UNIVERSITY OF LIVERPOOL

DEPARTMENT OF ELECTRICAL ENGINEERING & ELECTRONICS

Quadruped Robot Group Project

(ELEC330)

Assignment2 report

‘‘movement, automatic obstacle avoidance, object recognition, and navigation of the quadruped robot’’

Author: Zhibin Mo

Haoran Lu

Alex

Aidan

Project Supervisor: Lakany, Heba

Declaration of academic integrity

The standard University of Liverpool statement of academic integrity [6] should go here as follows:

I confirm that I have read and understood the University’s Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated, falsified or embellished data when completing the attached piece of work. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

Abstract

This report discusses in detail the design and implementation of a quadruped robot simulation work package based on ROS2 and Gazebo Sim. By standardizing the file format SDF, the world environment of this work package achieves high-precision physical parameter modeling and diversified scene configuration, including static and dynamic elements, sensor modules, and interactive visualization tools. The description file of the robot integrates GPU Lidar sensors and Bounding Box Camera to enhance environmental perception capabilities, and its control code functions cover object recognition, path planning, automatic obstacle avoidance, and navigation. Meanwhile, OpenCV was utilized to achieve simple obstacle recognition on the image data of the Bounding Box Camera. In addition, the report explores the algorithm implementation of robot joint control and dynamic performance optimization, achieving precise motion control through PID controllers. The summary indicates that the environment has modularity, flexibility, and scalability, which can efficiently support the development and validation of complex robot systems. At the same time, performance bottlenecks and future optimization directions are pointed out.

Contents

Introduction

In today's world, where robotics technologies are rapidly improving, the simulation environment has become indispensable in designing and testing systems for robotic applications. By being able to test the operation of robots prior to physical deployment, by doing this a developer can significantly reduce research and development costs while improving overall efficiency in testing. The SDF is a standardized file format that provides the opportunity for precise modeling of physical properties, scene layouts, and sensor configurations within a virtual environment. It hence facilitates efficient and adaptable development of robots.

This report discusses a simulation environment designed using SDF, aimed at creating a complex, realistic virtual world. It is useful in testing the behavior of robots and sensors under both dynamic and static conditions. This includes fixed elements such as ground planes, walls, slopes, and moving parts like boxes, cylinders, and spheres, enabling a wide range of real-world simulations. Additionally, the simulation achieves high accuracy and stability through meticulous configuration of physical parameters, including gravity, time step size, and atmospheric models.

The environment includes a physics engine, sensor modules, an IMU, and a buoyancy system for the multi-level simulation of robot behaviors. The array of GUI plug-ins, such as 3D views, scene management, and model selection tools, gives users an intuitive and interactive visualization experience. The use of ambient and directional lighting increases not only scene realism but also enables tests of robot vision systems.

In conclusion, this design of a simulation environment presents modularity, flexibility, and scalability; it robustly supports the development and validation of complex robotic systems. Its practical value lies in its ability to streamline and accelerate the iterative design process for robotics applications.

How to Launch

Our new package is called peter2. Once put it into workspace, and run the following comment:

colcon build

ros2 launch peter2 launch.py

After launching the launch file, a total of 5 windows will be launched, including two terminal windows, one Gazebo Sim window, one rviz window, and one OpenCV window. The figure 1 shows the launch status of the package.

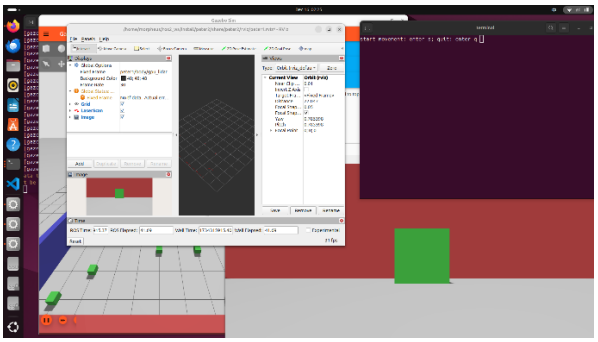

Figure 1. Launch status

Discussion

Overview of Simulation Environment/The World

Introduction

This world was created using gazebo sim, which called peter_world.sdf. The world contains several obstacles, multiple cubes, a light red cylinder and a dark red sphere, all of which are required. Also the world contains four different coloured walls. There is a ramp at the edge of the world file walls. The world contains physical characteristics, including a downward gravitational acceleration of 9.8m/s, atmospheric density, and ground friction coefficient. Previously we tried to create a world model in a ".dae" format, but abandoned its use because it was difficult to add to gazebo. Figure 1 shows an aerial view of the world.

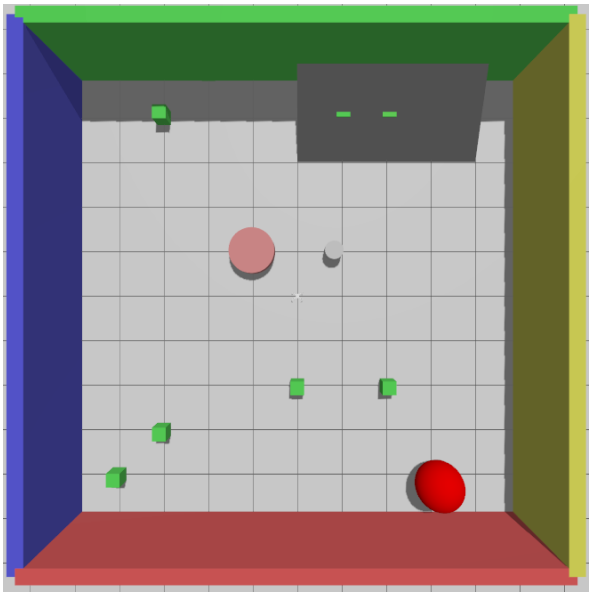

Figure 2. The aerial view of the world

Physics Parameter Design

Designing the physics parameters of the simulation environment plays an essential role in providing correct, realistic robot behavioral modeling. Therefore, it should include in detail most of the necessary parameters in a typical SDF file, for instance: time step size, real-time factors, gravity, magnetic field, atmospheric model. All these define jointly how the virtual world behaves; that is to say, how the simulations run on. Presently, let us talk about the main details around these properties.

1. Time Step Size and Real-Time Factors

Max Step Size (max_step_size):

The maximum step size determines the resolution of each physics update in the simulation. In this design, it is set to 0.001 seconds, corresponding to a high update frequency of 1000 Hz. This fine-grained time resolution is crucial for simulations involving fast dynamics or precise control, ensuring minimal numerical errors in physics calculations.

A smaller step size increases computational load but provides better accuracy for high-speed joint movements of robots or during the interaction of robots with small-size dynamic obstacles.

Real-Time Factor (real_time_factor):

Set to 1, this means the simulation runs in real time, where simulated time equals wall-clock time. Useful for systems (for example, sensor-driven navigation or reactive control algorithms) that should be tested based on real-time feedback.
Real-Time Update Rate (real_time_update_rate):
Set to 1000 updates per second, this parameter aligns with the time step size, maintaining consistency between simulation updates and real-time operation. It ensures a stable and responsive simulation environment.

2. Gravity Configuration

Value:
The gravity vector is set to 0 0 9.8 m/s^2, replicating Earth's standard gravitational force. This realistic setting is essential for simulating robots' locomotion, stability, and interactions with the ground or obstacles.
Impact on Robots:
Gravity directly affects the dynamics of multi-legged robots, as their dynamics must be carefully controlled with respect to joint torques against gravity. Moreover, gravity also takes effect during collision responses between the robot and other objects or the environment.

3. Magnetic Field

Magnetic Vector:
5.5645e-06 2.2876e-05 -4.2388e-05
This very mild magnetic presence allows the simulation of real-world conditions for robots with magnetometers or other magnetic-sensitive sensors.
Applications:
While not critical for all robots, magnetic field simulation is invaluable for navigation systems that rely on geomagnetic data, such as UAVs or autonomous vehicles.

4. Atmospheric Model

Type:
The atmospheric model is set to adiabatic, which provides a simplified representation of atmospheric conditions without complex thermodynamic effects.
Purpose:
While this simple model is sufficient for most robotics simulations, it can be easily extended for aerial robotics or even underwater simulations where atmospheric/fluid dynamics become important.

5. Integration with Plug-Ins

The physics parameters are strongly coupled with simulation plug-ins. This is instrumental in having the behavior of the virtual world aligned with the configured dynamics. For instance,
Physics Engine Plug-In: This ensures that gravity, friction, and collision forces are applied accurately.
The Buoyancy Plug-In uses uniform fluid density (set to 1000 kg/m^3) to simulate water or other fluid environments, allowing for realistic underwater robot testing.

