SweepR

Edwin Redhead, Xavier Parker, Kishan Grewal

Electronics Group 1, MEng Robotics and AI, Department of Computer Science, UCL

April 2025

1 Introduction

From their conception, teleoperated vehicles have played a pivotal role in robotics, progressing from early experimental models and consumer devices, such as radio-controlled (RC) cars, to sophisticated platforms used in industry, security, and research. Over time, advancements in sensing, control, and automation have enabled these systems to analyse their surroundings and adapt to dynamic environments, expanding their applications beyond simple remote operations.

Autonomous and semi-autonomous robots are now widely deployed in warehouses, military bases, and public spaces, where they monitor environments, detect intrusions, and adapt to dynamic conditions. Unlike systems such as fixed surveillance cameras, for example, these mobile systems can navigate obstacles, adjusting their routes based on real-time sensor data. Many rely on a combination of odometry and proximity sensing—using infrared (IR) and ultrasonic sensors—to estimate position, detect heat signatures, and gauge distances, even in low-visibility or high-risk environments.

Several well-established robotic platforms demonstrate the use of odometry and multi-sensor fusion for self-localisation and mapping. The Pioneer 3-DX [13], for example, employs wheel odometry to estimate its position within a space while integrating 16 ultrasonic sensors positioned around its chassis for full 360° obstacle detection. Similarly, the TurtleBot 3 [11] and 4 leverage IMUs (accelerometers and gyroscopes) alongside wheel odometry to enhance localisation accuracy, particularly in indoor environments. The Khepera IV [8], a compact research robot, integrates infrared proximity sensors with odometry for short-range obstacle detection and autonomous navigation. Meanwhile, the iRobot Create 2 [7], derived from the Roomba platform, offers an accessible platform for mobile robotics research, using odometry and bumper sensors to navigate confined spaces.

(a) Pioneer 3DX [13] (b) TurtleBot3 [11] (c) Khepera IV [8] (d) iRobot Create V2 [7] (e) mBot [9]

Figure 1: Existing differential drive robots for research and education

Beyond traditional robotics, motion-based controllers provide valuable inspiration for intuitive control mechanisms. A particularly relevant example is the Wii Remote [2], introduced in 2006. It introduced motion-based gameplay by tracking orientation and position wirelessly, a concept closely aligned with the

handheld controller in our project. The Wii Remote features a 3-axis accelerometer [2], though it lacks a gyroscope, which limits its ability to differentiate between tilting and linear acceleration. Instead, it estimates position by double integrating accelerometer data, using an infrared reference point (such as the Wii sensor bar or even candles) for external positional tracking. This method, while innovative, has limitations in sustained motion tracking, as accelerometer drift accumulates over time. Adding a gyroscope alongside the accelerometer would provide six degrees of freedom (6DOF), significantly enhancing orientation tracking and gesture recognition—a design choice we plan to incorporate into our system's handheld controller.

(a) Wii Controller [2]

(b) Haptic Smart Cane for the Visually impaired [12]

Figure 2: Additional existing systems of interest

While commercial-grade systems often rely on high-cost distance sensors such as LiDAR, many practical applications favour ultrasonic and infrared-based mapping, which, despite reduced precision, provide a cost-effective alternative for tasks like warehouse navigation and automated stock monitoring. These considerations inform the development of SweepR, a semi-teleoperated differential drive 2-wheeled robot designed to leverage odometry and sonar-based sensing for real-time localisation and general ground-level environmental mapping.

At its core, SweepR leverages wheel odometry and an onboard accelerometer for self-localisation, enabling it to detect sudden external disturbances such as potholes or abrupt pushes. To complement this, a buzzer—which usually outputs retro 8-bit music—also serves as an alert system, emitting a sharp danger signal in response to unexpected movements or collisions. Simultaneously, the robot's trajectory is displayed in real-time on a TFT screen, providing users with a visual representation of its movement.

The initial prototype will operate via a wired controller, with wireless communication via Bluetooth or WiFi planned for early integration. The controller itself will incorporate an accelerometer and gyroscope, allowing intuitive motion-based control—where tilting dictates movement and speed. Additionally, specific gestures, such as vigorous shaking, will trigger the robot's semi-autonomous mode, a feature to be implemented once the core system is fully established.

Inspired by the Pioneer 3-DX's [13] 360° sonar sensing and the design of modern smart haptic canes for visually impaired users [12], SweepR adopts an innovative approach to distance sensing. Rather than employing multiple fixed sensors, the robot features a short, lightweight arm-like structure mounted on top, sweeping continuously in an arc-like motion (with an optimised angle range to be determined). This motion allows a single sonar sensor (potentially paired with an IR sensor for short-range detection) to cover a wider field of view, reducing overall sensor costs while maintaining effective obstacle detection. In semi-autonomous mode, the goal is to enable the robot to navigate towards a wall and then follow its outline to map a room's perimeter—without human intervention. However, manual control remains available at all times, ensuring user oversight. Its modular design further allows for future upgrades, including real-time SLAM, machine learning-based navigation, or AI-driven autonomy.

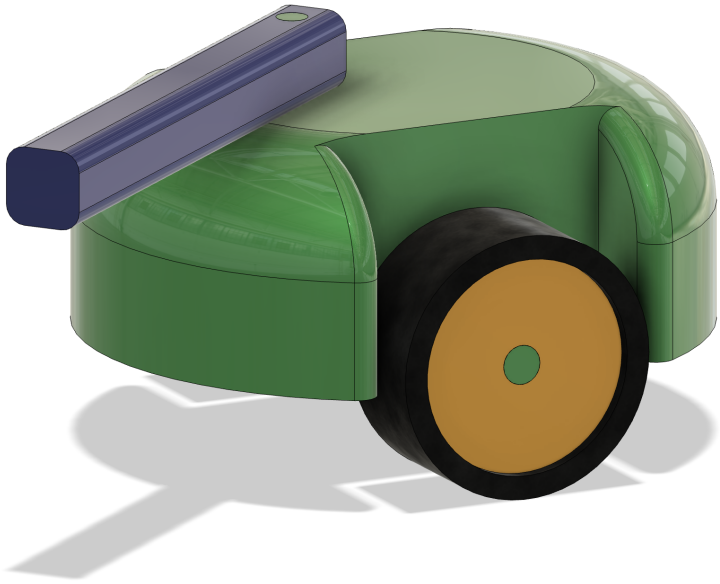


Figure 3: Early Concept Design

This project demonstrates how a cost-effective robotic system can incorporate motion-sensitive control and real-time mapping to perform practical tasks. Instead of relying on conventional button-based controls, gesture-based interactions provide a more dynamic and intuitive user experience, while the robot continuously tracks speed and distances. While it does not rival high-end commercial models, SweepR highlights how an affordable teleoperated platform can address meaningful challenges, from elementary safety inspection to basic industrial automation. Additionally, it offers educational potential, much like M-bots [9] or Cutebots, but with added complexity and functionality, making it a valuable tool for learning about robotics and sensor-driven navigation.

2 Theoretical Background

2.1 System Overview

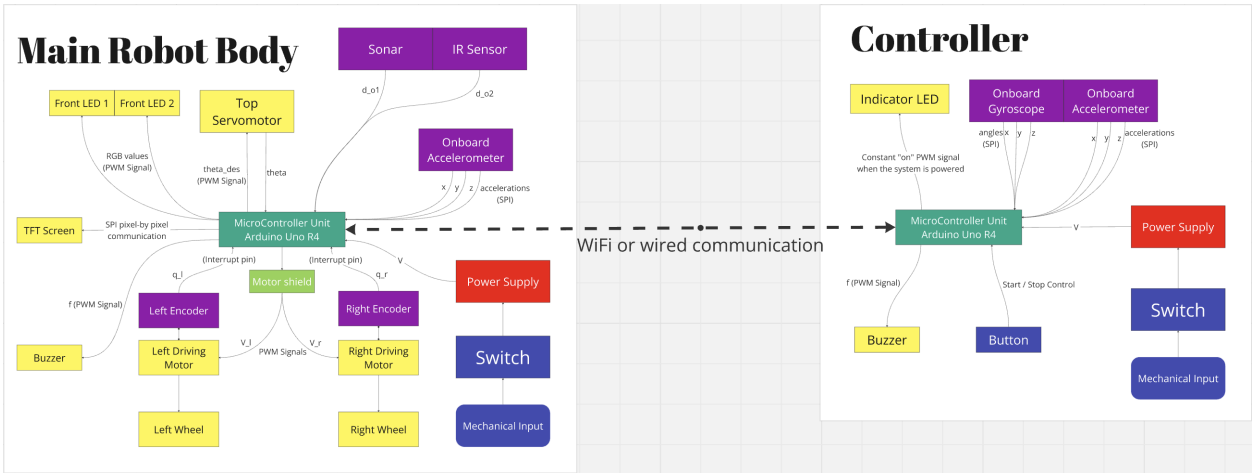


Figure 4: System Overview Diagram

The system establishes a seamless connection between the controller and the robot's body, enabling intuitive movement control and on-demand mapping. Communication is handled via Serial Communication (UART) at first to ensure fast and accurate transmission of commands. This will later be upgraded to wireless Wi-Fi communication, utilising the built-in capabilities of the Arduino Uno R4.

The controller integrates a Pmod ACL2 accelerometer and a Pmod GYRO gyroscope to track motion. A sudden, strong movement detected by the accelerometer triggers a switch between drive & localisation mode and mapping mode, with a KY-006 buzzer providing an audible alert upon transition.

A sensor fusion algorithm combines accelerometer and gyroscope data to enable tilt-based directional control: tilting forward increases speed, tilting backward decreases it, and tilting left or right induces turning while maintaining velocity.

On the robot's body, an ADXL345 accelerometer monitors sudden accelerations to detect collisions or if the robot has flipped over. The system ensures that movement commands are only executed when the robot is in a stable and operational state. If an unsafe condition is detected, a KY-006 buzzer on the body emits an alert.

Similarly, the servo motor controlling the arm will only activate if the arm is in a safe position, preventing motions that could cause mechanical stress or damage.

The onboard sensors scan the environment, detecting obstacles and measuring distances for real-time navigation and mapping. The processed data is displayed on a TFT screen, providing a visual representation of SweepR's localisation status or the mapped environment, depending on the selected mode.

Algorithm 1 Robot Control Loop

```
mode ← "drive"
θ_des ← vector(-60, 60, -60)
i ← 0
while True do
    distance ← get_distance(sonar)
    state ← get_state(accelerometer)
    [angular_v_l, angular_v_r, mode] ← get_control(controller, mode)
    control_motors(angular_v_l, angular_v_r, distance, mode, state)
    handle_servo(θ_des(i, state))
    set_led_colour(state)
    emit_buzzer(track, state)
    update_display()
    delay(Δt)
    if θ_des(i) = θ_des(end) then
        i ← 0
    end if
    i ← i + 1
end while
```

2.2 Distance Measurement Using the Ultrasonic Sensor

The ultrasonic sensor measures distance using the *time-of-flight* principle, where an ultrasonic pulse is emitted and its echo is received after bouncing off an object. The time taken for this round trip allows the calculation of distance.

Reading the Echo Signal

The sensor operates with a trigger and echo mechanism:

- The microcontroller (MCU) sends a HIGH pulse to the **Trigger** pin.

- The sensor emits an ultrasonic burst and waits for the reflection.
- Once the echo is received, the **Echo** pin goes HIGH for a duration proportional to the time taken for the wave to travel to the object and back.
- The MCU measures this HIGH duration (t) in microseconds.

Converting Time to Distance

Given that the speed of sound in air at room temperature is approximately $v = 343$ m/s $= 0.0343$ cm/µs, the distance to the object can be calculated using:

$$d = \frac{t \times 0.0343}{2} \tag{1}$$

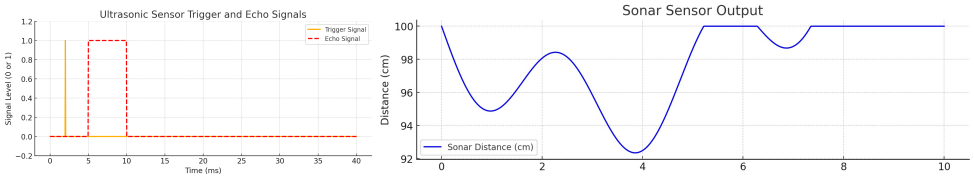
where:

- d is the one-way distance to the object (in cm),
- t is the measured pulse duration (in µs),
- The division by 2 accounts for the round trip of the sound wave.

Example Readings

Echo Pulse Duration (µs)	Distance (cm)
500 µs	8.58 cm
1000 µs	17.15 cm
5000 µs	85.75 cm
10000 µs	171.5 cm

Table 1: Example distance readings from the ultrasonic sensor



(a) Sonar Trigger Echo (b) Sonar Reading Example

Figure 5: Sonar triggering, reading and data conversion to distance

Algorithm 2 Get Distance from Sensors

```
function GET_DISTANCE(sensor)
    send_trigger(sensor)
    wait_echo(sensor)
    d ← compute_distance(sensor)
    return d
end function
```

The MCU continuously reads and converts these values, updating the distance measurement in real time. To improve accuracy.

2.3 Acceleration Processing Using the ADXL345

The ADXL345 accelerometer provides real-time acceleration data along three axes: X, Y, and Z. By analysing these values, we can determine the robot's state and detect anomalous events such as bumps, jumps, crashes, and tilting.

Raw Accelerometer Data

The ADXL345 outputs acceleration values in terms of gravitational acceleration (g) along the three axes:

$$\mathbf{a} = (a_x, a_y, a_z) \tag{2}$$

where:

- a_x, a_y, and a_z are the accelerations in their respective axes.
- The nominal value in a stationary state is $(0, 0, 1g)$ in a gravity-aligned orientation.

The sensor communicates with the MCU via SPI and provides new acceleration samples at a configurable rate. The acceleration threshold values must be carefully tuned to account for additional forces induced by the movement of the robot's arm.

Determining the Robot's State

A binary variable, *state*, is defined to represent normal operation ($state = 1$) or an anomalous condition ($state = 0$). The classification of the robot's state relies on acceleration thresholds:

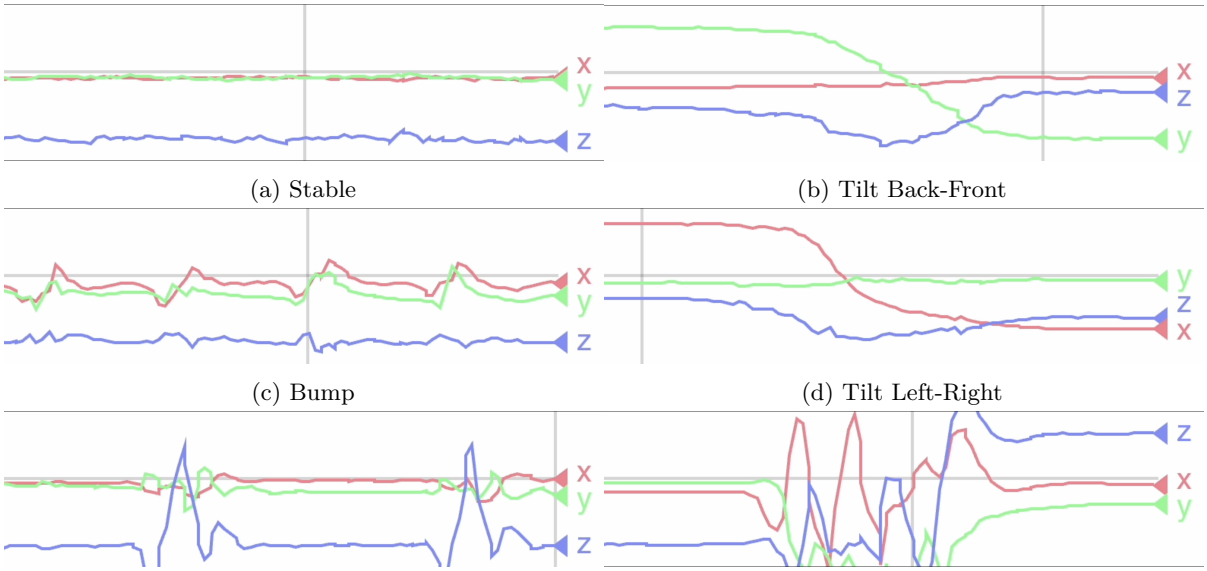
$$state = \begin{cases} 1, & \text{if acceleration remains within normal limits} \\ 0, & \text{if acceleration exceeds predefined thresholds} \end{cases} \tag{3}$$

The normal drive state is characterised by relatively stable acceleration values within expected limits. However, in specific events, abnormal readings are detected:

- **Bumping:** A short small spike in acceleration on one or more axes with visible recoil.
- **Jumping:** A sharp spike where a_z approaches zero or becomes negative due to free-fall conditions.
- **Tilting:** A sustained shift in 2 of the acceleration components relative the third.
- **Landing Upside Down:** A large impact force followed by erratic acceleration patterns before a near-inversion of the a_z component ($\approx -1g$), indicating an overturned state.

