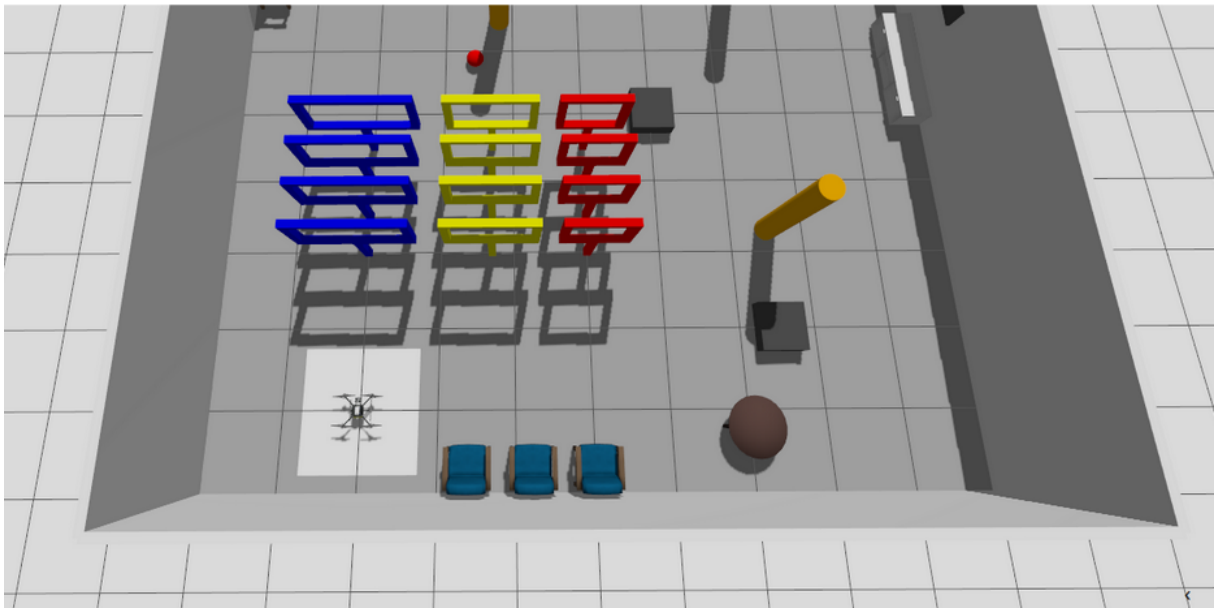

Gazebo and ROS2 for Robotics and Autonomous Vehicles

Daniel Aláez Gómez
Area of Languages and Computer Systems
Department of Statistics, Computer Science and Mathematics
Public University of Navarra

Abstract

This course provides a practical and comprehensive introduction to Gazebo and ROS2 as advanced simulation tools and environments for robotics and autonomous vehicles. It is designed for technical professionals with knowledge of digitalization, though not necessarily specialized in robotics, who wish to develop skills in open-source technologies and state-of-the-art simulation. The main objectives are to:

- Understand Gazebo and ROS2 as advanced tools and environments for robotics and autonomous vehicle simulation.
- Learn to create worlds and models of autonomous vehicles using standard specifications (SDF, URDF).
- Learn to integrate sensors (both virtual and real) and communicate them with ROS2.
- Launch simulations of autonomous vehicles such as rovers and drones controlled by PX4.
- Carry out practical examples with autonomous vehicles in realistic scenarios and simulated vehicles.

Through hands-on exercises and guided projects, participants will gain the necessary skills to simulate, test, and validate robotic systems and autonomous vehicles using open-source technologies, enabling rapid prototyping and validation before deploying solutions to real-world environments. This course is part of the TWiN Navarra project framework, focusing on digital twins.

Contents

1 Part 1: Theory & Examples

1.1 Introduction to Gazebo and ROS2

Gazebo and ROS2 are two of the most influential and powerful tools in the field of robotics, automation, and autonomous vehicle simulation. Their integration forms a robust foundation for experimenting, validating, and accelerating the development of applications in both industrial and research environments.

1.1.1 What is Gazebo?

Gazebo is an open-source simulator that enables the creation of complex virtual environments in which to test and validate robotic systems. Initially developed by Open Robotics, Gazebo has established itself as the standard in simulation thanks to its ability to represent realistic physics, multiple types of sensors, object interactions, and extensible environments. Over the years, it has evolved substantially: from its early versions to the current Gazebo Harmonic, passing through Ignition Gazebo and integrating improvements in visualization, physics, and scalability.

Main advantages:

- Enables realistic simulation of robots, vehicles, sensors, and environmental elements.
- Facilitates rapid prototyping and algorithm testing without risks or material costs.
- Supports integration with other tools (especially ROS/ROS2) for advanced and automated simulations.
- Highly customizable and features a large library of pre-existing models and worlds.

1.1.2 Introduction to ROS2

ROS (Robot Operating System) has become the largest open-source ecosystem for professional and educational robotics. Unlike other platforms, ROS is not an operating system per se, but rather a software layer that facilitates communication, modularization, and component reuse for robots of any type.

ROS2 represents the natural evolution of the first generation (ROS1), improving key aspects: real-time support, cross-platform communication, and robustness. ROS2 is designed to meet the current needs of industry and research in robotics, such as interoperability, security, and adaptability to all types of hardware.

Components and architecture:

- Modularity based on nodes that communicate with each other through messages (topics), services, parameters, and actions.
- Extensive ecosystem: controllers, simulators, perception and planning algorithms, packages for all types of hardware.
- Fostered and supported by a global community, universities, and major companies (Amazon, Intel, Open Robotics, etc.).
- Compatible with cloud and edge environments, as well as different operating systems (Linux, Windows, Mac).

1.1.3 Interoperability between Gazebo and ROS2: Use Cases

The joint work of Gazebo and ROS2 is key in modern development of robotics and autonomous vehicles. Thanks to their integration:

- It is possible to develop controllers and algorithms in ROS2 and test them directly in realistic virtual scenarios with Gazebo.
- Simulated sensors can emit data that flows as if they were real sensors, feeding perception, navigation, or AI algorithms.
- Users can create and modify custom virtual robots and see, in real time, how their behavior varies with changes in code or environment.

This allows reducing costs, errors, and accelerating validation before deploying systems to real environments, a critical aspect in industry and R&D.

1.2 Generation of Vehicles and Worlds with SDF and URDF

The creation of detailed models and scenarios is the foundation of advanced simulation in robotics. Gazebo and ROS2 use two standard languages to describe robots and worlds: **SDF** and **URDF**.

1.2.1 SDF Concepts (Simulation Description Format)

SDF (Simulation Description Format) is an XML-based file format developed specifically to describe everything that can exist in a simulated Gazebo environment: robots, vehicles, sensors, lights, landscapes, floors, etc. SDF is highly flexible and allows defining physical, visual, dynamic properties and complex interactions. It is the native format for Gazebo worlds and models.

- Allows defining multiple robots/environments simultaneously.
- Manages physical aspects (mass, friction, joints), visual (textures, materials) and functional (sensors, actuators) with a high level of detail.
- Facilitates reuse and extension of models for different uses and scenarios.

