# ROS2 Navigation

## Extra:  New Nav2 Features

Nav2 provides new features and tools that make creating robot applications easier.

In this unit, you will review the essential new features introduced in Nav2, which include:

- Basic Nav2 operations through the **Simple Commander API**
- Use of the Waypoint Follower and Task Executor plugins via **FollowWaypoints**
- Introduction to Keepout Zones and speed-restricted zones

You will then create a basic autonomous robotics demo based on Nav2. You will do this in a simulated warehouse where robots are often deployed:

You will use the AWS Small Warehouse World (https://github.com/aws-robotics/aws-robomaker-small-warehouse-world) and Neobotix's MP-400 (https://www.neobotix-robots.com/products/mobile-robots/mobile-robot-mp-400) mobile industrial robot.

# 1 Get Started!

Start by testing that you can navigate in this warehouse environment.

- Setup -

For this unit, you need to have a Nav2 system working correctly. You can download the required Nav2 packages from the following repo:

▶ Execute in Shell

In [ ]:
```
cd ~/ros2_ws/src
git clone https://bitbucket.org/theconstructcore/nav2_pkgs.git
```

After downloading the repo, make sure to compile your workspace:

▶ Execute in Shell

In [ ]:
```
cd ~/ros2_ws/
colcon build
source install/setup.bash
```

Now you are ready to start navigation!

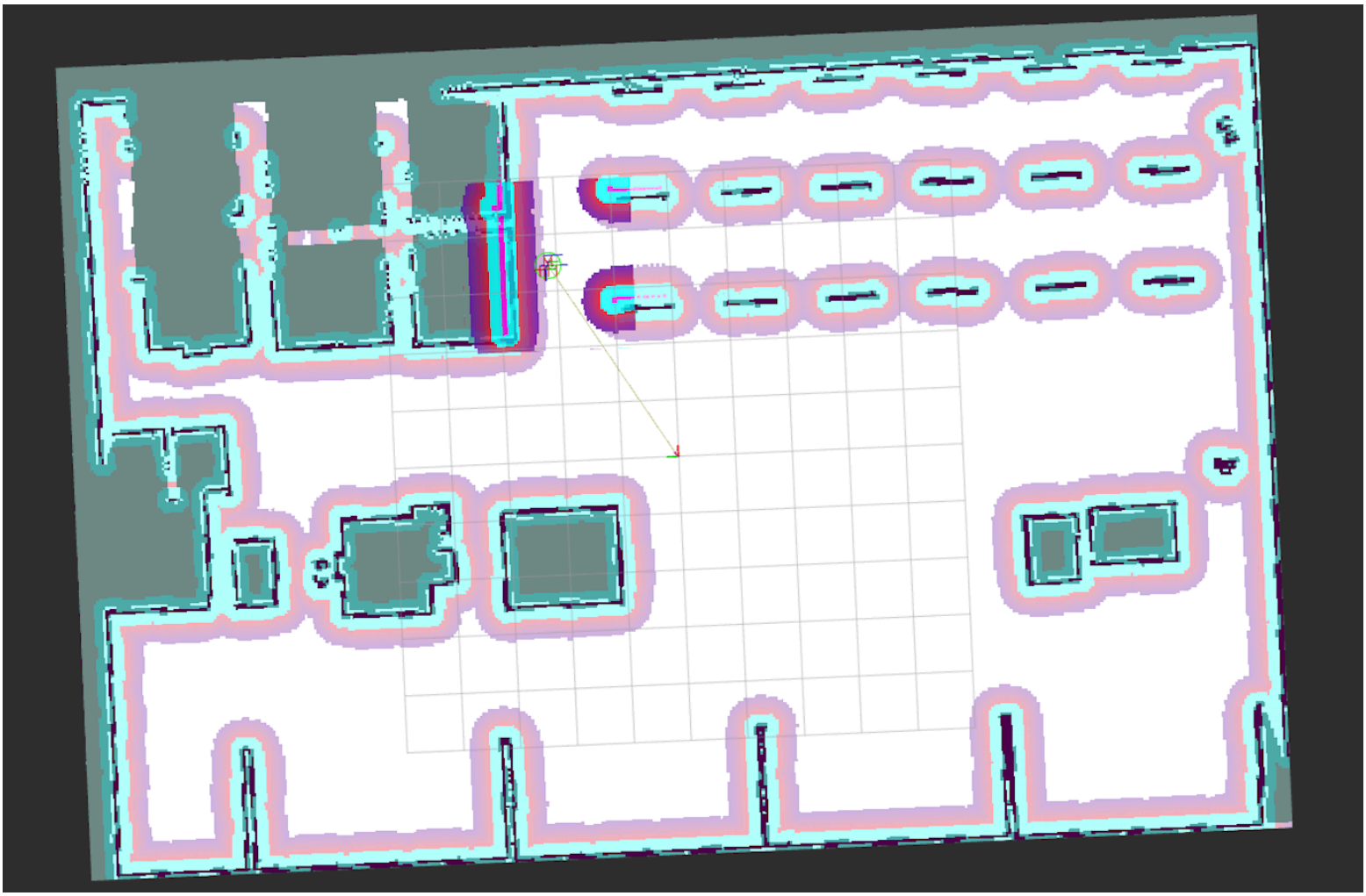
▶ Execute in Shell

In [ ]:
```
ros2 launch path_planner_server navigation.launch.py
```

- End Setup -

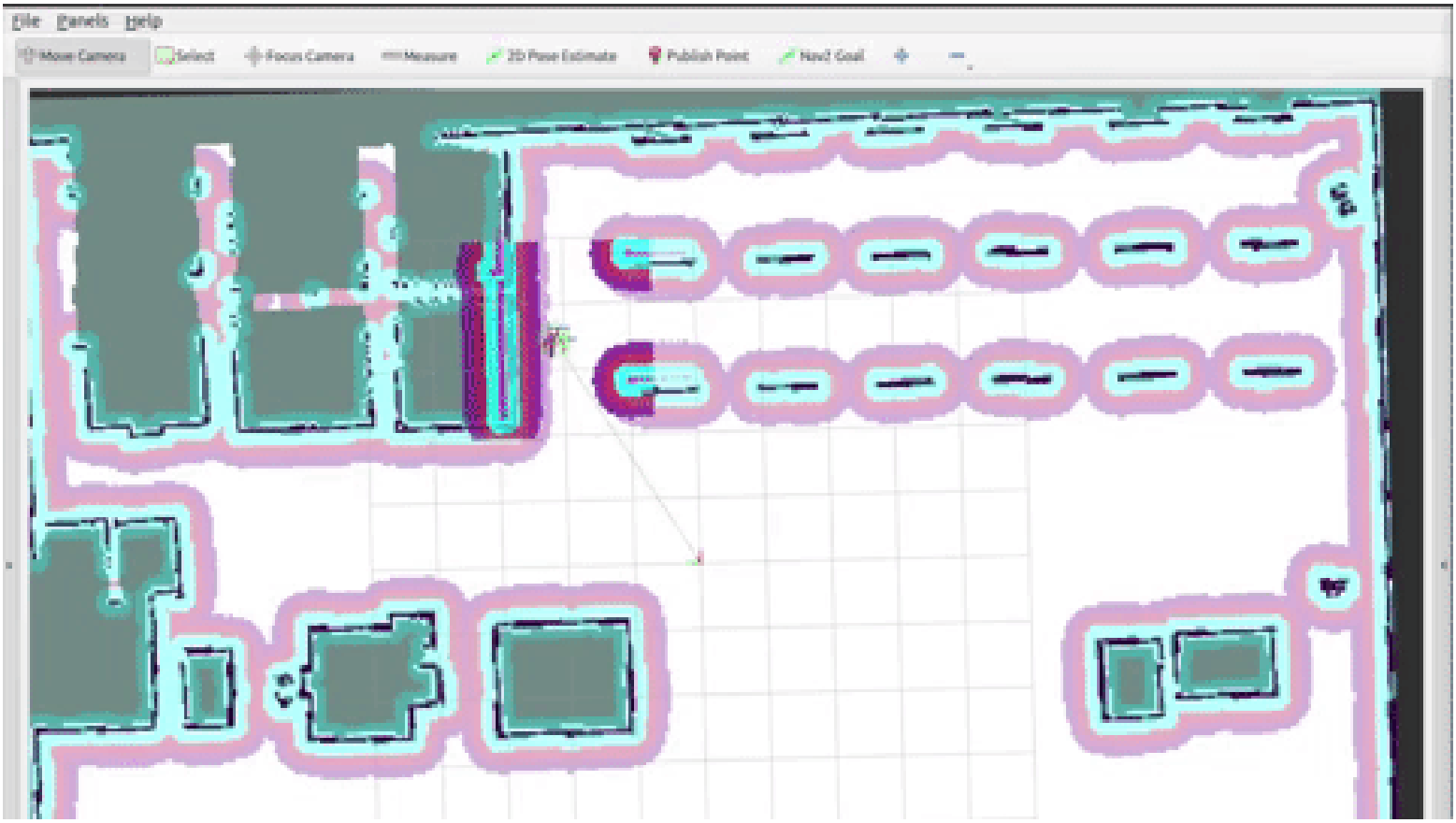
Set the robot's initial position using the **`2D Pose Estimation`** tool. You can also do this programmatically for a real system which you can mock up via (in a new terminal):

```
ros2 topic pub /initialpose geometry_msgs/msg/PoseWithCovarianceStamped "{header: {st
amp: {sec: 0}, frame_id: 'map'}, pose: {pose: {position: {x: 3.45, y: 2.15, z: 0.0},
orientation: {z: 1.0, w: 0.0}}}}"
```



You should now see the robot initialized at its position and the Costmaps updated in RVIZ2.

