robotignite ACADEMY

Unit 1: Set Up the Navigation Stack

SUMMARY

Estimated time to completion: **1 hour**

The first thing you will need to do to be able to use the TEB Local Planner is set up the Navigation Stack. And that's exactly what you are going to learn in this unit! For this, you will learn how to:

- Set up the gmapping node
- Set up the amcl node
- Set up the move_base node
- Set TEB as the Local Planner to be used

END OF SUMMARY

Set up the gmapping node

The first thing you need to do in order to set up the ROS Navigation Stack is create a map of the environment you want to navigate. For that, you are going to need the **slam_gmapping** node that the Navigation Stack provides. To see how to do this, follow the next exercise:

Exercise 1.1

a) First of all, let's create a new package where we'll put all the files related to navigation.

Execute in WebShell #1

In [ ]:

```
catkin_create_pkg teb_navigation
```

b) Inside this new package, let's create two new directories: one named **launch** and the other one named **config**.

c) Now, let's create a launch file to start our slam_gmapping node!

gmapping.launch

In [ ]:

```
<?xml version="1.0"?>

<launch>

 <arg name="scan_topic" default="scan" />

 <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
   <rosparam>
     odom_frame: odom
     base_frame: base_link
     map_frame: map

     map_update_interval: 0.5 # Publish new map

     maxUrange: 10.0 # Should be just less than sensor range
     maxRange: 12.0 # Should be just greater than sensor range
     particles: 100 # Increased from 80

     # Update frequencies
     linearUpdate: 0.3
     angularUpdate: 0.5
     temporalUpdate: 2.0
     resampleThreshold: 0.5

     # Initial Map Size
     xmin: -100.0
     ymin: -100.0
     xmax: 100.0
     ymax: 100.0
     delta: 0.05

     # All default
     sigma: 0.05
```

```
    kernelSize: 1
    lstep: 0.05
    astep: 0.05
    iterations: 5
    lsigma: 0.075
    ogain: 3.0
    lskip: 0
    llsamplerange: 0.01
    llsamplestep: 0.01
    lasamplerange: 0.005
    lasamplestep: 0.005

  </rosparam>
  <remap from="scan" to="$(arg scan_topic)"/>
 </node>
</launch>
```

The most important parameters in these file are:

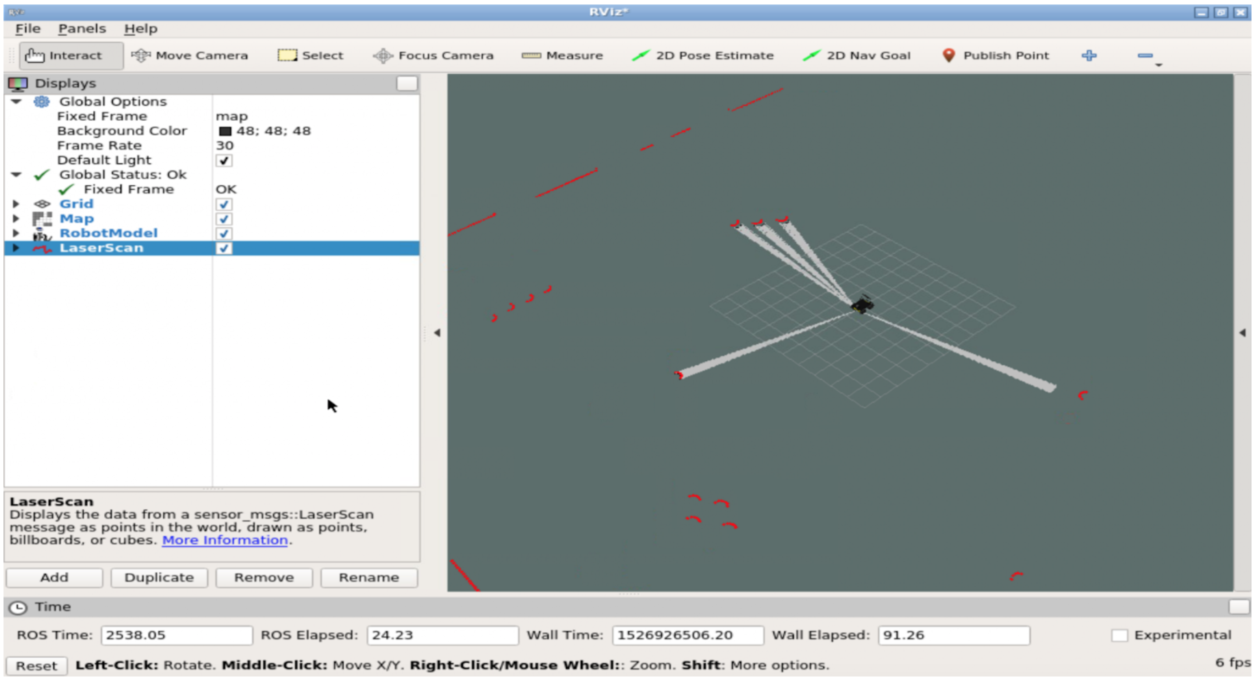
- **maxUrange**: This parameter sets how far your laser will reach to create the map. Greater range will create maps faster and its less probable that the robot gets lost. The downside is that it consumes more resources.

