An-Najah National University
Faculty of Engineering & Information Technology
Computer Engineering Department

Autonomous 3D LiDAR Mapping: Transforming 1D LiDAR to 3D LiDAR for Building Scanning with a Mobile Robot

Prepared By:
Rashid Sahem Nayef Hendawi
Mohammad Jafar Yousef AbuRehan

Supervised By:
Dr. Luai Malhis

A Graduation Project submitted to the Computer Engineering Department in partial fulfillment of the requirements for the degree of B.Sc. in Computer Engineering

Palestine
Jun, 2025

Abstract

In this project, we aim to convert 1D LiDAR into 3D LiDAR sensor using mirror mechanism to get a high light frequency and a wide angle for scanning the area. One of this project big approaches is to provide a cost-effective alternative to 3D LiDAR sensors, that are very expensive. The enhanced LiDAR system will be carried by a moving robot utilizing SLAM (Simultaneous Localization and Mapping) technology, included with a GPS sensor for positioning and obstacle avoidance. The primary abroach of this system will be a 3D point of cloud representation of the scanned environment that can be changed to a 3D Matrix or Virtual reality applications.

This project covers several aspects, starting from hardware design for the MEMS (mirror system), sensor integration, SLAM implementation for real-time data processing and the generation of the point of cloud. The development stage involves designing the mirror system (2D X-Y axis), implement and program the LiDAR data collection, processing the 3D transformation and testing the project in real world environments. To ensure reliability, smoothness and autonomously movement, GPS and obstacle detection sensors will be used.

3D LiDAR sensors already exist, but they are very expensive and not globally accessible. Our project aims a more cost-effective alternative, making 3D scanning technology widely available. Implementing the combination of LiDAR 1D sensor with SLAM navigation, GPS tracking, and obstacle detection is hard and limited. Our project offers the expansion of the low-cost 3D mapping technology.

Table of Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Problem Statement

Traditional 3D LiDAR sensors are prohibitively expensive and often inaccessible to researchers and small-scale developers. Although 1D LiDAR units offer precise distance measurements along a single beam, they lack the angular coverage necessary for comprehensive environment mapping. This project addresses the challenge of transforming a single-beam LiDAR into a cost-effective 3D scanning device using an orthogonal mirror system. By steering the beam across two axes, we generate a sparse 3D point cloud suitable for Simultaneous Localization and Mapping (SLAM) on a mobile platform. The resulting system must overcome limitations in sensor coverage, data latency, and mechanical precision to enable real-time mapping and navigation.

1.2 Objectives

The primary objectives of this project are:

1. Design and fabricate a two-mirror assembly that provides up to $\pm 10^\circ$ horizontal and $\pm 20^\circ$ vertical beam deflection for a 1D LiDAR sensor.
2. Integrate mirror-angle feedback using MPU6050 inertial measurement units to accurately reconstruct the 3D orientation of each LiDAR measurement.
3. Implement an ROS,2-based data acquisition and communication pipeline using ESP32 Wi-Fi UDP streaming of LiDAR, IMU, and wheel encoder data.
4. Fuse odometry and IMU data with `robot_localization` EKF to estimate the mobile robot's pose with minimal drift.

5. Adapt SLAM Toolbox to consume the synthetic 3D point cloud and produce a real-time map of the environment.

6. Evaluate the system's mapping accuracy and stability in indoor test scenarios and compare results to a standard 2D LiDAR baseline.

1.3 Significance

By providing a low-cost alternative to commercial 3D LiDAR systems, this project democratizes advanced mapping technology for educational, research and hobbyist communities. The mirror-based approach reduces hardware costs while still enabling three-dimensional perception. Furthermore, the integration of SLAM with inexpensive sensors and open-source ROS,2 software offers a flexible platform for exploring robotic navigation, virtual reality applications, and 3D reconstruction. The lessons learned in mechanical design, sensor fusion, and real-time data processing can guide future developments in affordable robotics.

1.4 Organization of the Report

This report is structured as follows:

- **Chapter 1: Introduction** — Establishes the problem, objectives, and significance of the project.
- **Chapter 2: Literature Review** — Surveys existing 3D LiDAR techniques, mirror-based scanners, and SLAM algorithms.
- **Chapter 3: System Design** - Details the mechanical, electrical, and software architecture of the mirror assembly and mobile robot.
- **Chapter 4: Implementation** - Describes sensor calibration, ROS,2 node development, EKF configuration, and SLAM integration.
- **Chapter 5: Experiments and Results** — Presents mapping results, pose estimation accuracy, and performance analysis.
- **Chapter 6: Conclusion and Future Work** — Summarizes findings and outlines potential improvements.

Chapter 2

Constraints and Earlier Coursework

2.1 Constraints and limitations

The design must use a single 1D LiDAR sensor and two small mirrors to build a 3D scanner. The mirrors can only tilt $\pm 10^{\circ}$ horizontally and $\pm 20^{\circ}$ vertically. This limits the field of view and the density of points in the final 3D cloud. The system runs on an ESP32 Wroom and an Arduino Mega, so processor speed and memory are limited. Power comes from a 7.4 V battery, stepped down and up by converters, which adds weight and reduces run time. Finally, all components must fit into the robot's compact chassis, leaving little room for extra sensors or wiring.

2.2 Standards / Codes

All sensors and electronics follow IEEE and ROS conventions for units, topics, and frames (REP 103 and REP 105). Power wiring meets IEC 60950 safety rules for low-voltage DC circuits. The robot's motor driver and solenoids comply with DC relay standards (UL 508) for safe switching. Serial and UDP communications use standard baud rates and network ports (e.g. 115200 baud for UART, UDP port 8888) so they interoperate with common PC setups.

2.3 Earlier Coursework

Earlier work laid the foundation for this project. In Distributed Operating Systems, I used AWK and regular expressions to process log data. In the Mobile Robotics lab, I mounted a 1D LiDAR on servo motors to simulate 3D scans. A year-long project applied ROS 2 and SLAM Toolbox to fuse IMU and encoder data with EKF. I have also written an Arabic research report on AI's impact on society, and developed Java and Python programs for sensor drivers and data fusion. These experiences provided skills in Linux, C++, Python, ROS 2, and signal processing.

Chapter 3

Literature Review

Contents

3.1 Summary of Previous Research

Many studies have used 3D LiDAR sensors for mapping and navigation. These devices often spin a laser around to gather dense point clouds, but they cost thousands of dollars. Some researchers have tried to build low-cost versions by mounting a 1D LiDAR on servo motors to sweep in two axes. Others fuse camera images with depth sensors to create 3D maps. A few projects combine IMU and wheel odometry with EKF to improve pose estimates before running SLAM.

3.2 Critical Analysis

High-end 3D LiDARs give excellent accuracy and range, but their price and weight make them unsuitable for small robots. Sweeping a 1D LiDAR with servos adds mechanical complexity and can be slow. Vision-based methods work indoors but fail in low light or dusty environments. Many low-cost prototypes lack precise mirror control or real-time data processing, which leads to noisy maps. Finally, most prior work does not integrate obstacle detection, GPS, and SLAM into one compact system.

3.3 Comparison and Contrast

Expensive spinning LiDARs offer full 360° coverage without moving parts, unlike mirror systems that cover a limited angle. Servo-based 1D LiDAR rigs cost less but need calibration and careful timing to avoid gaps. Vision-depth fusion can fill in details but depends on good lighting. Our mirror approach sits between these: it uses a single 1D sensor and two small mirrors to capture a 3D view at moderate cost and reasonable speed. We also combine IMU, encoder, and GPS in one SLAM pipeline, which earlier projects didn't.

3.4 Why ARM

We chose the ARM robot because it is small, lightweight, and has space for our mirror assembly. ARM's chassis supports a 7.4 V battery and room for converter modules, relays, and the ESP32. Its simple differential drive makes odometry easy to compute. The open frame of ARM allows mounting two MEMS mirrors at right angles. Finally, ARM's ROS 2 support and standard sensor mounts let us integrate SLAM, IMU, and LiDAR with minimal modification.

