Assignment 1

Group 6

Alberto Dorizza, alberto.dorizza@studenti.unipd.it
Federico Gelain, federico.gelain@studenti.unipd.it
Dario Mameli, dario.mameli@studenti.unipd.it

Repository at: https://bitbucket.org/ir2324-group-06/ir2324_group_06/src/main/
Additional material at: https://drive.google.com/drive/folders/1wlIDHWMwaEFmq8XOcLkxGTH3ORP3nTDt?usp=sharing

1 Our approach

Our group's solution to the problem can be summarized as follows:

1. the action client (**assignment1_client**) receives by command line position and orientation of *Pose_B*, and communicates it as an action goal to the action server (**assignment1_server**);
2. the action server guides the robot Tiago to the end of the corridor following a control law driven by the laser data. Afterwards the server sends the *Pose_B* to the **/move_base** topic as a goal;
3. while **move_base** handles the movements that the robot has to make in order to reach the desired goal, the action server monitors the feedback it receives from the node and sends it to the action client;
4. if the robot successfully reaches Pose_B, then the action server subscribes to the **/scan** topic collects the scanner data, and determines the position of the cylindrical tables, which will then be sent to the client using the action result.

1.1 Action client and server communication

The action file is defined in the following way:

```
#goal
#pose of Pose_B
float32 pose_B_x
float32 pose_B_y
float32 pose_B_yaw

#flag to activate the motion control law
int8 motion_control_law
---

#result
#positions of the obstacles
float32[] obstacles_positions_x
float32[] obstacles_positions_y

---

#feedback status
string status
```

Figure 1: Structure of the *Info.action* file

The client will send to the action server the pose the robot has to reach and if the motion control law has to be used or not. In former the action server use the motion control law to reach the end of the corridor and after that communicate the goal pose to the *move_base* node, instead in the latter case it communicate *Pose_B* immediately.
The position is provided as a couple of float values, corresponding to the x and y coordinates of

the origin of Pose_B (with respect to the absolute reference frame). In contrast, the orientation is provided as a triplet of angles Roll-Pitch-Yaw (in degrees) that specifies the set of rotations needed.
As for the feedback on the current status of the robot, we return the following messages:

- "The robot started moving using the motion control law"
- "The robot has reached the end of the corridor"
- "The robot started moving using move_base"
- "The robot has arrived at the goal location and it stopped"
- "The robot is starting the detection of the obstacles"
- "The robot has finished the detection of the obstacles"

1.2 Action server and *move_base* communication

The action server will act as the client in the communication with the move_base node. This means that it communicates the desired pose (acquired by the action client) as the action goal and monitors the state of the robot while it is moving.
The only feedback that we use from move_base is the state of the goal (either SUCCEEDED when the robot reaches pose_B or not, meaning that it failed to do so). We assume that the robot won't stop while trying to reach its final pose.

1.3 Action server and */scan* communication

After the robot has reached the desired pose, the server will subscribe to the /scan topic to analyze the data observed by the laser and determine the Cartesian coordinates of all the cylindrical tables (obstacles), which will be stored in the action result and sent to the action client. However, detecting these obstacles is a non-trivial task that is worth exploring. First of all, for each received scan, we transform the data into points in the Cartesian space, and then we visit them. Notice that all infinite ranges are discarded together with the first 20 and last 20 values since in this latter case we have partial occlusions with the robot's chassis. In the visiting process, the points are assigned to a **cluster** based on their Euclidean distances, following a DBSCAN methodology. If two points are closer than epsilon, then they are to be put in the same cluster. Epsilon should be tuned according to how close the cylinders are to the walls: in our simulation, 0.5 is a good choice, but if obstacles were to be placed closer to the walls, then a lower value would be needed. If they are farther, then we can use a higher value of epsilon.
Consequently, we calculate the **variance** for each cluster and then we apply a global threshold to check which clusters have a variance that falls within the threshold. These clusters are defined as obstacles. The threshold we have empirically found to best discriminate the cylinders from the walls is 0.012. This threshold is solely dependent on the shapes of the obstacles, and since they are all equal in our simulation, the threshold is very discriminating yet effective. If obstacles should change shape, then another threshold may be needed. Some false positives may be found if there are small and disconnected segments of walls.
Finally, the positions of the obstacles are determined by their **centroids**, which are the points whose coordinates are the averages over x and y of all the points for each cluster. It should be noted that these coordinates are expressed with respect to the robot's reference frame.
Moreover, all the clusters are plotted in a single 500x500 image, as shown in Figure 2.

