Last Update: 17Oct 2025

Autonomous Mobile Robot Diploma Outline

ROS 2 and Modern C++

Module 1: ROS 2 Fundamentals and Communication

This module establishes the foundational knowledge of the ROS 2 architecture and communication patterns.

- **1.1. Introduction & Environment Setup**
    - Iinroduction to ROS2 basics course
    - ROS2 Workspace, create your first ros2 workspace
- **1.2. ROS 2 Core Concepts**
    - ROS2 nodes, Overview on nodes and communication models
    - ROS2 packges, ROS2 packge types
    - ROS2 CPP package file system, ROS2 python package file system
- **1.3. Communication: Topics & Services**
    - Publish subscribe model, Multi pub-sub example
    - Server-Clinet comunication model, Multi server-client
- **1.4. ROS 2 Tools & CLI**
    - Turtlesim pkg Overview and filesystem, run Turtlesim using ROS2 CLI (Nodes)
    - Discover Twist msg and topics using ROS2 CLI, move Turtlesim using teleop
    - publsih msg to ros topic using CLI, publsih msg to ros topic using RQT publisher
    - Services CLI tools, call a service Using CLI, call a service Using RQT service caller
    - Discover RQT and rqt_graph plugin
- **1.5. Configuration and Launch**
    - Introduction to ROS paramters, Paramters CLI with Turtlesim
    - spawn service in Turtlesim, clear service in turtlesim
    - Intorduction to Launch files, Launch File Configration and Types, Why pyhon For ROS2 Launch files ?
    - Introduction to Chatter APP launch file, Create Chatter APP launch file, build and Run launch file

Module 2: Advanced ROS 2 Programming (C++ & Python)

This module covers the practical development of ROS 2 nodes in both C++ and Python, leveraging core programming concepts.

- **2.1. C++ Node Development & Concepts**
    - Intoduction to Node creation using CPP
    - **C++ Concepts Review:** classes and objects part 1 & 2, Class Inheritence part1 & 2, Classes conculusion

    - Intro to Bind Function, Bind function difintion and usage, bind with class callback member fuction
    - simple CPP ROS2 node with timer [CPP Concept]
    - Create First CPP ROS Node Composition part 1 & 2
    - create executable files from CPP node files, add dependencies to manifest files
- **2.2. Chatter App Implementation (C++)**
    - Chatter App overview, create chatter App CPP ROS pkg
    - write talker/listener CPP Node part1 & 2
    - Build and run talker/listener CPP Node, Test and Debug talker/listener CPP Node in [chatter App]
- **2.3. Chatter App Implementation (Python)**
    - Create Chatter APP ROS2 pyhon pkg
    - write talker/listener python Node part1 & 2
    - Build and run talker/listener python Node

Module 3: Robot Modeling and Simulation (URDF/Gazebo)

This module focuses on defining the robot's structure and testing its system in a simulated environment.

- **3.1. Robot Description Format**
    - Introduction to **URDF** (Unified Robot Description Format)
    - Defining the robot structure: **Links, Joints, and Frames**
    - Using **XACRO** for modular and simplified URDF creation
    - Adding **Visual and Collision** models (e.g., using STL files)
- **3.2. Coordinate Transforms (TF2)**
    - Understanding the **Transform (TF2) System** in ROS 2
    - Publishing static and dynamic transforms (e.g., base_link to sensor_frame)
    - Viewing the transform tree in **RViz2**
- **3.3. Gazebo Simulation**
    - Introduction to **Gazebo/Ignition** as a dynamic simulator
    - Adding physics, properties, and world elements
    - Spawning the custom URDF robot in Gazebo
    - Using **Gazebo ROS 2 Plugins** (e.g., for differential drive or LiDAR)

Module 4: Robot Kinematics and ROS 2 Control

This module covers the mathematical principles governing mobile robot motion and implements a robust, industry-standard control architecture using the `ros2_control` framework for the simulated robot.

4.1. Mobile Robot Kinematics (Theory)

- **Introduction to Mobile Robot Types:** Overview of common drive configurations (e.g., **Differential Drive**, Ackermann, Omnidirectional) and their applications.
- **Differential Drive Kinematics:**
    - Deriving **Forward Kinematics** (mapping wheel speeds to robot body velocity: $\dot{x},\dot{y},\dot{\theta}$).

    - Deriving **Inverse Kinematics** (mapping robot body velocity to individual wheel speeds: ωL,ωR).
- **Odometry Principles:**
    - Theory of **Dead Reckoning** and calculating robot pose (x,y,θ) from wheel encoder data.
    - Understanding and mitigating odometry drift and errors.

4.2. Low-Level Control Interface (Encoder & Command)

- **Odometry Publisher Node:** Developing a node (in C++ or Python) that simulates or reads **encoder data** (wheel velocities/positions).
- **Implementing the Odometry Calculation:** Coding the **Forward Kinematics** equations to convert wheel data into the robot's estimated pose.
- **Publishing Odometry:** Publishing the **`nav_msgs/Odometry`** message on the `/odom` topic.
- **Publishing TF:** Publishing the dynamic **odom→base_link** Transform using the odometry data.

