Title:

Autonomous driving and Vehicle tracking

Presented By:

Abdelrahman Shrief	Moamen mohamed
Asmaa Mohamed	Sondos Reda
Mohamed Hossam	Mohamed Alaa
Mahmoud Gaballah	Mohamed Nageh

Supervised By:

Prof. Gamal El-Sheikh

Prof. Heba Em

Cairo 2024_2025

Table of Contents

Table Of Figures:

Abstract

The automotive industry is transforming as vehicles integrate advanced digital technologies, enhancing usability and real-time data access. Among these advancements, self-driving vehicles have emerged as a key focus. Defined by the NHTSA as vehicles capable of operating without human intervention, autonomous cars are set to reshape transportation. The levels of autonomous driving range to 6 levels, from level 0 (no automation) to level 5 (full automation). Our project contributes by developing basic autonomous functions, such as driving in straight lines and following geometric patterns. This is done by using some components that determine the direction and control the direction of the vehicle through some codes from the C/C++ language.

Introduction

Nowadays, the majority of vehicles have been converted to digital form. Offer the driver improved ease of use and enhanced data, including up-to-the-minute traffic updates, performance stats evaluation of information similar to velocity, and playing audio online. Cloud computing, among other advancements, has made cars today highly technologically advanced, and there is a lot more yet to happen. Tomorrow's car will transform the automotive industry. Be a significant improvement from what is currently available. Self-driving or autonomous vehicles have become prevalent. a long-cherished aspiration—a dream that has consistently been unsuccessful—come into existence. Self-driving vehicles are now a reality. A highly specific and specialized In the market, the autonomous car sector is progressing quickly. Progress in incorporating numerous technologies from various sources Developing a self-driving car requires a complex ecosystem. To begin with, what exactly are autonomous vehicles? Per the information provided According to the National Highway Safety Administration (NHTSA), self-driving vehicles are cars capable of driving on their own without human assistance. No human intervention is necessary to steer. We are increasing speed and reducing speed. The definition above suggests self-driving vehicles equipped with autonomous technologies. Allow the vehicle to travel from Point A to Point B by executing all the necessary features needed for a vehicle to operate safely without any passengers inside. Despite the prevailing notion, driverless vehicles are considered a futuristic idea. Competition has commenced to introduce these vehicles onto our streets. These vehicles are causing a disturbance of unprecedented scale and reach. As we know that there are many levels of self-driving, we aspire to reach the highest level of these levels with new technologies. To reach all of this, we will learn about our first steps so that we can work on this. We have used some of the components that help us in controlling and obtaining the correct directions so that the vehicle can drive correctly. We will learn about each of them separately. We have also created a basic form for the application to control the vehicle's movement and direction remotely.

Autonomous Driving Technology

Early research and development of autonomous driving systems began decades ago. Significant milestones include the development of driver assistance systems in the 1980s and the DARPA Grand Challenges in the early 2000s. Commercial ADAS have been introduced in recent years.

Levels of Autonomy:

The Society of Automotive Engineers (SAE) has defined six levels of autonomy for self-driving cars, as shown in Figure 1 :

- ### Level 0: No Automation

    The driver controls all aspects of driving.

- ### Level 1: Driver Assistance

    The vehicle can assist with steering or acceleration/deceleration, but the driver is still in control.

- ### Level2: Partial Automation

    The vehicle can control both steering and acceleration/deceleration under certain conditions, but the driver must remain vigilant.

- ### Level 3: Conditional Automation

    The vehicle can handle most driving tasks under certain conditions, but the driver must be ready to take over.

- ### Level 4: High Automation

    The vehicle can drive itself under specific conditions, but a human driver may be needed in some situations.

- ### Level 5: Full Automation

    The vehicle can drive itself under all conditions, without any human input.
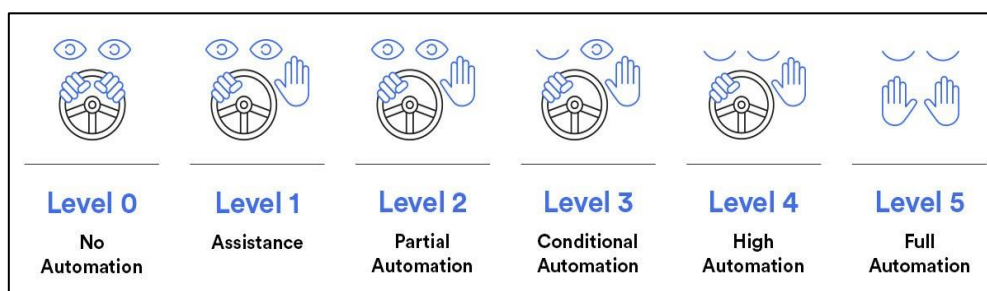

Figure 1 LEVELS OF AUTONOMY

Components Of Project :

