

Ministry of Higher Education

Pyramids High Institute (PHI) for Engineering and Technology

Electronics and Communication Engineering Department



Title:

Autonomous driving and Vehicle tracking

Presented By:

Abdelrahman Shrief

Sondos Reda

Asmaa Mohamed

Mohamed Alaa

Mohamed Hossam

Mohamed Nageh

Mahmoud Gaballah

Moamen Mohamed

Supervised By:

Prof. Gamal El-Sheikh

Prof. Heba Emara

Cairo 2024_2025

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.

1. 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 1.1:

- 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.

- Level 2: 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.

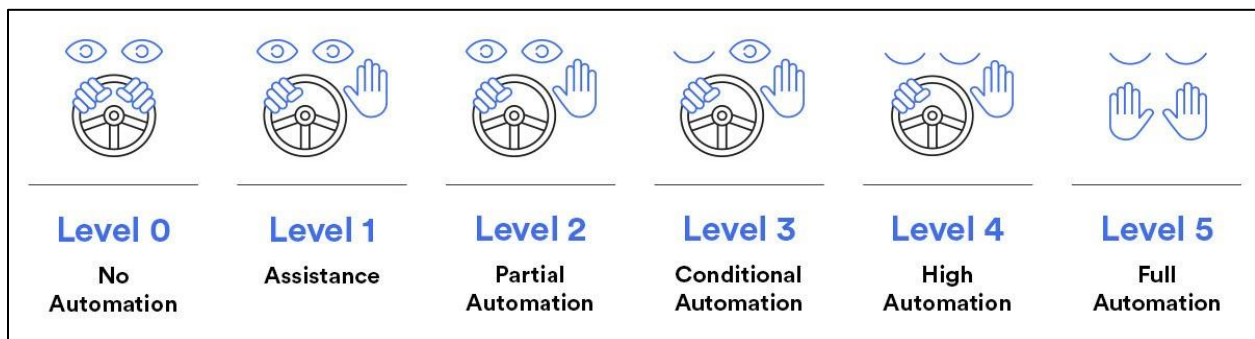


Fig 1.1

2. COMPONENTS OF PROJECT:

2.1 Arduino UNO

Arduino UNO is a microcontroller board based on the ATmega328P, as shown in figure 2.1. 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.

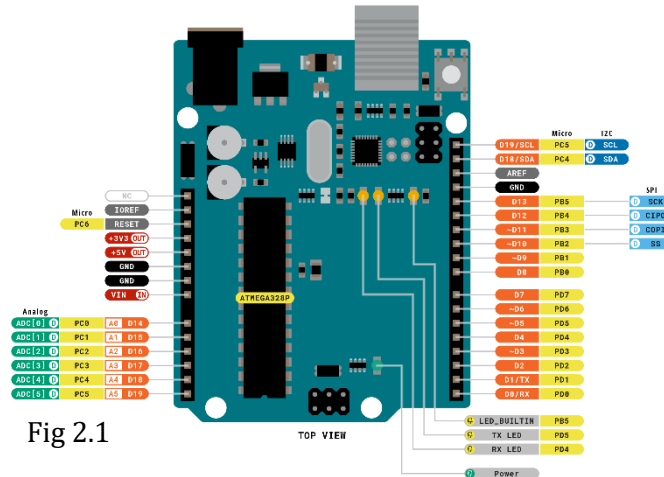


Fig 2.1

2.2 L298 motor driver:

The L298 motor driver is a popular dual H-bridge motor driver integrated circuit (IC) as shown in figure 2.2, 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.

