# TASK 1: Differential Drive Bot:

## 1. Different views of the bot:

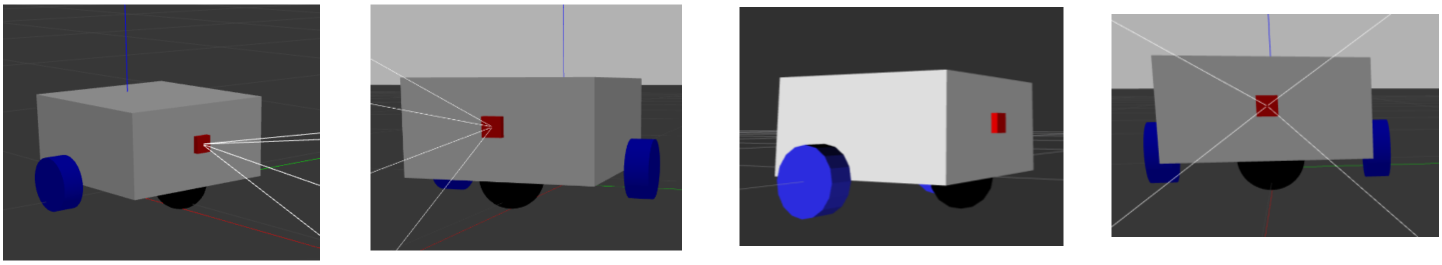

**Figure 1:** Different Views of Differential drive bot in Gazebo and Rviz

## 2. Architecture of the ROS code:

The robot is controlled using ROS 2 nodes integrated with Gazebo simulation. The architecture consists of the following key components:

- **Nodes & Topics:**

  1. **/teleop_twist_keyboard:** Publishes velocity commands (geometry_msgs/msg/Twist) to the /cmd_vel topic.
  2. **/diff_drive (via gazebo_ros_diff_drive plugin):**
     - Subscribes to /cmd_vel to control left and right wheel joints.
     - Publishes odometry data on /odom.
  3. **/robot_state_publisher:**
     - Subscribes to /joint_states and /robot_description to compute forward kinematics and publish TF transforms on /tf.
  4. **/gazebo:**
     - Provides simulation clock (/clock) and simulation control.
     - Publishes performance data (/performance_metrics) and interfaces with controllers.
  5. **/camera_controller:**
     - Simulated camera node that publishes:
       - Raw image data to /camera/image_raw
       - Camera parameters to /camera/camera_info
  6. /rqt_gui_py_node_8949:
     - Visualization or introspection node, subscribing to topics like /joint_states, /tf, and /rosout.

- **Data Flow:**

  1. **Keyboard input** → /teleop_twist_keyboard → /cmd_vel
  2. **Motion control** → /diff_drive reads /cmd_vel, updates wheel joints, publishes /odom
  3. **State publishing:**
     - /joint_states → /robot_state_publisher → /tf
     - /robot_description → /robot_state_publisher
  4. **Gazebo** manages the simulation world and provides /clock
  5. **Camera** streams through /camera/image_raw and /camera/camera_info