1.2.2 URDF Concepts (Unified Robot Description Format)

URDF is a format, also based on XML, oriented to describe mainly robots (their structure, joints, sensors and actuators). It is widely used in the ROS ecosystem, especially in the definition of robotic arms, drones, rovers and mobile manipulators.

- Its main objective is to define the geometry, kinematics and structure of a robot.
- It is simpler than SDF, but perfectly compatible through conversion tools (it is common to work with both standards in ROS2-Gazebo projects).
- Allows importing robots from existing libraries, or developing custom models and visualizing them simultaneously in simulation and visualizers like RViz.

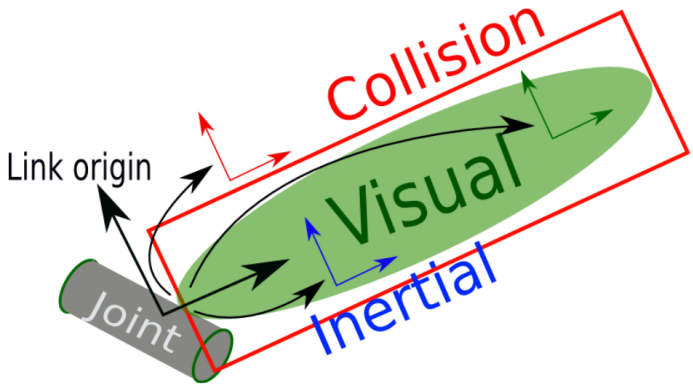

Figure 1: Esquema de *links* y *joints* con sus partes principales.

Links and Joints in URDF/SDF The definition of a robot through URDF/SDF is based on two key concepts:

- **Links:** correspond to the "rigid parts" of the robot (for example: chassis, wheels, arms, sensors), each with its physical and visual properties.
- **Joints:** articulations that connect the links and define how they move relative to each other (example: a wheel rotating relative to the chassis, a steering axis, etc.).

Basic example of a differential-type robot (URDF):

```
<robot name="mi_robot">
  <link name="base_link">
    <visual>
      <geometry><box size="0.4␣0.3␣0.1"/></geometry>
      <material name="azul"><color rgba="0␣0␣1␣1"/></material>
    </visual>
    <collision>
      <geometry><box size="0.4␣0.3␣0.1"/></geometry>
    </collision>
    <inertial>
      <mass value="8.0"/>
      <inertia ixx="0.1" iyy="0.1" izz="0.2"/>
    </inertial>
  </link>

  <link name="wheel_left">
    <visual>
      <geometry><cylinder length="0.05" radius="0.06"/></geometry>
      <material name="negro"><color rgba="0␣0␣0␣1"/></material>
    </visual>
  </link>

  <link name="wheel_right">
    <visual>
      <geometry><cylinder length="0.05" radius="0.06"/></geometry>
      <material name="negro"/>
    </visual>
  </link>
```

```
  <joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="wheel_left"/>
    <origin xyz="-0.18 0.13 0.03"/>
    <axis xyz="0 1 0"/>
  </joint>

  <joint name="right_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="wheel_right"/>
    <origin xyz="-0.18 -0.13 0.03"/>
    <axis xyz="0 1 0"/>
  </joint>
</robot>
```

In this example, a chassis (base_link) and two wheels connected by joints are defined. Each part can be customized in materials, masses, colors and dimensions.

Basic example in SDF:

```
<model name="mi_robot_sdf">
  <link name="base">
    ...
  </link>
  <link name="rueda_izquierda">
    ...
  </link>
  <joint name="union_rueda_izq" type="revolute">
    <parent>base</parent>
    <child>rueda_izquierda</child>
    <axis>
      <xyz>0 1 0</xyz>
      <limit>
        <lower>-1e16</lower>
        <upper>1e16</upper>
      </limit>
    </axis>
  </joint>
  ...
</model>
```

The syntax changes but the logic is the same: links (rigid parts) and joints (articulations/movements).

1.2.3 Creation of Autonomous Vehicle Models

The process starts from defining the vehicle structure (chassis, wheels, suspension...), adding virtual components (sensors, control modules) and establishing physical and kinematic constraints. Clarity in defining the model allows obtaining realistic and reusable simulations, both in own and collaborative projects.

The modularity of URDF/SDF allows building complex vehicles from simple parts, fostering reuse and scalable design. For example, a simulated drone consists of body, propellers, sensors and cameras, all configured as links and connected by joints.

1.2.4 Design and Customization of Virtual Worlds in Gazebo

A virtual world defines not only the test terrain or road, but all the elements with which the robot or vehicle will interact: obstacles, traffic areas, traffic, signage, virtual people, environmental

conditions...

- Worlds can be realistic (replicas of existing environments) or synthetic (controlled scenarios for specific tests).
- Textures, 3D objects, furniture elements can be loaded, and physics can be customized (gravity, fluid dynamics, etc.).

Basic example of a Gazebo world file (SDF):

```
<sdf version="1.6">
  <world name="entorno_pruebas">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://sun</uri>
    </include>
    <model name="cubo_obstaculo">
      <pose>1 1 0.5 0 0 0</pose>
      <link name="link">
        <visual name="visual">
          <geometry>
            <box>
              <size>1 1 1</size>
            </box>
          </geometry>
          <material>
            <ambient>1 0 0 1</ambient>
          </material>
        </visual>
        <collision name="collision">
          <geometry>
            <box><size>1 1 1</size></box>
          </geometry>
        </collision>
      </link>
    </model>
    <!-- Change world gravity: -->
    <gravity>0 0 -9.8</gravity>
    <!-- Add more elements, lights, environmental sensors, etc -->
  </world>
</sdf>
```

In this example, we create a world with a ground plane, a solar light source and a cube-type obstacle. All this can be extended with buildings, roads, pedestrians, weather conditions, etc.

To open a basic world like the example in Gazebo, simply execute the following command in the terminal, indicating the path to your `.world` file (for example, `entorno_pruebas.world`):

```
gazebo entorno_pruebas.world
```

If you use Gazebo for ROS 2 (Ignition Gazebo or Gazebo Fortress/Humble and successors):

```
ign gazebo entorno_pruebas.sdf
```

This will open the Gazebo graphical interface and load the defined environment, ready for simulation and customization.

Detailed customization of worlds allows simulating from completely controlled environments to replicas of real environments, ideal for testing navigation and perception algorithms under diverse conditions.

1.2.5 Adaptation and Reuse of Models for Different Scenarios

SDF/URDF models are designed to be modular and reusable: the same vehicle can be used in different environments or, with slight modifications, serve for other simulations, experiments or demonstrators.

Inside the `<world>` tags you can include any model located within Gazebo's servers or your local machine:

```
<include>
<name>Coke1</name>
        <pose>0 0.1 0 0 0 0</pose>
<uri>https://fuel.gazebosim.org/1.0/
OpenRobotics/models/Coke</uri>
</include>
```

This saves time in development, promotes collaboration and the adoption of open-source standards in the ecosystem of robotics applied to autonomous vehicles.

