ROS 2 Programming and Simulation

Gianluca Palli

gianluca.palli@unibo.it

DEI - LAR
University of Bologna
Viale del Risorgimento 2
40136 Bologna

Creating Your First ROS 2 Package

A package can be considered a container for your ROS 2 code.

If you want to be able to install your code or share it with others, then you'll need it organized in a package.

With packages, you can release your ROS 2 work and allow others to build and use it easily.

Package creation in ROS 2 uses ament as its build system and colcon as its build tool.

You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

What Makes Up a ROS 2 Package?

ROS 2 CMake packages each have their own minimum required contents:

- `package.xml` file containing meta information about the package
- `CMakeLists.txt` file that describes how to build the code within the package

While for **ROS 2 Python** packages:

- `package.xml` file containing meta information about the package
- `setup.py` containing instructions for how to install the package
- `setup.cfg` is required when a package has executables, so `ros2 run` can find them
- `/<package_name>` - a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

Packages in a Workspace

A single workspace can contain as many packages as you want, each in their own folder. You can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages

Best practice is to have a src folder within your workspace, and to create your packages in there

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

Create a CMake Package

Let's create a new package in the dev_ws workspace

```
$ mkdir -p ~/dev_ws/src
$ cd ~/dev_ws/src
$ ros2 pkg create --build-type ament_cmake <package_name>
```

You can use the optional argument --node-name which creates a simple Hello World type executable in the package

```
$ ros2 pkg create --build-type ament_cmake \
   --node-name my_cmake_node my_cmake_package
```

You will now have a new folder within your workspace's src directory called my_cmake_package
You can now build your first ROS package with the command:

```
$ cd ~/dev_ws
$ colcon build
```

This will build all the packages in your workspace. To build packages selectively use:

```
$ colcon build --packages-select my_cmake_package
```

Create a Python Package

Let's create a new package in the `dev_ws` workspace

```
$ cd ~/dev_ws/src
$ ros2 pkg create --build-type ament_python <package_name>
```

You can use the optional argument `--node-name` which creates a simple Hello World type executable in the package

```
$ ros2 pkg create --build-type ament_python \
  --node-name my_python_node my_python_package
```

You will now have a new folder within your workspace's `src` directory called `my_python_package`

You can now build your first ROS package with the command:

```
$ cd ~/dev_ws
$ colcon build
```

This will build all the packages in your workspace. To build packages selectively use:

```
$ colcon build --packages-select my_python_package
```

Use Your Package

In a new terminal, from inside the dev_ws directory, run the following command to source your workspace:

```
$ . install/local_setup.bash
```

Now that your workspace has been added to your path, you will be able to use your new package's executables

To run the executable you created using the --node-name argument during package creation, enter the command:

```
    $ ros2 run my_cmake_package my_cmake_node
    $ ros2 run my_python_package my_python_node
```

Which will return a message to your terminal:

```
  hello world my_cmake_package package
  Hi from my_python_package.
```

Customize Your CMake Package

Inside dev_ws/src/my_cmake_package, you will see the files and folders that ros2 pkg create automatically generated:

```
CMakeLists.txt include package.xml src
```

my_cmake_node.cpp is inside the src directory. This is where all your custom C++ nodes will go in the future.
From dev_ws/src/my_cmake_package, open package.xml

Input your name and email on the maintainer line if it hasn't been automatically populated for you. Then, edit the description line to summarize the package:

```
<description>Beginner client libraries tutorials practice\
 package</description>
```

Then update the license line. Since this package is only for practice, it's safe to use any license. We use Apache License 2.0:

```
<license>Apache License 2.0</license>
```

Don't forget to save once you're done editing.

Customize Your Python Package

Inside dev_ws/src/my_python_package, you will see the files and folders that ros2 pkg create automatically generated:

```
my_python_package package.xml resource setup.cfg setup.py test
```

my_python_node.py is inside the my_python_package directory. This is where all your custom Python nodes will go in the future.

The setup.py file contains the same description, maintainer and license fields as package.xml, so you need to set those as well. They need to match exactly in both files. The version and name (package_name) also need to match exactly, and should be automatically populated in both files.

Edit the maintainer, maintainer_email, and description lines to match package.xml

Don't forget to save once you're done editing.

Writing a Simple C++ Publisher and Subscriber

Navigate into `dev_ws/src`, and run the package creation command:

```
$ ros2 pkg create --build-type ament_cmake cpp_pubsub
```

Navigate into `dev_ws/src/cpp_pubsub/src` and download the example talker code by entering the following command:

```
$ wget -O publisher_member_function.cpp \
   https://raw.githubusercontent.com/ros2/examples/master\
   /rclcpp/topics/minimal_publisher/member_function.cpp
```

Examine the Publisher Code

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;
```

The top of the code includes the standard C++ headers

After the standard C++ headers is the `rclcpp/rclcpp.hpp` include which allows you to use the most common pieces of the ROS 2 system

Last is `std_msgs/msg/string.hpp`, which includes the built-in message type you will use to publish data

These lines represent the node's dependencies that have to be added to `package.xml` and `CMakeLists.txt`

Examine the Publisher Code (Cont.)

The next line creates the node class `MinimalPublisher` by inheriting from `rclcpp::Node`

```
class MinimalPublisher : public rclcpp::Node
```

The public constructor names the node `minimal_publisher` and initializes `count_` to 0. Inside the constructor, the publisher is initialized with the `String` message type, the topic name `topic`, and the required queue size to limit messages in the event of a backup. Next, `timer_` is initialized, which causes the `timer_callback` function to be executed twice a second

```
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = \
      this->create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(
    500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }
```

Examine the Publisher Code (Cont.)

The `timer_callback` function is where the message data is set and the messages are actually published

The `RCLCPP_INFO` macro ensures every published message is printed to the console

```
private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'",\
       message.data.c_str());
    publisher_->publish(message);
  }
```

Last is the declaration of the timer, publisher, and counter fields

```
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
```

Examine the Publisher Code (Cont.)

Following the `MinimalPublisher` class is `main`, where the node actually executes

`rclcpp::init` initializes ROS 2, and `rclcpp::spin` starts processing data from the node, including callbacks from the timer

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

Add Dependencies

Navigate one level back to the `dev_ws/src/cpp_pubsub` directory, where the `CMakeLists.txt` and `package.xml` files have been created for you

Open `package.xml` with your text editor and add a new line after the `ament_cmake` buildtool dependency and paste the following dependencies corresponding to your node's include statements:

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

This declares the package needs `rclcpp` and `std_msgs` when its code is executed

Add Dependencies (Cont.)

Now open the CMakeLists.txt file. Below the existing dependency find_package(ament_cmake REQUIRED), add the lines:

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

After that, add the executable and name it talker so you can run your node using ros2 run:

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)
```