If you need more information about all these parameters, please go to the slam_gmapping node docs: http://wiki.ros.org/gmapping

d) Now, you can proceed to start this launch file.

e) Let's now launch RVIZ to be able to visualize the mapping process. You will need to add the proper elements for visualizing the mapping process (**LaserScan**, **Map**, and **RobotModel**).

You should see something like this:



f) Now, you can start moving the robot around the environment to generate a full map. In order to move the robot with the keyboard, you can use the following command:

In [ ]:

```
roslaunch husky_launch keyboard_teleop.launch
```

g) You can also play with the values in the **maxUrange** parameter, to see how it affects the mapping process.

IMPORTANT: DO NOT CLOSE ANYTHING when you finalize the exercise (i.e. have created the full map). You will have to work with this.

End of Exercise 1.1

Great! So, you have now created a full map of the environment. Now what? Well, now it's time to save this map, so you can use it in the Path Planning system!

Saving the map

Another of the packages available in the ROS Navigation Stack is the map_server package. This package provides the map_saver node, which allows us to access the map data from a ROS Service, and save it into a file.

You can save the built map at anytime by using the following command:

In [ ]:

```
rosrun map_server map_saver -f name_of_map
```

This command will get the map data from the map topic, and write it out into two files: name_of_map.pgm and name_of_map.yaml.

Exercise 1.2

a) Save the map created in the previous exercise into a file.

Execute in WebShell #3

In [ ]:

```
roscd teb_navigation;
mkdir maps;
cd maps;
rosrun map_server map_saver -f my_map;
```

End of Exercise 1.2

You should end up with two new files: **my_map.yaml** and **my_map.pgm**.

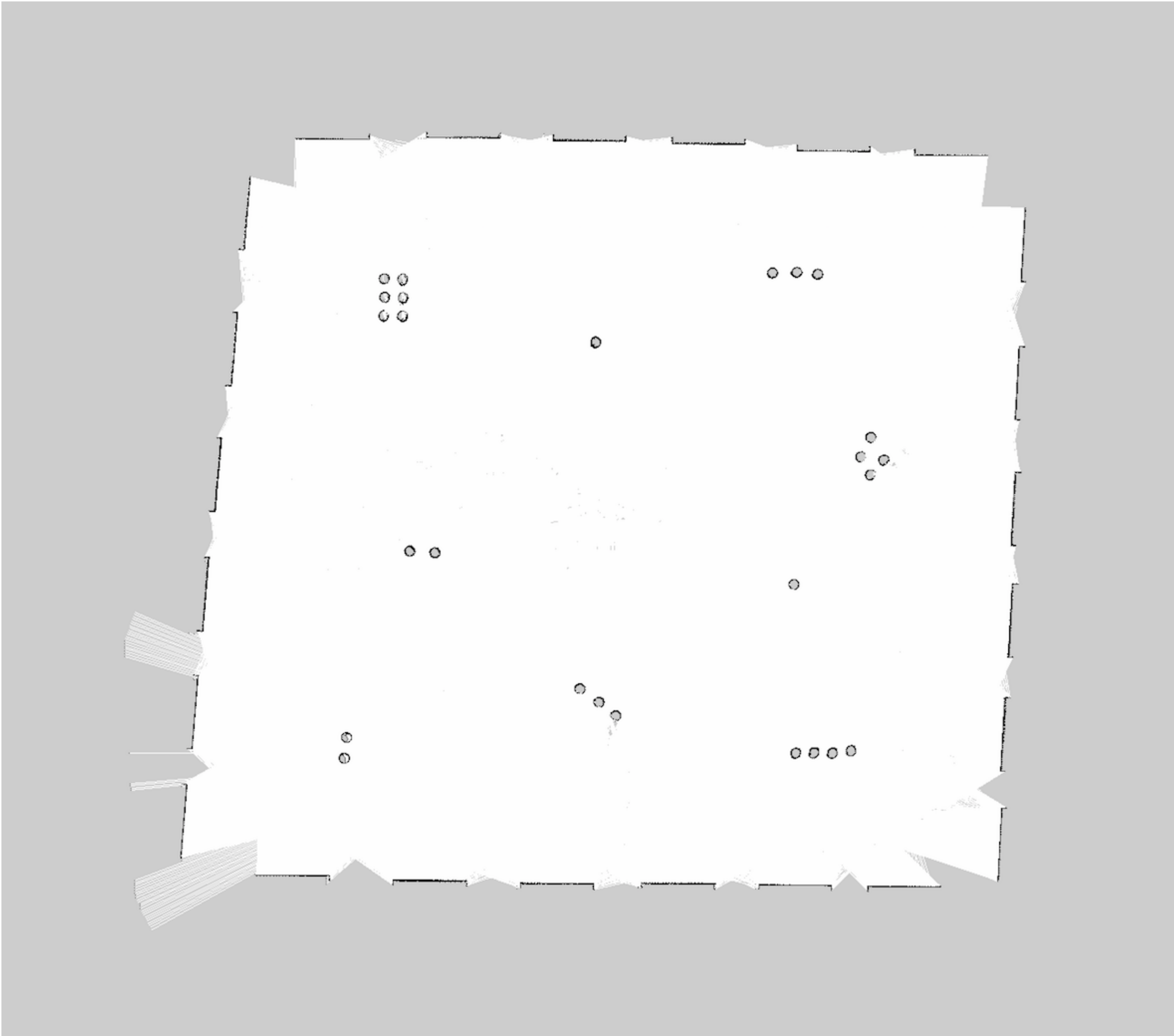
The PGM file is the one that contains the occupancy data of the map (the really important data); and the YAML file contains some metadata about the map, like the map dimensions and resolution, or the path to the PGM file.

