Intelligent Robotics Assignments Report

Pooya Nasiri [*], Arash Abdolhossein Pour Malekshah [†]

June 27, 2024

GROUP 43

- Pooya Nasiri
  Email: pooya.nasiri@studenti.unipd.it
  Arash Abdolhossein Pour Malekshah
  Email: arash.abdolhosseinpourmalekshah@studenti.unipd.it

- Bitbucket repository:
  `https://bitbucket.org/ir2324-group-43/ir2324_group_43/`
- Google Drive folder:
  `https://drive.google.com/drive/u/0/folders/1xXpOcEI8CoQ80vXw3xNiskqTXxp1L2BJ`

1 Introduction

In this project, we developed a multi-node ROS application for guiding a robotic platform through navigation, object detection, and manipulation tasks. The system consists of three primary nodes:

- **nodeA.cpp**: Responsible for navigation and obstacle avoidance.
- **nodeB.cpp**: Focuses on object detection using AprilTags.
- **nodeC.cpp**: Handles object manipulation and placement.

Two launch files simplify system execution: `mylaunch.launch` and `launch nodes.launch`.

1.1 Project Objectives

Objectives include developing navigation, object detection, and manipulation systems, ensuring seamless node communication via ROS.

1.2 System Architecture

Modular architecture enhances code clarity and maintainability, with each node performing distinct tasks.

1.3 Implementation Overview

Steps involved adaptation of existing code, development of custom services, integration of MoveIt!, and utilization of AprilTags.

2 Node A: Navigation and Obstacle Avoidance

The primary objective of `nodeA.cpp` is to guide the robot to a specific pose in front of the table using custom services and action clients for navigation and obstacle avoidance.

[*] Email: pooya.nasiri@studenti.unipd.it — Matricola 2071437
[†] Email: arash.abdolhosseinpourmalekshah@studenti.unipd.it — Matricola 2080671

2.1 Service Client for Obstacle Detection

`nodeA.cpp` includes a custom `ServiceClient` that communicates with `tiago_iaslab_simulation::ObstacleDetectionActi` for dynamic obstacle management.

2.2 Navigating Around Obstacles

Challenges like navigating around a cylindrical obstacle are addressed using partial pose adjustments.

2.3 Adjusting Torso Height with MoveIt!

The node optimizes object detection by adjusting the robot's torso height using MoveIt!.

2.4 Manipulating the Robot's Head

A `HeadMover` class manipulates the robot's head position to enhance object detection accuracy.

2.5 ROS Publishers and Message Handling

Several ROS publishers and message handlers coordinate object detection processes.

2.6 Completion Signal to Node B

After setup tasks, `nodeA.cpp` signals `nodeB.cpp` to commence operations, ensuring synchronized task execution.

2.7 Implementation Details

Key steps in implementing `nodeA.cpp` include setting up the service client, managing navigation goals, obstacle navigation logic, torso adjustment, head movement control, ROS communication setup, and task completion signaling.

3 Node B: Object Detection and Localization

The primary objective of `nodeB.cpp` is to identify and determine the positions of objects on the table using AprilTag detections. This node processes visual information from the robot's camera and communicates results to `nodeC.cpp` for manipulation tasks.

3.1 AprilTag Detection

`nodeB.cpp` subscribes to `tag_detections` to receive AprilTag detections, ensuring robust object identification and localization.

3.2 Tag Callback Function

The core `tagCallback` function transforms detected poses into the global map frame, accurately mapping object positions.

3.3 Storing Object Coordinates

Detected object coordinates are stored in the `objects` array, enabling `nodeB.cpp` to manage and track identified objects.

3.4 Initiation by Node A

`nodeB.cpp` waits for a signal from `nodeA.cpp` to begin object detection, ensuring synchronization with robot positioning.

3.5 Requesting Object Order

Upon initiation, `nodeB.cpp` requests object pick order via `/human_objects_srv`, enhancing system adaptability.

3.6 Multi-View Object Detection

Utilizing a multi-view strategy, `nodeB.cpp` ensures comprehensive object detection across different viewpoints.

3.7 Preparing Data for Node C

Finalized object information is encapsulated in `tiago_iaslab_simulation::pick_info` messages for `nodeC.cpp`, facilitating object manipulation tasks.

3.8 Implementation Details

Implementation steps include setting up subscription to `tag_detections`, processing detections with `tagCallback`, storing coordinates, signal-based initiation, human operator interaction for object order, multi-view detection strategy, and message preparation for `nodeC.cpp`.

4 Node C: Object Manipulation and Placement

The primary objective of `nodeC.cpp` is to manipulate and place objects identified by `nodeB.cpp` with precision and accuracy.

4.1 Receiving Data from Node B

`nodeC.cpp` receives `tiago_iaslab_simulation::pick_info` messages from `nodeB.cpp` containing object coordinates and pick order.

4.2 Planning Scene Interface

To ensure collision-free manipulation, `nodeC.cpp` sets up a planning scene interface with collision objects derived from `nodeB.cpp`.

4.3 Determining Optimal Table Side

Based on object positions, `nodeC.cpp` decides the optimal side of the table to start operations for efficient object handling.

4.4 Pick-and-Place Cycle

`nodeC.cpp` executes a cycle of pick-and-place operations:

1. Transforming coordinates to `base_footprint` frame.
2. Positioning end effector above the object.
3. Attaching and lifting the object safely.
4. Transporting the object to a visible position using movement clients.
5. Locating target tables with laser scan data and placing the object.
6. Preparing for subsequent object manipulations.

4.5 Laser Scan and Table Detection

Utilizes laser scan data to detect and locate tables in the room, converting polar coordinates to Cartesian for accurate positioning.

4.6 Implementation Details

Implementation includes data reception from `nodeB.cpp`, planning scene setup, optimal positioning determination, pick-and-place execution, and laser scan processing for table detection.