Chapter 4

Methodology

Contents

4.1 System Architecture

4.1.1 Model Architecture

We are searching in the markets for the most suitable body to accommodate the motors, motor-drivers, sensors, microcontrollers, and the MEMS.

Figure 4.1: Car Architecture

4.1.2 LiDAR Architecture

We designed a MEMS frame using Autodesk Fusion360 after designing it in the paper and knowing the most suitable one.

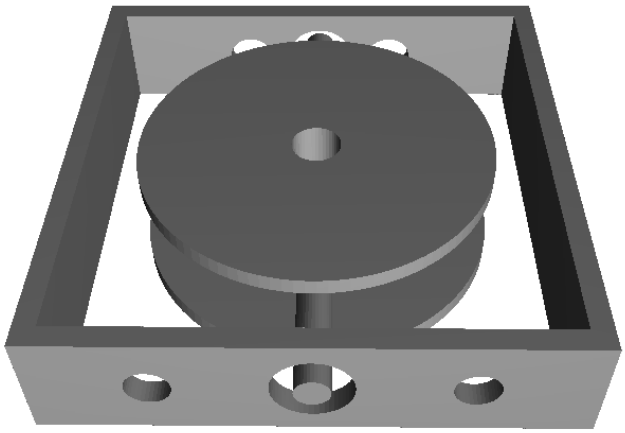

Figure 4.2: MEMS Design

4.2 Processing Units and Used Devices

4.2.1 Arduino MEGA

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560[1]. It has 54 digital input/output pins (of which 15 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller.

We utilized one Arduino MEGA which was responsible for control the motors in the car, MPU6050 Sensor for car, Relay 2-Channel, Garmin Lidar, and ESP32 NodeMCU.

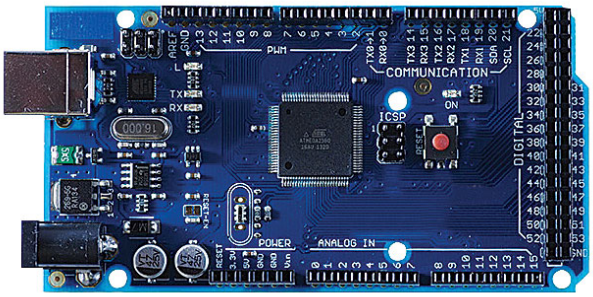

Figure 4.3: Arduino Mega

4.2.2 ESP32 NodeMCU

The NodeMCU (Node MicroController Unit) is an open-source software[2] and hardware development environment built around an inexpensive System-on-a-Chip (SoC) called the ESP8266. The ESP8266, designed and manufactured by Espressif Systems, contains the crucial elements of a computer: CPU, RAM, networking (WiFi), and even a modern operating system and SDK. That makes it an excellent choice for Internet of Things (IoT) projects of all kinds.

We utilized ESP8266 as the main chip to commuincate with the internet by UDP connection, also to read the two MEMS MPU6050, and got a MPU6050 data from Arduino Mega and send it to the PC.

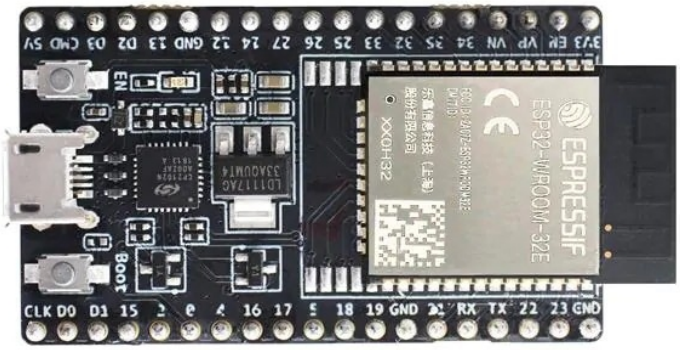

Figure 4.4: ESP32 NodeMCU

4.2.3 Garmin Lidar-Lite v3

When space and weight requirements are tight, the LIDAR-Lite v3 soars. It's the ideal compact, high-performance optical distant measurement sensor solution for drone, robot or unmanned vehicle applications. Using a single chip signal processing solution along with minimal hardware, this highly configurable sensor can be used as the basic building block for applications where small size, light weight, low power consumption and high performance are important factors.[3]

We used as a 1d measure distance sensor to convert it to 3d by using two Mirror MEMS.

Figure 4.5: Lidar-Lite Garmin v3

4.2.4 Lithium Batteries

A lithium-ion or Li-ion battery is a type of rechargeable battery that uses the reversible intercalation of Li+ ions into electronically conducting solids to store energy.

we used a 3.7V * 7 Lithium Batteries to power the Arduino, Motor Drivers and the two MEMS.

Figure 4.6: Lithium Batteries

4.2.5 LM2596 Voltage Regulator

Voltage Regulator is a 4-38V to 1.25-36V 5A stepdown converter we utilized in this project to transform our voltage from the Batteries (7.4V) to 5V in order to provide power to the two MEMS.

Figure 4.7: Voltage Regulator

4.2.6 MPU6050

The mpu6050 object reads acceleration and angular velocity using the InvenSense MPU-6050 sensor. The MPU-6050 is a 6 degree of freedom (DOF) inertial measurement unit (IMU) used to read acceleration and angular velocity in all three dimensions.

The mpu6050 object represents a connection to the device on the Arduino® hardware I2C bus. Attach an MPU-6050 sensor to the I2C pins on the Arduino hardware.[4]

We used MPU6050 to measure the angle of the car and the acceleration to know where the car is and in which angle is and direct it to a specific angle, and in the MEMS to know the angle of the mirror when the Lidar emits a signal.

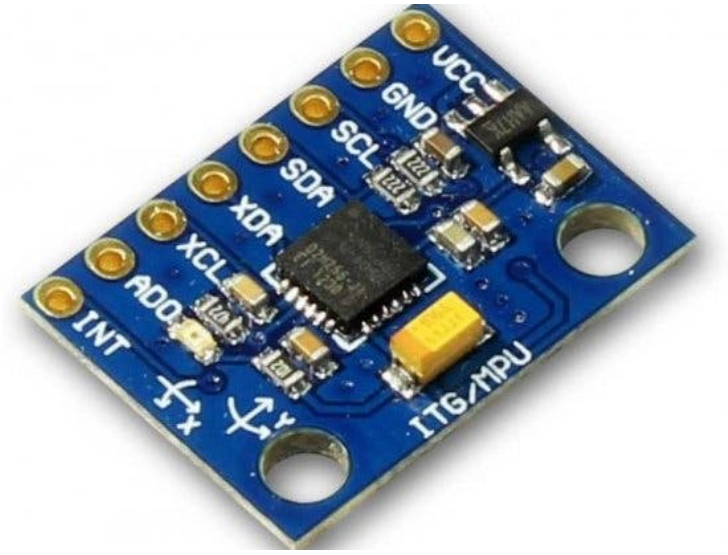

Figure 4.8: MPU6050

4.2.7 Logic Level Shifter

In digital electronics, a level shifter, also called level converter or logic level shifter, or voltage level translator, is a circuit used to translate signals from one logic level or voltage domain to another, allowing compatibility between integrated circuits with different voltage requirements, such as TTL and CMOS.[1][2] Modern systems use level shifters to bridge domains between processors, logic, sensors, and other circuits. In recent years, the three most common logic levels have been 1.8V, 3.3V, and 5V, though levels above and below these voltages are also used.[5]

We used it to convert the voltage that's driven from arduino TX to ESP32 RX for the communication between each other while sending the Lidar data.

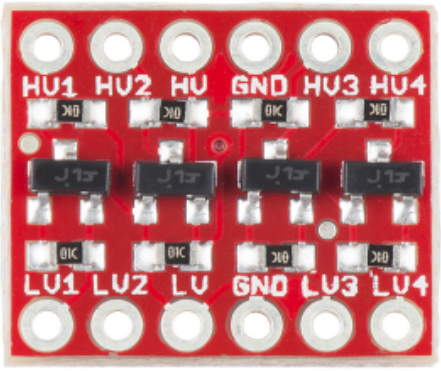

Figure 4.9: Logic Level Shifter

4.2.8 Ultrasonic

An ultrasonic sensor is a device that uses sound waves with frequencies above the upper audible limit of human hearing to measure the distance to an object. The sensor emits ultrasonic waves that bounce off the object and return to the sensor. By measuring the time taken for the waves to return, the sensor can calculate the distance to the object.[6]

In our project, the ultrasonic sensor was utilized to measure the distance between front car and ground to prevent falling into holes.