Arduino UNO

Arduino UNO is a microcontroller board based on the ATmega328P, as shown in Figure 2. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button.
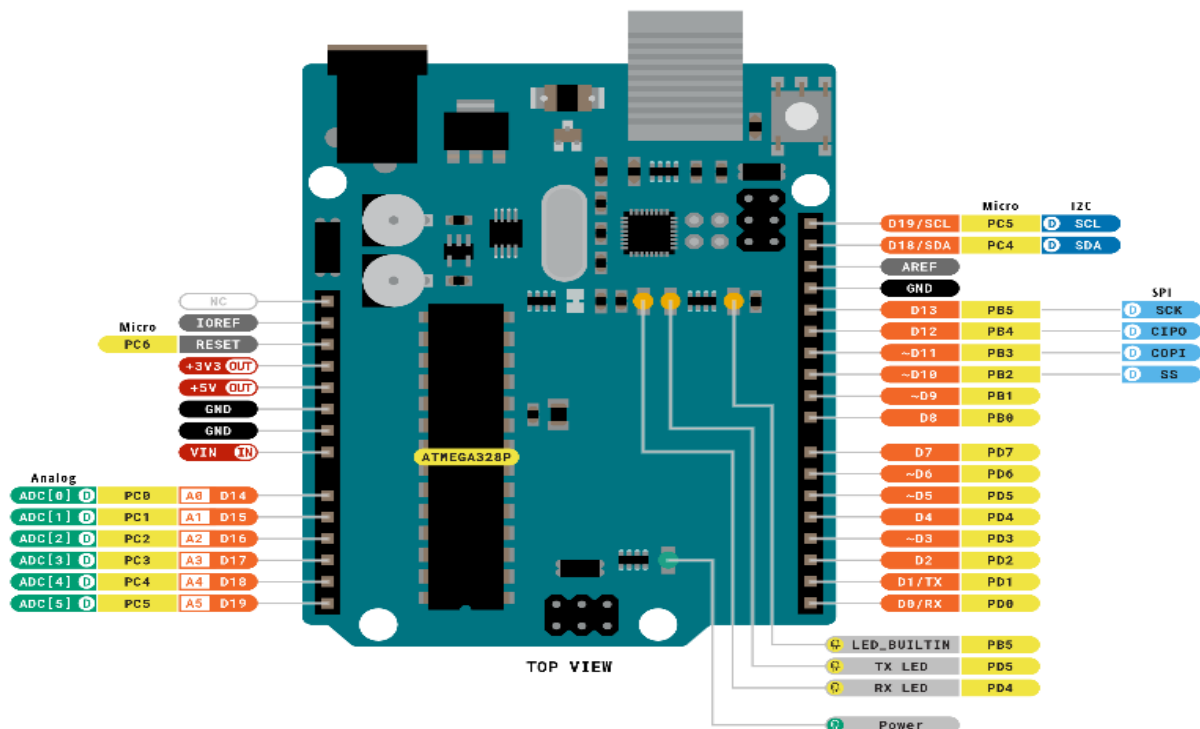
Figure 2 ARDUINO UNO PINOUT

L298 motor driver

The L298 motor driver is a popular dual H-bridge motor driver integrated circuit (IC) as shown in Figure 3, commonly used in projects requiring motor control :

- **Dual H-Bridge**
- **Maximum Voltage:** up to 46V
- **Current Handling:** The L298 can handle peak currents of up to 2A per channel and 4A if a heat sink is used .
- **Enable Pins:** Each H-bridge has an enable pin that allows for PWM (pulse-width modulation) control of motor speed.

Figure 3 L298 PINOUT

DC Motors

A DC motor is an electrical device that converts electrical energy into mechanical motion. It operates on direct current (DC) power and is used in autonomous vehicles, for functions like propulsion and steering. These motors enable the car to move, accelerate, decelerate, and navigate autonomously, playing a key role in the vehicle's overall functionality and movement capabilities. Powered by 3-12 volts, draws an average of 190 mA of current (max 250 mA), as shown in Figure 4 .
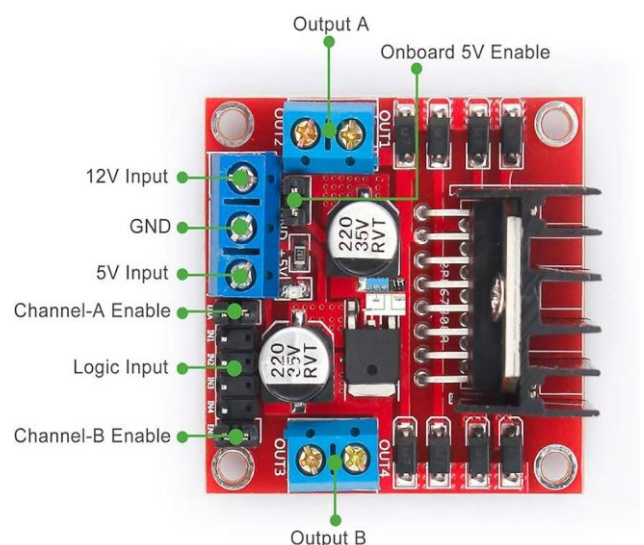

Figure 4 DC MOTORS + TIRES

MPU6050

The MPU-6050 is a Motion Tracking Device that combines a 3-axis gyroscope and a 3-axis accelerometer in a single chip. It is commonly used in various electronic applications for measuring motion and orientation, as shown in Figure 5 . Here are the specifications of the MPU-6050 :

- **Operating voltage :** 2.375 V to 3.46 V
- **Gyroscope :**
  - **Angular Velocity:** Measures rotational motion in degrees per second (dps).
  - **Sensitivity:** The sensitivity can be configured based on the selected range (e.g., ±250, ±500, ±1000, or ±2000 dps).
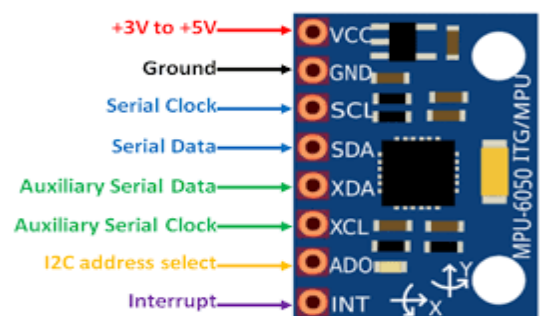
Figure 5 MPU6050 PINOUT

- **Temperature Sensor :** The MPU-6050 has an integrated temperature sensor that measures the device's temperature.

- **Accelerometer :**
  - **Acceleration:** Measures linear acceleration in three axes (X, Y, Z).
  - **Sensitivity:** The sensitivity can be configured based on the selected range (e.g., ±2g, ±4g, ±8g, or ±16g).

- **The Digital Low Pass Filter** can be configured to filter out high-frequency noise from sensor data.

ESP8266

The ESP8266 is a low-cost, highly integrated Wi-Fi microcontroller chip designed by **Espressif Systems**. It's widely used in IoT (Internet of Things) projects due to its ease of use, low power consumption, and robust wireless connectivity, as shown in Figure 6. Below are the key details about the ESP8266 :

- **Memory :**
  - **RAM:** 160 KB of on-chip SRAM.
  - **ROM:** 32 KB boot ROM.
  - **Flash:** The chip supports external QSPI flash memory.

