Intelligent Robotics Assigment 1

GROUP 05,
Sara Farris sara.farris@studenti.unipd.it,
Alberto Formaggio alberto.formaggio@studenti.unipd.it,
Michele Russo michele.russo.2@studenti.unipd.it,
Bitbucket: https://bitbucket.org/ros-group-5/ir2324_group_05/src/master/,
Additional Material: http://bit.ly/3v5BDt8

February 5, 2024

1 Introduction

This report delves into our innovative solution aimed at navigating narrow passages using laser data, leveraging the */move_base* topic for navigation, and employing *LiDAR-based* obstacle detection.

2 Code Structure

Our solution adopts an action client-server architecture, where the server will be responsible for reaching the position provided by the client and detect the obstacles. The code is well structured for modularity, facilitating testing and scalability. The code has been divided into 3 main components:

- **Robot.Action** This action is used from the action client to send the goal pose (point B) to the action server in terms of coordinates (x, y, ω).
- **Action client** reads from the terminal the desired pose that is sent to the *Action server* through a Robot.action action.
- **Action Server** is the core of all the crucial operations for this assignment, the server is responsible for the movement done by publishing on the topic /move_base if the robot is not in the narrow corridor. Being a server behaving also as a client for other actions, this server behaves as a proxy. Due to its importance, we would like to spend more words about the main tasks accomplished by the server:
  - **Robot's movement**: it moves the robot through narrow passages manually and also in the general environment by relying on the */move_base* topic
  - **Obstacle Detection**: it is done by using clustering functions, implemented in *scan_clusterizer.cpp* and general purpose functions, available in *utilities.cpp*.
- **Law.Action**: The action used by the action server (when behaving as a client) to communicate with the Control law server regarding the navigation in the narrow corridor.
- **Control Law server**: Upon requests of an action client, reads the scanner data and moves the robot inside a narrow corridor, until the distance to the closest wall is higher than the distance provided in the goal.

3 Implementation details

3.1 Navigation in Narrow Passages

We've implemented a *motion control law* for navigating narrow passages using *LiDAR* scans by relying on a reactive architecture. In practical terms, we've developed an action server housed in *ControLaw.cpp*, establishing communication with the main server discussed in section 2 through *law.action*.

This server receives a goal distance from the *Action server*, representing the maximum acceptable proximity of the robot to the closest wall in a narrow corridor. Simultaneously, it provides feedback on the robot's current position and communicates the outcome of its motion. The sent distance functions as a crucial threshold; if the robot's minimum distance to the closest wall surpasses this value, it triggers a halt, indicating that the robot perceives itself to be in a narrow corridor.

As we said the control's implementation relies on *LiDAR*'s data. Initially, we performed pre-processing by excluding the first and last 20 detections (which represent internal noise due to the chassis), by reducing the min and max angle by $20 \times$ angle_increment.

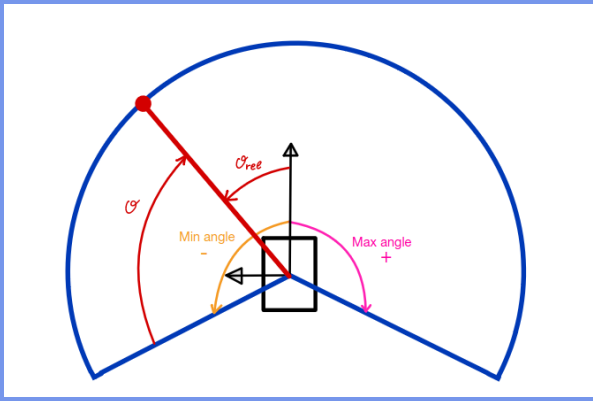

Figure 1: Lidar's view

Subsequently, we focused on the closest point detected, namely the minimum ρ, and its angle θ. We then translated the θ in the corresponding angle with respect to the scan reference frame so to understand its sign, we call this angle θ_{rel}. Specifically we used these two parameters as follows:

- ρ: We use the minimum distance measurement to assess whether an adjustment in orientation is required. If this value drops below a specified threshold, indicating that the robot is in close proximity to the wall, we proceed to adjust the angle accordingly. Conversely, if the value remains above the threshold, signifying that the robot is not in immediate proximity to the wall, it can continue moving in a straight line without requiring adjustments.
- $\boldsymbol{\theta_{\text{rel}}}$: if the distance is below the threshold an adjustment in the orientation is required therefore we analyze the θ_{rel} angle. A positive θ_{rel} indicates the necessity to rotate the robot leftward, while a negative θ_{rel} suggests a need to rotate the robot rightward.

