Moratello Matteo, Obetti Christian, Romano Domenico
Master Degree in Computer Engineering
A.Y. 2019/20 Course of Autonomous Robotics

03/18/2020

Report Homework 3

1 Introduction

In the 4.0 industry, autonomous navigation plays a crucial role. A robot that can move autonomously can be very useful in lots of situations, i.e. when moving heavy objects in an area or navigating in environments hardly accessible to humans.
It is also a rapidly growing need, especially for the flourishing autonomous driving cars. Many companies have decided to invest on it and an increasingly number of users is wanting to exploit it as comfort.
For this task, the aim was to make a 3-wheeled robot named Marrrtino navigate in an arena, from a START point to one of the DOCK positions, depending on which of them are free, going through CORRIDOR1, and return going through CORRIDOR2 to return at the START.

2 Arena description

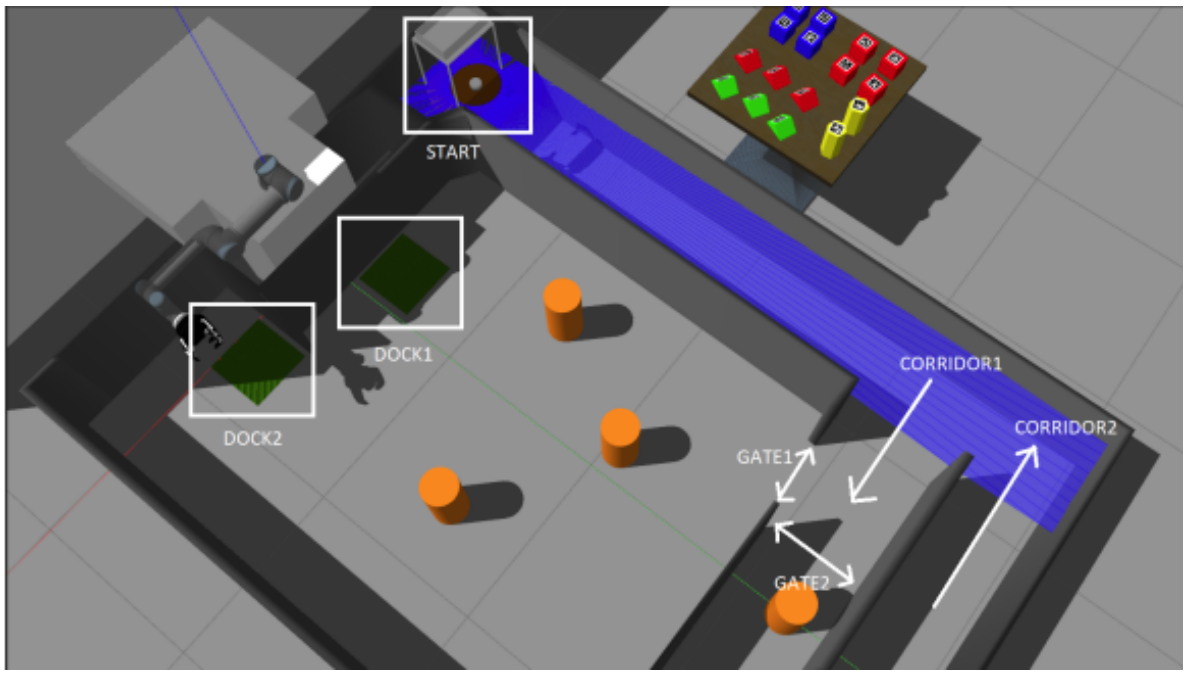

Figure 1: Arena design

As one can see from Figure 1, the arena is characterised by:

- a part made by corridors and narrow passages.
- an open space part.

The arena map is static, except for the orange obstacles, that are placed either in the open space or as gate blocker. As a consequence, we had to develop a ROS node that was able to navigate through both of these environments.
The final goal is to reache DOCK1 or DOCK2, depending of which of them is free, where the robot will be loaded by the manipulator.

3 Robot main features

Marrrtino is a three-wheeled robot and it is able to navigate forward, backward and rotate along its z-axis. It is equipped with a laser scanner, that allows to determine object distances around it. The laser scanner was also helpful to create the arena map.
For our purposes, two publishing nodes were very helpful:

- *scan*, that publishes laser informations.
- *marrtino_base_controller/odom*, that publishes the odometry, so the position of the robot with respect to the arena.

