RL-FSR Technical Project

Ciro Arena

March 9, 2022

Contents

Abstract

The following techincal project will present the developing of a control system for a wheeled mobile robot used in a logistic environment, specifically the robot must transport an object from a warehouse to a destination room. In order to know what is the destination room, the wheeled robot must detect a QR-code which contains a suitable ID of that room to know where to move the transported object. During the whole task, the robot will be able to localize itself into the map without using precise information provided by the simulator.

Chapter 1

Introduction

Let's start by presenting an existing imported model from Github, we will show you the wheeled robot which we used to reach the goal of our project.
Then we are going to build the scenario in which robot moves; this environment must be composed by 3 rooms: a warehouse and two destination rooms. We use a tool of Gazebo, Building editor, to create it.
After that we'll show you as the robot can create a map of the environment using SLAM techniques, since robot has to know the envoirnment in which he moves.
To complete our mission, we have to define the trajectory (path + time law) that the robot must follow. To find out the collision-free path starting from warehouse to a destination room, we employ a bi-directional RRT, along with the A* algorithm. Then we implement a trapezoidal velocity profile as suitable time law for each tract of the path, taking into account the maximum wheel velocity for the chosen robot.
Afterwards, to make the robot moves along the defined trajectory as precisely as possible, first we implement high level control composed by both trajectory tracking and regulation controller, and then low level control, that is transmissions for right and left wheel.
Finally, since we can't use odometry simulator as specified according to instructions, we'll implement a self-made odometry with Range-Kutta tecnique.

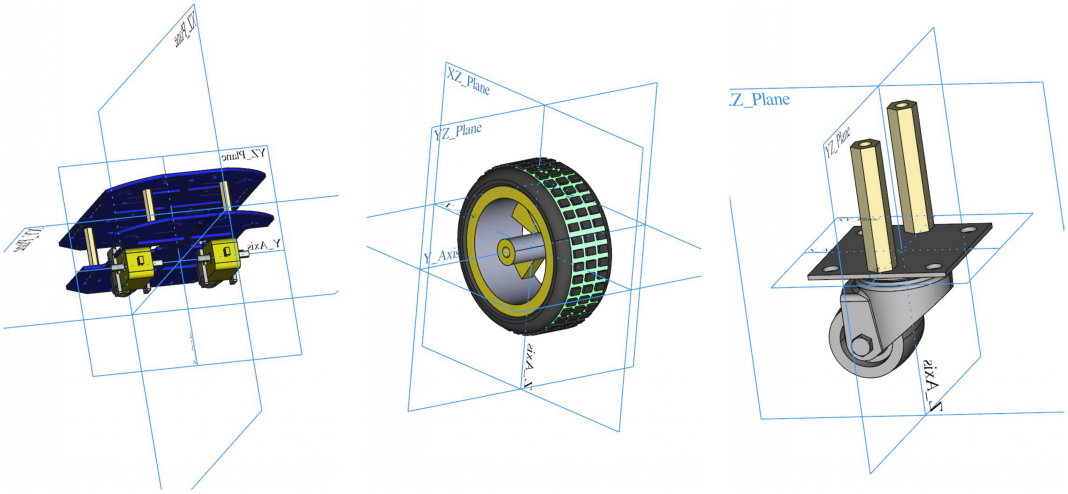
Chapter 2

Robot model

2.1 URDF model

We have already said that the mobile robot which we'll use for our purpose is a wheeled robot, specifically we'll use a differential drive robot as requested from project's essay. It is a non-holonomic wheeled robot and its kinematic and dynamic model is well described on [1].

We haven't used a commercial differential drive, that is a differential drive with datasheet and that we can buy, but we have used an existing imported model that we have downloaded from a repository on Github [2]. In this repository we have some packages, but we employ one only of them and it's called `my_robot_description` in which is contained urdf robot description. The author of the repository, first, has designed a CAD model of this robot, in fact in meshes folder we find every part of the differential drive robot in a file .dae that are previously converted from CAD files. Then he has employed these CAD model parts in order to write xacro file, which are contained in urdf folder. Below we show you pictures of the components designed



(a) *Chassis* **(b)** *Right and left wheel* **(c)** *Caster wheel*

Figure 2.1: Components of the differential drive robot

In order to include these files in the package of our technical project, named `ddr_pkg`, we obviously had to edit them modifying package name in which they are contained. Finally, besides chassis, fixed wheel and caster wheel, we have also available sensors: specifically in *urdf* folder, a camera sensor is modeled as a simple box link, while as in the directory *meshes* we have also a CAD model of the hokuyo sensor, that is a lidar sensor and that will be useful to create map of the scenario. Then if you want to know values of the kinematic and dynamic parameters, you can see file `properties.xacro` in meshes folder.

Now, as mentioned during Robotics Lab and according to what is written on [3], we can check whether urdf files contain errors thanks to ROS that provide us some commands. Before doing so, we must convert xacro file in urdf file, because such commands can be applied only to urdf files, so move us in the `urdf` folder and use the following command to get the conversion:

$ rosrun xacro xacro ddr_bot.xacro > ddr_generated.urdf

Once we have got urdf file, we can use the following command that let us to verify correctness of the urdf file

$ check_urdf ddr_generated.urdf

The `check_urdf` command will parse the urdf tag and show an error, if there are any. If everything is OK, it will output the following:

```
robot name is: differential_drive_robot
---------- Successfully Parsed XML ---------------
root Link: base_footprint has 1 child(ren)
    child(1):  chassis
        child(1):  camera_link
        child(2):  caster_wheel
        child(3):  hokuyo_link
        child(4):  left_wheel
        child(5):  right_wheel
```

Figure 2.2: result of the command check_urdf

If we want to view the structure of the robot links and joints graphically, we can use a command tool called `urdf_to_graphiz`:

$ urdf_to_graphiz ddr_generated.urdf

This command will generate a pdf file of the kinematic chain, as depicted in Fig. 2.3.

2.1.1 Display in RViz

Now we want to be able to visualize robot model that we have designed with urdf files, also to check whether every component is in the right position. RViz help us for this purpose. In order to do this, we must create 2 folders, called `launch` and `rviz`:

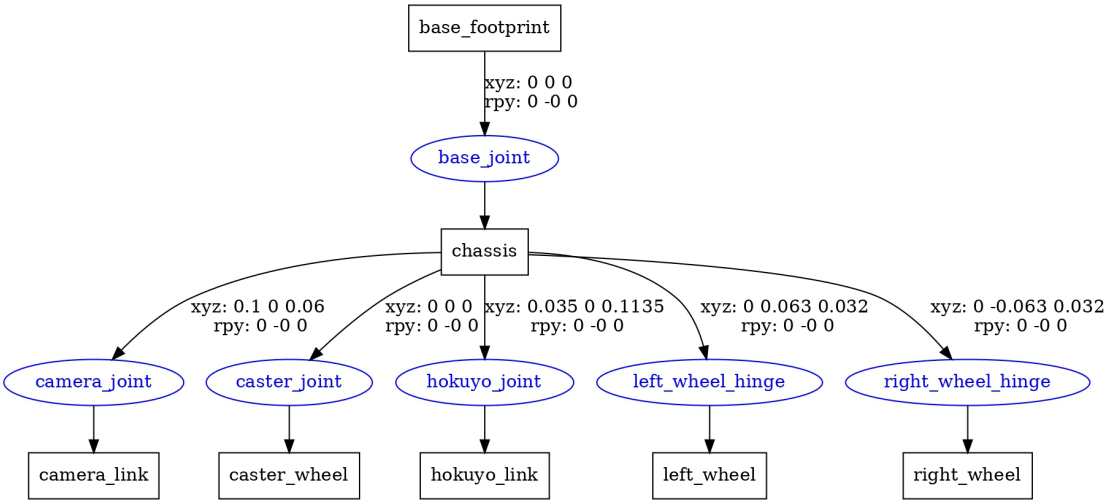


Figure 2.3: Graph of joint and links in the differential drive robot

- in the `launch` folder, we must create a file, that we call it `display_rviz.launch`, in which the conversion from xacro file to urdf file and display in RViz are realized;
- in the `rviz` folder we put a file `display_model.rviz` in which save every change of the model, rather than start RViz, click on `add`, set `RobotModel` and so on.

