ECE504 : INTERNET OF THINGS
FINAL PROJECT REPORT

“Gesture-controlled car”

Submitted To Prof. Anurag Lakhlani

Name	Enrollment No.
Preksha Morbia	AU2120225
Aarya Parekh	AU2120075
Shubham Shah	AU2120064

CONTENTS

Chapters:

1. Introduction
2. Market Survey
3. Block Diagram
4. Circuit Diagram
5. Arduino UNO/Mega OR Raspberry Pi Features
6. Program Flow Chart
7. Sensors
8. Actuators and Displays
9. Details of Communication Protocol
10. Complete Program
11. Summary
12. References

Appendices:

A. Datasheets
B. Programming Review
C. Trouble-Shooting
D. Real-Life Mounting of our System
E. Real-Life Scenarios

CHAPTER 1

❖ Introduction and Motivation:

Overview of the Project:

This project focuses on developing a gesture-controlled car using IoT technology. Gesture recognition systems are increasingly becoming a key part of human-computer interaction. Our system interprets specific gestures to control the car's movement, offering an intuitive and innovative solution for remote vehicle operation. The core idea is to enhance accessibility and control simplicity in IoT-enabled devices, making it easier to operate remotely or in constrained environments.

Uniqueness of the Project:

What makes this project unique is our approach to integrating gesture recognition with vehicular control using affordable and readily available IoT components. Initially, we explored alternative control methods such as EEG (electroencephalography) and EMG (electromyography) to make the system even more advanced. However, interpreting the raw data from EEG and EMG proved complex and time-intensive due to noise and the difficulty in extracting reliable signals. Hence, we pivoted to a more robust and user-friendly approach using motion-based gestures for control.

Learning Goals:

By undertaking this project, we aim to:

1. Deepen our understanding of gesture recognition technologies and their integration with IoT.
2. Learn about the interfacing of sensors like accelerometers and gyroscopes with microcontrollers.
3. Explore the challenges and solutions in real-time communication and control.
4. Build a complete hardware-software system, enhancing both practical and theoretical knowledge.

Usefulness of the Project:

This project has several practical applications:

- Accessibility: It can be adapted for users with limited mobility to operate devices or vehicles using simple gestures.
- Remote Operation: The system can be extended to control devices or vehicles remotely in hazardous or hard-to-reach areas.
- Learning Tool: This project serves as an excellent educational tool for understanding IoT systems and gesture recognition.
- Customization: The system can be tailored to control different IoT-enabled devices beyond vehicles, making it versatile and scalable.

CHAPTER 2

❖ Market Survey

"Myo-Controlled Robotic Arm" (Thalmic Labs, 2018)

This project utilizes the Myo armband to detect gestures through electromyography (EMG) and motion sensors, enabling precise control of robotic arms. The system is effective in applications such as rehabilitation and industrial automation. However, it relies on wearable devices and requires calibration for accurate gesture recognition. Unlike our Raspberry Pi-based robot, which uses a camera and OpenCV for gesture detection, the Myo system is limited to recognizing muscle signals and lacks flexibility for broader use cases. Our approach is also more cost-effective and eliminates the need for specialized hardware like the Myo armband.

"Gesture Control with Leap Motion Controller" (Leap Motion, 2020)

This system employs infrared cameras to track hand gestures in 3D space, offering a contact-free interaction method. It is primarily used for virtual reality applications and robotic control. While the Leap Motion system is easy to use and does not require wearables, it struggles in low-light environments and has a limited operational range of up to 2 feet. In contrast, our Raspberry Pi-based robot leverages image processing and a camera, which can be tuned to perform effectively in varying lighting conditions and offers a more flexible setup for educational and assistive purposes.

"Real-Time Gesture Recognition Using Raspberry Pi".

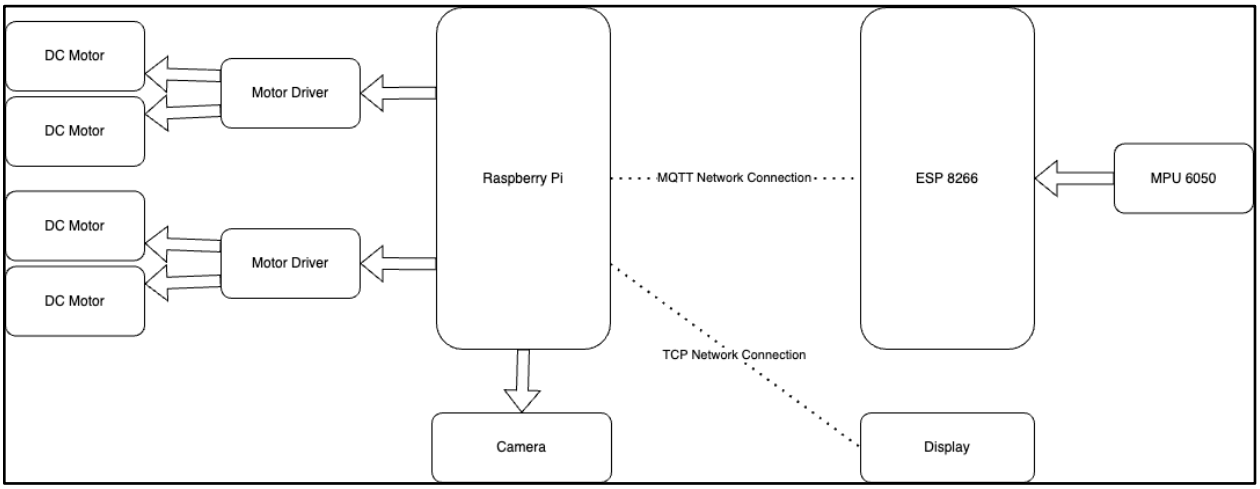
This research explores a Raspberry Pi-based robotic system that uses a camera to detect hand gestures via image processing. It is designed for educational and experimental purposes. While this system shares similarities with our project, such as affordability and modular design, it does not integrate features like additional sensors for obstacle detection. Our project incorporates ultrasonic sensors for obstacle avoidance and infrared sensors for improved navigation, making it more robust for real-world applications.