Finally, add the install(TARGETS...) section so ros2 run can find your executable:

```
install(TARGETS
  talker
  DESTINATION lib/${PROJECT_NAME})
```

You could build your package now, source the local setup files, and run it, but let's create the subscriber node first so you can see the full system at work

Write the Subscriber Node

Return to dev_ws/src/cpp_pubsub/src to create the next node. Enter the following code in your terminal:

```
$ wget -O subscriber_member_function.cpp \
   https://raw.githubusercontent.com/ros2/examples/master\
   /rclcpp/topics/minimal_subscriber/member_function.cpp
```

Entering ls in the console will now return:

```
publisher_member_function.cpp subscriber_member_function.cpp
```

Open the subscriber_member_function.cpp with your text editor

Examine the Subscriber Code

Now the node is named `minimal_subscriber`, and the constructor uses the node's `create_subscription` class to execute the callback

There is no timer because the subscriber simply responds whenever data is published to the `topic` topic

```
public:
  MinimalSubscriber() : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::String>(\
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback,\
      this, _1));
  }
```

The `topic_callback` function receives the string message data, and simply writes it to the console using the`RCLCPP_INFO` macro

The only field declaration in this class is the subscription

```
private:
  void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
  {
    RCLCPP_INFO(this->get_logger(), "I␣heard:␣'%s'", msg->data.c_str());
  }
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
```

Examine the Subscriber Code (Cont.)

The `main` function is exactly the same, except now it spins the `MinimalSubscriber` node

For the publisher node, spinning meant starting the timer, but for the subscriber it simply means preparing to receive messages whenever they come

Since this node has the same dependencies as the publisher node, there's nothing new to add to `package.xml`

Reopen `CMakeLists.txt` and add the executable and target for the subscriber node below the publisher's entries

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

install(TARGETS
  talker
  listener
  DESTINATION lib/${PROJECT_NAME})
```

Build and Run

It's good practice to run `rosdep` in the root of your workspace to check for missing dependencies before building:

```
$ rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your new package:

```
$ colcon build --packages-select cpp_pubsub
```

Open a new terminal, navigate to `dev_ws`, and source the setup files:

```
$ . install/setup.bash
```

Now run the talker node:

```
$ ros2 run cpp_pubsub talker
```

Open another terminal, source the setup files from inside `dev_ws` again, and then start the listener node:

```
$ ros2 run cpp_pubsub listener
```

Writing a Simple Python Publisher and Subscriber

Navigate into `dev_ws/src`, and run the package creation command:

```
$ ros2 pkg create --build-type ament_python py_pubsub
```

Navigate into `dev_ws/src/py_pubsub/py_pubsub` and download the example talker code by entering the following command:

```
$ wget https://raw.githubusercontent.com/ros2/examples/humble/\
  rclpy/topics/minimal_publisher\
  /examples_rclpy_minimal_publisher/publisher_member_function.py
```

Examine the Publisher Code

The first lines of code after the comments import `rclpy` so its `Node` class can be used

```
import rclpy
from rclpy.node import Node
```

The next statement imports the built-in string message type that the node uses to structure the data that it passes on the topic.

```
from std_msgs.msg import String
```

Next, the `MinimalPublisher` class is created, which inherits from (or is a subclass of) `Node`

```
class MinimalPublisher(Node):
```

Examine the Publisher Code (Cont.)

Following is the definition of the class's constructor

```
def __init__(self):
    super().__init__('minimal_publisher')
    self.publisher_ = self.create_publisher(String, 'topic', 10)
    timer_period = 0.5 # seconds
    self.timer = self.create_timer(timer_period, \
       self.timer_callback)
    self.i = 0
```

`create_publisher` declares that the node publishes messages of type `String`, over a topic named topic, and that the "queue size" is 10

Queue size is a required QoS (quality of service) setting that limits the amount of queued messages if a subscriber is not receiving them fast enough

Examine the Publisher Code (Cont.)

Next, a timer is created with a callback to execute every 0.5 seconds. `self.i` is a counter used in the callback

```python
def timer_callback(self):
    msg = String()
    msg.data = 'Hello␣World:␣%d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing:␣"%s"' % msg.data)
    self.i += 1
```

`timer_callback` creates a message with the counter value appended, and publishes it to the console with `get_logger().info`

Examine the Publisher Code (Cont.)

Lastly, the main function is defined

```
def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
```

First the `rclpy` library is initialized, then the node is created, and then it "spins" the node so its callbacks are called

Add Dependencies

Navigate one level back to the dev_ws/src/py_pubsub directory, where the setup.py, setup.cfg and package.xml files have been created for you

Open package.xml with your text editor and fill in the <description>, <maintainer> and <license> tags

```
<description>Examples of minimal publisher/subscriber
   using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

After the lines above, add the following dependencies corresponding to your node's import statements:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

This declares the package needs rclpy and std_msgs when its code is executed

Add an Entry Point

Open the setup.py file. Again, match the maintainer, maintainer_email, description and license fields to your package.xml

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples␣of␣minimal␣publisher/subscriber
␣␣␣␣using␣rclpy',
license='Apache␣License␣2.0',
```

Add the following line within the console_scripts brackets of the entry_points field

```
entry_points={
        'console_scripts': [
                'talker␣=␣py_pubsub.publisher_member_function:main',
        ],
},
```

You could build your package now, source the local setup files, and run it, but let's create the subscriber node first so you can see the full system at work

Write the Subscriber Node

Return to dev_ws/src/py_pubsub/py_pubsub to create the next node. Enter the following code in your terminal:

```
$ wget https://raw.githubusercontent.com/ros2/examples \
    /humble/rclpy/topics/minimal_subscriber \
    /examples_rclpy_minimal_subscriber/subscriber_member_function.py
```

Entering ls in the console will now return:

```
__init__.py publisher_member_function.py subscriber_member_function.py
```

Open the subscriber_member_function.py with your text editor

Examine the Subscriber Code

The constructor creates a subscriber with the same publisher arguments

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription # prevent unused variable warning
```

The subscriber's constructor and callback don't include any timer definition, because it doesn't need one. Its callback gets called as soon as it receives a message

Examine the Subscriber Code (Cont.)

The callback definition simply prints an info message to the console, along with the data it received

