Lab 2: ROS Nodes and Simulation Models

In this lab you will learn about the structure of a ROS package that includes the simulation world, robot model, and ROS nodes that process sensor information and generate robot commands. We will use Turtlebot 3 robot simulation in a customized environment. The source code you cloned from your instructor's Git repository in the previous lab is a ROS package designed specifically for this lab and your assignment will be to show that you understand the code, followed by adding your own code to make the robot do the required task.

Setup

Remember that everything must run inside the container that you created in the previous lab, so don't forget to enter the container in every terminal window that you open:

```
distrobox enter humble-e6911
```

To position your working directory on the top of your workspace type:

```
go_colcon
```

This is the top of your workspace. The source code of the lab package is under `src/prob_rob_labs_ros_2` relative to the top of your workspace. Three additional directories under your workspace have been generated during the build: `build/`, `install/`, and `log/`. Do not hand-modify them. If you remove them, you must do a full rebuild using both `cc_prebuild` and `cc_build`. You must also rebuild your package each time you touch any C/C++ code or if you add a new ROS node (Python or C/C++) or a new message definition. If you are only modifying Python code of existing modules, you do not need to rebuild. Each time you rebuild the package, you must re-source the environment setup for the newly built nodes to become visible, which you can do by either logging out of the container and re-entering it (environment setup will be picked up from .bashrc in that case) or manually by rerunning the .bashrc:

```
source ~/.bashrc
```

Keep this process in mind, you will be doing it often. Let us now run the simulation we will use in this lab. Type this:

```
ros2 launch prob_rob_labs turtlebot3_and_door_launch.py
```

A simulation of a Turtlebot robot equipped with a camera (we will use it later) standing in front of a glass door will start up. Your objective for this lab is to examine the code that implements this simulation, find out to which topic you need to publish to open the door, and then write a ROS node that will open the door, make the robot move forward through the door, stop, and close the door behind it. Because we are not using any sensors yet, the robot will have no sense of its location, so your motion will have to be based on commanding a specific velocity, holding it for enough time to make sure that the robot went through the door and then changing the command to stop the robot. Unlike the previous lab, you will create a complete ROS node with the launch file and integrate it with the top-level launch file.

Your objective is to understand the full structure of a ROS package that includes simulation models, simulation plugin, robot code, and auxiliary files. You will learn how to use standard packages that come with ROS and combine them with your own nodes that you will add to the provided lab package. Understanding this material is essential for all subsequent labs, so do not take shortcuts. Take time to understand every part of the system.

Launch File

The launch file is a Python file that describes which nodes from which packages must be started to execute your application. A launch file may include other launch files from other packages, they may evaluate arguments passed from the command line, or set ROS parameters, but eventually everything leads to spawning node executables.

Arguments are useful to control the behavior of our application at run time. Nodes can be started and launch files included conditionally depending on the value of the arguments. To learn more about Launch files, read this page:

https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Launch-Main.html

Locate the launch file that you have been instructed to use in the previous section, examine it, make an effort to understand what each line does (use the documentation above) and do your first assignment.

Assignment 1: Answer the following questions in your lab report:

1. With all arguments set at default (nothing specified in the command line), which launch files and from which packages will be included? Which will be skipped?
2. Which Launch file and from which package will be included if `run_door_opener:=true` is appended to the `ros2 launch` command above?
3. At which coordinates in the XY plane will the robot be spawned if no arguments are specified?
4. What would the Launch command look like if you were to spawn the robot at coordinates (-5, 1)?

The purpose of the above assignment is to give you an idea of the application launching process. In subsequent labs you will be writing your own nodes and your own launch files and integrating them with existing applications.