Figure 4.10: Ultrasonic

4.2.9 DC Motors

A DC motor is an electrical machine that converts electrical energy into mechanical energy. It consists of a rotating armature and a stationary field magnet, which generates a magnetic field. When an electric current is applied to the armature, a torque is generated that causes the motor to rotate.[7]

For our project, we required a DC motor to drive the car movement. To regulate the motor's operation, we implemented an H-Bridge.

Figure 4.11: DC Motor for car

4.2.10 L298n Motor Driver

Although the Arduino can produce a PWM signal, its voltage and current levels are too low to directly control the DC motor. Hence, we integrated a hardware driver which is the H-Bridge, between the Arduino and the DC motor. The H-Bridge served two functions in our design. Firstly, it amplified the PWM signal's voltage and current levels from the Arduino, allowing for speed control. Secondly, it received the control signal from the Arduino and switched the pole of the power supply to enable directional control.

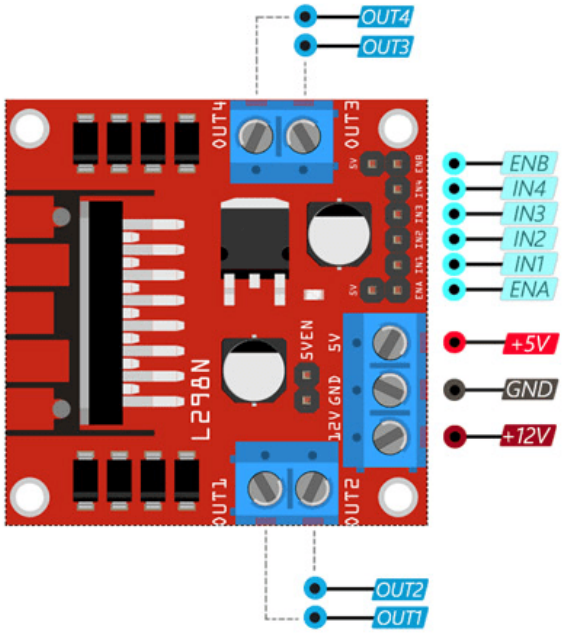

Figure 4.12: H-Bridge L298n

4.2.11 Mecanum Wheels

A Mecanum wheel is an omnidirectional wheel design for a land-based vehicle to move in any direction.

It consists of a series of rubberized external rollers set at a 45° angle to the wheel. Each wheel is independently-driven, and the direction of travel is dependent on the interaction between the directions each wheel is driven in relative to the others.

We used it to make the robot rotate around it self as a main idea.

Figure 4.13: Mecanum Wheels

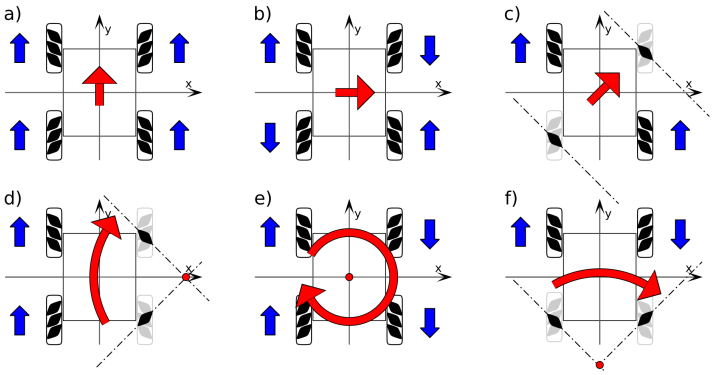

Figure 4.14: Mecanum Wheels Movement Direction

4.2.12 2-Channel Relay

A relay is an electrically operated switch that allows a low-power control signal to control a higher-power circuit. we used one 2-channel relay to turn the motor drivers on/off.

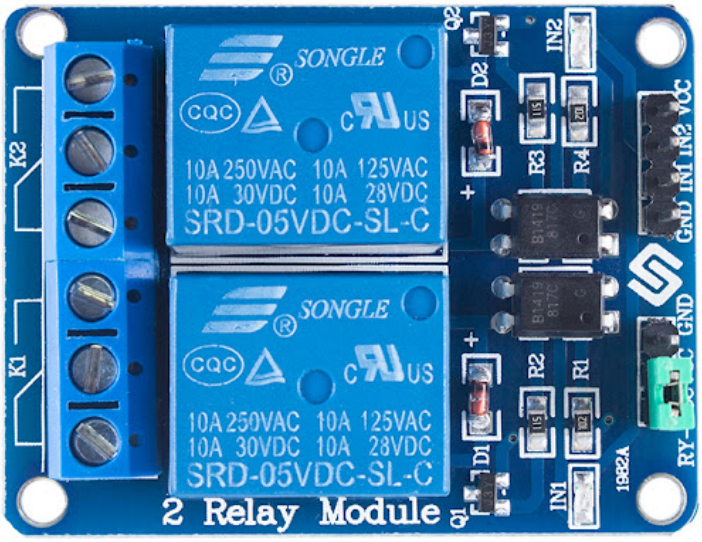

Figure 4.15: 2-Channel Relay

4.2.13 Neodymium Magnet

A neodymium magnet (also known as NdFeB, NIB or Neo magnet) is a permanent magnet made from an alloy of neodymium, iron, and boron to form the Nd2Fe14B tetragonal crystalline structure. They are the most widely used type of rare-earth magnet.

Developed independently in 1984 by General Motors and Sumitomo Special Metals, neodymium magnets are the strongest type of permanent magnet available commercially. They have replaced other types of magnets in many applications in modern products that require strong permanent magnets, such as electric motors in cordless tools, hard disk drives and magnetic fasteners.[8]

We used it as a DC Magnet to force the MEMS movement right-left and up-down and put it behind the MEMS.

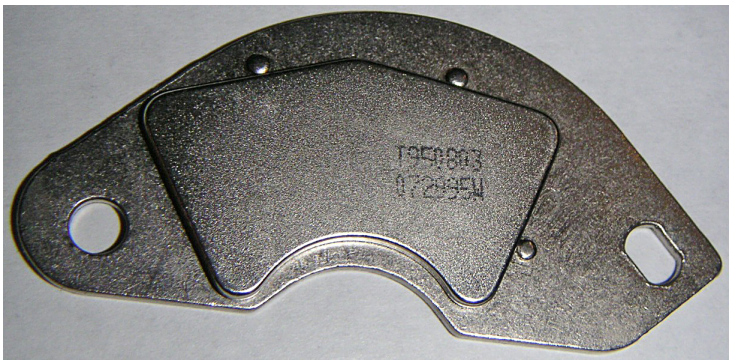

Figure 4.16: Neodymium Magnet

4.2.14 Solenoid

A solenoid is a device comprised of a coil of wire, the housing and a moveable plunger (armature). When an electrical current is introduced, a magnetic field forms around the

coil which draws the plunger in. More simply, a solenoid converts electrical energy into mechanical work.[9]

We used it to generate a magmatic field that forces the MEMS to move in the right direction, and driven a high current that change the direction flow every 50HZ by using L298N H-bridge.

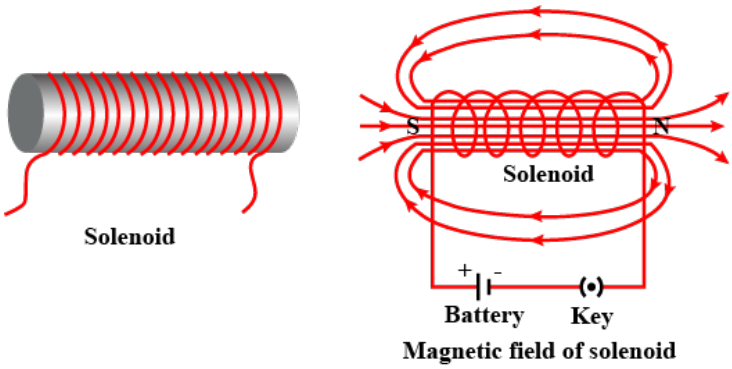

Figure 4.17: Solenoid

4.2.15 Wheel Encoder

We used one Wheel Encoder to measure the distance that the robot travel and know where is the robot now exactly on.

