

Here is the English translation of the provided `mBot机器人--从零开始的ROS2智能移动机器人制作教程.md` file.

mBot Robot - A Hands-on Tutorial for Building a Smart Mobile Robot with ROS2 from Scratch

Document Description

This document is a comprehensive tutorial for building an mBot smart mobile robot. It aims to help robotics enthusiasts, university students, and ROS2 developers build a mobile robot platform with autonomous navigation capabilities from scratch. The tutorial covers everything from design philosophy, hardware assembly, and principle analysis to ROS2 software development. It includes easy-to-understand explanations, in-depth mathematical derivations, complete code implementations, and detailed line-by-line comments.

Target Audience:

- Robotics enthusiasts with some ROS/Python foundation
- University students working on course/final year projects
- University teachers conducting ROS2 courses
- Technical enthusiasts interested in the principles of mobile robot navigation

Required Background:

- Basic Python programming knowledge
- Linux command line basics
- Basic understanding of electronic circuits
- Hands-on assembly skills

Learning Outcomes:

Upon completing this tutorial, you will be able to:

1. Understand the kinematics of a two-wheel differential robot
 2. Master the application of PID speed closed-loop control in motor drives
 3. Learn to use EKF for multi-sensor fusion
 4. Independently complete the assembly, debugging, and ROS2 environment setup for the mBot
 5. Implement SLAM mapping and autonomous navigation functions
 6. Perform secondary development and innovation based on this platform
-

Part 1: Getting to Know the mBot

1.1 What is mBot?

mBot is a medium-sized mobile robot platform for intelligent robot research and education, based on a classic two-wheel differential chassis design with a 30cm diameter. Its name "mBot" stands for "mobile robot basic platform", aiming to provide an out-of-the-box experimental platform for ROS2 learning and autonomous navigation algorithm validation.



Analogy: If we compare the mBot to a self-driving car, then:

- **QCS6490 Development Board** is the car's brain, responsible for thinking and processing information
- **Two Motor Driver Boards** are the car's engine and transmission, enabling movement
- **JY61P Gyroscope** is the car's inertial navigation system, knowing its own attitude
- **LDS-E110 LiDAR** is the car's eyes, perceiving the surrounding environment
- **CH344 Serial Chip** is the car's nervous system, connecting various sensors
- **Battery** is the car's fuel tank, providing power

Core Specifications:

- **Dimensions:** 30cm diameter × 25cm height
- **Weight:** Approximately 2.5kg
- **Drive Method:** Two-wheel differential drive + universal castor wheel
- **Control Core:** Qualcomm QCS6490 (8-core processor, 8GB RAM)
- **Operating System:** Ubuntu Desktop
- **Software Framework:** ROS2 Jazzy
- **Sensors:** LiDAR, IMU, Encoder motors
- **Power Supply:** 12V Lithium battery (approx. 2-hour runtime)

Part 2: Hardware Components

2.1 Hardware Bill of Materials

Component	Qty	Model	Purpose
DC Geared Encoder Motor	2	Custom	Provides power, with encoder feedback
Integrated Motor Driver	2	Atmega328P+A4950	Motor speed closed-loop control, PID tuning
Gyroscope	1	WitMotion JY61P	Attitude measurement, provides yaw angle

Component	Qty	Model	Purpose
Single-line LiDAR	1	Lanhai LDS-E110	Environment perception, SLAM mapping
USB to 4-port Serial Chip	1	CH344	Expands serial ports to connect peripherals
Main Control Board	1	Qualcomm QCS6490	Core controller, runs ROS2
Chassis	1	Aluminum/3D Printed	Carries all components, 30cm diameter
Battery	1	12V Li-ion	Powers the robot



2.2 Hardware Configuration & Interface Allocation

The mBot robot uses a modular design. All peripherals are connected to the main controller via serial ports expanded by the CH344 chip. To ensure fixed device names after each plug/unplug, we use udev rules to map serial ports to fixed names:

Fixed Device Name	Actual Device	Connected Device	Baud Rate	Function Description
/dev/ttyMOTOR_LEFT	/dev/ttyUSB0	Left Motor Driver	115200	Send left wheel target speed / receive actual speed
/dev/ttyMOTOR_RIGHT	/dev/ttyUSB1	Right Motor Driver	115200	Send right wheel target speed / receive actual speed
/dev/ttyIMU	/dev/ttyUSB2	JY61P Gyroscope	115200	Read IMU attitude data
/dev/ttyLIDAR	/dev/ttyUSB3	LDS-E110 LiDAR	230400	Read laser point cloud data



2.3 Integrated Motor Driver Board Design

The motor driver board uses an Atmega328P microcontroller + A4950 motor driver chip, running a PID speed closed-loop control algorithm. The main controller only needs to send the target speed value via serial port; the driver board automatically performs PID adjustment to make the motor run stably at the target speed, while providing real-time feedback of the current actual speed.



2.3.1 Hardware Connection Description

According to the system design requirements, the following pin assignments were determined:

- **Motor Control:** PWM output connected to pin D5, Direction control connected to pin D6
- **Encoder Input:** Phase A connected to pin D2 (INT0), Phase B connected to pin D3 (INT1)

2.3.2 Handwritten Incremental PID Implementation

```
/******  
 * Motor Driver Firmware - Incremental PID Speed Closed-Loop Control  
 * Platform: Atmega328P  
 *  
 * Pin Assignment:  
 * - Motor PWM Control: D5  
 * - Motor Direction Control: D6  
 * - Encoder Phase A: D2 (INT0)  
 * - Encoder Phase B: D3 (INT1)  
 *  
 * Communication Interface: Serial (115200)  
 * - Receive: Target speed (RPM)  
 * - Send: Actual speed (RPM)  
 *****/  
  
// ===== Pin Definitions =====  
#define PWM_PIN 5      // Motor PWM control pin (0-255), controls speed  
#define DIR_PIN 6      // Motor direction control pin (HIGH forward, LOW reverse)  
#define ENC_A_PIN 2    // Encoder Phase A (INT0), for speed measurement and  
direction  
#define ENC_B_PIN 3    // Encoder Phase B (INT1), for speed measurement and  
direction  
  
// ===== PID Parameters =====  
float Kp = 1.5;        // Proportional gain - determines response speed, larger =  
faster but prone to overshoot  
float Ki = 0.8;        // Integral gain - eliminates steady-state error, larger =  
faster elimination but may oscillate  
float Kd = 0.2;        // Derivative gain - suppresses overshoot, larger =  
smoother but sensitive to noise  
  
// ===== PID Variables =====  
float targetRPM = 0;   // Target speed (received via serial), unit:  
revolutions per minute  
float currentRPM = 0;  // Current speed (calculated from encoder), unit: RPM  
float lastError = 0;   // Previous error e(k-1), used for incremental  
calculation  
float prevError = 0;   // Error before last e(k-2), used for incremental  
calculation  
float lastOutput = 0;  // Previous output u(k-1), used for incremental  
calculation  
  
// ===== Encoder Variables =====
```

```

volatile long encoderCount = 0; // Encoder pulse count (modified in ISR), must
use volatile
int encoderPPR = 360;           // Encoder pulses per revolution (adjust
according to actual encoder)

// ===== Timing Variables =====
unsigned long lastTime = 0;
float controlPeriod = 0.05;     // Control period 50ms (20Hz), run PID
calculation every 50ms

void setup() {
    // Initialize serial communication (115200 baud)
    Serial.begin(115200);

    // Initialize pin modes
    pinMode(PWM_PIN, OUTPUT);
    pinMode(DIR_PIN, OUTPUT);
    pinMode(ENC_A_PIN, INPUT_PULLUP); // Enable internal pull-up resistor,
prevent signal floating
    pinMode(ENC_B_PIN, INPUT_PULLUP);

    // Initialize encoder interrupts
    // Encoder Phase A connected to D2, corresponding to interrupt 0, triggered
on CHANGE
    attachInterrupt(0, encoderISR_A, CHANGE);
    // Encoder Phase B connected to D3, corresponding to interrupt 1, triggered
on CHANGE
    attachInterrupt(1, encoderISR_B, CHANGE);

    // Initial state: motor stopped
    digitalWrite(DIR_PIN, LOW);
    analogWrite(PWM_PIN, 0);

    // Record start time
    lastTime = millis();
}

void loop() {
    // ===== 1. Receive serial command (target speed) =====
    // When serial data is available, read the target speed value
    if (Serial.available() > 0) {
        String cmd = Serial.readStringUntil('\n'); // Read one line of data
        cmd.trim();                               // Remove leading/trailing
spaces
        targetRPM = cmd.toFloat();                // Convert to float
    }

    // ===== 2. Execute PID control every 50ms =====
    if (millis() - lastTime >= controlPeriod * 1000) {

        // 2.1 Read encoder, calculate actual speed
        calculateRPM();

        // 2.2 Execute incremental PID calculation
        incrementalPID();
    }
}

```

```

    // 2.3 Control motor based on PID output
    setMotorOutput();

    // 2.4 Send actual speed back via serial
    Serial.println(currentRPM);

    // Update timer
    lastTime = millis();
}

/**
 * Incremental PID calculation function
 *
 * Formula:
 *  $\Delta u = K_p[e(k)-e(k-1)] + K_i \cdot e(k) + K_d[e(k)-2e(k-1)+e(k-2)]$ 
 *  $u(k) = u(k-1) + \Delta u$ 
 *
 * Physical meaning of this formula:
 * - Proportional delta: Adjusts based on error change trend, prevents overshoot
 * - Integral term: Adjusts based on current error magnitude, eliminates steady-state error
 * - Derivative delta: Adjusts based on error acceleration (second difference), predicts future trend
 */
void incrementalPID() {
    // Calculate current error
    float error = targetRPM - currentRPM;

    // Calculate delta  $\Delta u$ 
    float pDelta = Kp * (error - lastError);           // Proportional
    delta
    float iDelta = Ki * error;                           // Integral term
    float dDelta = Kd * (error - 2 * lastError + prevError); // Derivative
    delta
    float delta = pDelta + iDelta + dDelta;              // Total delta

    // Calculate new output  $u(k) = u(k-1) + \Delta u$ 
    float output = lastOutput + delta;

    // Output clamping (-255 to 255)
    // PWM value cannot exceed 255, negative means reverse
    if (output > 255) output = 255;
    if (output < -255) output = -255;

    // Save variables for next calculation
    prevError = lastError;    //  $e(k-2) = \text{old } e(k-1)$ 
    lastError = error;        //  $e(k-1) = \text{current } e(k)$ 
    lastOutput = output;      // Save current output
}

/**
 * Set motor output
 *
 * Determines direction based on the sign of output:
 * - output >= 0: forward, PWM = output

```

```

* - output < 0: reverse, PWM = -output
*/
void setMotorOutput() {
    if (lastOutput >= 0) {
        // Forward
        digitalWrite(DIR_PIN, HIGH);
        analogWrite(PWM_PIN, (int)lastOutput);
    } else {
        // Reverse
        digitalWrite(DIR_PIN, LOW);
        analogWrite(PWM_PIN, (int)(-lastOutput));
    }
}

/**
 * Calculate actual speed (RPM)
 *
 * Formula:
 * 
$$\text{RPM} = (\text{Pulse count} \times 60) / (\text{Encoder PPR} \times \text{Time in seconds})$$

 *
 * Where:
 * - Pulse count: Number of encoder pulses captured during controlPeriod
 * - 60: Converts per second to per minute
 * - Encoder PPR: Encoder pulses per revolution
 * - Time in seconds: controlPeriod (0.05 seconds)
 */
void calculateRPM() {
    // Disable interrupts, safely read count
    noInterrupts();
    long count = encoderCount;
    encoderCount = 0; // Reset count, prepare for next cycle
    interrupts();

    // Calculate raw speed
    float rawRPM = (count * 60.0) / (encoderPPR * controlPeriod);

    // First-order low-pass filter, reduce noise
    // Formula:  $\text{filtered} = \alpha * \text{raw} + (1-\alpha) * \text{filtered\_prev}$ 
    //  $\alpha = 0.3$ , meaning 30% trust new value, 70% trust historical value
    static float filteredRPM = 0;
    filteredRPM = filteredRPM * 0.7 + rawRPM * 0.3;
    currentRPM = filteredRPM;
}

/**
 * Encoder Phase A interrupt service routine
 *
 * Quadrature encoder principle:
 * - Forward rotation: On Phase A rising edge, Phase B is low
 * - Reverse rotation: On Phase A rising edge, Phase B is high
 *
 * By comparing the levels of Phase A and Phase B, rotation direction can be
determined
 */
void encoderISR_A() {
    // At the moment Phase A changes, read the level of Phase B

```

```

    if (digitalRead(ENC_A_PIN) == digitalRead(ENC_B_PIN)) {
        encoderCount++; // Phases same level → forward
    } else {
        encoderCount--; // Phases different level → reverse
    }
}

/**
 * Encoder Phase B interrupt service routine
 *
 * The purpose of Phase B interrupt is similar to Phase A, but the direction
 * logic is opposite:
 * - Forward rotation: On Phase B rising edge, Phase A is high
 * - Reverse rotation: On Phase B rising edge, Phase A is low
 */
void encoderISR_B() {
    if (digitalRead(ENC_A_PIN) == digitalRead(ENC_B_PIN)) {
        encoderCount--; // Forward
    } else {
        encoderCount++; // Reverse
    }
}

/**
 * Serial event handler - Dynamically adjust PID parameters
 * Send format: P1.5 I0.8 D0.2
 */
void serialEvent() {
    if (Serial.available() > 0) {
        String cmd = Serial.readStringUntil('\n');
        cmd.trim();

        if (cmd.startsWith("P")) {
            Kp = cmd.substring(1).toFloat();
        } else if (cmd.startsWith("I")) {
            Ki = cmd.substring(1).toFloat();
        } else if (cmd.startsWith("D")) {
            Kd = cmd.substring(1).toFloat();
        }
    }
}

```

Part 3: Software Environment

3.0 Environment Setup & Dependency Installation

Before compiling and running the mBot code, you need to install Ubuntu 24.04, ROS2 Jazzy, and all necessary dependency packages.

3.0.1 Install Ubuntu 24.04

1. Download the Ubuntu 24.04 LTS image from [Ubuntu official website](https://ubuntu.com/download/server)
2. Create a bootable USB drive and install it on the main controller (QCS6490 or x86 industrial PC)
3. Ensure the system can connect to the internet and configure the `apt` source (recommend using a domestic mirror for faster speed)

3.0.2 Install ROS2 Jazzy

ROS2 Jazzy is the official recommended version for Ubuntu 24.04, offering the best compatibility with system libraries.

```
# 1. Set up ROS2 software source
sudo apt update && sudo apt install curl
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
/usr/share/keyrings/ros-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-
archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release &&
echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list >
/dev/null

# 2. Update software source and install ROS2 Jazzy base packages
sudo apt update
sudo apt upgrade -y

# 3. Install ROS2 Jazzy Desktop (includes core components, RViz2, examples, etc.)
sudo apt install ros-jazzy-desktop -y

# 4. Install ROS2 build tools and dependencies
sudo apt install python3-colcon-common-extensions python3-rosdep python3-vcstool
git -y

# 5. Initialize rosdep
sudo rosdep init
rosdep update
```

3.0.3 Install Navigation2 & Related Dependencies (One-Click Install)

Navigation2 is the core framework for mBot's navigation functions. The following command will install all nav2-related packages at once:

```
# One-click install of the Navigation2 suite (includes planner, controller,
costmap, recovery behaviors, etc.)
sudo apt install ros-jazzy-nav* -y
```

This command automatically installs the following core packages (partial list):

- `ros-jazzy-nav2-bringup`: Navigation launch files
- `ros-jazzy-nav2-common`: Common libraries
- `ros-jazzy-nav2-costmap-2d`: Costmap

- `ros-jazzy-nav2-core`: Core libraries
- `ros-jazzy-nav2-msgs`: Message definitions
- `ros-jazzy-nav2-rviz-plugins`: RViz visualization plugins
- `ros-jazzy-nav2-simple-commander`: Python navigation API
- `ros-jazzy-nav2-smac-planner`: SMAC planner
- `ros-jazzy-nav2-dwb-controller`: DWB local planner
- ... and all other nav2 dependencies

3.0.4 Install LiDAR Driver Dependencies

The LDS-E110 LiDAR requires serial communication and CSM library support:

```
# Install serial communication library
sudo apt install libserial-dev -y

# Install CSM library (for PLICP laser odometry)
sudo apt install libcsml-dev -y

# Install GSL math library (CSM dependency)
sudo apt install libgsl-dev -y

# Install Eigen3 linear algebra library
sudo apt install libeigen3-dev -y
```

3.0.5 Install Other Tools and Development Libraries

```
# Compilation tools and debugging tools
sudo apt install build-essential cmake gdb -y

# Python related dependencies
sudo apt install python3-pip python3-numpy python3-scipy -y

# Serial debugging tools
sudo apt install minicom cutecom -y

# Network debugging tools
sudo apt install net-tools iputils-ping -y

# TF2 coordinate transform tools
sudo apt install ros-jazzy-tf2-ros ros-jazzy-tf2-geometry-msgs ros-jazzy-tf2-tools -y

# Robot state publisher (URDF support)
sudo apt install ros-jazzy-robot-state-publisher ros-jazzy-joint-state-publisher -y

# SLAM Toolbox mapping tool
sudo apt install ros-jazzy-slam-toolbox -y

# Extended Kalman Filter (robot_localization)
sudo apt install ros-jazzy-robot-localization -y
```

```
# LiDAR driver (Blue Ocean LiDAR ROS2 driver)
# Note: LDS-E110 requires manual compilation of the official Blue Ocean driver,
see later sections

# ===== Serial Port Permission Configuration (Important!) =====
# Add current user to the dialout group to gain serial port read/write
permissions
sudo usermod -a -G dialout $USER

# Note: You need to log out and back in for this to take effect!
echo "Please log out and log back in, or reboot the system for the group
permission to take effect"
```

3.0.6 Configure Workspace Environment

```
# Create ROS2 workspace
mkdir -p ~/evobase_ws/src
cd ~/evobase_ws

# Set environment variables (add the following to ~/.bashrc)
echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc
echo "source ~/evobase_ws/install/setup.bash" >> ~/.bashrc
echo "export ROS_DOMAIN_ID=0" >> ~/.bashrc # Optional: Set DDS domain ID
echo "export ROS_AUTOMATIC_DISCOVERY_RANGE=SUBNET" >> ~/.bashrc # Optional:
Limit node discovery range

# Apply configuration
source ~/.bashrc
```

3.0.7 Verify Installation

```
# Check ROS2 environment variable
echo $ROS_DISTRO
# Should output: jazzy

# Check if colcon is available
colcon --version

# Test ROS2 communication (open two terminals)
# Terminal 1:
ros2 run demo_nodes_cpp talker
# Terminal 2:
ros2 run demo_nodes_cpp listener

# If you see messages being sent/received, ROS2 installation is successful
```

Directory Structure after Dependency Installation:

/opt/ros/jazzy/	# ROS2 Jazzy installation directory
├─ bin/	# Executable files
├─ include/	# Header files
├─ lib/	# Library files
├─ share/	# Shared files (launch, config, etc.)
└─ setup.bash	# Environment activation script

```
~/evobase_ws/           # User workspace
├─ src/                  # Source code directory
│   ├─ evobase_bringup/
│   ├─ evobase_navigation2/
│   ├─ lidar_odom/
│   ├─ bluesea-ros2/
│   └─ ...
├─ build/                # Build intermediate files
├─ install/              # Installation files
└─ log/                  # Log files
```

3.0.8 Serial Device Permission Configuration (Important!)

The mBot robot's motor driver boards, LiDAR, IMU, and other devices are connected via USB serial ports. By default, Linux systems only allow `root` users and members of the `dialout` group to access serial devices. To allow normal users to use these devices, permission configuration is necessary.

Method 1: Add user to the dialout group (Recommended)

```
# Add current user to the dialout group
sudo usermod -a -G dialout $USER

# Verify if added successfully
groups $USER
# Output should include dialout

# Important: Logout and login again or reboot for group permission to take effect
# You can run the following command to refresh group permissions (no reboot
needed)
newgrp dialout
```

Note: The `newgrp` command only affects the current terminal. It is recommended to reboot the system or log out and back in.

Method 2: Temporarily grant 777 permissions (Not recommended, only for quick testing)

```
# Temporarily grant read/write permissions to all serial devices (lost after
reboot)
sudo chmod 777 /dev/ttyACM*
sudo chmod 777 /dev/ttyUSB*
```

⚠ Warning: `chmod 777` reduces system security. It is only recommended for temporary use during debugging. Use Method 1 for production environments.

3.1 Software Requirements

- **Operating System:** Ubuntu 24.04 LTS (Desktop version recommended, minimal install also works)
- **ROS2 Version:** Jazzy Jalisco (Official ROS2 version for Ubuntu 24.04)

- **Compiler:** GCC/G++ 11.4+, CMake 3.22+
- **Python Version:** Python 3.12 (system default)
- **Hardware Requirements:**
 - Main Controller: mBot base (QCS6490 or x86 industrial PC)
 - Remote Visualization: Another Ubuntu 24.04 virtual/physical machine (for running RViz2)

⚠ Important Notes:

- The LDS-E110 LiDAR equipped on the mBot has a **limited effective ranging distance** (usually 8-12 meters, practical reliable distance is about 6-8 meters). In **large open rooms or outdoor environments, the laser point cloud becomes too sparse, leading to SLAM mapping failure or localization loss.**
- **Please perform mapping and navigation tests in relatively small indoor environments (e.g., bedroom, living room, office, with side lengths of 5-10 meters),** ensuring that walls, furniture, and other obstacles are within the LiDAR's effective detection range.

3.2 Project Architecture

```
~/evobase_ws/src/
├─ evobase_bringup/      # Robot launch files
│   └─ config/          # Configuration files directory
│   └─ launch/          # Launch files directory
│   └─ scripts/         # Python scripts
│   └─ udev/            # Udev rules
├─ evobase_navigation2/  # Navigation configuration
│   └─ config/          # EKF configuration
│   └─ launch/          # Navigation launch files
│   └─ maps/            # Saved maps
│   └─ param/           # Navigation parameters
├─ lidar_odom/          # PLICP laser odometry
│   └─ config/          # PLICP configuration
│   └─ launch/          # Launch files
│   └─ src/             # Source code
├─ bluesea-ros2/        # Blue Ocean LiDAR driver
├─ slam_toolbox/        # slam_toolbox mapping
└─ navigation2/         # Navigation2 source code
```

Part 4: Core Algorithm Explanation

4.1 Two-Wheel Differential Kinematics Principles - How the Robot Moves

Imagine pushing a shopping cart: If you push both handles with equal force, the cart goes straight forward; if you push only the left handle, the cart turns right. The motion principle of the mBot robot is similar, except it uses two independent motors to drive the left and right wheels.

The mBot uses **two-wheel differential drive**, the most classic configuration in mobile robotics. Two wheels can independently control speed and direction, achieving various movements by adjusting the speed difference between the two wheels.

