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

```
$ 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 foxy -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/master\
  /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

```
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 \
    /master/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

```
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 foxy -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-foxy-gazebo-*
```

Install Cartographer

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

Install Navigation2

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

Install TurtleBot3 Packages

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

ROS Simulation (Cont.)

Install Turtlebot3 simulation packages

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

ROS Simulation (Cont.)

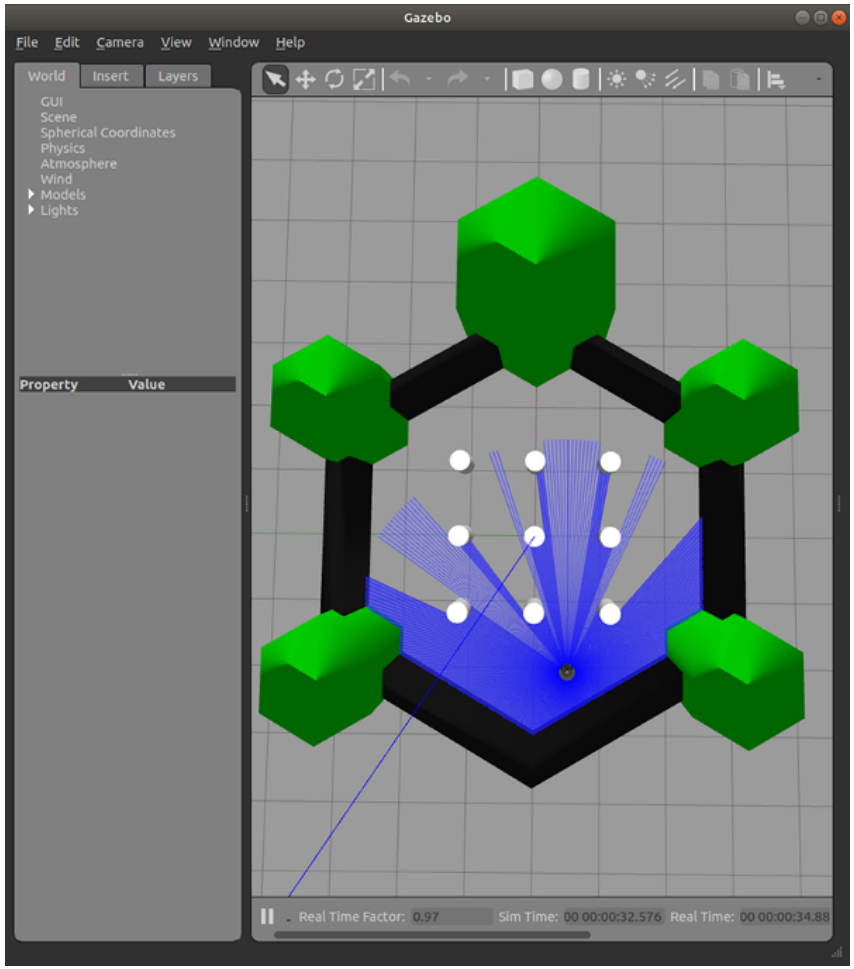
Launch Empty World Simulation

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

ROS Simulation (Cont.)

