Report Assignment 1 - Group 27

Bejaj Xhaku
Fonsato Andrea
Sambin Luca

December 23, 2022

Introduction

To realize this project, it is implemented the following procedure:

1. The action client sends poseB to the action server
2. To realize the bonus part, the action server verifies if Tiago is in the Starting Pose and it calls a function, using the topic /cmd_vel, to navigate through the narrow corridor. If Tiago is in a different position, the action server skips this part and sends directly the Pose_B to the move_base stack.
3. At the end of the corridor, the action server sends Pose_B to move_base stack
4. When the robot arrives to Pose_B, the action server waits 4 seconds, so that the robot is really stopped, it calls topic /scan and receives the laser message and it starts to detect the cylindrical obstacles
5. At the end of the detection, the action server publishes the results that are also printed on the client. The obstacles coordinates are expressed with respect to the robot's frame.
6. Finally, action client stops its execution, but the action server remains active for further requests

In the next sections, it is described the realization of the action client/server.

Action Server

Let explain the main variables and functions to implement the server. The class stores:

- a vector of PointSet representing the first, the middle and the last points of a set used to detect and analyze the obstacles;
- a vector of Circle used to store center and radius of the circles computed during the detection.

The *obstacle_detection* function processes a laser scan message to extract all valid points from the scan and store them in a vector of points. In particular the first and the last 19 measures are discarded since they represent the robot itself and therefore they are useless data. It then calls the *groupPoints* and *detectCircles* functions to group the points into sets and detect circles, respectively. The *groupPoints* function iterates over the points in the point vector and groups them into sets, of at least 5 points, based on their distance from each other. If the distance between two points is less than a maximum value equal to 0.15m, the function updates the end point and the number of points inside the set; otherwise a new group is started. The *detectCircles* function stores the cylindrical obstacles in the vector of circles and calls the functions *findCircle* and *is_a_Circle* to compute the circle and detect if a point is on the circle, respectively.

The *findCircle* function calculates the center and the radius of a circle, representing the cylindrical obstacle, given 3 points around the circumference. To find the center, first the function computes the perpendicular bisector between first point and second point but also between second and third point. Then, the intersection point between those two lines represents the center of the circle. This reasoning is based on the definition of perpendicular bisector, stating that a perpendicular bisector between two points is the set of all points equidistant from both. Therefore the computed intersection between these two bisectors must be the center of a circle. In the next step, the function computes the distance between the intersection point and any of the five point. This distance represents the radius, since in geometry the radius of a circle is defined as the distance between the center and any point of the circumference.

The equation of the two bisectors are $y_a = m_a(x - x_1) + y_1$ and $y_b = m_b(x - x_2) + y_2$, where the

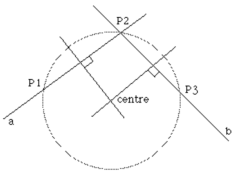

Figure 1: Perpendicular bisectors

slope of the line is given by $m_a = \frac{y_2-y_1}{x_2-x_1}$ and $m_a = \frac{y_3-y_2}{x_3-x_2}$. Then it is checked if the points are colinear proceding to discard them, otherwise it is computed the center which is the intersection of the two perpendicular lines passing through the midpoints of the segments P_1P_2 and P_2P_3. Solving for x, it gives $x = \frac{m_a m_b(y_1-y_3+m_b(x_1+x_2)-m_a(x_2+x_3)}{2(m_b-m_a}$ while the y value of the centre is obtained by substituting the x value into the equations of the perpendiculars like $y = -\frac{1}{m_a}(x - \frac{x_1+x_2}{2}) + \frac{y_1+y_2}{2}$.
The *is_a_Circle* function is used to detect if a certain point lies on the circumference or if it is outside the circle. This control is done checking if the distance between a point and the circle's center is greater than the radius' value, fixed at 0.25m where this value is justified by the fact that the diameter of the objects to be detected is about 0.4m. Then the *publishObstacles* function publishes the detected cylindrical obstacles in the result vector that is shown during the execution, converting from base_laser_link frame to base_link frame.
The main function of the class is the *executeCB* function, called when the action server receives a goal from the client. This function publishes feedback to the client, indicating that the goal has been received. It then waits for the move_base action server to come online and sets up a goal pose for the robot to move to. Next, the function enters a loop in which it checks for obstacles in the robot's environment using the *laser_msg* variable. If an obstacle is detected, the robot is commanded to move around it and the loop continues. If no obstacles are detected, the robot is allowed to move to the target pose and the loop stops. Finally, the function returns a result to the client, indicating the result of the detection of the cylindrical obstacles.

Action Client

The action server and client communicate using a specific action message that includes the goal position, all the detected obstacles and the feedback about the robot's status. The *doneCb* function, called when the goal is completed, prints the number of detected obstacles with their relative positions; the *activeCb* function only tells when the goal becomes active, while the *feedbackCb* tells that the goal is received, explaining the current Tiago status. In the main function the action client first initializes a ROS node, then it checks that the number of arguments provided by the user, representing the position (pose_B) to reach, is correct and finally it creates an instance of the client.
The action client waits for the action server to start before sending the goal message to it. Then it waits for a result from the server with a specified timeout of 300 seconds. If the goal finishes before the timeout, the state of the goal is reported. If the timeout is reached before the goal is completed, the user is notified that the goal did not finish in the allotted time.

Bonus part

To implement the navigation through the narrow corridor without the move_base stack, first the action server checks if Tiago is in the Starting Pose at the beginning of the corridor. Then, if this is true, the *manual_moving_routine* is called otherwise it navigates using the normal move_base stack, checking if the robot reaches Pose_B thanks to the *auto_moving_routine*.
While the robot is inside the corridor, the *manual_moving_routine* checks if it stays on the center of the two laser's readings (respectively 90° and -90°) with a threshold of about 10cm. If it's in center then it keeps going straight otherwise if it is too much on the right or on the left, first it changes the angular velocity and then, calling the *updateLaser* function, it fixes the position based on the laser's readings.
To stop the *manual_moving_routine* execution, two exiting conditions are set:

- the robot detects an obstacle at less than 2m in front of itself (safe condition);
- the robot reaches the end of the corridor.

Then it passes from the *manual_moving_routine* to the *auto_moving_routine* thanks to the second condition if everything went well.