Basic Motion Formulas:

Let the left wheel speed be v_l , right wheel speed be v_r , and the distance between the two wheels (wheelbase) be $L = 0.3\text{m}$. The overall motion of the robot can be decomposed into:

- **Forward Speed:**

$$v = \frac{v_l + v_r}{2}$$

— Average of left and right wheel speeds

- **Rotational Speed:**

$$\omega = \frac{v_r - v_l}{L}$$

— Speed difference divided by wheelbase

Intuitive Understanding:

- When $v_l = v_r$, left and right wheel speeds are equal, the robot moves straight forward or backward
- When $v_l = -v_r$, left and right wheel speeds are opposite, the robot rotates in place (like a tank turn)
- When $v_l > v_r$, the robot turns right (left wheel faster than right)
- When $v_l < v_r$, the robot turns left (right wheel faster than left)

Turning Radius calculation is also interesting:

$$R = \frac{v}{\omega} = \frac{L}{2} \cdot \frac{v_l + v_r}{v_r - v_l}$$

This formula tells us: the larger the speed difference between the two wheels, the smaller the turning radius; the closer the two wheel speeds, the larger the turning radius. When the two wheel speeds are exactly equal, the turning radius is infinite, meaning straight-line motion.

4.2 Dead Reckoning - How the Robot Knows Where It Is

If you walk with your eyes closed, you can only estimate how far you've walked and how much you've turned by feeling your steps. This is the basic idea of **dead reckoning**—estimating position by integrating motion.

In the mBot, the dead reckoning process is as follows:

1. Obtain the actual speeds of the left and right wheels from the encoders
2. Calculate the current linear velocity v and angular velocity ω
3. Multiply these velocities by the time interval Δt to get small displacements
4. Accumulate these small displacements to get the current position

Mathematical Expression:

Over a small time interval Δt , the robot's pose updates as:

$\Delta x = v \times \Delta t \times \cos(\theta)$	# X direction movement
$\Delta y = v \times \Delta t \times \sin(\theta)$	# Y direction movement
$\Delta \theta = \omega \times \Delta t$	# Change in orientation

Where (x, y, θ) is the robot's pose (position and orientation) in the world coordinate system. By continuously accumulating these small increments, the robot can estimate its position in real time.

```
# Dead Reckoning Implementation (with detailed comments)
def updateOdometry(left_rpm, right_rpm, dt):
    """
    Update robot pose using left and right wheel speeds
    left_rpm: Left wheel speed (revolutions per minute)
    right_rpm: Right wheel speed (RPM)
    dt: Time interval (seconds)
    """

    # Convert RPM to linear velocity (RPM to m/s)
    # Formula:  $v = (\text{RPM} * 2\pi * r) / 60$ 
    v_left = left_rpm * 2 * 3.14159 * wheel_radius / 60
    v_right = right_rpm * 2 * 3.14159 * wheel_radius / 60

    # Calculate robot velocity
    v = (v_left + v_right) / 2          # Linear velocity
    omega = (v_right - v_left) / wheel_base # Angular velocity

    # Update pose (using midpoint method for higher accuracy)
    delta_x = v * dt * cos(theta + omega * dt / 2)
    delta_y = v * dt * sin(theta + omega * dt / 2)
    delta_theta = omega * dt

    x += delta_x
    y += delta_y
    theta += delta_theta

    # Normalize angle to  $[-\pi, \pi]$ 
    theta = atan2(sin(theta), cos(theta))

    return x, y, theta, v, omega
```

Limitations of Dead Reckoning: Just like walking with your eyes closed leads to veering off course, the error in dead reckoning accumulates over time. Factors like wheel slip and uneven ground cause the estimated position to deviate from the actual position. This is why we need an IMU to assist with positioning.

4.3 PID Speed Closed-Loop Control - The Magic to Make Motors Obey

4.3.1 Why PID Control is Needed

Imagine you're driving a car and want to maintain a speed of 60km/h: If your current speed is 50km/h, you press the accelerator; if it's 70km/h, you release the accelerator or even brake. This idea of "adjusting control based on error" is the core of PID control.

The PID controller combines three components to make the motor reach the target speed quickly and accurately:

- **Proportional (P)** : Applies control based on the magnitude of the current error
 - Large error means more throttle, small error means less throttle

- But with only proportional control, there will be a steady-state error (always slightly off)
- **Integral (I)** : Accumulates past errors
 - If a small error persists for a long time, the integral term gradually increases until the error is eliminated
 - Like keeping your foot slightly on the accelerator if you notice you're consistently 2km/h below the target
- **Derivative (D)** : Predicts the future trend of the error
 - If the error is decreasing rapidly, the derivative term reduces the control amount preemptively to prevent overshoot
 - Like easing off the accelerator before reaching the target speed

PID Formula:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)dt + K_d \cdot \frac{de(t)}{dt}$$

Where $e(t)$ is the difference between target speed and actual speed.

4.3.2 Discrete PID Implementation

In digital systems, we need to discretize the continuous-time PID formula. Here is the implementation of an incremental PID:

```
# Incremental PID Implementation (with detailed comments)
class PIDController:
    def __init__(self, kp, ki, kd, dt):
        """
        Initialize PID controller
        kp: Proportional gain
        ki: Integral gain
        kd: Derivative gain
        dt: Control period (seconds)
        """
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.dt = dt
        self.last_error = 0      # Previous error e(k-1)
        self.integral = 0        # Accumulated integral value

    def compute(self, target, current):
        """
        Compute control output
        target: Target value
        current: Current value
        """
        error = target - current # Current error e(k)

        # Proportional term: current error
        P = self.kp * error

        # Integral term: accumulated error (with clamping to prevent integral
        windup)
```

```

# Integral windup means the integral term accumulates indefinitely,
causing excessive output
self.integral += error * self.dt
# Clamp [-100, 100], prevent integral from growing indefinitely
self.integral = max(min(self.integral, 100), -100)
I = self.ki * self.integral

# Derivative term: error change rate (e(k) - e(k-1)) / dt
derivative = (error - self.last_error) / self.dt
D = self.kd * derivative

# Total control output
output = P + I + D

# Update previous error
self.last_error = error
return output

```

Intuitive Understanding of the Three Gains:

- **Higher Kp:** Faster response, but prone to overshoot and oscillation (like an overreacting person)
- **Higher Ki:** Faster elimination of steady-state error, but may cause low-frequency oscillation
- **Higher Kd:** Better ability to suppress overshoot, but sensitive to noise (like being too nervous about predictions)

4.4 Multi-Sensor Fusion: Extended Kalman Filter

4.4.1 Why Fusion is Needed - Sensor "Personalities"

Wheel odometry and IMU are like two friends with different personalities:

Sensor	Advantages	Disadvantages	Analogy
Wheel Odometry	High short-term position accuracy, reliable x, y	Sudden changes during slip, heading drifts	A friend with good memory but prone to falls
IMU	Stable heading, unaffected by slip	Integral drift, cannot be used directly for position	A friend with good direction sense but bad with maps

Combining their strengths yields a more reliable pose estimate—this is **sensor fusion**.

4.4.2 Intuitive Understanding of Kalman Filter

The core idea of the Kalman filter is: **Don't fully trust any single sensor; instead, take a weighted average based on their respective reliability.**

This can be understood with a real-life example:

Suppose you want to know the current time:

- Your watch (like odometry) shows 10:00, but it was 2 minutes fast yesterday, so you trust it 70%
- The wall clock (like IMU) shows 10:05, but it occasionally stops, so you trust it 80%
- The Kalman filter combines this information to give the most likely time, say 10:03, along with an error range for this estimate.

Mathematically, the Kalman filter consists of two steps:

Step 1: Predict

Predict the current state based on the motion model (odometry data):

```
x_pred = motion_model(x_prev, control_input)
P_pred = prediction_covariance
```

Step 2: Update

Correct the prediction based on the observation (IMU data):

```
K = Kalman_gain (determines how much to trust prediction vs. observation)
x_new = x_pred + K × (observation - predicted_observation)
P_new = updated_covariance
```

The **Kalman gain K** is key: it automatically adjusts based on the error covariances of the prediction and observation. If the prediction is very accurate (small P), K is small, trusting the prediction more. If the observation is very accurate (small R), K is large, trusting the observation more.

4.4.3 EKF Application in mBot

For the mBot, the state we estimate includes:

```
x = [x, y, θ, v, ω] # Position, orientation, velocity, angular velocity
```

Prediction Step uses odometry data (motion model):

```
def predict(x, P, odom_data, dt):
    """
    Kalman filter prediction step
    x: Current state vector [x, y, theta, v, omega]
    P: State covariance matrix
    odom_data: Odometry data (contains velocity and angular velocity)
    dt: Time interval
    """
    # Get current velocity from odometry
    v = odom_data.v
    omega = odom_data.omega
    theta = x[2]

    # Predict new position (using motion model)
    x_pred = [
        x[0] + v * dt * cos(theta + omega * dt / 2), # new x
        x[1] + v * dt * sin(theta + omega * dt / 2), # new y
        x[2] + omega * dt,                          # new heading
```

```

        v,                                # velocity
        omega                             # angular velocity
    ]

    # State transition matrix F (Jacobian)
    # Used to linearize the nonlinear motion model
    F = np.array([
        [1, 0, -v*dt*sin(theta + omega*dt/2), dt*cos(theta + omega*dt/2),
        -0.5*v*dt**2*sin(theta + omega*dt/2)],
        [0, 1, v*dt*cos(theta + omega*dt/2), dt*sin(theta + omega*dt/2),
        0.5*v*dt**2*cos(theta + omega*dt/2)],
        [0, 0, 1, 0, dt],
        [0, 0, 0, 1, 0],
        [0, 0, 0, 0, 1]
    ])

    # Predict covariance
    # P_pred = F * P * F^T + Q
    # Q is the process noise covariance, representing model uncertainty
    P_pred = F @ P @ F.T + Q

    return x_pred, P_pred

```

Update Step uses IMU data (observation model):

```

def update(x_pred, P_pred, imu_data):
    """
    Kalman filter update step
    x_pred: Predicted state
    P_pred: Predicted covariance
    imu_data: IMU data (contains heading angle and angular velocity)
    """

    # IMU observations: heading angle and angular velocity
    z = np.array([imu_data.theta, imu_data.omega])

    # Observation matrix (only observes  $\theta$  and  $\omega$ )
    # H matrix represents mapping from state space to observation space
    H = np.array([
        [0, 0, 1, 0, 0],    # observes  $\theta$ 
        [0, 0, 0, 0, 1]     # observes  $\omega$ 
    ])

    # Observation noise matrix R
    # Represents the uncertainty of IMU measurements
    R = np.array([[0.01, 0], [0, 0.01]])

    # Calculate Kalman gain
    #  $S = H * P_{pred} * H^T + R$ 
    S = H @ P_pred @ H.T + R
    #  $K = P_{pred} * H^T * \text{inv}(S)$ 
    K = P_pred @ H.T @ np.linalg.inv(S)

    # State update
    y = z - H @ x_pred      # Residual (difference between observation and
                             prediction)

```

```

x_new = x_pred + K @ y      # New state
P_new = (np.eye(5) - K @ H) @ P_pred # New covariance

return x_new, P_new

```

Fusion Effect:

- When odometry has sudden changes due to wheel slip, the stable heading from the IMU constrains the state update
- When the IMU drifts, the position information from odometry corrects it
- The result is a smooth and accurate pose estimate

4.4.4 robot_localization Configuration

In actual ROS2 applications, we use the mature robot_localization package:

```

# ekf.yaml - EKF fusion configuration file
ekf_filter_node:
  ros__parameters:
    # Sensor configuration
    odom0: /wheel/odom
    odom0_config: [true, true, false,      # fuse x, y
                  false, false, true,     # fuse yaw
                  false, false, false,    # don't fuse velocity
                  false, false, false]    # don't fuse angular velocity

    imu0: /imu/data
    imu0_config: [false, false, false,    # don't fuse x, y
                  true, true, true,       # fuse roll, pitch, yaw
                  false, false, false,    # don't fuse velocity
                  true, true, true]       # fuse angular velocity

    # Output coordinate frames
    odom_frame: odom
    base_link_frame: base_link
    world_frame: odom

    # Fusion frequency 30Hz
    frequency: 30.0

```

4.5 In-depth Analysis of SLAM Mapping Algorithms - The Robot's "Blind Men and the Elephant"

4.5.1 The Fundamental Problem SLAM Solves - Map or Position First?

Imagine you are blindfolded and thrown into an unfamiliar room, armed only with a ruler to measure distances. You want to know the layout of the room (mapping) and simultaneously know where you are (localization). But—

- To know where you are, you need a map
- To draw a map, you need to know where you are

This is the famous "chicken or egg" problem in robotics, which scientists have named **SLAM** (Simultaneous Localization and Mapping).

The mBot is like this blindfolded person: it "feels" the surrounding environment through its LiDAR, senses its own motion through wheel encoders and IMU, gradually builds a map in its "mind" while exploring, and uses this map to determine its current position.

4.5.2 From "Walking with Eyes Closed" to "Seeing and Recognizing the Path" - The Evolution of SLAM

Stage 1: Dead Reckoning (Walking with Eyes Closed)

Just like walking with your eyes closed, relying only on the feeling of your steps to estimate position. This method is simple, but errors accumulate—it works for a few steps, but after a longer distance, you have no idea where you are.

Stage 2: Particle Filter (Walking with a Map)

Imagine you have a map, but you don't know your exact position. You can guess a few possible locations (particles) on the map, then take a few steps and look at the surroundings. If the scene matches what you'd expect from a guessed location, you trust that guess more; if it doesn't, you discard it. This is the **particle filter** idea, used by AMCL localization.

Stage 3: Graph Optimization (Drawing the Map While Exploring)

Now imagine you have neither a map nor your location, and you need to draw a map while exploring. What would you do?

You might:

1. First, take a few random steps, remembering the path you took (odometry constraints)
2. When you loop back to a familiar place, you realize, "Oh, I've been here before!" (loop closure detection)
3. Then you adjust the relationships between all remembered positions so that the entire path forms a closed loop (global optimization)

This is the basic idea of **Graph Optimization SLAM**, currently the most advanced method.

4.5.3 Intuitive Understanding of Graph Optimization SLAM - Explained with "Rubber Bands"

To understand graph optimization, think of a rubber band analogy:

Imagine you connect beads (representing the robot's different positions) with rubber bands (representing constraints):

- **Rubber bands between adjacent beads:** Represent odometry constraints. If you walk from point A to point B, a rubber band connects these two positions, trying to pull them to the distance and direction you actually traveled.
- **Rubber bands between non-adjacent beads:** Represent loop closure constraints. When you discover that point B and point Z are actually the same place, you add a rubber band between them, trying to pull them together.

Now, these rubber bands pull against each other. The task of graph optimization is: **Find the positions of all beads that minimizes the total stretch of all rubber bands.**

This is like solving a giant system of equations to find the most consistent robot trajectory and map.

Mathematical Form:

$$\text{Total Error} = (\text{Odometry Error}_1)^2 + (\text{Odometry Error}_2)^2 + \dots + (\text{Loop Closure Error}_1)^2$$

The goal of graph optimization is to minimize this "Total Error".

4.5.4 How Cartographer Works - The mBot's "Brain Memory System"

The cartographer algorithm supported by mBot is one of the most popular graph optimization SLAM methods. We can think of it as the robot's memory system:

Short-Term Memory (Frontend: Local Mapping)

Each step the robot takes, it stitches the current laser scan with the recent few scans to form a "small map" (called a submap). This process is fast, like our short-term memory, remembering only recent events.

Long-Term Memory (Backend: Global Optimization)

After the robot has traveled some distance, it accumulates many submaps. If it notices that the current view looks very similar to a previous submap (e.g., after looping back to the start), it realizes "I've been here before!"—this forms a **loop closure**.

After loop closure detection, the robot adjusts the positions of all submaps to make the entire large map consistent. This process is like organizing long-term memory, integrating scattered information into a coherent story.

Simple Analogy for Loop Closure Detection:

Imagine you are walking in an unfamiliar park:

- As you walk, you sketch the path you take on paper (mapping)
- After walking for a while, you suddenly notice a familiar trash can ahead—you realize you've looped back near the starting point (loop closure detected)
- You notice a gap between the start and your current position on the sketched path, so you adjust the entire path to form a closed loop (global optimization)

4.5.5 Map Formats and Applications - From Brain Memory to Physical Map

After the mBot finishes mapping, the map can be saved in two formats:

Format 1: pgm/yaml (Maps for Navigation)

This is the most common format, like a black-and-white photo:

- White areas (value 254): Free space, can walk freely
- Black areas (value 0): Obstacles, cannot walk
- Gray areas (value 205): Unknown area (not yet explored)

```
# Save map
ros2 run nav2_map_server map_saver_cli -f ~/maps/my_map
```

This generates two files:

- `my_map.pgm`: The map in black-and-white image format
- `my_map.yaml`: The map's configuration file (contains resolution, origin position, etc.)

Format 2: pbstream (Cartographer-specific map)

This is Cartographer's proprietary format, containing not only map information but also the robot's trajectory, constraint relationships, and other rich data. It can be used for continuing mapping or for pure localization mode.

```
# pbstream → pgm/yaml conversion
ros2 run cartographer_ros cartographer_pbstream_to_ros_map \
  -pbstream_filename map.pbstream -map_filestem map
```

4.6 AMCL Localization Principle - Finding Oneself on a Known Map

4.6.1 Intuitive Understanding of Particle Filter - Guess Where I Am?

Imagine this scenario: You have a map of a park, but you don't know where you are in the park. What would you do?

Step 1: Random Guesses (Initialization)

First, you randomly guess a few possible locations on the map. For example: "I might be near this pavilion, or maybe by that lake, or perhaps next to that big tree..." In mathematics, we call these guesses **particles**. Each particle contains three pieces of information: x-coordinate, y-coordinate, and facing direction (θ).

Step 2: Take a Few Steps (Motion Update)

You take a few steps forward, simultaneously updating each guessed position: "If I was near the pavilion before, I should be by the lake now; if I was by the big tree, I should be near the bench now..." This step accounts for the uncertainty in walking—after all, you don't know exactly how far you walked.

Step 3: Look Around (Observation Update)

You observe your surroundings: "Hey, there's a fountain ahead!" Then you check each guessed position: If a position should see a fountain, you give it a point (increase weight); if a position shouldn't have a fountain, you deduct points (decrease weight).

Step 4: Re-guess (Resampling)

Now, you discard the low-scoring guesses and scatter new guesses near the high-scoring ones. This is like "survival of the fittest"—believe the guesses that match observations, discard those that don't.

Then return to Step 2 and repeat. As you walk more and see more, all guesses will gradually converge on your true location.

This is the core idea of **particle filtering**. AMCL (Adaptive Monte Carlo Localization) is a mature algorithm based on this principle.

4.6.2 Mathematical Principle (Simplified)

Although explained with a real-life example, some mathematics are needed to make a robot actually implement this process:

State Representation:

Each particle is a triplet: (x, y, θ) , representing the robot's position and orientation.

Initialization:

If no initial information is available, scatter particles randomly across the entire map; if the approximate position is known (e.g., placed on a charging dock), scatter particles near that position.

Motion Update (Prediction):

$$\text{New Position} = \text{Old Position} + \text{Motion Command} + \text{Noise}$$

The "noise" here is crucial—because motion is never precise, we add a bit of random perturbation to each particle.

Observation Update (Weight Calculation):

$$\text{Weight} = \text{Likelihood of seeing the current LiDAR scan from this position}$$

How to calculate this "likelihood"?

1. Project the laser points onto the map based on the particle's position
2. Check whether each laser point should be free space or an obstacle according to the map
3. If the actual observation matches the map's expectation, the score is high

Resampling Timing:

Resampling is needed when most particles have low weights and only a few have high weights. Mathematicians invented a metric called **effective number of particles**:

$$\text{Effective Particles} = \frac{1}{\text{Sum of squares of all particle weights}}$$

When the effective number of particles falls below a threshold, perform resampling, discarding poor-quality particles.

4.6.3 Where Does AMCL's "Adaptive" Come From?

The "Adaptive" in AMCL's name has two meanings:

1. Adaptive Number of Particles

- When you are very certain of your position (all particles are concentrated), use fewer particles to reduce computation
- When you are uncertain of your position (particles are scattered), use more particles to improve robustness

2. Adaptive Resampling Strategy

Don't re-guess at every step; resample only when necessary. This avoids unnecessary randomness and makes localization more stable.

4.6.4 Starting AMCL on the mBot

```
# Start AMCL localization
ros2 launch nav2_bringup localization_launch.py map:=/path/to/map.yaml
```

After starting, you can see in RViz:

- Green arrows (particle cloud): Shows the distribution of all particles; the more concentrated, the more likely it's the robot's true position
- Red arrow: Shows the final estimated robot pose (weighted average of all particles)

4.7 In-depth Analysis of Navigation2 Navigation System - Making the Robot Walk on Its Own

Navigation2 is ROS2's navigation framework, essentially equipping the mBot with a complete "autonomous driving system". The core problem it solves is: **Given a map and a target point, how does the robot safely get there?**

4.7.1 Three Things the Navigation System Does

Comparing navigation to a human driving to an unfamiliar place, we need to do three things:

1. **Look at the map and plan the route:** Before departure, find a rough route from start to destination on the map (global path planning)
2. **Watch the road conditions and adjust in real-time:** While driving, avoid pedestrians and detour around obstacles (local path planning and obstacle avoidance)
3. **Handle unexpected situations:** What if you take a wrong turn? What if there's a traffic jam? (Recovery behaviors)

Navigation2 consists of multiple modules working together to accomplish these three tasks.

4.7.2 Costmap - The Robot's "Risk Map"

The costmap is the foundation for navigation decisions. It represents the environment as a 2D grid, where each cell has a **cost value** (0-255), indicating the "risk level" of traversing that cell.

Meaning of Cost Values:

- **0:** Completely safe, free to traverse
- **1-127:** Relatively safe, but might be near obstacles
- **128-253:** Dangerous, best to avoid
- **254:** Lethal obstacle (e.g., a wall), absolutely cannot traverse
- **255:** Unknown area (not yet explored)

The "Layer Cake" Structure of the Costmap:

The costmap is not a simple single layer, but is composed of multiple layers superimposed:

Layer 1: Static Layer

- Source: The map built by SLAM (pgm file)
- Purpose: Tells the robot where walls are and where free space is

- Analogy: Like the map app on your phone, showing buildings, roads, and other fixed information

Layer 2: Obstacle Layer

- Source: Real-time LiDAR scans
- Purpose: Marks temporarily appearing obstacles, such as pedestrians, chairs
- Analogy: Like real-time traffic conditions while driving—where there's congestion, where there's construction

Layer 3: Inflation Layer

- Source: Calculated based on the Static and Obstacle layers
- Purpose: Adds a "safety buffer" around obstacles, preventing the robot from getting too close
- Analogy: Like keeping a certain distance from strangers while walking, not brushing right up against them

Layer 4: Denoise Layer

- Source: Processes data from the Obstacle layer
- Purpose: Filters out false obstacles caused by sensor noise
- Principle: If an obstacle has no other obstacles around it, consider it noise and remove it
- Analogy: Like seeing a black dot in the distance, only to realize it's a plastic bag, not a real obstacle

Division of Labor Between Two Costmaps:

Navigation2 maintains two costmaps:

Map Type	Range	Resolution	Purpose
Global Costmap	Entire map	Lower (5-10cm/cell)	Plan the macro-level route from start to goal
Local Costmap	3-5 meters around robot	Higher (2-5cm/cell)	Real-time obstacle avoidance and micro-path adjustment

4.7.3 Global Path Planning - Finding a Main Route from A to B

The global planner's task is to find an optimal path from start to goal on the global costmap. The mBot supports multiple global planners:

1. NavFn Planner (Basic Navigation)

This is the simplest planner, treating the robot as a point and finding the shortest path on the map.

