Autonomous Hockey Robot in Simulation Report

GitHub Repo: https://github.com/WA6S7/autonomous-hockey-robot

Introduction

This project focuses on the design and implementation of an autonomous robotic system in simulation. Working collaboratively as a team, we aimed to build a robot that can sense its environment, make decisions, and act autonomously. This project involves the development of a simulated hockey-playing robot using ROS2 Jazzy and Gazebo. The robot operated within a virtual arena where it must detect and track a ball, navigate safely while attempting to score goals without human intervention.

Simulation & Robot Modelling

By Nikolina Filipov Pajic

My responsibility focused on the design and implementation of the simulation environment, robot modelling, and system integration within ROS2 and Gazebo. This work is represented by the following key files, each corresponding to a specific aspect of the system.

Simulation Environment

The virtual hockey arena was created and configured in the file `hockey_arena.sdf`. The file defines the environment, including walls, goals, and boundaries, as well as physics properties required for a realistic interaction between the robot and the puck. The arena serves as the primary testing and demonstration environment for autonomous behaviour.

Robot Modelling

The robot's physical structure and sensor configuration are defined in `hockey_robot.urdf.xacro`. The file describes the robot's base, wheel joints, caster links, and mounting points for the LiDAR and IMU sensors. Xacro macros were used to allow parameterisation and reuse, enabling the 2 robots to be spawned with unique prefixes while maintaining a single robot description.

Robot Spawning and Simulation Launch

The launch file spawn_robotcs.launch.py is responsible for starting the Gazebo simulation, loading the hockey arena, and spawning the 2 robotics with separate namespaces. It also launches the robot_state_publisher nodes required for publishing the robot's TF tree, ensuring correct coordinate transformation for sensor data and navigation

ROS-Gazebo Communication

Sensor data and control commands are exchanged between Gazebo and ROS2 using the bridge configuration defined in `bridge.launch.py`. This file establishes the necessary topic bridges for LiDAR scans, IMU data, and velocity commands (`/cmd_vel`), allowing higher-level perception and control nodes to interact with the simulated robot in real time.

Package Structure and Build Configuration

The files `package.xml` and `CMakeLists.txt` define the package metadata dependencies, and installation rules for both the robot description and the simulation world. These files ensure that the project follows ROS2 best practices and can be built and launched consistently across different systems.

LiDAR Puck Detection and Tracking

By Jeremy Galea

My contribution to the project focuses on LiDAR-only puck perception and a single bringup launch entrypoint that instantiates the full autonomy stack for both robots, so we do not need to stay using a multi-terminal system.

Puck Detection and Tracking

The goal of this node is to detect the puck directly from a 2D 'LaserScan' and publish its estimated position as a PoseStamped in the scar frame. Because the sensor is a single scan plane, the puck which originally imitated a real-life thin cylinder was changed to a ball to make it easier to detect and track, but will still be referred to as puck. Although the LiDAR mounting height is supposed to be at the diameter of the puck, the puck may still appear smaller than its physical diameter due to some minor inaccuracies in the LiDAR mounting height, and therefore, the detector is parameterised using an expected apparent diameter in the scan plane rather than assuming the true object diameter.

The following pipeline was adapted for the detection of the puck:

1. The node subscribes to a configurable scan topic '`scan_topic`' and filters raw ranges using `range_min` and `range_max` to reject the invalid and irrelevant returns before geometric processing.
2. Each valid beam is converted from polar coordinates to `(range, angle)` to Cartesian `(x, y)` in the LiDAR frame. This representation makes clustering and size estimation straightforward in metric space.
3. The projected points are then segmented into clusters using a distance-based split rule, so when the gap between consecutive points exceeds `cluster_gap`, a new cluster is started. Invalid points are skepped to make clustering robust to missing returns.
4. For each cluster with at least `min_points` points, the node computes a scalar width using the diagonal of the cluster's axis-aligned bounding box. Clusters are accepted

only if their width is within `puck_diameter ± diameter_tol`. Among the valid candidates, the best cluster minimises a score that prioritises diameter agreement and slightly prefers nearer objects, reducing false positives from similarly-sized clutter further away.

5. The selected cluster centroid is finally published as PoseStamped on puck_pose_topic. This output is directly consumable by the downstream control logic without requiring additional transforms.
6. To stabilise the puck estimate and bridge short detection dropouts, the node includes a constant-velocity Kalman Filter with state:

$$x = \left[p_x, p_y, v_x, v_y\right]^T$$

Full-System Bringup

By Jeremy Galea

While the simulation environment and bridges can be started independently, the project benefits from a single launch entrypoint that reliably boots the autonomy stack for both robots with correct namespacing and topic wiring. The bringup launch file provides that integration layer. This script publishes a puck pose estimate per robot namespace from raw LiDAR scans, and guarantees the correct launch order and correct parameterisation for the multi-robot operation, so each controller consumes the correct puck estimate and scan stream.

AI Control & Behaviour

By Liam Jake Vella

My contribution to the project focused on the design and implementation of the Finite State Machine (FSM) and the autonomous strategy that runs the robot's decision-making. This logic is programmed in the `simple_fsm.py` node, which processes perception data to execute manoeuvres such as locating the puck, approaching it, and avoiding obstacles.

Finite State Machine (FSM) Implementation

The robot's autonomy is driven by three primary states designed to ensure reliable interaction with the puck while maintaining safety within the arena:

- **SEARCH**: If the puck is not visible (no data received for over 1.0 second), the robot rotates at a constant angular velocity (0.8 rad/s) to scan the environment.
- **APPROACH**: Once the puck is detected via the LiDAR-based `puck_pose_lidar` topic, the robot uses a proportional (P) controller to align its heading with the puck and drive toward it. The linear velocity is dynamically adjusted based on distance, ensuring it maintains sufficient speed (0.3m/s minimum) to effectively "hit" or push the puck.
- **AVOID**: This state takes the highest priority. If the LiDAR detect returns an object within 0.35m that is not identified as the puck, the robot executes a recovery manoeuvre by backing up and turning to prevent collisions with walls or the opponent.

Control Logic and Navigation

The control node subscribes to the estimated puck position (`PoseStamped`) and raw `LaserScan` data. Key technical features of the implementation include:

- **Obstacle Discrimination**: The logic compares the distance and angle of LiDAR returns against the known puck position. This prevents the robot from misidentifying the puck as an obstacle, allowing it to get close enough to score while still avoiding walls.
- **Velocity Commands**: Decisions are translated into `geometry_msgs/Twist` commands sent to the `/cmd_vel` topic, which Gazebo uses to move the simulated robot.
- **Namespacing**: The node is designed to handle parameters like `robot_name` and `puck_topic`, allowing the same control logic to be deployed across multiple robots in the arena without conflict.

Known Issues

1. Due to issues with LiDAR, a ball was used instead of a Hockey Puck
2. Due to time constraints and other assignments, the ball is unable to move around the arena. Troubleshooting steps included using a custom utility script, `puck_kicker.py`, which bypasses standard physics interactions by calling the `gz service` to apply a high-force burst (500N) directly to the puck's link.