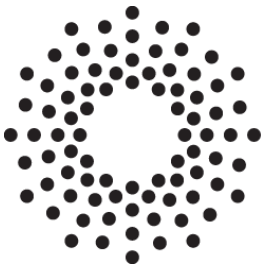

LONDON METROPOLITAN UNIVERSITY

CT7159 Sensors, Actuators and Control

Spring 2024-2025

2WD Mobile Robot Car

Logbook

Instructor	Instructor	Instructor
Ignacio García Zuazola	Dion Mariyanayagam	Marcio Lacerda

Student Name	University ID
Jorge Viñuela Perez	23038621

Table of Contents

Workshop 3: Actuators 5

Task 1: Construct a 2WD mobile robot car 5

Task 2: Implementation of DC motors with an H-Bridge Driver 5

Task 2a: Pulse Width Modulation 7

Task 3: Implementation of Servos 9

Task 3a: Servo fine control 9

Task 3b: Servo full rotation 10

Task 4: Implement the ultrasonic sensor 11

Task 5: Measure the accuracy of the ultrasonic sensor distance 13

Task 6: Filtering the ultrasonic sensor readings 13

Task 7: Design an obstacle avoidance car with variable speed 14

Workshop 4: Sensors 19

Task 1: Creating a Library 19

Task 2: Reading analogue pins 20

Task 3: Designing a Library for the BMP280 Sensor 21

Task 4: Implementing Filtering for BMP280 Sensor 22

Task 5: Temperature alarm 23

Task 6: Interfacing the MPU-6050 Sensor 23

Task 7: Implementing Filtering for MPU-6050 Sensor 24

Task 8: Robot Car Manipulation using MPU-6050 Sensor 27

Workshop 5: PID 29

Task 1: Creating an RC Filter Circuit 29

Task 2: PID on a RC Filter Circuit 30

Task 3: Implement PID with the MPU6050 on a Mobile Robot 30

Conclusion 33

Table of Figures

Figure 21. 2WD robot chassis with mounted motors, H-Bridge driver wiring, and battery pack and switch installation 5
Figure 22. H-Bridge motor control code for forward motion 6
Figure 23. Backward Motion Demonstration 6
Figure 24. PWM-based motor speed and direction control logic 7
Figure 25. PWM-based Speed ramp limit, motor stop, and automated direction change logic 8
Figure 26. PWM Implementation Demonstration 8
Figure 27. Servo Control Web Interface 9
Figure 28. Servo control for 90° and 180° positioning with PWM signal generation 9
Figure 29. Servo 90º and 180º Degrees Positioning Demonstration 10
Figure 30. Servo control Full Rotation with PWM signal generation 10
Figure 31. Servo Full Rotation Demonstration 11
Figure 32. Distance Measurement with Ultrasonic Sensor Implementation 12
Figure 33. Distance Measurement Demonstration 12
Figure 34. Distance Measurement Accuracy Test 13
Figure 35. Moving Average Filter Implementation for Distance Measurement 14
Figure 36. Raw and filtered ultrasonic distance data using a moving average filter 14
Figure 37. Obstacle Avoidance Car Web Interface 15
Figure 38. Emergency Brake Implementation 15
Figure 39. Motor Movement Logic Implementation 16
Figure 40. 2WD Robot Car Remote Movement Demonstration 16
Figure 41. Distance Measurement Implementation 17
Figure 42. Auto Mode Motor Implementation 17
Figure 43. Avoidance Car Demonstration 17
Figure 44. MyLibrary Header File (MyLibrary.h) 19
Figure 45. MyLibrary Source File (MyLibrary.cpp) 19
Figure 46. Implementation and Output of MyLibrary Example 20
Figure 47. Analogue Pin Reading Implementation 20
Figure 48. Analogue Pin Reading Example 20
Figure 49. BMP280 Configuration Setup Implementation 21
Figure 50. BMP280 Readings Printing Implementation 22
Figure 51. BMP280 Readings Output 22
Figure 52. Moving Average Filter Algorithm Applied to Temperature Readings 22
Figure 53. Raw and Filtered Temperature Readings 23

Figure 54. BMP280 Temperature Alarm Implementation 23
Figure 55. MPU6050 Wiring Connections 24
Figure 56. MPU-6050 Filtering Setup Function Implementation 24
Figure 57. MPU-6050 Readings Implementation 25
Figure 58. MPU-6050 Moving Average Filter 25
Figure 59. MPU-6050 Cumulative Average Filter 25
Figure 60. Pitch Raw Data And Filtered Output 26
Figure 61. Roll Raw Data And Filtered Output 26
Figure 62. MPU-6050 Real-time Readings 26
Figure 63. AccelerometerRead Function Implementation 27
Figure 64. Inclination-based motor speed control logic Implementation 27
Figure 65. Robot Car Manipulation with MPU-6050 Web Interface 28
Figure 66. Robot Car Manipulation with MPU-6050 Demonstration 28
Figure 67. RC Filter Circuit 29
Figure 68. Breadboard Mounting onto the 2WD Robot Car 29
Figure 69. PID on a RC Filter Output 30
Figure 70. Serial Output Stabilized at 80.00 30
Figure 71. PID Control Loop Function 31
Figure 72. GetAngleX function for MPU-6050 Sensor 31
Figure 73. Motor Speed And Direction Control Based On PID Output 32
Figure 74. PID with MPU-6050 on the 2WD Robot Car Demonstration 32

Workshop 3: Actuators

Task 1: Construct a 2WD mobile robot car

The first step of the project was to construct a two-wheeled drive (2WD) mobile robot car using the provided kit. The general instructions were to assemble the car, connect the motors to the motor driver, and then connect the motor driver to the microcontroller.

During the assembly process, specialized parts to mount both the ESP32 microcontroller and the L9110 motor driver neatly onto the new base were designed and 3D printed. After printing all the necessary parts, I proceeded to screw everything securely onto the base. Once the structure was stable, I mounted the ESP32 and the L9110 motor driver into their designated positions and connected them accordingly. A picture was taken at this stage to document the successful assembly and wiring.

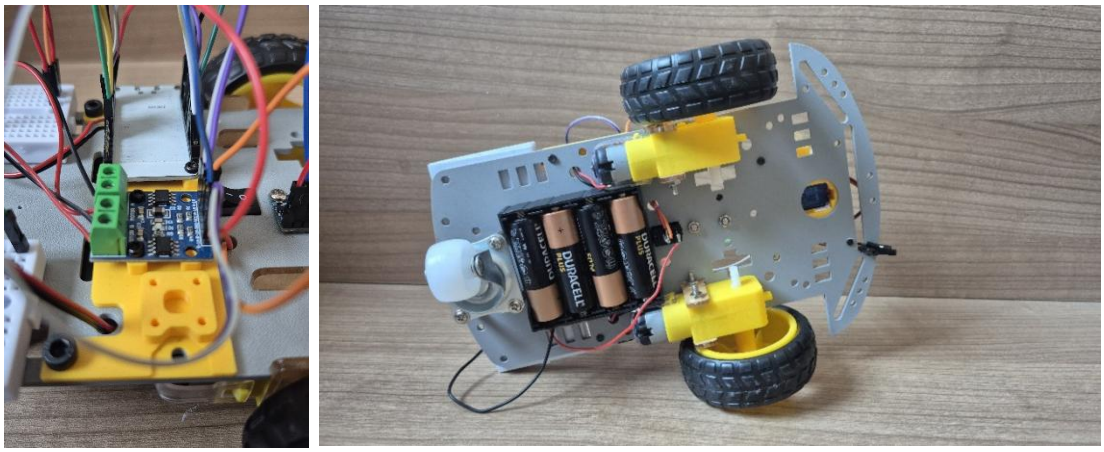

Figure 1. 2WD robot chassis with mounted motors, H-Bridge driver wiring, and battery pack and switch installation

Following that, the motors were installed into their correct slots on the base, a switch was positioned in an easily accessible location for convenient operation and the battery cables were soldered to the switch to allow for controlled power delivery to the system. A second picture was taken to capture the completed motor and power setup. With all the mechanical and electrical components properly mounted and connected, the basic construction phase of the 2WD mobile robot car was completed successfully.

Task 2: Implementation of DC motors with an H-Bridge Driver

This task was carried out to enable the basic directional movements of the 2WD mobile robot car: forward, backward, left, and right. For this purpose, the motors were connected to the H-Bridge driver outputs, with each motor responsible for driving one of the wheels. The right wheel motor was connected to the driver outputs B1A (GPIO 4) and B2B (GPIO 5), while the left wheel motor was connected to A1A (GPIO 32) and A2B (GPIO 33). These GPIO pins were configured as outputs in the microcontroller's setup routine to control the direction of the current flow through the motors.

```
// RIGHT WHEEL
int B1A = 4; // GPIO Digital Pin 4
int B2B = 5; // GPIO Digital Pin 5

// LEFT WHEEL
int A1A = 32; // GPIO Digital Pin 32
int A2B = 33; // GPIO Digital Pin 33

void setup() {
  pinMode(B1A, OUTPUT);
  pinMode(B2B, OUTPUT);
  pinMode(A1A, OUTPUT);
  pinMode(A2B, OUTPUT);
}

void loop() {
  digitalWrite(B1A, LOW);
  digitalWrite(B2B, HIGH);
  digitalWrite(A1A, LOW);
  digitalWrite(A2B, HIGH);
}

// LOW HIGH HIGH LOW --> CLOCKWISE ROTATION
// LOW HIGH LOW HIGH --> FORWARD
// HIGH LOW HIGH LOW --> BACKWARD
// HIGH LOW LOW HIGH --> ANTI-CLOCKWISE ROTATION
```

Figure 2. H-Bridge motor control code for forward motion

In the implemented code, the logic levels applied to these pins determined the rotation direction of each motor, enabling the desired manoeuvres. For instance, to move backward, both motors were driven with HIGH-LOW combinations, causing both wheels to rotate in the same direction.

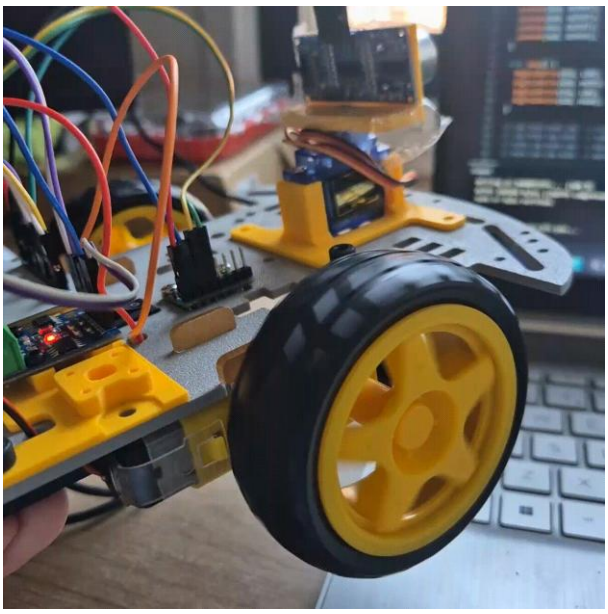

Figure 3. Backward Motion Demonstration

One of the disadvantages found of this code was the lack of control. As it executed continuously and without delay, the car began moving immediately and indefinitely upon uploading the program, which made it difficult to perform precise testing or stop the car without physically disconnecting power. This led to create a control website in future stages of the project.