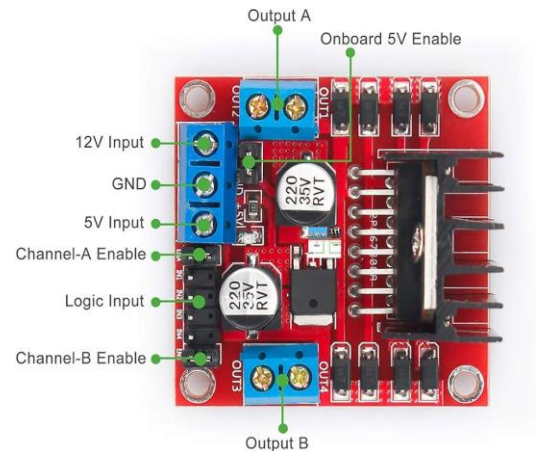


Fig 2.2

2.3 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 2.3.



Fig 2.3

2.4 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 2.4. Here are the specifications of the MPU-6050:

- Operating voltage: 2.375V to 3.46V.
- Gyroscope:
 1. Angular Velocity: Measures rotational motion in degrees per second (dps).
 2. Sensitivity: The sensitivity can be configured based on the selected range (e.g., ± 250 , ± 500 , ± 1000 , or ± 2000 dps).
- Temperature Sensor: The MPU-6050 has an integrated temperature sensor that measures the device's temperature.
- Accelerometer:
 1. Acceleration: Measures linear acceleration in three axes (X, Y, Z).
 2. Sensitivity: The sensitivity can be configured based on the selected range (e.g., $\pm 2g$, $\pm 4g$, $\pm 8g$, or $\pm 16g$).
- The Digital Low Pass Filter can be configured to filter out high-frequency noise from sensor data.

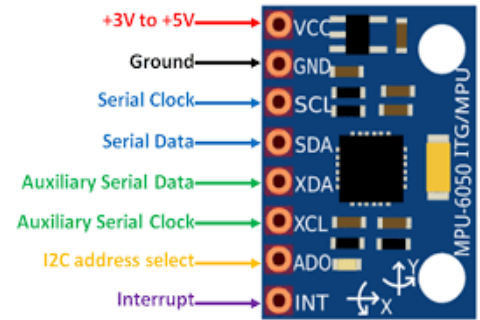


Fig 2.4

2.5 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 2.5. Below are the key details about the ESP8266:

- Memory:
 1. RAM: 160 KB of on-chip SRAM.
 2. ROM: 32 KB boot ROM.
 3. Flash: The chip supports external QSPI flash memory.
- Power:
 1. Operating Voltage: 3.0V to 3.6V.
 2. Power Consumption: Low power consumption with sleep modes.
- Programming Languages: Typically programmed in C/C++.

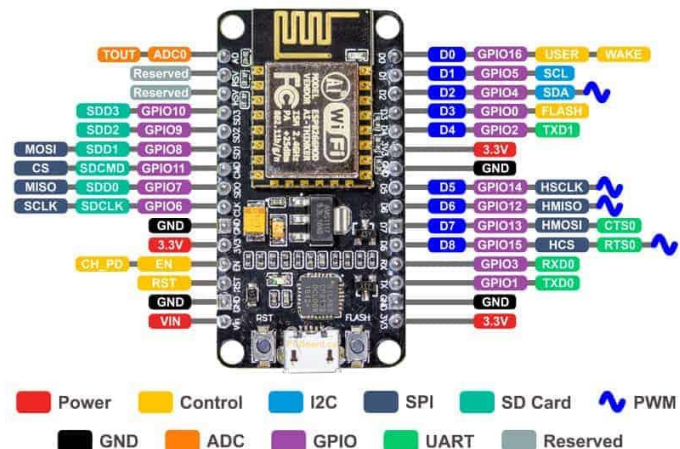


Fig 2.5

3. CIRCUIT DIAGRAM:

We will show how to connect the components to each other as shown in figure 3.1.