1.3 Sensor Integration and Communication with ROS2

Proper sensor integration is fundamental in robotics and autonomous vehicles, as sensors enable robots to perceive the environment, make decisions, and execute actions efficiently and safely. Gazebo and ROS2 provide a robust framework for simulation, integration, and management of different types of sensors, as well as for real-time information flow between the simulated environment and control algorithms.

1.3.1 Types of Compatible Sensors (Virtual and Real)

In robotics, sensors enable robots to capture information about their own state and the surrounding environment. There are many types of sensors, among which the following stand out:

- **Position and orientation sensors:** Encoders, IMU (Inertial Measurement Unit), GPS.
- **Distance sensors:** LiDAR, ultrasonic, infrared, radar.
- **Vision sensors:** RGB cameras, depth cameras, stereo cameras.
- **Contact and force sensors:** Tactile sensors, load cells.

Gazebo allows simulating all these sensors realistically, emulating both their physical operation and their possible errors, noise, or limitations. This is key for developing robust tests before deploying solutions to real hardware.

Furthermore, ROS2 supports connection with real physical sensors, allowing the development and integration of solutions that can later work in the real world with minimal modifications.

1.3.2 Sensor Configuration in the Simulator

Configuring sensors in Gazebo involves:

- Defining them in the robot or scenario description (SDF or URDF), specifying their type, parameters, and location.

- Adjusting their technical characteristics: detection range, sampling frequency, accuracy, aperture angle, etc.
- Associating simulation plugins, which are responsible for emulating their behavior and generating data messages.

Practical Example: Minimal Robot with LiDAR Sensor and ROS2 The following shows how to create a very simple robot with a LiDAR sensor using URDF and Gazebo plugins. The step-by-step process for launching the simulation and visualizing data in real time is included:

1. URDF Definition with Sensor and Plugin

Save the following content in a file called `robot_lidar.urdf`:

```
<robot name="robot_lidar">
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.2␣0.2␣0.1" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <box size="0.2␣0.2␣0.1" />
      </geometry>
    </collision>
    <inertial>
      <mass value="1.0"/>
      <origin xyz="0␣0␣0" />
      <inertia ixx="0.01" iyy="0.01" izz="0.02"
      ixy="0.0" ixz="0.0" iyz="0.0"/>
    </inertial>
  </link>

  <!-- LIDAR SENSOR: Hokuyo plugin for Gazebo specific to ROS2 -->
  <gazebo>
    <plugin name="gazebo_ros_head_hokuyo_laser"
    filename="libgazebo_ros_ray_sensor.so">
      <robotNamespace>/</robotNamespace>
      <frameName>base_link</frameName>
      <topicName>scan</topicName>
      <updateRate>5</updateRate>
      <ray>
        <scan>
          <horizontal>
            <samples>360</samples>
            <resolution>1</resolution>
            <min_angle>-1.57</min_angle>
            <max_angle>1.57</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.12</min>
          <max>3.5</max>
        </range>
        <noise>
          <type>gaussian</type>
```

```
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
    </plugin>
  </gazebo>
</robot>
```

This defines a simple robot with a LiDAR sensor. The plugin `libgazebo_ros_ray_sensor.so` publishes the topic `/scan` (type `sensor_msgs/LaserScan`).

2. Launch the Robot in Gazebo (using ROS2)

Place the URDF file where your workspace can find it and launch:

```
# Terminal 1: Launch Gazebo with ROS2
gazebo --verbose
# or:
ros2 launch gazebo_ros gazebo.launch.py
```

In another terminal, launch the robot spawn:

```
ros2 run gazebo_ros spawn_entity.py -entity robot_lidar
-file /ruta/completa/robot_lidar.urdf
```

3. Visualize Sensor Data from Terminal

With the simulation running, open another terminal and check the topic published by the LiDAR:

```
ros2 topic list
# You should see /scan among the listed topics

ros2 topic echo /scan
# You will see LaserScan type messages showing simulated
# distances in the environment
```

4. Visualize the Sensor and Environment in RViz2

Open RViz2:

```
rviz2
```

- Add the "LaserScan" display.
- Set the topic to `/scan` and the "fixed frame" to `base_link`.
- You should see the LiDAR data drawn as points or lines.

Notes:

- You can adjust sensor parameters (range, frequency...) by editing the plugin in the URDF.
- If you want to try with a camera, change the plugin to `libgazebo_ros_camera.so` and the topic would be `/camera/image_raw`.

This practical case illustrates the complete flow from URDF definition, sensor inclusion, and visualization both from terminal and in RViz2, using Gazebo's plugin infrastructure for ROS2.

The ability to adjust parameters such as noise or physical limitations is very useful for creating realistic and relevant simulations, where algorithms must be robust against adverse conditions typical of the real world.

1.3.3 Data Communication and Topics in ROS2

One of the fundamentals of ROS2 is its message-based communication system. Each sensor, whether simulated or real, generates information that is sent through **topics** (communication channels). Other nodes can subscribe to these topics to process or visualize the data.

- The data generated by sensors can contain images, point clouds (3D), distance readings, accelerations, positions, etc.
- These messages follow standards defined by the ROS community (for example, `sensor_msgs/Image` for images, `sensor_msgs/LaserScan` for LiDAR, etc.). See link: https://docs.ros2.org/foxy/api/sensor_msgs/index-msg.html
- The ROS2 publish/subscribe system allows this information to be distributed, processed, and reused by different parts of the robotic system simultaneously.

This favors modularity and flexible system design, being able to replace, update, or add sensors without needing to redo the main control software.

1.3.4 Synchronization and Sensor-ROS2 Data Flows

In real and simulated robotic applications, **synchronization** of information is key: data from different sensors are usually used together (for example, sensor fusion for localization or advanced perception).

- ROS2 provides mechanisms to synchronize data from different sources, adjusting arrival times and correlating information.
- Publication frequency and reception delay can be configured, and there are specific tools for debugging and visualizing data flows.

Well-managed data flows enable building reliable applications in localization, navigation, mapping, advanced perception, and decision-making in autonomous vehicles.

Sensor integration and communication in Gazebo and ROS2 are the gateway to realistic experiments and the development of solutions transferable to the real world. Deepening these fundamentals is essential to advance in the design of intelligent robots and autonomous vehicles.

1.3.5 Visualizing and Publishing Topics with rqt

rqt is a modular graphical tool based on the Qt library, designed to facilitate visualization, analysis, and debugging of robotic systems built with ROS 2. Its main function is to offer a visual interface that allows real-time inspection of the information published and subscribed by system nodes, as well as parameters, transforms, and logging messages, all without relying exclusively on command-line tools.

rqt is composed of a collection of plugins, each focused on a specific aspect of the system. Some of the most commonly used plugins in ROS 2 include:

- **rqt_graph**: Displays an interactive graph representing active nodes and topics, enabling users to visualize how different system components communicate.
- **rqt_plot**: Allows real-time plotting of values published on a topic, which is useful for analyzing sensor data or control variables.