Therefore, writing the following command in the shell:

$ roslaunch ddr_pkg display_rviz.launch

we get as result Fig. 2.4

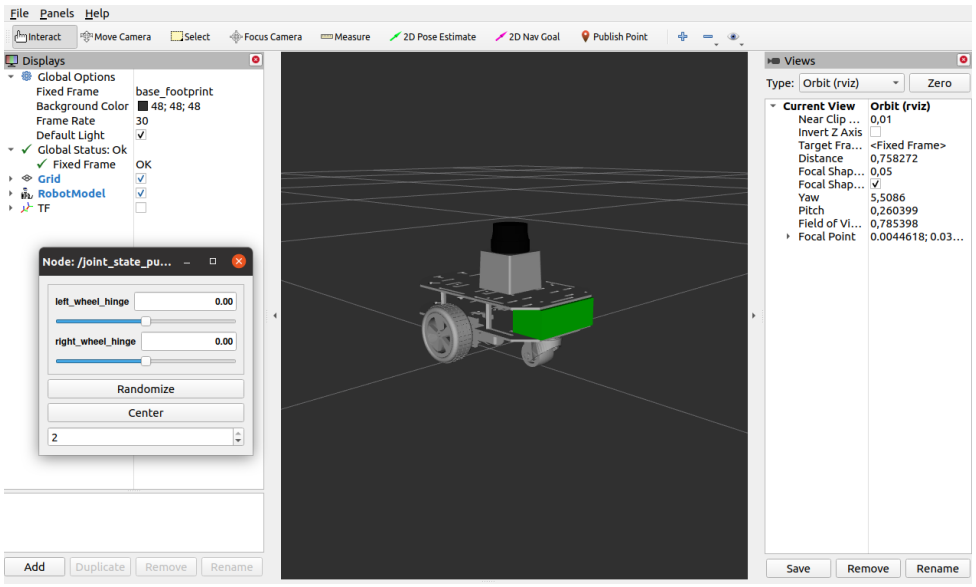


Figure 2.4: Differential drive robot in RViz

Chapter 3

Scenario

The next step is designing the scenario, that is the envoirnment in which differential drive robot moves and performs the task. According to what is specified in the essay, our scenario must be composed by 3 rooms: a warehouse and two destination rooms, that have one or more doors. An example of the working environment is depicted in Fig 3.1.

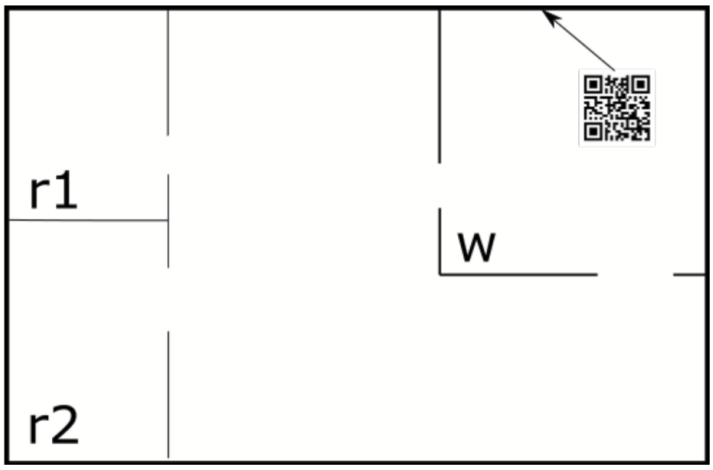


Figure 3.1: Sketch of scenario

After designing scenario, we have to put in the warehouse a QR-code that our robot has to detect in order to know where to go.

3.1 Building editor

For the purpose of designing scenario, we will make use of a tool of Gazebo, named *Building editor*. To learn how to use this tool, we watched [4], in this video are shown every step that we have attended to reach our goal. Final result is depicted in Fig. 3.2.

Once this work is saved as, for example, *Scenario*, a folder with the same name is created: it contains only the sdf model of the scenario. At this point, let's create a new directory inside of our package `ddr_pkg`, and let's call it `models`. In this folder we will put sdf models created that we possibly want to spawn in Gazebo. Consequently we put just created folder `Scenario` in the folder `models`. Then, since we want to spawn in Gazebo

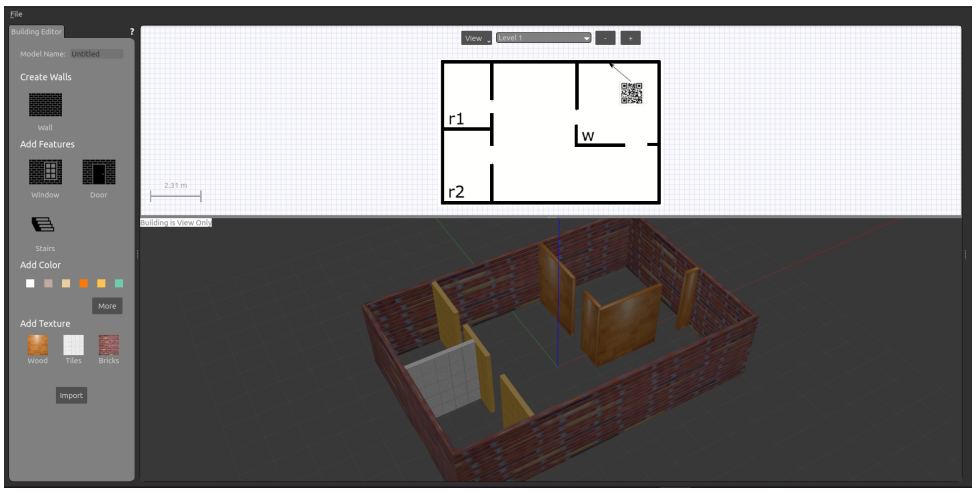


Figure 3.2: Scenario designed by building editor

also a QR-code, it needs to put, besides `Scenario`, inside of `models` another folder that contains sdf model of QR-code that robot must detect. Next, before to spawn robot and scenario in Gazebo, we must create in `ddr_pkg` another folder that we call `worlds`. In this directory we write a xml file in which we include empty world of Gazebo and all sdf models in `models` that we want.

At this point we can create file launch that allows us to spawn robot within a scenario in Gazebo: we name this file launch as `spawn_gazebo.launch`. Using following command:

$ roslaunch ddr_pkg spawn_gazebo.launch

we get as consequence Fig. 3.3.

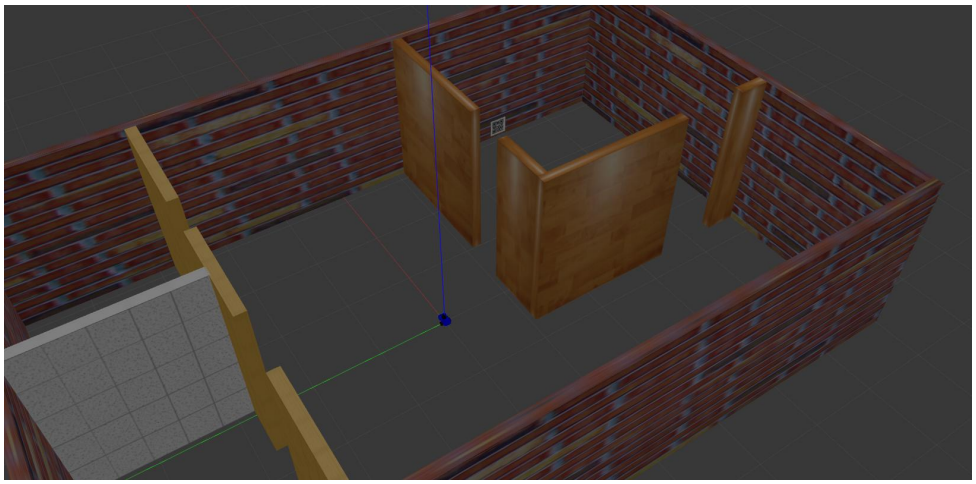


Figure 3.3: Robot and QR-code inside of scenario spawned in Gazebo

Chapter 4

Navigation

After creating and spawing in Gazebo our differential drive robot and Scenario, now we have to take care of mission. First step is creating map of envoirnment, because planner that we will use, employs a static map, so that's why we must generate offline our map, before starting for the mission. In order to do this, we have to exploit a node for the teleoperation to explore scenario, and simultaneously, use a SLAM tecnique to create a 2D map of the envorinment.

