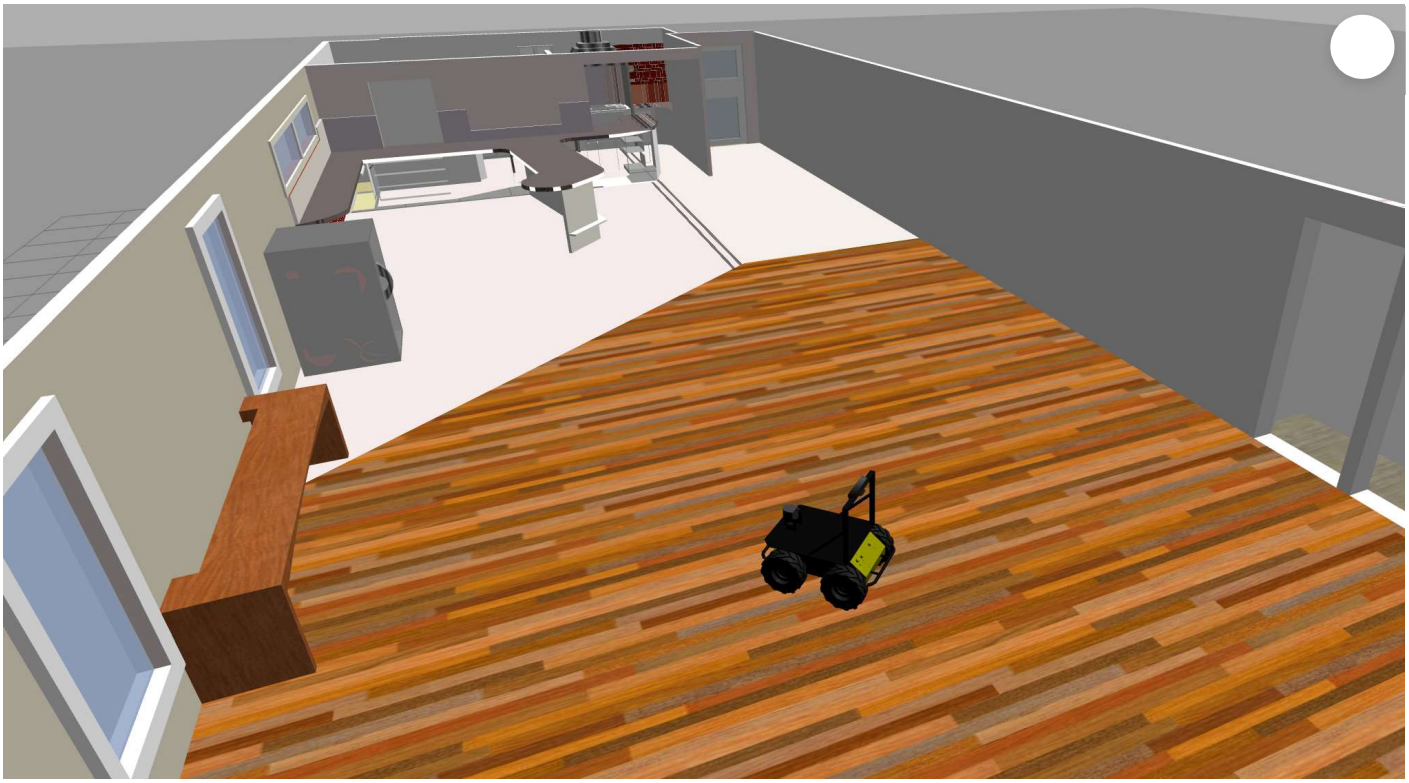
ROS Navigation in 5 Days

Unit 4: Path Planning Part 1



- Summary -

Estimated time to completion: **3 hours**

What will you learn with this unit?

- Visualize Path Planning in Rviz
- Basic concepts of the move_base node
- What is the Global Planner?
- What is the Global Costmap?

- End of Summary -

We're arriving at the end of the course, guys! For now, we've seen how to create a map of an environment, and how to localize the robot in it. So, at this point (and assuming everything went well), we have all that we need in order to perform Navigation. That is, we're now ready to plan trajectories in order to move the robot from pose A to pose B.

In this chapter, you'll learn how the Path Planning process works in ROS, and all of the elements that take place in it. But first, as we've been doing in previous chapters, let's have a look at our digital best friend, RViz.

4.1 Visualize Path Planning in Rviz

As you've already seen in previous chapters, you can also launch RViz and add displays in order to watch the Path Planning process of the robot. For this chapter, you'll basically need to use 3 elements of RViz:

- Map Display (Costmaps)
- Path Displays (Plans)
- 2D Tools

- Exercise 4.1 -

a) Execute the next command in order to launch the move_base node.

▶ Execute in Shell #1

In [ ]:
```
roslaunch husky_navigation move_base_demo.launch
```

b) Open the Graphic Interface and execute the following command in order to start RViz.

▶ Execute in Shell #2

In [ ]:
```
rosrun rviz rviz
```

c) Properly congifure RViz in order to visualize the necessary parts.

Visualize Costmaps

- Click the Add button under Displays and chose the Map element.
- Set the topic to **/move_base/global_costmap/costmap** in order to visualize the global costmap
- Change the topic to **/move_base/local_costmap/costmap** in order to visualize the local costmap.
- You can have 2 Map displays, one for each costmap.

Visualize Plans

- Click the Add button under Displays and chose the Path element.
- Set the topic to **/move_base/NavfnROS/plan** in order to visualize the global plan.
- Change the topic to **/move_base/DWAPlannerROS/local_plan** in order to visualize the local plan.
- You can also have 2 Path displays, one for each plan.

d) Use the 2D Pose Estimate tool in order to provide an initial pose for the robot.

e) Use the 2D Nav Goal tool in order to send a goal pose to the robot.

- End of Exercise 4.1 -

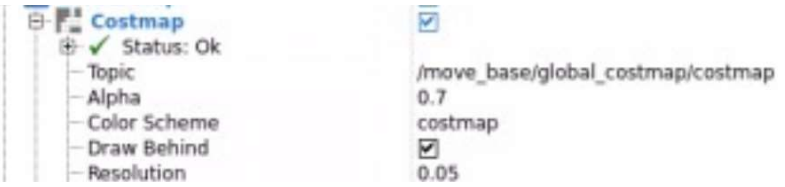
- Notes for Exercise 4.1 -

Check the following notes in order to complete the exercise:

Note 1: Bear in mind that if you don't set a 2D Nav Goal, the planning process won't start. This means that until you do, you won't be able to visualize any plan in RViz.

Note 2: In order for the 2D tools to work, the Fixed Frame at Rviz must be set to map.

Note 3: You can change the color scheme used for showing the costmaps at the *Color Scheme* option in the RViz configuration:

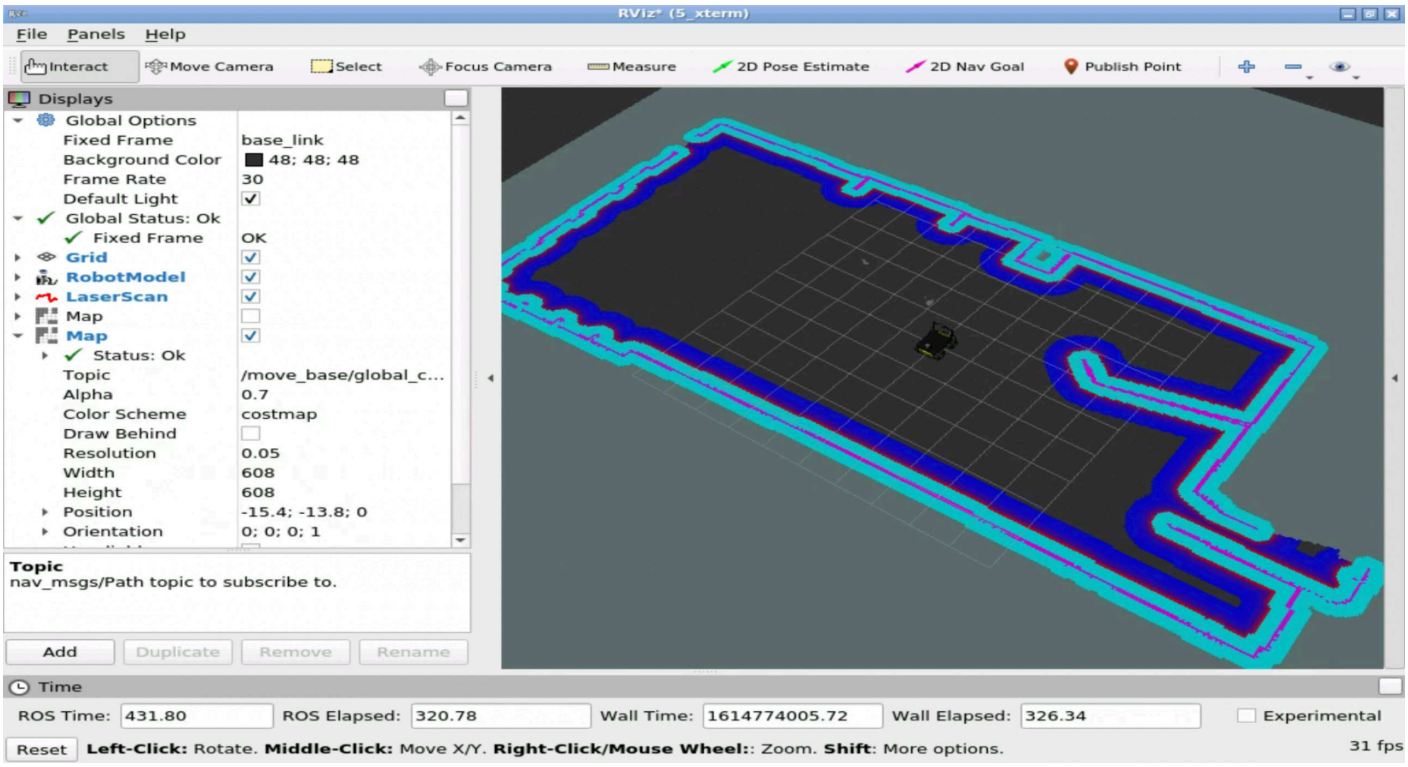


Note 4: Also, you can change the color used for showing the path at the *Color* option in the RViz configuration.