Conclusion:

The market survey highlights various approaches to gesture-controlled systems, such as the Myo armband and Leap Motion Controller, which rely on specialized hardware or specific environmental conditions. Comparatively, Raspberry Pi-based systems provide an affordable and accessible solution for gesture recognition, often used in research and education.

CHAPTER 3

❖ PROJECT BLOCK DIAGRAM:



Input components:

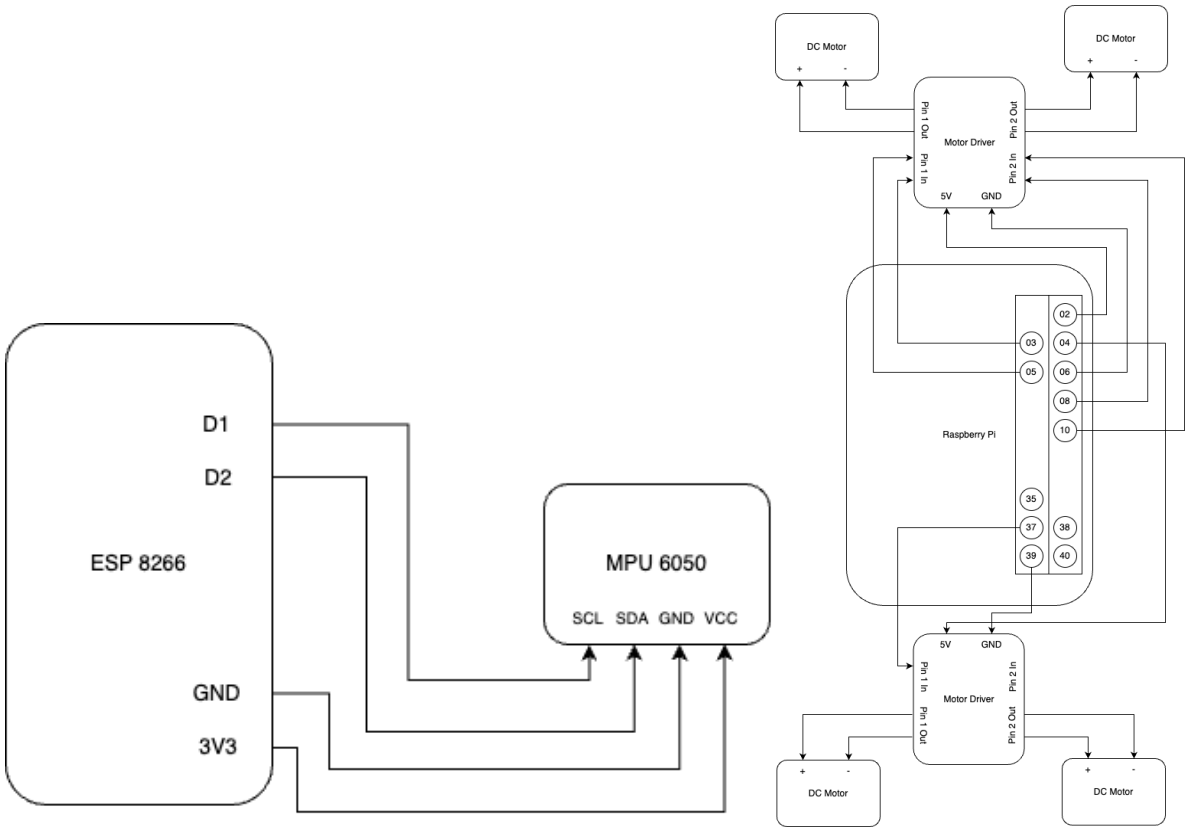
- **MPU6050 (Accelerometer + Gyroscope):** Captures hand movement and orientation to detect gestures.

Output components:

- **Motor Driver (L298N) and DC Motors:** Receives signals from the Raspberry Pi to control the movement of the car (forward, backward, left, right).
- **Display (Live Streaming via Raspberry Pi Camera):** Displays real-time video stream of the car's surroundings.

CHAPTER 4

❖ Circuit Diagrams



4.1 Components

1. **Raspberry Pi (RPi)**
    - Function: Serves as the central control unit, running the gesture recognition software and processing inputs from the accelerometer and gyroscope.
    - Purpose: Processes sensor data and sends control signals to the motor driver for car movement.

2. **Accelerometer + Gyroscope (MPU6050)**
    - Function: Detects the orientation and movement of the user's hand in 3D space (X, Y, Z axes).
    - Purpose: Captures raw gesture data and transmits it to the Raspberry Pi via I2C for gesture recognition and motion control.

3. **Motor Driver (L298N)**
    - Function: Controls the direction and speed of the DC motors.
    - Purpose: Receives processed commands from the Raspberry Pi to drive the motors forward, backward, left, or right.

4. **DC Motors**
    - Function: Converts electrical signals from the motor driver into mechanical motion for the car's wheels.
    - Purpose: Facilitates the physical movement of the car.

5. **Power Supply**
    - Function: Provides power to all components, including the Raspberry Pi, motor driver, and motors.
    - Purpose: Ensures uninterrupted operation of the system.

6. **Camera**

The connections between these components are shown in the circuit diagram, with emphasis on the data transmission from the MPU6050 to the Raspberry Pi and the control signals sent to the motor driver for car movement.

4.2 List of Sensors/Actuators/Displays

➔ **Accelerometer + Gyroscope (MPU6050)**

➔ **Camera**
➔ **DC Motors**
➔ **Motor Driver**
➔ **Screen Display**

4.4 Why These Sensors Were Selected

- **Accelerometer + Gyroscope (MPU6050):**
  - Offers combined functionality, high precision, and low power consumption, making it ideal for gesture-controlled systems.
  - Provides accurate 3D motion data via I2C, simplifying integration with the Raspberry Pi.
- **DC Motors:**
  - Reliable, simple to use, and capable of providing sufficient torque and speed to drive the car.
- **Motor Driver (L298N):**
  - Allows for control of both speed and direction of the motors and integrates well with the Raspberry Pi.