Threshold Calibration

To ensure reliable state detection, threshold values for acceleration changes must be adjusted dynamically. The motion of the robot's arm introduces additional accelerations, which can be misinterpreted as anomalies. To mitigate false positives, a calibration step will be required.



(a) Stable (b) Tilt Back-Front

(c) Bump (d) Tilt Left-Right

(e) Jump (f) Upside Down

Figure 6: Example Accelerometer Readings from a Micro:bit device

2.4 Controller

The handheld controller enables the user to command the robot through predetermined gestures. It incorporates an accelerometer, a gyroscope, a buzzer, and a microcontroller.

Reading Measurements

The voltage outputs from the accelerometer and gyroscope are used to determine the controller's acceleration and angular velocity. The relationship for the accelerometer is:

$$a = \frac{V_{out} - V_{offset}}{S},$$

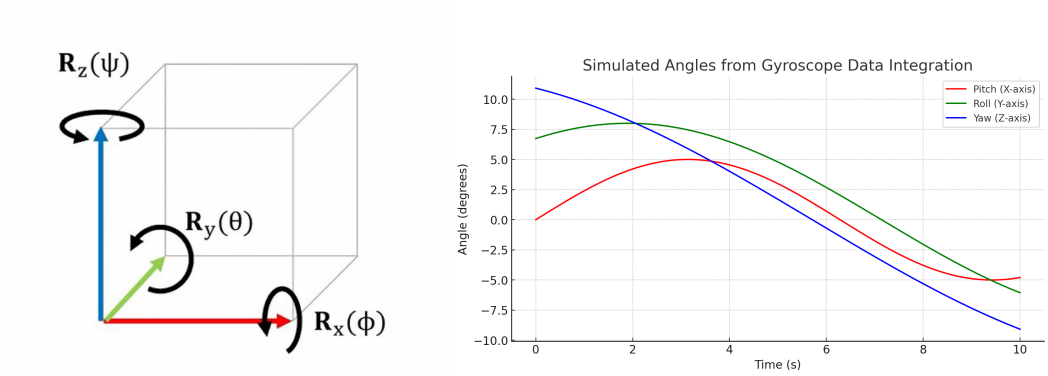
where

$$V_{out} = \text{accelerometer output voltage}, \quad V_{offset} = \text{zero-voltage reading}, \quad S = \text{sensitivity}.$$

Similarly, the angular velocity ω is computed as:

$$\omega = \frac{V_{out} - V_{offset}}{S}.$$

The gyroscope's angular velocity readings are integrated over time to estimate the controller's angular displacement.



(a) Rotation References

(b) Example Gyroscope Reading

Figure 7: Gyroscope Reference Frame and Example

Mode Change

A strong, sudden movement detected by the controller's accelerometer triggers a switch between mapping mode and a movement-only mode. The controller continuously monitors acceleration, and if the measured acceleration exceeds a predetermined threshold (e.g., $\pm 2g$ m/s^2), a mode change is initiated. Note that the constant gravitational acceleration g is always present and accounted for in the measurements.

Movement

Once a mode change has been registered, the robot's movement is controlled by the tilting of the controller. Tilting the controller forward or backward commands forward or reverse acceleration, respectively, while a sideways tilt results in a turn. Although the gyroscope can track these motions independently, its output is often noisy; therefore, sensor fusion of both the gyroscope and accelerometer data is employed to improve the accuracy of tilt detection.

Movement Equations

The accelerometer continuously measures the acceleration due to gravity, which allows us to determine the controller's tilt when at rest. The measured acceleration vector A_p is expressed as:

$$A_p = \begin{bmatrix} A_{px} \\ A_{py} \\ A_{pz} \end{bmatrix} = R \cdot \vec{g},$$

where R is the rotation matrix that fully describes the controller's three-dimensional orientation. Since the rotations about the orthonormal axes are independent, R can be represented as $R(\psi)R(\theta)R(\phi)$ [10]; however, these rotations are non-commutative, so the order is significant. There are six possible orders, with four yielding a full three-angle representation and two yielding only two angles when applied to the initial gravitational vector $\vec{g} = [0, 0, 1]^T$.

For example, one rotation order is:

$$R_{xyz} \cdot \vec{g} = \begin{bmatrix} -\sin\theta \\ \cos\theta \sin\phi \\ \cos\phi \cos\theta \end{bmatrix},$$

and another is:

$$R_{yxz} \cdot \vec{g} = \begin{bmatrix} -\cos\theta \sin\phi \\ \sin\theta \\ \cos\phi \cos\theta \end{bmatrix}.$$

These rotation matrices have a determinant of 1, ensuring they represent pure rotations. By normalising the acceleration vector, the sideways tilt θ and the forward tilt ϕ can be computed as:

$$\theta = -\arcsin \frac{A_{px}}{|A_p|},$$

$$\phi = \arctan \frac{A_{py}}{A_{pz}}.$$

The loss of the third angle ψ is acceptable, as rotation about that axis is not used as a command input. Thus, the accelerometer effectively provides two angular measurements (yielding a 5 DOF capability), and when combined with the gyroscope, these sensors complement each other to reduce noise and enhance the overall user experience.

Algorithm 3 Motion Detection and Control Algorithm

```
function GET_CONTROL(mode)
    A_p ← get_state(accelerometer)
    if |A_p| > threshold then
        if θ > 0 then
            car mode = "drive"
        else
            car mode = "mapping"
        end if
        Buzzer sound
    else
        θ_a, φ_a ← accelerometer_equations(A_p)
    end if
    θ̇_g, φ̇_g = gyroscope values
    Store angles θ, φ in a list
    θ_g & φ_g ← integration(angles list)
    if |θ_a − θ_g| and |φ_a − φ_g| < threshold then
        if θ > threshold then
            if θ > 0 then
                return Motor voltages to turn right, mode
            else
                return Motor voltages to turn left, mode
            end if
        end if
        if φ > threshold then
            if φ > 0 then
                return Motor voltages to drive forward, mode
            else
                return Motor voltages to drive backwards, mode
            end if
        end if
    else
        return 0 voltage
    end if
end function
```

2.5 Motor Control

Driving Motor Control

We use the angular velocities computed from the controller data for each wheel, the distance to an obstacle, the mode ("drive" or "map") and the current functional or abnormal state (binary value) to determine what

instructions should be given to the motors and which voltage to apply.

Algorithm 4 Motor Control Based on Sensor Inputs

```
function CONTROL_MOTORS(angular_v_l, angular_v_r, distance, mode, state)
    if distance < 15cm then                                          ▷ Obstacle detected
        if mode = "drive" then
            restrict_motion(turning_or_reverse)                      ▷ Prevent forward movement
        else
            follow_obstacle()                                        ▷ Engage wall-following behaviour
            adjust_arm_angle(follow)                                 ▷ Modify servo position accordingly
        end if
    else if state = 0 then                                           ▷ Stop motors if inactive
        set_motor_pwm(left_motor, 0)
        set_motor_pwm(right_motor, 0)
    else                                  ▷ Convert speed inputs to voltage and drive motors
        V_l ← map_speed_to_voltage(left_speed)
        V_r ← map_speed_to_voltage(right_speed)
        set_motor_pwm(left_motor, V_l)
        set_motor_pwm(right_motor, V_r)
    end if
end function
```

Servo Motor Control

Position the arm controlled by the servo motor to the next desired angle from a vector spanning back and forth between $-60°$ and $60°$ (or a better angle value to be determined from testing).

Algorithm 5 Handle Servo Motor

```
function HANDLE_SERVO(theta_des)
    set_pwm(servo, theta_des)
end function
```

2.6 Localisation and Mapping – Odometry Equations

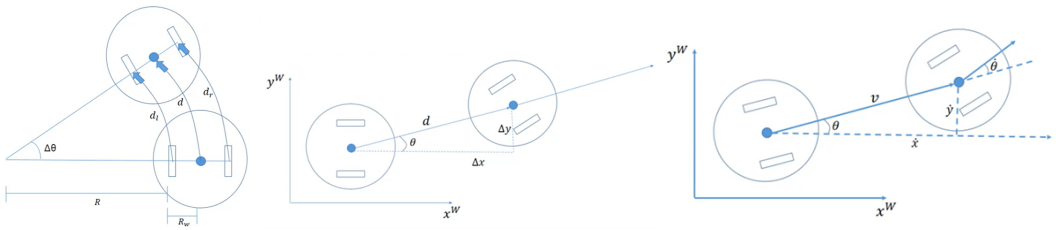
Overview of Wheeled Robot Control

Wheeled robots, such as self-driving cars and mobile robotic platforms, rely on wheel-based locomotion. Among various configurations, differential drive robots are particularly common due to their simplicity and ease of control. In a differential drive robot, two independently driven wheels enable precise control over both speed and direction [3].

How a Differential Drive Robot Moves

The motion of a differential drive robot can be described as follows:

- Both wheels moving at the same speed and in the same direction → Straight motion (forward/backward).
- Wheels moving at equal speeds but in opposite directions → Rotation in place.
- One wheel moving faster than the other → Curved motion.



(a) Arc Approximation (b) Finding Position (c) Finding Velocity

Figure 8: Odometry Parameter Visualisation

Using Wheel Encoders for Tracking

Wheel encoders measure the number of rotations each wheel undergoes, providing an estimate of the distance travelled during each short period of t. Since t approaches 0, we can effectively consider displacement as a series of small arcs. The distance D a wheel covers is therefore given by:

$$D_{r/l} = \frac{\left(S_t - S_{t-1}\right) C}{Q},$$

where

$$D_{r/l} = \text{distance travelled by respective wheel,}$$

$C =$ wheel circumference, $\quad Q =$ total encoder ticks per full rotation, $\quad S =$ encoder tick counts for that wheel.

Calculating the Robot's Position and Orientation

The robot's position is computed at the midpoint between its wheels, with its trajectory approximated by a circular arc. The change in orientation is determined by:

$$\Delta\theta = \frac{D_r - D_l}{2R_w},$$

and the average distance travelled is:

$$D = \frac{D_r + D_l}{2}.$$

The positional changes in the X and Y coordinates are then calculated using:

$$\Delta X = D\cos(\theta), \quad \Delta Y = D\sin(\theta),$$

where

$$\Delta X = \text{change in } X, \quad \Delta Y = \text{change in } Y, \quad \theta = \text{orientation angle of the body.}$$

Estimating Velocity

The forward velocity of the robot is estimated as the average of the left and right wheel velocities:

$$v_r = \frac{D_r}{\Delta t}, v_l = \frac{D_l}{\Delta t},$$

$$v = \frac{v_r + v_l}{2},$$

and the angular velocity is given by:

$$\omega = \frac{r\left(v_r - v_l\right)}{2R_w},$$

where

$$v_l = \text{velocity of the left wheel,} \quad v_r = \text{velocity of the right wheel,}$$

$$v = \text{velocity of the body's centre,} \quad \omega = \text{angular velocity of the body,}$$

$$r = \text{wheel radius,} \quad R_w = \text{radius from centre to wheels.}$$

In this configuration, if the right wheel moves faster, the robot turns left, and vice versa.

Summary

1. Read encoder values from both wheels.
2. Calculate the distance each wheel has traveled.
3. Determine the change in orientation ($\Delta\theta$).
4. Update the robot's position in the X and Y directions using trigonometric relationships.
5. Compute velocity based on the displacement over time.

In addition to wheel odometry, accelerometer data is processed to estimate position and speed. Although accelerometer-based measurements are less precise, they provide an independent check to ensure the accuracy of the wheel encoders.

Mapping Overview

In addition to relative localisation derived from odometry and accelerometer data, we can integrate sonar sensor readings to construct an environmental map of the space in which SweepR operates. The sonar provides distance measurements to nearby obstacles, but these measurements must be associated with the robot's global position and orientation to build an accurate spatial representation.

The sonar sensor is mounted on an actuated arm, which rotates at an angle θ relative to the robot's main body. The absolute position of an obstacle detected by the sonar can be determined by transforming the measured range into the global coordinate frame.

Given:

- The robot's global position (X_r, Y_r).
- The robot's orientation θ_r in the global frame.
- The sonar arm's angular position θ_s relative to the robot.
- The measured distance d_s from the sonar sensor to the detected obstacle.

The global coordinates of the detected obstacle (X_o, Y_o) are given by:

$$X_o = X_r + d_s \cos(\theta_r + \theta_s),$$
$$Y_o = Y_r + d_s \sin(\theta_r + \theta_s).$$

This transformation accounts for both the robot's current orientation and the sonar's relative angle, ensuring that distance measurements are correctly placed within a global reference frame.

To construct an environmental map, multiple sonar readings are taken over time at small intervals as the robot moves. By accumulating these readings, we can generate a point cloud or vectorial representation of the outline of the room or obstacles in the robot's vicinity. Handling the data representation of the map in an efficient manner, optimised of the MCU will however be challenging, which is why we consider mapping as an additional feature if this project's time frame permits it.

2.7 Human Interaction Elements

The robot's body will feature two front-facing RGB LEDs that change colour based on its operational state. Initially, a simple implementation will indicate a normal operating state with green lights and an abnormal state with red lights. However, if additional states are introduced—such as detecting an obstacle but remaining operational—the RGB capabilities of these LEDs will allow for a more nuanced visual representation of the robot's status.

Algorithm 6 Set LED Colour Based on the Robot's State

```
function SET_LED_COLOUR(state)
    (R, G, B) ← state_to_colour(state)
    send_pwm(led1, R, G, B)
    send_pwm(led2, R, G, B)
end function
```

LEDs

As mentioned in the introduction, SweepR's buzzer will play retro music tracks while patrolling to provide a more engaging interaction. However, when an abnormal state is detected, the buzzer will serve as an alert system, producing a series of high-pitched warning sounds to notify the user of an issue.

Buzzer

Algorithm 7 Emit Buzzer Sound Based on State

```
function EMIT_BUZZER(track, state)
    if state = 1 then                          ▷ Normal operation: Play a track
        play_sound(track)
    else if state = 0 then                     ▷ Alert: Modify pitch based on sensors
        adjust_pitch(state)
    else
        stop_buzzer()
    end if
end function
```

The TFT display will initially visualise the robot's trajectory, updating its position at regular intervals t. If the mapping function is implemented, the display could also be used to render a real-time environmental map.