1.4 Motion control

In this last section, we describe how we implemented the motion control law to drive the robot from any point close to the initial position to the end of the corridor.
For this task, given the robot's holonomic constraints, only two variables can be manipulated: the **linear velocity** along x and the **angular velocity** around z. In our implementation, the linear velocity is assumed constant and equal to 0.1, since trying to set it proportional to the distance of the walls wasn't working well (the robot didn't have enough time to turn and would get stuck against the wall).
The real challenge is determining the correct angular velocity given the scan information on the environment, and for that, we need to compute the proper angles. Since we assume the goal

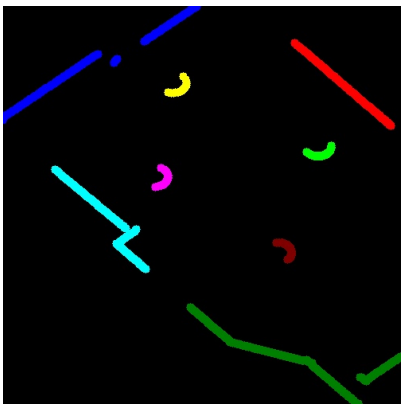

Figure 2: Plot of the clusters detected in position (11, 0), with -45° orientation, epsilon = 0.5, threshold = 0.012 using OpenCV library. Yellow, magenta, dark red, and green are the colors of the obstacles. They are specified also as ROS_INFO in the server terminal

position to be the farthest scan detection, we want to find its corresponding angle in order to adjust the orientation toward the goal. This angle is thus called *goal angle*.
After this, we determine the range at which the closest point is found in the first half of the scan data (meaning between the minimum angle of the scan +20 and the median) together with its exact position in the robot's reference frame. Then we determine the range of the closest point in the second half (between the median and the maximum angle -20), together with its exact position. The reason for +20 and -20 is explained in Section 1.3. These two points are therefore considered the closest possible collision points to the robot from the right (meaning in the half-plane $y < 0$) and the left (half-plane $y > 0$).
Now we need to determine whether these points are actually in the collision path. To understand this, we check whether the absolute value of the components along x and y are within 0.7 and 0.2 respectively. If the point is on the right and is on the collision path, we calculate a counterclockwise rotation around z with a magnitude inversely proportional to the distance between the robot's reference frame and the point. If it's on the left and is on the collision path, we calculate a clockwise rotation around z with a magnitude inversely proportional to the distance between the robot's reference frame and the point.
In summary, we are left with three components to reach the desired rotation: the goal angle, a clockwise rotation angle, and a counterclockwise rotation angle. These values will form the error in the motion control law, which seeks to regulate the angular velocity of the robot.
In particular, the error is defined as:

$$error = (goal\_angle - current\_angle) + z\_rot\_clockwise + z\_rot\_counterclockwise \tag{1}$$

In this equation, we can notice that *current_angle* is always equal to 0 due to the laser scan rotating together with the chassis of the robot (and thus with the robot's reference frame as well), and so the error equation can be reduced to:

$$error = goal\_angle + z\_rot\_clockwise + z\_rot\_counterclockwise \tag{2}$$

Finally, the control law is given by the following formula:

$$angular\_vel\_z = kp \cdot error - kd \cdot (error - prev\_error) \tag{3}$$

Notice that the imposed angles are in reality angular velocities by implicitly assuming the angles are spanned in 1 second. The parameters kp and kd are the PD controller parameters to be tuned, which regulate the angular velocity. The idea of the PD controller as implemented is that kp is responsible for driving the robot to the goal angle, while kd is tasked with lowering sudden bursts in the change of the angular speed. In our simulations, we have used $kp = 1$, $kd = 0.66$, but they are not universally suited for any initial condition in the pose of the robot at the beginning of the corridor and should be tuned accordingly.
The commands on the linear and angular velocities are published in the *cmd_vel* topic.