- **rqt_console**: Shows logging messages generated by nodes (debug, info, warn, error levels), facilitating debugging.
- **rqt_tf_tree**: Represents the hierarchy of transformations (tf) between the robot's reference frames.
- **rqt_image_view**: Visualizes images published on topics of type `sensor_msgs/Image`, such as those from real or simulated robot cameras.
- **rqt_reconfigure** (or its counterparts in ROS 2): Allows dynamic adjustment of parameters for nodes running in the system.

When using Gazebo (or Ignition Gazebo) with ROS 2, the simulator publishes information about the simulated world (positions, sensors, cameras, etc.) as ROS 2 topics. **rqt** can subscribe to these topics to visualize and analyze the data. For example, you can use `rqt_image_view` to observe the image generated by a simulated robot camera in Gazebo, `rqt_plot` to chart real-time LiDAR sensor readings, or `rqt_graph` to examine how control and simulation nodes communicate.

To launch **rqt** in ROS 2, simply use the following command:

```
rqt
```

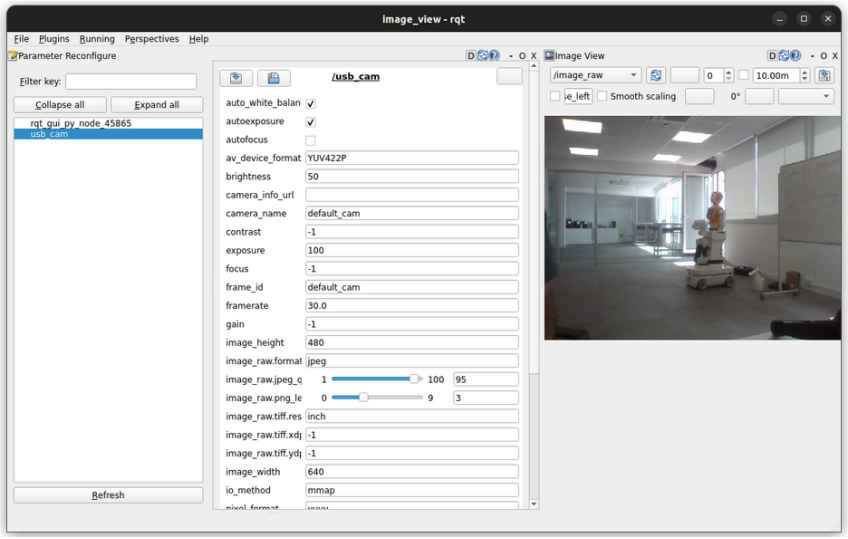

Figure 2: Interface principal de rqt mostrando la visualización de nodos, tópicos y herramientas de análisis en ROS2.

1.3.6 Recording Repeatable Simulations with rosbag

One of the most powerful tools in ROS 2 is the ability to record (log) messages flowing through topics and later replay them for analysis, debugging, or sharing experiments. The command-line tool for this is `ros2 bag`. This functionality allows you to capture published sensor data, control commands, internal states, or entire simulations and save them to a file or database for future use. For more details, refer to the official ROS documentation.

Why is this important?

- **Experiment Repeatability:** Enables you to replay exactly the same messages that were published during a particular run, making it easier to replicate tests or simulations.
- **Debugging Made Easy:** If a system behaves unexpectedly, the message flow can be recorded and thoroughly reviewed.
- **Collaboration and Data Sharing:** Anyone can record a session, send it to another person, and that person can replay it in their own environment to see exactly the same data.

Basic Steps

1. **Preparation:** Launch the ROS 2 nodes that publish the messages of interest. For example, in a Turtlesim simulation, start the visualizer node and the teleoperation node.
2. **Choose Topics:** Use `ros2 topic list` to see the active topics and decide which ones to include in the recording.
3. **Record with ros2 bag record:** For example:

   ```
   ros2 bag record /turtle1/cmd_vel
   ```

   This command creates a bag file or database folder containing messages from the topic `/turtle1/cmd_vel`. You can record multiple topics at once, and use the `-o <name>` option to set the bag name. There is also a `-a` option to record **all** topics.

Inspecting Recordings You can check info about a recorded bag using:

```
ros2 bag info <bag_file>
```

For example:

```
ros2 bag info subset
```

This command will show how many messages from each topic were recorded, the duration of the bag, the size on disk, and more.

Replaying Data To replay recorded messages as they were originally published, use:

```
ros2 bag play <bag_file>
```

For example:

```
ros2 bag play subset
```

In a Turtlesim simulation, you would see the turtle follow the exact path drawn during the recording.

Considerations and Best Practices

- Execute the recording command from the directory where you want the output, as `ros2 bag record` will create the database folder or file there.
- Recording only the necessary topics reduces file size and makes data management easier.
- Checking the publication frequency of topics can help interpret the recording (for example, comparing the number of messages for `/turtle1/pose` vs `/turtle1/cmd_vel`).

- When replaying, make sure nodes are active and subscribed to the same topics; otherwise, they won't receive the messages.
- To use a bag for simulation (e.g., with Gazebo or a real robot), it is useful to also record sensor states, commands, and feedback messages, enabling full analysis of the event chain.

1.4 Control of Autonomous Vehicles with PX4 and Simulation

The control of autonomous vehicles requires coordinating both hardware (sensors, actuators) and high-level software (perception and planning algorithms) and real-time firmware. The **PX4** firmware is one of the international standards for the control of drones and rovers, widely adopted in robotics, automotive and aerospace sectors, particularly in the field of research.

1.4.1 Introduction to PX4 Firmware

PX4 is an open-source firmware for flight controllers of autonomous systems, such as drones, rovers and submarines. It runs on embedded control boards and directly manages vehicle actuators and sensors, providing flight, navigation, failsafe and telemetry functions.

- Support for multiple vehicle types (multirotor, fixed wing, rover, VTOL...)
- Communicates via the MAVLink protocol with control stations, ROS2, simulators and mission applications.
- Allows manual, semi-autonomous and fully autonomous control.

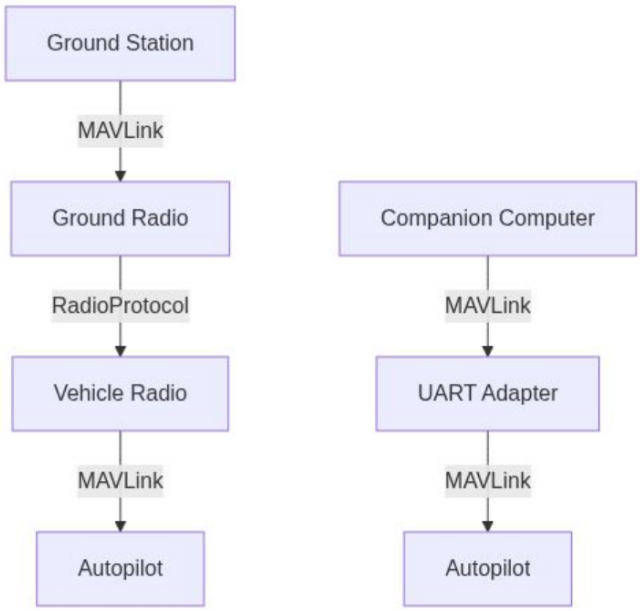

Figure 3: The two main control schemes for autonomous vehicles: (1) Ground Control Station (GCS) and (2) Companion Computer.

