robotignite ACADEMY

Unit 2: Configuration and Optimization

SUMMARY

Estimated time to completion: **2 hours**

What will you learn with this unit?

- How to launch the Optimization node
- How to customize the planner parameters
- How to visualize the trajectories in RVIZ

END OF SUMMARY

So, at this point, you have already set up your Navigation System with the TEB Local Planner. But what if your robot needs a different setup? Depending on the robot you are using and the usage you want, the parameters you configure for your planner will vary. Hopefully, the **teb_local_planner** provides a nice and easy-to-use tool that will allow you to test how modifying some parameters will affect the plans generated. So... let's begin!

Configure Optimization

The **teb_local_planner** package comes with a ROS node, which is called **test_optim_node**, which optimizes a trajectory between a fixed start and goal pose. First, we will configure the planning parameters of a single trajectory (Timed-Elastic-Band) between start and goal.

Optimization of a Single Trajectory

Follow the next exercise to see how you can use the **test_optim_node** to optimize your planning trajectories.

Exercise 2.1

a) First of all, you'll need to deactivate the parallel planning using the ROS parameter server. This will allow us to focus on one single trajectory.

Execute in WebShell #1

In [ ]:

```
rosparam set /test_optim_node/enable_homotopy_class_planning False
```

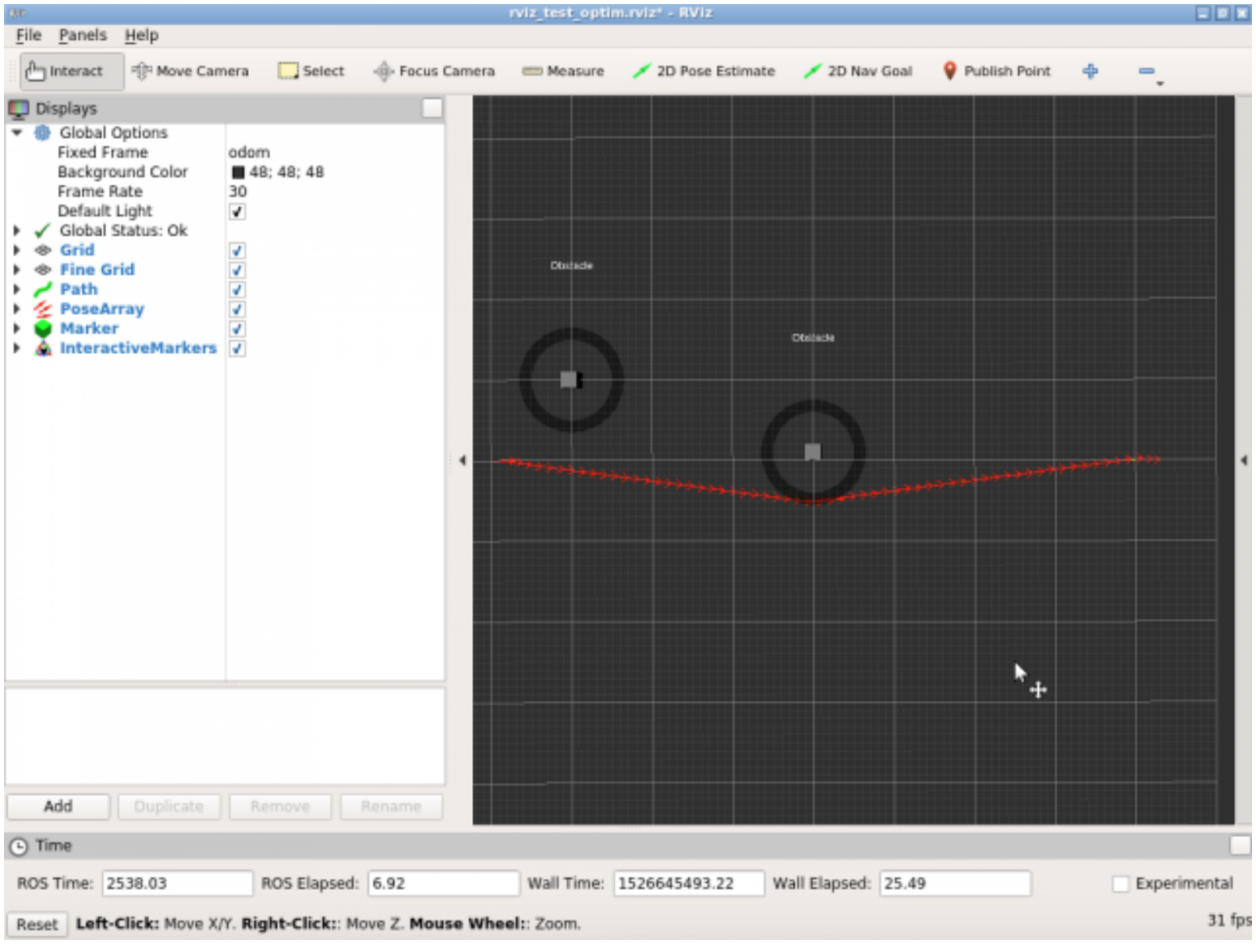
b) Next, let's launch the **test_optim_node** in combination with a preconfigured RVIZ node for visualization:

Execute in WebShell #1

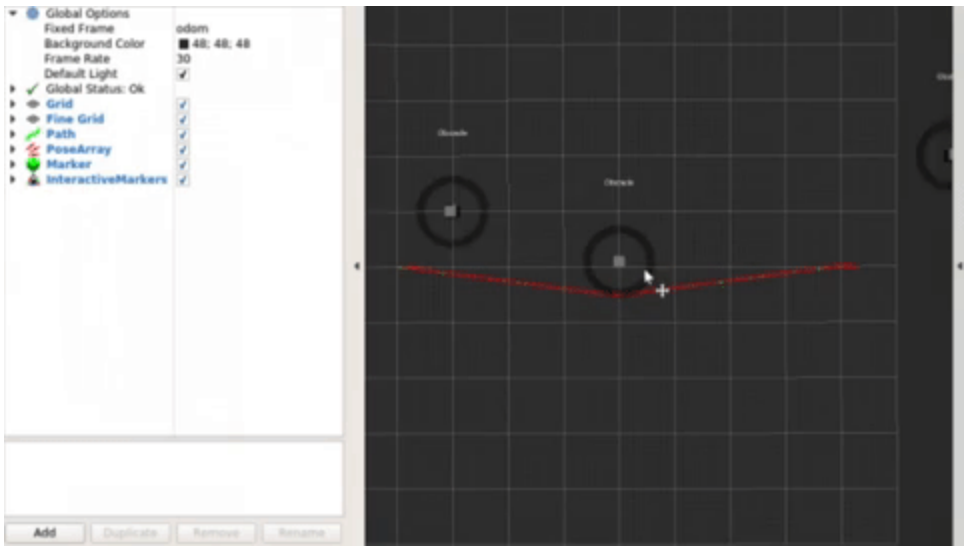
In [ ]:

```
roslaunch teb_local_planner test_optim_node.launch
```

Now, if you open the Graphic Tools by hitting this icon  you will see something like this:



As you can see, three obstacles appear shaped like cubes. They are represented as an interactive_markers type, which means that they can be moved around by clicking and holding the circle that appears around each obstacle. Have a look at the following gif:



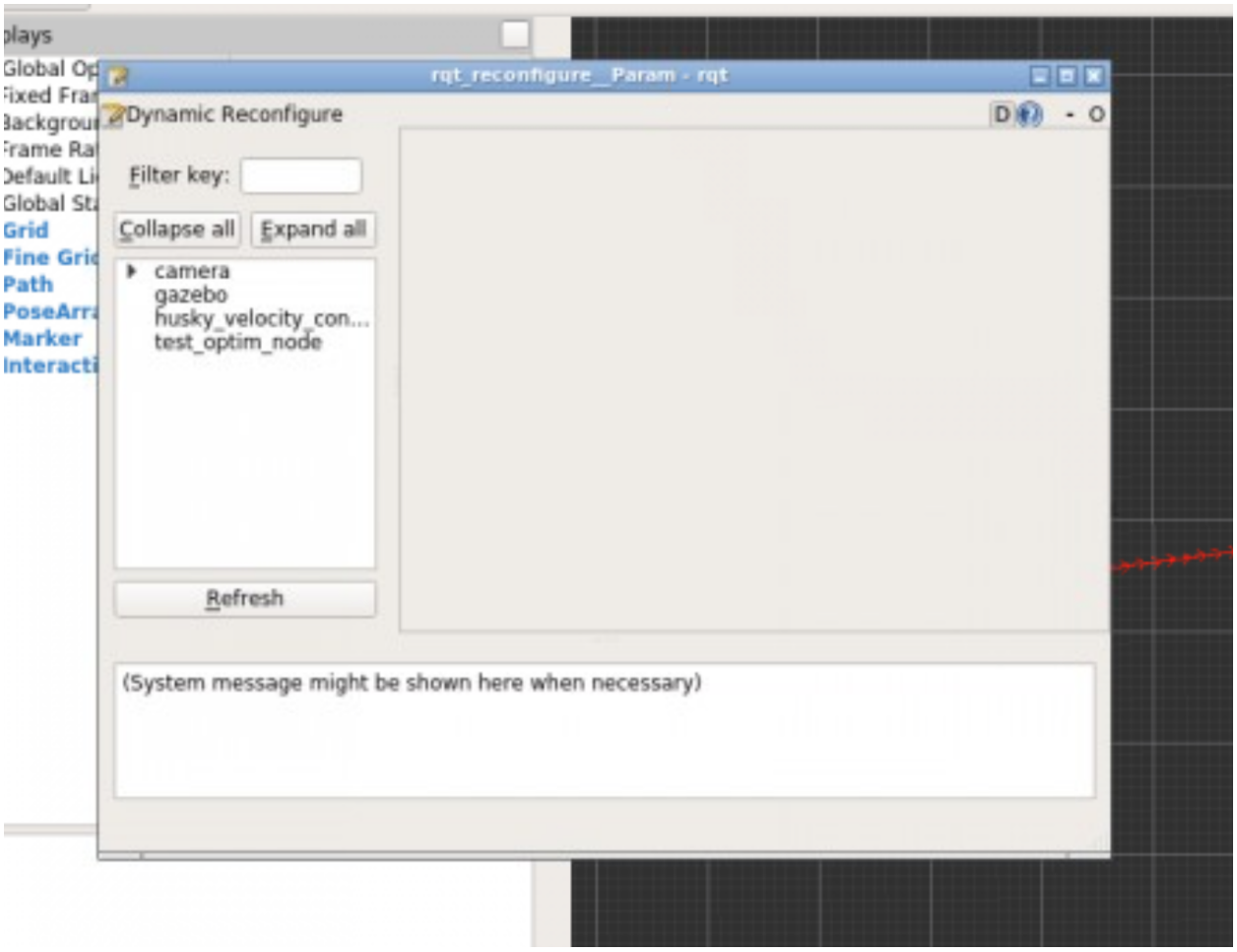
Move the obstacles around and see how they affect the trajectory calculated by the local planner.
c) Now, let's try to optimize these trajectories by modifying the local planner parameters. For that, execute the following command:

Execute in WebShell #2

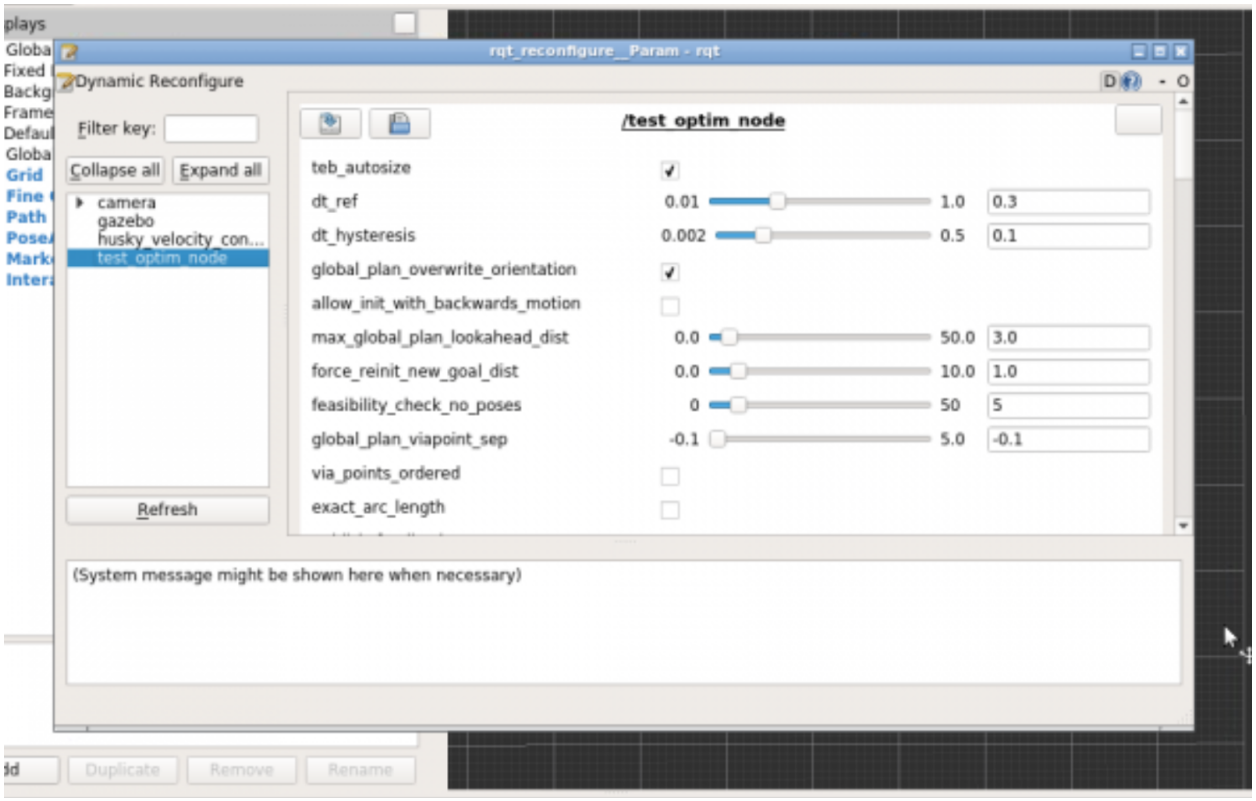
In [ ]:

```
rosrun rqt_reconfigure rqt_reconfigure
```

After a few seconds, a new screen, like this one, should appear:



Now, just click on the **test_optim_node** on the left menu, and the parameters will appear in the center of the screen. Try to modify some of the parameters and see how they affect the trajectory that is being calculated by the local planner.



NOTE: Modify the parameters only slightly, since some parameter sets could lead to unwanted convergence behavior or a bad performance (especially by changing the optimization parameters).
NOTE 2: You can change between screens (RVIZ and rqt) by using the bottom menu.