4.1 Joy teleop node

Let's start to control our differential drive robot inside of scenario. In order to do this, we can progress in different ways: using `Navigation stack`, with or without an action client written by us, or using a node that teleoperates robot with pc keyboard or a joypad. In this case, to control robot, we are going to use a PS4 joypad by a suitable node. In order to do this we have visited [5], and we have followed every instruction written there.
After connecting joypad to PC, first thing to do is installing this suitable node:

$ sudo apt-get install ros-noetic-joy

Then, we need to start the joy node, but first let's tell the joy node which joystick device to use, that in our case is `js1`:

$ rosparam set joy_node/dev "/dev/input/js1"

Now we can run joy node with following

$ rosrun joy joy_node

Thanks to this node, called `/joy_node`, our joypad is able to comunicate with ROS. In fact if we press some buttons of the joypad, `/joy_node` publishes messages on a topic called `/joy`. Kind of these messages is `sensor_msgs/Joy` and we can see them with:

$ rosrun echo /joy

At this point, our goal is: establish a comunication between joypad and robot, because we want to control it in Gazebo, pressing in a suitable way buttons of joypad. How can we do this? We have to create a cpp node that subscribes to `/joy` and that exploits these `sensor_msgs/Joy` to publish suitable velocity values on topic `/cmd_vel` making robot

move. This node will be called `ddr_joypad_node`.
In order to write such node, we can use the example provided from [5]. Such example has been done to control turtle of `turtlesim_node`, so for this reason we have to modify it in such a way that adapt it for our application. So let's create a file cpp called `joypad_teleop.cpp` and let's put it inside of folder `src`.

Now, instead setting joystick device in the parameters list, running `/joy_node` and then running `ddr_teleop_twist_joy`, we may proceed in a simpler way, writing a file launch that performs all these steps automatically. Also in this case, we can take a file launch already done from [5] and then modify it for our purpose. Let's name such file as `joypad.launch` e let's put it in `launch` folder. At this time, we can start exploration of the scenario using analog sticks of the joypad. First let's spawn robot within scenario in Gazebo

$ roslaunch ddr_pkg spawn_gazebo.launch

and then let's start nodes to connect joypad with ROS and control our robot

$ roslaunch ddr_pkg joypad.launch

Now we are able to control differential drive robot inside of envorinment. In order to check whether different nodes are connected as we expected, we might use a ROS tool called `rqt_graph`. To use it, it needs to write on terminal

$ rqt_graph

In this way it will be shown following graph

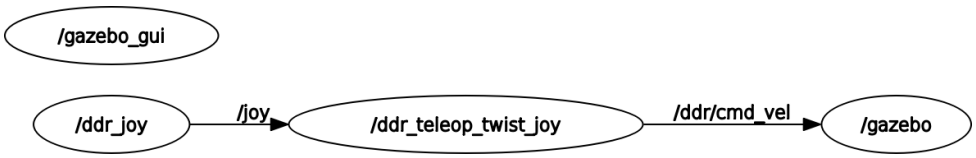


Figure 4.1: rqt_graph output

4.2 SLAM

Next step is building a map of scenario, and in order to this we are going to make use of SLAM (Simultaneous Localization And Mapping) tecniques. SLAM is the main instrument which allows robot to explore unknown envorirments and one of the most famous tools to do 2D SLAM is called *gmapping*. There's a package that implements gmapping and we must download it. It contains a node called `slam_gmapping`, which is the implementation of SLAM and helps to create a 2D occupancy grid map from the laser scan data and the mobile robot pose. That's why at the start, we have mounted on the top of the robot a laser scan.

As usual, we need to create a proper launch file to start 2d SLAM and also in this case we take one already done and then we modify it. We can take it from [3] and we have to be careful of editing laser scan topic's name and then odom frame and base frame according

to what we wrote in `my_ddr.gazebo`. So let's call such file as `gmapping_ddr.launch` and, as usual, let's place it in `launch` folder.

4.3 Building map

At this point, we are ready to build map of scenario. In order to reach this goal we have to execute following steps:

1. Let's spawn robot and scenario in Gazebo

   $ roslaunch ddr_pkg spawn_gazebo.launch

2. Establish a link between ROS and joypad

   $ roslaunch ddr_pkg joypad.launch

3. Start gmapping

   $ roslaunch ddr_pkg gmapping_ddr.launch

4. Now we are already able to generate map teleoperating robot with joypad within scenario, but if we want to visualize map while it is being created, we may open RViz, click on `Add` and select `TF` and then topic `Map`. In this way we see in RViz, robot frame that moves in the map that is being created thanks to lidar. It should appear a result like this:

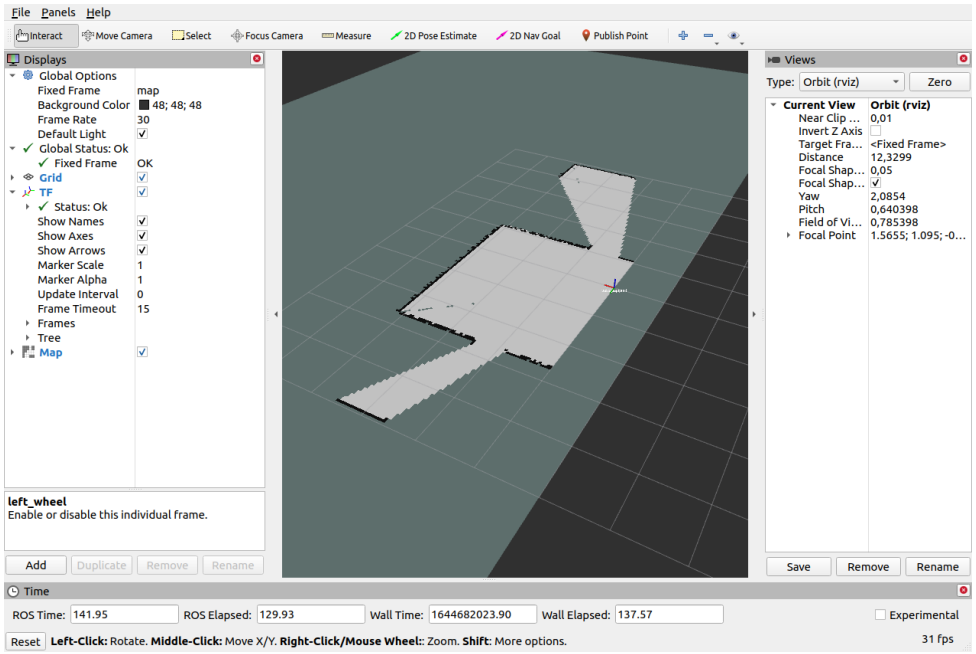


Figure 4.2: Visualization of map creation

Moreover, to avoid opening RViz, selecting `TF` and `Map` whenever we want to build map, we can save it as `map.rviz` in `rviz` folder.