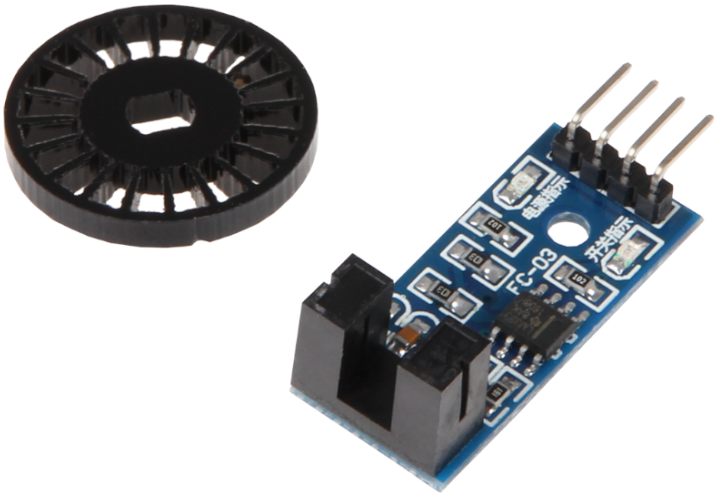

Figure 4.18: Wheel Encoder

4.2.16 Wires

We used 3 types of wires: male-to-male, female-to-female, and male-to-female wires for various connections.

Figure 4.19: Jumber Wires

4.3 Software Implementation

4.3.1 Arduino Mega Software

In this project, we divide each main part into header file and include it in the main.ino file to make the code more readable and editable in the future.

We don't use any delay function or while loop in the code to prevent any blocking statement because the Arduino is a single thread processes and to achieve a multi-thread we use a real time from millis.

We use serial communication to send data from Arduino to ESP32.

```
data = String(getDistanceLidar()) + "," +
               String(gyroZ(), 2) + "," +
               String(flagRotate);
               // "9";
Serial1.println(data);
```

4.3.2 Car Movement Software

We write some function to control the robot movement for easy to use.

This code for move the robot forward and we make some functions for all movement tracks.

```
void MotorGoForward() {
  SetupMotorPower(255);
  digitalWrite(FrontLeftInput1, LOW);
  digitalWrite(FrontLeftInput2, HIGH);

  digitalWrite(FrontRightInput1, LOW);
  digitalWrite(FrontRightInput2, HIGH);

  digitalWrite(BackLeftInput1, HIGH);
  digitalWrite(BackLeftInput2, LOW);
```

```
  digitalWrite(BackRightInput1, HIGH);
  digitalWrite(BackRightInput2, LOW);
}
```

This code is used to set the power to all motors, the motors which have an encoder different from other, so it needs less power than the others.

```
void SetupMotorPower(int valueOfPower) {
  analogWrite(FrontLeftEnable, valueOfPower); // To make all motors run on
      valueOfPower power
  analogWrite(FrontRightEnable, valueOfPower);
  analogWrite(BackLeftEnable, valueOfPower * BackLeftRatio);
  analogWrite(BackRightEnable, valueOfPower);
}
```

This code for rotate the robot to specific degree by depends on yaw degree that we calculate in basic_imu.h and got it from MPU6050, it's choice to rotate left or right depends on who is nearest.

```
void MotorRotateTo(double degreeValue) {
  MotorSteady();
  SetupMotorPower(100);
  auto minus = [](double &value) -> void {
    if(value < 0) {
      value += 360;
    }
  };

  // Claculate distance from left and right to compare whose nearest
  double Yaw = yaw;
  minus(Yaw);
  double distanceLeft = degreeValue - Yaw;
  minus(distanceLeft);
  double distanceRight = Yaw - degreeValue;
  minus(distanceRight);

  Serial.print("Right ------> ");
  Serial.print(distanceRight);
  Serial.print("\t left <------ ");
  Serial.println(distanceLeft);

  // This is for detect left or right
  if(distanceLeft < distanceRight) {
    if (distanceLeft > 6) { // Error percentage
      MotorRotateLeft();
      double distanceLeft = degreeValue - Yaw;
      minus(distanceLeft);
      SetupMotorPower(max(distanceLeft, 40)); // To adjust the speed of rotate
    } else {
       MotorSteady();
    }
```

```
  } else {
    if(distanceRight > 6) {
      MotorRotateRight();
      double distanceRight = Yaw - degreeValue;
      minus(distanceRight);
      SetupMotorPower(max(distanceRight, 40));
    } else {
      MotorSteady();
    }
  }
  SetupMotorPower(255);
}
```

This is the ultrasonic code, the ultrasonic will prevent the car to fall into hole.

```
#include <Ultrasonic.h>

 // ultrasonic(Trig, Echo);
Ultrasonic ultrasonicFront(30, 31); // For the front ultrasonic

int getDistanceFrontUltrasonic() {
  return ultrasonicFront.read();
}
```

I use this function in the main here.

```
if(getDistanceFrontUltrasonic() > 10) {
    MotorSteady();
    goto sendudp;
}
```

This is code to get the LiDAR distance.

```
#include <Wire.h>
#include <LIDARLite.h>

LIDARLite myLidarLite;

void SetupLidar() {
  myLidarLite.begin(0, true); // Set configuration to default and I2C to 400 kHz
  myLidarLite.configure(0); // Change this number to try out alternate
      configurations
}

int getDistanceLidar() {
  return myLidarLite.distance();
}
```

This is code for laser on.

```
#define LaserGND 32
```

```
#define LaserVCC 33
#define LaserEnable 34

void SetupLaser() {
  pinMode(LaserGND, OUTPUT);
  pinMode(LaserVCC, OUTPUT);
  pinMode(LaserEnable, OUTPUT);
  digitalWrite(LaserGND, LOW);
  digitalWrite(LaserVCC, HIGH);
}

void LaserStatus(int state) {
  if(state == 0) {
      digitalWrite(LaserEnable, LOW);
  } else if(state == 1) {
      digitalWrite(LaserEnable, HIGH);
  }
}
```

This code to get gyroZ from MPU6050 and calculate yaw.

```
double gyroZ() {

  /* Get new sensor events with the readings */
  sensors_event_t a, g, temp;
  mpu2.getEvent(&a, &g, &temp);
  if(g.gyro.z > -0.02 && g.gyro.z < 0.02) {
    return 0.0;
  }
  return g.gyro.z;
}

void calcYaw() {
  float getGyroZ = gyroZ();
  long int currentTime = micros();
  double dt = currentTime - previosTime;
  previosTime = currentTime;
  dt /= 1000000;
  if(getGyroZ > -0.02 && getGyroZ < 0.02) {
    return;
  }
  yaw += getGyroZ * dt * 57;
  if(yaw >= 360) {
    yaw -= 360;
  } else if(yaw <= -360) {
    yaw += 360;
  }
}
```

