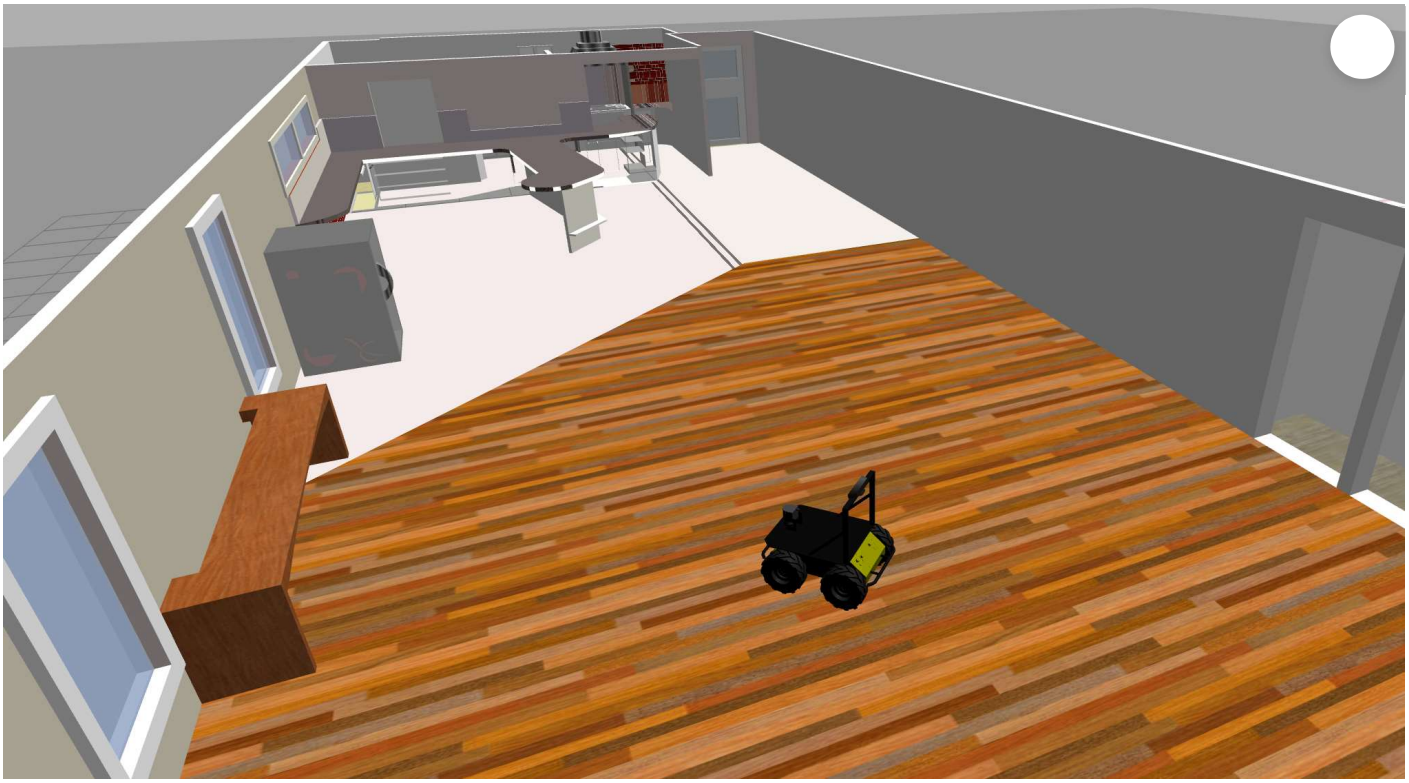
ROS Navigation in 5 Days

Unit 5: Path Planning Part 2 (Obstacle Avoidance)



- Summary -

Estimated time to completion: **3 hours**

What will you learn with this unit?

- How does Obstacle Avoidance work
- What is the Local Planner?
- What is the Local Costmap?
- Path Planning Recap
- How to use the Dynamic Reconfigure and other Rviz tools

- End of Summary -

Until this point, you've seen how ROS plans a trajectory in order to move a robot from a starting position to a goal position. In this chapter, you will learn how ROS executes this trajectory, and avoids obstacles while doing so. You will also learn other important concepts regarding Path Planning, which we missed in the previous chapter. Finally, we will do a summary so that you can better understand the whole process.

Are you up to the challenge? Let's get started then!

5.1 Local Planner

Once the global planner has calculated the path to follow, this path is sent to the local planner. The local planner, then, will execute each segment of the global plan (let's imagine the local plan as a smaller part of the global plan). So, **given a plan to follow (provided by the global planner) and a map, the local planner will provide velocity commands in order to move the robot**.

Unlike the global planner, the **local planner monitors the odometry and the laser data**, and chooses a collision-free local plan (let's imagine the local plan as a smaller part of the global plan) for the robot. So, the local planner **can recompute the robot's path on the fly** in order to keep the robot from striking objects, yet still allowing it to reach its destination.

Once the local plan is calculated, it is published into a topic named **/local_plan**. The local planner also publishes the portion of the global plan that it is attemting to follow into the topic **/global_plan**. Let's do an exercise so that you can see this better.

- Exercise 5.1 -

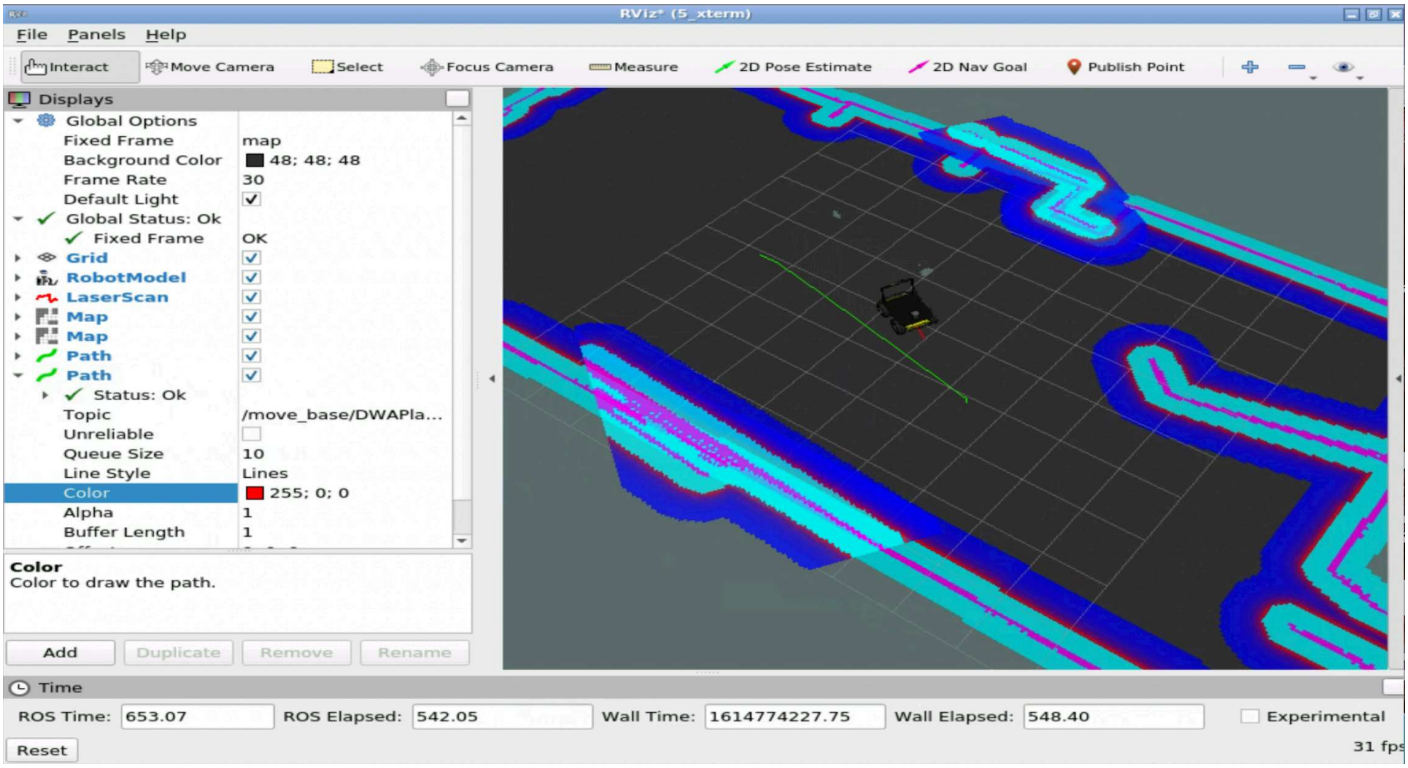
a) Open Rviz and add Displays in order to be able to visualize the **/global_plan** and the **/local_plan** topics of the local planner.

b) Send a Goal Pose to the robot and visualize both topics.

- End of Exercise 5.1 -

- Expected Result for Exercise 5.1 -

Global plan in green and Local plan in red:



- End of Expected Result -

- Notes for Exercise 5.1 -

Note 1: Remember you can change the color used for showing the path at the *Color* option in the RViz configuration.

- End of Notes -

As for the global planner, different types of local planners also exist. Depending on your setup (the robot you use, the environment it navigates, etc.) and the type of performance you want, you will use one or another. Let's have a look at the most important ones.

base_local_planner

The base local planner provides implementations of the *Trajectory Rollout* and the *Dynamic Window Approach (DWA)* algorithms in order to calculate and execute a global plan for the robot.

Summarizing, the basic idea of how this algorithms works is as follows:

- Discretely sample from the robot's control space
- For each sampled velocity, perform forward simulations from the robot's current state to predict what would happen if the sampled velocity was applied.
- Evaluate each trajectory resulting from the forward simulation.
- Discard illegal trajectories.
- Pick the highest-scoring trajectory and send the associated velocities to the mobile base.
- Rinse and Repeat.

DWA differs from Trajectory Rollout in how the robot's space is sampled. Trajectory Rollout samples are from the set of achievable velocities over the entire forward simulation period given the acceleration limits of the robot, while DWA samples are from the set of achievable velocities for just one simulation step given the acceleration limits of the robot.

DWA is a more efficient algorithm because it samples a smaller space, but may be outperformed by Trajectory Rollout for robots with low acceleration limits because DWA does not forward simulate constant accelerations. In practice, DWA and Trajectory Rollout perform similarly, so **it's recommended to use DWA because of its efficiency gains**.

The DWA algorithm of the base local planner has been improved in a new local planner separated from this one. That's the DWA local planner we'll see next.

dwa_local_planner

The DWA local planner provides an implementation of the *Dynamic Window Approach* algorithm. It is basically a re-write of the base local planner's DWA (Dynamic Window Approach) option, but the code is a lot cleaner and easier to understand, particularly in the way that the trajectories are simulated.

So, for applications that use the DWA approach for local planning, the dwa_local_planner is probaly the best choice. This is the **most commonly used option**.

eband_local_planner

The eband local planner implements the *Elastic Band* method in order to calculate the local plan to follow.

teb_local_planner

The teb local planner implements the *Timed Elastic Band* method in order to calculate the local plan to follow.

5.2 Change the Local Planner

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

In [ ]:
```
base_local_planner: "base_local_planner/TrajectoryPlannerROS" # Sets the Traje

base_local_planner: "dwa_local_planner/DWAPlannerROS" # Sets the dwa local pla

base_local_planner: "eband_local_planner/EBandPlannerROS" # Sets the eband loc

base_local_planner: "teb_local_planner/TebLocalPlannerROS" # Sets the teb loca
```

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

In [ ]:
```
<arg name="base_local_planner" default="dwa_local_planner/DWAPlannerROS"/>
```

- End of Exercise 5.2 -

Change the local planner in the ***my_move_base.launch*** file to use the teb_local_planner and check how it performs.

NOTE: Make sure to switch back to the DWAPlanner after you finish with this Exercise.