To speed up procedure, we might write a proper launch file that executes automatically steps 2, 3 and 4. We have created such file and we have named it `create_map.launch`

5. Finally, after teleoperating differential drive robot in the envoirnment and building the whole map, in RViz we should get a result as reported in Fig.4.3. If you

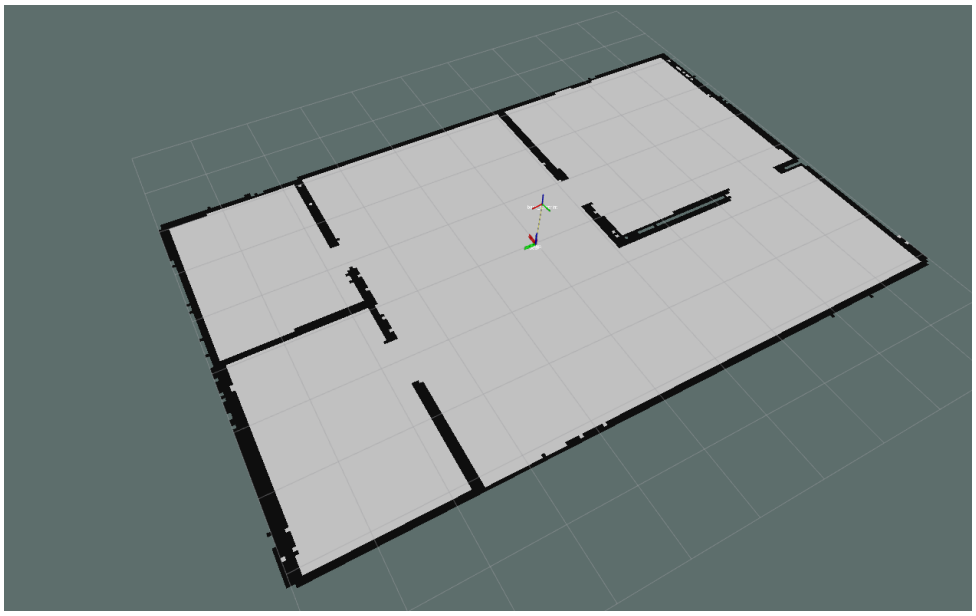


Figure 4.3: Completed map in RViz

remember, since we must use as planner a RRT with A*, we must have a map of the envoirnment already created, therefore we need saving such map. First thing, in order to find and correctly load the map in the ROS system, let's move into the `ddr_pkg` folder, and let's create `maps` directory.

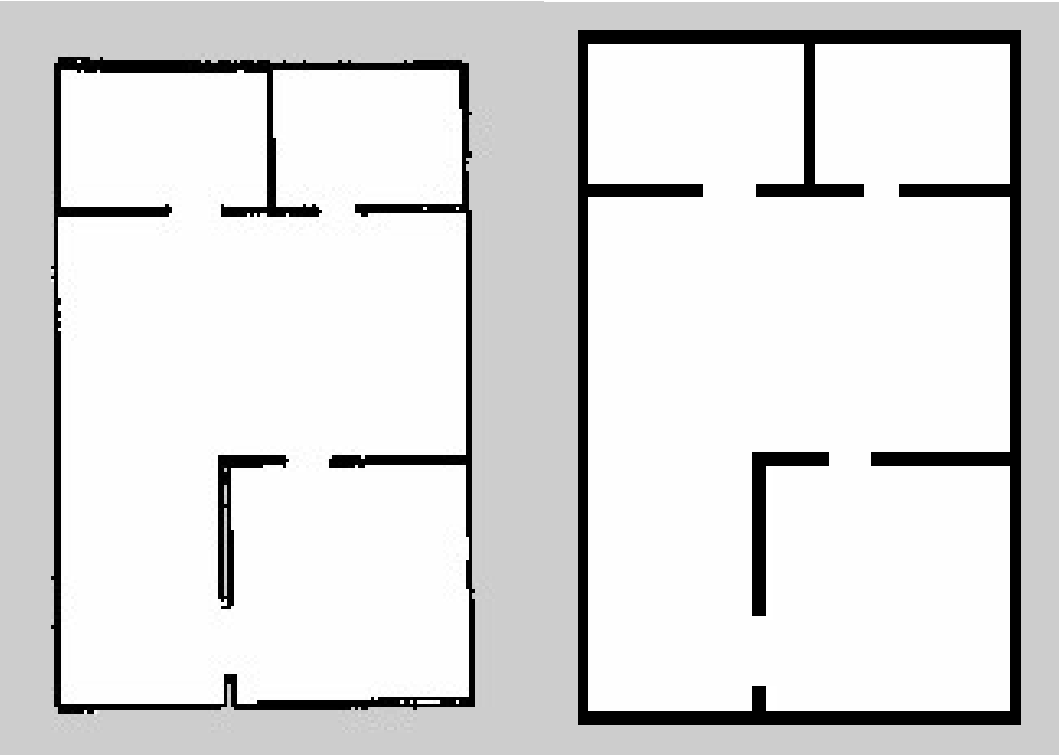
$ mkdir maps
$ cd maps

In this folder we are going to save such map, and we do it thanks to node `map_saver` of the `map_server` package:

$ rosrun map_server map_saver -f map_scenario

This command will listen to the map topic and save into the image, specifically such command generates two files: one is the YAML file, which contains the map metadata and the image name, `map_scenario.yaml`, and second is the image, `map_scenario.pgm`, which has the encoded data of the occupancy grid map.

At the end, if we don't like how map image is created, e.g. because contours are not perfectly rectilinear, or during the navigation we were not been able to scan a small part of contour wall with our lidar and so on, we can make also use of a photo editing application to define strongly obstacles. So we have decided to use `Gimp` to retouch the map image generating in this way a new map. Make a comparison in Fig. 4.4 to notice



(a) *Map generated with gmapping* **(b)** *Map image edited by Gimp*

Figure 4.4: Comparison between two maps

differences. Such new image too is stored in `maps` directory and from now on we are going to use it. To refer at such image we must modify in a suitable way image's name in `map_scenario.yaml`.

4.4 AMCL

Up to this point we applied SLAM tecnique to generate map, because planner that we must use, RRT with A*, needs a static map of the envoirnment. For this reason, during the whole task, the robot must be able only to localize itself into the map. So in this case, the SLAM problem consists only in the Localization part. One popular way to solve localization problem is to use `AMCL` algorithm (Adaptive Monte Carlo Localization) to localize a robot into a given map.

AMCL is a probabilistic localization system for a robot moving in 2D. It uses a particle filter to track the pose of a robot in a known map. The ROS amcl package provides nodes for localizing the robot on a static map. The amcl node subscribes the laser scan data, laser scan based maps, and the TF information from the robot. The amcl node estimates the pose of the robot on the map and publishes its estimated position with respect to the map.

As for gmapping, we must start to write a proper launch file, `amcl_ddr.launch`, that will run the node and customize its configuration setting a list of parameters. As usual, instead of writing from skratch such file launch, we take one already made like that is

provided during Robotics Lab course, and then we modify some paramters as laser scan topic's name, yaml file's name contained into `maps` and finally base and odom frame. Now we can launch `amcl_ddr.launch` by following command:

$ roslaunch ddr_pkg amcl_ddr.launch

But in order to visualize robot inside of the map created, that is able to localize itself, we may open RViz and add some display type as `tf` and topic's name as `/map` and `/laser_scan` (for now), as we have already done for gmapping, and let's save it as `navigation.rviz`. At the end, we have written a proper launch file, `localization_ddr.launch`, that runs AMCL and opens RViz, making automatically previous two steps.

4.5 Package zbar

As it's specified from essay, our robot has to start from a random point, and then move in the warehouse, in which there is a QR-code that must be read from camera upon robot. In order to allow the QR-code detection, we had to install a package to enable such detection. Such package is called `zbar_ros` and we can install it typing

$ sudo apt-get install ros-noeatic-zbar-ros

This package provides us a node, called `barcode_reader_node`, that realizes the detection. And to run it, we must type

$ rosrun zbar_ros barcode_reader_node

It subscribes to a topic called necessarily `/image`, so in this case it's mandatory modify xacro file, changing camera topic name from `/ddr/camera` to `/image`, and then it publishes on a topic, called `/barcode`, a `std_msgs/String` message, to whom we subscribe in our planner.cpp because we need to know what is the destination room. So, in summary

- robot camera publishes `sensor_msgs/Image` data on `/image` topic
- `barcode_reader_node` subscribes to `/image` topic, converts `sensor_msgs/Image` data into `std_msgs/String` data and then publishes them on `/barcode` topic
- At the end, planner node has to subscribe to `/barcode` topic to know string message coded in the QR-code, so we create a proper callback in `planner.cpp`.