my_map.yaml

In [ ]:

```
image: my_map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

my_map.pgm



Set up the AMCL node

So, after generating the map, the next thing we'll need to do is be able to localize the robot into that map. If we don't do this, the map would be totally useless. Right?

For that, we are going to use the **AMCL** node from the Navigation Stack. So, as you did for the mapping process, let's create a launch file to start this node.

Exercise 1.3

a) Inside your package, create a new launch file to start the localization node.

amcl.launch

In [ ]:

```
<?xml version="1.0"?>

<launch>
  <arg name="map_file" default="$(find teb_navigation)/maps/my_map.yaml"/>
```

```xml
 <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />

 <arg name="use_map_topic" default="true"/>
 <arg name="scan_topic" default="scan" />

 <node pkg="amcl" type="amcl" name="amcl">
   <param name="use_map_topic" value="$(arg use_map_topic)"/>
   <!-- Publish scans from best pose at a max of 10 Hz -->
   <param name="odom_model_type" value="diff"/>
   <param name="odom_alpha5" value="0.1"/>
   <param name="gui_publish_rate" value="10.0"/>
   <param name="laser_max_beams" value="60"/>
   <param name="laser_max_range" value="12.0"/>
   <param name="min_particles" value="500"/>
   <param name="max_particles" value="2000"/>
   <param name="kld_err" value="0.05"/>
   <param name="kld_z" value="0.99"/>
   <param name="odom_alpha1" value="0.2"/>
   <param name="odom_alpha2" value="0.2"/>
   <!-- translation std dev, m -->
   <param name="odom_alpha3" value="0.2"/>
   <param name="odom_alpha4" value="0.2"/>
   <param name="laser_z_hit" value="0.5"/>
   <param name="laser_z_short" value="0.05"/>
   <param name="laser_z_max" value="0.05"/>
   <param name="laser_z_rand" value="0.5"/>
   <param name="laser_sigma_hit" value="0.2"/>
   <param name="laser_lambda_short" value="0.1"/>
   <param name="laser_model_type" value="likelihood_field"/>
   <!-- <param name="laser_model_type" value="beam"/> -->
   <param name="laser_likelihood_max_dist" value="2.0"/>
   <param name="update_min_d" value="0.25"/>
   <param name="update_min_a" value="0.2"/>
   <param name="odom_frame_id" value="odom"/>
   <param name="resample_interval" value="1"/>
   <!-- Increase tolerance because the computer can get quite busy -->
   <param name="transform_tolerance" value="1.0"/>
   <param name="recovery_alpha_slow" value="0.0"/>
   <param name="recovery_alpha_fast" value="0.0"/>
   <remap from="scan" to="$(arg scan_topic)"/>
 </node>
```