- End of Exercise 5.2 -

- Notes for Exercise 5.2 -

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_local_planner
```

```
ubuntu@ip-172-31-32-206:~$ rosparam get /move_base/base_local_planner
teb_local_planner/TebLocalPlannerROS
```

Important Note: Make sure to switch back to the *DWAPlannerROS* after you finish with the previous Exercise.

- End of Notes -

As you would expect, each local planner also has its own parameters. These parameters will be different depending on the local planner you use. In this course, we'll be focusing on the DWA local planner parameters, since it's the most common choice. Anyways, if you want to check the specific parameters for the other local planners, you can have a look at them here:

base_local_planner: http://wiki.ros.org/base_local_planner (http://wiki.ros.org/base_local_planner)

eband_local_planner: http://wiki.ros.org/eband_local_planner (http://wiki.ros.org/eband_local_planner)

teb_local_planner: http://wiki.ros.org/teb_local_planner (http://wiki.ros.org/teb_local_planner)

5.2.1 DWAPlannerROS 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 *DWAPlannerROS* planner:

In [ ]:
```
DWAPlannerROS:
  # Robot configuration parameters
  acc_lim_x: 2.5
  acc_lim_y: 0
  acc_lim_th: 3.2

  max_vel_x: 0.5
  min_vel_x: 0.0
  max_vel_y: 0
  min_vel_y: 0

  max_vel_trans: 0.5
  min_vel_trans: 0.1
  max_vel_theta: 1.0
  min_vel_theta: 0.2

  # Goal Tolerance Parameters
  yaw_goal_tolerance: 0.1
  xy_goal_tolerance: 0.2
  latch_xy_goal_tolerance: false
```

The most important parameters for the DWA local planner are the following:

Robot Configuration Parameters

- **/acc_lim_x (default: 2.5)**: The x acceleration limit of the robot in meters/sec^2
- **/acc_lim_th (default: 3.2)**: The rotational acceleration limit of the robot in radians/sec^2
- **/max_vel_trans (default: 0.55)**: The absolute value of the maximum translational velocity for the robot in m/s
- **/min_vel_trans (default: 0.1)**: The absolute value of the minimum translational velocity for the robot in m/s
- **/max_vel_x (default: 0.55)**: The maximum x velocity for the robot in m/s.
- **/min_vel_x (default: 0.0)**: The minimum x velocity for the robot in m/s, negative for backwards motion.
- **/max_vel_theta (default: 1.0)**: The absolute value of the maximum rotational velocity for the robot in rad/s
- **/min_vel_theta (default: 0.4)**: The absolute value of the minimum rotational velocity for the robot in rad/s

Goal Tolerance Parameters

- **/yaw_goal_tolerance (double, default: 0.05)**: The tolerance, in radians, for the controller in yaw/rotation when achieving its goal
- **/xy_goal_tolerance (double, default: 0.10)**: The tolerance, in meters, for the controller in the x and y distance when achieving a goal
- **/latch_xy_goal_tolerance (bool, default: false)**: If goal tolerance is latched, if the robot ever reaches the goal xy location, it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so.

- Exercise 5.3 -

a) Open the **my_move_base_params.yaml** file you created in the previous Chapter to edit it.

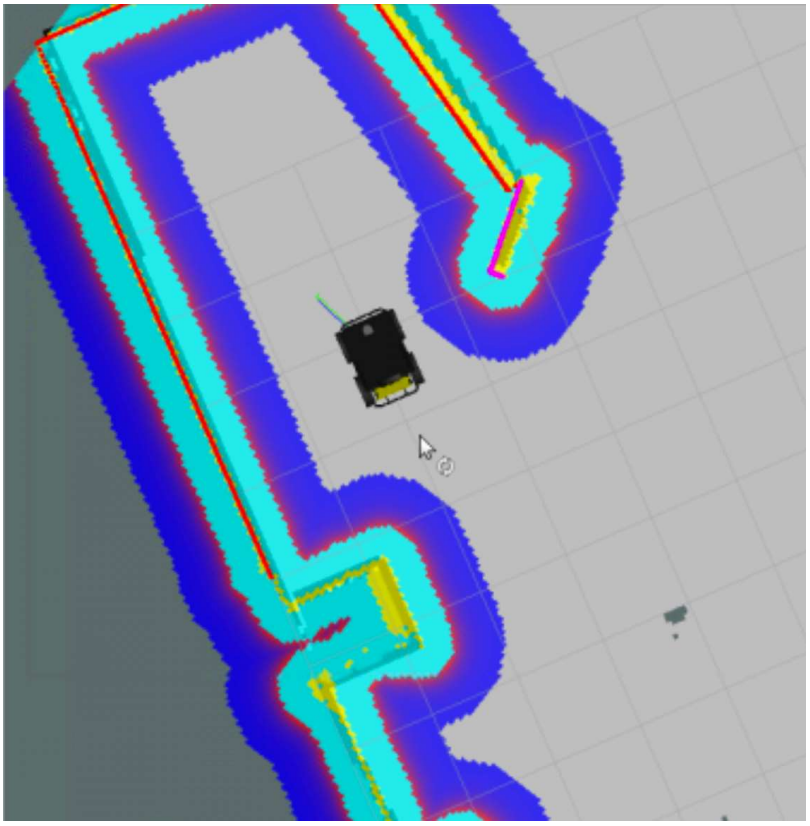
b) Modify the ***xy_goal_tolerance*** parameter of the DWAPlannerROS and set it to a higher value.

c) Check if you notice any differences in the performance.

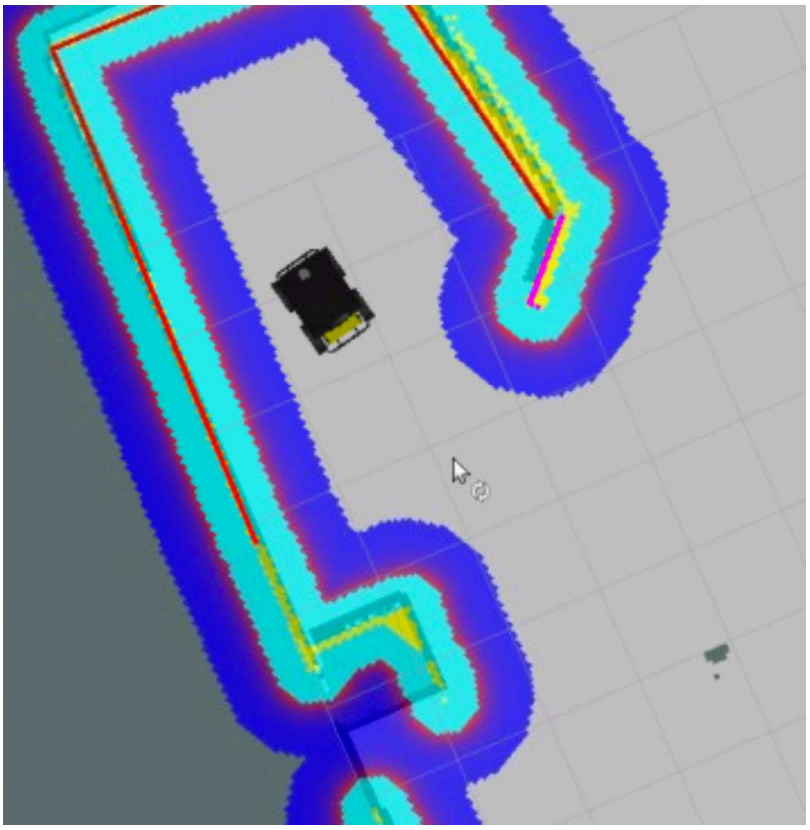
- End of Exercise 5.3 -

- Expected Result for Exercise 5.3 -

High XY tolerance:



Low XY tolerance:



- End of Expected Result -

As you've seen in the exercise, the higher you set the goal tolerances in your parameters file, the less accurate the robot will be in order to set a goal as reached.

As you can see in your parameters file, there are also other parameters which are commented (hence the planner is taking their default values):

In [ ]:
```
# # Forward Simulation Parameters
  # sim_time: 2.0
  # sim_granularity: 0.02
  # vx_samples: 6
  # vy_samples: 0
  # vtheta_samples: 20
  # penalize_negative_x: true

  # # Trajectory scoring parameters
  # path_distance_bias: 32.0 # The weighting for how much the controller shoul
  # goal_distance_bias: 24.0 # The weighting for how much the controller shoul
  # occdist_scale: 0.01 # The weighting for how much the controller should att
  # forward_point_distance: 0.325 # The distance from the center point of the
  # stop_time_buffer: 0.2  # The amount of time that the robot must stThe absc
  # scaling_speed: 0.25 # The absolute value of the veolicty at which to start
  # max_scaling_factor: 0.2 # The maximum factor to scale the robot's footprin