Now, use the **`Nav2 Goal`** button to request the robot to go to a specific goal to test it out!




**Now you are navigating!**

Play around and get a feel for it. If you find yourself in a bad state at any time during this tutorial, feel free to close out the application and relaunch it with the instructions above.

---

## 2 Simple Commander API

The **Nav2 Simple Commander** provides a set of methods to interact through Python3 code with the Nav2 system. So, you can think of it as a Python API that allows you to interact easily with the navigation stack.

In this unit, you will review important methods for creating an application for a navigation robot. This includes: **goToPose()**, **goThroughPoses()** and **followWaypoints()**.

- Setup -

For this unit you must install the Nav2 Simple Commander module manually. You can do it with the following commands:

▶ Execute in Shell

In [ ]:
```
sudo apt update
sudo apt install ros-galactic-nav2-simple-commander
```

- End Setup -

## 2.1 Navigate To Pose

The **NavigateToPose** action is most suitable for point-to-point navigation requests or other tasks that can be represented in a behavior tree with a boundary condition pose, such as dynamic object following.

Find below the structure of the NavigateToPose action.

NavigateToPose.action

```
#goal definition
geometry_msgs/PoseStamped pose
string behavior_tree
---
#result definition
std_msgs/Empty result
---
# feedback definition
geometry_msgs/PoseStamped current_pose
builtin_interfaces/Duration navigation_time
builtin_interfaces/Duration estimated_time_remaining
int16 number_of_recoveries
float32 distance_remaining
```

As you can see, the action's primary inputs are the **`pose`** you would like the robot to navigate to and the (optional) **`behavior_tree`** to use. It uses the default behavior tree in the BT Navigator if none is specified. During the action's execution, you get feedback with essential information like the robot's pose, how much time has elapsed, the estimated time remaining, the distance remaining, and the number of recoveries executed while navigating to the goal. This information can be used to make good autonomy decisions or track progress.

### 2.1.1 Demo

In the following demo, you use the `NavigateToPose` action to have your robot drive from its staging point to a shelf for a human to place an item on the robot. Then you will drive to the pallet jack for shipping on the next truck out of the warehouse.

Start by creating a new package named **nav2_new_features**, where you place all the scripts created in this unit.

▶ Execute in Shell

In [ ]:
```
ros2 pkg create --build-type ament_python nav2_new_features --dependencies rc
```

Now, inside this package, create a new folder named **scripts** and add a Python script named **navigate_to_pose.py**:

navigate_to_pose.py

In [ ]:

```
#! /usr/bin/env python3
# Copyright 2021 Samsung Research America
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import time
from copy import deepcopy

from geometry_msgs.msg import PoseStamped
from rclpy.duration import Duration
import rclpy

from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult

# Shelf positions for picking
shelf_positions = {
    "shelf_A": [-3.829, -7.604],
    "shelf_B": [-3.791, -3.287],
    "shelf_C": [-3.791, 1.254],
    "shelf_D": [-3.24, 5.861]}

# Shipping destination for picked products
shipping_destinations = {
    "recycling": [-0.205, 7.403],
    "pallet_jack7": [-0.073, -8.497],
    "conveyer_432": [6.217, 2.153],
    "frieght_bay_3": [-6.349, 9.147]}

'''
Basic item picking demo. In this demonstration, the expectation
is that a person is waiting at the item shelf to put the item on the robot
and at the pallet jack to remove it
(probably with a button for 'got item, robot go do next task').
'''



def main():
    # Recieved virtual request for picking item at Shelf A and bringing to
    # worker at the pallet jack 7 for shipping. This request would
```

```
# contain the shelf ID ("shelf_A") and shipping destination ("pallet_jack7"
####################
request_item_location = 'shelf_C'
request_destination = 'pallet_jack7'
####################

rclpy.init()

navigator = BasicNavigator()

# Set your demo's initial pose
initial_pose = PoseStamped()
initial_pose.header.frame_id = 'map'
initial_pose.header.stamp = navigator.get_clock().now().to_msg()
initial_pose.pose.position.x = 3.45
initial_pose.pose.position.y = 2.15
initial_pose.pose.orientation.z = 1.0
initial_pose.pose.orientation.w = 0.0
navigator.setInitialPose(initial_pose)

# Wait for navigation to activate fully
navigator.waitUntilNav2Active()

shelf_item_pose = PoseStamped()
shelf_item_pose.header.frame_id = 'map'
shelf_item_pose.header.stamp = navigator.get_clock().now().to_msg()
shelf_item_pose.pose.position.x = shelf_positions[request_item_location][0]
shelf_item_pose.pose.position.y = shelf_positions[request_item_location][1]
shelf_item_pose.pose.orientation.z = 1.0
shelf_item_pose.pose.orientation.w = 0.0
print('Received request for item picking at ' + request_item_location + '.'
navigator.goToPose(shelf_item_pose)

# Do something during your route
# (e.x. queue up future tasks or detect person for fine-tuned positioning)
# Print information for workers on the robot's ETA for the demonstration
i = 0
while not navigator.isTaskComplete():
    i = i + 1
    feedback = navigator.getFeedback()
    if feedback and i % 5 == 0:
        print('Estimated time of arrival at ' + request_item_location +
              ' for worker: ' + '{0:.0f}'.format(
                  Duration.from_msg(feedback.estimated_time_remaining).nanc
              + ' seconds.')

result = navigator.getResult()
if result == TaskResult.SUCCEEDED:
    print('Got product from ' + request_item_location +
          '! Bringing product to shipping destination (' + request_destinat
```

```
        shipping_destination = PoseStamped()
        shipping_destination.header.frame_id = 'map'
        shipping_destination.header.stamp = navigator.get_clock().now().to_msg(
        shipping_destination.pose.position.x = shipping_destinations[request_de
        shipping_destination.pose.position.y = shipping_destinations[request_de
        shipping_destination.pose.orientation.z = 1.0
        shipping_destination.pose.orientation.w = 0.0
        navigator.goToPose(shipping_destination)

    elif result == TaskResult.CANCELED:
        print('Task at ' + request_item_location +
              ' was canceled. Returning to staging point...')
        initial_pose.header.stamp = navigator.get_clock().now().to_msg()
        navigator.goToPose(initial_pose)

    elif result == TaskResult.FAILED:
        print('Task at ' + request_item_location + ' failed!')
        exit(-1)

    while not navigator.isTaskComplete():
        pass

    exit(0)


if __name__ == '__main__':
    main()
```

- Notes -

Code extracted from the official Nav2 documentation (https://github.com/ros-planning/navigation2/tree/main/nav2_simple_commander/nav2_simple_commander).

- End of Notes -

Look at the code and identify where you are using the Simple Commander API and what the expected robot behavior should be. Do not worry if you do not understand the entire code because it will be reviewed later in the unit.

When you are done reviewing the code, compile the package:

▶ Execute in Shell

In [ ]:

```
cd ros2_ws
colcon build
source install/setup.bash
```

Test the new code! First, you need to have the Navigation system up and running.

