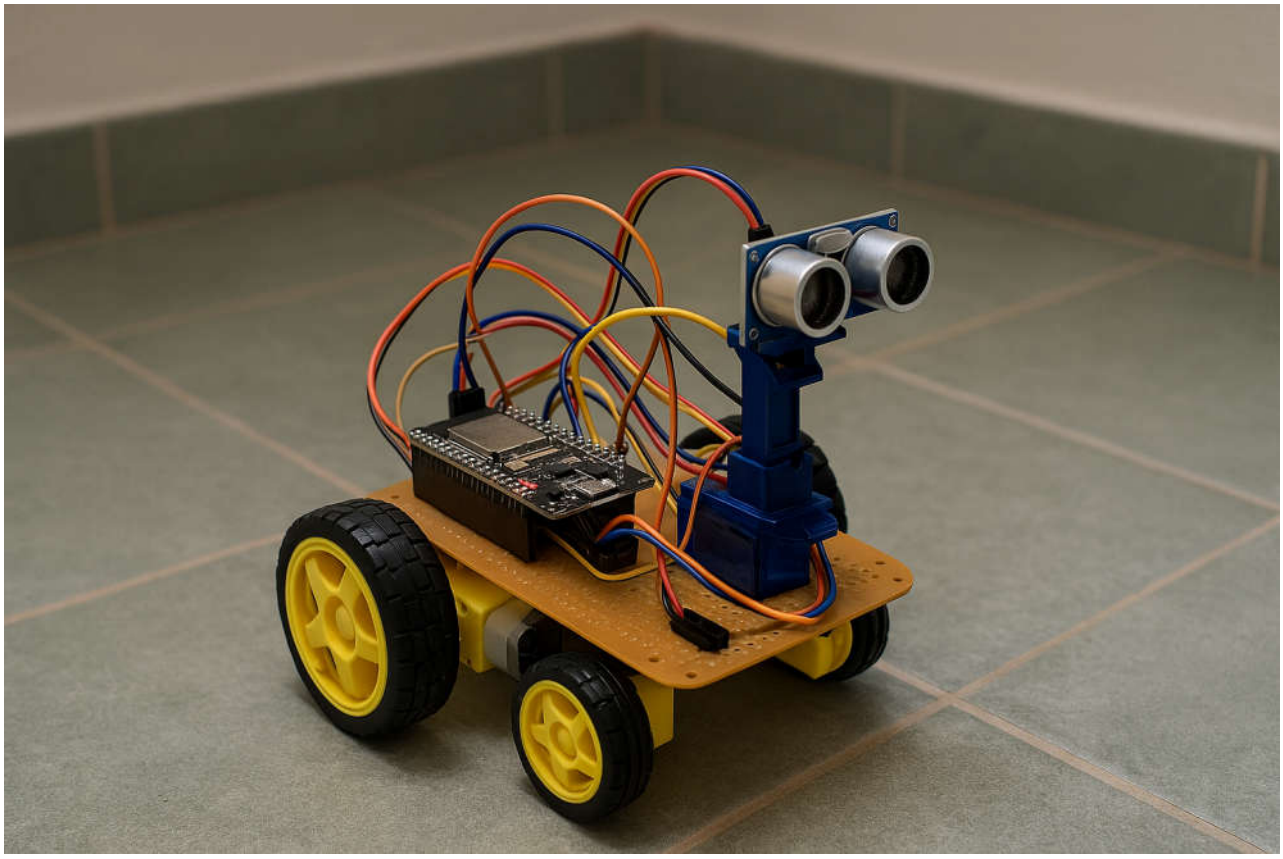
Visvesvaraya National Institute of Technology (VNIT), Nagpur

Electronic Product Engineering Workshop

DOCUMENTATION

Kartik Mundkar	BT23ECE034
Vaibhav S. Gurav	BT23ECE035
Kapil Mahale	BT23ECE037
Prajwal Bhagat	BT23ECE045

2D Mapping and Visualization using ESP 32



AIM :

Real-Time SLAM System Using ESP32 and Ultrasonic Sensors for Environment Mapping.

ABSTRACT :

This project presents a Simultaneous Localization and Mapping (SLAM) system implemented using an ESP32-based Slam-Car equipped with an MPU-9250 IMU sensor and an ultrasonic sensor. The system aims to provide accurate 2D mapping of the environment by combining data acquisition and visualization. The proposed solution focuses on low-cost hardware integration and real-time data visualization to aid in efficient environment mapping.

INTRODUCTION :

SLAM is a critical technology in robotics, enabling autonomous systems to navigate unknown environments while simultaneously building a map. The integration of cost-effective hardware like the ESP32, IMU, and ultrasonic sensors offers an accessible yet powerful solution for real-time mapping. This project focuses on developing a Slam-Car that collects position data using sensor inputs, applies mapping algorithms, and visualizes the mapped area in a 2D representation.