END of Exercise 2.1

Optimization of Multiple Trajectories in Distinctive Topologies

Great! So now you've seen how you can test and optimize the setup for a single trajectory. But let me tell you that this is not the usual behavior of a TEB Local Planner. By default, parallel planning is enabled, which provides better results, but also requires more computational resources.
Let's see how we can test the parallel planning in the following exercise!

Exercise 2.2

a) First of all, let's enable the extended planner:

Execute in WebShell #1

In [ ]:

```
rosparam set /test_optim_node/enable_homotopy_class_planning True
```

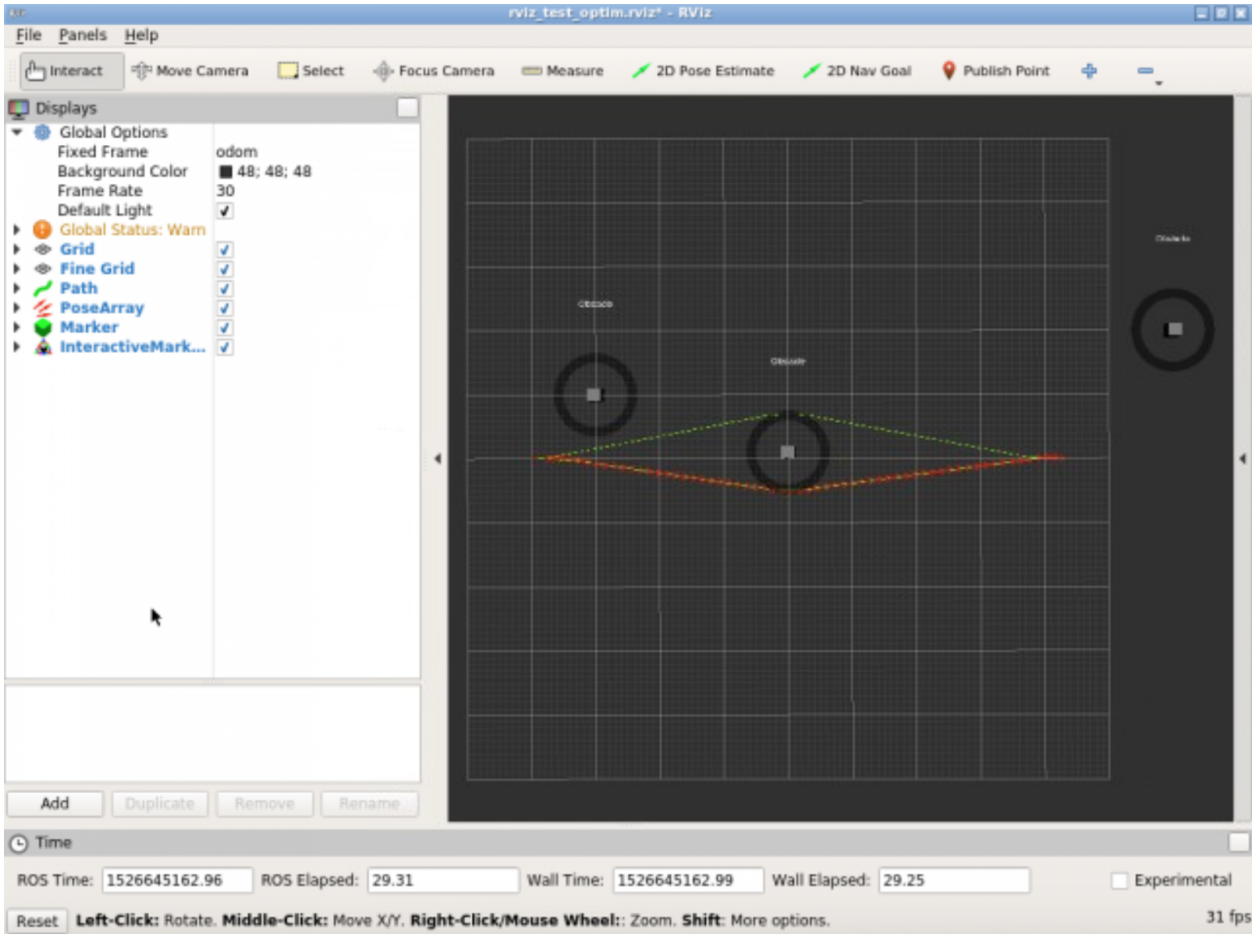
b) Now, let's launch the **test_optim_node** in combination with a preconfigured RVIZ node for visualization:

Execute in WebShell #1

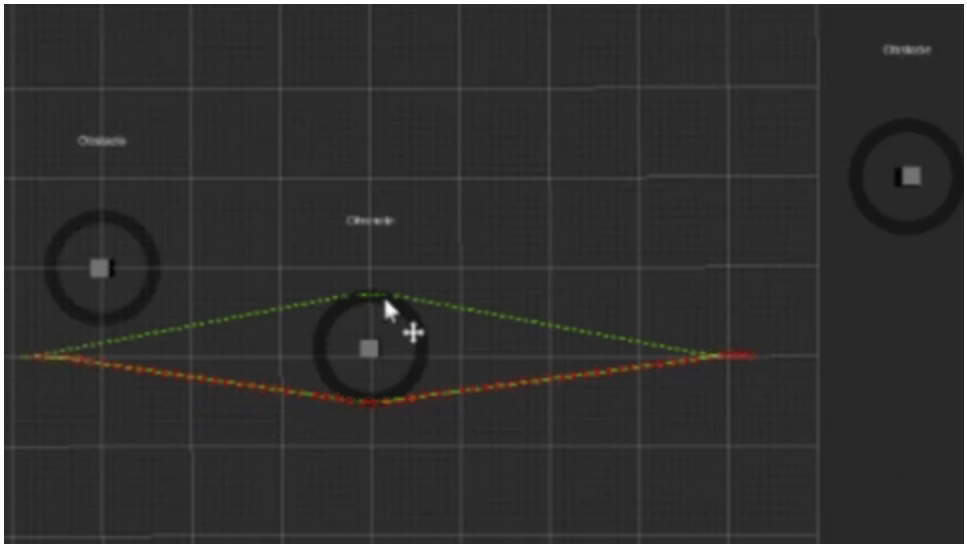
In [ ]:

```
roslaunch teb_local_planner test_optim_node.launch
```

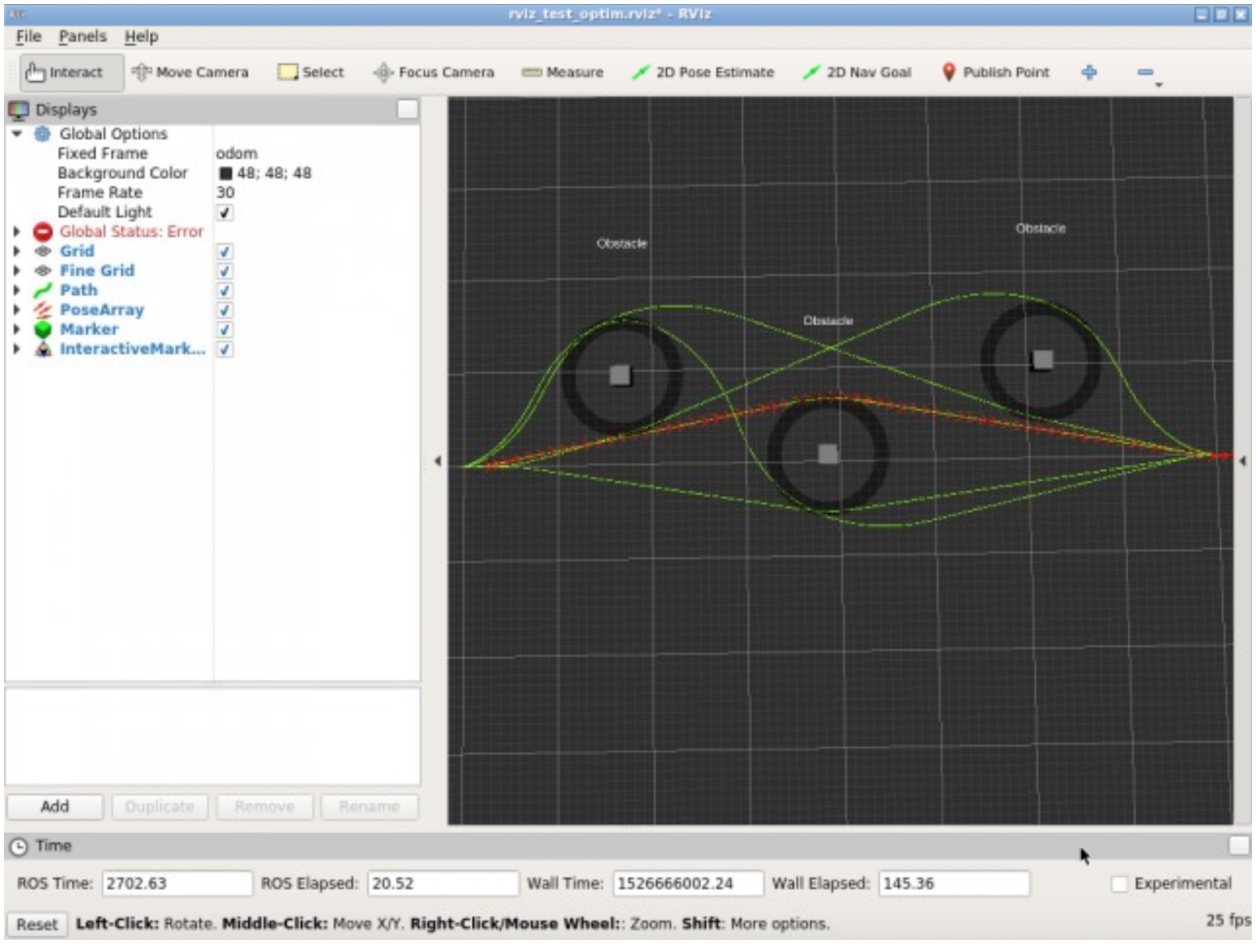
Now, if you open the Graphic Tools by hitting this icon  you will see something like this:



Like in the previous exercise, you can move all the obstacles around by holding the circle that appears around them.



