ENEE 467: Robotics Project Laboratory
Introduction to Programming with ROS 2
Lab 3

Learning Objectives

The purpose of this lab is to provide a hands-on introduction to the ROS 2 workflow. After completing this lab students should be able to:

- Create, configure, and build ROS 2 Python packages
- Communicate with a ROS 2 environment via topics and actions
- Create simple nodes for publishing and subscribing to topics
- Use the ROS 2 command line interface to interact with a ROS 2 environment.

ROS 2 provides client libraries for both C++ and Python. Throughout this course we will focus on the Python client, though many existing packages are written in C++. Therefore this lab will provide an introduction to dealing with packages written in both languages (or a combination of the two).

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup.

Robot Operating System

Robot Operating System (ROS) is a bit of a misnomer, as ROS is not an operating system. ROS is a set of software libraries and tools for building robot applications, including drivers, algorithms for planning and control, as well as a rich set of tools for developing and interacting with systems.

ROS 2 is the second generation of ROS, providing many quality of life improvements such as security and reliability improvements, and support for large-scale embedded systems. The current long-term support version of ROS 2 is Jazzy Jalisco released in May 2024. ROS 2 Jazzy has tier 1 support for Ubuntu 24.04 (Noble) and Windows 10. The laboratory computers have Ubuntu 24.04 and ROS 2 Jazzy preinstalled. For those interested in installing ROS 2 Jazzy on their own system, please see the instructions at: Installation.

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

List of Exercises

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

> **Question 0.** A question for you.

Boxes with a leaf symbol (like the one below) are extra theoretical or implementation details provided for your enrichment and insight. We recommend that you read them during the lab:

> Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text commands will be set in black teletype font, e.g., `a = [0, 0, 0]`, while Python type hints will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and links to an external online reference are highlighted in blue, eg. see: Wikipedia.

1. Lab Procedure

1.1 Prequesites

The software for this lab is packaged and distributed as a Docker container. Ensure that the `lab-3` Docker image is build on your system:

```
docker image ls
```

and check to see that the following line is included in the output:

```
lab-3-image   latest    0d1b7b7788f3   2 days ago       4.47GB
```

Next, download the code for this lab from the `lab-3` repository:

```
cd ~/Labs
git clone git@github.com:ENEE467-F2025/lab-3.git
cd lab-3
```

Finally, launch the Docker container with the following commands:

```
cd docker
docker compose -f lab-3-compose.yml run --rm lab-3-docker
```

You should be greeted with the following prompt indicating you have a terminal inside the active container:

```
robot@desktop:~$
```

where `robot` is the name of the logged user and `desktop` is the name of the virtual machine (which may be different in your case).

1.2 Terminal Management with `tmux`

`tmux` is a terminal multiplexer that allows you to switch easily between several programs in one terminal. Since we only have one terminal connected to the Docker container, and we will need to run multiple commands concurrently, we need a way to divide the terminal into multiple sessions. Fortunately, tmux provides this capability for us. Launch `tmux` with the following command:

```
tmux
```

All commands in tmux are triggered by a prefix key followed by a command key. To split the terminal into a left and right pane, use the shortcut `Ctrl-A B`. That means press `Ctrl` and `A` at the same time then release them, then press `B`. Your single terminal window should now be split into two panes.

To split a pane into a top and bottom pane, use the shortcut `Ctrl-A V`.

To navigate between panes, press the command keys `Ctrl-A` then press an arrow key pointing in the direction of the pane you wish to navigate to. The shortcut is `Ctrl-A <arrow key>`.

Closing a pane is the same as exiting a terminal session. Navigate to the pane you wish to close and hit `Ctrl-D` or type `exit`.

1.3 Terminal Management with `screen`

Another popular choice for running processes concurrently in a single terminal window is `screen`. To attach a process using screen, run the following command:

```
screen -S <session_name> <process>
# example
screen -S a_python_shell python3
```

To detach from a running screen, type the command `Ctrl-A D`. That means press `Ctrl` and `A` at the same time then release them, then press `D`. To list all of the currently running screens, type the command:

```
screen -ls
```

which will return a list of processes and their IDs. To reconnect to a screen, type the following command:

```
screen -x <pid or name>
```

`screen` and `tmux` have their advantages and disadvantages when it comes to terminal multiplexing. Be aware that using `screen` runs the process in the background if you are not currently attached to the shell whereas `tmux` will display all active terminals.