- **Power :**
  - **Operating Voltage :** 3.0 V to 3.6 V.
  - **Power Consumption :** Low power consumption with sleep modes.
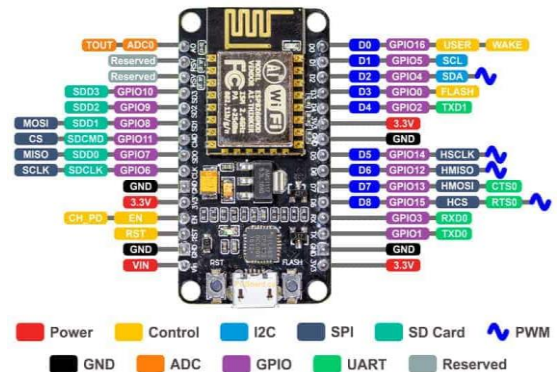
Figure 6 ESP8266 PINOUT

- **Programming Languages :** Typically programmed in C/C++.

HC-020k

The HC-020K Encoder Module is a versatile device commonly used in robotics and automation projects. It features an incremental rotary encoder that translates rotational position into digital signals, allowing for precise motion control and feedback. Key features include:

Figure 7 HC-020k PINOUT

- **Resolution**: The encoder provides high-resolution output, enabling fine control over motor positions.

- **Output Type**: Typically outputs two signals (A and B) that are 90 degrees out of phase, allowing for direction detection.

- **Compatibility**: Often used with microcontrollers like Arduino or Raspberry Pi for easy integration.

- **Applications**: Suitable for applications in robotics, CNC machines, and any project requiring accurate rotational position feedback.

Circuit Diagram

We will show how to connect the components to each other as shown in Figure 8 .

Figure 8 CIRCUIT DIGRAM

Arduino Code Flowchart

Figure 9 FLOW CHART OF ARDUINO CODE
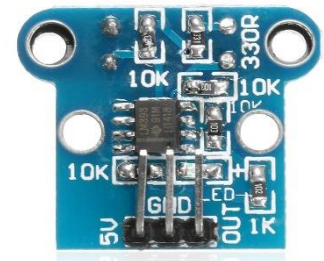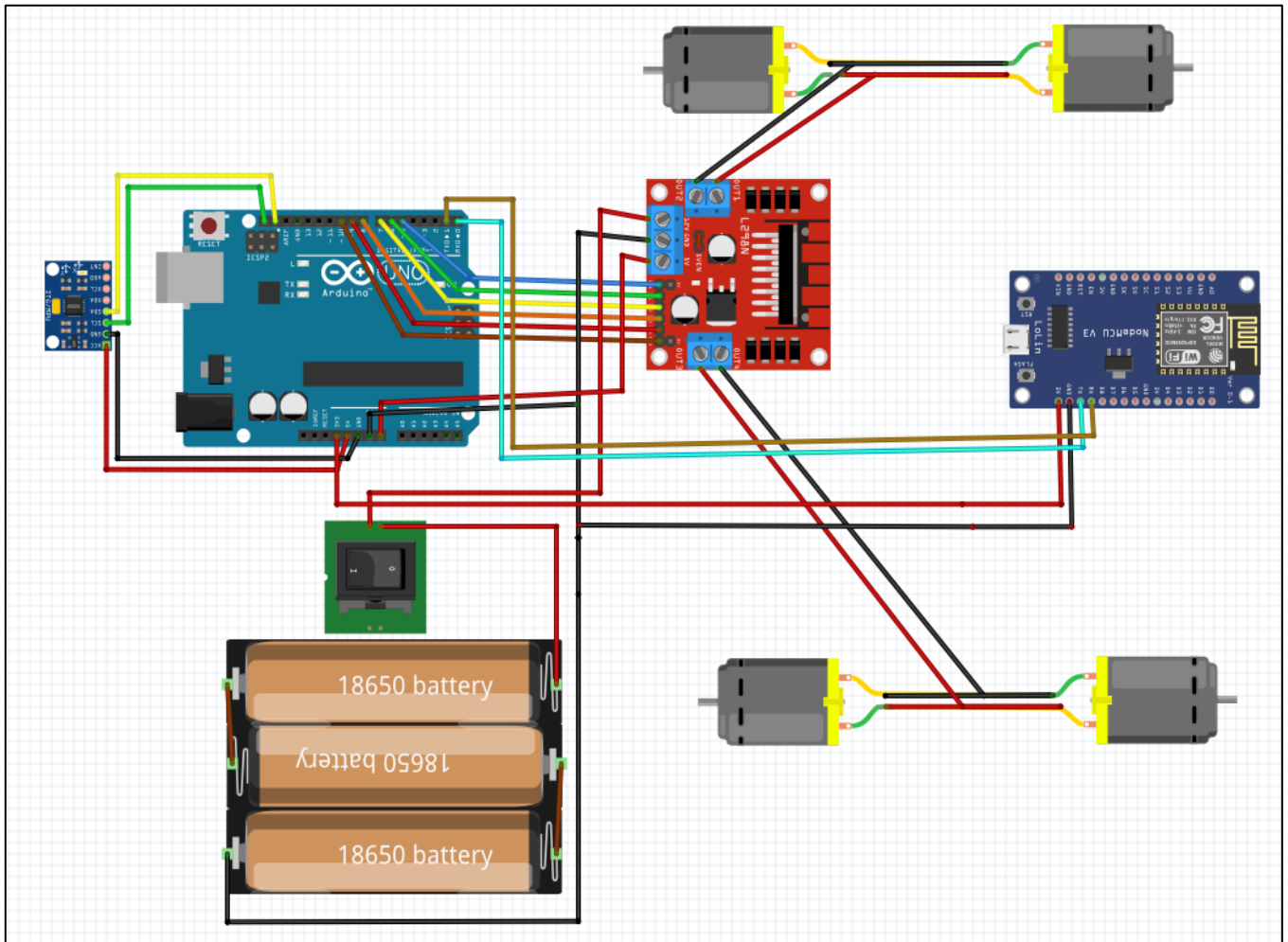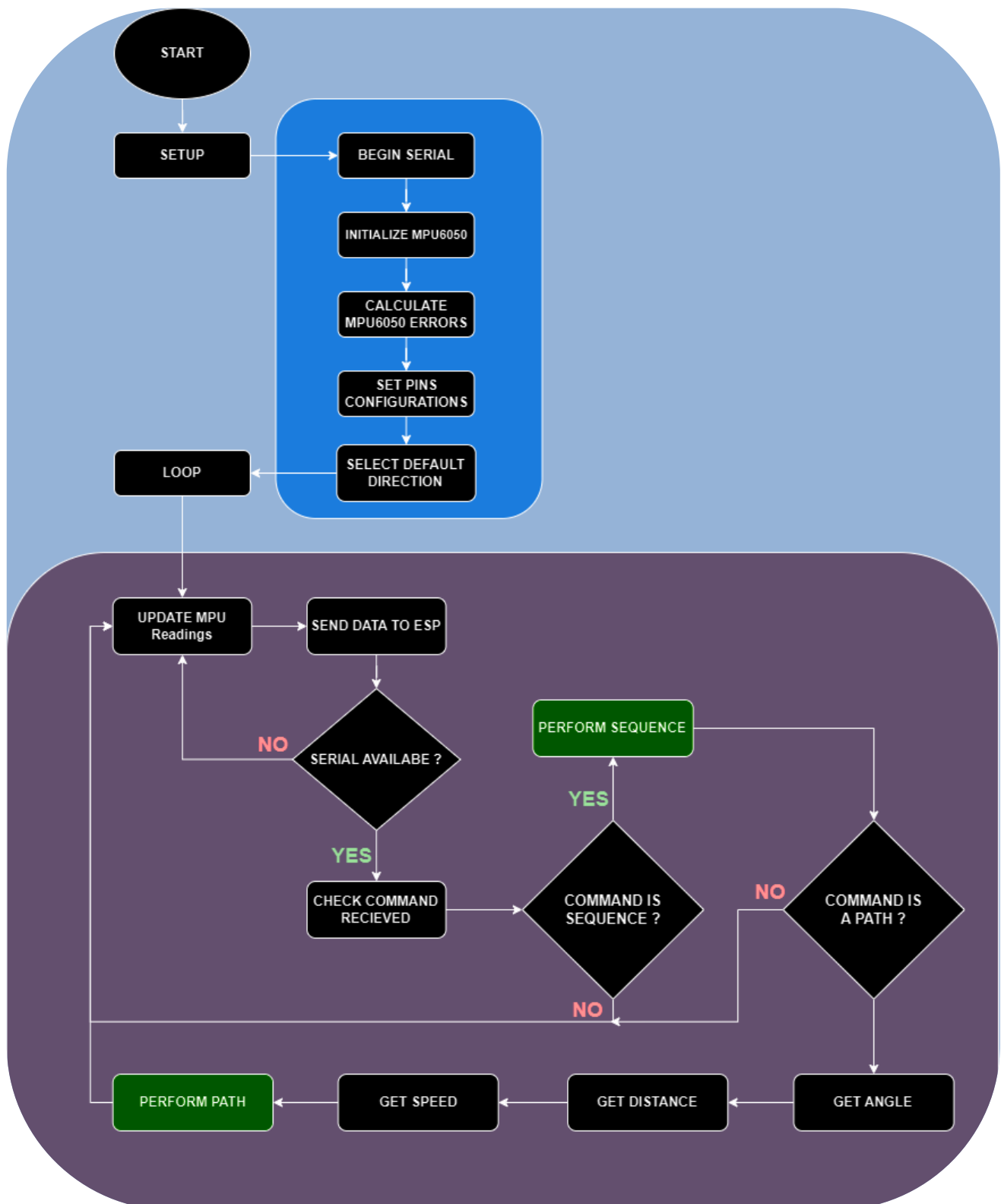
Explain MPU6050 Equations With Complementary Filter