TFT Display

Algorithm 8 Update Robot Trajectory on TFT Display

```
function UPDATE_DISPLAY
    (x, y) ← read_position(encoders, x, y)
    plot_trajectory(x, y)
    refresh_display()
end function
```

2.8 Mechanics of the System

Arm Movement – Motor Equations

The servomotor-driven rotating bar positions the ultrasonic (and possibly infrared) sensors used for obstacle detection, avoidance, and mapping. The motor's performance is characterized by the relationship between torque, speed, current, and voltage.

The torque generated by the motor is directly proportional to the armature current:

$$\tau_m = K_m \, I_a \tag{4}$$

where

τ_m is the torque, K_m is the motor's torque constant, I_a is the armature current.

Thus, increasing the current enhances the torque, which in turn helps the motor overcome the inertia of the rotating bar and initiate movement.

As the motor rotates, it also produces a back ElectroMotive Force (EMF) that opposes the applied voltage:

$$V_m = K_b \, \omega_m \tag{5}$$

where

V_m is the back EMF, K_b is the back EMF constant, ω_m is the motor's angular velocity.

With an increase in speed, the back EMF grows, naturally limiting the maximum rotational speed. This interplay between torque, current, speed, and voltage ensures a smooth and controlled motion of the sensor bar.

Body Movement – Equations of Motion

To track the movement of the robot's body, data from both the motor encoders and the ADXL accelerometer are employed along with the standard kinematic equations:

$$v = u + at,$$

$$s = ut + \tfrac{1}{2}at^2,$$

$$v^2 = u^2 + 2as,$$

where

s is displacement, u is the initial velocity, v is the final velocity, a is the acceleration measured by the ADXL345, t is

By continuously processing the acceleration data, the system estimates velocity and position. Although wheel odometry provides the primary measurement for position and speed, the accelerometer-based estimates serve as an important cross-check to ensure that the odometry readings are reliable and the overall system is functioning correctly.

3 Electronics Composition

3.1 Components list

The system consists of the following electronic components and modules:

- **Microcontroller:**
  - 1st Arduino Uno R4 (Controller)
  - 2nd Arduino Uno R4 (Body)
- **Sensors:**
  - Pmod ACL2: 3-axis MEMS Accelerometer (Controller) - 3.3V, 12 bits resolution
  - Pmod GYRO: 3-axis Digital Gyroscope (Controller) - 3.3V, 250/500/2000 dps resolutions exist, we will go with 250dps as our tilts will be slow.
  - ADXL345 Accelerometer (Body) - 3.3V, 10 bits resolution
  - HC-SR04 Ultrasonic Sensor (Body) - 5V, Range: 2cm to 400cm
  - (Infrared Sensor (Body))
- **Actuators:**
  - 2 DC Motors (Body) with Encoders and Motor Shield - specifications TBD

  - 1 Servomotor to rotate the arm (Body) - specification TBD
- **Indicators:**
  - 1 Green or Red LED (Controller)
  - 2 KY-006 Buzzer (Controller & Body) - Tone generation range: 1.5 to 2.5kHz
  - 2 RGB LEDs (Body)
- **Display:**
  - Adafruit 2478, TFT LCD Display 2.4in (Body) - 5 pins, 3-5V
- **Mechanical Input:**
  - 1 Button (Controller)
  - 2 Switches (Controller & Body)
- **Power and Wiring:**
  - 2 Power Supplies (Controller & Body)
  - 2 Perforated Prototyping Boards (Controller & Body)
  - Necessary Wiring for sensors, motors, and communication (Controller & Body)
- **Mechanical elements:**
  - 2 Motor-driven Wheels (Body)
  - 1 Front Caster Wheel(Body)
  - Laser-cut Chassis (Body)

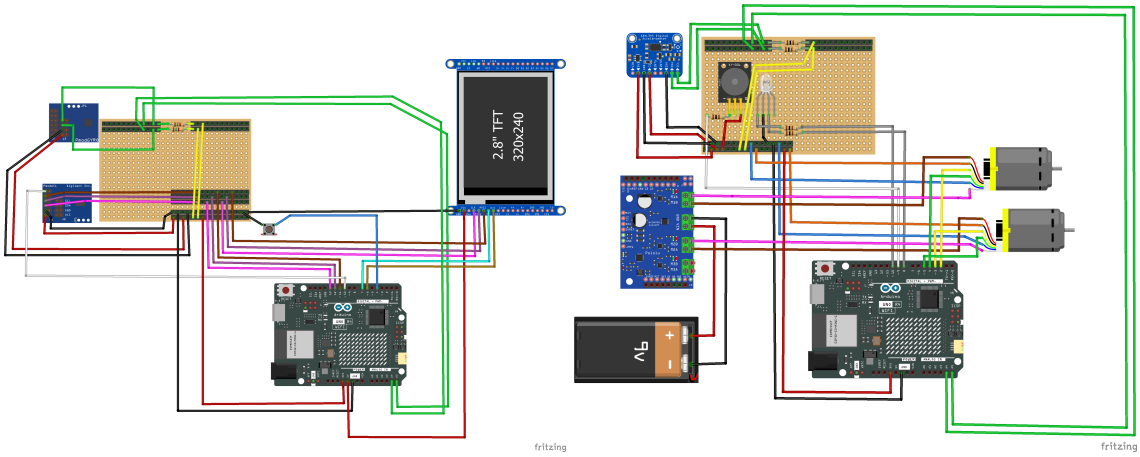
3.2 Interfacing & Communication Protocols

The various components of the SweepR system interface with the microcontroller using a combination of communication protocols, as outlined below:

- **Accelerometers and Gyroscope:** The ADXL345 accelerometer as well as the Pmod gyroscope module will communicate with the MCU via I2C protocol. The Pmod accelerometer however will need to be wired using the Serial Peripheral Interface (SPI).
- **Ultrasonic Sensor:** Uses digital I/O pins for its trigger and echo signals. This sensor does not employ a bus protocol such as I2C or SPI; instead, it relies on simple digital signaling.
- **DC Motors and Motor Controllers:** Operated using PWM (Pulse-Width Modulation) signals for precise speed control, with additional digital outputs for managing direction.
- **Wheel Encoders:** Provide incremental rotation information through digital pulses, which are captured by the MCU via interrupt-driven pins rather than over a communication bus.
- **Servo Motor:** Controlled by PWM signals generated by the MCU. The PWM signal sets the desired angular position directly.
- **LED Indicators:** Driven by PWM outputs for variable brightness and colour control.
- **Buzzer:** Driven by PWM signals that set the frequency for sound generation, enabling auditory alerts and feedback based on system events.
- **TFT Display:** Communicates via SPI, which enables high-speed data transfer necessary for updating graphical information on the display.

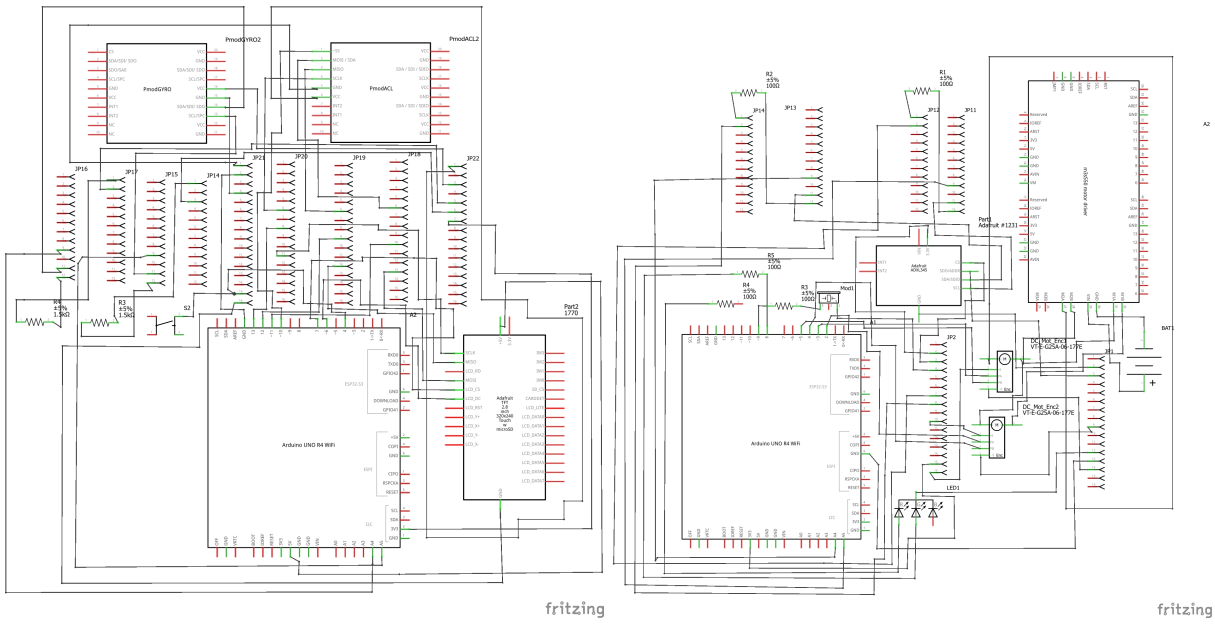
4 Electrical and Electronics Design

4.1 Electrical Diagrams and Schematics



(a) Controller (b) Main Body

Figure 9: Visual renditions of the electrical schematics using Fritzing



(a) Controller (b) Main Body

Figure 10: Electrical Schematics from Fritzing

4.2 Datasheet Analysis

The use of datasheets greatly aided us in getting the components to work as intended. When working with the TFT screen, the datasheet [1] detailed how to connect all the wires, enabling us to efficiently set up the display. In addition, it provided a library that allowed us to program the screen's logic in an intuitive manner.

For the Pmod Gyroscope and Accelerometer, the datasheets [4] [5] again facilitated a quick and straightforward setup, allowing us to interface the Arduino with the components effectively. The example code provided

by the manufacturers proved to be robust and precisely what we needed. Helpfully, this code included the necessary coefficients to convert the raw values into angular velocity from the gyroscope and acceleration from the accelerometer.

The ADXL345 datasheet [14] also offered an example wiring layout that suited our project well. The only issue arose from the datasheet suggesting the use of the Arduino's 5V output, which the accelerometer is rated for. However, we found that it frequently overheated under this configuration. Since the datasheet stated the component could operate down to 3V, we switched to using the 3.3V output instead. The example code included with the ADXL345 also contained angle calculation functions, which, upon inspection, used the same mathematics as derived in Section 2.4. Therefore, we retained the code as it functioned effectively.

Conversely, the lack of a suitable datasheet [15] for the motors proved to be a challenge. One example we found online featured a different wire colour scheme from ours, which led us to mistakenly plug the encoder wires into the motor power terminals on the motor shield—costing us valuable lab time while troubleshooting why the motors wouldn't rotate.

To implement WiFi communication between the two Arduino R4 boards, we referred to the official Arduino documentation [6]. Adapting the example code provided there enabled us to achieve the desired wireless functionality.

4.3 Communication and Inter-Connectivity

The primary interconnectivity architecture remained unchanged from what was outlined in Section 3.3. However, we encountered critical issues with the I2C communication protocol when integrating the ADXL345 accelerometer and the Pmod gyroscope. The data returned was either incorrect or not received at all. This was resolved by including 1.5 kΩ pull-up resistors on each I2C line between the Arduino and the sensors (note that for 3.3 V logic, 1.5 kΩ was selected; for 5 V logic, 4.7 kΩ would have been used).

For motor encoder readings, interrupt pins were assigned using the `attachInterrupt` function to accurately detect state changes from the encoders' A channel. The B channel which detects the direction in which the motor is spinning was not used as we lacked sufficient interrupt pins. Instead, we determined direction from the value passed via PWM signal to the motor from the MCU. Regarding SPI communication, both the Pmod accelerometer and the TFT screen required dedicated SPI lines (MISO, MOSI, SCK, and CS). By designating unique Chip Select (CS) pins for each device and remapping input pins on the Arduino, we enabled both devices to operate concurrently on the SPI bus.

Motor control was managed using the Pololu Motor Shield. The Motoron library for Arduino simplified this process by abstracting the low-level motor logic. Motor speed was set programmatically, with a value of 3000 corresponding to maximum speed. Negative values reversed the motor direction as previously stated.

To enable full WiFi communication between the controller and the body unit, both Arduino R4 boards were connected to a local WiFi network (SSID: `eduino`) using `WiFi.begin(ssid, password)`. IP addresses were dynamically assigned and exchanged between devices. Data transmission was handled via UDP on predefined ports (e.g., 2000), with each device listening on the corresponding port to receive messages. Communication followed a structured message format: `"v1=motor_speed_left, v2=motor_speed_right, mode=mode"` and `"x=x, y=y"`. To avoid board rate issues, both boards had to have the same rate (115200) to ensure both receive and send messages at the same rate.

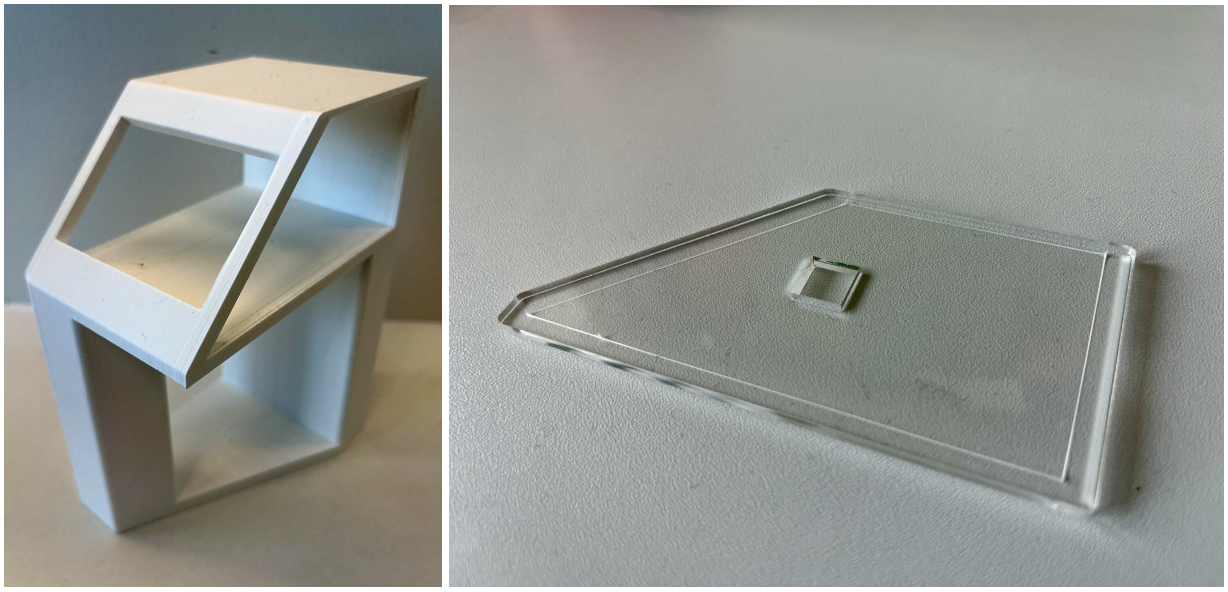
For the LED and buzzer components, 100 Ω resistors were added in series to regulate current flow and ensure desired brightness and sound levels.

