Intelligent Robotics Project

Assignment 1 - Group 20

Marco Mustacchi, Valentina Zaccaria,
Damiano Zanardo

1 Structure and Usage

Commands to execute the code with optional point:
cmd console 1

```
>>> roslaunch tiago_iaslab_simulation start_simulation.launch world_name:=ias_lab_room_full
```

cmd console 2

```
>>> roslaunch tiago_iaslab_simulation navigation.launch
```

cmd console 3 (first wait until Tiago has tucked its arm)

```
>>> roslaunch tiago_iaslab_simulation move_scan.launch target:="float_x float_y float_yaw"
↪   [corridor:=true]
```

Example:

```
roslaunch tiago_iaslab_simulation move_scan.launch target:="11 -0.5 -1.52" corridor:=true
```

The values must be given in the `map` frame and the positions of the obstacles are returned with respect to the `base_laser_link` frame. This choice is not restrictive, because the positions are returned in a `std::vector` of `geometry_msgs::PointStamped`, therefore each point contains the information of the frame with respect to which it is expressed and one can change frame simply by using the `tf2` and `tf2_ros` packages. The program also displays the markers of positions in RViz.

1.1 Structure

There are 4 nodes in the package folder (their implementations are inside the src folder):

- `client_node`: action client that creates the action client goal with the target pose and send it to `move_server_node`. The action client implements also a callback to the feedback of actions server and prints the task's current status of the robot each time that the status changes.
- `move_server_node`: sends the goal "target pose" to `move_base`, when the goal is reached asks to `scanner_node` the service of obstacles position detection. When everything is done, it sends to the client all the positions as a result.
- `scanner_node` that provides the service of cylindrical obstacles detection.
- `corridor_node`: it detects when the robot is in a corridor, if so pauses the `move_base` navigation and starts the motion control law.

2 Navigation with Navigation Stack

The desired pose in the `map` frame is given through the command line as requested. The user has to provide the parameters x and y coordinates and the orientation expressed with the yaw angle, since it is very intuitive. The action server `move_base` goal requires a target orientation expressed with respect to unit quaternions, therefore the conversion is made with the package `tf2` and normalization. The goal is first sent to `move_server` which acts as action client and it sends the goal to `move_base` action server.

2.1 Feasibility of the target pose

Once the user has inserted the desired pose, the program checks if it is feasible. To do that, it checks the occupancy grid of the static map, subscribing to the topic `"\map"` and exploiting the package `costmap_2d` to find the cell index starting from the coordinates and the cell value (in particular if it was `costmap_2d::FREE_SPACE`).

3 Cylindrical Obstacle Detection

Once the goal position is reached, the main action server `move_server` calls the service node `scanner_server` to detect the cylindrical tables.

1. The `scanner_server` node waits for a single message to arrive on the `"\scan"` topic, where laser sensor publishes.
2. First, it clusters the "valid" 2D laser scans by dividing them basing on big jumps in radial distance difference between subsequent points. At the same time, the node transforms each point from polar to Cartesian coordinates in the frame `base_laser_link`, which is the frame of the laser.
3. Then, clusters with too few number of points are discarded.
4. Exploiting the equation of a circle through 3 points, for each cluster the node take the first, the middle and the last point and compute the radius and the center of the circle passing through them. One can be more accurate by taking several tuples of three points. However, taking these three points seems to be sufficient for this application purposes.
5. If the radius is too large, it is unlikely that the cluster points come from cylindrical obstacles. More likely, they represent a line, which in principle should have infinite radius but in practice it is very big. Therefore, a threshold or radius value decides if that cluster represents a cylindrical object or not.
6. If the cluster is marked as cylindrical table, then the `scanner_server` sends the coordinates of the computed center (final position of the obstacles in `base_laser_link` frame) to the main action server `move_server` which sends them as a result to the action client.

Other methods have been tried. The first exploited the fact that cylindrical tables are far away from walls. It considered one cluster at time, it transformed the Cartesian coordinates of all points in the map frame and then it mapped them in the cells of the static map of the environment. It checked if there were some points that belong to map's occupied cells. If so the cluster had points that belong to a wall, therefore it was neglected as obstacle. This solution failed when the scanned points come only from the big desk with books. Another tried solution was to use Hough Transform for each cluster to check if there were lines inside it. The main drawback was that the accumulator angle and distance resolutions and threshold depend too much on where the scan was made.

4 Narrow passages: motion control law

4.1 Narrow Passage Detection

To detect narrow passages, the ranges values at $\pm 90\,\deg$ of the laser scan (with respect to the x axis of the robot frame) are exploited. If the sum of these two lateral distances is below a certain threshold, the narrow passage is detected.

4.2 Motion Control law implementation

1. When an action goal is sent to `move_base` action server, the node `corridor_node` starts to check if the agent is in a narrow passage (detection described above).
2. If it is the case, the node pauses the navigation of `move_base` by publishing "true" instead of "false" in the `"\pause_navigation"` topic.
3. The robot starts moving with the new motion control law. The node publish in the topic `"\cmd_vel"` to control the velocity. In particular, it imposes a linear velocity of $0.5m/s$ (in the robot frame, so it moves forward) and an angular velocity along z axis (to allow a rotation in the x-y plane) which is proportional to the $-$ difference between the right and the left walls.
$$\text{velocity.angular.z} = 0.5 \times -(\text{distanceRight} - \text{distanceLeft}) \tag{1}$$
4. When the robot is near to the target or when it exits from the narrow corridor, `corridor_node` publishes to the `"\pause_navigation"` topic false and the navigation with `"move_base"` starts again.