MPU6050 has two main functions in Arduino code as follows :

Calculate_IMU_error :

This function calculate_IMU_error(), is designed to compute the error values for both accelerometer and gyroscope sensors of the Inertial Measurement Unit (IMU) MPU6050.

1. Accelerometer Data Error Calculation :

```
while (c < 200) {
  Wire.beginTransmission(MPU);
  Wire.write(0x3B);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 6, true);
  AccX = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
  AccY = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
  AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
  AccErrorX = AccErrorX + ((atan((AccY) / sqrt(pow((AccX), 2) + pow((AccZ), 2))) * 180 / PI));
  AccErrorY = AccErrorY + ((atan(-1 * (AccX) / sqrt(pow((AccY), 2) + pow((AccZ), 2))) * 180 / PI));
  c++;
}
AccErrorX = AccErrorX / 200;
AccErrorY = AccErrorY / 200;
```

- The function uses an I2C protocol (Wire library) to communicate with the IMU sensor (MPU).
- It sends a command to read the accelerometer data (starting from register 0x3B for the MPU6050).
- It then requests 6 bytes of data from the IMU. The accelerometer provides three 16-bit values (X, Y, and Z axes).
- The 6 bytes are split into three 16-bit values for the X, Y, and Z axes.
- The raw accelerometer data is divided by 16384.0 to convert it into units of gravitational force (g), assuming the sensitivity scale is set to ±2g (for the MPU6050).
- The function calculates the pitch (AccErrorX) and roll (AccErrorY) angles from the accelerometer data using trigonometric formulas.
- The error is accumulated over 200 readings. The formula uses the atan() (arctangent) function to derive the angle in degrees from the accelerometer's raw data.
- After collecting 200 readings, the accumulated error is divided by 200 to compute the average error, which represents the systematic bias in the sensor readings.