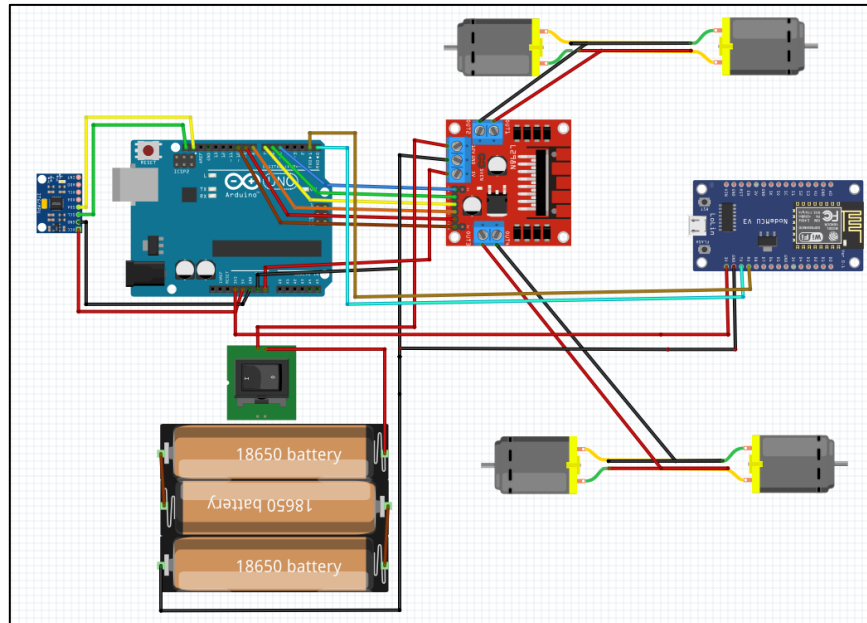


Fig 3.1

4. ARDUINO CAR CODE:

```
#include <Wire.h>
#include <MPU6050_light.h>
#define buadRate 9600
// String constants used for parsing serial commands
String sequenceCommand = "executeSequence" , pathCommand = "executePath";
String path_angleCommand = "angle" , path_angleDirCommand = "dir";
String path_distanceCommand = "distance" , path_speedCommand = "speed";
String path_substringCommand = "&" , path_equalCommand = "=";

int pathDetails[3]; // Array to store path params (angle, distance, speed)
MPU6050 mpu(Wire); // MPU6050 sensor object for reading IMU data
// Motor control pins
#define motorL1 7
#define motorL2 6
#define motorR1 9
#define motorR2 8
#define motorEN1 5
#define motorEN2 10
#define sensor 2
// Speed settings
#define speedTurn 80
#define speed_error_factor 11
```

```

// Shape dimensions and speeds
#define squareLength      100
#define squareSpeed       150
#define triangleLength    100
#define triangleSpeed     150
#define rectangleLength   100
#define rectangleWidth    150
#define rectangleSpeed    150

int steps = 0; // Tracks the number of steps (encoder counts)
int distance = 0; // Distance traveled based on encoder steps
float yaw; // Variable to hold the yaw angle from the MPU6050
void setup() {
    Serial.begin(buadRate); // Init serial communication at a baud rate of 9600
    Wire.begin(); // Initialize I2C communication
    mpu.begin(); // Initialize the MPU6050 sensor
    mpu.calcOffsets(); // Calibrate MPU6050 sensor
    mpu.update(); // Update sensor readings
    yaw = mpu.getAngleZ(); // Initialize yaw angle
    pinMode(sensor, INPUT); // Set sensor pin as input
    pinMode(motorR1, OUTPUT); // Set motor control pins as outputs
    pinMode(motorR2, OUTPUT);
    pinMode(motorL1, OUTPUT);
    pinMode(motorL2, OUTPUT);
    pinMode(motorEN1, OUTPUT);
    pinMode(motorEN2, OUTPUT);
    moveForward(); // Start by moving the car forward
}

void loop() {
    mpu.update(); // Continuously update MPU6050 sensor readings
    yaw = mpu.getAngleZ(); // Get the current yaw angle
    sendData("false", yaw, distance, 0); // Send current data over serial
    if (Serial.available()) {
        // Check if data is available on the serial port
        checkCommand(Serial.readStringUntil('\r')); // Read command and process it
    }
}