or Launch Turtlebot3 World Simulation

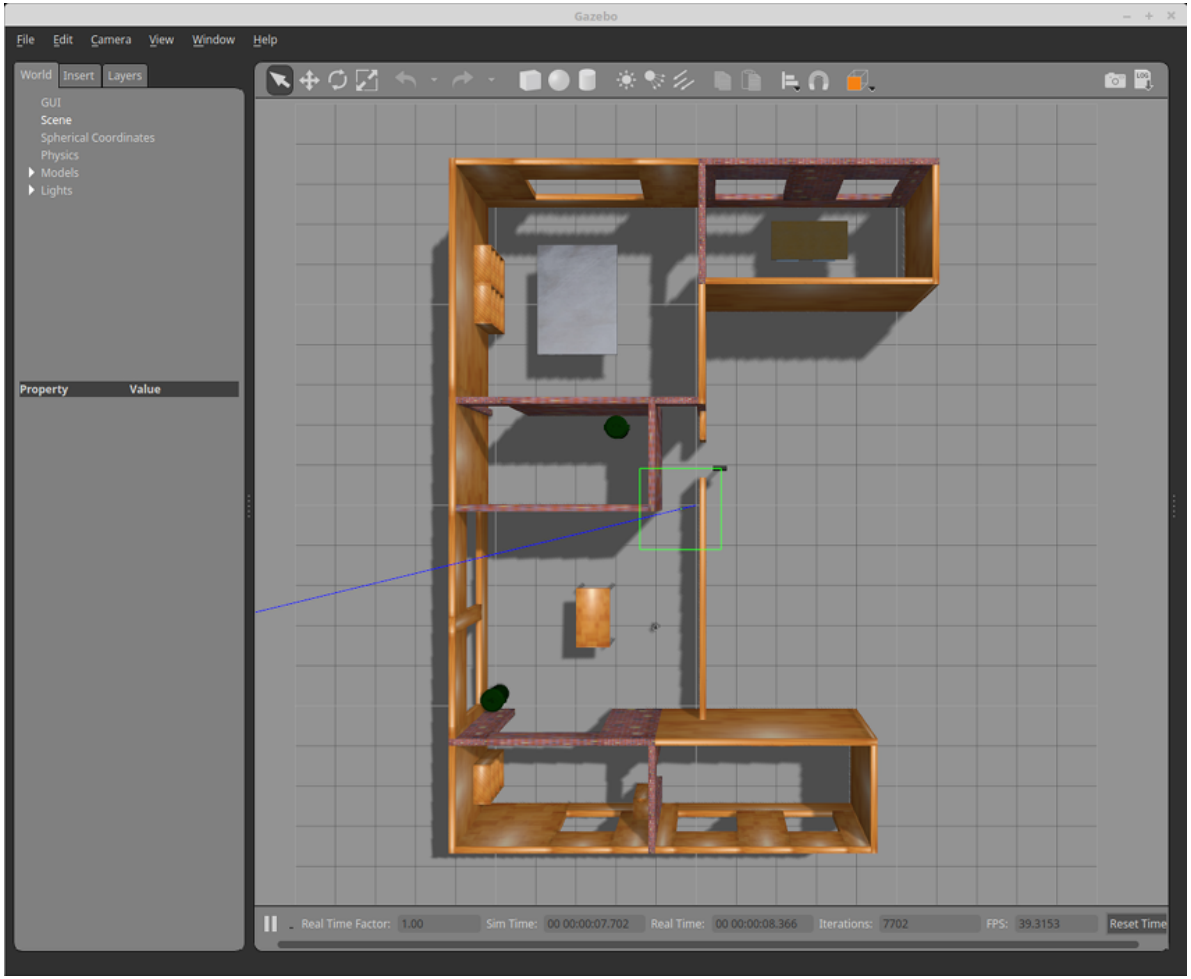
```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```



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
```