Core Formula (A* algorithm):

$$f(n) = g(n) + h(n)$$

- $g(n)$: Distance already traveled from the start to the current cell
- $h(n)$: Estimated straight-line distance from the current cell to the goal
- $f(n)$: The overall "cost-effectiveness" of this cell; smaller is better

2. Smac Planner (Advanced Navigation)

This planner is smarter; it considers the robot's motion characteristics, generating more natural and feasible paths.

4.7.4 Local Path Planning - DWB Controller

How DWB (Dynamic Window Approach) works:

1. **Sample:** Sample multiple velocity commands near the current velocity
2. **Predict:** Simulate the future trajectory for each sampled velocity
3. **Evaluate:** Evaluate each trajectory using a cost function
4. **Select:** Execute the optimal trajectory

Evaluation Metrics:

Score = Path Distance Score + Goal Distance Score + Obstacle Distance Score + Velocity Score + Steering Smoothness Score

4.7.5 Behavior Tree - Navigation's "Decision Flowchart"

The Behavior Tree is Navigation2's brain, organizing navigation logic in a tree structure, making the entire navigation process clear and understandable.

NavigateToPose Behavior Tree (making the robot reach a specified point):

```
<BehaviorTree>
  <!-- Main task: reach the goal point, allow retries up to 6 times if failed -->
  <RecoveryNode number_of_retries="6">

    <!-- Normal navigation flow -->
    <PipelineSequence>

      <!-- Plan path (re-plan every second because the environment may change) -->
    >
      <RateController hz="1.0">
        <ComputePathToPose/>
      </RateController>

      <!-- Follow the path -->
      <FollowPath/>
    </PipelineSequence>

    <!-- Recovery flow (if normal navigation fails, execute the following) -->
    <Sequence>
      <!-- Try recovery methods in turn -->
      <RoundRobin>
        <ClearCostmap/>    <!-- Clear costmap (possibly a false obstacle) -->
        <Spin/>            <!-- Rotate in place (re-observe surroundings) -->
        <Wait/>            <!-- Wait a while (let people pass) -->
        <BackUp/>          <!-- Back up a bit (change direction) -->
      </RoundRobin>
    </Sequence>
  </RecoveryNode>
```

Coordinate System	Description	Parent Coordinate	Analogy
map	Map coordinate system, globally fixed	None	World map
odom	Odometry coordinate system	map	GPS position
base_footprint	Robot's projection on the ground	odom	Your footprint
base_link	Robot's center	base_footprint	Your body
imu_link	IMU sensor	base_link	Your balance organ
laser_frame	LiDAR	base_link	Your eyes



Part 6: Detailed Explanation of Core Nodes

In this part, we will deeply analyze the core ROS2 node code of the mBot robot. Different from the hardware and algorithms discussed earlier, here we focus on the software implementation—how each node works, the purpose of each part of the code, and how they collaborate.

We will use a "section-by-section parsing" approach, breaking the complete code into functional blocks and explaining their purpose and principles block by block. This allows us to see the big picture of the code while understanding the design intent behind every detail.

6.1 Motor Driver Node (evobase_cmd_odom.py)

The motor driver node is the "actuator" of the entire system, responsible for three core tasks:

1. **Receive Commands:** Subscribe to the `/cmd_vel` topic to get target velocities
2. **Control Motors:** Send target speeds to the motor driver boards via serial port
3. **Calculate Odometry:** Read encoder feedback, compute robot pose and publish it

This node is like a car's "engine control unit," turning abstract "velocity commands" into concrete "motor rotations."

6.1.1 Node Initialization and Configuration

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
import serial
import time
import math
import os
import yaml
import tf2_ros
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from geometry_msgs.msg import TransformStamped
from ament_index_python.packages import get_package_share_directory
```

Code Explanation:

- `rclpy`: ROS2's Python client library, required for all ROS2 nodes
- `serial`: Python serial communication library, used to communicate with motor driver boards
- `yaml`: Parse YAML configuration files, read robot parameters
- `tf2_ros`: Used for publishing TF transforms (coordinate transformations)
- `Twist`: Velocity message type, containing linear and angular velocity
- `Odometry`: Odometry message type, containing position and orientation

Analogy: These imports are like a chef preparing ingredients—each library has a specific purpose, together they create a complete dish.

6.1.2 Loading Configuration File

```
def load_config():
    """Load configuration file"""
    try:
        package_share_directory = get_package_share_directory('evobase_bringup')
        config_path = os.path.join(package_share_directory, 'config',
                                    'evobase_config.yaml')
    except Exception as e:
        config_path = os.path.join(os.path.dirname(os.path.dirname(__file__)),
                                    'config', 'evobase_config.yaml')

    try:
        with open(config_path, 'r') as f:
            config = yaml.safe_load(f)
        return config, config_path
    except FileNotFoundError:
        print(f"Error: Configuration file '{config_path}' not found.")
        exit()
```

Code Explanation:

- The function first tries to find the config file in the install directory (runtime path)
- If not found, it tries the development directory (development time path)
- Use `yaml.safe_load` to safely parse the YAML file
- If the configuration file doesn't exist, print an error and exit

Configuration File Example (evobase_config.yaml):

```
serial_ports:
  left: "/dev/drive_left"
  right: "/dev/drive_right"

robot_params:
  wheel_base: 0.3           # wheelbase (meters)
  wheel_radius: 0.05        # wheel radius (meters)
  encoder_cpr: 360          # Encoder counts per revolution
  gear_ratio: 30.0          # Gear reduction ratio
  left_wheel_dir: 1          # Left wheel direction coefficient
  right_wheel_dir: 1        # Right wheel direction coefficient
```

Why design it this way? Putting parameters in a configuration file instead of hardcoding them allows:

- Easy parameter adjustment (no recompilation)
- Adaptation to different robot configurations (just modify the config file)
- Easier version control and team collaboration

6.1.3 Class Definition and Parameter Declaration

```
class CmdVelSerialCommander(Node):
    """Motor driver node class"""

    def __init__(self):
        super().__init__('cmd_vel_serial_commander')

        # Declare parameters, allowing override from launch files
        self.declare_parameter('publish_tf', True)
        self.declare_parameter('odom_topic', 'odom')
        self.declare_parameter('base_frame', 'base_footprint')

        # Get parameter values
        self.publish_tf = self.get_parameter('publish_tf').value
        self.odom_topic = self.get_parameter('odom_topic').value
        self.base_frame = self.get_parameter('base_frame').value

        # Load configuration file
        self._load_and_initialize_config()
```

Code Explanation:

- Inherits from the `Node` class, the base class for all ROS2 nodes
- `declare_parameter` declares node parameters, allowing override in launch files
- `get_parameter` retrieves the actual value of a parameter
- `_load_and_initialize_config` (private method) loads and initializes the configuration

Purpose of Parameters:

- `publish_tf`: Whether to publish TF transforms (not needed in some scenarios)
- `odom_topic`: Name of the odometry topic (remappable)
- `base_frame`: Name of the robot's base frame

Design Philosophy: Parameterized design makes the node more flexible and adaptable to different use cases.

6.1.4 Serial Port Initialization and Motor Stop

```
def _open_serial(self, port):
    """Open serial port"""
    try:
        ser = serial.Serial(port, BAUDRATE, timeout=0.5)
        self.get_logger().info(f"Successfully opened serial port: {port}")
        return ser
    except serial.SerialException as e:
        self.get_logger().error(f"Failed to open serial port {port}: {e}")
        return None

# called in __init__
self.ser_left = self._open_serial(self.SERIAL_PORT_LEFT)
self.ser_right = self._open_serial(self.SERIAL_PORT_RIGHT)
```

```

if self.ser_left is None or self.ser_right is None:
    self.get_logger().error("Failed to open one or both serial ports. Node will exit.")
    rclpy.shutdown()

# Give motor drivers some time to initialize
time.sleep(2)
self._stop_motors(self.ser_left, self.ser_right)

```

Code Explanation:

- `_open_serial` attempts to open a serial port, setting baud rate and timeout
- Uses `self.get_logger()` for logging (more professional than print, with timestamps and levels)
- If serial port opening fails, logs an error and exits
- `time.sleep(2)` waits for motor driver boards to initialize
- `_stop_motors` sends zero velocity, ensuring motors are initially stopped

Why need a timeout? `timeout=0.5` means wait at most 0.5 seconds if no data is available, preventing the program from hanging.

Why need initial stop? Prevents unexpected motor rotation when the program starts, ensuring safety.

6.1.5 Linear Velocity to Encoder Pulse Conversion

```

def _velocity_to_encoder_pulse(self, v_linear):
    """
    Convert linear velocity (m/s) to encoder pulses needed within
    MOTOR_CONTROL_DT (10ms).

    Formula derivation:
    1. Distance = velocity × time
    2. Revolutions = distance / wheel circumference
    3. Pulses = revolutions × encoder CPR × gear ratio

    Combined: pulses = v * dt * (CPR * GR / circumference)
    """
    pulses = v_linear * PULSE_CONVERSION_FACTOR
    return int(round(pulses))

```

Code Explanation:

- This is the most critical conversion function in motor control
- `PULSE_CONVERSION_FACTOR` is calculated during config loading:

```

WHEEL_CIRCUMFERENCE = 2 * math.pi * WHEEL_RADIUS
PULSE_CONVERSION_FACTOR = (ENCODER_CPR * GEAR_RATIO / WHEEL_CIRCUMFERENCE) *
MOTOR_CONTROL_DT

```

Mathematical Derivation:

Assume:

- Linear velocity $v = 0.5 \text{ m/s}$
- Control period $dt = 0.01 \text{ s}$ (10ms)
- Wheel radius $r = 0.05 \text{ m}$
- Encoder CPR = 360
- Gear ratio $GR = 30$

Calculation process:

1. Distance traveled in 10ms: $s = v \times dt = 0.5 \times 0.01 = 0.005 \text{ m}$
2. Wheel circumference: $C = 2\pi r = 2 \times 3.14 \times 0.05 = 0.314 \text{ m}$
3. Revolutions needed: $n = s / C = 0.005 / 0.314 = 0.0159 \text{ revolutions}$
4. Motor revolutions (considering gear ratio): $n_{\text{motor}} = n \times GR = 0.0159 \times 30 = 0.477 \text{ revolutions}$
5. Encoder pulses: $p = n_{\text{motor}} \times \text{CPR} = 0.477 \times 360 = 172 \text{ pulses}$

So a speed of 0.5 m/s corresponds to 172 pulses needed per 10ms.

Why rounding? Pulse count must be an integer because the motor driver board receives integers.

6.1.6 cmd_vel Callback Function

```
def cmd_vel_callback(self, msg: Twist):
    """
    Callback function for /cmd_vel messages
    """
    self.last_cmd_vel_time = self.get_clock().now()

    linear_x = msg.linear.x
    angular_z = msg.angular.z

    left_target_pulses, right_target_pulses = self._cmdvel_to_wheel_targets(
        linear_x, angular_z
    )

    self._set_target_velocity(self.ser_left, left_target_pulses)
    self._set_target_velocity(self.ser_right, right_target_pulses)

    if left_target_pulses != 0 or right_target_pulses != 0:
        self.get_logger().info(
            f"Received cmd_vel: v={linear_x:.2f} m/s, w={angular_z:.2f} rad/s ->
            f"L_target={left_target_pulses}, R_target={right_target_pulses}
            pulse/10ms"
        )
```

Code Explanation:

- Called each time a `/cmd_vel` message is received
- First updates the timestamp (used for the watchdog)
- Extracts linear velocity `linear_x` and angular velocity `angular_z`
- Calls `_cmdvel_to_wheel_targets` to calculate target pulses for left and right wheels

- Sends the values to the motor driver boards via serial port
- Logs information (only for non-zero speeds to avoid log flooding)

Design Philosophy:

- **Real-time:** Callback function should be as fast as possible, avoid complex calculations
- **Idempotence:** Even if the same command is received repeatedly, it is handled correctly
- **Debuggability:** Logging input and output facilitates debugging

6.1.7 cmd_vel to Wheel Speed Conversion

```
def _cmdvel_to_wheel_targets(self, v_robot, w_robot):
    """
    Convert robot linear and angular velocity to target pulse counts for left and
    right wheels

    Parameters:
        v_robot: Robot linear velocity (m/s)
        w_robot: Robot angular velocity (rad/s)

    Returns:
        (left_target, right_target): Left and right wheel target pulse counts
    """
    # Apply angular velocity direction coefficient to correct rotation direction
    w_robot *= self.angular_velocity_direction

    # Standard differential drive model
    v_l = v_robot - w_robot * WHEEL_BASE / 2.0
    v_r = v_robot + w_robot * WHEEL_BASE / 2.0

    # Convert linear velocities to encoder pulses
    left_target = self._velocity_to_encoder_pulse(v_l)
    right_target = self._velocity_to_encoder_pulse(v_r)

    # Clamp (prevent motor overload)
    left_target = max(min(left_target, 60), -60)
    right_target = max(min(right_target, 60), -60)

    # Multiply by direction coefficients to adapt to assembly/wiring/encoder
    direction
    left_target *= LEFT_WHEEL_DIR
    right_target *= RIGHT_WHEEL_DIR

    return left_target, right_target
```

Code Explanation:

1. **Apply direction coefficient:** `angular_velocity_direction` corrects rotation direction. If the robot turns clockwise instead of counterclockwise, set this coefficient to -1 to flip it.
2. **Differential model:** According to the kinematics formula from Section 4.1:

$$v_l = v - \frac{\omega L}{2}, \quad v_r = v + \frac{\omega L}{2}$$
3. **Convert to pulses:** Calls `_velocity_to_encoder_pulse` introduced earlier

4. **Clamping:** Limits pulses to the range [-60, 60] to prevent motor overload. 60 pulses/10ms corresponds to approximately 1.7 m/s (based on previous calculation).
5. **Direction coefficients:** `LEFT_WHEEL_DIR` and `RIGHT_WHEEL_DIR` correct for motor installation direction. If the left wheel is installed backwards, set this coefficient to -1.

Why are direction coefficients needed? Because:

- Motor wiring might be reversed
- Encoder A/B phases might be reversed
- The definition of left/right wheels might not match reality

Using software coefficients avoids rewiring.

6.1.8 Watchdog Mechanism

```
def watchdog_callback(self):
    """
    Watchdog timer callback function
    Automatically stops motors if no cmd_vel message is received for a certain
    period
    """
    time_since_last_msg = (self.get_clock().now() -
self.last_cmd_vel_time).nanoseconds / 1e9

    if time_since_last_msg > self.cmd_vel_timeout_duration:
        self.get_logger().warn(
            f"No /cmd_vel message received for {self.cmd_vel_timeout_duration}
seconds, stopping motors!"
        )
        self._stop_motors(self.ser_left, self.ser_right)
        self.last_cmd_vel_time = self.get_clock().now()
```

Code Explanation:

- This is a timer callback function executed every 0.1 seconds
- Calculates the time elapsed since the last `cmd_vel` message
- If no new command is received for more than 0.5 seconds, automatically stops motors
- Updates the timer to avoid repeated stopping

Why is a watchdog needed?

- **Safety:** If the control program crashes, the robot won't keep moving
- **Fault tolerance:** During network latency or packet loss, the robot automatically stops
- **Energy saving:** Automatically stops motors when no movement is required

Analogy: Like a car's autonomous driving system—if the steering wheel has no input for a long time, the system assumes the driver may have lost consciousness and applies the brakes.

6.1.9 Odometry Initialization

```
def _init_odom(self):
    """Initialize odometry related variables and publishers"""
    self.x = 0.0
    self.y = 0.0
    self.theta = 0.0
    self.left_total_ticks = 0.0
    self.right_total_ticks = 0.0
    self.last_left_total_ticks = 0.0
    self.last_right_total_ticks = 0.0
    self.odom_pub = self.create_publisher(Odometry, self.odom_topic, 10)
```

Code Explanation:

- `x, y, theta`: Robot's position and orientation in the odom coordinate frame
- `left/right_total_ticks`: Cumulative pulse counts for left/right wheels (from startup)
- `last_left/right_total_ticks`: Cumulative pulse counts from the last calculation (used to compute increments)
- `odom_pub`: Creates an odometry publisher with QoS=10

Design Philosophy:

- Uses cumulative values instead of incremental values to avoid error accumulation
- Computes increments by subtracting the last cumulative value from the current one, unaffected by restarts
- QoS=10 means buffer up to 10 messages, discarding the oldest if exceeded

6.1.10 Reading Encoder Data

```
def read_and_print_encoders(self):
    """
    Timer callback function: reads, parses encoder feedback, and publishes
    odometry messages
    """
    delta_l_ticks = 0.0
    delta_r_ticks = 0.0

    # --- Read left wheel data ---
    if self.ser_left and self.ser_left.is_open:
        while self.ser_left.in_waiting > 0:
            try:
                line_bytes = self.ser_left.readline()
                line = line_bytes.decode('utf-8',
errors='ignore').replace('\x00', '').strip()
                if not line: continue
                parts = line.split()
                if len(parts) >= 2:
                    target = float(parts[0])
                    actual = float(parts[1])
                    delta_l_ticks += actual * LEFT_WHEEL_DIR
            except Exception as e:
```

```

        self.get_logger().error(f"Error reading from left serial port:
{e}")

        break

    # --- Read right wheel data (similar to left) ---
    # ... similar code omitted ...

    # Update total pulse counts
    self.left_total_ticks += delta_l_ticks
    self.right_total_ticks += delta_r_ticks

    # Calculate increments for odometry
    delta_for_odom_l = self.left_total_ticks - self.last_left_total_ticks
    delta_for_odom_r = self.right_total_ticks - self.last_right_total_ticks

    self.last_left_total_ticks = self.left_total_ticks
    self.last_right_total_ticks = self.right_total_ticks

    # Publish odometry
    self._update_and_publish_odom(delta_for_odom_l, delta_for_odom_r)

```

Code Explanation:

1. **Read all pending data:** Uses a `while ser.in_waiting > 0` loop to read all data in the buffer
2. **Decode and clean:** Decodes byte data to UTF-8 string, removes null characters and leading/trailing spaces
3. **Parse data:** Motor driver boards send data in the format "target actual", separated by a space
4. **Accumulate increments:** Adds the actual pulse count multiplied by the direction coefficient
5. **Update total counts:** Adds the current increment to the total count
6. **Calculate increments for odometry:** Subtracts the previously recorded total count from the new total count
7. **Update last recorded values:** Prepares for the next calculation
8. **Publish odometry:** Calls `_update_and_publish_odom`

Why use a `while` loop to read all data?

- Serial data is streaming; multiple lines of data may be received at once
- If not all read, the next timer tick might read old data
- Ensures the most recent data is processed

Why accumulate `actual` instead of just using the last one?

- If multiple data points are received within 10ms, all pulses need to be accumulated
- For example, when the motor rotates at high speed, multiple feedback messages might be received within 10ms

6.1.11 Odometry Update and Publishing

```
def _update_and_publish_odom(self, delta_left_ticks, delta_right_ticks):
    """
    Compute and publish odometry from pulse increments
    """

    # Convert pulse increments to distance increments
    dist_left = delta_left_ticks / (ENCODER_CPR * GEAR_RATIO) *
WHEEL_CIRCUMFERENCE
    dist_right = delta_right_ticks / (ENCODER_CPR * GEAR_RATIO) *
WHEEL_CIRCUMFERENCE

    # Kinematics calculation
    delta_s = (dist_left + dist_right) / 2.0                # Average distance
traveled
    delta_theta = (dist_right - dist_left) / WHEEL_BASE     # Change in angle
    delta_theta *= self.angular_velocity_direction          # Apply direction
coefficient

    # Update global pose (using midpoint method for higher accuracy)
    self.x += delta_s * math.cos(self.theta + delta_theta / 2.0)
    self.y += delta_s * math.sin(self.theta + delta_theta / 2.0)
    self.theta += delta_theta

    # Normalize angle to  $[-\pi, \pi]$ 
    self.theta = math.atan2(math.sin(self.theta), math.cos(self.theta))

    # Create and publish odometry message
    now = self.get_clock().now().to_msg()
    odom_msg = Odometry()
    odom_msg.header.stamp = now
    odom_msg.header.frame_id = 'odom'
    odom_msg.child_frame_id = self.base_frame

    # Set position
    odom_msg.pose.pose.position.x = self.x
    odom_msg.pose.pose.position.y = self.y
    odom_msg.pose.pose.position.z = 0.0

    # Set orientation (Euler to quaternion)
    q = self.euler_to_quaternion(0, 0, self.theta)
    odom_msg.pose.pose.orientation.x = q[0]
    odom_msg.pose.pose.orientation.y = q[1]
    odom_msg.pose.pose.orientation.z = q[2]
    odom_msg.pose.pose.orientation.w = q[3]

    # Estimate velocity
    dt = 0.1 # Same as timer frequency
    vx = delta_s / dt
    vth = delta_theta / dt
    odom_msg.twist.twist.linear.x = vx
    odom_msg.twist.twist.angular.z = vth

    self.odom_pub.publish(odom_msg)
```

Code Explanation:

Step 1: Convert pulses to distance

```
dist_left = delta_left_ticks / (ENCODER_CPR * GEAR_RATIO) * WHEEL_CIRCUMFERENCE
```

- `delta_left_ticks`: Left wheel pulse increment
- Dividing by `ENCODER_CPR * GEAR_RATIO` gives motor revolutions
- Multiplying by `WHEEL_CIRCUMFERENCE` gives actual distance traveled

Step 2: Kinematics calculation

```
delta_s = (dist_left + dist_right) / 2.0 # Average distance traveled  
delta_theta = (dist_right - dist_left) / WHEEL_BASE # Change in angle
```

This is exactly the kinematics formula from Section 4.1:

- Average distance traveled = (left distance + right distance)/2
- Change in angle = (right distance - left distance)/wheelbase

Step 3: Midpoint method pose update

```
self.x += delta_s * math.cos(self.theta + delta_theta / 2.0)  
self.y += delta_s * math.sin(self.theta + delta_theta / 2.0)
```

Why use the midpoint method instead of the current angle directly?

- Using the current angle `theta` assumes rotation happens after translation, which introduces error
- Using the new angle `theta + delta_theta` assumes rotation happens before translation, also introducing error
- The midpoint method assumes rotation is evenly distributed during translation, providing higher accuracy

Analogy: Like walking while turning—the actual path is an arc. The midpoint method more accurately describes this motion.

Step 4: Angle normalization

```
self.theta = math.atan2(math.sin(self.theta), math.cos(self.theta))
```

- Ensures the angle is always within the range $[-\pi, \pi]$
- Prevents numerical issues from unbounded angle growth

Step 5: Euler to quaternion conversion

