SYSC 4805 Final Report

L1-G8
Team Beaver

Course:SYSC4805
Lab Section: L1-G8
Team Name: Beaver
Date: 11/11/2024

Fatima Chughtai 101195748
Aniesh Sabnani 101192026
Aaranan Sathiendran 101196339
Joseph Vretenar 101234613

Table of Contents

Objective

Snow plows are an essential part of the community in the winter, as they significantly reduce the tedious labor and potential safety hazards that come with removing snow from roads, driveways, parking lots, etc. while keeping them safe for driving and walking [1]. To mimic this real world application, the objective of this project is to design and implement a scale model of an autonomous snowplow system within a variety of preset size and performance constraints.

The robot's design will firstly mimic the safety requirements of a real world snowplow by using a variety of distance sensors to accurately detect obstacles and move the robot accordingly to avoid them while also using line follower sensors to remain within a designated plowing area [2]. It will also utilize additional sensors such as wheel encoders for ensuring the plow moves at a safe, regulated speed. Lastly, the robot will use an algorithm to determine the best path to push as many lightweight wooden blocks as it can out of the arena's perimeter in the fastest time possible based on the surroundings detected by the sensors [2].

The key deliverables of this project include: 1) developing a functional prototype of the robot, 2) creating algorithms to ensure the robot can successfully avoid obstacles and stay within the boundary while pushing out as many snow blocks as possible, and 3) creating proper documentation, reports, and presentations on the robot's design, functionality and performance. The project will be considered complete successfully when the robot is able to avoid all the obstacles, stay within the set boundaries, and push out a majority of the blocks out of the area within the allowed operation time.

Overall Deliverables

Below, Table 1 outlines the overall deliverables that will be incrementally completed to ensure the successful completion of the autonomous snowplow project.

Table 1: Overall deliverables for autonomous snowplow project

Deliverable Code	Deliverable Name	Deliverable Description	Deadline
OD1	Project Proposal	Complete document outlining project objective, deliverables, requirements, test plan, schedule, cost, and responsibilities	October 14th
OD2	Chassis Design	Determine placement for plough, boards and sensors, and install components to complete chassis assembly	October 21st
MD1	Progress Report	Complete a detailed progress report that outlines what we have accomplished in the past. This	November 18th

		includes the overall architecture of the system, UML diagrams, working code and more.	
OD3	Presentation	Present motivation behind project objective and explanation of design choices for implemented solution to class	November 20th
MD2	Robot Complete	Have all sensors mounted along with being fully programmed and operational	December 2nd
OD4	Demo	Set up, demonstrate, and explain robot design to TAs and participate in code review session	December 2nd
MD3	Final Report	Complete a report that contains all the information that is in the progress report. As well as, adding results from integration testing.	December 6th

List of Functional Requirements

To ensure the autonomous snowplow meets all the required design goals, its design will satisfy the following requirements:

1. Physical Constraints:
   1.1. The robot's design shall not exceed 226 mm in width, 262 mm in length, and 150 mm in height [2].
   1.2. The integrated plow shall not extend the total width of the robot by more than 50 mm (25 mm in each direction), and shall not extend the total length by more than 30 mm [2].

2. Operational
   2.1. The robot's speed shall not exceed 30 cm/s at any point during its operation [2].
   2.2. The robot shall be able to successfully remove snow cubes from the testing arena without any human intervention or assistance.

3. Snow Plowing
   3.1. The integrated plow shall permit the robot to push wooden snow cubes with a side length of 20 mm outside the testing area [2].
   3.2. The robot shall be able to remove at least 50% of the snow cubes from the testing area within a maximum time frame of 5 minutes.

4. Boundary Detection
    4.1. When the robot detects a boundary to the testing area, the robot shall stop and turn accordingly to prevent the robot's wheels from crossing the boundary by no more than 5 cm [2].

5. Obstacle Avoidance
    5.1. The robot shall be able to detect an obstacle along its path within a maximum proximity of 10 cm.
    5.2. When the robot detects an obstacle, it shall be able to correct its path to avoid making any physical contact with the obstacle while staying in the testing area.

List of Detailed Deliverables

Table 2 below outlines a detailed breakdown that explains how each task will contribute to the incremental development of the autonomous snowplow.

Table 2: Detailed deliverables for autonomous snowplow project

	Deliverable Code	Detailed Deliverables	Description
Milestone 1	RD1	Designing a chassis	Create a 3D sketch of how the snow plow robot will look including component placement.
	WD1	Project Proposal	Complete document outlining project objective, deliverables, requirements, test plan, schedule, cost, and responsibilities.
	RD2	Obstacle detection	Use the various distance sensors on the robot to detect objects along its path within a given proximity.
	RD3	Obstacle avoidance	Use the sensor data from detection to determine the best path forward to avoid the obstacle along the robot's path.
	RD4	Boundary detection	Implement logic to interact with distance and line detection sensors to detect and react to boundaries of the testing area.
	RD5	Speed of Robot	Determine the speed of the robot through the encoder wheel readings.
	RD6	Turning the Robot	Use the Arduino Nano Every to ensure rotation through controlling the motors.

	Deliverable Code	Detailed Deliverables	Description
Milestone 2	RD7	Part Integration	Putting together the robot by attaching all the sensors and the snow plow. Ensuring that chassis follows the size restrictions
	WD2	Progress Report	Create a detailed progress report outlining the milestones accomplished.
Milestone 3	TI1	Unit/Integration Testing	Performing Unit and Integration tests on the system, ensuring that all requirements are met and no liabilities/loose ends exist.
	WD3	Final Report	Create a final report going through all data that was observed through the project along with the solution.
	WD4	Presentation	Present the final product, discussing the events leading up to our final design, including the preparation, objectives, technical aspects and more
	WD5	Demo	Demonstrate functioning robot inside the testing area and explain its functionalities to the TAs

Figure 1 below provides a work breakdown structure diagram that provides a visual interpretation of all the tasks that will be completed during the incremental development of the autonomous snowplow robot.

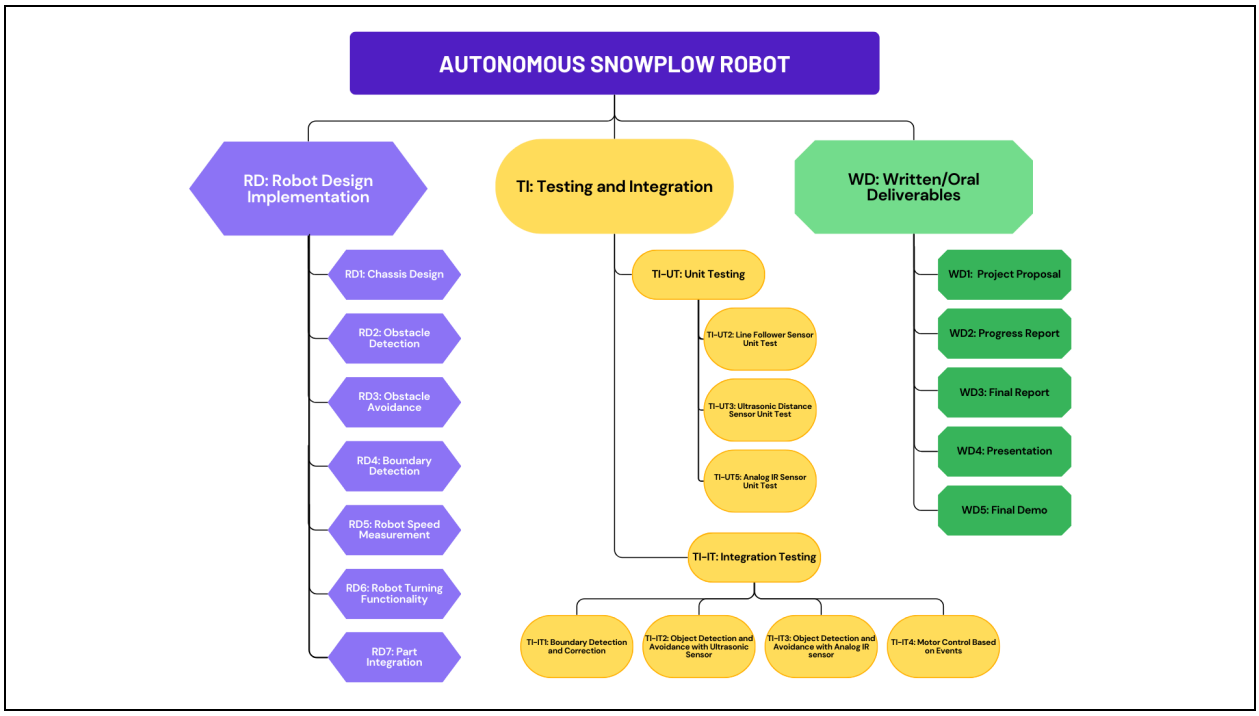

Figure 1: Work breakdown structure diagram for all tasks related to autonomous snowplow

Testing Plan

Quality assurance is a crucial component to the successful implementation of the autonomous snow plow. Table 2 below presents a structured unit testing plan that will be followed to validate the functionality of each individual component of the robot.

Table 3: Unit Test Descriptions and Pass/Fail Criteria

Test	Description	Pass/Fail Criteria
Line follower sensor	Successfully detect a black boundary line and generate an interrupt signal.	Pass: Robot system accurately detects a black line and generates an interrupt signal Fail: Robot system is unable to detect a black line and does not send an interrupt signal
Ultrasonic distance sensor	Successfully detect an object set at a known, close distance in the robot's path, and accurately measure the distance to it using the ultrasonic distance sensor.	Pass: Robot successfully detects an object at a known, close distance and measures the distance to the object accurately within a set threshold Fail: Robot is unable to detect an

Test	Description	Pass/Fail Criteria
		object, or the measured distance to the object falls outside the set threshold
Obstacle detection IR sensor	Successfully detect an object set at a known distance in the robot's path, and accurately measure the distance to it using the obstacle detection IR sensor.	Pass: Robot successfully detects an object at a known distance and measures the distance to the object accurately within a set threshold Fail: Robot is unable to detect an object, or the measured distance to the object falls outside the set threshold

Similarly, Table 4 below provides a breakdown of all the integration tests that will be performed to validate that all the robot's individual components interact with each other properly, and that their combined functionalities satisfy the robots initial design goals.