  # # Oscillation Prevention Parameters
  # oscillation_reset_dist: 0.25 #How far the robot must travel in meters befc
```

Let's also have a look at them:

Forward Simulation Parameters

- **/sim_time (default: 1.7)**: The amount of time to forward-simulate trajectories in seconds
- **/sim_granularity (default: 0.025)**: The step size, in meters, to take between points on a given trajectory
- **/vx_samples (default: 3)**: The number of samples to use when exploring the x velocity space
- **/vy_samples (default: 10)**: The number of samples to use when exploring the y velocity space
- **/vtheta_samples (default: 20)**: The number of samples to use when exploring the theta velocity space

- Exercise 5.4 -

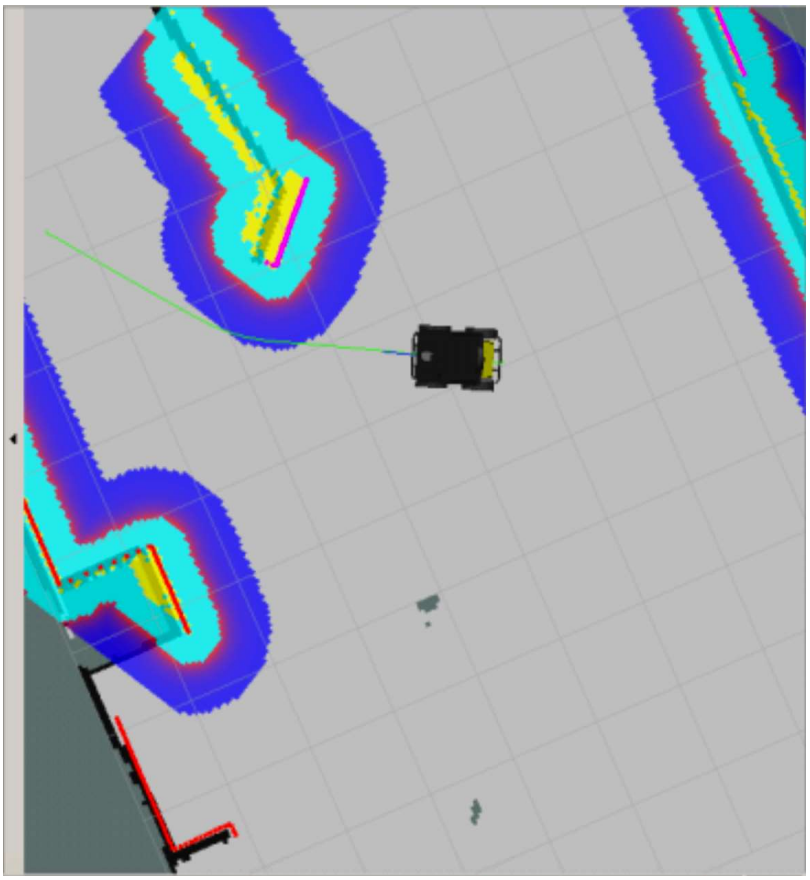
a) Modify the ***sim_time*** parameter in the local planner parameters file and set it to 4.0.

b) Check if you notice any differences in the performance or visualization of the local planner.

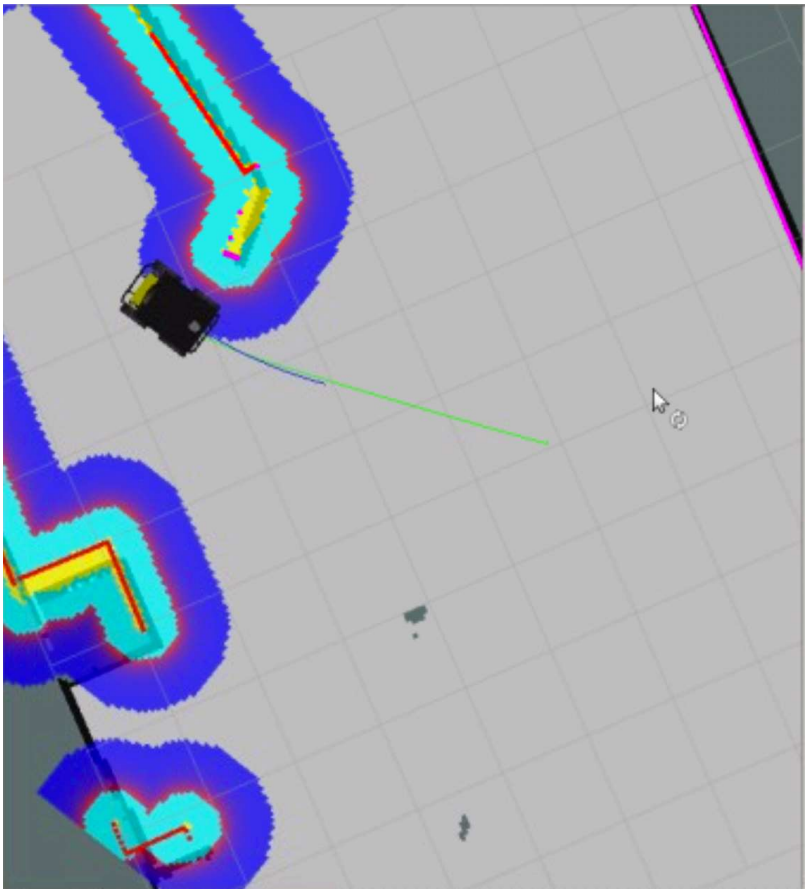
- End of Exercise 5.4 -

- Expected Result for Exercise 5.4 -

Regular sim_time (local plan in blue):



High sim_time (local plan in blue):



- End of Expected Result -

As you can see from the exercise above, the higher the **sim_time** parameter is, the longer the local plan calculated will be. However, bear in mind that this will also increase the computation resources used.

Trajectory Scoring Parameters

- **/path_distance_bias (default: 32.0)**: The weighting for how much the controller should stay close to the path it was given
- **/goal_distance_bias (default: 24.0)**: The weighting for how much the controller should attempt to reach its local goal; also controls speed
- **/occdist_scale (default: 0.01)**: The weighting for how much the controller should attempt to avoid obstacles

Here you have an example of the dwa_local_planner_params.yaml:

In [ ]:
```
# # Trajectory scoring parameters
  # path_distance_bias: 32.0 # The weighting for how much the controller shoul
  # goal_distance_bias: 24.0 # The weighting for how much the controller shoul
  # occdist_scale: 0.01 # The weighting for how much the controller should att
  # forward_point_distance: 0.325 # The distance from the center point of the
  # stop_time_buffer: 0.2  # The amount of time that the robot must stThe absc
  # scaling_speed: 0.25 # The absolute value of the veolicty at which to start
  # max_scaling_factor: 0.2 # The maximum factor to scale the robot's footprin
```

- Exercise 5.5 -

Take a chance to modify these paremeters by yourself and see how they affect the planning process. For instance, you could try to change the **path_distance_bias** parameter in the local planner parameters file.

Pay special attention to check if you notice any differences in the performance.

- End of Exercise 5.5 -

In the global planner section, we already introduced you to costmaps, focusing on the global costmap. So, now it's time to talk a little bit about the local costmap.

5.3 Local Costmap

The first thing you need to know is that the **local planner uses the local costmap in order to calculate local plans**.

Unlike the global costmap, the local costmap is created directly from the robot's sensor readings. Given a width and a height for the costmap (which are defined by the user), it keeps the robot in the center of the costmap as it moves throughout the environment, dropping obstacle information from the map as the robot moves.

Let's do an exercise so that you can get a better idea of how the local costmap looks, and how to differentiate a local costmap from a global costmap.

- Exercise 5.6 -

a) Open Rviz and add the proper displays in order to visualize the global and the local costmaps.

b) Execute the following command in order to spawn an obstacle in the room.

Check if you already have the **object.urdf** file in your workspace. If you don't have it yet, you'll need to execute the following command in order to move it to your workspace.

▶ Execute in Shell #2

In [ ]:
```
cp /home/simulations/public_sim_ws/src/all/turtlebot/turtlebot_navigation_gaze
```

Now, spawn the object.

▶ Execute in Shell #2

In [ ]:
```
rosrun gazebo_ros spawn_model -file /home/user/catkin_ws/src/object.urdf -urdf
```

c) Launch the keyboard Teleop and move close to the spawned object.

▶ Execute in Shell #2

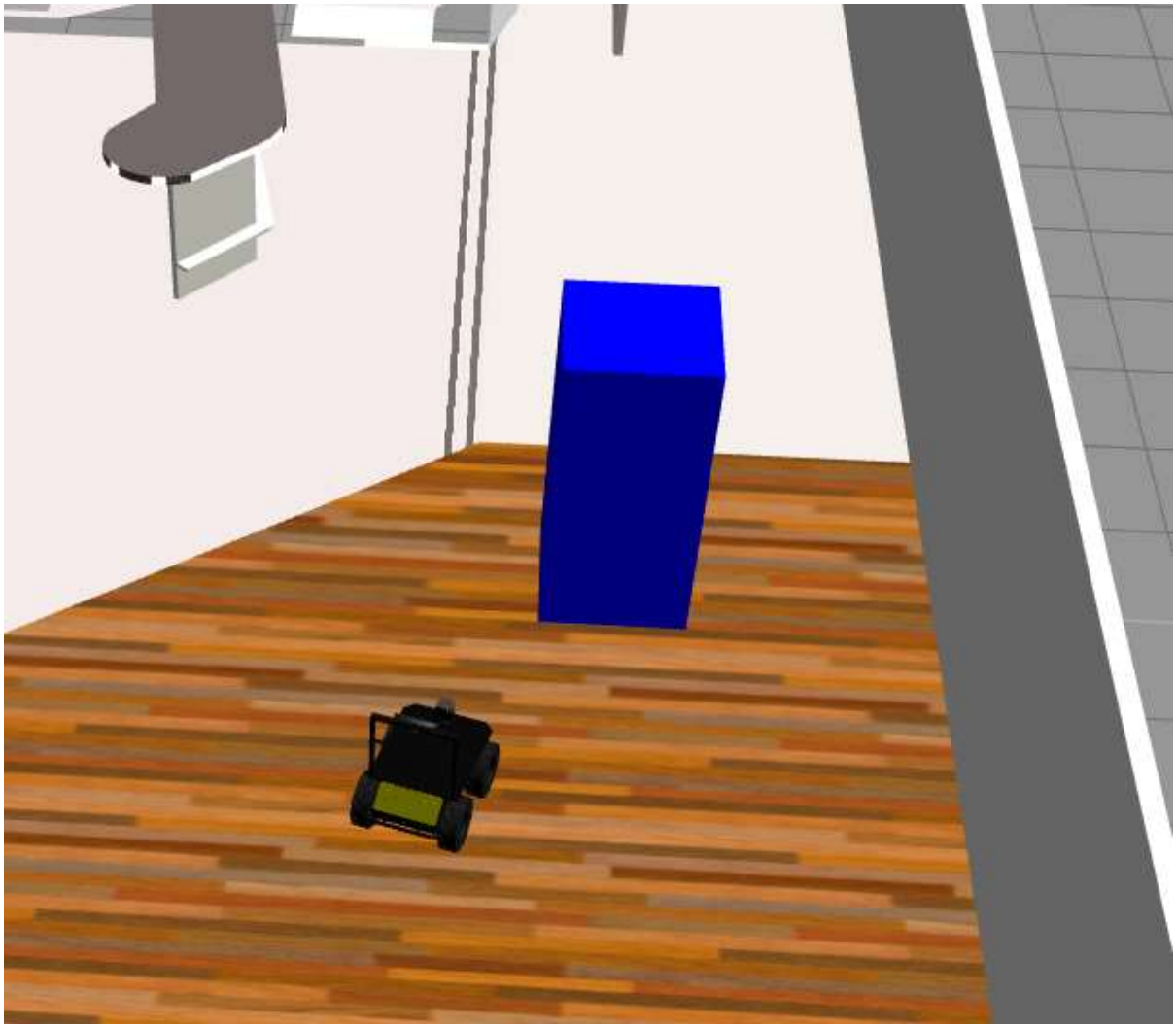
In [ ]:
```
roslaunch husky_launch keyboard_teleop.launch
```

d) Check the differences between the global and local Costmaps.

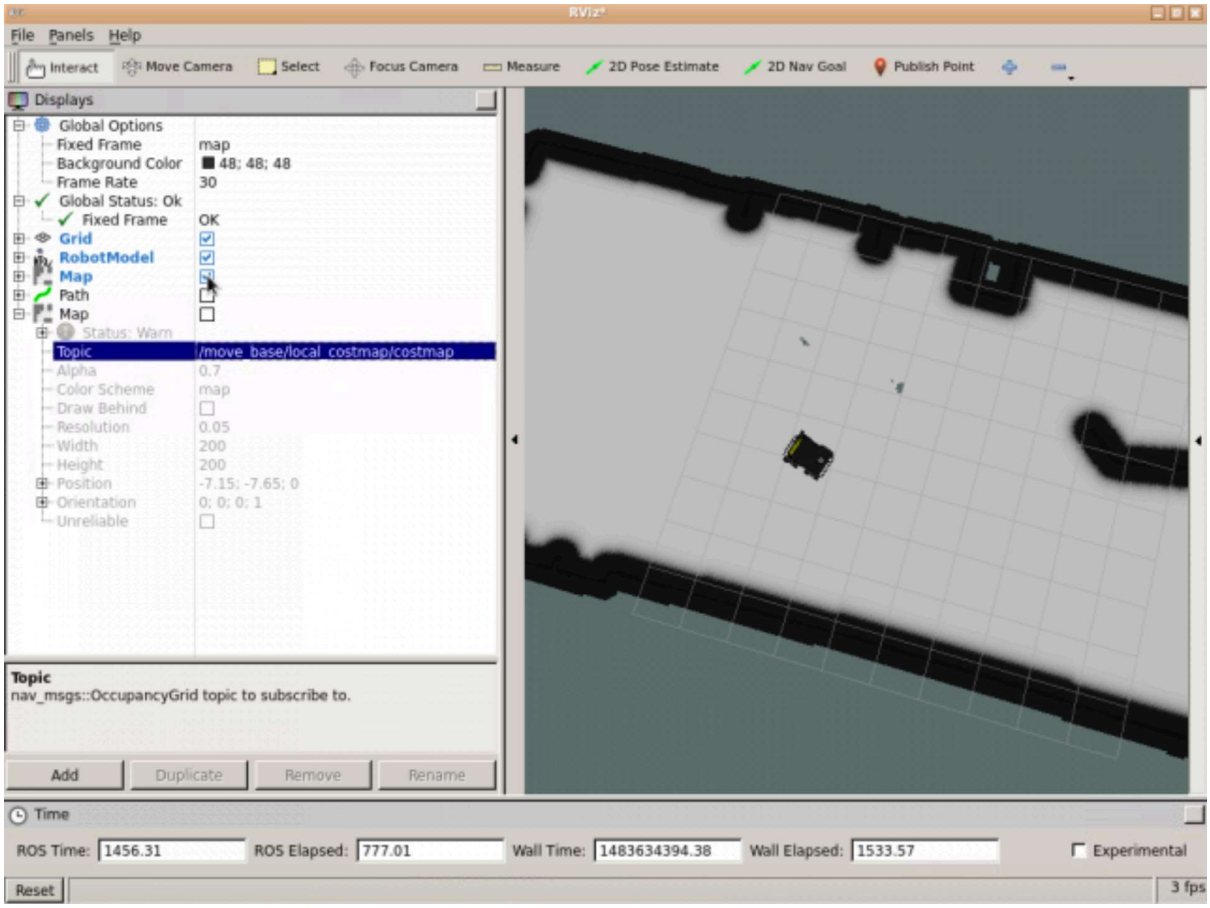
- End of Exercise 5.6 -

- Expected Result for Exercise 5.6 -

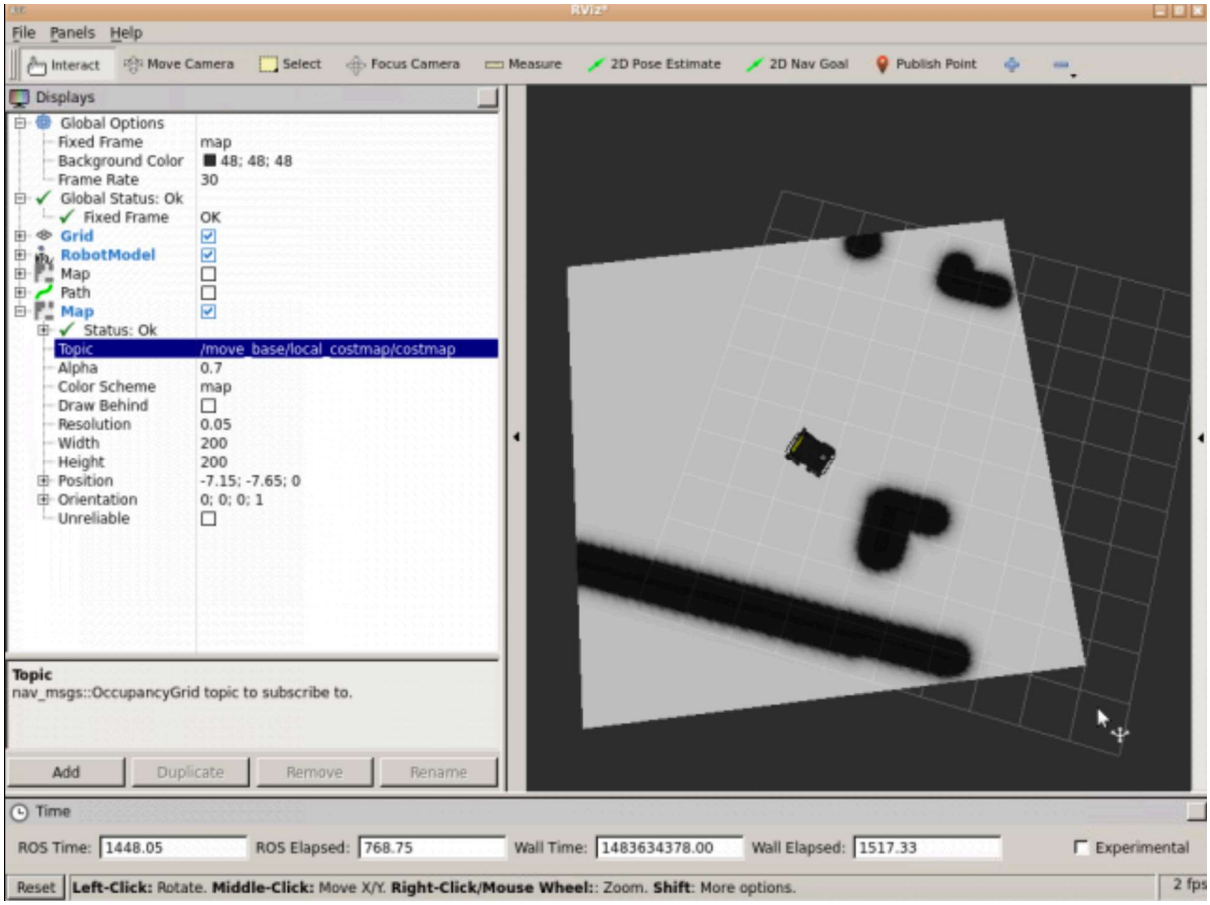
Husky facing the spawned obstacle:



Global Costmap (Obstacle doesn't appear):



Local Costmap (Obstacle does appear):



- End of Expected Result -