1.4 Check your ROS 2 Installation

First and foremost your environment must be setup to interact with the installed ROS 2 packages and binaries. This is achieved by sourcing the ROS 2 setup file located in `/opt/ros/jazzy`. To configure your system to work with ROS 2, run the following command in a new terminal:

```
source /opt/ros/jazzy/setup.bash
```

For your convenience, this command has been added to the shell startup script so every time a new window is opened, ROS 2 is automatically sourced. You can verify that the environment is properly configured with the following command:

```
printenv | grep -i ROS
```

Verify that the environmental variables like `ROS_DISTRO` and `ROS_VERSION` are set.

Exercise 1. Random Numbers

We want to create two simple ROS nodes for generating random numbers. The first node will consist of a publisher that generates and publishes two uniform random numbers u_1 and u_2 every second on the topic `/random_numbers`. The second node will subscribe to the topic `/random_numbers` and use the uniform random numbers to generate a Normal random number then print it to the console. To generate a Normal random number from two uniform random numbers, we can use the Box-Muller Transform. Let u_1 and u_2 be uniform random numbers on the interval $(0,\ 1)$. Then

$$z = \sqrt{-2 \log u_1} \cos(2\pi u_2) \qquad (\dagger)$$

is a standard Normal random number. Here log represents the natural logarithm.

Inside the src folder is an ament_python package named random_numbers. Inside this package is the template for two nodes: simple_publisher and simple_subscriber. If you get stuck, refer to the ROS2 lecture slides on ELMS!

Deliverables: Complete the two node class files for simple_publisher and simple_subscriber by doing the following:

Simple Publisher:

a. Create a publisher inside the __init__ method that publishes a message type Float32MultiArray to the topic random_numbers. Set the qos_profile to 10. Be sure to assign the publisher to a class variable so you can access it in a callback.

b. Create a timer inside the __init__ method with a period of 1.0 seconds. Set the callback function to the class function self.timer_callback.

c. Complete the class function timer_callback by publishing msg to the ROS network using your class's publisher.

Simple Subscriber:

d. Create a subscriber inside the __init__ that subscribes to the topic /random_numbers and targets the callback function self.sub_callback.

e. Complete the class function sub_callback by implementing the Box-Muller transform (†) using the variable u_1 and u_2. Print the resulting Normal random number to the consol using ROS's logging function, self.get_logger().info("...").

Building the Package:

Once you complete the node class files, build the package by navigating to the workspace root ros2_ws and building the package with colcon:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select random_numbers
```

The package's setup.py has already been populated for you. Once building is complete, source the workspace:

```
source install/setup.bash
```

and run the nodes in two separate tmux panes:

```
# pane 1
ros2 run random_numbers simple_publisher
# pane 2
ros2 run random_numbers simple_subscriber
```

Ensure that simple_subscriber is producing Normal random numbers by looking at the console output. To verify that the network is properly configured, run rqt_graph in a third

pane and verify that `simple_publisher` is publishing to `simple_subscriber` over a topic named `random_numbers`.

Include a screenshot of `simple_subscriber`'s terminal output as well as a screenshot of `rqt_graph` in your lab report.

2. The Turtlesim Node

Turtlesim is a simple 2D mobile robot simulator packaged with ROS installations for tutorials. Check that the `turtlesim` package is installed with the following terminal command:

```
ros2 pkg executables turtlesim
```

This command outputs the available package specific executables. The expected return for the above command is:

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

If the `turtlesim` package is not installed, it can be installed via apt with the following command (sudo required):

```
sudo apt install ros-jazzy-turtlesim
```

Now that we ensured the `turtlesim` package is installed on the system, we can start the `turtlesim` node. First launch `tmux` and split the window into two panes. Next in one of the panes run the `turtlesim` node with the following command:

```
ros2 run turtlesim turtlesim_node
```

You should see the simulator window appear with a random turtle sprite in the center. The ROS 2 cli provides tools for listing the available nodes, topics, services, and actions as well as their details. To see all public nodes that are currently running, navigate to the other pane and enter the following command:

```
ros2 node list
```

To see more info about a node including published and subscribed topics, run the following command:

```
ros2 node info <node_name>
```

For example, substituting `/turtlesim` for `<node_name>` should reveal that the turtle sim has the following subscriptions and publications:

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
```

ROS 2 provides a `list` command for topics, services, and actions in addition to nodes. The respecitve commands are:

```
ros2 topic list
ros2 service list
ros2 action list
```

To show the message type for a topic, service, or action, include the type flag (`-t`) at the end of the command. Running `ros2 topic list -t` should reveal the following topics and types:

```
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

To learn more about a topic, service, or action, try running the following `info` command for a respective interface:

```
ros2 <interface_type> info <interface_name>
```

Question 1. What is the average publishing rate for the topic `/turtle1/pose`? (**Hint:** try running the command `ros2 topic --help` to see the available commands for topics).

To see the quality of service settings for a topic, include the verbose flag (`-v` or `--verbose`). For example, the topic `/turtle1/cmd_vel` has the following QoS profile:

```
QoS profile:
  Reliability: RELIABLE
  History (Depth): KEEP_LAST (7)
  Durability: VOLATILE
  Lifespan: Infinite
  Deadline: Infinite
  Liveliness: AUTOMATIC
  Liveliness lease duration: Infinite
```

Properly configured QoS settings are important when trying to establish communication channels as certain configurations are not compatable (see: Quality of Service settings). When unsure about the QoS settings for a particular topic, the previous ROS 2 cli command provides a quick and easy way to check.

To kill a process running in a terminal, send a interrupt signal (`SIGINT`) by entering `Ctrl + C` in the terminal.

3. Creating a Package to Control the Turtle

Now we want to create a simple control node to drive the turtle around to goal locations. To start we need a ROS 2 workspace to house our package in which the node will reside. a typical ROS 2 workspace has the following structure:

```
ros2_ws/
|-- src/
|   |-- pkg_1
|   `-- pkg_2
|-- build
|-- install
`-- log
```

where `ros2_ws` is the root of the workspace, `src` contains the source code for packages, and `build`, `install`, and `log` are automatically generated during the build procedure.

The downloaded repository contains a `src` directory that is mounted in the Docker container with two packages in addition to the `random_numbers` package we just finished. Ignore these packages for now, we will need them later. Navigate to the source directory:

```
cd ~/ros2_ws/src
```

ROS 2 provides two different build types: `ament_cmake` (default) for C++ applications and `ament_python` for pure Python applications.

> **Question 2.** What build type should you use if your application consists of both C++ and Python code?

We wish to create a Python node so run the following command to create a package named `turtle_control`:

```
ros2 pkg create --build-type ament_python turtle_control
```

Change directories to the newly created package and list its contents:

```
cd turtle_control && ls
```

You should see the following output:

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

The first thing we want to do is configure the file `package.xml` to add any dependencies that we think our node might need. Our control node requires the turtlesim package so lets add it as a dependency. Open the file `package.xml` and add the following element within the `<package>` tags:

```
<depend>turtlesim</depend>
```

Although the `package.xml` doesn't affect your package at build-time, it is used by dependency managers like `rosdep` when installing package dependencies. It is a crucial habit that every time you add a new (non-standard) dependency to your application that you include it in your `package.xml`. This increases the reusability and portability of your application.

The code for you node will reside inside the folder with the same name as the package. Change directories to the folder `turtle_control` and create a new Python file named `turtle_control.py`:

```
cd turtle_control
touch turtle_control.py
```

Open the file `turtle_control.py`. We are now ready to start writing the code for the control node!

3.1 Writting the Python Node

The easiest way to write and manage nodes in Python is to extend the Python client's node class. Below is the template for a basic Python node.