CHAPTER 5

5.1 Overview of the Raspberry Pi

In this project, we have used the **Raspberry Pi** as the central control unit. Below are its key features, with a focus on those relevant to the gesture-controlled car:

- **GPIO Pins:** 40 GPIO pins allow integration with sensors (e.g., MPU6050) and motor drivers (e.g., L298N).
- **Processing Power:** Equipped with a quad-core ARM processor (depending on the model), enabling real-time data processing for gesture recognition.
- **Operating System:** Runs a full-fledged Linux-based OS (e.g., Raspbian), providing flexibility to implement Python-based gesture detection algorithms.
- **Connectivity:** Supports Wi-Fi, Bluetooth, and Ethernet, though only GPIO-based wired connections were used in this project.
- **Cost-Effective:** Affordable for IoT and robotics projects compared to industrial-grade solutions.
- **Compatibility:** Seamlessly works with a variety of sensors and actuators using libraries available in Python or C++.

5.2 Comparison of IoT Platforms

The table below compares the Raspberry Pi with other popular IoT development platforms:

Feature	Raspberry Pi	Arduino UNO/Mega	BeagleBone	Intel Galileo	Intel Edison
Processor	Quad-core ARM Cortex (RPi 4)	8-bit AVR (UNO), 32-bit ARM (Mega)	ARM Cortex-A8	Intel Quark	Intel Atom
Clock Speed	Up to 1.5 GHz	16 MHz (UNO), 84 MHz (Mega)	1 GHz	400 MHz	500 MHz
RAM	2GB to 8GB (RPi 4)	None	512MB	256MB	1GB
GPIO Pins	40	14 (UNO), 54 (Mega)	92	20	40
Connectivity	Wi-Fi, Bluetooth, Ethernet	None (requires modules)	Wi-Fi, Ethernet	Ethernet	Wi-Fi, Bluetooth
Cost	$35-$75	$20-$40	$50-$100	$50-$80	$50-$100

5.3 Strengths and Weaknesses of Platforms

Platform	Strengths	Weaknesses
Raspberry Pi	Powerful, supports multitasking and complex programs, large community, versatile connectivity.	Higher power consumption, overkill for simple projects, requires external hardware for ADC.
Arduino	Simple, low power consumption, ideal for beginner-level projects, great for real-time control.	Limited processing power, no native OS, lacks networking features (needs additional modules).
BeagleBone	High processing power, extensive GPIO, built-in storage, good for industrial IoT applications.	More expensive, less beginner-friendly, smaller community support.
Intel Galileo	Combines Arduino simplicity with x86 compatibility, decent processing power for IoT projects.	Slower processing compared to Raspberry Pi and BeagleBone, smaller user base.
Intel Edison	Compact, built-in Wi-Fi/Bluetooth, excellent for wearable and compact IoT devices.	Expensive, harder to source, discontinued, requires custom breakout boards.

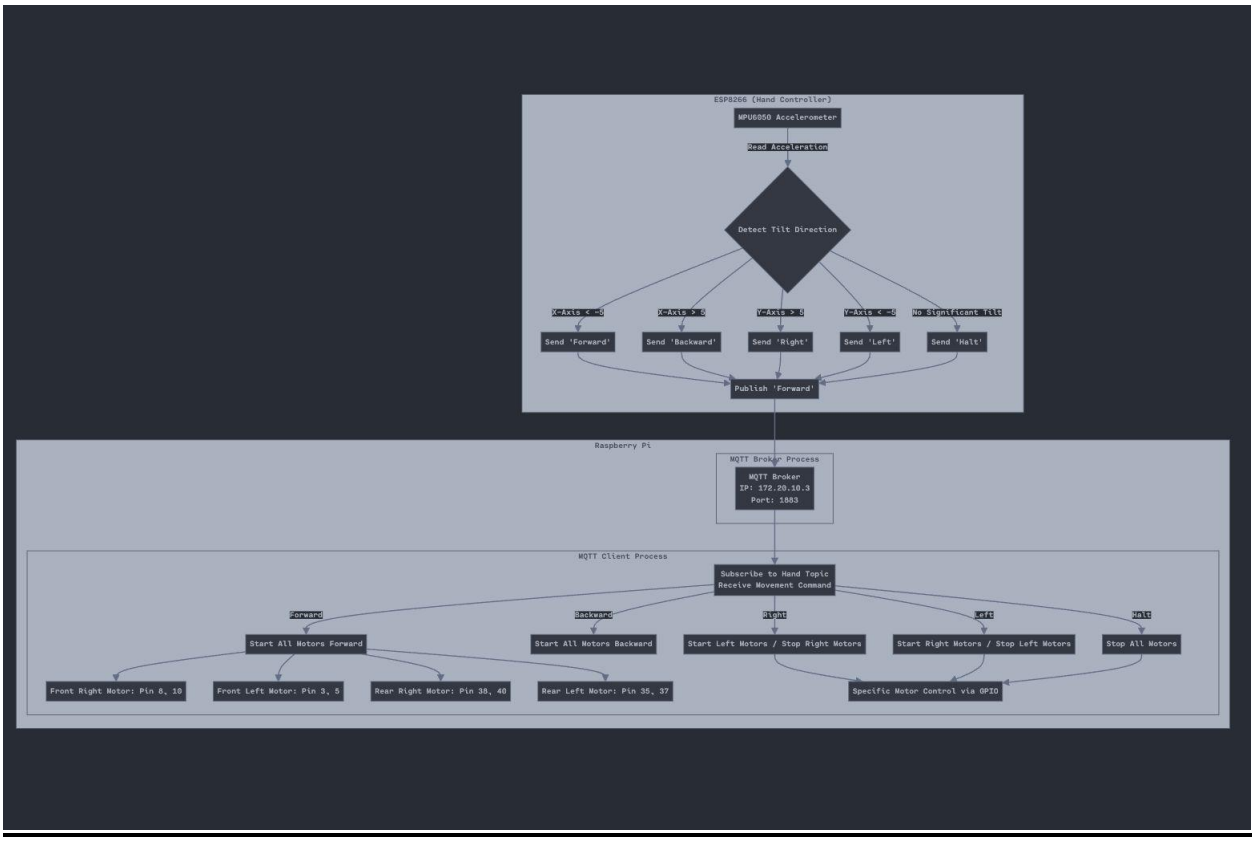
5.4 Platform Selection Rationale

We selected the Raspberry Pi for this project due to the following reasons:

1. **Processing Power**: It handles real-time gesture data processing from the MPU6050 effectively.
2. **Connectivity**: Its GPIO pins allow direct integration with the motor driver and accelerometer without additional hardware.
3. **Flexibility**: The ability to run Python scripts simplified the implementation of gesture recognition algorithms.
4. **Cost-Effectiveness**: Provides an optimal balance between cost and functionality for IoT robotics projects.

CHAPTER 6

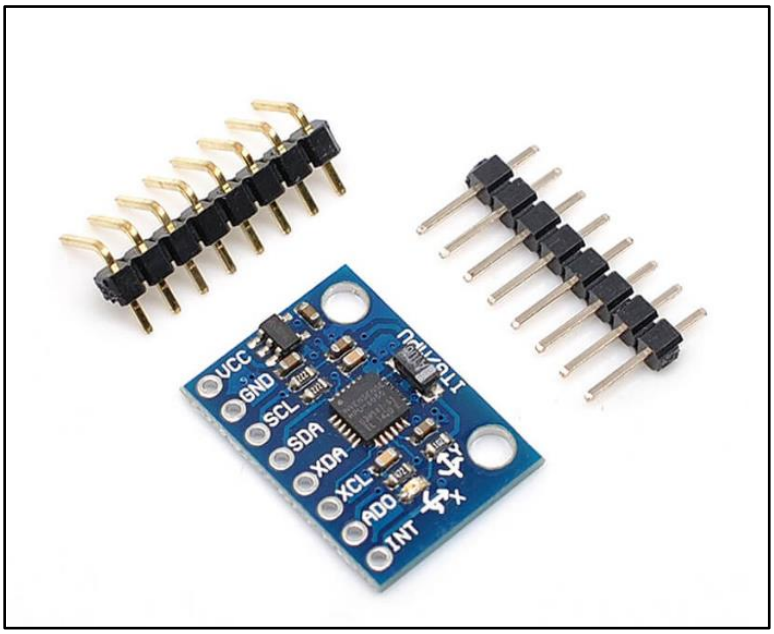
❖ PROGRAM FLOWCHART:



CHAPTER 7

DETAILS OF SENSORS:

7.1 Accelerometer + Gyroscope (MPU6050)



Attribute	Details
Operating Principle	Measures hand movement and orientation using a 3-axis accelerometer and 3-axis gyroscope. Data is transmitted to Raspberry Pi via I2C for gesture recognition.
Dimensions	Approx. 20mm x 15mm (with breakout board)
Weight	3g
Power Supply	3.3V - 5V
Measurement Range	Accelerometer: ±2g to ±16g; Gyroscope: ±250°/s to ±2000°/s
Current Consumption	~5mA
Pins/Connections	**VCC**: 3.3V to 5V; **SCL**: I2C Clock; **SDA**: I2C Data; **GND**: Ground
Interface	I2C
Application	Gesture control and motion tracking.

7.2 Camera (Raspberry Pi Camera Module)

Attribute	Details
Operating Principle	Uses a CMOS sensor to capture light and convert it into digital signals. Streams real-time video through the CSI interface to the Raspberry Pi.
Dimensions	25mm x 24mm x 9mm
Weight	3g
Power Supply	5V (via CSI ribbon cable)
Resolution	8 MP, 1080p at 30fps, 720p at 60fps
Current Consumption	~250mA
Pins/Connections	Connected to Raspberry Pi via CSI connector for video and power transfer.
Interface	CSI (Camera Serial Interface)
Application	Live video streaming and surveillance.

7.3 Interfacing Details

Sensor/Component	Pins Used	Connection Description
MPU6050 (Accelerometer + Gyroscope)	SDA, SCL, VCC, GND	Connected via I2C to Raspberry Pi GPIO2 (SDA) and GPIO3 (SCL). Powered by 3.3V.
Raspberry Pi Camera Module	CSI Ribbon Port	Connected to the dedicated CSI interface of the Raspberry Pi for video streaming.
Motor Driver (L298N)	GPIO Pins (IN1-4)	Raspberry Pi controls motors through GPIO pins on L298N driver.

7.4 Summary of Inputs and Outputs:

- **Input:** The MPU6050 (Accelerometer + Gyroscope) detects hand gestures and orientation, sending data to the Raspberry Pi.
- **Output:** The system controls the Motor Driver (L298N) and DC Motors for movement and streams live video via the Raspberry Pi Camera.