▶ Execute in Shell

In [ ]:
```
ros2 launch path_planner_server navigation.launch.py
```

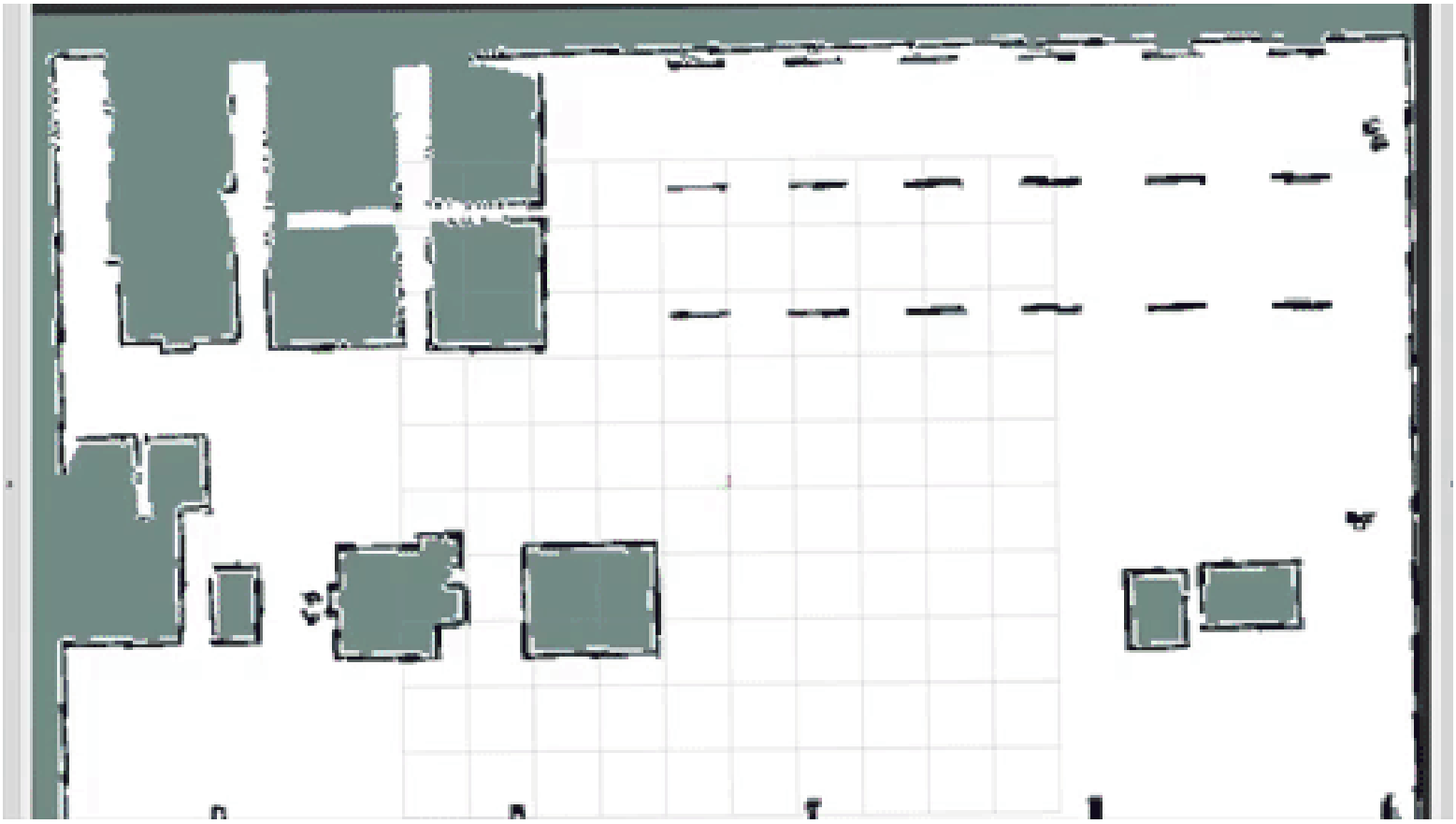
- Notes -

In this case, you do not need to set the robot's initial pose from RVIZ2 since the Python script has done that.

- End of Notes -

▶ Execute in Shell

In [ ]:
```
python3 ros2_ws/src/nav2_new_features/scripts/navigate_to_pose.py
```



### 2.1.2 Code Review

Great! So now that you have seen the Simple Commander API in action. Analyze in more depth what is going on in the Python script you have executed. Pay special attention to the part where you use the API. Start at the beginning:

In [ ]:
```
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult
```

This line of code is one of the most important. You are importing the **`BasicNavigator`** class, which gives you access to the Simple Commander API. So, whenever you want to use the Simple Commander API, import this class.

In [ ]:
```
navigator = BasicNavigator()
```

Of course, you also need to instantiate the class to use it.

In [ ]:
```
navigator.setInitialPose(initial_pose)
```

Here, you are using the **`setInitialPose()`** method, which will set the robot's initial pose (same as doing it with the 2D Pose Estimate tool from RVIZ2). In this case, you are specifying the initial pose in the **`initial_pose`** variable (it must be of the type **`PoseStamped`** ).

In [ ]:
```
navigator.waitUntilNav2Active()
```

The **`waitUntilNav2Active()`** method blocks the program's execution until Nav2 is entirely online and lifecycle nodes are in the active state.

In [ ]:
```
navigator.goToPose(shelf_item_pose)
```

Here, you are using the **`goToPose()`** method, which requests the robot to navigate to the specified pose. In this case, you are specifying the pose in the **`shelf_item_pose`** variable (it must be of the type **`PoseStamped`** ).

In [ ]:
```
while not navigator.isTaskComplete():
```

The **`isNavComplete()`** method will return **`True`** only after the robot has reached the goal. It will return **`False`** while it is still in process.

In [ ]:
```
feedback = navigator.getFeedback()
```

The **`getFeedback()`** method returns the feedback from the **`NavigateToPose`** action server.

In [ ]:
```
result = navigator.getResult()
```

The **`getResult()`** method returns the result from the **`NavigateToPose`** action server.

- Notes -

You can review all the Nav2 Simple Commander methods here: https://navigation.ros.org/commander_api/index.html (https://navigation.ros.org/commander_api/index.html)

- End of Notes -

## 2.2 Navigate Through Poses

The **NavigateThroughPoses** action is most suitable for pose-constrained navigation requests or other tasks represented in a behavior tree with a set of poses. This will **NOT** stop at each waypoint, but drive through them as a pose constraint.

NavigateThroughPoses.action

```
#goal definition
geometry_msgs/PoseStamped[] poses
string behavior_tree
---
#result definition
std_msgs/Empty result
---
#feedback definition
geometry_msgs/PoseStamped current_pose
builtin_interfaces/Duration navigation_time
builtin_interfaces/Duration estimated_time_remaining
int16 number_of_recoveries
float32 distance_remaining
int16 number_of_poses_remaining
```

As you can see, the action's inputs are nearly identical to NavigateToPose, except now you take in a vector of **`poses`** . The feedback is also similar, only containing the new **`number_of_poses_remaining`** field to track progress through the via-points.

### 2.2.1 Demo

In this demo, use the **`NavigateThroughPoses`** action to have your robot drive from its staging point throughout the warehouse on a known route. The `NavigateThroughPoses` action, like `NavigateToPose` , can deviate in the presence of obstacles, as you will see with the pallet jack in this demo. Once it finishes the route, it starts over and continues until stopped.

Create a new Python script named **navigate_through_poses.py**:

navigate_through_poses.py

In [ ]:

```
#! /usr/bin/env python3
# Copyright 2021 Samsung Research America
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import time
from copy import deepcopy

from geometry_msgs.msg import PoseStamped
from rclpy.duration import Duration
import rclpy

from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult

'''
Basic security route patrol demo. In this demonstration, the expectation
is that there are security cameras mounted on the robots recording or being
watched live by security staff.
'''


def main():
    rclpy.init()

    navigator = BasicNavigator()

    # Security route, probably read in from a file for a real application
    # from either a map or drive and repeat.
    security_route = [
        [1.792, 2.144],
        [1.792, -5.44],
        [1.792, -9.427],
        [-3.665, -9.427],
        [-3.665, -4.303],
        [-3.665, 2.330],
        [-3.665, 9.283]]

    # Set your demo's initial pose
    initial_pose = PoseStamped()
```

```
    initial_pose.header.frame_id = 'map'
    initial_pose.header.stamp = navigator.get_clock().now().to_msg()
    initial_pose.pose.position.x = 3.45
    initial_pose.pose.position.y = 2.15
    initial_pose.pose.orientation.z = 1.0
    initial_pose.pose.orientation.w = 0.0
    navigator.setInitialPose(initial_pose)

    # Wait for navigation to activate fully
    navigator.waitUntilNav2Active()

    # Do security route until dead
    while rclpy.ok():
        # Send your route
        route_poses = []
        pose = PoseStamped()
        pose.header.frame_id = 'map'
        pose.header.stamp = navigator.get_clock().now().to_msg()
        pose.pose.orientation.w = 1.0
        for pt in security_route:
            pose.pose.position.x = pt[0]
            pose.pose.position.y = pt[1]
            route_poses.append(deepcopy(pose))
        navigator.goThroughPoses(route_poses)

        # Do something during your route (e.x. AI detection on camera images fo
        # Print ETA for the demonstration
        i = 0
        while not navigator.isTaskComplete():
            i = i + 1
            feedback = navigator.getFeedback()
            if feedback and i % 5 == 0:
                print('Estimated time to complete current route: ' + '{0:.0f}'.
                      Duration.from_msg(feedback.estimated_time_remaining).nanc
                      + ' seconds.')

                # Some failure mode, must stop since the robot is clearly stuck
                if Duration.from_msg(feedback.navigation_time) > Duration(secor
                    print('Navigation has exceeded timeout of 180s, canceling t
                    navigator.cancelTask()

        # If at the end of the route, reverse the route to restart
        security_route.reverse()

        result = navigator.getResult()
        if result == TaskResult.SUCCEEDED:
            print('Route complete! Restarting...')
        elif result == TaskResult.CANCELED:
            print('Security route was canceled, exiting.')
            exit(1)
```

```
        elif result == TaskResult.FAILED:
            print('Security route failed! Restarting from the other side...')

    exit(0)


if __name__ == '__main__':
    main()
```

- Notes -

Code extracted from the official Nav2 documentation (https://github.com/ros-planning/navigation2/tree/main/nav2_simple_commander/nav2_simple_commander).