Chapter 5

Planner

Come to this point, we must plan the trajectory that our robot should follow to move from a random point to final destination. So, now, let's take care of *planning problem*.
In general, the problem of planning a trajectory for a robot, manipulator or mobile robot, can be broken down in finding a path and defining a timing law on the path. Let's start to find path considering the *motion panning* problem. Such problem consists of finding a collision free path, that is, in the presence of obstacles, we have to plan motions that enable the robot to execute the assigned task without colliding with them.

Planner that we had to implement, as specified from essay, is *RRT*, *Rapidly-exploring Random Tree*, along A* algorithm. In general, RRT is a single-query probabilistic method, that is, it tends to explore only a subset of C_{free} that is relevant for solving the problem at hand. It's so called because it makes use of a data structure called RRT (Rapidly-exploring Random Tree). The incremental expansion of an RRT relies on a simple randomized procedure to be repeated at each iteration. However, the RRT alone is not enough to find a path, because it generates only a graph in free configuration space, without finding a specific path, because in this graph we may indentify more than one path. In order to find an optimum path which links start node and final node in that graph, we must use a graph search algorithm, that is A*. Let's to talk about RRT and then A*. You can find the file .cpp developed in the `src/Planning` directory, and file .h in the `include/ddr_pkg` directory.

5.1 Bi-directional RRT

In general when we talk about RRT, we refer to that procedure in which a Tree is generated starting from start configuration, and after a finite number of iterations that Tree is linked to goal configuration, $\boldsymbol{q}_g$. But there exist different versions of the RRT, and we decided to implement *bi-directional RRT*, in which we have 2 trees, one starting from start node, and the other starting from goal node. So the procedure described previously is repeated twice for both trees.
We made this decision because generating two trees, and not only one, is obviously a more

efficient solution, in fact for tha same number of iterations, using 2 trees instead of one entails a further reduction of the time needed to compute a solution. Moreover we work in a scenario in which there are extensive free region, so that's why it's easier whether we use a bi-directional RRT rather than a simple RRT, because there's free space to generate more than one tree. On the contrary, if we had had a scenario with more cramped space, maybe two trees wouldn't have been necessary. From this point forward, let's indicate start configuration by $\boldsymbol{q}_s$ and goal configuration by $\boldsymbol{q}_g$.

So we have already said that to speed up the search for a free path going from $\boldsymbol{q}_s$ to $\boldsymbol{q}_g$, we're going to implement the bidirectional RRT method that uses two trees. Let's indicate by T_s and T_g these two trees, respectively rooted at $\boldsymbol{q}_s$ and $\boldsymbol{q}_g$. We have used the same symbology also in the code. At each iteration, both trees are expanded with a randomized mechanism that we describe below, i.e. every single step is repeated twice for T_s and T_g:

1. The first step is the generation of a random configuration $\boldsymbol{q}_{rand}$ according to a uniform probability distribution in configuration space $\mathcal{C}$. In order to do this, we used `random` C++ library.

2. The next step is finding configuration $\boldsymbol{q}_{near}$ in T that is closer to $\boldsymbol{q}_{rand}$. That operation is implemented in a function called `find_qnear`.

3. Once $\boldsymbol{q}_{near}$ is found we are ready to generate a new candidate configuration $\boldsymbol{q}_{new}$ to be added for our Tree. Such $\boldsymbol{q}_{new}$ is produced on the segment joining $\boldsymbol{q}_{near}$ to $\boldsymbol{q}_{rand}$ at a predefined distance δ from $\boldsymbol{q}_{near}$. That operation is performed by a function that we have called `calculate_qnew`, in which we calculate unit vector between $\boldsymbol{q}_{near}$ and $\boldsymbol{q}_{rand}$, and then we multiply its x and y coordinates by δ.

4. Before adding that new configuration generated, we have to run a collision check to verify that both $\boldsymbol{q}_{new}$ and the segment going from $\boldsymbol{q}_{near}$ to $\boldsymbol{q}_{new}$ belong to $\mathcal{C}_{free}$. In order to do that, we wrote a c++ function called `collision_check` in which we achieve a correspondence between 2D point of Gazebo and a pixel on map image saved in `maps` directory. We do this because if a map point in the image is black, then it corresponds to a point in Gazebo that is occupied by an obstacle, otherwise it's free. Finally in order to discretize segment which links $\boldsymbol{q}_{near}$ and $\boldsymbol{q}_{new}$ we used same procedure to generate $\boldsymbol{q}_{new}$, but multiplying for a fraction of δ.
   With any luck, T is expanded by incorporating $\boldsymbol{q}_{new}$ and the segment joining it to $\boldsymbol{q}_{near}$, otherwise steps 1, 2 and 3 are repeated

5. After a certain number of expansion steps, the algorithm enters a phase where it tries to connect the two trees by extending each one of them towards the other. This is realized by calculating, for every T_s configuration, distance between them and every T_g configuration. In this way we get a matrix, where T_s nodes are ordered by rows, and T_g nodes by column; at the end, taking the minimum value calculated

in that matrix, we find linking to connect T_s and T_g. As usual we must run a collision check for above linking, and if it is collision-free, the extension is complete and the two trees have been connected; otherwise we should extend one of the two trees, e.g. T_s, adding others nodes to it and then trying again to connect 2 trees according to previuosly described procedure. If collision check fails also in this case, at this point, T_s and T_g exchange their roles and the connection attempt is repeated.

5.2 A*

Until this moment, we implemented RRT to generate a graph, where code is been structured in the following way: a function called `rrt` takes as input start and goal configurations, $\boldsymbol{q}_s$ and $\boldsymbol{q}_g$, and provides as output a graph. In above function there's another function called `generate_tree` that generates T_s and T_g, that in turn performs steps described in the previous section. So, after RRT is finished, a graph is been generated, and then we can apply A* to find minimum path. In general, when we talk about *minimum path*, we refer to a path got according to the minimum cost, because every linking between 2 nodes are labelled by a positive number called *weights*, and sum of these weights is the *cost*. In our case the nodes generally represent points in configuration space, so the weight of an arc is the length of the path that it represents. Therefore the minimum path is the shortest path among those joining starting node, N_s, and goal node, N_g. In order to calculate that cost for every node of the graph, we must use a cost function $f(N_i)$ defined as

$$f(N_i) = g(N_i) + h(N_i)$$

where:

- $g(N_i)$ is the cost of the path from N_s to N_i
- $h(N_i)$ is a heuristic estimate of the cost $h*(N_i)$ of the minimum path between N_i and N_g .

In our case, since the nodes represent points in configuration space, that is in particulat a Euclidean space, such heuristic was been chosen as the Euclidean distance between N_i and N_g. In fact, the length of the minimum path between N_i and N_g is bounded below by the Euclidean distance.

As usual, to implement A* in C++, we have written a function in `Astar.cpp` file, called `Astar_search` that takes as input the graph provided from RRT, and provides as output the the shortest path. That A* was been written in C++ along the lines of pseudocode depicted in [1] and that we report below in Fig 5.1. Compared to RRT, now it doesen't need to make particular explanations about reasons on how to implement algorithm, beacuse there aren't many versions of A* unlike RRT, so we have written, by C++, pseudocode depicted above and stop. Code was been properly commented in order to identify every step of algorithm in a very simple way.