Constructing the World

The world is constructed by Gazebo by passing it the World file. In our case, that file is constructed from the argument `world_name` which can be set by the user and the default value is `door`. So the world file we are using is located in `worlds/door.world` under the `prob_rob_labs` package (make sure you understand why before proceeding). This file contains some common properties of our world (e.g., location of light sources, direction and intensity of gravity, wind, position of the observer, etc.) and also specifies objects and their placement. The World file as well as individual object models follow the so-called SDF specification. The full specification relevant for describing the world is here:

http://sdformat.org/spec?ver=1.5&elem=world

Each included object leads to the SDF file that describes the object model. Our world is simple and consists of a few glass-wall segments and a glass door. You can easily tell from the World file that the model names are called `hinged_glass_door` and `glass_wall_segment`. Gazebo will look for these models at several presumed locations. There are also environment variables that tell Gazebo where to look for models. Gazebo also has the ability to identify, find, and download community-shared models available in its public repository on the Internet. The exact process of locating the model and resolving its name is complicated, but all you need to know is that our `prob_rob_labs` repository is set up so that Gazebo will find models placed in `models/` directory under `models` package. So, navigate to that directory and examine the two models used in our world.

Object Models and Plugins

First, examine the glass wall model. The central file in each model is the `model.sdf` file that describes the geometry, appearance, and physical properties of the object. The glass wall model is very simple and the two main sections that you can spot are the “collision” and “visual” section. The former describes the geometry of the body that can collide with other objects or with our robot. The latter describes how the object looks in the world. The two can differ. Because the wall object is a static rigid body, there is no need to describe the inertial properties. Deciding which objects are static and which are subject to world dynamics is essential for building an efficient model. If you make everything dynamic, the physics engine will have a more complex model to solve.

The door model is more interesting. Because it is a dynamic model, it has movable links attached on joints (the door hinge in this case) and inertial properties must be specified. This model has been derived from a publicly available model called hinged_door. It is the same model, except that the door is made of glass and the hinge is “motorized”.

Note that the SDF file only describes the physical properties of your model, but there are no inherent means to apply forces or torques to objects from a ROS node. This is where the Gazebo plugin comes into play. Our door is actually a simple model of the door with a motorized hinge. The “motor” that applies the torque to the hinge, thus opening or closing the door, is controllable from the ROS topic. The following line instantiates the plugin:

```
<plugin name="hinge_control_plugin" filename="libhinge_control.so">
</plugin>
```

The libhinge_control.so file is a dynamically loadable library that Gazebo loads. In each simulation step, the solver will pass the control to the code inside the plugin that models additional behavior of the model beyond simple dynamics. Typically sensors and actuators are modeled using plugins. This provides a great degree of flexibility to the models, because the plugin can do anything that the model developer designs it to do.

The source code of the plugin is in `models/src/hinge_control.cpp`, and the build environment has been set up to build the `.so` file and put it at the location where Gazebo’s model loader will find it. You can read about plugins here:

https://classic.gazebosim.org/tutorials?tut=plugins_hello_world&cat=write_plugin

The way our plugin works is that it creates a ROS node named after the model instance, subscribes to a ROS topic on which the user or another program can specify the torque, reads the specified torque from the topic and applies it to the hinge.

Assignment 2: Examine the plugin code and answer the following in your lab report:

1. In which line of which SDF file is the joint called “hinge” that the plugin controls specified and what is its type?
2. What is the name and mass of the link that hangs off the joint that the plugin is controlling?
3. The model name associated with the door is `hinged_glass_door`. What is the name and the type of the topic to which you must publish to open or close the door? Hint: look at the plugin source code. Even if you are not an expert in C++, you should be able to figure out where the code subscribes to the topic. Then use `ros2 topic list` and `ros2 topic info` to examine the topic and make sure you got it right. You can also use the ros2 `node list` and `ros2 node info` to examine the nodes.
4. Use the ros2 `topic pub` command to open the door. Hint: Type ros2 topic pub –help first to understand the command syntax and also read about the `ros2-topic` utility at https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html
5. What is the minimum torque you had to use for the door to open?