- End of Notes -

Look at the code and identify where you are using the Simple Commander API and what the expected robot behavior should be. Do not worry if you do not understand the entire code because it will be reviewed later in the unit.

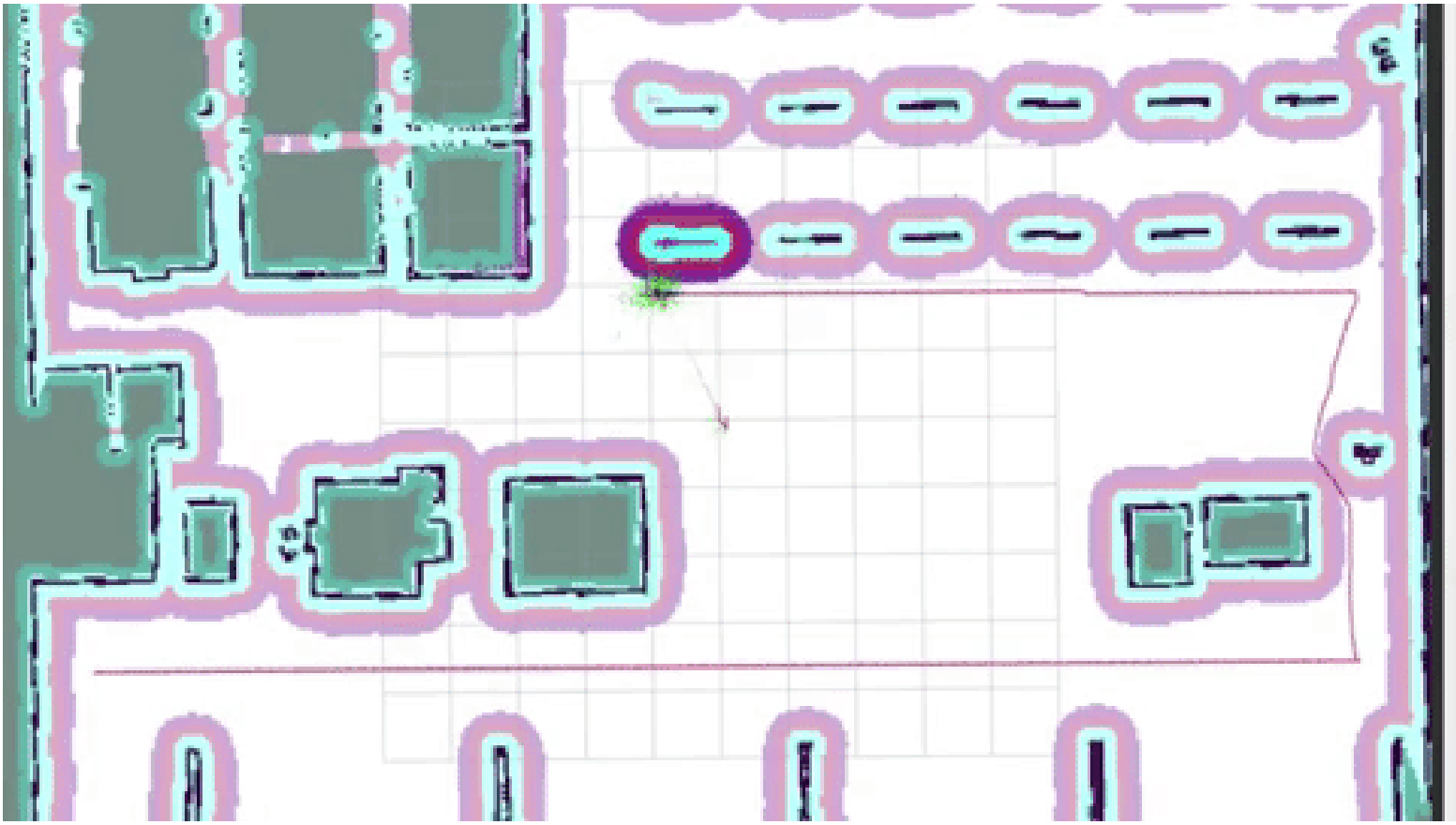
Test the new code! First, you need to have the Navigation system up and running.

▶ Execute in Shell

In [ ]:
```
ros2 launch path_planner_server navigation.launch.py
```

▶ Execute in Shell

In [ ]:
```
python3 ros2_ws/src/nav2_new_features/scripts/navigate_through_poses.py
```



### 2.2.2 Code Review

As you can see, the code is similar to the first script, with a couple of modifications.

In [ ]:
```
security_route = [
        [1.792, 2.144],
        [1.792, -5.44],
        [1.792, -9.427],
        [-3.665, -9.427],
        [-3.665, -4.303],
        [-3.665, 2.330],
        [-3.665, 9.283]]
```

First, define a set of poses you want the robot to go through.

In [ ]:
```
for pt in security_route:
        pose.pose.position.x = pt[0]
        pose.pose.position.y = pt[1]
        route_poses.append(deepcopy(pose))
```

Append these poses to a **`route_poses`** vector. Note that each pose must be defined as a PoseStamped message.

In [ ]:
```
navigator.goThroughPoses(route_poses)
```

Then, use this vector as input for calling the **`goThroughPoses()`** method. This method requests the robot to navigate through a set of poses.

In [ ]:
```
if Duration.from_msg(feedback.navigation_time) > Duration(seconds=180.0):
                print('Navigation has exceeded timeout of 180s, canceling the r
                navigator.cancelTask()
```

Note that in this script, you are also adding a timeout. If the navigation task takes longer than 180 seconds, you will cancel the current task using the **`cancelTask()`** method.

## 2.3 Waypoint Following

The **FollowWaypoints** action is most suitable for simple autonomy tasks where you wish to stop at each waypoint and execute a behavior (e.x. pausing for 2 seconds, taking a picture, waiting for someone to place a box on it, etc.). The Nav2 waypoint follower server contains `TaskExecutor` plugins to execute a task at each waypoint.

FollowWaypoints.action

```
#goal definition
geometry_msgs/PoseStamped[] poses
---
#result definition
int32[] missed_waypoints
---
#feedback definition
uint32 current_waypoint
```

As you can see, the API is simple. It takes in the set of poses, where the last pose is the final goal. The feedback is the current waypoint ID it is executing and returns a vector of missed waypoint IDs at the end should any be unnavigable.

### 2.3.1 Launch the FollowWaypoints action

The **FollowWaypoints** action is not started by default with the navigation system. Therefore, add it to your navigation launch file if you want to use it.

Begin by creating a configuration file for it. In the **config** folder of your **path_planner_server** package, add a new configuration file named **waypoint_follower.yaml**.

waypoint_follower.yaml

In [ ]:
```
waypoint_follower:
  ros__parameters:
    loop_rate: 20
    stop_on_failure: false
    waypoint_task_executor_plugin: "wait_at_waypoint"
    wait_at_waypoint:
      plugin: "nav2_waypoint_follower::WaitAtWaypoint"
      enabled: True
      waypoint_pause_duration: 200
```

In this example, load the default plugin **`WaitAtWaypoint`** . These plugins pause the robot for a specified time after reaching each waypoint. The time to wait can be configured with the **`waypoint_pause_duration`** parameter (specified in milliseconds).

There are other plugins available such as **PhotoAtWaypoint** or **InputAtWaypoint**. If you want to learn more about them, review the official documentation here (https://navigation.ros.org/configuration/packages/configuring-waypoint-follower.html).

Now it is time to add the **FollowWaypoints** action to your launch file. For this, a couple of modifications will be needed.

First, add the new configuration file path:

Add to navigation.launch.py

In [ ]:
```
waypoint_follower_yaml = os.path.join(get_package_share_directory(
        'path_planner_server'), 'config', 'waypoint_follower.yaml')
```

Next, add a new Node element to start the **waypoint_follower** server:

Add to navigation.launch.py

In [ ]:
```
Node(
    package='nav2_waypoint_follower',
    executable='waypoint_follower',
    name='waypoint_follower',
    output='screen',
    parameters=[waypoint_follower_yaml]),
```

Finally, and important, you need to add the new node to be started on the lifecycle manager:

Add to navigation.launch.py

In [ ]:
```
Node(
    package='nav2_lifecycle_manager',
    executable='lifecycle_manager',
    name='lifecycle_manager',
    output='screen',
    parameters=[{'autostart': True},
                {'node_names': ['map_server',
                                'amcl',
                                'controller_server',
                                'planner_server',
                                'recoveries_server',
                                'bt_navigator',
                                'waypoint_follower']}])
```

And that is it! Compile after you have added the new modifications.

▶ Execute in Shell

In [ ]:
```
cd ros2_ws
colcon build
source install/setup.bash
```

### 2.3.2 Demo

In this demo, use the **`FollowWaypoints`** action to have your robot drive from its staging point to a set of inspection points. The Nav2 waypoint follower **`TaskExecutor`** plugin takes images and RFID scans of the shelves that can be analyzed for inventory management.

