Robotic Algorithms - Mobile Robot Assembly and Navigation

Muhie Al Haimus, 230388549

December 2024

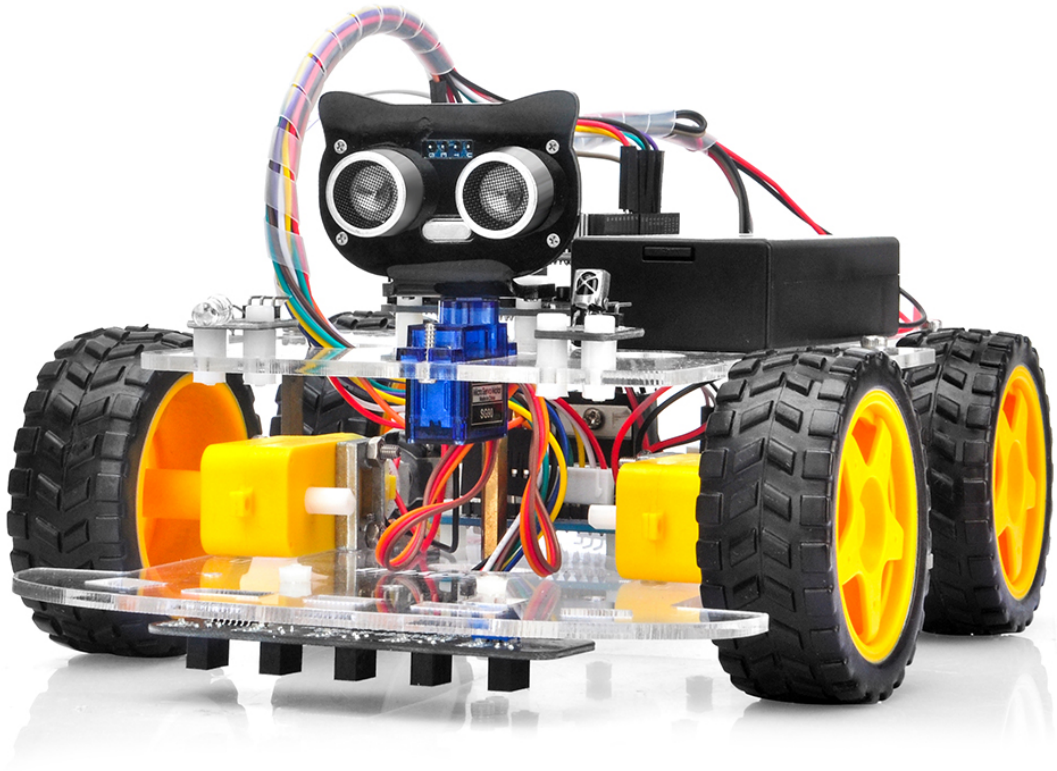


Figure 0: OSOYOO V2.1 Robot Car [1].

Contents

Abstract. From self driving cars to warehouse automation: robotic algorithms are crucial to navigate unknown, non-deterministic environments. To test these pathfinding procedures, an experimental analysis was conducted through the use of a mobile robot kit. An addition of various sensors and actuators allowed for the production of a Bug Algorithm.

Keywords— Unknown Environments, Non-deterministic, Algorithms, Mobile Robots, Pathfinding, Sensors and Actuators, Bug Algorithm,

1 Introduction

The purpose of this laboratory is to build an understanding on how to fabricate a functioning four wheel drive mobile robot and the communication protocols required to interface with sensors and actuators. Furthermore, this lab it should provide an understanding on how both sensors and actuators truly work from accelerometers to ultrasonic sensors and the problems associated with these sensors, if any. These sensors and actuators will then provide a foundation from which local robot navigation algorithms can be produced, rigorously tested and practically investigated to understand their benefits and limitations in a real-world environment.

The main focus was to implement Bug Algorithms (BAs), they are known for their simple operation and low memory requirements, ideal for tiny robots with little resources as compared to much more complex algorithms like Simultaneous Localisation And Mapping (SLAM) which require a multitude of fusion filters and sensors to operate which are too computationally expensive [2].

The main principle behind BAs is that they are not aware of information of the environment, only the relative position of the target (goal). Only when the algorithms come into contact (or by proximity depending on the sensors used) with obstacles and walls, it reacts locally, allowing the robot to track the edge and move towards the target [3].

In this laboratory, two Bug Algorithms were explored; aptly named Bug 1 and Bug 2. Both of these algorithms operate by navigating obstacles and using the Euclidean distance between the position of the robot and the goal. The difference lies within the path taken around obstacles; Bug 1 uses an exhaustive search approach [4] meaning it navigates round the object once to find the optimal leave point (the shortest distance to the goal from the obstacle that it is navigating). Whereas Bug 2 only uses the Euclidean distance between the starting position of the robot and the goal, when navigating obstacles. As soon as Bug 2 reaches point along the line of the Euclidean distance, it immediately takes this path; also known as a greedy search approach [4].

2 Materials and Methods

2.1 Fabrication of Mobile Robot

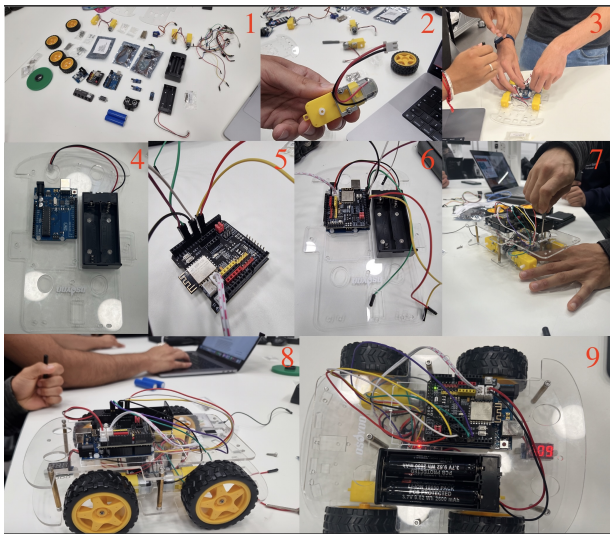


Figure 1: Fabrication steps of mobile robot

The fabrication process of the mobile robot began by 1) unboxing the kit and checking if all of the parts were present as expected. The next step in the fabrication process was to assemble the four motor assemblies, this was conducted by attaching the metal motor bracket to the motors with two 2) M3 hex screws per motor. The motor mounts

were then 3) connected with an additional two M3 hex screws to the frame. The Arduino UNO Microcontroller was mounted along with the battery case on the upper frame 4) of the four-wheel drive robot. 5) For the electronics assembly six jumper wires were connected via digital PWM pins (6,7,8,9,11,12) on the UART shield to the L298N H-Bridge motor controller. Power was also supplied to the H-bridge via a separate 12V line to the UART shield. The UART 6) ESP-32 Wi-Fi shield was then attached to the UNO. Five copper pillars 7) were used to attach the bottom and top frame with an additional ten M3 hex screws. The four skid drive 8) wheels were then connected to the shaft of the motors. Lastly, 9) two 3.7v Lithium Ion batteries were inserted into the battery case and the robot was powered on.

2.1.1 Supplementary Fabrication - Addition of Extra Sensors

To allow the mobile robot to interact with its environment, additional sensors are required. In this case the following were used: a line tracking IR module, an Ultrasonic sensor attached to an actuator to allow it to rotate precisely between $0 - 180°$ and an accelerometer for odometry of the robot.