Task 2a: Pulse Width Modulation

To achieve variable speed control of the DC motors, Pulse Width Modulation (PWM) was implemented. This technique allowed the adjustment of motor speeds while maintaining consistent supply voltage.

The process began by defining the motor pins and initializing the PWM_SPEED variable with a value of 140, as it is the minimum value that the motors require for them to rotate. A direction flag, dir, was also declared to alternate between forward (0) and backward (1) motion.

```
22  void loop() {
23
24    Serial.print("PWM_SPEED: ");
25    Serial.print(PWM_SPEED);
26    Serial.print(" | Direction: ");
27    Serial.println(dir);
28
29    if (dir == 0) {
30        analogWrite(MOTOR_A1, PWM_SPEED);
31        analogWrite(MOTOR_A2, 0);
32        analogWrite(MOTOR_B1, PWM_SPEED);
33        analogWrite(MOTOR_B2, 0);
34    } else {
35        analogWrite(MOTOR_A1, 0);
36        analogWrite(MOTOR_A2, PWM_SPEED);
37        analogWrite(MOTOR_B1, 0);
38        analogWrite(MOTOR_B2, PWM_SPEED);
39    }
```

Figure 4. PWM-based motor speed and direction control logic

In the loop() function, as shown in Figure 24, the current PWM speed and direction were printed to the serial monitor for observation. Based on the value of dir, the appropriate motor control signals were applied:

- For forward motion, analogWrite() applied PWM_SPEED to MOTOR_A1 and MOTOR_B1, while MOTOR_A2 and MOTOR_B2 were set to 0.
- For backward motion, the opposite pins received the PWM signal, reversing the motor direction.

For testing purposes, a program where the wheels accelerate in different directions was implemented. As shown in the Figure 25, once PWM_SPEED exceeded 255, the following sequence occurred:

1. **PWM_SPEED reset**: The speed value was set to 0.
2. **Motors stopped**: All motor pins were set to 0 to ensure a complete stop.
3. **Delay added**: A 500 ms pause allowed time to observe the stop.
4. **Direction switched**: The value of dir was toggled to reverse the direction.
5. **Status printed**: A message indicating a direction change was output to the serial monitor.
6. **PWM_SPEED reinitialized**: The speed was reset to 140 to start a new speed ramp in the opposite direction.

```
  PWM_SPEED += 10;

  // Speed limit, change of direction
  if (PWM_SPEED >= 260) {

    PWM_SPEED = 0;

    analogWrite(MOTOR_B1, 0);
    analogWrite(MOTOR_B2, 0);
    analogWrite(MOTOR_A1, 0);
    analogWrite(MOTOR_A2, 0);

    Serial.println("Stopping motors...");
    delay(500);

    // Direction change
    if (dir == 0) {
      dir = 1;
    } else {
      dir = 0;
    }

    Serial.println("Change of direction...");
    delay(500);

    PWM_SPEED = 140;
  }

  delay(500);
}
```

Figure 5. PWM-based Speed ramp limit, motor stop, and automated direction change logic

This cycle continuously repeated, causing the robot to alternate between forward and backward motion with gradually increasing speed, followed by a brief stop and reversal, as shown in the following GIF.

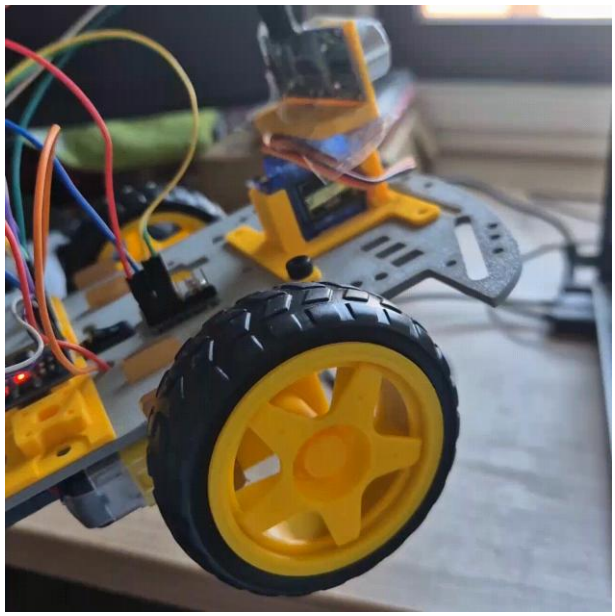

Figure 6. PWM Implementation Demonstration

Although a PWM value of 140 was determined to be the minimum required to initiate wheel movement, it was observed that, at times, the wheels would remain stuck. This inconsistency was likely due to factors such as motor friction, slight variations in motor performance, or insufficient torque at low speeds, causing the motors to occasionally fail to overcome static friction.

Task 3: Implementation of Servos

For this task, a servo motor was mounted at the front of the 2WD robot car to enable precise control of mechanical movement. It was physically secured to the chassis, ensuring stability during operation. Electrical connections were then established: the servo's power pin (5V) was connected to the 5V output of the microcontroller, the ground (GND) pin was connected to the common ground, and the signal (SIG) pin was connected to GPIO 2.

Task 3a: Servo fine control

Following the successful mounting and wiring of the servo, the next objective was to implement control allowing the servo to rotate precisely 90 degrees and then 180 degrees.

To facilitate easier and more reliable command transmission, a web interface was created using the ESP32's Wi-Fi capabilities. Through this hotspot, a web server hosted an HTML page that included control buttons. These buttons allowed remote commands to be sent to the microcontroller to execute the desired servo motions without requiring physical connection or manual code changes.

Figure 7. Servo Control Web Interface

When the "Move servo" button was pressed, the servo was commanded to rotate to 90 degrees, pause for 500 ms, and then proceed to 180 degrees. This sequence was programmed using the moveServo() function, which translated the desired angle into the appropriate pulse width modulation (PWM) signal. The function maintained the required pulse width for a short duration to ensure the servo reached and held the target position accurately.

```
75  void handleMove() {
76      moveServo(90);
77      delay(500);
78      moveServo(180);
79      server.send(200, "text/plain", "");
80  }
81
82  void moveServo(int angle) {
83      int pulseWidth = map(angle, 0, 180, 500, 2500); // 500-2500 µs
84      unsigned long startTime = millis();
85      while (millis() - startTime < 500) {
86          digitalWrite(servoPin, HIGH);
87          delayMicroseconds(pulseWidth);
88          digitalWrite(servoPin, LOW);
89          delay(20); //  ~50Hz
90      }
91      currentAngle = angle;
92      Serial.println("Current Angle: " + String(currentAngle) + "°");
93  }
```

Figure 8. Servo control for 90° and 180° positioning with PWM signal generation

The use of the web interface significantly improved the testing and control process, enabling quick and repeatable servo movements without modifying the code or relying on physical inputs.

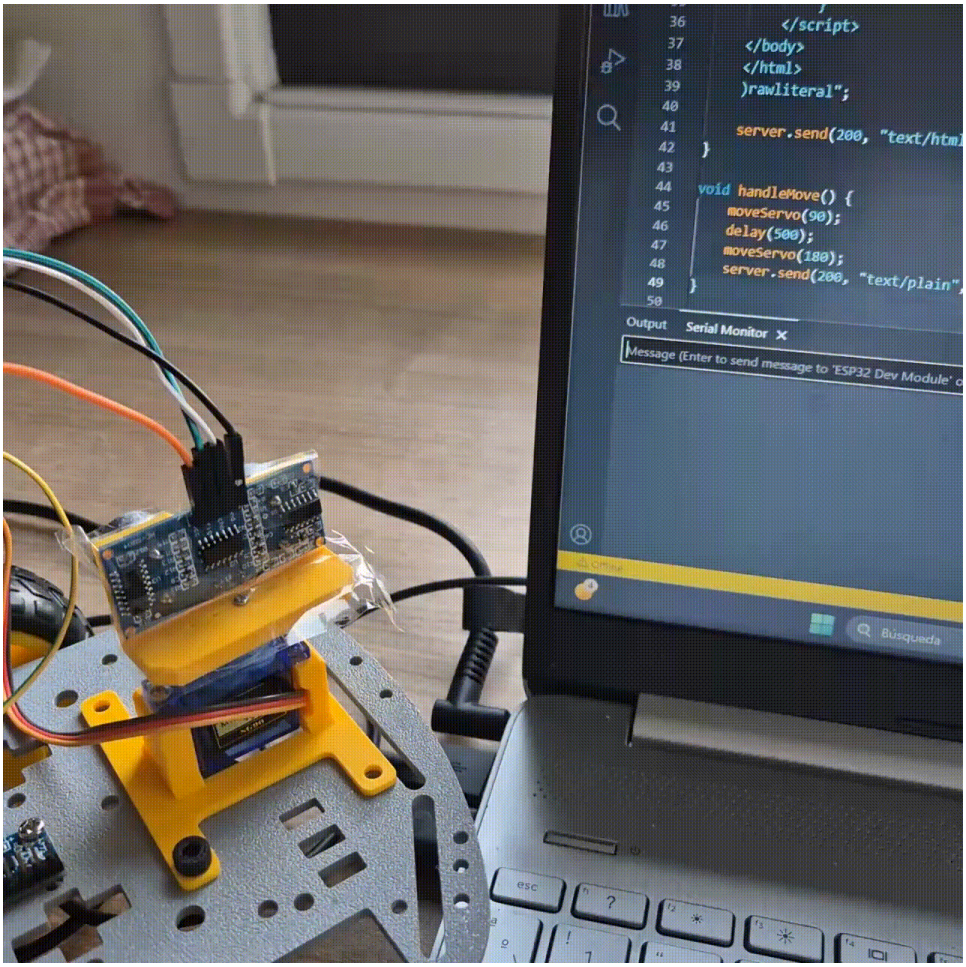

Figure 9. Servo 90º and 180º Degrees Positioning Demonstration

Task 3b: Servo full rotation

Building upon the previous task, the objective was to implement continuous incremental movement of the servo, achieving a full 180-degree sweep in small steps. This would allow smoother rotation and better control compared to abrupt position changes.

```
51  void handleFullRotation() {
52      for (int angle = 0; angle <= 180; angle += 9) {
53          moveServo(angle);
54          delay(20); // 100ms entre cada movimiento
55      }
56      server.send(200, "text/plain", "Rotación completa realizada.");
57  }
```

Figure 10. Servo control Full Rotation with PWM signal generation

A new function, triggered by the "Full Rotation" button on the previously created web interface, was programmed to incrementally move the servo from 0° to 180° in 9-degree steps. This relatively large step size was intentionally chosen to increase the speed of the sweep for demonstration purposes, allowing the full rotation to be completed quickly during testing. However, in practical applications where precise control is required, the sweep would typically be executed in 1-degree steps, by simply changing the increment

value in the loop from 9 to 1. After each increment, a 20 ms delay was applied to maintain an approximate 50 Hz PWM frequency, ensuring that the servo had sufficient time to reach each new position before advancing.