2. Gyroscope Data Error Calculation :

- The gyroscope error calculation begins in the same way as the accelerometer, except it reads from a different register (0x43 for the MPU6050).
- The function requests 6 bytes of data corresponding to the gyroscope's X, Y, and Z axis readings.
- Like the accelerometer, the 6 bytes are split into three 16-bit values for the X, Y, and Z axes of the gyroscope.

```
c = 0;
 while (c < 200) {
   Wire.beginTransmission(MPU);
   Wire.write(0x43);
   Wire.endTransmission(false);
   Wire.requestFrom(MPU, 6, true);
   GyroX = Wire.read() << 8 | Wire.read();
   GyroY = Wire.read() << 8 | Wire.read();
   GyroZ = Wire.read() << 8 | Wire.read();
   GyroErrorX = GyroErrorX + (GyroX / 131.0);
   GyroErrorY = GyroErrorY + (GyroY / 131.0);
   GyroErrorZ = GyroErrorZ + (GyroZ / 131.0);
   c++;
 }
```

- The raw gyroscope data is scaled by dividing it by 131.0. This value assumes the gyroscope is set to a sensitivity of ±250 degrees per second (dps) for the MPU6050, where each unit represents 1/131.0 degrees per second.
- The error for each axis is accumulated over 200 readings.

3. Averaging the Gyroscope Errors :

- After collecting 200 readings, the accumulated gyroscope error is divided by 200 to compute the average error, representing the bias in the sensor.

```
  GyroErrorX = GyroErrorX /
200;
  GyroErrorY = GyroErrorY /
200;
```

Update_mpu_readings :

This function update_mpu_readings() reads data from an MPU6050 sensor and calculates roll, pitch, and yaw using a complementary filter.

1. Reading Accelerometer Data :

- The function communicates with the MPU6050 via I2C and sends a request to start reading accelerometer data from register 0x3B.
- The function requests 6 bytes of data, corresponding to the accelerometer

```
  Wire.beginTransmission(MPU);
  Wire.write(0x3B);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 6,
true);
```

values for the X, Y, and Z axes (each value is 16-bit, hence 2 bytes per axis).

2. Extracting and Normalizing Accelerometer Data :

```
AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;
AccY = (Wire.read() << 8 | Wire.read()) / 16384.0;
AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0;
```

- The 6 bytes of data are split into three 16-bit integers for the X, Y, and Z axes using bitwise operations (<< 8 |).
- The raw accelerometer values are divided by 16384.0 to convert them to units of gravitational acceleration (g). This scaling factor corresponds to the MPU6050 being set to a ±2g range (as per the sensor's datasheet).

3. Calculating Roll and Pitch from Accelerometer Data :

```
accAngleX = (atan(AccY / sqrt(pow(AccX, 2) + pow(AccZ, 2))) * 180 / PI) - AccErrorX;
accAngleY = (atan(-1 * AccX / sqrt(pow(AccY, 2) + pow(AccZ, 2))) * 180 / PI) - AccErrorY;
```

- The function uses trigonometric formulas to compute roll (accAngleX) and pitch (accAngleY) angles based on the accelerometer data.

- atan() calculates the arctangent of the given ratio, and then the result is converted from radians to degrees (* 180 / PI).

- The calculated accelerometer errors (AccErrorX, AccErrorY) are subtracted to compensate for the inherent bias of the sensor, which was determined using the calculate_IMU_error() function.

- Formulas used :

  ➢ **Roll** (rotation about the X-axis):

  $$\text{Roll} = \tan^{-1}\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right)$$

  ➢ **Pitch** (rotation about the Y-axis):

  $$\text{Pitch} = \tan^{-1}\left(\frac{-A_x}{\sqrt{A_y^2 + A_z^2}}\right)$$

  ➢ **Roll** (rotation about the Z-axis):

  $$\text{Yaw} = \int g_z \, . \, dt$$

> The MPU6050 doesn't directly calculate yaw using accelerometer, so we have to integrate the gyroscope's z-axis angular velocity (g_z) over time to get yaw.

4. Reading Gyroscope Data :

```
previousTime = currentTime;
currentTime  = millis();
elapsedTime  = (currentTime - previousTime) / 1000.0;
Wire.beginTransmission(MPU);
Wire.write(0x43);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
```

- The function records the time between the current and previous readings (elapsedTime), which is used to compute the angular displacement from the angular velocity.
- elapsedTime is measured in seconds by dividing the time difference in milliseconds by 1000.
- The function starts reading from register 0x43 to get gyroscope data.
- 6 bytes are requested to get the X, Y, and Z angular velocities.

5. Extracting and Scaling Gyroscope Data :

```
GyroX = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroY = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroZ = (Wire.read() << 8 | Wire.read()) / 131.0;
```

- Like the accelerometer, the 6 bytes are split into three 16-bit integers for the X, Y, and Z axes.
- The raw gyroscope values are divided by 131.0 to convert them into degrees per second (dps). This scaling factor assumes the gyroscope is set to a sensitivity of ±250 degrees per second (as per the sensor's datasheet).

6. Correcting Gyroscope Data Using Error Values:

- The gyroscope readings are corrected by subtracting pre-calculated error values (GyroErrorX, GyroErrorY, GyroErrorZ), which were determined using the calculate_IMU_error() function.

```
GyroX = GyroX - GyroErrorX;
GyroY = GyroY - GyroErrorY;
GyroZ = GyroZ - GyroErrorZ;
```

- This corrects the gyroscope bias for more accurate readings.

7. Calculating Gyroscope Angles:

```
gyroAngleX = gyroAngleX + GyroX * elapsedTime;
gyroAngleY = gyroAngleY + GyroY * elapsedTime;
yaw = yaw + GyroZ * elapsedTime;
```

- The angular velocities from the gyroscope are integrated over time to compute the angular displacement.

- Since gyroscope values are in degrees per second, multiplying by elapsedTime (in seconds) gives the angle in degrees.
- gyroAngleX, gyroAngleY, and yaw represent the angles calculated solely from the gyroscope data.

8. **Applying Complementary Filter:**

```
roll = 0.96 * gyroAngleX + 0.04 * accAngleX;
pitch = 0.96 * gyroAngleY + 0.04 * accAngleY;
```

- The complementary filter combines the gyroscope and accelerometer angles to get more stable and accurate values for roll and pitch.
- The gyroscope provides quick response to changes, but it tends to drift over time. The accelerometer gives more stable values but can be noisy.
- The complementary filter formula weights the gyroscope data more heavily (0.96) but still uses a small portion of accelerometer data (0.04) to counteract drift.
- The final roll and pitch angles are a blend of both gyroscope and accelerometer data, where the filter helps in maintaining both responsiveness and long-term stability.