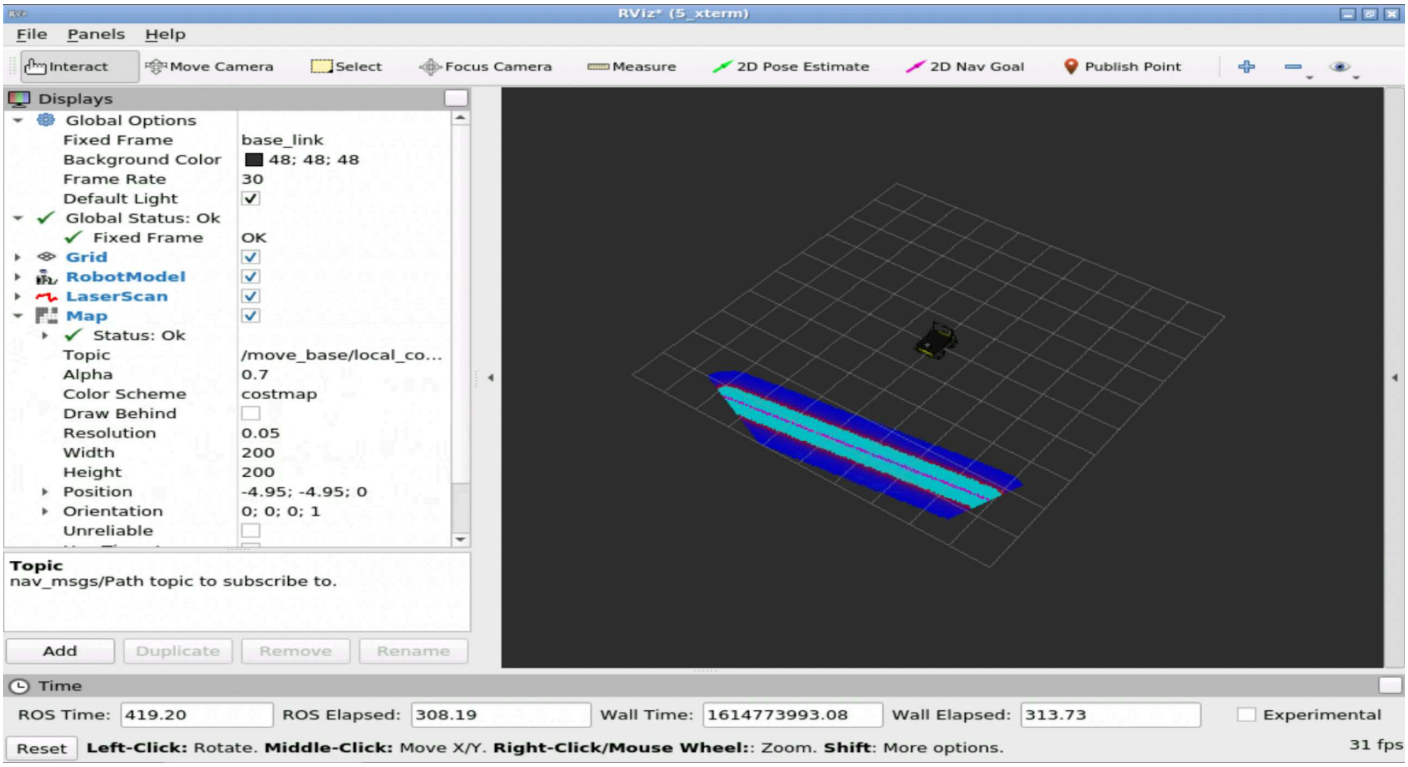
- End of Notes -

- Expected Result for Exercise 4.1 -

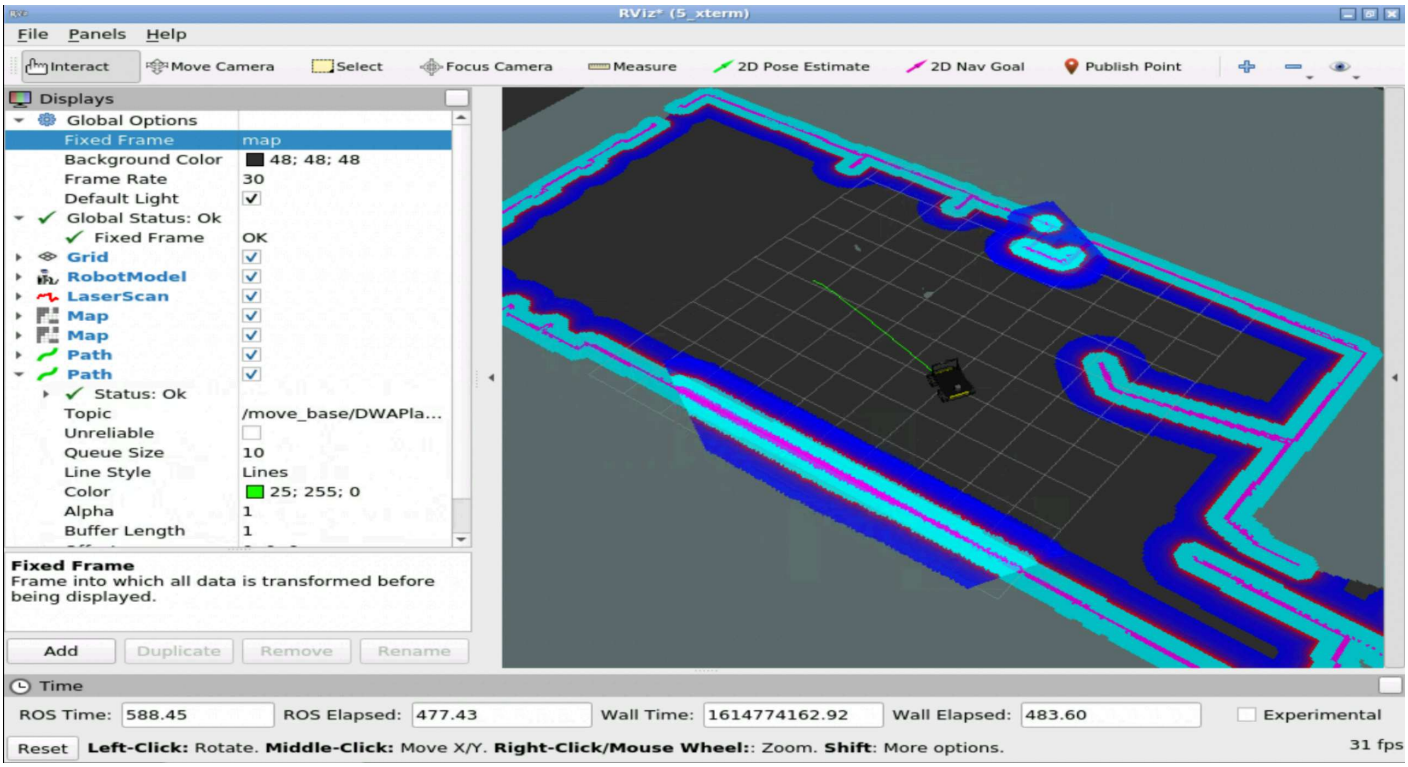
Global Costmap:



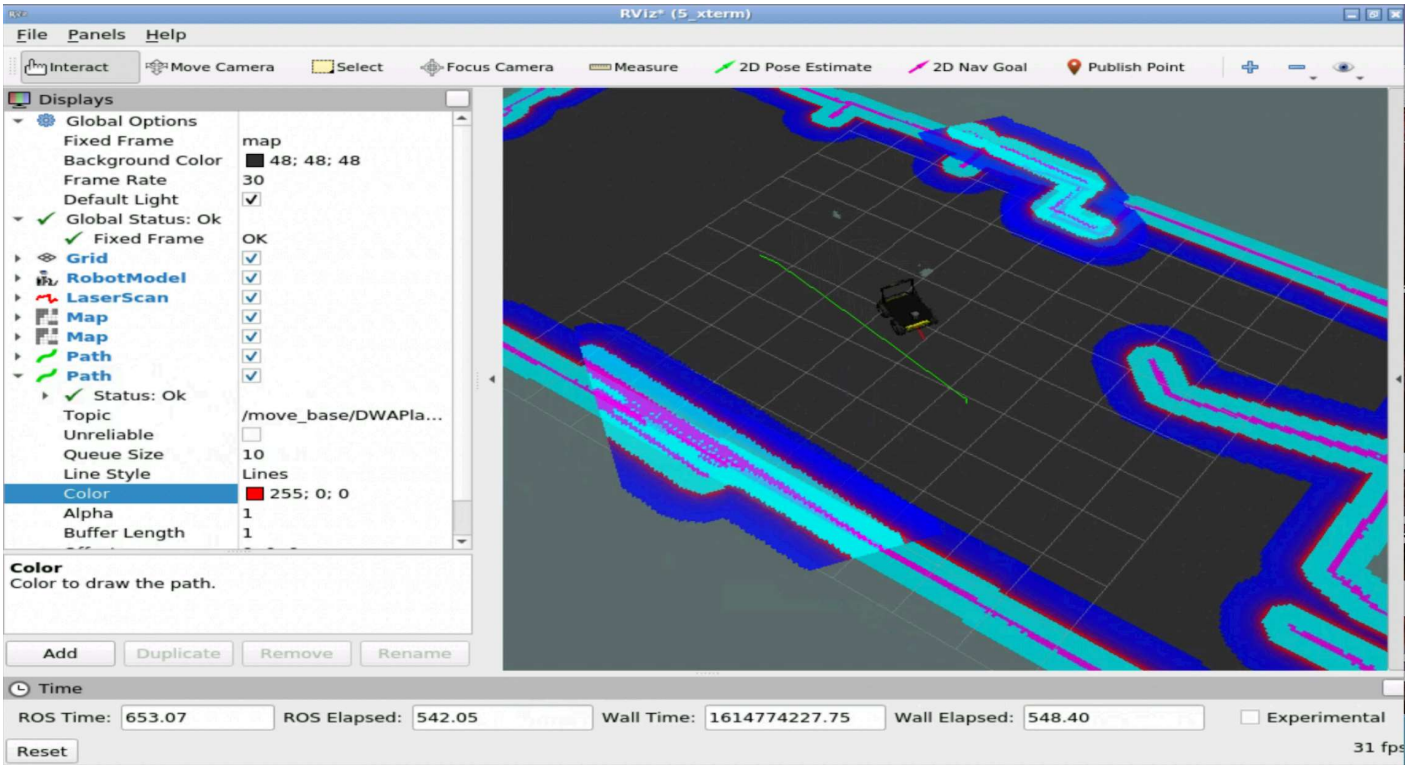
Local Costmap:



Global Plan:



Local Plan (in red) and Global Plan (in green):



- End of Expected Result -

REMEMBER: Remember to save your RViz configuration in order to be able to load it again whenever you want. If you don't remember how to do it, check the Mapping Chapter.