4 Implementation

We decided to structure our node relying on the arena features:

- for the narrow corridors and passages we developed a series of goal-based functions. Since the path along this part of the arena is mainly static, we wrote some maze-related algorithms.
- for the open space, since local dynamic obstacles may be present, we relied on the DWA planner, that plans both locally and globally.

First of all, we identified some phases, each indicating that the robot was navigating in a specific part of the arena. After that, we identified some states to characterize the robot in a specific moment.

4.1 Phases overview

Identified as strings, phases are pretty self-explicative and can be found in the source. When changing phases, the node sets a set of parameters that will be used for the next phase, such as the new goal, the default state, ecc...

4.2 States overview

States are identified by integers:

- 0 indicates the robot must fix its yaw with respect to the goal.
- 1 indicates the robot must proceed forward to reach the goal.
- 2 indicates the robot is done with the current goal and has to change phase to set a new one.
- 3 indicates the robot is navigating through a corridor.
- 4 indicates the robot is navigating in the open space.
- 5 indicates the robot has reached a load/dock position.
- 6 indicates the robot is currently stuck in the open space.
- 7 indicates the robot has completely finished a round.

States 0, 1, 3 and 7 are related to the maze navigation algorithms, 4, 5, 6 are related to open space navigation, state 2 is shared.

4.3 Maze algorithms

To navigate this part of the arena, we made strong use of odometry information. In order to let the robot be as much autonomous as possible, we designed two simple evaluation functions with respect to the robot position and the current goal:

- **err_yaw**, which represents the difference between the desired yaw and the current robot yaw (Figure 2).
- **err_pos**, which represent the euclidean distance between the goal and the current robot position (Figure 3).

As a general case, our first idea has been to implement two functions, fix_yaw and go_straight_on, each one responsible to minimize the respective evaluation function.
When in state 0, the robot evaluates its yaw error. If the value is greater than a tolerance threshold, a Twist message with an angular rotation is sent, otherwise, the state is changed to 1 and at the next iteration, go_straight_on is called, which will check the yaw error once more, then the distance error and then, depending on these values it will send a Twist message with a linear speed or change state back to 0.

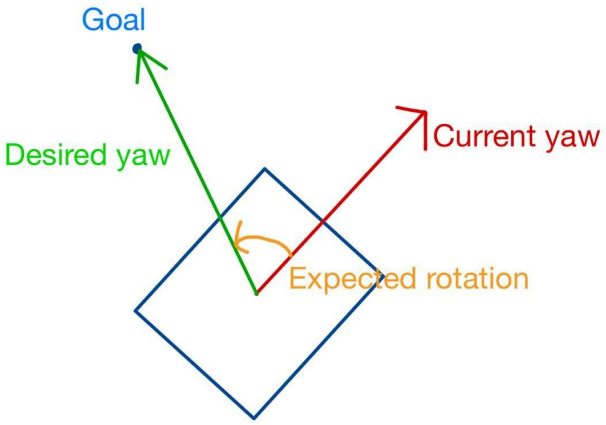

Figure 2: Yaw error

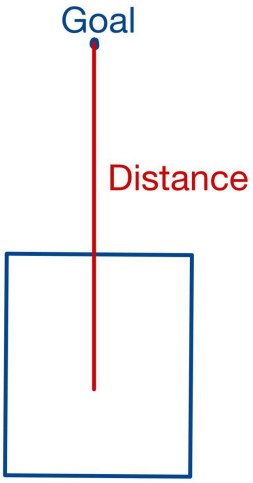

Figure 3: Position error

After the first experiments, we tried to improve our node by implementing a function that makes use of laser informations to follow corridors. This function is used only in phases where a corridor navigation is expected.
At each iteration the position error is evaluated (Figure 4). If bigger then a designed threshold, state is changed to 1 to make the robot navigate forward, but before exiting the function, a Twist message with an angle correction is sent. To achieve this task, front right and front left lasers with an angle of 45° are taken (an average around it to avoid scan errors) and their measure is evaluated. The total width of the corridor is computed and so also the center of it. This point is compared to marrrtino's position and so the rotation angle is computed.