$A^\star$ algorithm
1 **repeat**
2 find and extract N_{best} from OPEN
3 **if** $N_{\text{best}} = N_g$ **then** exit
4 **for** each node N_i in ADJ(N_{best}) **do**
5 **if** N_i is *unvisited* **then**
6 add N_i to T with a pointer toward N_{best}
7 insert N_i in OPEN; mark N_i *visited*
8 **else if** $g(N_{\text{best}}) + c(N_{\text{best}}, N_i) < g(N_i)$ **then**
9 redirect the pointer of N_i in T toward N_{best}
10 **if** N_i is not in OPEN **then**
10 insert N_i in OPEN
10 **else** update $f(N_i)$
10 **end if**
11 **end if**
12 **until** OPEN is empty

Figure 5.1: A* pseudocode

5.3 Visualization markers

After performing RRT along A* in C++, we have to verify that everything works properly, but in order that we can check the code correctness, we need to *see* nodes generated and what is the minimum path. Luckily there are some tools to do it. First, we could use opencv library to bring up nodes in the map image in a very similar way to what we used for collision check, or else we could use *visualization markers* that are basic shapes markers can be visualized in RViz. In order to use them we must be careful to add as dependency `visualization_msgs`, and to visualize them in a RViz window, don't forget to add as display type `Marker`. As we already said, there are visualization marker of different shapes: we used spheres to represent nodes, and arrows to represent archs. Then we used colour red for T_s and blue for T_g. Then, in order to highlight minimum path found with A*, we used green nodes. Result is shown in Fig. 5.2

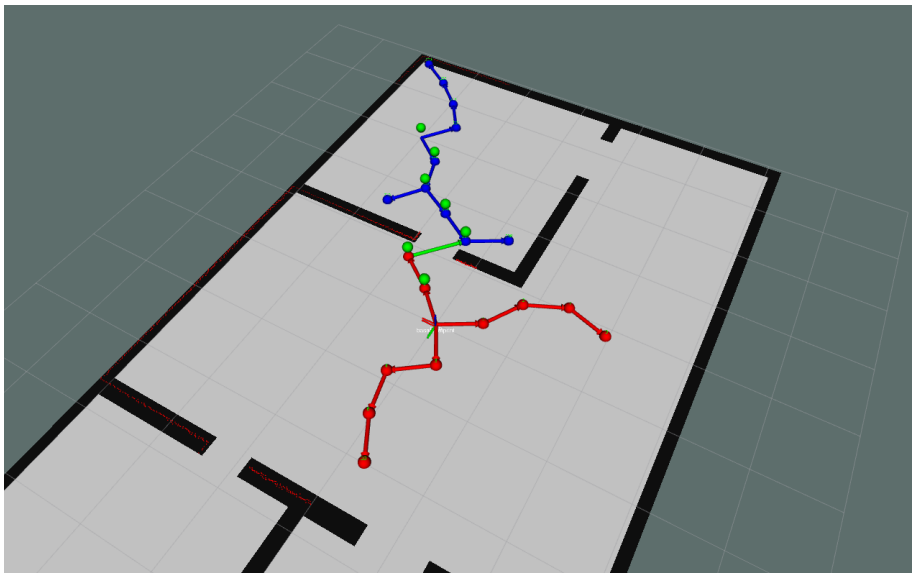


Figure 5.2: Visualization markers on RViz

Finally, to be even safer, that every node is located in the 2D position correctly calculated, we can use another kind of shape that is called, `TEXT_VIEW_FACING`, which let us to show text in RViz. Specifically we used it to show nodes ID upon them.

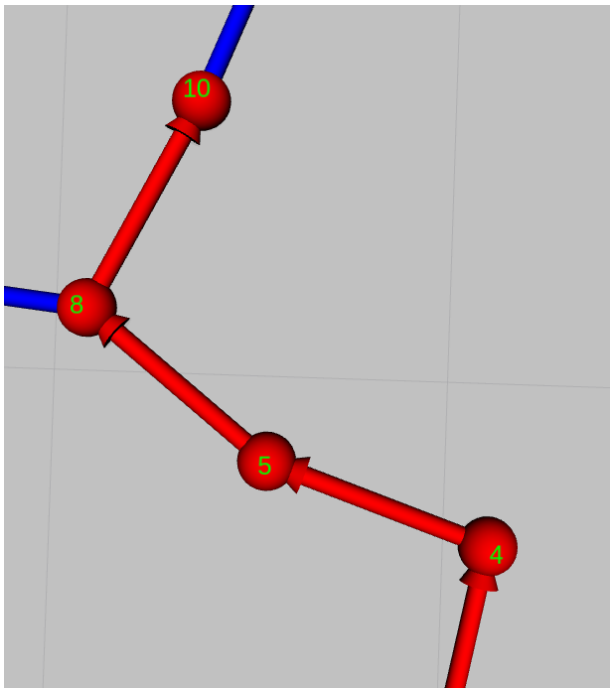


Figure 5.3: text view facing visualization marker

That code can be read in `visualize.cpp` file.

5.4 Trajectory

Until now we took care of exploring configuration space, without defining a specific path, infact we just found trajectory way points in $\mathcal{C}_{free}$, but without interpolating them. So it's clear that, since our controller has to take as input a trajecotry, we must define a certain path primitive between two adjacent nodes and a time law on it.

Remember that we are using a differential drive robot, that is a non-holonomic robot, that's why if we wanted to get a motion without interruptions, we should apply a proper RRT version for non-holonomic robots. Since it would be to much difficult for us, we implemented the RRT version just seen. We say this, because in such case we must consider linear segments to connect each node of the graph to the next one. As result, to define a proper rectilinear path, we may use same parametric representation already studied on [1], that is:

$$\boldsymbol{p}(s) = \boldsymbol{p}_i + \frac{s}{\|\boldsymbol{p}_f - \boldsymbol{p}_i\|}(\boldsymbol{p}_f - \boldsymbol{p}_i) \tag{5.1}$$

where $\boldsymbol{p}_f$ and $\boldsymbol{p}_i$ are, respectively, final and initial point of the rectilinear path, and s is arc length of the primitive path and it's a function of the time such that $s(t_i) = 0$ and

$s(t_f) = \|\boldsymbol{p}_f - \boldsymbol{p}_i\|$. And its time derivative is

$$\dot{\boldsymbol{p}}(s) = \frac{\dot{s}}{\|\boldsymbol{p}_f - \boldsymbol{p}_i\|}(\boldsymbol{p}_f - \boldsymbol{p}_i) \tag{5.2}$$

Now path definition is complete.

At this point we just have to define a proper timing law on these rectilinear paths. In order to do this, we have to choice in a suitable way values that $s(t)$ has to assume in a certain time interval. Also in this case we can use methods studied during Robotics Foundation course on [1], in particular tecniques to implement Point-to point motion as polynomials of third or fifth order, and trapezoidal velocity profile. For no particular reason, we decided to assign to $s(t)$ values according to trapezoidal velocity profile, coding in C++ the same formulas that we can find on [1].

Strictly speaking, we should specify rectilinear path primitives and timing law on them, inside of exact same file in which we coded in C++ planner. Actually, because of difficulties to send controller, from planner, every value of $\boldsymbol{p}(s)$ and $s(t)$, we have coded them within class controller.

Chapter 6

Controller

After defining collision free trajectory, thanks to the planner, now we must ensure that our robot follows such trajectory as precise as possibile, i.e. the error must tend asymptotically to zero. So, at the moment, we have to deal with the projecting of the controller, which has to take as input target position and velocity provided from planner, and then it has to provide as output angular velocities of right and left wheels to make sure that differential drive robot follows asymptotically trajectory previously defined.

In general the control problem can be broken down in two basic problems:

- *Trajectory tracking*: the robot must asymptotically track a desired Cartesian trajectory, starting from an initial configuration that may or may not be "matched" with the trajectory.

- *Regulation*: the robot must asymptotically reach a given posture (position and orientation), without considering particular path to follow.

In our case we have to solve both kinds of problems, because we have already said that path provided from planner, consisting of many segments that connect two by two nodes of minimum path, and if we want to produce a movement without interruptions, we shoud have used another kind of RRT suitable only for non-holonomic robots. Therefore, to follow trajectory given by our planner, we have to track a rectilinear path from strating configuration $\boldsymbol{q}_0$ to desired configuration $\boldsymbol{q}_d$ corresponding to two adjacent nodes, hence we have to use a trajectory tracking tecnique to do it. Then, in order to align robot along next segment of the path, it has to rotate on the spot according to a certain orientation. We can do it by a posture regulation tecnique.

6.1 I/O Linearization

Let's start to talk about what trajectory tracking controller to implement. We have just said that mobile robot has to follow a rectilinear path between 2 adjacent nodes, and at the end of such path, robot must stop because it has to rotate on the spot to align

with next segment. So we have to implement a controller that allows robot to follow such trajectory, making it stop at the end of the path.