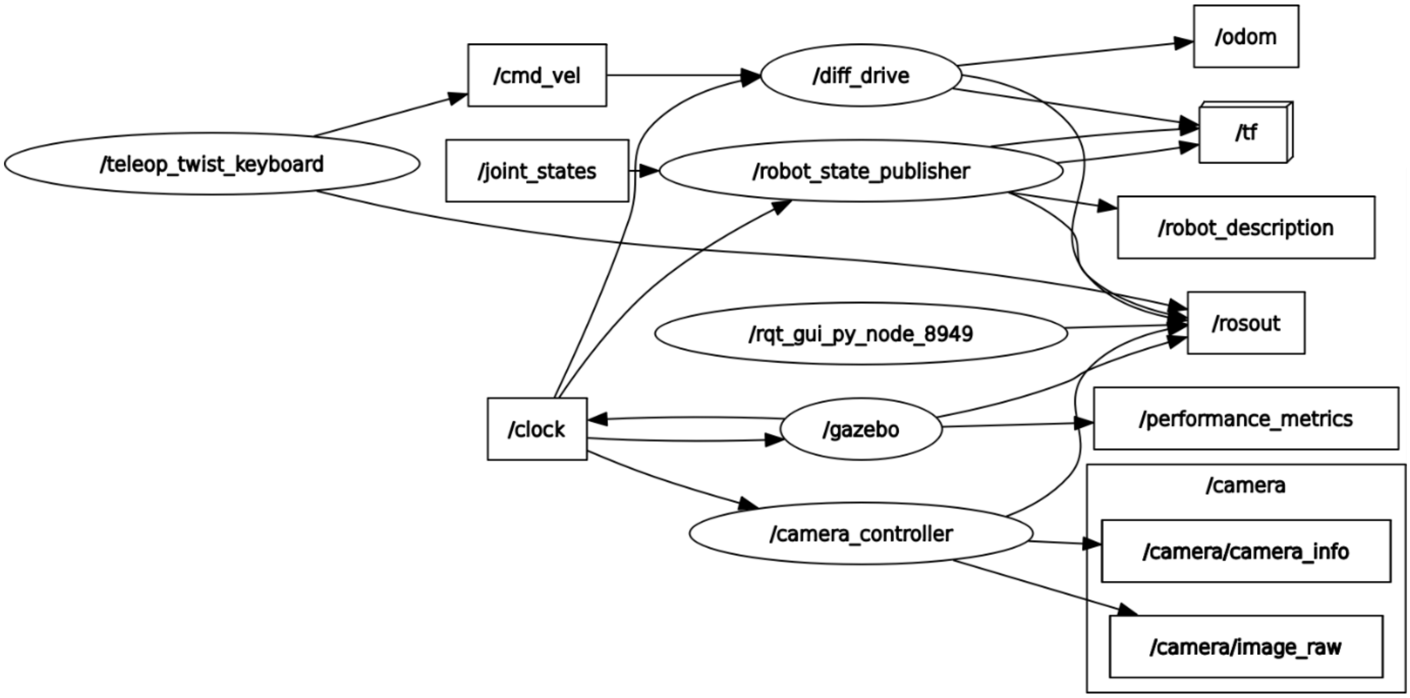


**Figure 2:** rqt graph of Task 1

## 3. Different views of the World:

**Figure 3:** World view 1

**Figure 4:** World view 2

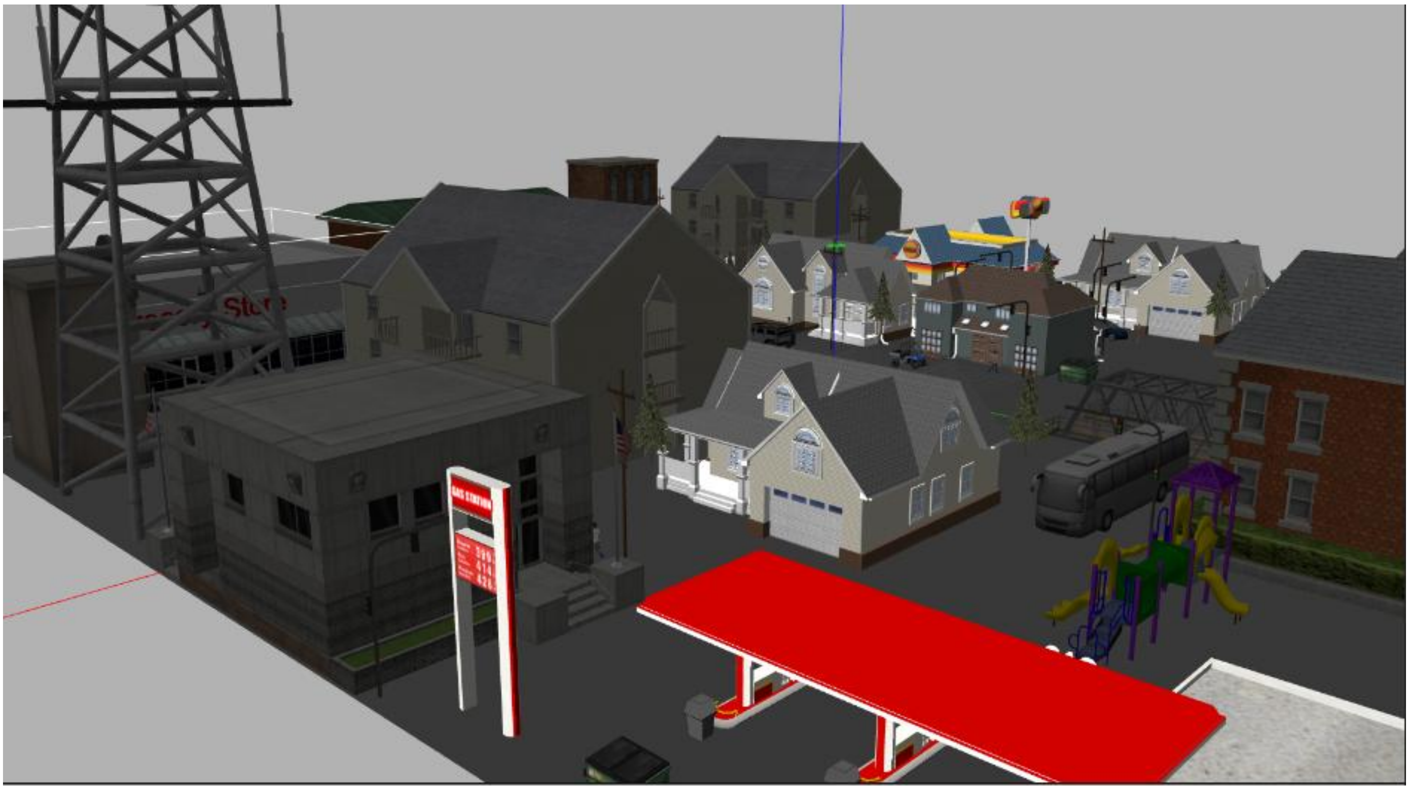

**Figure 5:** World view 3

## 4. Bot spawning in the World:


**Figure 6:** Differential drive bot spawning in the world

## 5. Camera feed from different locations in the world:

**Figure 7:** Camera feed 1 by differential drive bot

**Figure 8:** Camera feed 2 by differential drive bot

# TASK 2: Artoo's Day Out!