METHODOLOGY:

The project is structured into three key phases to ensure an efficient and systematic approach:

1. SLAM-Car Design

The development of the ESP32-based autonomous vehicle involves:

- **Microcontroller Selection:** The ESP32-S is chosen due to its built-in WiFi and Bluetooth capabilities, which allow for real-time data transmission.
- **Motor and Motion System:** The vehicle is powered by **DC motors** connected to an **L293D motor driver module**, enabling bidirectional movement and precise speed control.
- **IMU Sensor (MPU-9250):** The **Inertial Measurement Unit (IMU)** captures acceleration and angular velocity data, helping track the car's orientation and movement.

- **Ultrasonic Sensor (HC-SR04):** Measures distances by sending and receiving ultrasonic pulses, allowing the system to detect obstacles and generate a spatial map of the environment.
- **Custom Car Frame:** A lightweight, rigid chassis is designed to house all components securely and minimize vibrations, ensuring accurate sensor readings.

2. Data Acquisition

The ESP32 continuously collects data from the sensors, processes it in real-time, and transmits it for visualization. This step involves:

- **IMU Data Processing:** The accelerometer and gyroscope readings from the **MPU-9250** are processed to estimate the vehicle's movement and orientation.
- **Ultrasonic Sensor Readings:** The **HC-SR04** measures the distance to nearby objects, helping to build an environmental map.
- **Noise Reduction Techniques:** Raw sensor data contains noise and inaccuracies. To improve positional accuracy, the following filtering methods are implemented:
    - **Complementary Filtering:** Combines accelerometer and gyroscope data to obtain a more stable orientation estimate.
    - **Kalman Filtering:** An advanced algorithm that refines sensor readings by predicting and correcting errors based on previous measurements.

3. Data Representation

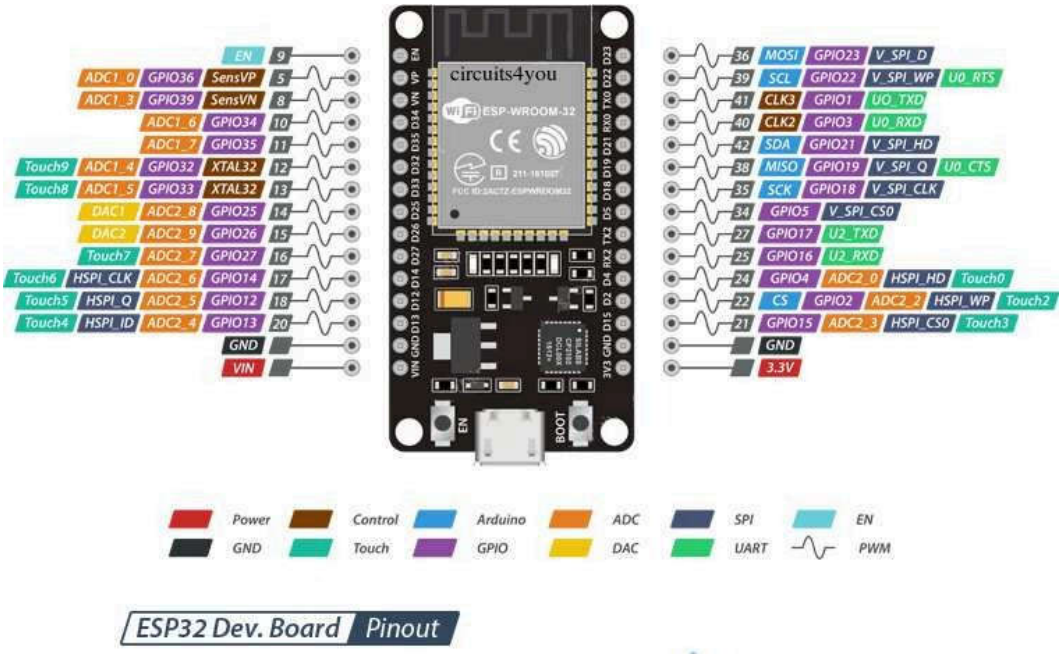
After data is acquired and processed, it is visualized to create a **real-time 2D and 3D representation** of the environment. This phase includes:

- **Python-based Data Processing:** The ESP32 transmits sensor data to a computer running a Python script that processes the information.
- **Graphical Visualization:**
    - **Matplotlib** is used for static and animated 2D plotting.
    - **Plotly** provides interactive 3D visualizations of the mapped environment.
- **Real-time Updates:** The visualization platform continuously refreshes the map, enabling real-time tracking of the vehicle's position and detected obstacles.