That's awesome, right? But what has just happened? What was that 2D Nav Goal tool I used in order to move the robot? And what's a Costmap? How does ROS calculate the trajectories?

Keep calm!! By the end of this chapter, you'll be able to answer all of those questions. But let's go step by step, so that you can completely understand how the whole process works.

4.2 The move_base package

The move_base package contains the **move_base node**. Doesn't that sound familiar? Well, it should, since you were introduced to it in the Basic Concepts chapter! The move_base node is one of the major elements in the ROS Navigation Stack, since it links all of the elements that take place in the Navigation process. Let's say it's like the Architect in Matrix, or the Force in Star Wars. Without this node, the ROS Navigation Stack wouldn't make any sense!

Ok! We understand that the move_base node is very important, but... what is it exactly? What does it do? Great question!

The **main function of the move_base node is to move the robot from its current position to a goal position**. Basically, this node is an implementation of a *SimpleActionServer*, which takes a goal pose with message type *geometry_msgs/PoseStamped*. Therefore, we can send position goals to this node by using a *SimpleActionClient*.

This Action Server provides the topic **move_base/goal**, which is the input of the Navigation Stack. This topic is then used to provide the goal pose.

- Exercise 4.2 -

a) In a WebShell, visualize the *move_base/goal* topic.

b) As you did in the previous exercise, send a goal to the robot by using the 2D Nav Goal tool in RViz.

c) Check what happens in the topic that you are listening to.

- End of Exercise 4.2 -

- Expected Result for exercise 4.2 -

```
ubuntu@ip-172-31-44-229:~$ rostopic pub /move_base/goal move_base_msgs/MoveBaseActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
    secs: 0
    nsecs: 0
  id: ''
goal:
  target_pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: 'map'
    pose:
      position:
        x: 1.16
        y: -4.76
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.75
        w: 0.66"
publishing and latching message. Press ctrl-C to terminate
```

- End of Expected Result -

So, each time you set a Pose Goal using the 2D Nav Goal tool from RViz, what is really happening is that a new message is being published into the move_base/goal topic.

Anyway, this is not the only topic that the move_base Action Server provides. As every action server, it provides the following 5 topics:

- **move_base/goal (move_base_msgs/MoveBaseActionGoal)**
- **move_base/cancel (actionlib_msgs/GoalID)**
- **move_base/feedback (move_base_msgs/MoveBaseActionFeedback)**
- **move_base/status (actionlib_msgs/GoalStatusArray)**
- **move_base/result (move_base_msgs/MoveBaseActionResult)**

- Exercise 4.3 -

Without using Rviz, send a pose goal to the move_base node.

a) Use the command line tool in order to send this goal to the Action Server of the move_base node.

b) Visualize through the webshells all of the topics involved in the action, and check their output while the action is taking place, and when it's done.

- End of Exercise 4.3 -

- Notes for Exercise 4.3 -

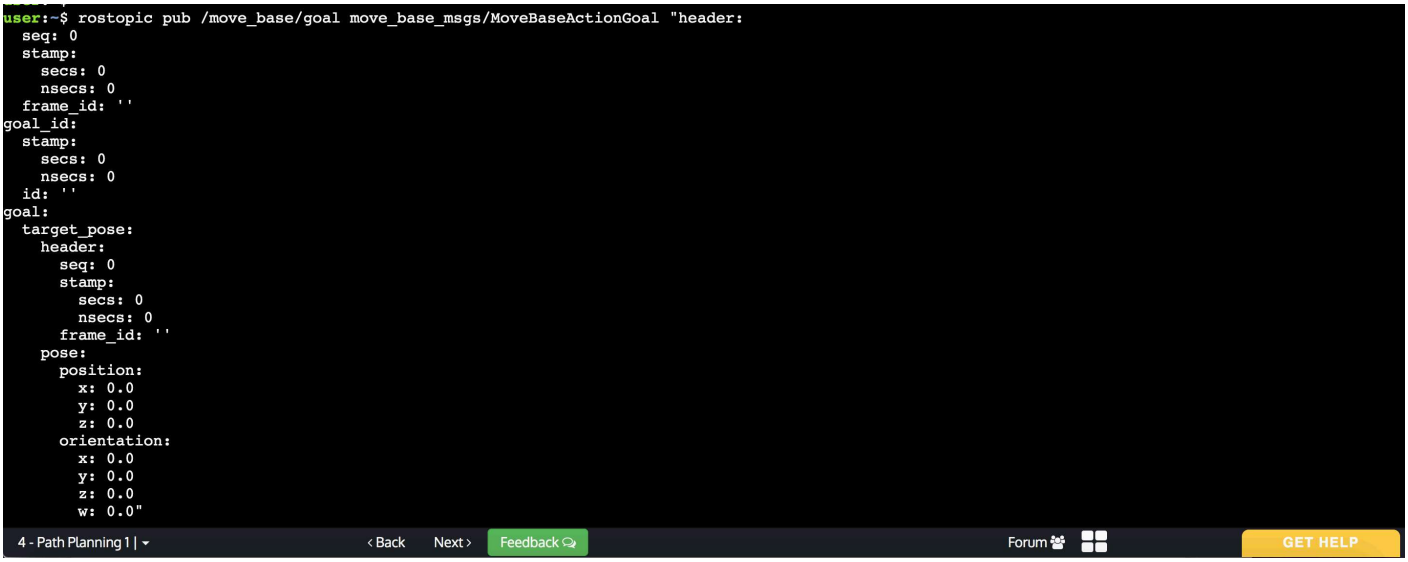
Check the following notes in order to complete the exercise:

Note 1: Remember that the SimpleAcionServer subscribes to the */move_base/goal* topic in order to read the pose goal.

Note 2: In order to see an example of a valid message for the */move_base/goal* topic, you can listen to the topic while you send a pose goal via the 2D Nav Goal tool of RViz.

Note 3: Keep in mind that in order to be able to send goals to the Action Server, the move_base node must be launched.

Note 4: In order to be able to properly fill the message for the /move_base/goal topic, you will need to first maximize the Web Shell. Otherwise, the message will collapse and it will be impossible to modify. Check out the image below:



- End of Notes -

- Expected Result for Exercise 4.3 -

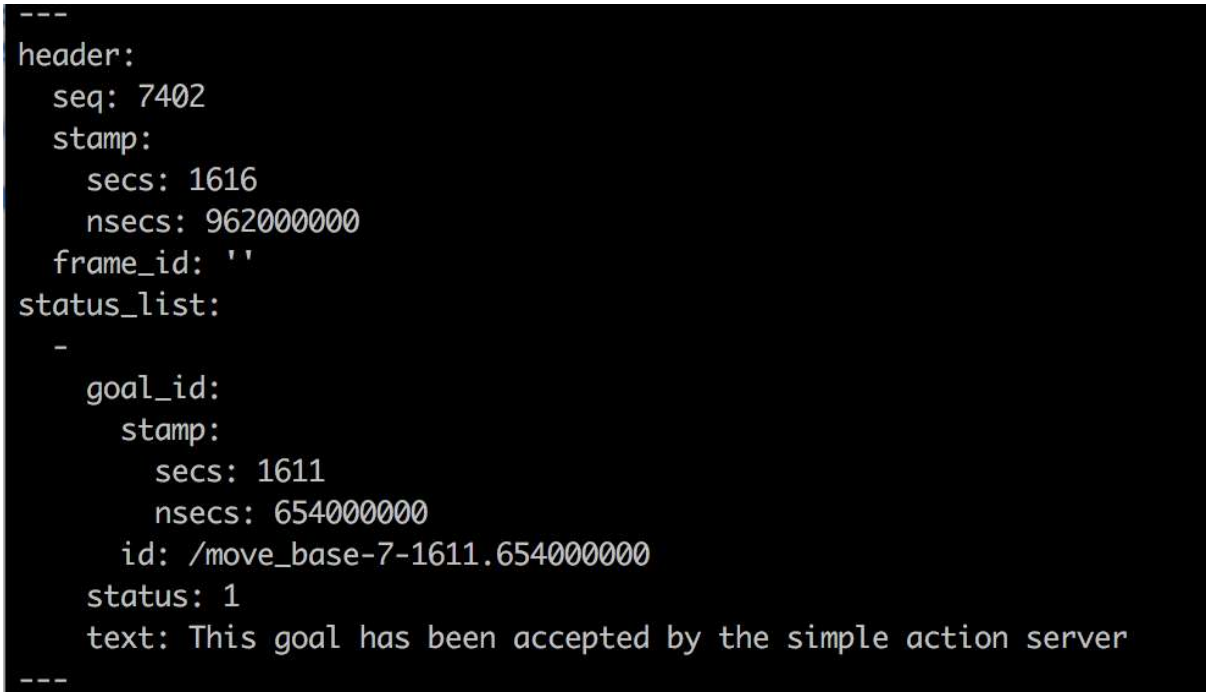
Sending goal:

```
ubuntu@ip-172-31-44-229:~$ rostopic pub /move_base/goal move_base_msgs/MoveBaseActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
    secs: 0
    nsecs: 0
  id: ''
goal:
  target_pose:
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: 'map'
    pose:
      position:
        x: 1.16
        y: -4.76
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.75
        w: 0.66"
publishing and latching message. Press ctrl-C to terminate
```