The Process of Changing Direction of The Car

The direction of the car is controlled by adjusting the motor pin signals based on yaw readings from the MPU6050 sensor.
There are three main functions to change direction of the car:

Forward Turn :

- The function moveForward() sets the right motor (motorR1 and motorR2) and the left motor (motorL1 and motorL2) to rotate forward, as shown in code.
- This makes both wheels rotate in the forward direction, causing the car to move straight ahead .

```
void moveForward() {
  digitalWrite(motorR1, LOW);
  digitalWrite(motorR2, HIGH);
  digitalWrite(motorL1, LOW);
  digitalWrite(motorL2, HIGH);
}
}
```

Right Turn :

- The function moveRight() rotates the right motor backward (motorR1, HIGH; motorR2, LOW;) and the left motor forward (motorL1, LOW; motorL2, HIGH;) , as shown in code .
- This combination causes the car to pivot to the right as the left motor pushes forward while the right motor moves backward .

```
void moveRight() {
  digitalWrite(motorR1, HIGH);
  digitalWrite(motorR2, LOW);
  digitalWrite(motorL1, LOW);
  digitalWrite(motorL2, HIGH);
}
```

Left Turn :

- The function moveLeft() rotates the left motor backward (motorL1, HIGH; motorL2, LOW;) and the right motor forward (motorR1, LOW; motorR2, HIGH;) , as shown in code .
- This causes the car to pivot to the left.

The Process of Moving Car In A Path

The car moves along a path by following a sequence of steps, which include turning to specific angles and moving forward for set distances. The function SetCarPath() is the core of this mechanism, where the car adjusts its yaw (turning angle) and controls its forward movement.

For example, the car was given an order to turn right at 90-degree angle and then move forward one meter with speed 150 for the path, the sequence will be as the following :

Turn to the Desired Angle :
- The car first turns to the desired angle (90-degree) using the turnCar(initial_angle, speedTurn) function.
- This ensures the car is facing the correct direction before moving forward.
- The yaw value from the MPU6050 sensor is continuously monitored, and the car adjusts its direction until the yaw matches the target angle (initial_angle).

Move Forward :

- Once the car is oriented correctly, it moves forward using the moveForward() function.
- The car will keep moving forward until it has covered the specified distance (one meter).
- Distance is calculated using an encoder (HC-020K). Each encoder step is counted, and the distance is computed as:

$$distance = \left(\frac{steps}{90.0}\right) * 100$$

> Note :
> - Every 90 step from encoder represented a meter (100 cm).
> - Distance is calculated by cm.

- The motors' speeds are adjusted dynamically to keep the car moving straight, based on the yaw angle, as shown in code :

```
analogWrite(motorEN1, speed + (yaw- initial_angle)*speed_adjust_factor + speed_error_factor);
analogWrite(motorEN2, speed - (yaw- initial_angle)*speed_adjust_factor - speed_error_factor);
```

Stop After Covering Distance :

When the car reaches the specified distance, it stops using the instantStop() function, which halts both motors.

➢ Detailed Breakdown of Path Movement:

The SetCarPath() function takes three parameters:

- **initial_angle**: the angle the car should turn to.
- **path_distance**: the distance the car should move forward after turning.
- **speed**: the speed at which the car should move.

➢ Step-by-Step Movement in a Path :

1. Turn to Angle :

- The car starts by turning to the specified initial_angle using turnCar(). This function compares the current yaw with the desired yaw and adjusts the motor direction accordingly :
- If the car needs to turn right (i.e., the current yaw is greater than the target yaw), it calls moveRight().
- If the car needs to turn left, it calls moveLeft().
- Once the car reaches the target yaw with range $\left(^+_-\right)$ 1 degree, it stops and proceeds to move forward.

2. Move Forward:

- After the turn is completed, the car moves forward using the moveForward() function. The motors are controlled such that the car travels straight.
- The distance the car travels is tracked by counting the encoder steps and converting those steps into distance.

3. Adjust Speed to Stay on Path :

- During the forward movement, the car continuously checks its yaw angle to ensure it's moving straight along the path. If the car deviates from the desired angle:
- The speed of the left and right motors is adjusted to correct the direction, based on how far the yaw deviates from the initial angle.

4. Stop at the Desired Distance :

- The car keeps moving forward until the total distance covered (tracked via encoder steps) matches path_distance. Once this condition is met, the car stops using the instantStop() function, bringing both motors to a halt.

The Process of Moving Car In A Sequence

One of the options is to set a path for the car to move in specific shapes like a square, rectangle, or triangle, there are functions that instruct the car to follow a sequence of turns and forward movements, Let's break down how each shape is defined and how you can set the path for the car to move in these patterns.

Moving in a Square Path

In the moveSquare() function, as shown in code and the car moves in a square by:
- Moving forward along each side of the square.
- Turning 90 degrees at each corner.
- **SetCarPath(yaw, length, speed):** Moves the car forward by the length of one side of the square.
- **SetCarPath(yaw + 90, length, speed):** After each side, the car turns 90 degrees (right turn) and moves forward along the next side.
- This sequence repeats four times, forming a square.

```
void moveSquare(float length, char speed) {
  SetCarPath(yaw, length, speed);          // Move forward for the first side
  SetCarPath(yaw - 90, length, speed);     // Turn and move for the second side
  SetCarPath(yaw - 90, length, speed);     // Turn and move for the third side
  SetCarPath(yaw - 90, length, speed);     // Turn and move for the last side
  turnCar(yaw - 180, speedTurn);           // Final turn to complete the square
}
```

Moving in a Rectangle Path

- The moveRectangle() function works similarly as shown in code but takes both length and width as inputs since the rectangle has different side lengths .
- The car alternates between moving forward by the rectangle's length and width, turning 90 degrees after each side to form the shape .

```
void moveRectangle(float length, float width, char speed) {
  SetCarPath(yaw, length, speed);          // Move forward for the first side (length)
  SetCarPath(yaw - 90, width, speed);      // Turn and move for the second side (width)
  SetCarPath(yaw - 90, length, speed);     // Turn and move for the third side (length)
  SetCarPath(yaw - 90, width, speed);      // Turn and move for the fourth side (width)
  turnCar(yaw - 90, speedTurn);            // Final turn to complete the rectangle
}
```

Moving in a Triangle Path :

- The moveTriangle() function defines an equilateral triangle, where the car moves forward and turns 120 degrees at each corner, as shown in code.