```
</launch>
```

The most important parameters in these files are:

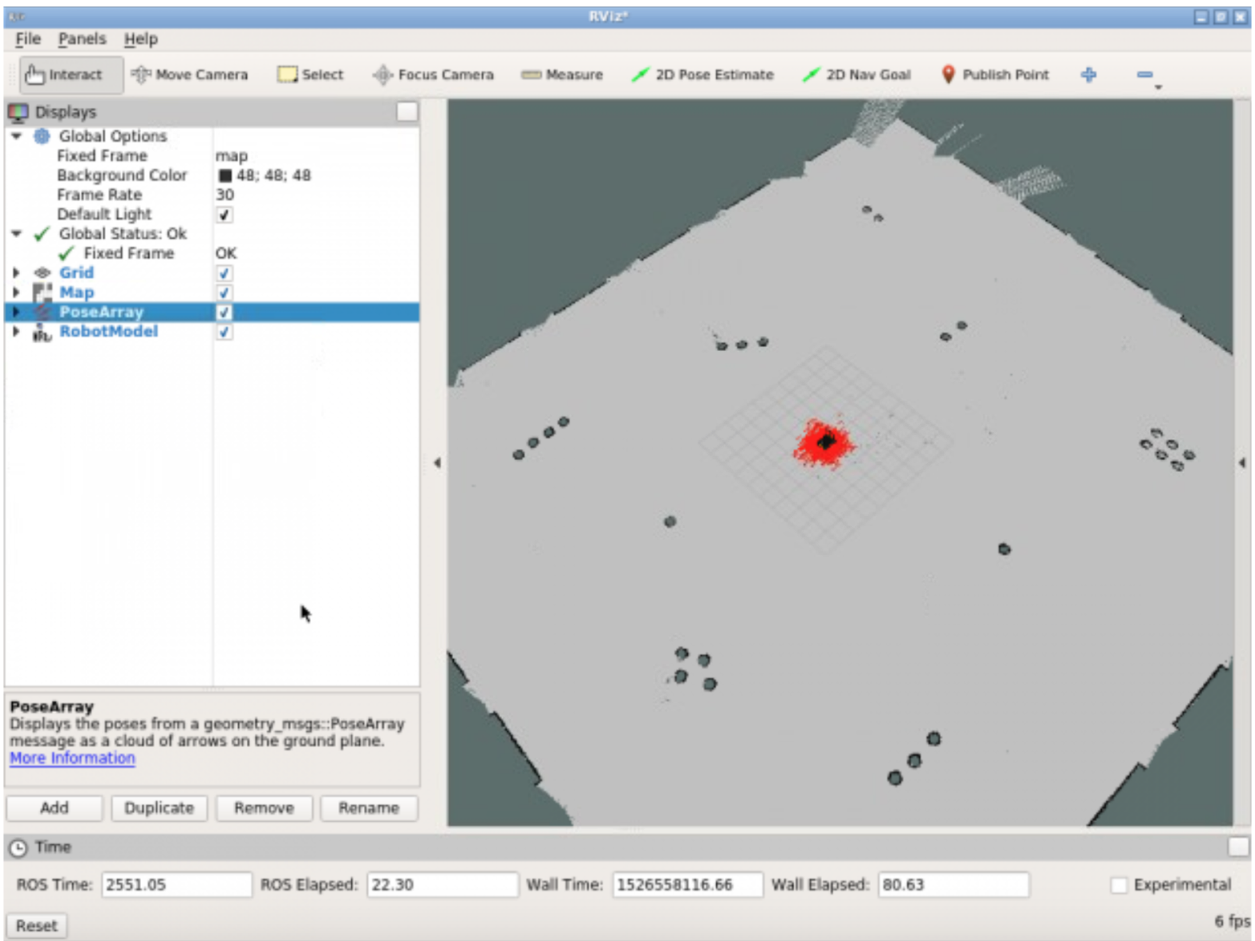
- **min_particles, max_particles**: This parameter sets the number of particles that the filter will use to localize the robot. The more you use, the more precise the localization will be, but the more resources it will consume.
- **laser_max_range**: Maximum range of the laser beams.

If you need more information about all these parameters, please go to the AMCL node docs: http://wiki.ros.org/amcl.

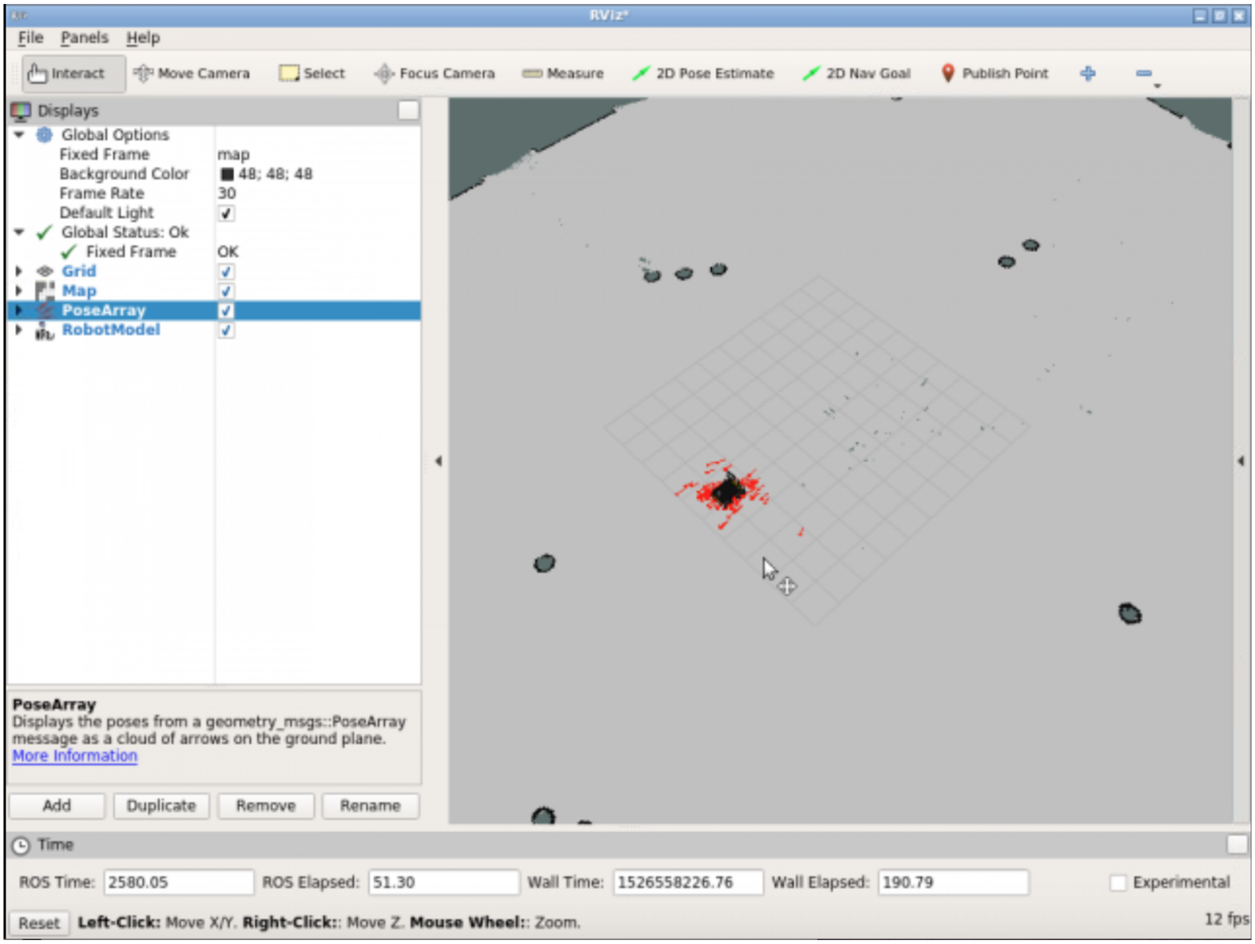
d) Now, you can start this launch file.

e) Let's launch RVIZ in order to be able to visualize the localization process. You can use the same setup you used for the mapping process, adding one more display: **Pose Array**.

You should see something like this:



f) Now, you can start moving the robot around the environment to localize the robot. As you move the robot, you will see in RVIZ how the particles keep getting closer, which means that the estimated poses of the robot are getting closer to the real place. This is a test of how good your localization system is working.



g) You can also play with the values in the **min_particles** and **max_particles** parameters, to see how they affect the localization process.

End of Exercise 1.3

Great! So, at this point, you've already built a map of the environment and you are able to localize the Vehicle on the map. This means that everything is ready to navigate the robot!

Set up the move_base node

For doing Path Planning, you'll need to combine everything you've done up until now. Plus, you will have to use the **move_base** node from the Navigation Stack, which will manage the Path Planning system for you. So, as you've done in the previous exercises, let's create our launch file in order to launch the Path Planning system. This time, though, you'll have some extra work to do, since there are a lot of parameters involved that you'll need to set. But don't worry, you can follow the next exercise and be guided through the process!

Exercise 1.4

a) Inside your package, create a new launch file to start the move_base node.

move_base.launch

In [ ]:

```
<launch>

 <!--- Run AMCL -->
```

```
 <include file="$(find teb_navigation)/launch/amcl.launch" />

 <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">

   <param name="base_global_planner" value="navfn/NavfnROS"/>
   <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS"/>
   <rosparam file="$(find teb_navigation)/config/base_global_planner.yaml"
command="load"/>
   <rosparam file="$(find teb_navigation)/config/base_local_planner.yaml"
command="load"/>

   <!-- observation sources located in costmap_common.yaml -->
   <rosparam file="$(find teb_navigation)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />
   <rosparam file="$(find teb_navigation)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />

   <!-- local costmap, needs size -->
   <rosparam file="$(find teb_navigation)/config/local_costmap_params.yaml"
command="load" ns="local_costmap" />
   <param name="local_costmap/width" value="10.0"/>
   <param name="local_costmap/height" value="10.0"/>

   <!-- static global costmap, static map provides size -->
   <rosparam file="$(find teb_navigation)/config/global_costmap_params.yaml"
command="load" ns="global_costmap"/>

 </node>

</launch>
```

We can see there two main parts:

- **AMCL**: The node that localizes the robot based on laser readings
- **move_base**: The node that will provide Path Planning and Obstacle Avoidance

So, you will just need to create all these parameter files that are being loaded for the move_base node.

b) Create all the parameter files required by the move_base node.

costmap_common_params.yaml

In [ ]:

```
footprint: [[-0.5, -0.33], [-0.5, 0.33], [0.5, 0.33], [0.5, -0.33]]
footprint_padding: 0.01
```

```
robot_base_frame: base_link
update_frequency: 4.0
publish_frequency: 3.0
transform_tolerance: 0.5

resolution: 0.05

obstacle_range: 5.5
raytrace_range: 6.0

#layer definitions
static:
   enable: true
   map_topic: /map
   subscribe_to_updates: true

obstacles_laser:
   enabled: true
   observation_sources: laser
   laser: {data_type: LaserScan, clearing: true, marking: true, topic: /scan,
inf_is_valid: true}

inflation:
   enabled: true
   inflation_radius: 1.0
```

Just note the observation_sources parameters, which use the topic/scan to read laser readings. The camera is not used for navigation. Also note that the ray trace range is only 3.5 meters. This is just to make detections faster and not meant for areas that aren't close enough.

local_costmap_params.yaml

In [ ]:

```
global_frame: odom
robot_base_frame: base_link
rolling_window: true

plugins:
 - {name: obstacles_laser,           type: "costmap_2d::ObstacleLayer"}
 - {name: inflation,                 type: "costmap_2d::InflationLayer"}
```

Note that the static_map parameter is set to False. This is because the local costmap is built from the laser readings, not from any static map.

global_costmap_params.yaml

In [ ]:

```
global_frame: map
robot_base_frame: base_link
rolling_window: false
track_unknown_space: true

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

Note that the static_map parameter here is set to True. This is because the global costmap is built from the static map you created in the previous steps.

base_global_planner.yaml

In [ ]:

```
controller_frequency: 5.0
recovery_behaviour_enabled: true

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

base_local_planner.yaml

In [ ]:

```
TrajectoryPlannerROS:
 # Robot Configuration Parameters
 acc_lim_x: 2.5
 acc_lim_theta:  3.2

 max_vel_x: 1.0
 min_vel_x: 0.0

 max_vel_theta: 1.0
 min_vel_theta: -1.0
 min_in_place_vel_theta: 0.2

 holonomic_robot: false
 escape_vel: -0.1

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

```
 # Forward Simulation Parameters
 sim_time: 2.0
 sim_granularity: 0.02
 angular_sim_granularity: 0.02
 vx_samples: 6
 vtheta_samples: 20
 controller_frequency: 20.0

 # Trajectory scoring parameters
 meter_scoring: true # Whether the gdist_scale and pdist_scale parameters
should assume that goal_distance and path_distance are expressed in units of
meters or cells. Cells are assumed by default (false).
 occdist_scale:  0.1 #The weighting for how much the controller should
attempt to avoid obstacles. default 0.01
 pdist_scale: 0.75  #     The weighting for how much the controller should
stay close to the path it was given . default 0.6
 gdist_scale: 1.0 #     The weighting for how much the controller should
attempt to reach its local goal, also controls speed  default 0.8

 heading_lookahead: 0.325  #How far to look ahead in meters when scoring
different in-place-rotation trajectories
 heading_scoring: false  #Whether to score based on the robot's heading to
the path or its distance from the path. default false
 heading_scoring_timestep: 0.8   #How far to look ahead in time in seconds
along the simulated trajectory when using heading scoring (double, default:
0.8)
 dwa: true #Whether to use the Dynamic Window Approach (DWA)_ or whether to
use Trajectory Rollout
 simple_attractor: false
 publish_cost_grid_pc: true

 # Oscillation Prevention Parameters
 oscillation_reset_dist: 0.25 #How far the robot must travel in meters before
oscillation flags are reset (double, default: 0.05)
 escape_reset_dist: 0.1
 escape_reset_theta: 0.1

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

```
max_vel_x: 2.0
min_vel_x: 0.0
max_vel_y: 0
min_vel_y: 0

max_trans_vel: 2.0
min_trans_vel: 0.1
max_rot_vel: 2.0
min_rot_vel: 0.2

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

# # 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
should stay close to the path it was given
# goal_distance_bias: 24.0 # The weighting for how much the controller
should attempt to reach its local goal, also controls speed
# occdist_scale: 0.01 # The weighting for how much the controller should
attempt to avoid obstacles
# forward_point_distance: 0.325 # The distance from the center point of the
robot to place an additional scoring point, in meters
# stop_time_buffer: 0.2  # The amount of time that the robot must stThe
absolute value of the veolicty at which to start scaling the robot's
footprint, in m/sop before a collision in order for a trajectory to be
considered valid in seconds
# scaling_speed: 0.25 # The absolute value of the velocity at which to start
scaling the robot's footprint, in m/s
# max_scaling_factor: 0.2 # The maximum factor to scale the robot's
footprint by