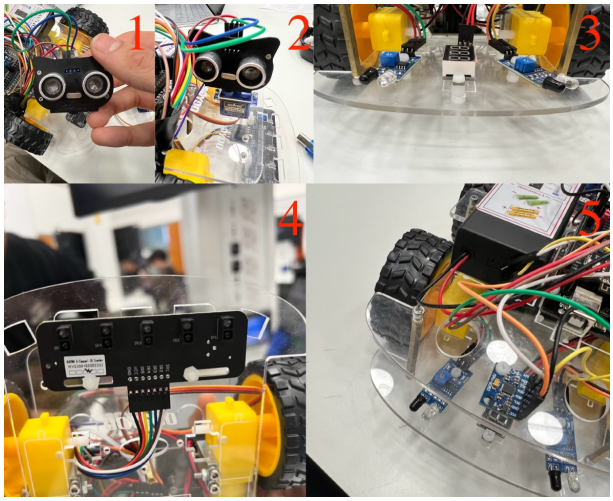


Figure 2: Additional fabrication steps of mobile robot

The servo motor was affixed to the top chassis of the mobile robot with two self-tapping Philips head screws. 1) Attachment of supplementary sensors began by fastening an HC-SRO4 Ultrasonic sensor to a plastic bracket. This bracket was attached directly to the 2) servo with a Philips head self tapping screw. 3) The IR obstacle avoidance sensors were attached to the back of the bottom chassis of the robot and connected to power, ground and two digital pins on the microcontroller (one digital pin for each module). 4) The IR line tracking module was attached to the front, bottom chassis and connected to the microcontroller over five analogue pins and power (one analogue pin for each sensor for a total of five sensors.) 5) The accelerometer was attached to the top chassis and connected to the microcontroller over I^2C by connecting the SDA and SCL pins together.

2.2 Arduino Uno Microcontroller

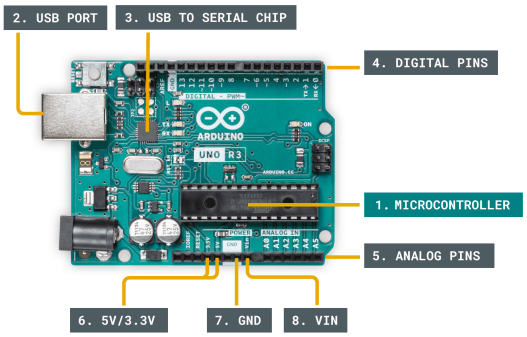


Figure 3: Labelled diagram of the UNO microcontroller [5]

The Arduino UNO is fully open-source microcontroller which is powered by an 8 bit ATmega328 Figure 3 (1). The ATmega328 is a low-power Reduced Instruction Set Computer (RISC). It contains 6 PWM channels, a real time counter, a 6 bit and 8 bit Analogue to Digital Converter (ADC) and is I^2c compatible. The ATmega328 only has 2kB of SRAM, which is tiny. However, this plays to a key strength of Bug Algorithm since they require very little memory to operate.

The UNO has 13 digital pins which can be used for reading Pulse Width Modulation (PWM) data, for example from the Ultrasonic sensor. It can also emit PWM signals required when interfacing with the L298N. Additionally, it has 5 analogue pins which are used to read continuous data such as the readings from a line tracking unit which contains five IR sensors, whom, each output an analogue reading.

2.2.1 Interfacing with mobile robot

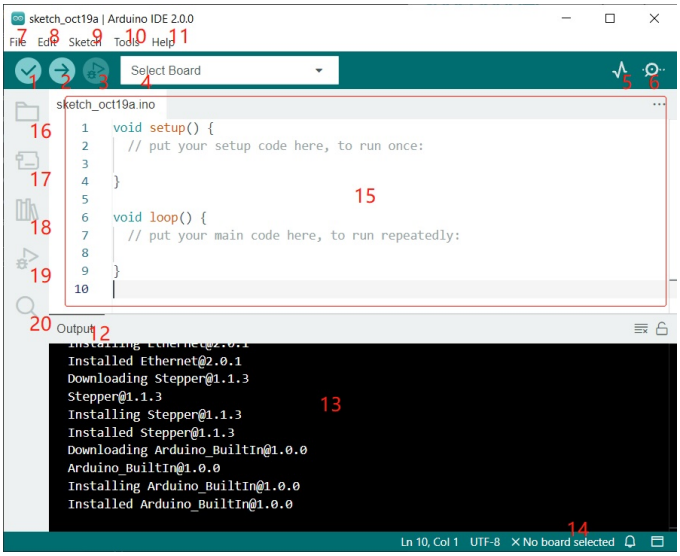


Figure 4: A numbered diagram showing the features of the Ardiuno IDE [6].

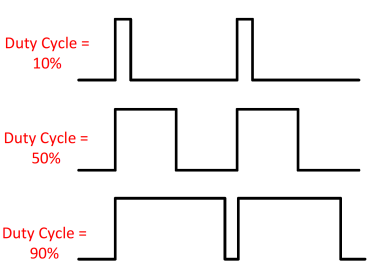
The UNO is connected to a host computer by USB and communication is initiated through UART. The UNO can be programmed with the Arduino Interactive Development Environment (IDE). It provides a easy to use graphical interface. It contains all the common IDE features as seen in Figure 4 such as: a code editor with syntax highlighting (15), auto-completion, a debugger (19), installing dependencies (20), file management (16), and an in IDE terminal (13). Where the IDE is Unique is with its serial monitor (5). Which allows a user to monitor communication over USB, this is especially useful for reading data from the mobile robot sensors' for experimental purposes and for calibration. A program can be uploaded to the UNO by first selecting the board type and the port to which it is connected to; (4) it can then verified and compiled to the Board (3).

The Universal Asynchronous Receiver-Transmitter (UART) protocol is used to send data between a computer and Arduino board to uploading a code, or read data directly from an Arduino [5]. UART is independent of a clock signal. Both devices must agree on a communication speed baud rate (in bits per second). The UART protocol only needs 3 pins to operate RX, TX and a shared ground between the devices. A UART signal consists of start bit which is a transition from an idle high signal to a low voltage. Then a set of data bits are sent (0 is low and 1 is high), an optional error checking bit can be sent which is called a parity bit which checks all the previously sent bits, and is set to zero or one to make the total number of high signals either even or odd (depending on which type of parity). Lastly a stop bit is sent which is a transition from a low to high voltage.

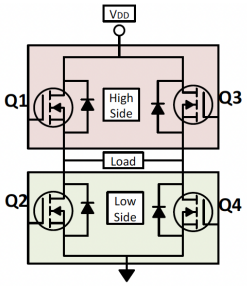
The Arduino programming language is a variant of C++ with extra functions, classes and methods built in. For example, it contains a built in class called Serial which allows for the output of characters to the serial monitor at an agreed upon baud rate. Furthermore, it also contains pulse width abstraction methods which allows the user to move the mobile robot with simple commands.

3 Robot Locomotion & Navigation Algorithms

The locomotion of the mobile robot consists of four geared DC motors which are controlled using PWM with the L298N dual H-Bridge motor controller. A H-Bridge is a cleaver arrangement of four electrically activated switches called mosfets and as seen in Figure 5b. By choosing which mosfets are activated, the arrangement changes the



(a)



(b)

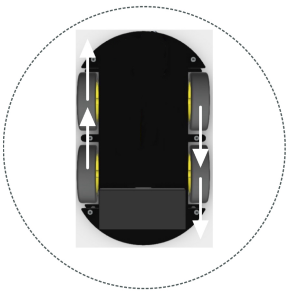
Figure 5: (a) Pulse Width Modulation Scheme for H-Bridge [7]. (b) Mosfet and Diode configuration for H-Bridge [7].

polarity of the motors, hence, switching their direction. PWM is a square wave signal and is used in conjunction with the H-bridge to control the speed of the motors, this is achieved by rapidly switching the motor on or off via the mosfet. The duty cycle is the percentage of time that the square wave is held high against the total time period a graphical representation of the is can be found in Figure 5a.