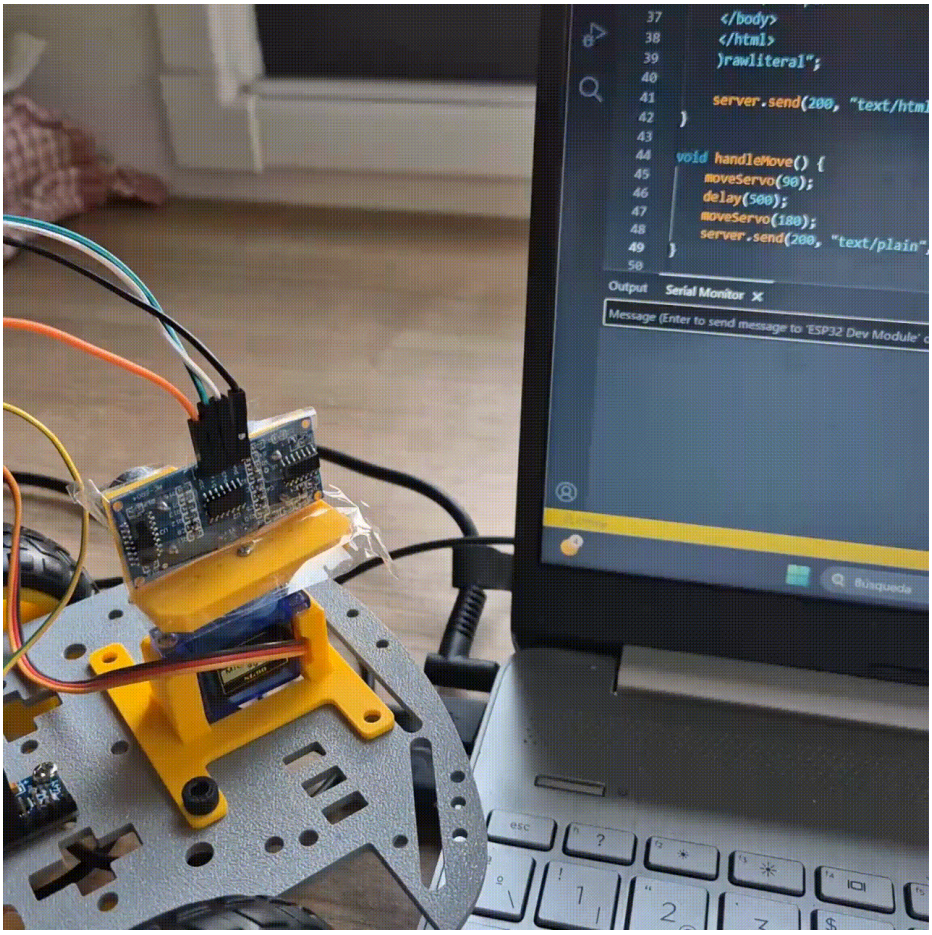

Figure 11. Servo Full Rotation Demonstration

This method allowed for a visibly smooth and controlled motion across the full rotation range. The web interface continued to provide a convenient way to send commands and observe the servo's performance without modifying the code or using manual hardware controls.

Task 4: Implement the ultrasonic sensor

The ultrasonic sensor was mounted securely on top of the previously installed servo at the front of the robot. This mounting approach was chosen to allow the servo to rotate the ultrasonic sensor in future tasks, enabling distance measurements at different angles for obstacle detection or mapping applications. The sensor's electrical connections were then established as follows:

- VCC (5V) connected to the microcontroller's 5V output.
- GND connected to the common ground.
- Trigger (Trig) connected to GPIO 19.
- Echo connected to GPIO 23.

```
void loop() {
  // Send a pulse to the trig pin to trigger the ultrasonic sensor
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // Measure the duration of the echo pulse
  long duration = pulseIn(echoPin, HIGH);

  // Calculate the distance in centimeters
  distance = duration / 58;

  // Print the distance to the serial monitor
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");
  // Wait for 500 milliseconds before taking another measurement
  delay(500);
}
```

Figure 12. Distance Measurement with Ultrasonic Sensor Implementation

In the implemented code, the Trig pin was configured as an output and the Echo pin as an input. During each cycle of the loop(), the following sequence was executed:

1. The Trig pin was set LOW briefly, then HIGH for 10 microseconds to send an ultrasonic pulse.
2. The time taken for the echo pulse to return was measured using the pulseIn() function.
3. The distance to an object was calculated by dividing the duration by 58, a value derived from the speed of sound, simplifying the calculation to return the distance in centimetres.
4. The measured distance was printed to the Serial Monitor.
5. A 500 ms delay was applied before the next measurement to allow easy observation and prevent excessive readings.

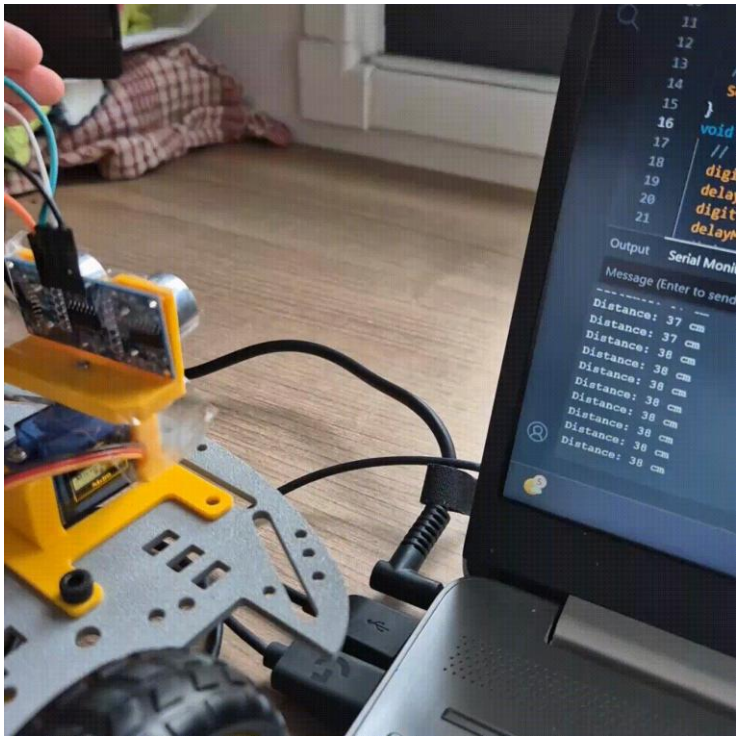

Figure 13. Distance Measurement Demonstration

Task 5: Measure the accuracy of the ultrasonic sensor distance

The ultrasonic sensor was tested using a ruler to verify the accuracy of its distance measurements. Multiple readings were taken at various known distances, and the results consistently matched the expected values within an acceptable margin of error. This confirmed that the sensor was functioning correctly and no calibration or code adjustments were necessary. A figure has been included to illustrate the accuracy of the measurements obtained during the test.

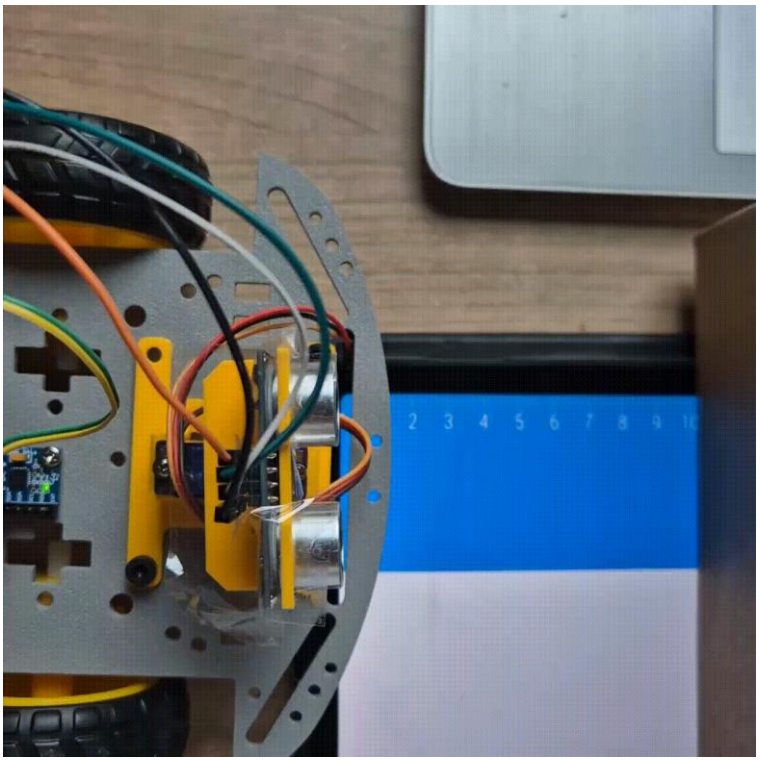

Figure 14. Distance Measurement Accuracy Test

Task 6: Filtering the ultrasonic sensor readings

To improve the reliability of the distance measurements provided by the ultrasonic sensor, a moving average filter was implemented to smooth out fluctuations and reduce the impact of noise. This was based on filtering techniques previously studied in the Filtering Systems workshop.

In the updated code (Figure 35), an array was used to store the 10 most recent distance measurements. Each new reading replaced the oldest value in the array, maintaining a continuous stream of data for averaging. After updating the array, the sum of all stored values was calculated and divided by the number of readings to compute the moving average. The filtered value represented a smoothed version of the raw distance data, reducing the effect of sudden spikes or brief inaccuracies often caused by sensor noise or environmental factors.

```
// Measure the duration of the echo pulse
long duration = pulseIn(echoPin, HIGH);

// Calculate the distance in centimeters
distance = duration / 58;

total = total - readings[readIndex];   // Delete oldest reading
readings[readIndex] = distance;         // Save new reading
total = total + readings[readIndex];    // Add new reading
readIndex = (readIndex + 1) % numReadings;

average = total / numReadings; // Average

// Print the distance to the serial monitor
Serial.print("Raw(cm):");
Serial.print(distance);
Serial.print(",");

Serial.print("Filtered(cm):");
Serial.println(average);
```

Figure 15. Moving Average Filter Implementation for Distance Measurement

Both the raw and filtered distance values were sent to the Serial Monitor, allowing visualization using the Serial Plotter, as shown in the graph below. The blue line represents the raw distance data, which fluctuates significantly due to noise and brief measurement inconsistencies. The orange line shows the filtered data, which follows the general trend of the raw data but with reduced variation, providing a smoother and more stable output. This demonstrates the effectiveness of the filter in reducing noise and improving the consistency of distance measurements, especially during rapid changes or when the sensor momentarily struggles to detect accurate distances.