6. Balancing Accuracy and Performance

While high update rates and small-time steps improve accuracy, they increase computational demand. This design strikes a balance by:
Setting a fine step size suitable for precise robot dynamics.
Utilizing optimized collision detection and simplified atmospheric modeling to maintain real-time operation.

Model Definition

The Model Definition module is the keystone of the simulation environment. In other words, it details how things are in the static and dynamic parts of the virtual world. The SDF file would comprise a very

detailed model that includes static structures such as a ground plane and/or wall, and dynamic elements: boxes, cylinders, spheres, etc. These will have precisely calculated geometries, collision properties, and visual attributes with dimensions to emulate real events or situations. Below is a detailed explanation of the models, supported by the respective code snippets.

1. Ground Plane

The ground plane forms the foundational surface for all interactions in the simulation.

Code Snippet:

```
<model name='ground_plane'>
      <static>true</static>
      <link name='link'>
        <collision name='collision'>
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>100 100</size>
            </plane>
          </geometry>
          <surface>
            <friction>
              <ode>
               <mu>1</mu>
               <mu2>1</mu2>
              </ode>
            </friction>
            <bounce/>
            <contact/>
          </surface>
        </collision>
        <visual name='visual'>
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>100 100</size>
            </plane>
          </geometry>
          <material>
            <ambient>0.800000012 0.800000012 0.800000012 1</ambient>
            <diffuse>0.800000012 0.800000012 0.800000012 1</diffuse>
            <specular>0.800000012 0.800000012 0.800000012 1</specular>
          </material>
        </visual>
        <pose>0 0 0 0 0 0</pose>
        <inertial>
          <pose>0 0 0 0 0 0</pose>
          <mass>1</mass>
          <inertia>
            <ixx>1</ixx>
            <ixy>0</ixy>
            <ixz>0</ixz>
            <iyy>1</iyy>
            <iyz>0</iyz>
            <izz>1</izz>
```

```
          </inertia>
        </inertial>
        <enable_wind>false</enable_wind>
      </link>
      <pose>0 0 0 0 0 0</pose>
      <self_collide>false</self_collide>
    </model>
```

Analysis:

Static Property: The static attribute is set to true, indicating that the ground plane does not move or interact dynamically.

Geometry: A large 100x100 plane is defined with a normal vector (0, 0, 1) to represent a flat surface.

Visual Properties: Ambient, diffuse, and specular light properties are configured to provide a realistic appearance under different lighting conditions.

Collision: Ensures the plane interacts physically with other models.

2. Walls

The walls enclose the simulation environment, providing boundaries for robot movement.

Code Snippet (Left Wall):

```
<model name='wall_left'>
      <pose>-5 0 0.15 0 0 0</pose>
      <link name='wall_left_link'>
        <collision name='wall_left_collision'>
          <geometry>
            <box>
              <size>0.3 10 3</size>
            </box>
          </geometry>
          <surface>
            <friction>
              <ode/>
            </friction>
            <bounce/>
            <contact/>
          </surface>
        </collision>
        <visual name='wall_left_visual'>
          <geometry>
            <box>
              <size>0.3 10 3</size>
            </box>
          </geometry>
          <material>
            <ambient>0.8 0.1 0.1 1</ambient>
            <diffuse>0.8 0.1 0.1 1</diffuse>
            <specular>0.8 0.1 0.1 1</specular>
          </material>
        </visual>
      </link>
      <static>true</static>
    </model>
```

Analysis:

Geometry: The wall is modeled as a box with dimensions 0.3x10x3 meters.

Position: The pose places the wall at (-5, 0, 0.15), creating one of the four boundary walls.

Visual Properties: Distinct ambient and diffuse colors (0.8, 0.1, 0.1) make the wall easily identifiable in the simulation.
Collision: Ensures the wall interacts with dynamic models, preventing objects from leaving the environment.

3. Dynamic Models

Dynamic models simulate movable obstacles or interactive objects in the environment.

Example 1: Obstacle Box
Code Snippet:

```
<model name='obstacle_box1'>
  <pose>-2 0 0.15 0 0 0</pose>
  <link name='obstacle_box_link'>
    <collision name='obstacle_box_collision'>
      <geometry>
        <box>
          <size>0.3 0.3 0.3</size>
        </box>
      </geometry>
      <surface>
        <friction>
          <ode/>
        </friction>
        <bounce/>
        <contact/>
      </surface>
    </collision>
    <visual name='obstacle_box_visual'>
      <geometry>
        <box>
          <size>0.3 0.3 0.3</size>
        </box>
      </geometry>
      <material>
        <ambient>0.1 0.8 0.1 1</ambient>
        <diffuse>0.1 0.8 0.1 1</diffuse>
        <specular>0.1 0.8 0.1 1</specular>
      </material>
    </visual>
  </link>
  <static>false</static>
</model>
```

Analysis:
Geometry: A cube with dimensions 0.3x0.3x0.3 meters is defined.
Position: Initially placed at (-2, 0, 0.15) in the simulation world.
Material: Bright green (0.1, 0.8, 0.1) for easy identification.
Dynamic Behavior: Unlike static models, these boxes can interact physically with other objects and respond to forces.
Example 2: Obstacle Cylinder
Code Snippet:

```
<model name='obstacle_cylinder'>
      <pose>1 1 0.1 0 0 0</pose>
      <link name='obstacle_cylinder_link'>
        <collision name='obstacle_cylinder_collision'>
          <geometry>
            <cylinder>
              <radius>0.5</radius>
              <length>0.3</length>
            </cylinder>
          </geometry>
          <surface>
            <friction>
              <ode/>
            </friction>
            <bounce/>
            <contact/>
          </surface>
        </collision>
        <visual name='obstacle_cylinder_visual'>
          <geometry>
            <cylinder>
              <radius>0.5</radius>
              <length>0.3</length>
            </cylinder>
          </geometry>
          <material>
            <ambient>0.8 0.3 0.3 1</ambient>
            <diffuse>0.8 0.3 0.3 1</diffuse>
            <specular>0.8 0.3 0.3 1</specular>
          </material>
        </visual>
      </link>
      <static>false</static>
    </model>
```

Analysis:

Geometry: A cylinder with a radius of 0.5 meters and height of 0.3 meters.

Position: Placed at (1, 1, 0.1).

Purpose: Serves as a larger obstacle to test robot navigation and collision avoidance.

4. Light Source

A directional light source (sun) is included to illuminate the simulation environment.

Code Snippet:

```
 <light name='sun' type='directional'>
      <pose>0 0 10 0 0 0</pose>
      <cast_shadows>true</cast_shadows>
      <intensity>1</intensity>
      <direction>-0.5 0.10000000000000001 -0.90000000000000002</direction>
      <diffuse>0.800000012 0.800000012 0.800000012 1</diffuse>
      <specular>0.200000003 0.200000003 0.200000003 1</specular>
      <attenuation>
        <range>1000</range>
        <linear>0.01</linear>
        <constant>0.90000000000000002</constant>
        <quadratic>0.001</quadratic>
      </attenuation>
```

```
      <spot>
        <inner_angle>0</inner_angle>
        <outer_angle>0</outer_angle>
        <falloff>0</falloff>
      </spot>
    </light>
```

Analysis:
Position: Located at (0, 0, 10), simulating sunlight.
Direction: Casts light diagonally with a vector (-0.5, 0.1, -0.9).
Purpose: Enhances visual realism and supports testing of vision-based robot sensors.

Visualization and Interaction Design

The Visualization and Interaction Design of the simulation environment ensures users can effectively interact with and observe the virtual world. In this section, by integrating several GUI plugins and designing intuitive interfaces, the visual experience will be smooth, hence enhancing usability and debugging. The following is a detailed analysis of the visualization and interaction components based on the SDF configuration.

1. GUI Plugins for Visualization

The simulation environment leverages several GUI plugins to provide dynamic and informative visualizations.

1.1 3D View Plugin

Code Snippet:

```
<plugin filename="MinimalScene" name="3D View">
          <gz-gui>
            <title>3D View</title>
            <property type="bool" key="showTitleBar">false</property>
            <property type="string" key="state">docked</property>
          </gz-gui>

          <engine>ogre2</engine>
          <scene>scene</scene>
          <ambient_light>0.4 0.4 0.4</ambient_light>
          <background_color>0.8 0.8 0.8</background_color>
          <camera_pose>-6 0 6 0 0.5 0</camera_pose>
          <camera_clip>
            <near>0.25</near>
            <far>25000</far>
          </camera_clip>
        </plugin>
```

Analysis:
Purpose: Provides a real-time 3D visualization of the simulation environment.
Features:
Ambient light (0.4, 0.4, 0.4) and background color (0.8, 0.8, 0.8) create a visually pleasant scene.
Camera configuration:
Positioned at (-6, 0, 6) with a focus angle of 0.5 radians.
Clipping distances are set between 0.25 and 25000 meters to accommodate both close-up and large-scale views.

