ROBOTICS LAB 6
Custom Teleoperation, Launch Files & Plotting

MUNADI SIAL

SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

Overview

This lab will involve the following:

- Keystroke Detection
- Simultaneous Publishing and Subscribing
- Custom Teleoperation Program
- Create a Launch File for multiple node execution
- RQt Plotting

Teleoperation

- Teleoperation is the movement of a robot from one place to another when the user provides an input (typically from a keyboard)

- The goal of robotics is to make fully autonomous systems that can move without user input. However, manual teleoperation is much simpler than autonomous behavior and so is a good starting point for building robots

- Teleoperation is also useful to control the autonomous robot when it malfunctions during tests

Teleoperation

- In the previous lab, we used the following command to use teleoperation in the robot simulation

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

- This lab will focus on building our own teleoperation program from scratch so as to deepen understanding of ROS

- Although there is already a teleoperation program in ROS, creating custom teleoperation from scratch will give valuable insights into ROS programming for practical scenarios

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

This is the class code for a node that will publish twist messages. The node will cause the robot to toggle between moving and stopping after every 1.5 seconds.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The **create_publisher** function from *Node* base class is used to make the node a publisher

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The node will publish messages of type **Twist**. These messages will be sent to the **cmd_vel** topic. The cmd_vel is a common topic name for robot velocities.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The **create_timer** function in the *Node* base class is used to define a timer. The timer is not mandatory and is only used to publish messages at regular intervals of time.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The node will publish the twist message after every 1.5 seconds.

Twist Messages - Review

```python
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The timer callback is the function that is executed after every 1.5 seconds. The messages will be published from the timer callback

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The message is a Twist object; msg is a vector of six variables:

linear.x angular.x,
linear.y angular.y
linear.z angular.z

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The isMoving is a boolean flag that keeps account of the robot’s motion. It is True when the robot is moving and False if it is stopped.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The appropriate variables (linear.x) in the msg object are set before publishing.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The message is published to the **cmd_vel** topic. The robot's motion will be affected as soon as the message is published.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

The isMoving flag is changed for the next callback. In this case, the flag is inverted.

Twist Messages - Review

```
class VelocityPub(Node):
    def __init__(self):
        super().__init__('twist_publisher')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        timer_period = 1.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.isMoving = False # flag to check if robot is moving