Table 4: Integration Tests, Descriptions and Pass/Fail Criteria

Test	Description	Pass/Fail Criteria
Boundary Detection and Correction	Successfully redirect the robot using the motors upon detection of a black boundary line.	Pass: The robot successfully detects a black boundary line, and the robot controls the motors accordingly to turn itself and avoid the line without the wheels crossing it by more than 5 cm. Fail: The robot is unable to detect the black boundary line, or its wheels cross the boundary by more than 5 cm.
Object Detection and Avoidance with Ultrasonic Sensor	Successfully detect an object accurately using the ultrasonic sensor, and correct the direction of the robot using the motors to avoid it	Pass: The robot successfully detects an object and accurately measures its distance, and controls the motors to redirect the robot accordingly without making any contact with the obstacle. Fail: The robot is unable to detect the object and/or makes contact with the obstacle
Object Detection and Avoidance with Analog IR distance sensor	Successfully detect an object accurately using the analog IR distance sensor, and correct the direction of the	Pass: The robot successfully detects an object and accurately measures its distance, and controls the motors to redirect the robot accordingly without

	robot using the motors to avoid it	making any contact with the obstacle. Fail: The robot is unable to detect the object and/or makes contact with the obstacle
Motor Control based on Events	The Arduino Due successfully controls the motors in the correct way based on the event sent by a sensor.	Pass: The Arduino Due executes the correct command based on the event received by the sensor. Fail: The Arduino Due executes the incorrect command based on the event received by the sensor.

List of Activities

The following list of activities will be completed to satisfy all the deliverables outlined in the Work Breakdown Structure.

1. Using Fusion360 to design an attachable snow plow
2. Using the tools provided to attach the snowplow and motors onto the chassis
3. Setup the Ultrasonic sensor on the chassis
4. Program the Ultrasonic sensor to detect obstacles for avoidance
5. Setup the analog IR distance sensor on the chassis
6. Program the analog IR distance sensor to detect objects on the edge for turning and avoidance
7. Setup the Line following sensor on the chassis
8. Program the line follower to detect boundaries and correct course
9. Setup the Watchdog timer
10. Setup the state machine for the due
11. Make sure the states correctly change from the sensors
12. Setup motor control based on the current state
13. Testing the integration of all the sensors together to make sure they are working as expected
14. Testing the control of the robot through the sensors

Schedule Network Diagram

Figure 2 below shows the project's schedule network diagram, which states all the milestones and connects them together based on their dependencies. The diagram also shows the expected time each milestone will take. The critical paths are shown using the blue arrows.

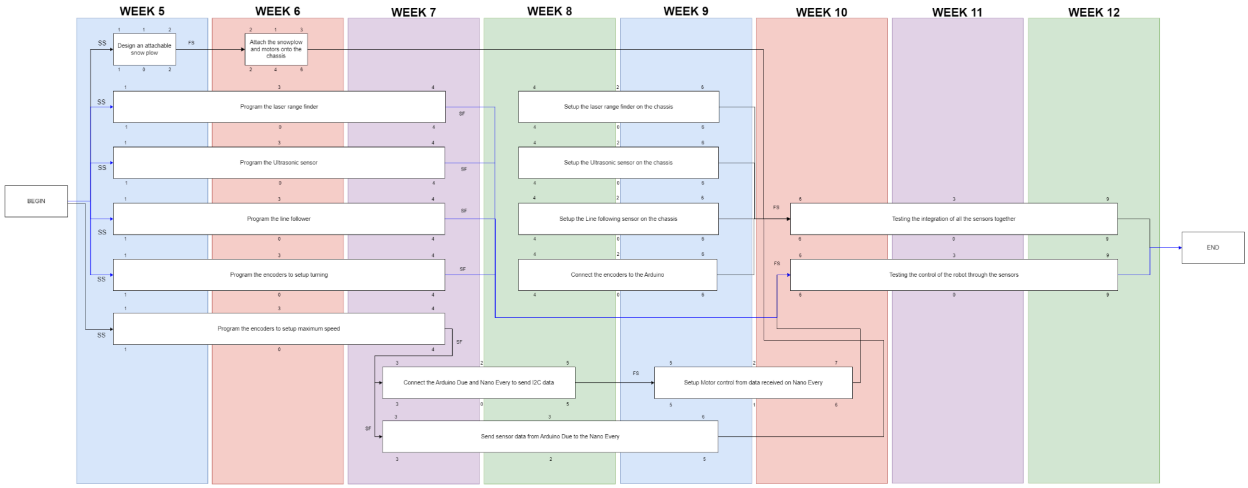

Figure 2: Schedule Network diagram for milestones

Gantt chart

Figure 3 below provides a Gantt chart that displays a timeline following all the previously mentioned activities.

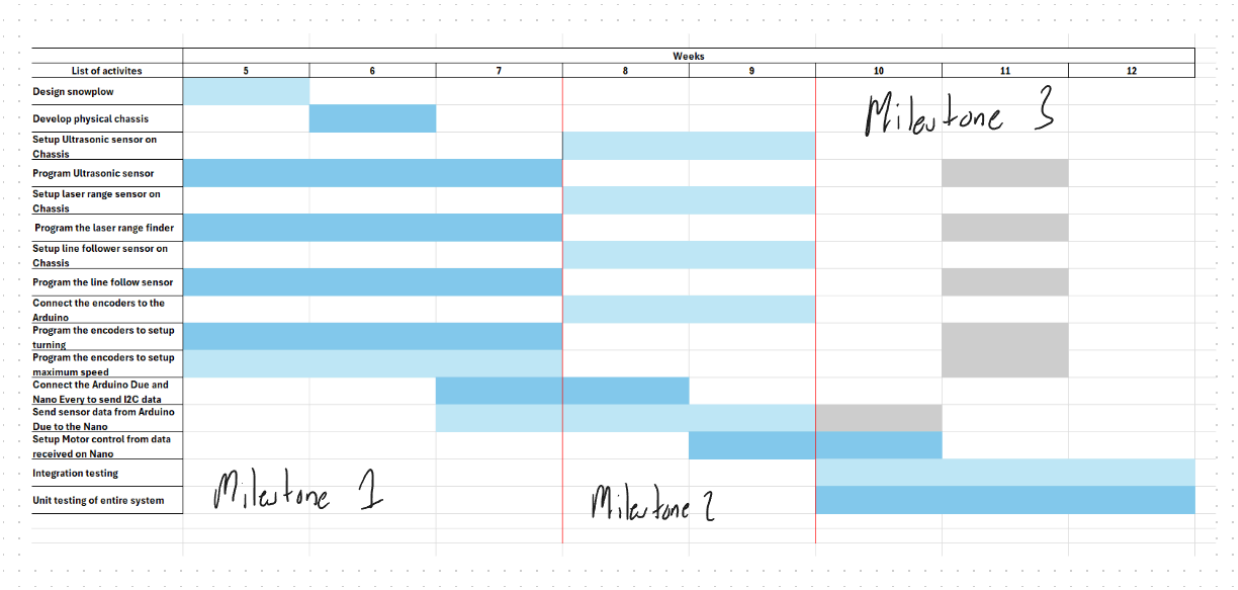

Figure 3: Gantt chart detailing activities, milestones and contingencies and its duration (in weeks)

When observing Figure 3, the red line outlines our big milestones. The first one is having all the sensors and peripherals needed for our robot design to be programmed to meet functional requirements. The second big milestone requires the setup of the programmed peripheral devices on the robot chassis, ensuring that it accurately meets the dimensions of the requirements and our design. Finally, the last big milestone is for testing, crucial to ensure that our system meets all requirements we have set out for.

A one week contingency period was given to development of the peripherals as shown in Figure 1 (the gray blocks) during Milestone 3 where testing is performed, accordingly making adjustments based on the results of the test.

Planned Value Analysis Figure

Cost assessment is a crucial component to determining the feasibility of a solution for any engineering problem. This section provides a concise cost breakdown for the autonomous snowplow project, accounting for the cost of the hardware as well as the work of the developer.

Assuming we do 6 hours of work per week per person, the total cost for this project would be $50/hour * 6 hours/week * 5 weeks total * 4 people + $500 (one-time cost of the course kit at the start of the project), which is about $6500.

Based on this assumption, the planned value for each week can be calculated by adding the cost of work per week per person ($50/hour * 6 hours/week * 4 people = $1200) to the previous week's planned value.

Similarly, the actual cost for each week can be calculated by adding the cost of work per week per person based on the actual number of hours of work completed ($50/hour * x hours/week * 4 people) to the previous week's actual cost.

Lastly, the earned value for each week can be calculated by taking a ratio of the number of deliverables completed that week (i.e the percentage of project completion) and multiplying it by the total project budget.

Table 5 below shows all the values calculated to perform the planned value analysis.

Table 5: Planned Value Analysis Calculations

Week	Funding Requirements/Planned Value (PV)	Earned Value (EV)	Actual Cost (AC)
September 30	$1,700.00	$1,700.00	$1,700.00
October 7	$2,900.00	$2,900.00	$2,900.00
October 14	$4,100.00	$3,792.86	$3,700.00
October 21	$5,300.00	$5,578.57	$4,700.00
October 28	$6,500.00	$5,357.14	$5,700.00
November 4	$7,700.00	$6,250.00	$6,500.00
November 11	$8,900.00	$7,142.86	$7,900.00
November 18	$10,100.00	$8,928.57	$8,700.00

November 25	$11,300.00	$10,714.29	$10,500.00
December 2	$12,500.00	$12,500.00	$12,500.00

Figure 4 below illustrates a planned value analysis figure based on the estimated planned value analysis calculations provided above.