When developing autonomous robotic applications, we typically encounter two main control schemes:

1. **Ground Control Station (GCS):** This is the most common scheme. It involves a ground control station connected to a transmitter, which converts operator commands into MAVLink messages. These messages are transmitted via radio waves to a receiver installed on the vehicle, where the messages are relayed to the autopilot to execute commands and/or send information back to the ground control station.

2. **Companion Computer:** This is a computer that replaces the ground control station and is often installed on the autonomous vehicle itself. The companion computer is responsible for combining information from multiple sensors and feedback from the controller to make decisions regarding the behavior or control of the autopilot, to which it connects via cable to exchange messages using MAVLink or other protocols.

1.4.2 Integration of PX4 SITL (Software In The Loop) with Gazebo

PX4 offers SITL (Software In The Loop) functionality, which allows running the complete firmware on a computer, simulating the real electronics, and connecting with Gazebo to experiment without the need for physical hardware.

The typical flow for development and testing is:

1. Compile/launch PX4 in SITL simulator mode (without real flight board)
2. Launch Gazebo with the desired vehicle model and environment
3. Connect Gazebo (generating physics and sensors) with PX4 SITL via the **MAVLink** protocol
4. Allow control from the control station (QGroundControl, ROS2 commands or autopilot scripts)

Basic Scheme:

```
ROS2 ↔ GAZEBO ↔ PX4 SITL ↔ QGroundControl / Autonomous commands
```

1.4.3 ROS2-PX4 Integration Architecture

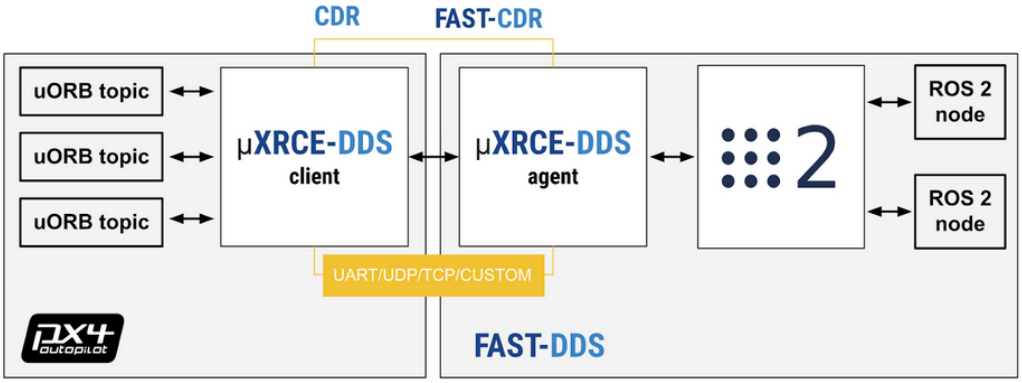

Figure 4: Diagrama de la arquitectura de integración ROS2-PX4 mediante Micro XRCE-DDS Agent.

Communication between ROS2 and PX4 is one of the pillars for developing high-level robotic applications on PX4-controlled platforms, whether through simulation or on real hardware. This architecture allows ROS2 nodes to access, process and generate commands for the vehicle through a standardized message flow.

Key Components of the Architecture:

- **uORB:** It is PX4's internal messaging system, responsible for communicating internal firmware modules. For example, control modules read and publish uORB messages to exchange information about sensors, states and internal commands.
- **ROS2:** It is the middleware that manages communication between user nodes, perception, planning and high-level control algorithms, using the topics, services and actions paradigm.
- **Micro XRCE-DDS Agent (MicroXRCEAgent):** This agent acts as a bridge between PX4 (and its uORB messages) and the ROS2 ecosystem. It translates and publishes relevant autopilot topics in ROS2 topics and vice versa, using the DDS standard required by ROS2. It is especially designed for embedded devices with limited resources and replaces tools like micrortps_agent, being the currently recommended option in the official PX4 documentation.

Message Flow Scheme:

```
Physical Sensors/Actuators
        ↑↓
     [PX4 Firmware]
        ↑↓  (uORB: internal PX4 messages)
     [MicroXRCE-DDS Agent]
        ↑↓  (uXRCE-DDS: serial/UDP, RTPS)
     [ROS 2 Nodes]
        ↑↓  (DDS: topics/services/actions)
     [User applications, visualization, control]
```

- Relevant PX4 messages (uORB topics) are sent externally via MicroXRCEAgent, which uses the Micro uXRCE-DDS protocol.
- This agent publishes and subscribes data in standard ROS2 topics, making it possible for any ROS2 node to interact with the PX4 system.

Basic Installation of the PX4-ROS2 Bridge

1. **Clone and install ROS2, PX4 and dependencies for the MicroXRCE bridge:**
   - Install ROS2 Foxy, Humble or supported distribution (according to your system and PX4).
   - Install necessary dependencies for PX4 tools.
2. **Build PX4 with MicroXRCE support:**
   - Compile PX4 in SITL mode with MicroXRCE support:

     ```
     cd ~/src/Firmware
     make px4_sitl_default
     ```

3. **Launch the Micro XRCE-DDS Agent and ROS2 nodes:**
   - In a terminal, run the agent:

     ```
     MicroXRCEAgent udp4 -p 8888
     ```

     You can also use serial mode if you connect a physical autopilot via USB/serial, but UDP is the typical method in SITL simulation.
   - In another terminal, launch PX4 SITL or connect your real autopilot.
4. **Topic publication and subscription:**
   - ROS2 nodes can now access messages published by PX4, such as sensors (IMU, GPS), vehicle state, control commands, etc. Similarly, you can publish commands from ROS2 to PX4 (for example, to perform offboard control).

Offboard Control from ROS2 Offboard control allows sending high-level navigation and movement commands from a ROS2 node, shifting control from the firmware to custom algorithms.

- The *OFFBOARD* mode must be enabled in PX4 (via service/flight mode).
- For example, you can make position, velocity or attitude commands from a Python/C++ node that publishes the appropriate messages to the RTPS topics "bridged" by the agent.
- Example of useful tools and packages: `px4_msgs`, `ros2_control`, `offboard_control_py`.

Useful Resources:

- https://docs.px4.io/main/en/ros2/user_guide#ros2-launch - ROS2-PX4 Integration Guide (PX4 Docs)
- https://github.com/PX4/px4_msgs - px4_msgs repository
- https://docs.px4.io/main/en/ros2/ros2_offboard_control.html - Offboard control example in ROS2

With this architecture, it is possible to simulate, develop and validate advanced control, perception and autonomy algorithms in ROS2 and then transfer them to real hardware with minimal adaptation.