## 1. Different views of the bot:

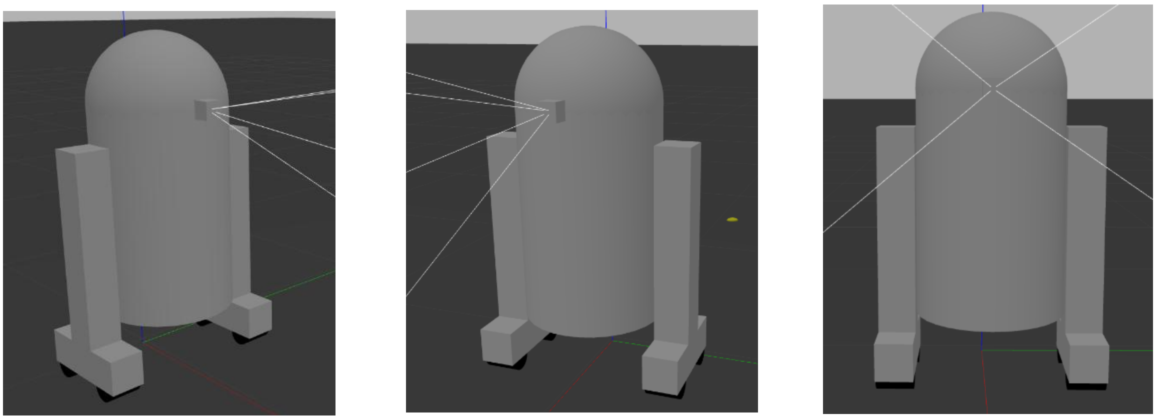

**Figure 9:** Different views of the R2-D2 bot in Gazebo

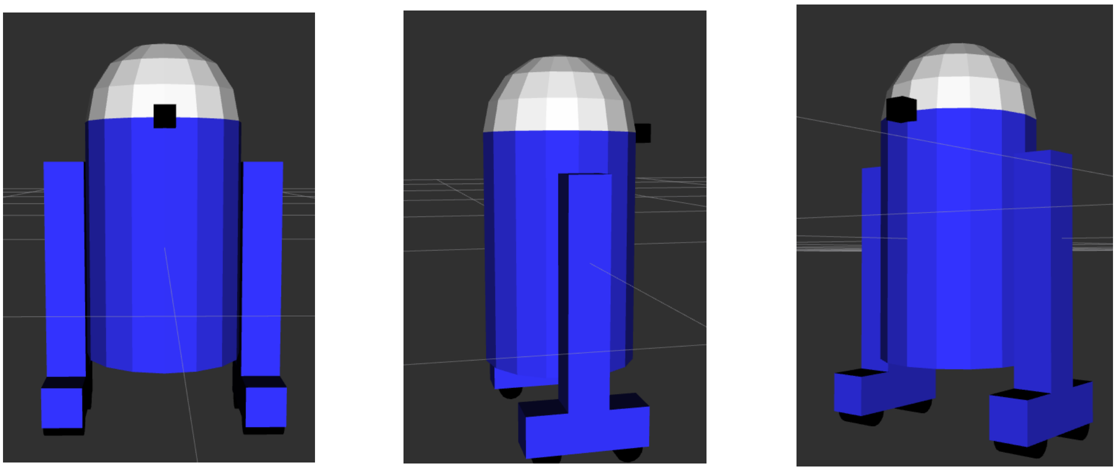

**Figure 10:** Different views of the R2-D2 bot in Rviz

## 2. Architecture of the ROS code:

The ROS 2 architecture for the R2D2 robot is modular, clearly separating different functionalities into nodes, configuration files, and launch mechanisms. Below is a breakdown of each component involved in the simulation:

**a. URDF and Robot Description**

- The robot's structure is defined in a Xacro file (r2d2.urdf.xacro) that describes all physical components:
    - **Base** (base_link): A cylindrical main body.
    - **Head** (head_link): Spherical top part, connected with a revolute joint.
    - **Camera** (camera_link): A fixed joint on the head with a simulated camera.
    - **Legs and Feet**: Two side legs (left_leg, right_leg) and feet (left_foot, right_foot) to support the body.
    - **Wheels**: Front wheels are continuous joints, back wheels are fixed.

**b. Gazebo Integration**

- Gazebo plugins are added for:
    - **Camera sensor** (mounted on the head).
    - **Differential Drive Plugin** to simulate robot mobility using front wheels.
    - **ROS 2 Control Plugin** (gazebo_ros_control) enables joint control through ROS interfaces.

**c. ros2_control Configuration**

- Defined in the ros2_control block of the URDF and head_controller.yaml.
- Includes:
    - joint_state_broadcaster for state feedback of all joints.
    - head_position_controller for controlling head_joint.

**d. Launch System**

- A single launch file does the following:
    - Launches Gazebo with a custom world.
    - Starts robot_state_publisher with processed URDF.
    - Starts ros2_control_node with controller YAML config.
    - Spawns robot entity in Gazebo.
    - Spawns required controllers after a delay (joint_state_broadcaster and head_position_controller).

**e. Control Nodes**

- **Manual Rotation Node** (rotate_head): Publishes a fixed angle to the head controller.
- **Keyboard Teleop Node** (head_teleop): Uses keyboard inputs ('j' and 'l') to rotate the head left/right.
- **Bridge Node** (twist_to_head): Converts geometry_msgs/Twist commands into Float64 values for head_position_controller.

This architecture enables easy debugging, clean separation of simulation, control, and user input, and ensures extensibility for adding more features like full-body motion or sensors.