4.4 Mechanical Design

Controller Design

A minimalist yet ergonomic design approach was adopted for the controller. A custom 3D-printed structure (Fig. 11.a) was created to ensure a comfortable grip and to facilitate tilt-based control. The design accommodates a range of hand sizes with ease. The upper portion of the controller features a dedicated

housing for the electronics, including a cutout for the screen. The middle section holds the microcontroller unit (MCU) and the majority of the internal wiring. At the rear, a foam parallelepiped with slits was inserted to securely position both the Pmod accelerometer and the gyroscope while keeping them parallel for accurate orientation tracking. Finally, side walls (Fig. 11.b) were laser-cut from 5 mm acrylic sheets to complete the housing. These walls include openings for the power cable and a push-button interface, ensuring neat cable management and accessibility.



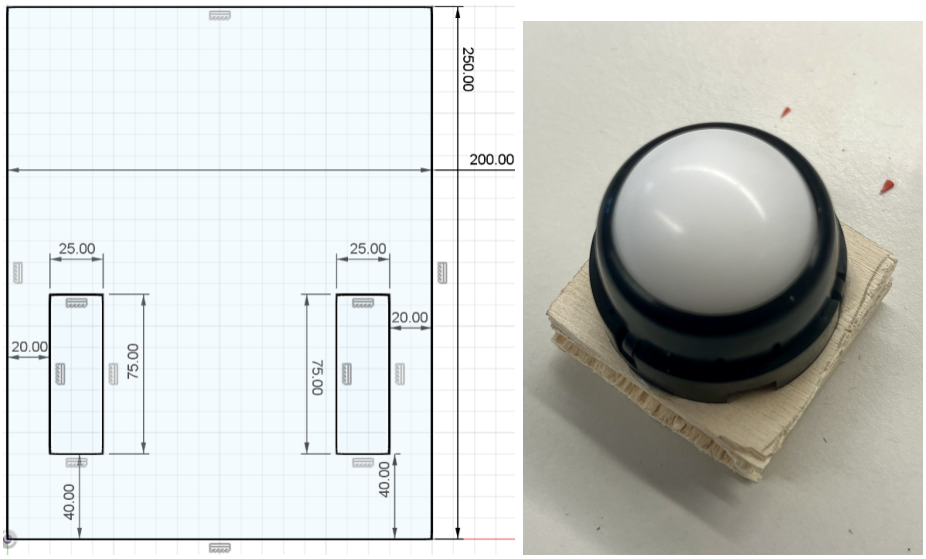
(a) 3D-Printed Controller Housing

(b) Laser-cut Controller Housing Side-walls

Figure 11: Controller Housing

Main Body

As a result of the limited time we had on offer crossed with sub-optimal time management skills, the mechanical system of the main body remained intentionally simple. A single laser-cut plywood plank of 5mm formed the primary support platform, as illustrated in Fig.12. Motors were mounted using zip ties and a caster ball, whose base was modified to achieve the desired height and meet the platform perpendicularly.



(a) CAD Sketch of the main support board (b) Caster Ball

Figure 12: Mechanical Elements for the Main Robot Body

Electronic components were fixed to the platform using a combination of tape and Blu Tack. The perforated board was stuck into a second foam parallelepiped (taking advantage of the excess length of soldered components), which was then secured onto the main support, helping to keep the circuit organised despite the strategically improvised layout.

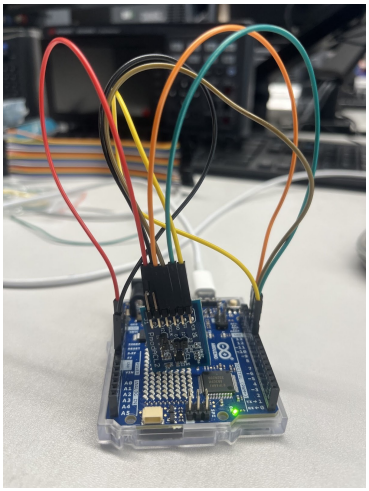
5 Experimental Evaluation

5.1 Electrical Assembly

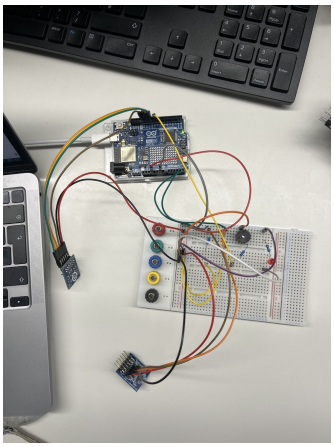
The assembly process began with soldering header pins onto the ADXL345 and the TFT display to enable reliable wire connections. Prior to permanent soldering, we developed and validated the circuit layout on a breadboard, incrementally adding one component at a time. Each addition was thoroughly tested using component-specific code (often provided by the manufacturer) before progressing, as illustrated in Fig. 13.

After successful validation, we transitioned to soldering the circuits onto perforated boards. External modules (e.g., accelerometers, TFT) were not directly soldered to facilitate reusability and prevent damage during revisions. Plug-and-play header connections were used instead. Separate boards were designed for the controller and the body system to ensure functional independence.

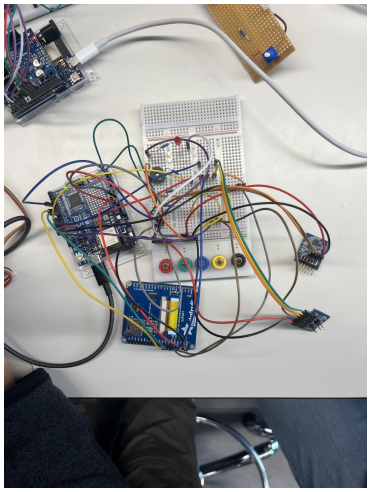
We selected perforated boards with pre-laid copper strips on the reverse side (Fig. 14b) to simplify connections and reduce the number of solder joints. Power rails (GND, 3.3 V, 5 V) were consolidated across rows to improve consistency and ease of debugging. Figures 14a through 15 depict the final soldered assemblies.



(a) Accelerometer only

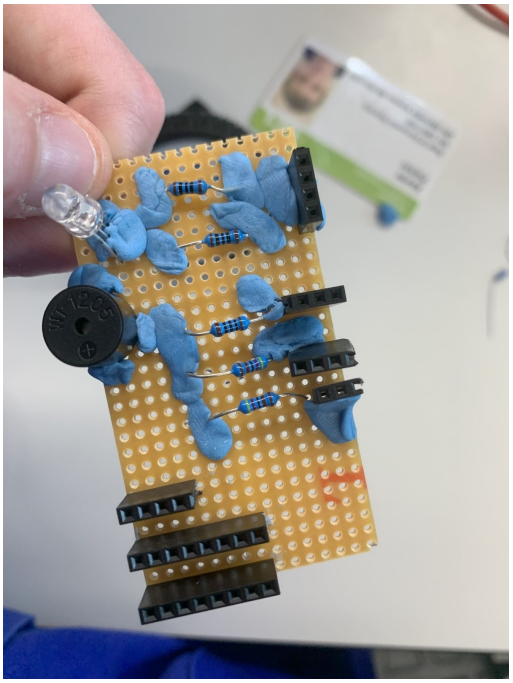


(b) Addition of gyroscope, buzzer and LED

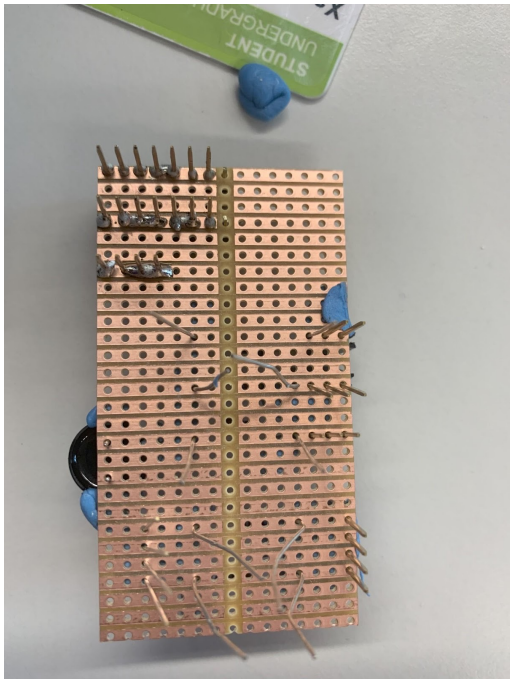


(c) Complete setup with screen

Figure 13: Breadboard prototyping



(a) Component layout on body's perforated board. Blu tack is temporarily used prior to the final soldering.



(b) Underside of the board shown in Figure 14a. The pre-laid copper tracks minimise inter-component wiring but require careful layout to prevent short circuits.

Figure 14: Component arrangement on the perforated board, with top and underside views

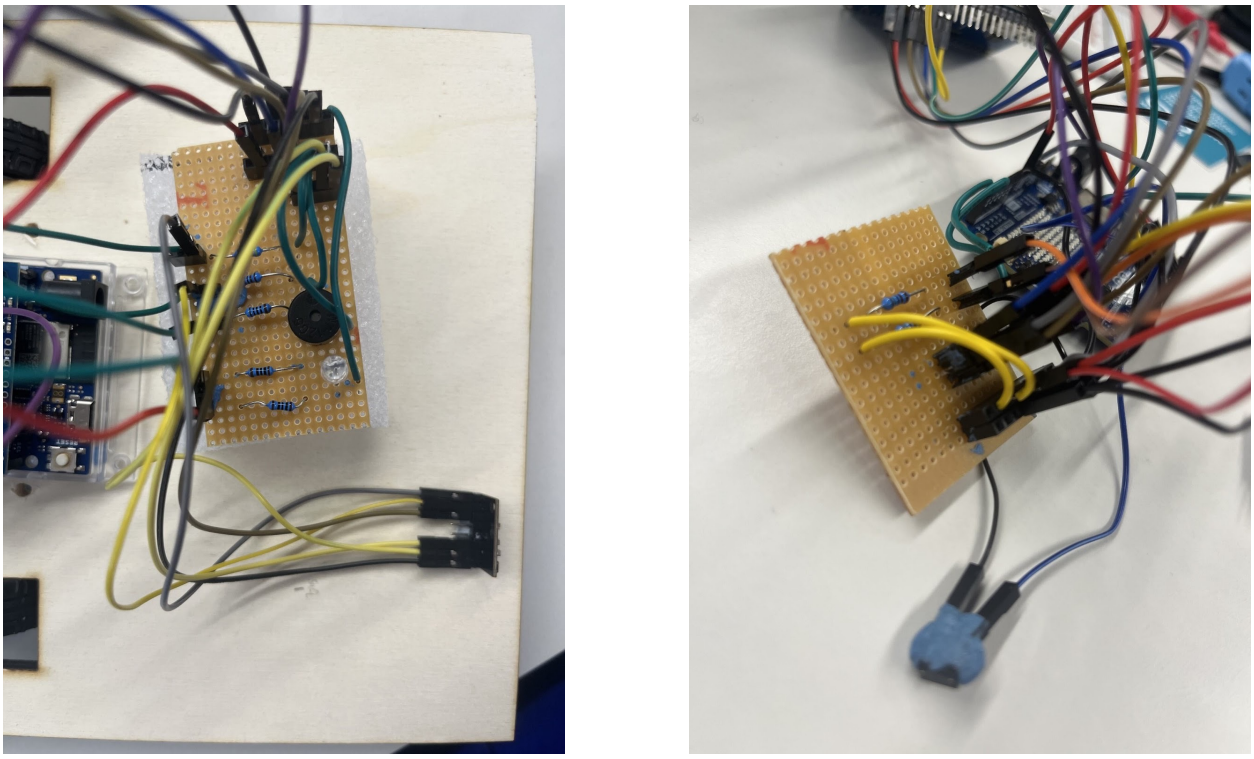


Figure 15: Final soldered circuits for the controller and body. Each has dedicated header connections for GND, 5 V, and 3.3 V leading to their respective Arduino boards