```
# import the Python ROS 2 client
import rclpy
# import the Node class
from rclpy.node import Node

# create a class that inherits Node
class TurtleControl(Node):

    def __init__(self):
        # call the Node class constructor and give your node a name
        super().__init__("turtle_control_node")


# main entry point
def main(args=None):
    # initialize the ROS 2 client
    rclpy.init(args=args)
    # create an instance of your node
    turtle_control_node = TurtleControl()

    # graceful shutdown
    try:
        # spin, i.e. run the node
        rclpy.spin(turtle_control_node)

    except (KeyboardInterrupt, rclpy.executors.ExternalShutdownException):
        print("\nExiting...")
        turtle_control_node.destroy_node()
```

Now with our node created, we have to add an entry point so colcon knows where to find the code. Open the file `setup.py` and navigate to the field `entry_points`. This is where we add the path to our main function. Add the following line within the `console_scripts` brackets of the `entry_points` field.

```
entry_points={
    'console_scripts': [
        'turtle_control_node = turtle_control.turtle_control:main',
    ],
},
```

We are now ready to build the package and run our node! Navigate back to the root directory of

the workspace:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select turtle_control
```

Here we use the flag `--symlink-install` to install a symbolic link to our file `turtle_control.py` that way we can continue to edit it without rebuilding. We also use the flag `--packages-select` to tell the build system that we only want to build the package `turtle_control`. If not otherwise specified, colcon will build all of the packages located inside the `src` folder. Once it completes building, we have to source the workspace to tell ROS 2 where the new package files are located.

```
source install/setup.bash
```

Now our package is sourced and we can list its executables:

```
ros2 pkg executables turtle_control
```

Which should output the following:

```
turtle_control turtle_control_node
```

We can run our new node with the following command:

```
ros2 run turtle_control turtle_control_node
```

List the currently running nodes in a new terminal pane and make sure you can see your node running.

3.2 Adding a Subscriber to the Node

In order to create a feedback controller for our turtle, we need a way to obtain the current position and orientation of the turtle. Luckily the simulator node publishes the current pose (pose = position + orientation) so all we need to do is add a subscriber to our node and we will have access to the pose data. The turtlesim node publishes pose data using the custom message type `turtlesim/msg/Pose`. To see the available fields in this message, enter the following command in a terminal window:

```
ros2 interface show turtlesim/msg/Pose
```

Which returns the following output:

```
float32 x
float32 y
float32 theta

float32 linear_velocity
float32 angular_velocity
```

Our feedback controller needs information about the current position and orientation. The pose message also contains information on the current linear and angular velocity, but we can ignore these for now. To store the current pose of the turtle in our node's memory, modify the file `turtle_control.py` with the following changes:

```python
# import pose message
from turtlesim.msg import Pose

class TurtleControl(Node):

    def __init__(self):
        # call the Node class constructor and give your node a name
        super().__init__("turtle_control_node")

        # class variables for current pose
        self.x: float = None
        self.y: float = None
        self.theta: float = None

        # create a subscriber
        self.create_subscription(
            msg_type=Pose,
            topic="/turtle1/pose",
            callback=self.pose_callback,
            qos_profile=10
        )

    # create a subscriber callback to process pose data
    def pose_callback(self, msg: Pose):
        # store the message fields in memory
        self.x = msg.x
        self.y = msg.y
        self.theta = msg.theta
```

Since we built our node with the flag `--symlink-install` we do not need to rebuild in order for the changes to take place. Rerun the `turtle_control_node` and take a look at its subscriptions with `ros2 node info`. You should now see the following:

```
/turtle_control_node
  Subscribers:
    /turtle1/pose: turtlesim/msg/Pose
```

Meaning our node is subscribed to the turtle's pose!

3.3 Deriving a Feedback Controller

The turtle is an example of a non-holonomic system, which roughly speaking means there are constraints on its velocity that prevents motion in certain directions. The kinematics of the turtle are described by the Dubins path or Unicycle model. The position and orientation of the turtle is a three-vector $(x,\ y,\ \theta)$ where $x,\ y$ describes the center of the unicycle in the world frame system and θ describes the orientation of the turtle relative to the x-axis. The control input to the turtle is a *twist*, which describes the body linear and angular velocity. More precisely, the control input for the turtle is a linear velocity the direction of its heading, v_x, and an angular velocity about its axis of rotation, ω_z.