Create a new Python script named **follow_waypoints.py**:

follow_waypoints.py

In [ ]:

```
#! /usr/bin/env python3
# Copyright 2021 Samsung Research America
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import time
from copy import deepcopy

from geometry_msgs.msg import PoseStamped
from rclpy.duration import Duration
import rclpy

from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult


def main():
    rclpy.init()

    navigator = BasicNavigator()

    # Inspection route, probably read in from a file for a real application
    # from either a map or drive and repeat.
    inspection_route = [
        [3.461, -0.450],
        [3.461, -2.200],
        [3.661, -4.121],
        [3.661, -5.850]]

    # Set your demo's initial pose
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.header.stamp = navigator.get_clock().now().to_msg()
    initial_pose.pose.position.x = 3.45
    initial_pose.pose.position.y = 2.15
    initial_pose.pose.orientation.z = 1.0
    initial_pose.pose.orientation.w = 0.0
    navigator.setInitialPose(initial_pose)

    # Wait for navigation to activate fully
```

```
    navigator.waitUntilNav2Active()

    # Send your route
    inspection_points = []
    inspection_pose = PoseStamped()
    inspection_pose.header.frame_id = 'map'
    inspection_pose.header.stamp = navigator.get_clock().now().to_msg()
    inspection_pose.pose.orientation.z = 1.0
    inspection_pose.pose.orientation.w = 0.0
    for pt in inspection_route:
        inspection_pose.pose.position.x = pt[0]
        inspection_pose.pose.position.y = pt[1]
        inspection_points.append(deepcopy(inspection_pose))
    nav_start = navigator.get_clock().now()
    navigator.followWaypoints(inspection_points)

    # Do something during your route (e.x. AI to analyze stock information or u
    # Print the current waypoint ID for the demonstration
    i = 0
    while not navigator.isTaskComplete():
        i = i + 1
        feedback = navigator.getFeedback()
        if feedback and i % 5 == 0:
            print('Executing current waypoint: ' +
                  str(feedback.current_waypoint + 1) + '/' + str(len(inspectior

    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        print('Inspection of shelves complete! Returning to start...')
    elif result == TaskResult.CANCELED:
        print('Inspection of shelving was canceled. Returning to start...')
        exit(1)
    elif result == TaskResult.FAILED:
        print('Inspection of shelving failed! Returning to start...')

    # go back to start
    initial_pose.header.stamp = navigator.get_clock().now().to_msg()
    navigator.goToPose(initial_pose)
    while not navigator.isTaskComplete():
        pass

    exit(0)


if __name__ == '__main__':
    main()
```

- Notes -

Code extracted from the official Nav2 documentation (https://github.com/ros-planning/navigation2/tree/main/nav2_simple_commander/nav2_simple_commander).

- End of Notes -

Look at the code and identify where you are using the Simple Commander API and what the expected robot behavior should be. Do not worry if you do not understand the entire code because it will be reviewed later in the unit.

Test the new code! First, you need to have the Navigation system up and running.

▶ Execute in Shell

In [ ]:
```
ros2 launch path_planner_server navigation.launch.py
```

- Notes -

Take a moment to ensure that you have correctly launched the **FollowWaypoints** server. Run the command:

▶ Execute in Shell

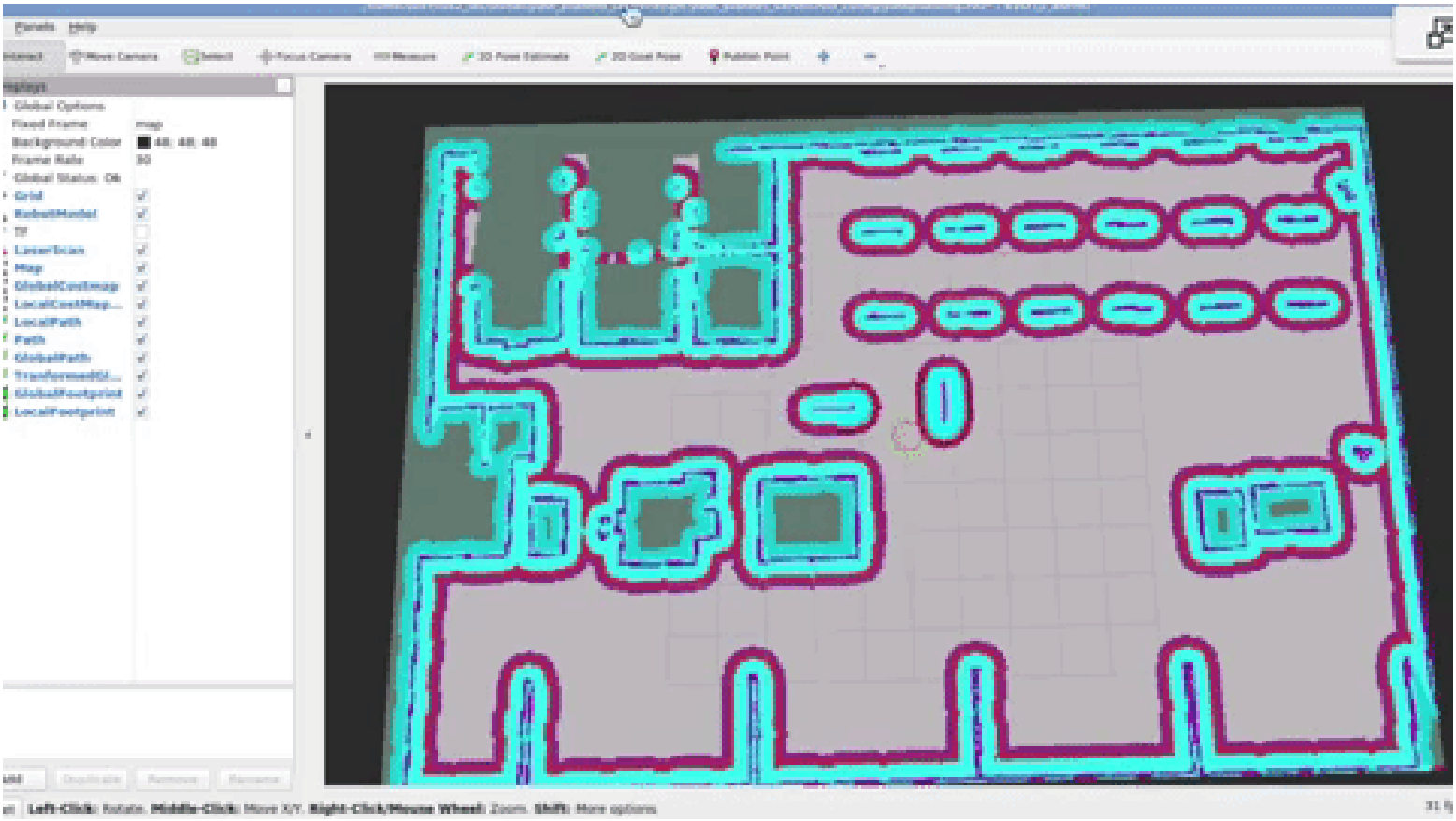
In [ ]:
```
ros2 action list
```

If everything is okay, you should see the **`/follow_waypoints`** among the list of actions available.

- End of Notes -

▶ Execute in Shell

In [ ]:
```
python3 ros2_ws/src/nav2_new_features/scripts/follow_waypoints.py
```



### 2.3.2 Code Review

In [ ]:
```
navigator.followWaypoints(inspection_points)
```

Like you did before, use the **`inspection_points`** vector as input for calling the **`followWaypoints()`** method. This method requests the robot to follow a set of waypoints (list of `PoseStamped` messages). This will execute the chosen TaskExecutor plugin at each pose.

In [ ]:
```
i = 0
while not navigator.isTaskComplete():
    i = i + 1
    feedback = navigator.getFeedback()
    if feedback and i % 5 == 0:
        print('Executing current waypoint: ' +
              str(feedback.current_waypoint + 1) + '/' + str(len(inspection_poi
```

In this code, you are printing the robot's current waypoint. However, remember that you could do other (more meaningful) tasks here.

# 3 Costmap Filters

Nav2 provides an exciting set of features in the form of Costmap filters. You can think of a Costmap filter as an extra layer (mask) added to your regular Costmaps. The exciting thing is that with this extra mask, you can make your robot have a specific behavior depending on the area (zone) on the map.

For instance, you can make the robot avoid going to a particular zone of the map (also known as **Keepout Zones**) or limit the speed of the robot depending on the map's area (**Speed Restricted Zones**).

