GROUP 24
Member1: Osti Simone, simone.osti@studenti.unipd.it
Member2: Russo Christian Francesco, christianfrancesco.russo@studenti.unipd.it
Member3: Spinato Matteo, matteo.spinato@studenti.unipd.it
Repository link: https://bitbucket.org/ir2324-group-24/assignment1/src/master/
Google Drive link to video (running with rosrun): https://drive.google.com/file/d/1Z-mMuHPBGhvbLuxtiSOAq_1E3DQC7M9l/view?usp=sharing
Google Drive link to video (running with roslaunch): https://drive.google.com/file/d/1ZxTsEt6o3sQUtkMql7f3S6hb0vMGOkpT/view?usp=sharing
Note: we have implemented the control law (extra points part of the assignment)!

Report Assignment 1 of Intelligent Robotics

How to run the program

First way (suggested) = in different terminals:

- **Start the simulation:** `roslaunch tiago_iaslab_simulation start_simulation.launch world_name:=robotics_library`
- **Navigation stack:** `roslaunch tiago_iaslab_simulation navigation.launch`
- **Run action server:** (example) `rosrun assignment1 main_action_server_node _CL_flag:=true`
  Note: the `CL_flag` allows to enable (`true`) or disable (`false`) the narrow corridor control law.
- **Run action client:** (example) `rosrun assignment1 main_action_client_node _x:=11.0 _y:=0.0 _theta:=90.0`
  Note: the input coordinates `x`, `y`, `theta` represent the input target pose the Tiago robot has to reach, where `theta` is expressed in degrees.

Second way (alternative) = in different terminals:

- **Start the simulation:** `roslaunch tiago_iaslab_simulation start_simulation.launch world_name:=robotics_library`
- **Navigation stack:** `roslaunch tiago_iaslab_simulation navigation.launch`
- **Run launch:** (example) `roslaunch assignment1 tiago_navigation.launch`

Structure of the project

Program structure and files organization

Program structure:

- **`main_action_client_node.cpp`:** contains the main of the action client. This file receives the `target_pose` (x, y, θ) in input from the user and instantiate a `NavigationClient` action client.
- **`main_action_server_node.cpp`:** contains the main of the action server. This file receives the `CL_flag` in input from the user and instantiate a `NavigationServer` action server.
- **`NavigationClient.h/.cpp`:** is the class of the action client. This class calls the action server (topic `/navigation`) that executes all the tasks. Moreover, it implements callbacks to

the feedback of the action server (reflecting the status of the robot) and prints the tasks current *status* in the terminal.

- **`NavigationServer.h/.cpp`:** is the class of the action server. This class executes the following tasks:
    - Narrow corridor control law (if enabled) to cross the narrow corridor;
      Note: the narrow corridor control law is executed (if enabled) only for the first action client connecting to the server, then (once the robot has crossed the corridor) it is disabled by the server, in order to avoid its execution (potentially dangerous for the robot) for the next action client connecting to the same server.
    - `move_base` stack navigation to reach the `target_pose` starting from the current position of the robot;
    - Movable obstacles detection, where the final list of `obstacle_positions` (x, y) of the obstacles is sent as `result` to the action client.
- **`ObstaclesDetector.h/.cpp`:** this class allows to detect the obstacles. For all the details about how obstacles detection works, see the section **Obstacles detection**.
- **`NarrowCorridorControlLaw.h/.cpp`:** this class allows to perform the control law for crossing the narrow corridor (if enabled). For all the details about how narrow corridor control law works, see the section **Narrow corridor control law**.
- **`util/Point.h/.cpp`:** contains some util structs which represent a polar point and a cartesian point, plus some helper functions.
- **`util/Circle.h/.cpp`:** contains a util struct which represents a circle, plus some helper functions.

Note: sources and header (C++) files are respectively organized in the folders `src` and `include`.

Action file: the action file `Navigation.action` has the following structure:

- **Goal:**
    - `std_msgs/Header header`
    - `geometry_msgs/Pose target_pose`
- **Result:**
    - `std_msgs/Header header`
    - `geometry_msgs/Pose[] obstacle_positions`
- **Feedback:**
    - `std_msgs/Header header`
    - `string status`

Launch file: `tiago_navigation.launch` runs the action client and the action server (with input parameters).

Old folder: contains some files related to previously tested approaches and some files containing step by step prints for performing the debug of the currently implemented approaches.