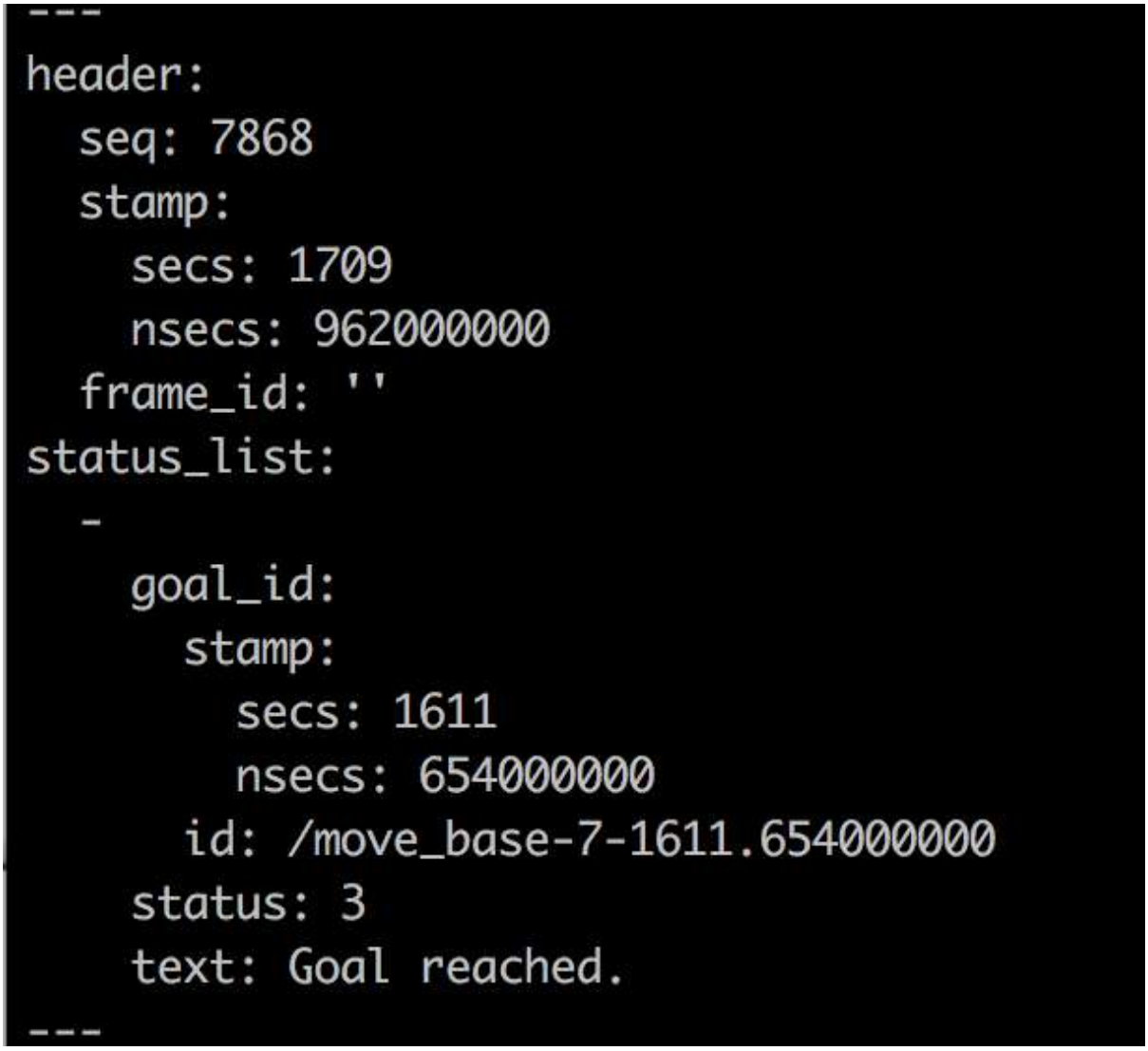
Echo feedback:

```
^Cubuntu@ip-172-31-44-229:~$ rostopic echo /move_base/feedback
WARNING: no messages received and simulated time is active.
Is /clock being published?
header:
  seq: 388
  stamp:
    secs: 1611
    nsecs: 654000000
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1611
      nsecs: 654000000
    id: /move_base-7-1611.654000000
  status: 1
  text: This goal has been accepted by the simple action server
feedback:
  base_position:
    header:
      seq: 0
      stamp:
        secs: 1611
        nsecs: 640000000
      frame_id: map
    pose:
      position:
        x: 0.936578248096
        y: -4.23249236439
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.716234530584
        w: 0.697859654372
---
```

Echo status accepted:



Echo status reached:


---
header:
seq: 7868
stamp:
secs: 1709
nsecs: 962000000
frame_id: ''
status_list:
-
goal_id:
stamp:
secs: 1611
nsecs: 654000000
id: /move_base-7-1611.654000000
status: 3
text: Goal reached.

Echo result:

```
ubuntu@ip-172-31-44-229:~$ rostopic echo /move_base/result
WARNING: no messages received and simulated time is active.
Is /clock being published?
header:
  seq: 6
  stamp:
    secs: 1625
    nsecs: 254000000
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1611
      nsecs: 654000000
    id: /move_base-7-1611.654000000
  status: 3
  text: Goal reached.
result:
```

- End of Expected Result -

- Exercise 4.4 -

a) Create a new package named **send_goals**. Add rospy as a dependency.

b) Inside this package, create a file named **send_goal_client.py**. Write into this file the code for an Action Client in order to send messages to the Action Server of the move_base node.

c) Using this Action Client, move the robot to three different Poses of the Map. When the robot has reached the 3 poses, start over again creating a loop, so that the robot will keep going to these 3 poses over and over.

- End of Exercise 4.4 -

- Expected Result for Exercise 4.4 -

The robot moves infinitely to the 3 poses given.

- End of Expected Result -

So, at this point, you've checked that you can send pose goals to the move_base node by sending messages to the /move_base/goal topic of its Action Server.

When this node **receives a goal pose**, it links to components such as the *global planner, local planner, recovery behaviors, and costmaps*, and **generates an output, which is a velocity command** with the message type *geometry_msgs/Twist*, and sends it to the **/cmd_vel** topic in order to move the robot.

The move_base node, just as you saw with the slam_gmapping and the amcl nodes in previous chapters, also has parameters that you can modify. For instance, one of the parameters that you can modify is the frequency at which the move_base node sends these velocity commands to the base controller. Let's check it with a quick exercise.

- Exercise 4.5 -

a) Create a new package named ***my_move_base_launcher***. Inside this package, create 2 directories, one named ***launch*** and the other one named ***params***. Inside the *launch* directory, create a new file named ***my_move_base.launch***. Inside the *params* directory, create a new file named ***my_move_base_params.yaml***.

b) Have a look at the ***move_base_demo.launch*** and ***move_base.launch*** files of the *husky_navigation* package. Check their structure and see how they work.

c) Copy the contents of the ***move_base.launch*** file to your file ***my_move_base.launch***.

d) Modify the ***my_move_base.launch*** file so that it also loads the ***map_server*** and ***amcl*** nodes. Pay attention to how it is done in the ***move_base_demo.launch*** file.

e) Now, have a look at the ***planner.yaml*** file of the *husky_navigation* package.

f) Copy the contents of the ***planner.yaml*** file to your file ***my_move_base_params.yaml***.

g) Modify the ***my_move_base.launch*** file so that it loads your parameters file **my_move_base_params.yaml**.

i) Launch the ***my_move_base.launch*** file, and send a goal to your robot!

- End of Exercise 4.5 -

At this point, all you know is that sending a pose goal to the move_base node activates some kind of process, which involves other nodes, and that results in the robot moving to that goal pose. That's very interesting, but... what is this process that is going on? How does it work? What are these other nodes that take place?

Dont't worry, you'll be able to answer all of those questions by the end of this chapter. But for now, let's start by introducing one of the main parts that take place in this process: the **global planner**.

4.3 Global Planner

When a new goal is received by the move_base node, this goal is immediately sent to the global planner. Then, the **global planner is in charge of calculating a safe path in order to arrive at that goal pose**. This path is calculated before the robot starts moving, so it will **not take into account the readings that the robot sensors are doing** while moving.

Each time a new path is planned by the global planner, this path is published into the **/plan** topic. Let's do an exercise to check this.

- Exercise 4.6 -

a) Open Rviz and add a Display in order to be able to visualize the Global Plan.