Change the obstacle configuration and observe what's happening. You see many plans at the same time, right? Something like this:



What's actually happening is that the current best trajectory (in the sense of the cheapest optimization cost) is highlighted by showing the individual poses (as red arrows) at each trajectory configuration.

c) Let's now launch the rqt_reconfigure node again to customize the parameters.

Execute in WebShell #2

In [ ]:

```
rosrun rqt_reconfigure rqt_reconfigure
```

Try to modify some of the parameters and see how they affect the trajectory that is being calculated by the local planner.

END of Exercise 2.2

How obstacle avoidance works

Obstacle avoidance is achieved as a part of the optimization process. Basically, the optimization process aims for the trajectory with the **minimum cost** among all the trajectories that are calculated. For instance, if one of the trajectories violates a desired distance from obstacles, the cost of that trajectory will increase, so it will likely be discarded at the end. The allowed minimum distance to an obstacle can be configured in the parameter **min_obstacl_dist**. So, let's say that we set this value

to 0.2 meters; then, if a trajectory has a distance to an obstacle that is below 0.2 meters, the cost of this trajectory will increase.

But there are many more penalty (cost) terms. Therefore, the optimizer may have a small violation (increase of the cost) in order to achieve an overall better cost for the trajectory. You can adjust the optimization weights (scaling of the individual costs) on the parameter **weight_obstacle**. But be careful. If you set the values too high here, it may lead to poor behavior.

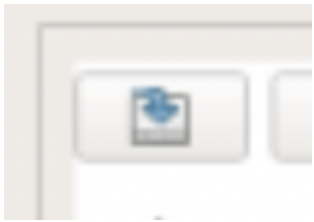
You can also shift penalty terms. For instance, by adding a small extra margin to the min_obstacle_dist parameter, you implicitly increase the cost value at 0.2 meters. There's also one parameter to shift all penalty terms at once: it is done with the **penalty_epsilon** parameter. But be careful here, since this parameter will have a huge impact on the optimization results.
A feasibility check is performed after the optimizer returns a trajectory and before the velocity commands are sent to the robot. The purpose of this check is to identify an invalid/infeasible trajectory that might be produced by the optimizer.
Currently, the algorithm iterates the first n poses (n = **feasibility_check_no_poses** parameter), starting from the current robot pose, and checks whether those poses are collision-free. For detecting if a collision occurs, the **costmap footprint** is used (not the footprint mode, which you'll see in the next unit).

The value of **feasibility_check_no_poses** should not be too high, since the optimizer might not be converged completely. Figuratively, small obstacle violations in the (far) future could be corrected while the robot is moving towards the goal.
If you are driving in narrow environments, make sure to configure the obstacle avoidance behavior (local planner and global planner) properly. Otherwise, the local planner might reject an infeasible trajectory (from its point of view), but the global planner, in contrast, could think that the selected (global) plan is feasible; the robot could get stuck.