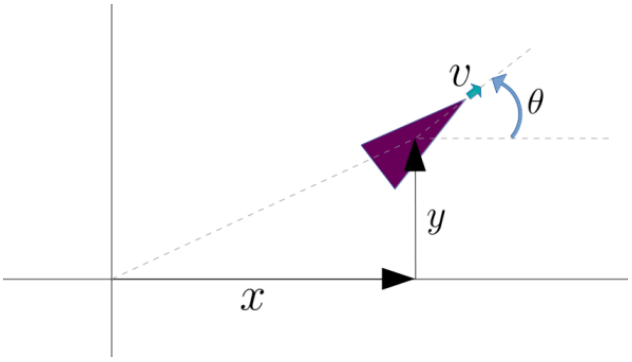

Figure 1: The Unicycle model.

Question 3. What are the fields in the message type `geometry_msgs/msg/Twist`?

The evolution of the state of the turtle is described by the following set of differential equations:

$$\begin{aligned}\dot{x} &= v_x \cos\theta \\ \dot{y} &= v_x \sin\theta \\ \dot{\theta} &= \omega_z.\end{aligned}$$

Our goal is to construct and implement a feedback controller that drives the turtle to a desired goal location. To do this we will construct a proportional controller of the form:

$$\begin{aligned}v_x &= K_1 d_e \\ \omega_z &= K_2 \theta_e,\end{aligned} \tag{1}$$

where d_e is the error in the distance between the desired position and the current position of the turtle, θ_e is the error in the relative orientation of the turtle and the goal position and the current orientation of the turtle, and K_1, K_2 are positive scalar gains.

Given a desired goal position $(x_d,\ y_d)$, the error in the distance is simply the length of the vector pointing from the current position of the turtle to the goal position:

$$d_e = \sqrt{(x_d - x)^2 + (y_d - y)^2}. \tag{2}$$

The orientation error is a little more tricky. What we really want is the angle between the current heading of the turtle and the vector pointing from the current position to the goal. In addition, we want the sign of the angle to indicate what direction the turtle must turn to reach the goal with as little effort as possible. We use the right-handed convention with the normal to the plane pointing out of the screen. A positive angle implies a counterclockwise rotation about the normal, and a negative angle implies a clockwise rotation. The (signed) angle is given by:

$$\theta_e = \text{atan2}\left((y_d - y)\cos\theta - (x_d - x)\sin\theta,\ (x_d - x)\cos\theta + (y_d - y)\sin\theta\right), \tag{3}$$

where atan2 is the 2-argument arctangent (see: Atan2).

3.4 Implementing the Controller

So far our turtle control node consists of a subscriber to store current pose information. We will need two more components to implement the controller: 1) a publisher to publish the computed control input, and 2) a timer that will run at a constant frequency and determine the rate in which control inputs are computed. ROS 2 is an asynchronous, concurrent programming language and in general while loops are not allowed inside subscriber callbacks (as they need to be non-blocking). Instead, if we want a particular function to run continuously at a fixed rate, we implement it as a timer callback.

To create a publisher, we must first import the message type we intend to publish. The topic `/turtle1/cmd_vel` has the message type `geometry_msgs/msg/Twist`. Add the following code to the top of `turtle_control.py`:

```
# import twist message
from geometry_msgs.msg import Twist
# import numpy for some useful math functions
import numpy as np
```

To create a publisher, add the following block to the turtle control node's `__init__` method:

```
self.cmd_vel_publisher = self.create_publisher(
    msg_type=Twist,
    topic="/turtle1/cmd_vel",
    qos_profile=10
)
# create a class variable for the desired position
self.goal: tuple = None # tuple of the form (x_d: float, y_d: float)
```

Next, we will create a method to compute the control input. Create a class method called `compute_control` to the turtle control node based on the following template:

```
def compute_control(self):
    # wait to compute control until we recieve pose data
    if self.x is None or self.goal is None:
        return

    # compute v_x and omega_z here
    # hint: use np.arctan2 for an atan2 implementation
    # v_x = ...
    # omega_z = ...

    msg = Twist()
    msg.linear.x = v_x
    msg.angular.z = omega_z

    self.cmd_vel_publisher.publish(msg)
```

You will need to write the code to implement equations (1), (2), and (3) within this method. Recall that the current pose is stored in the three state variables `self.x`, `self.y`, and `self.theta`, and the desired goal position is stored as a 2-tuple `self.goal`. For the gains K_1, K_2, start with something small ($0.1 \leq K_1 \leq 0.5$ and $0.5 \leq K_2 \leq 1$).