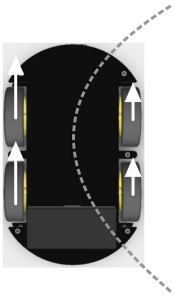
The Arduino can interface with the H-Bridge by using the built in `digitalWrite(args:motorPin,(HIGH/LOW))` method, which sets the motor's direction to either forward or reverse. The motor's speed can be set by the `analogWrite(hBridgePwmPin,(0-255))` method, which takes a value from 0-255 where 0 is off and 255 is max speed. This method adjusts the duty cycle of the pulse, obfuscating it from the programmer, thus making it easier to program.

Combining these methods for speed and direction, for each motor; enabled the creation of bespoke methods such as `moveForwards`, `moveBackwards`, `moveRight` and `moveLeft`. These are easy to use and reduce the need for code duplication, furthermore, they are self explanatory and do not require the underlying knowledge to operate them; meaning a programmer use them with ease.

3.1 Skid Steering Kinematics



(a)



(b)

Figure 6: (a) The direction of movement for each individual motor, inducing a point turn within the smart car. (b) The direction and magnitude of each individual motor inducing a swerve or skid within the robot.

By changing the relative magnitude and direction of each individual motor, allows one to manipulate the locomotion of the smart car. It is not only possible to do the rudimentary movement such as forwards and backwards but skid steering also allows for both point turns and skids.

Point turns allow the robot to turn in place, which is great for sharp turns and repositioning of the robot. These turns are achieved by moving the left and right motors in equal and opposite direction. In Figure 6a the locomotion causes the robot move counter-clockwise.

In contrast Skid steer works in a similar fashion to traditional car steering without the need for a complex rack and pinion system with lots of moving parts. By increasing the speed of the left motors it and decreasing the speed of the right motors as shown in Figure 6b, it causes the robot to turn right.

3.2 Sensors and Actuators for Robot Navigation

3.2.1 Ranging - HC-SR04 Ultrasonic Sensor

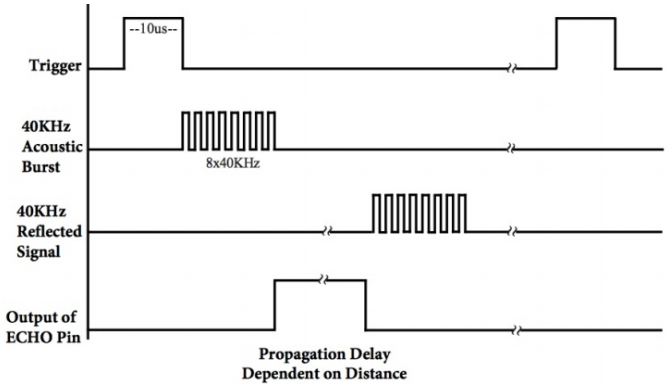


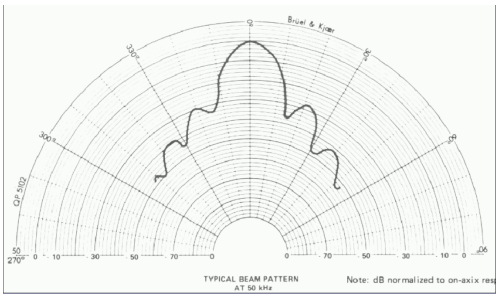
Figure 7: Pulse Width Modulation Scheme for ultrasonic sensor [8].

The mobile robot contains numerous sensors used for ranging and mapping. Most notably is the HC-SR04, which is an PWM controlled, Ultrasonic sensor, which tracks distances to obstacles in the vicinity of the robot. These sensors use two piezoelectric transducers to transmit and receive waves with a frequency well above human hearing at 40kHz.

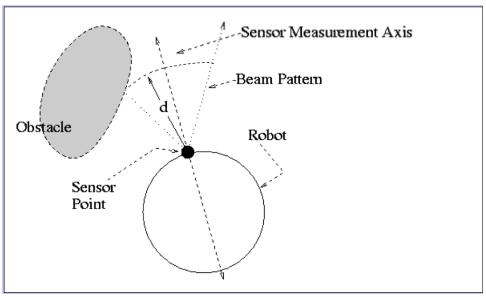
The HC-SR04 sensor has 4 pins, GND and 5V for power and two others which need to be connected to the digital pins on the Arduino, being Trigger and Echo. To emit an Ultrasonic pulse the Trigger pin must be provided with a $10\mu s$ square wave that transitions from 0 to V_{cc} (where $V_{cc} = 5V$ *or* $3.3V$). This then causes the Trigger to emit 8x40KHz short pulses. Then, the Echo pin is initiated to high or V_{cc} by the on-board chip. Once the short burst is received by the Echo pin it is set to low or $0V$.

The time between between sending and receiving the burst is outputted. This time corresponds the distance from the sensor to the object and then back once again. This means that the distance must be divided by two to get the absolute distance to the object [8].

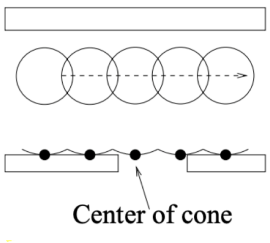
3.2.2 Problems associated with Ultrasonic sensors



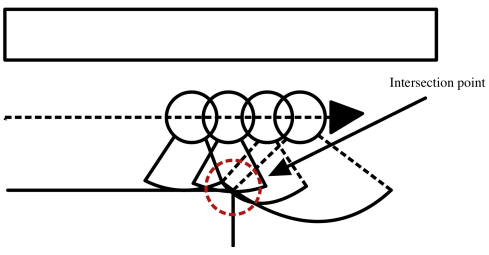
(a)



(b)



(c)



(d)

Figure 8: (a) Beam pattern of ultrasonic sensor (b) (Naïve) Sensor Model (c) Problems with Naïve (centre-line sensor) Model in Robotics. Visualisation of the Arc traversable method [4].

Azimuth uncertainty: Ultrasonic sensors (US) emit beams cone-shaped beams of waves, not just a singular point. This means that at large distances it is hard to pinpoint which object the wave reflected back from. One cannot just assume that the object is directly in front of the sensor as this would cause an large uncertainty in the distance measurement.
Specular reflections: Objects may reflect sound waves differently which could mean that the distance reflected back echo transducer not be as expected. For example if we had two mirror-like surfaces reflecting the sound between each other, before reaching the echo transducer, the value received would not be the distance to the first mirror.
Multi pass: Is again to do with mirror-like surfaces, if the reflective object is placed at an angle to the US it could reflect the sound wave in a different direction, causing it to never be received by the sensor.
Sensitivity to pressure and temperature changes: This is a pretty self explanatory problem, sound waves travel at different speeds in different pressures/temperatures. This means that additional sensors would have to be used in conjunction to calibrate the sensor for different conditions. for example with a robot that handles nuclear waste.
Minium Range: Each ultrasonic has different minium range (blind zone) since the sensors' emitter and receiver have to be rapidly turned on and off and they both cannot be on at once. This is because the sensor must switch between transmitting and receiving sound, so it can't do both at the same time.