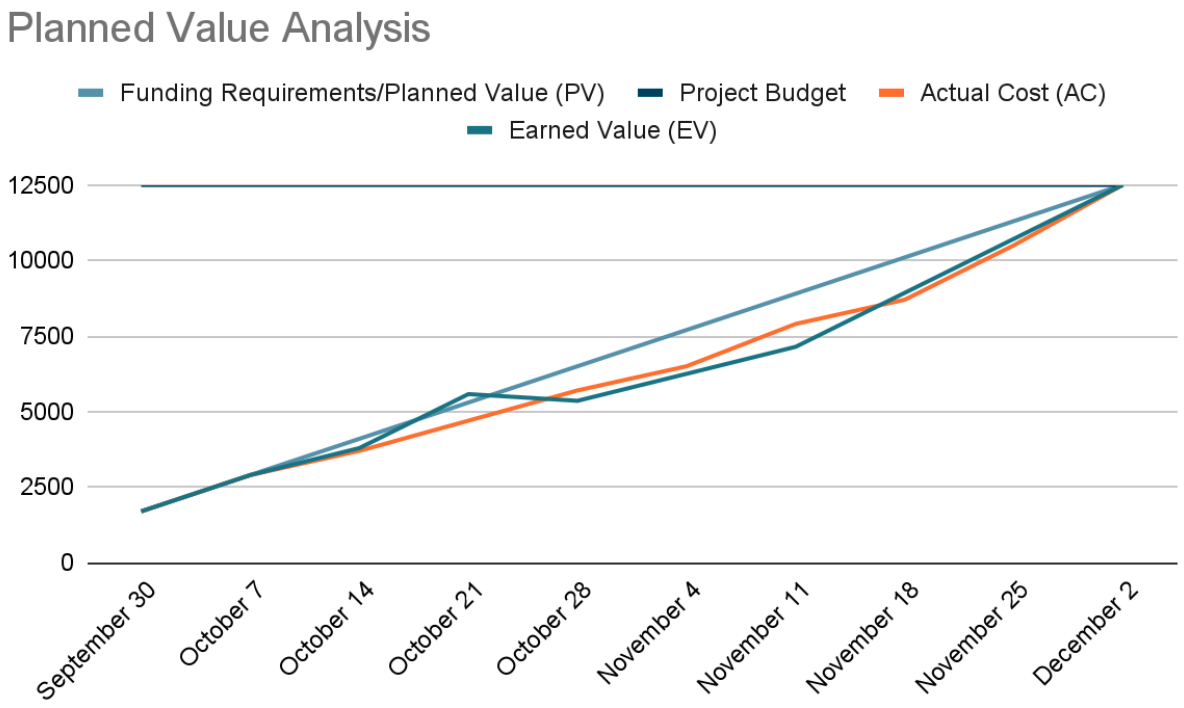

Figure 4: Planned value analysis figure based on funding and dates

Responsibility Assignment Matrix

Table 6 below summarizes the activities that each developer will be responsible for completing to ensure the successful completion of the autonomous snowplow. The activities have been equally distributed to ensure maximum efficiency.

Table 6: Responsibility Assignment Matrix for project activities

Requirements Assignment Matrix		
Project Name:		Autonomous Snowplow
Project Description:		Move snow, avoid obstacles
Responsible	**Approver**	**Requirement Description**
Joseph	Fatima	Design snowplow
Fatima	Aniesh	Finalize robot build
Aniesh	Joseph	Setup ultrasonic sensor
Aniesh	Fatima	Program ultrasonic sensor
Fatima	Joseph	Setup laser range finder
Fatima	Aaranan	Program laser range finder
Aaranan	Joseph	Setup line follower
Aaranan	Aniesh	Program line follower
Joseph	Fatima	Setup Watchdog timer
Joseph	Aaranan	Program turning
Joseph	Aniesh	Program speed
Aniesh	Joseph	Setup state machine
Fatima	Aniesh	Send sensor data for the state machine
Joseph	Aaranan	Setup motor control from data
Aaranan	Joseph	Test integration of sensors
Aniesh	Fatima	Test control of robot from sensors

Architecture of System

The robot is composed of many different components that are all working together in sync. Figure 5 below shows the updated deployment diagram of our system containing changes made since the progress report.

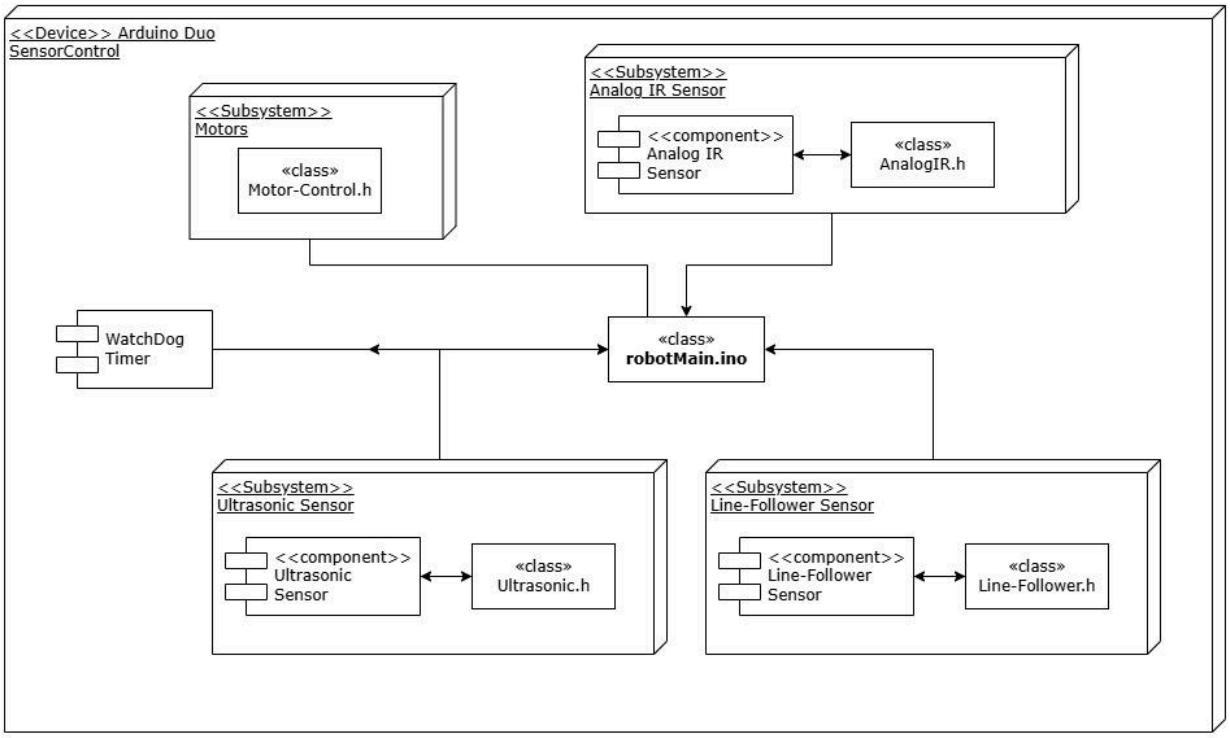

Figure 5: System Architecture Deployment Diagram

As shown in Figure 5, the robot is run using the Arduino Due, which is responsible for all the controls in the robot. The Due holds the system's main code, the watchdog timer, and the subsystems for interfacing with each sensor. Since we have transitioned to a super loop approach instead of servicing interrupt handlers for each sensor, the main code now runs the super loop, which processes the robot's current state based on the changes in values read by each sensor subsystem. The laser range finder subsystem has also been replaced with the analog IR sensor subsystem due to technical difficulties with the I2C interface during integration. The main code then uses the motor control class to control the robot's direction based on the current state.

To implement the described system, Joseph implemented the motor control subsystem and watchdog timer. Aniesh implemented the ultrasonic sensor subsystem, Aaranan implemented the line follower sensor subsystem, and Fatima implemented the analog IR distance sensor subsystem. The subsystems were then integrated in robotMain.ino with the help of all group members during labs 9 and 10.

State Chart

As the system now uses a super loop instead of interrupt handlers, the robot's behavior now follows the new state machine illustrated in Figure 6 below. After the robot completes its setup, it transitions to the main super loop, where it instantly moves to the check boundary state. If no boundaries are detected, it moves to the check ultrasonic sensor state. If no objects are detected by the ultrasonic sensors, it moves to the check analog IR sensor state. If no objects are detected by the analog IR sensors, the robot moves to the forward state, then instantly returns to the loop state.

However, if any of the sensor check states detect a boundary or obstacle, the robot moves to the stop state, then transitions to a turn state depending on the type of boundary or object detected. If an object or boundary is detected on the right of the robot, it will move to the turn left state. If an object is detected on the left of the robot or a front or left boundary is detected, it will move to the turn right state. Once the operations in the selected turn state are complete, the robot moves to the forward state, then instantly returns to the loop state.

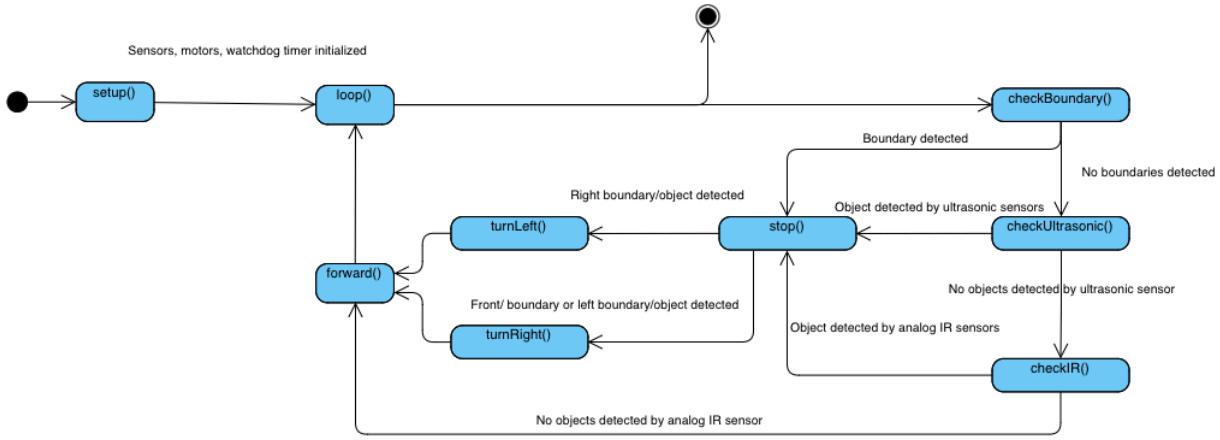

Figure 6: State Chart of the overall system

Subsystem Sequence Diagrams

There are 3 main sensors that will be used to make sure that the robot is able to avoid obstacles and stay within the boundaries. The sensors being used are the line follower sensor, Analog IR sensor, and the ultrasonic sensor.