Now we need to create a timer that will execute the function `compute_control` at a fixed frequency. Add the following code block to the turtle control node's `__init__` method:

```
# create a timer to execute the controller
self.create_timer(
    timer_period_sec=0.03, # roughly 30 Hz
    callback=self.compute_control
)
```

Now we need a way to pass goal positions to the turtle. We can do this by implementing another subscriber over a topic name `/turtle1/goal`. The goal message needs to contain two floats, one for the desired x position and one for the desired y position. A good message choice for this is `std_msgs/msg/Float32MultiArray` which has the field:

```
float32[]         data          # array of data
```

Add the following import statement to `turtle_control.py` to import the float array message:

```
# import float array message
from std_msgs.msg import Float32MultiArray
```

Next create a class function to process the goal message:

```
def process_goal(self, msg: Float32MultiArray):
    # store the goal in memory and prevent indexing error
    if len(msg.data) >= 2:
        self.goal = (msg.data[0], msg.data[1])
```

Finally, add the following block to the turtle control node's `__init__` method to create the new subscriber:

```
self.create_subscription(
    msg_type=Float32MultiArray,
    topic="/turtle1/goal",
    callback=self.process_goal,
    qos_profile=10
)
```

Now our turtle is ready to receive goal commands and navigate to them!

3.5 Sending a Goal Message from the Command Line

Ensure that both the turtlesim node and the control node are running. Next enter the following command in a new terminal to publish one message to the topic `/turtle1/goal`:

```
ros2 topic pub -1 /turtle1/goal std_msgs/msg/Float32MultiArray \
    "data: [2.0, 7.0]"
```

You should now see the turtle navigate to the goal position!

Question 4. Try playing around with the scalar gains K_1, K_2. What happens when you increase/decrease them? You will have to rerun the control node for the changes to take place.

Figure 2: Turtle navigating to a goal with a left-handed turn.

Exercise 2. Goal Publisher Node

We can send goals to the control node using ROS's topic cli for publishing. These commands are very verbose, and for more complex data types, fillining in fields in one line can be tedious. Instead, we would like to make a node that reads the minimal information required from the terminal and programmatically constructs a message to publish. In order to do this, we will need to make a new node.

Deliverables: Create a new node that takes goal inputs from the terminal and publishes them to the topic `/turtle1/goal`. To do this, complete the following:

a. Create a new node class file inside the `turtle_control` package source directory named `goal_publisher.py`. Create a class inside this file that extends the Node base class.

b. Inside the class, create a publisher that publishes to the topic `/turtle1/goal` with message type `Float32MultiArray`.

c. Inside the class, create a class method named `send_goal` with two arguments for the x and y goal positions. Inside the method, create a `Float32MultiArray` message and populate the data field with the goal position (x, y). Finally, publish the message using your class's publisher.

d. Add an entry point to your file in the form of a `main` function. Inside this function we will need to read user input from the terminal and call our class function `send_goal`. After initializing `rclpy` and instantiating your node, create an infinite while loop. Inside the loop, read user input with the Python function `input`. User input should be of the

form two floats separated with a comma. Once you process the input and call your class's `send_goal` function, you will need to let `rclpy` "spin" your node. Typically, this is done asynchronously using the `rclpy.spin` method. However, since we want to continuously read user input, we need a way to spin synchronously. Thankfully, ROS provides this functionality with the function `rclpy.spin_once` which flushes all callbacks currently registered to the executor's stack, but not any new ones registered after calling the function. Be sure to specify `timeout_sec` to be zero or greater. If left unspecified, `spin_once` will block forever. An example entry point is provided for you below:

```
def main(args=None):
    rclpy.init(args=args)
    my_node = GoalPublisher()
    while 1:
        data = input("Goal Position x,y: ")
        x, y = data.split(",")
        try:
            x = float(x)
            y = float(y)
        except ValueError:
            print("Wrong data format!")
            continue

        my_node.send_goal(x, y)
        rclpy.spin_once(my_node, timeout_sec=0)
```

Add your new node's entry point to the package's `setup.py`. Rebuild and source the package then demonstrate your goal publisher working by sending simplified goal commands from the terminal.