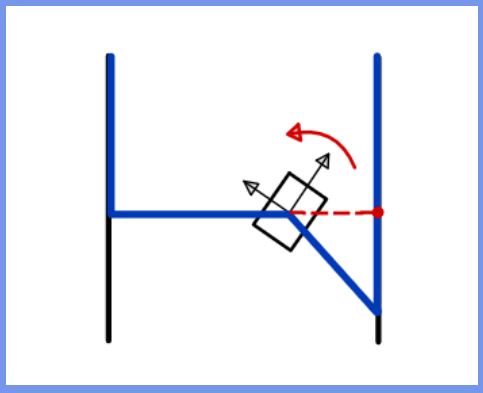

Figure 2: Robot approaching the right wall

Referring to Figure 2, the robot's trajectory indicates a movement toward the right wall. Upon reaching a proximity where the red dotted line falls below the defined threshold, the robot initiates a rotation and shifts its direction towards the left, since in this case the θ_{rel} is positive. The robot continuously assesses its position in relation to the goal to determine task completion using odometry.

This method enables the robot to successfully reach the goal and send a positive result to the main server. Eventually, the main server utilizes /move_base topic to navigate towards the final goal.

3.2 Obstacle Detection

Once the robot reaches its intended position, it starts the task of detecting cylindrical objects in the room. The main aim here is to find where the cylinders are, ignoring where the walls are located. Even though we already know there are exactly four cylinders, we designed our solution to be more flexible, in fact, it's built to detect the number of circles that can be detected by using *LiDAR's* scans, in that specific robot pose.

3.2.1 Object grouping

The readings provided by the *LiDAR* are in polar coordinates: each reading is a pair (ρ_i, θ_i). For each reading, the laser are done in the range of angles $\theta_i \in [\theta_{min}, \theta_{max}]$ and the angle gets increased by θ_{inc} at each step. So, reading i is associated to an angle $\theta = \theta_{min} + i \cdot \theta_{inc}$. These readings are used to generate a point in the Cartesian space $\mathbf{p_i} = (x_i, y_i)$ by using the formulas $x_i = \rho_i cos(\theta_i)$ and $y = \rho_i sin(\theta_i)$ 3

There is the need now to group together the laser readings that belong to each object. This is done by relying on the concept of unsupervised clustering (we used the *cv::partition()* [1] function). At the beginning, each point p_i is assigned to a cluster S_i. Then, iteratively, $\mathbf{p_i} \in S_j$ if $\exists \mathbf{p_k} \in S_j$: $d_{L2}(\mathbf{p_i}, \mathbf{p_k}) < d_{th}$ where:

- $d_{L2}(x, y)$ is the euclidean (L2) distance between the points x and y.
- d_{th} is a threshold distance defined by the user. We found that 0.2 was providing good results in our example.

The result is reported in Figure 4

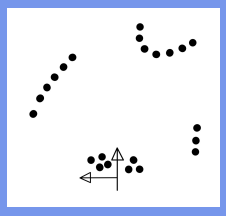

Figure 3: The raw scanner readings

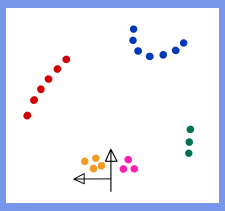

Figure 4: Scanner readings assigned to the corresponding cluster

3.2.2 Cluster Refinement

During our experiments, we realized that some *LiDAR* readings are inside the robot's chassis. Furthermore, there were some clusters containing only few points and thus it would be impossible to extract their geometrical features. In this step therefore we:

1. Remove all clusters such that $d_{L2}(\mathbf{0}, \bar{\mathbf{c}}_\mathbf{j}) < d_{max}$: where $\bar{\mathbf{c}}_\mathbf{j}$ is the centroid of cluster S_j and d_{max} is a user-defined threshold. We obtained good results with $d_{max} = 0.04$.
2. Remove all clusters such that $|S_j| \leq N_{min}$. Here, N_{min} is the minimum number of points a cluster should have to be considered valid. In the experiment, $N_{min} = 5$.