```


5. ARDUINO HELPER FUNCTION:

```
/**
 * Function to set the car's path and control its movement.
 * This involves turning the car to a specific angle and then moving forward
 * for a specified distance while adjusting the speed based on the yaw angle.
 * @param initial_angle - The target yaw angle to turn the car to.
 * @param path_distance - The distance the car should travel after turning.
 * @param speed - The speed at which the car should move.
 */
void SetCarPath(signed int initial_angle, int path_distance, char speed) {
    mpu.update(); // Update sensor readings
    turnCar(initial_angle, speedTurn); // Turn the car to the specified angle
    moveForward(); // Start moving forward
    distance = 0; // Reset distance traveled
    steps = 0; // Reset step count

    // Continue moving until the car reaches the specified distance
    while (distance < path_distance) {
        // Adjust motor speeds based on the current yaw angle
        analogWrite(motorEN1, speed + (yaw - initial_angle) * speed_adjust_factor + speed_error_factor);
        analogWrite(motorEN2, speed - (yaw - initial_angle) * speed_adjust_factor - speed_error_factor);

        if (digitalRead(sensor)) { // Check if the sensor detects a step
            steps += 1;
            distance = (steps / 90.0) * 100; // Convert steps to distance
            while (digitalRead(sensor)); // Wait for the sensor to clear
        }
        mpu.update(); // Update sensor readings
        yaw = mpu.getAngleZ(); // Update yaw angle
    }
    sendData("true", yaw, distance, speed); // Send final status
    instantStop(); // Stop the car
}
```



```

/**
 * Function to send the current status of the car via serial communication.
 * @param carIsMoving - Indicates if the car is currently moving ("true" or "false").
 * @param angle - The current yaw angle of the car.
 * @param dis - The distance traveled by the car.
 * @param speed - The current speed of the car.
 */
void sendData(String carIsMoving, signed int angle, int dis, int speed) {
    Serial.print("status:carIsMoving=");
    Serial.print(carIsMoving);
    Serial.print("&dir=");

    // Determine the direction based on the angle (right or left)
    if (angle < 0) {
        Serial.print("r"); // Right
    } else {
        Serial.print("l"); // Left
    }

    Serial.print("&angle=");
    Serial.print(abs(angle)); // Send the absolute value of the angle
    Serial.print("&distance=");
    Serial.print(dis); // Send the distance traveled
    Serial.print("&speed=");
    Serial.print(speed); // Send the current speed
    Serial.print("\r"); // End of message
}

/**
 * Function to execute a sequence of movements based on the command received.
 * @param str - The command string indicating the sequence to execute (e.g., "moveSquare").
 */
void performSequence(String str) {
    Serial.print("executeSequence:success=true&status=inProgress\r");
    mpu.update(); // Update sensor readings
    // Execute specific movement based on the command
    if (str == "moveSquare") {
        moveSquare(squareLength, squareSpeed);
    }
    if (str == "moveRectangle") {
        moveRectangle(rectangleLength, rectangleWidth, rectangleSpeed);
    }
}

```

```

if (str == "moveTriangle") {
    moveTriangle(triangleLength, triangleSpeed);
}
Serial.print("executeSequence:success=true&status=completed\r");
}
/**
 * Function to parse and execute commands received via serial communication.
 * It determines whether the command is a sequence or a path and acts accordingly.
 * @param str - The command string received via serial communication.
 */
void checkCommand(String str) {
    mpu.update(); // Update sensor readings
    // Check if the command is a sequence command
    if (str.indexOf(sequenceCommand) > -1) {
        str.remove(str.indexOf(sequenceCommand), str.indexOf(sequenceCommand)+sequenceCommand.length()+6);
        performSequence(str); // Execute the sequence command
    }
    // Check if the command is a path command
    if (str.indexOf(pathCommand) > -1) {
        String param, val;
        str.remove(0, str.indexOf(pathCommand) + pathCommand.length() + 1);
        // Parse the parameters from the command string
        while (str.length() > 0) {
            mpu.update();
            param = str.substring(0, str.indexOf(path_equalCommand)); // Get parameter name
            str.remove(0, str.indexOf(path_equalCommand) + 1);

            val = str.substring(0, str.indexOf(path_substringCommand)); // Get parameter value
            str.remove(0, str.indexOf(val) + val.length());
        }
        // Assign values to the pathDetails array based on parameter names
        if (param == path_angleCommand) {
            pathDetails[0] = val.toInt(); // Set angle
        }
        if (param == path_angleDirCommand && val == "r") {
            pathDetails[0] = -pathDetails[0]; // Reverse angle direction if "r"
        }
        if (param == path_distanceCommand) {
            pathDetails[1] = val.toInt(); // Set distance
        }
    }
}