4. Implementing the Controller as an Action Server

Whenever making a request for something in ROS 2, whether it be a request for motion, a computation, or data, it is best done using a request-response method like a service or an action (see: Topics vs Services vs Actions). We now want to create an action server for the turtle controller that revives motion requests and executes them. To begin, navigate to the source folder within your ROS 2 workspace:

```
cd ~/ros2_ws/src
```

Inside the folder are templates for two packages that we will finish developing: 1) `turtle_control_interfaces` which houses the custom interfaces we will use to interact with our action server, and 2) `turtle_controller_server` which contains the source code for the action server node.

4.1 Creating an Action Interface

Much like messages and services, action interfaces use the interface definition language (IDL) to describe the action's goal, result, and feedback messages. An action definition file (`.action`) has the following structure:

```
# Request
<request_type> <request_fieldname>
---
# Result
<response_type> <response_fieldname>
---
# Feedback
<feedback_type> <feedback_fieldname>
```

Actions can also consist of fields from message types. In the folder `turtle_control_interfaces/msg` is a file `Goal.msg` that defines a custom message for a turtle goal. The message has two fields: `x` and `y`, both with type `float32` representing the coordinates of the goal position.

Navigate to the folder `turtle_control_interfaces/action` and open the file `SendGoal.action`. Notice that the request fields consist of a `Goal` message, a `float32 goal_tol` representing the minimum distance to the goal position the turtle must reach to have successfully navigated, and a `float32 time_out` representing the maximum time allotted for navigation. ROS 2 interface definitions allow for fields with default values. Defining a default value is done by adding a third element to the field definition line, i.e:

```
<field_type> <fieldname> <field_default_value>
```

The result field consists of a `bool success` indicating whether or not navigation was completed successfully. The feedback field is empty as we will not be using it.

4.2 Writing the Action Server Node

Open the file `action_server.py` within the `turtle_controller_server` package. This file houses the source code that will be used to build the controller action server. Most methods have

been populated for your convenience. Inspect the file. Notice that some values like the controller gains and the controller rate are implemented as ROS parameters. This allows you to configure your node at runtime without the need to edit the source code. You can set parameters directly from the command line using the following syntax:

```
ros2 run package_name executable_name --ros-args -p param_name:=param_value
```

The class `TurtleControllerServer` has the following structure:

```
class TurtleControllerServer(Node):

    def __init__(self):
        # declare parameters and initialize the node

    def pose_callback(self, msg: Pose):
        # process turtle pose data and store in memory

    def goal_callback(self, goal_request: SendGoal.Goal) -> GoalResponse:
        # process goal requests and either accept or reject them

    def execute_callback(self, goal_handle: ServerGoalHandle):
        # main method, execute a request for navigation
```

Question 5. Why do we need to place the action server callbacks and publisher/subscriber callbacks in separate callback groups? What would happen if they were all registered to the same mutually-exclusive callback group? (**Hint:** see: Using Callback Groups).

Similarly to Exercise 2. where we implemented the controller in a topical fashion, your task is to implement the controller code in the action server's main callback, `execute_callback`. The callback contains a while loop where your code must be implemented. The loop runs at a constant frequency determined by the parameter `controller_rate` and must compute the next control input at each time step.

```
 # compare elapsed time to timeout
while T1 - self.get_clock().now() <= timeout:
    x_e = x_d - self.x
    y_e = y_d - self.y
    c = np.cos(self.theta)
    s = np.sin(self.theta)

    # compute errors
    theta_e = # YOUR CODE HERE
    d_e =     # YOUR CODE HERE

    v = Twist()
    v.linear.x = self.K_1*d_e
    v.angular.z = self.K_2*theta_e

    self.cmd_vel_publisher.publish(v)
```

```
    # check to see if we have reached the goal
    if d_e <= goal_tol:
        break

    # this is a blocking call running inside a callback
    # must use a reentrant callback group to safely process
    # this without deadlock
    self.rate.sleep()
```