❖ Hardware and Software Design :

★ Hardware Components :

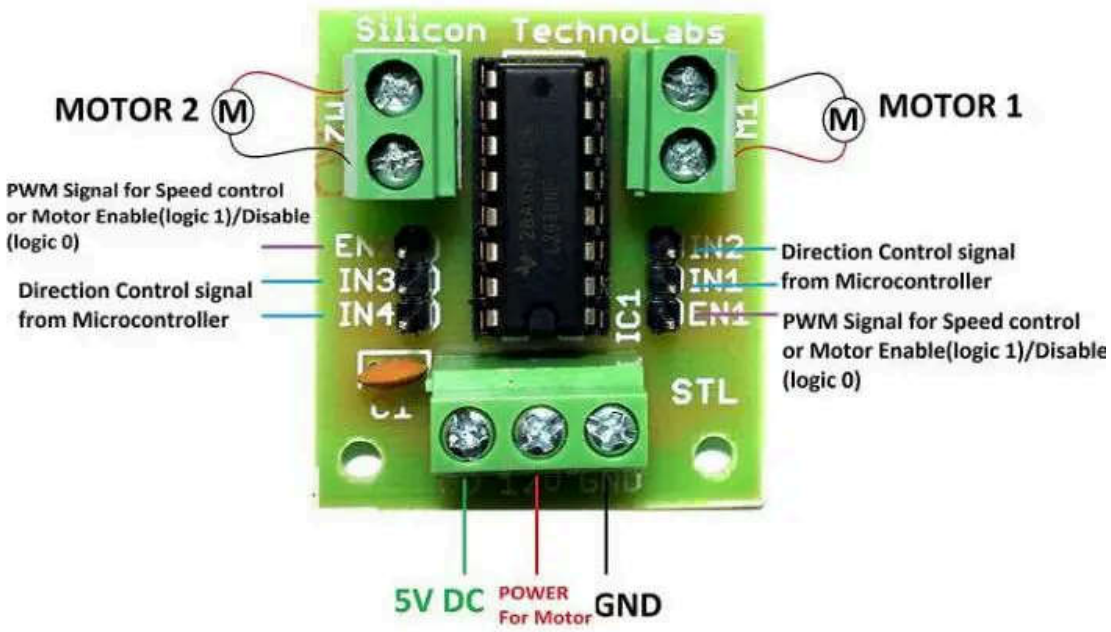
1. ESP32 (Microcontroller)



The **ESP32** is a powerful and versatile microcontroller that serves as the brain of this system. It is responsible for receiving, processing, and transmitting data between various sensors, the motor driver, and the motion controller.

- **Wi-Fi & Bluetooth Connectivity**: Enables wireless control via a mobile phone and data transmission to a server or laptop.
- **Multiple GPIO Pins**: Facilitates interfacing with multiple sensors, actuators, and modules.
- **High Processing Power**: Handles real-time processing tasks like sensor data acquisition, motor control, and communication.
- **Analog and Digital I/O Support**: Can interface with both digital sensors like the ultrasonic sensor and analog sensors if needed.

2. L293D Motor Driver

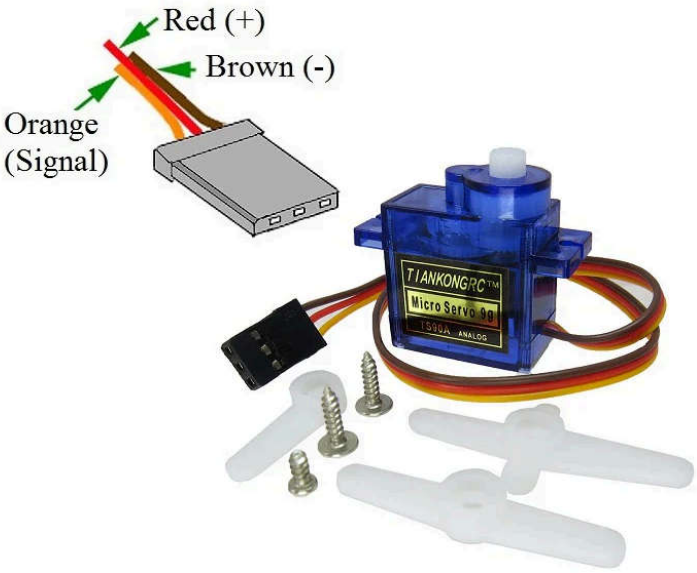


The **L293D** is a dual H-Bridge motor driver IC that controls the DC motors connected to the car. Since the ESP32 cannot directly drive motors due to limited current output, the **L293D** acts as an interface to amplify the control signals.

- **Controls Two Motors Independently**: Allows movement in all directions (forward, backward, left, and right).
- **Handles High Current & Voltage**: Operates at up to **36V and 600mA per channel**.
- **PWM Control**: Allows speed variation using **Pulse Width Modulation (PWM)** signals.

- **Bidirectional Motor Control**: Supports both forward and reverse movement.

3. Servo Motor (SG-90)



The **SG-90 servo motor** is used to rotate the ultrasonic sensor, allowing it to scan the environment for mapping.