1.4.4 Launch and Control of Rover and Drone Simulations

Practical Example: Quadrotor Drone Simulation

1. **Prepare PX4 and Gazebo**
   - Install PX4 Firmware and Gazebo (you can use containers or install from source/package)
   - Enter the PX4 firmware directory:

     ```
     cd ~/src/Firmware  # or where you have PX4 cloned
     make px4_sitl gazebo
     ```

   - This will launch Gazebo with a standard drone (for example, Iris) and the PX4 firmware connected.
2. **QGroundControl** (optional for visualization and control)

- Open QGroundControl (downloadable for Linux/Windows/Mac)
- It will automatically detect the simulated vehicle

3. **Arming, takeoff and basic control**

   - From QGroundControl you can "arm" motors, perform "takeoff", move the drone manually or programmatically.
   - A simple and direct way to control the simulation is using PX4's integrated commands (commander module) from the terminal where SITL is running (PX4 provides an interactive prompt in the console after launch):

     ```
     # To arm the motors
     commander arm

     # To perform automatic takeoff to a specific altitude
     # (for example, 3 meters)
     commander takeoff 3

     # To land
     commander land
     ```

   - There are also commands available to disarm, change flight mode, abort landing, etc. Check all possible commands with:

     ```
     commander help
     ```

   - These commands are especially useful for quick tests in simulation and for understanding PX4 firmware state flows, without needing to launch additional nodes in ROS2 or depend on external tools like MAVROS, whose compatibility and support may be limited in recent versions.

4. **Telemetry and monitoring**

   To visualize topics in ROS2 in a PX4 simulation, it is necessary to use the ROS2 bridge provided by `px4_ros_com` together with Micro XRCE-DDS Agent (`MicroXRCEAgent`). Before continuing, make sure you have correctly installed PX4 messages (`px4_msgs`) and the `px4_ros_com` package. This allows direct communication between PX4 and native ROS2 nodes without needing to use MAVROS.

   Once the environment is configured (PX4 with `px4_ros_com` and `MicroXRCEAgent` running), simulated sensors such as IMU, GPS, camera and LiDAR will publish information through native ROS2 topics, under the namespace `/fmu/out/`. Examples of useful topics:

   - Vehicle state:

     ```
     ros2 topic echo /fmu/out/vehicle_status
     ```

   - Global position information (GPS):

     ```
     ros2 topic echo /fmu/out/vehicle_gps_position
     ```

   - IMU:

     ```
     ros2 topic echo /fmu/out/sensor_combined
     ```

   - Additional information (for example, odometry, pose estimator, etc.) will also be available under `/fmu/out/`.

These topics can be visualized with tools like `ros2 topic echo`, or integrated into RViz2 and other ROS2 nodes for advanced real-time monitoring.

PX4 Rover Simulation

- Change the vehicle model (for example, to an Ackermann-type rover):

  ```
  make px4_sitl gz_rover_ackermann
  ```

- The flow is identical: you can teleoperate, send waypoints or automate trajectories.

1.4.5 Practical Scenarios: Automation and Teleoperation

The main use cases in PX4-Gazebo simulation include:

- **Manual teleoperation:** With physical controller (joystick/gamepad), QGroundControl or ROS2 commands (publishing on velocity/cmd topics).
- **Basic automation:** Sending mission commands directly from Python/C++ scripts using MAVSDK, or from ROS2 nodes (e.g. waypoint tracking, autonomous execution of a predetermined route, takeoff-landing, etc.)
- **Navigation and autonomy testing:** Integration of perception and localization algorithms from ROS2 on Gazebo sensor topics, sending movement commands to PX4.

Minimal example of automation from Python script using MAVSDK (takeoff and landing):

```
import asyncio
from mavsdk import System

async def run():
    drone = System()
    await drone.connect(system_address="udp://:14540")
    print("Waiting for connection...")
    async for state in drone.core.connection_state():
        if state.is_connected:
            print(f"Connected!")
            break
    print("Arming...")
    await drone.action.arm()
    print("Taking off...")
    await drone.action.takeoff()
    await asyncio.sleep(6)
    print("Landing...")
    await drone.action.land()
asyncio.run(run())
```

QGroundControl: Essential Tool for Configuration, Control and Mission Monitoring
QGroundControl is the recommended ground control station (GCS) for working with PX4, both in simulation and with real hardware. It offers an intuitive and cross-platform graphical interface (Windows, Linux, MacOS, Android/iOS) that covers the entire operational and technical workflow:

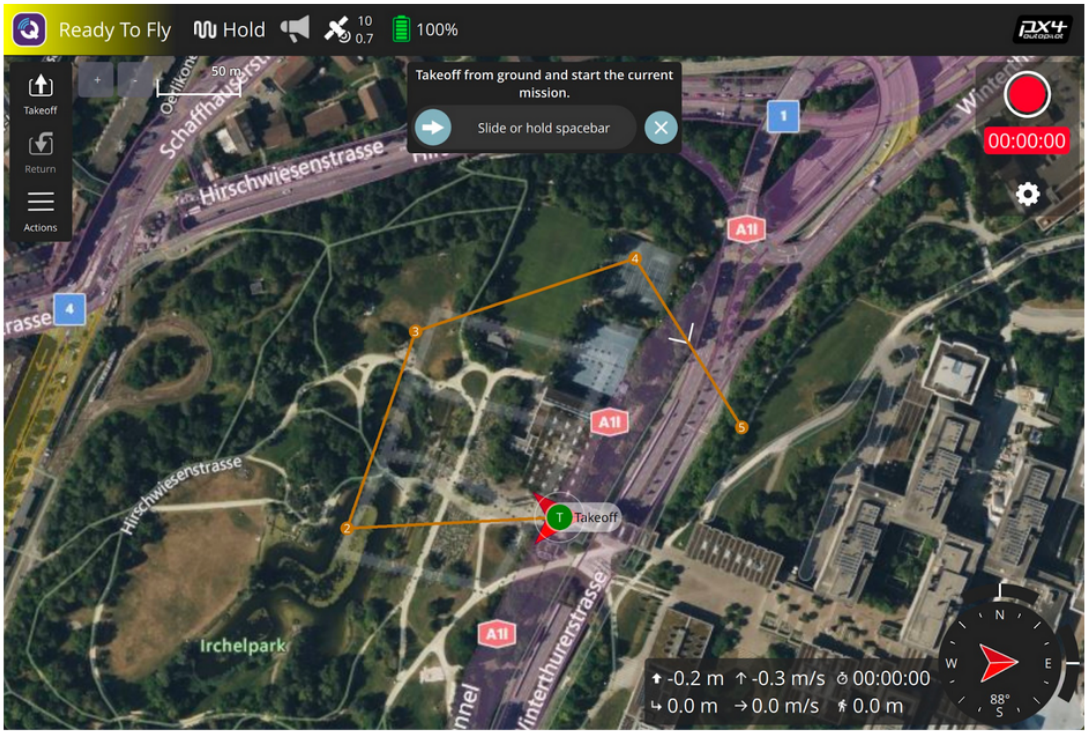

Figure 5: QGroundControl interface for configuration, planning and real-time monitoring in PX4 simulations and real vehicles.