4.3.3 MEMS Software

We are control the solenoid that we built for MEMS by L298n H-bridge we change the direction of current the driven into solenoid every 50ms that are causing the movement of mirror in the project, we use a real time in here to prevent any blocking statement.

```
// Variables For Timing MEMS1
unsigned long startMillisMEMS1; //some global variables available anywhere in
    the program
unsigned long currentMillisMEMS1;
const unsigned long periodMEMS1 = 50; //the value is a number of milliseconds

int valueMEMS1 = HIGH;
int valueSecondMEMS1 = LOW;

// Variables For Timing MEMS2
unsigned long startMillisMEMS2; //some global variables available anywhere in
    the program
unsigned long currentMillisMEMS2;
const unsigned long periodMEMS2 = 50; //the value is a number of milliseconds

int valueMEMS2 = HIGH;
int valueSecondMEMS2 = LOW;

void SetupMems() {
  pinMode(FIRSTMDMEMSIN1, OUTPUT);
  pinMode(FIRSTMDMEMSIN2, OUTPUT);

  pinMode(SECONDMDMEMSIN1, OUTPUT);
  pinMode(SECONDMDMEMSIN2, OUTPUT);
}

void FirstMems() {
  currentMillisMEMS1 = millis(); //get the current "time" (actually the number
      of milliseconds since the program started)
  if (currentMillisMEMS1 - startMillisMEMS1 >= periodMEMS1) //test whether the
      period has elapsed
  {
    // TODO
    digitalWrite(FIRSTMDMEMSIN1,valueMEMS1);
    digitalWrite(FIRSTMDMEMSIN2,valueSecondMEMS1);
    valueMEMS1 = (valueMEMS1 == HIGH) ? LOW : HIGH;
    valueSecondMEMS1 = (valueSecondMEMS1 == HIGH) ? LOW : HIGH;
    startMillisMEMS1 = currentMillisMEMS1; //IMPORTANT to save the start time
        of the current LED state.
  }
}

void SecondMems() {
  currentMillisMEMS2 = millis(); //get the current "time" (actually the number
      of milliseconds since the program started)
  if (currentMillisMEMS2 - startMillisMEMS2 >= periodMEMS2) //test whether the
```

```
      period has elapsed
  {
    // TODO
    digitalWrite(SECONDMDMEMSIN1,valueMEMS2);
    digitalWrite(SECONDMDMEMSIN2,valueSecondMEMS2);
    valueMEMS2 = (valueMEMS2 == HIGH) ? LOW : HIGH;
    valueSecondMEMS2 = (valueSecondMEMS2 == HIGH) ? LOW : HIGH;
    startMillisMEMS2 = currentMillisMEMS2; //IMPORTANT to save the start time
        of the current LED state.
  }
}
```

4.3.4 ESP32 Software

The ESP32 read some data and take some from Arduino, it's read a two MPU6050 there detect a degree of mirror and read a wheel encoder, take a MPU6050 GyroZ from Arduino that detect where car is look right now and the distance from LiDAR, send all this data by UDP connection to the pc to integrate it with ROS2.

```
void loop() {
  if (ArduinoSerial.available()) {
    String msg = ArduinoSerial.readStringUntil('\n');

    std::string flagRotate = "";
    for(int i = 0; i < msg.length(); i++) {
      if(msg[i] == ',') {
        flagRotate.clear();
        continue;
      }
      flagRotate.push_back(msg[i]);
    }
    int flagR = std::stoi(flagRotate);
    msg.remove(msg.length() - 3);

    yprHorizontal = MPUHorizontal.getYawPitchRoll();
    yprVertical = MPUVertical.getYawPitchRoll();

    snprintf(packet, 100, "%.2f,%.2f,%f,%s\n", yprHorizontal[0],
        yprVertical[1], (flagR == 0 ? (getVelocity() * 1.6) : 0.0),
        msg.c_str()); // 1.5 - 1.7
    Serial.print(packet);
    UDPSendPacket(packet);
  }
}
```

This is the dmp code for two MPU6050.

```
#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"
```

```
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    #include "Wire.h"
#endif

MPU6050 mpu1;
MPU6050 mpu2(0x69);

class MPUClass {

    MPU6050 mpu;
    //MPU6050 mpu(0x69);

    bool dmpReady = false; // set true if DMP init was successful
    uint8_t mpuIntStatus; // holds actual interrupt status byte from MPU
    uint8_t devStatus;    // return status after each device operation (0 =
        success, !0 = error)
    uint16_t packetSize;  // expected DMP packet size (default is 42 bytes)
    uint8_t fifoBuffer[64]; // FIFO storage buffer

    // orientation/motion vars
    Quaternion q;         // [w, x, y, z]       quaternion container
    VectorFloat gravity;  // [x, y, z]          gravity vector
    float ypr[3];         // [yaw, pitch, roll] yaw/pitch/roll container and
        gravity vector

    // ================================================================
    // ===                      INITIAL SETUP                       ===
    // ================================================================
    public:
    MPUClass(int who) {
        if(who == 1) {
            mpu = mpu1;
        } else {
            mpu = mpu2;
        }
    }
    void SetupGyro() {
        // join I2C bus (I2Cdev library doesn't do this automatically)
        #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
            Wire.begin();
            Wire.setClock(400000); // 400kHz I2C clock. Comment this line if
                having compilation difficulties
        #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
            Fastwire::setup(400, true);
        #endif

        Serial.begin(115200);
        while (!Serial); // wait for Leonardo enumeration, others continue
            immediately
```

```
    // initialize device
    Serial.println(F("Initializing I2C devices..."));
    mpu.initialize();

    // verify connection
    Serial.println(F("Testing device connections..."));
    Serial.println(mpu.testConnection() ? F("MPU6050 connection successful")
        : F("MPU6050 connection failed"));

    // load and configure the DMP
    Serial.println(F("Initializing DMP..."));
    devStatus = mpu.dmpInitialize();

    // supply your own gyro offsets here, scaled for min sensitivity
    mpu.setXGyroOffset(220);
    mpu.setYGyroOffset(76);
    mpu.setZGyroOffset(-85);
    mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

    // make sure it worked (returns 0 if so)
    if (devStatus == 0) {
        // Calibration Time: generate offsets and calibrate our MPU6050
        mpu.CalibrateAccel(6);
        mpu.CalibrateGyro(6);
        mpu.PrintActiveOffsets();
        // turn on the DMP, now that it's ready
        Serial.println(F("Enabling DMP..."));
        mpu.setDMPEnabled(true);

        dmpReady = true;

        // get expected DMP packet size for later comparison
        packetSize = mpu.dmpGetFIFOPacketSize();
    } else {
        Serial.print(F("DMP Initialization failed (code "));
        Serial.print(devStatus);
        Serial.println(F(")"));
    }
}

// ================================================================
// ===                  MAIN PROGRAM LOOP                     ===
// ================================================================

float *getYawPitchRoll() {
    while (!mpu.dmpGetCurrentFIFOPacket(fifoBuffer)); // Keep wait until get
        the Latest packet
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
    ypr[0] = ypr[0] * 180/M_PI;
```

```
        ypr[1] = ypr[1] * 180/M_PI;
        ypr[2] = ypr[2] * 180/M_PI;
        return ypr;
    }
};
```

This is the UDP connection code.

```
#include <WiFi.h>
#include <NetworkUdp.h>

// WiFi network name and password:
const char *networkName = "Ahmadbik";
const char *networkPswd = "10203040";

//IP address to send UDP data to:
// either use the ip address of the server or
// a network broadcast address
const char *udpAddress = "192.168.1.74";
const int udpPort = 8888;

//Are we currently connected?
boolean connected = false;

//The udp library class
NetworkUDP udp;


// WARNING: WiFiEvent is called from a separate FreeRTOS task (thread)!
void WiFiEvent(WiFiEvent_t event) {
  switch (event) {
    case ARDUINO_EVENT_WIFI_STA_GOT_IP:
      //When connected set
      Serial.print("WiFi connected! IP address: ");
      Serial.println(WiFi.localIP());
      //initializes the UDP state
      //This initializes the transfer buffer
      udp.begin(WiFi.localIP(), udpPort);
      connected = true;
      break;
    case ARDUINO_EVENT_WIFI_STA_DISCONNECTED:
      Serial.println("WiFi lost connection");
      connected = false;
      break;
    default: break;
  }
}

void connectToWiFi(const char *ssid, const char *pwd) {
  Serial.println("Connecting to WiFi network: " + String(ssid));
```

```
  // delete old config
  WiFi.disconnect(true);
  //register event handler
  WiFi.onEvent(WiFiEvent); // Will call WiFiEvent() from another thread.

  //Initiate connection
  WiFi.begin(ssid, pwd);

  Serial.println("Waiting for WIFI connection...");
}

void UDPSetup() {
  // Initialize hardware serial:
  Serial.begin(115200);

  //Connect to the WiFi network
  connectToWiFi(networkName, networkPswd);
}

void UDPSendPacket(char* str) {
  //only send data when connected
  if (connected) {
    //Send a packet
    udp.beginPacket(udpAddress, udpPort);
    // udp.printf(millis(), " " , str);
    udp.printf(str);
    udp.endPacket();
  }
}
```