3.2.3 Lines Removal

Once only meaningful clusters remain, they categorically fall into two types: lines or cylinders.

Initially, we remove all the clusters that are associated with lines. To do so, we convert each cluster in binary image with size $(\frac{\text{cluster width}}{\text{resolution}}, \frac{\text{cluster height}}{\text{resolution}})$, where the cluster width and height can be found

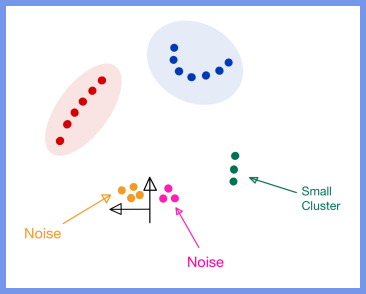

Figure 5: Removal of the noisy clusters

by looking at the x coordinate of the left-most, right-most pixels and at the y coordinate of the top-most and bottom-most pixels. The resolution defines how many pixels to use for each variation of x and y equal to 1.

After clustering, we apply the *Hough transform* to detect lines in each image. We focus on determining if at least one line is present; if so, we confidently discard the cluster. We configure the *cv::HoughLines [2]* function to consider a line valid only if at least 20 points align along it.

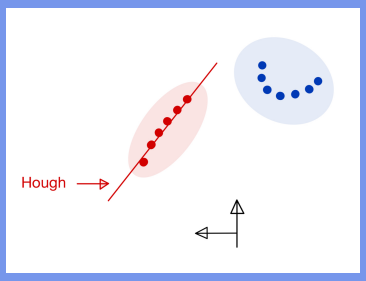

Figure 6: Detection of lines by using Hough transform.

3.2.4 Circle Detection

Finally, we are left only with few clusters. It is very likely that they are mostly circles, but there may still be some false positive. The following approach was adopted at this stage:

1. For each cluster, we try to interpolate a circle passing through three points. We picked them distributed equally to divide the points into 4 parts. Based on those 3 points it is possible to compute the coordinates of the center and the corresponding radius.

2. We discard all the circles that have a radius $r > R_{max}$. The idea behind this is that it is possible to interpolate a circle even over points that belong to a line, however the resulting circle will have a very large radius.

3. Finally, you look at the *MAE* between the points of the cluster and the approximated circle. If $MAE = \frac{1}{n}\sum_{i=0}^{n} |r_j - d_{L2}(\mathbf{c_j}, \mathbf{p_{ij}})| < MAE_{max}$ then the circle is classified as a positive example.

In the end, the center of the circles in the robot's reference frame (base_link) is returned and is converted by using a transform to the odom reference frame and then a transformation to the Gazebo reference frame.

$A = (x_A, y_A), B = (x_B, y_B), C = (x_C, y_C)$

$$(x1 - xc)^2 + (y1 - yc)^2 = r^2$$
$$(x2 - xc)^2 + (y2 - yc)^2 = r^2$$
$$(x3 - xc)^2 + (y3 - yc)^2 = r^2$$

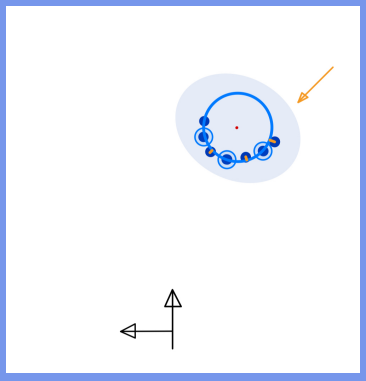

Figure 7: Keeping only the clusters that correspond to circles

References

[1] Partition. [Online]. Available: https://docs.opencv.org/4.x/d6/d00/operations$_$8hpp.html

[2] Hough transform. [Online]. Available: https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html

4 Work Distribution

The order of the names corresponds to the amount of contribution.

- **Object Grouping**: Michele Russo, Sara Farris
- **Cluster Refinement**: Alberto Formaggio
- **Lines Removal**: Michele Russo, Alberto Formaggio
- **Circle Detection**: Alberto Formaggio
- **Navigation in the narrow corridor**: Sara Farris, Alberto Formaggio
- **Navigation with move base***: Sara Farris, Alberto Formaggio, Michele Russo
- **Transformation of the coordinates between reference frames**: Sara Farris, Alberto Formaggio, Michele Russo

* all the people nominated contributed equally