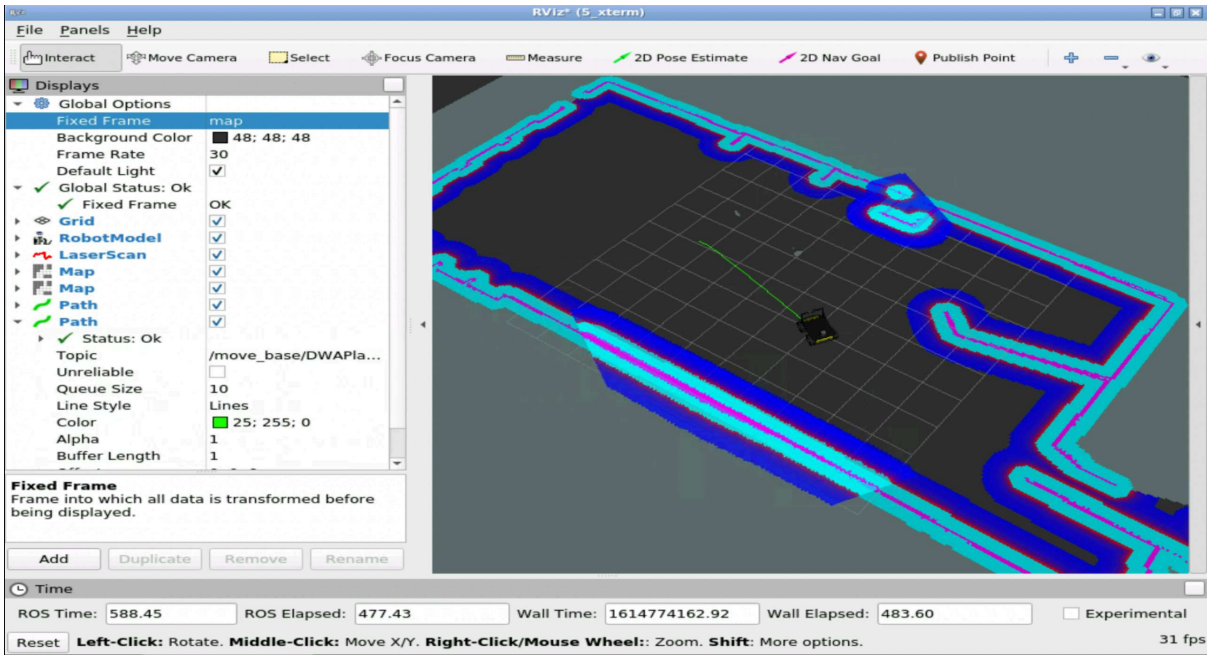
b) Subscribe to the topic where the Global Planner publishes its planned path, and have a look at it.

c) Using the 2D Nav Goal tool, send a new goal to the move_base node.

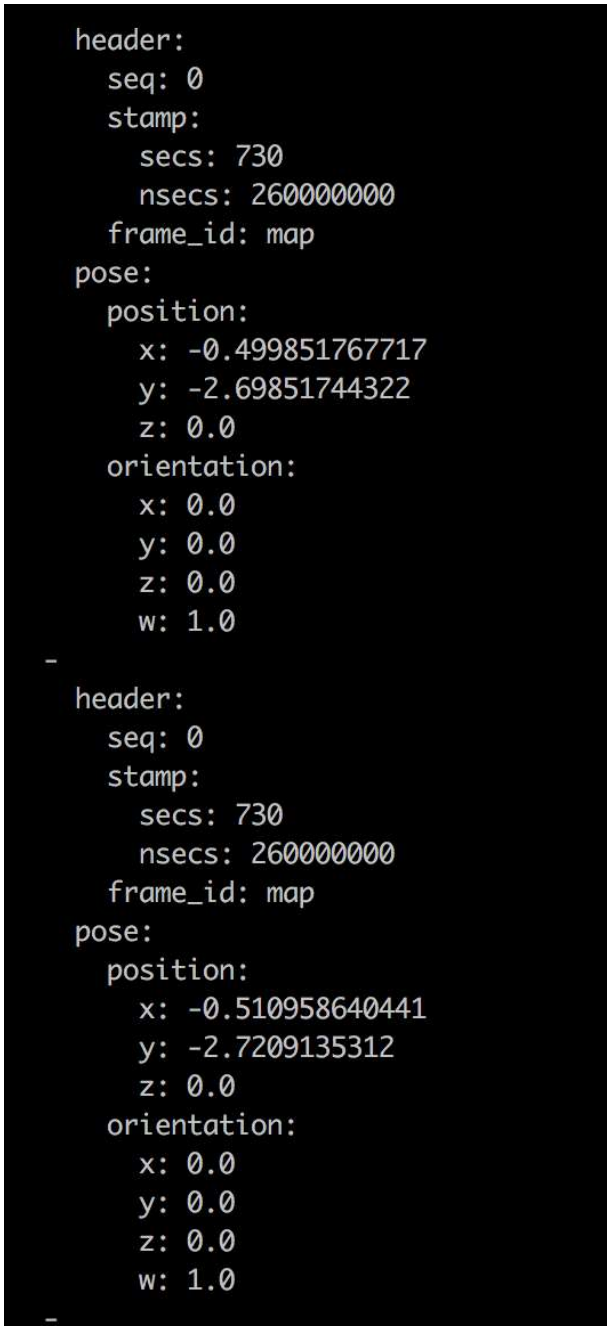
- End of Exercise 4.6 -

- Expected Result for Exercise 4.6 -

Global Plan in RViz:



Global Plan topic:



- End of Expected Result -

You've probably noticed that when you send a goal in order to visualize the path plan made by the global planner, the robot automatically starts executing this plan. This happens because by sending this goal pose, you're starting the whole navigation process.

In some cases, you might be interested in just visualizing the global plan, but not in executing that plan. For this case, the move_base node provides a service named **/make_plan**. This service allows you to calculate a global plan without causing the robot to execute the path. Let's check how it works with the next exercise.

- Exercise 4.7 -

Create a Service Client that will call one of the services introduced above in order to get the plan to a given pose, without causing the robot to move.

a) Create a new package named **make_plan**. Add rospy as a dependency.

b) Inside this package, create a file named **make_plan_caller.py**. Write the code for your Service Client into this file.

- End of Exercise 4.7 -

- Notes for Exercise 4.7 -

Check the following notes in order to complete the exercise:

Note 1: The type of message used by the /make_plan service is nav_msgs/GetPlan.

Note 2: When filing this message in order to call the service, you don't have to fill all of the fields of the message. Check the following message example:

```
ubuntu@ip-172-31-44-229:/opt/ros/indigo/share/husky_rviz_launchers/rviz$ rosservice call /move_base/make_plan "start:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: 'map'
  pose:
    position:
      x: 1.16
      y: -4.76
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.75
      w: 0.66
goal:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: 'map'
  pose:
    position:
      x: 1.16
      y: -4.50
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.75
      w: 0.66
tolerance: 0.0"
```

- End of Notes -

- Expected Result for Exercise 4.7 -

Returned Plan:

```
plan:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs:         0
    frame_id: ''
  poses:
    -
      header:
        seq: 0
        stamp:
          secs: 2151
          nsecs: 238000000
        frame_id: map
      pose:
        position:
          x: 1.15000024661
          y: -4.79999986589
          z: 0.0
        orientation:
          x: 0.0
          y: 0.0
          z: 0.0
          w: 1.0
    -
      header:
        seq: 0
        stamp:
          secs: 2151
          nsecs: 238000000
        frame_id: map
      pose:
        position:
          x: 1.15000024661
          y: -4.74999986514
          z: 0.0
        orientation:
          x: 0.0
```

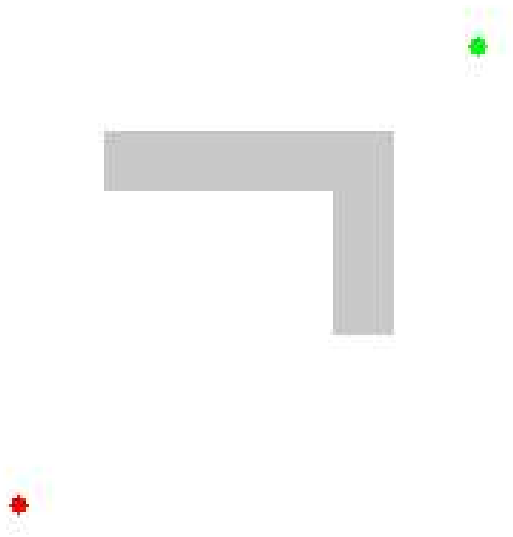
- End of Expected Result -

So, you now know that the first step of this navigation process is to calculate a safe plan so that your robot can arrive to the user-specified goal pose. But... how is this path calculated?

There exist different global planners. Depending on your setup (the robot you use, the environment it navigates, etc.), you would use one or another. Let's have a look at the most important ones.

4.3.1 Navfn

The Navfn planner is probably the most commonly used global planner for ROS Navigation. It uses Dijkstra's algorithm in order to calculate the shortest path between the initial pose and the goal pose. Below, you can see an animation of how this algorithm works.



4.3.2 CarrotPlanner

The carrot planner takes the goal pose and checks if this goal is in an obstacle. Then, if it is in an obstacle, it walks back along the vector between the goal and the robot until a goal point that is not in an obstacle is found. It, then, passes this goal point on as a plan to a local planner or controller. Therefore, this planner does not do any global path planning. It is helpful if you require your robot to move close to the given goal, even if the goal is unreachable. In complicated indoor environments, this planner is not very practical.

This algorithm can be useful if, for instance, you want your robot to move as close as possible to an obstacle (a table, for instance).

4.3.3 GlobalPlanner

The GlobalPlanner is a more flexible replacement for the Navfn planner we saw before. It allows you to change the algorithm used by Navfn (Dijkstra's algorithm) to calculate paths for other algorithms. These options include support for A$*$, toggling quadratic approximation, and toggling grid path.

4.4 Change the Global Planner

The global planner used by the move_base node it's specified in the **base_global_planner** parameter. It can be set in a parameter file, like in the example below:

In [ ]:
```
base_global_planner: "navfn/NavfnROS" # Sets the Navfn Planner

base_global_planner: "carrot_planner/CarrotPlanner" # Sets the CarrotPlanner

base_global_planner: "global_planner/GlobalPlanner" # Sets the GlobalPlanner
```

Or it can be set directly in the launch file, like it's done in our case:

In [ ]:
```
<arg name="base_global_planner" default="navfn/NavfnROS"/>
```

- Exercise 4.8 -

a) Open Rviz and add a display in order to be able to visualize the global plan.

b) Send a goal using the 2D Nav Goal tool. This goal must be "inside" an obstacle. Check what happens.

c) Modify the **my_move_base.launch** file so that it now uses the CarrotPlanner.

d) Repeat step b, and check what happens now.