# # Oscillation Prevention Parameters
```

```
# oscillation_reset_dist: 0.25 #How far the robot must travel in meters before oscillation flags are reset (double, default: 0.05)
```

These are some parameters related to the local planners.

NOTE: If you want more details on how all these parameter files work, you can have a look at the **ROS Navigation in 5 Days Course**.

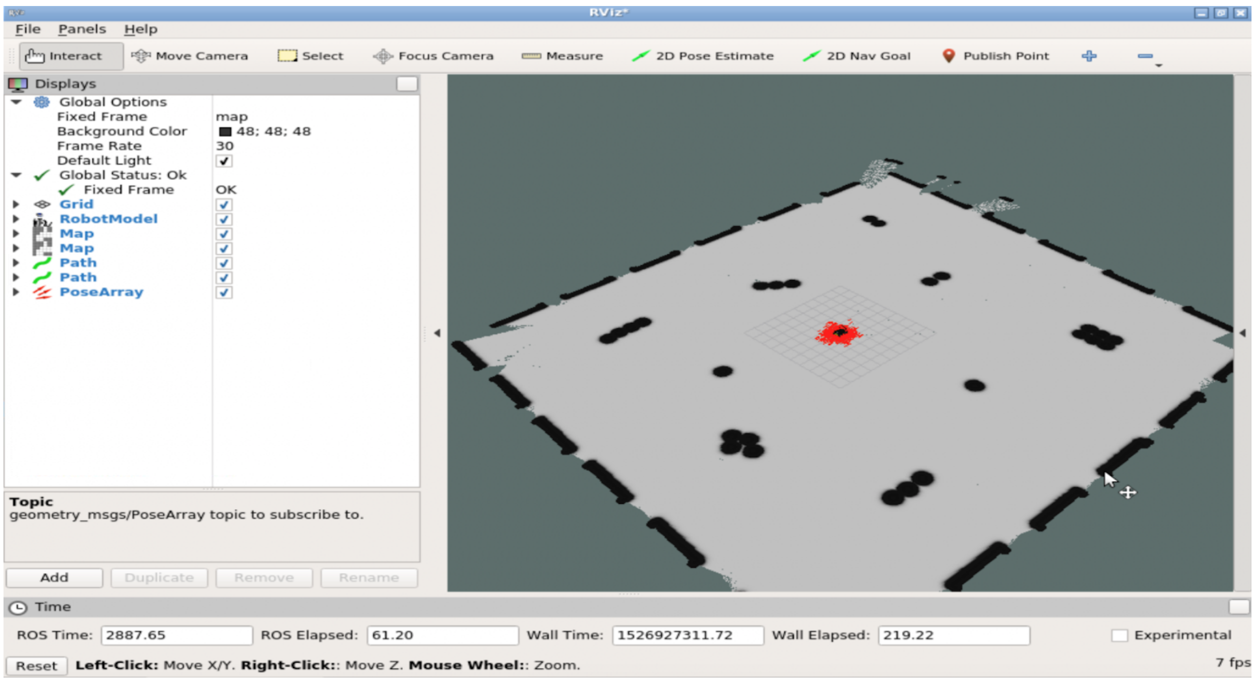
c) Execute your launch file to start the navigation system.

In [ ]:

```
roslaunch teb_navigation move_base.launch
```

d) Let's now launch RVIZ in order to be able to visualize the Path Planning process. For that, you will need to add Displays for the costmaps (local and global) and the plans.

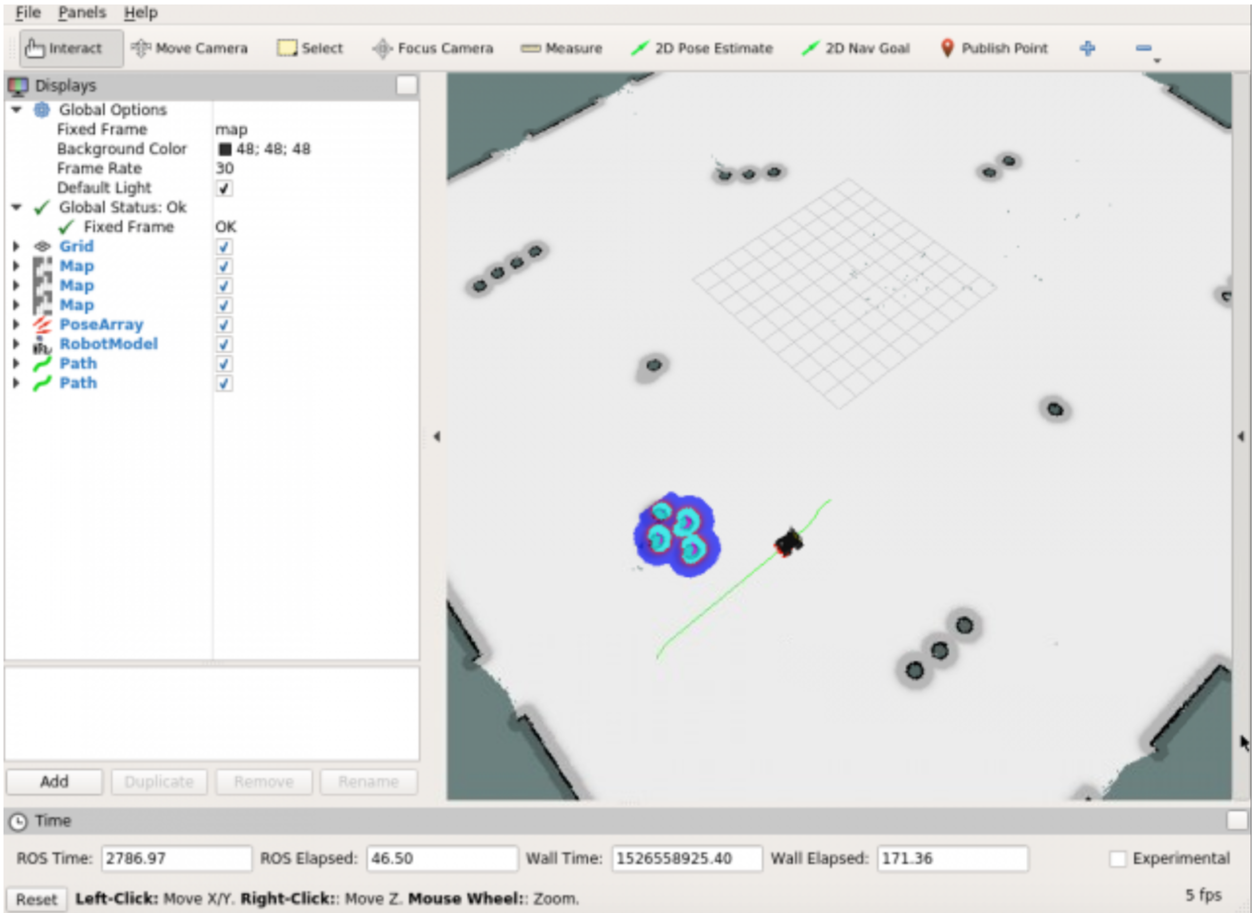
You should see something like this:



e) Use the 2D Pose Estimate tool in RVIZ to localize the robot in the map, just as you did in the demo of the previous unit.

f) Use the 2D Nav Goal tool in RVIZ to send a goal (desired pose) to the robot, just as you did in the demo of the previous unit.

You should now see the Husky robot going to that position in the simulation. In RVIZ, you can also visualize the planned path that it follows.



g) You can also play with some of the values in the parameter files you created previously, to see how they affect the Path Planning process.

End of Exercise 1.4

Set up the TEB Local Planner

So, there you have it. Now, your mobile robot can navigate around one position at a time. But until now, the TEB Local Planner has been nowhere to be seen, right? Let's change that!
Once we have set up the whole Navigation Stack, configuring it to use the TEB Local Planner is quite simple. You can follow the below steps to see how to do it:

Steps to do it:

The first step will be to modify the **move_base.launch** file. Specifically, you will need to modify these two lines:

In [ ]:

```
<param name="base_local_planner"
value="teb_local_planner/TebLocalPlannerROS"/>

<rosparam file="$(find teb_navigation)/config/teb_local_planner.yaml"
command="load"/>
```

As you can see, we have just modified the value of the parameter **base_local_planner** to load **TebLocalPlannerROS**, instead of DWAPlannerROS. And we have also modified the parameters file to load, so that now it will load **teb_local_planner.yaml**. But... we don't have this file yet, right? Well then, let's create it!
Here you have an example of how this file could be created:
teb_local_planner.yaml

In [ ]:

```
TebLocalPlannerROS:

odom_topic: odom

# Trajectory
 teb_autosize: True
dt_ref: 0.3
dt_hysteresis: 0.1
global_plan_overwrite_orientation: True
allow_init_with_backwards_motion: False
max_global_plan_lookahead_dist: 3.0
feasibility_check_no_poses: 5

# Robot

max_vel_x: 2.0
max_vel_x_backwards: 2.2
max_vel_y: 0.0
max_vel_theta: 2.0
acc_lim_x: 2.5
acc_lim_theta: 2.5
min_turning_radius: 0.0 # diff-drive robot (can turn on place!)

footprint_model:
  type: "point"

# GoalTolerance

xy_goal_tolerance: 0.2
yaw_goal_tolerance: 0.1
free_goal_vel: False

# Obstacles

min_obstacle_dist: 0.75 # This value must also include our robot radius,
since footprint_model is set to "point".
```

```
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.5
obstacle_poses_affected: 30
costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5

# Optimization

no_inner_iterations: 5
no_outer_iterations: 4
optimization_activate: True
optimization_verbose: False
penalty_epsilon: 0.1
weight_max_vel_x: 2
weight_max_vel_theta: 1
weight_acc_lim_x: 1
weight_acc_lim_theta: 1
weight_kinematics_nh: 1000
weight_kinematics_forward_drive: 1
weight_kinematics_turning_radius: 1
weight_optimaltime: 1
weight_obstacle: 50
weight_dynamic_obstacle: 10 # not in use yet
weight_adapt_factor: 2

# Homotopy Class Planner

enable_homotopy_class_planning: True
enable_multithreading: True
simple_exploration: False
max_number_classes: 4
selection_cost_hysteresis: 1.0
selection_obst_cost_scale: 1.0
selection_alternative_time_cost: False
roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_keypoint_offset: 0.1
obstacle_heading_threshold: 0.45
visualize_hc_graph: False
```

Alright! Everything's ready now. Pretty simple, right? So, the final step will be to test everything. For that, just launch your **move_base.launch** file again.

Execute in WebShell #1

In [ ]:

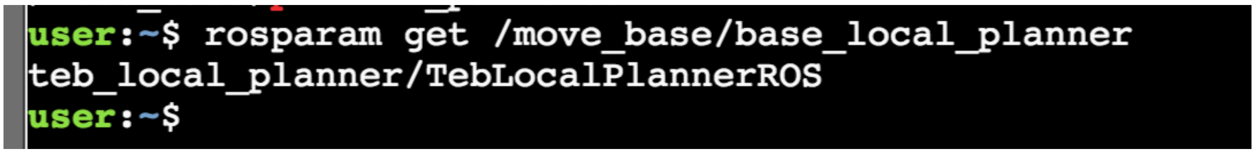
```
roslaunch teb_navigation move_base.launch
```

You can now repeat the process you followed in **Exercise 1.4** to send goals to your repeat. Also, you can check the local planner you are currently using by executing the following command:

In [ ]:

```
rosparam get /move_base/base_local_planner
```

If you've done everything right, you should see something like this:



Congratulations!! You are now capable of creating your own navigation set up with the TEB Local Planner.