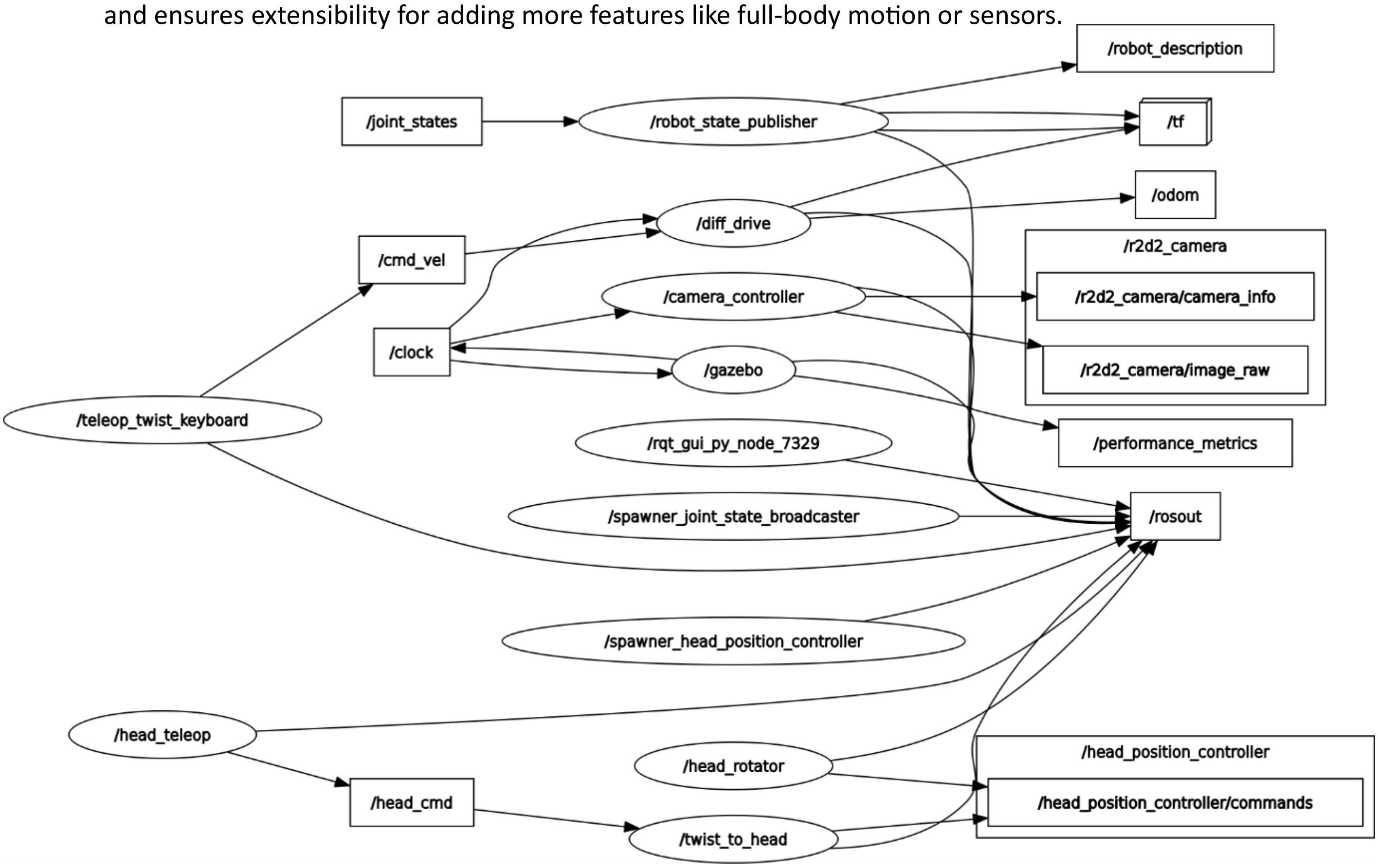


**Figure 11:** rqt graph of Task 2

## 3. Bot spawning in the World:


**Figure 12:** R2-D2 bot spawning in the world

## 4. Camera feed from different locations in the world:

**Figure 13:** Camera feed 1 by R2-B2 bot

**Figure 14:** Camera feed 2 by R2-D2 bot

## 5. Head Rotation Mechanism:

The head of the R2D2 robot is implemented as a separate link (head_link) connected to the body (base_link) via a **revolute joint** called head_joint. This allows rotational movement around the vertical axis (z-axis), simulating the turning of the robot's head.

**Implementation Details:**

- **URDF Configuration**:
  - The joint type is revolute with defined motion limits (-6.28 to 6.28 radians).
  - A transmission is specified using SimpleTransmission, enabling it to receive position commands via ros2_control.
- **ros2_control Setup**:
  - A JointPositionController is configured for head_joint.
  - The controller subscribes to /head_position_controller/commands topic expecting Float64 messages.
- **Command Interfaces**:
  - Three interfaces are exposed for the joint: position, velocity (state), and position (command).
  - A hardware interface plugin (gazebo_ros2_control/GazeboSystem) bridges Gazebo and the controller system.
- **User Interaction**:
  - Users can manually control the head rotation using:
    - rotate_head node (sends a fixed command).
    - head_teleop node (keyboard-based left/right turns).
    - Or programmatically by publishing to /head_cmd which is processed by twist_to_head.

**Result:** This mechanism ensures smooth, realistic rotation of the head in simulation and demonstrates effective integration between Gazebo physics and ROS 2 control interfaces.

The head of the R2D2 robot (top hemisphere) is made to rotate using the **ROS 2 Control framework** in conjunction with **Gazebo simulation**. Here's a detailed breakdown of how this mechanism is defined and controlled:

**URDF Configuration**

**1. Joint Definition:**

```
<joint name="head_joint" type="revolute">
  <parent link="base_link"/>
  <child link="head_link"/>
  <origin xyz="0 0 0.3"/>
  <axis xyz="0 0 1"/>
  <limit lower="-6.28" upper="6.28" effort="10.0" velocity="5.0"/>
  <dynamics damping="1.0" friction="0.5"/>
</joint>
```

- This allows the head_link to rotate about the Z-axis.
- Limits are set from $-2\pi$ to $2\pi$ radians.

**2. Transmission Block:**

```xml
<transmission name="head_joint_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="head_joint">
    <hardwareInterface>position</hardwareInterface>
  </joint>
  <actuator name="head_motor">
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

**3. ros2_control Plugin:**

```xml
<ros2_control name="R2D2System" type="system">
  <hardware>
    <plugin>gazebo_ros2_control/GazeboSystem</plugin>
  </hardware>
  <joint name="head_joint">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
  </joint>
</ros2_control>
```

**4. Controller Configuration (head_controller.yaml):**

```yaml
controller_manager:
  ros__parameters:
    update_rate: 50
    head_position_controller:
      type: position_controllers/JointPositionController
      joint: head_joint
    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster
```

This YAML defines the controller type and maps it to head_joint, allowing external nodes to send position commands.

---

.