# Mapping & SLAM Fundamentals using TurtleBot3 and ROS2

## 1. Purpose & Objective

Purpose:
This document provides a structured technical overview of Simultaneous Localization and Mapping (SLAM) principles as implemented on the TurtleBot3 platform using ROS2. It focuses on system architecture, data flow, and practical execution.

Learning Objectives:
Upon reviewing this document, the learner will be able to:

- Define core **SLAM**, **mapping**, and **localization** concepts.
- Describe the hardware and software architecture of a TurtleBot3 performing SLAM.
- Trace the data flow from sensor inputs to map outputs.
- Execute a basic SLAM pipeline in simulation or on a physical robot.
- Save and load occupancy grid maps for later use.

**Why SLAM is Needed:**
In mobile robotics, a robot must navigate an initially **unknown** environment. SLAM solves two interdependent problems simultaneously:

- Mapping: Constructing a representation of the environment.
- Localization: Estimating the robot's pose (position and orientation) within that map.
  Without a prior map, a robot cannot localize accurately. Without accurate localization, it cannot build a consistent map. SLAM is therefore a foundational capability for autonomous exploration, navigation, and task execution.

## 2. Definitions & Concepts

SLAM (Simultaneous Localization and Mapping): A computational problem where a robot builds a map of an unknown environment while simultaneously tracking its location within that map.

**Mapping**: The process of creating a spatial model (map) of the environment from sensor data.

**Localization**: The process of determining a robot's pose (x, y, yaw) relative to a map or a coordinate frame.

Occupancy Grid Map: A discrete grid representation of the environment where each cell (pixel) holds a probability value indicating whether it is occupied (obstacle), free (traversable space), or unknown.

TF (Transform Library): A ROS2 system that keeps track of multiple coordinate frames (e.g., `base_link`, `odom`, `map`, `laser`) and their relationships over time. It answers questions like "Where is the laser scanner relative to the robot's center?"

Odometry: The use of data from motion sensors (wheel encoders, IMU) to estimate the change in the robot's pose over time. It provides a relative, drift-prone pose estimate (`odom` frame relative to `base_link`).

# 3. System Architecture

## TurtleBot3 Hardware (Burger/Waffle Models)

- Chassis: Differential drive mobile base.
- Sensing for SLAM:
    - 2D LiDAR (LDS-01/02): A 360° planar laser scanner. Primary sensor for perceiving wall and obstacle geometry.
    - Wheel Encoders: Built into Dynamixel motors. Provide angular displacement of each wheel.
    - IMU (Inertial Measurement Unit): Measures angular velocity and linear acceleration. Used for improving orientation estimation.

## ROS2 Node Architecture

The SLAM system is composed of several interconnected nodes:

1. `turtlebot3_core` / `turtlebot3_node`: The main robot driver node.
    - Responsibility: Interfaces with hardware (motors, encoders, IMU). Publishes raw odometry (`/odom`) and sensor data. Subscribes to velocity commands (`/cmd_vel`).

2. `rplidar_node` (or similar): The LiDAR driver node.
    - Responsibility: Publishes laser scan messages (`/scan`) containing arrays of range measurements.
3. `robot_state_publisher`: A standard ROS2 node.
    - Responsibility: Uses the URDF robot description and joint states (wheel positions) to broadcast the static transform from `base_link` to `laser` (and other links) via the TF tree.
4. `slam_toolbox` (or `slam_gmapping`): The core SLAM algorithm node.
    - Responsibility: Subscribes to `/scan`, `/tf`, and `/odom`. Fuses this data to compute the most likely map and the robot's pose within it. Publishes the occupancy grid (`/map`) and the map-to-odom transform (`/tf`).