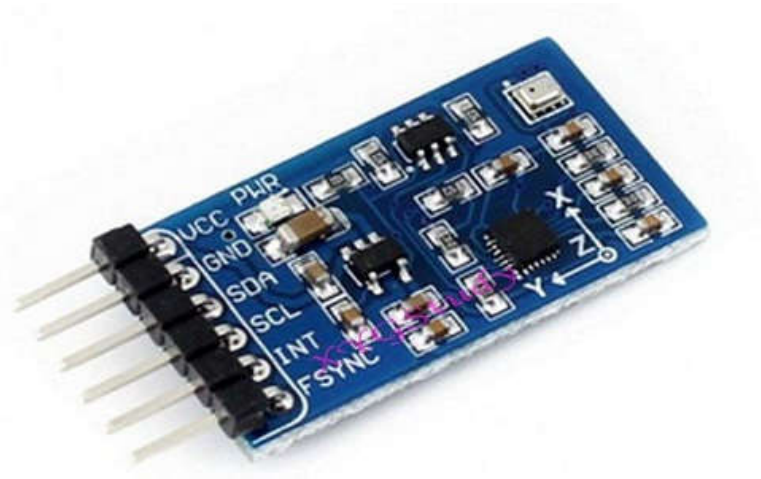
- **Precise Angular Control:** It moves in defined steps between 0° and 180°, ensuring accurate directional sensing.
- **Low Power Consumption**: Suitable for embedded applications powered by batteries.
- **Compact and Efficient:** Offers high torque in a small form factor with low power consumption—ideal for battery-powered embedded systems.
- **Enhanced Data Collection:**The controlled movement helps gather distance data from multiple angles, enhancing the accuracy of environmental visualization.

4. Ultrasonic Sensor (HC-SR04)

The **HC-SR04 ultrasonic sensor** is used for distance measurement and obstacle detection. It works by sending ultrasonic pulses and measuring the time taken for the echo to return.

- **Obstacle Detection:** Efficiently identifies objects within a range of 2cm to 400cm, making it suitable for navigation and mapping.
- **High Accuracy:** Offers reliable readings with a typical error margin of ±3mm, ensuring precise measurements..
- **Simple Interface:** Operates using four pins—VCC, GND, Trigger, and Echo—making it easy to integrate with microcontrollers.
- **Supports 2D Mapping:** Enables spatial visualization by collecting distance data at various angles when paired with a rotating servo.
- **Real-Time Sensing:** Continuously monitors surroundings to detect obstacles and measure distances on the go.

5. IMU Sensor (MPU-9250)



The **MPU-9250** is an **Inertial Measurement Unit (IMU)** sensor that tracks the orientation and movement of the car. It consists of:

- **Accelerometer**: Measures acceleration in **X, Y, and Z** axes.
- **Gyroscope**: Tracks angular velocity and rotational motion.
- **Magnetometer**: Detects changes in the Earth's magnetic field for direction sensing.
- **Real-time Motion Analysis**: Helps with precise movement and stabilization.

6. Motion Controller

The car is controlled remotely using a **laptop** via **Wi-Fi or Bluetooth**. The ESP32 receives commands to navigate in different directions.

- **Wireless Communication**: Uses **Wi-Fi/Bluetooth** for long-range control.
- **User Interface**: A mobile app with buttons/sliders for movement control.
- **Real-Time Command Execution**: Instant response to user inputs

.

★ Software Framework :

The implementation of this project requires both **software tools** for development and **libraries** to interface with hardware components efficiently. This section describes the key software environments, programming frameworks, and libraries used.

1. Development Environment

1.1 PlatformIO (VS Code)

Why PlatformIO?

PlatformIO is an advanced open-source ecosystem for IoT development. It is an excellent alternative to the traditional Arduino IDE, offering a more professional and scalable development environment. It is built on top of **Visual Studio Code**, which brings powerful features such as intelligent code completion, Git integration, and extensions support.

Key Features:

- **Integrated Library Manager –** Simplifies adding, updating, and removing libraries with version control to avoid conflicts.
- **Modular Code Structure –** Encourages the use of multiple source and header files, making large projects easier to manage and debug.
- **Cross-platform Build System –** Supports over 1000 embedded boards and multiple frameworks including Arduino, ESP-IDF, STM32Cube, and Zephyr.
- **Built-in Debugging –** Offers advanced debugging tools that help in setting breakpoints, monitoring variables, and stepping through code.
- **Multi-environment Support –** Allows managing different configurations (e.g., for different boards or setups) within a single project.

2. Programming Libraries

2.1 IMU Sensor (MPU9250) – Motion Tracking

Library Used :- MPU9250.h

Purpose:

Reads **accelerometer, gyroscope, and magnetometer** data.
Helps determine the **cart's orientation and movement**.
Uses **algorithms (e.g., complementary filter, Kalman filter)** to improve accuracy.