The line follower sensors are used to make sure that the robot stays within the boundaries of the arena by detecting when it reaches a boundary and ensuring that it corrects the robot's direction based on which boundary was detected. There are two line follower sensors placed under the plow at the front of the robot; one on the left edge of the plow, and one on the right edge. The superloop gives the line follower the highest priority, checking and analyzing the data

obtained. If the sensor readings indicate a boundary was detected, it will transition the robot to the appropriate direction state to avoid the boundary. Figure 7 shows the sequence diagram of how the line-follower sensor subsystem behaves.

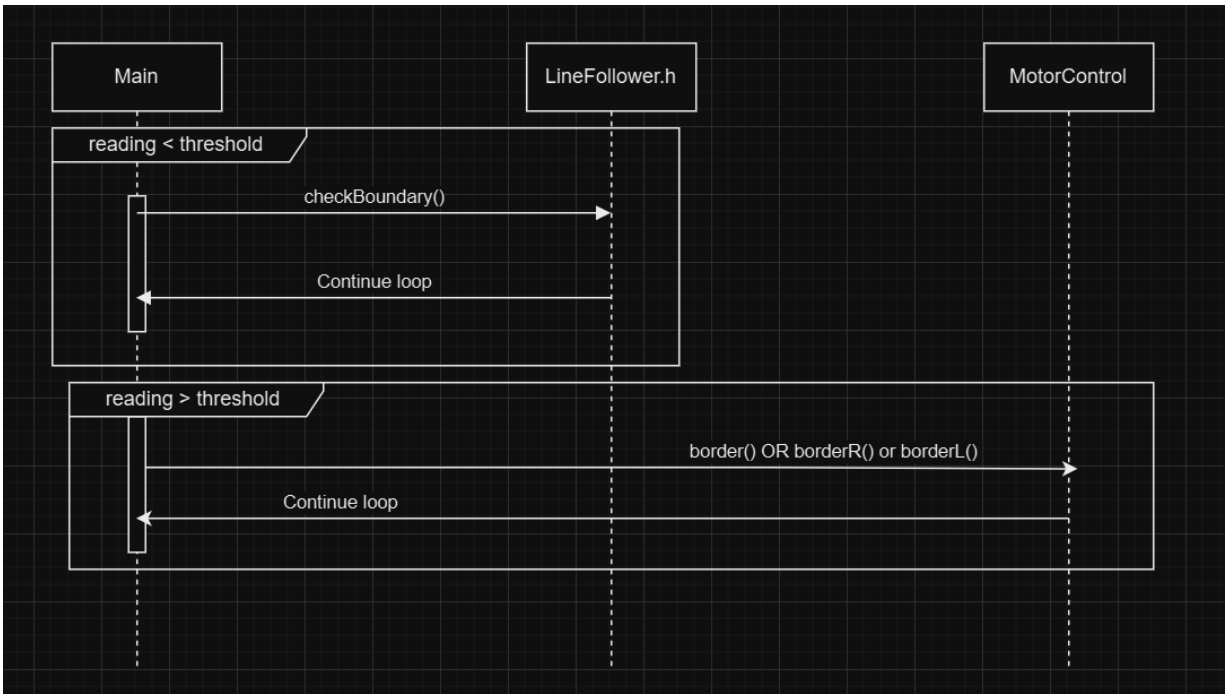

Figure 7: Line Follower Sensor Sequence Diagram

Both the ultrasonic and analog IR sensors are used to measure how far away obstacles are from the robot. The responsibility of the analog IR sensor is to cover the field of view that is not being covered by the ultrasonic sensor. The sequence diagram of the laser range sensor can be seen in Figure 8.

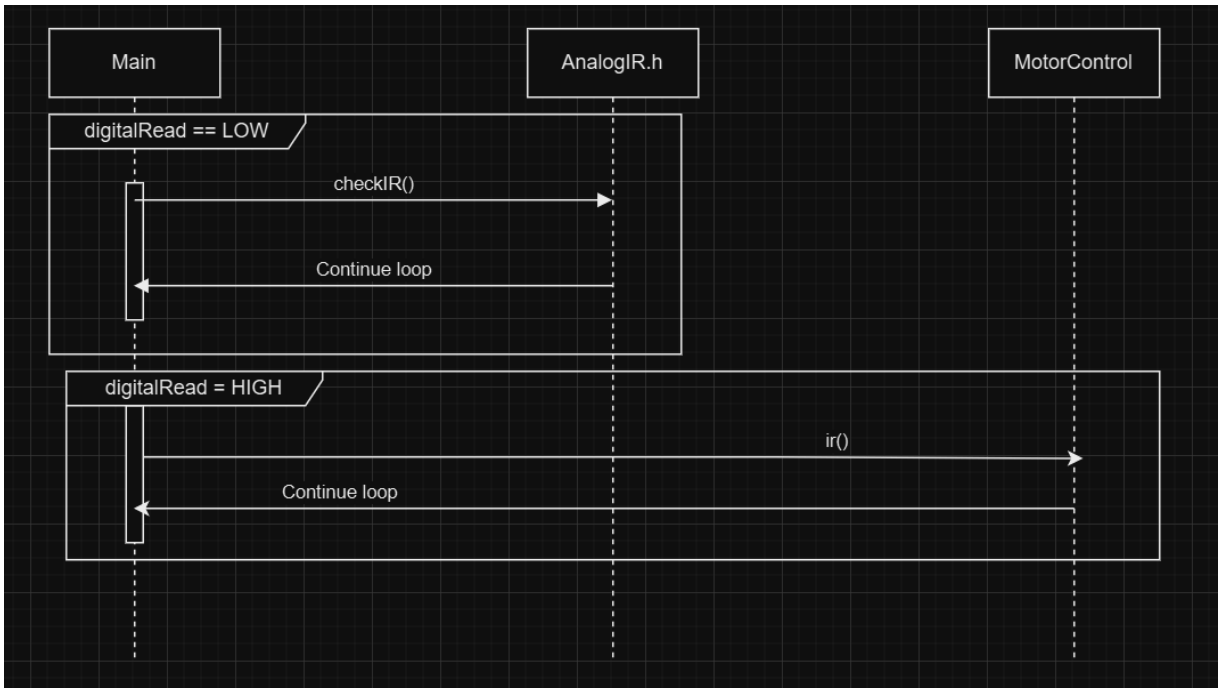

Figure 8: Analog IR Sequence Diagram

The ultrasonic sensor is placed in the two ends of the robot to maximize its field of coverage given the pulses output in an angle. The ultrasonic would be the primary source of detecting objects given it can accurately detect objects within a reasonable distance. Figure 9 shows the sequence diagram of the ultrasonic sensor and how it responds.

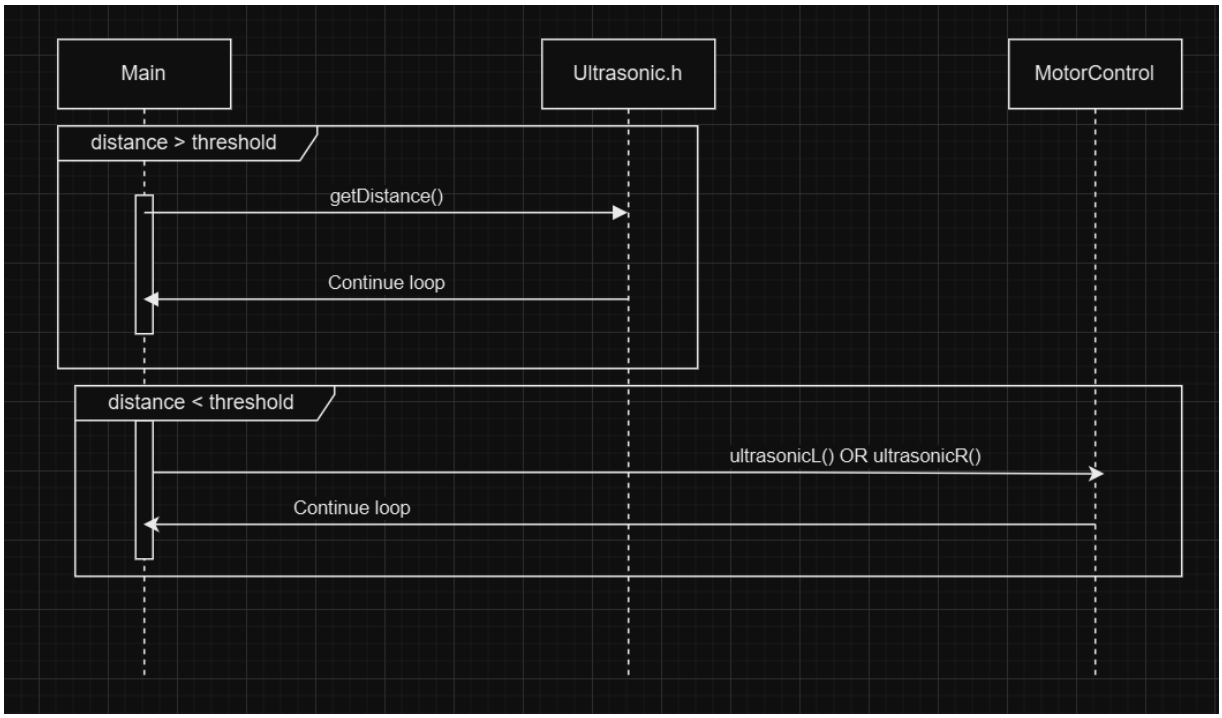

Figure 9: Ultrasonic Sensor Sequence Diagram

Watchdog Timer

A watchdog timer is used in electronic systems to detect and recover from malfunctions. It works by expecting a regular signal from the system to indicate it is functioning correctly. If the system fails to send this signal within a preset time interval, the watchdog timer assumes the system has crashed or become unresponsive and automatically resets it.
Within the automated snow plow, the watchdog timer will be set up to detect if the system has malfunctioned. Upon the system malfunction the watchdog timer should reset the system. This will ensure that upon a sensor malfunction the system will stop and restart, and that the snowplow will not crash into obstacles or leave the boundary.