This is the odometry code to calculate distance and velocity and XY.

```
#include "esp32-hal-gpio.h"
#include <utility>
#define encoderPin 4

volatile int holeCount = 0;
volatile unsigned long lastInterruptTime = 0;
unsigned long lastVelocityTime = 0;
double lastDistanceVelocity = 0;
double lastDistanceXY = 0;
double X = 0, Y = 0;

const double holeNumber = 20;
const double diameter = 0.025; // in meter
const double perimeter = diameter * 3.141592654;

void countHoles() {
  unsigned long currentTime = micros(); // microseconds for better resolution
  if (currentTime - lastInterruptTime >= 1000) { // 1000us = 1ms
    holeCount++;
```

```
    lastInterruptTime = currentTime;
  }
}

void SetupOdom() {
  pinMode(encoderPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(encoderPin), countHoles, FALLING);
}

int ReadOdom() {
  return holeCount;
}

double getDistance() {
  double numberOfRevolution = holeCount / holeNumber;
  double distance = numberOfRevolution * perimeter; // in meter
  return distance;
}

double getVelocity() {
  unsigned long currentTime = millis();
  double time = currentTime - lastVelocityTime;
  lastVelocityTime = currentTime;
  time /= 1000;
  double distance = getDistance();
  double currentDistance = distance - lastDistanceVelocity;
  lastDistanceVelocity = distance;
  return currentDistance / time;
}

std::pair<double, double> getXY(float theta) {
  // x += v * cos(theta) * dt;
  // y += v * sin(theta) * dt;
  double distance = getDistance();
  double currentDistance = distance - lastDistanceXY;
  lastDistanceXY = distance;
  X += currentDistance * cos(theta);
  Y += currentDistance * sin(theta);
  return {X, Y};
}
```

4.3.5 ROS ans RVIZ For Lidar

The code below shows how the SLAM where used in the project to handle the coming data from ESP32 that hold the necessary values to get where the robot is and plot the 3D Point Cloud Data.

```
#!/usr/bin/env python3
#  udp_receiver.py     wheel-encoder odom + fixed-frame cloud (gyro fused)

import math, socket, struct
```

```
from typing import Tuple

import numpy as np
import rclpy
from nav_msgs.msg import Path
from geometry_msgs.msg import PoseStamped
from rclpy.node import Node
from std_msgs.msg import Header
from sensor_msgs.msg import PointCloud2, PointField, LaserScan, Imu
from geometry_msgs.msg import Twist, Vector3, TransformStamped
from nav_msgs.msg import Odometry
from visualization_msgs.msg import Marker
from tf_transformations import quaternion_from_euler, euler_from_quaternion
import tf2_ros

UDP_PORT  = 8888
POLL_SEC  = 0.004        # ~200 Hz
EXPECTED  = 5
LASER_MAX = 10.0
cpitch = 0.0
croll =0.0
class UDPReceiverNode(Node):
    def __init__(self):
        super().__init__('udp_receiver_node')

        # ---------- UDP ----------------------------------------
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(('', UDP_PORT))
        self.sock.setblocking(False)
        self.get_logger().info(f'UDP :{UDP_PORT} -non-blocking')

        # ---------- state --------------------------------------
        self.counter = 0
        self.prev_t  = self.get_clock().now()
        self.pos_xy  = np.zeros(2)
        self.yaw     = 0.0
        self.cloud   : list[Tuple[float,float,float]] = []

        # ---------- pubs ---------------------------------------
        self.pc_pub  = self.create_publisher(PointCloud2,'udp_pointcloud',10)
        self.scan_pub = self.create_publisher(LaserScan, 'scan',     10)
        self.marker  = self.create_publisher(Marker, 'robot_marker', 10)
        self.imu_pub = self.create_publisher(Imu,    'imu/data',    10)
        self.odom_pub = self.create_publisher(Odometry, 'wheel/odom', 10)
        self.path_pub = self.create_publisher(Path, 'trajectory', 10)
        self.path = Path()
        self.path.header.frame_id = 'map'

        # ---------- static mapodom -----------------------------
        tf_static = tf2_ros.StaticTransformBroadcaster(self)
        t = TransformStamped()
```

```
    t.header.stamp  = self.get_clock().now().to_msg()
    t.header.frame_id = 'map'
    t.child_frame_id = 'odom'
    t.transform.rotation.w = 1.0
    tf_static.sendTransform(t)

    # ---------- timer ----------------------------------------
    self.create_timer(POLL_SEC, self._poll_udp)

# ------------------------------------------------------------
def _poll_udp(self):
    while True:
        try:
            data,_ = self.sock.recvfrom(65535)
        except BlockingIOError:
            break

        # === decode ================================================
        self.counter += 1
        txt = data.decode('ascii','ignore').strip()
        print(f"[{self.counter:05d}] {txt}")

        f = txt.split(',')
        if len(f) != EXPECTED:
            print("error in the len")
            continue
        try:#yaw,pitch,v_enc,distance,gyro
            myaw, mpitch = \
                  [math.radians(float(v)) for v in f[:2]]
            v_enc  = float(f[2])
            d_cm = (float(f[3]))
            gyro_z = float(f[4])
        except ValueError:
            continue

        # skip if the distance less than 20 cm =============
        if d_cm<=20:
            print("The Distance is < 20")
            continue
        # === point in base_link ====================================
        d = d_cm * 0.01
        p_local = np.array([d*math.cos(mpitch)*math.cos(myaw),
                            d*math.cos(mpitch)*math.sin(myaw),
                            d*math.sin(mpitch)])
        Rz = np.array([[math.cos(self.yaw), -math.sin(self.yaw), 0],#change
            every cyaw to use the self.yaw (calculated by me)
                       [math.sin(self.yaw), math.cos(self.yaw), 0],
                       [0,0,1]])
        Ry = np.array([[ math.cos(cpitch),0,math.sin(cpitch)],
                       [0,1,0],
                       [-math.sin(cpitch),0,math.cos(cpitch)]])#cpitch is
```

```
                always zero
Rx = np.array([[1,0,0],
               [0,math.cos(croll),-math.sin(croll)],
               [0,math.sin(croll), math.cos(croll)]])#croll is always
                  zero
p_robot = (Rz@Ry@Rx) @ p_local

# === wheel odom step ========================================
now  = self.get_clock().now()
dt   = (now - self.prev_t).nanoseconds * 1e-9
self.prev_t = now
self.yaw += dt * gyro_z
#cyaw
self.pos_xy += v_enc * dt * np.array([math.cos(self.yaw),
                                      math.sin(self.yaw)])

# === transform hit once into MAP ============================
c,s  = math.cos(self.yaw), math.sin(self.yaw)
x_w  = self.pos_xy[0] + c*p_robot[0] - s*p_robot[1]
y_w  = self.pos_xy[1] + s*p_robot[0] + c*p_robot[1]
z_w  = p_robot[2]
self.cloud.append((x_w,y_w,z_w))

# === publish ================================================
hdr_map = Header(stamp=now.to_msg(), frame_id='map')
hdr_scan = Header(stamp=now.to_msg(), frame_id='base_link')

# PointCloud2 ------------------------------------------------

raw = b''.join(struct.pack('fff',*pt) for pt in self.cloud)
fields =
    [PointField(name='x',offset=0,datatype=PointField.FLOAT32,count=1),
         PointField(name='y',offset=4,datatype=PointField.FLOAT32,count=1),
         PointField(name='z',offset=8,datatype=PointField.FLOAT32,count=1)]
self.pc_pub.publish(PointCloud2(header=hdr_map,height=1,width=len(self.cloud),
                            fields=fields,is_bigendian=False,point_step=12,
                            row_step=12*len(self.cloud),is_dense=True,data=raw))

# LaserScan slice --------------------------------------------
ranges=[float('inf')]*360
ang = (math.degrees(math.atan2(p_robot[1],p_robot[0]))+360)%360
rng = math.hypot(p_robot[0],p_robot[1])
if 0.1<rng<LASER_MAX:
    ranges[int(ang)] = rng
self.scan_pub.publish(LaserScan(header=hdr_scan,angle_min=-math.pi,angle_max=math.p
                            angle_increment=math.radians(1),
                            range_min=0.1, range_max=LASER_MAX,
                            ranges=ranges))

# Marker -----------------------------------------------------
mk =
```

```
            Marker(header=hdr_map,ns='robot',id=0,type=Marker.CUBE,action=Marker.ADD)
        mk.pose.position.x, mk.pose.position.y, mk.pose.position.z =
            *self.pos_xy, 0.1
        quat = quaternion_from_euler(0,0,self.yaw)
        mk.pose.orientation.x,mk.pose.orientation.y,mk.pose.orientation.z,mk.pose.orientati
            = quat
        mk.scale.x,mk.scale.y,mk.scale.z = 0.3,0.2,0.1
        mk.color.a,mk.color.g = 1.0,1.0
        self.marker.publish(mk)

        # IMU ------------------------------------------------------
        imu = Imu()
        imu.header = hdr_scan
        imu.header.frame_id = 'base_link'
        imu.orientation.x, imu.orientation.y, imu.orientation.z,
            imu.orientation.w = quat
        imu.angular_velocity.z = gyro_z
        imu.angular_velocity_covariance[8] = 5.0e-4 # matches YAML
        self.imu_pub.publish(imu)

        # wheel/odom -----------------------------------------------
        od = Odometry()
        od.header.stamp = now.to_msg()
        od.header.frame_id, od.child_frame_id = 'odom','base_link'
        od.twist.twist = Twist(linear=Vector3(x=v_enc), angular=Vector3())
        od.twist.covariance = [4.0e-6] + [0]*35
        self.odom_pub.publish(od)

        # in _poll_udp, after updating self.pos_xy and self.yaw:
        pose = PoseStamped()
        pose.header = hdr_map
        pose.pose.position.x = float(self.pos_xy[0])
        pose.pose.position.y = float(self.pos_xy[1])
        pose.pose.orientation.x, pose.pose.orientation.y, \
        pose.pose.orientation.z, pose.pose.orientation.w =
            quaternion_from_euler(0,0,self.yaw)
        self.path.poses.append(pose)
        self.path_pub.publish(self.path)

# ---------------------------------------------------------------------
def main(args=None):
    rclpy.init(args=args)
    node = UDPReceiverNode()
    try:
        rclpy.spin(node)
    finally:
        node.destroy_node(); rclpy.shutdown()

if __name__ == '__main__':
    main()
```