3.2.3 Minimising Azimuth Uncertainty (Arc Transversable Median (ATM))

Since the beam pattern emitted by the US is not a point but rather a Gaussian function as shown in Figure 8a. Due to this fact, one cannot assume that the distance processed by the US is exactly in the centre this called a Naïve approach. This Naïve assumption falls short when the robot is trying to navigate small gaps as seen in Figure 8c.

When a robot gets further away from a corner (but is still being detected by the Ultrasonic sensor;) the angle of the reflected sound wave back towards the ultrasonic sensor increases. From this an arc can be drawn around the hit point. If the distances were also stored from just before moving off the corner, more arcs could be produced. The intersection between these arcs dictates where a corner is as shown in Figure 8d.

3.2.4 Odometry - Inertial Measurement Unit (IMU)

An Inertial Measurement Unit (IMU) uses a clever arrangement of microelectromechanical systems; where each component may only be a few microns across. These systems allow the IMU to measure the acceleration in a particular direction on the mobile robot. Furthermore, IMU's also contain an a gyroscope to detect the angular velocity of the mobile robot. Lastly a IMU may contain a magnetometer which measures the earth's magnetic field. This combination of sensors measures in nine degrees of freedom which allows the robot to rotate accurately and track the position of the robot by using integration of the acceleration values. Sensor fusion techniques such as a kalman filter could be used to improve the accuracy of these readings.

3.2.5 Actuator - Servo Motor

A servo motor is a special type of motor that contains a closed loop control system so it to precisely move between angle in a given range, the SG-90 servo motor used within the smart car has a range of 0-180 degrees. The servo motor also operates over PWM, just like the Ultrasonic sensor. Programming the servo motor is simple, the arduino IDE contains libraries that can be included within the codebase which abstract the underlying pulse widths. When using a servo motor in the arduino IDE a servo object must be created from which the `servo.attach(arg:digitalPin)` method can be called which dictates the digital pin the servo is attached to. Then the `servo.write(arg:angle)` can be called which determines the position of the servo motor in degrees (between 0-180.) These two methods make it extremely simple to control the servo motor.

3.3 Algorithms

3.3.1 Line Tracking Algorithms

The line tracking algorithm used the IR sensor attached to the bottom chassis of the robot to operate. The IR sensor works by emitting a beam of inferred light and measures how much light is reflected back. For a light surface a '0' would be returned as most if not all the light is reflected back to the receiver. Whereas for a darker object a '1' would be returned as only a small amount of light would be sensed by the receiver.

Within the codebase sensor values are encoded as a 5 character string, with each character corresponding to the value of a single sensor on the IR module, from this one can infer if the robot is on the right or left of the robot and can adjust the direction of motion of the robot accordingly.

The code in Listing 1 uses `if` conditions to determine whether the line is on the right or left side of the robot. The motor speed within this example has fixed values, however this could be slightly improved by using a proportional controller to adjust the speed linearly.

The proportional control approach might see a marginal increase in the efficiency of the overall algorithm, Since the sensor only returns discrete values (reducing the sampling interval).

Listing 1: Pseudocode for line tracking

```
/*A proccedure using values from IR line tracker to
follow a black line. This method can be called in the loop method to run infinitely. /*
Proccedure LineTrack(){
        String SENSOR VALUE = Call GetSensorValues // Get sensor values from line tracker
        from all five IR sensors
        if (SENSOR VALUE CORRESPONDS TO BEING ON FAR RIGHT OF LINE){
                //The black line is in the right of
                the car, turn right to get back on the tape
                SET MOTORS FAST
                RIGHT
        }
        if (SENSOR VALUE CORRESPONDS TO BEING ON SLIGHT RIGHT OF LINE){
                // Robot is slightly right
                of line so it must move
                right to get back on the tape.
                SET MOTORS MEDIUM
                RIGHT
        }
        if (SENSOR VALUE CORRESPONDS TO BEING ON FAR LEFT OF LINE){
                // Robot is on left of line
                so it must move left to get
                back on the tape.
                SET MOTORS FAST
                LEFT
        }
        IF (SENSOR VALUE CORRESPONDS TO BEING ON SLIGHTLY LEFT OF LINE) {
                // Robot is on slightly left of line
                so it must move left to get back on the tape.
                SET MOTORS MEDIUM
                LEFT
        }

        if (ALL IR SENSORS COVERED){
                //The robot is at the end of the
                line, so it needs to stop.
                STOP
        }
}
```

3.3.2 Wall Following Algorithm & Obstacle and Avoidance Algorithm (Left Turning Robot)

Wall following and obstacle avoidance within the robot was created by leveraging position of the servo motor and manipulating of the locomotion of the robot. The initial approach to solving this problem was to turn the servo-fully to the right (0 degrees), then if smart car was heading towards the wall it would turn right and the robot would turn left if it was moving away from the wall to counteract this.

This approach fell short in two key areas. Firstly, the robot was unable to navigate sharp corners this was due to the point turn approach of locomotion for the robot and not taking advantage of the skid steering, causing the robot to get stuck between the two sides of the corner. Secondly, the robot was extremely slow, since it had to keep adjusting it's angle of attack, and would rarely move forwards parallel to the wall or obstacle, rather it would move askew to the wall.

Resolving these issues was achieved by changing the position of the servo. Then to allow for the robot to move around sharp edges a new locomotion method was employed by allowing the robot to swerve left and right rather

than it turning in place, allowing the robot to curve round corners in a fluent manner. This was implemented by sending half speed to one side of the motors and full power to the other side.

Listing 2: Pseudocode for Wall Following Algorithm

```
// A Proccedure used to avoid obstacles and walls. (left turning robot)
Proccedure WallFollowing(DISTANCE, TARGET DISTANCE){
        MOVE SERVO 35 // Move Servo Motor position to 35 degrees,
        facing the right side of the robot but not fully to the right)
        if (DISTANCE > TARGETDISTANCE + 10){ // When the smart car is too far from the wall
                SWERVE Right // Swerve the smart car so it can
                peer around corners and not get stuck on sharp edges;
                STOP
        }
        else if (DISTANCE < TARGETDISTANCE − 10){ // when the smart car is too close to the wall
                SWERVE LEFT
                STOP
        }
        else { // Car is within the bounds +/− 10 cm so it can continue forwards
                FORWARD // Move the smart car forward
                STOP
        }
        DISTANCE = NEW DISTANCE // Get distance of ultrasonic sensor after every move.
}
```

3.3.3 Proportional Control approach to Wall Following

Proportional Controller approach can improve the wall following model of the smart robot by adjusting the speed powering the motors. For example if the robot is straying away from the wall, a multiplier can be used to linearly control the robot rather than using a discrete, threshold value to determine if the robot goes right, left or forwards like in Listing 2.

The equation that governs the proportional control for wall following can be given by,

$$MOTOR\_SPEED = GAIN * |CURRENT\_DISTANCE - TARGET\_DISTANCE| \tag{1}$$

Listing 3: Pseudocode for Wall Following Algorithm

```
// A Proccedure used to avoid obstacles and walls with proportional control.
Proccedure proportionalWallFollow(DISTANCE, TARGET DISTANCE){
        MOVE SERVO 35 // Move Servo Motor position to 35 degrees,
        GAIN = 50
        facing the right side of the robot but not fully to the right)
        IF (DISTANCE > TARGETDISTANCE){ // When the smart car is too far from the wall
                SWERVE Right // Swerve the smart car so it can
                SPEED = GAIN * |CURRENT_DISTANCE−TARGET_DISTANCE| // smart car
                reaches max speed at 5.1 cm away from the target distance
                STOP
        }
        ELSE IF (DISTANCE < TARGETDISTANCE){ // when the smart car is too close to the wall
                SWERVE LEFT
                SPEED = GAIN∗|CURRENT_DISTANCE−TARGET_DISTANCE| // smart car reaches
                max speed at 5.1 cm away from the target distance
                STOP
        }
        DISTANCE = NEW DISTANCE // Get distance of ultrasonic sensor after every move.
}
```

3.3.4 Goal Tracking

This is an algorithm that is yet to be implemented to the mobile robot. However, some methods were investigated within this paper such as Wi-Fi and Bluetooth to test their feasibility for this task. These approaches would use

the Received Signal Strength Indicator (RSSI) values (measured in decibels) to check if the robot is getting closer or further away from the goal, code for this can be found in Appendix B.