    def timer_callback(self):
        msg = Twist()
        if self.isMoving == False:
                msg.linear.x = 1.0
                self.get_logger().info('GO !!!')
        if self.isMoving == True:
                msg.linear.x = 0.0
                self.get_logger().info('Stop')
        self.publisher_.publish(msg)
        self.isMoving = not(self.isMoving)
```

This is optional. The **get_logger** function in the *Node* base class is used to log information on the terminal.

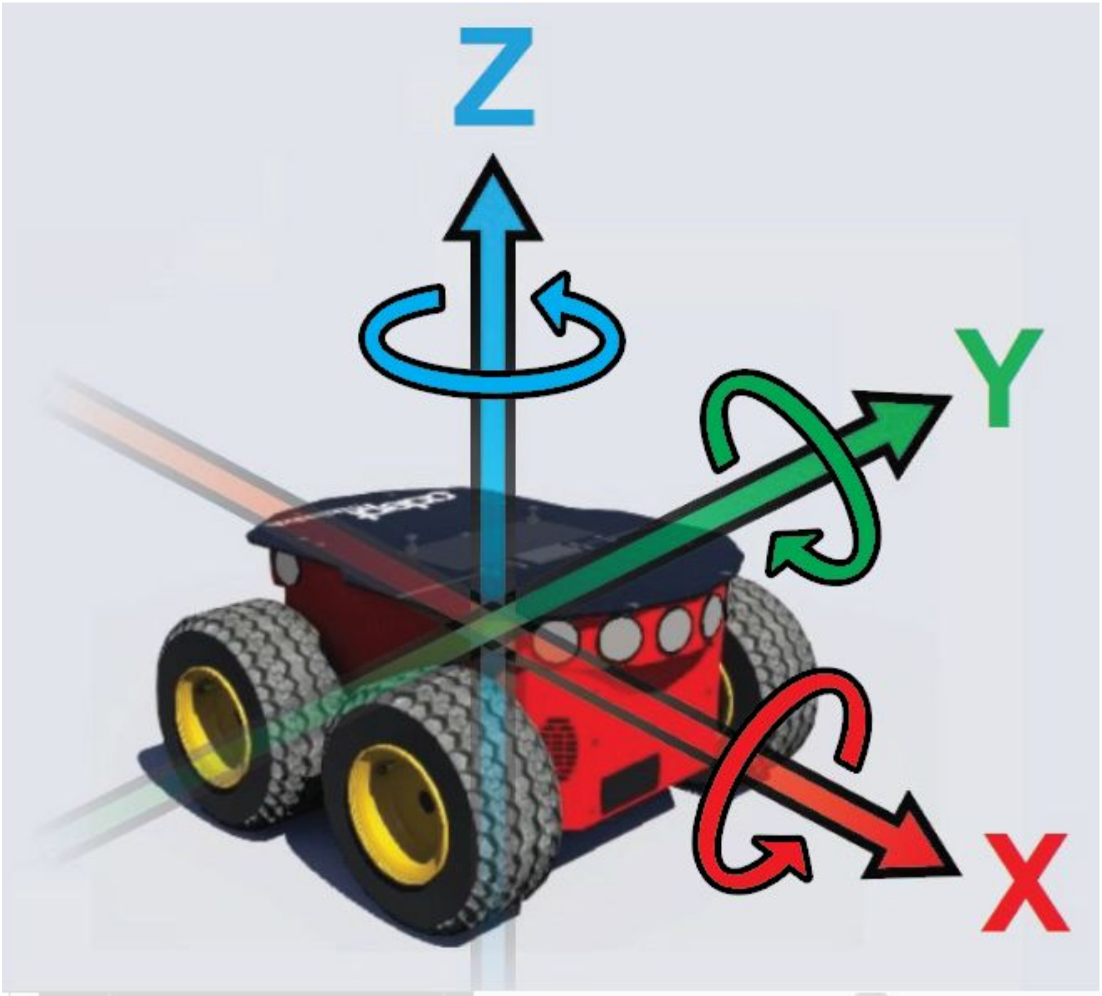
Twist Messages - Review

In mobile robots, the convention is to use:

- **X-axis** for forward and backward direction of robot