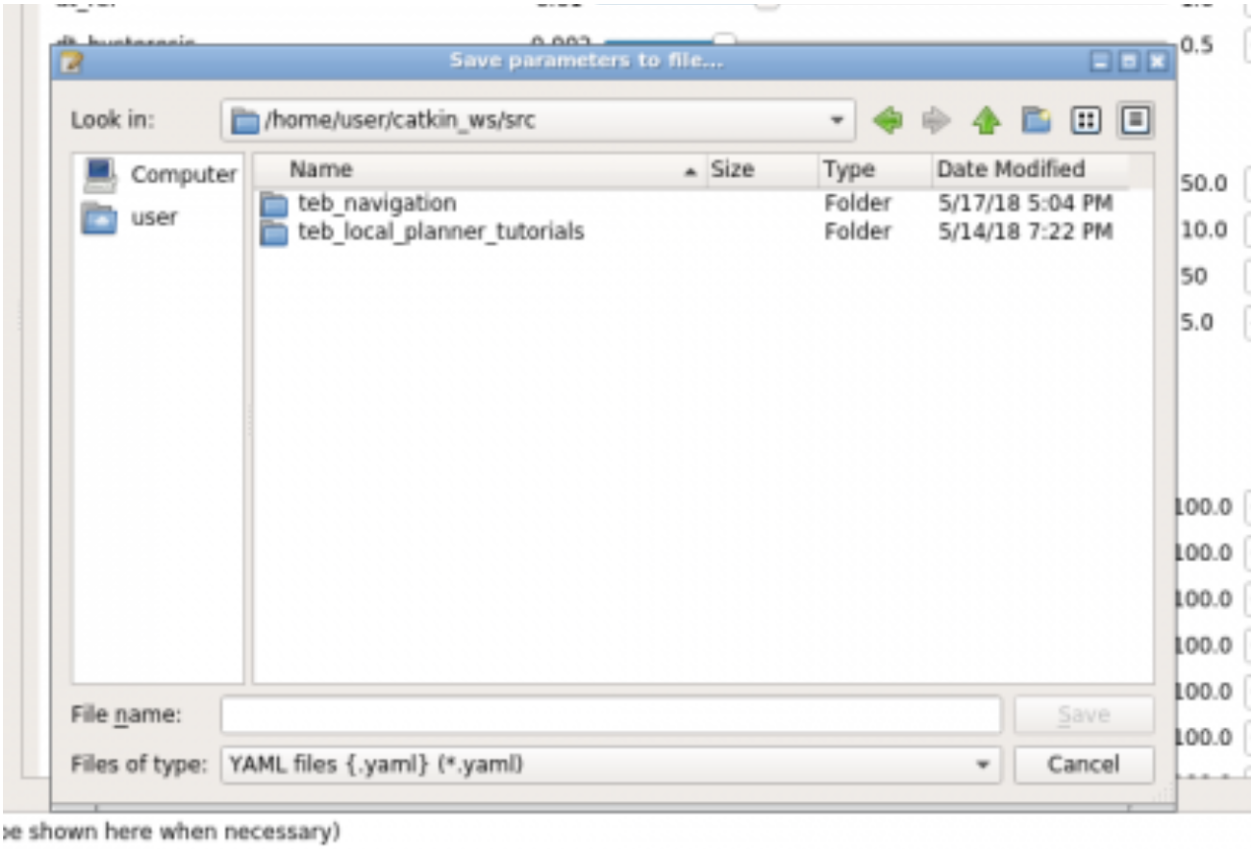
Save the parameters

You may have noticed that when you close the **rqt_reconfigure** node, all the modifications you've made to the parameters are lost. When you launch the node back again later, the parameters will be set to their defaults, losing all the work you've done.

Thankfully, rqt_reconfigure allows you to save your parameters to a file very easily. You just have to click on the Save button that appears right above the parameters...



...and select the folder where you want to save them.



ɔe shown here when necessary)

NOTE: We recommend that you save your parameter files somewhere inside **catkin_ws/src**.

Inspect optimization feedback

Great! So, now you've learned how to launch the **test_optim_node** to optimize your plans by customizing the **teb_local_planner** parameters. Also, you've seen how to visualize the resulting trajectories in RVIZ, which makes the whole process (which is quite complicated) much more easy to understand.

However, these trajectories you were visualizing in RVIZ didn't contain any temporal information. They were only trajectories represented into the space. And, as you may have already seen, you had some parameters that were related to the velocities of the robot. So, we will need to add some temporal information to our trajectories, since velocity depends on time. For this, the teb_local_planner package provides a feedback message, **teb_local_planner/FeedbackMsg**, which contains all this information. Furthermore, the feedback message contains all the alternative trajectories, and the currently selected one is stored in an index.

This feedback message can be published into a topic named **/test_optim_node/teb_feedback**. This feedback message, though, is disabled by default in order to reduce the computational resources used, but can be easily enabled.

Visualize the velocity profile

In the following exercise, you are going to create a small Python script that subscribes to the /test_optim_node/teb_feedback topics, and generates a plot with the velocity information associated with the trajectories.

Exercise 2.3

a) Inside your navigation package, create a new Python script named **visualize_velocities.py**. Copy the following code into it:

visualize_velocities.py

In [ ]:

```
#!/usr/bin/env python
import rospy, math
from teb_local_planner.msg import FeedbackMsg, TrajectoryMsg,
TrajectoryPointMsg
from geometry_msgs.msg import PolygonStamped, Point32
import numpy as np
import matplotlib.pyplot as plotter

def feedback_callback(data):
 global trajectory

 if not data.trajectories: # empty
   trajectory = []
   return
 trajectory = data.trajectories[data.selected_trajectory_idx].trajectory


def plot_velocity_profile(fig, ax_v, ax_omega, t, v, omega):
 ax_v.cla()
 ax_v.grid()
 ax_v.set_ylabel('Trans. velocity [m/s]')
 ax_v.plot(t, v, '-bx')
 ax_omega.cla()
 ax_omega.grid()
 ax_omega.set_ylabel('Rot. velocity [rad/s]')
 ax_omega.set_xlabel('Time [s]')
 ax_omega.plot(t, omega, '-bx')
 fig.canvas.draw()


def velocity_plotter():
 global trajectory
 rospy.init_node("visualize_velocity_profile", anonymous=True)

 topic_name = "/test_optim_node/teb_feedback" # define feedback topic here!
```

```
 rospy.Subscriber(topic_name, FeedbackMsg, feedback_callback, queue_size = 1)

 rospy.loginfo("Visualizing velocity profile published on '%s'.",topic_name)
 rospy.loginfo("Make sure to enable rosparam 'publish_feedback' in the
teb_local_planner.")

 # two subplots sharing the same t axis
 fig, (ax_v, ax_omega) = plotter.subplots(2, sharex=True)
 plotter.ion()
 plotter.show()


 r = rospy.Rate(2) # define rate here
 while not rospy.is_shutdown():

   t = []
   v = []
   omega = []

   for point in trajectory:
     t.append(point.time_from_start.to_sec())
     v.append(point.velocity.linear.x)
     omega.append(point.velocity.angular.z)

   plot_velocity_profile(fig, ax_v, ax_omega, np.asarray(t), np.asarray(v),
np.asarray(omega))

   r.sleep()

if __name__ == '__main__':
 try:
   trajectory = []
   velocity_plotter()
 except rospy.ROSInterruptException:
   pass
```

b) Next, we'll enable the publishing of the feedback message, then we'll start the test_optim_node, and finally, we'll run our Python script.