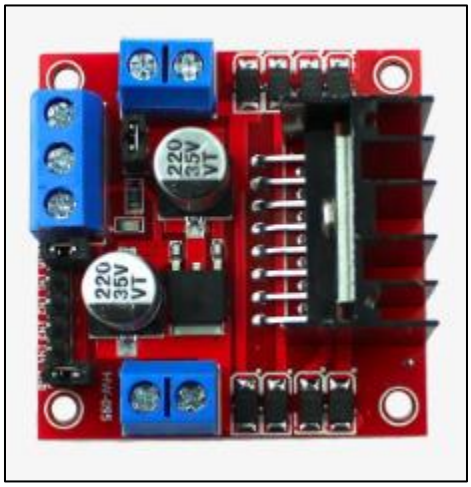
CHAPTER 8

❖ DETAILS OF ACTUATORS:

8.1 DC Motors

Attribute	Details
Operating Principle	DC motors convert electrical energy into mechanical motion through the interaction of magnetic fields inside the motor. The direction of rotation is controlled by reversing the polarity of the input voltage, while speed is controlled by varying the input voltage.
Dimensions	Varies, but typical small motors: 35mm x 28mm
Power Ratings	Operating Voltage: 6V to 12V (depends on motor type and application). Current consumption depends on load.
Pins/Connections	Typically 2 pins for controlling direction and 1 for speed control via the motor driver.
Type of Interface	Controlled by the Raspberry Pi via a motor driver (L298N) using GPIO pins.
Application	Used for movement in robotic projects such as driving the wheels of a gesture-controlled car.

8.2 Motor Driver - L298N



Attribute	Details
Operating Principle	The L298N motor driver is used to control the direction and speed of DC motors. It takes control signals from the Raspberry Pi and switches the direction of current flow to drive the motors forward or backward.
Dimensions	43mm x 43mm x 27mm
Power Ratings	Operating Voltage: 5V to 12V, Maximum Current: 2A per channel for each motor.
Pins/Connections	IN1, IN2, IN3, IN4: Control pins; ENA, ENB: Enable pins for motor operation; VCC: Power supply for motors; GND: Ground.
Type of Interface	Controlled via GPIO pins from Raspberry Pi.
Application	Used to control the motors of the robot by regulating motor speed and direction.

CHAPTER 9

❖ Communication Protocol

MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for resource-constrained devices and low-bandwidth, high-latency, or unreliable networks. Its simplicity and efficiency make it ideal for Internet of Things applications and scenarios requiring real-time communication between devices.

Key Features of MQTT:

1. Lightweight and Efficient: MQTT minimizes overhead, making it suitable for devices with limited computational and memory resources.

2. Publish/Subscribe Model: Communication occurs between clients through a central broker:

- Publisher: Sends messages to the broker.
- Subscriber: Receives messages from the broker based on topic subscriptions.
- Broker: Manages message distribution between publishers and subscribers.

3. Topic-based Messaging: Messages are categorized using hierarchical topic strings (e.g., esp8266/hand). Subscribers can listen to specific topics or use wildcards (e.g., esp8266/# to subscribe to all subtopics under home).

4. Quality of Service (QoS): MQTT offers three levels of message delivery assurance:

- QoS 0: At most once (fire-and-forget).
- QoS 1: At least once (delivery is guaranteed, but duplicates may occur).
- QoS 2: Exactly once (ensures no duplicates).

5. Retained Messages: The broker can retain the last message on a topic, allowing new subscribers to immediately receive the latest update.

6. Will Messages: In case a client disconnects unexpectedly, the broker can automatically publish a pre-defined message to a topic, indicating device status.

7. Security: Supports SSL/TLS for encryption and username/password authentication for secure communication.

Advantages of MQTT:

- Low power consumption, enabling use in battery-powered devices.
- Efficient in scenarios with intermittent network connectivity.
- Scalable, supporting large numbers of connected devices.
- Simple to implement and configure.

Protocol Workflow:

1. Connection Establishment: A client initiates a connection with the broker, specifying its credentials, client ID, and optional "Last Will" message.

2. Topic Subscription: A client subscribes to one or more topics to receive relevant messages.

3. Message Publishing: A publisher sends a message to the broker, specifying the topic. The broker routes the message to all subscribers of the topic.

4. Disconnection: Clients can gracefully disconnect by sending a disconnect message to the broker.

Use in the Project:

In our project, MQTT is used to enable communication between ESP8266 modules and the Raspberry Pi 3. The ESP devices act as publishers, sending sensor data to the MQTT broker hosted on the Raspberry Pi. The Raspberry Pi processes the data and also acts as a client and subscribes to these data and commands the car.

- **Broker Implementation:** The broker is hosted using Mosquitto.

- **Topics Defined:** esp8266/hand

- **QoS Level:** 1

CHAPTER 10

❖ COMPLETE PROGRAM:

There are two main code files:

mv.py on raspberry pi and
pubsub_client.py on esp 8266
mv.py on raspberry pi:

```
from enum import Enum
import RPi.GPIO as gpio
gpio.setmode(gpio.BOARD)

mfr_pins = (8, 10)      ## motor front right pins
mfl_pins = (3, 5)       ## motor front left pins

mrr_pins = (38, 40)     ## motor rear right pins
mrl_pins = (35, 37)     ## motor rear left pins

servo_pin = (12, )

all_mpins = mfr_pins + mfl_pins + mrr_pins + mrl_pins
all_pins = all_mpins + servo_pin

gpio.setup(
      all_pins,
      gpio.OUT
  )

class Motor(Enum):
```

```
  mfr = 0
  mfl = 1

  mrr = 2
  mrl = 3

def forward():

  start_motor(Motor.mfr)
  start_motor(Motor.mfl)
  start_motor(Motor.mrr)
  start_motor(Motor.mrl)

def backward():

  start_motor(Motor.mfr, True)
  start_motor(Motor.mfl, True)
  start_motor(Motor.mrr, True)
  start_motor(Motor.mrl, True)

def right():

  start_motor(Motor.mfl)
  start_motor(Motor.mrl)
  stop_motor(Motor.mfr)
  stop_motor(Motor.mrr)

def left():

  start_motor(Motor.mfr)
  start_motor(Motor.mrr)
```

```
  stop_motor(Motor.mfl)
  stop_motor(Motor.mrl)

def halt():

  stop_motor(Motor.mfr)
  stop_motor(Motor.mfl)
  stop_motor(Motor.mrr)
  stop_motor(Motor.mrl)

def isMotorValid(motor: Motor) -> tuple:

  if motor == Motor.mfr:
      pins = mfr_pins

  elif motor == Motor.mfl:
      pins = mfl_pins

  elif motor == Motor.mrr:
      pins = mrr_pins

  elif motor == Motor.mrl:
      pins = mrl_pins

  else:
      raise Exception(f"Invalid Motor: {motor}")

  return pins

def start_motor(motor: Motor, reverse: bool = False) -> None:
```

```
    pins = isMotorValid(motor)

    if not reverse:
        high, low = pins
    else:
        low, high = pins

    print(high, low)
    gpio.output(high, gpio.HIGH)
    gpio.output(low, gpio.LOW)

def stop_motor(motor: Motor) -> None:

    pins = isMotorValid(motor)

    # gpio.output(pins[0], gpio.LOW)
    # gpio.output(pins[1], gpio.LOW)
    gpio.cleanup(pins)
   gpio.setup(pins, gpio.OUT)

if __name__ == "__main__":

    try:
        while True:

            i = input()
            if i == 'w':
                forward()
            elif i == 's':
                backward()
            elif i == 'a':
```

```
            left()
        elif i == 'd':
            right()
        elif i == 'h':
            halt()

  except:
      gpio.cleanup()
```