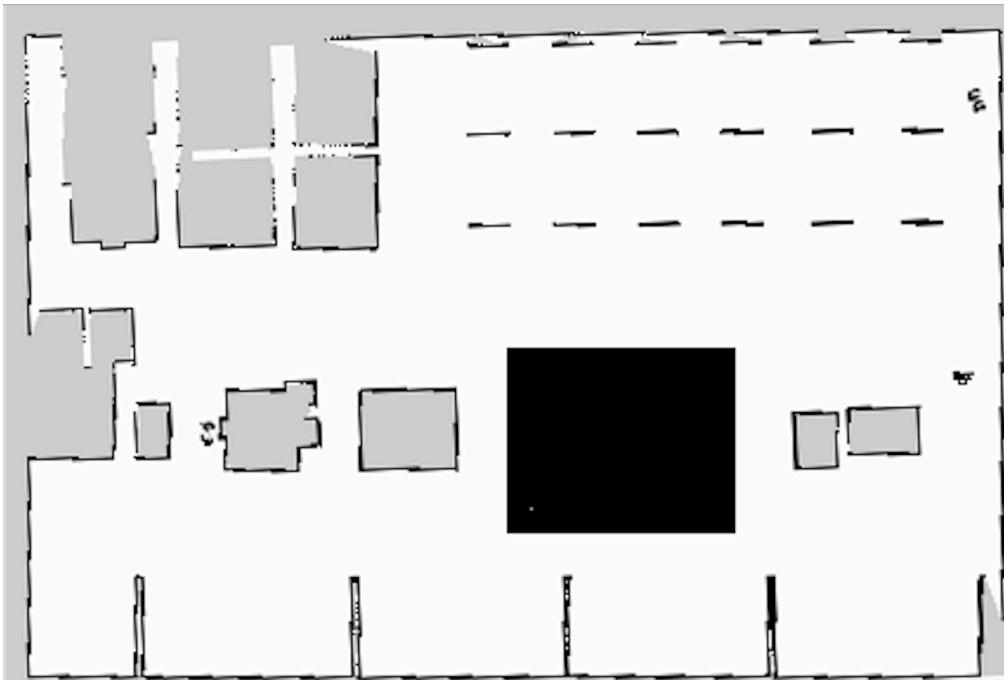
## 3.1 Keepout Mask

### 3.1.1 Edit the map

Review how to make the robot avoid certain zones of the environment by using a Keepout Mask filter.

The Keepout Mask is a file similar to a map, containing the mask to apply as the Keepout Zone. Therefore, to apply a mask, you will need a map file of the environment. So start by downloading the file **map_rotated.pgm**. You can easily download the file from the IDE by right-clicking on it and selecting the `Download` option.

Now, you will paint the map area you want your robot to avoid in **BLACK**. A simple example is the following:



You might ask, "Why do I have to paint in black?" The shade of each pixel on the mask means encoded information for the specific Costmap filter you will use. The incoming mask file is being read by the Map-Server and converted into OccupancyGrid values from [the 0..100] range where:

- 0 means free cell
- 100 means occupied cell

For the **Keepout Filter** mask, the higher the value, the more restricted the area. Therefore, a black area corresponds to a value of 100 (the robot can not navigate through it). For values in between, the robot will be allowed to move in these areas, but its presence there will be “undesirable” (the higher the value, the sooner planners will try to get the robot out of this area).

Once you have edited your map, upload it again. Rename the edited map file to **map_keepout.pgm** and upload it to the **maps** folder of the **map_server** package.

Finally, you will also have to create the associated YAML file of the map:

map_keepout.yaml

In [ ]:
```
image: map_keepout.pgm
resolution: 0.050000
origin: [-7.000, -10.500000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

### 3.1.2 Launch the Costmap Filter nodes

Once you have created the edited map files, it is time to launch the required nodes. But before that, update the configuration files. First, you will update the Costmap configuration, which can be found in **planner_server.yaml** and **controller.yaml** files.

Start by adding a **`filter`** parameter:

In [ ]:
```
filters: ["keepout_filter"]
```

And, of course, you have to add the configuration for the filter:

In [ ]:
```
keepout_filter:
    plugin: "nav2_costmap_2d::KeepoutFilter"
    enabled: True
    filter_info_topic: "/costmap_filter_info"
```

As you can see, you specify the filter to use, **`KeepoutFilter`** .

- Notes -

Remember that you must apply the changes to the global and local Costmaps. In the end, you should have something like this:

planner_server.yaml

In [ ]:
```
...

global_costmap:
  global_costmap:
    ros__parameters:
      ...
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      filters: ["keepout_filter"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::KeepoutFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"

...
```

controller.yaml

In [ ]:
```
...

local_costmap:
  local_costmap:
    ros__parameters:
      ...
      plugins: ["voxel_layer", "inflation_layer"]
      filters: ["keepout_filter"]
      ...
      keepout_filter:
        plugin: "nav2_costmap_2d::KeepoutFilter"
        enabled: True
        filter_info_topic: "/costmap_filter_info"

...
```

- End of Notes -

Now, set extra parameters for the new filter nodes you will launch. You will be launching two new nodes:

- **Costmap Filter Info Server**: This node will publish `nav2_msgs/CostmapFilterInfo` messages. These messages contain metadata such as the filter type or data conversion coefficients.
- **Mask Map Server**: This node will publish `OccupancyGrid` messages.

Both messages are required to be published as a pair to generate the Costmap filter. If you want to learn more about this intra-process, you can review the [design document (https://github.com/ros-planning/navigation2/blob/main/doc/design/CostmapFilters_design.pdf)](https://github.com/ros-planning/navigation2/blob/main/doc/design/CostmapFilters_design.pdf).

Start by creating a new configuration file named **`filters.yaml`**, where you place the parameters required by these nodes:

filters.yaml

In [ ]:

```yaml
costmap_filter_info_server:
  ros__parameters:
    use_sim_time: true
    filter_info_topic: "/costmap_filter_info"
    type: 0
    mask_topic: "/keepout_filter_mask"
    base: 0.0
    multiplier: 1.0

filter_mask_server:
  ros__parameters:
    use_sim_time: true
    frame_id: "map"
    topic_name: "/keepout_filter_mask"
    yaml_filename: "/home/user/ros2_ws/src/nav2_pkgs/map_server/maps/map_keepou
```

- Notes -

The **`type`** parameter sets the type of Costmap filter used. The values are:

- 0 for Keepout Zones/preferred lanes filter
- 1 for speed filter (if the speed limit is specified in % of maximum speed)
- 2 for speed filter (if the speed limit is specified in absolute value (m/s))

The **`mask_topic`** parameter sets the topic to publish the filter mask. Therefore, the topic name specified must be the same as the **`topic_name`** parameter of the Map-Server.

The **`base`** and **`multiplier`** are coefficients used for applying the mask for the filter. They are used in the following formula: `filter_space_value = base + multiplier * mask_value`

For Keepout Zones, they need to be set to 0.0 and 1.0, respectively, to explicitly show that you have a one-to-one conversion from OccupancyGrid values -> to a filter value space.

- End of Notes -

Now it is time to add the filter nodes to your launch file. For this, a few modifications will be needed in the **`navigation.launch.py`** file.

First, add the new configuration file path:

Add to navigation.launch.py

In [ ]:
```
filters_yaml = os.path.join(get_package_share_directory(
        'path_planner_server'), 'config', 'filters.yaml')
```

Next, two new Node elements:

Add to navigation.launch.py

In [ ]:
```
Node(
    package='nav2_map_server',
    executable='map_server',
    name='filter_mask_server',
    output='screen',
    emulate_tty=True,
    parameters=[filters_yaml]),