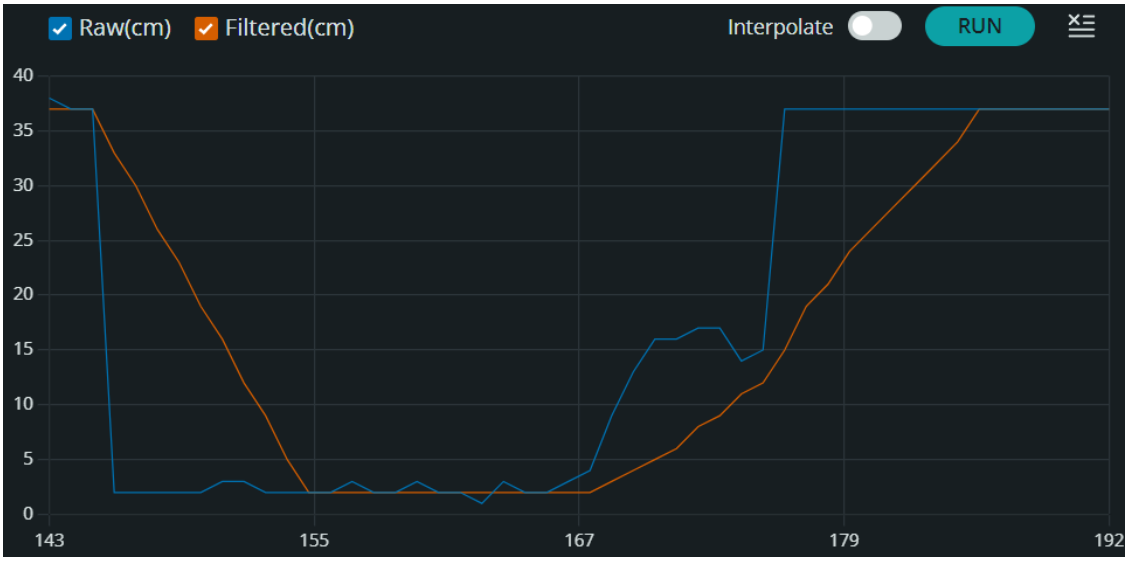

Figure 16. Raw and filtered ultrasonic distance data using a moving average filter

Task 7: Design an obstacle avoidance car with variable speed

For this task, all previously developed subsystems were fully integrated to create a functional obstacle avoidance robot with both manual and automatic control, using PWM for motor speed, ultrasonic distance sensing with filtering, and a web interface for user commands and distance monitoring.

Web Interface and Wi-Fi Communication

A Wi-Fi hotspot was created by the ESP32 to serve an HTML web page, accessible by any Wi-Fi-enabled device. The web page provided:

- Manual control buttons (W, A, S, D) for moving the car forward, backward, rotate left, or rotate right.
- A real time distance display showing the latest filtered distance from the ultrasonic sensor.
- Buttons to activate and deactivate Auto Mode, where the robot operated autonomously.

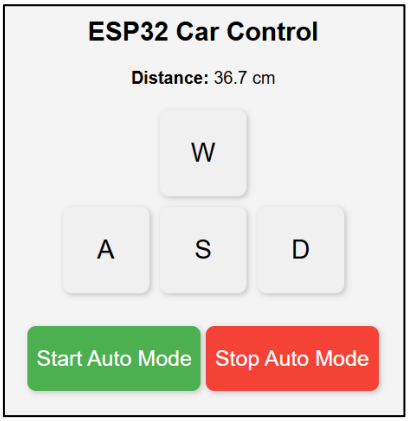

Figure 17. Obstacle Avoidance Car Web Interface

Creating the web interface and ensuring its stability posed a significant challenge. Initially, the system would crash when receiving constant or overlapping commands. For instance, when multiple directional buttons were held down or pressed repeatedly, the car would get stuck in the motion. To overcome this, and emergency brake was implemented.

```
191     // Freno de seguridad
192     if (!modoAuto && millis() - lastCommandTime > timeoutInterval) {
193         stopMotors();
194     }
```

Figure 18. Emergency Brake Implementation

During manual control mode, the code continuously monitored the time elapsed since the last valid movement command was received. If no command had been received for more than 500 milliseconds, the motors were stopped. This emergency brake ensured that if the user accidentally released the controls, lost connection, or the web interface failed, the robot would immediately stop instead of continuing to move uncontrolled.

PWM Motor Control

The motors were controlled using PWM signals. Three speed levels were defined:

- **PWM_SPEED (250)** for full speed forward movement.
- **PWM_SPEED_GIRO (140)** for turning manoeuvres, where one motor moved slower to allow the car to pivot.
- **PWM_AUTO (140)** for automatic forward movement at a reduced speed during Auto Mode.

```
218 void applyMotorMovement() {
219     if (movingForward) {
220         analogWrite(MOTOR_A1, PWM_SPEED);
221         analogWrite(MOTOR_A2, 0);
222         analogWrite(MOTOR_B1, PWM_SPEED);
223         analogWrite(MOTOR_B2, 0);
224     } else if (movingBackward) {
225         analogWrite(MOTOR_A1, 0);
226         analogWrite(MOTOR_A2, PWM_SPEED);
227         analogWrite(MOTOR_B1, 0);
228         analogWrite(MOTOR_B2, PWM_SPEED);
229     } else if (turningLeft) {
230         analogWrite(MOTOR_A1, 0);
231         analogWrite(MOTOR_A2, PWM_SPEED_GIRO);
232         analogWrite(MOTOR_B1, PWM_SPEED_GIRO);
233         analogWrite(MOTOR_B2, 0);
234     } else if (turningRight) {
235         analogWrite(MOTOR_A1, PWM_SPEED_GIRO);
236         analogWrite(MOTOR_A2, 0);
237         analogWrite(MOTOR_B1, 0);
238         analogWrite(MOTOR_B2, PWM_SPEED_GIRO);
239     } else {
240         stopMotors();
241     }
242 }
```

Figure 19. Motor Movement Logic Implementation

Commands from the web interface updated the movement state flags (movingForward, movingBackward, turningLeft, turningRight) which then triggered the appropriate PWM outputs to the motors. The motor control logic allowed for smooth forward and backward movement as well as turning combinations.

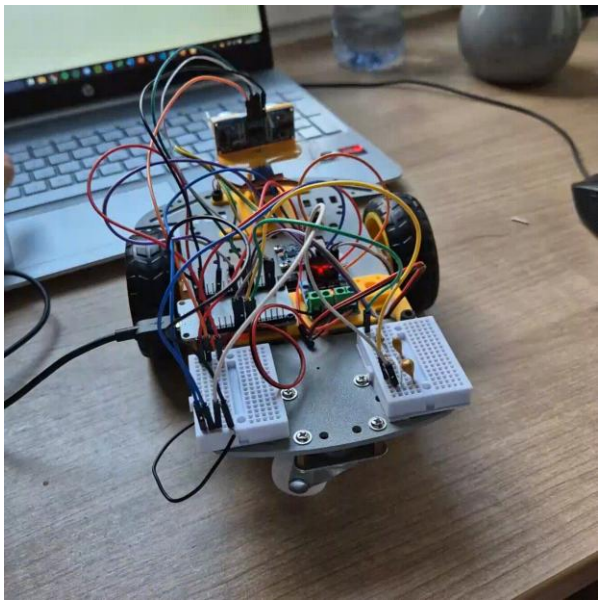

Figure 20. 2WD Robot Car Remote Movement Demonstration

Ultrasonic Distance Measurement and Filtering

An HC-SR04 ultrasonic sensor, already mounted on the servo, was used to measure the distance ahead. Distance measurements were taken every 400 ms and filtered using an exponential moving average filter with a coefficient α = 0.4. This filtering smoothed out the rapid fluctuations common to ultrasonic sensors and provided more reliable data for the obstacle avoidance algorithm, as seen in the previous tasks. A global variable (distanciaActual) was updated with the latest raw and filtered distance values.

```
if (millis() - lastDistanceCheck > 400) {
    distanciaActual = measureDistance();
    lastDistanceCheck = millis();
}
```

Figure 21. Distance Measurement Implementation

Auto Mode Behaviour

When Auto Mode was activated, the following logic was applied:

- If the filtered distance was 10 cm or greater, the car continued moving forward at the preset PWM_AUTO speed.
- If the distance dropped below 10 cm, the car immediately stopped to avoid collision.

```
179     // Auto Mode
180     if (modoAuto) {
181         if (distanciaActual >= 10.0) {
182             analogWrite(MOTOR_A1, PWM_AUTO);
183             analogWrite(MOTOR_A2, 0);
184             analogWrite(MOTOR_B1, PWM_AUTO);
185             analogWrite(MOTOR_B2, 0);
186         } else {
187             stopMotors();
188         }
189     }
```

Figure 22. Auto Mode Motor Implementation

Originally, the control logic was intended to implement proportional speed control, where the car's speed would decrease smoothly from 100% PWM at 10 cm down to 0% PWM at 5 cm as an obstacle approached, as asked. However, during testing, it was observed that due to the minimum operational speed of the motors and the physical inertia of the car, even when the motors were commanded to stop, the car required approximately 5 cm of distance to fully decelerate. As a result, by the time the ultrasonic sensor detected an obstacle at 10 cm and the speed began decreasing, the car would naturally come to a complete stop by around 5 cm without needing a gradual speed reduction.

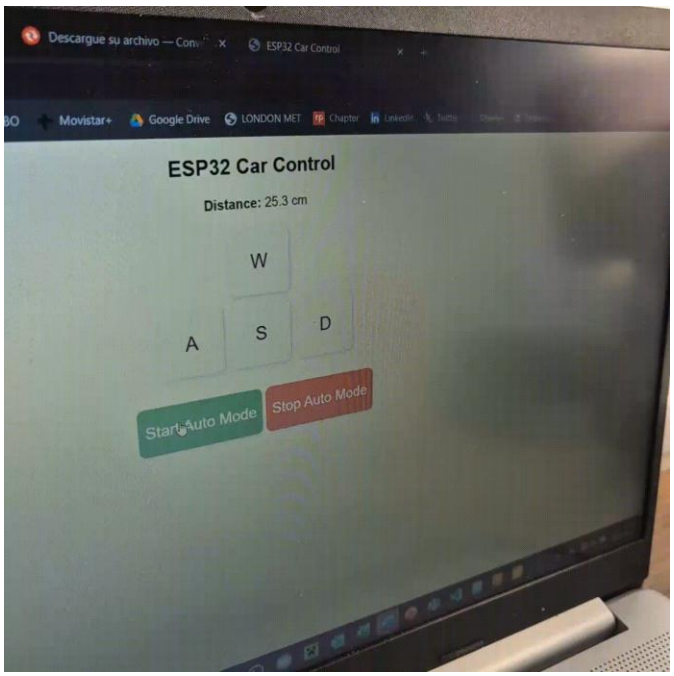

Figure 23. Avoidance Car Demonstration

Because of this behaviour, implementing the mapping function for proportional speed control was deemed unnecessary at this stage. Instead, a simpler and more reliable control was applied: the car moved at a constant speed until an object was detected within 10 cm, at which point the motors were immediately stopped. This simplified approach ensured effective obstacle avoidance while maintaining the safety and responsiveness of the system. Additionally, this decision provided a clear foundation for future improvements, where proportional control can be reintroduced if finer speed management becomes necessary or if the braking distance can be reduced mechanically.