Your ROS Application

In this exercise, you will add a new node to the `prob_rob_labs` package. To do that, you must create several new files and modify existing files that control the build. The process is tedious and error-prone, but it can be easily automated. The script that does all of the above for you is available under `tools/` directory. Go to that directory and type:

```
new_ros_node <name_of_your_new_node>
```

Your new node is supposed to open the door and move the robot through the door, so give it a meaningful name. To see what the script did, you can type `git status` and it will show you new and modified files. After the “boilerplate” files are created you must rebuild. Type this:

```
cc_build
```

The above command is an alias that you should have added during the setup. To see what it actually does type ‘`alias cc_build`’ (it goes to the workspace, invokes Colcon tool to build, and returns to the original directory). This will build the new node and after re-sourcing your .bashrc the node will become visible. You can type this and the node will start but not do anything:

```
ros2 launch prob_rob_labs <your_node_name>_launch.py
```

The new node is just a boilerplate which will spin in the ROS loop waiting for events that will never happen because the node is not set up to process any events nor does it subscribe to any topics. Examine the code and note its structure. After initializing the node in `main()` function the program instantiates the class that implements the node. This is where you will be adding your code by implementing new methods and attributes. The boiler node simply sets a timer callback that enters the `heartbeat()` function periodically (which simply logs the entry and does nothing).

To make the node do something, initialize publishers and subscribers in the constructor and implement subscriber callbacks. If you need something to execute periodically, you can use the timer callback. General architecture of the ROS node is that the callbacks process the incoming data and store the results in some globally visible variables (typically in class attributes). Processed data can be published from the same or another topic callback or in the timer callback.

Assignment 3: Add the code to the boilerplate node that you have just created to perform the following task:

1. Open the door.
2. Move the robot through the door.
3. Stop the robot once it went through the door.
4. Close the door behind.

Submit the code and a video screencast that shows that your program works.

Hint: Create the publishers in the constructor (you need a publisher for door control topic and robot velocity command topic), then implement the task as a state machine and progress the state inside the heartbeat callback.

Submit the code with your lab report. The robot may not push the door open and you may not use command-line tools while your program is running to hand-open the door for the robot. Your program must properly open the door by publishing to the topic.

We are still not using any sensors, so your control node will be rather simplistic. It will run in an open loop and simply rely on elapsed time to know when to move, stop, and when to manipulate the door and it will make the robot drive straight and hope that it does not skid from the expected path. As the course progresses, we will be adding perception and creating more advanced controllers.

Useful tip: As you test your code, you will probably want to place the robot back to its starting position without restarting the simulation. You can do that by typing this:

```
ros2 service call /reset_world std_srvs/srv/Empty
```

Node Parameters

Most likely, your solution to Assignment 4 ended up with several hard-coded constants (e.g. velocity at which the robot moves, travel time of the robot, etc.). A well written node will parametrize these values allowing the user to specify them at run time. This is especially useful when some of these parameters need to be determined experimentally. Instead of changing the code before each experiment, you can just start the node with a different parameter value.

Assignment 4: Modify the code from Assignment 3 to parametrize the forward speed commanded to the robot. Submit the code.

Hint: Use command line arguments to pass the parameter value to the Launch file, followed by assigning the argument value to the ROS node parameter, followed by reading the parameter in your code as explained in this tutorial:

https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Using-Parameters-In-A-Class-Python.html

You need to declare the parameter in your code and then use `get_parameter_value()` method to read it.

Examining topics

The speed parameter that you created in the previous assignment sets the velocity that your program commands to the robot. That does not mean that the robot will actually move at that velocity. It has its physical limits and finite acceleration. Odometry topic provides robot's internal estimate of its linear and angular velocity. You can examine it like this:

```
ros2 topic echo /odom
```

or if you are interested in a specific field you can add `--field <path_to_field>` argument to isolate the field of interest. For example, if you are interested in linear velocity along x-axis it would be:

```
ros2 topic echo /odom --field twist.twist.linear.x
```

See the `Twist` message (message type used on odometry topic) definition for more information about individual fields.

Assignment 5: Run your program multiple times and vary the speed parameter, while examining the odometry topic, until you determine the maximum velocity that the robot can achieve. Report the result. Submit the output of the `echo` command.