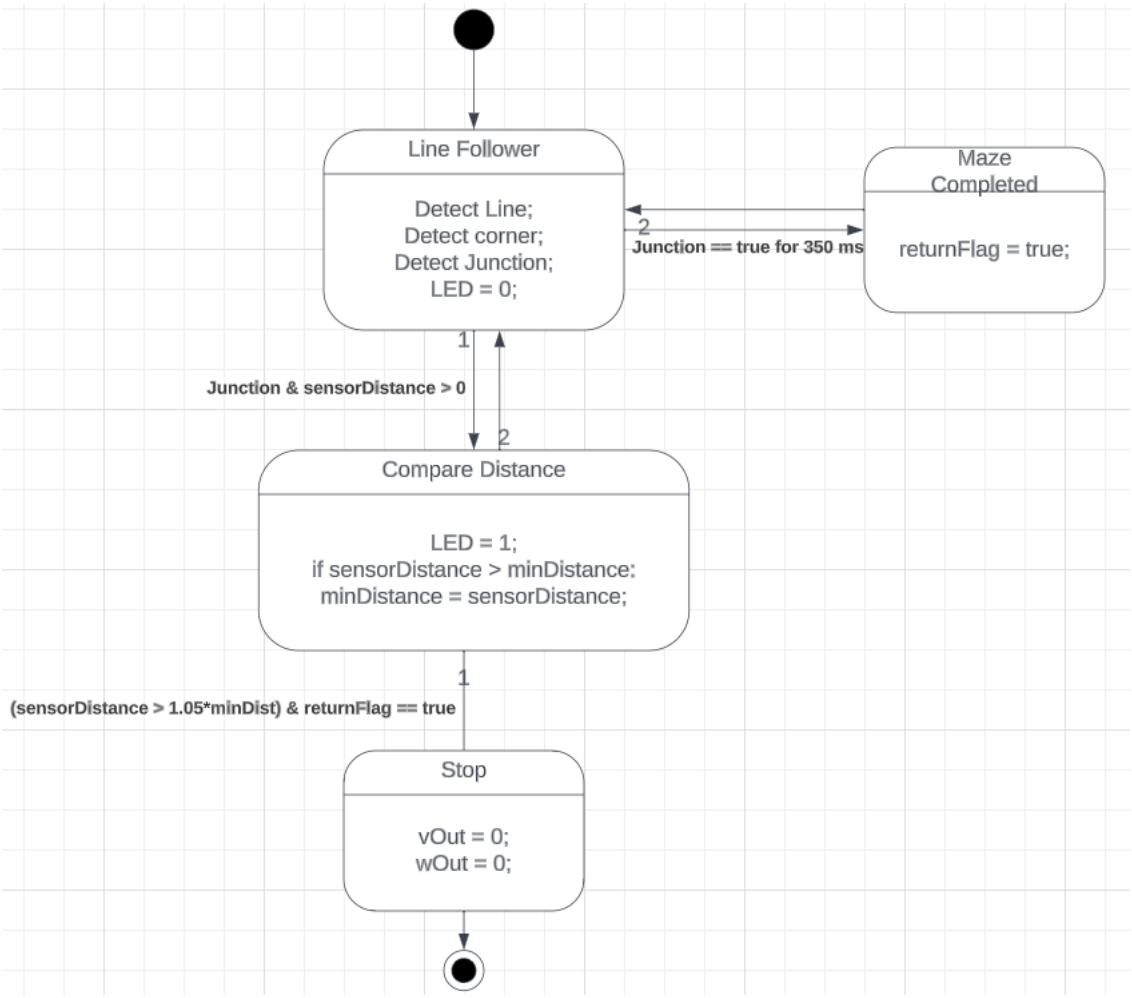

Figure 12: Object Detection UML Diagram

This report defines 'Hunt Solving' as any logic related to ensuring all objects are detected, and the car returning to the closest object from the detect line.

A simplistic approach is used in hunt solving. Such an approach is motivated by the concept that the simplest approach is least susceptible to errors, thus proving to be the most robust. To ensure that the car advanced through the track - instead of returning on the path that it had already driven - the previous turn was stored in memory. When a junction was detected, the car would turn in the direction of the previous turn. For example, if the car had previously turned right onto a branch, then the car would turn right again when it returned from the branch. This is illustrated graphically in the figure below.

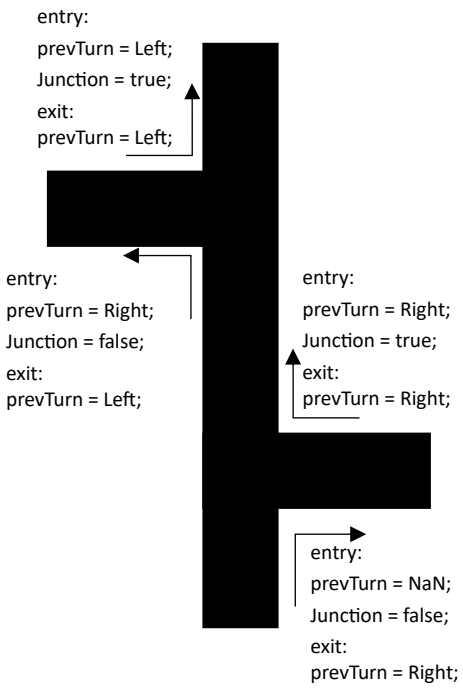

Figure 13: Junction Decision Making Diagram

Thus, the decision at a turn it determined by the state of the 'junction' and 'prevTurn' variables.

A further distinction must be made between a junction (between main path and branch) and a detect line. Since the car does not hold any knowledge of which branch it is on, the two are physically identical. However, a detect line has an object in front of it. As a result, the car checks the ultrasonic sensor to determine if a given line is a detect line or junction and acts accordingly. This is shown in the UML diagram in Fig. 8.