Uses the Ogre2 rendering engine for high-quality graphics.

1.2 Scene Management Plugin

Code Snippet:

```
  <plugin filename="GzSceneManager" name="Scene Manager">
            <gz-gui>
              <property key="resizable" type="bool">false</property>
              <property key="width" type="double">5</property>
              <property key="height" type="double">5</property>
              <property key="state" type="string">floating</property>
              <property key="showTitleBar" type="bool">false</property>
            </gz-gui>
          </plugin>
```

Analysis:

Purpose: Allows users to manage and organize scene elements, including models, lights, and other entities.

Features: Provides an intuitive interface to navigate and control the simulation's scene hierarchy.

1.3 Camera Tracking Plugin

Code Snippet:

```
 <plugin filename="CameraTracking" name="Camera Tracking">
            <gz-gui>
              <property key="resizable" type="bool">false</property>
              <property key="width" type="double">5</property>
              <property key="height" type="double">5</property>
              <property key="state" type="string">floating</property>
              <property key="showTitleBar" type="bool">false</property>
            </gz-gui>
          </plugin>
```

Analysis:

Purpose: Automatically tracks selected entities, such as robots or dynamic obstacles, to keep them in the camera's focus.

Application: Particularly useful for observing robot movement and interactions in real-time.

1.4 World Statistics Plugin

Code Snippet:

```
 <plugin filename="WorldStats" name="World stats">
            <gz-gui>
              <title>World stats</title>
              <property type="bool" key="showTitleBar">false</property>
              <property type="bool" key="resizable">false</property>
              <property type="double" key="height">110</property>
              <property type="double" key="width">290</property>
              <property type="double" key="z">1</property>

              <property type="string" key="state">floating</property>
              <anchors target="3D View">
                <line own="right" target="right"/>
                <line own="bottom" target="bottom"/>
              </anchors>
            </gz-gui>

            <sim_time>true</sim_time>
            <real_time>true</real_time>
            <real_time_factor>true</real_time_factor>
            <iterations>true</iterations>
          </plugin>
```

Analysis:
Purpose: Provides detailed statistics about the simulation's progress.
Displayed Metrics:
Simulation time (sim_time) and real time (real_time).
Real-time factor, indicating the performance of the simulation.
Number of iterations, showing how many physics steps have been computed.

2. Interaction Features

2.1 Model Manipulation

Users can dynamically spawn, move, and rotate models within the environment using GUI controls.
Example: The Transform Control Plugin allows precise adjustments of model positions and orientations.

2.2 Selection and Inspection

Select Entities Plugin:
Enables users to select specific models or elements in the scene for detailed inspection or manipulation.
Component Inspector Plugin:
Displays detailed information about a selected entity's properties, such as pose, collision geometry, and visual attributes.

3. Lighting Configuration

Lighting plays a critical role in enhancing realism and testing robot vision systems.
Ambient Lighting:

```
   <ambient_light>0.4 0.4 0.4</ambient_light>
```

Provides uniform lighting across the scene, ensuring all objects are visible.
Directional Lighting:

```
<light name='sun' type='directional'>
      <pose>0 0 10 0 0 0</pose>
      <cast_shadows>true</cast_shadows>
      <intensity>1</intensity>
      <direction>-0.5 0.10000000000000001 -0.90000000000000002</direction>
      <diffuse>0.800000012 0.800000012 0.800000012 1</diffuse>
      <specular>0.200000003 0.200000003 0.200000003 1</specular>
      <attenuation>
        <range>1000</range>
        <linear>0.01</linear>
        <constant>0.90000000000000002</constant>
        <quadratic>0.001</quadratic>
      </attenuation>
      <spot>
        <inner_angle>0</inner_angle>
        <outer_angle>0</outer_angle>
        <falloff>0</falloff>
      </spot>
    </light>
```

Mimics sunlight, casting realistic shadows and enhancing depth perception.

System Implementation and Analysis

The focus of the System Implementation and Analysis module is to show how this simulation environment is set up and works. This includes an analysis of how physical parameters, plugins, and models are put together in one interactive virtual world. It offers comprehensive details on the architecture, the flow of how different components of the system will work, and some consideration for performance.

1. System Structure

The simulation environment is built on a modular framework using SDF, where each component contributes to the overall functionality:

Physical Core: Defines the environment's dynamics, including gravity, time step, and real-time factors.
Plugins: Extend the system's capabilities, including physics, sensors, scene broadcasting, and interaction controls.
Models: Serve as the building blocks of the environment, incorporating static elements (ground, walls) and dynamic entities (obstacles, movable objects).
Visualization Layer: Provides an interactive GUI for monitoring and controlling the simulation.

2. Workflow and Integration

The system operates through the coordinated interaction of its components:

2.1 Physical Core

The physics engine is configured through SDF properties and the gz-sim-physics-system plugin. This setup ensures realistic dynamics by processing:
Gravity: Simulates real-world force acting on all objects.
Collisions: Detects and resolves physical interactions between models.
Time Step: Maintains consistent updates at high precision (1000 Hz).
Key Integration:

```
    filename="gz-sim-physics-system"
           name="gz::sim::systems::Physics">
      </plugin>
```

2.2 Plugin System

Plugins are the backbone of the simulation, handling essential functions:

Physics Engine: Manages real-time dynamics for models and obstacles.
Sensors System: Simulates cameras, lidar, and other sensors.
Scene Broadcaster: Ensures the simulation state is visualized correctly in the GUI.
Example: The IMU System Plugin provides inertial data for robots.

```
  filename="gz-sim-imu-system"
           name="gz::sim::systems::Imu">
      </plugin>
```

2.3 Model Interaction

Each model integrates multiple attributes—geometry, collision, and visuals:

Static Models (e.g., Ground Plane): Provide foundational elements without dynamic properties.
Dynamic Models (e.g., Obstacles): Include mass, inertia, and friction to interact with the environment realistically.
Collision Configuration Example:

```
<collision name='wall_left_collision'>
        <geometry>
          <box>
            <size>0.3 10 3</size>
          </box>
        </geometry>
```

```
          <surface>
            <friction>
              <ode/>
            </friction>
            <bounce/>
            <contact/>
          </surface>
        </collision>
```

3. Operational Features

3.1 Real-Time Interaction

The system supports real-time interaction through GUI plugins:

Add or remove models dynamically.

Inspect and modify properties of objects, such as pose, geometry, or friction coefficients.

3.2 Sensor Simulation

Integrates GPU-based sensors (lidar, cameras) for environmental perception.

Generates sensor data streams that mimic real-world robotic applications, such as obstacle detection and path planning.

3.3 Lighting and Visualization

Use directional and ambient lighting to create a realistic simulation space.

Enables shadow casting and brightness adjustments to test vision algorithms under varying conditions.

4. Performance Analysis

4.1 Strengths

Accuracy: High-resolution physics simulation (1000 Hz) ensures precise dynamic modeling.

Flexibility: Modular design allows easy addition or modification of models and plugins.

Visualization: Comprehensive GUI support improves usability and debugging.

4.2 Limitations

Computational Demand: High precision and large-scale scenes can stress processing resources, especially with complex models or multiple sensors.

Collision Handling: Detailed collision geometries can increase computation time, requiring optimization for real-time performance.

4.3 Optimization Strategies

Reduce sensor update rates or resolution for simpler scenarios.

Simplify collision geometries for models to improve computation efficiency.

Use hierarchical LOD (Level of Detail) for visual elements to reduce rendering overhead.

5. Application Scenarios

The implemented system is versatile and can support a wide range of robotics applications:

Navigation Testing: Dynamic obstacles and sensor integration enable testing path-planning algorithms.

Control System Validation: Real-time physics and interaction provide a realistic testbed for robot controllers.

Behavioral Simulations: Complex scenarios like multi-agent interactions or environment mapping can be simulated accurately.

Summary

This code provides a very modular simulation environment for both static and dynamic models, physical parameter settings, sensor simulations, and interaction with the environment through a built SDF file format. By defining basic models such as ground, walls, and obstacles, it incorporates multiple plug-ins, including physics engines, IMU systems, and sensor modules, creating a realistic and

adaptive virtual world. The high-precision physical simulation and real-time interaction of the environment make it well fit for tasks such as robot navigation, path planning, and dynamic control.