```

```

if (param == path_speedCommand) {
    pathDetails[2] = val.toInt(); // Set speed
}
str.remove(0, str.indexOf(path_substringCommand) + 1);
}
Serial.print("executePath:success=true&status=inProgress\r");
SetCarPath(yaw + pathDetails[0], pathDetails[1], pathDetails[2]); // Execute the path command
Serial.print("executePath:success=true&status=completed\r");
}
}

/**
 * Function to move the car in a square pattern.
 * The car moves forward along the sides of a square, turning 90 degrees at each corner.
 * @param length - The length of each side of the square.
 * @param speed - The speed at which the car should move.
 */
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
}

/**
 * Function to move the car in a triangle pattern.
 * The car moves forward along the sides of an equilateral triangle, turning 120 degrees at each corner.
 * @param length - The length of each side of the triangle.
 * @param speed - The speed at which the car should move.
 */
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
}

```

```

/**
 * Function to move the car in a rectangular pattern.
 * The car moves forward along the sides of a rectangle, turning 90 degrees at each corner.
 * @param length - The length of the rectangle.
 * @param width - The width of the rectangle.
 * @param speed - The speed at which the car should move.
 */
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
}
/**
 * Function to turn the car to a specific yaw angle.
 * The car will adjust its direction until the desired angle is reached.
 * @param ang - The target yaw angle to turn the car to.
 * @param speed - The speed at which the car should turn.
 */
void turnCar(signed int ang, int speed) {
    while (1) {
        mpu.update(); // Update sensor readings
        yaw = mpu.getAngleZ(); // Get the current yaw angle

        if (yaw > (ang + 1)) { // If the car needs to turn right
            moveRight();      // Turn the car to the right
            applyCarSpeed(speed); // Apply the turning speed
        } else if (yaw < (ang - 1)) { // If the car needs to turn left
            moveLeft();       // Turn the car to the left
            applyCarSpeed(speed); // Apply the turning speed
        } else {
            instantStop(); // Stop the car when the desired angle is reached
            break;
        }
    }
}
}

```

```

/**
 * Function to apply a specified speed to both motors of the car.
 * @param speed - The speed value to apply to the motors (0-255).
 */
void applyCarSpeed(int speed) {
    analogWrite(motorEN1, speed); // Set speed for the right motor
    analogWrite(motorEN2, speed); // Set speed for the left motor
}

/**
 * Function to immediately stop the car by setting all motor pins high.
 */
void instantStop() {
    digitalWrite(motorR1, HIGH);
    digitalWrite(motorR2, HIGH);
    digitalWrite(motorL1, HIGH);
    digitalWrite(motorL2, HIGH);
}
// Movement control functions

/**
 * Function to move the car forward.
 * The right and left motors are set to rotate in the forward direction.
 */
void moveForward() {
    digitalWrite(motorR1, LOW);
    digitalWrite(motorR2, HIGH);
    digitalWrite(motorL1, LOW);
    digitalWrite(motorL2, HIGH);
}

/**
 * Function to turn the car to the right.
 * The right motor moves backward and the left motor moves forward.
 */
void moveRight() {
    digitalWrite(motorR1, HIGH);
    digitalWrite(motorR2, LOW);
    digitalWrite(motorL1, LOW);
    digitalWrite(motorL2, HIGH);
}