Node(
    package='nav2_map_server',
    executable='costmap_filter_info_server',
    name='costmap_filter_info_server',
    output='screen',
    emulate_tty=True,
    parameters=[filters_yaml]),
```

Finally, and important, you need to add the new nodes to be started on the lifecycle manager:

Add to navigation.launch.py

In [ ]:
```
Node(
    package='nav2_lifecycle_manager',
    executable='lifecycle_manager',
    name='lifecycle_manager',
    output='screen',
    parameters=[{'autostart': True},
                {'node_names': ['map_server',
                                'amcl',
                                'controller_server',
                                'planner_server',
                                'recoveries_server',
                                'bt_navigator',
                                'waypoint_follower',
                                'filter_mask_server',
                                'costmap_filter_info_server']}])
```

And that is it! Make sure to compile after you have added the new modifications.

▶ Execute in Shell

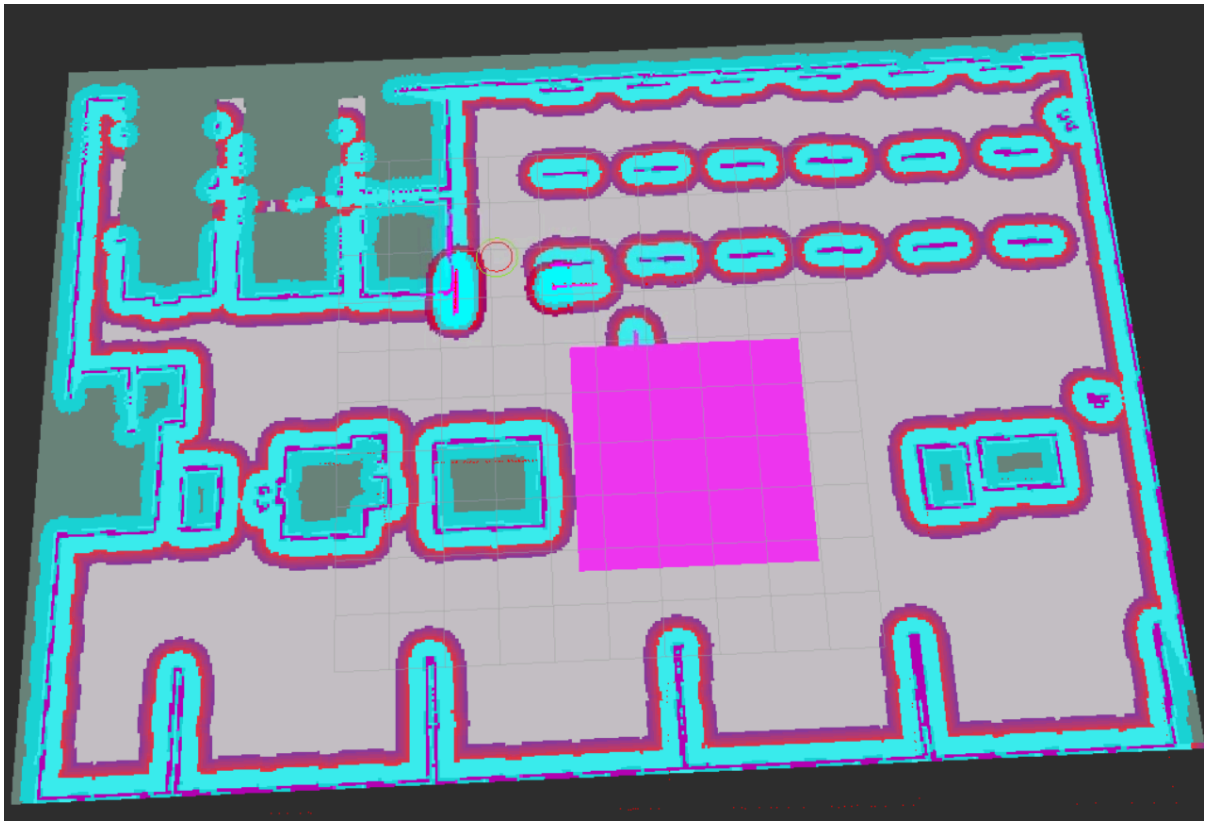
In [ ]:
```
cd ros2_ws
colcon build
source install/setup.bash
```

Test the new code!

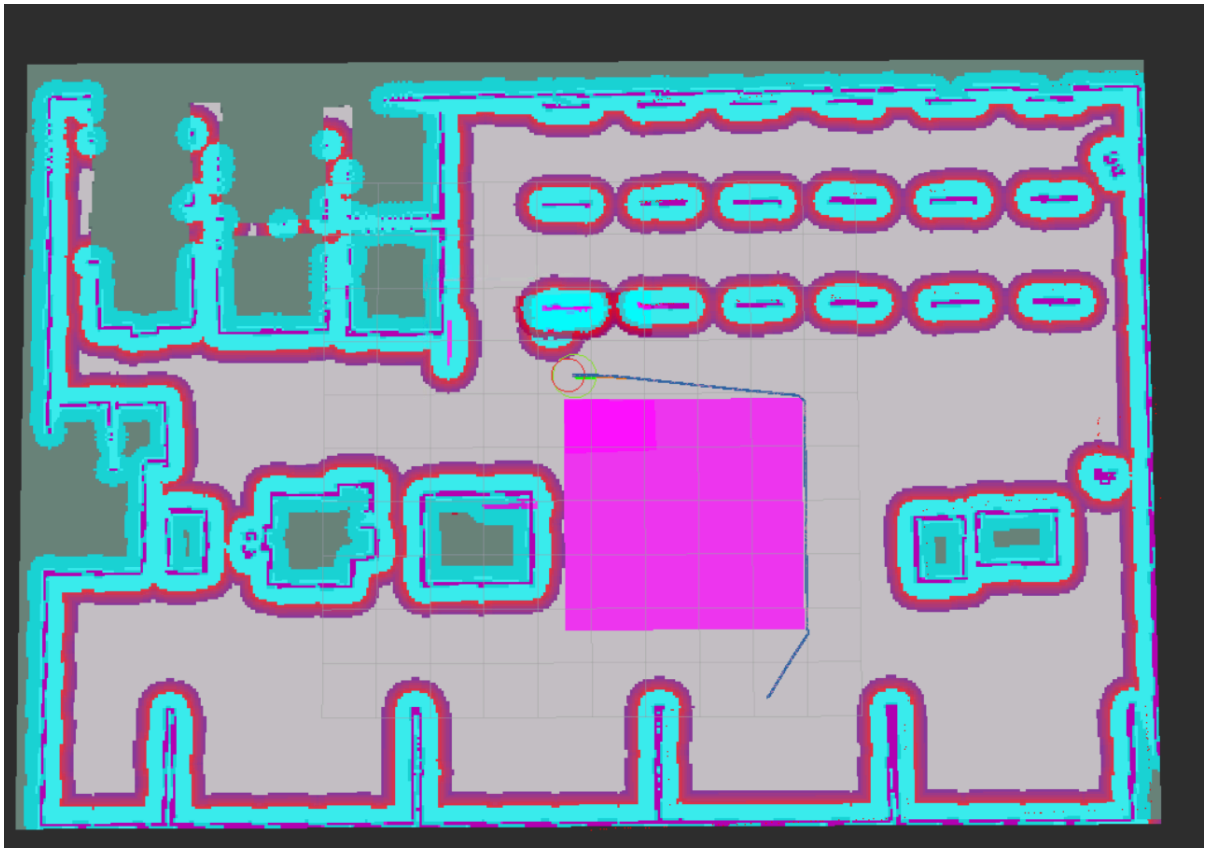
▶ Execute in Shell

In [ ]:
```
ros2 launch path_planner_server navigation.launch.py
```

If you followed all the instructions correctly, you get a Costmap like the one shown below:



Now, when you send a goal near the Keepout Zone, the plan will avoid the pink area.



- Warning -

When trying to navigate near the Keepout Zone, you might end up getting errors like this:

In [ ]:
```
[controller_server-3] [ERROR] [1654803915.474550722] [DWBLocalPlanner]: 1.00:
```

If you get this error, you can open the **`controller.yaml`** parameters and remove the critic named **`ObstacleFootprint`** from the list of `critics`.

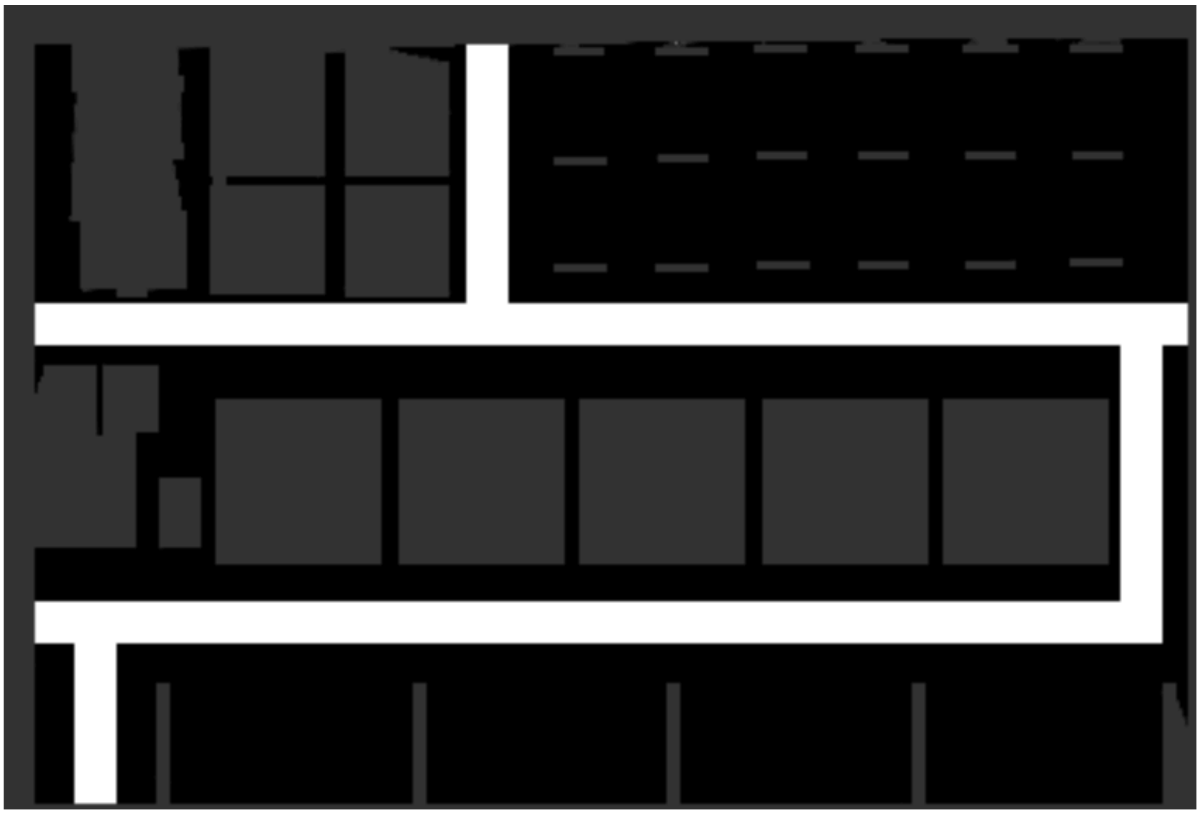
In [ ]:
```
#critics: ["RotateToGoal", "Oscillation", "ObstacleFootprint", "GoalAlign",