So, as you've seen in the previous exercise, the **local costmap does detect new objects that appear in the simulation, while the global costmap doesn't**.

This happens, as you may have already deduced, because the global costmap is created from a static map file. This means that the costmap won't change, even if the environment does. The local costmap, instead, is created from the robot's sensor readings, so it will always keep updating with new readings from the sensors.

Since the global costmap and the local costmap don't have the same behavior, the parameters file must also be different. Let's have a look at the most important parameters that we need to set for the local costmap.

5.3.1 Local Costmap Parameters

The parameters you need to know are the following:

- **global_frame**: The global frame for the costmap to operate in. In the local costmap, this parameter has to be set to "/odom".
- **robot_base_frame**: The name of the frame for the base link of the robot.
- **rolling_window**: Whether or not to use a rolling window version of the costmap. If the static_map parameter is set to true, this parameter must be set to false. In the local costmap, this parameter has to be set to "true."
- **update_frequency (default: 5.0)**: The frequency in Hz for the map to be updated.
- **width (default: 10)**: The width of the costmap.
- **heigth (default: 10)**: The height 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.

As you can see, they are same as for the global costmap. However, we've added a couple of parameters which are interesting when working with local costmaps: **width** and **heigth**. We've also added **update_frequency**, which we will see later in this Unit.

Let's now do a quick exercise to test the *width* and *heigth* parameters.

- Exercise 5.7 -

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

b) Copy the contents of the ***costmap_local.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 local costmap parameters file you just created.

d) Launch Rviz and visualize the local costmap again. Visualize both the map and costmap modes.

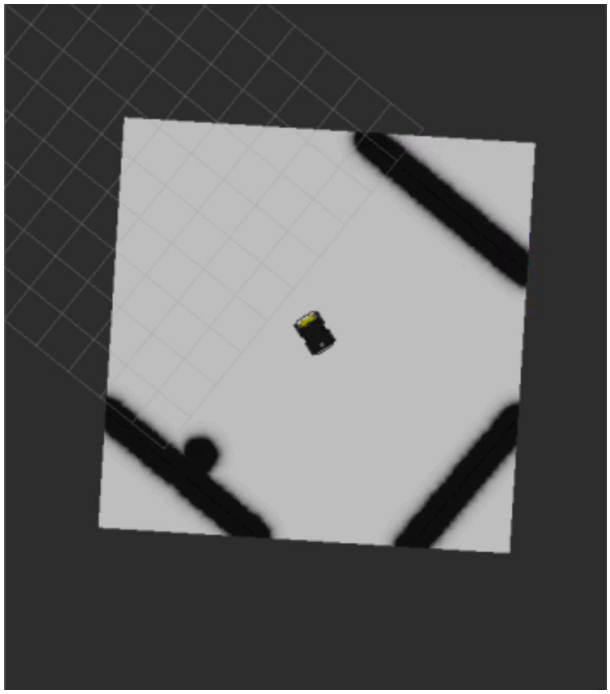
e) Modify the **width** and **height** parameters and put them to 5. Visualize the costmap again.

NOTE: Keep in mind that, for your case, the **width** and **heigth** parameters are being loaded directly from the launch file. So, you will have to remove from the launch file and add them to the parameters file.

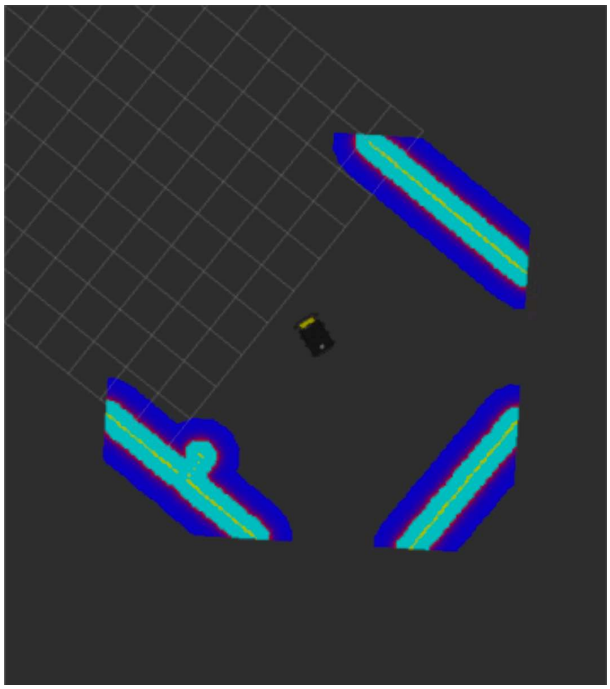
- End of Exercise 5.7 -

- Expected Result for Exercise 5.7 -

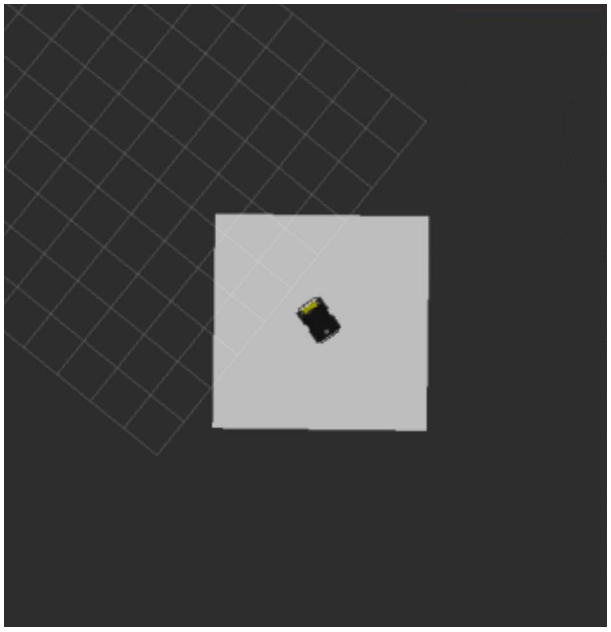
10x10 costmap (map view):



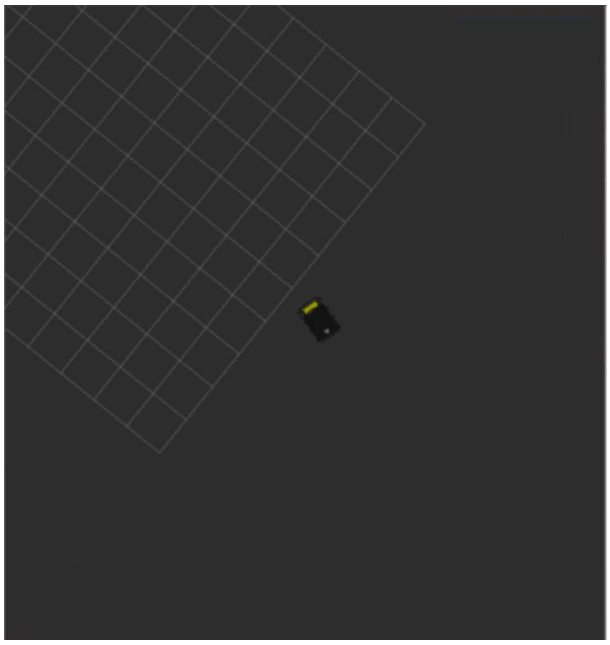
10x10 costmap (costmap view):



5x5 costmap (map view):



5x5 costmap (costmap view):



- End of Expected Result -

As you've seen in the previous exercise, it's very important to set a correct width and height for your costmap. Depending on the environment you want to navigate, you will have set one value or another in order to properly visualize the obstacles.

Let's keep working with the local costmap parameters. **Despite the parameters being the same as for the global costmap, they are not set with the same values**.

For instance, for the local costmap the **rolling_window** parameter will be set to *true*. This way we are indicating that we don't want the costmap to be initialized from a static map (as we did with the global costmap), but to be built from the robot's sensor readings.

Also, since we won't have any static map, the **global_frame** parameter needs to be set to **odom**.

You can see this in the **my_local_costmap_params.yaml** file you created in the previous exercise:

In [ ]:
```
global_frame: odom
rolling_window: true
```

Just as we saw for the global costmap, layers can also be added to the local costmap. In the case of the local costmap, you will usually add these 2 layers:

- **costmap_2d::ObstacleLayer**: Used for obstacle avoidance.
- **costmap_2d::InflationLayer**: Used to inflate obstacles.

So, you will end up with something like this:

In [ ]:
```
plugins:
    - {name: obstacle_layer,      type: "costmap_2d::ObstacleLayer"}
    - {name: inflation_layer,     type: "costmap_2d::InflationLayer"}
```

- Notes -

VERY IMPORTANT: **Note that the **obstacle layer uses different plugins for the **local costmap** and the **global costmap**. For the local costmap, it uses the **costmap_2d::ObstacleLayer**, and for the global costmap it uses the **costmap_2d::VoxelLayer**. This is very important because it is a common error in Navigation to use the wrong plugin for the obstacle layers.

- End of Notes -

Let's add one final parameter! As you've already seen through the exercises, the local costmap keeps updating itself. These update cycles are made at a rate specified by the **update_frequency** parameter. Each cycle works as follows:

1. Sensor data comes in.
2. Marking and clearing operations are performed.
3. The appropriate cost values are assigned to each cell.
4. Obstacle inflation is performed on each cell with an obstacle. This consists of propagating cost values outwards from each occupied cell out to a specified inflation radius.

- Exercise 5.8 -

a) In the local costmap parameters file, change the **update_frequency** parameter of the map to be slower.

b) Repeat Exercise 5.6 again, and see what happens now.

- End of Exercise 5.8 -

- Expected Result for Exercise 5.8 -

The object in the costmap is spawned with a little delay.

- End of Expected Result -

Now, you may be wondering... what are the marking and clearing operations you mentioned above?

As you already know, the costmap automatically subscribes to the sensor topics and updates itself according to the data it receives from them. Each sensor is used to either **mark** (insert obstacle information into the costmap), **clear** (remove obstacle information from the costmap), or both.

A **marking** operation is just an index into an array to change the cost of a cell.
A **clearing** operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported.

The marking and clearing operations can be defined in the **obstacle layer**.

At this point, we can almost say that you already know how to configure both global and local costmaps. But if you remember, there's still a paramters file we haven't talked about. That's the **common costmap parameters file**. These parameters will affect both the global and the local costmap.

5.3.2 Common Costmap Parameters

Basically, the most important parameters you'll have to set in this file are the following:

- **footprint**: Footprint is the contour of the mobile base. In ROS, it is represented by a two-dimensional array of the form [x0, y0], [x1, y1], [x2, y2], ...]. This footprint will be used to compute the radius of inscribed circles and circumscribed circles, which are used to inflate obstacles in a way that fits this robot. Usually, for safety, we want to have the footprint be slightly larger than the robot’s real contour.
- **robot_radius**: In case the robot is circular, we will specify this parameter instead of the footprint.
- **layers parameters**: Here we will define the parameters for each layer.

Each layer has its own parameters.

Obstacle Layer

The obstacle layer is in charge of the **marking and clearing operations**.

As you already know, the costmap automatically subscribes to the sensor topics and updates itself according to the data it receives from them. Each sensor is used to either mark (insert obstacle information into the costmap), clear (remove obstacle information from the costmap), or both.

A marking operation is just an index into an array to change the cost of a cell.
A clearing operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported.

The marking and clearing operations can be defined in the obstacle layer.

In order to configure the obstacle layer, we need to first set a name for the layer and then set the **observation_sources** parameter.

- **observation_sources (default: "")**: A list of observation source names separated by spaces. This defines each of the *source_name* namespaces defined below.

You can chek how it is done in the **costmap_common.yaml** file from the *husky_navigation* package:

In [ ]:
```
obstacles_laser: # Name of the layer
    observation_sources: laser # We define 1 observation_source named laser
```

Now we can define the specific parameters for this *observation_source*. Each *source_name* in *observation_sources* defines a namespace in which parameters can be set:

- **/source_name/topic (default: source_name)**: The topic on which sensor data comes in for this source. Defaults to the name of the source.
- **/source_name/data_type (default: "PointCloud")**: The data type associated with the topic, right now only "PointCloud," "PointCloud2," and "LaserScan" are supported.
- **/source_name/clearing (default: false)**: Whether or not this observation should be used to clear out freespace.
- **/source_name/marking (default: true)**: Whether or not this observation should be used to mark obstacles.
- **/source_name/inf_is_valid (default: false)**: Allows for Inf values in "LaserScan" observation messages. The Inf values are converted to the laser's maximum range.
- **/source_name/max_obstacle_height (default: 2.0)**: The maximum height of any obstacle to be inserted into the costmap, in meters. This parameter should be set to be slightly higher than the height of your robot.