2.2 Servo Motor Control – Rotating the Ultrasonic Sensor

Library Used:- ESP32Servo.h *(Arduino built-in library)*

Purpose:

Controls the **servo motor** to rotate the ultrasonic sensor 180° for scanning.
Ensures **precise movement** for accurate mapping.

WORKING :

An IMU (Inertial Measurement Unit) gives us:

- **Acceleration (aX, aY, aZ)** → From the **accelerometer**
- **Rotation/Angular Velocity (gX, gY, gZ)** → From the **gyroscope**
- **Magnetic field (mX, mY, mZ)** → From the **magnetometer**

IMU Data Challenges and Filtering Techniques -

When working with raw data from an Inertial Measurement Unit (IMU) like the MPU9250, several common challenges can affect the accuracy and stability of measurements. One major issue is **sensor noise —** small, random fluctuations in the data caused by external vibrations, electronic interference, or inherent sensor limitations. Another issue is drift, particularly noticeable in gyroscopes, where the sensor's output slowly changes over time even when there's no movement, causing a gradual accumulation of error in orientation estimates. **Bias or offset** is also a concern, where the sensor shows non-zero values when stationary; for instance, an accelerometer may read 0.1g on the X-axis despite being perfectly still. Lastly, spikes or outliers—sudden, extreme readings—can appear due to jerks, quick impacts, or glitches in sensor hardware.

To address these problems and make the IMU data suitable for accurate surface mapping, we apply a series of calibration and filtering steps. The process starts with **sensor calibration**, where the IMU is kept still, and readings are averaged to determine baseline offsets. These biases are subtracted from future data, and gravitational acceleration (typically ~9.81 m/s^2 on the Z-axis) is removed to isolate only the motion-related components. This step ensures that the system starts with a clean, corrected reference frame.

Following calibration, a **Low Pass Filter (LPF)** is applied to smooth out high-frequency noise. It suppresses rapid changes in data caused by small vibrations or electrical interference, keeping the slow, more meaningful variations in motion. On the other hand, a **High Pass Filter (HPF)** is used to counter slow-changing errors like gyroscope drift. It removes the low-frequency components from the data, helping maintain accuracy over longer periods by focusing only on fast, actual movements.

To combat occasional spikes or abnormal values, a **Median Filter** is introduced. This filter collects a short window of recent readings (e.g., the last five), sorts them, and selects the middle value. Unlike averaging, this method is highly effective at ignoring extreme outliers without distorting the signal.

Finally, we use a **Kalman Filter** for sensor fusion — combining data from the accelerometer and gyroscope. The accelerometer provides reliable long-term orientation based on gravitational pull but reacts slowly and can be noisy. The gyroscope is fast and smooth but prone to drift. By intelligently blending the two, the Kalman Filter generates accurate, real-time estimates of the sensor's orientation and movement, essential for reliable SLAM operations and precise 2D surface mapping

Understanding Sensor Fusion:

- **Gyroscope:**
  - Measures how fast the device is rotating (angular velocity).
  - Very responsive and smooth over short periods
  - But it accumulates small errors over time , this is called drift.
  - So, in the short term, we trust the gyroscope because it gives fast, real-time motion changes.

- **Accelerometer:**
  - Measures linear acceleration, including the force of gravity.
  - From this, we can estimate the tilt/orientation (like roll and pitch).
  - However, it is noisy and sensitive to small vibrations.
  - But in the long term, it gives a stable reference to correct the drift of the gyroscope.

- Kalman filter is like a smart system that "learns" from both sensors and predicts the most likely true value.
  This results in accurate, stable orientation and movement even in long-term operation.

Orientation & Position Tracking (Using IMU)-

Accurate orientation and position tracking using an IMU relies heavily on a few key constants that help refine raw sensor data. One of the most fundamental is **gravity** ($g = 9.81\ m/s^2$). When the IMU is stationary, the accelerometer naturally senses gravity—usually on the Z-axis. To isolate true motion, we subtract this gravitational pull during calibration, ensuring that future readings reflect actual acceleration. Another crucial constant is dt (delta time), representing the time between two IMU readings. For example, with dt = 0.05 seconds, the system samples at 20 Hz. This time step is essential for numerical integration: converting acceleration to velocity and then velocity to position. It also allows for real-time orientation updates using gyroscope data via simple equations like

`velocity += acceleration * dt` and `yaw += angular_velocity * dt`.

To improve data stability, we apply smoothing techniques using constants like **ALPHA_GYRO**, a factor between 0 and 1 that blends the fast, real-time updates from the gyroscope with the more stable (but slower) corrections from the accelerometer. A higher value means the system reacts quickly but may drift, while a lower value produces smoother, more accurate long-term tracking. Similarly, **ACCEL_ALPHA** is used in a low-pass filter to clean up accelerometer readings. A high alpha value allows rapid updates (but lets noise through), while a low alpha yields smoother results with reduced high-frequency noise. These parameters ensure that motion tracking feels natural and realistic, even in noisy environments.