Approaches adopted

Target pose construction

The `target_pose` provided in input by the user is composed by the two spatial coordinates x_{in} and y_{in} for the position and by the (yaw) angle θ_{in} (in degrees) for the orientation.
In ROS the `geometry_msgs/Pose` defines a data structure made up by two main parts:

- **`position`**: represents the three-dimensional position (x, y, z) in the world reference frame.
- **`orientation`**: represents three-dimensional orientation in the world reference frame by means of a quaternion, composed of the (x, y, z, w) components.

The `target_pose` $(x_{in}, y_{in}, \theta_{in})$ provided in input by the user must then be converted into the quaternion notation by the client as follows:

- **`position`**: $x = x_{in}$, $y = y_{in}$, $z = 0$.
- **`orientation`**: $x = 0$, $y = 0$, $z = \sin(\theta_{rad}/2)$, $w = \cos(\theta_{rad}/2)$, with $\theta_{rad} = \frac{\theta_{in} \cdot \pi}{180°}$.

the `target_pose` obstained is then send to the server.

Obstacles detection

The obstacles detection task is implemented in the function `ObstaclesDetector::detect_obstacles_positions` which reads the data coming from the laser sensor (topic `/scan`) and executes the following algorithm to perform the detection of the movable cylindrical obstacles, which is subdivided into the following steps:

1) **Get laser scan parameters and laser scan data (`ranges`).**
2) **Detect obstacles points:**
   - Remove the range values corresponding to the blind angles of the robot ($[0,19]$ and $[646,665]$), so the valid range indices are $[20,645]$.
   - Initialize the vector `obstacles_points` (vector of cartesian coordinates).
   - For each (polar) point $\boldsymbol{r}_i = (\rho_i, \theta_i)$ in the laser scan data (in the valid range of indices):
     - If $\rho_i < \infty$, then it is an obstacle point, so convert it from polar coordinates to cartesian coordinates and store it in `obstacles`;
     - Else discard it.
3) **Group obstacles points in obstacles:**
   - Initialize the vector `obstacles` (vector of vector of cartesian coordinate points).
   - Initialize the vector `curr_obstacle` (vector of cartesian coordinate points).
   - For each (cartesian) point $\boldsymbol{x}_i$ in the `obstacles_points` vector:
     - If $\boldsymbol{x}_i$ is the first point of the `obstacles_points` vector (i.e., $\boldsymbol{x}_0$), then store it in `curr_obstacle` (start forming the set of points of the first detected obstacle);
     - If $\boldsymbol{x}_i$ is the last point of the `obstacles_points` vector (i.e., $\boldsymbol{x}_{n-1}$), then:
       - Store it in `curr_obstacle`;
       - Store `curr_obstacle` (the set of points of the last detected obstacle) in `obstacles`;
       - Clear the `curr_obstacle` vector (optional).
     - Else:

- If $d(\boldsymbol{x}_i, \boldsymbol{x}_{i-1}) \leq K \cdot d(\boldsymbol{x}_i, \boldsymbol{x}_{i+1})$, then store $\boldsymbol{x}_i$ in `curr_obstacle` (this point belongs to the `curr_obstacle`)
- Else (this point does not belong to the `curr_obstacle`):
    - If `curr_obstacle` is not empty, then:
        - Store `curr_obstacle` in `obstacles` (store the last detected obstacle);
        - Clear the `curr_obstacle` vector (initialize a new set of points for the next obstacle);
    - Store $\boldsymbol{x}_i$ in the new vector `curr_obstacle`.

Note: $d(\cdot,\cdot)$ is the Euclidean distance, K is a hyperparameter factor to avoid bad groupings of obstacles points.

4) Filter obstacles to extract (the central point of) movable cylindrical obstacles only:

- Initialize the vector `movable_cylindrical_obstacles` (vector of cartesian coordinate points).
- For each obstacle $\boldsymbol{o}_i$ in the `obstacles` vector:
    - If $|\boldsymbol{o}_i| < N$ then discard that obstacle (discard obstacles with less than $N \geq 3$ points).
    - Try to fit a circle model on the distribution of points of $\boldsymbol{o}_i$, by using the set of points $\{\boldsymbol{x}_0, \boldsymbol{x}_m, \boldsymbol{x}_{n-1}\} \in \boldsymbol{o}_i$ (i.e., the first, the middle, and the last one),
        - Fail if the three points are collinear (e.g., case of a wall), with a certain tolerance error ε, in this case just discard $\boldsymbol{o}_i$.
    - Compute the center O_i and the radius r_i of the fitted circle.
    - Filter $\boldsymbol{o}_i$ if the fitted circle has radius r_i greater than a certain value R (for objects with almost collinear points, that fit huge circles).
    - Filter $\boldsymbol{o}_i$ if its points do not all have a distance $d(\boldsymbol{x}_i, O_i) \in [r_i - \varepsilon, r_i + \varepsilon]$ (i.e., filter all obstacles which do not have a circular base), see **Figure 1**.
    - At this point, $\boldsymbol{o}_i$ has a circular base, so it is a movable cylindrical obstacle, then store its center O_i in `movable_cylindrical_obstacles`.