When the robot detects an obstacle, the values from these sensors can be combined obstacle avoidance to create a fully functional Bug 1 algorithm.

When the robot is navigating around an Obstacle for the first time a unique RSSI value could be stored within the program for each movement the robot makes and once the robot has made one complete cycle (determined by the IMU) of the obstacle the lowest RSSI value could be computed. On the next cycle the lowest RSSI value can be compared to the current one and once it is close or matches the lowest value the robot knows when to leave at that point and the process repeats until it reaches the goal.

In theory this fully functional Bug 1 algorithm would have a space complexity of just $O(1)$ as it would only have to store the lowest RSSI value within the program. This reinforces the fact that BAs have low memory requirements.

3.3.5 Bug algorithms

A 'pseudo' Bug 2 algorithm was implemented into the robot by combining the line tracking algorithm and obstacle avoidance algorithms. These procedures interacted with one another by using `while` statements and constantly switching between each mode. For example, the initial state the robot defaults to line following, and when an obstacle detected within 30cm of the US the mode switches to the wall and obstacle avoidance mode, again similarly if the robot detects a black line after navigating the obstacle it will then activate the line tracking mode and this process repeats until the goal is reached. Having to switch modes constantly, is a key limitation of Arduino since it only has one core and only one process can happen at the time. Instead, a much faster approach would be to have each process running at the same time, using parallelisation on multiple cores.

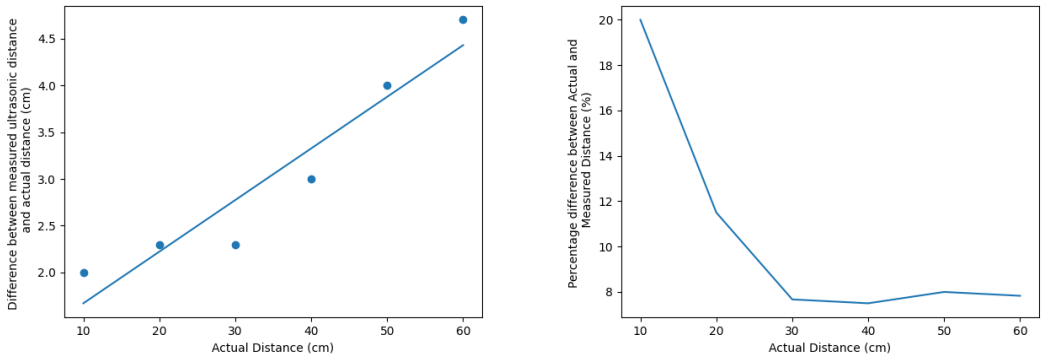
Listing 4: Pseudocode for a Bug 2 algorithm

```
/* Two methods that interact with one another to produce a Psudo Bug Algorithm
by constantly switching between line tracking and wall/obstacle avoidance /*

/* An extension of Listing 1 with and additional while loop */
Begin Line Track Bug(){
        SET NO OBSTACLES true
        while (NO OBSTACLE){
                CALL LineTrack // the method used to track the black line (as seen in Listing 1)
                if (ULTRASONIC DISTANCE IS LESS THAN THE TARGET DISTANCE){
                        TURN RIGHT // turn right until the robot is perpendicular to the wall
                        NO OBSTACLES false //exit out of while loop
                        CALL ObstacleAvoidanceBug(DISTANCE, TARGET DISTANCE)
                        // start obstacle avoidance
                }
        }
}

/* An extension of Listing 2 with and additional while loop */
Begin ObstacleAvoidanceBug(DISTANCE, TARGET DISTANCE){
        SET NO LINE true
        while (NO LINE){ // Check if their is no line to follow.
                CALL WallFollow // call obstacle avoidance procedure (as seen in Listing 2)
                if (LINE TRACKER MODULE DETECTS BLACK LINE){
                        // go back to line following
                        NO LINE false // exit out of while loop
                        CALL LineTrackBug // Start line tracking
                }
        }
}
```

4 Results

4.1 Ultrasonic Distance Sensor Readings

The readings from the sensor were experimentally gathered by using the code found in Appendix B. By placing a non-reflective object in front of the ultrasonic sensor it would return the raw distance from the sensor in centimetres. These values were then compared with the distance measured from a ruler. When gathering the distances, the ruler was placed behind the PCB of the ultrasonic sensor to avoid any random error. However, this introduced a systematic error as the transducer is enclosed in a metal casting so it is hard to quantify where the sound waves were emitted from. Therefore the readings will always be an underestimate from the true values. Each point was measured three times to calculate an average, and the raw data can be found in the Appendix A.



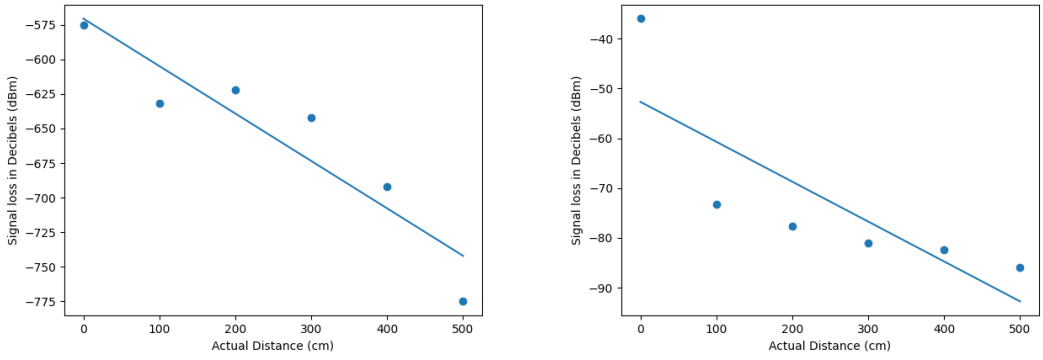
(a) (b)

Figure 9: (a) Errors associated with HC-04 ultrasonic sensor in centimetres (b) Percentage errors associated with HC-04 ultrasonic sensor.

As expected, the results from Figure 9a show that all of the values measured were an underestimate of the true distance. As the distance increased the accuracy of the module also increased as shown in Figure 9b from 20% at 10cm to 7.83% at 60cm. In the environment tested, above 60cm, the module was susceptible to Azimuth uncertainty a key problem associated with Ultrasonic sensors, as mentioned previously. The environment used for the BAs is within the 50-30cm range so the Ultrasonic sensor is suitable for this task.

4.2 Wi-Fi Module Readings & Bluetooth Readings

A Raspberry Pi Pico W was used as a Wireless Access Point (WAP) in conjunction with the smart robot to conduct this test. This experiment was used to check the suitability of Wi-Fi to be used for the goal in BAs by using the RSSI values to determine signal loss between the goal and the robot, this could be used to estimate the distance to the goal and produce a fully functional Bug 1 algorithm.