critics: ["RotateToGoal", "Oscillation", "GoalAlign", "PathAlign", "PathDist",
```

- End Warning -

- Notes -

Although it is not covered in this unit, you can also invert this logic for **preferred lanes** rather than **Keepout** Zones. Below, see an example where you set the main warehouse lane as the preferred lane of travel, but the robot can navigate off of it for the final approach to the goal when necessary.



- End of Notes -

## 3.2 Speed Limits

### 3.2.1 Edit the map

The principle of the Speed Limits Filter is similar to the one used for Keepout Zones. However, in this case, the mask values will have a different meaning: encoded speed limits for the areas corresponding to the cell on the map.

As you already know, `OccupancyGrid` values belong to the [0..100] range. For the Speed Filter, a 0 value means no speed limit in the area corresponding to the map. Values from the [1..100] range are being linearly converted into a speed limit value by the following formula: `speed_limit = filter_mask_data * multiplier + base`

Simplifying means that the lighter the percentage of gray used, the lower the speed limit will be, and vice versa.

Start by downloading the file **map_rotated.pgm** again.

Now, paint in **GREY** (use different tones for different areas) the areas of the map where you want your robot to have speed limits.

An example is the following:

Once you have edited your map, upload it again. Rename the edited map file to **map_speed.pgm** and upload it to the **maps** folder of the **map_server** package.

Finally, create the associated YAML file of the map:

map_speed.yaml

In [ ]:
```
image: map_speed.pgm
resolution: 0.050000
origin: [-7.000, -10.500000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

### 3.1.2 Configure the Speed Limit nodes

In this unit, only apply the speed filter to the global Costmap. So, first, update the **`filter`** parameter:

Add to planner_server.yaml

In [ ]:
```
# For Keepout Zones
#filters: ["keepout_filter"]

# For Speed Limits
filters: ["speed_filter"]
```

And, of course, you have to add the configuration for the filter:

Add to planner_server.yaml

In [ ]:
```
speed_filter:
    plugin: "nav2_costmap_2d::SpeedFilter"
    enabled: True
    filter_info_topic: "/costmap_filter_info"
    speed_limit_topic: "/speed_limit"
```

As you can see, you specify the filter to use, **`SpeedFilter`** .

Also, you need to specify the same **`speed_limit_topic`** to the **`controller_server`** configuration:

controller.yaml

In [ ]:
```
...

controller_server:
  ros__parameters:
    ...
    # For Speed Limits
    speed_limit_topic: "/speed_limit"
    ...
```

Now update the file **`filters.yaml`** to set the required parameters for the speed filter:

filters.yaml

In [ ]:
```
costmap_filter_info_server:
  ros__parameters:
    use_sim_time: true
    filter_info_topic: "/costmap_filter_info"
    # For Keepout Zones
    #type: 0
    #mask_topic: "/keepout_filter_mask"
    #base: 0.0
    #multiplier: 1.0
    # For Speed Limits
    type: 1
    mask_topic: "/speed_filter_mask"
    base: 100.0
    multiplier: -1.0

filter_mask_server:
  ros__parameters:
    use_sim_time: true
    frame_id: "map"
    # For Keepout Zones
    #topic_name: "/keepout_filter_mask"
    #yaml_filename: "/home/user/ros2_ws/src/map_server/maps/map_keepout.yaml"
    # For Speed Limits
    topic_name: "/speed_filter_mask"
    yaml_filename: "/home/user/ros2_ws/src/nav2_pkgs/map_server/maps/map_speeds
```

And that is it! Compile after you have added the new modifications.

Execute in Shell

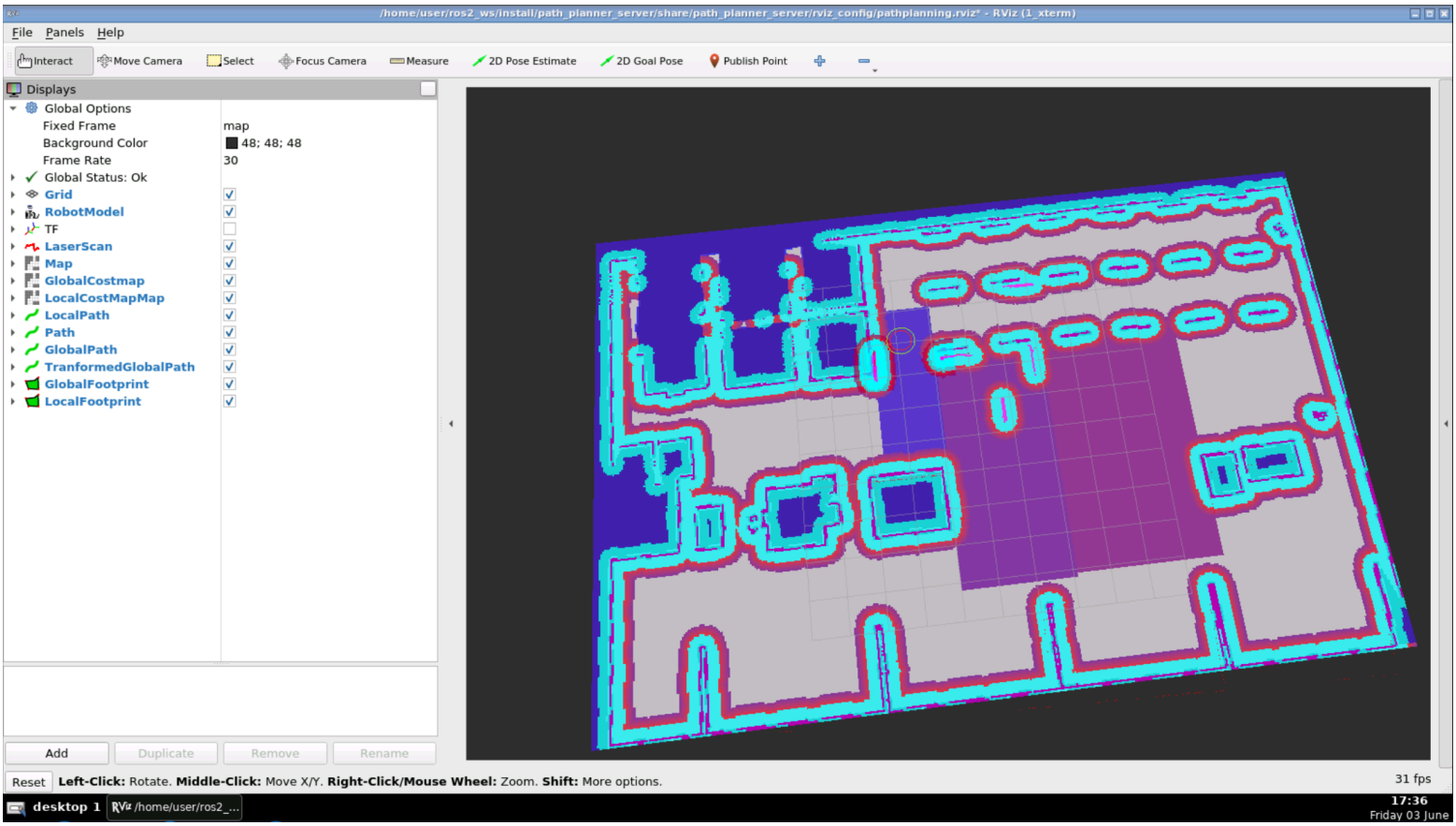
In [ ]:
```
cd ros2_ws
colcon build
source install/setup.bash
```

Test the new code!

▶ Execute in Shell

In [ ]:
```
ros2 launch path_planner_server navigation.launch.py
```

If you followed all the instructions correctly, you get a Costmap like the one shown below:




For visualizing the filter mask, add a Map display with the following configuration:

| | |
|---|---|
| Map | ☑ |
| Status: Ok | |
| Topic | /speed_filter_mask |
| Depth | 5 |
| History Policy | Keep Last |
| Reliability Policy | Reliable |
| Durability Policy | Transient Local |
| Filter size | 10 |
| Update Topic | /speed_filter_mask_updates |
| Alpha | 0.7 |
| Color Scheme | map |
| Draw Behind | ☐ |
| Resolution | 0.05 |
| Width | 286 |
| Height | 423 |
| Position | -7; -10.5; 0 |
| Orientation | 0; 0; 0; 1 |
| Use Timestamp | ☐ |

Send a Nav2 Goal to the robot and check how the speed limit changes depending on the robot's navigation area.

You can also visualize the **`/speed_limit`** topic for getting extra data. Every time the robot enters a new speed-limited area, a message like the following will be published in this topic:

```
user:~$ ros2 topic echo /speed_limit
header:
  stamp:
    sec: 704
    nanosec: 820000000
  frame_id: map
percentage: true
speed_limit: 80.0
---

header:
  stamp:
    sec: 750
    nanosec: 560000000
  frame_id: map
percentage: true
speed_limit: 57.0
---
```