```
def euler_to_quaternion(self, roll, pitch, yaw):
    qx = math.sin(roll/2) * math.cos(pitch/2) * math.cos(yaw/2) -
    math.cos(roll/2) * math.sin(pitch/2) * math.sin(yaw/2)
    qy = math.cos(roll/2) * math.sin(pitch/2) * math.cos(yaw/2) +
    math.sin(roll/2) * math.cos(pitch/2) * math.sin(yaw/2)
    qz = math.cos(roll/2) * math.cos(pitch/2) * math.sin(yaw/2) -
    math.sin(roll/2) * math.sin(pitch/2) * math.cos(yaw/2)
    qw = math.cos(roll/2) * math.cos(pitch/2) * math.cos(yaw/2) +
    math.sin(roll/2) * math.sin(pitch/2) * math.sin(yaw/2)
    return [qx, qy, qz, qw]
```

ROS2 uses quaternions to represent orientation because they:

- Avoid gimbal lock issues associated with Euler angles
- Make interpolation easier
- Are more compact to store and transmit

6.1.12 Publishing TF Transforms

```
# In _update_and_publish_odom, after publishing the odometry message:
if self.publish_tf and self.tf_broadcaster:
    t = TransformStamped()
    t.header.stamp = now
    t.header.frame_id = 'odom'
    t.child_frame_id = self.base_frame
    t.transform.translation.x = self.x
    t.transform.translation.y = self.y
    t.transform.translation.z = 0.0
    t.transform.rotation.x = q[0]
    t.transform.rotation.y = q[1]
    t.transform.rotation.z = q[2]
    t.transform.rotation.w = q[3]
    self.tf_broadcaster.sendTransform(t)
```

Code Explanation:

- Creates a `TransformStamped` message containing timestamp, parent frame, and child frame
- Sets translation (x, y, z) and rotation (quaternion)
- Sends the transform to the TF tree via `sendTransform`

Why are TF transforms needed?

- Allows all nodes to know the robot's current position
 - Enables LiDAR point clouds to be transformed to the odom frame
 - Allows the navigation system to plan paths relative to the map frame
-

6.2 Debugging Scripts Explained

Besides the core nodes, the project includes several practical debugging scripts. While not strictly necessary, they are very useful during development and debugging.

6.2.1 Temperature Monitoring Script (check_temp.py)

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

def get_temperatures():
    """Read temperature sensor data for AidLux system"""
    thermal_zones = [d for d in os.listdir('/sys/class/thermal/') if
d.startswith('thermal_zone')]
    temperatures = []

    for zone in thermal_zones:
        try:
            with open(f'/sys/class/thermal/{zone}/temp', 'r') as f:
                temp = int(f.read().strip()) / 1000.0
            with open(f'/sys/class/thermal/{zone}/type', 'r') as f:
                type_name = f.read().strip()
            temperatures.append((zone, type_name, temp))
        except:
            pass

    return temperatures
```

Code Explanation:

- Linux systems (including AidLux) expose temperature sensor information via `/sys/class/thermal/`
- Directories like `thermal_zone0`, `thermal_zone1` correspond to different sensors
- The `temp` file stores the temperature value (usually in millidegrees Celsius, divide by 1000 for Celsius)
- The `type` file stores the sensor type (e.g., "cpu-thermal", "gpu-thermal")

```
if __name__ == "__main__":
    temps = get_temperatures()
    print("Android device temperature sensors:")

    categories = {
        'CPU': [],
        'GPU': [],
        'Memory': [],
        'Other': []
    }

    for zone, type_name, temp in temps:
        if temp > 0:
            if 'cpu' in type_name.lower():
                categories['CPU'].append(temp)
            elif 'gpu' in type_name.lower():
```

```

        categories['GPU'].append(temp)
    elif 'ddr' in type_name.lower():
        categories['Memory'].append(temp)
    else:
        categories['Other'].append(temp)
    print(f"{zone}: {type_name} - {temp:.1f}°C")

print("\nCategory average temperatures:")
for cat, temps_list in categories.items():
    if temps_list:
        avg = sum(temps_list) / len(temps_list)
        print(f"{cat}: {avg:.1f}°C ({len(temps_list)} sensors)")

```

Why is temperature monitoring needed?

- During high-load operations (like SLAM mapping), the CPU/GPU may overheat
- Overheating can cause throttling, impacting performance
- Extreme cases can damage hardware

Usage Example:

```
python3 src/evobase_bringup/scripts/check_temp.py
```

Example output:

```

--- Discovered temperature sensors ---
thermal_zone0: cpu-thermal - 45.2°C
thermal_zone1: gpu-thermal - 42.8°C
thermal_zone2: ddr-thermal - 38.5°C

```

6.2.2 Dynamic Temperature Monitoring (dynamic_check_temp.py)

```

#!/usr/bin/env python3
import os
import time

def display_average_temperatures(temps):
    """Calculate and print category average temperatures"""
    categories = {
        'CPU': [],
        'GPU': [],
        'Memory (DDR)': [],
        'Other': []
    }

    for _, type_name, temp in temps:
        if temp > 0:
            if 'cpu' in type_name.lower():
                categories['CPU'].append(temp)
            elif 'gpu' in type_name.lower():
                categories['GPU'].append(temp)
            elif 'ddr' in type_name.lower():
                categories['Memory (DDR)'].append(temp)
            else:

```

```

        categories['Other'].append(temp)

    print("=" * 40)
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Device category average temperatures:")
    print("-" * 40)

    for cat, temps_list in categories.items():
        if temps_list:
            avg = sum(temps_list) / len(temps_list)
            print(f" - {cat}: {avg:.1f}°C ({len(temps_list)} sensors)")
        else:
            print(f" - {cat}: No valid data")
    print("=" * 40)

if __name__ == "__main__":
    UPDATE_INTERVAL = 5 # Update every 5 seconds

    try:
        while True:
            temps = get_temperatures()
            display_average_temperatures(temps)
            time.sleep(UPDATE_INTERVAL)
    except KeyboardInterrupt:
        print("\nProgram terminated.")

```

Code Explanation:

- Extends `check_temp.py` by adding a loop
- Updates and displays temperatures every 5 seconds
- Catches `Ctrl+C` to exit gracefully

Use Cases:

- Monitor temperature during long SLAM mapping sessions
- Test cooling effectiveness
- Identify hot spots

6.2.3 Serial Device Finder (`find_serial.py`)

```

#!/usr/bin/env python3
import serial.tools.list_ports

def find_cp2102_by_vid_pid(target_vid=0x10C4, target_pid=0xEA60):
    """Find CP2102 devices with specified VID and PID"""
    found_devices = []
    ports = serial.tools.list_ports.comports()

    print("--- List of discovered serial devices ---")
    for port in ports:
        print(f" Device Path: {port.device}")
        print(f" Description: {port.description}")
        print(f" Hardware ID: {port.hwid}")
        print(f" Vendor ID (VID): {hex(port.vid) if port.vid else 'N/A'}")

```

```

        print(f"    Product ID (PID): {hex(port.pid) if port.pid else 'N/A'}")
        print(f"    Serial Number: {port.serial_number if port.serial_number else
'N/A'}")
        print("-" * 20)

        if port.vid == target_vid and port.pid == target_pid:
            found_devices.append(port.device)

    return found_devices

```

Code Explanation:

- Uses `serial.tools.list_ports.comports()` to list all serial devices
- Each device contains: device path, description, hardware ID, VID, PID, serial number
- Filters target device by VID/PID (CP2102 default VID=0x10C4, PID=0xEA60)

What are VID and PID?

- VID (Vendor ID): Assigned by the USB organization
- PID (Product ID): Assigned by the vendor
- Each USB device has a unique VID/PID combination

Why is this script needed?

- Confirm devices are recognized by the system
- View device VID/PID for writing udev rules
- Debug serial connection issues

Usage Example:

```
python3 src/evobase_bringup/scripts/find_serial.py
```

Example output:

```

--- List of discovered serial devices ---
Device Path: /dev/ttyUSB0
Description: CP2102 USB to UART Bridge Controller
Hardware ID: USB VID:PID=10C4:EA60 SER=0001
Vendor ID (VID): 0x10c4
Product ID (PID): 0xea60
Serial Number: 0001
-----

```

6.2.4 Encoder Data Reader (read_encoders_once.py)

```

#!/usr/bin/env python3
import serial
import time

SERIAL_PORT_LEFT = '/dev/drive_left'
SERIAL_PORT_RIGHT = '/dev/drive_right'
BAUDRATE = 115200

```

```

def main():
    ser_left = serial.Serial(SERIAL_PORT_LEFT, BAUDRATE, timeout=0.5)
    ser_right = serial.Serial(SERIAL_PORT_RIGHT, BAUDRATE, timeout=0.5)

    try:
        while True:
            # Read left wheel data
            if ser_left.in_waiting > 0:
                line = ser_left.readline().decode('utf-8',
errors='ignore').strip()
                if line:
                    print(f"[Left] {line}")

            # Read right wheel data
            if ser_right.in_waiting > 0:
                line = ser_right.readline().decode('utf-8',
errors='ignore').strip()
                if line:
                    print(f"[Right] {line}")

            time.sleep(0.05)
    except KeyboardInterrupt:
        print("\nExiting.")
    finally:
        ser_left.close()
        ser_right.close()

```

Code Explanation:

- Directly opens the serial ports for left and right motors (using fixed device names)
- Loops reading and printing raw data
- Reads every 50ms to avoid high CPU usage
- Closes serial ports on `Ctrl+C` exit

Example Output:

```

[Left] 0 0
[Right] 0 0
[Left] 100 98
[Right] 100 101
[Left] 200 198
[Right] 200 202

```

Each line format: "Target Pulses Actual Pulses"

Purpose:

- Verify serial communication is working
- Observe PID control effectiveness (is actual close to target?)
- Check encoder direction correctness

6.2.5 Endurance Test (send_cmd_time_test.py)

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import time

class CmdVelTestNode(Node):
    def __init__(self):
        super().__init__('cmd_vel_test_node')

        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.timer = self.create_timer(0.1, self.publish_cmd_vel)
        self.start_time = time.time()

        self.twist_msg = Twist()
        self.twist_msg.linear.x = 0.5 # Linear velocity 0.5 m/s

        self.get_logger().info('Starting endurance test, linear velocity: 0.5 m/s')
        self.msg_count = 0

    def publish_cmd_vel(self):
        self.cmd_vel_pub.publish(self.twist_msg)
        self.msg_count += 1

        if self.msg_count % 100 == 0:
            elapsed_time = time.time() - self.start_time
            minutes = int(elapsed_time // 60)
            seconds = int(elapsed_time % 60)
            self.get_logger().info(f'Running: {minutes}m {seconds}s, Messages sent: {self.msg_count}')

    def destroy_node(self):
        stop_msg = Twist()
        self.cmd_vel_pub.publish(stop_msg)
        elapsed_time = time.time() - self.start_time
        minutes = int(elapsed_time // 60)
        seconds = int(elapsed_time % 60)
        self.get_logger().info(f'Endurance test finished! Total runtime: {minutes}m {seconds}s')
        super().destroy_node()
```

Code Explanation:

- Creates a ROS2 node that publishes `/cmd_vel` messages at 10Hz
- Sets linear velocity to 0.5 m/s, angular velocity to 0 (straight line motion)
- Prints runtime every 10 seconds (100 messages)
- Sends a stop message and prints total runtime when the node is destroyed

Purpose:

- Test battery endurance

- Long-duration stability testing
- Motor and driver board reliability testing

Why continuous sending instead of just once?

- The watchdog mechanism will stop motors, so continuous sending is necessary
- Simulates real control scenarios (remote controller continuously sending commands)

6.3 PLICP LiDAR Odometry Node

PLICP (Point-to-Line Iterative Closest Point) is an efficient LiDAR odometry algorithm. Unlike traditional ICP (point-to-point matching), PLICP uses point-to-line distance, converging faster and achieving higher accuracy, especially suitable for structured environments (like indoor corridors, rooms, etc.).

This node is responsible for three core tasks:

1. **Receive LiDAR Data:** Subscribes to the `/scan` topic to get LiDAR point clouds
2. **Fuse Wheel Odometry:** Subscribes to the `/odom_wheel` topic to get velocity information from wheel odometry
3. **Compute Pose:** Estimates robot motion by matching consecutive LiDAR scans using the PLICP algorithm and publishes the fused odometry

Analogy: If the motor driver node is the robot's "muscles" and "proprioception," the PLICP node is the robot's "vision" and "spatial perception." It corrects errors estimated by muscle movement by observing changes in the surrounding environment.

6.3.1 Header File Definition (plicp_odometry.hpp)

```
/*
 * PLICP LiDAR Odometry - ROS2 Version (with wheel Odometry Fusion)
 *
 * This header file defines the interface of the PlicpOdometry class, including:
 * - Member variables (state, parameters, TF, etc.)
 * - Public methods (constructor, destructor)
 * - Private methods (callbacks, core algorithms, helper functions)
 */

#ifndef PLICP_ODOMETRY_ROS2__PLICP_ODOMETRY_HPP_
#define PLICP_ODOMETRY_ROS2__PLICP_ODOMETRY_HPP_

#include <chrono>
#include <memory>
#include <string>
#include <vector>

// ROS2 Core
#include <rclcpp/rclcpp.hpp>
#include <rclcpp_components/register_node_macro.hpp>

// Message Types
#include <sensor_msgs/msg/laser_scan.hpp>
#include <geometry_msgs/msg/transform_stamped.hpp>
#include <nav_msgs/msg/odometry.hpp>
```

```
// TF2 Coordinate Transform Library
#include <tf2/utils.h>
#include <tf2/LinearMath/Transform.h>
#include <tf2_ros/buffer.h>
#include <tf2_ros/transform_listener.h>
#include <tf2_ros/transform_broadcaster.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>

// CSM Library (Canonical Scan Matcher) - Implementation of PLICP algorithm
#include <csm/csm_all.h>
```

Code Explanation:

Header File Inclusion Explanation:

- `rc1cpp/rc1cpp.hpp`: ROS2's C++ client library, required for all ROS2 nodes
- `sensor_msgs/msg/laser_scan.hpp`: LiDAR message type
- `nav_msgs/msg/odometry.hpp`: Odometry message type
- `tf2_*` series: Coordinate transform related, used for publishing and listening to TF
- `csm/csm_all.h`: CSM library, implements the PLICP algorithm (core!)

Why so many header files?

- ROS2 nodes require various functionalities: communication, coordinate transforms, algorithm libraries, etc.
- Each function corresponds to one or more header files
- This is a characteristic of C++ projects—explicitly include all required dependencies

```
namespace plicp_odometry_ros2
{

class PlicpOdometry : public rc1cpp::Node
{
public:
    // Constructor - allows component-style loading
    explicit PlicpOdometry(const rc1cpp::NodeOptions & options =
rc1cpp::NodeOptions());

    // Destructor - cleanup resources
    virtual ~PlicpOdometry();

private:
    // ===== ROS2 Communication Components =====

    // LiDAR data subscriber
    rc1cpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr
laser_scan_subscription_;

    // wheel odometry subscriber (for fusion)
    rc1cpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr wheel_odom_sub_;

    // Fused odometry publisher
```

```
rc1cpp::Publisher<nav_msgs::msg::Odometry>::SharedPtr odom_publisher_;
```

Code Explanation:

- The class inherits from `rc1cpp::Node`, making it a standard ROS2 node
- The constructor has a default parameter `options`, supporting component-style loading (more flexible)
- Declares three ROS2 communication members: two subscribers and one publisher

What is component-style loading?

- Traditional way: each node is an independent executable
- Component way: multiple nodes can run in the same process, with higher communication efficiency
- The `options` parameter in the constructor is designed to support this mode

```
// ===== Time Management =====
rc1cpp::Time last_icp_time_;      // Time of last ICP execution
rc1cpp::Time current_time_;      // Current time

// ===== Motion State Estimation =====
geometry_msgs::msg::Twist latest_velocity_;      // Most recently estimated
velocity
geometry_msgs::msg::Twist current_wheel_twist_;  // Current wheel odometry
velocity
bool has_wheel_data_;      // whether wheel odometry data has been received

// ===== TF2 Coordinate Transform System =====
std::shared_ptr<tf2_ros::Buffer> tf_buffer_;      // TF buffer
std::shared_ptr<tf2_ros::TransformListener> tf_listener_; // TF listener
std::shared_ptr<tf2_ros::TransformBroadcaster> tf_broadcaster_; // TF
broadcaster

// ===== Fixed Transforms Between Frames =====
tf2::Transform base_to_laser_;    // Transform from base_link to laser_frame
tf2::Transform laser_to_base_;    // Transform from laser_frame to base_link
(inverse)
```

Code Explanation:

- **Time management:** Records the time of last ICP execution to compute time differences
- **Motion state:** Stores velocity information for prediction and publishing
- **TF system:** Buffer, listener, broadcaster, a complete TF processing component
- **Fixed transforms:** The transform between `base_link` and `laser_frame`, defined by URDF, is fixed

Why need both `base_to_laser_` and `laser_to_base_`?

- LiDAR data is in the `laser_frame` coordinate frame
- Robot pose is in the `base_link` coordinate frame
- PLICP matching needs to be performed in a common coordinate frame, so transforms are needed

```

// ===== Pose State Variables =====
tf2::Transform base_in_odom_;           // Pose of base_link in odom frame
tf2::Transform base_in_odom_keyframe_;  // Pose at keyframe

// ===== System State Parameters =====
bool initialized_; // whether initialized

// ===== Configuration Parameters =====
std::string odom_frame_; // Odometry frame name
std::string base_frame_; // Base frame name

// ===== Keyframe Detection Parameters =====
double kf_dist_linear_; // Linear motion threshold (meters)
double kf_dist_linear_sq_; // Square of linear motion threshold (avoids
repeated sqrt)
double kf_dist_angular_; // Angular motion threshold (radians)
int kf_scan_count_; // Force keyframe update interval
int scan_count_; // Current scan count

```

Code Explanation:

What is a keyframe?

- A keyframe is a selected laser scan frame used as a reference
- Not every frame is used as a reference, otherwise computation is too high
- Only update the keyframe when the robot has moved far enough or turned enough

Why are keyframes needed?

- Computational efficiency: matching every frame is too computationally expensive
- Accuracy: when two frames are too close, matching is poor (little information)
- Robustness: matching between keyframes makes loop closure detection easier

Threshold parameters:

- `kf_dist_linear_`: Move more than 0.1 meters, update keyframe
- `kf_dist_angular_`: Rotate more than 5 degrees (approx. 0.087 rad), update keyframe
- `kf_scan_count_`: Force update at most every 10 frames, preventing never updating when stationary

```

// ===== CSM Library Interface Data Structures =====
sm_params input_; // CSM input parameters
sm_result output_; // CSM output results
LDP prev_ldp_scan_; // Previous LiDAR scan frame (CSM format)

// ===== Private Method Declarations =====

// Initialize parameters
void initParams();

// Create cache (pre-compute cos/sin values)
void createCache(const sensor_msgs::msg::LaserScan::SharedPtr scan_msg);

```

```

// Get transform from base_link to laser_frame
bool getBaseToLaserTf(const std::string & frame_id);

// Convert ROS2 LaserScan to CSM's LDP format
void laserScanToLDP(const sensor_msgs::msg::LaserScan::SharedPtr scan_msg,
LDP & ldp);

// Core PLICP matching function
void scanMatchWithPLICP(LDP & curr_ldp_scan, const rclcpp::Time & time);

// Get predicted motion (using wheel odometry)
void getPrediction(double & prediction_change_x, double &
prediction_change_y,
double & prediction_change_angle, double dt);

// Create TF transform from translation and angle
void createTfFromXYTheta(double x, double y, double theta, tf2::Transform &
t);

// Publish TF transform and odometry
void publishTFAndOdometry();

// Determine if a new keyframe is needed
bool newKeyframeNeeded(const tf2::Transform & d);

// Callback functions
void scanCallback(const sensor_msgs::msg::LaserScan::SharedPtr scan_msg);
void wheelOdomCallback(const nav_msgs::msg::Odometry::SharedPtr msg);
};

} // namespace plicp_odometry_ros2

#endif // PLICP_ODOMETRY_ROS2__PLICP_ODOMETRY_HPP_

```

Header File Summary:

- Class structure is clear: member variables at the top, member methods at the bottom
- All core functions have corresponding methods
- Uses a namespace to avoid naming conflicts

6.3.2 Constructor and Initialization (plicp_odometry.cpp)

```

/*
 * PLICP LiDAR Odometry - Core Implementation File
 */

#include "plicp_odometry_ros2/plicp_odometry.hpp"

namespace plicp_odometry_ros2
{

// Constructor
PlipcOdometry::PlipcOdometry(const rclcpp::NodeOptions & options)
: Node("plicp_odometry", options)
{

```

```

RCLCPP_INFO(this->get_logger(), "\033[1;32m----> PLICP odometry (wheel
Fusion) started.\033[0m");

// Initialize TF buffer
tf_buffer_ = std::make_shared<tf2_ros::Buffer>(this->get_clock());
tf_listener_ = std::make_shared<tf2_ros::TransformListener>(*tf_buffer_);
tf_broadcaster_ = std::make_shared<tf2_ros::TransformBroadcaster>(*this);

// Subscribe to LiDAR data
laser_scan_subscription_ = this-
>create_subscription<sensor_msgs::msg::LaserScan>(
    "scan",
    rclcpp::SensorDataQoS(), // Use sensor-specific QoS settings
    std::bind(&PlipcOdometry::scanCallback, this, std::placeholders::_1));

// Subscribe to wheel odometry data
wheel_odom_sub_ = this->create_subscription<nav_msgs::msg::Odometry>(
    "/odom_wheel",
    rclcpp::QoS(10),
    std::bind(&PlipcOdometry::wheelOdomCallback, this,
std::placeholders::_1));

// Publish fused odometry
odom_publisher_ = this->create_publisher<nav_msgs::msg::Odometry>("odom",
50);

// Initialize parameters
initParams();

// Initialize state variables
scan_count_ = 0;
initialized_ = false;
has_wheel_data_ = false;

// Initialize pose (identity matrix)
base_in_odom_.setIdentity();
base_in_odom_keyframe_.setIdentity();

// Initialize CSM input parameters
input_.laser[0] = 0.0;
input_.laser[1] = 0.0;
input_.laser[2] = 0.0;
output_.cov_x_m = nullptr;
output_.dx_dy1_m = nullptr;
output_.dx_dy2_m = nullptr;

// Initialize velocity variables
latest_velocity_.linear.x = 0.0; latest_velocity_.linear.y = 0.0;
latest_velocity_.linear.z = 0.0;
latest_velocity_.angular.x = 0.0; latest_velocity_.angular.y = 0.0;
latest_velocity_.angular.z = 0.0;

current_wheel_twist_.linear.x = 0.0; current_wheel_twist_.angular.z = 0.0;
}

```

Code Explanation:

Step 1: Initialize TF system

```
tf_buffer_ = std::make_shared<tf2_ros::Buffer>(this->get_clock());  
tf_listener_ = std::make_shared<tf2_ros::TransformListener>(*tf_buffer_);  
tf_broadcaster_ = std::make_shared<tf2_ros::TransformBroadcaster>(*this);
```

- `Buffer`: Stores all TF transforms
- `Listener`: Listens for TF transforms, automatically updates the buffer
- `Broadcaster`: Broadcasts TF transforms, allowing other nodes to know the robot's pose

Step 2: Create subscribers and publisher