The code in Figure 10 below shows how the watchdog timer is setup. The timeout time is defined at the top as 10000, which is in ms, this translates to 10s. In the setup function, the watchdog timer is enabled and set to that watchdog timeout period of 10s. Then at the start of the loop, the watchdog timer has to be reset which will make sure that the system doesn't crash because it reached the timeout period.

```
1   int watchdogTime = 10000; // 10s timeout
2
3   void watchdogSetup(void) {
4   }
5
6   void setup() {
7     watchdogEnable(watchdogTime);
8     // setup code here
9   }
10
11  void loop() {
12    watchdogReset();
13    // rest of loop here
14  }
```

Figure 10: Sample watchdog timer implementation

GitHub Repository/Function Handlers

The implementations of all the function handlers for the ultrasonic, line follower, laser range finder sensor, and analog IR sensor subsystems as well as the motor control subsystem and main superloop script can be found in our GitHub repository.

Unit Testing

Ultrasonic Sensor

The unit test for the ultrasonic test was designed to ensure that the ultrasonic module is capable of processing the data received correctly based on the set threshold value.

As discussed above, our design describes the functional use of one ultrasonic sensor. The main purpose of the sensor is to detect any objects at a close range of within 15 cm. The unit test was created around this. A set of predetermined ranges of data was created representing the possible distance values that the ultrasonic sensor could produce.

```
distance received: 25  threshold value: 15
ACTUAL RETURN: No obsclates detected  EXPECTED RETURN: No obstacles detected
distance received: 16  threshold value: 15
ACTUAL RETURN: No obsclates detected  EXPECTED RETURN: No obstacles detected
distance received: 22  threshold value: 15
ACTUAL RETURN: No obsclates detected  EXPECTED RETURN: No obstacles detected
distance received: 4  threshold value: 15
ACTUAL RETURN Obstacle Detected! Sending Alert  EXPECTED RETURN: Obstacle Detected! Sending Alert
distance received: 1  threshold value: 15
ACTUAL RETURN Obstacle Detected! Sending Alert  EXPECTED RETURN: Obstacle Detected! Sending Alert
distance received: 14  threshold value: 15
ACTUAL RETURN Obstacle Detected! Sending Alert  EXPECTED RETURN: Obstacle Detected! Sending Alert
```

Figure 11: Output after executing the unit test

When observing Figure 11, the first 3 distance values all exceed the threshold value. As a result, from the ultrasound's perspective there are no obstacles in its programmed range to detect. However, the final 3 values are all less than the threshold value, which suggests that there is an obstacle detected. The "Actual Return" output was a result of the algorithm that compares the distance input with the threshold, within the "checkObject" function in Figure 12.

```
#include "ultrasonic.h"

int DISTANCE = 0;
int TRG_PIN = 7;
int ECHO_PIN = 8;
bool CHECK = false;



void setup(){
    Serial.begin(9600);
}

void loop(){
    //Test 1:
    //Values that meet threshold value of 15 are inputted
    //Verify that no objects are detected
    //Should return "No obstacles detected"
    DISTANCE = 25;
    rx_ISR_test();
    if (CHECK){
    CHECK = false;
    checkObject();
    }
    Serial.println("Expected Return: No obstacles detected");
    DISTANCE = 16;
    rx_ISR_test();
    if (CHECK){
    CHECK = false;
    checkObject();
    }
    Serial.println("Expected Return: No obstacles detected");
    DISTANCE = 22;
    rx_ISR_test();
    if (CHECK){
    CHECK = false;
    checkObject();
    }
    Serial.println("Expected Return: No obstacles detected");
```

```
    //Test 2
    //Values that DO NOT meet threshold are inputted
    //Should return "Obstacle Detected! Sending Alert"
    DISTANCE = 4;
    rx_ISR_test();
    if (CHECK){
    CHECK = false;
    checkObject();
  }
    Serial.println("Expected Return: Obstacle Detected! Sending Alert");

    DISTANCE = 1;
    rx_ISR_test();
    if (CHECK){
    CHECK = false;
    checkObject();
  }
    Serial.println("Expected Return: Obstacle Detected! Sending Alert");

    DISTANCE = 14;
    rx_ISR_test();
    if (CHECK){
    CHECK = false;
    checkObject();
  }
    Serial.println("Expected Return: Obstacle Detected! Sending Alert");


}

void rx_ISR_test(){
    rxISR();

}
```

Figure 12: Unit test code for the Ultrasonic sensor

Table 7 below summarizes the testing conditions and successful execution results of each unit test case for the ultrasonic sensor.

Table 7: Summary of unit tests and results for ultrasonic sensors

Test case	Test case conditions	Expected Output	Actual Output	Pass/Fail
Verify no objects are detected when placed at a distance greater than threshold (15 cm)	Simulate object placed at known distance = 25 cm	No obstacles detected	No obstacles detected	Pass
Verify no objects are detected when placed at a distance greater than threshold (15 cm)	Simulate object placed at known distance = 16 cm	No obstacles detected	No obstacles detected	Pass
Verify no objects are detected when placed at a distance greater than threshold (15 cm)	Simulate object placed at known distance = 22 cm	No obstacles detected	No obstacles detected	Pass
Verify object detected when placed at a distance	Simulate object placed at known distance = 4 cm	Obstacle Detected! Sending Alert	Obstacle Detected! Sending Alert	Pass

less than threshold (15 cm)				
Verify object detected when placed at a distance less than threshold (15 cm)	Simulate object placed at known distance = 1 cm	Obstacle Detected! Sending Alert	Obstacle Detected! Sending Alert	Pass
Verify object detected when placed at a distance less than threshold (15 cm)	Simulate object placed at known distance = 14 cm	Obstacle Detected! Sending Alert	Obstacle Detected! Sending Alert	Pass

Laser range finder

The first test we ran to make sure that laser range finder worked correctly was making sure that the function does constant polling correctly, we created a unit test to see if the output is what we would desire.

```
11  // Test 1: Checks to see if the function used for constantly polling is working correctly
12  void polling_test() {
13    Serial.println("POLLING TEST STARTED");
14    int sensor_test_value[] = {500, 450, 300, 500, 101, 200, 400, 400, 500, 500};
15    int arraySize = sizeof(sensor_test_value) / sizeof(sensor_test_value[0]);
16    for (int i = 0; i < arraySize; i += 2) {
17      int distanceLeft = sensor_test_value[i];
18      int distanceRight = sensor_test_value[i + 1];
19      check_sensors(distanceLeft, distanceRight);
20    }
21    Serial.println("POLLING TEST END");
22    Serial.println();
23  }
```

Figure 13: Polling test Code

The unit test creates some dummy data which is passed into the check sensor function to make sure that it is working correctly. The check_sensor function takes in the data and checks if it is within the desired threshold, if it is then it prints the value it reads. Otherwise, if it is lower than the threshold, it sets flags for the interrupts to turn right or left respective to the sensor that reads the value.

```
78  void check_sensors(int distanceLeft, int distanceRight) {
79    // Check the left sensor distance
80    if (distanceLeft <= 100) {
81      TURN_RIGHT = true; // Trigger interrupt and stop sensor reading until the interrupt is serviced
82      right_interrupt_triggered = true;
83    } else if (sensors[0].timeoutOccurred()) {
84      Serial.print("TIMEOUT");
85    } else {
86      Serial.print(distanceLeft);
87    }
88    Serial.print('\t'); // Tab space between readings
89    // Check the right sensor distance
90    if (distanceRight <= 100) {
91      TURN_LEFT = true; // Trigger interrupt and stop sensor reading until the interrupt is serviced
92      left_interrupt_triggered = true;
93    } else if (sensors[1].timeoutOccurred()) {
94      Serial.print("TIMEOUT");
95    } else {
96      Serial.print(distanceRight);
97    }
98    Serial.println();
99    delay(500);
100 }
```

Figure 14: Check Sensor Code

In the following figure, you can see the output of the unit test running successfully.

```
POLLING TEST STARTED
500     450
300     500
101     200
400     400
500     500
POLLING TEST END :)
```

Figure 15: Output of polling test

The second test we ran was to make sure that the interrupts were being serviced correctly for the right laser range sensor.

```
25  // Test 2: Validate interrupt servicing for RIGHT detection
26  void object_right_test() {
27    Serial.println("OBJECT RIGHT TEST STARTED");
28
29    // Reset relevant flags
30    TURN_LEFT = false;
31    left_interrupt_triggered = false;
32
33    // Call the sensor check function
34    check_sensors(500, 88);
35
36    // Check if the interrupt was triggered
37    if (!TURN_LEFT) {
38        Serial.println("FAIL: Left interrupt not triggered.");
39        return;
40    }
41
42    // Simulate servicing the interrupt
43    if (TURN_LEFT) {
44        object_Detected_Right();
45    }
46
47    // Verify if the interrupt was serviced correctly
48    if (!left_interrupt_triggered && !TURN_LEFT) {
49        Serial.println("PASS: Left interrupt serviced correctly.");
50    } else {
51        Serial.println("FAIL: Left interrupt not serviced correctly.");
52    }
53
54    // Reset flags
55    TURN_LEFT = false;
56    left_interrupt_triggered = false;
57
58    Serial.println("OBJECT RIGHT TEST ENDED");
59    Serial.println();
60  }
```