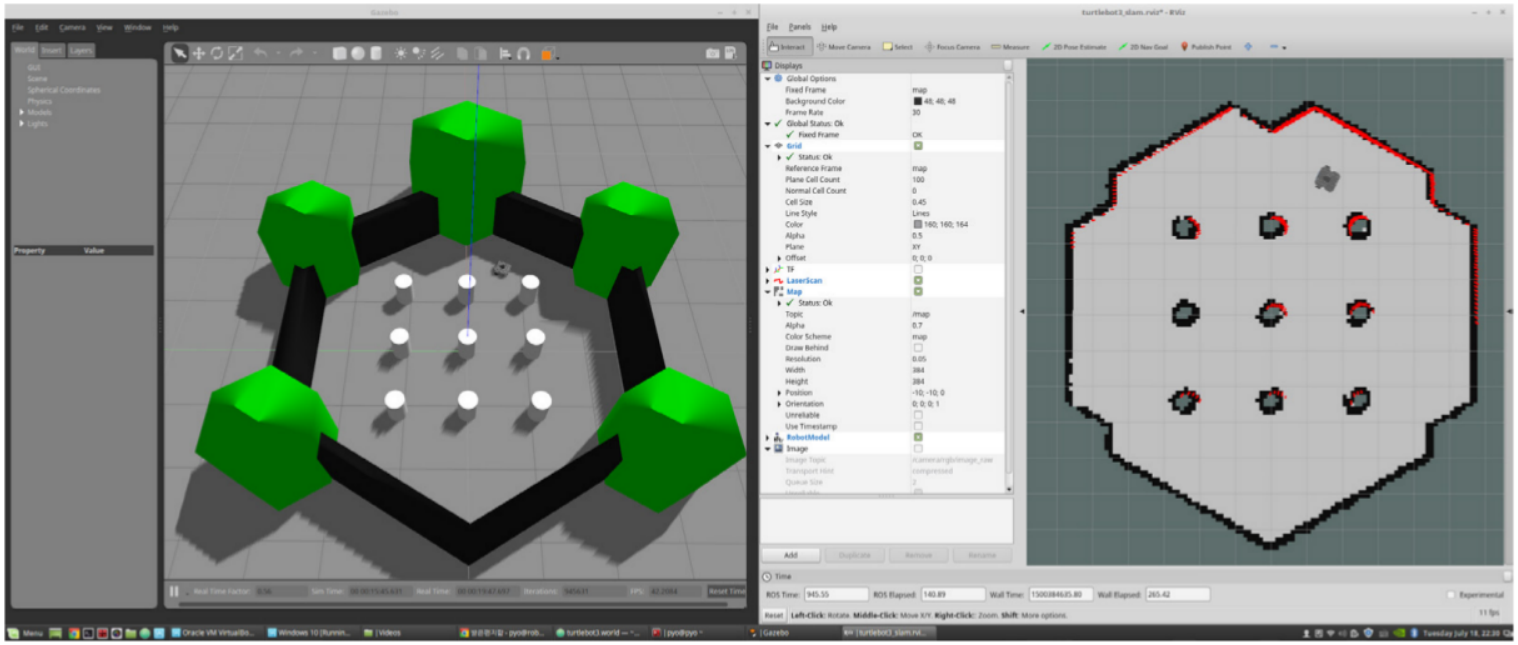
SLAM Simulation

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
```



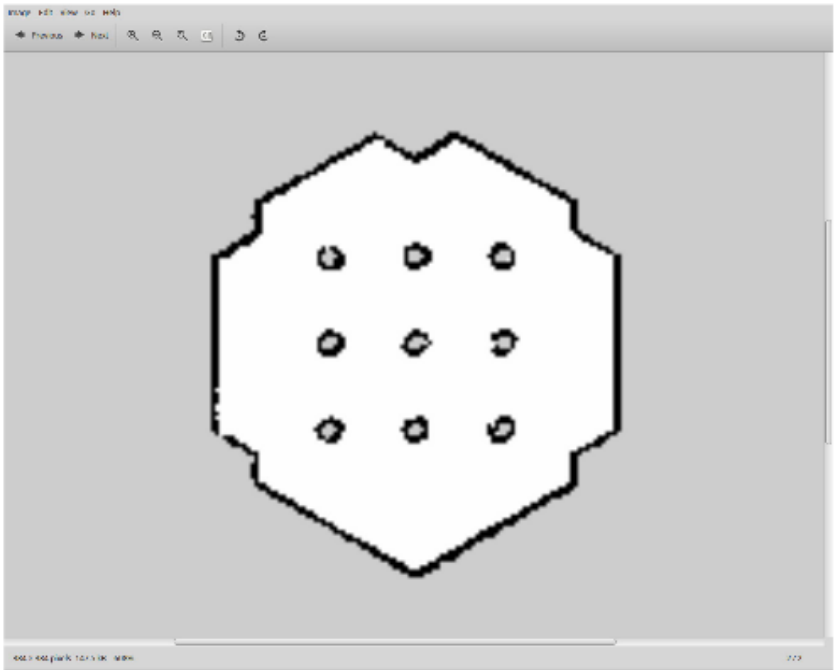
SLAM Simulation (Cont.)

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
```