Graphically: below is graphically represented the last step of the algorithm:

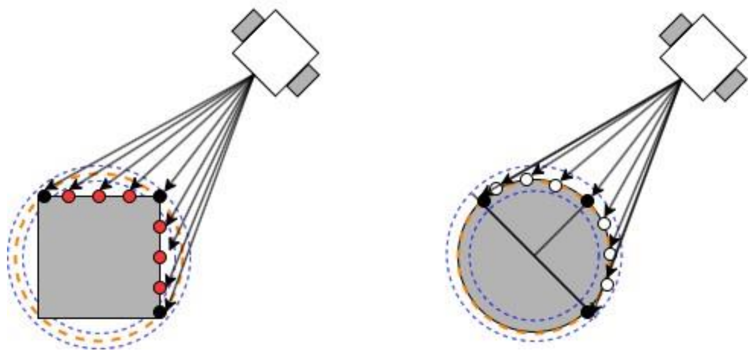

Figure 1 – Representation of how the detection of a movable cylindrical obstacle works

where the orange dash line represents the fitted circle, the two blue dash lines represent respectively the upper and the lower bound (tolerance error: $[r - \varepsilon, r + \varepsilon]$) on the fitted circle, the black points represent the points used for fitting the circle, the white points represent the points falling in the circle fitted model and the red points represent the points falling out the circle fitted model.

As we can see, the first obstacle (a table) does not have a circular shape, indeed many of its points fall out of the range $[r-\varepsilon, r+\varepsilon]$, whereas the second obstacle (a movable cylindrical obstacle) has a circular shape, indeed all its points fall into the range $[r-\varepsilon, r+\varepsilon]$.

Note: before coming up with this approach, which works really well in practice, we tried many other different approaches. One noteworthy to mention is the hierarchical clustering (single linkage) approach, for clusterizing the obstacles, which, unlike what expected, turned out to have really bad results, in addition to the fact that was really slow. We left the relative code in the folder `old` of the project.

Narrow corridor control law

The obstacles detection task is implemented in the function `NarrowCorridorControlLaw::control_law_navigation` which, iteratively, reads the data coming from the laser sensor (topic `/scan`) and sends (by setting up a publisher) the commands of the wheels speeds to the Tiago differential drive robot (topic `/mobile_base_controller/cmd_vel`). To control the Tiago's wheels speeds, it executes the following algorithm, which consists of an infinite while loop which iteratively executes the following steps (see **Figure 2**):

1) **Get laser scan parameters and laser scan data (*ranges*).**
2) **Compute the (Euclidean) distance between the robot and the left and right walls:**
   - Compute the average distances within a window of N points around the two angles $-\pi/2$ and $\pi/2$ (i.e., orthogonally to the robot heading).
3) **Check if the robot is in the narrow corridor:**
   - Check whether $|d_{LW} - d_{RW}| < W$, with d_{LW} and d_{LR} the distances between the left and right walls and the robot, and W the width of the corridor.
4) **Control law:**
   - If the robot is in the narrow corridor, then:
     - Set `was_in_narrow_corridor = true`.
     - Adjust wheels velocity as follows:
       - Define an adjusting factor A, and a smoothing factor S, with $S > A$.
       - Compute the adjusting velocity step A_s and the smoothing velocity step S_s.
       - If $|v_L - v_R| < V$ (prevents the speeds of the two wheels to diverge, causing the robot to rotate on itself, hence V is related to the maximum steering speed), then:
         - If $d_{LW} - d_{RW} > 0$ (the robot is getting close to the right wall), then increase v_R and decrease v_L as follows:
           $$v_L = v_L + A_s, \qquad v_R = v_R - A_s$$
         - If $d_{LW} - d_{RW} < 0$ (the robot is getting close to the left wall), then increase v_L and decrease v_R as follows:
           $$v_L = v_L - A_s, \qquad v_R = v_R + A_s$$
         - Else, the robot is in the center of the narrow corridor (do nothing).