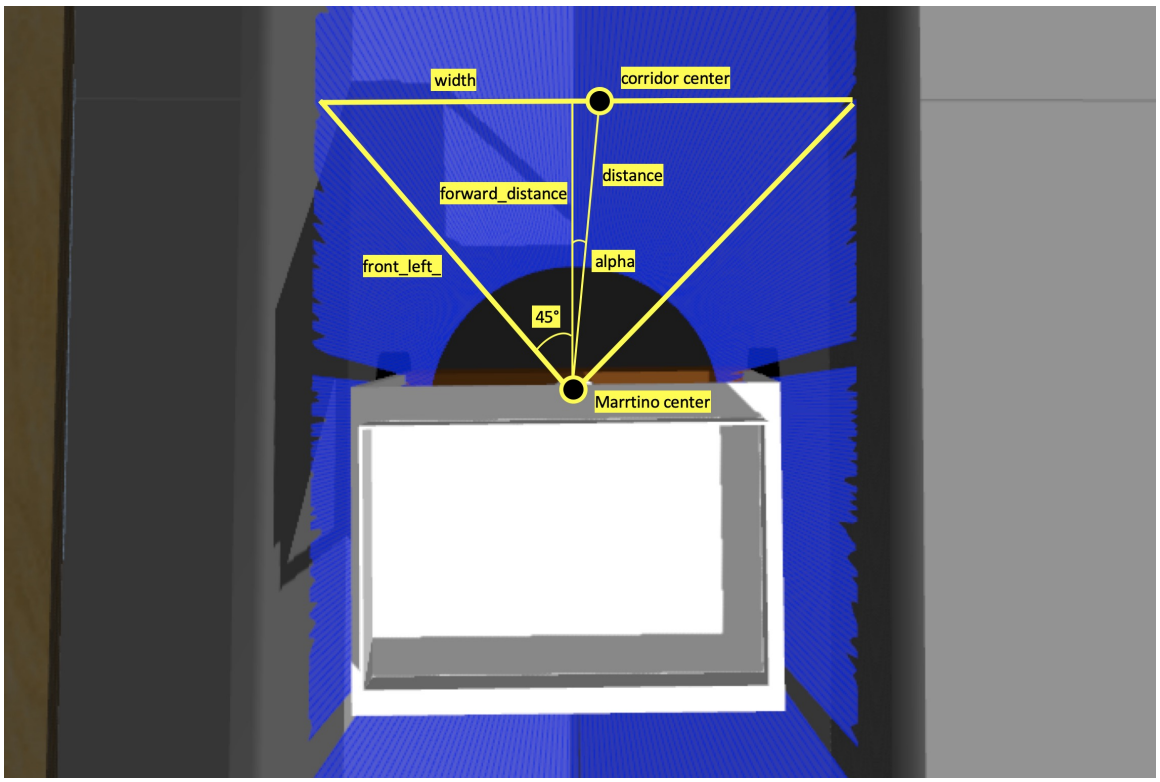

Figure 4: Corridor navigation

This allowed us to get a more stable navigation, although by designed suitable goals, we could achieve the same results with just fix_yaw and go_straight_on.

4.4 Gate choice

The robot could enter the open space either going through GATE1 or GATE2. These gates are mutually exclusive, that means that only one of them is opened when the robot arrives at the gate selection. A gate is closed if there is an obstacle in its center. To understand which gate is open, as one can see in Figure 5, the robot relies on the front laser informations: if the measured distance is bigger than a designed threshold, then GATE2 is open and the robot will proceed, otherwise it will turn and go through GATE1.