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

The same Gazebo environment will be used for Navigation

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

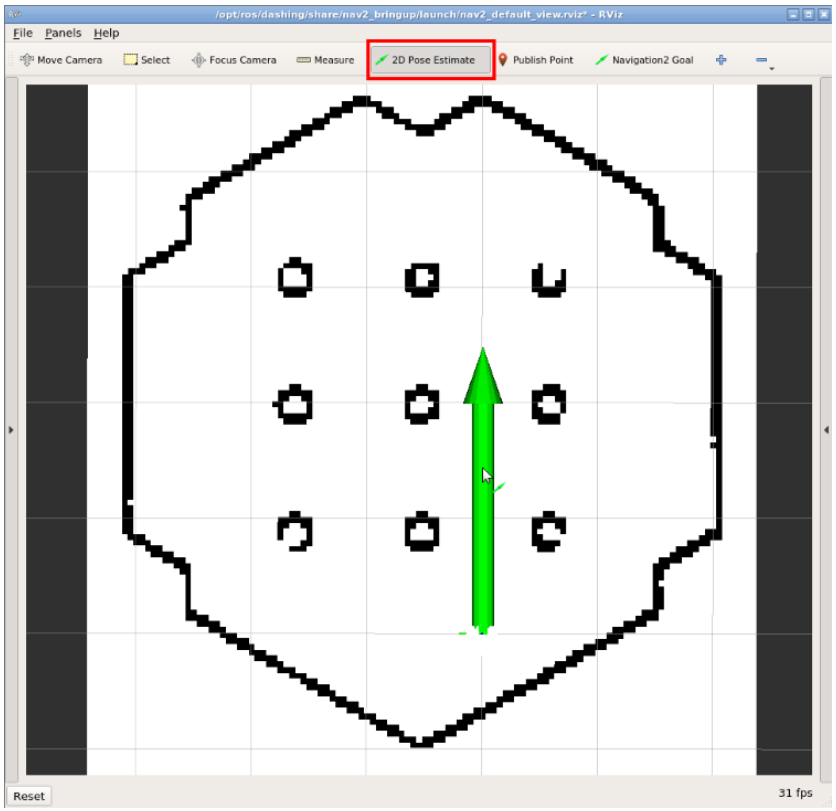
Open a new terminal and run the Navigation2 node