```
laser_scan_subscription_ = this->create_subscription<sensor_msgs::msg::LaserScan>  
(  
    "scan",  
    rclcpp::SensorDataQoS(),  
    std::bind(&PlipOdometry::scanCallback, this, std::placeholders::_1));
```

- Subscribes to the `/scan` topic, using sensor-specific QoS settings
- `SensorDataQoS`: Optimized for sensor data, ensuring reliability
- `std::bind`: Binds the callback function, `_1` is a placeholder for the first parameter

Why use `SensorDataQoS`?

- Sensor data requires reliable transmission (no dropped frames)
- But doesn't need long history caching (only cares about latest data)
- This is ROS2's standard QoS setting for sensor data

Step 3: Initialize state variables

```
initialized_ = false;  
has_wheel_data_ = false;  
base_in_odom_.setIdentity();
```

- `initialized_`: Marks whether the first frame of data has been received
- `has_wheel_data_`: Marks whether wheel odometry has been received
- `base_in_odom_`: Initialized to identity matrix (position 0,0, angle 0)

6.3.3 Parameter Initialization

```
void PlipOdometry::initParams()  
{  
    // Frame names  
    odom_frame_ = this->declare_parameter("odom_frame", "odom");  
    base_frame_ = this->declare_parameter("base_frame", "base_link");  
  
    // Keyframe detection parameters  
    kf_dist_linear_ = this->declare_parameter("kf_dist_linear", 0.1);
```

```

kf_dist_angular_ = this->declare_parameter("kf_dist_angular", 5.0 * (M_PI /
180.0));
kf_dist_linear_sq_ = kf_dist_linear_ * kf_dist_linear_;
kf_scan_count_ = this->declare_parameter("kf_scan_count", 10);

// CSM parameters - Relaxed parameters optimized for tracked vehicles
input_.max_angular_correction_deg = this-
>declare_parameter("max_angular_correction_deg", 90.0);
input_.max_linear_correction = this-
>declare_parameter("max_linear_correction", 1.0);
input_.max_iterations = this->declare_parameter("max_iterations", 20);
input_.epsilon_xy = this->declare_parameter("epsilon_xy", 0.000001);
input_.epsilon_theta = this->declare_parameter("epsilon_theta", 0.000001);
input_.max_correspondence_dist = this-
>declare_parameter("max_correspondence_dist", 1.0);
input_.sigma = this->declare_parameter("sigma", 0.010);
input_.use_corr_tricks = this->declare_parameter("use_corr_tricks", 1);
input_.restart = this->declare_parameter("restart", 1);

// Point-to-line distance mode (core of PLICP)
input_.use_point_to_line_distance = this-
>declare_parameter("use_point_to_line_distance", 1);

// Outlier handling parameters
input_.outliers_maxPerc = this->declare_parameter("outliers_maxPerc", 0.90);
input_.outliers_remove_doubles = this-
>declare_parameter("outliers_remove_doubles", 1);
}

```

Code Explanation:

Keyframe Parameters:

- `kf_dist_linear_ = 0.1`: Update keyframe if movement exceeds 10cm
- `kf_dist_angular_ = 5°`: Update keyframe if rotation exceeds 5 degrees
- `kf_scan_count_ = 10`: Force update at most every 10 frames

Core CSM Parameters:

Parameter	Value	Purpose
<code>max_angular_correction_deg</code>	90°	Maximum allowed angular correction per match
<code>max_linear_correction</code>	1.0m	Maximum allowed linear translation per match
<code>max_iterations</code>	20	Maximum number of iterations
<code>epsilon_xy</code>	1e-6	Translation convergence threshold (meters)
<code>epsilon_theta</code>	1e-6	Angular convergence threshold (radians)
<code>max_correspondence_dist</code>	1.0m	Maximum correspondence distance; points beyond this are ignored

Parameter	Value	Purpose
<code>sigma</code>	0.01	Standard deviation of LiDAR measurement noise
<code>use_point_to_line_distance</code>	1	Use point-to-line distance (core of PLICP)
<code>outliers_maxPerc</code>	0.9	Keep 90% of correspondences, discard the rest as outliers

Why do parameters need to be configurable?

- Different environments require different parameters (corridor vs. open area)
- Different robots have different motion characteristics (tracked vs. wheeled)
- Facilitates debugging and optimization

6.3.4 Wheel Odometry Callback

```
void PlicpOdometry::wheelodomCallback(const nav_msgs::msg::Odometry::SharedPtr
msg)
{
    // Only take the velocity twist, used for prediction
    current_wheel_twist_ = msg->twist.twist;
    has_wheel_data_ = true;
}
```

Code Explanation:

- This is the simplest function in the entire PLICP node
- When a wheel odometry message is received, extracts the velocity information
- Sets the `has_wheel_data_` flag to indicate data has been received

Why only take velocity and not position?

- Position information (x, y, theta in odom) may already contain drift
- Velocity information (twist) is an instantaneous value, relatively reliable
- PLICP uses velocity for prediction and LiDAR for correction

6.3.5 LiDAR Callback Function

```
void PlicpOdometry::scanCallback(const sensor_msgs::msg::LaserScan::SharedPtr
scan_msg)
{
    current_time_ = scan_msg->header.stamp;

    // Initialization when first data is received
    if (!initialized_) {
        createCache(scan_msg);
        if (!getBaseToLaserTf(scan_msg->header.frame_id)) {
            RCLCPP_WARN(this->get_logger(), "Waiting for TF (base->laser)...");
            return;
        }
    }
    laserScanToLDP(scan_msg, prev_ldp_scan_);
}
```

```

        last_icp_time_ = current_time_;
        initialized_ = true;
        return;
    }

    LDP curr_ldp_scan;
    laserScanToLDP(scan_msg, curr_ldp_scan);
    scanMatchWithPLICP(curr_ldp_scan, current_time_);
}

```

Code Explanation:

Initialization branch (first data received):

1. `createCache`: Pre-computes sin/cos values for each angle, improving efficiency
2. `getBaseToLaserTf`: Gets the static transform from base_link to laser_frame
3. `laserScanToLDP`: Converts the ROS2 message to CSM format
4. `last_icp_time_`: Records the timestamp of the first frame
5. `initialized_ = true`: Marks as initialized

Normal processing branch (second frame and beyond):

1. Converts the current frame to CSM format
2. Calls `scanMatchWithPLICP` to perform matching

Why cache sin/cos values?

- Each LiDAR frame has 360 points, each requiring angle calculation
- sin/cos calculations are time-consuming; pre-computation saves significant CPU time
- This is a typical space-for-time optimization

6.3.6 Creating Cache (Pre-computing Trigonometric Functions)

```

void PlicpOdometry::createCache(const sensor_msgs::msg::LaserScan::SharedPtr
scan_msg)
{
    a_cos_.clear();
    a_sin_.clear();
    double angle;
    for (size_t i = 0; i < scan_msg->ranges.size(); i++) {
        angle = scan_msg->angle_min + static_cast<double>(i) * scan_msg-
>angle_increment;
        a_cos_.push_back(cos(angle));
        a_sin_.push_back(sin(angle));
    }
    input_.min_reading = scan_msg->range_min;
    input_.max_reading = scan_msg->range_max;
}

```

Code Explanation:

- Based on the angle range of the first LiDAR frame, calculates the cos and sin for each angle
- Stores them in two vectors for direct use later

- Also saves the minimum/maximum effective distance of the LiDAR

Mathematical Principle:

- Each LiDAR point can be represented in polar coordinates: (r, θ)
- Convert to Cartesian coordinates: $x = r \cdot \cos\theta$, $y = r \cdot \sin\theta$
- Pre-computing $\cos\theta$ and $\sin\theta$ avoids repeated calculations

6.3.7 Getting the base_link to laser_frame Transform

```
bool PlicpOdometry::getBaseToLaserTf(const std::string & frame_id)
{
    geometry_msgs::msg::TransformStamped transform_stamped;
    try {
        transform_stamped = tf_buffer_ -> lookupTransform(
            base_frame_, frame_id, this -> now(), tf2::durationFromSec(1.0));
    } catch (const tf2::TransformException & ex) {
        RCLCPP_WARN(this -> get_logger(), "TF Error: %s", ex.what());
        return false;
    }
    tf2::fromMsg(transform_stamped.transform, base_to_laser_);
    laser_to_base_ = base_to_laser_.inverse();
    return true;
}
```

Code Explanation:

- `lookupTransform`: Looks up the transform from `frame_id` to `base_frame_` in the TF tree
- Parameters: source frame, target frame, timestamp, timeout
- `tf2::fromMsg`: Converts ROS2 message to TF2 data type
- `inverse()`: Computes the inverse transform

Why is this transform needed?

- LiDAR data is in the `laser_frame` coordinate frame
- Robot pose is in the `base_link` coordinate frame
- PLICP matching needs to be performed in a common coordinate frame
- With this transform, LiDAR points can be converted to the `base_link` frame

6.3.8 Converting ROS2 LaserScan to CSM's LDP Format

```
void PlicpOdometry::laserScanToLDP(const sensor_msgs::msg::LaserScan::SharedPtr
scan_msg, LDP & ldp)
{
    unsigned int n = scan_msg -> ranges.size();
    ldp = ld_alloc_new(n); // Allocate memory

    for (unsigned int i = 0; i < n; i++) {
        double r = scan_msg -> ranges[i];
        if (r > scan_msg -> range_min && r < scan_msg -> range_max) {
            ldp -> valid[i] = 1; // Valid point
            ldp -> readings[i] = r; // Distance value
        }
    }
}
```

```

    } else {
        ldp->valid[i] = 0;        // Invalid point
        ldp->readings[i] = -1;
    }
    ldp->theta[i] = scan_msg->angle_min + static_cast<double>(i) * scan_msg->angle_increment;
    ldp->cluster[i] = -1;        // Initially not part of any cluster
}
ldp->min_theta = ldp->theta[0];
ldp->max_theta = ldp->theta[n - 1];

// Initialize odometry and true pose (not used)
ldp->odometry[0] = 0.0; ldp->odometry[1] = 0.0; ldp->odometry[2] = 0.0;
ldp->true_pose[0] = 0.0; ldp->true_pose[1] = 0.0; ldp->true_pose[2] = 0.0;
}

```

Code Explanation:

What is LDP?

- LDP = Laser Data Point, a data structure defined by the CSM library
- Contains: validity, distance, angle, clustering information for each point
- It is the input format for CSM algorithms

Conversion process:

1. Allocate memory: `ld_alloc_new(n)`, where n is the number of points (usually 360)
2. For each point:
 - Check if the distance is within the valid range
 - If valid, set `valid=1` and `readings=r`
 - If invalid, set `valid=0` and `readings=-1`
 - Set the angle `theta[i]`
 - Initialize cluster to -1 (not clustered)
3. Set minimum/maximum angles
4. Initialize odometry and true_pose (not used, set to 0)

Why is the `valid` flag needed?

- LiDAR may not detect certain directions (out of range, absorption, etc.)
- These invalid points cannot be used for matching and need to be flagged

6.3.9 Getting Predicted Motion (Using Wheel Odometry)

```

void PlicpOdometry::getPrediction(double & prediction_change_x,
                                   double & prediction_change_y,
                                   double & prediction_change_angle,
                                   double dt)
{
    double vx = 0.0;
    double vy = 0.0;
    double vth = 0.0;
}

```

```

// [Core] Prefer wheel odometry velocity
if (has_wheel_data_) {
    vx = current_wheel_twist_.linear.x;
    vy = current_wheel_twist_.linear.y;
    vth = current_wheel_twist_.angular.z;
} else {
    vx = latest_velocity_.linear.x;
    vy = latest_velocity_.linear.y;
    vth = latest_velocity_.angular.z;
}

// Remove tiny noise values
if (std::abs(vx) < 1e-4) vx = 0.0;
if (std::abs(vth) < 1e-4) vth = 0.0;

prediction_change_x = vx * dt;
prediction_change_y = vy * dt;
prediction_change_angle = vth * dt;

// Normalize angle
if (prediction_change_angle >= M_PI) prediction_change_angle -= 2.0 * M_PI;
else if (prediction_change_angle < -M_PI) prediction_change_angle += 2.0 *
M_PI;
}

```

Code Explanation:

Why is prediction needed?

- PLICP is an iterative optimization algorithm that requires an initial guess
- A good initial guess speeds up convergence and improves accuracy
- Wheel odometry provides this guess

Prediction calculation:

```

Δx = vx × dt
Δy = vy × dt
Δθ = vth × dt

```

- Assumes the robot moves at constant velocity during dt
- This is the simplest motion model

Noise removal:

- Velocities less than 0.0001 m/s are treated as 0
- Prevents small jitter from affecting the prediction

Angle normalization:

- Ensures the angle is within the range $[-\pi, \pi]$
- Prevents angle accumulation errors

6.3.10 Core PLICP Matching Function

This is the most complex part of the node, parsed section by section:

```
void PlicpOdometry::scanMatchwithPLICP(LDP & curr_ldp_scan, const rcLcPP::Time &
time)
{
    // Set reference frame and current frame
    input_.laser_ref = prev_ldp_scan_;
    input_.laser_sens = curr_ldp_scan;

    // 1. Get prediction from wheel odometry
    double dt = (time - last_icp_time_).seconds();
    double pr_ch_x, pr_ch_y, pr_ch_a;
    getPrediction(pr_ch_x, pr_ch_y, pr_ch_a, dt);

    // Convert prediction to initial guess in the laser frame
    tf2::Transform prediction_change;
    createTfFromXYTheta(pr_ch_x, pr_ch_y, pr_ch_a, prediction_change);

    // Transform to laser frame
    prediction_change = prediction_change * (base_in_odom_ *
base_in_odom_keyframe_.inverse());
    tf2::Transform prediction_change_lidar;
    prediction_change_lidar = laser_to_base_ * base_in_odom_.inverse() *
prediction_change * base_in_odom_ * base_to_laser_;

    input_.first_guess[0] = prediction_change_lidar.getOrigin().getX();
    input_.first_guess[1] = prediction_change_lidar.getOrigin().getY();
    input_.first_guess[2] = tf2::getYaw(prediction_change_lidar.getRotation());
}
```

Code Explanation:

Step 1: Set input parameters

```
input_.laser_ref = prev_ldp_scan_; // Reference frame (previous keyframe)
input_.laser_sens = curr_ldp_scan; // Current frame
```

Step 2: Get predicted motion

```
double dt = (time - last_icp_time_).seconds(); // Time difference
getPrediction(pr_ch_x, pr_ch_y, pr_ch_a, dt); // Predicted  $\Delta x$ ,  $\Delta y$ ,  $\Delta \theta$ 
```

Step 3: Coordinate transformation (most complex part)

```
prediction_change = prediction_change * (base_in_odom_ *
base_in_odom_keyframe_.inverse());
```

- `base_in_odom_`: Current pose
- `base_in_odom_keyframe_`: Pose at keyframe
- Multiplying them gives the transform from keyframe to current

```
prediction_change_lidar = laser_to_base_ * base_in_odom_.inverse() *
prediction_change * base_in_odom_ * base_to_laser_;
```

This complex transformation chain completes the conversion from the base_link frame to the laser_frame frame:

1. base_to_laser_: From base_link to laser_frame
2. base_in_odom_: From odom to base_link
3. prediction_change: Predicted transform
4. base_in_odom_.inverse(): Inverse transform
5. laser_to_base_: From laser_frame to base_link

Why is such a complex transformation needed?

- The prediction is calculated in the base_link frame
- But the CSM library expects the initial guess in the laser_frame frame
- A series of coordinate transformations are needed to convert

```
// Free old memory
if (output_.cov_x_m) { gsl_matrix_free(output_.cov_x_m); output_.cov_x_m =
nullptr; }
if (output_.dx_dy1_m) { gsl_matrix_free(output_.dx_dy1_m); output_.dx_dy1_m =
nullptr; }
if (output_.dx_dy2_m) { gsl_matrix_free(output_.dx_dy2_m); output_.dx_dy2_m =
nullptr; }

// 2. Execute PLICP algorithm
sm_icp(&input_, &output_);

tf2::Transform corr_ch;

if (output_.valid) {
    // === Matching successful, use PLICP result ===
    tf2::Transform corr_ch_l;
    createTfFromXYTheta(output_.x[0], output_.x[1], output_.x[2], corr_ch_l);
    corr_ch = base_to_laser_ * corr_ch_l * laser_to_base_;

    // Update velocity estimate
    latest_velocity_.linear.x = corr_ch.getOrigin().getX() / dt;
    latest_velocity_.angular.z = tf2::getYaw(corr_ch.getRotation()) / dt;
} else {
    // === Matching failed, use predicted value (Dead Reckoning) ===
    RCLCPP_WARN(this->get_logger(), "PLICP Failed/Drop (dt=%.3f), using wheel
prediction!", dt);

    // Reconstruct the predicted transform matrix
    tf2::Transform dr_pose;
    createTfFromXYTheta(pr_ch_x, pr_ch_y, pr_ch_a, dr_pose);
    corr_ch = dr_pose;

    // Don't update latest_velocity_, keep the previous value
}
```

```

// Update pose
base_in_odom_ = base_in_odom_keyframe_ * corr_ch;

// Publish TF and odometry
publishTFAndOdometry();

// Determine if keyframe needs updating
if (newKeyframeNeeded(corr_ch)) {
    ld_free(prev_ldp_scan_);          // Free old keyframe
    prev_ldp_scan_ = curr_ldp_scan; // Update keyframe
    base_in_odom_keyframe_ = base_in_odom_;
} else {
    ld_free(curr_ldp_scan_);          // Free current frame
}

last_icp_time_ = time;
}

```

Code Explanation:

Step 4: Free old memory

```
if (output_.cov_x_m) { gsl_matrix_free(output_.cov_x_m); ... }
```

- The CSM library uses GSL (GNU Scientific Library) matrices
- Need to free the previous results before each call to prevent memory leaks

Step 5: Execute PLICP algorithm

```
sm_icp(&input_, &output_);
```

- This is the core function of the CSM library
- Input: reference frame, current frame, initial guess, various parameters
- Output: matching result (success flag, displacement, angle, covariance, etc.)

Step 6: Handle matching result

When matching succeeds:

```
createTfFromXYTheta(output_.x[0], output_.x[1], output_.x[2], corr_ch_1);
corr_ch = base_to_laser_ * corr_ch_1 * laser_to_base_;
```

- `output_.x[0], x[1], x[2]`: Displacement and angle computed by PLICP (in laser_frame)
- Convert back to base_link frame

When matching fails:

```
createTfFromXYTheta(pr_ch_x, pr_ch_y, pr_ch_a, dr_pose);
corr_ch = dr_pose; // Use predicted value
```

- Prints a warning message

- Uses the wheel odometry prediction as the result

Step 7: Update pose

```
base_in_odom_ = base_in_odom_keyframe_ * corr_ch;
```

- New pose = keyframe pose × relative transform

Step 8: Determine if keyframe needs updating

```
if (newKeyframeNeeded(corr_ch)) {
    ld_free(prev_ldp_scan_);          // Free old keyframe
    prev_ldp_scan_ = curr_ldp_scan_; // Update keyframe
    base_in_odom_keyframe_ = base_in_odom_;
} else {
    ld_free(curr_ldp_scan_);          // Free current frame
}
```

- If movement is large enough, set the current frame as the new keyframe
- Otherwise, free the current frame (no need to save)

6.3.11 Creating TF Transform (from translation and angle)

```
void PlicpOdometry::createTfFromXYTheta(double x, double y, double theta,
tf2::Transform & t)
{
    t.setOrigin(tf2::Vector3(x, y, 0.0));
    tf2::Quaternion q;
    q.setRPY(0.0, 0.0, theta);
    t.setRotation(q);
}
```

Code Explanation:

- This is a helper function to create a TF transform from translation (x, y) and angle θ
- `setOrigin`: Sets the translation vector
- `setRPY`: Creates a quaternion from Euler angles (Roll=0, Pitch=0, Yaw= θ)
- `setRotation`: Sets the rotation

6.3.12 Publishing TF and Odometry

```
void PlicpOdometry::publishTFAndOdometry()
{
    // Publish TF transform
    geometry_msgs::msg::TransformStamped tf_msg;
    tf_msg.header.stamp = current_time_;
    tf_msg.header.frame_id = odom_frame_;
    tf_msg.child_frame_id = base_frame_;
    tf_msg.transform = tf2::toMsg(base_in_odom_);
    tf_broadcaster_>sendTransform(tf_msg);

    // Publish odometry message
    nav_msgs::msg::Odometry odom_msg;
```

```

odom_msg.header.stamp = current_time_;
odom_msg.header.frame_id = odom_frame_;
odom_msg.child_frame_id = base_frame_;
tf2::toMsg(base_in_odom_, odom_msg.pose.pose);

if (has_wheel_data_) {
    odom_msg.twist.twist = current_wheel_twist_;
} else {
    odom_msg.twist.twist = latest_velocity_;
}
odom_publisher_ -> publish(odom_msg);
}

```

Code Explanation:

Publishing TF:

- `frame_id = odom_frame_`: Parent frame (odom)
- `child_frame_id = base_frame_`: Child frame (base_link)
- Transform relation: `base_in_odom_` (pose of base_link in odom)

Publishing Odometry:

- Pose: Converted from `base_in_odom_`
- Velocity: Prefer wheel odometry velocity (if available)
- Otherwise, use the velocity estimated by PLICP

Why publish both TF and odometry?

- TF: For other nodes to perform coordinate transformations
- Odometry: For the navigation system (contains velocity and covariance)

6.3.13 Determining if a New Keyframe is Needed

```

bool PlipcOdometry::newKeyframeNeeded(const tf2::Transform & d)
{
    scan_count_++;
    // Angular change exceeds threshold
    if (std::abs(tf2::getYaw(d.getRotation())) > kf_dist_angular_) return true;
    // Reached forced update frame count
    if (scan_count_ >= kf_scan_count_) {
        scan_count_ = 0;
        return true;
    }
    // Translation exceeds threshold
    double x = d.getOrigin().getX();
    double y = d.getOrigin().getY();
    if (x * x + y * y > kf_dist_linear_sq_) return true;
    return false;
}

```

Code Explanation:

Three conditions (update if any is met):

1. **Large angular change:** Rotation exceeds threshold (e.g., 5 degrees)
2. **Frame count reached:** Processed enough frames (e.g., 10 frames)
3. **Large translation change:** Movement exceeds threshold (e.g., 0.1 meters)

Why is the frame count condition needed?

- If the robot is stationary, the angle and translation conditions will never be triggered
- But the environment might change even when stationary (e.g., someone walks by)
- Forced updates can capture these changes

6.4 Node Summary

The PLICP LiDAR odometry node is a typical sensor fusion node:

Inputs:

- LiDAR data (`/scan`)
- Wheel odometry velocity (`/odom_wheel`)

Outputs:

- Fused odometry (`/odom`)
- TF transform (`odom` → `base_link`)

Core Algorithm:

- Uses wheel odometry for prediction
- Uses PLICP for correction
- Keyframe mechanism for efficiency
- Falls back to prediction when matching fails

Design Features:

- Modular: Clear functional separation
- Configurable: Many adjustable parameters
- Robust: Failure handling mechanism
- Efficient: Optimizations like precomputation and keyframes

Through this node, we achieve tight coupling fusion of LiDAR and wheel odometry, ensuring both short-term accuracy (wheel odometry) and correcting long-term drift (LiDAR), providing a stable and reliable pose estimate for the navigation system.

Part 7: Deployment and Usage Guide

7.1 Quick Start Command Summary