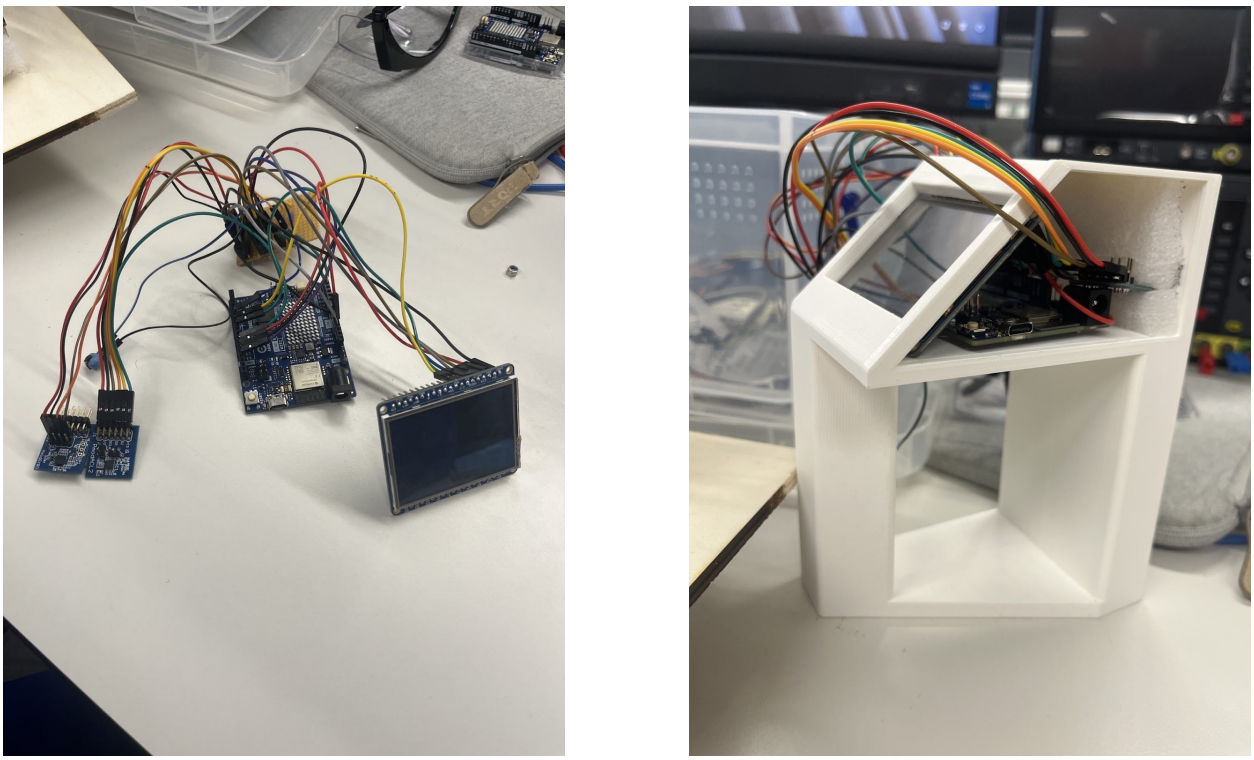


Figure 16: Finalised controller board installed into the 3D-printed housing

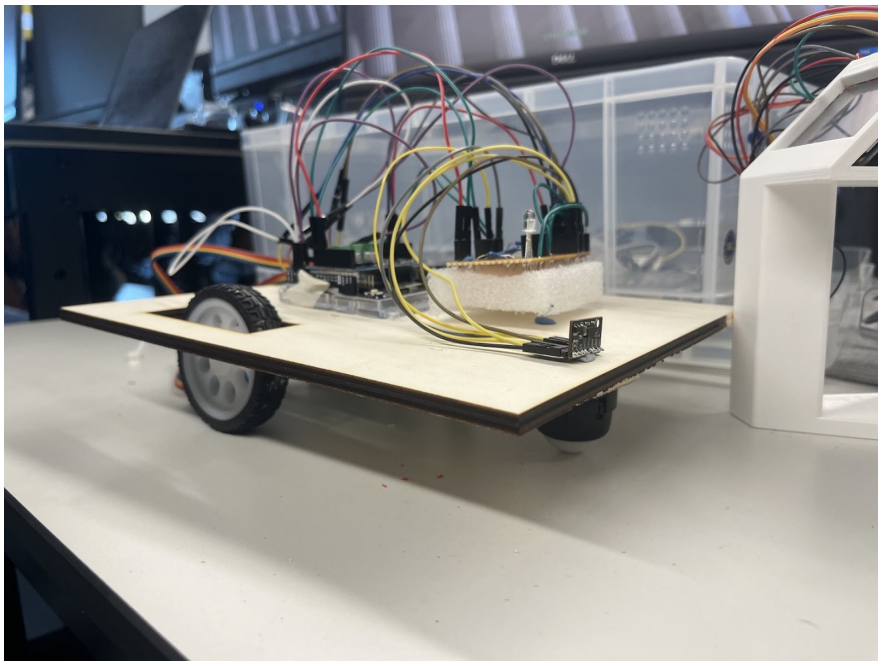


Figure 17: Fully assembled body module including wiring and mounted board

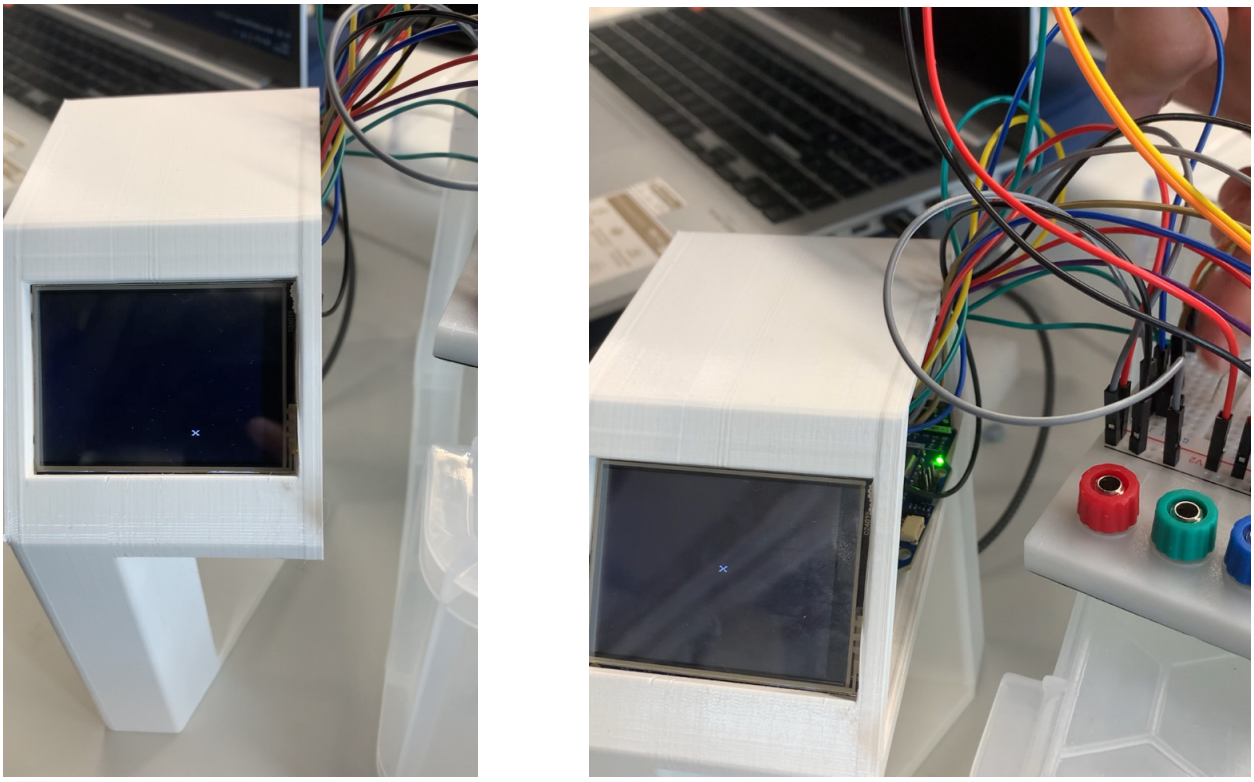
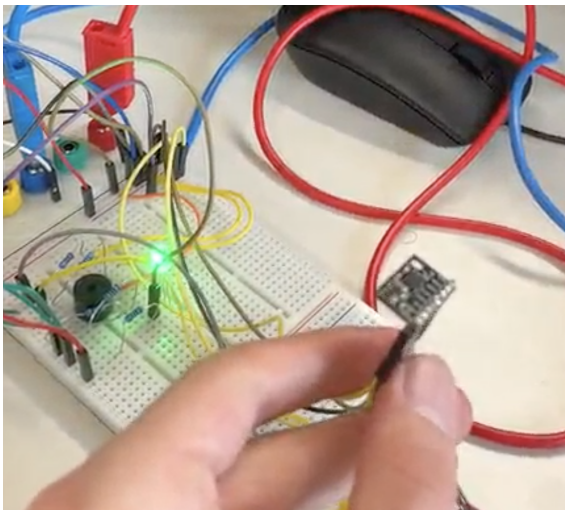
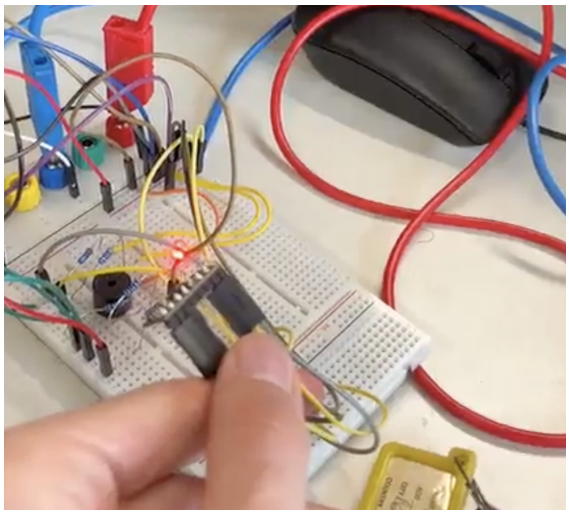


Figure 18: Display showing real-time position from odometry with an "x" marker. Pressing the button re-centres the position display.



(a) LED green when upright



(b) LED red when tilted

Figure 19: Crash detection visualised. Buzzer sounds provide audible feedback.

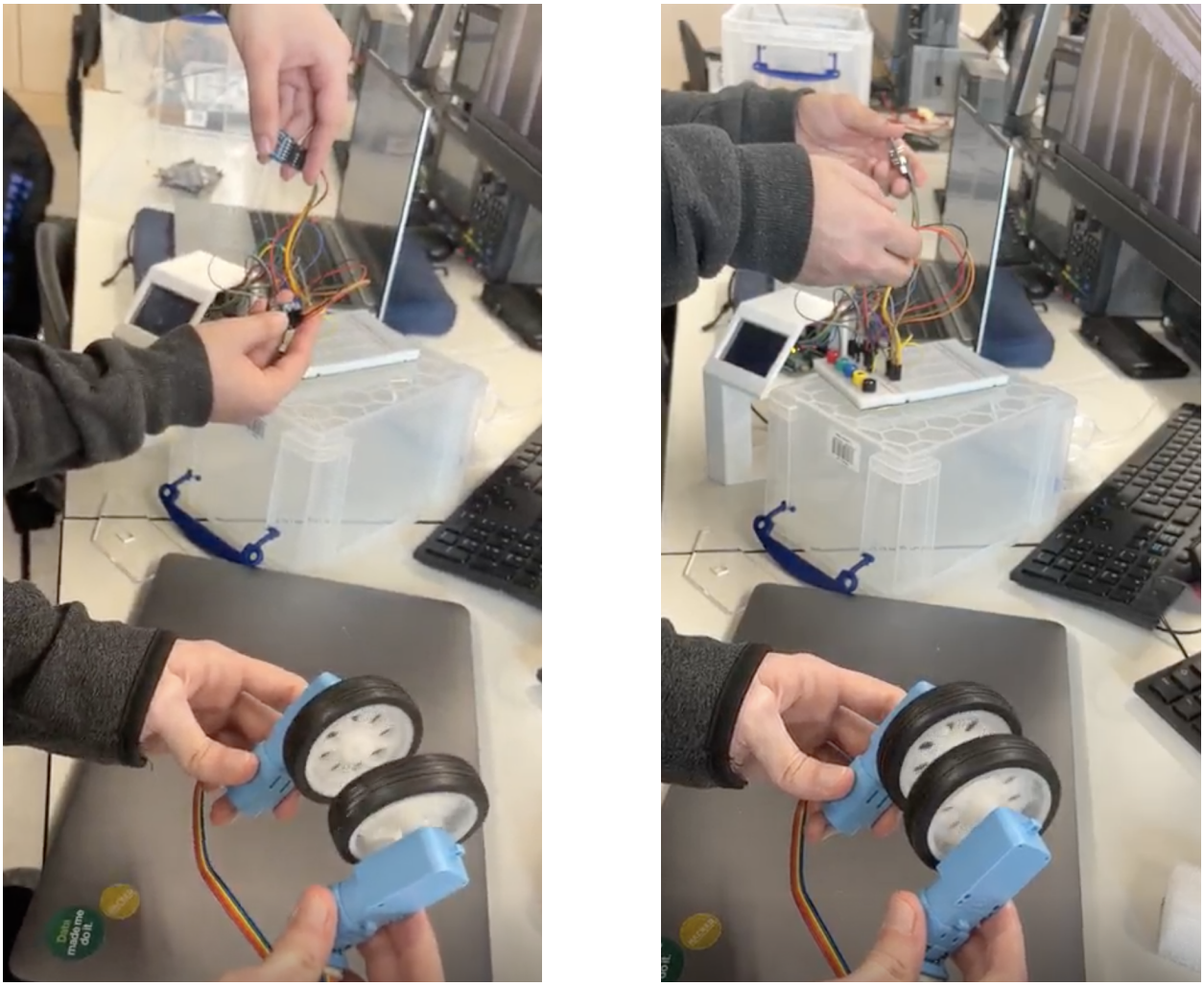


Figure 20: Tilt recognition in action: Tilting the controller sideways by more than 30° causes one motor to accelerate (left), enabling the car to turn. Tilting it forward causes both motors to rotate at the same speed (right), resulting in straight-line motion.

6 Discussion and Conclusion

6.1 Accomplishments

Although the project was not fully completed by the imposed deadline, it nonetheless yielded several positive outcomes. Our primary objective was to design a small mobile robot, wirelessly connected to a controller via WiFi, capable of using odometry for localisation tasks within a relative coordinate space. This core functionality was achieved successfully.

All critical components intended for integration were used effectively. Interestingly, not using every single component initially planned allowed us to concentrate on maximising the potential of those we did include. The buzzer, for example, successfully played six distinct tracks during standard operating conditions: the Star Wars main theme, He's a Pirate (Pirates of the Caribbean), Hedwig's Theme (Harry Potter), Korobeiniki (Tetris theme), the Pac-Man theme, and the Super Mario Bros theme. Moreover, a 2000 Hz tone was employed as an auditory indicator of abnormal behaviour—highly noticeable and effective in crowded environments.

The buzzer was paired with an RGB LED to enable visual state signalling: green for normal operation, red for anomalies — thus making status information accessible to deaf users as well. The ADXL345 accelerometer functioned as intended, detecting tilts over 30° and excessive acceleration in any direction. On the controller side, sensor fusion between the gyroscope and accelerometer enabled smooth differential-drive motor control. The TFT screen displayed the robot's real-time position within relative space, while the integrated button successfully reset this position to the screen's centre.

6.2 Failures

Despite these accomplishments, the project ultimately did not meet all initial specifications outlined in Sections 1 to 3, and therefore cannot be considered fully successful. However, these shortcomings were primarily due to sub-optimal time management and an overly ambitious scope within a tight manufacturing window, especially considering a simple mechanical system had to be manufactured in addition to the electronics part.

Key areas for improvement include:

1. **Time Management:** Allocating an additional three-hour session for mechanical assembly could have been decisive in achieving full completion.
2. **Scope Adjustment:** Focusing solely on building a mobile robot with reliable wireless control, while enhancing code functionality using available components, would have been more realistic.
3. **Planning:** We underestimated the GPIO demands of the system architecture outlined in Fig. 4, making it infeasible to implement using only one Arduino R4. Consequently, elements such as the TFT screen had to be reassigned to the controller, where it was arguably used to better effect.
4. **Hardware Debugging:** We relied too heavily on manufacturer datasheets and code intended for Arduino R3 boards, overlooking key compatibility differences with the R4. Notably, we encountered persistent I2C communication issues with our devices—costing us over 10 days (from Monday 3$^{\text{rd}}$ to Thursday 13$^{\text{th}}$ March), until the man himself, Alex Charitonidis, stepped in and helped us resolve the problem by implementing properly adapted pull-up resistors.

6.3 Future Work & Possible Improvements

Although the dismantling of the system limits direct continuation, several improvements and follow-up directions can be identified:

Firstly, a natural next step would be to realise the complete SweepR system outlined in Sections 1–3. However, while originally intended as an affordable mobile robot for simple patrol and educational use, the current prototype revealed cost challenges. Scaling up—especially with additional sensors—would require either a secondary Arduino board or a higher-end MCU like the STM32 to accommodate GPIO demands. Thus, the current system could serve as a testbed for designing a more cost-effective and scalable robot.

Secondly, the absence of distance sensors led us to abandon the mapping mode initially considered. Inspired by the iRobot Create V2 (Fig. 1.d, Ref. [3]), one possible alternative would have been to also use the ADXL345 accelerometer as a bumper sensor. With proper mechanical adaptations for absorbing light impacts, this method could enable basic mapping functionality without relying on additional costly ultrasonic or infrared distance sensors. This idea remains viable for simulation in the Arduino IDE and could be implemented as a proof-of-concept without requiring a physical build.

Thirdly, removing the controller entirely and shifting to autonomous robot operation could substantially reduce production costs. Eliminating the 3D-printed controller, the Pmod ACL2 and GYRO, the TFT screen, and a second Arduino R4 board would streamline both cost and complexity.

In summary, while the project fell short of its original ambitions, it nonetheless demonstrated key subsystems and design strategies that could be refined and expanded. By building on these achievements and addressing the identified limitations, this project still holds potential as a stepping stone towards the development of more accessible, low-cost robotic platforms for low-risk patrol tasks and educational environments.

