Selected Technical Contributions

Quadruped Robot Simulation Integration

(ROS 2 Jazzy + Gazebo Harmonic)

Prepared by: Alireza Sharifan
Independent Robotics Integration Engineer (MSc. Mechanical. Eng.)

Citation: Sharifan, A. (2025). Quadruped Simulation Suite: Initial Release.
DOI: 10.5281/zenodo.16916383

Objective

Evaluate open-source ROS 2 simulation assets for quadruped robots and integrate them into a modern ROS 2 Jazzy + Gazebo Harmonic pipeline, enabling simulation, sensor emulation, and presentation-ready demonstrations. The project addressed plugin compatibility gaps and extended sensor analysis capabilities beyond existing public documentation.

Technical Contributions:

1. Model Visualization Pipeline

1.1. Loaded go2.usd in Blender from unitree_model.

1.2. Verified mesh integrity using Blender (imported USD, exported DAE/OBJ/STL).

1.3. Rendered the Go2 robot model in both RViz and Gazebo Harmonic using various mesh formats (STL, OBJ, and DAE).

2. URDF Integration and TF Frame Validation

2.1. Integrated go2.urdf from unitree_rl_gym, with corrected mesh paths to ensure compatibility.

2.2. Rendered the URDF model in both RViz and Gazebo (static view).

2.3. Confirmed active joint and TF publishers via rqt_graph during system bring-up.

2.4. Echoed /robot_description to ensure the URDF was correctly published from Gazebo.

2.5. Echoed /tf_static to verify that all fixed joint frames were correctly broadcasted.

2.6. Echoed /tf to confirm dynamic joint transforms during interaction.

2.7. Used joint_state_publisher_gui to manually configure joint angles and observe real-time TF updates.

2.8. Performed joint configuration testing, ensuring that joint interactions reflected intended poses.

The resulting TF matrix and RPY values were monitored to validate the joint-level transformations.

2.9. Verified forward kinematic pose consistency, confirming that dependent links responded correctly to parent joint movements, in line with the robot's kinematic hierarchy.

2.10. Generated a TF tree PDF to visually confirm complete frame connectivity with no orphan nodes.

3. Custom TF Frame Insertion via Mesh-Based Estimation

3.1. Precisely calculated transform in Blender by directly measuring the base.dae mesh geometry.

Final pose relative to the base frame: x = 0.230312, y = 0, z = 0.088754, roll = 0, pitch = 0, yaw = 0

3.2. Manually added a virtual camera frame to the Go2 robot using static_transform_publisher.

3.3. Visualized the custom frame in RViz to confirm alignment and placement accuracy relative to the base link.

4. Motion Control Experiments (Attempted)

Tried 4 different environments to integrate ros2_control:

4.1 Ubuntu 24.04 + ROS 2 Jazzy + Gazebo Harmonic
libgz_ros2_control.so missing; manual build failed due to ROS 2 Jazzy & Harmonic API mismatch.

4.2 Docker: ROS 2 Foxy + Gazebo Classic
URDF failed in Classic due to missing or unbuildable ROS 2 plugin support under Humble.

4.3 Docker: ROS 2 Humble + Gazebo Harmonic
Reached technical dead-end due to outdated repos and lack of support for newer toolchain versions.

4.4 Docker: ROS 2 + Working Plugin
Launch failed due to .xacro path not resolving inside Docker despite multiple rewrites.

Conclusion:
Despite methodical experimentation across multiple environments, full joint actuation remained blocked due to version-specific plugin incompatibilities and missing toolchain support — highlighting current ecosystem limitations rather than implementation errors.

Sensor Data Analysis Context

Due to limited actuator control in simulation (e.g., ros2_control was not yet functional), real sensor motion feedback was unavailable. Therefore, both real-world simulation triggers (via SDF) and mock publishers were used to test the full sensor data pipeline in ROS 2.

5. LiDAR Sensor Data Analysis (real and mock)

5.1 Real SDF Test

5.1.1. For the sensor simulation, the torso section of the robot was converted from URDF to SDF—including the base, the upper and lower head segments, and the LiDAR mount—using the actual mesh file base.dae.