- End of Exercise 4.8 -

- Notes for Exercise 4.8 -

Check the following notes in order to complete the exercise:

Note 1: To make sure you've properly changed the global planner, you can use the following command:

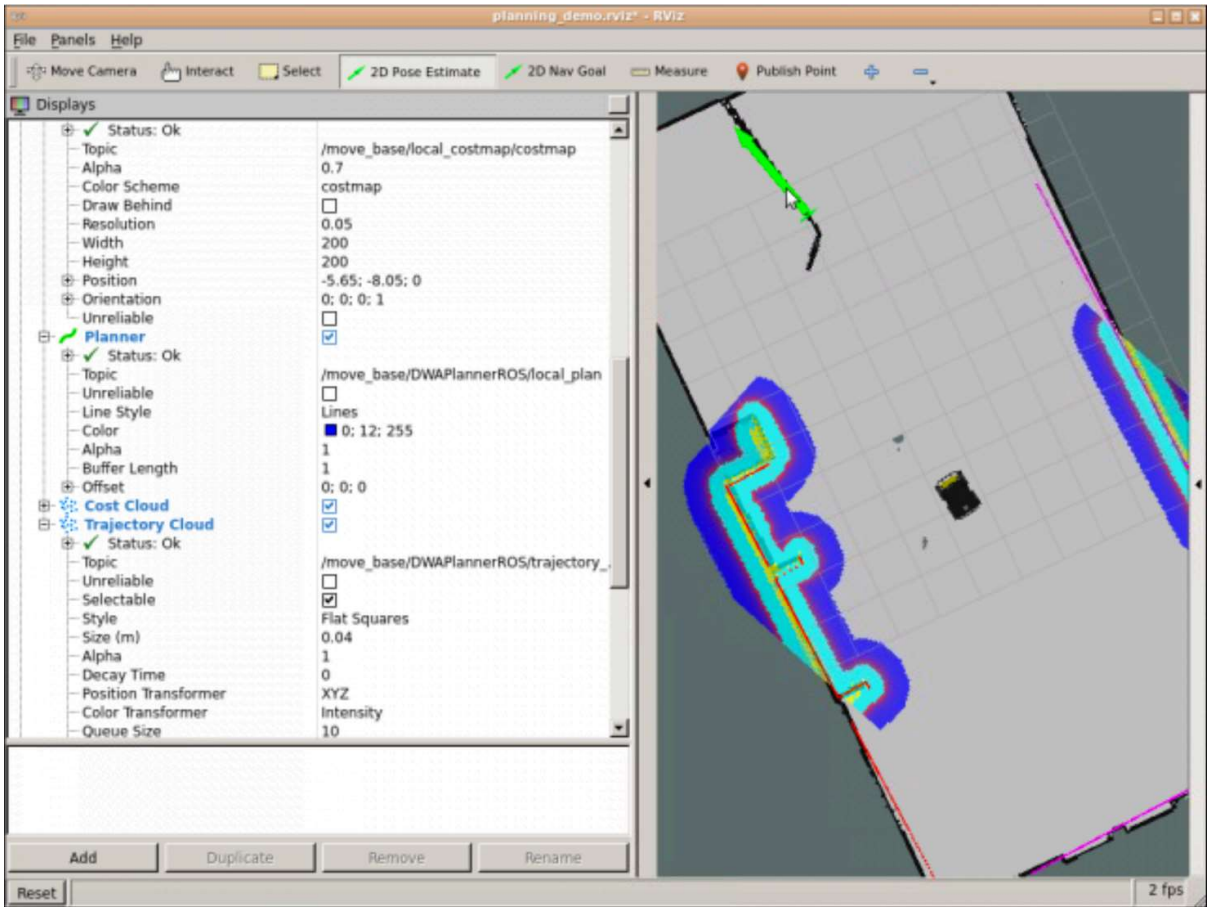
In [ ]:
```
rosparam get /move_base/base_global_planner
```

```
ubuntu@ip-172-31-44-26:~$ rosparam get /move_base/base_global_planner
carrot_planner/CarrotPlanner
```

- End of Notes -

- Expected Result for Exercise 4.8 -

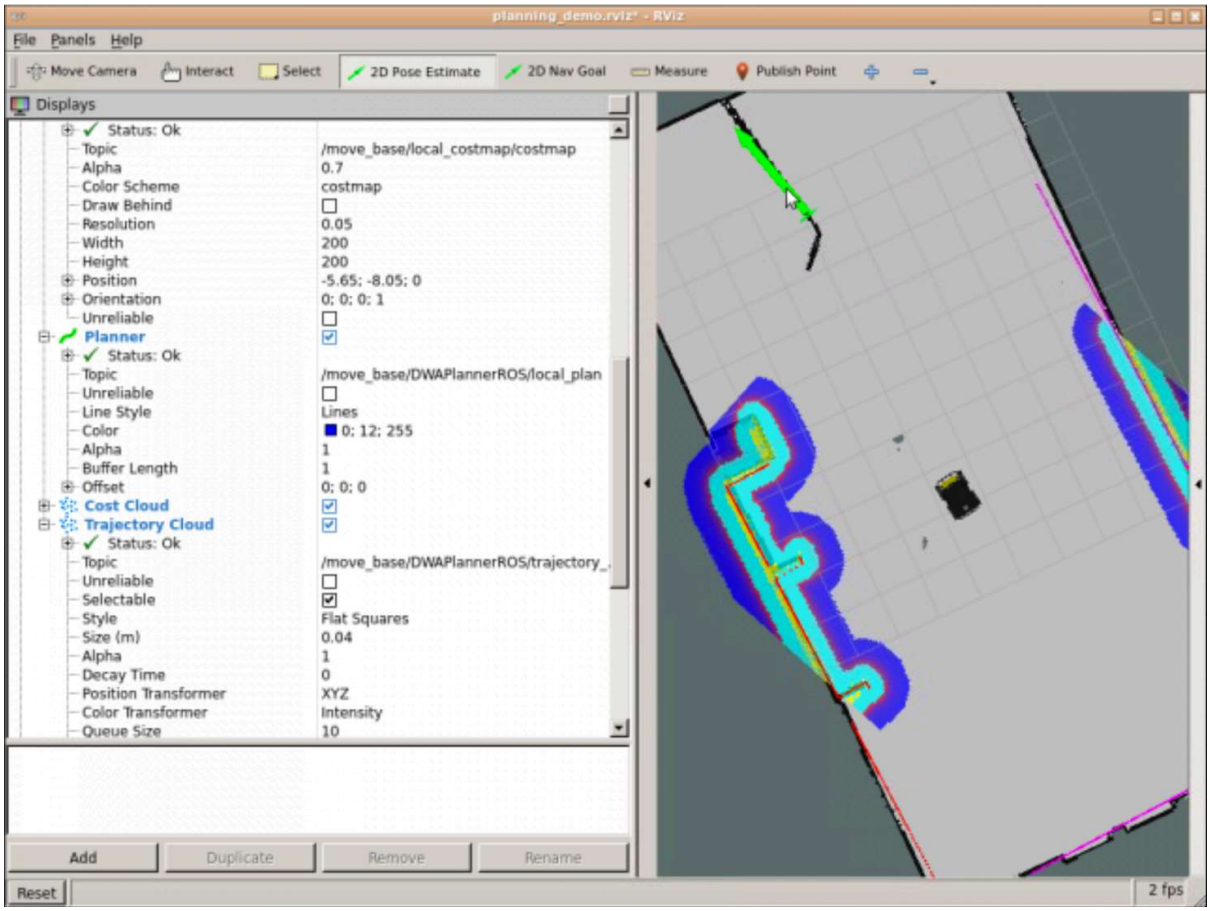
Sending goal "inside" an obstacle:



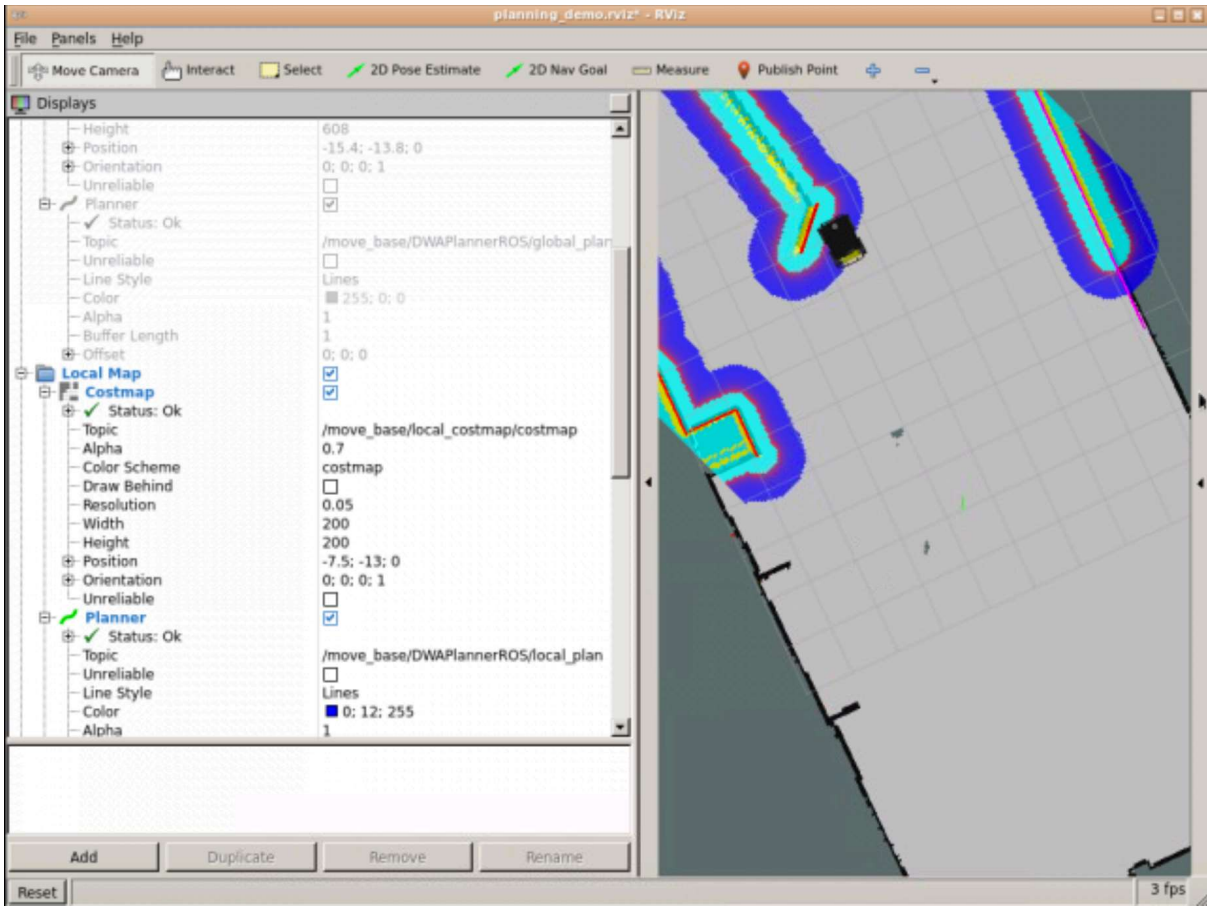
Navfn planner response:

```
[ WARN] [1485825088.248058940, 847.853000000]: Clearing costmap to unstuck robot (3.000000m).
[ WARN] [1485825089.177535176, 848.253000000]: Rotate recovery behavior started.
[ WARN] [1485825100.899294797, 853.953000000]: Clearing costmap to unstuck robot (1.840000m).
[ WARN] [1485825101.778941714, 854.353000000]: Rotate recovery behavior started.
[ERROR] [1485825112.392342099, 860.053000000]: Aborting because a valid plan could not be found. Even after executing all recovery behaviors
```