```
# Terminal 1: Start base control, LiDAR, and PLICP LiDAR odometry
ros2 launch evobase_bringup combined.launch.py
```

```
# Terminal 2: Start SLAM mapping
```

```
ros2 launch slam_toolbox online_async_launch.py

# Terminal 3: Teleoperate the robot for mapping
ros2 run teleop_twist_keyboard teleop_twist_keyboard

# Terminal 4: Save the map
ros2 run nav2_map_server map_saver_cli -f ~/my_map

# Terminal 5: Start Navigation2 navigation
ros2 launch evobase_navigation2 navigation2.launch.py
map:=/home/aidlux/my_map.yaml
```

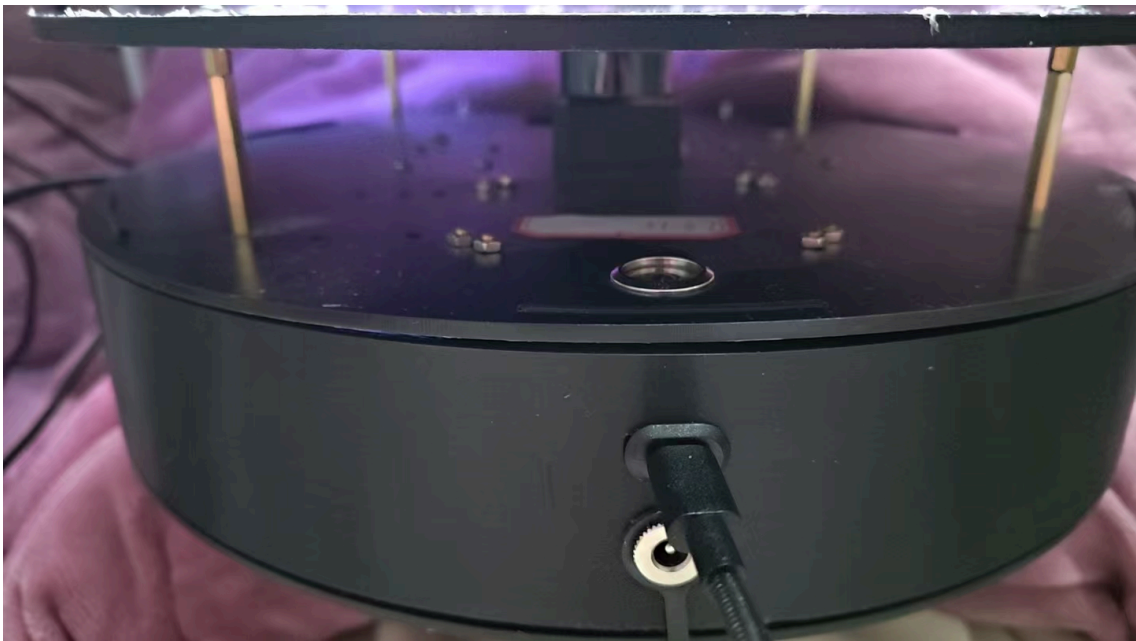
7.2 Startup Process

7.2.1 Power On and Configuration

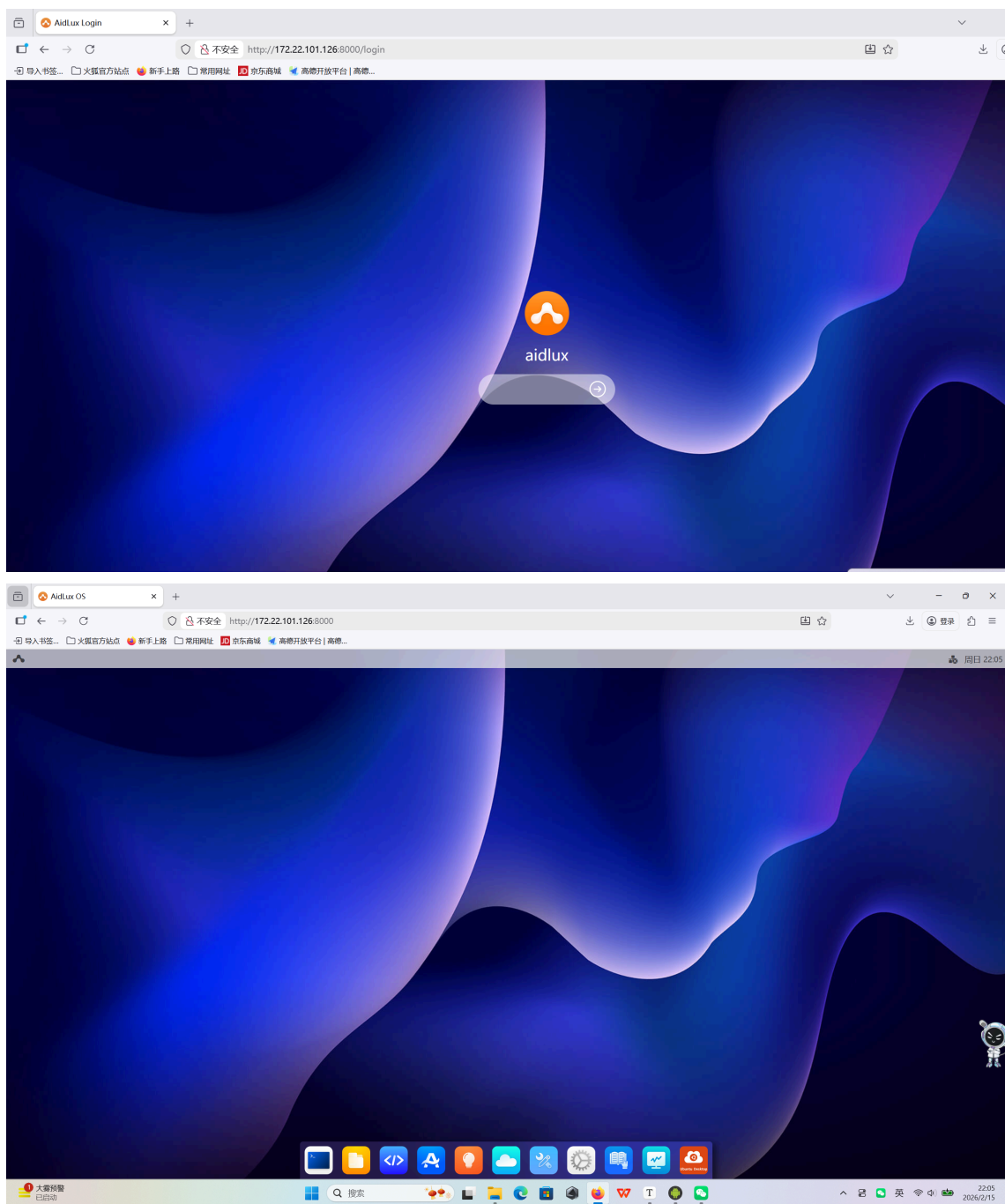
1. Press the power button to turn on the device. The internal lighting should be visible through the bottom.



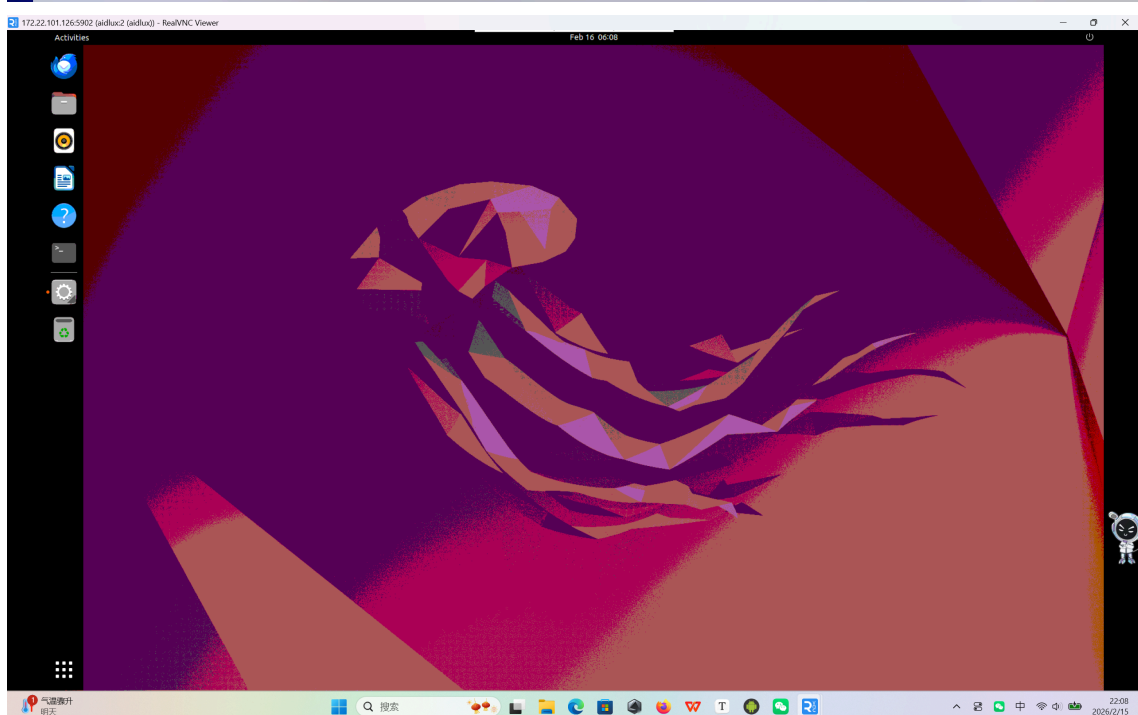
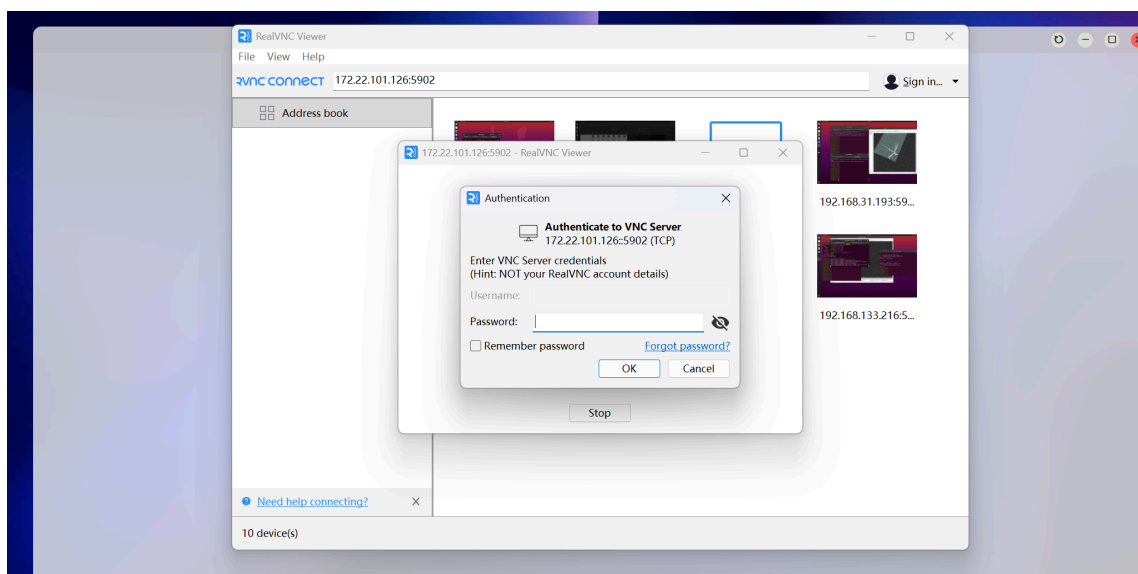
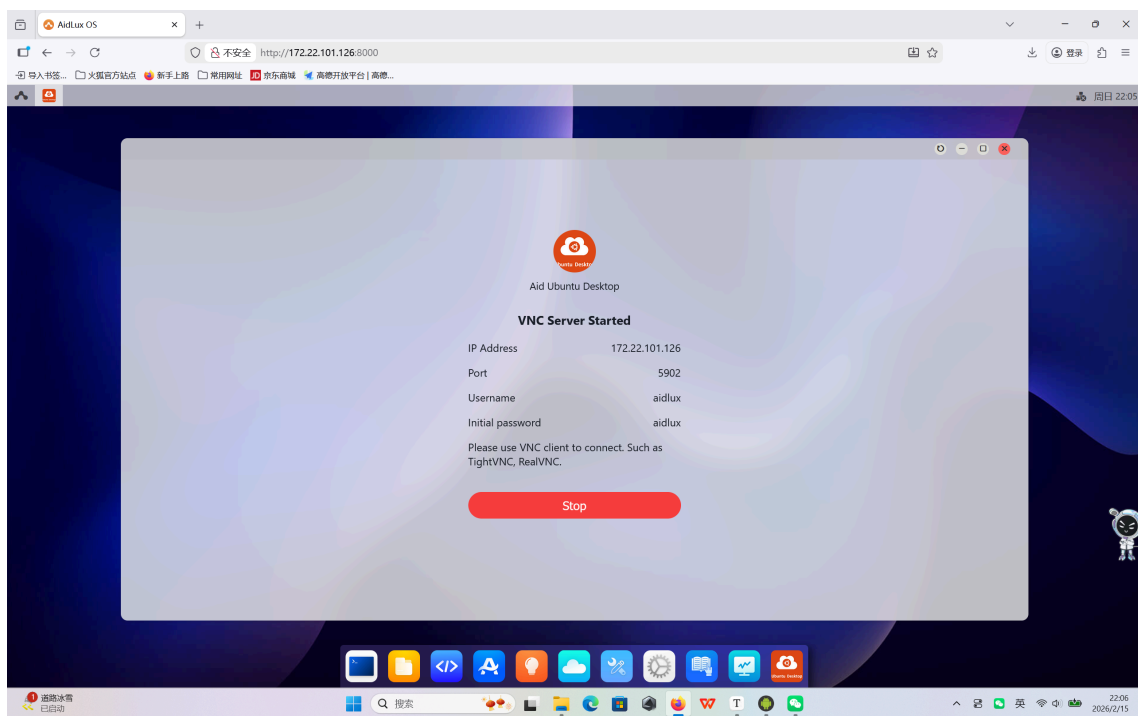
2. Use a Type-C cable for a one-cable connection for network configuration, using screen casting software for operation.



3. After obtaining the device IP address, log in via the web interface.



4. Open ubuntu-desktop and a VNC client to log in.



5. Check serial port device permissions (must be done on first use):

```
# Check if current user is in the dialout group
groups
# If dialout is not in the output, run the following command
sudo usermod -a -G dialout $USER

# Check serial port device permissions
ls -l /dev/ttyUSB* /dev/ttyACM*
# Should show crw-rw-rw- or that the user has read/write permissions

# If no permissions, temporary solution:
sudo chmod 666 /dev/ttyUSB* /dev/ttyACM*
```

7.2.2 Mapping

⚠ Important Notes Before Mapping:

1. Environmental Requirements:

- The effective ranging distance of the LDS-E110 LiDAR is **0.1-12 meters**, but the practical reliable distance is about **6-8 meters**
- **Please choose a small room for mapping** (recommended area 10-30 square meters, such as a bedroom, study, small living room)
- Avoid mapping in large open halls, excessively long corridors (over 8 meters), outdoor environments, etc.
- The room should have sufficient features (walls, furniture, corners, etc.), avoiding completely empty spaces.

2. Typical Failure Scenarios:

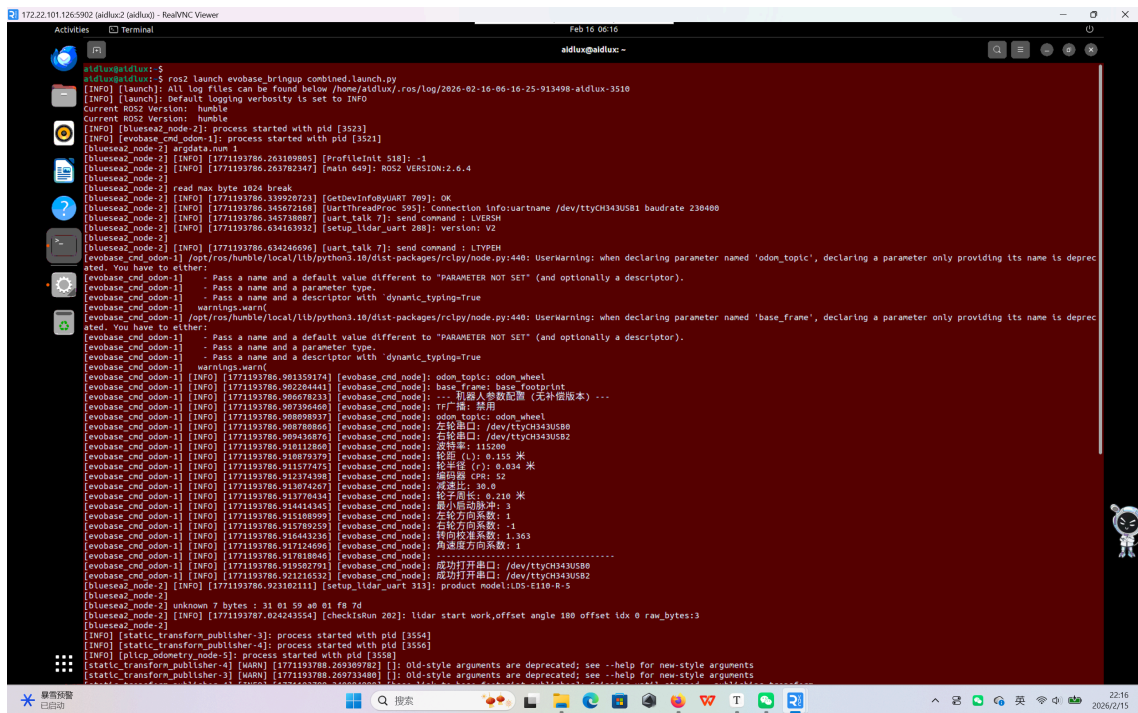
- Large open hall: LiDAR point cloud is too sparse, SLAM cannot extract features → mapping fails or map deforms
- Long corridor (>10 meters): The end of the corridor is beyond the LiDAR's range → unable to close loops, map stretches
- Glass walls/mirrors: Laser penetrates or causes specular reflection → generates false point clouds

3. Mapping Tips:

- Move the robot **slowly at low speed (0.2-0.3 m/s)**
- Try to follow **rectangular paths**, ensuring each area is scanned 2-3 times
- When encountering narrow passages, let the robot **rotate in place** before passing through
- After mapping is complete, **do not save immediately**; let the robot walk around the explored area once more to ensure loop closure

4. Open a terminal and start the base and sensors:

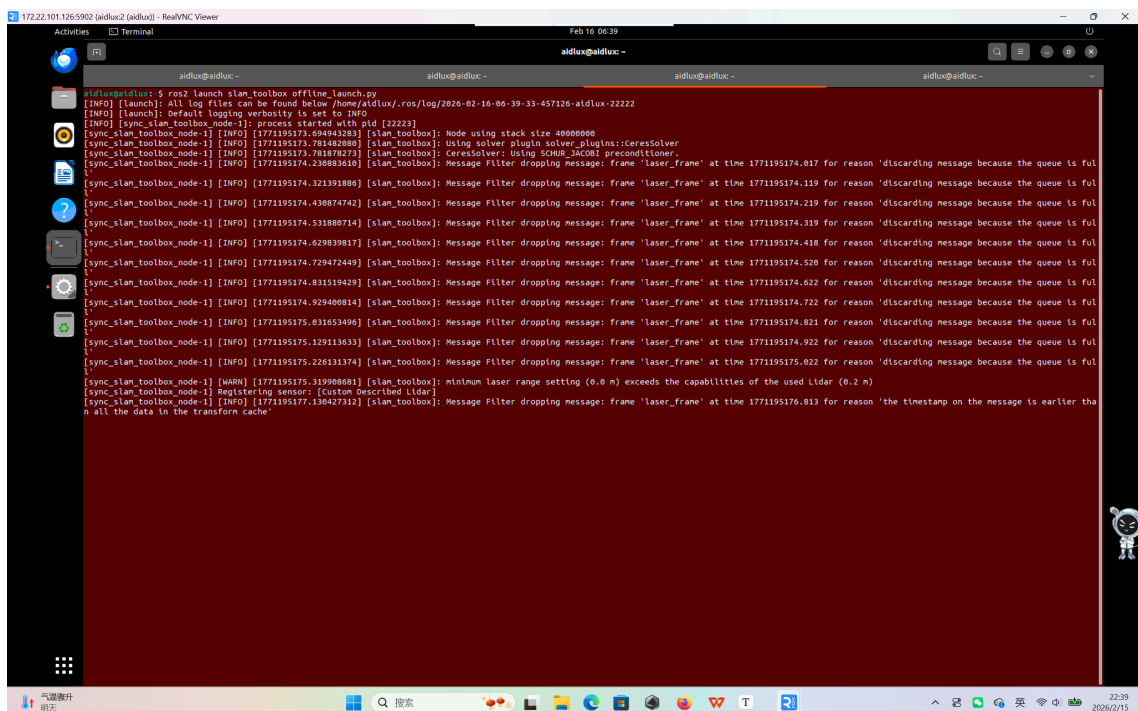
```
ros2 launch evobase_bringup combined.launch.py
```