```
def listener_callback(self, msg):
    self.get_logger().info('I␣heard:␣"%s"' % msg.data)
```

The `main` definition is almost exactly the same, replacing the creation and spinning of the publisher with the subscriber

```
minimal_subscriber = MinimalSubscriber()

rclpy.spin(minimal_subscriber)
```

Since this node has the same dependencies as the publisher, there's nothing new to add to `package.xml`. The `setup.cfg` file can also remain untouched.

Add an Entry Point

Reopen setup.py and add the entry point for the subscriber node below the publisher's entry point. The entry_points field should now look like this:

```
entry_points={
        'console_scripts': [
                'talker␣=␣py_pubsub.publisher_member_function:main',
                'listener␣=␣py_pubsub.subscriber_member_function:main',
        ],
},
```

Build and Run

It's good practice to run `rosdep` in the root of your workspace to check for missing dependencies before building:

```
$ rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your new package:

```
$ colcon build --packages-select py_pubsub
```

Open a new terminal, navigate to `dev_ws`, and source the setup files:

```
$ . install/setup.bash
```

Now run the talker node:

```
$ ros2 run py_pubsub talker
```

Open another terminal, source the setup files from inside `dev_ws` again, and then start the listener node:

```
$ ros2 run py_pubsub listener
```

ROS Simulation

The default ROS simulator is Gazebo. To install Gazebo 11

```
$ sudo apt-get install ros-humble-gazebo-*
```

Install Cartographer

```
$ sudo apt install ros-humble-cartographer
$ sudo apt install ros-humble-cartographer-ros
```

Install Navigation2

```
$ sudo apt install ros-humble-navigation2
$ sudo apt install ros-humble-nav2-bringup
```

Install TurtleBot3 Packages

```
$ sudo apt install ros-humble-dynamixel-sdk
$ sudo apt install ros-humble-turtlebot3-msgs
$ sudo apt install ros-humble-turtlebot3
```

ROS Simulation (Cont.)

Install Turtlebot3 simulation packages

```
$ mkdir -p ~/turtlebot3_ws/src/
$ cd ~/turtlebot3_ws/src/
$ git clone -b humble-devel \
   https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/turtlebot3_ws
$ rosdep install -i --from-path src --rosdistro humble -y
$ colcon build --symlink-install
$ . install/setup.bash
```

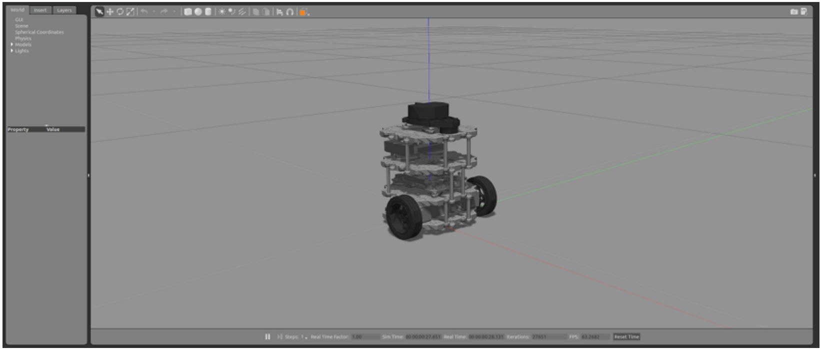
Update Keyrings

```
$ sudo rm -f /usr/share/keyrings/ros-archive-keyring.gpg

$ sudo curl -sSL \
  https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
  -o /usr/share/keyrings/ros-archive-keyring.gpg

$ echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] \
  http://packages.ros.org/ros2/ubuntu \
  $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | \
  sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

ROS Simulation (Cont.)

Launch Empty World Simulation

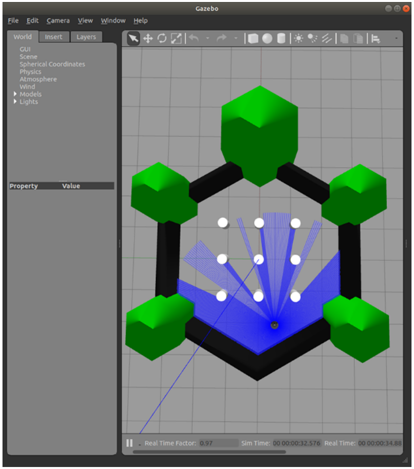
```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo empty_world.launch.py
```



ROS Simulation (Cont.)

or Launch Turtlebot3 World Simulation

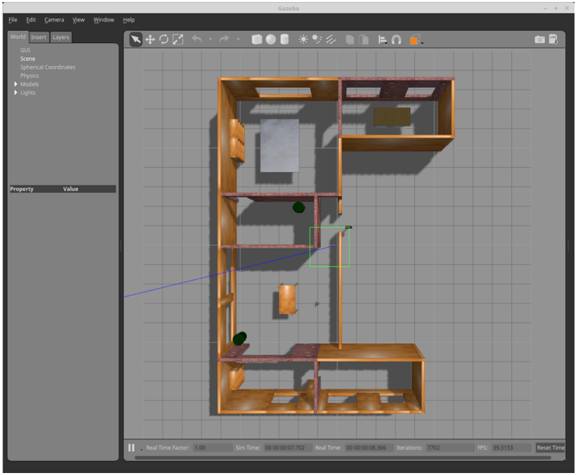
```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```



ROS Simulation (Cont.)

or Launch Turtlebot3 House Simulation

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```



ROS Simulation (Cont.)

Open a new terminal and run the teleoperation node to move the robot in the simulation environment

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 run turtlebot3_teleop teleop_keyboard
```

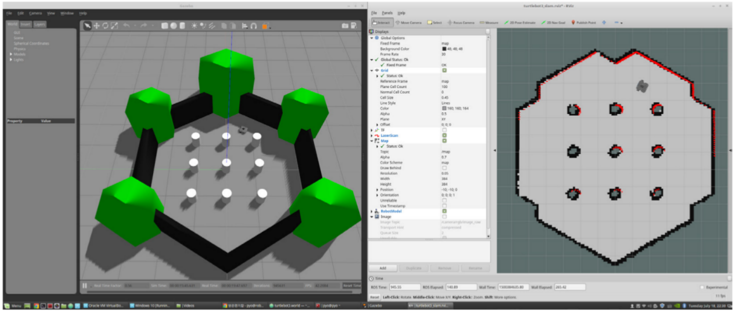
SLAM Simulation

Let's create a map with SLAM in the TurtleBot3 World

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Open a new terminal and run the SLAM node

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_cartographer cartographer.launch.py \
   use_sim_time:=True
```



SLAM Simulation (Cont.)