Finally, the car returns to the closest object upon completion of the maze by checking each object again. If the distance recorded on the return path is within 5% of the minimum distance, the car stops. This is demonstrated in Fig. 12.

Such a design model minimizes possible errors, since only one value is stored to determine the location of the closest object. That is, the distance of that object. Since the line tracking is extremely robust, there is minimal chance that the robot is unable to return to that object – even if it misses it the first time.

The model was simple to design due to the modularity of the software. More specifically, the line tracking algorithm, in addition to the method through which the car proceeds through the maze, is extremely similar in this section in comparison to the exploration model described earlier in the report (line tracking and motion control). As a result, these algorithms could easily be adapted and applied to this section, with minimal error due to the extensive testing which was conducted prior. Due to the simplicity of the design, no interrupts or additional functions were called during this section.

Implementation

The hunt solving model has a relatively simple implementation, and consists of 3 subsections, other than the:

- The ultrasonic sensor
- Logic to ensure the car advances through the maze
- Logic to determine the end of the maze

The ultrasonic sensor is configured using a Simulink mask, which handles the distance calculation based on the time difference between the trigger and echo pulses. Each new distance measurement is compared to the current minimum distance, which is replaced if the newer measurement is smaller. Thus, only one value is stored as a reference for the location of the closest object.

To ensure that the car advances through the maze, the previous turn was stored and used when a junction was detected. This variable, labelled 'prevTurn', was created as local data in the Stateflow chart. Since a right turn had priority over a left turn in the chart, in the case where a junction is detected as the first turn, the car would turn right.

Finally, to determine when the end of the maze was reached, a condition was added to the junction state. This is described by the pseudocode below and had a higher priority over other state transitions.

Inside junction state

{

If (junction == true for 350 ms):

% i.e. outer sensors detecting black line while car is moving forward with no angular velocity % for 350 ms

    finishLine == true; % actual variable name may differ

end

}

If this condition was reached, a separate section of the Stateflow chart was activated. In this section, the car would turn around return to each object, checking the distance of each. This is synonymous with having a returnFlag, as described in earlier sections.

Results

The various sections of the hunt solving algorithm had mixed success. To start, the end of maze detection was executed without a flaw. The risk of this implementation was a false trigger of the returnFlag. In such a scenario, the car would turn around and return the closest object up until that point in time. However, this was not the case in reality, and the car successfully identified the end of the maze.

The ultrasonic sensor was accurate in the final demonstration, but false readings due to unwanted noise proved to be a major obstacle leading up to the final model. To overcome this obstacle, an interval threshold was set on the distance reading such that the value would only be considered valid if it were between 2mm and 4m. These boundary values were chosen based of the HC-SR04 detection range, as specified in the datasheet. With this threshold in place, the sensor accurately recorded distances, and the closest object was successfully identified.

The downfall of the hunt-solving model was its efficiency. While other methods could have skipped redundant branches, this implementation went through the maze in the same manner as before – thus checking every object and dead end. The justification for choosing such a model was once more the robustness thereof. By minimizing the sources of error, the probability that the car would return to the closest object was extremely high, especially when coupled with the line tracking model, which would never lose the line.

In summary, the success of the hunt solving algorithm depends on perspective. While the model was not the most efficient, it was the most robust, and required the least memory. As such, the model could be extrapolated to a much larger maze, with an arbitrary number of branches, sub-branches, and objects.

Conclusion:

By integrating various sensors, control algorithms, and programming, this report has demonstrated the steps required to create an autonomous robotic car. Through collaboration and problem solving, the process of designing, assembling, and fine tuning this robot has provided valuable experience that is relevant to anyone in the field of robotics, mechatronics, and artificial intelligence.

By using KiCad to design the PCB, the circuit was successfully implemented. Using a Veroboard, the circuit was built and connected to the robot. Using MATLAB to design the control algorithm, the logic required for the robot to complete the maze was achieved. Stateflow played a major role in this logic, switching between states of line-following, turning, and measuring objects. By integrating the above-mentioned techniques, the robotic car was able to complete the maze, line tracking successfully, accurately detecting, and measuring objects and successfully returned to the closest object once the maze was completed.

This report has illustrated the remarkable potential of robotic systems in accomplishing complex challenges and stands as a beacon of progress and inspiration for future endeavours in the dynamic field of Mechatronics.

Git Repository:

Please click here to view the GitHub Repository of the page.

Bibliography:

[1] R. Santos, "Complete Guide for Ultrasonic Sensor HC - SR04," *Random Nerd Tutorials*, 2019. https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/

[2] "L298N Motor Driver Module," *Components101*, Apr. 13, 2021. https://components101.com/modules/l293n-motor-driver-module

[3] Arduino, "Nano 33 IoT | Arduino Documentation," *docs.arduino.cc*. https://docs.arduino.cc/hardware/nano-33-iot

[4] "Logic Level Converter Bi-Directional - Micro Robotics," *www.robotics.org.za*, 2023. https://www.robotics.org.za/LEVEL-4P

[5] "Line_Tracking_Sensor_for_Arduino_V4_SKU_SEN0017-DFRobot," *wiki.dfrobot.com*, 2023. https://wiki.dfrobot.com/Line_Tracking_Sensor_for_Arduino_V4_SKU_SEN0017

[6] DFRobots, "Turtle: 2WD Arduino Mobile Robot Platform wiki- DFRobot,"

wiki.dfrobot.com.
https://wiki.dfrobot.com/2WD_Mobile_Platform_for_Arduino__SKU_ROB0005_
(accessed Oct. 22, 2023).