- Else (this generally happens after the robot has steered to adjust its trajectory and align itself with the center of the corridor), smooth the steering by re-balancing the speeds of the two wheels (to avoid oscillations leading to crashes against the walls of the corridor) as follows:
  - If $v_L > v_R$, then:
    $$v_L = v_L - S_s, \qquad v_R = v_R + S_s$$
  - If $v_L < v_R$, then:
    $$v_L = v_L + S_s, \qquad v_R = v_R - S_s$$
- Set the linear speed of the (differential drive) robot as the average linear speed of the two wheels:
$$v = \bar{v} = \frac{v_R + v_L}{2}$$
- Set the steering/angular speed of the (differential drive) robot as the corrected average angular speed:
$$\omega = \bar{\omega} = K \cdot \frac{v_R - v_L}{D}$$
- Send the command to the robot.

- If the robot was in the narrow corridor and now it is no longer in the narrow corridor (i.e., the robot is exit from the narrow corridor), then:
  - Set the linear and the steering speeds to zero (stop it):
$$v = \omega = 0$$
- Else (if the robot was not entered in the narrow corridor yet), let the robot proceed straight at constant speed:
$$v = C, \qquad \omega = 0$$

Note: v_L is the speed of the left wheel, v_R is the speed of the right wheel, v is the linear speed of the robot, ω is the steering speed of the robot, D is the distance between the two wheels, d_{iW} is the Euclidean distance between the robot and a wall.

Graphically: below is graphically represented the functioning of the algorithm:

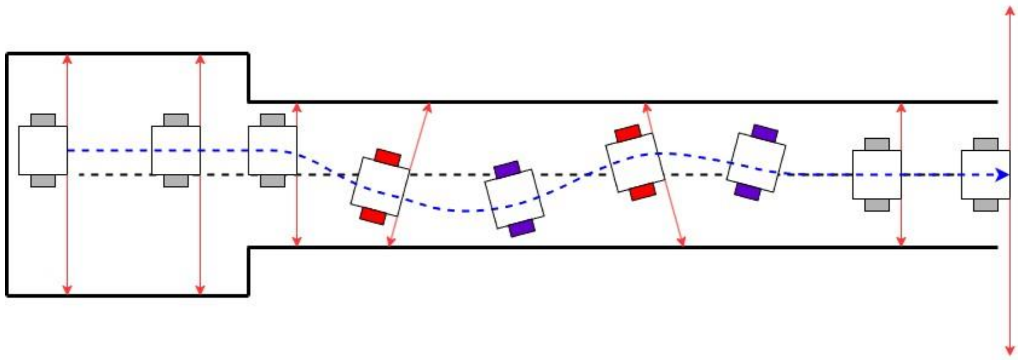

Figure 2 - Representation of the functioning of the narrow corridor control law

where the black dash line represents the center of the corridor, the blue dash line represents the trajectory of the robot, the red arrows represent the average distances between the robot and each wall, the gray wheels represent the fact that the wheels turn at their respective constant speed, the red wheels represent the wheels velocities (and then trajectory) adjustment, and the purple wheels represent the wheels velocities (and then trajectory) smoothing.

As we can see, the robot starts its trajectory out of the narrow corridor, with the two wheels at the same constant speed; when the robot enters the corridor, it starts the control law, i.e., its sequence of adjustments and smoothings, in order to fit its trajectory with the center of the corridor; finally, when the robot exits the corridor, it stops.

Note: if enabled (with `CL_flag=true`), the control law allows the robot to cross the corridor. After that, the robot proceeds reaching the target pose using the `move_base` stack.
If you call two times in the same simulation the launch file provided, the robot at the second shot will re-execute the control law and if has not a corridor in front of it, it will crash, hence the second time you have to disable the control law (with `CL_flag=false`).
For this reason, we suggest you run client and server separately; in this way when more clients send `target_pose`'s to the server with the control law enabled, the server will execute the control law only for the first client and will automatically disable it for the other ones.

Note: the narrow corridor control law navigation simulates a reactive approach, the move_base stack navigation is a deliberative approach, hence, the ensemble of the two approaches provides a hybrid navigation approach.

Members contribution to the project

Osti Simone:

- Implemented and written report about the target pose construction.
- Testing.

Russo Christian Francesco:

- Designed, implemented and written report about the base structure of the program.
- Designed, implemented and written report about the obstacles detection.
- Designed, implemented and written report about the narrow corridor control law.

Spinato Matteo:

- Creating and configuring the repository.
- Designing and implementing the base structure of the program.
- Tuning parameters and testing.