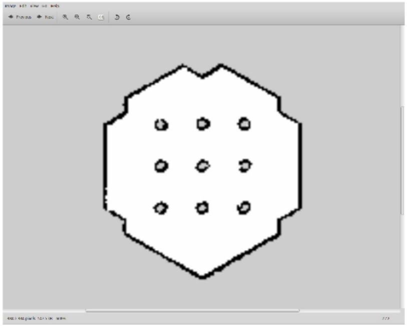
Open a new terminal and run the teleoperation node to move the robot in the simulation environment in order to create the map

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 run turtlebot3_teleop teleop_keyboard
```

Save the Map

When the map is created successfully, open a new terminal and save the map

```
$ ros2 run nav2_map_server map_saver_cli -f ~/map
```



Navigation Settings

Now we need to change a param file of our turtlebot from the package we have installed. Edit the burger.yaml file:

```
$ sudo nano \
 /opt/ros/humble/share/turtlebot3_navigation2/param/burger.yaml
```

Locate the parameter **robot_model_type** and change it from

```
    robot_model_type: "differential"
```

to:

```
    robot_model_type: "nav2_amcl::DifferentialMotionModel"
```

Save and close.

Navigation

The Turtlebot 3 World environment can be used for Navigation

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

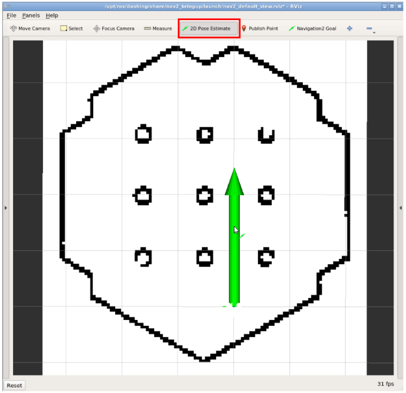
NOTE: Any other environment is fine after crating a map!

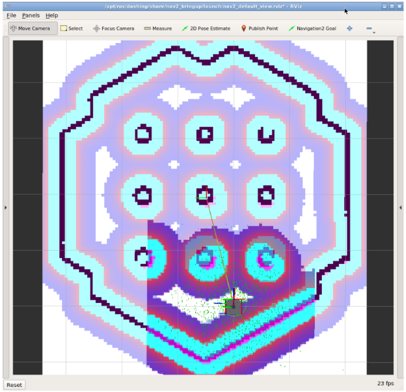
Open a new terminal and run the Navigation2 node

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py \
use_sim_time:=True map:=$HOME/map.yaml
```

Initial Pose Estimation

1. Click the 2D Pose Estimate button in the RViz2 menu
2. Click on the map where the actual robot is located and drag the large green arrow toward the direction where the robot is facing
3. Repeat step 1 and 2 until the LDS sensor data is overlayed on the saved map



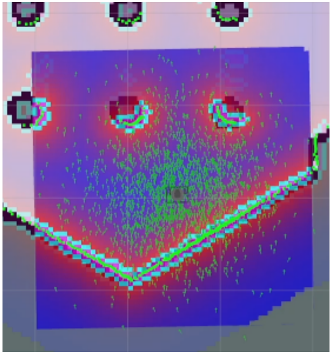


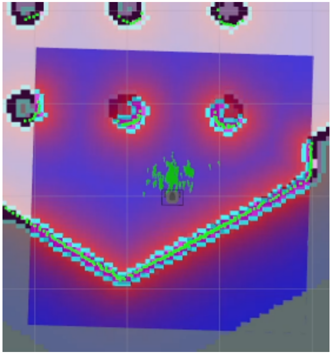
Initial Pose Estimation (Cont.)

Use teleoperation node to precisely locate the robot on the map

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 run turtlebot3_teleop teleop_keyboard
```

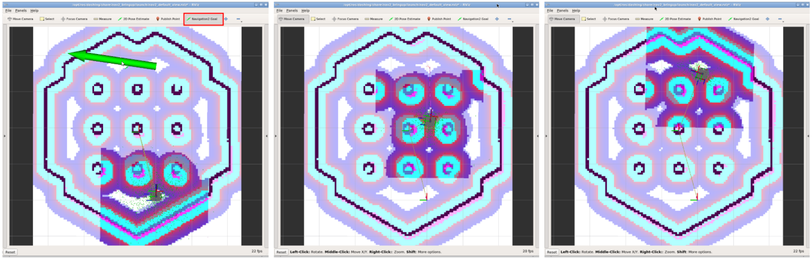
Move the robot back and forth a bit or rotate it to collect the surrounding environment information and narrow down the estimated location of the TurtleBot3 on the map which is displayed with tiny green arrows





Set Navigation Goal by Hand

- Click the Navigation2 Goal button in the RViz2 menu.
- Click on the map to set the destination of the robot and drag the green arrow toward the direction where the robot will be facing.
    - The green arrow is a marker specifying the destination of the robot
    - The root of the arrow is x, y coordinate of the destination, and the angle θ is determined by the orientation of the arrow
    - As soon as x, y, θ are set, TurtleBot3 will start moving to the destination immediately



Autonomous Exploration

Use the Explore Lite package for autonomous exploration.
Download the package from Github (or from Virtuale) and build it

```
$ cd ~/turtlebot3_ws/src
$ git clone https://github.com/robo-friends/m-explore-ros2.git
$ cd ..
$ colcon build
$ . install/local_setup.bash
```

Launch Simulation (try also other environments)

```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Explore Lite requires the navigation stack to work. So launch Navigation with SLAM (Simultaneous Localization and Mapping) and no map

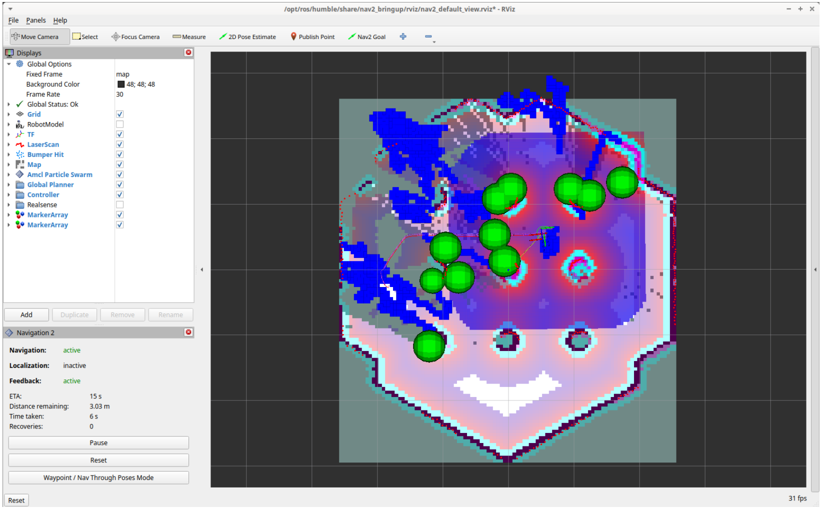
```
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py \
  use_sim_time:=True slam:=True
```

Launch Explore Lite

```
$ ros2 launch explore_lite explore.launch.py
```

Autonomous Exploration (Cont.)