Advantages:
Modular Design: The flexible architecture is easy to expand and customize.
High-Precision Simulation: Detailed physical parameter settings ensure that the simulation results are very similar to those in a real environment.
Comprehensive Functionality: The integration of diverse sensors and dynamic plug-ins accommodates complex task requirements.
Visualization support with a rich set of GUI plug-ins allows for intuitive scene manipulation and monitoring.

Disadvantages:
High Computational Demands: High-resolution simulations may easily cause performance bottlenecks, especially in complex scenarios.
Steep Learning Curve: Beginners may find it hard to understand and use the code efficiently.

Future Potential:
This code offers extreme scalability and will be taken even further for enhancements towards much more advanced use cases: testing swarm robotics, drone dynamics, and fusion of multimodal sensors. Availing the systems of more effective collision detection algorithms and layered detail management in scenarios would increase their simulation resolution with less computational overhead and will pave the path to robust support for robotic system developments.

Robot Sdf file

1.Functional Objectives of the Code

The project plans to simulate a quadruped robot model that could move in four directions within the virtual environment. The movement of the robot will utilize posture designs developed in our first assignment. Moreover, it is designed to acquire distance information from the environment through a laser radar system (GPU Lidar). Such data will be important for developing path-planning algorithms and implementing obstacle avoidance. I will try to analyze and give elaboration of how we have structured and developed the SDF file in this module to gain these functionalities. Figure 3 shows the overview of the quadruped robot.

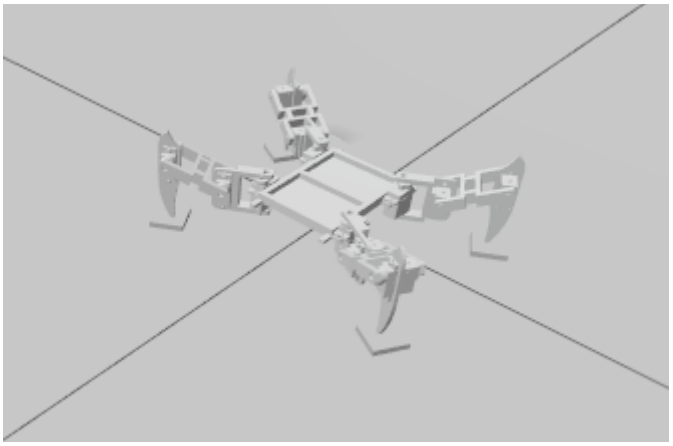

Figure 3. Quadruped robot

1.1 Environmental Sensing Components：

The simulation incorporates two major features that are required for effectively perceiving the environment: GPU Lidar and the Bounding Box Camera. These two tools emulate some advanced sensing systems used in robotics, thus laying a base for environmental awareness and interaction.

GPU Lidar: Advanced Laser Scanning for Navigation：

The Lidar is a simulated laser scanning device developed to model the real-world Lidar used for environmental sensing in robotics. It works by sending laser beams in a pre-specified horizontal and vertical scanning pattern and capturing a complete spatial dataset. In this code, Lidar makes 720 horizontal samples with a resolution of 1° over a 90° field of view. These range measurements enable a robot to detect obstacles and build accurate maps of environments.

Main features of the GPU Lidar are as follows: It possesses a range and resolution whereby the distances within 0.05 m to 10 m are measured with an accuracy of 0.01 m for the establishment of obstacles. It grants real-time environmental feedback by running at an update rate of 10 Hz during decision-making processes.

Physics Simulation: Lidar performs an accurate simulation of realistic properties; thus, it can act like a robust navigation and obstacle avoidance tool within the simulation.

This is very important in tasks like the identification of safe paths, collision avoidance, and dynamic trajectory adjustment due to changes in the environment.

Bounding Box Camera: Object Detection and Visual Processing

The Bounding Box Camera supplements the Lidar with enhanced visual sensing. It simulates a camera with object detection functionality, returning 2D bounding boxes for every object within its field of view. These bounding boxes provide precise information about object positions, enabling the robot to identify and track environmental features.

Key features of the Bounding Box Camera include:

High-Resolution Imaging: Captures images at a resolution of 1920x1080 pixels, ensuring clarity and detail.

Field of View and Clipping Range: Works within a 60° horizontal field of view and a clipping range of 0.1 m to 10 m, offering an optimal balance of range and precision.

Dynamic Scene Adaptation: Processes up to 30 frames per second, making it well-suited for real-time navigation, object recognition, and tracking in dynamic environments.

This simulates the operation of state-of-the-art robotic vision and thus is instrumental for performing challenging tasks such as visual navigation and making decisions under rapidly changing conditions.

1.2Integration and Application：

The quadruped robot integrates both the GPU Lidar with its accurate measurements of distance and the Bounding Box Camera with object detection capability, allowing the robot to understand its environment. While it greatly improves the accuracy of the environment perception, this dual-sensor approach will also provide the robot with solid tools for performing path planning and avoiding obstacles. These components constitute the backbone of the robot's sensory architecture and thus enable operation in realistic simulated environments. This module has focused on how these have been integrated into the SDF file and how central they are to making a robot system highly functional and adaptive.

1.3 Revolute Joint

The Revolute Joint is one of the most used joint types in robotic mechanical structures. It allows two links to rotate around a fixed axis, thus effectively emulating real-world articulated parts. In the robot

model, a revolute joint is defined by its rotation axis, angle limits, and dynamic properties such as spring stiffness and damping. These parameters enable the precise control of the joint, ensuring that its functionality aligns with both physical and computational requirements.

The following code provides a revolute joint that connects different links together, allowing for flexible movement within the robot. An example is the setting of its rotational limits to specific angular displacements to avoid movements which are unrealistic or undesirable and will result in destabilizing the robot. By this design, the robot can realize the following:
Quadruped gait walking: This will ensure that the leg movements are well-coordinated to provide stable locomotion.
Grasping by robotic arm: It allows the exact positioning and manipulation of objects.
Directional adjustment during navigation: Providing rotational flexibility for effective path alignment.

A key feature of the revolute joint is its ability to accurately simulate physical behaviors. This is achieved by combining dynamic properties with motion control algorithms, such as PID control. The PID controller adjusts the joint's movements by balancing proportional, integral, and derivative gains, enabling smooth and responsive control.

The revolute joint, being a fundamental component of robotic design, lays the basic platform for flexible, accurate, and realistic mechanical movements. Its flexibility and precision are the essence of any simulation that could mimic real robotic behaviors.

Code:

```
<joint name='jlf1' type='revolute'>
      <pose relative_to='body'>-0.04 -0.04 0.04 -1.5708 5.5511151231257827e-17
-0.52359999999999995</pose>
      <parent>body</parent>
      <child>llf1</child>
      <axis>
        <xyz>0 1 0</xyz>
        <limit>
          <lower>-3.14</lower>
          <upper>3.14</upper>
          <effort>10</effort>
          <velocity>10</velocity>
        </limit>
        <dynamics>
          <spring_reference>0</spring_reference>
          <spring_stiffness>0</spring_stiffness>
        </dynamics>
      </axis>
    </joint>
```

In the code that we designed, the revolute joint ‘jlf1’ is set to be of the revolute type and connects the main link ‘body’ of the robot to the sub-link ‘llf1’. The <pose> element sets the position and orientation of this joint with precision, with respect to <body>. Position coordinates, -0.04 -0.04 0.04, describe the offset of the joint, while the rotation parameters set an initial posture of the joint as -1.5708 radians around the X-axis and -0.5236 radians around the Z-axis. These settings will allow for precise alignment of the joint in the robot model.

The <axis> element defines the rotation axis of the joint as '[0, 1, 0]', which means it rotates around the Y-axis. For controlling the movement of the joint, its angular range is limited to between '-3.14' and '3.14' radians. Moreover, the joint is given a maximum torque limit of 10 N·m and a maximum speed of 10 radians/second, so that its movements remain physically realistic and within safe operational parameters.

Elastic restoring forces and damping are set to zero to let the joint move freely without any resistance due to internal spring forces or due to damping effects. Position control is very accurate using a PID controller plugin that drives the joint according to proportional, integral, and derivative gains. The plugin publishes the status of the joint-including position, velocity, and applied torque-in real time for efficient monitoring and feedback control.

These configurations enable 'jlf1' to achieve those flexibilities and accuracies crucial for robotic simulations, thus enabling it to execute complex movements with due physical realism and precision in control.

1.4 friction, collision, and visual effects

In robotic simulation, it is important to accurately model real-world environments in terms of friction, collision, and visual effects for close-to-actual results. The properties are carefully defined by the geometry, surface characteristics, and material specifications of the robot's links, so that realistic interactions can be simulated.

