

Assignment1

Bino Matteo 2158293	matteo.bino@studenti.unipd.it
Caregnato Gianluca 2157859	gianluca.caregnato@studenti.unipd.it
Meneghetti Federico 2157857	federico.meneghetti.2@studenti.unipd.it
Github:	group_number_4_assignment_1
Video:	Video_group_number_4_assignment_1

1 Introduction

We developed a robotic system capable of autonomous navigation and perception in a simulated environment. The main objectives of the project were to estimate the robot's final goal pose by detecting two fixed AprilTags viewed from a camera, to implement the standard navigation strategy (ROS2 nav2) to let the robot navigate to the goal position and eventually implement a custom controller (bonus points), and finally to detect tables (cylindrical obstacles) from any available sensor and output their positions and radii.

The following sections detail the project design in matter of package and code division, as well as the algorithms and logic used to achieve these tasks.

2 Robot goal pose

The package `find_goal` has the objective to identify the robot's goal position (final position) as the midpoint between two AprilTags visible from a service camera.

2.1 AprilTags detection

To identify the AprilTags, we developed a launch file that creates a *node container* with *two composable nodes*:

- `Image_proc::RectifyNode`, which allows image rectification. Specifically, the node subscribes to the `/rgb_camera/image` and `/rgb_camera/camera_info` topics published by `ros_gz_bridge` node. `RectifyNode` then processes the data and publishes the rectified image in the `/rgb_camera/image_rect` topic.
- `Apriltag_ros::AprilTagNode`, which enables AprilTags detection. The node subscribes to the `/rgb_camera/image` and `/rgb_camera/camera_info` topics and publishes a `apriltag_msgs/msg/AprilTagDetection` message in the topic `apriltag/detections`, which makes possible to obtain some information about the detected tags, such as family, ID, corners, etc. Furthermore, it publishes in `/tf` the transformations between the tag frames (`tag36h11:1` and `tag36h11:10`) and `/odom`.
The `AprilTagNode` is configured via the `tags_36h11.yaml` file, which specifies the tag family (36h11), the size (5 cm), the Tag IDs to search for (1 and 10), and enables the publication of TF frames.

2.2 Pose estimation

Once the `RectifyNode` and `AprilTagNode` nodes are started and correctly configured, the `find_goal_position_node.cpp` node calculates and publishes the robot's goal position.

Specifically, the node subscribes to the `apriltag/detections` topic, which allows it to automatically obtain the tag frame IDs and, using the transformations in `/tf`, obtains the pose of Tag 0 (P_0) and Tag 10 (P_{10}) with respect to `/odom`. The mean point between the AprilTags is then calculated and published in the `/goal_pose` topic.

3 Navigation and Control

To navigate the environment efficiently, we implemented a hybrid state machine that switches between a standard planner and a custom reactive controller depending on the environment (e.g., corridor vs. open room).

3.1 Standard Navigation (Nav2)

For open-space navigation, we utilize the ROS2 Navigation2 stack, interfacing with it through a custom `NavGoalSubscriberNode`. This node orchestrates the navigation process by bridging the goal estimation system with the Nav2 action server.

3.1.1 Initialization and Localization

Upon startup, the node waits for the Nav2 stack to initialize. Once the `navigate_to_pose` action server is available, the node publishes an initial pose estimate to the `/initialpose` topic. This step is crucial for initializing the Adaptive Monte Carlo Localization (AMCL) algorithm, setting the robot's starting position at the origin (0,0) with zero orientation.

3.1.2 Goal Handling

The node subscribes to the `/goal_pose` topic to receive target coordinates from the AprilTag detection system. When a new goal is received:

1. **Validation:** The system checks if a navigation task is already in progress to prevent conflicting commands.
2. **Transformation:** The goal is stamped with the `map` frame and the current timestamp to ensure compatibility with the global costmap.
3. **Action Execution:** The goal is sent asynchronously to the Nav2 action server via the `NavigateToPose` interface.

3.2 Custom Controller (Bonus Task)

When the turtlebot enters the corridor, the robot performs a custom navigation. To realize this feature, two packages have been developed: `detection_corridor`, which recognizes when the robot is in the corridor, and `corridor_manual_nav`, which sends velocity commands to the robot.

3.2.1 Corridor Detection

The switching mechanism is governed by a `CorridorDetectorNode` that processes LIDAR data to identify corridor features. The detection pipeline consists of:

1. **Feature Extraction:** The RANSAC algorithm fits linear models to the laser points to identify parallel wall segments.
2. **Validation:** The system verifies geometric constraints, ensuring the walls are parallel and the corridor width is within feasible limits ($0.5m - 1.2m$).
3. **Hysteresis:** To prevent state flickering, the system requires 2 consecutive detections to engage the custom controller and 5 consecutive failures to revert to Nav2.