pubsub_client.py on esp 8266:

```
// ------------ MQTT Imports -----------------

#include "PubSubClient.h" // Connect and publish to the MQTT broker
#include "ESP8266WiFi.h"  // Enables the ESP8266 to connect to the local network (via WiFi)

// ------------ MQTT Imports -----------------
// ------------ Hand Imports -----------------

#include <MPU6050.h>

// ------------ Hand Imports -----------------

// ------------ MQTT Helpers -----------------

String currState = "";

// WiFi
const char* ssid = "aarya";                 // Your personal network SSID
const char* wifi_password = "12345678"; // Your personal network password

// MQTT
```

```
const char* mqtt_server = "172.20.10.3";  // IP of the MQTT broker
const char* hand_topic = "esp8266/hand";
// const char* head_topic = "esp8266/head";
const char* mqtt_username = "admin"; // MQTT username
const char* mqtt_password = "admin"; // MQTT password
const char* clientID = "esp8266"; // MQTT client ID

// Initialise the WiFi and MQTT Client objects
WiFiClient wifiClient;
// 1883 is the listener port for the Broker
PubSubClient client(mqtt_server, 1883, wifiClient);

void connect_MQTT() {
 Serial.print("Connecting to ");
 Serial.println(ssid);

 // Connect to the WiFi
 WiFi.begin(ssid, wifi_password);

 // Wait until the connection has been confirmed before continuing
 while (WiFi.status() != WL_CONNECTED) {
   delay(50);
   Serial.print(".");
 }

 // Debugging - Output the IP Address of the ESP8266
 Serial.println("WiFi connected");
 Serial.print("IP address: ");
 Serial.println(WiFi.localIP());

 // Connect to MQTT Broker
```

```
 // client.connect returns a boolean value to let us know if the connection was
successful.
 // If the connection is failing, make sure you are using the correct MQTT Username and
Password (Setup Earlier in the Instructable)
 if (client.connect(clientID)) {
   Serial.println("Connected to MQTT Broker!");
 }

 else {
   Serial.println("Connection to MQTT Broker failed...");
 }
}

// ------------ MQTT Helpers ----------------
// ------------ Hand Helpers ----------------

MPU6050 mpu;
String readCommand() {
  Vector normAccel = mpu.readNormalizeAccel();

 if (normAccel.XAxis < -5) {
   return "Forward";
 } else if (normAccel.XAxis > 5) {
   return "Backward";
 } else if (normAccel.YAxis > 5) {
   return "Right";
 } else if (normAccel.YAxis < -5) {
   return "Left";
 } else {
   return "Halt";
 }
```

```
}

// ------------- Hand Helpers -----------------

void setup() {
  // ------------- MQTT Setup ---------------

 connect_MQTT();

 // ------------- MQTT Setup -----------------
 // ------------- Hand Setup -----------------
  // put your setup code here, to run once:
 Serial.begin(115200);

 Serial.println("Initialize MPU6050");

 while(!mpu.begin(MPU6050_SCALE_2000DPS, MPU6050_RANGE_2G))
 {
   Serial.println("Could not find a valid MPU6050 sensor, check wiring!");
   delay(500);
 }

 // ------------- Hand Setup -----------------
}

void loop() {
 // put your main code here, to run repeatedly:
 while (WiFi.status() != WL_CONNECTED) {
   connect_MQTT();
 }
```

```
 String newState = readCommand();
 if (newState != currState) {
   if (client.publish(hand_topic, newState.c_str())) {
     Serial.print("State Updated to ");
     Serial.println(newState);
     currState = newState;
   }
 }
 delay(10);
}
```

CHAPTER 11

SUMMARY OF WORKING OF PROJECT:

1. **Gesture Control with MPU6050:**
   The car is controlled through hand gestures detected by the MPU6050 accelerometer and gyroscope. The accelerometer senses the tilt of the hand, and based on the tilt direction, the Raspberry Pi processes these inputs to control the movement of the car.
   - Forward/Backward Movement: Tilting the hand forward or backward moves the car in the corresponding direction.
   - Left/Right Turns: Tilting the hand left or right steers the car accordingly.

2. **Motor Control with L298N Motor Driver:**
   The L298N motor driver receives the processed signals from the Raspberry Pi and drives the DC motors to move the car in the desired direction. The motors respond based on the forward, backward, left, or right commands.

3. **Live Streaming with Raspberry Pi Camera:**
   The Raspberry Pi Camera streams live video, providing real-time feedback to the user. The video feed is displayed via the Raspberry Pi, allowing the user to monitor the car's surroundings during operation.

4. **Real-time Feedback and Control:**
   The system enables intuitive control with hand gestures, while the live video provides continuous updates on the car's environment. The integration of these technologies makes the gesture-controlled car interactive and user-friendly.

Project Timeline

1. **Requirement Gathering and System Design (Weeks 1–2)**:
   The project started with research and understanding the components, followed

by designing the overall system, including the hardware setup and software integration.