```

```

/**
 * Function to move the car backward.
 * The right and left motors are set to rotate in the reverse direction.
 */
void moveBackward() {
    digitalWrite(motorR1, HIGH);
    digitalWrite(motorR2, LOW);
    digitalWrite(motorL1, HIGH);
    digitalWrite(motorL2, LOW);
}

/**
 * Function to turn the car to the left.
 * The right motor moves forward and the left motor moves backward.
 */
void moveLeft() {
    digitalWrite(motorR1, LOW);
    digitalWrite(motorR2, HIGH);
    digitalWrite(motorL1, HIGH);
    digitalWrite(motorL2, LOW);
}

```

6. HOW THE CAR MOVES (FORWARD, RIGHT, LEFT)?

The movement 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 move the car:

1. Forward Movement:

- The function `moveForward()` sets the right motor (motorR1 and motorR2) and the left motor (motorL1 and motorL2) to rotate forward, as shown in code figure6.1.
- 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);
}

```

Fig 6.1

2. 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 figure6.2.
- 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);  
}
```

Fig 6.2

3. 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 figure6.3.
- This causes the car to pivot to the left.

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

Fig 6.3

7. How the car moves 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:

1. 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).

2. 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:

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

- The motors' speeds are adjusted dynamically to keep the car moving straight, based on the yaw angle, as shown in code figure 7.1:

```
// Continue moving until the car reaches the specified distance
while (distance < path_distance) {
    // Adjust motor speeds based on the current yaw angle
    analogWrite(motorEN1, speed+(yaw-initial_angle)*speed_adjust_factor+speed_error_factor);
    analogWrite(motorEN2, speed-(yaw-initial_angle)*speed_adjust_factor-speed_error_factor);
}
```

Fig 7.1

3. 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, 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.

8. Set a path for the car to move (square, rectangle, triangle)

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.

1. Moving in a Square Path:

In the moveSquare() function, as shown in code figure 8.1 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  
}
```

Fig 8.1

2. Moving in a Rectangle Path:

- The moveRectangle() function works similarly as shown in code figure 8.2 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  
}
```

Fig 8.2

3. 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 figure 8.3.
- SetCarPath(yaw + 120, length, speed): The car turns 120 degrees after each side, forming an equilateral triangle.

```
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  
}
```

Fig 8.3

9. MATLAB CODE:

We display the car's data as it moves using matlab, as shown in figure 9.1.

```
clc,clear,close all  
load gp_task_1_data.txt  
x=gp_task_1_data';  
  
subplot(211)  
plot(x(1,:), 'r', 'linewidth', 2); grid on  
title('distance')  
subplot(212)  
plot(x(2,:), 'b', 'linewidth', 2); grid on  
title('angle')
```

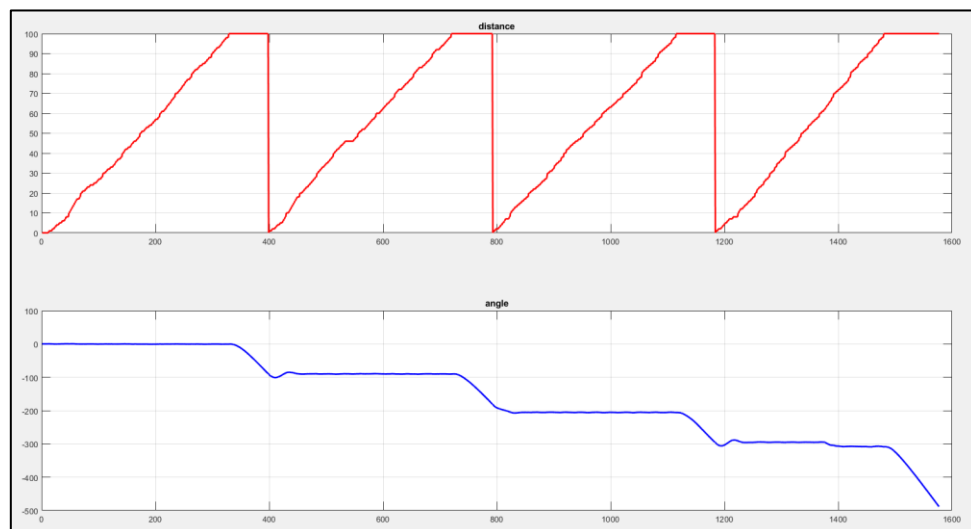


Fig 9.1

10. 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.

a. 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.