Logical Control Flow:

1. Sensor data (LiDAR scans, encoder ticks) is published by driver nodes.
2. `robot_state_publisher` calculates and broadcasts static transforms.
3. The `slam_toolbox` node:
   a. Listens for incoming scans.
   b. Uses the TF tree to relate the scan to the robot's base.
   c. Uses odometry as an initial motion estimate.
   d. Matches the current scan against the growing map to refine the robot's pose (scan matching).
   e. Integrates the aligned scan into the probabilistic occupancy grid.
   f. Publishes the updated map and the corrected robot pose.

# 4. Data Flow Explanation

## Inputs

- Sensor Topics:
    - `/scan` (`sensor_msgs/msg/LaserScan`): Raw distance measurements from the LiDAR.
    - `/imu` (`sensor_msgs/msg/Imu`): Inertial data (optional but recommended).
    - `/odom` (`nav_msgs/msg/Odometry`): Raw pose estimate from wheel encoders.
- Transform (TF) Tree: Dynamic transforms, particularly `odom -> base_link`.

### Processing

1. Data Acquisition: SLAM node subscribes to `/scan` and `/odom`.
2. Motion Prediction: The latest odometry provides a prediction of how the robot has moved since the last scan.
3. Scan Matching: The algorithm (e.g., iterative closest point) aligns the new scan with the existing map based on the predicted pose, correcting for odometry drift.
4. Pose Update: The robot's estimated pose in the `map` frame is updated.
5. Map Update: The aligned scan is integrated into the occupancy grid map, updating cell probabilities.
6. Transform Publishing: The node calculates and publishes the continuous transform from the `map` frame to the `odom` frame. This "corrects" the drifting odometry.

### Outputs

- Mapping Output:
  - `/map` (`nav_msgs/msg/OccupancyGrid`): The 2D occupancy grid map.
- Localization Output:
  - TF Broadcast: The `map -> odom` transform. This allows any node to query the robot's corrected pose in the map frame by chaining transforms: `map -> odom -> base_link`.

## 5. Inputs and Outputs Table

| Component | Input | Output | Description |
|---|---|---|---|
| LiDAR Driver | Serial/USB data from LiDAR | `/scan` (`LaserScan`) | Converts raw sensor readings to a standardized laser scan message. |

| | | | |
|---|---|---|---|
| Robot Driver | Encoder ticks, IMU data | `/odom` (`Odometry`) | Calculates and publishes dead-reckoning odometry. |
| robot_state_publisher | URDF, Joint States | Static TF (`base_link` -> `laser`) | Computes and broadcasts the fixed transform from robot base to sensor. |
| slam_toolbox Node | `/scan`, `/odom`, `/imu`, TF | `/map` (`OccupancyGrid`), TF (`map` -> `odom`) | Core SLAM processing. Fuses inputs to produce a map and localized robot pose. |
| Teleop Node | Keyboard/twist commands | `/cmd_vel` (`Twist`) | Publishes velocity commands to drive the robot for mapping. |

## 6. SLAM Pipeline Explanation

1. Initialization: The SLAM node starts with an empty map. The robot's initial pose is set to (0,0,0) in the `map` frame.
2. LiDAR Scan Acquisition: A new `/scan` message arrives. It contains an array of range measurements for fixed angles.
3. Odometry Integration: The node checks the latest `/odom` message. The difference between the current and previous odometry pose gives a motion

prior (Δx, Δy, Δθ). This predicts where the new scan *should* appear in the existing map.

4. Scan Matching (Localization): The algorithm takes the predicted pose and performs scan-to-map matching. It adjusts the predicted pose (x, y, yaw) to find the alignment that best fits the new scan points into the existing probabilistic map. This corrected pose is the robot's updated localization.
5. Map Update (Mapping): Using the newly corrected pose, each point in the LiDAR scan is projected onto the occupancy grid. The probability values of the affected cells are updated (e.g., cells along the beam become more "free," cells at the endpoint become more "occupied").
6. Loop Closure (Advanced): Over time, as the robot revisits areas, `slam_toolbox` recognizes this and performs loop closure. It corrects accumulated drift by adjusting the entire pose history and map, ensuring global consistency.