Key Functionalities:

- **Initial configuration and vehicle adjustment:** Allows parameterizing the firmware, sensors (IMU, compass, GPS...), radio configuration, flight modes, failsafes and much more through visual assistants and user-friendly menus.
- **Autonomous mission planning:** Uses a "point & click" mission editor on the map, where you can define waypoints, specific commands (takeoff, land, survey, RTL, etc.) and constraints for flights or rover routes. It is possible to save, load and replicate missions easily.
- **Real-time telemetry:** Shows mission state (position, attitude, velocity, trajectory and objectives in real time), sensor readings, battery state and advanced telemetry. Allows viewing the vehicle on satellite maps, digital terrain, etc.
- **Manual control and supervision:** Direct control via joystick, gamepad or from the touch interface. Activates and changes between flight modes, arm/disarm, and critical commands.
- **Logging and analysis:** Facilitates downloading, visualization and analysis of flight/mission logs, helping in debugging and performance evaluation tasks.

Integration with Simulators and PX4 SITL

- Easily connects to simulations in PX4 SITL (such as those deployed in Gazebo) automatically detecting vehicles via MAVLink.

- Commands executed in QGroundControl (mission planning, arm/disarm, takeoff, landing...) are reflected in the simulated environment in real time, being very useful for practicing and testing without risks.
- Supports simultaneous connection with other ROS2/MAVSDK nodes, allowing combining automation flows and manual control.

Example of Typical Workflow

1. **Start PX4 SITL and the simulator (e.g. Gazebo).**
2. **Launch QGroundControl on your PC:** It will automatically detect the virtual vehicle.
3. **Perform the configuration you need** (parameters, sensors, flight modes).
4. **Design and load an autonomous mission** or use manual control.
5. **Monitor the simulation:** Visualize various data in real time and react to events (failsafes, signal losses, etc.).
6. **Analyze the logs** downloaded from QGC after each mission.

QGroundControl is, therefore, the essential tool both in testing phases and in real operations, allowing to close the complete cycle: configuration → planning → execution → analysis.

Download link and documentation: https://docs.qgroundcontrol.com

Key Notes:

- SITL integration allows developing, debugging and testing everything without risks or physical hardware.
- All mission elements can be monitored and recorded for later analysis.
- Most examples and practices can be done with both drones and rovers (changing model and command type).
- It is possible to combine real and simulated sensor data for advanced testing (hardware-in-the-loop, HIL).

Useful References:

- Official PX4 documentation: https://docs.px4.io/main/en/simulation/gazebo.html
- MAVSDK Python: https://mavsdk.mavlink.io/main/en/python

This section establishes the technical and practical foundations for professional and research work with robots and autonomous vehicles using advanced simulation, allowing experimentation without risks and facilitating direct transfer to real hardware when necessary.

1.5 Introduction to Isaac Sim, Isaac Lab and Omniverse for AI-Based Simulation

In recent years, simulation for robotics has evolved beyond testing traditional control algorithms, becoming a key tool for training and validating systems based on Artificial Intelligence (AI), deep learning and reinforcement learning. NVIDIA's tools—Isaac Sim, Isaac Lab and Omniverse—represent the state of the art in physical and visual simulation, providing advanced functionalities for research and industry.

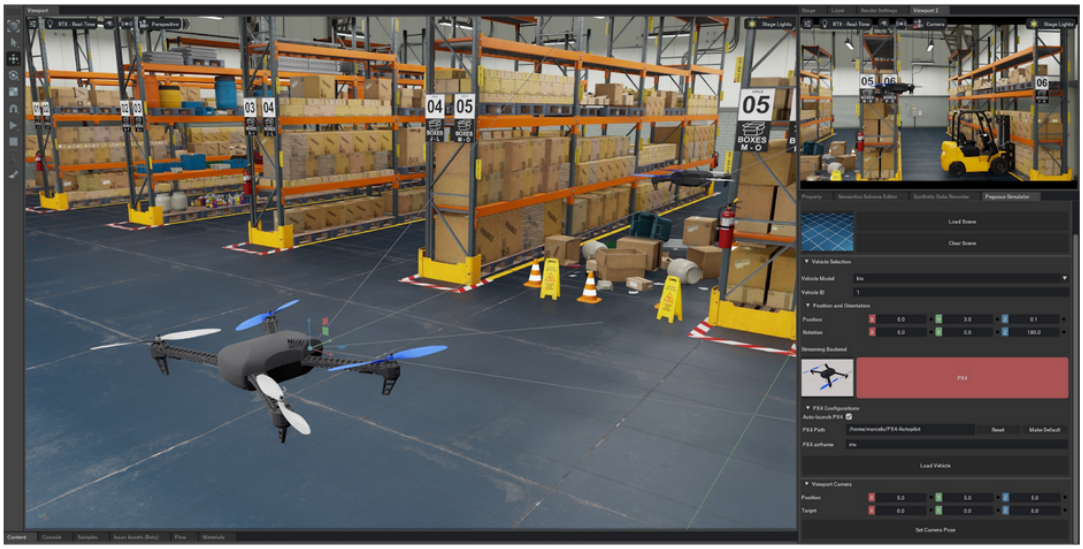

Figure 6: Vista de Isaac Sim mostrando un dron PX4 simulado en un entorno 3D realista.

1.5.1 1. Isaac Sim

Isaac Sim is a next-generation robotic simulator, developed by NVIDIA on the Omniverse platform. It is specifically designed for training, validation and integration of robots that require advanced perception, realistic physical simulation, and massive generation of synthetic data.

- Uses RTX graphics technology and represents ultra-realistic worlds with dynamic lighting, photorealistic materials and high-fidelity physical simulation.
- Allows simulating complex environments (factories, warehouses, cities), manipulation of soft objects, multi-robots, advanced sensors (3D LiDAR, stereo cameras, ToF, RGB-D), and human-robot interaction.
- It is deeply integrated with AI workflows: allows training vision models, detection, segmentation and reinforcement learning.
- Can generate large volumes of automatically labeled synthetic data, vital for training robust deep learning algorithms.
- Supports Python scripting and direct connectivity with ROS2 for "sim2real" validation (transfer of policies from simulation to the real world).

Typical use cases:

- Training of manipulation robots, mobile robots and autonomous vehicles.
- Generation of synthetic datasets for perception algorithms.
- Automatic testing of navigation, SLAM, manipulation, computer vision algorithms, etc.

1.5.2 2. Isaac Lab

Isaac Lab is a suite of tools and environments, complementary to Isaac Sim, specifically oriented to training and benchmarking reinforcement learning agents in realistic robotic scenarios.

- Includes a collection of predefined tasks and scenarios (for example, manipulation, locomotion, human-machine interaction).
- Allows training agents through different AI algorithms (PPO, DDPG, SAC, etc.) with access to photorealistic physics and advanced sensors.
- Leverages Omniverse's parallel infrastructure for distributed training and accelerated experience generation.
- Offers integration with AI and robotics frameworks, facilitating development, testing and comparison of custom algorithms.

