ROS2 LiDAR Data and Bump and Go Roomba

Table of Contents

Introduction 2

Objective 2

Background 3

Ignition Gazebo Sim 3

2D LiDAR Technology 5

ROS2: *sensor_msgs/msg/LaserScan* 5

Bring up 7

Diff Drive Simulation Bring up 7

GoBilda Hardware Bring up 7

Foxglove Visualization (Running Locally to Visualize GoBilda Data) 7

Lab Assignment 8

Tasks 8

Submission 10

Introduction

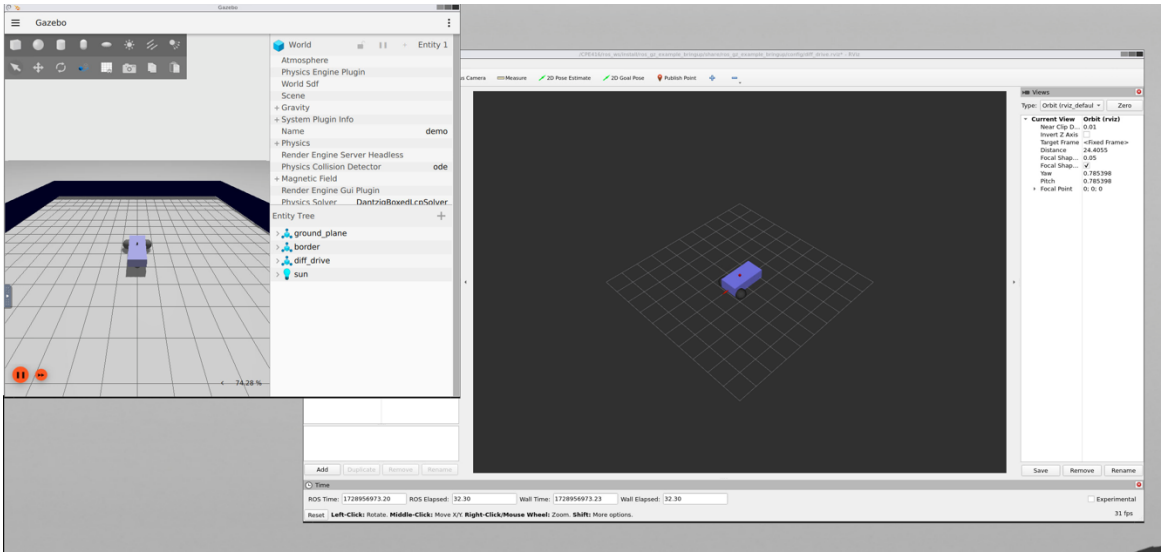
Objective

Objective:
This lab introduces the use of the Ignition Gazebo Sim simulator as a vital tool for developing and testing robotics software in a safe, controlled virtual environment. Before deploying code to a physical GoBilda robot, we will implement a classic "Bump and Go" behavior entirely in simulation.

- The core of this lab involves programming a Finite State Machine (FSM) that enables the robot to navigate by using data from a 2D LiDAR sensor.
- The robot will move forward until it detects an obstacle, then execute a backup and turn maneuver to avoid a collision.
- You will learn to parse the standardized LaserScan message in ROS2, interpret the robot's surroundings, and issue appropriate motor commands
- This exercise with simulation and your physical robotics, providing a practical framework for programming a simple autonomous behavior.

Background

Ignition Gazebo Sim



Testing your code on a physical robot is the ultimate way to assess its real-world performance. However, debugging on a physical platform isn't always ideal for several reasons: access to the robot may be limited due to costs, we might need to test in different environments, or there's a risk that faulty code could damage the robot. Fortunately, physics simulators provide an effective alternative, allowing us to simulate robot models, sensors, and environments to evaluate our code under various conditions. Gazebo Sim is the official simulator for the ROS2 community. While Gazebo works independently for simulating a variety of robots, it also offers a ROS API to connect with topics and nodes, enabling integration into the ROS ecosystem.

Note that there is deprecated version of Gazebo (called Gazebo Classic now) that was previously very common and is still used for certain projects because of the support. We will focus exclusively on Gazebo Sim in this class.