Friction is crucial in how the robot interacts with contact surfaces. The <friction> element models this, defining coefficients such as mu and mu2, which enable the simulation of movement and sliding behavior on various surfaces, providing realistic resistance to motion. This becomes important when simulating tasks such as walking, where stable contact with the ground is required.

The <collision> tag defines the collision properties in detail, including geometry-for example, a mesh model in the <mesh> tag-and surface contact properties like elasticity and a recovery force due to collision. This will make the robot behave dynamically in a way that is in accord with physical laws while interacting with the environment or other objects. For instance, this is a setting of a collision in which, in case the robot's link touches the surface, determines the deformation and dissipation of energy.

Visual effects are introduced using the <visual> tag, which describes the appearance geometry and material properties of the link. These visual attributes enhance not only the realism of the simulation but also become crucial in sensor simulation, such as object recognition and environmental perception.

These physical properties synergize in the simulation process. For example, friction and collision directly affect the dynamic behavior of the robot in terms of stability and locomotion, while the appearance contributes to a more appealing and useful simulation environment for generating sensor data and for environment awareness.

With such a detailed configuration of physical properties, simulation gains much realism, enabling the reproduction of complex real-world environments. This realism provides a solid foundation for testing, verifying, and optimizing robotic behaviors, ensuring that simulation insights confidently can be translated into practical applications.
Code:

```
<collision name='llf1_collision'>
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <mesh>
            <scale>1 1 1</scale>
            <uri>model://description/meshes/llf1.STL</uri>
          </mesh>
        </geometry>
          <surface>
            <friction>
              <ode>
               <mu>1</mu>
               <mu2>1</mu2>
              </ode>
            </friction>
            <bounce/>
            <contact/>
          </surface>
      </collision>
```

For instance, here is how to set collision properties of link llf1 within the sample code by means of the <collision> element, which elaborately represents its geometry and surface characteristics in detail with regards to physical behaviour.

First, the collision geometry is defined using the <mesh> tag inside the <geometry>element. This mesh file, model://description/meshes/llf1.STL, depicts the 3D model of the link llf1. By maintaining the <scale> at 1 1 1, the simulation considers the true dimensions of the 3D model for the physical boundary conditions, accurate and compatible with the actual design dimensions of the robot.

Further, the surface characteristics of the link are being set through <surface> by collisions with other objects: Friction coefficients <mu> and <mu2> set to 1 simulating a high-friction environment. A high value for friction stabilizes the robot when in contact with any surface. It helps minimize sliding at each step of the walk and may help with tasks of balance in rough terrain conditions.
The <bounce> element is empty, which means that the link does not have elastic recovery properties. Consequently, it will not bounce back upon collision, adding to the more realistic and stable nature of the simulation.
The <contact> element enables collision detection, thus allowing the simulation to register and respond to interactions between llf1 and other objects. However, the properties of the contact points are not explicitly defined in this configuration for simplicity.

The key feature of this setup is the replication of the dynamic behavior of the link llf1 during interactions in the simulated environment. For example, if the robot moves, then the contact friction configured between llf1 and the ground significantly affects the stability and movement dynamics. The absence of any rebound effect makes the behavior predictable for the robot and in control during collisions.

This setup provides an effective simulation of real collision behaviors while keeping the model simple and flexible. By properly tuning friction and geometric properties, complex physical interactions can be handled in the simulation, allowing the behaviors of robots to be close to real-world scenarios. This will definitely provide a very sound basis for debugging and optimization of the robot system, smoothly transitioning from simulation to practice.

2. Logic and Algorithms Supporting the Functions

The code implements a modular physical construction of the robot through a hierarchical structure of links and joints. Each link is defined with characteristics including inertia, collision, and vision properties that specify the nature of its physical behavior inside the simulation environment. On the other hand, joints serve to connect these links with each other and specify their movement modes, which may be rotational or translational. This is a modular and hierarchical design that allows the robot model to be extended or modified easily. For instance, new links or joints can be added, or the physical properties of existing components can be adjusted for specific simulation or application requirements.

2.1 Sensor Simulation

GPU Lidar:

The gpu_lidar simulation uses some parameter configurations in terms of scanning resolution, sample size, and horizontal and vertical field of view. The example is as follows:

Horizontal configuration: 720 sampling points, with a resolution of 1 ° covering a range of 90 ° in front of the robot.

Vertical configuration: Only one sample, indicating that the radar is a two-dimensional radar.

The<allways_on>and<visualize>parameters are used to generate and visualize laser_stcan data in real-time. This enables robots to perceive the distance of objects in their surrounding environment, laying the foundation for tasks such as obstacle detection and map drawing. The actual range of the lidar is shown in Figure 4.

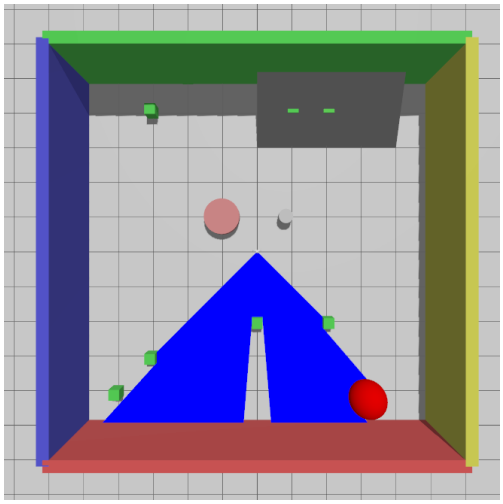

Figure 4. The actual range of the lidar

Bounding Box Camera:

The Bounding Box Camera is supported with additional sensory input in the form of defining a field of view and clipping range through parameters such as horizontal_fov and clip. Example:

This world utilizes a camera with a horizontal field of view of about 60° (1.047 radians), and the clipping range is set between 0.1 to 10 meters.

This camera publishes 2D bounding box data to represent such objects' positions via a topic called boxes for the robot's internal use. Such data comes in handy for many other tasks, such as Visual Navigation and Object Tracking applications by allowing the robot to serve useful purposes in dynamic environments, too.

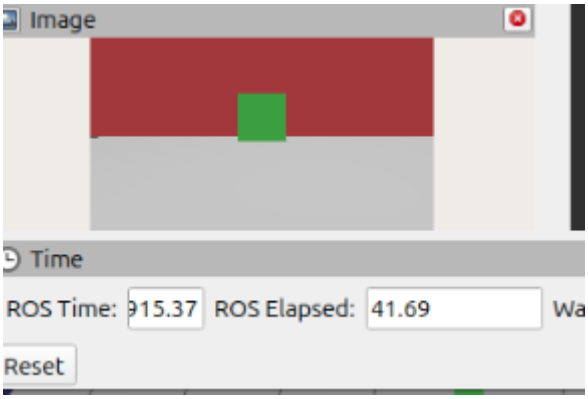

Figure 5. The image of the camera

2.2 Dynamics and Motion Control

PID Controller Plugin:

The JointPositionController plugin allows the precise motion control of each joint by tuning the proportional, integral, and derivative gains. The parameters can be tuned for flexible control of the joint position and response speed. Example:

p_gain=1, i_gain=0.1, d_gain=0.01 : Smooth and stable movement of the joint.

Inertia Tensor and Mass Definition:

For instance, the inertia tensor, such as <ixx>, <iyy>, and mass of each link are explicitly defined so that dynamic behaviors obey real-world physical laws. In this case, these settings will be crucial for dynamic simulations of tasks such as walking, grasping, and force analysis to provide realistic and predictable performance.

2.3 Plugin System

JointStatePublisher:

This plugin publishes real-time joint states, including current angle, velocity, and torque, to enable an interaction between the robot and any external system. Subscribing to this state in any ROS environment, for example, allows for the monitoring of robot behavior and feedback control, hence improving system responsiveness.

JointPositionController:

In addition to allowing PID-based control, this plugin accepts target position commands to achieve accurate joint movements. Along with the dynamic properties of the joints (e.g., torque and speed limits), the plugin is highly customizable. Such flexibility in the framework means the robot can adapt easily and efficiently to a range of simulation tasks.

It is these components and algorithms combined in code that provide not only a detailed and accurate model of the robot's physical structure and behavior but also support complex tasks such as path planning, environmental perception, and dynamic control. In this way, the system designed here provides a modular, sensor-rich, and dynamically accurate platform for testing and expanding robotic applications, allowing for easier transitions between simulations and real-world performance.

3. Evaluation and Summary

3.1 Improvements we made after our last failure

In the previous model design, we developed a simulation model of a quadruped robot inspired by a spider, drawing from elements of our own observations of real-life motion patterns. In the detailed modeling process, the robot's foot structure was first realized using points that were formed by the intersection of two arcs with different radii. As a result, each foot bottom was effectively a line segment in three-dimensional space.