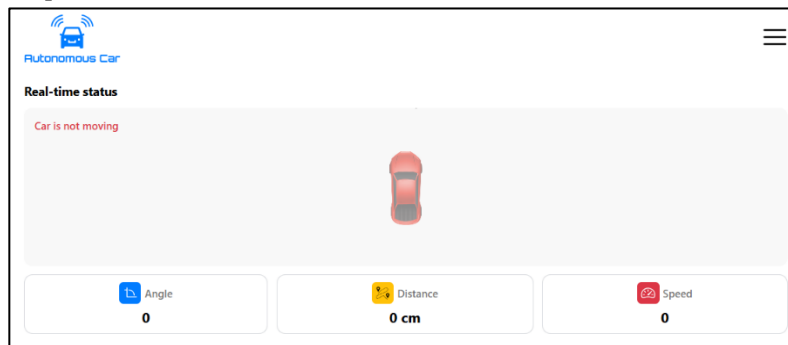


Fig 10.1

b. Executed Path:

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.

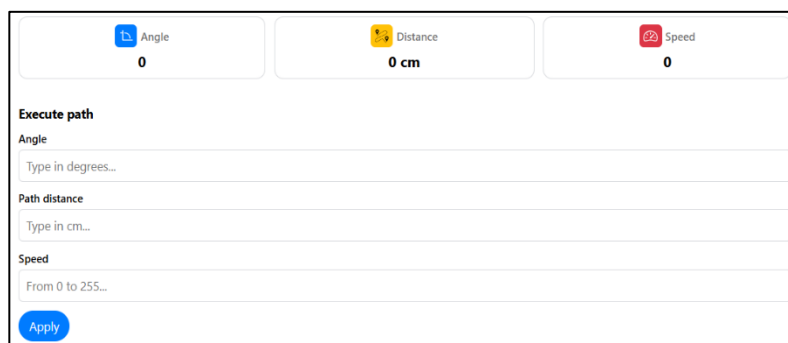
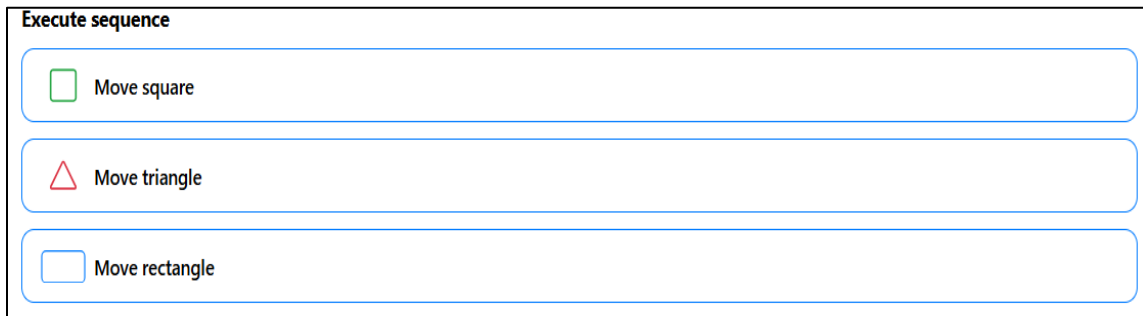


Fig 10.2

c. Executed Sequence (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.



The image shows a web interface titled "Execute sequence". It contains three buttons stacked vertically. The first button has a green square icon and the text "Move square". The second button has a red triangle icon and the text "Move triangle". The third button has a blue rectangle icon and the text "Move rectangle".

Fig 10.3

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.