Simulation Description Files (SDF)

An SDF file is an XML-based format used to describe the properties of a robot, its components, and the environment in which it operates. It is commonly used in robotics simulators like Gazebo to define models, including robots, sensors, lights, and objects, as well as to specify their physical characteristics, joints, dynamics, and interactions. SDF files provide a structured way to store information about all aspects of a simulation, making it easier to recreate a consistent virtual environment for testing and development.

Which is the *sdf* file that describes the robot we are simulating. There is a separate *sdf* file that describes the *world* that the robot will be operating in.

Key tags of an SDF File:

- Model Definition: <model>: Describes the overall robot or object in the simulation. This section can contain nested components like links, joints, sensors, and more.

- Link: <link>: Represents a rigid body in the simulation (e.g., a part of a robot). A link could be a robot's wheel, arm, or chassis. It defines properties such as mass, inertia, visual appearance, and collision geometry.

- Joint: <joint>: Defines the connections between different links. A joint specifies how two parts of a robot move relative to each other, such as rotational or translational movement.

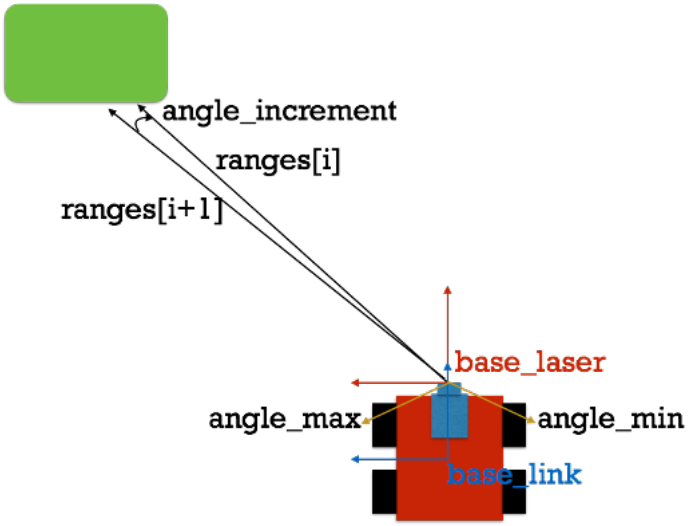
- Sensors: <sensor>: Describes sensors attached to the robot, such as cameras, LIDARs, or IMUs. It includes parameters like the sensor's type, position, and properties.

- Visual and Collision Elements:
    - <visual>: Specifies how a link or object appears in the simulation, such as its color, texture, and mesh file.
    - <collision>: Defines the geometry used for collision detection during the simulation. This could be simpler shapes (like boxes or spheres) or more complex meshes.

- World Properties: <world>: Defines the environment in which the robot operates, such as gravity, lighting, terrain, and other factors influencing the simulation.

An SDF file is loaded by a simulator (such as Gazebo), which interprets the XML data and creates a virtual environment or robot model based on the definitions within the file. The simulation runs by using the described models and their interactions (e.g., joint movements, sensor readings, and collisions).

2D LiDAR Technology



A 2D LiDAR (Light Detection and Ranging) sensor measures distances to surrounding objects by emitting laser pulses in a horizontal plane and detecting the time it takes for the light to bounce back. The sensor rapidly rotates a laser beam—typically using a mirror or rotating head—to scan the environment in a limited arc (field of view).

By calculating the time of flight or phase shift of the reflected light, the LiDAR determines the distance to each point. Combining these measurements yields a 2D map of ranges versus angles, which represents the nearby environment's shape and structure in a single horizontal slice.

In robotics, a 2D LiDAR (Light Detection and Ranging) is commonly mounted at a fixed height on a mobile robot to perceive walls, obstacles, and open spaces. The resulting **scan data**, often in the form of polar coordinates (r, θ), can be converted into Cartesian coordinates for mapping and localization tasks. Because LiDAR operates using light, it provides high accuracy and resolution over a broad range—often up to tens of meters—though its performance can be affected by surface reflectivity, lighting conditions, and environmental factors like dust or fog.

ROS2: *sensor_msgs/msg/LaserScan*

In ROS2, the sensor_msgs/msg/LaserScan message type is the standard format for representing 2D LiDAR data. Each message corresponds to a single full scan from the LiDAR and contains an array of range measurements that describe distances to obstacles around the sensor.