```
aidlux@aidlux:~$ ros2 launch evobase bringup_combined.launch.py
[INFO] [launch]: All log files can be found below /home/aidlux/.ros/log/2026-02-16-06-16-25-913498-aidlux-3510
[INFO] [launch]: Default logging verbosity is set to INFO
Current ROS2 Version: humble
[INFO] [bluesae2_node-2]: process started with pid [3523]
[INFO] [evobase_cnd_odom-1]: process started with pid [3521]
[bluesae2_node-2] argdata.num 1
[bluesae2_node-2] [INFO] [1771193786.263109985] [ProfileInit 518]: 1
[evobase_cnd_odom-1] [INFO] [1771193786.263782347] [main 649]: ROS2 VERSION:2.6.4
[bluesae2_node-2] [bluesae2_node-2] read max byte 1024 break
[bluesae2_node-2] [INFO] [1771193786.339920723] [GetDevInfoByUART 769]: OK
[bluesae2_node-2] [INFO] [1771193786.346072168] [UartThreadProc 395]: Connection Info:uartname /dev/ttyCH343USB1 baudrate 238400
[bluesae2_node-2] [INFO] [1771193786.345748897] [uart_talk 7]: send Command 1 LVERS04
[bluesae2_node-2] [INFO] [1771193786.634163932] [setup_lidar_uart 288]: version: V2
[bluesae2_node-2] [INFO] [1771193786.634246696] [uart_talk 7]: send command 1 LTYPEH
[evobase_cnd_odom-1] /opt/ros/humble/local/lib/python3.10/dist-packages/rcldp/node.py:440: UserWarning: when declaring parameter named 'odon_topic', declaring a parameter only providing its name is deprecated. You have to either:
- Pass a name and a default value different to "PARAMETER NOT SET" (and optionally a descriptor).
- Pass a name and a parameter type.
- Pass a name and a descriptor with dynamic_typing=True
[evobase_cnd_odom-1] warnings.warn(
[evobase_cnd_odom-1] /opt/ros/humble/local/lib/python3.10/dist-packages/rcldp/node.py:440: UserWarning: when declaring parameter named 'base_frame', declaring a parameter only providing its name is deprecated. You have to either:
- Pass a name and a default value different to "PARAMETER NOT SET" (and optionally a descriptor).
- Pass a name and a parameter type.
- Pass a name and a descriptor with dynamic_typing=True
[evobase_cnd_odom-1] [INFO] [1771193786.981359174] [evobase_cnd_node]: odon_topic: odon_wheel
[evobase_cnd_odom-1] [INFO] [1771193786.982204441] [evobase_cnd_node]: base_frame: base_footprint
[evobase_cnd_odom-1] [INFO] [1771193786.986679233] [evobase_cnd_node]: ... 删除参数数据项 (无升级版本) ...
[evobase_cnd_odom-1] [INFO] [1771193786.987396468] [evobase_cnd_node]: TF广播: 禁用
[evobase_cnd_odom-1] [INFO] [1771193786.988998937] [evobase_cnd_node]: odon_topic: odon_wheel
[evobase_cnd_odom-1] [INFO] [1771193786.988788866] [evobase_cnd_node]: 左轮串口: /dev/ttyCH343USB0
[evobase_cnd_odom-1] [INFO] [1771193786.989436876] [evobase_cnd_node]: 右轮串口: /dev/ttyCH343USB2
[evobase_cnd_odom-1] [INFO] [1771193786.910131866] [evobase_cnd_node]: 波特率: 115200
[evobase_cnd_odom-1] [INFO] [1771193786.910879379] [evobase_cnd_node]: 轴距 (L): 0.155 米
[evobase_cnd_odom-1] [INFO] [1771193786.91577475] [evobase_cnd_node]: 轮半径 (r): 0.034 米
[evobase_cnd_odom-1] [INFO] [1771193786.912314388] [evobase_cnd_node]: 转向轴 Ctr: 52
[evobase_cnd_odom-1] [INFO] [1771193786.913074267] [evobase_cnd_node]: 减速比: 30.0
[evobase_cnd_odom-1] [INFO] [1771193786.913779434] [evobase_cnd_node]: 轮子直径: 0.218 米
[evobase_cnd_odom-1] [INFO] [1771193786.914414345] [evobase_cnd_node]: 最小启动脉冲: 3
[evobase_cnd_odom-1] [INFO] [1771193786.915108999] [evobase_cnd_node]: 左轮方向系数: 1
[evobase_cnd_odom-1] [INFO] [1771193786.915789258] [evobase_cnd_node]: 右轮方向系数: -1
[evobase_cnd_odom-1] [INFO] [1771193786.916443236] [evobase_cnd_node]: 转向校准系数: 1.363
[evobase_cnd_odom-1] [INFO] [1771193786.917124656] [evobase_cnd_node]: 角速度方向系数: 1
[evobase_cnd_odom-1] [INFO] [1771193786.917818946] [evobase_cnd_node]: .....
[evobase_cnd_odom-1] [INFO] [1771193786.919582791] [evobase_cnd_node]: 成功打开串口: /dev/ttyCH343USB0
[evobase_cnd_odom-1] [INFO] [1771193786.921216532] [evobase_cnd_node]: 成功打开串口: /dev/ttyCH343USB2
[bluesae2_node-2] [INFO] [1771193786.923102111] [setup_lidar_uart 313]: product model:LDS E110-R-5
[bluesae2_node-2] [bluesae2_node-2] unknown 7 bytes : 31 01 59 a0 01 f8 fd
[bluesae2_node-2] [INFO] [1771193787.024243554] [checkLidar 282]: lidar start work,offset angle 180 offset idx 0 raw_bytes:3
[bluesae2_node-2] [INFO] [static_transform_publisher-3]: process started with pid [3554]
[INFO] [static_transform_publisher-4]: process started with pid [3556]
[INFO] [alice_odymetry_node-3]: process started with pid [3558]
[static_transform_publisher-4] [WARN] [1771193788.269389782] []: Old-style arguments are deprecated: see --help for new-style arguments
[static_transform_publisher-3] [WARN] [1771193788.269733480] []: Old-style arguments are deprecated: see --help for new-style arguments
```

5. Open a terminal and start the mapping algorithm:

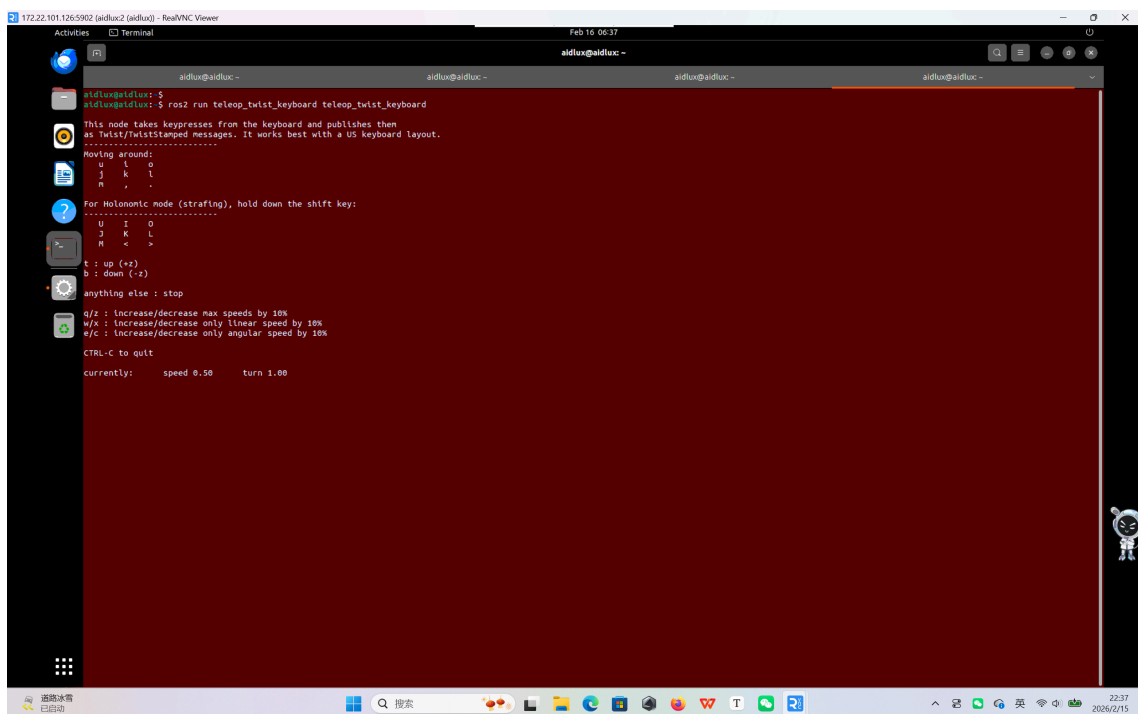
```
ros2 launch slam_toolbox offline_launch.py
```



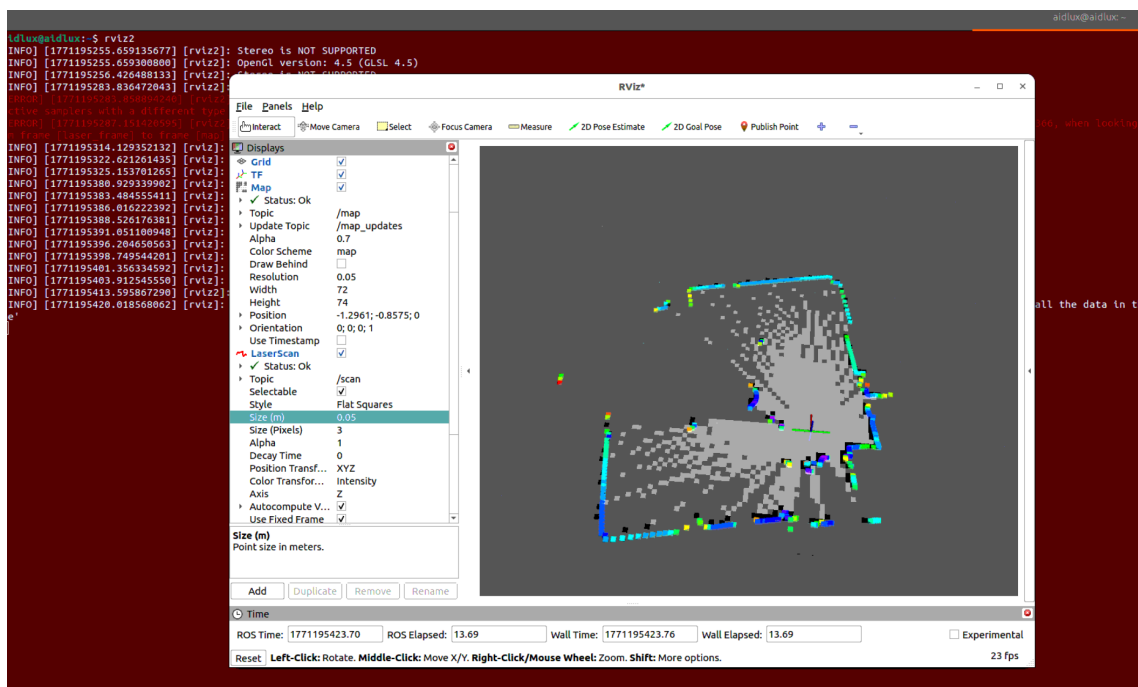
```
aidlux@aidlux:~$ ros2 launch slam_toolbox offline_launch.py
[INFO] [launch]: All log files can be found below /home/aidlux/.ros/log/2026-02-16-06-39-33-457126-aidlux-2222
[INFO] [sync_slam_toolbox_node-1]: process started with pid [22223]
[sync_slam_toolbox_node-1] [INFO] [1771193786.694943283] [slam_toolbox]: Node using stack size 4096000
[sync_slam_toolbox_node-1] [INFO] [1771193786.781482088] [slam_toolbox]: Using solver plugin solver_plugins:CeresSolver
[sync_slam_toolbox_node-1] [INFO] [1771193786.781878273] [slam_toolbox]: CeresSolver: Using SCHUR_JACOBI preconditioner,
[sync_slam_toolbox_node-1] [INFO] [1771193786.786836168] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.017 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.321391886] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.119 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.438874742] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.219 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.531880714] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.319 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.629839817] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.418 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.729472449] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.520 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.831519429] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.622 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193784.929408814] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.722 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193785.031653496] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.821 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193785.129131633] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193784.922 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [INFO] [1771193785.226131374] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193785.022 for reason 'discarding message because the queue is full'
[sync_slam_toolbox_node-1] [WARN] [1771193785.319986881] [slam_toolbox]: minimum laser range setting (0.0 m) exceeds the capabilities of the used Lidar (0.2 m)
[sync_slam_toolbox_node-1] Registering sensor: [Custom Described Lidar]
[sync_slam_toolbox_node-1] [INFO] [1771193787.156477315] [slam_toolbox]: Message Filter dropping message: frame 'Laser_frame' at time 1771193786.813 for reason 'the timestamp on the message is earlier than all the data in the transform cache'
```

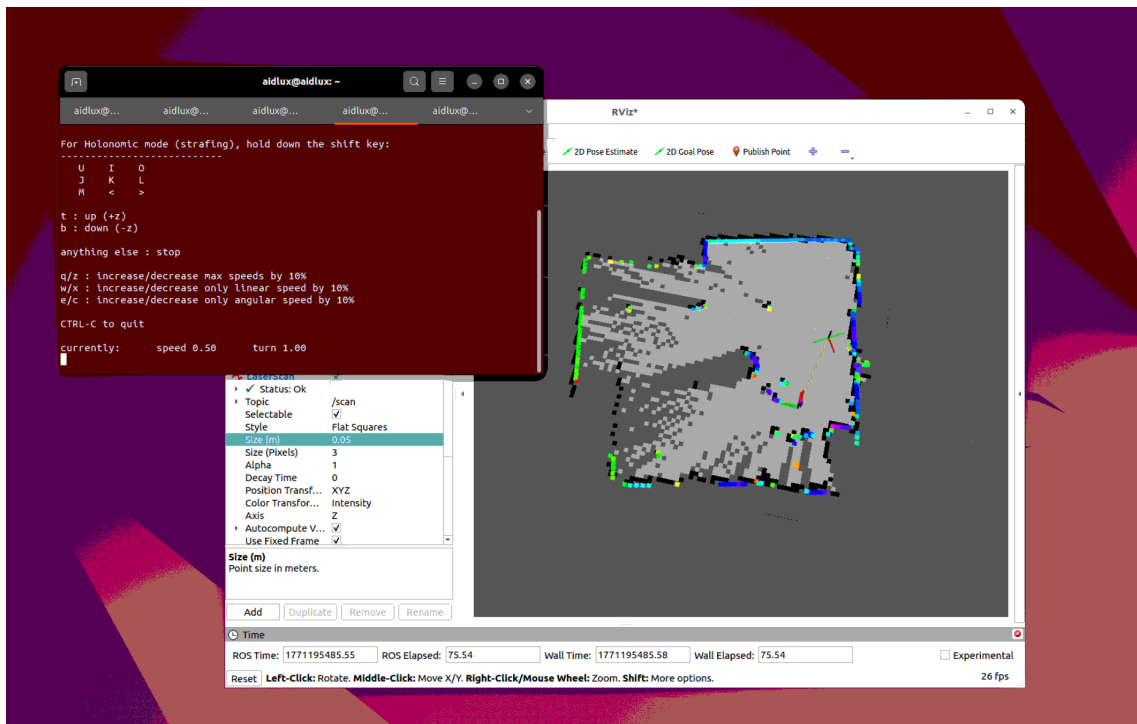
6. Open a terminal and start keyboard teleoperation:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```



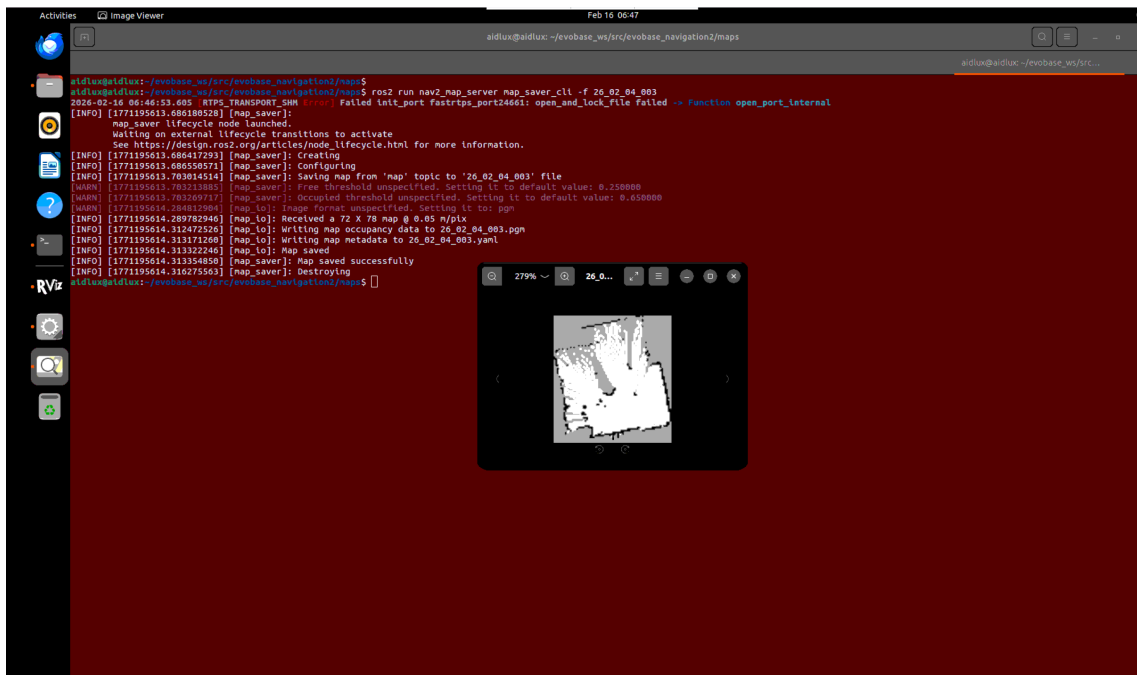
7. Open RViz2 for data visualization:





8. Save the map data:

```
ros2 run nav2_map_server map_saver_cli -f <map_name>
```



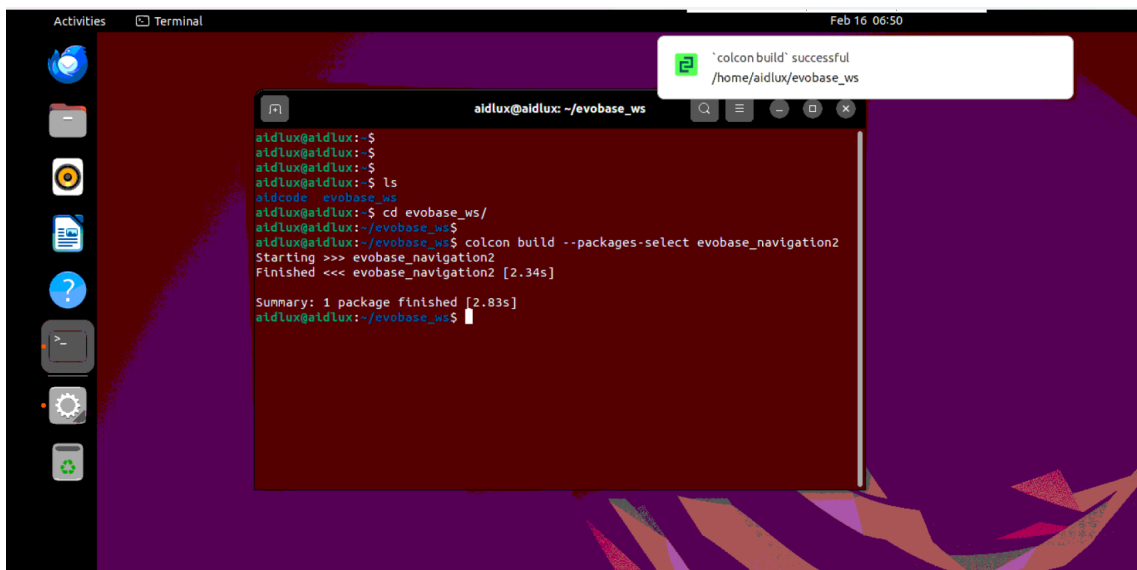
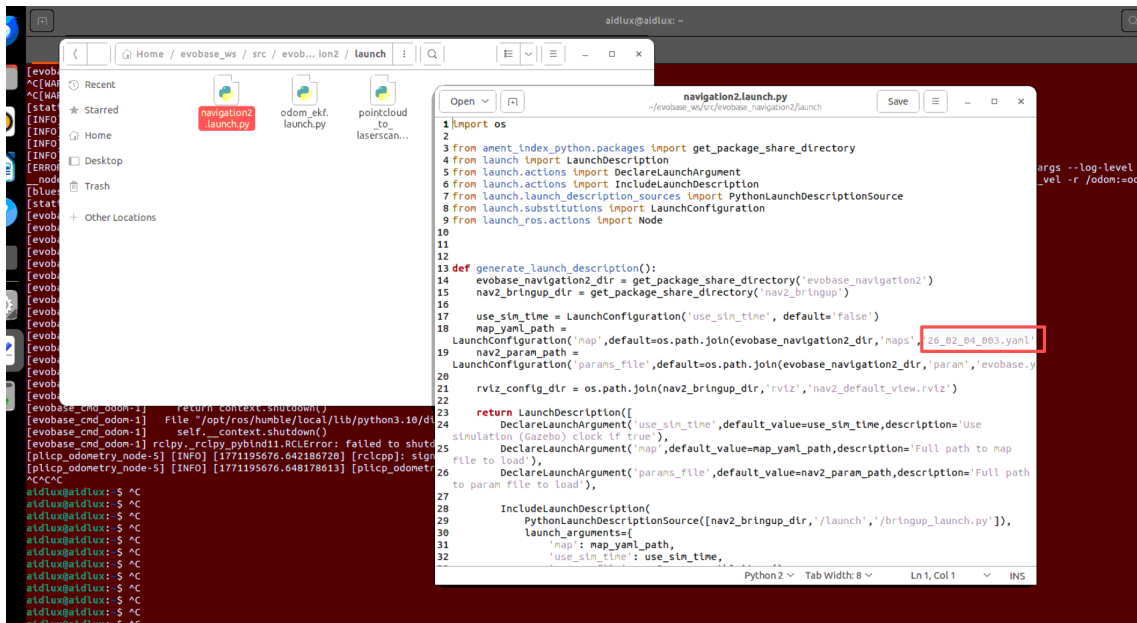
7.2.3 Navigation

Pre-navigation Checklist:

- ☐ The map has been built in a **small room consistent with the current environment**
- ☐ The robot's position roughly corresponds to the map's origin (initial localization)
- ☐ There are no temporary obstacles in the LiDAR's field of view (e.g., opened doors, moved chairs)
- ☐ RViz2 has been started on a remote PC (if the mBot main controller has no GUI)

1. Modify the code to use the newly saved map:

```
colcon build --packages-select evobase_navigation2
```



2. Open a terminal and start the base and sensors:

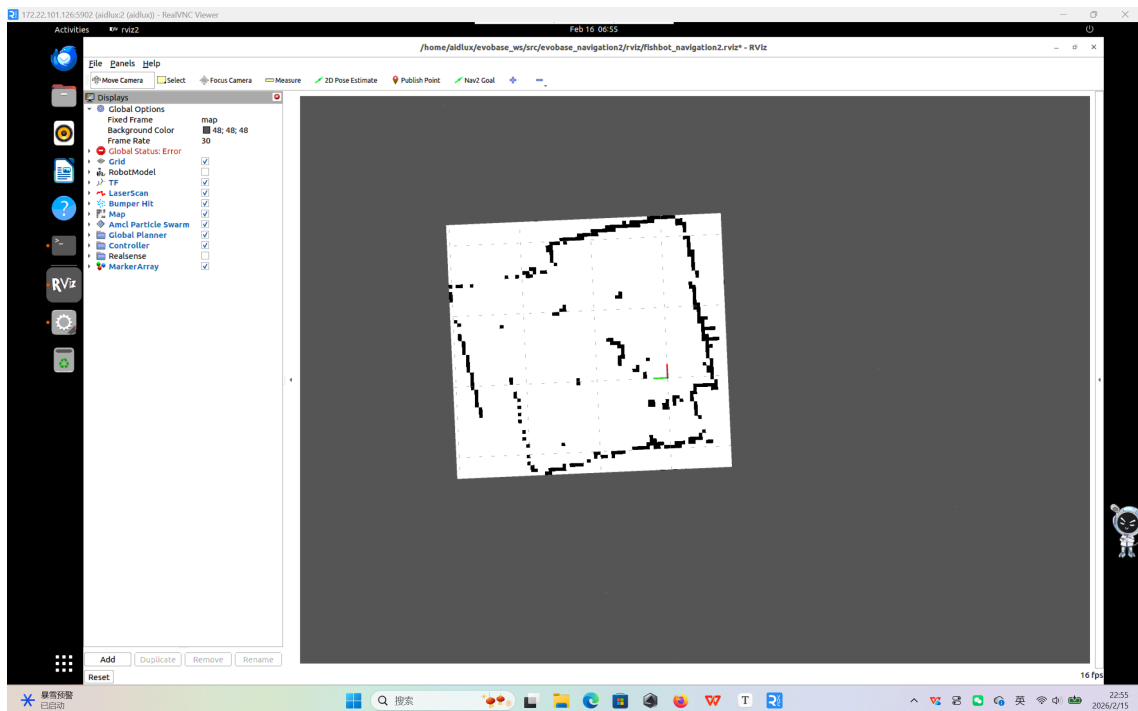
```
ros2 launch evobase_bringup combined.launch.py
```