(a) (b)

Figure 10: (a) A linear regression analysis on the average RSSI values from the WIFI module at different distances (b) A linear regression analysis on the average RSSI values from the Bluetooth module at different distances.

The Bluetooth data was gathered in a similar fashion by using the Bluetooth module on the robot shield and a mobile phone to receive the RSSI values. For both Wi-Fi and Bluetooth each point was measured three times to calculate and average and a linear regression was conducted on each graph to test the the correlation and accuracy of the results.

4.3 Testing of Navigation Algorithms

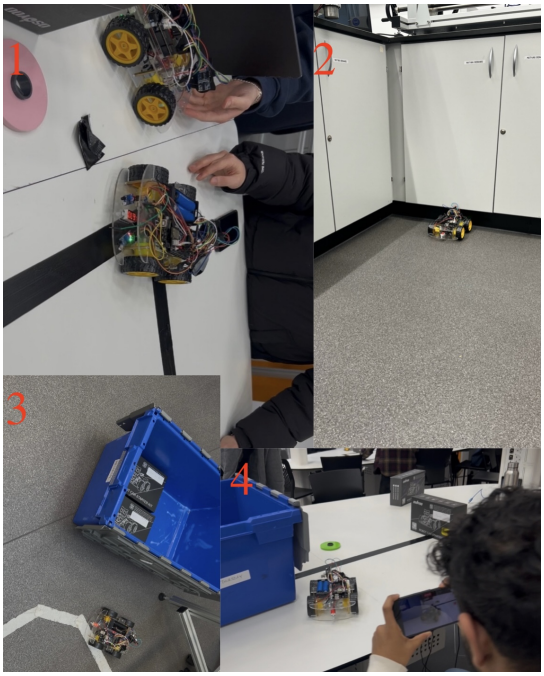


Figure 11: 1) Testing environment for line tracking algorithms. 2) Testing environment for wall following algorithms. 3) Testing environment for Obstacle Avoidance algorithms. 4) Testing environment for the 'pseudo' Bug 2 algorithm.

4.3.1 Line tracking

The line tracking algorithm really struggled when navigating sharp corners, it was unable to navigate for which the angle of turn exceeded 45°. In our test case the robot was never able to negotiate these turns. Since, when turning, all of the line tracking sensors become covered which causes the robot to stop as seen in Listing 1.

4.3.2 Wall Following

This algorithm performed well on most walls around the test environment. The robot managed to maintain a good following distance between the walls; within the specified target distance. However, the wall following algorithm really struggled when navigating concave corners as seen in Figure 11, where the robot would not be able predict the corner due to the servos position and would cause the robot to crash into the wall. This was not the case for convex corners.

4.3.3 Obstacle Avoidance

The Obstacle Avoidance algorithm worked well on both square and cylindrical objects throughout the test cases, the only place the algorithm struggled was on very thin objects such as chair legs.

4.3.4 Bug 2

This strategy was tested thoroughly, the robot had to navigate 3 different objects in succession, with line following tape in between them: the first object was a large box; second was a small box and last was a small cylindrical object. Table 1 summarises the results of these experiments.

Table 1: Summary of Bug algorithm test results

Test	Was the robot able to navigate the section object?			Comments
	Object 1	Object 2	Object 3	
1	Yes	Yes	No	Robot overshot the very small cylindrical object
2	Yes	No	No	Robot managed to navigate first object but got stuck when switching to line following.
3	Yes	Yes	Yes	Robot managed to traverse all objects
4	Yes	Yes	No	Robot's tires got stuck on the second object
5	Yes	No	No	Robot took the corner too sharply when performing obstacle avoidance causing it to crash into the smaller box
6	No	No	No	Robot didn't turn far enough in the initial turn causing the ultrasonic sensor to not be parallel with the object that needed to be navigated.
7	Yes	No	No	Robot's battery voltage was low causing the left motors to not turn when set to lower speeds (as it uses skid steer here.)

5 Discussion

The robot smart car was capable of a "pseudo" Bug 2 algorithm (found in Appendix B). Since BAs are usually represented in a simulation environment, these assume complete accuracy from all sensors and actuators. The source material was also presented within a simulated environment. This meant that for each component of the 'pseudo' Bug 2 algorithm it had some key differences with the simulator.

The reason why the real world example is called a 'pseudo' Bug 2 algorithm is due to the robot not having any information of the location of the goal, instead a black line was used to act as the straight line vector between the start and the goal; this line could then be traversed with the line tracking module. Obstacles along the path of black tape were navigated in the same way as in the source material, although, since a ultrasonic sensor was used instead of a bump switch, thus, the robot needed to have a minium following distance around the object.

The overall reliability of the Bug 2 was mixed. Throughout the trials, on average, the mobile robot traversed 1.42 out of the 3 obstacles along the path. This algorithm failed for a multitude of different reasons, some partially due to the hardware such as the battery being too low and some from the limitations of the software such as getting stuck when all sensors are covered with the line tracker. These results reinforce the unpredictability of creating such algorithms in a non-deterministic, real-world environment.

The Bug 1 algorithm could not be implemented within the allocated time, however some methods were investigated by combining the obstacle avoidance with a proper goal tracking algorithm. As mentioned previously goal tracking could be implemented by harnessing the power of Wi-Fi + Bluetooth module. From these results, the

approach most suitable for this task would be using Wi-Fi rather than Bluetooth due to the small reduction in signal strength beyond 200cm in Figure 10b. In contrast to Wi-Fi where it only seems to struggle at close ranges, Which can be solved with the addition of an IR emitter at the goal which can be received by the pre-installed IR receiver module in the kit.

In theory this Bug 1 algorithm would be identical to the one presented within the source material, as the position of the goal would always be known by the robot, thus an estimate for the distance and direction of the goal could be found.

The Bug 2 algorithm produced has a constant space complexity of $O(1)$, and in theory the Bug 1 algorithm that could be produced would have constant memory complexity of $O(1)$ as well. This is due to the fact that neither of algorithms need to store much data using complex data structures. This reinforces the simple working principle of BAs and little need for memory.

6 Future Work

Generally, each individual component of the Bug 2 worked well, but some further optimisations could be made. Firstly, in the environment tested the Ultrasonic sensor was affected by Azimuth uncertainty and the minium range problem when navigating obstacles. A time of flight sensor could used to eliminate these problems as they use light rather than sound for ranging.

Although, skid steer provides a range of different movement options as mentioned previously. The most versatile drive system would be using a 4-wheel-mecanum drive as it allows a robot to navigate in every direction (omni-directionally). This would be especially useful for implementing a simulation environment as massively abstracts the complexities in modelling the movement of the robot.

The addition of a camera could also allow for more complex ranging strategies by using fiducial markers such as April tags, especially for the goal tracking as they can be used for both distance and for gathering relative angles from the robot to the tag with a monocular vision system.

7 Conclusions

In this report, robotic navigation algorithms were explored. From this the following has been discovered:

1. Bug Algorithms have very low memory requirements.
2. Bug Algorithms can be implemented on low performance hardware.
3. Wi-Fi communication is suitable to implement a Bug 1 algorithm on the mobile robot.
4. The uncertainty of the real world makes it difficult to produce a reliable Bug algorithm.

References

[1] OSOYOO. "Osoyoo v2.1 robot car for arduino: Introduction model2019012400." (May 2020), [Online]. Available: https://osoyoo.com/2020/05/12/v2-1-robot-car-kit-for-arduino-tutorial-introduction/.

[2] K. McGuire, G. de Croon, and K. Tuyls, "A comparative study of bug algorithms for robot navigation," *Robotics and Autonomous Systems*, vol. 121, p. 103 261, 2019, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2019.103261. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889018306687.