Now we know three different trajectory tracking controllers: Control based on approximate linearization, Non linear control and Input/Output Linearization. The only one that satisfys the above requirements is I/O Linearization because the other two controllers allow robot to follow trajectory, in terms of position and orientation, but they can be used only for *persistent* trajectories, i.e. robot at the end of the path has to move with a non-zero linear/angular velocity.

Compared to them, I/O Linearization doesn't necessarily require a non zero velocity at the end of the trajectory, so it's the good chioce for our case, but such controller allows robot to follow trajectory only in terms of position. Orientation is free, it is not controlled. That's why we have to resort to a posture regulation. Before talking about posture reguation controller, though, we should illustrate all the steps we have followed to implement such trajectory tracking controller:

- Let's start with targets, to talk about how target position, $y_{1,d}$ and $y_{2,d}$, and target velocity, $\dot{y}_{1,d}$ and $\dot{y}_{2,d}$, are generated. We obtain them thanks to 5.1 and 5.2 respectively, which are obtained in turn thanks to trapezoidal velocity profile tecnique, that it's used to calculate arc length s and its time derivative $\dot{s}$, as we have already said.

- Then we have to feed back to such targets, input of our system, the system output. Unitil now, output of differential drive we have considered was composed by coordinates x and y of the centre of the unicycle and the orientation θ. In order to make possible a linearization, in this case we have to consider a different output for the position:
$$\begin{cases} y_1 = x + b\cos\theta \\ y_2 = y + b\sin\theta \end{cases} \tag{6.1}$$
So, not more the centre of the unicycle, but a point on the sagittal axis that is a length b from the centre of the unicycle. Such values can be estimated subscribing to odometry topic obtaining in this way x, y e θ, while parameter b is chosen by us.

- After that, once computed reference positions and velocities, and then actual output pose estimated 6.1, we are ready to calculate control input to be provided to our control system
$$\begin{cases} u_1 = \dot{y}_{1,d} + k_1(y_{1,d} - y_1) \\ u_2 = \dot{y}_{2,d} + k_2(y_{2,d} - y_2) \end{cases} \tag{6.2}$$
where k_1 and k_2 are two positive gains set by us.

- Since we should provide linear and angular velocities to differential drive robot to make it move, finally, we have to calculate v and ω in function of control inputs u_1

and u_2. We can do that according to following equation

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = T^{-1}(\theta) = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{6.3}$$

where

$$T(\theta) = \begin{bmatrix} \cos\theta & -b\sin\theta \\ \sin\theta & b\cos\theta \end{bmatrix}$$

The control system of such controller is represented in Fig.

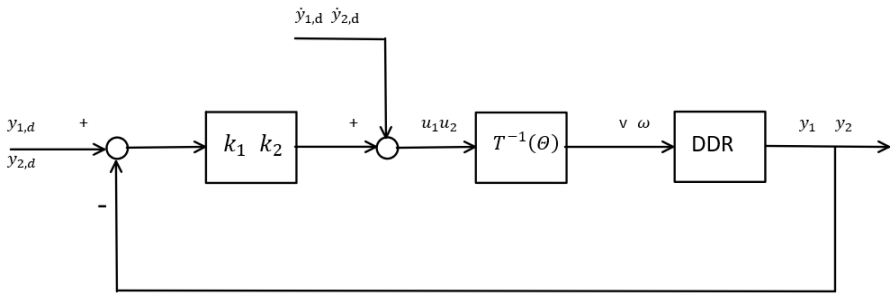


Figure 6.1: Block scheme of IO Linearization

6.2 Posture regulation

When we arrive at the and of the path, thanks to trapezoidal velocity profile implemented for arc length, robot stops. That's when posture regulation has to be run. As usual there are different kinds of regulation: cartesian regulation and posture regulation. The former allows robot to reach a given position, ignoring the orientation, whereas the latter includes also the orientation. Since we want that robot to rotate on the spot, we have to implement a posture regulation controller, that is able to regulate the whole configuration vector (cartesian position and vehicle orientation), but customized only for the orientation part. So we must use formulae on [1] and then particularizing them in order to carry out a posture regulation only for the orientation. Let's see how we ought to proceede.

Let's define (x_i, y_i) coordinates of initial position and (x_f, y_f) coordinates of final position. As reported on [1], first of all we should express the problem in polar coordinates defining ρ, γ and δ. As we have already said, though, we have to apply an *orientation regulation*, then in this case we don't need to ρ and δ. So we must define only γ:

$$\gamma = atan2(y_f - y_i, x_f - x_i) - \theta \tag{6.4}$$

Therefore linear velocity will be equal to zero, whereas angular velocity can be computed through the following modified formula:

$$\omega = k_2\gamma + k_1 \sin\gamma\cos\gamma \tag{6.5}$$

where k_1 and k_2 are positive gains chosen by us, and totally different from gains defined for trajectory tracking.

Such parameters values, such that gains for trajectory tracking controller and posture regulation controller, distance b etc. are specified in a file called `plan_ctrl_param.yaml` within `config` folder.

6.3 Low level control

We have understood that the aim of above controllers is calculating linear and angular velocity that our robot must have so that it follows the desired trajectory with an error which tends to zero. In ROS we could use such velocities because in ROS there is `/ddr/cmd_vel` topic on which we can publish linear and angular velocities values and robot moves. In the reality though we can't proceed in this way, because robot's wheels have velocity controllers, so we have to provide as input angular velocities of right and left wheel, ω_R and ω_L, and not linear and angular velocities v and ω. We know that it's possible express v and ω on the basis of ω_R and ω_L thanks to following formulae

$$v = \frac{\rho_W}{2}(\omega_R + \omega_L) \qquad \omega = \frac{\rho_W}{d}(\omega_R - \omega_L) \tag{6.6}$$

where ρ_W is the wheel radius and d is the distance between two wheels.

Since we have just obtained v and ω through above controllers, we can use 6.6 in order to get the inverse expression, that is ω_R and ω_L computed from v and ω. Adding and subtracting side to side of two equations, it's quite easy to get that:

$$\omega_R = \frac{2v + d\omega}{2\rho_W} \qquad \omega_L = \frac{2v - d\omega}{2\rho_W} \tag{6.7}$$

Afterwards we must publish ω_R and ω_L values so calculated on available topics created by us thanks to another yaml file, called `ddr_low_control.yaml`, contained in `config` folder.

Chapter 7

Odometry

Last part of such project is devoted to Odometry. As specified from essay, meanwhile the robot performs the task, the robot must be able to localize itself into the map without using precise information provided by the simulator. Also, in order to implement the controller, we don't have to feed back to the controller the pose retrieved from the odometry of the simulator. Therefore, for our project, we have to create a custom odometry according to what has been studied during Field and Service Robotics course.

First of all, let's say that we have implemented odometry as a C++ node, in a file called `odometry.cpp` in the folder `src`. On the basis of what is written in [1], we used following formulae to estimate position and orientation (yaw angle) of a mobile robot modelled as a unicycle:

$$x_{k+1} = x_k + \frac{v_k}{\omega_k}(\sin\theta_{k+1} - \sin\theta_k) \tag{7.1}$$

$$y_{k+1} = y_k - \frac{v_k}{\omega_k}(\cos\theta_{k+1} - \cos\theta_k) \tag{7.2}$$

$$\theta_{k+1} = \theta_k + \omega_k T_s \tag{7.3}$$

As we can notice, the first two are not defined for $\omega_k = 0$; so, in this case, we have to use others formulae, as for example the second-order Runge–Kutta integration method, which is exact over line segments

$$x_{k+1} = x_k + v_k T_s \cos\left(\theta_k + \frac{\omega_k T_s}{2}\right) \tag{7.4}$$

$$y_{k+1} = y_k + v_k T_s \cos\left(\theta_k + \frac{\omega_k T_s}{2}\right) \tag{7.5}$$

In the implementation, in order to handle such singularity, we have used a conditional instruction.