A front-facing LiDAR sensor of type gpu_ray was configured with angular and range parameters aligned with the Livox L1 model used in the real platform, providing a representative scan behavior for simulation purposes.

The vertical offset was carefully computed from the URDF file based on the full leg joint chain, from hip to foot, to reflect the LiDAR’s real-world mounting height.

5.1.2. Simulated a dynamic obstacle in the SDF world by using the Translation Gizmo along the X-axis to move a box-shaped object back and forth in front of the stationary robot, generating /scan data from the LiDAR sensor.

5.1.3. Tracked real-time changes in LiDAR scan data during obstacle motion using rqt_plot and terminal output from the /scan topic (bridged via ros_gz_bridge). The front-facing beam was inferred based on the LiDAR’s angular configuration and scan index geometry.

5.2 Mock Scan Test

5.2.1. Placed a virtual wall marker 1.5 m ahead of the LiDAR sensor in RViz to visually confirm scan response at the defined proximity threshold.

The placement distance (1.5 m) was derived from the computed link and joint offsets of the sensor mount.

5.2.2. Injected synthetic laser scan messages to simulate proximity alert behavior at a 0.2 m threshold, which was visually confirmed in RViz by the front-facing LiDAR beam, configured with the same SDF parameters as in the real-sensor simulation (Livox L1), stopping in front of the virtual obstacle (wall marker).

5.2.3. Triggered terminal alerts on close-range detection.

5.2.4. Recorded /scan topic data and extracted timestamps, time intervals, range count, mid-beam range, jitter, dropouts, and number of points per scan, into a CSV file.

5.2.5. Plotted LiDAR middle-beam range over time, scan time deviations (jitter and drops), and point count per scan to assess temporal consistency, signal stability, and data completeness.

5.2.6. Performed automated anomaly detection based on inter-scan timing and beam range discontinuities, with a custom Python script (pandas/matplotlib).

5.2.7. Interpreted and documented LiDAR plots to support developer teams and inform sensor evaluation processes.

5.2.8. Identified misalignment in LiDAR mounting in go2.urdf and proposed corrections.

6. IMU Sensor Data Analysis (mock)

6.1. Visualized real-time IMU data (acceleration and orientation) in RViz from the mock sensor mounted on Go2 in Gazebo.

The synthetic IMU data was generated based on typical specifications inspired by the Bosch BMI088, a widely used 6-axis inertial measurement unit in robotics applications.

6.2. Visualized a fixed forward-facing pose in RViz using the goal_pose_tf_broadcaster node to conceptually align the simulated heading direction, thereby validating the consistency of the mock IMU data simulating forward motion along the X-axis.

6.3. Recorded /imu topic data and extracted timestamps, time intervals, 3D acceleration, 3D angular velocity, sensor biases, estimated linear velocity (via numerical integration), pose, and orientation into a CSV file.

6.4. Visualized raw IMU data from the /imu/data topic, including unprocessed 3D linear acceleration and angular velocity. This data contains inherent sensor noise (included in the visualization of Section 6.7).

6.5. Performed static bias estimation by averaging the raw IMU output from the /imu/data topic over low-motion (operational stationary) segments in the initial part of the recording, and applied offset correction to both accelerometer and gyroscope data to eliminate constant offset and improve accuracy in estimating velocity and position (Used as the baseline in Section 6.7).

6.6. Performed dynamic bias estimation based on rolling average smoothing of the bias-corrected IMU signals (obtained as described in Section 6.5) and applied numerical integration (Euler/rectangular method) to reduce high-frequency sensor noise and limit cumulative integration errors, helping produce more accurate velocity and 2D trajectory estimates (as visualized in Section 6.7).

6.7. Plotted IMU time-series data including raw 3D linear acceleration and gyroscope measurements (as recorded in Section 6.3), 3D linear acceleration and gyroscope measurements (before and after static bias correction), smoothed 3D linear acceleration and gyroscope data, inter-sample time differences (Δt), integrated 2D trajectory (X–Y), and estimated linear velocities (computed from smoothed acceleration data as described in Section 6.6), along with orientation in both Euler (RPY) and quaternion formats.