[3] K. Liu, "A comprehensive review of bug algorithms in path planning," *Applied and Computational Engineering*, vol. 33, pp. 259–265, Jan. 2024. DOI: 10.54254/2755-2721/33/20230278.

[4] P. K. Althoefer and S. Hussain. "Introduction to robotics." (Sep. 2024), [Online]. Available: https://qmplus.qmul.ac.uk/pluginfile.php/4385606/mod_resource/content/1/Intro%20to%20Robotics-4_Sensors_QMUL%20%282024%29.pdf.

[5] K. Söderby. "Arduino docs." (Jul. 2024), [Online]. Available: https://docs.arduino.cc/learn/starting-guide/getting-started-arduino/#quick-reference.

[6] xiemeiping. "Introduce of arduino ide." (Oct. 2022), [Online]. Available: https://docs.sunfounder.com/projects/3in1-kit/en/latest/arduino_start/introduce_ide.html.

[7] M. Prakash Arjun. Toa. "Current sensing in an h-bridge." (Apr. 2023), [Online]. Available: https://www.ti.com/lit/ab/sboa174d/sboa174d.pdf?ts=1730999645085&ref_url=https%253A%252F%252Fwww.google.com%252F.

[8] C. Schmitz. "Ultrasonic (us) sensor." [Accessed 30 October 2024]. (Sep. 2024), [Online]. Available: https://www.electroschematics.com/hc-sr04-datasheet/.

8 Appendix

A Data

Table 2: Experimental values from the ultrasonic sensor, calculated averages and percentage difference from the theoretical values.

Actual Distance (cm)	Measured distance (cm)			Average (cm)	Error (cm)	Percentage Error (%)
	1	2	3			
0	Minium Distance Error	Minium Distance Error	Minium Distance Error	0	0	0
10	8	8	8	8	2.00	20.0
20	17	18	18	17.7	2.30	11.5
30	28	28	27	27.7	2.30	7.67
40	38	37	36	37.0	3.00	7.50
50	46	46	46	46.0	4.00	8.00
60	55	56	55	55.3	4.70	7.83

Table 3: Experimental values from the Bluetooth module

Actual Distance (cm)	Signal loss (dBm)			Average (dBm)
	1	2	3	
0	-39	-35	-34	-36
100	-76	-76	-68	-73.33
200	-76	-74	-83	-77.66
300	-86	-76	-81	- 81
400	-81	-76	-90	-82.33
500	-88	-85	-85	-86

Table 4: Experimental values from the Wi-Fi module

Actual Distance (cm)	Signal loss (dBm)			Average (dBm)
	1	2	3	
0	-572	-582	-572	-575
100	-632	-632	-632	-632
200	-622	-622	-622	-622
300	-642	-642	-642	-642
400	-702	-682	-692	-692
500	-772	-782	-772	-775

B Codes

Listing 5: Code to get distance measurement from ultrasonic sensor in cm.

```
/* A method used to return the ultrasonic reading of the HC-SR04 in cm */
int getDistance(){
        digitalWrite(TRIG_PIN, LOW); // Clears the Pin
        delayMicroseconds(2);
        digitalWrite(TRIG_PIN, HIGH);
        delayMicroseconds(10); // Sets the trigPin on HIGH state for 10 micro seconds
        digitalWrite(TRIG_PIN, LOW); // Reads the echoPin, returns the sound wave travel
         time in microseconds
        long duration = pulseIn(ECHO_PIN, HIGH); // Calculating the distance
        int distance = duration * 0.034 / 2;
        return distance; // returns distance reading in cm
}
```

Listing 6: Wall following algorithm code

```
/* A method used to avoid obstacles and walls, this function can be called by the main loop so it can run forever
void leftWallFollowAvoidOstacles(int distance, int targetDistance){
        anyPosition(35); // Move Servo Motor position to 35 degrees
        if (distance > targetDistance + 10){ // When the smart car is too far from the wall
                swerveRight(180); // Swerve the smart car so it can
                peer around corners and not get stuck on sharp edges;
                the Swerve function takes an additional pwm speed parameter from 0 (off) to 255 (max speed)
                delay(50); // Leaves the motors on for 50ms
                Stop(); // Stop all of the motors
        }
        else if (distance < targetDistance - 10){ // when the smart car is too close to the wall
                swerveLeft(180);
                delay(50);
                Stop();
        }
        else{ // Car is within the limits, so it can continue forwards
```

```
                setMotorSpeed(180); // Set the motor speed to aprox. 70% of the max speed
                moveForward(); // Move the smart car forward
                delay(10); // Delay has to be short as the smart car could encounter a sharp corner
                Stop(); // Stop all motors
            }
            distance = getDistance(); // Get distance of ultrasonic sensor after every move.
            return;
        }
```

Listing 7: Code to get distance measurement from ultrasonic sensor in cm.

```
        /*A function using values from IR line tracker to follow a black line;
        to follow a white line the inverse of values would have to be used */
        void lineTracking(){
            String sensorVal = readsensorValues(); // Get sensor values from all five IR sensors
            if (sensorVal == "10000")
            {
                //The black line is in the left of the car, need left turn
                setMotorSpeed(FAST_SPEED,FAST_SPEED);
                moveLeft(); // Turn left
            }
            if (sensorVal =="10100" || sensorVal=="01000" || sensorVal=="01100"
            || sensorVal =="11100" || sensorVal=="10010" || sensorVal=="11010")
            // Robot is on the right so it must need to move left.
            {
                setMotorSpeed(180);
                moveLeft(); // Move left

            }
            if (sensorVal=="00001"){ //The black line is on the right of the car, so it must turn right to stay on the
                setMotorSpeed(180);
                moveRight(); //Turn right

            }
            if (sensorVal=="00011" || sensorVal=="00010"
            || sensorVal=="00101" || sensorVal=="00110" || sensorVal=="00111"
            || sensorVal=="01101" || sensorVal=="01111" || sensorVal=="01011" || sensorVal=="01001")
            // Robot is almost in the center of the .
            {
                setMotorSpeed(180);
                moveRight(); // Move right
            }

            if (sensorVal=="11111"){
                Stop(); //The robot is at the end of the line, so it needs to stop.
            }
        }
```

Listing 8: Code for outputting RSSI values to the Arduino Terminal.

```
void setup()
{
    Serial.begin(9600); // initialize serial for debugging
    Serial1.begin(115200); // initialize serial for ESP module
    Serial1.print("AT+CIOBAUD=9600\r\n");
    Serial1.write("AT+RST\r\n");
    Serial1.begin(9600); // initialize serial for ESP module

    WiFi.init(&Serial1); // initialize ESP module

```

```
11          // check for the presence of the shield
12          if (WiFi.status() == WL_NO_SHIELD) {
13                  Serial.println("WiFi shield not present");
14                  while (true); // don't continue
15          }
16
17          // Serial.print("Attempting to start AP ");
18          // Serial.println(ssid);
19          //AP mode
20          //status = WiFi.beginAP(ssid, 10, "", 0);
21
22          //STA mode
23          while ( status != WL_CONNECTED) {
24                  Serial.print("Attempting to connect to WPA SSID: ");
25                  Serial.println(ssid);
26                  // Connect to WPA/WPA2 network
27                  status = WiFi.begin(ssid, pass);
28          }
29
30          Serial.println("You're connected to the network");
31          Serial.println();
32
33
34
35          printWifiStatus();
36
37          Udp.begin(localPort);
38
39          Serial.print("Listening on port ");
40          Serial.println(localPort);
41  }
42          void printWifiStatus()
43          {
44  while (true){
45                  Serial.println(WiFi.RSSI());
46  }
```