Role of IMU and Odometry: Odometry provides a high-frequency, short-term motion estimate but suffers from drift (wheel slip, uneven floors). The IMU provides robust orientation data, especially for yaw, which helps correct odometric drift during rotation. The SLAM algorithm uses these as a prior, but the final pose is dominated by the accurate LiDAR scan-to-map alignment.

# 7. Practical Example (TurtleBot3 in Simulation)

## Package Installation

```bash
# Update package lists
sudo apt update

# Install TurtleBot3 ROS2 packages
sudo apt install ros-humble-turtlebot3*

# Install SLAM Toolbox
sudo apt install ros-humble-slam-toolbox
```

Setup: This example uses ROS2 Humble and the TurtleBot3 in Gazebo simulation.

1. Launch the Simulation World and Robot:

```
export TURTLEBOT3_MODEL=waffle

ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

    - Action: Starts Gazebo, spawns a TurtleBot3 in a predefined world, and launches all necessary drivers (simulated LiDAR, IMU, odometry).
2. Launch the SLAM Node (`slam_toolbox`):

```
ros2 launch slam_toolbox online_async_launch.py

Or

ros2 launch turtlebot3_cartographer cartographer.launch.py
```

    - Action: Starts the `slam_toolbox` node with a default configuration. It will now subscribe to `/scan`, `/odom`, and `/imu` topics from the simulation.
3. Control the Robot for Exploration:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

    - Action: Starts a node that listens to keyboard inputs (WASD) and publishes corresponding `Twist` messages to the `/cmd_vel` topic, driving the robot.
4. Observe Active Topics:

```
ros2 topic list
```

    - Expected Topics: `/scan`, `/odom`, `/map`, `/tf`, `/cmd_vel`.

## 8. Example Code Snippets

### Start SLAM

```
ros2 launch turtlebot3_cartographer cartographer.launch.py
```

OR using slam_toolbox:

```
ros2 launch turtlebot3_slam slam.launch.py
```

slam_toolbox subscribes automatically to:

- `/scan`
- `/odom`
- `/tf`

### SAVING THE MAP

Once mapping is complete:

```
ros2 run nav2_map_server map_saver_cli -f tb3_map
```

Generated files:

- tb3_map.pgm
- tb3_map.yaml

## 9. Expected Results

During Operation:

- In RViz (visualization tool), you will see:
    1. Robot Model: A URDF model of the TurtleBot.
    2. Laser Scans: Red points/circles appearing around the robot, showing real-time LiDAR hits on obstacles.

3. Growing Map: A grey-scale grid map (`/map`) being drawn in real-time. Dark cells (high probability) are obstacles, light cells are free space, and grey is unknown.
4. Pose Estimate: The robot's arrow in RViz should move smoothly and correspond accurately with the real/simulated robot's movement. The `map -> odom` transform continuously corrects any visual drift.

Final Map Output:

- A coherent, globally consistent 2D floor plan of the environment.
- Walls and permanent obstacles are clearly defined.
- Open spaces are clear.
- The map should be metrically accurate, allowing for precise navigation.

## 10. Conclusion

This document has detailed the fundamental process of performing SLAM with TurtleBot3 and ROS2. The learner should now understand how sensor data flows through the system to solve the interdependent problems of localization and mapping, resulting in a usable occupancy grid.

Next Step – Navigation: The output of the SLAM process is the `/map` topic and the established `map` coordinate frame. This is the critical input for the Navigation2 (Nav2) stack. Nav2 uses this map to perform:

- Localization: (`amcl` node) precisely tracks the robot within the now-known map.
- Path Planning: Calculates obstacle-free paths from start to goal.
- Motion Control: Executes the path by sending `cmd_vel` commands.

Thus, a successful SLAM operation provides the foundational world model required for fully autonomous robot navigation.