Figure 16: Object Right Detection Test Code

The unit test would check if there was an object that was detected on the right sensor, if there was it triggered the interrupt that would send an interrupt to tell the motors to turn left. Once the interrupt was serviced, it resets the flags and allows both the sensors to continue to poll. The output of the object right test running successfully can be seen in the figure below.

```
OBJECT RIGHT TEST STARTED
500
PASS: Left interrupt serviced correctly.
OBJECT RIGHT TEST ENDED
```

Figure 17: Output of Object Right Detection Test Code

A similar test was written to check if there was an object that was detected on the left sensor. Instead, it would trigger an interrupt that would tell the motors to turn right. Figure 18 and 19 show the code for the object left detection test and the output of running the test respectively.

```
// Test 3: Test to make sure that it responds correctly to object detection on the left
void object_left_test() {
  Serial.println("OBJECT LEFT TEST STARTED");

  // Reset relevant flags
  TURN_RIGHT = false;
  right_interrupt_triggered = false;

  // Call the sensor check function
  check_sensors(90, 188);

  // Check if the interrupt was triggered
  if (!TURN_RIGHT) {
      Serial.println("FAIL: Right interrupt not triggered.");
      return;
  }

  // Simulate servicing the interrupt
  if (TURN_RIGHT) {
      object_Detected_Left();
  }

  // Verify if the interrupt was serviced correctly
  if (!right_interrupt_triggered && !TURN_RIGHT) {
      Serial.println("PASS: Right interrupt serviced correctly.");
  } else {
      Serial.println("FAIL: Right interrupt not serviced correctly.");
  }

  // Reset flags
  TURN_RIGHT = false;
  right_interrupt_triggered = false;

  Serial.println("OBJECT LEFT TEST ENDED");
  Serial.println();
}
```

Figure 18: Object Left Detection Test Code

```
OBJECT LEFT TEST STARTED
        188
PASS: Right interrupt serviced correctly.
OBJECT LEFT TEST ENDED
```

Figure 19: Output of Object Right Detection Test Code

Table 8 below summarizes the testing conditions and successful execution results of each unit test case for the laser range finder sensor.

Table 8: Summary of unit tests and results for laser range finder sensors

Test case	Test case conditions	Expected Output	Actual Output	Pass/Fail
Verify function handler for reading data from laser range finder is outputting expected data	Simulate object distances for left and right sensors: 500 450 300 500 101 200 400 400 500 500	500 450 300 500 101 200 400 400 500 500	500 450 300 500 101 200 400 400 500 500	Pass

Verify signal telling motors to turn left generated when object within threshold (100 cm) of right sensor	Simulate object placed at known distance = 500 cm from left sensor, and known distance = 88 cm from right sensor	Left interrupt serviced correctly	Left interrupt serviced correctly	Pass
Verify signal telling motors to turn right generated when object within threshold (100) of left sensor	Simulate object placed at known distance = 90 cm from left sensor, and known distance = 188 cm from right sensor	Right interrupt serviced correctly.	Right interrupt serviced correctly.	Pass

Line follower

The unit test for the line follower sensor subsystem was designed to meet the predetermined pass/fail criteria of successfully generating an interrupt when a black boundary line is detected. The test cases implemented validate proper boundary detection by testing against the function handlers shown in Figure 20 below.

```ino
#include "LineFollower.h"

void initializeSensors() {
  pinMode(LEFT_SENSOR_RIGHT_PIN, INPUT);
  pinMode(LEFT_SENSOR_LEFT_PIN, INPUT);
  pinMode(LEFT_SENSOR_MIDDLE_PIN, INPUT);

  pinMode(RIGHT_SENSOR_RIGHT_PIN, INPUT);
  pinMode(RIGHT_SENSOR_LEFT_PIN, INPUT);
  pinMode(RIGHT_SENSOR_MIDDLE_PIN, INPUT);
}

int readSensorAverage() {
  int numReadings = 10;
  int total = 0;
  SensorReadings temp;
  for (int i = 0; i < numReadings; i++) {
    temp = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
    // total += (temp.leftReading + temp.middleReading + temp.rightReading) / 3;
    total += temp.leftReading;
    delay(10); // Small delay between readings
  }
  return total / numReadings; // Return the average
}

int calibrateThreshold() {
  int whiteAverage = 0;
  int blackAverage = 0;
  // Prompt user to place on white surface
  Serial.print("Place sensors on a white surface.  Calibration will begin in ");
  for (int i = 5; i > 0; i--) {
    Serial.print(i);
    Serial.println(" seconds...");
    delay(1000); // 1-second delay
  }
  whiteAverage = readSensorAverage();
  // Prompt user to place on black surface
  Serial.print("Place sensors on a black surface.  Calibration will begin in ");
  for (int i = 5; i > 0; i--) {
    Serial.print(i);
    Serial.println(" seconds...");
    delay(1000); // 1-second delay
  }
  blackAverage = readSensorAverage();
  int threshold = (whiteAverage + blackAverage) / 2;
  Serial.println("Threshold calibrated succesfully.  Threshold = " + String(threshold));
  return threshold;
}


SensorReadings readSensorValues(int leftPin, int middlePin, int rightPin) {
  SensorReadings sensor;
  sensor.leftReading = analogRead(leftPin);
  sensor.middleReading = analogRead(middlePin);
  sensor.rightReading = analogRead(rightPin);
  return sensor;
}

void printSensorReadings() {
  Serial.println("Left Sensor left: " + String(leftSensor.leftReading) + " | Left Sensor middle: " + String(leftSensor.middleReading) + " | Left Sensor right: " + String(leftSensor.rightReading));
  Serial.println("Right Sensor left: " + String(rightSensor.leftReading) + " | Right Sensor middle: " + String(rightSensor.middleReading) + " | Right Sensor right: " + String(rightSensor.rightReading));
}

void checkBoundary(SensorReadings &left_sensor, SensorReadings &right_sensor) {
  // if((left_sensor.leftReading > dynamicThreshold || left_sensor.middleReading > dynamicThreshold || left_sensor.rightReading > dynamicThreshold) && (right_sensor.leftReading > dynamicThreshold || right_sensor.middleReading > dynamicThreshold || right_sensor.rightReading > dynamicThreshold)) {
  if((left_sensor.leftReading > STATIC_THRESHOLD || left_sensor.middleReading > STATIC_THRESHOLD || left_sensor.rightReading > STATIC_THRESHOLD) && (right_sensor.leftReading > STATIC_THRESHOLD || right_sensor.middleReading > STATIC_THRESHOLD || right_sensor.rightReading > STATIC_THRESHOLD)) {
    //DEBUG
    Serial.println("FRONT EDGE DETECTED.  TURNING AROUND");
    CORRECTION_DIRECTION = 'B';
  }

  // else if(left_sensor.leftReading > dynamicThreshold) {
  else if(left_sensor.leftReading > STATIC_THRESHOLD) {
    //DEBUG
    Serial.println("LEFT BOUNDARY DETECTED.  TURNING RIGHT");
    CORRECTION_DIRECTION = 'R';
  }

  // else if(right_sensor.rightReading > dynamicThreshold) {
  else if(right_sensor.rightReading > STATIC_THRESHOLD) {
    // //DEBUG
    Serial.println("RIGHT BOUNDARY DETECTED.  TURNING LEFT");
    CORRECTION_DIRECTION = 'L';
  }

  else {
    //DEBUG
    Serial.println("No boundaries detected");
    CORRECTION_DIRECTION = '\0';
  }
}

void boundaryISR() {
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  checkBoundary(leftSensor, rightSensor);
}
```

Figure 20: Function handlers tested in line follower sensor unit test

Figure 21 below shows the unit test script implementing the test cases chosen to validate the line follower sensor's function handlers.

```ino
#include "LineFollower.h"

  // Declare sensor objects
  SensorReadings leftSensor, rightSensor;
  char CORRECTION_DIRECTION = '\0';

void assertEqualsBoolean(bool expected, bool actual, String testName) {
  if (expected == actual) {
    Serial.println(testName + ": PASS");
  } else {
    Serial.println(testName + ": FAIL");
    Serial.println("  Expected: " + expected ? "true" : "false");
    Serial.println("  Actual: " + actual ? "true" : "false");
  }
}

void assertEqualsChar(char expected, char actual, String testName) {
  if (expected == actual) {
    Serial.println(testName + ": PASS");
  } else {
    Serial.println(testName + ": FAIL");
    Serial.println("  Expected: " + expected);
    Serial.println("  Actual: " + actual);
  }
}

void countdown(int seconds) {
  Serial.println("Starting next test in " + String(seconds) + "seconds...");
  for (int i = seconds; i > 0; i--) {
    delay(1000); // 1-second delay
  }
  Serial.println("Starting next test...");
}

void setup() {
  // Initialize the sensors
  initializeSensors();
  Serial.begin(9600);
}

void loop() {

  //Test case 1: Read sensor values on white surface and verify sensors correctly detect white (< STATIC_THRESHOLD)
  Serial.println("Test 1a: White surface detection test - Move left sensor to white surface");
  countdown(5);
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  bool leftSensorWhiteSurfaceDetected = (leftSensor.leftReading < STATIC_THRESHOLD && leftSensor.middleReading < STATIC_THRESHOLD && leftSensor.rightReading < STATIC_THRESHOLD);
  assertEqualsBoolean(true, leftSensorWhiteSurfaceDetected, "Left sensor white surface detection test");

  Serial.println("Test 1b: White surface detection test - Move right sensor to white surface");
  countdown(5);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  bool rightSensorWhiteSurfaceDetected = (rightSensor.leftReading < STATIC_THRESHOLD && rightSensor.middleReading < STATIC_THRESHOLD && rightSensor.rightReading < STATIC_THRESHOLD);
  assertEqualsBoolean(true, rightSensorWhiteSurfaceDetected, "Right sensor white surface detection test");

  //Test case 2: Read sensor values on black surface and verify sensors correctly detect black (> STATIC_THRESHOLD)
  Serial.println("Test 2a: Black surface detection test - Move left sensor to Black surface");
  countdown(5);
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  bool leftSensorBlackSurfaceDetected = (leftSensor.leftReading > STATIC_THRESHOLD && leftSensor.middleReading > STATIC_THRESHOLD && leftSensor.rightReading > STATIC_THRESHOLD);
  assertEqualsBoolean(true, leftSensorBlackSurfaceDetected, "Left sensor black surface detection test");

  Serial.println("Test 2b: Black surface detection test - Move right sensor to black surface");
  countdown(5);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  bool rightSensorBlackSurfaceDetected = (rightSensor.leftReading > STATIC_THRESHOLD && rightSensor.middleReading > STATIC_THRESHOLD && rightSensor.rightReading > STATIC_THRESHOLD);
  assertEqualsBoolean(true, rightSensorBlackSurfaceDetected, "Right sensor black surface detection test");

  //Test case 3: Boundary detection test for front edge (verify correction direction is 'B')
  Serial.println("Test 3: Boundary detection test (front edge detection) - Move both sensors to black surface");
  countdown(5);
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  checkBoundary(leftSensor, rightSensor);
  assertEqualsChar('B', CORRECTION_DIRECTION, "Boundary detection test (front edge detection)");

  //Test case 4: Boundary detection test for left edge (verify correction direction is 'R')
  Serial.println("Test 4: Boundary detection test (left edge detection) - Move left sensor to black surface");
  countdown(5);
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  checkBoundary(leftSensor, rightSensor);
  assertEqualsChar('R', CORRECTION_DIRECTION, "Boundary detection test (left edge detection)");

//Test case 5: Boundary detection test for right edge (verify correction direction is 'L')
  Serial.println("Test 5: Boundary detection test (right edge detection) - Move right sensor to black surface");
  countdown(5);
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  checkBoundary(leftSensor, rightSensor);
  assertEqualsChar('L', CORRECTION_DIRECTION, "Boundary detection test (right edge detection)");

//Test case 6: Boundary detection test for no boundaries (verify correction direction is '\0')
  Serial.println("Test 6: Boundary detection test (no boundaries) - Move both sensors to white surface");
  countdown(5);
  leftSensor = readSensorValues(LEFT_SENSOR_LEFT_PIN, LEFT_SENSOR_RIGHT_PIN, LEFT_SENSOR_MIDDLE_PIN);
  rightSensor = readSensorValues(RIGHT_SENSOR_LEFT_PIN, RIGHT_SENSOR_RIGHT_PIN, RIGHT_SENSOR_MIDDLE_PIN);
  printSensorReadings();
  checkBoundary(leftSensor, rightSensor);
  assertEqualsBoolean('\0', CORRECTION_DIRECTION, "Boundary detection test (no boundaries)");

  Serial.println("Unit test complete.");
  delay(10000);
}
```