A Program Code

A.1 Controller Arduino Code

```
#include "ADXL362.h"
#include <SPI.h>
#include <Wire.h>
#include <L3G.h>
#include <math.h>
#include <WiFiS3.h>
#include <WiFiUdp.h>
#include <TFT_eSPI.h>  // Hardware-specific library
#include <SPI.h>
#include <math.h>

// Define custom SPI pins for Arduino R4 (ESP32-S3)
#define SPI_MISO 13
#define SPI_MOSI 11
#define SPI_SCK 12
#define SPI_CS 10  // Chip Select (CS), change if needed

TFT_eSPI tft = TFT_eSPI();  // Invoke custom library
// Define screen properties
#define TEXT_HEIGHT 16
#define BOT_FIXED_AREA 0
#define TOP_FIXED_AREA 16
#define YMAX 320

#define BUTTON_PIN 4
#define DEBOUNCE_TIME 25

// Scrolling region
uint16_t yStart = TOP_FIXED_AREA;
uint16_t yArea = YMAX - TOP_FIXED_AREA - BOT_FIXED_AREA;
uint16_t yDraw = YMAX - BOT_FIXED_AREA - TEXT_HEIGHT;

uint16_t xPos = 0;  // X position tracker
byte data = 0;      // Serial byte input

int blank[19] = { 0 };  // Pre-allocate memory
int loopCount = 0;      // Counter to display something different every loop

// Accelerometer object
ADXL362 acc(SPI_CS);

// Gyroscope object
L3G gyro;

```

```
// Global variables for accelerometer readings
MeasurementInMg xyzlow;
MeasurementInMg xyz;
float temp;

// Global variables for gyroscope readings
float gyroX, gyroY, gyroZ;
float theta_g = 0;
float phi_g = 0;
float psi_g = 0;

// Replace with your network credentials
const char ssid[] = "eduino";
const char password[] = "password";

// body is 20, controller is 22
// IP address of the receiver Arduino
const IPAddress peerIP(192, 168, 243, 20);  // Update with the actual IP address
const unsigned int sendPort = 2000;         // Port to send data to

// Local port to listen on
const unsigned int listenPort = 3000;

WiFiUDP Udp;
char incomingPacket[50];  // Buffer for incoming data

int mode = 0;

void setup() {
  Serial.begin(115200);

  // Initialise accelerometer
  check(acc.init());
  acc.printRegisters();
  Serial.print(F("\r\nCHIP REVISION : "));
  Serial.println(acc.getRevisionId());
  check(acc.activateMeasure());
  acc.printRegisters();

  // Initialise gyroscope
  Wire.begin();
  if (!gyro.init()) {
    Serial.println("Failed to autodetect gyro type!");
    while (1)
      ;
  }
  gyro.enableDefault();

  // is below necessary
  // while (!Serial) { ; }  // Wait for Serial Monitor

  // Connect to Wi-Fi
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nConnected to WiFi");

  // Start UDP
  Udp.begin(listenPort);
  Serial.print("Listening on port ");
  Serial.println(listenPort);

  tft.init();
```

```
  tft.setRotation(0);
  tft.fillScreen(TFT_BLACK);

  // Print initial message
  tft.setTextColor(TFT_WHITE, TFT_BLACK);

  setupScrollArea(TOP_FIXED_AREA, BOT_FIXED_AREA);

  // button code

  pinMode(BUTTON_PIN, INPUT_PULLUP);
}

int last_steady_state = LOW;       // the previous steady state from the input pin
int last_flickerable_state = LOW;  // the previous flickerable state from the input pin
int current_state;                 // the current reading from the input pin

unsigned long last_debounce_time = 0;  // the last time the output pin was toggled

float x = 0;
float y = 0;

void loop() {
  unsigned long timer = millis();  // Get current time

  // Read accelerometer values
  xyzlow = acc.getXYZLowPower(ad_range_2G);
  xyz = acc.getXYZ(ad_range_2G);
  temp = acc.getTemperature();

  // Read gyroscope values
  gyro.read();
  gyroX = gyro.g.x * 8.75 / 1000.0;  // Convert raw values to dps
  gyroY = gyro.g.y * 8.75 / 1000.0;
  gyroZ = gyro.g.z * 8.75 / 1000.0;

  // Example variables

  // Create the message
  int* controls = get_control(&mode);

  char message[50];
  snprintf(message, sizeof(message), "v1=%d, v2=%d, mode=%d", controls[0], controls[1],
      mode);

  // Send the message via UDP
  Udp.beginPacket(peerIP, sendPort);
  Udp.write(message);
  Udp.endPacket();

  Serial.print("Sent message: ");
  Serial.println(message);
  int xa, ya;
  int packetSize = Udp.parsePacket();
  if (packetSize) {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);

    // Read the packet into the buffer
    int len = Udp.read(incomingPacket, sizeof(incomingPacket) - 1);
    if (len > 0) {
      incomingPacket[len] = '\0'; // Null-terminate the string
    }

    Serial.print("Received message: ");
    Serial.println(incomingPacket);

    // Parse the received data
```

```
180
181       if (sscanf(incomingPacket, "x=%d, y=%d", &xa, &ya) == 2) {
182         Serial.print("Parsed values - x: ");
183         Serial.print(xa);
184         Serial.print(", y: ");
185         Serial.print(ya);
186       } else {
187         Serial.println("Failed to parse message.");
188       }
189     } else{
190       Serial.println("No message :(");
191     }
192
193     // two delays? - delay(10); // Wait for a second before sending the next packet
194
195     // Ensure loop runs at a stable rate
196     // delay(100 - (millis() - timer)); // uncomment if needed, comes from docs
197
198
199     // read the state of the switch/button:
200     current_state = digitalRead(BUTTON_PIN);
201
202     // If the switch/button changed, due to noise or pressing:
203     if (current_state != last_flickerable_state) {
204       // reset the debouncing timer
205       last_debounce_time = millis();
206       // save the the last flickerable state
207       last_flickerable_state = current_state;
208     }
209
210     if ((millis() - last_debounce_time) > DEBOUNCE_TIME) {
211       // if the button state has changed:
212       if (last_steady_state == HIGH && current_state == LOW) {
213         // Serial.println("The button is pressed");
214         loopCount++;
215       } else if (last_steady_state == LOW && current_state == HIGH) {
216         // Serial.println("The button is released");
217
218
219       }
220       // save the the last steady state
221       last_steady_state = current_state;
222     }
223
224     // Store the last drawn position
225     static float prev_x = 0;
226     static float prev_y = 0;
227
228     // Clear the previous position before drawing a new "x"
229     tft.setTextColor(TFT_BLACK, TFT_BLACK);
230     tft.drawCentreString("x", round(prev_x), round(prev_y), 2);
231
232     // Draw the new "x"
233     tft.setTextColor(TFT_WHITE, TFT_BLACK);
234     tft.drawCentreString("x", round(x), round(y), 2);
235
236     // Update previous position
237     prev_x = x;
238     prev_y = y;
239
240     // Move to next position
241     x = 100 + xa;
242     y = 150 + ya;
243
244     // Handle serial input
245     if (Serial.available()) {
246       data = Serial.read();
247
```

```
    // Scroll when necessary
    if (data == '\r' || xPos > 231) {
      xPos = 0;
      yDraw = scroll_line();
    }

    if (data > 31 && data < 128) {
      xPos += tft.drawChar(data, xPos, yDraw, 2);
      blank[(18 + (yStart - TOP_FIXED_AREA) / TEXT_HEIGHT) % 19] = xPos;
    }
  }
  delay(10); // 10 ms
}

void check(short code) {
  if (code <= 0) {
    Serial.print(F("\r\n**** ERROR: "));
    Serial.print(code);
    Serial.println(F(" ****"));
    if (code == -110)
      Serial.println(F("Device not connected? Check wiring or noisy power supply."));
    else if (code >= -104 && code <= -102)
      Serial.println(F("Check power supply stability."));
    delay(3000);
  } else {
    Serial.println(F("----------------------------------------"));
  }
}

// Scroll display
int scroll_line() {
  int yTemp = yStart;
  tft.fillRect(0, yStart, blank[(yStart - TOP_FIXED_AREA) / TEXT_HEIGHT], TEXT_HEIGHT,
      TFT_BLACK);

  yStart += TEXT_HEIGHT;
  if (yStart >= YMAX - BOT_FIXED_AREA)
    yStart = TOP_FIXED_AREA + (yStart - YMAX + BOT_FIXED_AREA);

  scrollAddress(yStart);
  return yTemp;
}

// Set scrolling area
void setupScrollArea(uint16_t tfa, uint16_t bfa) {
  tft.writecommand(ILI9341_VSCRDEF);
  tft.writedata(tfa >> 8);
  tft.writedata(tfa);
  tft.writedata((YMAX - tfa - bfa) >> 8);
  tft.writedata(YMAX - tfa - bfa);
  tft.writedata(bfa >> 8);
  tft.writedata(bfa);
}

// Update scroll pointer
void scrollAddress(uint16_t vsp) {
  tft.writecommand(ILI9341_VSCRSADD);
  tft.writedata(vsp >> 8);
  tft.writedata(vsp);
}

int* get_control(int* mode_a) {
  static int array[2];

  MeasurementInMg a_p_low = acc.getXYZLowPower(ad_range_2G);
  MeasurementInMg a_p = acc.getXYZ(ad_range_2G);

```

```
  double mag_a_p = sqrt((a_p.x / 1000) * (a_p.x / 1000) + (a_p.y / 1000) * (a_p.y / 1000) + (a_p.z / 1000) * (a_p.z / 1000));
  double* angles = RP_calculate(a_p);

  if (mag_a_p > (1.5 * 9.81)) {
    if (*mode_a == 1) {
      *mode_a = 0;
    } else {
      *mode_a = 1;
    }
    //buzzer logic
  }

  float theta_g_vel, phi_g_vel, psi_g_vel;
  gyro.read();
  theta_g_vel = gyro.g.x * 8.75 / 1000.0;  // Convert raw values to dps
  phi_g_vel = gyro.g.y * 8.75 / 1000.0;
  psi_g_vel = gyro.g.z * 8.75 / 1000.0;


  float time_gap = 0.01;
  float threshold_difference = 5;
  float threshold = 20;


  if (theta_g_vel < 0.8 and theta_g_vel > -0.58) {
    theta_g_vel = 0;
  }

  theta_g += theta_g_vel * time_gap;
  phi_g += phi_g_vel * time_gap;
  psi_g += psi_g_vel * time_gap;

 // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
  // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
    // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
      // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
        // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
          // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
        // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
      // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
    // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
  // !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think
// !!! angles is an array of length 2, how is angles[2] being accessed? it should be angles[0] and angles[1] I think

  if (fabs(theta_g - angles[0]) > threshold_difference && fabs(phi_g - angles[1]) > threshold_difference) {
    if (fabs(angles[0]) > threshold) {
      if (angles[0] > 0) {
        array[0] = 500;
        array[1] = 1000;
      } else {
        array[0] = 1000;
        array[1] = 500;
      }
    } else if (fabs(angles[1]) > threshold) {
      if (angles[1] > 0) {
```

```
        array[0] = 1000;
        array[1] = 1000;
      } else {
        array[0] = -1000;
        array[1] = -1000;
      }
    }
  } else {
    array[0] = 0;
    array[1] = 0;
  }

  //   // Display accelerometer values
  // Serial.print(F("ACC (8-bit) x: "));
  // Serial.print(a_p_low.x / 1000.0);
  // Serial.print(F(" y: "));
  // Serial.print(a_p_low.y / 1000.0);
  // Serial.print(F(" z: "));
  // Serial.print(a_p_low.z / 1000.0);
  // Serial.println("");

  // Display gyroscope values
  // Serial.print("GYRO theta: ");
  // Serial.print(theta_g);
  // Serial.print(" phi: ");
  // Serial.print(phi_g);

  // Serial.print(" ACC Theta: ");
  // Serial.print(angles[0]);
  // Serial.print(" phi: ");
  // Serial.print(angles[1]);
  // Serial.println(" ");

  // Serial.print("motor speed left: ");
  // Serial.print(array[0]);
  // Serial.print(", motor speed right: ");
  // Serial.print(array[1]);

  return array;
}


// calculate the Roll&Pitch
double* RP_calculate(MeasurementInMg acc) {
  static double angles[2];
  double x_Buff = float(acc.x);
  double y_Buff = float(acc.y);
  double z_Buff = float(acc.z);

  double roll = atan2(y_Buff, z_Buff) * 57.3;
  double pitch = atan2((-x_Buff), sqrt(y_Buff * y_Buff + z_Buff * z_Buff)) * 57.3;

  angles[0] = roll;
  angles[1] = pitch;

  return angles;
}
```

Listing 1: Arduino code for the controller

A.2 Main Body Arduino Code

```
#include <Wire.h>
#include <Motoron.h>
#include <math.h>
#include <WiFiS3.h>
#include <WiFiUdp.h>
```

```
#include <tracks.h>

// ADXL345 Accelerometer
#define DEVICE (0x53)
#define TO_READ (6)
char str[512];

// Buzzer and LED pins
#define BLUE_PIN 7 // Probably useless
#define GREEN_PIN 8 // PWM
#define BUZZER_PIN 9
#define RED_PIN 10 // PWM

// State Detection Thresholds
#define ACCEL_THRESHOLD 5000   // Adjust based on sensitivity
#define TILT_THRESHOLD 30.0    // Max tilt angle before state = 0 (degrees)
#define UPSIDE_DOWN_THRESHOLD -9.81 // Approx -1g, indicating inversion
int state = 1; // Default to normal operation

// Motor Encoder Definitions
const int encoder_ticks_per_rev = 292.7 / 2;
const float wheel_radius = 63.0 / 2.0;
const float wheel_circumference = 2.0 * M_PI * wheel_radius;
const float wheel_body_separation = 300.0;
const int motor_speed = 2000;
volatile int motor_speed_l = 1000;
volatile int motor_speed_r = 1000;

#define encoder_a_l 2
#define encoder_a_r 3
#define encoder_b_l 4
#define encoder_b_r 5

MotoronI2C mc;

volatile int voltage_sign_l = -1;
volatile int voltage_sign_r = 1;

volatile long encoder_ticks_l = 0;
volatile long encoder_ticks_r = 0;
long prev_ticks_l = 0, prev_ticks_r = 0;

float x = 0.0, y = 0.0, theta = 0.0, v = 0.0, w = 0.0;
unsigned long prev_time = 0;

// Accelerometer Variables
byte buff[TO_READ];
int regAddress = 0x32;
int x_a, y_a, z_a;
double roll = 0.00, pitch = 0.00;

// Music Variables
int *melodies[] = {melody1, melody2, melody3, melody4, melody5, melody6};
int *durations[] = {durations1, durations2, durations3, durations4, durations5, durations6};
int noteCounts[] = {
    sizeof(durations1) / sizeof(int),
    sizeof(durations2) / sizeof(int),
    sizeof(durations3) / sizeof(int),
    sizeof(durations4) / sizeof(int),
    sizeof(durations5) / sizeof(int),
    sizeof(durations6) / sizeof(int)};

int noteIndex = 0;
int trackIndex = 0;
unsigned long previousNoteTime = 0;
bool isPlaying = false;
const int totalTracks = 6;