Compact Message Definition

```
std_msgs/msg/Header header
float angle_min
float angle_max
float angle_increment
float time_increment
float scan_time
float range_min
float range_max
float[] ranges
float[] intensities
```

The message type also includes important data such as:

- *angle_min (float32)*: The start angle of the *scan* (radians)
- *angle_max (float32)*: The end angle of the *scan* (radians)
- *angle_increment (float32)*: The angular distance between single beams of the scan (radians)
- *ranges (float32[])*: the actual scan data, one index per beam.
- *scan_time (float32)*: the time between scans (seconds)

This data structure is extremely useful because it standardizes how different LiDAR devices communicate scan data, making it easier to integrate with mapping, localization, and navigation algorithms in ROS2. For example, the SLAM Toolbox or Nav2 stack can subscribe directly to a LaserScan topic to build occupancy grids or detect obstacles.

Bring up

Diff Drive Simulation Bring up

To bring up the Gazebo Simulation of our robots, you will need to follow in the instructions provided in the *gobilda_sim* repo.

GoBilda Hardware Bring up

Follow the instructions provided in the *gobilda_driver* starting from the section: ***Compiling the Driver***

That will get you started with the software to command the robot via velocities and read scan data from a topic.

Foxglove Visualization (Running Locally to Visualize GoBilda Data)

Check out the official Foxglove documentation to get started.
Foxglove is a powerful, cross-platform visualization tool that we'll use throughout this course to better understand and interpret data from our sensors and robots. One of the key advantages of

Foxglove is that it can be installed locally on your laptop and run independently — no ROS 2 installation is required on your machine.

Before downloading the desktop application, you'll need to create a free Foxglove account. Once you're in, pay special attention to the Overview and Frameworks → ROS 2 sections of the documentation, as they'll give you a solid foundation for integrating Foxglove with the tools we'll be using.

(Finally, note that the ros-humble-foxglove-bridge package should have been installed on the Orin when you installed dependencies using the rosdep commands.)

Lab Assignment

Bump and Go Roomba

The Bump and Go behavior uses the robot's 2D LiDAR sensor to detect nearby obstacles in front of the robot. The behavior I'd like you to program can be specified as follows:

- The robot moves forward at constant speed,
- When it detects an obstacle 1m in front:
  - it first moves backward. The robot should move backwards for either 1 second or until it detects an obstacle behind it (1m).
  - then turn in place until a *clear space* in front of it is detected. We can define clear space as at least 4m meters.
- If at any point the laser scanner fails, the robot stops completely. In this case *fails* means that the laser is not publishing *any* data on the topic.

Although the behavior is simple, proper design patterns are recommended since our code, can start to fail as we solve subsequent problems that may arise. In this case, we will use a Finite State Machine (FSM).

Tasks

1. Finite State Machine (FSM) -- (10 points)

Design an FSM for the GoBilda robot that achieves the behavior above. Include at least these states: forward, back, turn, stop

Describe the transitions between the states (what events/sensor readings trigger each change.)

2. Align the Simulated LiDAR – (20 points)

Update the simulated LiDAR sensor pose to match the physical LiDAR on your robot.

File to edit:

gobilda_sim/ros_ws/src/ros_gz_project/ros_gz_example_description/diff_drive/model.sdf

Look for the following tag line: *<link name="lidar_link">*

Note: syntax errors in the *model.sdf* file can prevent the node from running.

3. Implement the FSM on the Real Robot (30 points)

Write a ROS2 Node that runs your FSM on the physical GoBilda robot.

Executable Name: *bump_and_go*

Safety first: Remember to test in the simulator before running on the real robot to reduce risk of accidents.

For full credit:

- The robot should only start its back-up and spin when it detects an obstacle *in front* of the robot
- The robot should not hit anything while it's backing up

Tips:

Remember, the Lidar sensor returns data in its own frame; how do you calculate the information in the robot's frame?

How do you read a LaserScan message to only select the beams in front of the robot? What about behind the robot?

Submission

You are expected to demo your code in the next lab section. Make sure that I get your name down after your successful demo.

After you've demo'ed your code and passed, I will need you to submit your source code. Compress your entire ROS2 workspace and upload the files to canvas.