Figure 21: Unit test script for line follower sensor

As shown in Figure 21, test case 1 firstly validates that the sensor correctly reads a white surface and does not generate a false interrupt, and test case 2 validates that the sensor correctly reads a black boundary when the line follower sensors cross one (i.e the sensor readings are determined to be beyond the predetermined threshold for a black surface). Test cases 3-6 then verify that the checkBoundary() function generates the correct mock interrupt signal (a character representing the direction the motors should turn to avoid the boundary) based on the type of boundary detected. A front boundary detected should generate a mock interrupt to turn the robot 180º, a left boundary should generate an interrupt to turn the robot right, and a right boundary should generate an interrupt to turn the robot left.

Figure 22 below shows the successful execution of the described test cases, validating that the line follower sensor function handlers implemented meet the subsystem's success criteria by successfully detecting a black boundary line and generating an interrupt to correct the robot's direction accordingly. Note that the threshold value used for testing purposes was 800, but this value is subject to change based on the robot's operating environment.

```
Test 1a: White surface detection test - Move left sensor to white surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 786 | Left Sensor middle: 792 | Left Sensor right: 780
Right Sensor left: 0 | Right Sensor middle: 0 | Right Sensor right: 0
Left sensor white surface detection test: PASS
Test 1b: White surface detection test - Move right sensor to white surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 786 | Left Sensor middle: 792 | Left Sensor right: 780
Right Sensor left: 666 | Right Sensor middle: 310 | Right Sensor right: 500
Right sensor white surface detection test: PASS
Test 2a: Black surface detection test - Move left sensor to Black surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 836 | Left Sensor middle: 836 | Left Sensor right: 835
Right Sensor left: 666 | Right Sensor middle: 310 | Right Sensor right: 500
Left sensor black surface detection test: PASS
Test 2b: Black surface detection test - Move right sensor to black surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 836 | Left Sensor middle: 836 | Left Sensor right: 835
Right Sensor left: 886 | Right Sensor middle: 886 | Right Sensor right: 889
Right sensor black surface detection test: PASS
Test 3: Boundary detection test (front edge detection) - Move both sensors to black surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 829 | Left Sensor middle: 829 | Left Sensor right: 828
Right Sensor left: 894 | Right Sensor middle: 885 | Right Sensor right: 891
FRONT EDGE DETECTED.  TURNING AROUND
Boundary detection test (front edge detection): PASS
Test 4: Boundary detection test (left edge detection) - Move left sensor to black surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 813 | Left Sensor middle: 814 | Left Sensor right: 812
Right Sensor left: 681 | Right Sensor middle: 662 | Right Sensor right: 681
LEFT BOUNDARY DETECTED.  TURNING RIGHT
Boundary detection test (left edge detection): PASS
Test 5: Boundary detection test (right edge detection) - Move right sensor to black surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 757 | Left Sensor middle: 747 | Left Sensor right: 744
Right Sensor left: 928 | Right Sensor middle: 927 | Right Sensor right: 930
RIGHT BOUNDARY DETECTED.  TURNING LEFT
Boundary detection test (right edge detection): PASS
Test 6: Boundary detection test (no boundaries) - Move both sensors to white surface
Starting next test in 5seconds...
Starting next test...
Left Sensor left: 763 | Left Sensor middle: 753 | Left Sensor right: 742
Right Sensor left: 469 | Right Sensor middle: 365 | Right Sensor right: 319
No boundaries detected
Boundary detection test (no boundaries): PASS
Unit test complete.
```

Figure 22: Sample terminal output for line follower unit test execution

Table 9 below summarizes the testing conditions and successful execution results of each unit test case for the line follower sensor.

Table 9: Summary of unit tests and results for line follower sensors

Test case	Test case conditions	Expected Output	Actual Output	Pass/Fail
Verify line follower sensors correctly detect white surface (< threshold)	Line follower sensors placed on white surface	-True (Left sensor white surface detected) -True (Right sensor white surface detected)	-True (Left sensor white surface detected) -True (Right sensor white surface detected)	Pass
Verify line follower sensors correctly detect black surface (> threshold)	Line follower sensors placed on black surface	-True (Left sensor black surface detected) -True (Right sensor black surface detected)	-True (Left sensor black surface detected) -True (Right sensor black surface detected)	Pass
Verify signal telling motors to turn around generated when front boundary detected	Both line follower sensors placed on black surface	-FRONT EDGE DETECTED. TURNING AROUND -Correction direction = ‘B’	-FRONT EDGE DETECTED. TURNING AROUND -Correction direction = ‘B’	Pass
Verify signal telling motors to turn right generated when left boundary detected	Left line follower sensor placed on black surface	-LEFT BOUNDARY DETECTED. TURNING RIGHT -Correction direction = ‘R’	-LEFT BOUNDARY DETECTED. TURNING RIGHT -Correction direction = ‘R’	Pass
Verify signal telling motors to turn left generated when right boundary detected	Right line follower sensor placed on black surface	-RIGHT BOUNDARY DETECTED. TURNING LEFT -Correction direction = ‘L’	-RIGHT BOUNDARY DETECTED. TURNING LEFT -Correction direction = ‘L’	Pass

System Integration Testing

After individually programming and testing all the sensors, we incrementally integrated them all together into one system. We had designed our snowplow to function under a superloop, where the sensors gather data and analyze it to make the appropriate motor control.

We started by firstly integrating the line follower subsystem with the motor control subsystem to ensure that the robot met the passing criteria of successfully detecting a black boundary line and controlling the motors accordingly to turn itself and avoid the line without the wheels crossing it by more than 5 cm.

Table 10 below contains the results of the integration test, validating the consistency and accuracy of the integrated system.

Table 10: Trial results of line follower and motor control integration test

Trial number	Length plow exceeded boundary by (mm)
1	10.6
2	11.1
3	11.3
4	8.9
5	13.3

After successfully implementing boundary detection and avoidance, we added the ultrasonic sensor subsystem to begin integrating obstacle detection and avoidance. Finally, after the ultrasonic sensor subsystem was successfully integrated, we added the laser range sensor subsystem to complete the robot's functionality. While integrating all the sensors, we encountered an issue with the I2C communication of the laser range finders. Laser range sensors were not able to work correctly in the superloop implementation that we had come up with, so we pivoted and instead decided to use the IR Obstacle Avoidance Sensor which was easier to implement.

Table 11 below contains the result of the final integration test, validating the consistency and accuracy of the complete obstacle detection and avoidance system implemented.

Table 11: Trial results of full integration test for obstacle detection and avoidance

Trial number	Obstacle distance before robot turns (mm)
1	15
2	14.9
3	15
4	15
5	14.8

Control Charts

Control charts are useful for monitoring the stability of processes over a period of time. The following three test scenarios were executed during integration testing to create control charts validating the reliability and accuracy of the implemented robot, with each scenario consisting of a minimum of five test runs.

Speed of Motors

The first test scenario created to validate the stability of the system was the test for motor speed. For this scenario, trial runs were conducted to verify that the system consistently met the requirement stating that the robot speed should not exceed 30 cm/s. Figure 23 below shows the control chart for the trial results.