However, this design, upon its simulation in the virtual environment, caused great instability in movement. Due to the limited contact surface of the paw and the ground, it is very challenging to maintain balance, especially during dynamic motion.

We re-designed the structure of the paw by adding a circular plane at the base of each leg. It enlarged the contact area between his feet and the ground; thus, this increased the stability of the robot noticeably. In our code, the enhancement was done around the core of the “fit” definition so that it fit right into the model. With that modification, the robot's motion had been stabilized, and most of the foundation grounds of locomotion behaviors and basic interactions with the virtual world had been laid. As shown in Figure 4.

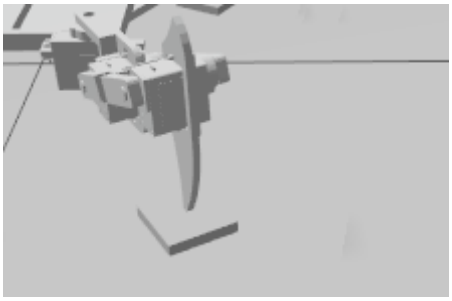

Figure 6. Robot Foot

Code:

```
<joint name='fit1' type='revolute'>
 <pose relative_to='llf3'>0 0 0.06 0 0 0</pose>
 <parent>llf3</parent>
 <child>fitl1</child>
 <axis>
   <xyz>1 0 0</xyz>
   <limit>
     <lower>-3.14</lower>
     <upper>3.14</upper>
     <effort>100</effort>
     <velocity>1</velocity>
   </limit>
   <dynamics>
     <spring_reference>0</spring_reference>
     <spring_stiffness>0</spring_stiffness>
   </dynamics>
 </axis>
</joint>
```

The revolute joint fit1 is an essential component of the robot model for rotational connection between the parent link llf3 and the child link fitl1. The design of this joint is highly flexible, with a number of carefully configured parameters that define its spatial position, rotation axis, motion range, and dynamic constraints. These settings make sure the behavior of the joint in simulation reflects real-world physical performance as accurately as possible.

fit1 spatial position is exactly aligned with regard to llf3 so that proper placement and attachment in the structure of the robot can be realized. The rotation axis is explicitly set to enable rotation regarding its purpose within the motion of the robot. Range of motion for this joint was set not to go out of reasonable limits so that behavior does not look unrealistic and would lead to unphysical unstable movements.

Dynamic constraints, such as torque limits and speed limits, are included to balance flexibility with control. These constraints allow the joint to handle the demands of complex tasks-such as adapting to uneven terrain or maintaining precise positioning-while keeping the simulation stable and manageable.

This thoughtful configuration of fit1 will support the robot's capability to perform intricate and demanding operations while preserving overall stability and controllability of the simulation. It is a basic building block of the robot's mechanical system.

3.2 Advantages and Disadvantages

This code represents several strengths. The modularity of the code is well designed with a clear hierarchy of links and joints, hence easy to extend and maintain. It offers comprehensive functionality because it integrates physical simulation, sensor systems, and motion control all together, which allows for very realistic robotic behavior. Support for simulators like Gazebo means real-time interaction out-of-the-box and easily interfaces with external tools such as ROS. Precise configuration for PID controllers and friction parameters further adds to the accuracy of motion, with close correspondence to real-world dynamics. Further Flexibility to modify the robot's structure and functions right from the SDF file, support for various simulation plugins and sensor types, and detailed material and collision properties visualizations are remarkably useful for debugging and observations.

However, this code is not without its own inadequacies. Its complexity is overwhelming for a beginner and requires a lot of effort to comprehend the overall structure and functionality. Performance limitations: High-resolution sensors, including GPU Lidar, may impact simulation speed, especially on less capable systems. In addition, some places lack comments, such as the reason for choosing PID parameters, which reduces code readability. Optimizing complexity in collision geometry and performing high-frequency updates from sensors will yield a better performance. Lastly, although the code is modular and flexible, scalability would come through implementing more types of sensors, adding controllers, and enabling the interface dynamically for adjusting parameters.

Autonomous navigation

Joint_controller.py

Introduction

This code implements the control of robot joints and obstacle detection, and automatic obstacle avoidance based on the data returned by sensors. This code imports the essential libraries and modules to create a ROS2-based quadruped robot controller. It includes the rclpy library and the Node class to define a ROS2 node, and message types like Float64 for joint control commands and LaserScan for LIDAR sensor data. Further, it utilizes the threading module for the running of some activities concurrently, such as handling ROS2 communication and user inputs all at once. These will serve as the building blocks of making a functional robot control system that is truly responsive.

1. QuadrupedSpider Class

The QuadrupedSpider class inherits from the Node of ROS2 and is responsible for creating the node joint_controller.

```
class QuadrupedSpider(Node):
    def __init__(self):
        super().__init__('joint_controller')
```

Twelve ROS publishers were created for each leg's three joints (out of a total of four legs) in the class to control joint motion.

```
      self.publishers_joints = {
            'jlf1': self.create_publisher(Float64, 'jlf1_topic', 100),
            'jlf2': self.create_publisher(Float64, 'jlf2_topic', 100),
            'jlf3': self.create_publisher(Float64, 'jlf3_topic', 100),
            'jrf1': self.create_publisher(Float64, 'jrf1_topic', 100),
            'jrf2': self.create_publisher(Float64, 'jrf2_topic', 100),
            'jrf3': self.create_publisher(Float64, 'jrf3_topic', 100),
            'jlr1': self.create_publisher(Float64, 'jlr1_topic', 100),
            'jlr2': self.create_publisher(Float64, 'jlr2_topic', 100),
            'jlr3': self.create_publisher(Float64, 'jlr3_topic', 100),
            'jrr1': self.create_publisher(Float64, 'jrr1_topic', 100),
            'jrr2': self.create_publisher(Float64, 'jrr2_topic', 100),
            'jrr3': self.create_publisher(Float64, 'jrr3_topic', 100),
        }
```

The subscriber subscribes to the LIDAR data topic lidar scan to receive laser scanning data for obstacle detection.

```
    self.lidar_subscription = self.create_subscription(
            LaserScan,
            'lidar_scan',
            self.lidar_callback,
            10
        )
```

At the same time, by using a timer, the publish_joint_positions method is called every 0.2 seconds to achieve timed release of joint control instructions.

```
  self.timer = self.create_timer(0.2, self.publish_joint_positions)  # Timer
for publishing joint positions
```

In addition, variables such as active (motion state), step (gait phase), obstacle_detected (whether obstacles are detected), and obstacle_ste (obstacle position) are defined in the class as the core state parameters of robot behavior.

```
  self.active = False
        self.step = 0  # Gait phase counter
        self.obstacle_detected = False
        self.obstacle_side = None  # Track the side of the obstacle
```

These functions lay the foundation for robot motion control and obstacle detection.

2.lidar_callback

The Lidar callback method has processed the incoming LIDAR data whenever a new scan is received. It separated the LIDAR distance data by range into left and right, subtracted 5 to each segment to avoid conflict with each other, and also computes its minimum data in every section. If an obstacle is detected within 0.3 meters, the obstacle is detected, and obstacle side variables are updated to indicate the presence of the obstacle and whether the obstacle is on the left or right side for subsequent disguised work.

```
    left_ranges = msg.ranges[:(len(msg.ranges)//2)-5]
        right_ranges = msg.ranges[(len(msg.ranges)//2)-5:]

        min_left_distance = min(left_ranges)
        min_right_distance = min(right_ranges)
```

If an obstacle is on the left with a shorter distance, the self.obstacle_detected will set True; self.obstacle_side will set 'left' or 'right', it logs the right-side obstacle. If no obstacle is detected, the state is reset. This function provides essential data for the robot to adapt its gait based on environmental conditions.

```
   if min_left_distance < 0.3 or min_right_distance < 0.3:
            self.obstacle_detected = True
            if min_left_distance < min_right_distance:
                self.obstacle_side = 'left'
                self.get_logger().info(f"Obstacle detected on the left at
distance: {min_left_distance}")
            else:
                self.obstacle_side = 'right'
                self.get_logger().info(f"Obstacle detected on the right at
distance: {min_right_distance}")
        else:
            self.obstacle_detected = False
            self.obstacle_side = None
```

toggle_active

The toggle active method toggles the state of the robot's movement. Upon invocation, it changes the active variable between True and False, which respectively turns on or off the robot's motion. If motion is turned on, it logs "Motion started.", and when motion is turned off, it logs "Motion stopped.". This simple mechanism makes the movement state of the robot dynamic for more interactive control.

publish_joint_positions

The publish_joint_positions method is the core of the robot's gait control, executed every 0.2 seconds to issue joint angle commands. It first checks the active state. If active, it generates control messages for each joint. In the absence of obstacles, it executes a four-phase gait where legs are lifted and swung in sequence, controlled by the step % 4 logic.