Mechanical Assembly and Challenges

Throughout this stage, several hardware challenges arose:

- **Connection Rearrangement:** As additional components (motors, servo, ultrasonic sensor, and ESP32) were connected, the limited GPIO availability and physical layout required multiple iterations of connection rearrangement to avoid conflicts and ensure reliable wiring.
- **Mechanical Impact:** During manual movement tests, the car struck a bump, causing the battery pack to detach, which highlighted the need for better mechanical stability. The battery pack was remounted, and improvements were made to the motor solder joints and power switch wiring to prevent similar failures during future use.
- **Wi-Fi Server Crashes:** The ESP32 web server would initially crash or freeze under repeated or fast command inputs. Adjustments were made to the server's event handling to resolve these issues.

Results

The completed system demonstrated reliable manual control and fully autonomous forward motion with obstacle detection and stopping behaviour. The web interface allowed intuitive user interaction, while the filtered ultrasonic sensor data ensured stable obstacle detection. The experience also provided valuable insights into real-world challenges related to combining mechanical, electronic, and software components in a mobile robot platform.

Workshop 4: Sensors

Task 1: Creating a Library

To begin the workshop, a custom Arduino library was created to perform the simple arithmetic operation of adding two numbers. This task introduced the structure and syntax required for developing reusable and modular code components for Arduino-based projects. The following steps were followed:

1. **Library Folder Creation**: A new folder was created within the Arduino library directory and named MyLibrary.
2. **Header File (MyLibrary.h)**: The header file defined the class MyLibrary, inside which, a constructor and a public method addNumbers(int num1, int num2) were declared. A private variable _number was also declared, though it was not directly used in this example but reserved for possible future uses.

```
#ifndef MyLibrary_h
#define MyLibrary_h

#include <Arduino.h>

class MyLibrary {
  public:
    MyLibrary();
    int addNumbers(int num1, int num2);
  private:
    int _number;
};

#endif
```

Figure 24. MyLibrary Header File (MyLibrary.h)

3. **Source File (MyLibrary.cpp)**: The source file implemented the constructor and the addNumbers function, which returned the sum of the two integer arguments provided when called.

```
#include "MyLibrary.h"

MyLibrary::MyLibrary() {
  _number = 0;
}

int MyLibrary::addNumbers(int num1, int num2) {
  return num1 + num2;
}
```

Figure 25. MyLibrary Source File (MyLibrary.cpp)

After creating the library, a simple Arduino sketch was written to demonstrate its functionality by adding 350 and 80. The library was included, and an instance of the MyLibrary class was created. The addNumbers() method was used to compute the sum of 350 and 80, and the result was printed to the Serial Monitor. The program correctly displayed the result, confirming the successful implementation of the custom library.

```cpp
#include <MyLibrary.h>

MyLibrary myLib;

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

  int result = myLib.addNumbers(350, 80);

  Serial.print("The result of 350 + 80 is: ");
  Serial.println(result);
}

void loop() {

}
```

```
The result of 350 + 80 is: 430
```

Figure 26. Implementation and Output of MyLibrary Example

Task 2: Reading analogue pins

For this task, the objective was to read an analogue input and apply the previously created addNumbers function from the custom library to process the data. The A0 analogue pin (GPIO 36 on the ESP32) was used for reading the input signal.

```cpp
#include "MyLibrary.h"

MyLibrary myLib;

void setup() {
  Serial.begin(115200);
  delay(1000);
}

void loop() {
  int analogValue = analogRead(36);
  int result = myLib.addNumbers(analogValue, 100);

  Serial.println(result);

  delay(200);
}
```

Figure 27. Analogue Pin Reading Implementation

The value obtained from pin A0 was added to 100 using the addNumbers() method from MyLibrary, explained in the previous task. The result was printed to the Serial Monitor and could be visualized using the Serial Plotter for a clearer representation of changes over time.

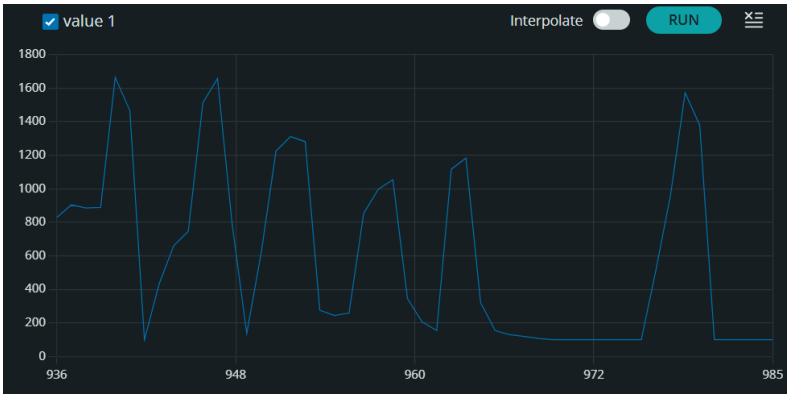

Figure 28. Analogue Pin Reading Example

As a simple demonstration signal, the analogue pin was touched with a finger or left untouched to create variable readings. This caused noticeable changes in the plotted values, successfully demonstrating the acquisition and processing of analog signals using the custom library.

Task 3: Designing a Library for the BMP280 Sensor

The objective of this task was to implement a functional interface with the BMP280 environmental sensor, which can measure temperature, pressure, and approximate altitude. Rather than creating a completely new library from scratch, which would have required rewriting complex calibration and compensation routines, the Adafruit_BMP280 library was employed.

The Adafruit_BMP280 library includes all the essential functions needed to communicate with the BMP280 sensor, retrieve raw data, and convert it into meaningful measurements by applying the calibration coefficients stored inside the sensor itself. In the main program, the BMP280 sensor was connected to the ESP32 microcontroller via I2C communication.

```
14  void setup() {
15    Serial.begin(115200);
16    while (!Serial) delay(100);
17
18    Serial.println(F("BMP280 test"));
19
20    unsigned status;
21    status = bmp.begin(0x76);
22
23    if (!status) {
24      Serial.println(F("Could not find a valid BMP280 sensor, check wiring or try a different address!"));
25      Serial.print("SensorID was: 0x"); Serial.println(bmp.sensorID(), 16);
26      Serial.println("ID of 0xFF probably means a bad address, a BMP 180 or BMP 085");
27      Serial.println("ID of 0x56-0x58 represents a BMP 280,");
28      Serial.println("ID of 0x60 represents a BME 280.");
29      Serial.println("ID of 0x61 represents a BME 680.");
30      while (1) delay(10);
31    }
32
33
34    bmp.setSampling(
35      Adafruit_BMP280::MODE_NORMAL,
36      Adafruit_BMP280::SAMPLING_X2,
37      Adafruit_BMP280::SAMPLING_X16,
38      Adafruit_BMP280::FILTER_X16,
39      Adafruit_BMP280::STANDBY_MS_500
40    );
41  }
```

Figure 29. BMP280 Configuration Setup Implementation

The program then attempted to initialize the BMP280 sensor using the bmp.begin() method. If the sensor was not detected, the program printed detailed error messages indicating possible causes, such as incorrect wiring or an invalid sensor address, and halted further execution. This diagnostic feedback was helpful for ensuring reliable sensor connections.

The selected configuration for the sensor included normal mode operation, moderate oversampling for temperature and high oversampling for pressure to enhance measurement accuracy, and strong filtering to minimize the effects of rapid environmental changes or electrical noise. The standby time was set to 500 milliseconds, balancing measurement responsiveness with power efficiency.

```
43  void loop() {
44    Serial.print(F("Temperature = "));
45    Serial.print(bmp.readTemperature());
46    Serial.println(" *C");
47
48    Serial.print(F("Pressure = "));
49    Serial.print(bmp.readPressure());
50    Serial.println(" Pa");
51
52    Serial.print(F("Approx altitude = "));
53    Serial.print(bmp.readAltitude(1015));
54    Serial.println(" m");
55
56    Serial.println();
57    delay(2000);
58  }
```

Figure 30. BMP280 Readings Printing Implementation

During each cycle of the loop() function, the program read the temperature, pressure, and calculated altitude values from the BMP280. These values were printed to the Serial Monitor at two-second intervals. The temperature was reported in degrees Celsius, the pressure in Pascals, and the altitude in meters. The altitude calculation used a sea-level pressure reference of 1015 hPa, which was an appropriate value for the testing environment (London). The consistent and accurate output confirmed that the BMP280 sensor was correctly interfaced and that the Adafruit library provided reliable data processing.

```
Temperature = 20.40 *C
Pressure = 101100.00 Pa
Approx altitude = 11.60 m
```

Figure 31. BMP280 Readings Output

This task successfully demonstrated not only the use of an external sensor library but also the ability to configure the sensor, acquire environmental data, and display it in a user-friendly manner. The groundwork was now prepared for applying filtering to the BMP280 data, which would be the focus of the next task.

Task 4: Implementing Filtering for BMP280 Sensor

For this task, a Moving Average Filter same as the one used for filtering the distance measured by the HC-SR04 ultrasonic sensor was implemented to smooth the temperature readings.

```
23  // --- Moving Average Filter ---
24  total = total - readings[readIndex];   // Remove the oldest reading
25  readings[readIndex] = rawTemperature;  // Store the new reading
26  total = total + readings[readIndex];   // Add the new reading
27  readIndex = (readIndex + 1) % numReadings; // Move to next index
28
29  average = total / numReadings; // Calculate the average
30
31  // --- Print both raw and filtered values ---
32  Serial.print("Raw(°C):");
33  Serial.print(rawTemperature);
34  Serial.print(",");
35  Serial.print("Filtered(°C):");
36  Serial.println(average);
```

Figure 32. Moving Average Filter Algorithm Applied to Temperature Readings

Both the raw and filtered temperature values were printed to the Serial Plotter, which made it possible to compare the noisy input data against the filtered result in real-time.