```
172.22.101.126:5902 (a1d1ux2 (a1d1ux) - RoshINC Viewer
Activities Terminal
Feb 10 06:16
a1d1ux@a1d1ux: ~
a1d1ux@a1d1ux: $ ros2 launch evobase bringup combined.launch.py
[INFO] [launch]: All log files can be found below /home/a1d1ux/.ros/log/2026-02-16-06-16-25-913498-a1d1ux-3510
[INFO] [launch]: Default logging verbosity is set to INFO
Current ROS2 Version: humble
[INFO] [blues2_node-2]: process started with pid [3523]
[INFO] [evobase_cnd_odom-1]: process started with pid [3521]
[blues2_node-2] argdata.num 1
[blues2_node-2] [INFO] [1771193786.263109885] [ProfileInit 518]: 1
[blues2_node-2] [INFO] [1771193786.263782347] [main 649]: ROS2 VERSION:2.6.4
[blues2_node-2]
[blues2_node-2] read max byte 1024 break
[blues2_node-2] [INFO] [1771193786.339926723] [GetDevInfoByUART 789]: OK
[blues2_node-2] [INFO] [1771193786.345672168] [UartFireAndProc 395]: Connection Info:uartname /dev/ttyCH343USB1 baudrate 238408
[blues2_node-2] [INFO] [1771193786.345748897] [uart_talk 71]: send command 1 LVERS0
[blues2_node-2] [INFO] [1771193786.634163932] [setup_lidar_uart 288]: version: V2
[blues2_node-2] [INFO] [1771193786.634246696] [uart_talk 7]: send command 1 LTYPEH
[evobase_cnd_odom-1] /opt/ros/humble/local/lib/python3.10/dist-packages/rcldpy/node.py:440: UserWarning: when declaring parameter named 'odom_topic', declaring a parameter only providing its name is deprecated. You have to either:
- Pass a name and a default value different to "PARAMETER NOT SET" (and optionally a descriptor).
- Pass a name and a parameter type.
[evobase_cnd_odom-1] - Pass a name and a descriptor with 'dynamic_typing=True'
[evobase_cnd_odom-1] warnings.warn()
[evobase_cnd_odom-1] /opt/ros/humble/local/lib/python3.10/dist-packages/rcldpy/node.py:440: UserWarning: when declaring parameter named 'base_frame', declaring a parameter only providing its name is deprecated. You have to either:
- Pass a name and a default value different to "PARAMETER NOT SET" (and optionally a descriptor).
- Pass a name and a parameter type.
[evobase_cnd_odom-1] - Pass a name and a descriptor with 'dynamic_typing=True'
[evobase_cnd_odom-1] warnings.warn()
[evobase_cnd_odom-1] [INFO] [1771193786.981359174] [evobase_cnd_node]: odom_topic: odom_wheel
[evobase_cnd_odom-1] [INFO] [1771193786.982204444] [evobase_cnd_node]: base_frame: base_footprint
[evobase_cnd_odom-1] [INFO] [1771193786.986679231] [evobase_cnd_node]: ... 删除人参数配置 (无升级版本) ...
[evobase_cnd_odom-1] [INFO] [1771193786.987396468] [evobase_cnd_node]: TF广播: 禁用
[evobase_cnd_odom-1] [INFO] [1771193786.989898977] [evobase_cnd_node]: odom_topic: odom_wheel
[evobase_cnd_odom-1] [INFO] [1771193786.989878886] [evobase_cnd_node]: 左轮串口: /dev/ttyCH343USB0
[evobase_cnd_odom-1] [INFO] [1771193786.989436876] [evobase_cnd_node]: 右轮串口: /dev/ttyCH343USB2
[evobase_cnd_odom-1] [INFO] [1771193786.91613366] [evobase_cnd_node]: 电机: 15280
[evobase_cnd_odom-1] [INFO] [1771193786.910879379] [evobase_cnd_node]: 轴距 (L): 0.155 米
[evobase_cnd_odom-1] [INFO] [1771193786.91577475] [evobase_cnd_node]: 轮半径 (r): 0.034 米
[evobase_cnd_odom-1] [INFO] [1771193786.912314384] [evobase_cnd_node]: 转向角 CPH: 52
[evobase_cnd_odom-1] [INFO] [1771193786.913074267] [evobase_cnd_node]: 减速比: 30.8
[evobase_cnd_odom-1] [INFO] [1771193786.913778414] [evobase_cnd_node]: 轮子直径: 0.218 米
[evobase_cnd_odom-1] [INFO] [1771193786.914414345] [evobase_cnd_node]: 最小启动脉冲: 3
[evobase_cnd_odom-1] [INFO] [1771193786.915108999] [evobase_cnd_node]: 左轮方向系数: 1
[evobase_cnd_odom-1] [INFO] [1771193786.915789258] [evobase_cnd_node]: 右轮方向系数: 1
[evobase_cnd_odom-1] [INFO] [1771193786.916443236] [evobase_cnd_node]: 转向控制系数: 1.363
[evobase_cnd_odom-1] [INFO] [1771193786.917124696] [evobase_cnd_node]: 角速度方向系数: 1
[evobase_cnd_odom-1] [INFO] [1771193786.917818946] [evobase_cnd_node]: .....
[evobase_cnd_odom-1] [INFO] [1771193786.919592791] [evobase_cnd_node]: 成功打开串口: /dev/ttyCH343USB0
[evobase_cnd_odom-1] [INFO] [1771193786.921245523] [evobase_cnd_node]: 成功打开串口: /dev/ttyCH343USB2
[blues2_node-2] [INFO] [1771193786.923102111] [setup_lidar_uart 313]: product model:LDS-E110-R-5
[blues2_node-2]
[blues2_node-2] unknown 7 bytes: 31 01 59 a0 01 F8 FD
[blues2_node-2] [INFO] [1771193787.024243554] [checkLidar 282]: lidar start work,offset angle 180 offset idx 0 raw_bytes:3
[blues2_node-2]
[INFO] [static_transform_publisher-3]: process started with pid [3554]
[INFO] [static_transform_publisher-4]: process started with pid [3556]
[INFO] [static_transform_publisher-5]: process started with pid [3558]
[static_transform_publisher-3] [WARN] [1771193788.269389782] []: Old-style arguments are deprecated; see --help for new-style arguments
[static_transform_publisher-3] [WARN] [1771193788.269733480] []: Old-style arguments are deprecated; see --help for new-style arguments
```

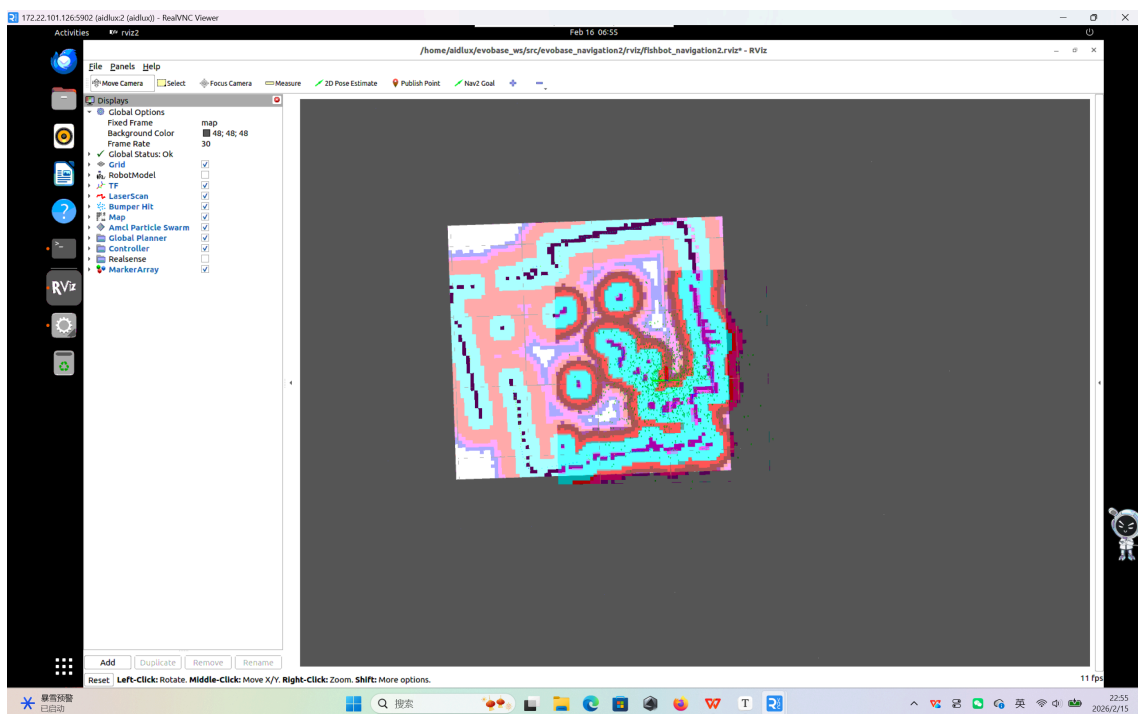
3. Open a terminal and start the navigation algorithm:

ros2 launch evobase_navigation2 navigation2.launch.py

```
a1d1ux@a1d1ux: ~
a1d1ux@a1d1ux: $ ros2 launch evobase_navigation2 navigation2.launch.py
[INFO] [launch]: All log files can be found below /home/a1d1ux/.ros/log/2026-02-16-06-53-21-506586-a1d1ux-28686
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [component_container_isolated-1]: process started with pid [28723]
[component_container_isolated-1] [INFO] [1771196802.79872275] [nav2_container]: Load Library: /home/a1d1ux/evobase_ws/install/nav2_controller/lib/libcontroller_server_core.so
[component_container_isolated-1] [INFO] [1771196802.801516126] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_controller::ControllerServer>
[component_container_isolated-1] [INFO] [1771196802.801657712] [nav2_container]: Instantiate class: rclcpp_components::NodeFactoryTemplate<nav2_controller::ControllerServer>
[component_container_isolated-1] [INFO] [1771196802.934986884] [controller_server]:
controller_server lifecycle node launched.
[component_container_isolated-1]
Waiting on external lifecycle transitions to activate
[component_container_isolated-1] See https://design.ros2.org/articles/node_lifecycle.html for more information.
[component_container_isolated-1] [INFO] [1771196803.082498563] [controller_server]: Creating controller server
[component_container_isolated-1] [INFO] [1771196803.083362735] [local_costmap.local_costmap]:
local_costmap lifecycle node launched.
[component_container_isolated-1]
Waiting on external lifecycle transitions to activate
[component_container_isolated-1] See https://design.ros2.org/articles/node_lifecycle.html for more information.
[component_container_isolated-1] [INFO] [1771196803.094312153] [local_costmap.local_costmap]: Creating costmap
[component_container_isolated-1] [INFO] [1771196803.097675997] [nav2_container]: Load Library: /home/a1d1ux/evobase_ws/install/nav2_map_server/lib/libmap_server_core.so
[INFO] [launch_ros.actions/load_composable_nodes]: Loaded node '/controller_server' in container '/nav2_container'
[component_container_isolated-1] [INFO] [1771196803.142163968] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_map_server::CostmapFilterInfoServer>
[component_container_isolated-1] [INFO] [1771196803.142254947] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_map_server::MapSaver>
[component_container_isolated-1] [INFO] [1771196803.142271248] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_map_server::MapServers>
[component_container_isolated-1] [INFO] [1771196803.142282185] [nav2_container]: Instantiate class: rclcpp_components::NodeFactoryTemplate<nav2_map_server::MapServers>
[component_container_isolated-1] [INFO] [1771196803.221014399] [map_server]:
map_server lifecycle node launched.
[component_container_isolated-1]
Waiting on external lifecycle transitions to activate
[component_container_isolated-1] See https://design.ros2.org/articles/node_lifecycle.html for more information.
[component_container_isolated-1] [INFO] [1771196803.221288191] [map_server]: Creating
[INFO] [launch_ros.actions/load_composable_nodes]: Loaded node '/map_server' in container '/nav2_container'
[component_container_isolated-1] [INFO] [1771196803.232473123] [nav2_container]: Load Library: /home/a1d1ux/evobase_ws/install/nav2_smoother/lib/libsmoother_server_core.so
[component_container_isolated-1] [INFO] [1771196803.253844374] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_smoother::SmootherServer>
[component_container_isolated-1] [INFO] [1771196803.253938113] [nav2_container]: Instantiate class: rclcpp_components::NodeFactoryTemplate<nav2_smoother::SmootherServer>
[component_container_isolated-1] [INFO] [1771196803.329955998] [smoother_server]:
smoother_server lifecycle node launched.
[component_container_isolated-1]
Waiting on external lifecycle transitions to activate
[component_container_isolated-1] See https://design.ros2.org/articles/node_lifecycle.html for more information.
[component_container_isolated-1] [INFO] [1771196803.342257715] [smoother_server]: Creating smoother server
[INFO] [launch_ros.actions/load_composable_nodes]: Loaded node '/smoother_server' in container '/nav2_container'
[component_container_isolated-1] [INFO] [1771196803.346210185] [nav2_container]: Load Library: /home/a1d1ux/evobase_ws/install/nav2_amcl/lib/libamcl_core.so
[component_container_isolated-1] [INFO] [1771196803.370412121] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_amcl::AmclNode>
[component_container_isolated-1] [INFO] [1771196803.370506388] [nav2_container]: Instantiate class: rclcpp_components::NodeFactoryTemplate<nav2_amcl::AmclNode>
[component_container_isolated-1] [INFO] [1771196803.451966648] [amcl]:
amcl lifecycle node launched.
[component_container_isolated-1]
Waiting on external lifecycle transitions to activate
[component_container_isolated-1] See https://design.ros2.org/articles/node_lifecycle.html for more information.
[component_container_isolated-1] [INFO] [1771196803.444957966] [amcl]: Creating
[component_container_isolated-1] [INFO] [1771196803.449658462] [nav2_container]: Load Library: /home/a1d1ux/evobase_ws/install/nav2_planner/lib/libplanner_server_core.so
[INFO] [launch_ros.actions/load_composable_nodes]: Loaded node '/amcl' in container '/nav2_container'
[component_container_isolated-1] [INFO] [1771196803.459288832] [nav2_container]: Found class: rclcpp_components::NodeFactoryTemplate<nav2_planner::PlannerServer>
[component_container_isolated-1] [INFO] [1771196803.459386060] [nav2_container]: Instantiate class: rclcpp_components::NodeFactoryTemplate<nav2_planner::PlannerServer>
[component_container_isolated-1] [INFO] [1771196803.541906131] [planner_server]:
```



4. Set the initial pose:



5. Set the navigation goal:



7.3 Debugging Tools

Check Temperature:

single check

```
python3 src/evobase_bringup/scripts/check_temp.py
```

Dynamic monitoring (updates every 5 seconds)

```
python3 src/evobase_bringup/scripts/dynamic_check_temp.py
```

Check Encoder Data:

```
python3 src/evobase_bringup/scripts/read_encoders_once.py
```

Endurance Test:

```
ros2 run evobase_bringup send_cmd_time_test.py
```

Check Serial Devices:

```
python3 src/evobase_bringup/scripts/find_serial.py
```

ROS2 Debugging Commands:

```
# List all topics
ros2 topic list

# Echo topic data (once)
ros2 topic echo /scan --once

# Check topic frequency
ros2 topic hz /scan

# Publish a velocity command
ros2 topic pub /cmd_vel geometry_msgs/Twist "{linear: {x: 0.2}, angular: {z: 0.0}}"

# View TF tree
ros2 run tf2_tools view_frames
evince frames.pdf

# View node graph
rqt_graph
```

7.4 Common Issues and Solutions

Issue	Possible Cause	Solution
Serial device names not fixed	udev rules not configured	Install udev rules to fix device names
Motor not running or jittering	Insufficient power / PID parameters not tuned	Check power supply, adjust PID parameters
PLICP matching fails	Abnormal LiDAR data / Insufficient environment features	Check LiDAR, adjust parameters
Robot hits wall during navigation	Inflation radius too small / Map accuracy insufficient	Increase inflation radius, re-map
AMCL localization fails	Incorrect initial pose / Map mismatch	Set correct initial pose

7.5 RViz2 Remote Visualization Configuration

Why run RViz2 remotely?

The mBot's main controller (QCS6490 or industrial PC) typically **does not have an integrated GPU or has weak 3D rendering capabilities**. RViz2 requires real-time rendering of LiDAR point clouds, costmaps, robot models, and other 3D content, which has certain graphics performance requirements. Therefore, a **client-server separation architecture** is recommended:

- **mBot Base:** Runs core nodes like base driver, sensor drivers, SLAM, navigation (no GUI needed)
- **Remote PC/VM:** Runs RViz2 for data visualization and interactive control

Configuration Method

Step 1: Ensure Network Connectivity

- Connect the mBot main controller and remote PC to the **same local network** (or configure routing)
- Record the IP addresses of both devices:

```
# On mBot main controller
ip addr show | grep inet
# Example: 192.168.1.100

# On remote PC
ip addr show | grep inet
# Example: 192.168.1.200
```

Step 2: Configure ROS2 Environment Variables (Required on both sides)

Add the following to the `~/.bashrc` file on both the mBot main controller and the remote PC:

```
# Set ROS_DOMAIN_ID (ensure both sides match, range 0-101)
export ROS_DOMAIN_ID=0

# Set automatic discovery range (optional, improves stability)
export ROS_AUTOMATIC_DISCOVERY_RANGE=SUBNET

# If crossing subnets or encountering communication issues, manually set ROS_IP
# On mBot main controller:
export ROS_IP=192.168.1.100
# On remote PC:
export ROS_IP=192.168.1.200
```

Then run `source ~/.bashrc` for the configuration to take effect.

Step 3: Test ROS2 Communication

Run on the mBot main controller:

```
ros2 run demo_nodes_cpp talker
```

Run on the remote PC:

```
ros2 run demo_nodes_cpp listener
```

If messages are received, communication is working.

Step 4: Install ROS2 Jazzy on the Remote PC (Client Only)

If the remote PC is a VM or another physical machine, install ROS2 Jazzy Desktop as well:

```
# Refer to section 3.0.2 for installing ROS2 Jazzy
sudo apt install ros-jazzy-desktop -y

# Install navigation packages (for RViz2 plugins)
sudo apt install ros-jazzy-nav* -y
```

Note: The remote PC only needs the ROS2 base environment and RViz2; **there is no need to compile the mBot workspace.**

Step 5: Start RViz2 and Connect to mBot

Method 1: Direct launch (automatic node discovery)

```
# On the remote PC
rviz2
```

Then in the RViz2 interface:

- 1. Click the `Add` button in the bottom left
- 2. Select the desired display type (e.g., `LaserScan`, `Map`, `RobotModel`)
- 3. Set `Fixed Frame` to `odom` or `map`
- 4. Subscribe to the corresponding topics (e.g., `/scan`, `/map`, `/odom`)

Method 2: Use a pre-configured RViz2 configuration file

```
# On the remote PC, if the mBot workspace is cloned, use the provided rviz config
rviz2 -d ~/evobase_ws/src/evobase_bringup/rviz/evobase.rviz
```

Step 6: Troubleshooting Common Issues

Issue	Solution
RViz2 shows no topics	Check if <code>ROS_DOMAIN_ID</code> matches; check if firewall is blocking UDP ports (7400-7500)
Point cloud or map not updating	Check if <code>Fixed Frame</code> is correct; check if nodes on mBot are running normally
RViz2 is slow or crashes	Reduce point cloud display density (adjust <code>Size</code> parameter in <code>LaserScan</code> display); reduce number of displayed layers

Issue	Solution
Slow RViz2 rendering in VM	Enable 3D acceleration; allocate more video memory (at least 256MB); use latest versions of VMware or VirtualBox

Performance Optimization Suggestions:

- Recommended remote PC configuration: 4-core CPU, 8GB RAM, graphics card supporting OpenGL 3.3+
- For VMs, it is recommended to use VMware Workstation or VirtualBox 7.0+ and install Guest Additions
- On wireless networks, RViz2 may have high latency; a wired network is recommended

Appendices

A.1 Quick Command Reference

```
# workspace management
cd ~/evobase_ws
colcon build --symlink-install
source install/setup.bash

# Launch robot
ros2 launch evobase_bringup combined.launch.py

# Mapping
ros2 launch slam_toolbox online_async_launch.py
ros2 run nav2_map_server map_saver_cli -f ~/my_map

# Navigation
ros2 launch evobase_navigation2 navigation2.launch.py map:=~/my_map.yaml

# Teleoperation
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

A.2 Configuration File Paths

```
# Robot parameter configuration
evobase_bringup/config/evobase_config.yaml

# EKF fusion configuration
evobase_navigation2/config/ekf.yaml

# Navigation parameter configuration
evobase_navigation2/param/evobase.yaml

# PLICP configuration
lidar_odom/config/plicp_odometry.yaml
```

A.3 Udev Rules File and Serial Port Permission Configuration

Installing udev Rules

```
# Navigate to the udev rules directory
cd ~/evobase_ws/src/evobase_bringup/udev

# View the content of the rules file
cat 99-evobase.rules

# Install rules (copy to system directory)
sudo cp 99-evobase.rules /etc/udev/rules.d/

# Reload udev rules
sudo udevadm control --reload-rules
sudo udevadm trigger

# Re-plug the USB devices and check if mapping is successful
ls -l /dev/ttyMOTOR_* /dev/ttyIMU /dev/ttyLIDAR
```

Testing Device Mapping

```
# Run the test script
./test_device_mapping.sh
```

Expected output:

```
/dev/ttyMOTOR_LEFT -> /dev/ttyUSB0
/dev/ttyMOTOR_RIGHT -> /dev/ttyUSB1
/dev/ttyIMU -> /dev/ttyUSB2
/dev/ttyLIDAR -> /dev/ttyUSB3
```

Manual Serial Port Permission Configuration (if udev rules do not set MODE)

If the udev rules do not include `MODE="0666"`, configure permissions manually:

```
# Method 1: Add user to dialout group (recommended)
sudo usermod -a -G dialout $USER
# Logout and login again for the change to take effect

# Method 2: Temporarily grant 777 permissions to all serial devices (debugging only)
sudo chmod 777 /dev/ttyUSB* /dev/ttyACM*
```

Uninstalling udev Rules

```
# Delete the rules file
sudo rm /etc/udev/rules.d/99-evobase.rules

# Reload udev
sudo udevadm control --reload-rules
sudo udevadm trigger
```

Document Information:

- Version: 2.0.0
- Last Updated: February 2026
- License: MIT/Apache 2.0