Add your code for computing the control inputs then change directories back to the root of the workspace and build. Be sure to use the `--symlink-install` flag so you can continue to make changes without rebuilding. Once building has finished, source the workspace, start the turtlesim node if not already running, and start the new node with the following command:

```
ros2 run turtle_controller_server turtle_controller_server_node
```

Verify that the action server is advertising itself by listing the available actions in a new terminal:

```
ros2 action list
```

The new action `/turtle_goal` should be included in the output.

4.3 Sending an Action Goal from the Command Line

The ROS 2 action cli includes a command for sending goals directly from the command line. The syntax for sending a goal is:

```
ros2 action send_goal <action_name> <action_type> <values>
```

Where `<values>` typically takes the form of a YAML dictionary (see: Understanding actions).

To send a goal to the turtle,

```
ros2 action send_goal /turtle_goal turtle_control_interfaces/action/SendGoal \
    "{goal: {x: 1.0, y: 9.0}, goal_tol: .1}"
```

Exercise 3. Simple Action Client Node

Similarly to publishing to topics from the terminal, sending action goals using the cli can be very verbose. It is preferable to have a node send action goals while interacting with it in a more user-friendly fashion. To do so, we need a node that takes in user input from the terminal and programmatically constructs an action goal. Nodes responsible for sending action goals are referred to as *action clients* whereas nodes that process action goals are referred to as *action servers*. The action client code is provided for you. Inside the `turtle_controller_server` package, open the file `action_client.py`. To run the action client, ensure the `turtlesim` and `turtle_controller_server` nodes are running. Then in a separate pane, run the following command:

```
ros2 run turtle_controller_server turtle_controller_client_node
```

Deliverables: To complete the exercise, answer the following questions:

a. Inside the file `action_client.py`, the node class has three functions for processing communication with the action server. These functions are:

   1. `send_goal`
   2. `response_callback`
   3. `result_callback`

   Qualitatively explain why there are three methods for processing communications with the server. How does this relate to the notion of an action? (**Hint:** Actions consist of three parts: a goal, feedback, and a result).

b. Try sending some goals using the client. Notice that the client node blocks until navigation is complete. What happens if you send an invalid goal? (Try sending a goal far out of bounds).

c. In the entry point function `main`, callbacks are processes synchronously using the function `rclpy.spin_once` inside of a while loop. Compared to the goal publisher node, we need to run `spin_once` multiple times to ensure all action callbacks are processed. To verify that all callbacks have been flushed, we use the custom class function `is_done()`. How many times is `spin_once` called when an action goal is accepted? How many times when an action goal is rejected?

d. ROS uses futures when referring to tasks that are scheduled but have not completed yet. Subsequently, each future is registered to a callback that will execute when the future completes and process any resulting data. How does the use of futures and their associated callbacks support the asynchronous nature of ROS actions, and why is this design important for action clients like the one implemented in `action_client.py`?

5. Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. If you wrote any Python programs during the exercises, include these files as a separate attachment, appending your team's number and last names (in alphabetical order) to the file, e.g., **Jane Doe**, **Alice Smith**, and **Richard Roe** in **Team 90** submitting their modified `main.py` would submit the file `main_Team90_Doe_Roe_Smith.py` and the report `Lab_<LabNumber>_Report_Team90_Doe_Roe_Smith.pdf`, where `<LabNumber>` is the number for that week's lab, e.g., **0**, **1**, **2**, etc. Submit your lab report along with any code to ELMS under the assignment for that week's lab. You must complete and submit your lab report by **Friday, 11:59 PM** of the week following the lab session. See the **Files** section on ELMS for a lab report template named **Lab_Report_Template.pdf**. Below is a rubric breaking down the grading for this week's lab assignment:

Component	Points
Lab Report and Formatting	10
Question 1.	8
Question 2.	8
Question 3.	8
Question 4.	8
Question 5.	8
Exercise 1.	20
Exercise 2.	20
Exercise 3.	10
Total	100