- **/source_name/obstacle range (default: 2.5)**: The default maximum distance from the robot at which an obstacle will be inserted into the cost map, in meters. This can be overridden on a per-sensor basis.
- **/source_name/raytrace_range (default: 3.0)**: The default range in meters at which to raytrace out obstacles from the map using sensor data. This can be overridden on a per-sensor basis.

You can chek how it is done in the **costmap_common.yaml** file from the *husky_navigation* package:

In [ ]:
```
laser: {data_type: LaserScan, clearing: true, marking: true, topic: scan, inf
```

As you can see in our example, we are just declaring 1 *observation_source*, which is for the laser. This means that our Husky robot will build its local costmap based on the data received from its laser.

- Exercise 5.9 -

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

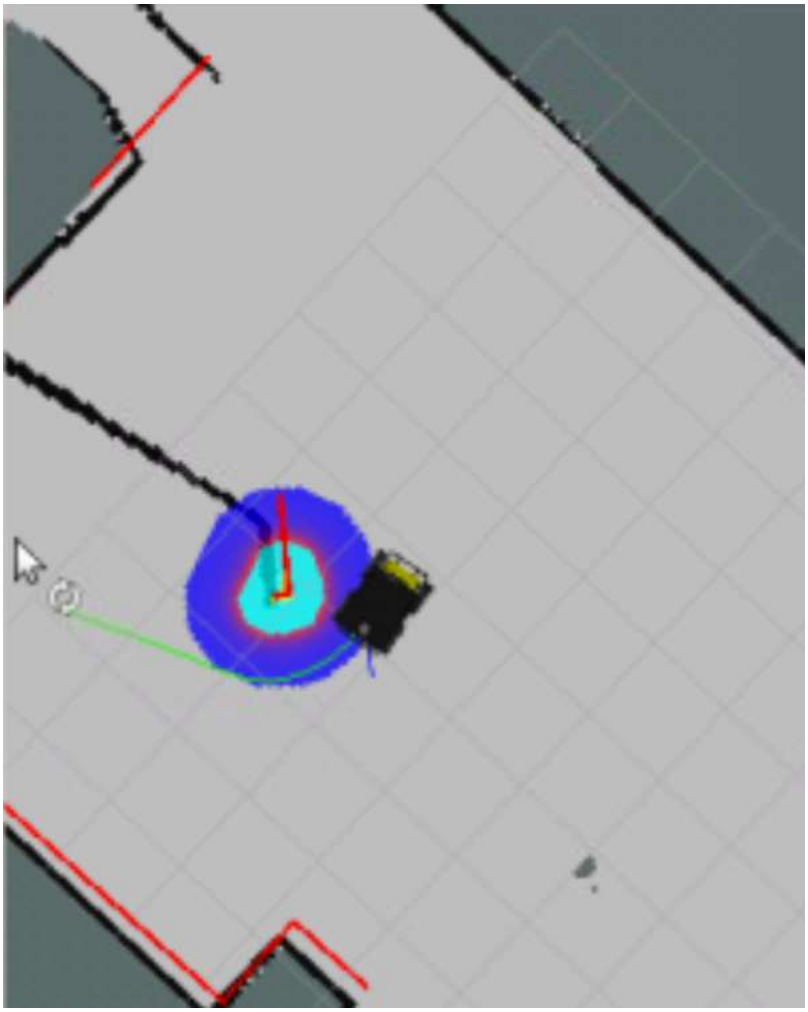
b) Copy the contents of the ***costmap_common.yaml*** file of the *husky_navigation* package into this new file.

a) Now, modify the **obstacle_range** parameter and set it to 1.

b) Move the robot close to an obstacle and see what happens.

- End of Exercise 5.9 -

- Expected Result for Exercise 5.9 -



- End of Expected Result -

Inflation Layer

The inflation layer is in charge of performing inflation in each cell with an obstacle.

- **inflation_radius (default: 0.55)**: The radius in meters to which the map inflates obstacle cost values.
- **cost_scaling_factor (default: 10.0)**: A scaling factor to apply to cost values during inflation.

- Exercise 5.10 -

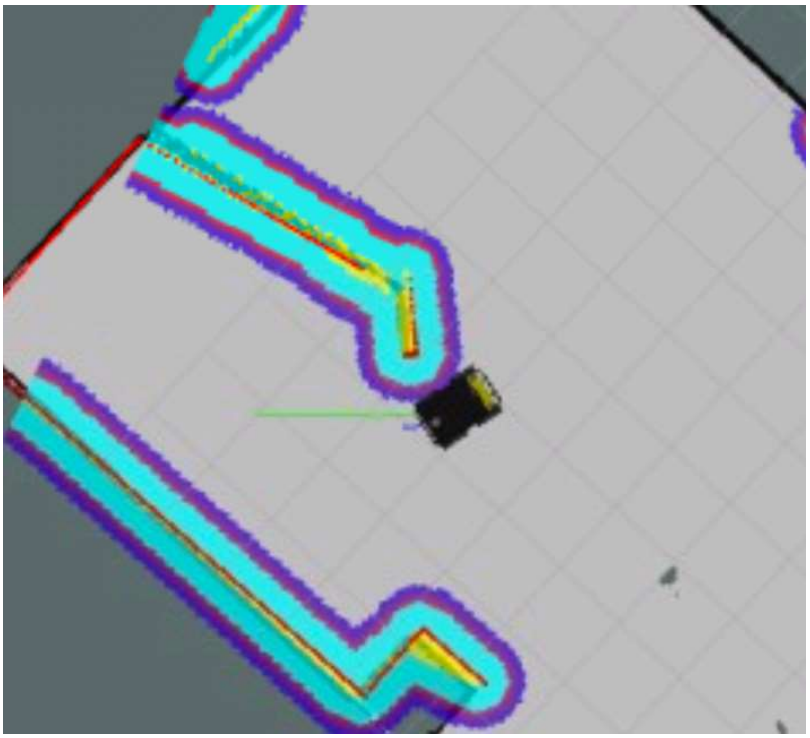
a) Now, modify the **inflation_radius** parameter of the costmap to be slower.

b) Move close to an object and check the difference.

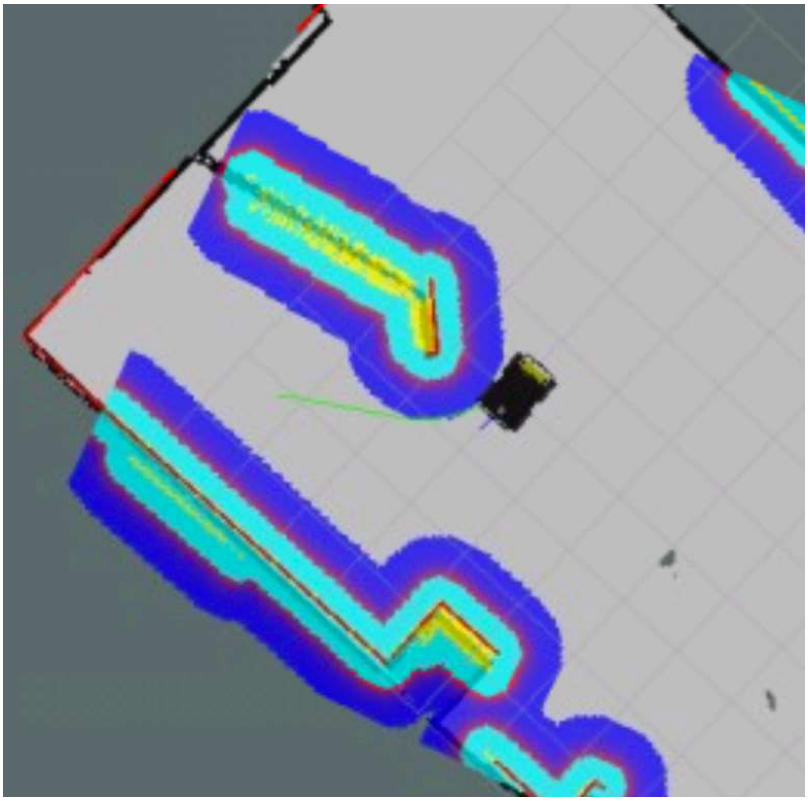
- End of Exercise 5.10 -

- Expected Result for Exercise 5.10 -

Low inflation:



High inflation:



- End of Expected Result -

Static Layer

The static layer is in charge of providing the static map to the costmaps that require it (global costmap).

- **map_topic (string, default: "map")**: The topic that the costmap subscribes to for the static map.

5.4 Recovery Behaviors

It could happen that while trying to perform a trajectory, the robot gets stuck for some reason. Fortunately, if this happens, the ROS Navigation Stack provides methods that can help your robot to get unstuck and continue navigating. These are the **recovery behaviors**.

The ROS Navigation Stack provides 2 recovery behaviors: **clear costmap** and **rotate recovery**.

In order to enable the recovery behaviors, we need to set the following parameter in the move_base parameters file:

- **recovery_behavior_enabled (default: true)**: Enables or disables the recovery behaviors.

You can chek how it is done in the **my_move_base_params.yaml** file from the *my_move_base_launcher* package:

In [ ]:
```
recovery_behaviour_enabled: true
```

5.4.1 Rotate Recovery

Bascially, the rotate recovery behavior is a simple recovery behavior that attempts to clear out space by rotating the robot 360 degrees. This way, the robot may be able to find an obstacle-free path to continue navigating.

It has some parameters that you can customize in order to change or improve its behavior:

Rotate Recovery Parameters

- **/sim_granularity (default: 0.017)**: The distance, in radians, between checks for obstacles when checking if an in-place rotation is safe. Defaults to 1 degree.
- **/frequency (default: 20.0)**: The frequency, in HZ, at which to send velocity commands to the mobile base.

Other Parameters

- **/yaw_goal_tolerance (double, default: 0.05)**: The tolerance, in radians, for the controller in yaw/rotation when achieving its goal
- **/acc_lim_th (double, default: 3.2)**: The rotational acceleration limit of the robot, in radians/sec^2
- **/max_rotational_vel (double, default: 1.0)**: The maximum rotational velocity allowed for the base, in radians/sec
- **/min_in_place_rotational_vel (double, default: 0.4)**: The minimum rotational velocity allowed for the base while performing in-place rotations, in radians/sec

These parameters are also set in the move_base parameters file.

5.4.2 Clear Costmap

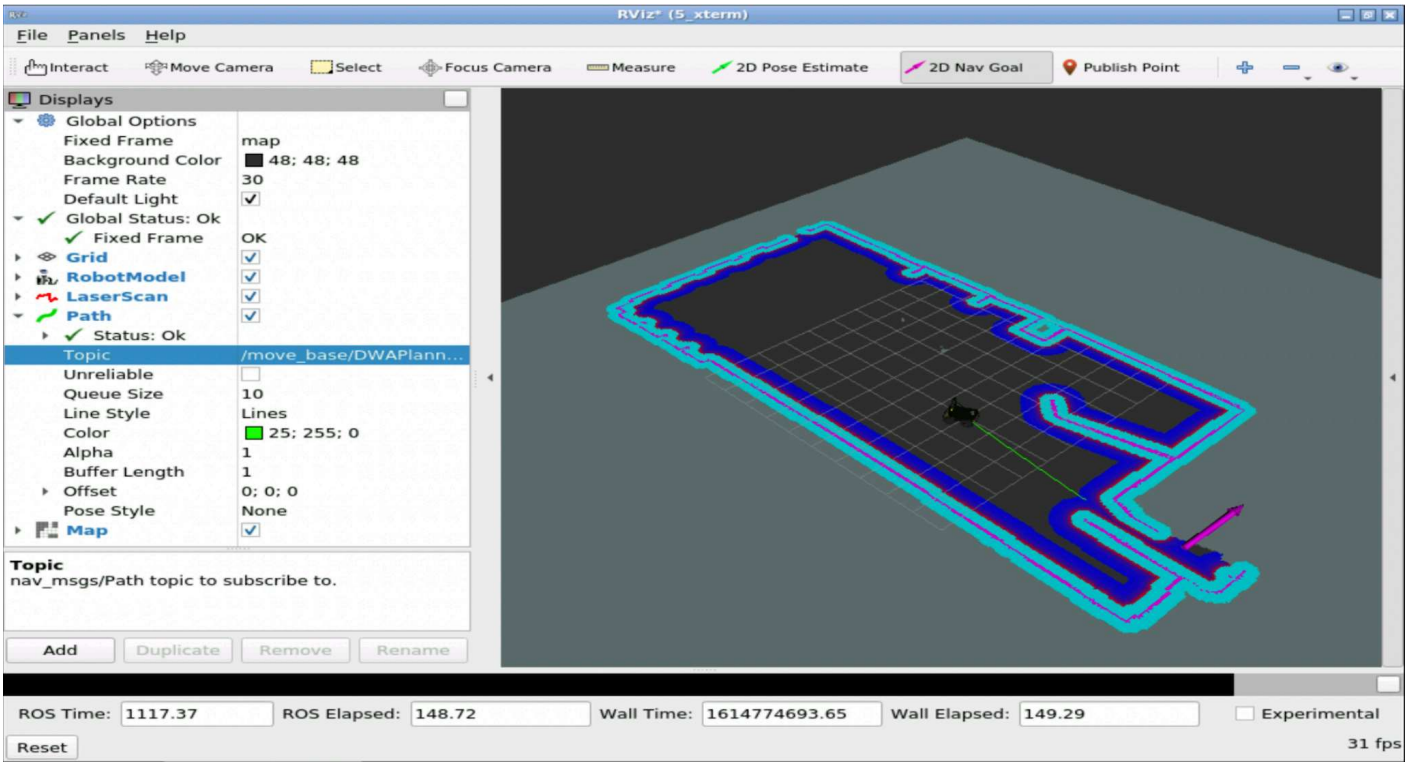
The clear costmap recovery is a simple recovery behavior that clears out space by clearing obstacles outside of a specified region from the robot's map. Basically, the local costmap reverts to the same state as the global costmap.

Let's try a quick exercise to put the recovery behaviors under test.

- Exercise 5.11 -

Send a navigation goal to the robot which is not reachable and check if the Recovery Behaviors are triggered.

For instance, you could set a goal inside the kitchen of the simulated world. Check the image below:



- End of Exercise 5.11 -

- Expected Result for Exercise 5.11 -

Output from move base node:

```
[ INFO] [1614774620.440536275, 1044.464000000]: Got new plan
[ WARN] [1614774620.443822772, 1044.468000000]: DWA planner failed to produce path.
[ WARN] [1614774620.641293863, 1044.664000000]: Clearing both costmaps to unstuck robot (3.00m).
[ INFO] [1614774621.042992963, 1045.064000000]: Got new plan
[ WARN] [1614774621.045880994, 1045.067000000]: DWA planner failed to produce path.
[ WARN] [1614774621.243453410, 1045.264000000]: Rotate recovery behavior started.
[ INFO] [1614774627.969798788, 1051.964000000]: Got new plan
[ WARN] [1614774628.575323044, 1052.567000000]: DWA planner failed to produce path.
```

Rotate Recovery Behavior:

- End of Expected Result -

The move_base node also provides a service in order to clear out obstacles from a costmap. This service is called **/move_base/clear_cotmaps**.

Bear in mind that by clearing obstacles from a costmap, you will make these obstacles invisible to the robot. So, be careful when calling this service since it could cause the robot to start hitting obstacles.

- Exercise 5.12 -

a) If there's not one yet, add an object to the scene that doesn't appear in the global costmap. For instane, the object in Exercise 5.6.

b) Move the robot so that it detects this new obstacle in the local costmap.

c) Turn the robot so that it doesn't see anymore the obstacle (the laser beams don't detect it).

d) Perform a call to the **/clear_costmaps** service through the WebShell, and check what happens.

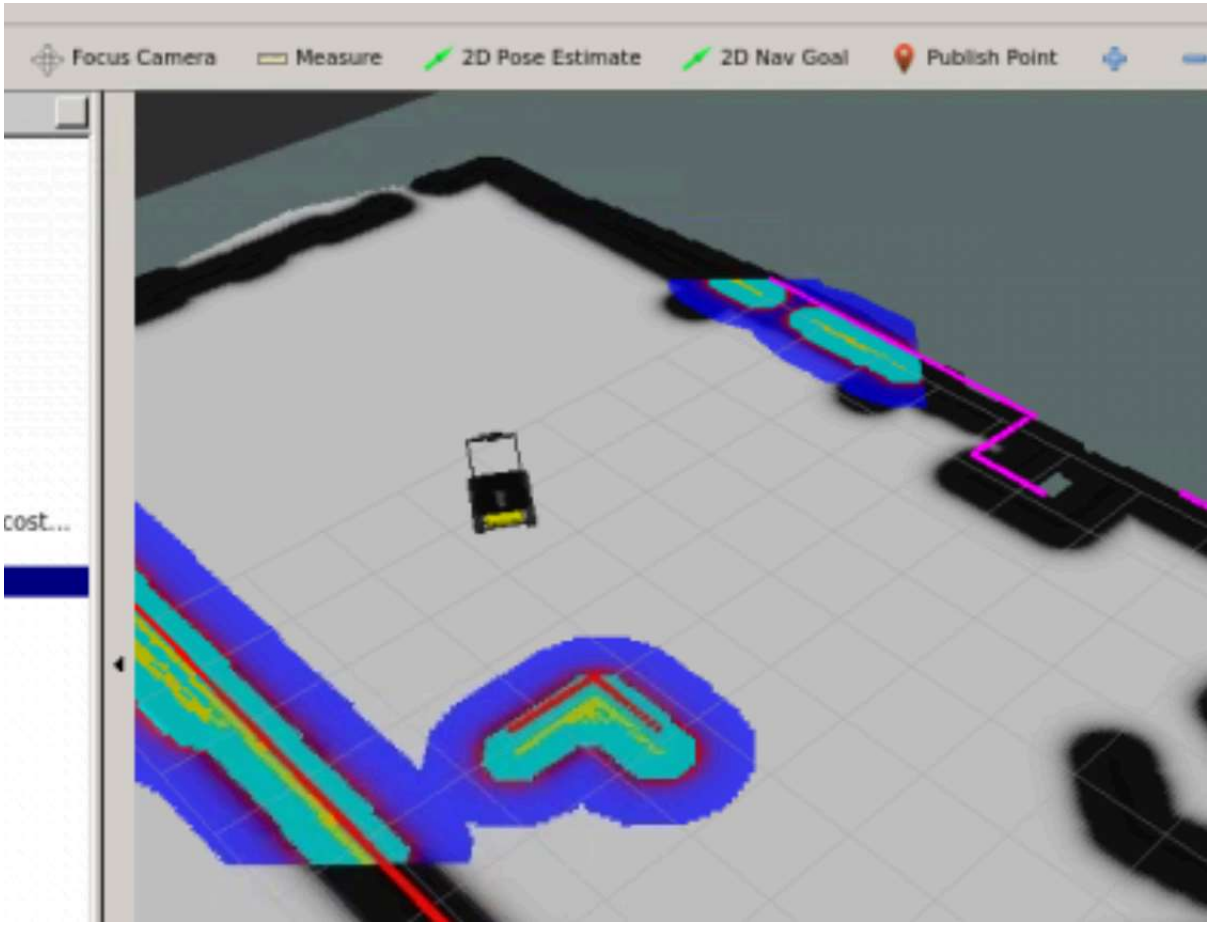
▶ Execute in Shell #1

In [ ]:
```
rosservice call /move_base/clear_costmaps "{}"
```

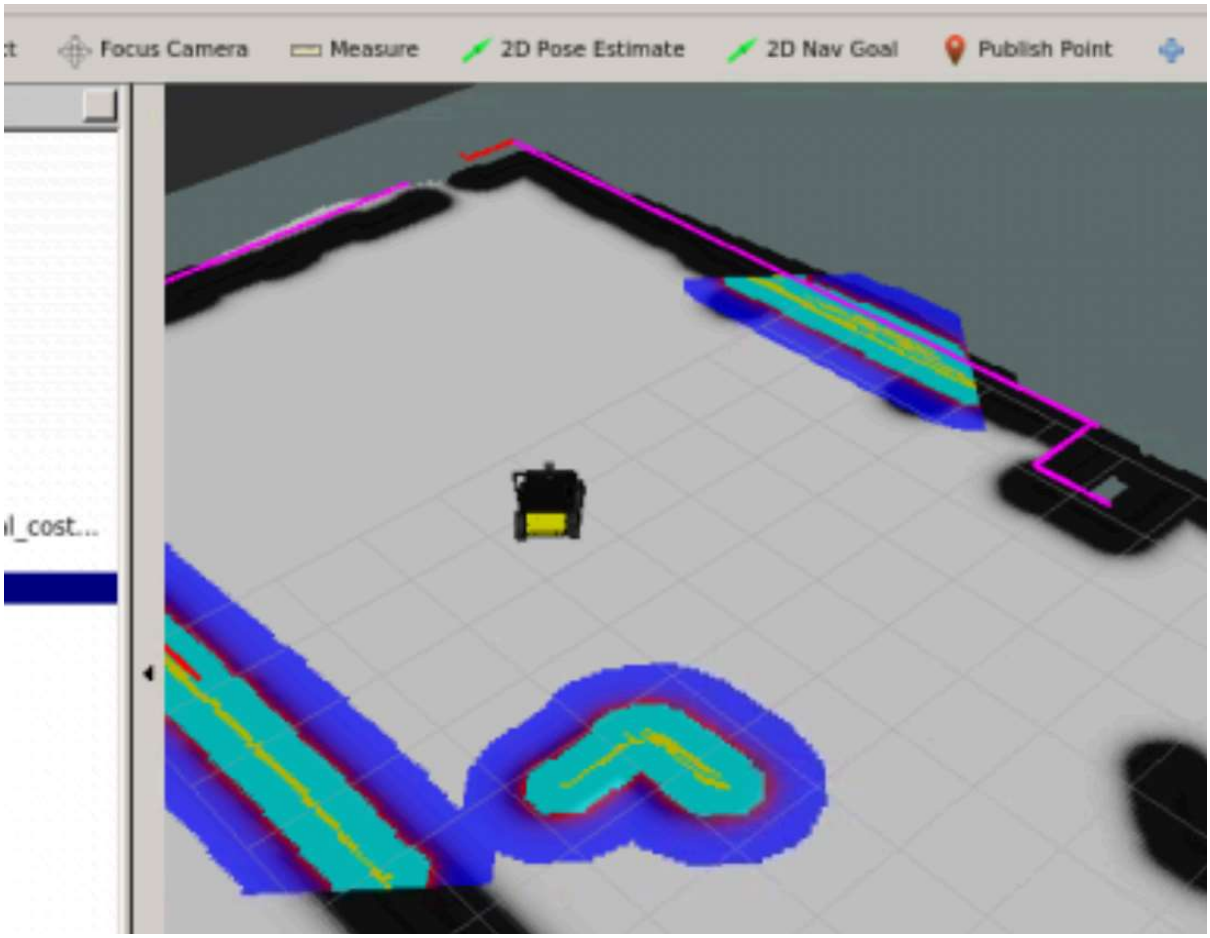
- End of Exercise 5.12 -

- Expected Result for Exercise 5.12 -

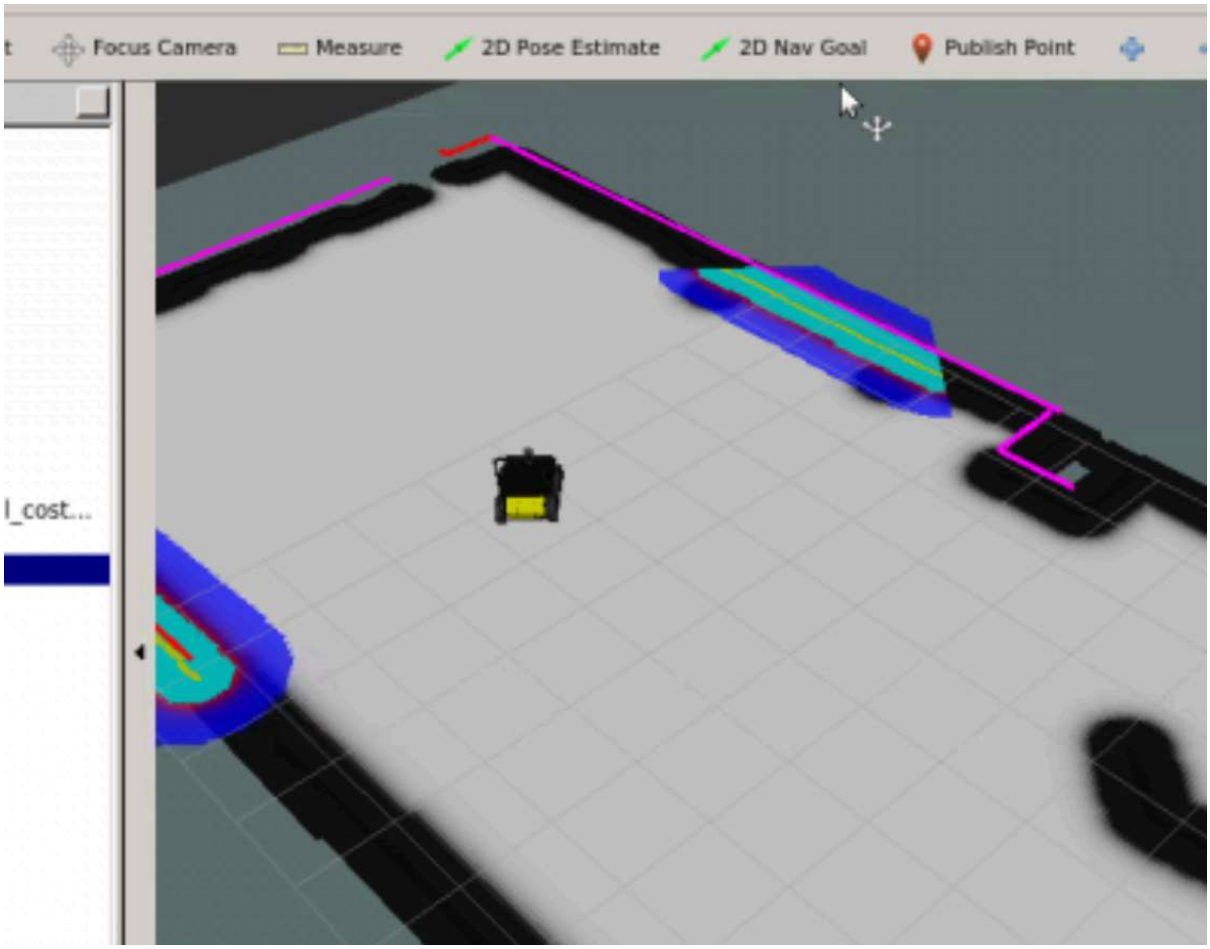
Object detected by the laser and placed into the local Ccostmap:



Husky turns and laser doesn't detect the object anymore, but it still appears in the local costmap.



After calling the **/move_base/clear_costmaps** service, the object is cleared from the local costmap:



- End of Expected Result -

5.5 Recap

Congratulations! At this point, you've already seen almost all of the important parts that this chapter covers. And since this is the last chapter of the course, this means that you are very close to knowing how to deal with ROS Navigation in its entirety!

Anyways, you may be overwhelmed with all of the information that you've received about Path Planning. That's why I think this is a good moment to do a summary of all that you've seen in this chapter up until now. Let's begin!

The move_base node

The move_base node is, basically, the node that coordinates all of the Path Planning System. It takes a goal pose as input, and outputs the necessary velocity commands in order to move the robot from an initial pose to the specified goal pose. In order to achieve this, the move_base node manages a whole internal process where it take place for different parts:

- global planner
- local planner
- costmaps
- recovery behaviors

The global planner

When a new goal is received by the move_base node, it is immediately sent to the global planner. The global planner, then, will calculate a safe path for the robot to use to arrive to the specified goal. The global planner uses the global costmap data in order to calculate this path.

There are different types of global planners. Depending on your setup, you will use one or another.

The local planner

Once the global planner has calculated a path for the robot, this is sent to the local planner. The local planner, then, will execute this path, breaking it into smaller (local) parts. So, given a plan to follow and a map, the local planner will provide velocity commands in order to move the robot. The local planner operates over a local costmap.

There are different types of local planners. Dependng on the kind of performance you require, you will use one or another.

Costmaps

Costmaps are, basically, maps that represent which points of the map are safe for the robot to be in, and which ones are not. There are 2 types of costmaps:

- global costmap
- local costmap

Basically, the difference between them is that the global costmap is built using the data from a previously built static map, while the local costmap is built from the robot's sensor readings.

Recovery Behaviors