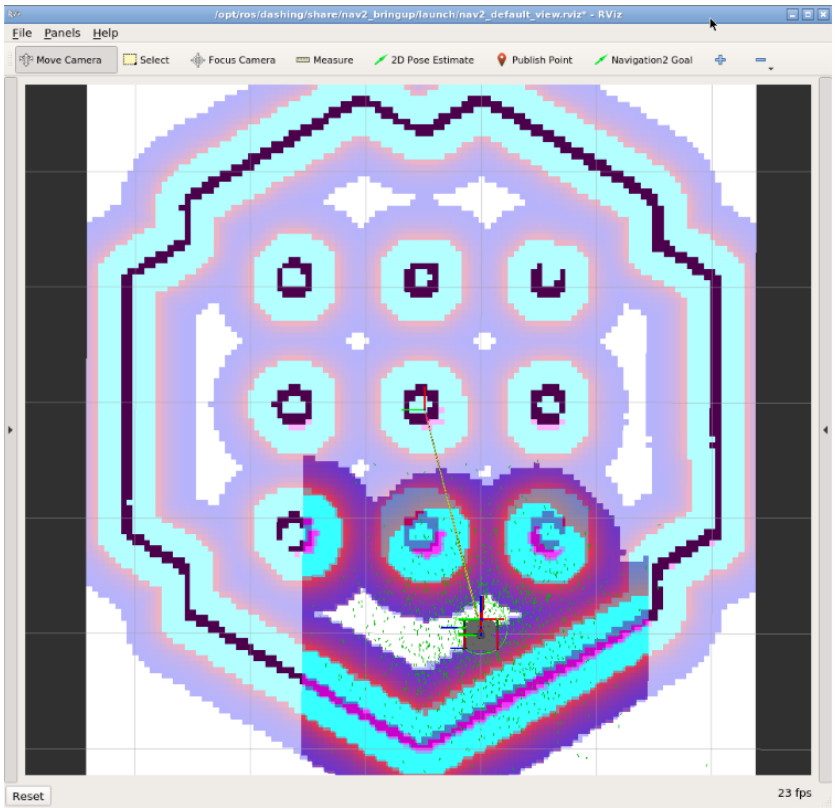
```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py \
   use_sim_time:=True map:=$HOME/map.yaml
```

Initial Pose Estimation must be performed before running the Navigation as this process initializes the AMCL parameters that are critical in Navigation. The robot has to be correctly located on the map with the LDS sensor data that neatly overlaps the displayed map