However, to calculate position and orientation we have to know linear velocity and angular velocity in a certain sample time, v_k e ω_k, but in the reality we haven't sensors to estimate those velocities. Rather, we have sensors to measure angular velocities of right wheel and left wheel, or even better, incremental encoders to calculate displacement of

right and left wheel every sampling time. Consequently, we have structured the code in the following way:

- you subscribe on topic `/joint_states` and you take right and left wheels'positions in every sampling time interval in order to compute displacement between two consecutive sampling instants of right and left wheel, $\Delta\phi_R$ e $\Delta\phi_L$;
- After that, we must use such displacements of right and left wheels to calculate linear and angular displacement, Δs e $\Delta\theta$, of differential drive robot by following formulae

$$v_k T_s = \Delta s = \frac{\rho}{2}(\Delta\phi_R + \Delta\phi_L) \qquad \omega_k T_s = \Delta\theta = \frac{\rho}{d}(\Delta\phi_R - \Delta\phi_L)$$

- Finally we substitute such new expressions just calculated in the above formulae used to compute odometry with numerical integration.

Then, we are going to publish results that we obtain on a topic, that we name always in the same way as `/ddr/odom`, we'll publish on it the same message type which is published on odometry topic of the simulator, that is `nav_msgs/Odometry`, and then we fill its fields with same values. We do this, because in this way it's not necessary modify file launch about gmapping or amcl. This node has to be run before planner and controller.

7.1 Imu sensor

Odometry projected in this way, unfortunately, doesn't work very well, in fact, because of derive phenomenon and because of robot that rotates on the spot many times during the path, orientation estimation gets worse very quickly compared to position estimation, and so after few minutes of simulation, robot can't localize itself anymore precicely. In fact the orientation error affects also position coordinates because the latter depends on the former according to 7.1 and 7.2. Therefore during the task, even though we obtained a collision free path, robot collides against an obstacle. For this reason we decided to use an Imu sensor in order to estimate orientation in a very accurate way.

To do that, we had to add a plugin in robot xacro files, and then in `odometry.cpp` we had to subscribe to `/ddr/imu` topic. Proceeding in this way, also the position error disappears.

Chapter 8

Graphics and results

At the end of such project, in order to verify that everything works correctly, it's necessary to report the most meaningful graphics to make sure the efficiency of control system that we have designed along the custom odometry. We will simulate the complete task and we will plot graphics of tracking error (for position) and regulation error (for orientation) using a ROS package called `rqt_multiplot`.

So let's run the simulation, and then let's start to plot error about x and y coordinates. To do that we have created a new topic, called `/error_topic` on which we publish `geometry_msgs/Pose2D` data, and in `rqt_multiplot` we will report trend of x error and y error respect to data of topic `/clock`, that is respect to time. So results are shown below:

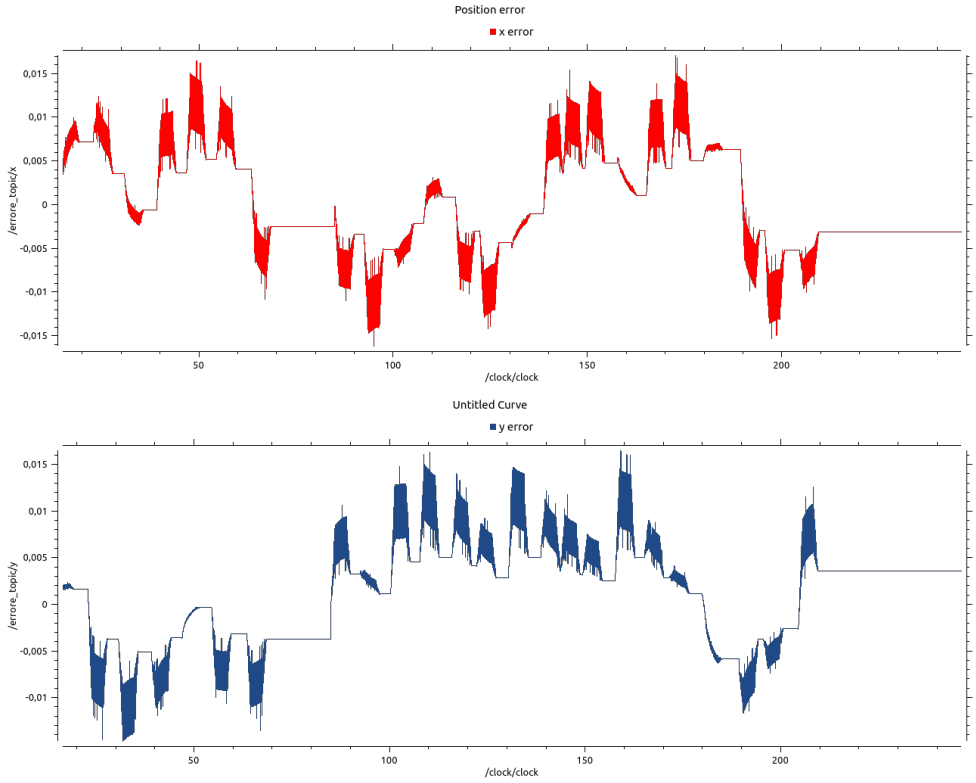


Figure 8.1: Position error

Regarding orientation, we have a regulation error, and not a tracking error as before,

so, first, the orientation error is not always very small, close to zero, such as x and y errors, but it's huge at the beginning and then tends asimptotically to zero; second, in the code we have set a treshold for the orientation error, because we coded that when the orientation error is lower than treshold (i.e. 0.1), then robot can move along rectilinear path to reach next path point. So orientation error never tends to zero, but it tends to such treshold that we have set close to zero, but it isn't a problem for us:

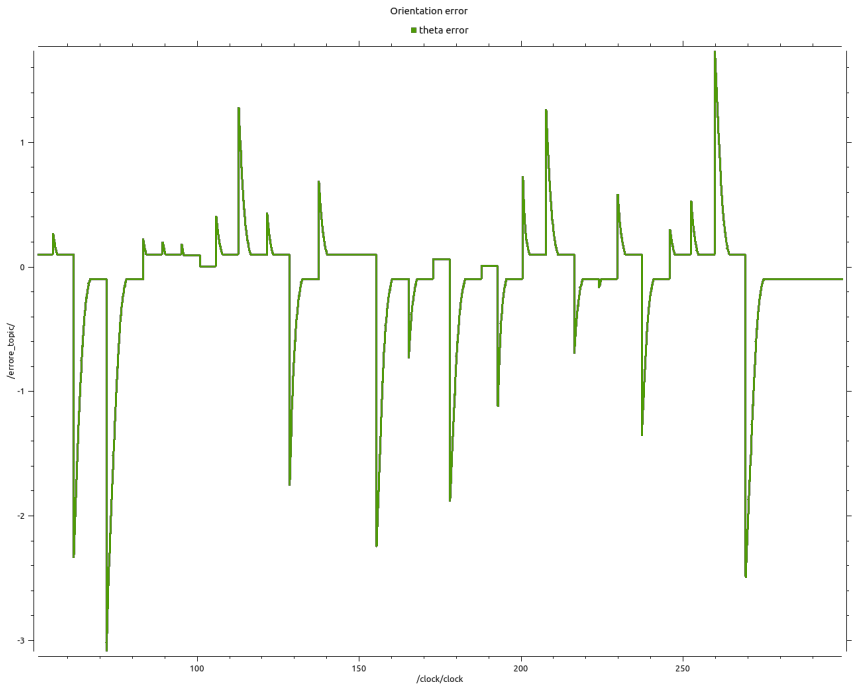


Figure 8.2: Orientation error

Bibliography

[1] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo. *Robotics Modelling, Planning and Control.* Springer, 2009.

[2] `https://github.com/eborghi10/my_ROS_mobile_robot`

[3] L. Joseph, J. Cacace. *Mastering ROS for Robotics Programming*, Second Edition. Packt, 2018.

[4] `https://www.youtube.com/watch?v=Y2LRHl79b3g&t=632s`

[5] `http://wiki.ros.org/joy/Tutorials`