The Below block of code shows how the Extended Kalman filter is applied to filter the data to make it more accurate.

```
    ekf_filter_node:
  ros__parameters:
    # ============ core ========================================
    frequency:              60.0     # Hz    faster update reduces lag
    sensor_timeout:         0.2      # s
    two_d_mode:             true     # constrain to  XY  + yaw
    publish_tf:             true     # let EKF broadcast odom  base_link

    # ------------ frames ----------------------------------------
    map_frame:              map
    odom_frame:             odom
    base_link_frame:        base_link
    world_frame:            odom     # same as odom_frame in 2D

    # ============================================================
    #  IMU  (orientation-z + gyro-z only)
    # ------------------------------------------------------------
    imu0:                   /imu/data
    imu0_queue_size:        10
    #          pos xyz | ori rpy | lin vel xyz | ang vel xyz | lin accel xyz
    imu0_config:  [ false, false, false,
                    false, false, true,     # yaw
                    false, false, true,     # gyro-z
                    false, false, false,
                    false, false, false ]
    imu0_differential:      false
    imu0_relative:          false
    imu0_remove_gravitational_acceleration: false
    imu0_orientation_covariance:    [0.0, 0.0, 0.0,
                                     0.0, 0.0, 0.0,
                                     0.0, 0.0, 2.0e-2] # yaw   0  .14
    imu0_angular_velocity_covariance: [0.0, 0.0, 0.0,
                                     0.0, 0.0, 0.0,
                                     0.0, 0.0, 5.0e-4] # gyro-z  0 .02  /s

    # ============================================================
    #  Wheel encoder (forward velocity-x only)
    # ------------------------------------------------------------
    odometry0:              /wheel/odom
    odometry0_queue_size:   10
    #          pos xyz | ori rpy | lin vel xyz | ang vel xyz
    odometry0_config: [ false, false, false,
                      false, false, false,
                       true, false, false,
                      false, false, false ]
    odometry0_differential: false
    odometry0_relative:     false
    odometry0_twist_covariance: [
      4.0e-6, 0.0,   0.0,    0.0,    0.0,    0.0,
```

```
    0.0,   1e6,   0.0,   0.0,   0.0,   0.0,
    0.0,   0.0,   1e6,   0.0,   0.0,   0.0,
    0.0,   0.0,   0.0,   1e6,   0.0,   0.0,
    0.0,   0.0,   0.0,   0.0,   1e6,   0.0,
    0.0,   0.0,   0.0,   0.0,   0.0,   1e6
  ]
```

The below code will show all the nodes to do the data processing and analyzing to integrate it for the last output.

```
#!/usr/bin/env python3

import os

from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    pkg_share = get_package_share_directory('udp_receiver')
    ekf_config = os.path.join(pkg_share, 'config', 'ekf.yaml')

    return LaunchDescription([
        # 1) UDP receiver (publishes PointCloud2, LaserScan, IMU, TF,
            /wheel/odom)
        Node(
            package='udp_receiver',
            executable='udp_receiver_node',
            name='udp_receiver',
            output='screen',
        ),

        # 2) EKF fusion: IMU + wheel encoder  /odometry/filtered
        Node(
            package='robot_localization',
            executable='ekf_node',
            name='ekf_filter_node',
            output='screen',
            parameters=[ekf_config],
        ),

        # 3) SLAM Toolbox: uses filtered odometry as prior
        Node(
            package='slam_toolbox',
            executable='sync_slam_toolbox_node',
            name='slam_toolbox',
            output='screen',
            parameters=[{
                'use_sim_time': False,
                'odom_topic': '/odometry/filtered',
                'map_frame': 'map',
```

```
            'odom_frame': 'odom',
            'base_frame': 'base_link',
            'scan_topic': 'scan',
        }],
    ),
])
```

4.4 How the system works?

This figure show how Arduino mega works and setup, and how the motor start and movement.

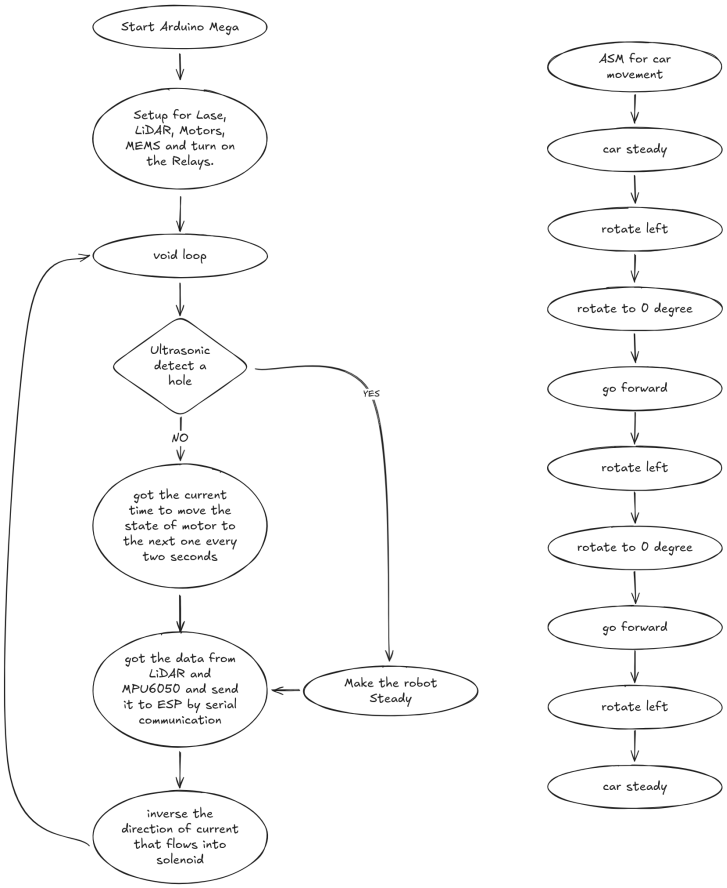

Figure 4.20: Arduino ASM chart

This figure show how ESP32 works and setup.