Sending goal "inside" an obstacle:



Carrot Planner Response:



- End of Expected Result -

Each planner also has its own parameters in order to customize its behaviour. These parameters are usually located in a YAML file. Depending on which global planner you use, the parameters to set will be different. In this course, we will have a look at the parameters for the Navfn planner because it's the one that is most commonly used. If you are interested in checking the parameters you can set for the other planners, you can have a look at them here:

carrot planner: http://wiki.ros.org/carrot_planner (http://wiki.ros.org/carrot_planner)

global planner: http://wiki.ros.org/global_planner (http://wiki.ros.org/global_planner)

4.4.1 Navfn Parameters

If you check your file **my_move_base_params.yaml** inside the package **my_move_base_launcher**, you will se the parameters seet for the Navfn planner:

In [ ]:

```
NavfnROS:
  allow_unknown: true # Specifies whether or not to allow navfn to create plan
  default_tolerance: 0.1 # A tolerance on the goal point for the planner.
```

- Notes -

Inside this file, you will also find parameters for other planners (*TrajectoryPlannerROS* and *DWAPlannerROS*). Don't worry about these now, since we will check them in the next Unit of the course.

- End of Notes -

Let's have a look at what are the most important parameters for the Navfn planner:

- **/allow_unknown (default: true)**: Specifies whether or not to allow navfn to create plans that traverse unknown space. NOTE: if you are using a layered costmap_2d costmap with a voxel or obstacle layer, you must also set the track_unknown_space param for that layer to be true, or it will convert all of your unknown space to free space (which navfn will then happily go right through).
- **/planner_window_x (default: 0.0)**: Specifies the x size of an optional window to restrict the planner to. This can be useful for restricting NavFn to work in a small window of a large costmap.
- **/planner_window_y (default: 0.0)**: Specifies the y size of an optional window to restrict the planner to. This can be useful for restricting NavFn to work in a small window of a large costmap.
- **/default_tolerance (default: 0.0)**: A tolerance on the goal point for the planner. NavFn will attempt to create a plan that is as close to the specified goal as possible, but no farther away than the default tolerance.
- **/visualize_potential (default: false)**: Specifies whether or not to visualize the potential area computed by navfn via a PointCloud2.

- Notes -

As you can see in our parameters file, not all the parameters are set. In this case, the planner will just take the default value for the parameter.

- End of Notes -

So... summarizing:

Until now, you've seen that a global planner exists that is in charge of calculating a safe path in order to move the robot from an initial position to a goal position. You've also seen that there are different types of global planners, and that you can choose the global planner that you want to use. Finally, you've also seen that each planner has its own parameters, which modify the way the planner behaves.

But now, let me ask you a question. When you plan a trajectory, this trajectory has to be planned according to a map, right? A path without a map makes no sense. Ok, so... can you guess what map the global planner uses in order to calculate its path?

You may be tempted to think that the map that is being used is the map that you created in the Mapping Chapter (Chapter 2) of this course... but, let me tell you, that's not entirely true.

There exists another type of map: **the costmap**. Does it sound familiar? It should since you were introduced to it back in the first exercise of this chapter.

4.5 Costmaps

A costmap is a map that represents places that are safe for the robot to be in a grid of cells. Usually, the values in the costmap are binary, representing either free space or places where the robot would be in collision.

Each cell in a costmap has an integer value in the range {0,255}. There are some special values frequently used in this range, which work as follows:

- **255 (NO_INFORMATION)**: Reserved for cells where not enough information is known.
- **254 (LETHAL_OBSTACLE**: Indicates that a collision-causing obstacle was sensed in this cell
- **253 (INSCRIBED_INFLATED_OBSTACLE)**: Indicates no obstacle, but moving the center of the robot to this location will result in a collision
- **0 (FREE_SPACE)**: Cells where there are no obstacles and, therefore, moving the center of the robot to this position will not result in a collision

There exist 2 types of costmaps: **global costmap** and **local costmap**. The main difference between them is, basically, the way they are built:

- The **global costmap** is created from a static map.
- The **local costmap** is created from the robot's sensor readings.

For now, we'll focus on the global costmap since it is the one used by the global planner. So, **the global planner uses the global costmap in order to calculate the path to follow**.

Let's do an exercise so that you can have a better idea of how a global costmap looks.

- Exercise 4.10 -

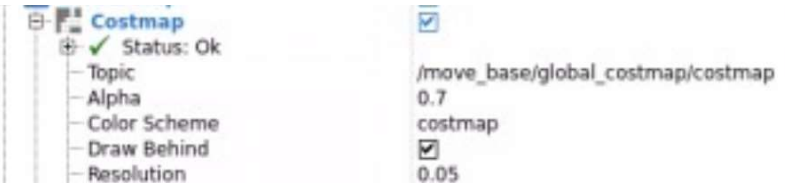
Launch Rviz and add the necessary display in order to visualize the global costmap.

- End of Exercise 4.10 -

- Notes for Exercise 4.10 -

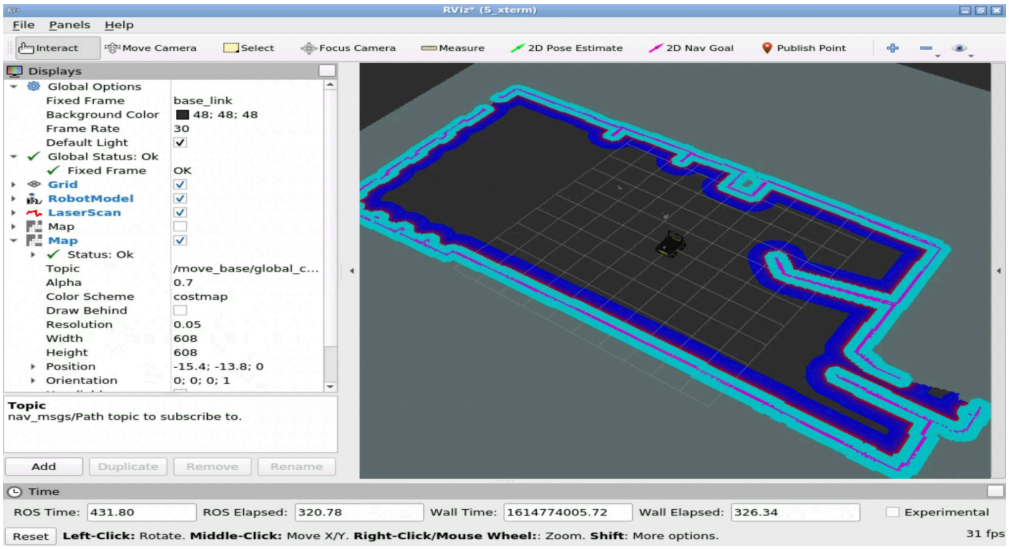
Check the following notes in order to complete the exercise:

Note 1: You can change the colours used for the global costmap at the Color Scheme parameter in the RViz configuration:



- End of Notes -

- Expected Result for Exercise 4.10 -



- End of Expected Result -

So, now that you've seen how the global costmap looks, let's learn some more about it.

4.5.1 Global Costmap

The global costmap is created from a user-generated static map (as the one we created in the Mapping Chapter). In this case, the costmap is initialized to match the width, height, and obstacle information provided by the static map. This configuration is normally used in conjunction with a localization system, such as amcl. This is the method you'll use to initialize a **global costmap**.

The global costmap also has its own parameters, which are defined in a YAML file. Next, you can see an example of a global costmap parameters file.

In [ ]:
```
global_frame: map
rolling_window: false

plugins:
  - {name: static,                  type: "costmap_2d::StaticLayer"}
  - {name: inflation,               type: "costmap_2d::InflationLayer"}
```

- Notes -

An important thing to take into account is that, since the **ROS Noetic** distribution, frames cannot be specified with an slash **"/"** symbol before them.

In [ ]:
```
global_frame: /map # NOT VALID IN NOETIC
global_frame: map # VALID IN NOETIC
```

- End of Notes -

Costmap parameters are defined in 3 different files:

- A YAML file that sets the parameters for the global costmap (which is the one you've seen above). Let's name this file *global_costmap_params.yaml*.
- A YAML file that sets the parameters for the local costmap. Let's name this file *local_costmap_params.yaml*.
- A YAML file that sets the parameters for both the global and local costmaps. Let's name this file *common_costmap_params.yaml*.

Fow now, we'll focus on the global costmap parameters since it's the costmap that is used by the global planner.