When obstacles are detected, the gait adapts based on the obstacle's position:

Left-side obstacle: Executes a right-turn gait.

Right-side obstacle: Executes a left-turn gait.

Joint angle commands are sent via ROS publishers to the corresponding topics. This adaptive logic allows the robot to navigate dynamically based on LIDAR feedback.

Keyboard Listener

The keyboard_listener function will start another terminal and continuously listen for user input of robot control commands. It enables users to open or close programs using's' and exit programs using 'q'. The incoming input is processed using Python's match case statement. It is called the toggle_active method, which is used to change the motion state of the robot, and in the case of input errors, it will issue an error message. It makes user tasks very simple and interactive, allowing for real-time processing of robots

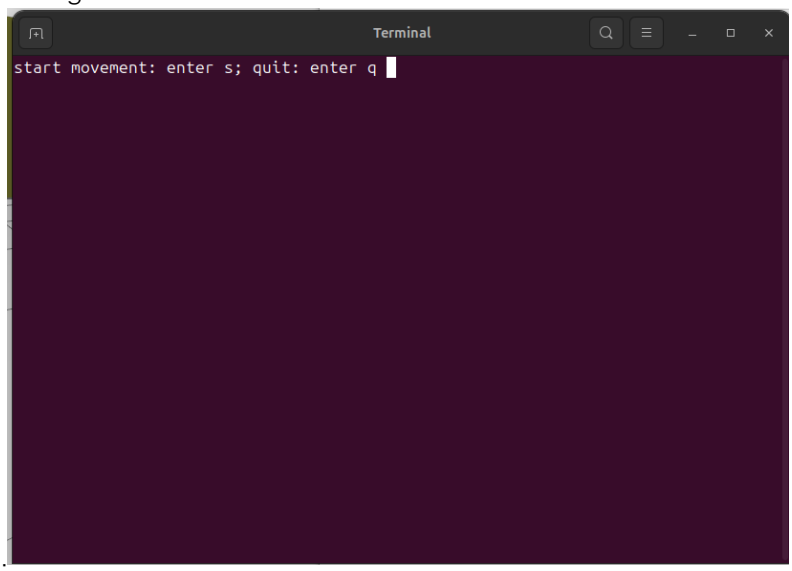

Figure 6. Keyboard Listener terminal

Image_listener.py (OpenCV)

Introduction

This program will open an OpenCV window to perform real-time image processing and shape detection based on data input from the robot camera. It utilizes the library 'rclpy' to create ROS 2 nodes and subscribe to specific image themes (/boxes_image). Use OpenCV to process the incoming real-time images to identify and annotate basic geometric shapes like rectangles and circles captured by the camera.

ImageListener Class

The ImageListener class is the core component of this application, inherited from rclpy.node. Node and implement the core functions of ROS 2 node. This type is responsible for subscribing to the specified image theme (/boxes_image) and processing the received image messages.

image_callback

The image callback function processes images coming from the /boxes_image theme and carries out several important steps towards detecting and annotating geometric shapes. First, it uses the CvBridge library to convert an incoming ROS 2 image message into OpenCV-compatible BGR images, thereby carrying out the subsequent image processing. Convert the BGR image to grayscale with cv2.cvtClor to reduce the computational complexity, then cv2.Gaussian Blur to apply Gaussian blur and smoothen out the image for minimization of noise. Now, the Canny edge detection algorithm-cv2.Canny, which extracts the outstanding edges necessary for contour recognition is applied next. Use cv2.findContours for the further processing of the edges detected to detect contours of potential shapes.

For each contour, this function uses the cv2.approxinPolyDP polygon approximation to simplify the shape, reducing the number of vertices to analyze.

Shapes are categorized depending on their geometrical shape: contours with four vertices are recognized as rectangles that are outlined with a bounding box in green color and labeled as "Rectangle." Smooth curves with more than eight vertices are considered circles and were approximated using cv2.minEnclosingCircle, labeled in blue color with "Circle" label. The resulting image is the annotated shapes presented in runtime through the function call of cv2.imshow for visualization. In addition, it has a powerful error handling to log any exceptions that would occur while processing, in order to maintain the application alive and debuggable.

The image callback function processes images coming from the /boxes_image theme and carries out several important steps towards detecting and annotating geometric shapes. First, it uses the CvBridge library to convert an incoming ROS 2 image message into OpenCV-compatible BGR images, thereby carrying out the subsequent image processing. Convert the BGR image to grayscale with cv2.cvtClor to reduce the computational complexity, then cv2.Gaussian Blur to apply Gaussian blur and smoothen out the image for minimization of noise. Now, the Canny edge detection algorithm-cv2.Canny, which extracts the outstanding edges necessary for contour recognition is applied next. Use cv2.findContours for the further processing of the edges detected to detect contours of potential shapes.

For each contour, this function uses the cv2.approxinPolyDP polygon approximation to simplify the shape, reducing the number of vertices to analyze.

Shapes are categorized depending on their geometrical shape: contours with four vertices are recognized as rectangles that are outlined with a bounding box in green color and labeled as "Rectangle." Smooth curves with more than eight vertices are considered circles and were approximated using cv2.minEnclosingCircle, labeled in blue color with "Circle" label. The resulting image is the annotated shapes presented in runtime through the function call of cv2.imshow for visualization. In addition, it has a powerful error handling to log any exceptions that would occur while processing, in order to maintain the application alive and debuggable.

Limitations

Although recognition of cubes and spheres can be observed from the openCV window during runtime. However, due to the intense shaking of quadruped robots, the recognized images are quite unstable. Optimizing the gait of quadruped robots to reduce shaking during movement will help solve related problems. The actual operating situation is shown in Figure 7.

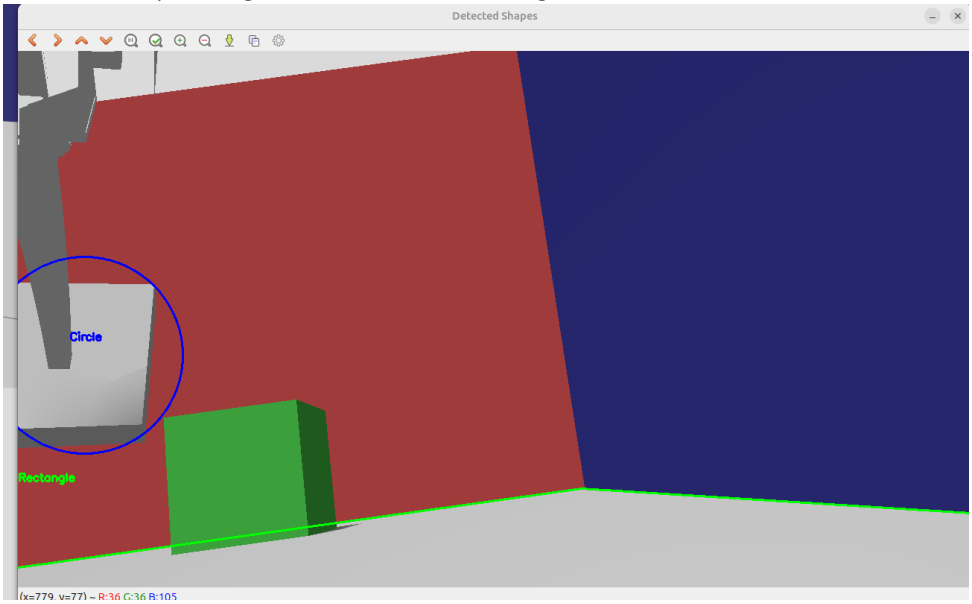

Figure 7. The actual operation of object recognition

Map generation

Map generation involves using sensor data, such as LiDAR scans, to create the spatial representation of the environment-such as 2D occupancy grids. Map generation is thus a very crucial aspect when it comes to robotics regarding navigation, path planning, and obstacle avoidance. In ROS, the common SLAM algorithm that is used for map generation includes gmapping, which requires laser scan data from the topic called /scan. Although we attempted to launch 'slam_toolbox' to produce maps, however, in this case LiDAR publishes on '/lidar_scan', which is probably not directly compatible with how gmapping expects it, hence the generation of the map fails.