2. **Component Setup and Integration (Weeks 3–4)**:
   The hardware components were assembled, including the **MPU6050**, **Raspberry Pi**, **DC motors**, and **motor driver**. The software was configured to interface with these components, allowing basic motion control.

3. **Coding and Integration of Features (Weeks 5–7)**:
   The gesture recognition software was developed, and live streaming was integrated into the Raspberry Pi. This phase included writing Python code for the communication between the **MPU6050** and **Raspberry Pi**, and controlling the motors.

4. **Testing and Debugging (Weeks 8–9)**:
   Rigorous testing was conducted to ensure that the gesture controls worked smoothly and that the video feed was consistent. Debugging of hardware and software components ensured reliable performance.

5. **Final Integration and Documentation (Weeks 10–11)**:
   Final adjustments were made, and documentation was prepared for the project report. The car was fully integrated, with the gesture control system, live streaming, and motor controls functioning together.

CHAPTER 12

References

Raspberry-GPIO-Python, "raspberry-gpio-python Wiki," SourceForge, [Online]. Available: https://sourceforge.net/p/raspberry-gpio-python/wiki/Home/. [Accessed: Nov. 20, 2024].

AdaFruit, "AdaFruit MPU6050," [Online]. Available: https://adafruit.github.io/Adafruit_MPU6050/html/class_adafruit___m_p_u6050.html. [Accessed: Nov. 20, 2024].

Predictable Designs, "How to Connect an ESP32 WiFi Microcontroller to a Raspberry Pi Using IoT MQTT," [Online]. Available: https://predictabledesigns.com/how-to-connect-esp32-microcontroller-to-raspberry-pi-using-iot-mqtt/. [Accessed: Nov. 20, 2024].

MQTT, "MQTT," [Online]. Available: https://mqtt.org/. [Accessed: Nov. 20, 2024].

PubSubClient, "Pub Sub Client," [Online]. Available: https://pubsubclient.knolleary.net/. [Accessed: Nov. 20, 2024].

UpsideDownLabs, "BioAmp EXG Pill GitHub," [Online]. Available: https://github.com/upsidedownlabs/BioAmp-EXG-Pill. [Accessed: Nov. 20, 2024].
\

Neuphony, "Neuphony Synapse Manual," [Online]. Available: https://neuphony.gitbook.io/synapse. [Accessed: Nov. 20, 2024].

Appendix A

Datasheets

Please refer to the following folder containing all the datasheets of the sensors, actuators and displays.

Appendix B

Programming Review:

This section provides a comparison between C, Python, and two other popular programming languages: Java and JavaScript. The comparison is based on factors such as syntax, performance, usability, and suitability for IoT applications.

C:

- A procedural programming language known for its speed and low-level memory access.
- Widely used in embedded systems and hardware-level programming.

Python:

- An interpreted, high-level language known for its simplicity and readability.
- Popular in IoT for its extensive libraries and community support.

Java:

- A general-purpose, object-oriented language designed for cross-platform compatibility.
- Commonly used for Android app development and backend IoT applications.

JavaScript:

- A lightweight, interpreted scripting language primarily used in web development.
- Increasingly popular in IoT for real-time web-based interfaces.

Comparison of Programming Languages

Feature	C	Python	Java	JavaScript
Syntax	Complex and rigid	Simple and highly readable	Moderately complex	Simple but verbose
Execution	Compiled	Interpreted	Compiled to bytecode	Interpreted
Performance	Extremely fast, low-level	Slower due to interpretation	Fast, optimized JVM	Slower, browser-dependent
IoT Suitability	Ideal for hardware control	Ideal for scripting and data processing	Suitable for middleware	Best for real-time web interfaces
Portability	Hardware-dependent	Cross-platform	Highly portable (JVM-based)	Highly portable (browser-based)

Strengths and Weaknesses

1. **C:**
    - Strengths: Extremely fast, efficient for microcontrollers, precise hardware control.
    - Weaknesses: Complex syntax, manual memory management, steep learning curve.
2. **Python:**
    - Strengths: Easy to learn, extensive libraries for IoT, rapid prototyping.
    - Weaknesses: Slower execution, higher resource consumption, less suited for hardware.
3. **Java:**
    - Strengths: Platform-independent, secure, suitable for backend and middleware.
    - Weaknesses: Memory-intensive, slower than C for real-time tasks.

4. **JavaScript:**
    - Strengths: Real-time interactivity, ideal for IoT web interfaces.
    - Weaknesses: Limited for hardware-level programming, browser-dependent performance.

Conclusion

- **C excels in low-level hardware programming and resource-constrained devices.**
- **Python is versatile for scripting and data analysis tasks.**
- **Java is reliable for middleware and cross-platform applications.**
- **JavaScript is best suited for IoT dashboards and real-time web interfaces.**

Appendix C

Trouble-Shooting:

Debugging Issue: Motor Driver Not Responding

Problem Encountered:
During the initial testing of the car's motion control, the DC motors failed to activate even though the gesture inputs from the accelerometer (MPU6050) were being successfully processed by the Raspberry Pi. The motor driver (L298N) showed no indication of receiving signals, resulting in the car being completely stationary.

Root Cause Analysis:

1. **Faulty Wiring:**
    - Upon checking the connections, one of the GPIO pins on the Raspberry Pi meant to control the motor driver was found to be loose. This caused intermittent or no signal to reach the input pin of the L298N motor driver.
    -
2. **Incorrect Logic Level Compatibility:**
    - The Raspberry Pi uses 3.3V logic for its GPIO pins, while the L298N motor driver typically expects a 5V signal for reliable operation. The mismatch led to the motor driver not recognizing the control signals.

3. **Power Supply Issues:**
    - The motor driver was initially powered by the same source as the Raspberry Pi, causing insufficient current supply when the motors were activated. The motors require higher current for operation, which was not being delivered consistently.