To prevent accumulated errors from ruining position estimates, we introduce two smart corrections. **VEL_DECAY**, typically set around 0.95, reduces velocity gradually at each step, simulating natural friction or drag. This helps stop the velocity from growing endlessly when there's no real motion. **ZUPT** (Zero Velocity Update Threshold) is another clever addition—it detects near-zero acceleration (e.g., $< 0.05\ m/s^2$) and forces velocity to zero, effectively freezing position updates when the cart isn't moving. This is especially useful when the robot stops frequently, ensuring that small sensor noise doesn't get mistakenly integrated.

Gyroscope – Tracking Orientation (Yaw)-

The gyroscope is a key component in tracking a robot's orientation particularly its yaw, which represents rotation around the Z-axis — essentially, the direction the robot is facing. The gyroscope measures angular velocity, indicating how fast the device is turning. By integrating this angular velocity over time using the formula `yaw += gyro_z * dt`, we can estimate how much the robot has rotated. This makes the gyroscope extremely useful for capturing quick, real-time changes in orientation during motion.

However, gyroscopes aren't perfect. Even small inaccuracies like sensor bias or noise accumulate over time, leading to what's known as drift — where the yaw angle keeps changing slowly even when the robot isn't actually turning. To tackle this, we use exponential smoothing to blend the gyroscope's fast but potentially drifting output with a more stable correction. The updated yaw is calculated using the formula:

`yaw = ALPHA_GYRO * (prev_yaw + gyro_z * dt) + (1 - ALPHA_GYRO) * correction`,

where `prev_yaw + gyro_z * dt` is the raw gyroscope estimate, and `correction` can come from a more stable reference like an accelerometer, magnetometer, or compass. The constant **ALPHA_GYRO**, which ranges between 0 and 1, controls how much we trust the gyroscope versus the correction source.

Tuning **ALPHA_GYRO** is critical. A high value (e.g., 0.98) favors the gyroscope, giving fast, responsive tracking but potentially allowing more drift. A lower value (e.g., 0.80) leans more on the correction, reducing drift but possibly making the response feel slower. When tuned well, this method provides a smooth and accurate yaw estimation that remains stable over long durations, essential for reliable navigation and surface mapping.

Accelerometer (Detecting Movement & Tracking Position) -

The accelerometer is responsible for measuring linear acceleration along the X and Y axes, which tells us how the robot is moving in a straight line, as opposed to rotating (which is handled by the gyroscope). From these readings, we can estimate both velocity (how fast the robot is moving) and position (how far it has traveled). However, a major challenge in using accelerometer data is that it's inherently noisy, primarily due to mechanical vibrations and sensor imperfections. When we integrate acceleration to get velocity, even tiny errors can accumulate quickly, and repeated integration to derive position leads to even greater drift over time.

To mitigate these issues, we apply a series of filtering and transformation techniques. First, a **Low-Pass Filter (LPF)** is used to smooth out high-frequency noise from raw acceleration readings. This filter blends current and previous values using a constant α (typically between 0.1 and 0.3) to produce more stable and realistic acceleration data for velocity estimation. Next, we convert the filtered acceleration from the **local frame** (robot-based) to the **global frame** (world-based). This transformation is necessary because the accelerometer reports motion relative to the robot's current orientation, but for mapping and position tracking, we need motion referenced to the environment. Using the current yaw angle from the gyroscope, we rotate the acceleration using the formulas:

```
ax_global = ax * cos(yaw) - ay * sin(yaw)
ay_global = ax * sin(yaw) + ay * cos(yaw)
```

This ensures that the acceleration is aligned with real-world X and Y directions, even when the robot turns.

Once the data is cleaned and aligned, we perform integration to compute velocity:
`vx += ax_global * dt`, `vy += ay_global * dt`.
From velocity, we then integrate again to estimate position using:
`X += vx * dt`, `Y += vy * dt`.

To reduce long-term drift, we also apply a method called **ZUPT** (Zero Velocity Update). When the total acceleration falls below a certain threshold (e.g., when the robot is stationary), we assume the robot has stopped and manually reset the velocity to zero:

`if abs(ax) < ZUPT and abs(ay) < ZUPT: vx, vy = 0, 0`.

This prevents tiny fluctuations from being interpreted as movement and helps maintain position stability when the robot is idle. Frame conversion and proper filtering are therefore critical to achieve reliable movement tracking and mapping in real-world environments.

Ultrasonic Sensor + Servo (Surface Mapping)-

The **ultrasonic sensor** plays a key role in obstacle detection by emitting sound waves and measuring the time it takes for the echo to return. Using the **time-of-flight** principle, it calculates the **distance to nearby objects**, much like a mini radar. This sensor is mounted on a **servo motor**, which systematically rotates in fixed angular steps, allowing it to scan the surroundings in a 180° or 360° sweep. For each servo angle, the ultrasonic sensor provides the distance to the nearest obstacle in that direction, giving us a local view of the environment.