```

```
// Replace with your network credentials
const char ssid[] = "eduino";
const char password[] = "password";

// body is 20, controller is 22
// IP address of the receiver Arduino
const IPAddress peerIP(192, 168, 243, 22);  // Update with the actual IP address
const unsigned int sendPort = 3000;         // Port to send data to

// Local port to listen on
const unsigned int listenPort = 2000;

WiFiUDP Udp;
char incomingPacket[50]; // Buffer for incoming data

void handleEncoderA_L()
{
  if (motor_speed_l > 0)
    encoder_ticks_l++;
  else
    encoder_ticks_l--;
}

void handleEncoderA_R()
{
  if (motor_speed_r > 0)
    encoder_ticks_r++;
  else
    encoder_ticks_r--;
}

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

  // Motoron Initialisation
  mc.reinitialize();
  mc.disableCrc();
  mc.clearResetFlag();
  mc.setMaxAcceleration(1, 140);
  mc.setMaxDeceleration(1, 300);
  mc.setMaxAcceleration(3, 140);
  mc.setMaxDeceleration(3, 300);

  // Encoder Interrupts
  pinMode(encoder_a_l, INPUT);
  pinMode(encoder_b_l, INPUT);
  pinMode(encoder_a_r, INPUT);
  pinMode(encoder_b_r, INPUT);
  attachInterrupt(digitalPinToInterrupt(encoder_a_l), handleEncoderA_L, RISING);
  attachInterrupt(digitalPinToInterrupt(encoder_a_r), handleEncoderA_R, RISING);

  // LED & Buzzer Setup
  pinMode(BUZZER_PIN, OUTPUT);
  pinMode(RED_PIN, OUTPUT);
  pinMode(GREEN_PIN, OUTPUT);
  pinMode(BLUE_PIN, OUTPUT);

  // Accelerometer Initialisation
  writeTo(DEVICE, 0x2D, 0);
  writeTo(DEVICE, 0x2D, 16);
  writeTo(DEVICE, 0x2D, 8);

  prev_time = millis();

  // WiFi Setup
  Serial.print("Connecting to ");
```

```
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nConnected to WiFi");

  // Start UDP
  Udp.begin(listenPort);
  Serial.print("Listening on port ");
  Serial.println(listenPort);

}

void loop() {
  // mc.setSpeed(1, 2000);
  // mc.setSpeed(3, -2000);

  updateAccelerometer();
  checkRobotState();
  updateOdometry();
  playMusicNonBlocking();
  handleWiFi(x, y);

  // Serial.print("The acceleration info of x, y, z are:");
  sprintf(str, "%d %d %d", x_a, y_a, z_a);
  // Serial.print(str);
  Serial.write(10);
  //Roll & Pitch calculate
  RP_calculate();
  // Serial.print("Roll:"); Serial.println( roll );
  // Serial.print("Pitch:"); Serial.println( pitch );
  // Serial.println("");

  delay(10); // delay 10 ms
}

// Odometry Update
void updateOdometry()
{
  unsigned long current_time = millis();
  float dt = (current_time - prev_time) / 1000.0;

  long current_ticks_l = encoder_ticks_l;
  long current_ticks_r = encoder_ticks_r;

  float dL = ((current_ticks_l - prev_ticks_l) * wheel_circumference) /
      encoder_ticks_per_rev;
  float dR = ((current_ticks_r - prev_ticks_r) * wheel_circumference) /
      encoder_ticks_per_rev;

  prev_ticks_l = current_ticks_l;
  prev_ticks_r = current_ticks_r;

  float dTheta = (dR - dL) / (2.0 * wheel_body_separation);
  float D = (dR + dL) / 2.0;

  theta += dTheta;
  x += D * cos(theta);
  y += D * sin(theta);

  float vL = dL / dt, vR = dR / dt;
  v = (vL + vR) / 2.0;
  w = wheel_radius * (vR - vL) / (2.0 * wheel_body_separation);

```

```
  prev_time = current_time;

  // Serial Output
  // Serial.print("X:");
  // Serial.print(x);
  // Serial.print("\tY:");
  // Serial.print(y);
  // Serial.print("\tZero:");
  // Serial.println(0);

  // Motor Control
  mc.setSpeed(1, -motor_speed_l);
  mc.setSpeed(3, motor_speed_r);
  // mc.setSpeed(1, 2000);
  // mc.setSpeed(3, -2000);
}

// Accelerometer Update
void updateAccelerometer()
{
  readFrom(DEVICE, regAddress, TO_READ, buff);
  x_a = (int16_t)((buff[1] << 8) | buff[0]);
  y_a = (int16_t)((buff[3] << 8) | buff[2]);
  z_a = (int16_t)((buff[5] << 8) | buff[4]);

  RP_calculate();
}

// Roll & Pitch Calculation
void RP_calculate()
{
  roll = atan2(y_a, z_a) * 57.3;
  pitch = atan2(-x_a, sqrt(y_a * y_a + z_a * z_a)) * 57.3;
}

// Finding the robot's state from accelerometer values
void checkRobotState() {
    // Convert raw accelerometer data to meaningful acceleration values
    double ax = x_a * 0.0039; // Assuming ADXL345 scale (LSB to g conversion)
    double ay = y_a * 0.0039;
    double az = z_a * 0.0039 * 9.81; // Convert to m/s^2

    // Check for Bumping (Short small spikes in acceleration)
    if (abs(ax) > ACCEL_THRESHOLD || abs(ay) > ACCEL_THRESHOLD || abs(az) >
        ACCEL_THRESHOLD) {
        state = 0;
    }
    // Check for Jumping (Free fall: az approaches 0 or becomes negative)
    else if (az < 1.0) {
        state = 0;
    }
    // Check for Tilting (Roll or Pitch exceeding threshold)
    else if (abs(roll) > TILT_THRESHOLD || abs(pitch) > TILT_THRESHOLD) {
        state = 0;
    }
    // Check for Landing Upside Down (Large impact + az near -1g)
    else if (az < UPSIDE_DOWN_THRESHOLD) {
        state = 0;
    }
    else {
        state = 1;
    }

    updateIndicators();
}

void updateIndicators() {
    if (state == 1) {
```

```
275         setColour(0, 255, 0);
276         isPlaying = false; // Allow normal song playback
277     } else {
278         setColour(255, 0, 0);
279         tone(BUZZER_PIN, 200); // High-pitched beep for error
280         isPlaying = true; // Stop music
281     }
282 }
283 
284 // RGB LED Control
285 void setColour(int R, int G, int B)
286 {
287   analogWrite(RED_PIN, R);
288   analogWrite(GREEN_PIN, G);
289   analogWrite(BLUE_PIN, B);
290 }
291 
292 // Non-blocking Music Playback
293 void playMusicNonBlocking()
294 {
295   unsigned long currentMillis = millis();
296   int *melody = melodies[trackIndex];
297   int *trackDuration = durations[trackIndex];
298   int numNotes = noteCounts[trackIndex];
299 
300   // If the state is 0, override music with an alarm sound
301   if (state == 0)
302   {
303     tone(BUZZER_PIN, 2000); // High-pitched alarm
304     return; // Exit the function to prevent normal music from playing
305   }
306   else
307   {
308     noTone(BUZZER_PIN); // Stop beeping when state is not 0
309   }
310 
311   // If enough time has passed, move to the next note
312   if (!isPlaying || (currentMillis - previousNoteTime > (1000 / trackDuration[noteIndex]) *
          1.3))
313   {
314     noTone(BUZZER_PIN); // Stop previous note
315 
316     int duration = 1000 / trackDuration[noteIndex];
317     tone(BUZZER_PIN, melody[noteIndex], duration);
318 
319     previousNoteTime = currentMillis;
320     isPlaying = true;
321 
322     // Move to the next note
323     noteIndex++;
324 
325     // If the song finishes, cycle to the next track
326     if (noteIndex >= numNotes)
327     {
328       noteIndex = 0;
329       trackIndex = (trackIndex + 1) % totalTracks;
330     }
331   }
332 }
333 
334 void handleWiFi(float x, float y)
335 {
336   // Send
337   char message[50];
338   snprintf(message, sizeof(message), "x=%d, y=%d", (int)x, (int)y);  // Cast to int if
          needed
339 
340   // Send the message via UDP
```

```
  Udp.beginPacket(peerIP, sendPort);
  Udp.write(message);
  Udp.endPacket();

  Serial.print("Sent message: ");
  Serial.println(message);

  // Receive
  int packetSize = Udp.parsePacket();
  if (packetSize)
  {
    int len = Udp.read(incomingPacket, sizeof(incomingPacket) - 1);
    if (len > 0)
      incomingPacket[len] = '\0';

    Serial.print("Received message: ");
    Serial.println(incomingPacket);

    int v1, v2, mode;
    if (sscanf(incomingPacket, "v1=%d, v2=%d, mode=%d", &motor_speed_l, &motor_speed_r,
        &mode) == 3)
    {
      Serial.print("Parsed v1: "); Serial.print(motor_speed_l);
      Serial.print(" v2: "); Serial.print(motor_speed_r);
      Serial.print(" mode: "); Serial.println(mode);

      // Optional: You can use v1, v2, mode to change behaviour
    }
  }
  else
  {
    Serial.print("sad :(");
  }
}

// I2C Communication
void writeTo(int device, byte address, byte val)
{
  Wire.beginTransmission(device);
  Wire.write(address);
  Wire.write(val);
  Wire.endTransmission();
}

void readFrom(int device, byte address, int num, byte buff[])
{
  Wire.beginTransmission(device);
  Wire.write(address);
  Wire.endTransmission();
  Wire.requestFrom(device, num);
  for (int i = 0; Wire.available(); i++)
    buff[i] = Wire.read();
}
```