This binary state determines whether the robot follows the global Nav2 plan or the local reactive controller.

3.2.2 Control Logic

The controller uses the `LaserScan` data to keep the robot centered between the walls and allow it to exit the corridor (it is assumed that the walls are approximately parallel, and that there are no obstacles or impediments in the corridor). Each sensor ray corresponds to an angle according to the formula:

$$\theta_i = \text{angle_min} + i \cdot \text{angle_increment},$$

and together they produce a visual field equal to $[\text{angle_min}, \text{angle_max}] = [0.0, 6.28] \approx [0, 2\pi]$. Specifically, the angles correspond to the following orientations:

$$0 \text{ rad} = 0^\circ \text{ (front)}, \quad \frac{\pi}{2} = 90^\circ \text{ (left)}, \quad \pi = 180^\circ \text{ (back)}, \quad \frac{3\pi}{2} = 270^\circ \text{ (right)}, \quad 2\pi = 360^\circ \text{ (front)}.$$

To calculate the distance from the walls, the following two angle windows are then selected:

$$\text{left side: } \theta \in \left[\frac{\pi}{2} - \Delta\theta, \frac{\pi}{2} + \Delta\theta \right], \quad \text{right side: } \theta \in \left[\frac{3\pi}{2} - \Delta\theta, \frac{3\pi}{2} + \Delta\theta \right], \quad \text{with } \Delta\theta = 10^\circ$$

Then, these angles are converted into indices of the `scan->ranges` array using the inverse formula:

$$i = \frac{\theta - \text{angle_min}}{\text{angle_increment}}$$

rounded to the nearest integer and clamped to the range $[0, \text{ranges.size()} - 1]$.

This way, we obtain 4 indices (`left_start`, `left_end` and `right_start`, `right_end`) with which the average distance of the robot from the wall is calculated ($d_{\text{right}}, d_{\text{left}}$), considering only valid measurements. If there are no valid values, a high default distance (10 m) is considered. Then, to make the robot equidistant from the walls, a controller proportional to the centering error is used:

$$\omega = -K_p \cdot e \quad \text{with} \quad e = d_{\text{right}} - d_{\text{left}},$$

where ω is the angular velocity and $K_p = 0.8$ is the proportional gain (to avoid excessive rotations, a maximum angular velocity has been set). Essentially, if the robot is farther from the right wall, then the error will be positive, resulting in a negative angular velocity ($w < 0$): the robot rotates to the left (and vice versa).

As for the linear speed, it is set to a constant value so that the robot moves forward and exits the corridor.

4 Perception: Table Detection

The table detection module is implemented as a ROS 2 node named `circle_detector`. The algorithm processes raw 2D laser scans to identify, track, and publish cylindrical obstacles (representing tables) in the environment. The pipeline consists of four main stages: point cloud accumulation, clustering, geometric fitting, and temporal tracking.

4.1 Point Cloud Accumulation and Filtering

To mitigate the sparsity of individual LIDAR scans and account for robot motion, raw `/scan` data is accumulated into a global persistent point cloud. Upon receiving a scan, the points are transformed from the sensor frame to the static `map` frame. To manage memory and remove dynamic objects, a temporal "aging" mechanism is implemented using the point cloud intensity field, in which new points are initialized with an intensity of 0, and aged by $\Delta t = (\text{current scan time stamp} - \text{point time stamp})$ at every subsequent scan. Points are removed if their intensities exceed the defined time to live (3.0s) or if they are outside a defined radius around the robot (6m).

4.2 Clustering

The filtered point cloud is segmented into distinct clusters using Euclidean Cluster Extraction. A Kd-Tree structure is utilized for efficient nearest-neighbor search. Points are grouped into the same cluster if the distance between them is less than 0.5m. Clusters containing fewer than 5 points are discarded as noise.

4.3 Geometric Circle Fitting

For each extracted cluster, we apply an Algebraic Least Squares method to fit a circle model. The problem is formulated as solving the linear system $Ax = b$ to minimize the residuals of the equation:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (1)$$

where (x_c, y_c) is the center and r is the radius. The fit quality is evaluated using the Root Mean Square Error (RMSE). A cluster is classified as a valid table if the calculated radius r falls within $(0.02m - 0.25m)$ and the RMSE is below 0.04.