Steps Taken to Resolve the Issue:

1. **Securing the Wiring:**

- The GPIO connections from the Raspberry Pi to the L298N motor driver were rechecked and properly secured. Dupont connectors were replaced with soldered connections to ensure stability during operation.

2. **Level Shifter Integration:**
   - A logic level shifter was added between the Raspberry Pi and the motor driver to convert the 3.3V signals to 5V. This ensured that the L298N could accurately detect the control signals sent by the Raspberry Pi.

3. **Dedicated Power Supply for Motors:**
   - A separate power supply (12V battery) was used to power the L298N motor driver and the DC motors. This ensured that the motors received sufficient current without overloading the Raspberry Pi's power source.

4. **Testing Signal Transmission:**
   - Using a multimeter, the voltage levels on the motor driver's input pins were tested to confirm the proper transmission of signals from the Raspberry Pi. The outputs to the motors were also verified.

5. **Code Debugging:**
   - Minor issues in the motor control code were resolved, such as ensuring the correct GPIO pins were assigned for motor control and adjusting the pulse width modulation (PWM) signals for speed regulation.

Outcome:

After these steps, the motor driver began responding accurately to the control signals, and the DC motors functioned as intended. The car moved forward, backward, and turned in response to gestures, completing the desired functionality. The troubleshooting process highlighted the importance of proper wiring, logic level compatibility, and sufficient power supply in IoT hardware setups

Appendix D

Real-Life Mounting of the System:

Deployment Scenario

In a real-life application, the gesture-controlled car system could be mounted and deployed in several environments, such as:

- **Smart Warehouses:** For moving goods efficiently without manual intervention.
- **Assistance Robots:** Providing mobility assistance to individuals with disabilities.
- **Delivery Robots:** Automating the delivery of items within offices, hospitals, or residential areas.

Placement and Protection Strategies

1. **Weather Protection:**
    - Enclose the system in a weatherproof casing made of durable materials like polycarbonate or aluminum.
    - Use rubber seals around openings to prevent water and dust ingress.
    - Ensure vents are shielded to allow airflow without exposing the internal components to rain or debris.

2. **Environmental Safety:**
    - Mount shock-absorbing materials inside the casing to protect delicate components, like the Raspberry Pi and sensors, from vibrations or impacts.
    - Include a cooling mechanism (e.g., heat sinks or small fans) to prevent overheating in high-temperature environments.

3. **Protection from Damage or Vandalism:**
    - Install the system in tamper-proof enclosures with lockable access points to prevent intentional damage.
    - Paint the chassis with scratch-resistant coatings and reinforce with impact-resistant frames.
    - Use hidden wiring or protective conduits to minimize exposure to wires.

Maintenance of Power Supply

1. **Power Sources:**
    - Use rechargeable lithium-ion battery packs capable of delivering sufficient voltage and current for extended operation.
    - For stationary or semi-stationary applications, include an option for AC power supply with a backup battery for uninterrupted power.

2. **Energy Efficiency:**
    - Optimize power consumption by using energy-efficient components and algorithms. For instance, switch to low-power modes when the system is idle.

3. **Solar Power Integration:**
    - Attach solar panels to the system to charge the batteries in outdoor deployments, ensuring longer operational periods without human intervention.

Ensuring Internet Connectivity

1. **Wi-Fi Network:**
    - In areas with established infrastructure, connect the system to a stable Wi-Fi network for remote monitoring and control.
    - Use signal boosters or mesh networks to extend coverage in large spaces.

2. **Cellular Network (4G/5G):**
    - For deployment in areas without Wi-Fi, equip the system with a 4G/5G dongle or SIM module for internet access.
    - Implement redundancy by switching between cellular and Wi-Fi networks based on signal availability.

3. **Offline Operation:**
    - Design the system to function independently of internet connectivity by pre-programming tasks or storing data locally, with periodic syncing when the connection is available.

Long-Term Maintenance

1. **Regular Inspections:**
    - Schedule routine checks for hardware wear and tear, such as battery health, motor functionality, and sensor calibration.

2. **Software Updates:**
    - Enable over-the-air (OTA) updates to improve functionality, fix bugs, and enhance security remotely.

3. **Security Measures:**
    - Implement encryption for all communication to prevent unauthorized access or hacking attempts.
    - Install GPS tracking to locate the system in case of theft or loss.

Appendix E

Real-Life Scenarios

Implementing the gesture-controlled car system in real-life scenarios can pose several challenges. Below are three key challenges and the corresponding solutions:

1. Signal Interference in Gesture Recognition

- **Challenge:**
  In crowded environments or areas with high electromagnetic interference (e.g., warehouses or urban settings), the accelerometer and gyroscope signals might be affected, leading to inaccurate gesture recognition or delayed responses.
- **Resolution:**
  - Implement signal filtering techniques like Kalman filtering to eliminate noise from sensor data.
  - Test and calibrate the sensors in the intended environment to fine-tune sensitivity settings.
  - Use error correction algorithms to improve the robustness of gesture detection.

2. Battery Drain and Limited Operational Time

- **Challenge:**
  The system relies on battery power for mobility and processing, which can limit the operational period, especially in outdoor applications or prolonged usage scenarios.
- **Resolution:**
  - Use high-capacity lithium-ion batteries with advanced power management circuits to optimize energy usage.
  - Integrate solar panels for continuous recharging in outdoor environments.
  - Enable low-power modes when the system is idle to conserve battery life.

3. Durability and Safety in Harsh Conditions

- **Challenge:**
  Real-life deployment in outdoor or industrial environments exposes the system to

weather conditions, dust, vibrations, and accidental collisions, which can damage components or impair functionality.

- **Resolution:**
    - Encase the system in a robust, weatherproof housing made from impact-resistant materials.
    - Install shock-absorbing mounts to protect sensitive electronics from vibrations and impacts.
    - Regularly inspect and replace wear-prone components, such as motors and wiring.
    - Introduce obstacle detection mechanisms to avoid collisions during operation.