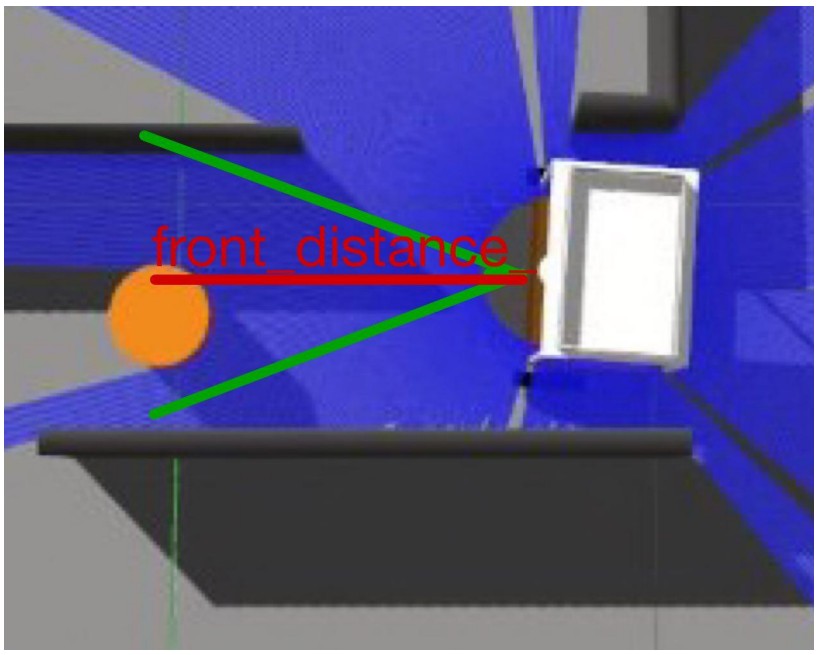

Figure 5: Gate choice and front laser scan

4.5 Open space navigation

Open space is composed by a squared area where random obstacles may be placed. This obstacles are not mapped before so the robot must implement a way to avoid them and reach one of the docking positions where it will be loaded. To achieve this result, we decided to use the DWA planner and MoveBase, a library that contains some useful classes to handle navigation.

Briefly, the Dynamic Window Approach algorithm performs the following steps:

1. Discretely sample in the robot's control space (dx,dy,dtheta).
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

In particular, the most important step is the third. DWA evaluation function value depends on three parameters: path_distance_bias, goal_distance_bias and occdist_scale. To get good results, the tuning of the first two parameters has been difficult. Since the robot has low mobility in general, we decided to give the highest importance possible to goal_distance_bias, which is related to local planning. The higher this parameter, the more the robot will prefer local to global planning. This choice has been made also because the global path for our case was very easy to compute since the robot was navigating in an open space.

As a consequence for this choice, when avoided obstacles, the robot tended to stuck next to walls, that's why we decided to go for some back-up routines to handle cases in which robot was stuck.

When entering the open space, MoveBaseClient sets the goal in terms of pose. The goal is one of the dock positions, which are also mutually exclusive. The default dock position is DOCK1. If it is occupied by an obstacle, MoveBase will soon start to fail its planning. Every 5 seconds, our node will exit the waitForResult() loop and check the current state, which for our case could be:

- SUCCEEDED, the robot will refine its position through handleLoadPosition().
- ACTIVE, the robot will check the distance completed before the last check: if it is greater than a certain threshold, the robot will return inside the waitForResult() loop and proceed, otherwise it will change goal:
  - if the current goal is DOCK1, the new goal will be DOCK2.
  - otherwise, a random goal is taken inside a range in front of it (Figure 6). If the random can be reached, the robot will start back the initial loop, otherwise it will continue setting new random goals.

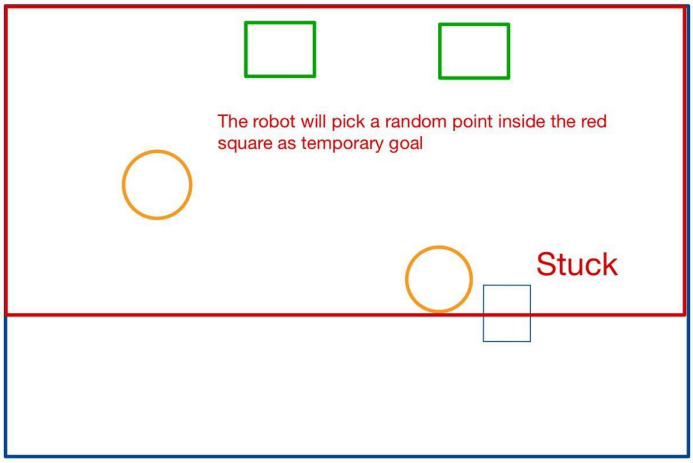

Figure 6: Typical stuck when navigating to DOCK positions

When exiting the open space, the only goal is GATE2. As anticipated before, near the walls the robot may stuck (Figure 7). For the cases when this happens, we implemented a routine that checks laser scans around the robot (180°) and based on where the goal is. The routine will check if there is an obstacle between its position and the goal and if not it will exit MoveBase and proceed using fix_yaw() and go_straight_on() to complete the navigation.