The main challenges are mismatch of topic names, possible incompatibility in message types, and missing or wrong transformations (tf) between the LiDAR frame and the robot base frame. Moreover, differences in sensor configuration (e.g., resolution or range) may need tuning of gmapping's parameters with respect to the LiDAR specifications.

The topic can be remapped from /lidar_scan to /scan, and data conversion can be implemented if that is what is necessary for it to agree with the sensor_msgs/LaserScan format. Verify accurate frame transformations, since gmapping relies on accurate pose information. Otherwise, explore other SLAM packages, such as Hector SLAM or Cartographer, which is more flexible regarding input data.

By remapping topics, adjusting parameters, and considering alternative mapping frameworks, the robot can successfully generate maps, enabling effective navigation and understanding of its surroundings.

Launch.py

1. Objective

The primary goal of the `launch.py` script is to enable the simulation of a quadruped robot within a virtual environment. Additionally, it sets up the necessary framework for the robot to perform key tasks, including movement in four directions, obstacle avoidance, and environmental mapping using integrated sensors and algorithms.

2. Functionality Overview

2.1 Simulation Environment Setup

To begin with, the script integrates seamlessly with Gazebo, launching a simulation environment defined by the `peter_world.sdf` file. Alongside this, it employs the `ros_gz_sim` package to spawn the robot (`peter1`) into the environment, ensuring that the robot is properly initialized within the simulation.

2.2 Sensor Integration

On one hand, the script incorporates GPU Lidar, simulating distance measurements crucial for SLAM and obstacle detection. On the other hand, it utilizes a bounding box camera to provide visual data, enabling tasks such as object detection, navigation, and planning.

2.3 State Management

To manage state efficiently, the `robot_state_publisher` node is employed, broadcasting the robot's status for other nodes and tools. Moreover, the `ros_gz_bridge` ensures smooth communication between Gazebo and ROS 2 topics, creating a unified framework for data sharing. Lastly, the `GZ_SIM_RESOURCE_PATH` environment variable is set to point to the required simulation resources, avoiding any runtime errors.

2.4 Real-Time Visualization and SLAM

For visualization, the script launches RViz using a preconfigured file (`peter1.rviz`), offering real-time insights into the robot's status. At the same time, it deploys an asynchronous SLAM node from the SLAM Toolbox, which try to processes Lidar data to build dynamic maps of the environment, enabling real-time navigation and obstacle avoidance.

3. Key Algorithms and Logic

3.1 Modularity and Design

The script's architecture is highly modular, with nodes assigned specific tasks such as publishing state information, mapping, and visualization. By leveraging Python's `launch` library, the simulation setup is further streamlined, allowing developers to define arguments and modularize components with ease.

3.2 Sensor Handling and Mapping

When it comes to sensor management, the parameter bridge plays a pivotal role, connecting topics like Lidar, camera, and joint states between Gazebo and ROS 2. This connection facilitates real-time processing of sensor data. Additionally, the SLAM algorithms operate on
the `/lidar_scan` topic, translating sensor input into actionable environmental maps that guide robot navigation.

3.3 Node Execution and Commands

To ensure smooth execution, the script uses the `ExecuteProcess` action to initiate the `joint_controller` and `image_listener` nodes. These nodes run in separate terminal windows, enabling parallel processing of robotic control and image data, which ultimately boosts the system's overall efficiency.

4. Improvements Made

Stability Enhancements

In response to the instability observed in earlier designs, the robot's foot components were reengineered by adding circular planes to the ends of its legs. This enhancement significantly improved the robot's stability during movement. Furthermore, collision and friction definitions were refined to achieve more realistic interactions with the environment.

Scalability and Flexibility

The design also focuses on scalability, with its node-based structure allowing for straightforward integration of new components, such as additional sensors or updated control algorithms. On top of that, the PID-based joint control ensures that the robot's movements are both precise and adaptable, catering to diverse operational scenarios.

5. Evaluation and Potential Improvements

Strengths

One of the standout features of this script is its modular design, which makes it easy to extend or customize for different use cases. In addition, its realistic simulation capabilities—integrating sensors, physics, and visualization—offer a robust framework for testing robotic functionalities. Lastly, the precise joint control mechanisms, including well-defined limits and dynamics, ensure accurate and reliable robotic operations.

Weaknesses

Despite its strengths, the script faces challenges in performance when handling high-resolution sensors like the GPU Lidar, which can slow down real-time simulations. Furthermore, the layered and complex structure of the code may be intimidating for newcomers, requiring a steeper learning curve to understand and adapt.

Suggestions for Improvement

To address performance concerns, reducing the resolution or update frequency of the sensors could lead to more efficient simulations, especially in resource-constrained environments. Better documentation, with detailed comments explaining specific parameters and sections of the code, could also enhance its accessibility. Finally, integrating dynamic configuration tools would enable real-time parameter adjustments, making the simulation more interactive and user-friendly.

Reflection

This project proved to be difficult with several things that slowed down the work and efficiency of the group. The first problem we had was that since assignment 2 is built upon the foundations of assignment 1, assignment 1 must be completed well to create a good foundation to build upon. However, our group struggled with assignment 1 which put us on the backfoot for this assignment. We had to spend the first week or so of this assignment on completing assignment 1 properly to even begin assignment 2. This immediately put us behind the curve of assignment 2 and over a week behind by the point we could start it. One of the main problems we highlighted in assignment 1 with our group was time management and procrastination. To tackle this problem, we attempted to create a breakdown of work packages and deliverables week on week to ensure we don't fall behind. However, that proved to be futile as it was not followed by everyone after the first couple weeks so it was discarded as the time spent on it did not pay off and would have been better spent on project work directly. Most of our productive work was done in the final week of the project, even though we were working consistently throughout we managed to get a breakthrough in the final lab session with a collection of systems now working together.

In the process of designing and developing a simulation environment for quadruped robots in this task, we have achieved significant results, but also exposed some areas that need improvement:

Balance between physical parameters and simulation accuracy

During the process of configuring the world file, we have learnt that although the high-precision time step setting (1000 Hz) improves simulation accuracy, it leads to a decrease in computational performance in complex scenarios. In the future, it is necessary to explore methods for dynamically adjusting time steps to achieve an optimized balance between accuracy and computational performance.

Stability of quadruped robot motion

The small foot contact area in early designs resulted in unstable movement. By improving the design by increasing the foot contact area, the stability of the robot during movement has been significantly enhanced.

Sensor data processing

Integrated GPU LiDAR and Bounding Box camera to achieve environmental perception function. However, in complex scenarios, real-time camera image data exhibits certain fluctuations, requiring further optimization of data filtering and fusion techniques, as well as optimization of robot gait codes.

Automatic navigation and obstacle avoidance performance

In terms of obstacle avoidance function, the robot can adjust its path in real-time based on LiDAR data. However, in complex obstacle environments, the smoothness of navigation paths still needs improvement.

Conclusion

Through this project, we have successfully constructed a modular and high-precision robot simulation environment, including ground, walls, various shaped obstacles, and environmental light sources. We have also developed robot control code based on ROS2 to achieve walking control and obstacle avoidance functions for quadruped robots. Additionally, we have integrated LiDAR and Bounding Box cameras, significantly improving environmental perception and real-time data processing capabilities. This platform provides a high-precision simulation tool that can be used to test the dynamic behavior and algorithms of robots. Its modular design facilitates quick adaptation to different types of robots and sensor requirements. In the future, this platform can be further expanded to more complex simulation environments, such as multi-layer dynamic scenes, and by integrating deep learning and reinforcement learning techniques, the intelligence and real-time performance of path planning algorithms can be further improved.

Lessons learned and skills acquired

This project has brought significant technological growth and valuable experience accumulation to the team. At the technical level, we have a deep understanding of the SDF file format and its application in robot simulation. We are proficient in using the ROS2 framework to implement node communication and sensor data processing, and have completed real-time image processing and enhanced visual data processing capabilities using OpenCV. At the same time, we have mastered the configuration and optimization of physical parameters in the simulation environment. In terms of team collaboration, efficient version control tools have been used to achieve orderly management of task division and problem tracking, and agile development methods have been adopted to ensure project progress and quality. In terms of problem-solving, we improved motion stability through iterative testing and optimized sensor

data flow to enhance the accuracy of the obstacle avoidance system. In addition, we have gained a deeper understanding of the overall architecture of robot systems, recognizing the collaborative effects between various modules such as hardware structure, motion control, environmental perception, and user interaction, accumulating rich experience and technical reserves for the development of future complex systems.

Division of responsibilities

Zhibin Mo: Code design, model design, environment configuration, environment testing, report writing

Haoran: gmapping design, sdf code analysis, main writing of reports

Alex: Report writing, research launch files

Aidan: Report polishing, designing World model