Edoardo Bastianello: 2053077
Stefano Binotto: 2052421
Gionata Grotto: 2052418

ASSIGNMENT 1 REPORT

CLIENT - “controller_client.cpp”:

Our Action Client requests, as input in the command line, 7 parameters that define the goal pose: 3 parameters for the position (x, y, z) and 4 parameters for the orientation (x, y, z, w). Next, it sends the goal to our user-defined Action Server and it subscribes to its feedback topic. Then, once the robot reaches the goal pose, it receives the result from the server (by subscribing to the relative result topic) and prints it (Note: in order to subscribe to both the feedback and result topic we used the CallbackQueue class functions). If the robot doesn’t reach the goal pose in a certain amount of time, it prints an error message.

SERVER - “robot_srv.cpp”:

Our class “PlannerRobot” implements the Action Server and is composed by the following functions:

1. ***move_towards_goal***: this function receives the goal pose from the Action Client and sends it to the “move_base” Action Server in order to make the robot navigate and reach that pose. Next, it publishes a feedback that specifies that the robot is moving. Then, until the robot reaches the goal, this function subscribes to the “nav_vel” topic and calls the *velocityCallback* function that publishes the feedback of the robot (see below). When the goal pose is reached, it subscribes to the “scan” topic and the *scanCallback* function gets called to compute the centers of the movable objects. Finally, the function sends the results to the Action Client. A time counter checks that the robot reaches the goal before a certain time limit (named “MAX_TIME_TO_REACH_GOAL*”* and equal to 200 seconds), otherwise it returns an error.
2. ***velocityCallback***: this function checks whether the robot is moving, is turning or is stopped. It reads the values from the “nav_vel” topic in order to analyze the linear and angular velocities. When all the velocities are zero, the server publishes the feedback status “The robot is stopped”. When the linear velocities are not zero, the feedback status published is “The robot is moving”. When the angular velocity with respect to the z axis is bigger than a threshold value we empirically chose, the server publishes the feedback status “The robot is turning”. We decided to use a threshold (“RADIUS_THRESHOLD_WHEN_TURNING” = 0.15 rad) to distinguish the “moving” and “turning” states because we found out the robot is always applying an angular velocity to stabilize, even though it seems to be going straight.
3. ***scanCallback***: this function publishes the feedback that specifies that the detection of the obstacles has started. Next, it divides the points scanned by the robot in sections by calling the *subdivide_in_sections* function. Then, it discards the sections that aren’t movable objects by calling the function *obstacles_finder*. Finally, it computes the centers of the detected objects by calling the *center_circle* function and publishes the feedback that specifies the detection of the obstacles has finished.
4. ***subdivide_in_sections***: this function subdivides the detections of the “ranges” array in different sections, depending on the distance between them. In particular, each

section is defined as the set of consequent detections of the “ranges” array having distance between them lower than a user-defined threshold named “THRESHOLD_DISTANCE” that is equal to 0.25. (Note: the first 19 points and the last 18 points of the “ranges” array have been discarded since they were wrong detections that hit the robot model).

5. ***obstacles_finder***: this function aims to find the movable obstacles given all the different detected sections that we found. For each section that we have, we firstly check that the section has more than 4 points and that its width is not larger than the diameter of the movable obstacles. If so, we need to check if the section we are analyzing has a circled-shape or not. To do so, we use the *center_circle* function to compute the center of its hypothetical circle. If the distance between this center and each point of its section is almost equal to the radius of the movable obstacles (we set an error of 0.02), then the section is confirmed to be a circle and consequently it’s a movable obstacle.
6. ***center_circle***: this function computes the center of a section considering 3 of its points (first one, middle one and last one) by solving a system of 3 equations.

ACTION FILE - “Move.action”:

This file contains:

- **goal:** *Pose_B*, defined as a *geometry_msgs/Pose;*
- **result:** obstacles, defined as an array of geometry_msgs/Point. This array will contain the coordinates of the centers of the detected movable obstacles.
- **feedback:** the feedback includes a header and a string named *status*, which will contain the current status of the robot.

ASSUMPTION:

We assumed that the movable objects aren’t attached to each other because if the robot detects the point of contact between each object it will consider it as one unique section (and consequently the two attached obstacles will be discarded). However, if the robot doesn’t see the point of contact it will rightly detect two sections.
Note: the coordinates of the centers are returned with respect to the robot pose.

COMMANDS TO RUN THE PROGRAM:

1)cmd console 1:
roslaunch tiago_iaslab_simulation start_simulation.launch world_name:=ias_lab_room_full
2)cmd console 2:
roslaunch tiago_iaslab_simulation navigation.launch
3)For the client, cmd console 3:
rosrun tiago_iaslab_simulation controller_client x y z x y z w
(where x y z x y z are the pose), for example:
rosrun tiago_iaslab_simulation controller_client 11.8 0 0 0 0 0.49 -1
4)For the server, cmd console 4:
rosrun tiago_iaslab_simulation robot_srv

SIDE NOTE:

The number of commits isn't indicative of the individual contribution since we mostly coded together.