To convert this local scan into a **global surface map**, we combine data from the servo, ultrasonic sensor, and IMU. The **servo angle** indicates where the sensor is pointing relative to the robot, while the **yaw** from the gyroscope gives the robot's current orientation in the world. Using this, we compute the true (X, Y) position of each obstacle with the formulas:

```
x_scan = X + distance * cos(angle + yaw)
y_scan = Y + distance * sin(angle + yaw).
```

Here, `distance` is from the ultrasonic sensor, `angle` is the servo's current rotation, `yaw` is the robot's orientation, and `(X, Y)` is the robot's current global position.

This approach allows us to place every scanned point correctly on a **global map**, regardless of the robot's movement or rotation. A dynamic **plot update routine** ensures that the surface map is continuously refreshed, zooming in or out as needed to fit the robot's exploration area. As the robot moves and scans, both its **live trajectory** and the **surrounding obstacles** are drawn on the canvas, creating a real-time visualization of the environment.

Data Communication & UI Control-

WebSockets – Real-Time Data Transfer

WebSockets create a two way connection between the car (ESP32) and the laptop's browser UI. Once connected, both sides can send and receive data at any time, instantly and without delay. This full-duplex communication allows the robot and the UI to exchange messages simultaneously, ensuring there is no lag in control or data display which makes it perfect for live mapping applications. Unlike HTTP, which requires repeated requests every few seconds, WebSockets keep the link open, allowing updates to be pushed automatically as soon as they are available.

Libraries Used:

- `ESPAsyncWebServer` (Arduino), or native WebSocket libs.

WebSocket Client

A JavaScript frontend running in a browser connects to the car's IP address using WebSocket. When the webpage loads, it establishes the WebSocket connection, enabling seamless communication. The browser listens for incoming messages, such as sensor data from the car, while also sending out messages, such as key press commands, to control the car.

JSON-Based Messaging

All messages sent over WebSocket are in JSON format, making them simple and human readable. This format allows the browser to easily update the plot with new data and enables the robot to react to key presses.

Live Interaction Flow (Real-Time Example):

While the car is moving and scanning, we can press the W key to send a command from the UI. The UI sends a JSON message like { "command": "forward" } to the car, which then receives the command and moves forward.

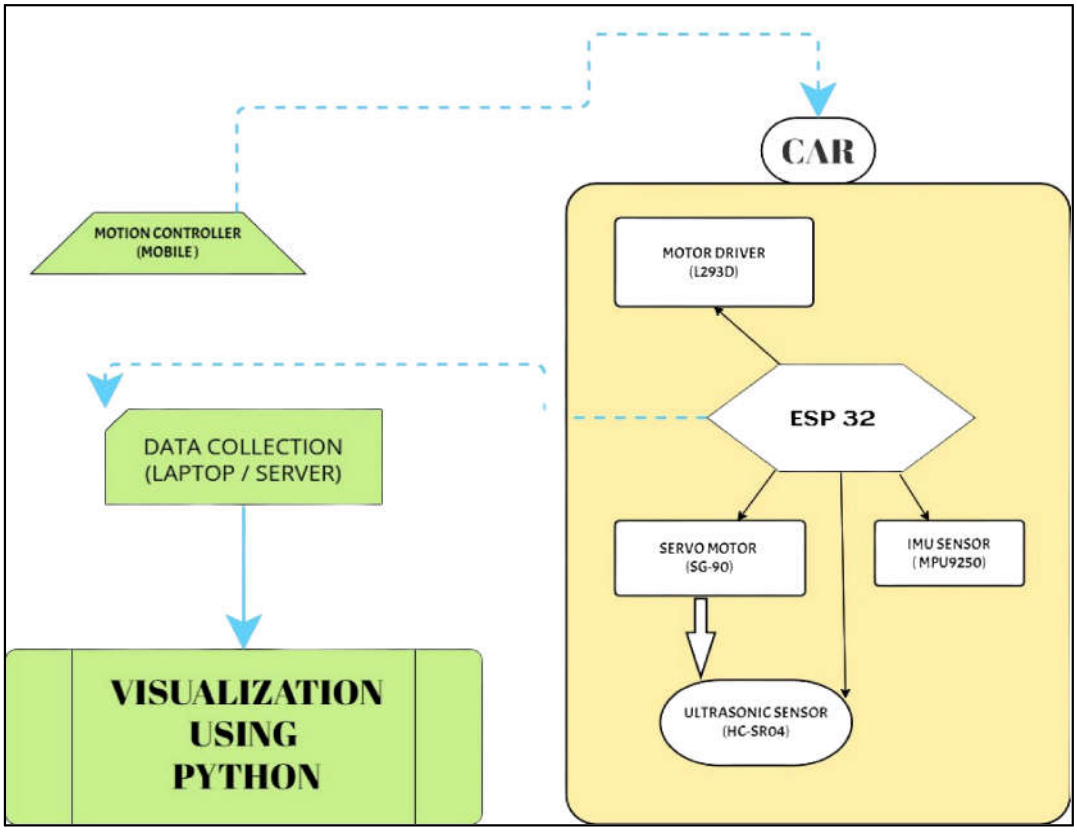
As the robot moves, it collects IMU and distance data and sends this information back to the UI in JSON format

example: { "yaw": 0.9, "x": 2.1, "y": 3.0, "scan": { "angle": 60, "distance": 95 } }.