4.3. Integrating with `ros2_control` and Simulation

- **Introduction to `ros2_control`:** Architecture overview (Controller Manager, Hardware Interface, Controllers).
- **Creating the ROS 2 Control Configuration:** Defining the hardware components (left/right wheel joints) and interfaces (position, velocity, effort).
- **Integrating with Gazebo/Ignition:** Using the **`ros2_control` Gazebo plugin** (or equivalent) to expose the simulated robot's joint interfaces to the ROS 2 system.
- **Configuring the Differential Drive Controller:** Creating the YAML configuration file for the **`diff_drive_controller`**, including kinematics parameters (wheel radius, track width) and PID gains.

4.4. Executing and Testing ROS 2 Control

- **Launch File Update:** Modifying the launch file to start the **`controller_manager`** and load the **`diff_drive_controller`**.
- **Controlling the Robot:**
    - The `diff_drive_controller` automatically subscribes to **`/cmd_vel`** (a `geometry_msgs/Twist` message).
    - Testing control by publishing `Twist` messages to `/cmd_vel` via the CLI and seeing the robot move in Gazebo.
- **Verification:**
    - Monitoring the **`diff_drive_controller`** state and published odometry.
    - Comparing the robot's odometry with its true pose in the Gazebo simulator.
    - Debugging common issues: Inverse kinematics errors and parameter tuning.

Module 5: Sensor Fusion and Advanced Localization (EKF)

This new module introduces probabilistic robotics to achieve a more robust and accurate estimate of the robot's pose by combining data from multiple sensors.

- **5.1. Introduction to Probabilistic Robotics**
  - Understanding the need for **Sensor Fusion** and the limitations of Odometry.
  - Introduction to the **Kalman Filter** and the **Extended Kalman Filter (EKF)**.
- **5.2. Sensor Data Preparation**
  - Reviewing IMU Data: Understanding the **`sensor_msgs/Imu`** message (linear acceleration, angular velocity, orientation).
  - Interfacing with a Simulated **IMU Sensor** (or real hardware).
  - Data Conditioning: Handling noise, biases, and sensor misalignment.
- **5.3. Implementing the Extended Kalman Filter (EKF)**
  - Introduction to the **`robot_localization`** package.
  - Configuring the **EKF Node** (using YAML parameters) to fuse:
    1. **Odometry** (`nav_msgs/Odometry`) for position/velocity.
    2. **IMU** (`sensor_msgs/Imu`) for orientation/angular velocity.
  - Understanding the →base_link transform and its relationship to the control loop.
- **5.4. Testing and Visualization**
  - Launching the EKF node and observing the fused pose on the **`/odometry/filtered`** topic.
  - Visualizing the improved localization accuracy in **RViz2**.
  - Tuning covariance matrices and filter parameters for optimal performance.

Module 6: DDS, Quality of Service (QoS), and Node Lifecycle

This module provides the necessary deep dive into the middleware layer and advanced node management required for a reliable, real-time system.

- **6.1. Introduction to DDS (Data Distribution Service)**
  - **DDS as the ROS 2 Middleware:** Understanding its role in networking and data transport.
  - ROS 2 Abstraction: The concept of **RMW (ROS Middleware)** and the various DDS vendors (Fast RTPS, Cyclone DDS, etc.).
  - DDS Discovery: How nodes and topics find each other in a distributed network.
- **6.2. Quality of Service (QoS) Policies**
  - Understanding the need for QoS in real-time systems.
  - **Key QoS Settings:**
    - **History:** Keep Last vs. Keep All.
    - **Reliability:** Best Effort vs. **Reliable** (Crucial for service and action communications).
    - **Durability:** Transient Local vs. Volatile (Important for map data).
    - **Liveliness and Deadline:** Configuring real-time guarantees.
  - Implementing and testing QoS settings in C++ and Python publishers/subscribers.
- **6.3. Managed Nodes and Node Lifecycle**
  - Introduction to **Lifecycle Nodes** and the need for deterministic startup/shutdown.
  - **Lifecycle States:** Understanding the sequence (Unconfigured, Inactive, Active, Finalized).
  - **Lifecycle Transitions:** Implementing the callbacks for transitions like `configure`, `activate`, and `deactivate`.
  - **System Management:** Using the `lifecycle_manager` in launch files to coordinate the startup of critical components (like Nav2 servers) for predictable behavior.

Module 7: Perception and Autonomous Navigation (Nav2)

This final module integrates all elements to achieve true autonomy.