Main advantages:

- Drastically reduces development and training time thanks to parallelization and physical/sensory realism.
- Extensible and customizable environments, ideal for research, academic challenges and industrial prototyping.
- Supports the complete cycle of: task definition → simulation → training → export and validation on real hardware.

1.5.3 3. Omniverse

NVIDIA Omniverse is a universal collaborative simulation platform for the design, visualization and validation of photorealistic and physically plausible virtual worlds.

- Allows connecting multiple design applications (CAD, DCC), simulation engines and AI pipelines.
- Uses USD (Universal Scene Description) as the central format, which facilitates interoperability between tools and reuse of scenarios/3D assets.
- Omniverse is the foundation on which Isaac Sim and other tools are built, supporting collaborative workflows, large-scale simulations and cloud streaming.
- It is a meeting point between experts in AI, robotics, graphic design and industrial optimization.

1.5.4 Advantages Compared to Open Source Alternatives like Gazebo

Advantages of Isaac Sim/Omniverse:

- Ultra-realistic visualization thanks to RTX technology and photorealistic rendering.
- High-precision physical simulation, including advanced contact, soft materials, fluids, etc.
- Direct integration with AI algorithms, reinforcement learning and deep learning pipelines.
- Automated generation of labeled synthetic data.
- Scalability: massive training and distributed simulation on GPU.
- Collaborative platform and native connectivity with professional modeling/animation tools.

Disadvantages and limitations:

- Closed code and dependent on the NVIDIA ecosystem; requires compatible graphics hardware (preferably RTX GPUs).
- Less flexibility for low-level customizations and access to all sources offered by 100% open source platforms.
- Requires a steeper learning curve and powerful computational resources to take full advantage of its potential.
- Not always the most efficient option for lightweight simulations, simple environments or rapid development.

Gazebo:

- It is open source, lightweight and highly customizable.
- Has a huge community, resources and examples.
- Suitable for teaching, quick tests, standard robotic integration and development on varied hardware.
- Greater compatibility with modest systems and public reference documentation.
- Lower visual and physical realism, but sufficient for most standard academic and industrial projects.

1.6 Practical Example of Control of Simulated Autonomous Vehicles in Realistic Scenarios

This section presents a comprehensive practical example that integrates the concepts and tools covered throughout the course. The objective is to guide participants through the complete process of configuring, deploying and controlling an autonomous vehicle simulation in a realistic scenario.

1.6.1 Challenge Statement: Scenario and Objective Definition

The practical exercise consists of creating a complete simulation environment where an autonomous vehicle (rover or drone) must navigate through a realistic scenario, using sensors for perception and ROS2 for control and decision-making.

Scenario definition:

- Create or use a pre-existing world in Gazebo that represents a realistic environment (urban area, industrial zone, natural terrain, etc.).
- Define the vehicle model with appropriate sensors (LiDAR, cameras, GPS, IMU).
- Establish navigation objectives: waypoint following, obstacle avoidance, area exploration, etc.

Objectives:

- Successfully integrate all components: Gazebo world, vehicle model, PX4 firmware, ROS2 nodes.
- Achieve autonomous navigation using sensor data and control algorithms.
- Monitor and visualize the vehicle's behavior in real time.
- Validate the simulation results and identify potential improvements.

1.6.2 Step by Step: Configuration, Deployment, Integration and Use of Sensors/Actuators

Step 1: Environment Preparation

1. Install and configure all necessary tools: Gazebo Harmonic, ROS2 Jazzy, PX4 Firmware.
2. Verify that all dependencies are correctly installed.
3. Prepare the workspace structure for the project.

Step 2: World and Vehicle Model Creation

1. Create or select a Gazebo world file (SDF format) with the desired scenario.
2. Define or import the vehicle model (URDF/SDF) with all necessary components.
3. Configure sensors in the model description, ensuring proper plugin configuration for ROS2 communication.

Step 3: PX4 Integration

1. Launch PX4 SITL with the appropriate vehicle model.
2. Configure MicroXRCEAgent for ROS2 communication.
3. Verify that PX4 topics are accessible from ROS2.

Step 4: ROS2 Node Development

1. Create ROS2 nodes for:
   - Sensor data processing (LiDAR, camera, GPS, IMU).
   - Navigation and path planning algorithms.
   - Control command generation for PX4.
2. Implement the communication logic between nodes using topics, services and actions.

Step 5: Launch and Testing

1. Create launch files to orchestrate all components.
2. Launch the complete simulation.
3. Monitor system behavior using RViz2 and command-line tools.
4. Adjust parameters and algorithms as needed.

1.6.3 Result Evaluation and Resolution of Common Problems

Evaluation criteria:

- Vehicle successfully completes the navigation mission.
- Sensors provide reliable and consistent data.
- Control algorithms respond appropriately to environmental conditions.
- System maintains stable performance throughout the simulation.

Common problems and solutions:

- **Communication issues between Gazebo and ROS2:** Verify plugin configuration, check topic names and namespaces.
- **PX4 connection problems:** Ensure MicroXRCEAgent is running and configured correctly, check UDP ports.
- **Sensor data not appearing:** Review plugin configuration in URDF/SDF, verify topic publication.
- **Performance issues:** Optimize world complexity, reduce sensor update rates, adjust physics parameters.
- **Control instability:** Review control algorithm parameters, check sensor data quality, adjust PX4 flight modes.

1.7 Questions, Closing and Reflection

1.7.1 Question Resolution

This final section provides an opportunity to address any remaining doubts or questions that participants may have regarding the concepts, tools and practices covered during the course. Common topics for discussion include:

- Advanced configuration and optimization techniques.
- Integration with other tools and platforms.
- Best practices for project development.
- Troubleshooting specific issues.
- Recommendations for further learning and specialization.

1.7.2 Final Conclusions and Reflection

The course has covered the fundamental aspects of using Gazebo and ROS2 for robotics and autonomous vehicle simulation. Key takeaways include:

- Understanding of the Gazebo and ROS2 ecosystems and their integration capabilities.
- Ability to create and configure robot models and simulation worlds using standard formats (SDF, URDF).
- Knowledge of sensor integration and data communication in ROS2.
- Experience with PX4 firmware for autonomous vehicle control.
- Awareness of advanced simulation tools like Isaac Sim for AI-based applications.

The combination of these tools provides a powerful platform for developing, testing and validating robotic systems before deploying them to real-world environments, significantly reducing development time, costs and risks.

1.7.3 Next Steps and Resource Recommendations

Recommended next steps:

- Practice with more complex scenarios and vehicle configurations.
- Explore advanced ROS2 features: services, actions, parameter management.
- Experiment with different sensor types and fusion techniques.
- Study PX4 advanced features: mission planning, failsafe configuration, parameter tuning.
- Investigate integration with real hardware (hardware-in-the-loop).

Useful resources:

- Official Gazebo documentation: https://gazebosim.org/docs
- ROS2 documentation: https://docs.ros.org/en/humble/
- PX4 documentation: https://docs.px4.io/
- Isaac Sim documentation: https://docs.omniverse.nvidia.com/app_isaacsim/
- ROS2 community forums and tutorials.
- Open source projects and examples on GitHub.

2 License and Usage Rights