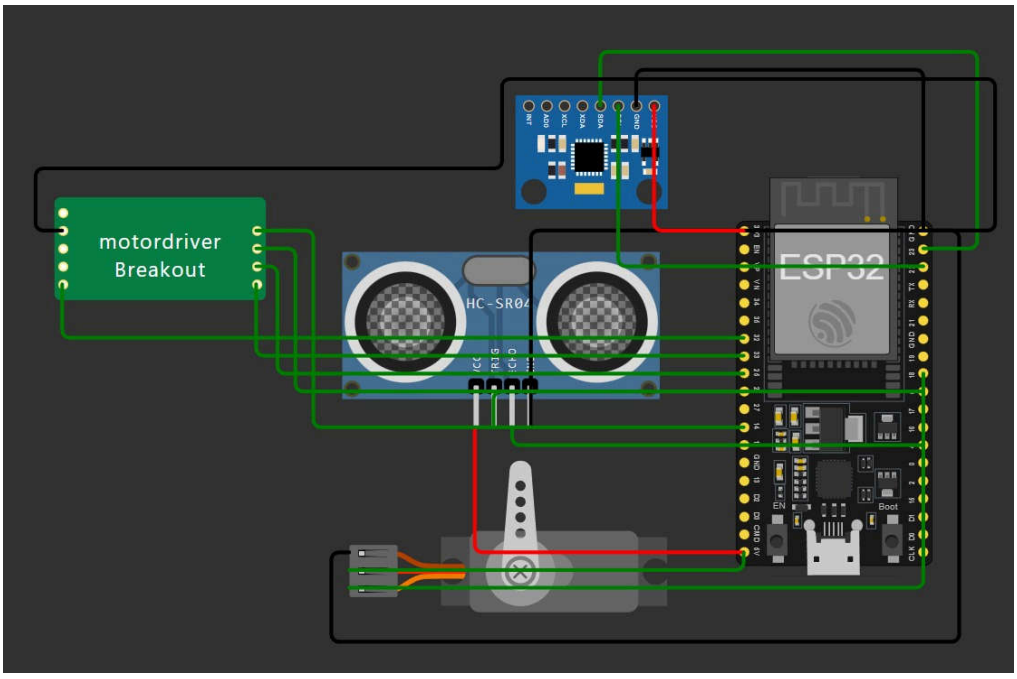
The UI receives this data and instantly updates the position and map on the plot. If you press the A key to turn left, the UI sends another command { "command": "left" }, prompting the car to turn left while continuing to scan. This cycle of sending commands and receiving data continues seamlessly as long as the WebSocket connection remains active.

WebSockets are ideal for this application because they address several key needs. For instant robot control, commands are sent immediately, ensuring responsive operation. Real-time mapping is achieved as sensor data streams to the UI without delay, allowing for up-to-date visualization. The bidirectional nature of WebSockets means that both the robot and the UI can continuously update each other, maintaining perfect synchronization. Additionally, everything is managed through a single interface, so all control and visualization happen conveniently.

FLOWCHART :



CIRCUIT :



RESULTS :

The system successfully tracks the Slam-Car's position and maps the environment in real-time. The ultrasonic sensor effectively measures distances, while the IMU sensor ensures motion tracking with minimal drift. The combination of both sensors enables precise data fusion for accurate mapping. The visual output accurately displays objects and obstacles within the environment, confirming the system's effectiveness in real-time mapping scenarios.

CONCLUSION :

This project demonstrates a practical, cost-effective SLAM solution using ESP32 hardware. Future improvements may involve integrating additional sensors for enhanced accuracy, improving data filtering techniques, and expanding visualization capabilities with web-based platforms for better accessibility.

REFERENCES :

1. Toward Accurate Position Estimation Using Learning to Prediction Algorithm in Indoor Navigation
2. ESP32 Pinout Reference - Last Minute Engineers
3. Kalman Filter for Beginners, Part 1 - Recursive Filters & MATLAB Examples

PROJECT BY -

KARTIK MUNDKAR	**BT23ECE034**
VAIBHAV S. GURAV	**BT23ECE035**
KAPIL MAHALE	**BT23ECE037**
PRAJWAL BHAGAT	**BT23ECE045**