The recovery behaviors provide methods for the robot in case it gets stuck. The Navigation Stack provides 2 different recovery behaviors:

- rotate recovery
- clear costmap

Configuration

Since there are lots of different nodes working together, the number of parameters available to configure the different nodes is also very high. I think it would be a great idea if we summarize the different parameter files that we will need to set for Path Planning. The parameter files you'll need are the following:

- **move_base_params.yaml**
- **global_planner_params.yaml**
- **local_planner_params.yamls**
- **common_costmap_params.yaml**
- **global_costmap_params.yaml**
- **local_costmap_params.yaml**

Besides the parameter files shown above, we will also need to have a launch file in order to launch the whole system and load the different parameters.

Overall

Summarizing, this is how the whole path planning method goes:

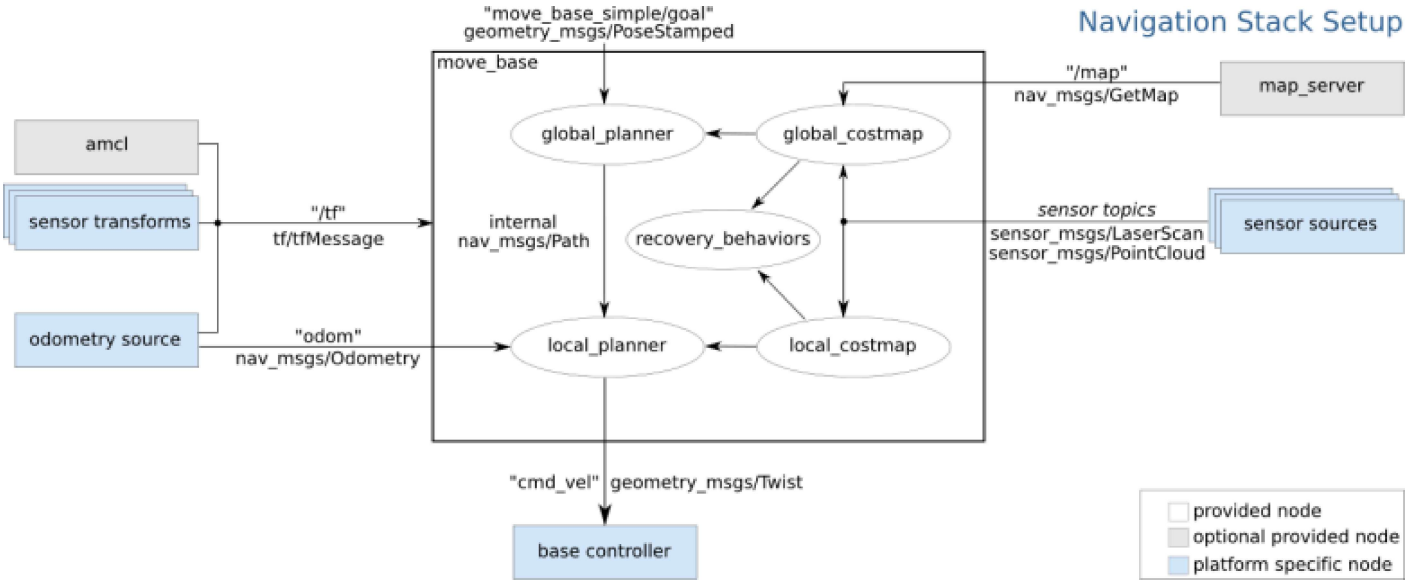
After getting the current position of the robot, we can send a goal position to the **move_base** node. This node will then send this goal position to a **global planner** which will plan a path from the current robot position to the goal position. This plan is in respect to the **global costmap**, which is feeding from the **map server**.

The **global planner** will then send this path to the **local planner**, which executes each segment of the global plan. The **local planner** gets the odometry and the laser data values and finds a collision-free local plan for the robot. The **local planner** is associated with the **local costmap**, which can monitor the obstacle(s) around the robot. The **local planner** generates the velocity commands and sends them to the base controller. The robot base controller will then convert these commands into real robot movement.

If the robot is stuck somewhere, the recovery behavior nodes, such as the **clear costmap recovery** or **rotate recovery**, will be called.

Now everything makes more sense, right?

So, with all the knowledge you've earned during this course, you can now look again at the below diagram and try to understand all the different elements that take part on it.



5.6 Dynamic Reconfigure

Until now, we've seen how to change parameters by modifying them in the parameters files. But, guess what... this is not the only way that you can change parameters! You can also change dynamic parameters by using the rqt_reconfigure tool. Follow the next steps:

- Exercise 5.14 -

a) Run the next command in order to open the rqt_reconfigure tool.

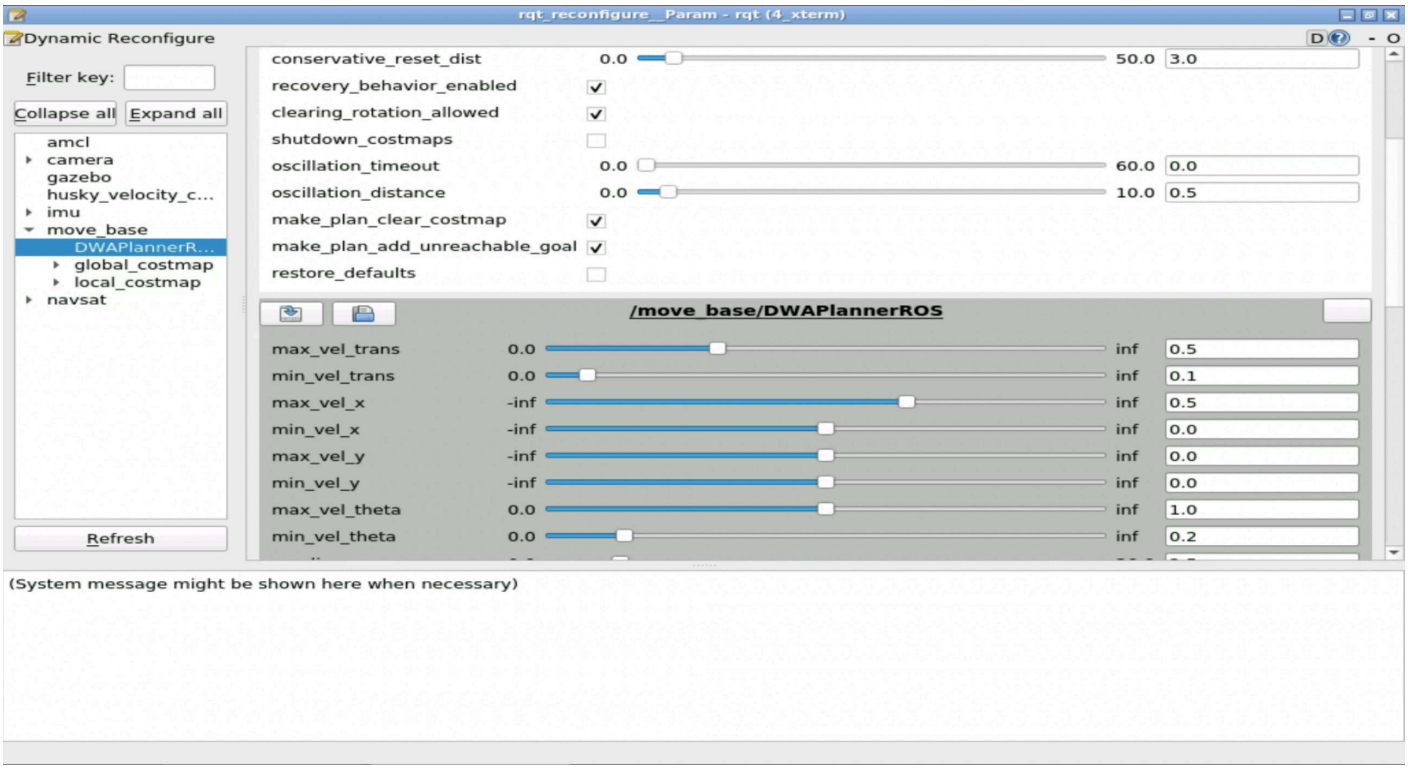
▶ Execute in Shell #2

In [ ]:
```
rosrun rqt_reconfigure rqt_reconfigure
```

- Open the move_base group.
- Select the DWAPlannerROS node.
- Play a little bit with the following 3 parameters:
  - **path_distance_bias**
  - **goal_distance_bias**
  - **occdist_scale**
- The above parameters are the ones involved in calculating the cost function, which is used to score each trajectory. More in detail, they define the following:
  - **path_distance_bias**: The weighting for how much the controller should stay close to the path it was given.
  - **goal_distance_bias**: The weighting for how much the controller should attempt to reach its local goal, also controls speed.
  - **occdist_scale**: The weighting for how much the controller should attempt to avoid obstacles.
- Open Rviz and visualize how the global and local plans change depending on the values set.

- End of Exercise 5.14 -

- Expected Result for Exercise 5.14 -



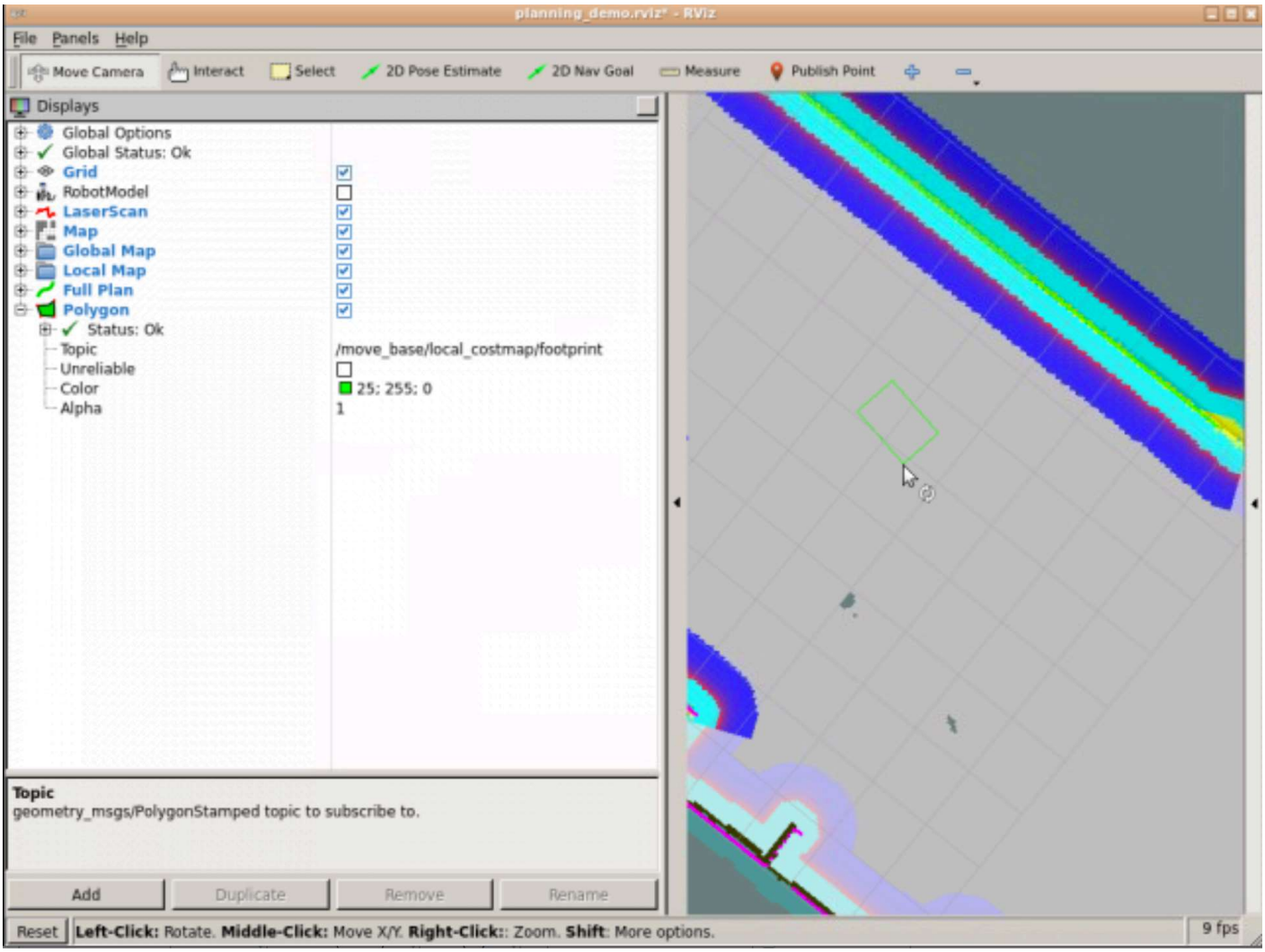
- End of Expected Result -

5.7 Other Useful Visualizations in Rviz

Until now, we've seen some ways of visualizing different parts of the move_base node process through Rviz. But, there are a couple more that may be interesting to know:

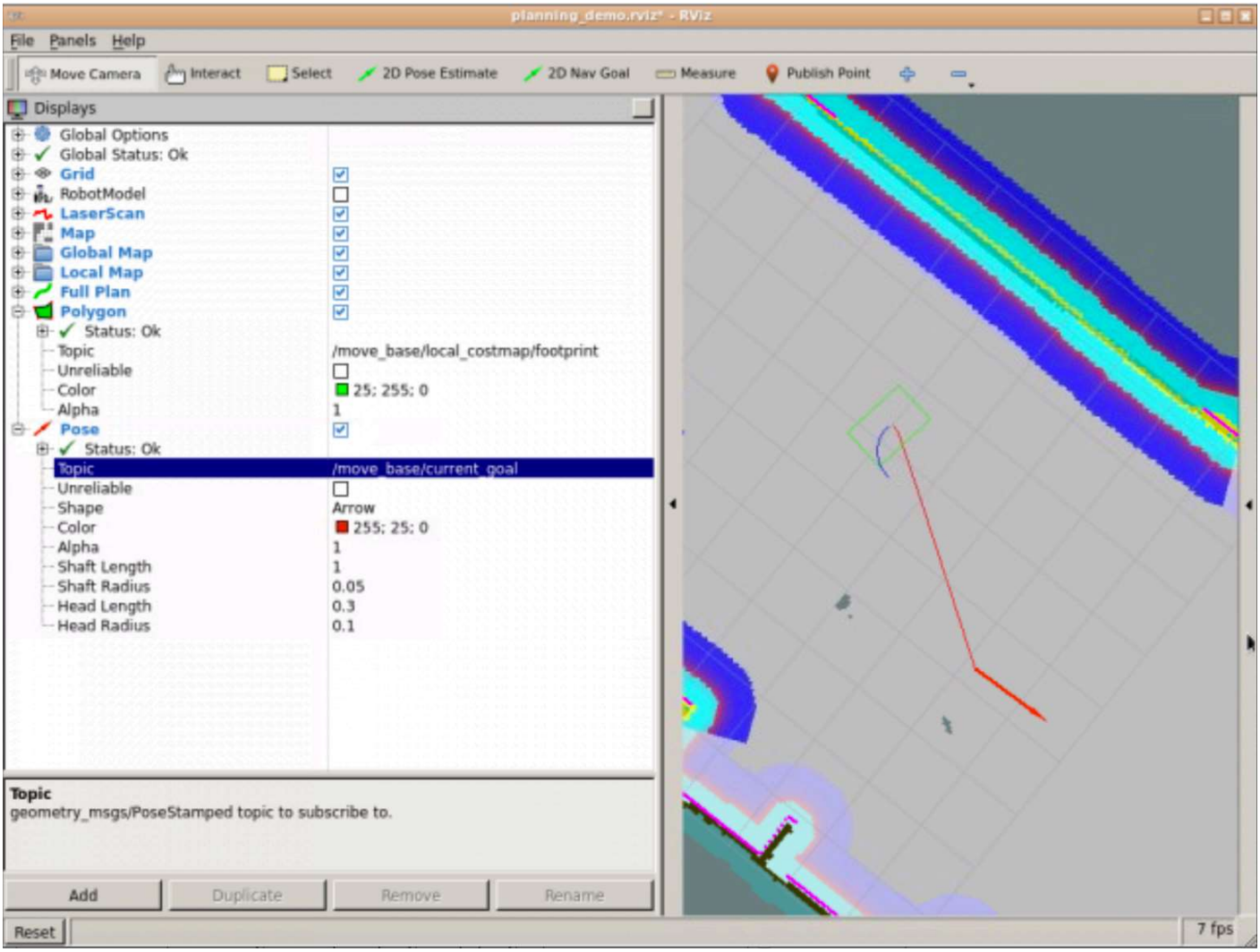
Robot Footprint

It shows the footprint of the robot.



Current Goal

To show the goal pose that the navigation stack is attempting to achieve, add a Pose Display and set its topic to /move_base_simple/goal. You will now be able to see the goal pose as a red arrow. It can be used to learn the final position of the robot.



- Exercise 5.15 -

Open Rviz and try to visualize the elements described above.

- End of Exercise 5.15 -