All data processing, bias estimation, and visualization in sections described above were implemented with a custom Python script (pandas/numpy).

6.8. Interpreted and documented IMU plots to support developer teams and inform sensor evaluation processes.

6.9. Analyzed the IMU frames defined in go2.urdf and recommended alignment refinements for improved pose stability.

7. Sensor Fusion with EKF (Single IMU Sensor)

7.1. In the absence of actuator control and external localization (e.g., GPS or odometry sources), a mock IMU-based dead-reckoning approach was implemented. Pose estimation was derived by integrating acceleration and orientation data, enabling real-time RViz visualization of the robot's inferred trajectory.

7.2. Although only one IMU sensor was available, a 9-state Extended Kalman Filter (EKF) was implemented using the robot_localization package to fuse acceleration and orientation data and estimate pose via dead-reckoning.

7.3. Recorded /ekf_odom topic data and extracted timestamps, inter-sample time intervals (Δt), 3D angular velocity, estimated sensor biases, linear velocity, and orientation into a structured CSV file.

7.4. Plotted EKF time-series data including gyroscope measurements corrected for static bias (as computed in Section 6.5), inter-sample time differences (Δt), linear velocities, and orientation in both Euler (RPY) and quaternion formats. Compared EKF results with the corresponding bias-corrected IMU data to evaluate consistency and the effectiveness of the EKF in attenuating sensor noise.

All EKF data analysis and visualization were implemented with a custom Python script (pandas/matplotlib).

7.5. Visualized EKF-based odometry in RViz to monitor real-time trajectory estimated from IMU dead-reckoning.

7.6. Interpreted and documented EKF plots to support development teams and inform sensor evaluation processes.

8. Sensor Behavior Extensions (Planned & Provisioned)

8.1. LiDAR-Based Motion Response (Interface-Ready, Behavior-Pending)

The LiDAR alert node is architected with optional behavior-control logic for motion responses:

8.1.1. Immediate halt via /cmd_vel with zero linear and angular velocity.

8.1.2. Directional evasion (e.g., rotate left) via a fixed angular twist command.

These motion commands are currently excluded from deployment, not due to lack of functionality, but to maintain testing isolation between perception and control modules in the current simulation phase. This design allows easy extension into teleop safeguards, autonomous navigation stacks (e.g., Nav2), and SLAM with minimal code change.

8.2. IMU Saturation Detection (Design Ready, Data-Limited)

A saturation detection mechanism was designed to identify accelerometer samples exceeding ±16g thresholds, typically indicating sensor clipping in high-impact motion (e.g., the BMI088 supports up to ±24g).

However, due to the absence of official IMU specifications (e.g., full-scale range) for the actual hardware, despite simulating based on typical BMI088 specifications, this logic has been intentionally held in reserve, awaiting verified sensor parameters to avoid false positives and ensure meaningful diagnostics.

8.3. More Accurate Integration Methods for Path and Velocity Estimation

Velocity and trajectory were computed using rectangular (Euler) integration. The system supports future upgrades to higher-order methods like trapezoidal or Runge-Kutta (RK4) for improved accuracy if required.

8.4. Sensor Fusion and Filter Extension for Future Integration

8.4.1 The robot_localization package supports a 15-state EKF model that fuses IMU, GPS, wheel odometry, and velocity data for full-state estimation. While currently limited to IMU-only dead-reckoning, the sensor design system is prepared for future expansion once compatible motion control and sensor plugins are available.

8.4.2. While the current EKF works best with predictable sensor data, the system is flexible enough to support more advanced filters, such as Unscented Kalman Filters (UKF) or Particle Filters, once realistic control (via ros2_control) and sensor plugins are integrated, especially for handling non-Gaussian noise and more complex localization requirements.

For corresponding visual references, please refer to the README of the GitHub repository.

Note: All simulation activities were preformed on publicly available open-source assets for the purpose of presenting technical capabilities.