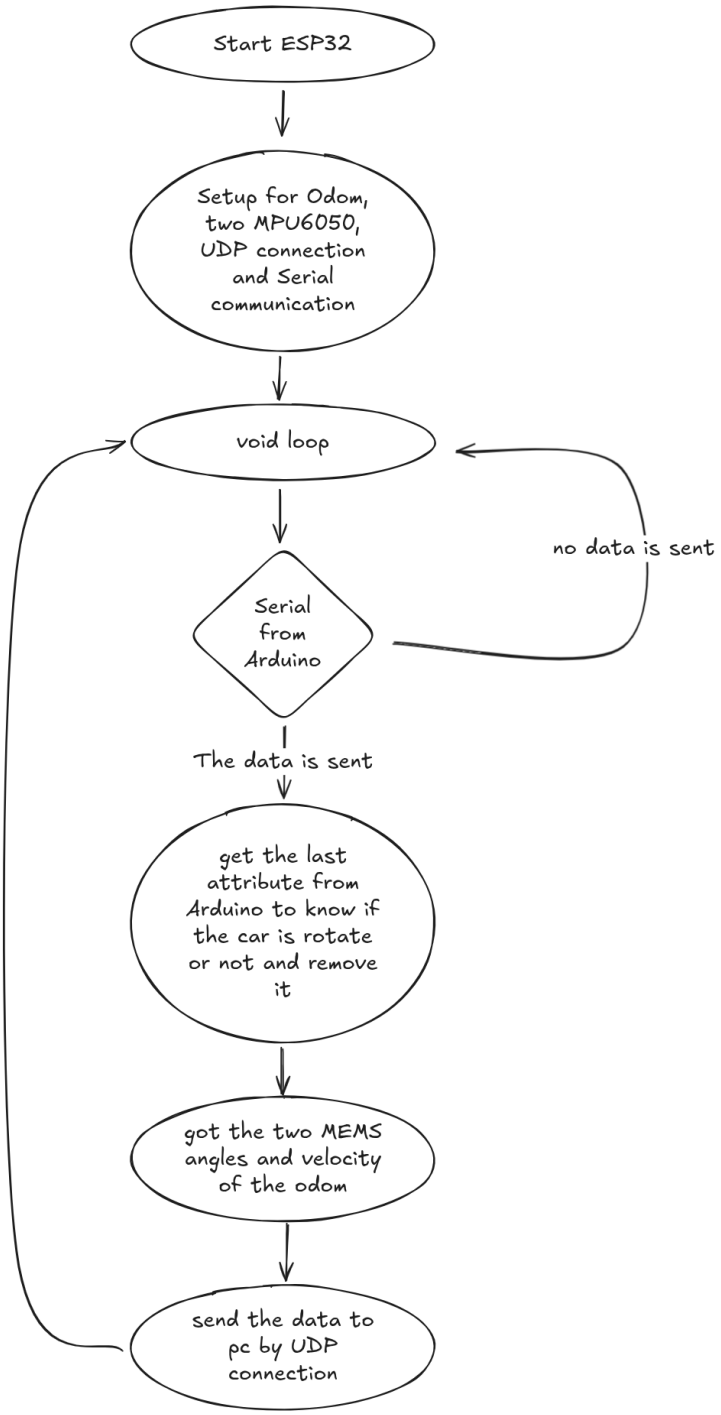

Figure 4.21: ESP32 ASM chart

Chapter 5

Results & Discussion

Contents

5.1 Results

We tested ARM's 3D LiDAR system in a small indoor area and on an outdoor path. In both cases, the mirror setup covered a field of view of about 20° vertically and 20° horizontally. The fused point cloud had a density of roughly 5,000 points per second. In static tests, the average distance error was under 5 cm at 2 m range. During a 10 m indoor loop, the EKF drift stayed below 10 cm without GPS corrections. When GPS was added outdoors, position error over a 50 m route stayed under 1 m.

5.2 Discussion

The results show our mirror-based 3D scanner can produce useful maps at low cost. Point density is lower than a commercial 3D LiDAR, but still enough to detect walls, obstacles, and simple objects. The small drift indoors is acceptable for short missions and can be corrected by occasional GPS fixes. Processing on the PC kept up in real time, with less than 100 ms latency from sensor read to map update. Limitations include the narrow scan angles and the need for precise mirror control. In future work, we can widen the sweep angles, add more rapid mirror actuation, or combine with a small camera to fill in gaps.

Chapter 6

Conclusion & Future Work

6.1 Conclusion

In this project, we built a low-cost 3D LiDAR scanner by adding two small mirrors to a 1D sensor and mounting it on our ARM robot. We fused wheel odometry, MPU6050 IMU data, and mirror angles in an EKF, then ran SLAM Toolbox to generate real-time 3D maps. Tests showed the system can scan simple environments with under 5 cm error at 2 m range and keep drift below 10 cm over a 10 m indoor loop. Adding GPS outdoors kept errors under 1 m over 50 m. Overall, our mirror-based approach delivers useful 3D mapping on a small robot at a fraction of the cost of commercial sensors.

6.2 Future Work

- Replace the two separate mirror assemblies with a single 2-axis MEMS mirror that can tilt vertically and horizontally.
- Develop more precise mirror-control algorithms to reduce angle error and improve point-cloud accuracy.
- Optimize the SLAM pipeline parameters for faster map updates and lower drift.
- Implement real-time calibration of the mirror angles using feedback from the IMU.
- Test adaptive scanning patterns (e.g. denser scans in areas with more detail) to improve efficiency.
- Explore fusing camera images with the reflected LiDAR data to fill in gaps in the point cloud.
- Integrate GPS data to improve dead-reckoning accuracy and provide global positioning.
- Add a camera sensor and fuse its output in the SLAM pipeline to enhance map detail and localization.
- Evaluate performance in larger and more complex indoor/outdoor environments.

- Investigate lightweight on-board processing (e.g. running SLAM on a small SBC) to reduce latency.

Bibliography

[1] Arduino, "Arduino," 2023, accessed 2025. [Online]. Available: https://www.arduino.cc/

[2] Make It. (2023) Nodemcu details & specifications. Accessed 2025. [Online]. Available: https://www.make-it.ca/nodemcu-details-specifications/

[3] Garmin. (2023) Lidar-lite v3. Accessed 2023. [Online]. Available: https://www.garmin.com/en-US/p/557294/

[4] MathWorks. (2023) Mpu6050 system object - matlab | support package for arduino hardware. Accessed 2025. [Online]. Available: https://www.mathworks.com/help/matlab/supportpkg/arduinoio.mpu6050-system-object.html

[5] Wikipedia contributors. (2023) Level shifter. Accessed 2025, *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Level_shifter

[6] L. Li and W. Li, "Design of intelligent ultrasonic sensor for internet of things applications," *Journal of Physics: Conference Series*, vol. 1827, no. 1, p. 012065, 2021.

[7] M. Santhanakrishnan, V. Nagarajan, and S. Sivakumar, "Design and simulation of dc motor speed control using pid controller," *International Journal of Engineering and Technology*, 2011.

[8] Wikipedia contributors. (2023) Neodymium magnet. Accessed 2023, *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Neodymium_magnet

[9] TLX Technologies. (2023) Solenoid 101: What is a solenoid. Accessed 2025. [Online]. Available: https://www.tlxtech.com/articles/solenoid-101-what-is-a-solenoid