4.5.2 Global Costmap Parameters

The parameters you need to know are the following:

- **global_frame (default: "map")**: The global frame for the costmap to operate in.
- **robot_base_frame (default: "base_link")**: The name of the frame for the base link of the robot.
- **rolling_window (default: false)**: Whether or not to use a rolling window version of the costmap.
- **plugins**: Sequence of plugin specifications, one per layer. Each specification is a dictionary with a **name** and **type** fields. The name is used to define the parameter namespace for the plugin. This name will then be defined in the **common_costmap_parameters.yaml** file, which you will see in the the next Unit. The type field actually defines the plugin (source code) that is going to be used.

Let's start by talking about the *rolling_window* parameter, which is one of the most important ones. So, by setting the *rolling_window* parameter to false, we will initialize the costmap by getting the data from a static map. This is the way you want to initialize a global costmap.

- Exercise 4.11 -

a) Add a file named ***my_global_costmap_params.yaml*** to the *params* directory of the package you created in Exercise 4.5.

b) Copy the contents of the ***costmap_global_static.yaml*** file of the *husky_navigation* package into this file.

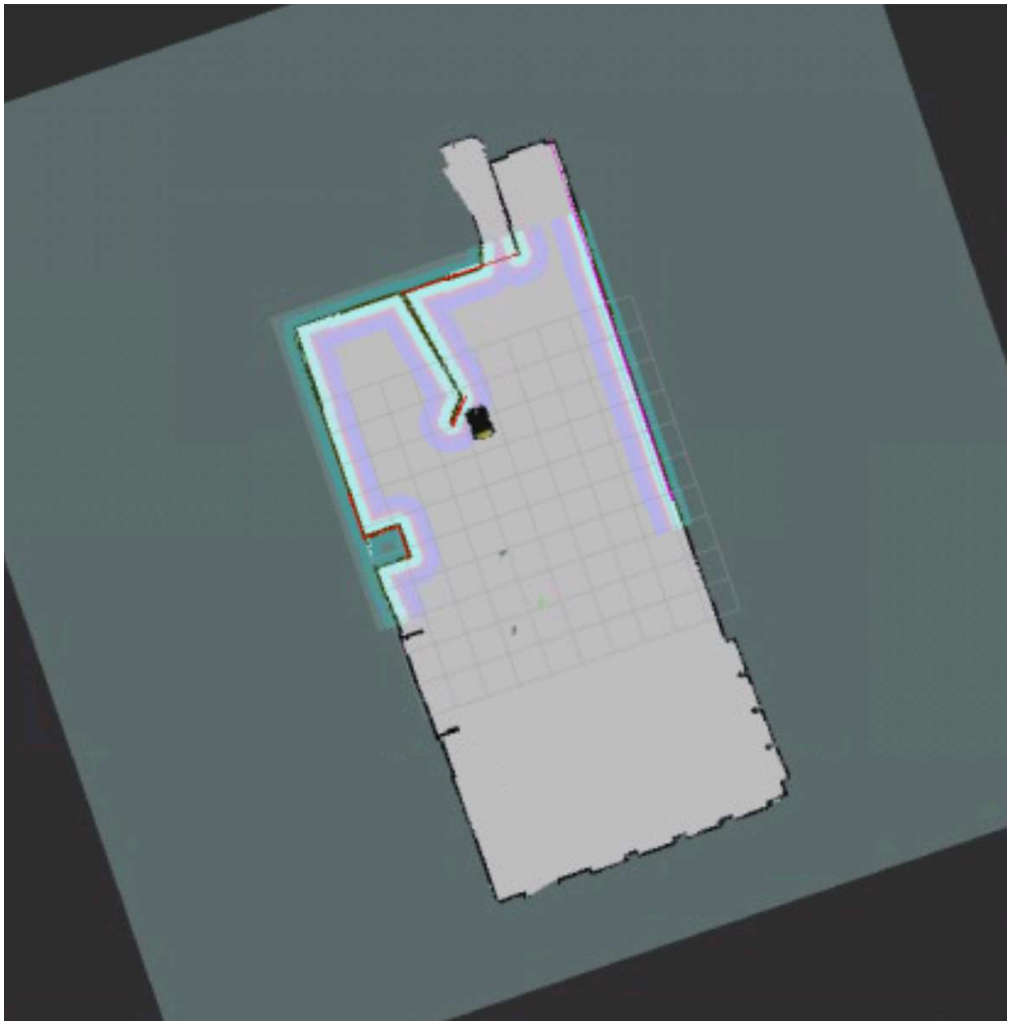
c) Modify the ***my_move_base.launch*** file you created in Exercise 4.5 so that it loads the global costmap parameters files you just created.

d) Change the ***rolling_window*** parameter to *true* and launch the move_base node again.

e) Check what changes you see in the visualization of the global costmap.

- End of Exercise 4.11 -

- Expected Result for exercise 4.11 -



- End of Expected Result -

The last parameter you need to know how to set is the plugins area. In the plugins area, we will add layers to the costmap configuration. Ok, but... what are layers?

In order to simplify (and clarify) the configuration of costmaps, ROS uses layers. Layers are like "blocks" of parameters that are related. For instance, the **static map, the sensed obstacles, and the inflation are separated into different layers**. These layers are defined in the ***common_costmap_parameters.yaml*** file, and then added to the ***local_costmap_params.yaml*** and ***global_costmap_params.yaml*** files.

To add a layer to a configuration file of a costmap, you will specify it in the plugins area. Have a look at the following line:

In [ ]:
```
plugins:
    - {name: static_map,       type: "costmap_2d::StaticLayer"}
```

Here, you're adding to your costmap configuration a layer named **static_map**, which will use the **costmap_2d::StaticLayer** plugin. You can add as many layers as you want:

In [ ]:
```
plugins:
    - {name: static_map,       type: "costmap_2d::StaticLayer"}
    - {name: inflation,        type: "costmap_2d::InflationLayer"}
```

For instance, you can see an example on the local costmap parameters file shown above. In the case of the global costmap, you will usually use these 2 layers:

- **costmap_2d::StaticLayer**: Used to initialize the costmap from a static map.
- **costmap_2d::InflationLayer**: Used to inflate obstacles.

You may have noticed that the layers are just being added to the parameters file. That's true. Both in the global and local costmap parameters file, the layers are just added. The specific parameters of these layers are defined in the **common costmap parameters** file. We will have a look at this file later on in the next Unit.

NEW IMPORTANT UPDATE FOR THIS COURSE !!!

(done on 23rd Feb 2021)

Starting from the 23rd of February, **the Unit 6 Project of this course is deprecated**. From that day on, the Unit 6 Project of this course must be done in a completely different way (see instructions below). The reason for this change is to include a more realistic project and to **provide you with practice with real robots**. Remember that everything is included in your subscription at no extra cost. **If you have already started the Unit 6 Project for this course, you can keep doing it and finish it**. You don't need to change to the new project. However, the Unit 6 Project will definitely be cancelled on March 31st, so you have to finish it before that date. If you haven't started the Unit 6 Project yet, then don't do it. Instead, go to the new project, the procedure for which is explained below. We are always thinking of ways to provide you with the best learning experience. We hope this change will help you better understand how to program robots with ROS.

Each course in this academy includes a project that must be solved applying the knowledge gained from the whole course.

It is now time that you start the project for this course!

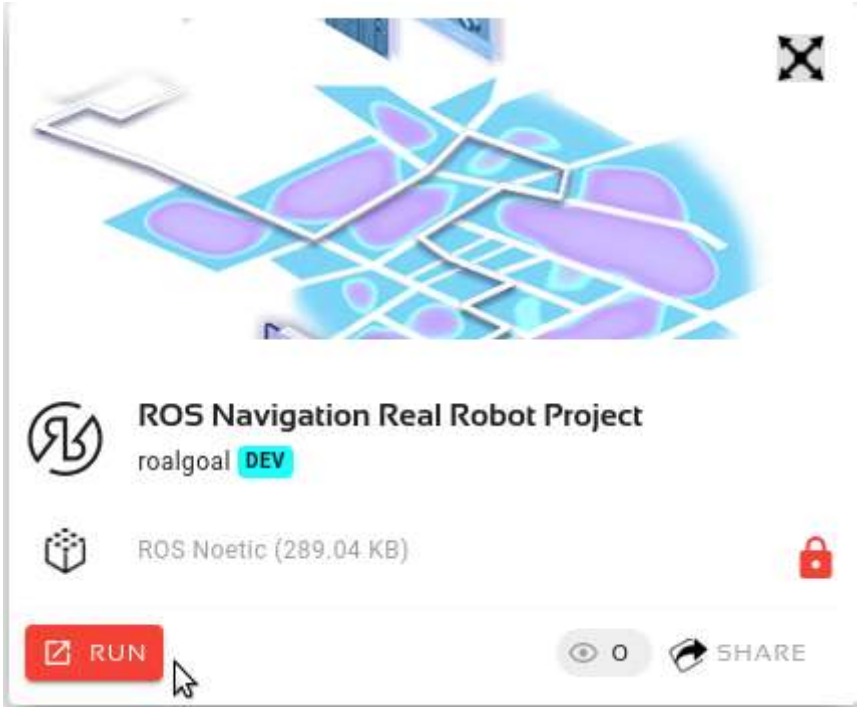
The project is going to be done in a different environment, which we call the **ROS Development Studio** (ROSDS). The ROSDS is an environment closer to what you will find when programming robots for companies. It is not as guided as this academy.

The ROSDS is included with your subscription, and is integrated inside The Construct. So no extra effort needs to be made by you. Well... you will need to expend some extra learning effort! But that's why you are here!

To start the project, you first need to get a copy of the ROS project (rosject), which contains the project instructions. Do the following:

1. **Copy the project rosject** to your ROSDS area (see instructions below).
2. Once you have it, go to the *My Rosjects* area in The Construct

1. **Open the rosject** by clicking *Run* on this course rosject



1. Then follow the instructions of the rosject to **finish the SECTION III and IV of the project**.

You can now copy the project rosject by clicking here (https://app.theconstructsim.com/#/l/3df8164e/). This will automatically make a copy of it.

You should finish SECTION III and IV of the rosject before attempting next unit of this course!

English proofread