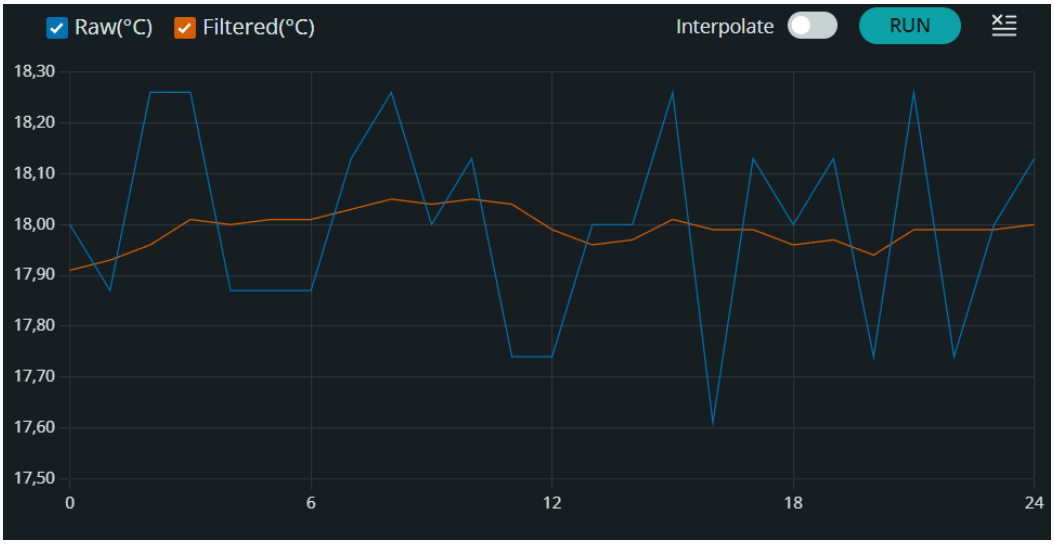

Figure 33. Raw and Filtered Temperature Readings

The filter operated as expected, significantly reducing the amplitude of the noise while preserving the overall trend of the temperature data. This demonstrated the effectiveness of the Moving Average approach in improving the reliability and readability of sensor measurements, once more.

Task 5: Temperature alarm

Building on the previous task, a temperature alarm was implemented by adding a warning system using the built-in LED of the ESP32. The existing Moving Average Filter continued to smooth the temperature readings, but if the filtered temperature exceeded 25 °C, the ESP32's onboard LED (GPIO 2) was turned on, and an alarm message was printed to the Serial Monitor. Otherwise, the LED remained off. This provided both a visual and textual indication whenever the temperature surpassed the defined threshold, demonstrating effective sensor-to-actuator integration.

```
// --- Temperature alarm ---
if (average > 25.0) {
  digitalWrite(ledPin, HIGH);
  Serial.println("ALARM: Temperature exceeds 25 °C!");
} else {
  digitalWrite(ledPin, LOW);
}
```

Figure 34. BMP280 Temperature Alarm Implementation

Task 6: Interfacing the MPU-6050 Sensor

For this task, the MPU-6050 accelerometer and gyroscope sensor was mounted directly onto the chassis of the 2WD car to prepare for future motion control and orientation tasks. The sensor was connected to the microcontroller using the I2C interface, with VCC, GND, SCL, and SDA lines securely wired.

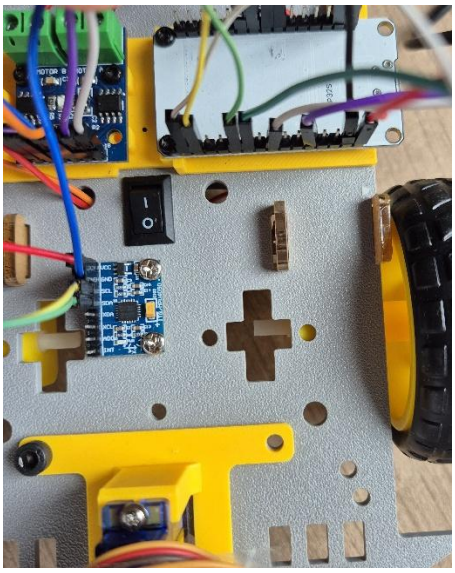

Figure 35. MPU6050 Wiring Connections

Although the workshop folder referenced a specific library ("mpu6050-master.zip"), this library was not provided. Instead, the Adafruit MPU6050 library was installed and prepared for use.

At this stage, the focus was exclusively on mounting the sensor and establishing reliable electrical connections. No code was implemented yet, as this task served as the hardware integration step for the MPU-6050 prior to performing any data acquisition or control algorithms.

Task 7: Implementing Filtering for MPU-6050 Sensor

To improve the quality of the readings acquired from the MPU-6050 sensor, two filtering techniques were implemented: a Moving Average Filter and a Cumulative Average Filter. The goal was to reduce short-term fluctuations and improve stability in the sensor readings, which are often affected by mechanical vibrations and sudden accelerations when mounted on a mobile robot.

```
17  void setup(void) {
18    Serial.begin(115200);
19    while (!Serial) delay(10);
20
21    if (!mpu.begin()) {
22      Serial.println("MPU6050 not found");
23      while (1) delay(10);
24    }
25
26    mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
27    mpu.setGyroRange(MPU6050_RANGE_500_DEG);
28    mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);
29
30    delay(100);
31  }
```

Figure 36. MPU-6050 Filtering Setup Function Implementation

In the setup() function of the code, the MPU-6050 sensor was initialized. If the sensor could not be detected, the program printed an error message and halted. After successful initialization, the accelerometer and gyroscope ranges were configured to provide

suitable sensitivity (8G for acceleration and 500 degrees/sec for gyroscope readings), and the internal digital filter bandwidth was set to 21 Hz to reduce high-frequency noise.

```
33  void loop() {
34    sensors_event_t a, g, temp;
35    mpu.getEvent(&a, &g, &temp);
36
37    float ax = a.acceleration.x;
38    float ay = a.acceleration.y;
39    float az = a.acceleration.z;
40
41    float pitch = atan2(ay, sqrt(ax * ax + az * az)) * 180 / PI;
42    float roll  = atan2(-ax, az) * 180 / PI;
```

Figure 37. MPU-6050 Readings Implementation

In the loop() function, the accelerometer data was retrieved using mpu.getEvent(), from which the pitch and roll angles were calculated. Computing pitch and roll from accelerometer data is essential because these two angles describe the orientation of the car relative to the horizontal plane, information critical for understanding how the robot tilts or rotates during movement, which will be used for future tasks involving balancing or orientation control.

Pitch was computed using the atan2() function applied to the Y-axis and the combination of the X and Z axes, while roll was computed using the X and Z axes. These formulas are standard methods for calculating tilt angles from raw acceleration data.

```
44    // Moving Average
45    pitch_buffer[buffer_index] = pitch;
46    roll_buffer[buffer_index] = roll;
47
48    float pitch_movavg = 0, roll_movavg = 0;
49    for (int i = 0; i < MOVING_AVG_SIZE; i++) {
50      pitch_movavg += pitch_buffer[i];
51      roll_movavg += roll_buffer[i];
52    }
53    pitch_movavg /= MOVING_AVG_SIZE;
54    roll_movavg  /= MOVING_AVG_SIZE;
55
56    buffer_index = (buffer_index + 1) % MOVING_AVG_SIZE;
```

Figure 38. MPU-6050 Moving Average Filter

Once computed, both pitch and roll were passed through a Moving Average Filter, which smoothed out short-term variations by averaging the most recent ten samples. This filter helped to minimize the effects of mechanical vibrations or sudden small movements that could otherwise cause unstable readings.

```
58    // Cumulative Average
59    sample_count++;
60    pitch_cum_avg += (pitch - pitch_cum_avg) / sample_count;
61    roll_cum_avg  += (roll - roll_cum_avg) / sample_count;
```

Figure 39. MPU-6050 Cumulative Average Filter

Additionally, a Cumulative Average Filter was applied to the pitch and roll values. This filter updated a long-term average with each new sample. However, it was noted that as the number of samples grew large, the cumulative average became very slow to respond to new changes, which limited its usefulness for continuously varying signals.

Finally, the raw, moving average, and cumulative average values for both pitch and roll were printed Serial Plotter, allowing for clear visualization of how each filtering method affected the sensor readings.