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



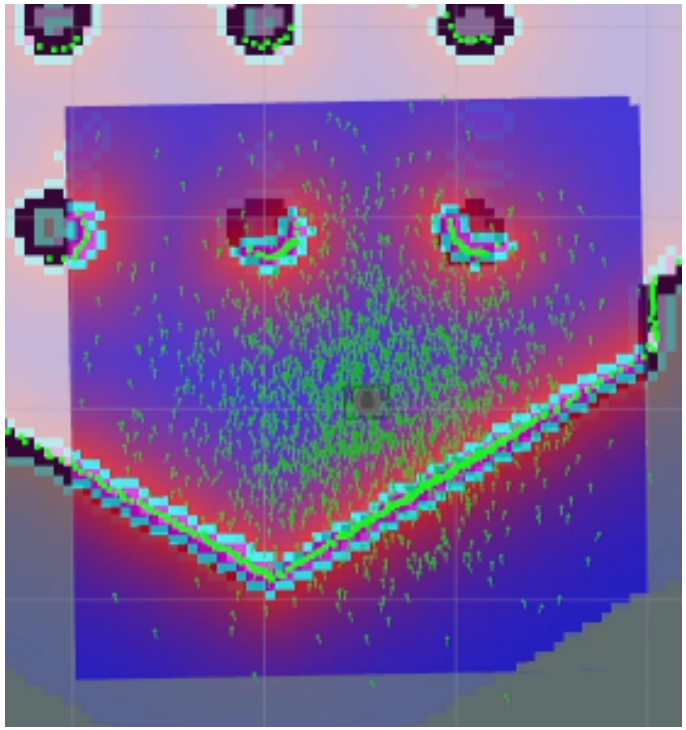


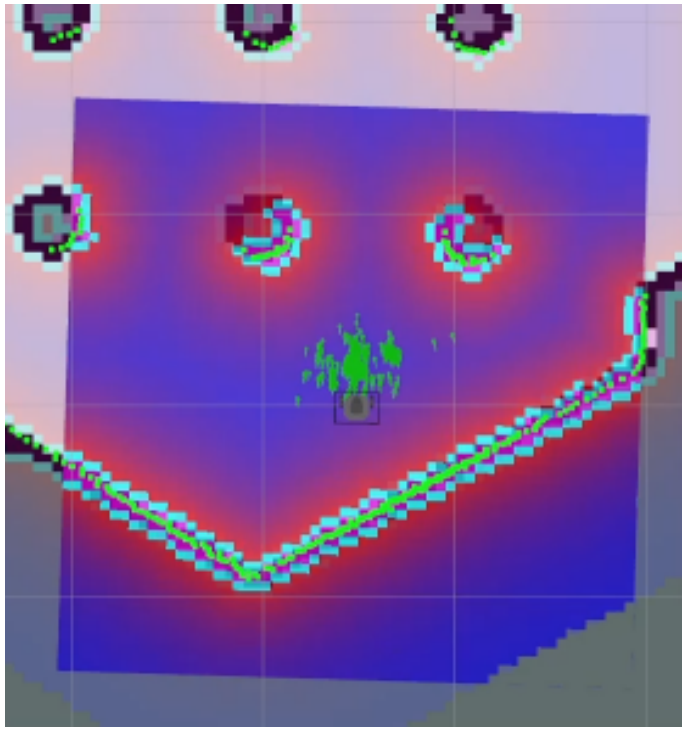
Initial Pose Estimation (Cont.)

Use teleoperation node to precisely locate the robot on the map

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

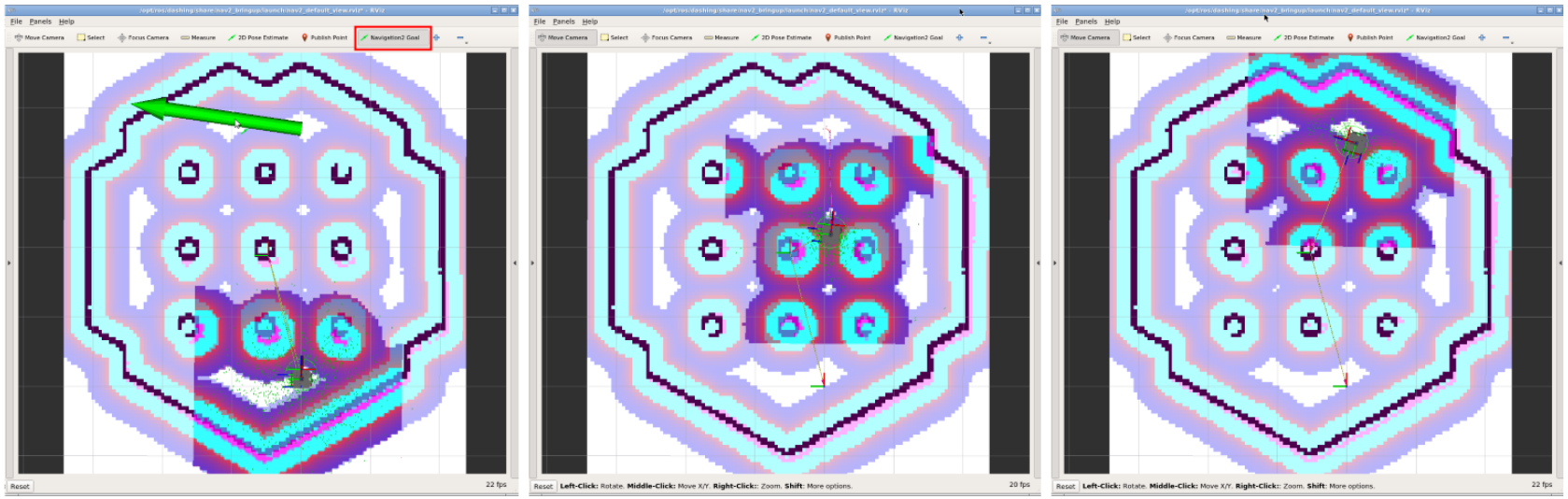
Move the robot back and forth a bit or rotate it to collect the surrounding environment information and narrow down the estimated location of the TurtleBot3 on the map which is displayed with tiny green arrows





Set Navigation Goal

- 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



Thanks!

Questions?

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