- **SetCarPath(yaw + 120, length, speed):** The car turns 120 degrees after each side, forming an equilateral triangle.

```c
void moveTriangle(float length, char speed) {
  SetCarPath(yaw, length, speed);          // Move forward for the first side
  SetCarPath(yaw - 120, length, speed);    // Turn and move for the second side
  SetCarPath(yaw - 120, length, speed);    // Turn and move for the third side
  turnCar(yaw - 180, speedTurn);           // Final turn to complete the triangle
}
```

MATLAB Code :

We display the car's data as it moves using MATLAB, as shown in Figure 10.

```matlab
clc,clear,close all
% Step 1: Load the data from the text file
fileID = fopen('gp_task_1_data.txt', 'r');
% Assuming the file has two columns: [distance,yaw_angle]
data = textscan(fileID, '%f,%f', 'Delimiter', '\n');
fclose(fileID);

% Extract yaw angle and distance
distances = data{1};   % In meters or the unit of your data
yaw_angles = data{2};  % In degrees

subplot(311)
plot(distances,'r','linewidth',2);grid on
title('distance')
subplot(312)
plot(yaw_angles,'b','linewidth',2);grid on
title('angle')

% Step 2: Initialize position
x = 0;  % Starting x-coordinate
y = 0;  % Starting y-coordinate

% Convert yaw angles from degrees to radians for trigonometric calculations
yaw_angles_rad = deg2rad(yaw_angles);

% Initialize arrays to store path coordinates
x_coords = zeros(length(distances), 1);
y_coords = zeros(length(distances), 1);

% Step 3: Calculate the car's path
for i = 1:length(distances)
    % Calculate change in position (delta x, delta y)
    delta_x = distances(i) * cos(yaw_angles_rad(i));
    delta_y = distances(i) * sin(yaw_angles_rad(i));

    % Update current position
    x = x + delta_x;
    y = y + delta_y;
```

```matlab
    % Store the coordinates
    x_coords(i) = x;
    y_coords(i) = y;
end

% Step 4: Plot the car's path
subplot(313)
plot(x_coords, y_coords, '-o');  % Line plot with markers
xlabel('X Position (m)');
ylabel('Y Position (m)');
title('Car Path Based on Yaw Angle and Distance');
grid on;
axis equal;  % Ensures equal scaling on both axes
```
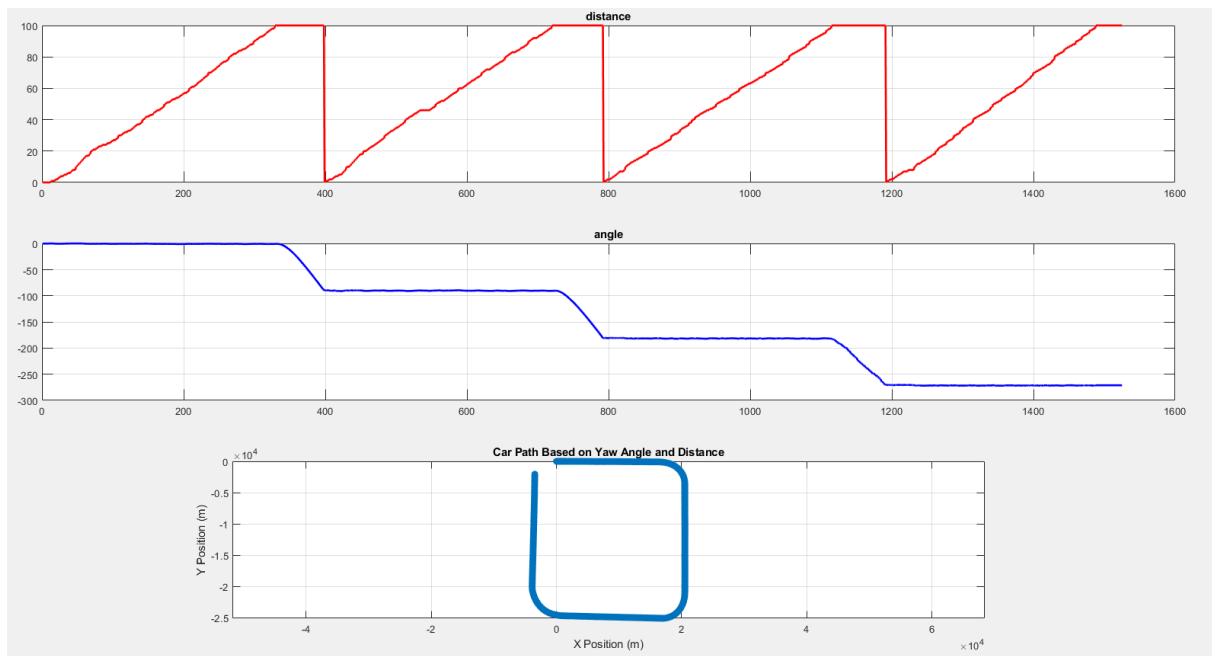
Figure 10 MATLAB CODE OUTPUT

Idea Of Web Application And How It Work :

The web application serves as the primary interface for controlling the car's movement, providing a simple platform for entering key parameters such as distance, angle, and speed. Designed with accessibility and simplicity in mind, the application allows users to send commands to the ESP module, which then communicates with the Arduino to execute precise movements.

The web app is accessible from any device with a browser, providing complete control. The underlying architecture integrates frontend and backend components, allowing for real-time data transmission and command execution. By abstracting the complexity of the underlying hardware, the web application allows users to focus on controlling the car's movement without requiring extensive technical knowledge by connecting to the ESP network which called **Autonomous Car** and password is **auto2244** ,then can go to **192.168.1.100** which operates as the Server Side.