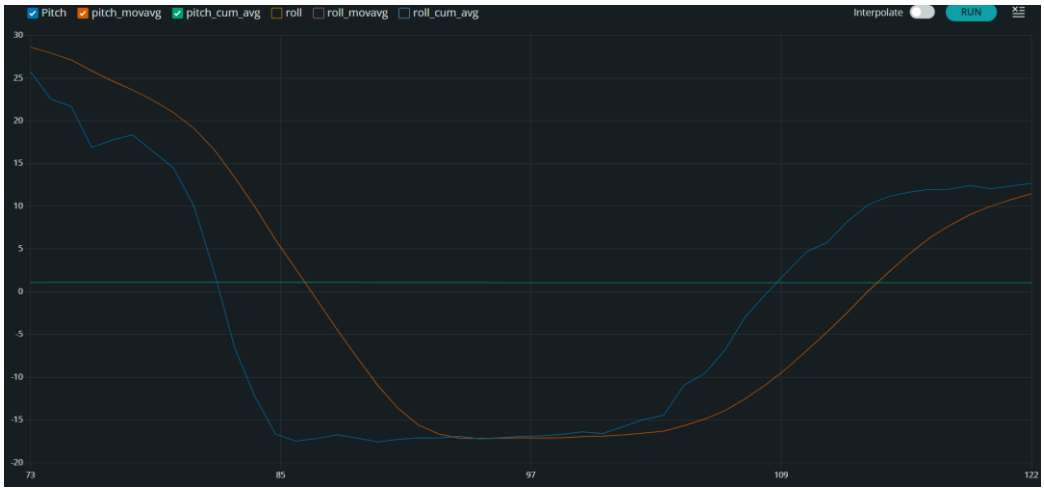

Figure 40. Pitch Raw Data And Filtered Output

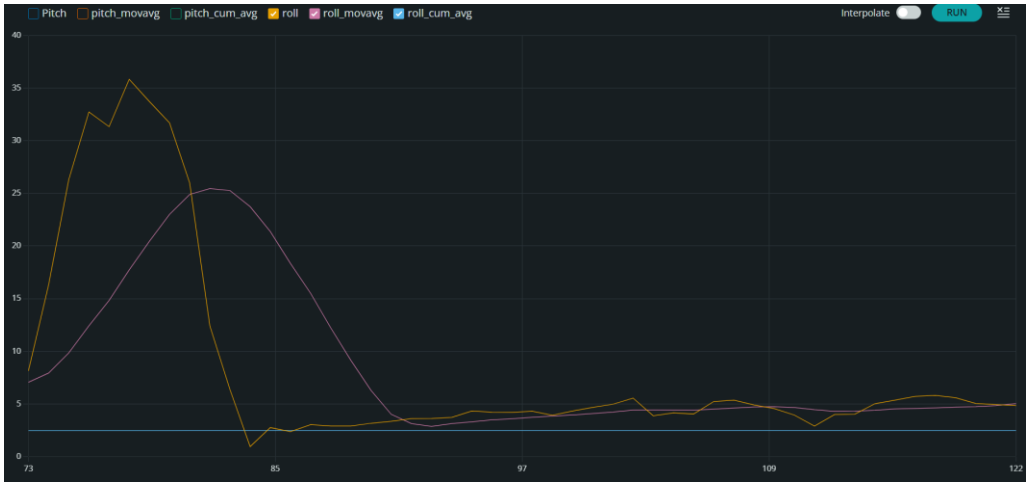

Figure 41. Roll Raw Data And Filtered Output

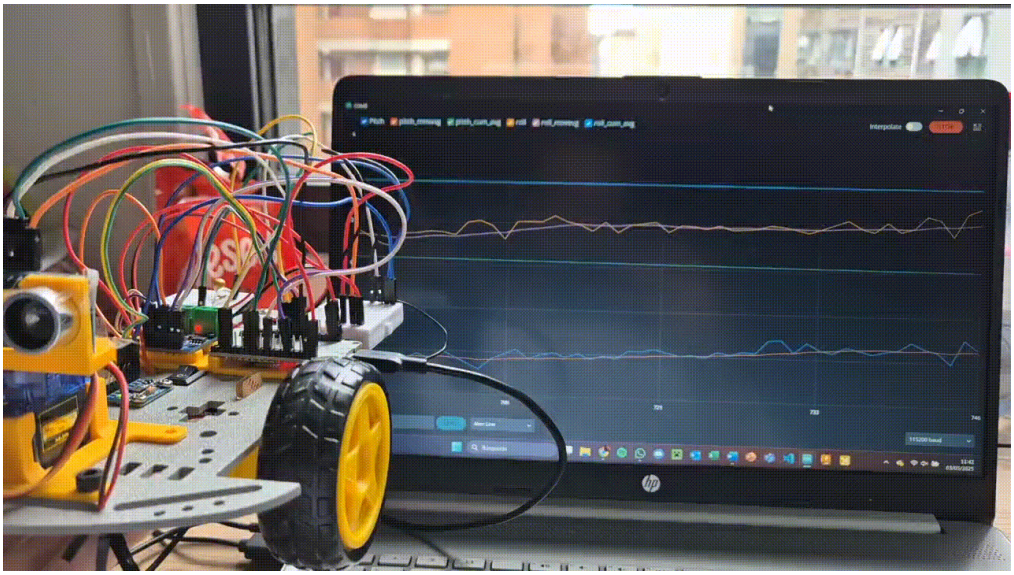

Figure 42. MPU-6050 Real-time Readings

Task 8: Robot Car Manipulation using MPU-6050 Sensor

The goal of this task was to enable the 2WD robot car to adjust its motor speed based on the tilt of the car. If the robot started going down a slope, it would reduce speed. Conversely, when going up a slope, it would increase speed to compensate. This behaviour was achieved by integrating the MPU-6050 sensor into the existing control system and using the pitch angle to determine inclination.

Inclination Measurement

A new function, accelerometerRead(), was added to acquire and process accelerometer data from the MPU-6050. Inside this function, the pitch angle (angleX) was computed using the same method as in the previous task. To prevent constant data acquisition and reduce processing load, readings were limited to once every 100 milliseconds.

```
317  void accelerometerRead(){
318      if(timeToUpdate < millis()){
319          timeToUpdate = millis() + 100;
320          sensors_event_t a, g, temp;
321          mpu.getEvent(&a, &g, &temp);
322
323          angleX = atan2(a.acceleration.y, a.acceleration.z) * 180.0 / PI;
324          float angleY = atan2(a.acceleration.x, a.acceleration.z) * 180.0 / PI;
325
326          Serial.print("Inclination angle X: ");
327          Serial.println(angleX, 1);
328      }
329
330  }
```

Figure 43. AccelerometerRead Function Implementation

Inclination-Based Speed Control

A new operation mode, “modoIncli”, was implemented and could be activated or deactivated through the web interface using two new buttons: Start Inclination Mode and Stop Inclination Mode. In the loop() function, when this mode was enabled, the car continuously checked the value of angleX:

```
294      if (modoIncli) {
295          if (angleX < -10.0) {
296              analogWrite(MOTOR_A1, PWM_SPEED+50);
297              analogWrite(MOTOR_A2, 0);
298              analogWrite(MOTOR_B1, PWM_SPEED+50);
299              analogWrite(MOTOR_B2, 0);
300          }
301          else if (angleX > 10.0) {
302              analogWrite(MOTOR_A1, PWM_SPEED-50);
303              analogWrite(MOTOR_A2, 0);
304              analogWrite(MOTOR_B1, PWM_SPEED-50);
305              analogWrite(MOTOR_B2, 0);
306          }
307          else {
308              analogWrite(MOTOR_A1, PWM_SPEED);
309              analogWrite(MOTOR_A2, 0);
310              analogWrite(MOTOR_B1, PWM_SPEED);
311              analogWrite(MOTOR_B2, 0);
312          }
313      }
```

Figure 44. Inclination-based motor speed control logic Implementation

- If the car was tilting upward, the motor speed would be increased.
- If the car was tilting downward, the motor speed would be decreased.
- If the car remained relatively level (between -10° and 10°), the default speed was maintained.

This simple proportional approach allowed the car to react automatically to slopes without requiring explicit commands from the user.

Web Interface Updates

The existing web interface was extended by adding a new pair of buttons specifically for inclination mode control. This allowed switching between manual driving, obstacle avoidance, and inclination-based driving modes without reprogramming the microcontroller.

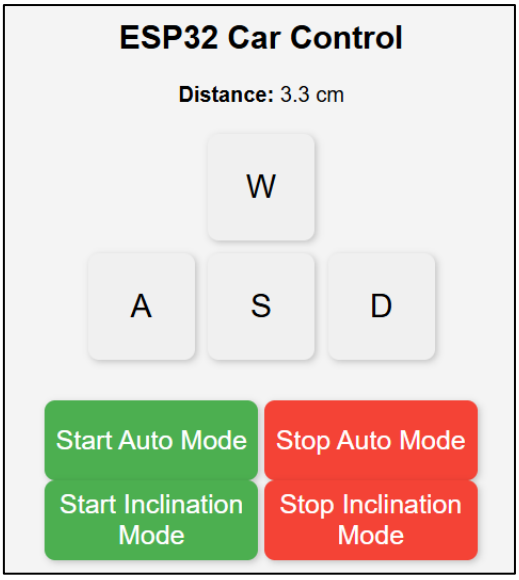

Figure 45. Robot Car Manipulation with MPU-6050 Web Interface

Integration and Testing

The system was tested by manually tilting the car at different angles while observing the motor behaviour. As it can be seen in the Figure 66, the motors correctly adjusted their speed based on the pitch angle, with noticeable acceleration when the car faced upward and deceleration when tilting downward.

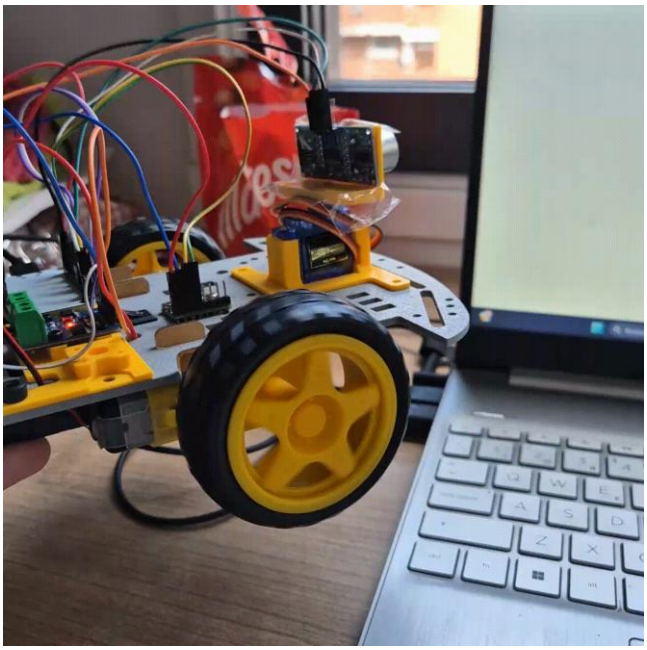

Figure 46. Robot Car Manipulation with MPU-6050 Demonstration

This successful implementation demonstrated that the robot could now respond to environmental conditions (slopes) using onboard sensors and adaptive control.

Workshop 5: PID

Task 1: Creating an RC Filter Circuit

To begin the task, the required passive components, resistors and capacitors, were carefully placed on a small breadboard to form the RC filter. The green wire, which serves as the filtered output, was connected to GPIO 32 of the ESP32, while the yellow wire, carrying the input signal, was connected to GPIO 33, and the white wire was connected to the system ground (GND). Special care was taken to ensure that the resistor and capacitor had secure connections and minimal lead length to reduce unwanted parasitic effects that could impact the filter's performance.

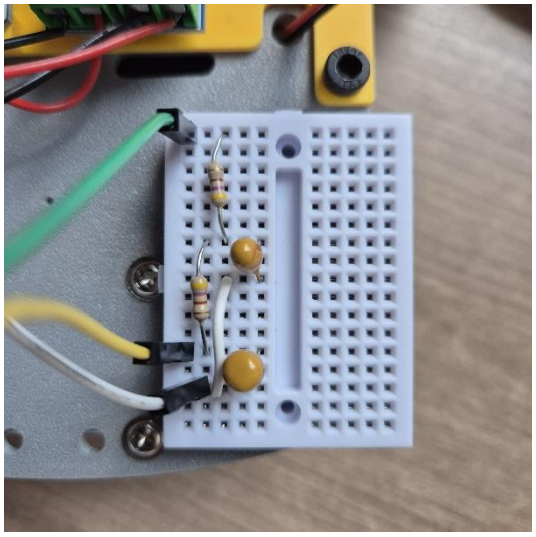

Figure 47. RC Filter Circuit

In preparation for the final version of the robot car, two small breadboards were permanently mounted at the back left and back right corners of the chassis. One breadboard was designated for the PID controller circuit, and the other was used to manage signal conditioning and power distribution, specifically providing a common point to connect all the ground and voltage wires in an organized manner. This approach allowed for better cable management and easier troubleshooting while ensuring system stability.

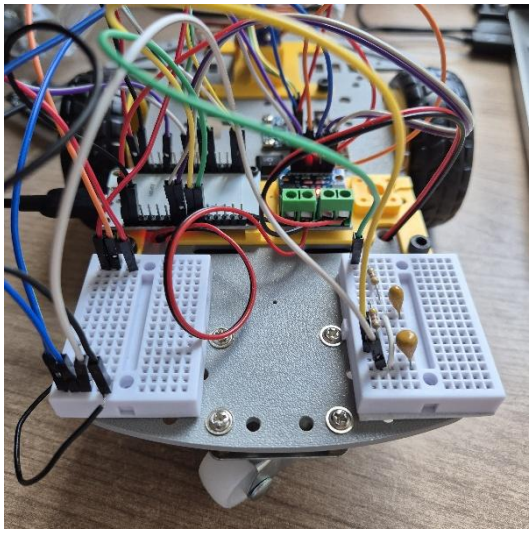

Figure 48. Breadboard Mounting onto the 2WD Robot Car

Task 2: PID on a RC Filter Circuit

Once the RC filter circuit was fully assembled and connected to the ESP32, the provided PID algorithm was implemented in the microcontroller. The input signal was read from GPIO 33 (connected to the filter input), while the output control signal was sent through GPIO 32 (connected to the filter output). The setpoint was defined as 80 (on a PWM scale of 0 to 255).