Listing 9: Final Bug algorithm code.

```
1   void bugRightTurn(int distance, int targetDistance){
2     //stage 1, need to get to the target distance
3     bugStageOne(targetDistance);
4     return;
5   }
6   /* get robot to the required distance */
7   void bugStageOne(int targetDistance){
8     bool reachedTarget = false;
9     centerServo();
10    int distance = getDistance(); // get distance of ultrasonic sensor
11    while(!reachedTarget){
12      if (distance > targetDistance + 5){
13        setMotorSpeed(128);
14        moveForward();
15        delay(100);
16        Stop();
17      }
18      else if (distance < targetDistance - 5){
19        setMotorSpeed(128);
20        moveBack();
21        delay(100);
```

```
22          Stop();
23        }
24        else{
25          reachedTarget = true;
26        }
27        distance = getDistance(); // get distance of ultrasonic sensor
28      }
29      bugStageTwo(targetDistance);
30      return;
31    }
32    /* A method used to rotate the robot 90 degrees */
33    void bugStageTwo(int targetDistance){
34      for(int i = 9; i >= 0; i--){
35        int distance = getDistance();
36        setMotorSpeed(128);
37        moveLeft();
38        delay(50);
39        Stop();
40        anyPosition(63);
41      }
42      bugStageThree(getDistance(), targetDistance);
43      return;
44    }
45
46
47
48    /* A method used to avoid obstacles after being detected in bugStageTwo and walls */
49    void bugStageThree(int distance, int targetDistance){
50      bool noLine = true;
51      while (noLine){ // Check if there is no line to follow.
52        if (distance > targetDistance + 5){ // When the smart car is too far from the wall
53          swerveLeft(255); // Swerve the smart car
54          so it can peer around corners and not get stuck on sharp edges; the Swerve function takes an additional
55          pwm speed parameter from 0 (off) to 255
56          (max speed)
57          delay(50); // Leaves the motors on for 50ms
58          Stop(); // Stop all of the motors
59        }
60        else if (distance < targetDistance - 5){ // when the smart car is too close to the wall
61        swerveRight(255);
62        delay(50);
63        Stop();
64        }
65      else{ // Car is within the limits, so it can continue forwards
66        setMotorSpeed(180); // Set the motor speed to ~ 70% of the max speed
67        moveForward(); // Move the smart car forward
68        delay(10); // Delay has to be short as the smart car could encounter a sharp corner
69        Stop(); //Stop
70       }
71       if (lineCheck() == false){
72        // go back to line following
73        noLine = false;
74       }
75
76      distance = getDistance(); // Get distance of ultrasonic sensor after every move.
77      }
78      autoTracking();
79      return;
80    }
81
```

```
82  char sensor[5];
83   /*read sensor value string, 1 stands
84   for black, 0 starnds for white, i.e 10000
85   means the first sensor(from left) detect
86   black line, other 4 sensors detected
87   white ground */
88  String read_sensor_values()
89  { int sensorvalue=32;
90    sensor[0]= !digitalRead(LFSensor_0);
91
92    sensor[1]=!digitalRead(LFSensor_1);
93
94    sensor[2]=!digitalRead(LFSensor_2);
95
96    sensor[3]=!digitalRead(LFSensor_3);
97
98    sensor[4]=!digitalRead(LFSensor_4);
99    sensorvalue +=sensor[0]*16+sensor[1]*8+sensor[2]*4+sensor[3]*2+sensor[4];
100
101   String senstr= String(sensorvalue,BIN);
102   senstr=senstr.substring(1,6);
103
104
105   return senstr;
106 }
107
108
109 void lineCheck(){
110     if (sensorval=="00000"){
111       // all good
112       return true;
113     }
114     else{
115       return false;
116     }
117 }
118
119 void autoTracking(){
120   bool noObstacles = true;
121   while (noObstacles){
122   String sensorval= read_sensor_values();
123     if (sensorval=="10000" )
124     {
125       //The black line is in the left of the car, need left turn
126       moveLeft(); //Turn left
127       setMotorSpeed(200);
128       Stop(); //The car front touch stop line, need stop
129
130     }
131     if (sensorval=="10100" || sensorval=="01000" ||
132     sensorval=="01100" || sensorval=="11100"
133     || sensorval=="10010" ||
134     sensorval=="11010")
135     {
136       moveLeft(); //Turn slight left
137       setMotorSpeed(150);
138       Stop(); //The car front touch stop line, need stop
139     }
140     if ( sensorval=="00001" ){ //The black line is on the right of the car, need right turn
141       moveRight(); //Turn right
```

```
142         setMotorSpeed(200);
143         Stop(); //The car front touch stop line, need stop
144       }
145       if (sensorval=="00011" || sensorval=="00010" ||
146       sensorval=="00101" || sensorval=="00110" ||
147       sensorval=="00111" || sensorval=="01101" ||
148       sensorval=="01111" || sensorval=="01011" ||
149       sensorval=="01001")
150       {
151         moveRight(); //Turn slight right
152         setMotorSpeed(SLOW_SPEED,0);
153         Stop(); //The car front touch stop line, need stop
154       }
155       if (sensorval=="01110"){
156         moveForward();
157         setMotorSpeed(200);
158         delay(250);
159         Stop(); //The car front touch stop line, need stop
160       }
161
162       if (sensorval=="11111"){
163         stop_Stop(); //The car front touch stop line, need stop
164         set_Motorspeed(0,0);
165       }
166       if (getDistance() < 30){
167         noObstacles = false;
168         bugStageOne();
169         return;
170       }
171     }
```

C IDE data

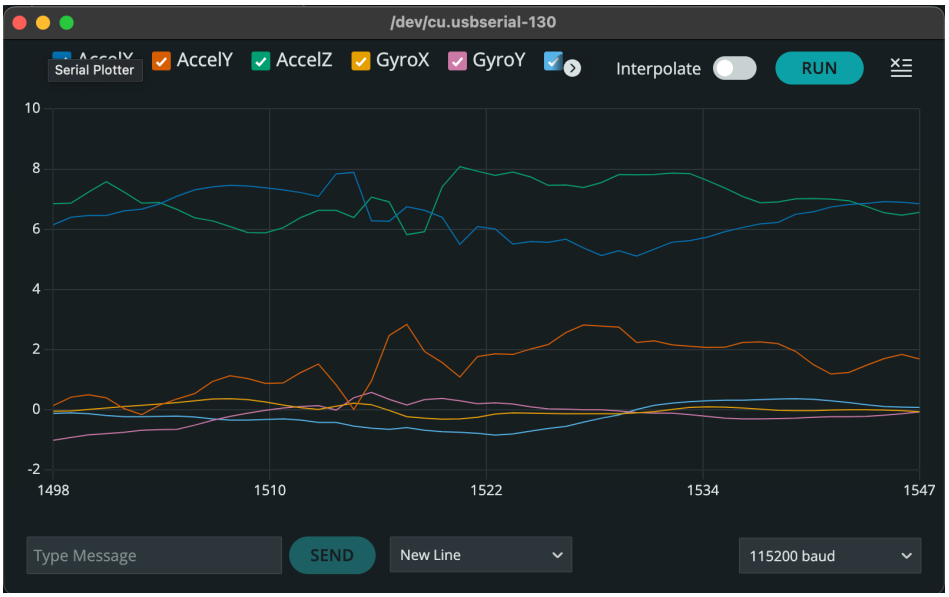


Figure 12: Accelerometer data on serial plot within the IDE