Activate visualization of /explore/frontiers to visualize exploration frontiers and markers



Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30

Navigation: active
Localization: inactive
Feedback: active

ETA: 15 s
Distance remaining: 3.03 m
Time taken: 6 s
Recoveries: 0

Create a launch file for autonomous exploration

Create a new python package

```
$ cd ~/turtlebot3_ws/src
$ ros2 pkg create --build-type ament_python \
    autonomous_exploration
```

Create the launch directory and the launch file

```
    $ cd autonomous_exploration
    $ mkdir launch
    $ cd launch
    $ nano autonomous_exploration.launch.py
```

You can use a different text editor in place of nano at your convenience

Structure of the Python launch file

The import section is required to load all the libraries

```python
import os

from ament_index_python.packages
        import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources
        import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
```

Import the Turtlebot model

```python
TURTLEBOT3_MODEL = os.environ['TURTLEBOT3_MODEL']
```

Structure of the Python launch file (Cont.)

Declare the `generate_launch_description()`

```
def generate_launch_description():
```

Define default parameters

```
    use_sim_time = LaunchConfiguration('use_sim_time', default='True')
    x_pose = LaunchConfiguration('x_pose', default='-2.0')
    y_pose = LaunchConfiguration('y_pose', default='-0.5')
```

Define Turtlebot parameters file

```
    param_file_name = TURTLEBOT3_MODEL + '.yaml'
    param_dir = os.path.join(
            get_package_share_directory('turtlebot3_navigation2'),
            'param',
            param_file_name)
```

Define RViz config file

```
    rviz_config_dir = os.path.join(
        get_package_share_directory('nav2_bringup'),
        'rviz',
        'nav2_default_view.rviz')
```

Structure of the Python launch file (Cont.)

Define Turtlebot simulation command with proper parameters

```
turtlebot3_world_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('turtlebot3_gazebo'),
            'launch'),
            '/turtlebot3_world.launch.py']),
    launch_arguments={
           'use_sim_time': use_sim_time,
           'x_pose': x_pose,
           'y_pose': y_pose
    }.items()
)
```

You may replace the simulation environment by replacing the `turtlebot3_world.launch.py` file with a different one, eventually from another package too

Structure of the Python launch file (Cont.)

Since Explore lite rely on the navigation stack, define navigation command with proper parameters

```
    navigation_cmd = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            os.path.join(
                get_package_share_directory('nav2_bringup'),
               'launch'), '/bringup_launch.py']),
            launch_arguments={
                'map' : '~/map.yaml',
                'use_sim_time': use_sim_time,
                'slam' : 'True',
                'params_file': param_dir}.items(),
         )
```

The `slam` parameter must be set to `True` to activate SLAM

The `map` parameter is need by `bringup_launch.py` but is not used in case `slam` is active

Structure of the Python launch file (Cont.)

Define RViz node command to visualize the results

```
rviz_cmd = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        arguments=['-d', rviz_config_dir],
        parameters=[{'use_sim_time': use_sim_time}],
        output='screen')
```

and the Explore Lite launch command

```
explore_lite_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
             get_package_share_directory('explore_lite'), 'launch'),
             '/explore.launch.py']),
         launch_arguments={
            'use_sim_time': use_sim_time}.items(),
)
```

Structure of the Python launch file (Cont.)

Finally, create the `LaunchDescription()` class, include all the commands and return the class output

```
    ld = LaunchDescription()

    # Add the commands to the launch description
    ld.add_action(turtlebot3_world_cmd)
    ld.add_action(navigation_cmd)
    ld.add_action(rviz_cmd)
    ld.add_action(explore_lite_cmd)

    return ld
```

Save the file

Setting up the package for the launch file

Include the following line in the `package.xml` file

```
<exec_depend>ros2launch</exec_depend>
```

Save the file

Modify the `setup.py` file including all launch files

```
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    # Include all launch files.
         (os.path.join('share', package_name, 'launch'),
              glob(os.path.join('launch', '*launch.[pxy][yma]*')))
    ],
```

Save the file

Building and testing the package

From the console

```
$ cd ~/turtlebot3_ws
$ colcon build
$ . install/local_setup.bash
```

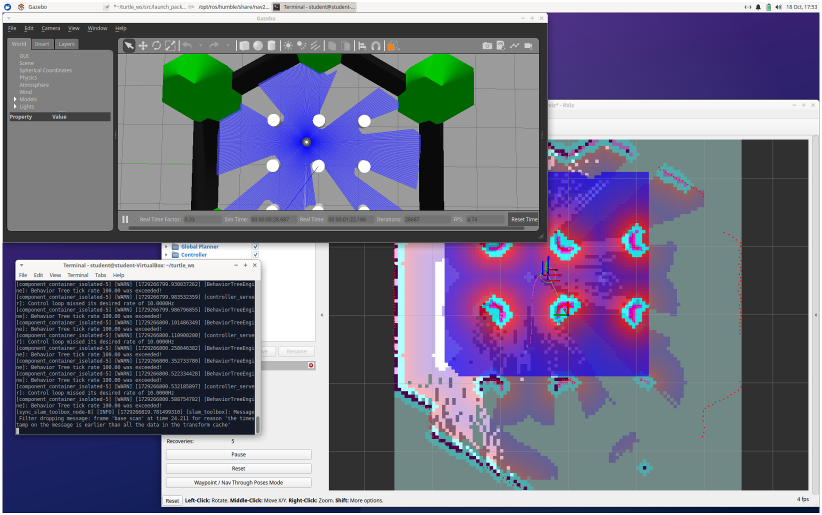
Launch the newly created launch file

```
$ ros2 launch autonomous_exploration \
   autonomous_exploration.launch.py
```

You may also play around with the robot position in the environment

```
$ ros2 launch autonomous_exploration \
   autonomous_exploration.launch.py x_pose:=0.5 y_pose:=2.0
```