Data Displayed (Angle, Distance, Speed):

The web application mainly displays the three critical parameters that govern the car's movement: angle, distance, and speed as shown in figure 10.1. These values are input by the user through an intuitive interface, where sliders or text fields can be used to specify precise numbers. Once set, this data is transmitted to the ESP module, which processes and relays the information to the Arduino. The application ensures that the displayed data is always accurate and up-to-date, reflecting the current state of the inputs. This functionality allows users to make quick adjustments and observe the immediate impact on the car's movement, providing a dynamic and responsive control experience.
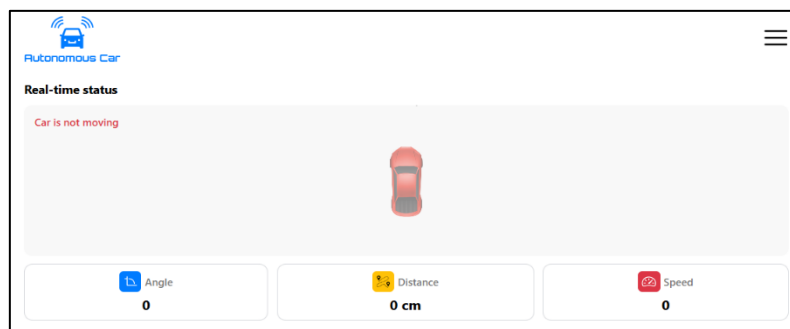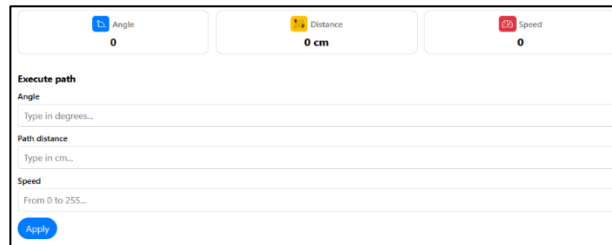

Figure 11 CAR STATUS WEB SECTION

Executed Path Section :

The executed path feature of the web application provides a visual representation of the path that the car will follow based on the inputted angle, distance, and speed, as shown in figure 10.2. This path is dynamically generated and displayed on the web app, giving users a clear understanding of the car's intended movement before execution.

This feature not only aids in planning and executing complex movements but also serves as a verification tool to ensure that the correct path is being followed as per the user's commands.
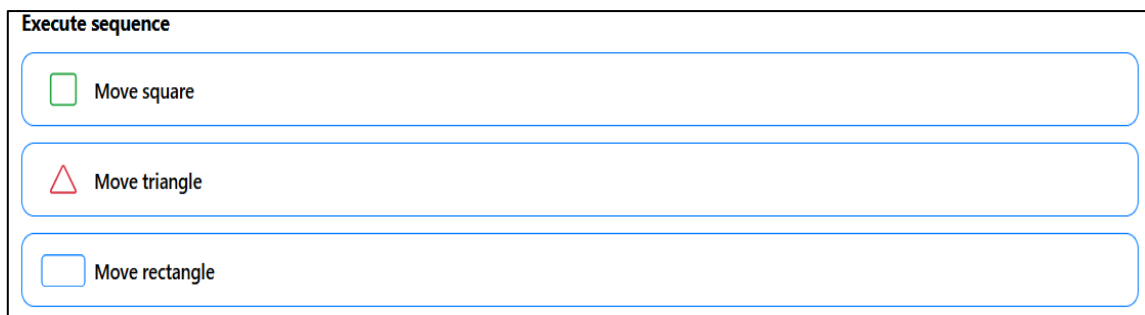

Figure 12 EXECUTE PATH WEB SECTION

Executed Sequence Section (Square, Rectangle, Triangle):

The web application supports predefined movement sequences such as squares, rectangles, and triangles, which can be selected by the user to command the car to execute specific geometric patterns, as shown in figure 10.3 . These sequences are pre-programmed into the system and can be triggered through simple button clicks on the web interface. Once a sequence is selected, the corresponding angles and distances are automatically calculated and sent to the car for execution. This feature simplifies the process of performing complex maneuvers, as the user does not need to manually input each angle and distance. Instead, the web app handles all calculations, ensuring accurate and consistent execution of the desired shape.

Figure 13 EXECUTE SEQUENCE WEB SECTION

Conclusion

This project demonstrates the potential for basic autonomous vehicle functionality by integrating hardware components such as Arduino, L298 motor driver, DC motors, MPU6050 sensor, and the ESP8266 Wi-Fi module. With the goal of developing a car capable of moving in straight lines and following geometric patterns autonomously, this project successfully outlines the use of microcontroller technology and sensors to achieve movement control. By using C/C++ code, it becomes possible to manipulate motor behavior, enabling the car to move forward, turn left or right, and follow predefined paths.

Additionally, the web application serves as a user-friendly interface for controlling the vehicle remotely, allowing users to adjust key parameters like speed, angle, and distance.

The application provides real-time feedback on the car's performance and displays the intended path visually. Moreover, with features such as geometric pattern execution, the project showcases how modern technology can be leveraged to simplify complex maneuvers, thus contributing to the foundation for more advanced autonomous driving systems.

This project serves as an initial step toward fully autonomous vehicle technology, demonstrating that basic autonomous functions can be achieved through the integration of essential components, programming, and user-friendly controls. Future developments could focus on enhancing sensor accuracy, improving decision-making algorithms, and scaling the system to more complex autonomous driving tasks.

Appendix

Project Files
https://github.com/abdo-shrief270/Graduation-Team-Tasks

Arduino Code
https://github.com/abdo-shrief270/Graduation-Team-Tasks/blob/master/Software/Car%20Software/Arduino%20Code/SELF-DRIVING_CAR_CODE/SELF-DRIVING_CAR_CODE.ino

ESP 8266 Code

https://github.com/abdo-shrief270/Graduation-Team-Tasks/blob/master/Software/Car%20Software/ESP%20Code/ESP_SERVER_SIDE/ESP_SERVER_SIDE.ino