Execute in WebShell #1

In [ ]:

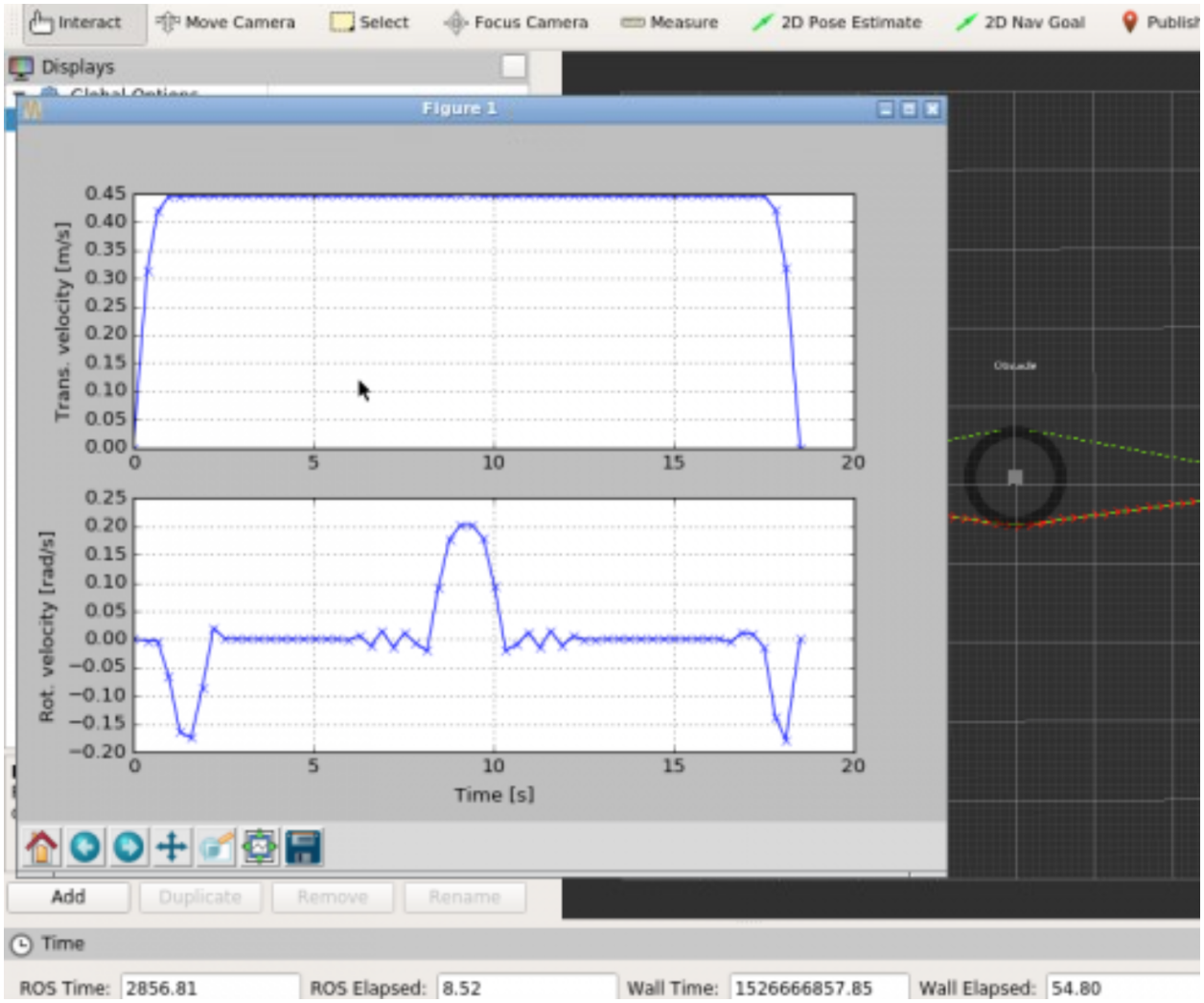
```
rosparam set /test_optim_node/publish_feedback true
roslaunch teb_local_planner test_optim_node.launch
```

Execute in WebShell #2

In [ ]:

```
rosrun teb_local_planner_tutorials visualize_velocity_profile.py # or call your own script here
```

If everything goes OK, you should see a plot with the velocities associated with the trajectory now.



END of Exercise 2.3

Congratulations!! You are now capable of optimizing your trajectories!