Building and testing the package



Terminal - student@student-VirtualBox: ~/turtle_ws

```
[component_container_isolated-5] [WARN] [1729266799.930037262] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[component_container_isolated-5] [WARN] [1729266799.983532359] [controller_serve
r]: Control loop missed its desired rate of 10.0000Hz
[component_container_isolated-5] [WARN] [1729266799.986796855] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[component_container_isolated-5] [WARN] [1729266800.101486349] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[component_container_isolated-5] [WARN] [1729266800.110900200] [controller_serve
r]: Control loop missed its desired rate of 10.0000Hz
[component_container_isolated-5] [WARN] [1729266800.258646382] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[component_container_isolated-5] [WARN] [1729266800.352733780] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[component_container_isolated-5] [WARN] [1729266800.522334428] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[component_container_isolated-5] [WARN] [1729266800.532185897] [controller_serve
r]: Control loop missed its desired rate of 10.0000Hz
[component_container_isolated-5] [WARN] [1729266800.588754782] [BehaviorTreeEngi
ne]: Behavior Tree tick rate 100.00 was exceeded!
[sync_slam_toolbox_node-8] [INFO] [1729266819.781499310] [slam_toolbox]: Message
 Filter dropping message: frame 'base_scan' at time 24.211 for reason 'the times
tamp on the message is earlier than all the data in the transform cache'
```

Testing the Bighouse Environment

Download the package from virtuale and build it

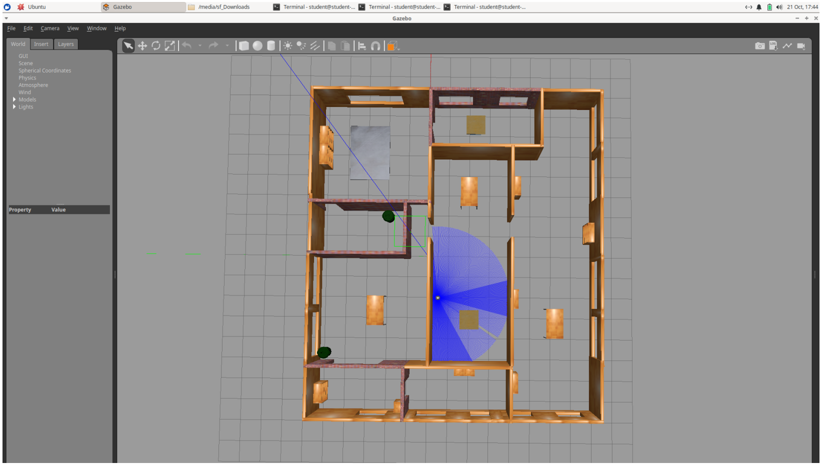
```
$ cd ~/turtlebot3_ws/src
$ wget --no-check-certificate https://tinyurl.com/54we4chu \
  -O turtlebot3_simulations.zip
$ unzip turtlebot3_simulations.zip
$ cd ..
$ colcon build
```

Install and launch

```
$ . install/local_setup.bash
$ ros2 launch turtlebot3_gazebo turtlebot3_bighouse.launch.py
```

Then activate autonomous exploration and save the map at the end

Building and testing the package



Autonomous Robot Localization

We will now create a node to perform self localization by checking the estimated pose covariance

Create a new python package

```
$ cd ~/turtlebot3_ws/src
$ ros2 pkg create --build-type ament_python \
    autonomous_localization
```

Create the a python script

```
    $ cd autonomous_localization/autonomous_localization
    $ nano autonomous_localization.py
```

You can use a different text editor in place of nano at your convenience

Autonomous Robot Localization

The import section is required to load all the libraries

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseWithCovarianceStamped, Twist
import time
import numpy as np
from nav_msgs.msg import Odometry
```

Define some useful parameter

```
TWO_PI = 6.28
ROTATION_VELOCITY = - 0.6
```

Autonomous Robot Localization

Create the Node Class

```
class InitialPositionNode(Node):

    def __init__(self):
        super().__init__('initial_position_node')
```

Create the AMCL subscriber to get the initial position

```
        self.odom_subscription = self.create_subscription(
           PoseWithCovarianceStamped, 'amcl_pose',
           self.amcl_callback, 10)
```

Create the initialpose publisher

```
        self.amcl_pose_publisher = self.create_publisher(
            PoseWithCovarianceStamped, 'initialpose', 10)
```

Subscriber to the odometry to get the robot current position estimate

```
        self.odometry_subscription = self.create_subscription(
            Odometry, 'odom', self.odom_callback, 10)
```

Autonomous Robot Localization

Create the publisher for the robot command

```
self.publisher = self.create_publisher(Twist, "cmd_vel", 10)
```

Initialize parameters and messages

```
self.covariance_treshold = 0.07

self.covariance_msg = PoseWithCovarianceStamped()
self.covariance_values = PoseWithCovarianceStamped()

self.tb3_pose = [0, 0, 0]
self.tb3_orientation = [0, 0, 0, 0]
```

Wait for the initial position to be obtained

```
self.get_logger().info('Waiting␣for␣the␣initial␣position...')
# Spin once to handle callbacks
rclpy.spin_once(self, timeout_sec=1)
```

Autonomous Robot Localization

In the AMCL callback, store the position covariance

```
    def amcl_callback(self, msg):
        self.amcl_position = msg.pose.pose.position
        self.amcl_orientation = msg.pose.pose.orientation
        self.covariance_msg.pose.covariance = msg.pose.covariance
```

Store the robot position in odometry callback

```
    def odom_callback(self, odom_msg):
        x_o = odom_msg.pose.pose.orientation.x
        y_o = odom_msg.pose.pose.orientation.y
        z_o = odom_msg.pose.pose.orientation.z
        w_o = odom_msg.pose.pose.orientation.w

        cov = np.zeros(36, dtype=np.float64)
        cov[0] = 200
        cov[7] = 200
        cov[35] = 1
        self.covariance_values = cov

        self.tb3_orientation = [x_o, y_o, z_o, w_o]
```

Autonomous Robot Localization

Publish the initial guess position on the topic initialpose