- **6.1. Sensor Integration and SLAM**
  - Interfacing with **LiDAR** and publishing `sensor_msgs/LaserScan`.
  - Using the **SLAM Toolbox** for 2D map creation.
- **6.2. The Nav2 Stack and Localization**
  - Nav2 Architecture and Core Components (Planner, Controller, Behavior Tree).
  - **AMCL (Adaptive Monte Carlo Localization):** Using AMCL to localize the robot on a pre-built map using LiDAR data.
- **6.3. Path Planning and Execution**
  - Configuring **Global Planners** (e.g., A*) and **Local Controllers** (e.g., DWA).
  - Sending navigation goals using **RViz2** and a custom **Action Client Node**.
- **6.4. Final Project Integration**
  - Creating a final, comprehensive launch file to run the entire stack: **Robot Model, `ros2_control`, EKF, SLAM/AMCL, and Nav2**.

Module 8: Microcontroller Hardware Control

This module covers connecting the high-level ROS 2 system on the Raspberry Pi to a low-level microcontroller (like an ESP32 or Arduino) that directly controls the robot's motors.

- **8.1. The Two-Brain Approach**
  - **Why use two computers?** We split the work. The **Raspberry Pi** is the "smart brain" running complex ROS 2 tasks like navigation. The **microcontroller (MCU)** is the "muscle brain," handling the fast, real-time job of spinning motors precisely.
  - They communicate over a simple **serial (USB) connection**.
- **8.2. Programming the Microcontroller**
  - The firmware on the MCU has two main jobs:
    - **Listen for commands**: It receives target wheel speeds from the Raspberry Pi (e.g., "left wheel, spin at 5 rad/s").
    - **Control motors & report back**: It uses a **PID control loop** to accurately drive the motors and reads **wheel encoders** to measure the actual speed, sending this data back to the Pi.
- **8.3. Bridging to `ros2_control`**
  - We create a special C++ node on the Raspberry Pi called a **Hardware Interface**.
  - This node acts as a **translator**:
    - It takes velocity commands from the ROS 2 navigation system.
    - It converts them into simple serial commands for the microcontroller.
    - It reads the encoder data from the serial port and publishes it as standard ROS 2 odometry messages.

- **8.4. Final Integration and Testing**
  - We update the robot's configuration file to tell `ros2_control` to use our new hardware interface translator.
  - With everything running, we can now send a `Twist` message to the `/cmd_vel` topic, and the physical robot will move. This confirms that the high-level ROS 2 commands are successfully controlling the low-level hardware.

Module 9: Robot Hardware Integration (Raspberry Pi)

This module covers the practical steps of setting up a Raspberry Pi as the onboard computer for the mobile robot, bridging the gap from simulation to a physical system.

- **9.1. Hardware Selection and Preparation**
  - Why Raspberry Pi?: Benefits for robotics (cost, form factor, GPIO, community).
  - Choosing a Model: Comparing Raspberry Pi 4 vs. 5 (RAM, processing power).
  - Essential Peripherals: Selecting an appropriate SD card, power supply, and cooling solution.
- **9.2. OS Installation and Headless Setup**
  - Choosing an Operating System: **Ubuntu Server** (e.g., 22.04 LTS) for ROS 2 compatibility.
  - Flashing the OS Image: Using the Raspberry Pi Imager.
  - **Headless Configuration**: Enabling SSH and configuring Wi-Fi for remote access without a monitor.
- **9.3. Onboard Software Installation**
  - Installing ROS 2: Following the official steps for Debian packages on an ARM64 architecture.
  - Environment Configuration: Sourcing ROS 2 and adding it to `.bashrc` for persistence.
  - Building a Test Workspace: Compiling a simple C++ publisher to verify the toolchain and ROS 2 installation.
- **9.4. Networking and Remote Development**
  - Configuring a **Static IP Address** for reliable network discovery.
  - Distributed ROS 2 System: Setting up communication between the Raspberry Pi and a development workstation using `ROS_DOMAIN_ID`.
  - Remote Workflow: Using Visual Studio Code with the **Remote-SSH extension** to edit, compile, and run code directly on the Pi.

Module 10: Robot Assembly: Mechanical & Electrical

This module provides a practical, hands-on guide to building the physical robot from individual components. 🛠️

- **8.1. Bill of Materials (BOM) & Tools**:
  - Reviewing the complete list of required parts: chassis, motors, encoders, motor driver, microcontroller, Raspberry Pi, sensors, battery, etc.
  - Overview of essential tools: soldering iron, multimeter, wire strippers, screwdrivers.
- **8.2. Mechanical Assembly**:
  - Step-by-step instructions for assembling the robot's chassis.
  - Mounting the DC motors, wheels, and wheel encoders.
  - Attaching the caster wheel for stability.

- **8.3. Electrical Wiring & Power System**:
  - Creating a power distribution system from the battery.
  - Using buck converters to supply the correct voltage (e.g., 5V) to the Raspberry Pi and microcontroller.
  - Wiring the motor driver to the microcontroller and motors.
- **8.4. Integrating Electronics & Sensors**:
  - Securely mounting the Raspberry Pi, microcontroller, and motor driver to the chassis.
  - Connecting the LiDAR and IMU sensors to the Raspberry Pi.
  - Finalizing the build with clean cable management to ensure reliability.