The PID constants were carefully tuned by trial and error, with final values being kp = 0.5, ki = 0.20, and kd = 0.002. These values ensured that the system could respond quickly while minimizing overshoot and oscillations. Initially, the output showed a significant deviation from the setpoint but stabilized rapidly after a few iterations. The error was reduced to less than 4%, fulfilling the task requirements.

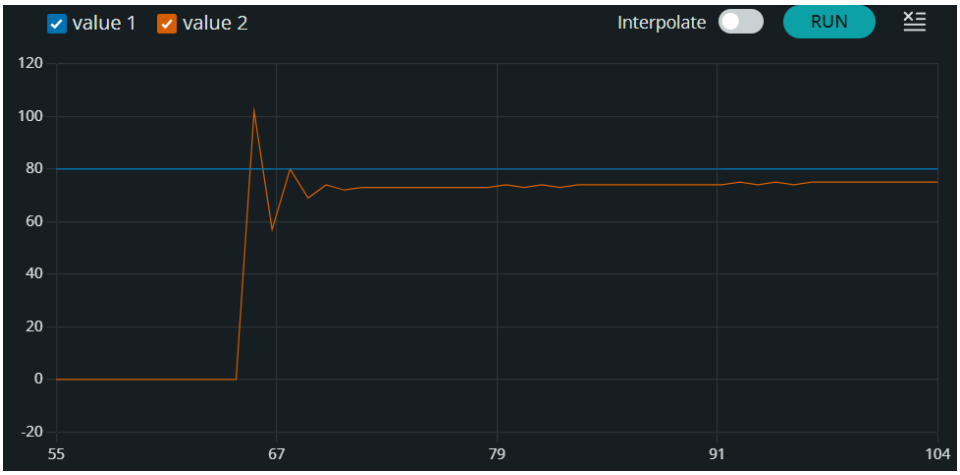

Figure 49. PID on a RC Filter Output

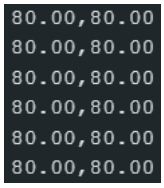

Figure 50. Serial Output Stabilized at 80.00

The PID algorithm calculated the proportional, integral, and derivative components based on the difference between the setpoint and the actual signal. The resulting output controlled the PWM duty cycle, adjusting the voltage applied to the RC circuit to reach and maintain the desired output voltage level.

Task 3: Implement PID with the MPU6050 on a Mobile Robot

For this final task, the MPU-6050 sensor was integrated with the mobile robot to implement an inclination-correcting PID system. The sensor was connected same way it was, which allowed the car to detect tilts forward or backward.

In the loop() function, the PID control logic was implemented to compute the motor speed based on the inclination angle error. First, a dead zone was applied: if the absolute value of the error was less than 2.0 degrees, the motor speed was set to zero and the integral term was reset to avoid wind-up. This prevented the motors from reacting to small fluctuations or sensor noise, which could otherwise cause unnecessary movement or oscillation.

```
47  void loop() {
48    float currentAngle = getAngleX();
49    float error = 0.0 - currentAngle;
50
51    // Dead Zone
52    if (abs(error) < 2.0) {
53      motorSpeed = 0;
54      integral = 0;
55    } else {
56      float P = Kp * error;
57      integral += Ki * error * (DT / 1000.0);
58      float D = Kd * (error - prevError) / (DT / 1000.0);
59      motorSpeed = constrain(P + integral + D, -255, 255);
60    }
61
62    prevError = error;
63    setMotorSpeed((int)motorSpeed);
64
65    delay(DT);
66  }
```

Figure 51. PID Control Loop Function

If the error exceeded this threshold, the PID calculation proceeded: the proportional term (P) was computed by multiplying the error by Kp, providing an immediate response proportional to the current error. The integral term (I) accumulated the error over time, scaled by Ki and the time step DT. This addressed any persistent offset that might keep the car from reaching the target inclination. The derivative term (D) was calculated as the rate of change of the error, scaled by Kd and DT, helping to counteract rapid changes and dampen oscillations. The sum of the P, I, and D terms was constrained to stay within the valid PWM range for motor control. Finally, the previous error was saved for use in the next derivative calculation, and the setMotorSpeed() function was called to apply the computed speed to the motors.

```
68  float getAngleX() {
69    sensors_event_t a, g, temp;
70    mpu.getEvent(&a, &g, &temp);
71
72    float angleX = atan2(a.acceleration.y, a.acceleration.z) * 180.0 / PI;
73
74    float offset = 0.45;
75    angleX += offset;
76
77    Serial.print("AngleX: ");
78    Serial.print(angleX);
79    Serial.print(" | ");
80
81    return angleX;
82  }
```

Figure 52. GetAngleX function for MPU-6050 Sensor

An offset of 0.45 degrees, inside the getAngleX() function, was added to the calculated angle to compensate for the fact that the MPU-6050 was mounted slightly tilted relative to the car's chassis (by less than 1 degree). Without this correction, the system would interpret the car as being permanently inclined even when it was on flat ground, causing unnecessary motor activation.

In the next section of the code, a setMotorSpeed() function was implemented, responsible for converting the PID output (which was a speed value between -255 and 255) into the appropriate PWM signals for the motor driver to control both motors' direction and speed.

```
94  void setMotorSpeed(int speed) {
95    speed = constrain(speed, -255, 255);
96    int pwm = getEffectivePWM(speed, MIN_PWM, MAX_PWM);
97
98    Serial.print("Speed: ");
99    Serial.print(speed);
100
101   if (speed > 0) {
102     analogWrite(MOTOR1_PIN1, pwm);
103     analogWrite(MOTOR1_PIN2, 0);
104     analogWrite(MOTOR2_PIN1, pwm);
105     analogWrite(MOTOR2_PIN2, 0);
106     Serial.print(" | Forward: ");
107     Serial.println(pwm);
108   } else if (speed < 0) {
109     analogWrite(MOTOR1_PIN1, 0);
110     analogWrite(MOTOR1_PIN2, pwm);
111     analogWrite(MOTOR2_PIN1, 0);
112     analogWrite(MOTOR2_PIN2, pwm);
113     Serial.print(" | Backward: ");
114     Serial.println(pwm);
115   } else {
116     analogWrite(MOTOR1_PIN1, 0);
117     analogWrite(MOTOR1_PIN2, 0);
118     analogWrite(MOTOR2_PIN1, 0);
119     analogWrite(MOTOR2_PIN2, 0);
120     Serial.println(" | Motors Stopped");
121   }
122 }
```

Figure 53. Motor Speed And Direction Control Based On PID Output

First, the input speed was adjusted into a practical PWM value that respected the motors' minimum and maximum PWM thresholds. This scaling ensured that if the PID controller output a low value (but not zero), the motors would still receive at least MIN_PWM (145), which was necessary to overcome static friction and initiate movement. Then, the direction and speed were applied accordingly.

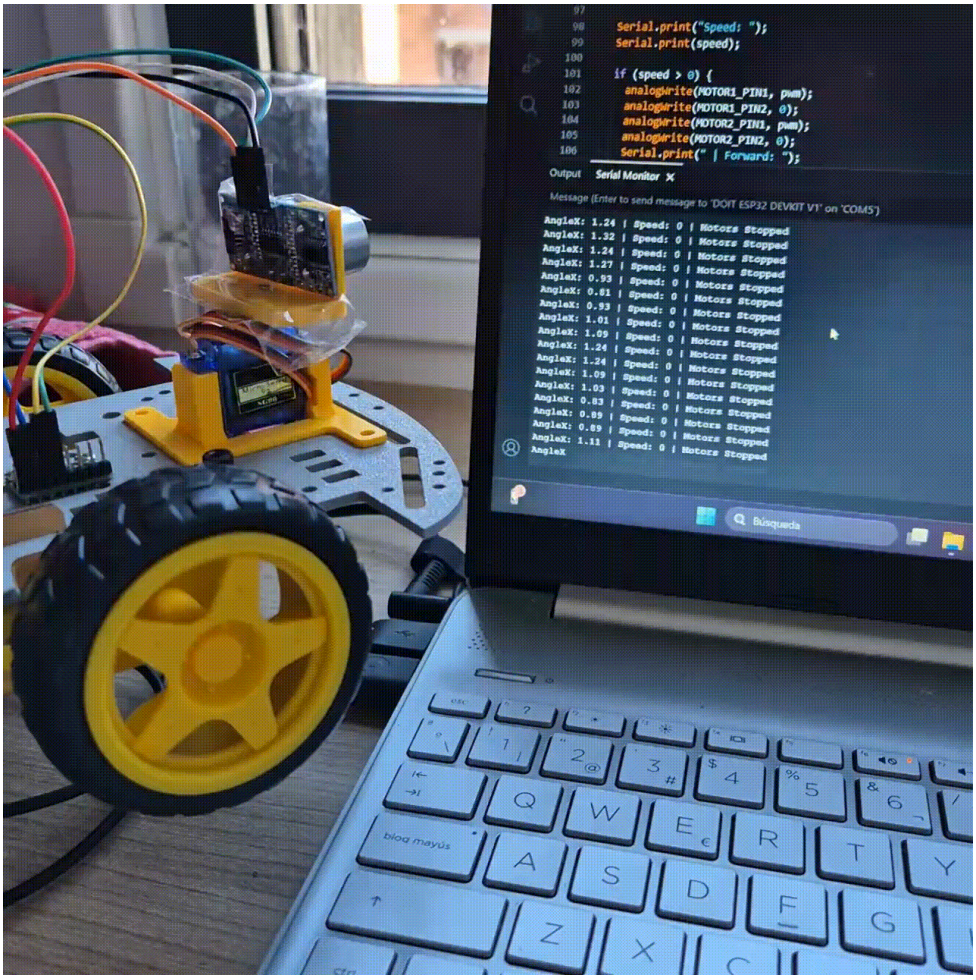

Figure 54. PID with MPU-6050 on the 2WD Robot Car Demonstration

After setting the motor signals, a Serial printout was provided to indicate whether the motors were moving forward, backward, or stopped, along with the actual PWM value applied. This was useful for monitoring the control behaviour in real time, as shown in the next Figure.

Conclusion

Throughout these workshops, a wide range of essential skills in electronics, programming, and control systems were developed. The integration of sensors such as ultrasonic, BMP280, and MPU6050 provided valuable experience in data acquisition, filtering, and interpreting sensor outputs. The implementation of web-based control interfaces enhanced understanding of IoT principles and user interaction. By constructing PID control systems and applying them to both RC circuits and mobile robots, practical knowledge of tuning control parameters and real-world system response was gained. Challenges such as noise filtering, sensor calibration, and mechanical stability were successfully addressed, reinforcing problem-solving abilities. Overall, these tasks provided a comprehensive foundation in robotics, embedded systems, and control engineering that will be applicable to future projects and professional work.