```
    def publish_initial_pose(self):
        rclpy.spin_once(self, timeout_sec = 1)
        initial_pose_msg = PoseWithCovarianceStamped()
        initial_pose_msg.pose.pose.position.x = float(self.tb3_pose[0])
        initial_pose_msg.pose.pose.position.y = float(self.tb3_pose[1])
        initial_pose_msg.pose.pose.position.z = float(self.tb3_pose[2])
        initial_pose_msg.pose.pose.orientation.x =
          float(self.tb3_orientation[0])
        initial_pose_msg.pose.pose.orientation.y =
          float(self.tb3_orientation[1])
        initial_pose_msg.pose.pose.orientation.z =
          float(self.tb3_orientation[2])
        initial_pose_msg.pose.pose.orientation.w =
          float(self.tb3_orientation[3])
        initial_pose_msg.pose.covariance = self.covariance_values
        initial_pose_msg.header.frame_id = 'map'
        initial_pose_msg.header.stamp = self.get_clock().now().to_msg()

        self.amcl_pose_publisher.publish(initial_pose_msg)
```

Autonomous Robot Localization

Check if localization is completed

```
def localization(self):
    while True:
        print('Localization␣in␣progress...')
        self.rotate()
        # update the covariance matrix from the topic amcl
        rclpy.spin_once(self, timeout_sec=1)

        if self.check_covariance():
            break
```

Autonomous Robot Localization

Check if the pose covariance is under the threshold

```
    def check_covariance(self):
        covariance_values = self.covariance_msg.pose.covariance

        if np.max(covariance_values) < self.covariance_treshold:
            self.get_logger().info("Covariance␣below␣the␣threshold.")
            self.get_logger().info("Robot␣is␣localized.")
            return True
        else:
            self.get_logger().warn("Covariance␣above␣the␣threshold.")
            return False
```

Autonomous Robot Localization

Define motion commands

```
    def rotate(self):
        vel_msg = Twist() #create a velocity message
        vel_msg.angular.z = ROTATION_VELOCITY
        self.publisher.publish(vel_msg)
        self.get_logger().info('Publishing:␣"%s"' % vel_msg)
        time.sleep(- TWO_PI / vel_msg.angular.z) # rotate for some time
        self.stop() # stop the robot to check the covariance

    def stop(self):
        vel_msg = Twist() #create a velocity message
        vel_msg.angular.z = 0.0
        vel_msg.linear.x = 0.0
        self.publisher.publish(vel_msg)
        self.get_logger().info('Publishing:␣"%s"' % vel_msg)
```

Autonomous Robot Localization

Main function

```
def main():
    rclpy.init()
    initial_position_node = InitialPositionNode()
    time.sleep(5)
    initial_position_node.publish_initial_pose()
    initial_position_node.localization()
    rclpy.shutdown()

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

Save the file

Modify the setup.py file including the entry point

```
    entry_points={
        'console_scripts': [
          'autonomous_localization=
␣␣␣␣␣␣␣␣␣␣␣␣␣autonomous_localization.autonomous_localization:main'
        ],
```

Building and testing the autonomous localization node

From the console

```
$ cd ~/turtlebot3_ws
$ colcon build
$ . install/local_setup.bash
```

Launch the simulation

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

You may also play around with the robot position in the environment
Open a new terminal and run the Navigation2 node

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py \
use_sim_time:=True map:=$HOME/map.yaml
```

Launch the newly created node in a new terminal

```
$ ros2 run autonomous_localization autonomous_localization
```

Building and testing the autonomous localization node

Try different motion strategies for localization:

- Alternated rotation and linear motions
- Braitenberg motion
- Wall following (bug algorithms)

Service client for localization reset

Implement a service client called `reset_node.py` to reset localization through the service call to `/reinitialize_global_localization`

```
import rclpy
from rclpy.node import Node
from std_srvs.srv import Empty


class MinimalClientAsync(Node):
    def __init__(self):
        super().__init__('reinitialize_global_localization_client')
        self.cli = self.create_client(Empty,
           'reinitialize_global_localization')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service␣not␣available,␣waiting...')
        self.req = Empty.Request()

    def send_request(self):
        self.future = self.cli.call_async(self.req)
```

Service client for localization reset

Continue...

```
def main():
    rclpy.init()
    minimal_client = MinimalClientAsync()
    minimal_client.send_request()
    while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
            except Exception as e:
                minimal_client.get_logger().info('Call␣failed␣%r' % (e,))
            else:
                minimal_client.get_logger().info('Call␣succeded!')
            break
    minimal_client.destroy_node()
    rclpy.shutdown()

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

Building and testing the reset node

Modify the `setup.py` file including the entry point

```
    entry_points={
       'console_scripts': [
         'autonomous_localization=
␣␣␣␣␣␣␣␣␣␣␣␣␣autonomous_localization.autonomous_localization:main',
         'reset_node=autonomous_localization.reset_node:main'
       ],
```

Save the file
From the console

```
$ cd ~/turtlebot3_ws
$ colcon build
$ . install/local_setup.bash
```

Launch the newly created node in a new terminal

```
$ ros2 run autonomous_localization reset_node
```

Send navigation goals stored in a text file

We will now create a node to snd navigation goals stored in a text file

Create a new python package

```
$ cd ~/turtlebot3_ws/src
$ ros2 pkg create --build-type ament_python \
   navigation_from_text
```

Create the a python script

```
    $ cd navigation_from_text/navigation_from_text
    $ nano navigation_from_text.py
```

You can use a different text editor in place of nano at your convenience

Send navigation goals stored in a text file

Create a navigation action client to send navigation goals read from a text file

The import section is required to load all the libraries

```
import rclpy
import time
from rclpy.node import Node
import os

from geometry_msgs.msg import PoseStamped
from tf2_ros import Duration

from action_msgs.msg import GoalStatus
from lifecycle_msgs.srv import GetState
from nav2_msgs.action import NavigateToPose
from geometry_msgs.msg import PoseStamped
from rclpy.action import ActionClient
```

Send navigation goals stored in a text file

Create the action client class

```
class NavigatorFromTextfile(Node):

    def __init__(self):
        super().__init__('action_server_control')

        self.nav_to_pose_client = ActionClient(self,
            NavigateToPose, 'navigate_to_pose')

        # Wait for the action server to be ready
        self.action_client.wait_for_server()

        print('Init␣completed')
```

Send navigation goals stored in a text file

Send a `NavToPose` action request and waits for completion

```
def goToPose(self, pose):
   self.debug("Waiting␣for␣'NavigateToPose'␣action␣server")
   while not self.nav_to_pose_client.wait_for_server(timeout_sec=1.0):
       self.info("'NavigateToPose'␣server␣not␣available,␣waiting...")

   goal_msg = NavigateToPose.Goal()
   goal_msg.pose = pose
   self.info('Navigating␣to␣goal:␣' + str(pose.pose.position.x)
       + '␣' + str(pose.pose.position.y) + '...')
   send_goal_future = self.nav_to_pose_client.send_goal_async(
                 goal_msg, self._feedbackCallback)
   rclpy.spin_until_future_complete(self, send_goal_future)
   self.goal_handle = send_goal_future.result()
   if not self.goal_handle.accepted:
       self.error('Goal␣to␣' + str(pose.pose.position.x) + '␣' +
                 str(pose.pose.position.y) + '␣was␣rejected!')
       return False
   self.result_future = self.goal_handle.get_result_async()
     return True
```