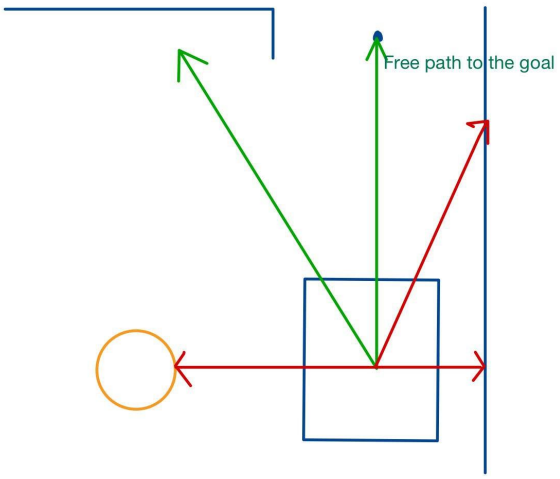

Figure 7: Typical stuck when navigating to GATE2 exiting from open space

4.6 Remarks from rviz

Rviz has been an useful tool to understand the reasons of most of open space fails. We decided to deepen one specific case, which was the exit from open space, in which quite often the robot stuck.

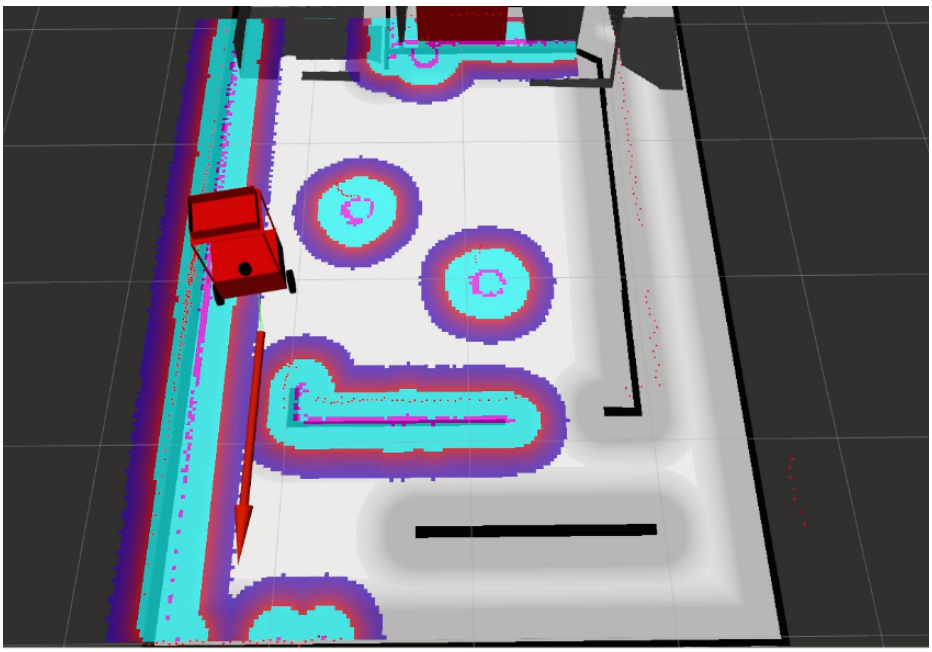

Figure 8: Stuck near the wall in open spaze in rviz

In this case the robot can't decide where to go, since the goal appears occluded. This may be a consequence of giving more importance on local rather than global planning.

5 Source files explanation

We decided to go for a simply structure. The key element is the Navigator class, which contains basically all the functions to perform the navigation. More details about each function can be found directly in the source.
We provided also a parameters file, which contains all the DWA parameters. This is done in order to

let us change parameters quickly.
The “main.cpp” file launches the node and handles ROS loop.

6 How to run the code

The following command runs the node:

$ roslaunch hw3 navigation.launch

7 Conclusion

It has been a difficult task for our group. Since corridors and passages were very narrow with respect to the robot size, parameters and goals position tuning was a task that took us a lot of time.
For example, since we opted for letting the robot turn on place, to complete turns we had to find two intermediate goals: one for starting the turn and one for completing. This choice was taken to avoid collisions with walls. Most of the times we noticed that motion may fail, even if the robot was few centimeters away from the ideal position, that’s why our tolerance parameters are very strict. Also computer’s processing speed had a role. The same algorithm runned in two different computers was running in one and failing in the other. However, the biggest issues were in open space. Although MoveBase brings a lot of useful functions, the understanding of its cycle is not that easy. Moreover, we could not find a smart way to handle planning failures, or better we could have done it reprojecting the entire source structure. Since we decided to keep our maze navigation part, our choices might be a little bit constrained.
Finally, we decided not to implement a real environment version for our node. The reason is that our node relies a lot on odometry that for some reasons is not performing very good in the real case. Indeed, as explained during the report, our node is goal based and goals are positions in the odometry map.