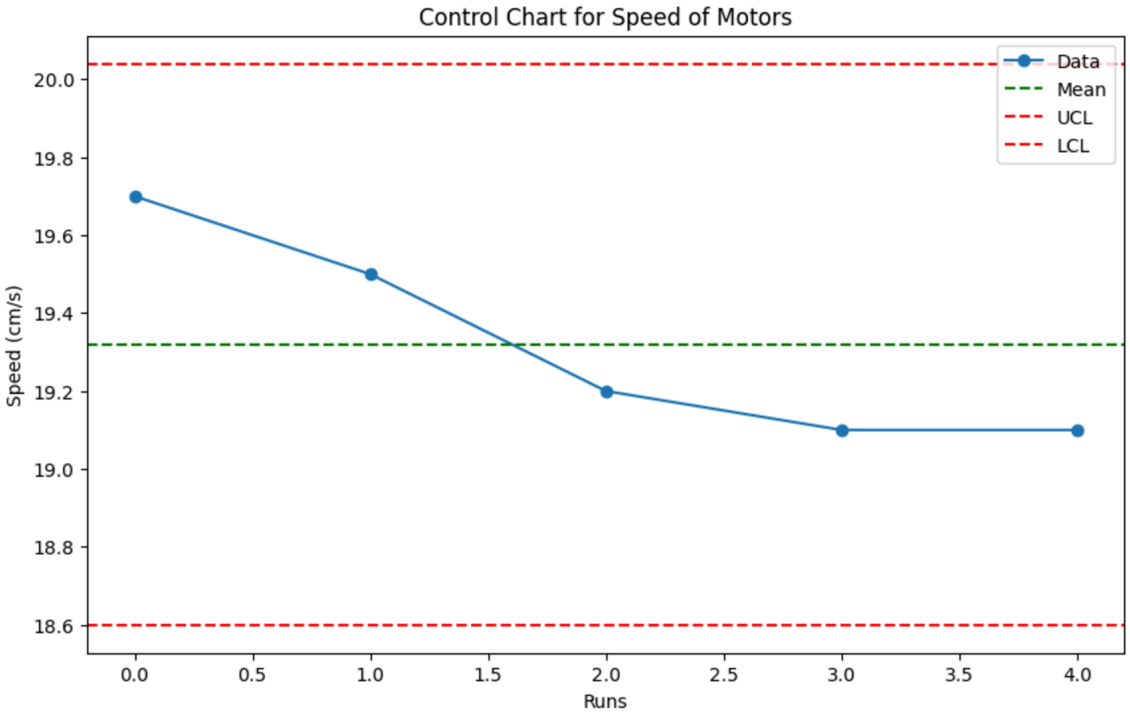

Figure 23: Control chart for speed of motors

How Far Plow Exits Boundary

The first test scenario created was the test for validating how far the plow exits the boundary when turning. For this scenario, trial runs were conducted to verify that the system consistently met the requirement stating that the robot should not cross the boundary by more than 5 cm. Figure 24 below shows the control chart for the trial results.

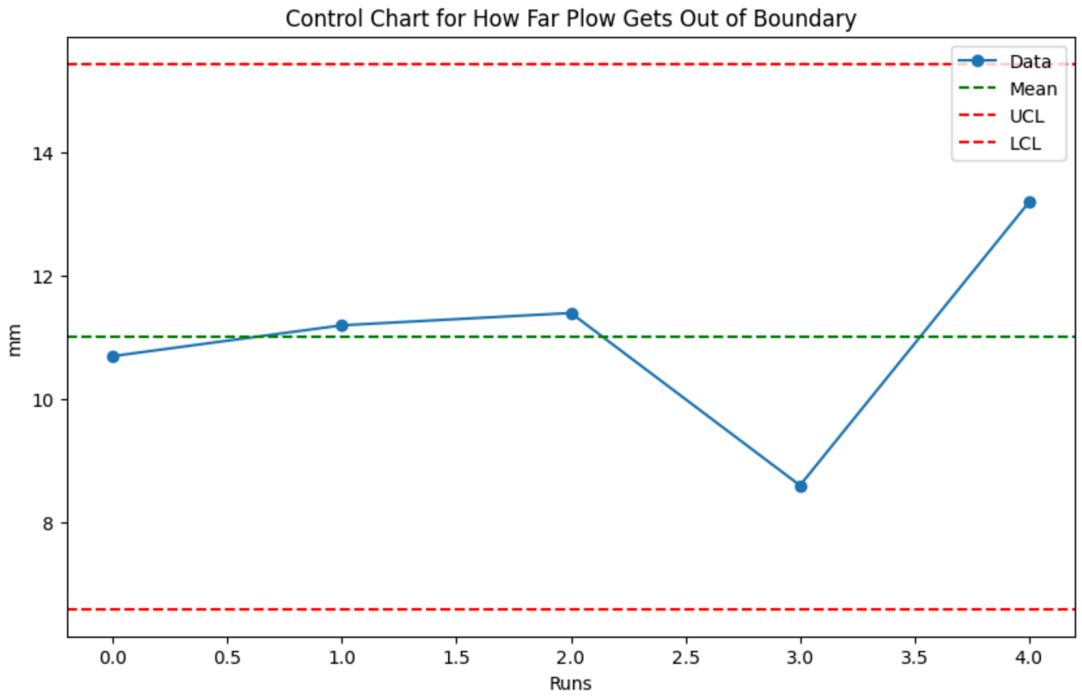

Figure 24: Control chart for how far robot gets out of boundary

Obstacle Distance Before Turns

The final test scenario created to validate the stability of the system was the test for obstacle distance before the robot turns. For this scenario, trial runs were conducted to verify that the system consistently met the requirement stating that the robot should be able to detect an obstacle along its path within a maximum proximity of 10 cm. Figure 25 below shows the control chart for the trial results.

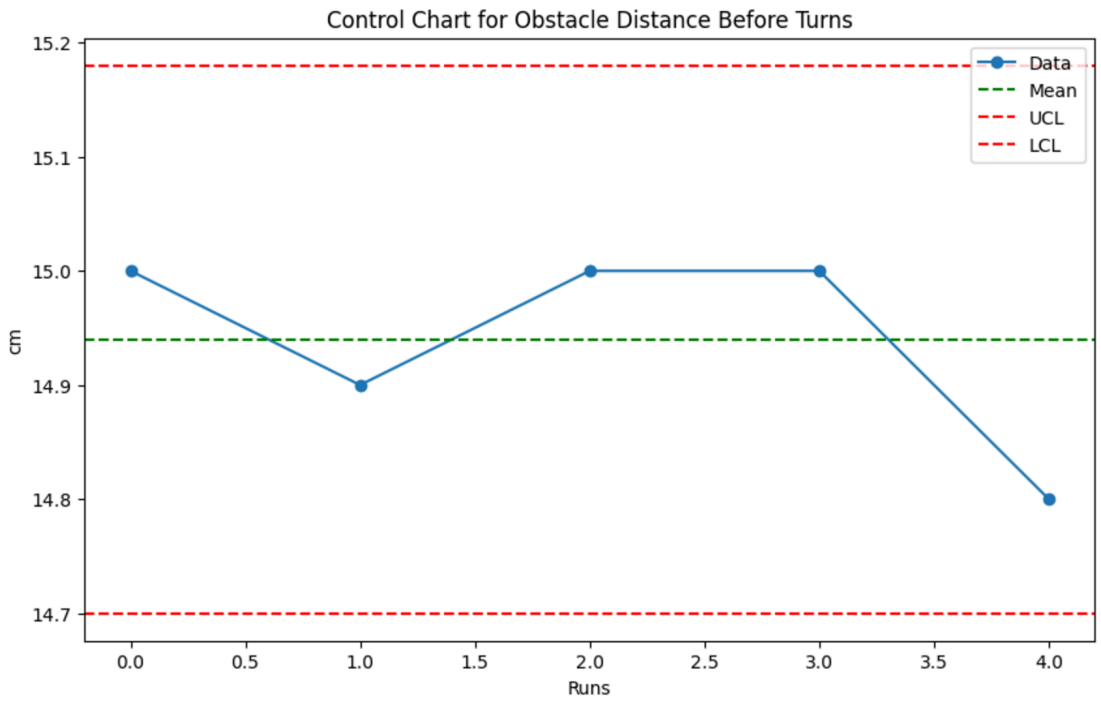

Figure 25: Control chart for obstacle distance before turns

Customer Testing

When performing customer testing we were able to meet all the functional requirements/success criteria that were given to us and were set ourselves at the beginning of the project, as well as those specified by the customer to satisfy the overall project objectives. The robot's chassis and plow was within the height, width, and length requirements that were provided. The robot did not exceed 30 cm/s while it was operating. It was successful in removing around 50% of snow cubes, specifically around 40 snow cubes were removed out of a total of 60 snow cubes. This was all done without any human intervention from the testing area and the plow not exceeding more than 20 mm outside of the boundary. The system was also flawless in detecting a boundary and turning to avoid the wheels from crossing the boundary.

There was a rare edge case though, where if the robot got into a corner, it would struggle for a while trying to get out, but eventually after a few attempts it would make its way out. To potentially prevent this in our future design, we would implement logic to turn at a random angle for every turn instead of a fixed one.

The obstacle detection worked well, but when the object was to the edge of the ultrasonic sensor the object would not be detected properly causing the robot to scrape against the object. This was because the ultrasonic pulse would reflect away from the sensor given the obstacles were angled. To avoid this in our future design, we would angle the ultrasonic sensor.

Table 12 below summarizes the results of our customer test trial. Note that only one trial was completed because the battery terminal connected to the motor driver board was disconnected during the second trial, and we were unable to reconnect it in time to complete a full second trial within the 5 minute time window.

Table 12: Results of customer testing during demo

Trial Number	Number of snow cubes cleared/total snow cubes	Number of soft obstacle hits	Number of hard obstacle hits	Number of times boundary crossed	Trial duration (min)
1	40/60	2	2	0	5:00
2	-	-	-	-	-

References

[1] sws warning lights, “The Importance of Highway Snowplow Lighting - Highway Snowplow Lights,” *SWS Warning Lights Inc.*, Jan. 10, 2024. https://warninglightsinc.com/highway-snowplow-lighting/ [accessed Oct. 18, 2024]

[2] M. Taha, “Project Description”, 2024. [Online]. Available: https://brightspace.carleton.ca/d2l/le/content/292069/viewContent/3933855/View

[3] M. Taha, “Project Proposal”, 2024. [Online]. Available: https://brightspace.carleton.ca/d2l/le/content/292069/viewContent/3933858/View