Send navigation goals stored in a text file

```
def isNavComplete(self):
    if not self.result_future: # task was cancelled or completed
        return True
    rclpy.spin_until_future_complete(self, self.result_future,
        timeout_sec=0.10)
    if self.result_future.result():
        self.status = self.result_future.result().status
        if self.status != GoalStatus.STATUS_SUCCEEDED:
            self.info('Goal␣with␣failed␣with␣status␣code:␣{0}'
               .format(self.status))
            return True
    else: # Timed out, still processing, not complete yet
        return False
    return True

def getFeedback(self):
    return self.feedback

def getResult(self):
    return self.status
```

Send navigation goals stored in a text file

Wait for the action server

```
    def waitUntilNav2Active(self):
        self.info('Wait␣for␣Nav2␣to␣be␣ready...')
        self._waitForNodeToActivate('amcl')
        self._waitForNodeToActivate('bt_navigator')
        self.info('Nav2␣is␣ready␣for␣use!')
        return
```

Send navigation goals stored in a text file

Waits for the node within the tester namespace to become active

```
def _waitForNodeToActivate(self, node_name):
    self.debug('Waiting␣for␣' + node_name + '␣to␣become␣active..')
    node_service = node_name + '/get_state'
    state_client = self.create_client(GetState, node_service)
    while not state_client.wait_for_service(timeout_sec=1.0):
        self.info(node_service + '␣service␣not␣available,␣waiting...

    req = GetState.Request()
    state = 'unknown'
    while (state != 'active'):
        self.debug('Getting␣' + node_name + '␣state...')
        future = state_client.call_async(req)
        rclpy.spin_until_future_complete(self, future)
        if future.result() is not None:
            state = future.result().current_state.label
            self.debug('Result␣of␣get_state:␣%s' % state)
        time.sleep(2)
    return
```

Send navigation goals stored in a text file

Default callbacks

```
def _feedbackCallback(self, msg):
     self.feedback = msg.feedback
     return

 def info(self, msg):
     self.get_logger().info(msg)
     return

 def warn(self, msg):
     self.get_logger().warn(msg)
     return

 def error(self, msg):
     self.get_logger().error(msg)
     return

 def debug(self, msg):
     self.get_logger().debug(msg)
     return
```

Send navigation goals stored in a text file

Print the time to arrival and, in case, abort the navigation if it's taking too long

```
    def navigation(self):
        i = 0
        while not self.isNavComplete():
            i = i + 1
            feedback = self.getFeedback()
            if feedback and i % 5 == 0:
                print('Estimated␣time␣of␣arrival:␣' + '{0:.0f}'.format(
                   Duration.from_msg(feedback.estimated_time_remaining).
                   nanoseconds / 1e9)
                    + '␣seconds.')
                 if Duration.from_msg(feedback.navigation_time) >
                    Duration(seconds=600.0):
                    self.cancelNav()
```

Send navigation goals stored in a text file

Continue...

```
        result = self.getResult()
        if result == GoalStatus.STATUS_SUCCEEDED:
            self.info('Goal␣succeeded!')
        elif result == GoalStatus.STATUS_CANCELED:
            self.info('Goal␣was␣canceled!')
        elif result == GoalStatus.STATUS_ABORTED:
            self.info('Goal␣failed!')
        else:
            self.info('Goal␣has␣an␣invalid␣return␣status!')
```

Send navigation goals stored in a text file

Main function

```
def main(args=None):
    rclpy.init()
    navigator = NavigatorFromTextfile()
    path_to_textfile =
      "/home/student/turtlebot3_ws/src/navigation_from_text/goals.txt"
    if os.path.exists(path_to_textfile):
        with open(path_to_textfile, "r") as file:
            lines = file.readlines()
            for line in lines:
                goal = line.split()
                goal_x = float(goal[0])
                goal_y = float(goal[1])
                goal_w = float(goal[2])
                goal_pose = PoseStamped()
                goal_pose.header.frame_id = 'map'
                goal_pose.header.stamp =
                   navigator.get_clock().now().to_msg()
```

Send navigation goals stored in a text file

Continue...

```
                goal_pose.pose.position.x = float(goal_x)
                goal_pose.pose.position.y = float(goal_y)
                goal_pose.pose.orientation.w = float(goal_w)
                navigator.goToPose(goal_pose)
                navigator.navigation()
                time.sleep(2) # wait a little
            navigator.info("All␣the", str(len(lines)) ,"goals␣processed")
    else:
        navigator.info("The␣specified␣path␣doesn’t␣exist")

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

Send navigation goals stored in a text file

Modify the `setup.py` file including the entry point

```
    entry_points={
        'console_scripts': [
          'navigation_from_text=
␣␣␣␣␣␣␣␣␣␣␣␣␣navigation_from_text.navigation_from_text:main'
        ],
```

Save the file

Create the navigation goals file `goals.txt` in the package base directory

```
0.5 0.5 1.0
0.5 -0.5 1.0
-0.5 0.5 1.0
```

Change and add waypoints and test the result

Building and testing the autonomous localization node

From the console

```
$ cd ~/turtlebot3_ws
$ colcon build
$ . install/local_setup.bash
```

Launch the simulation

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

You may also play around with the robot position in the environment
Open a new terminal and run the Navigation2 node

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py \
use_sim_time:=True map:=$HOME/map.yaml
```

Launch the newly created node in a new terminal

```
$ ros2 run navigation_from_text navigation_from_text
```

Testing the Wall Follower on the World and Bighouse Environment

Download the package from virtuale and build it

```
$ cd ~/turtlebot3_ws/src
$ wget --no-check-certificate https://tinyurl.com/2xu9h5dn \
  -O wall_follower_ros2.zip
$ unzip wall_follower_ros2.zip
$ cd ..
$ colcon build
```

Install and launch

```
$ . install/local_setup.bash
$ ros2 run wall_follower_ros2 wall_follower_ros2 \
  --ros-args --log-level DEBUG --log-level rcl:=INFO
```

Test also the launch files
Then activate simulation environments

Thanks!

Questions?

The only stupid question is the one you were afraid to ask but never did.
-Rich Sutton