Listing 2: Arduino code for the main body of the robot

```
#include <pitches.h>

// Star Wars

int melody1[] = {
    NOTE_AS4, NOTE_AS4, NOTE_AS4,
    NOTE_F5, NOTE_C6,
    NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F6, NOTE_C6,
    NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F6, NOTE_C6,
    NOTE_AS5, NOTE_A5, NOTE_AS5, NOTE_G5, NOTE_C5, NOTE_C5, NOTE_C5,
    NOTE_F5, NOTE_C6,
    NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F6, NOTE_C6,
```

```
13
14      NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F6, NOTE_C6,
15      NOTE_AS5, NOTE_A5, NOTE_AS5, NOTE_G5, NOTE_C5, NOTE_C5,
16      NOTE_D5, NOTE_D5, NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F5,
17      NOTE_F5, NOTE_G5, NOTE_A5, NOTE_G5, NOTE_D5, NOTE_E5, NOTE_C5, NOTE_C5,
18      NOTE_D5, NOTE_D5, NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F5,
19
20      NOTE_C6, NOTE_G5, NOTE_G5, REST, NOTE_C5,
21      NOTE_D5, NOTE_D5, NOTE_AS5, NOTE_A5, NOTE_G5, NOTE_F5,
22      NOTE_F5, NOTE_G5, NOTE_A5, NOTE_G5, NOTE_D5, NOTE_E5, NOTE_C6, NOTE_C6,
23      NOTE_F6, NOTE_DS6, NOTE_CS6, NOTE_C6, NOTE_AS5, NOTE_GS5, NOTE_G5, NOTE_F5,
24      NOTE_C6};
25
26  int durations1[] = {
27      9, 9, 9,
28      2, 2,
29      9, 9, 9, 2, 4,
30      9, 9, 9, 2, 4,
31      9, 9, 9, 2, 9, 9, 9,
32      2, 2,
33      9, 9, 9, 2, 4,
34
35      9, 9, 9, 2, 4,
36      9, 9, 9, 2, 8, 8,
37      3, 8, 8, 8, 8, 8,
38      9, 9, 9, 3, 8, 3, 8, 8,
39      3, 8, 8, 8, 8, 8,
40
41      8, 16, 2, 8, 8,
42      3, 8, 8, 8, 8, 8,
43      8, 8, 8, 3, 8, 3, 8, 8,
44      3, 8, 3, 8, 3, 8, 3, 8,
45      1};
46
47  // Korobeiniki (Tetris)
48  int melody2[] = {
49      NOTE_E5, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_C5, NOTE_B4,
50      NOTE_A4, NOTE_C5, NOTE_E5, NOTE_D5, NOTE_C5,
51      NOTE_B4, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_E5,
52      NOTE_C5, NOTE_A4, NOTE_A4,
53
54      NOTE_D5, NOTE_F5, NOTE_A5, NOTE_G5, NOTE_F5,
55      NOTE_E5, NOTE_C5, NOTE_E5, NOTE_D5, NOTE_C5,
56      NOTE_B4, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_E5,
57      NOTE_C5, NOTE_A4, NOTE_A4, REST,
58
59      NOTE_E5, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_C5, NOTE_B4,
60      NOTE_A4, NOTE_C5, NOTE_E5, NOTE_D5, NOTE_C5,
61      NOTE_B4, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_E5,
62      NOTE_C5, NOTE_A4, NOTE_A4,
63
64      NOTE_D5, NOTE_F5, NOTE_A5, NOTE_G5, NOTE_F5,
65      NOTE_E5, NOTE_C5, NOTE_E5, NOTE_D5, NOTE_C5,
66      NOTE_B4, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_E5,
67      NOTE_C5, NOTE_A4, NOTE_A4, REST,
68
69      NOTE_E5, NOTE_C5,
70      NOTE_D5, NOTE_B4,
71      NOTE_C5, NOTE_A4,
72      NOTE_GS4, NOTE_B4, REST,
73      NOTE_E5, NOTE_C5,
74      NOTE_D5, NOTE_B4,
75      NOTE_C5, NOTE_E5, NOTE_A5,
76      NOTE_GS5};
77
78  int durations2[] = {
79      4, 8, 8, 4, 8, 8,
80      3, 8, 4, 8, 8,
```

```
    4, 8, 8, 4, 4,
    4, 4, 2,

    3, 8, 4, 8, 8,
    3, 8, 4, 8, 8,
    4, 8, 8, 4, 4,
    4, 4, 2, 4,

    4, 8, 8, 4, 8, 8,
    3, 8, 4, 8, 8,
    4, 8, 8, 4, 4,
    4, 4, 2,

    4, 8, 4, 8, 8,
    4, 8, 4, 8, 8,
    4, 8, 8, 4, 4,
    4, 4, 4, 4,

    2, 2,
    2, 2,
    2, 2,
    2, 4, 8,
    2, 2,
    2, 2,
    4, 4, 2,
    2};

// Marios Bros

int melody3[] = {
    NOTE_E5, NOTE_E5, REST, NOTE_E5, REST, NOTE_C5, NOTE_E5,
    NOTE_G5, REST, NOTE_G4, REST,
    NOTE_C5, NOTE_G4, REST, NOTE_E4,
    NOTE_A4, NOTE_B4, NOTE_AS4, NOTE_A4,
    NOTE_G4, NOTE_E5, NOTE_G5, NOTE_A5, NOTE_F5, NOTE_G5,
    REST, NOTE_E5, NOTE_C5, NOTE_D5, NOTE_B4,
    NOTE_C5, NOTE_G4, REST, NOTE_E4,
    NOTE_A4, NOTE_B4, NOTE_AS4, NOTE_A4,
    NOTE_G4, NOTE_E5, NOTE_G5, NOTE_A5, NOTE_F5, NOTE_G5,
    REST, NOTE_E5, NOTE_C5, NOTE_D5, NOTE_B4,

    REST, NOTE_G5, NOTE_FS5, NOTE_F5, NOTE_DS5, NOTE_E5,
    REST, NOTE_GS4, NOTE_A4, NOTE_C4, REST, NOTE_A4, NOTE_C5, NOTE_D5,
    REST, NOTE_DS5, REST, NOTE_D5,
    NOTE_C5, REST,

    REST, NOTE_G5, NOTE_FS5, NOTE_F5, NOTE_DS5, NOTE_E5,
    REST, NOTE_GS4, NOTE_A4, NOTE_C4, REST, NOTE_A4, NOTE_C5, NOTE_D5,
    REST, NOTE_DS5, REST, NOTE_D5,
    NOTE_C5, REST,

    NOTE_C5, NOTE_C5, NOTE_C5, REST, NOTE_C5, NOTE_D5,
    NOTE_E5, NOTE_C5, NOTE_A4, NOTE_G4,

    NOTE_C5, NOTE_C5, NOTE_C5, REST, NOTE_C5, NOTE_D5, NOTE_E5,
    REST,
    NOTE_C5, NOTE_C5, NOTE_C5, REST, NOTE_C5, NOTE_D5,
    NOTE_E5, NOTE_C5, NOTE_A4, NOTE_G4,
    NOTE_E5, NOTE_E5, REST, NOTE_E5, REST, NOTE_C5, NOTE_E5,
    NOTE_G5, REST, NOTE_G4, REST,
    NOTE_C5, NOTE_G4, REST, NOTE_E4,

    NOTE_A4, NOTE_B4, NOTE_AS4, NOTE_A4,
    NOTE_G4, NOTE_E5, NOTE_G5, NOTE_A5, NOTE_F5, NOTE_G5,
    REST, NOTE_E5, NOTE_C5, NOTE_D5, NOTE_B4,

    NOTE_C5, NOTE_G4, REST, NOTE_E4,
    NOTE_A4, NOTE_B4, NOTE_AS4, NOTE_A4,
```

```
    NOTE_G4, NOTE_E5, NOTE_G5, NOTE_A5, NOTE_F5, NOTE_G5,
    REST, NOTE_E5, NOTE_C5, NOTE_D5, NOTE_B4,

    NOTE_E5, NOTE_C5, NOTE_G4, REST, NOTE_GS4,
    NOTE_A4, NOTE_F5, NOTE_F5, NOTE_A4,
    NOTE_D5, NOTE_A5, NOTE_A5, NOTE_A5, NOTE_G5, NOTE_F5,

    NOTE_E5, NOTE_C5, NOTE_A4, NOTE_G4,
    NOTE_E5, NOTE_C5, NOTE_G4, REST, NOTE_GS4,
    NOTE_A4, NOTE_F5, NOTE_F5, NOTE_A4,
    NOTE_B4, NOTE_F5, NOTE_F5, NOTE_F5, NOTE_E5, NOTE_D5,
    NOTE_C5, NOTE_E4, NOTE_E4, NOTE_C4,

    NOTE_E5, NOTE_C5, NOTE_G4, REST, NOTE_GS4,
    NOTE_A4, NOTE_F5, NOTE_F5, NOTE_A4,
    NOTE_D5, NOTE_A5, NOTE_A5, NOTE_A5, NOTE_G5, NOTE_F5,

    NOTE_E5, NOTE_C5, NOTE_A4, NOTE_G4,
    NOTE_E5, NOTE_C5, NOTE_G4, REST, NOTE_GS4,
    NOTE_A4, NOTE_F5, NOTE_F5, NOTE_A4,
    NOTE_B4, NOTE_F5, NOTE_F5, NOTE_F5, NOTE_E5, NOTE_D5,
    NOTE_C5, NOTE_E4, NOTE_E4, NOTE_C4,
    NOTE_C5, NOTE_C5, NOTE_C5, REST, NOTE_C5, NOTE_D5, NOTE_E5,
    REST,

    NOTE_C5, NOTE_C5, NOTE_C5, REST, NOTE_C5, NOTE_D5,
    NOTE_E5, NOTE_C5, NOTE_A4, NOTE_G4,
    NOTE_E5, NOTE_E5, REST, NOTE_E5, REST, NOTE_C5, NOTE_E5,
    NOTE_G5, REST, NOTE_G4, REST,
    NOTE_E5, NOTE_C5, NOTE_G4, REST, NOTE_GS4,
    NOTE_A4, NOTE_F5, NOTE_F5, NOTE_A4,
    NOTE_D5, NOTE_A5, NOTE_A5, NOTE_A5, NOTE_G5, NOTE_F5,

    NOTE_E5, NOTE_C5, NOTE_A4, NOTE_G4,
    NOTE_E5, NOTE_C5, NOTE_G4, REST, NOTE_GS4,
    NOTE_A4, NOTE_F5, NOTE_F5, NOTE_A4,
    NOTE_B4, NOTE_F5, NOTE_F5, NOTE_F5, NOTE_E5, NOTE_D5,
    NOTE_C5, NOTE_E4, NOTE_E4, NOTE_C4,

    // Game over sound
    NOTE_C5, NOTE_G4, NOTE_E4,
    NOTE_A4, NOTE_B4, NOTE_A4, NOTE_GS4, NOTE_AS4, NOTE_GS4,
    NOTE_G4, NOTE_D4, NOTE_E4};

int durations3[] = {
    8, 8, 8, 8, 8, 8, 8,
    4, 4, 8, 4,
    4, 8, 4, 4,
    4, 4, 8, 4,
    8, 8, 8, 4, 8, 8,
    8, 4, 8, 8, 4,
    4, 8, 4, 4,
    4, 4, 8, 4,
    8, 8, 8, 4, 8, 8,
    8, 4, 8, 8, 4,

    4, 8, 8, 8, 4, 8,
    8, 8, 8, 8, 8, 8, 8, 8,
    4, 4, 8, 4,
    2, 2,

    4, 8, 8, 8, 4, 8,
    8, 8, 8, 8, 8, 8, 8, 8,
    4, 4, 8, 4,
    2, 2,

    8, 4, 8, 8, 8, 4,
    8, 4, 8, 2,
```

```
217
218     8, 4, 8, 8, 8, 8, 8,
219     1,
220     8, 4, 8, 8, 8, 4,
221     8, 4, 8, 2,
222     8, 8, 8, 8, 8, 8, 4,
223     4, 4, 4, 4,
224     4, 8, 4, 4,
225
226     4, 4, 8, 4,
227     8, 8, 8, 4, 8, 8,
228     8, 4, 8, 8, 4,
229
230     4, 8, 4, 4,
231     4, 4, 8, 4,
232     8, 8, 8, 4, 8, 8,
233     8, 4, 8, 8, 4,
234
235     8, 4, 8, 4, 4,
236     8, 4, 8, 2,
237     8, 8, 8, 8, 8, 8,
238
239     8, 4, 8, 2,
240     8, 4, 8, 4, 4,
241     8, 4, 8, 2,
242     8, 4, 8, 8, 8, 8,
243     8, 4, 8, 2,
244
245     8, 4, 8, 4, 4,
246     8, 4, 8, 2,
247     8, 8, 8, 8, 8, 8,
248
249     8, 4, 8, 2,
250     8, 4, 8, 4, 4,
251     8, 4, 8, 2,
252     8, 4, 8, 8, 8, 8,
253     8, 4, 8, 2,
254     8, 4, 8, 8, 8, 8, 8,
255     1,
256
257     8, 4, 8, 8, 8, 4,
258     8, 4, 8, 2,
259     8, 8, 8, 8, 8, 8, 4,
260     4, 4, 4, 4,
261     8, 4, 8, 4, 4,
262     8, 4, 8, 2,
263     8, 8, 8, 8, 8, 8,
264
265     8, 4, 8, 2,
266     8, 4, 8, 4, 4,
267     8, 4, 8, 2,
268     8, 4, 8, 8, 8, 8,
269     8, 4, 8, 2,
270
271     // game over sound
272     4, 4, 4,
273     8, 8, 8, 8, 8, 8,
274     8, 8, 2};
275
276 // Pacman
277
278 int melody4[] = {
279
280     NOTE_B4, NOTE_B5, NOTE_FS5, NOTE_DS5,
281     NOTE_B5, NOTE_FS5, NOTE_DS5, NOTE_C5,
282     NOTE_C6, NOTE_G6, NOTE_E6, NOTE_C6, NOTE_G6, NOTE_E6,
283
284     NOTE_B4, NOTE_B5, NOTE_FS5, NOTE_DS5, NOTE_B5,
```

```
    NOTE_FS5, NOTE_DS5, NOTE_DS5, NOTE_E5, NOTE_F5,
    NOTE_F5, NOTE_FS5, NOTE_G5, NOTE_G5, NOTE_GS5, NOTE_A5, NOTE_B5};

int durations4[] = {
    16, 16, 16, 16,
    32, 16, 8, 16,
    16, 16, 16, 32, 16, 8,

    16, 16, 16, 16, 32,
    16, 8, 32, 32, 32,
    32, 32, 32, 32, 32, 16, 8};

// Pirates of the Caribbean

int melody5[] = {
    NOTE_E4, NOTE_G4, NOTE_A4, NOTE_A4, REST,
    NOTE_A4, NOTE_B4, NOTE_C5, NOTE_C5, REST,
    NOTE_C5, NOTE_D5, NOTE_B4, NOTE_B4, REST,
    NOTE_A4, NOTE_G4, NOTE_A4, REST,

    NOTE_E4, NOTE_G4, NOTE_A4, NOTE_A4, REST,
    NOTE_A4, NOTE_B4, NOTE_C5, NOTE_C5, REST,
    NOTE_C5, NOTE_D5, NOTE_B4, NOTE_B4, REST,
    NOTE_A4, NOTE_G4, NOTE_A4, REST,

    NOTE_E4, NOTE_G4, NOTE_A4, NOTE_A4, REST,
    NOTE_A4, NOTE_C5, NOTE_D5, NOTE_D5, REST,
    NOTE_D5, NOTE_E5, NOTE_F5, NOTE_F5, REST,
    NOTE_E5, NOTE_D5, NOTE_E5, NOTE_A4, REST,

    NOTE_A4, NOTE_B4, NOTE_C5, NOTE_C5, REST,
    NOTE_D5, NOTE_E5, NOTE_A4, REST,
    NOTE_A4, NOTE_C5, NOTE_B4, NOTE_B4, REST,
    NOTE_C5, NOTE_A4, NOTE_B4, REST,

    NOTE_A4, NOTE_A4,
    // Repeat of first part
    NOTE_A4, NOTE_B4, NOTE_C5, NOTE_C5, REST,
    NOTE_C5, NOTE_D5, NOTE_B4, NOTE_B4, REST,
    NOTE_A4, NOTE_G4, NOTE_A4, REST,

    NOTE_E4, NOTE_G4, NOTE_A4, NOTE_A4, REST,
    NOTE_A4, NOTE_B4, NOTE_C5, NOTE_C5, REST,
    NOTE_C5, NOTE_D5, NOTE_B4, NOTE_B4, REST,
    NOTE_A4, NOTE_G4, NOTE_A4, REST,

    NOTE_E4, NOTE_G4, NOTE_A4, NOTE_A4, REST,
    NOTE_A4, NOTE_C5, NOTE_D5, NOTE_D5, REST,
    NOTE_D5, NOTE_E5, NOTE_F5, NOTE_F5, REST,
    NOTE_E5, NOTE_D5, NOTE_E5, NOTE_A4, REST,

    NOTE_A4, NOTE_B4, NOTE_C5, NOTE_C5, REST,
    NOTE_D5, NOTE_E5, NOTE_A4, REST,
    NOTE_A4, NOTE_C5, NOTE_B4, NOTE_B4, REST,
    NOTE_C5, NOTE_A4, NOTE_B4, REST,
    // End of Repeat

    NOTE_E5, REST, REST, NOTE_F5, REST, REST,
    NOTE_E5, NOTE_E5, REST, NOTE_G5, REST, NOTE_E5, NOTE_D5, REST, REST,
    NOTE_D5, REST, REST, NOTE_C5, REST, REST,
    NOTE_B4, NOTE_C5, REST, NOTE_B4, REST, NOTE_A4,

    NOTE_E5, REST, REST, NOTE_F5, REST, REST,
    NOTE_E5, NOTE_E5, REST, NOTE_G5, REST, NOTE_E5, NOTE_D5, REST, REST,
    NOTE_D5, REST, REST, NOTE_C5, REST, REST,
    NOTE_B4, NOTE_C5, REST, NOTE_B4, REST, NOTE_A4};

int durations5[] = {
```

```
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8,

    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8,

    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 8, 4, 8,

    8, 8, 4, 8, 8,
    4, 8, 4, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 4,

    4, 8,
    // Repeat of First Part
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8,

    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8,

    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 8, 8,
    8, 8, 8, 4, 8,

    8, 8, 4, 8, 8,
    4, 8, 4, 8,
    8, 8, 4, 8, 8,
    8, 8, 4, 4,
    // End of Repeat

    4, 8, 4, 4, 8, 4,
    8, 8, 8, 8, 8, 8, 8, 8, 4,
    4, 8, 4, 4, 8, 4,
    8, 8, 8, 8, 8, 2,

    4, 8, 4, 4, 8, 4,
    8, 8, 8, 8, 8, 8, 8, 8, 4,
    4, 8, 4, 4, 8, 4,
    8, 8, 8, 8, 8, 2};

// Harry Potter

int melody6[] = {
    REST, NOTE_D4,
    NOTE_G4, NOTE_AS4, NOTE_A4,
    NOTE_G4, NOTE_D5,
    NOTE_C5,
    NOTE_A4,
    NOTE_G4, NOTE_AS4, NOTE_A4,
    NOTE_F4, NOTE_GS4,
    NOTE_D4,
    NOTE_D4, REST, NOTE_D4,

    NOTE_G4, NOTE_AS4, NOTE_A4,
    NOTE_G4, NOTE_D5,
    NOTE_F5, NOTE_E5,
```

```
    NOTE_DS5, NOTE_B4,
    NOTE_DS5, NOTE_D5, NOTE_CS5,
    NOTE_CS4, NOTE_B4,
    NOTE_G4,
    NOTE_AS4,

    NOTE_D5, NOTE_AS4,
    NOTE_D5, NOTE_AS4,
    NOTE_DS5, NOTE_D5,
    NOTE_CS5, NOTE_A4,
    NOTE_AS4, NOTE_D5, NOTE_CS5,
    NOTE_CS4, NOTE_D4,
    NOTE_D5,
    REST, NOTE_AS4,

    NOTE_D5, NOTE_AS4,
    NOTE_D5, NOTE_AS4,
    NOTE_F5, NOTE_E5,
    NOTE_DS5, NOTE_B4,
    NOTE_DS5, NOTE_D5, NOTE_CS5,
    NOTE_CS4, NOTE_AS4,
    NOTE_G4};

int durations6[] = {
    2, 4,
    3, 8, 4,
    2, 4,
    1.33,
    1.33,
    3, 8, 4,
    2, 4,
    1.33,
    4, 4, 4,

    3, 8, 4,
    2, 4,
    2, 4,
    2, 4,
    3, 8, 4,
    2, 4,
    1,
    4,

    2, 4,
    2, 4,
    2, 4,
    2, 4,
    3, 8, 4,
    2, 4,
    1,
    4, 4,

    2, 4,
    2, 4,
    2, 4,
    2, 4,
    3, 8, 4,
    2, 4,
    1};
```

Listing 3: Custom, adapted track melodies and durations for the buzzer

References

[1] Lady Ada. *Adafruit 2.4" Color TFT Touchscreen Breakout*, 2016. Accessed: 2025-03-19.

[2] Marshall Brain. How the wii works, n.d. Accessed: 2025-02-20.

[3] Kok Seng Chong and Lindsay Kleeman. Accurate odometry and error modelling for a mobile robot. Technical Report MECSE-1996-6, Intelligent Robotics Research Centre (IRRC), Department of Electrical and Computer Systems Engineering, Monash University, Clayton, Victoria 3168, Australia, 1996.

[4] Digilent. *Pmod GYRO Reference Manual*, 2016. Accessed: 2025-03-03.

[5] Digilent. *Pmod ACL2 Reference Manual*, unknown. Accessed: 2025-03-03.

[6] Jacob Hylén. *UNO R4 WiFi Network Examples*, 2025. Accessed: 2025-03-19.

[7] IRobot. *iRobot Create 2 Open Interface (OI)*, 2015. Accessed: 2025-02-21.

[8] Kteam. *Khepera IV*, 2016. Accessed: 2025-02-19.

[9] Makeblock. *mBot*, 2017. Accessed: 2025-02-22.

[10] Mark Pedley. Tilt sensing using a three-axis accelerometer, 2013. Accessed: 2025-02-20.

[11] ROBOTIS. *TurtleBot3*, 2011. Accessed: 2025-02-21.

[12] Patrick Slade and et al. Multimodal sensing and intuitive steering assistance improve navigation and mobility for people with impaired vision. *Science Robotics*, 6(59):eabg6594, 2021.

[13] Adept Technology. *Pioneer 3D-x manual*, 2011. Accessed: 2025-02-21.

[14] unknown. *Analog device ADXL345 user guide*, unknown. Accessed: 2025-03-10.

[15] unknown. *DG01D-E motor with encoder*, unknown. Accessed: 2025-03-03.