4.4 Temporal Tracking and Association

To ensure stable detection across frames, a tracking layer maintains a list of known obstacles. New detections are associated with existing tracked circles if their center lies within radius 0.2m from any existing detection. The update logic incorporates a hysteresis factor $\epsilon = 0.005$. An existing tracked circle is updated with new parameters if the new detection matches spatially and satisfies:

$$RMSE_{new} - \epsilon < RMSE_{old} \quad (2)$$

This ensures that the system looks for the most accurate geometric fit while slightly preferring newer circles, even if not optimal. Circles that are not re-observed within 1.0s are purged from the system, since the environment could have changed or they could have been misleading detections due to LIDAR errors.

Finally, the tracked circles are transformed from the `map` frame to the `odom` frame and published to the `/circle_obstacles` topic for navigation. This topic contains an array of custom circles that are defined as `geometry_msgs/Point center`, `float64 radius`, `float64 fit_rmse`.

5 Results and Discussion

The goal pose is correctly estimated, and the robot is able to reach it. Furthermore, the robot is able to navigate the corridor using our custom navigation controller. The table detection reacts smoothly to changes in position, but it might be possible that at the start of the simulation, for a brief period of time (1 second), a non-existing table is detected. This is due to the fact that, in order to detect thin tables, we need to lower the minimum points per cluster to 5. If the points on a wall are slightly misaligned, they could fit a large circle. However, after few scans, the point cloud is populated with enough points to correct the false detection.

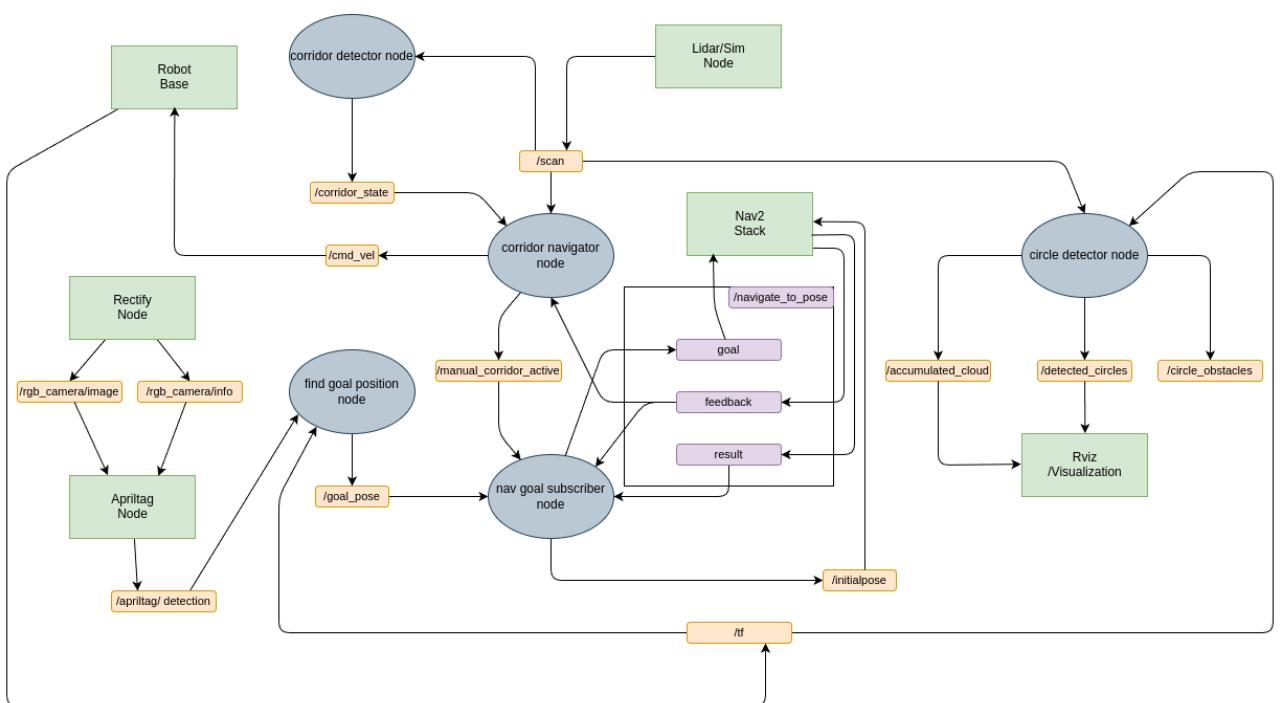


Figure 1: diagram of the most important nodes. Green nodes are provided by ir_launch, blue nodes are developed by us, orange labels are the topics, purple labels are the actions.