- **Y-axis** for sideways (or lateral) direction of robot
- **Z-axis** for vertical direction (aligned with gravity)

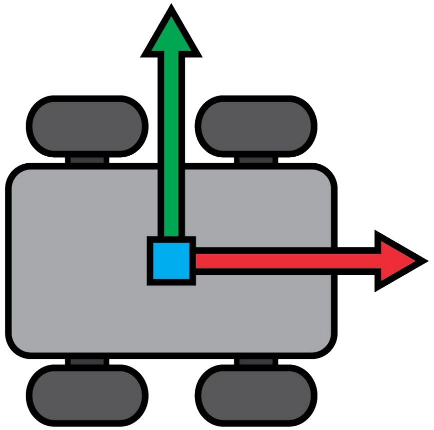
The twist messages are

linear.x	**angular.x**
linear.y	**angular.y**
linear.z	**angular.z**

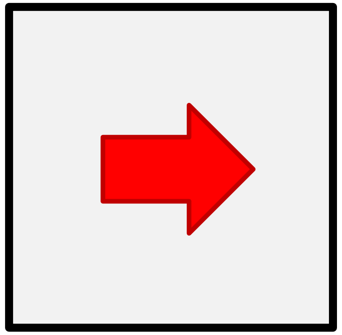


Twist Messages - Review

- The robot's **x-axis** and **y-axis** are shown
- The **z-axis** is pointing out of the plane
- The robot has only the **linear.x** and **angular.z** velocities due to the constraints from its design

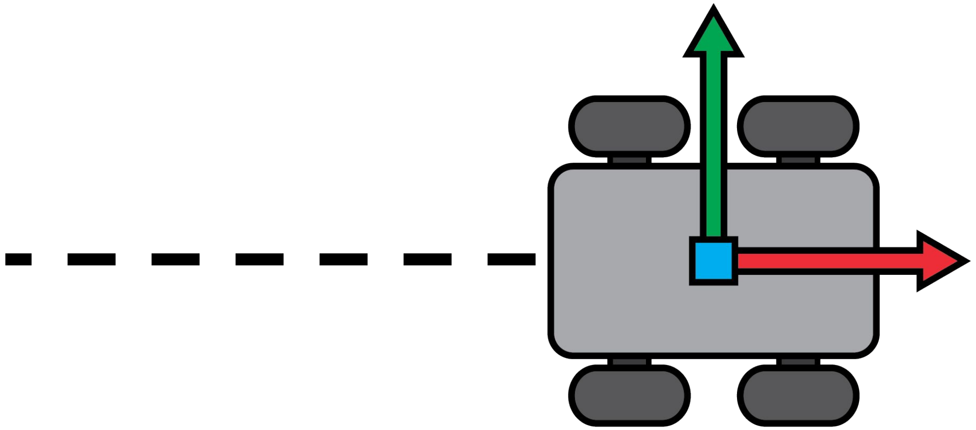


Twist Messages - Review

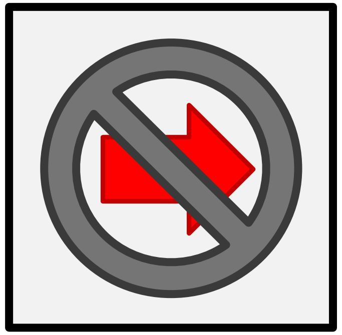


- To move forward, set the twist message:

```
msg.linear.x = 1.0
```

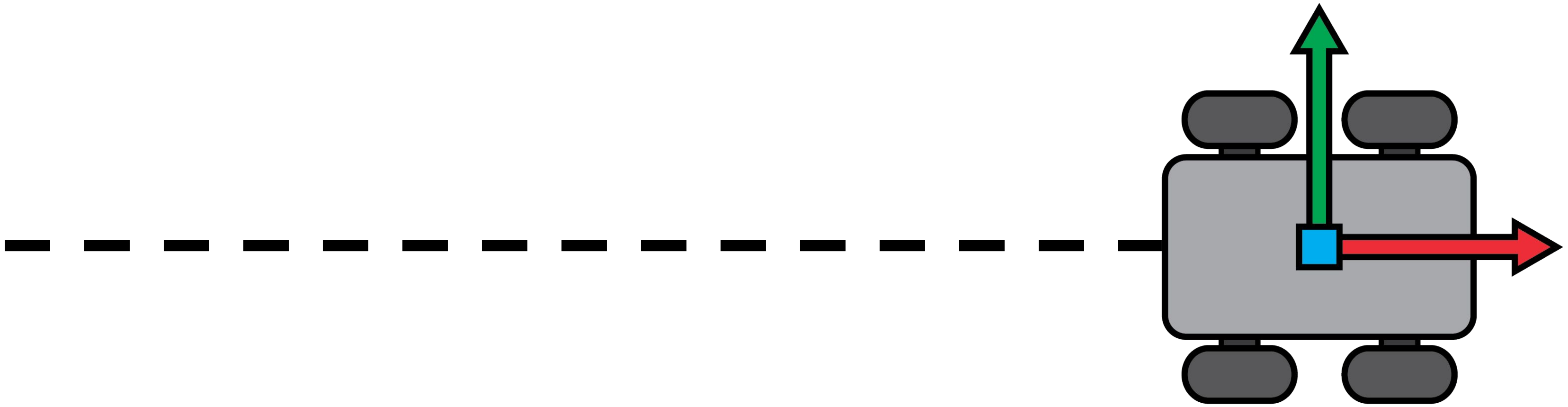


Twist Messages - Review



- To stop the robot, set the twist message:

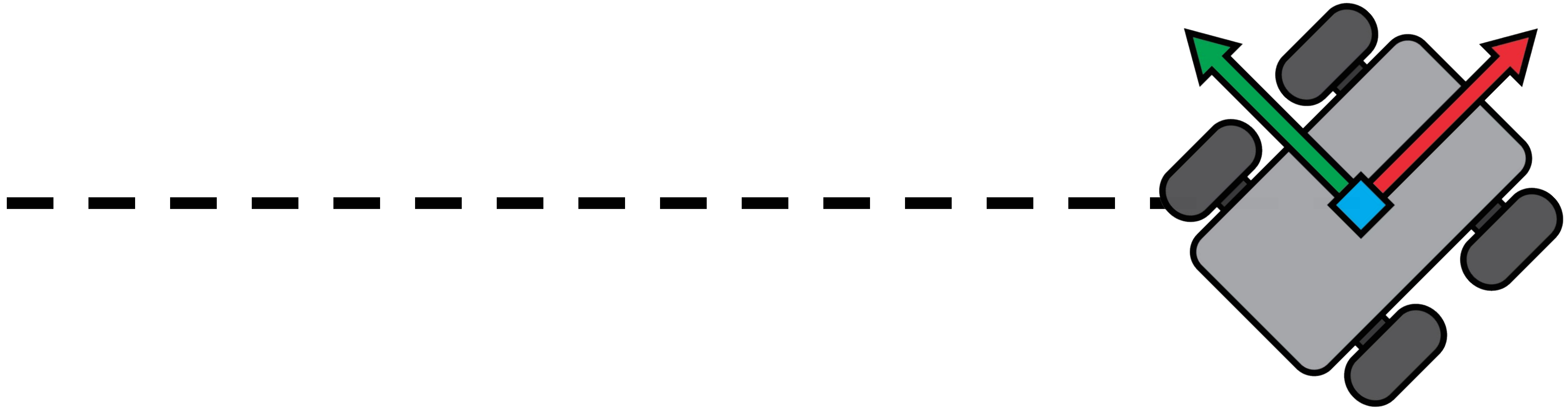
```
msg.linear.x = 0.0
```



Twist Messages - Review

- To start turning, set the twist message:

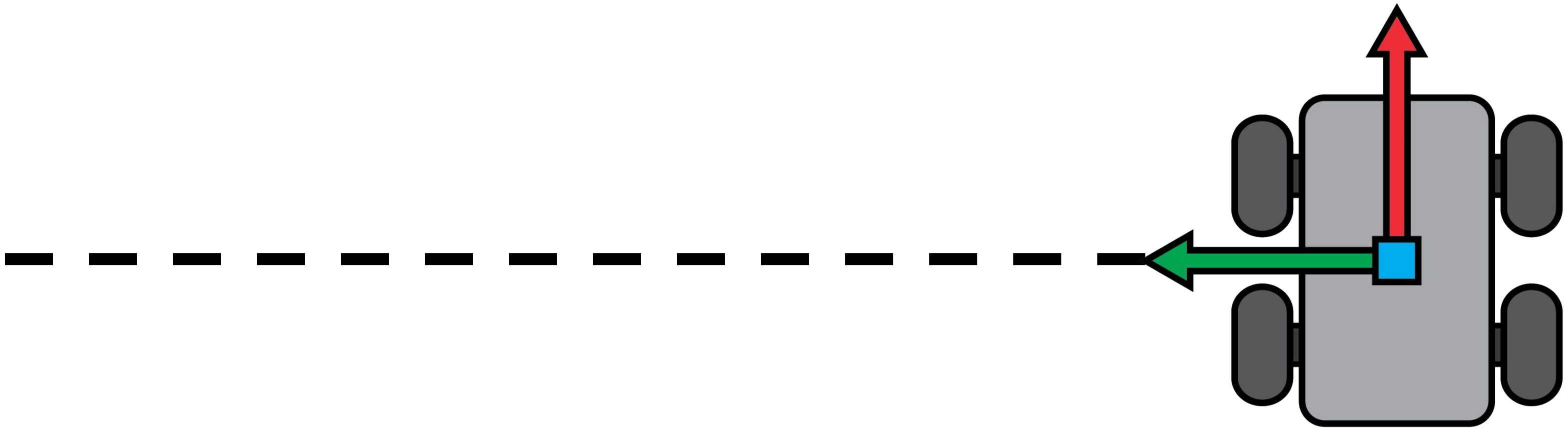
```
msg.angular.z = 1.0
```



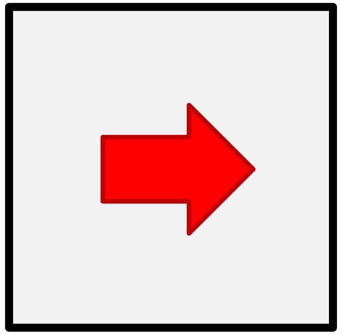
Twist Messages - Review

- To stop turning, set the twist message:

```
msg.angular.z = 0.0
```

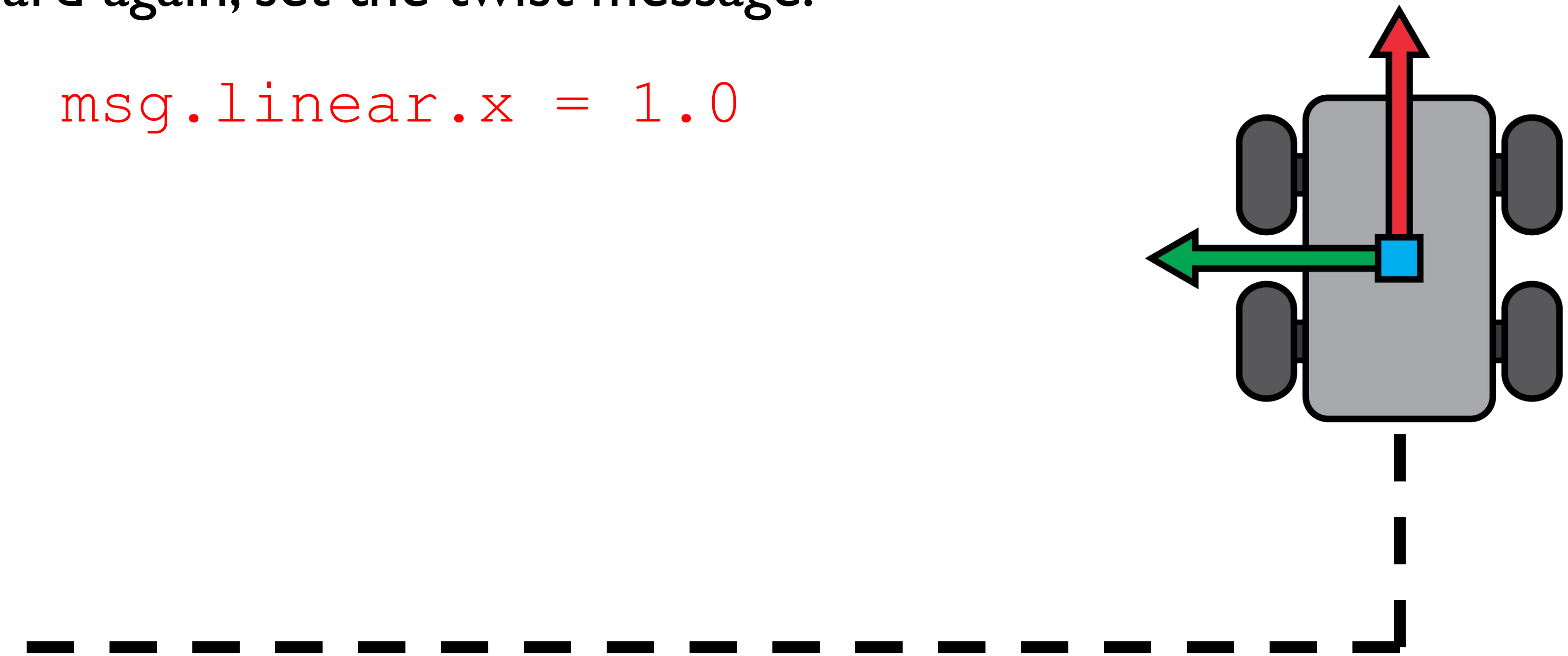


Twist Messages - Review



- Notice that the robot's **x-axis** has turned
- To move forward again, set the twist message:

```
msg.linear.x = 1.0
```



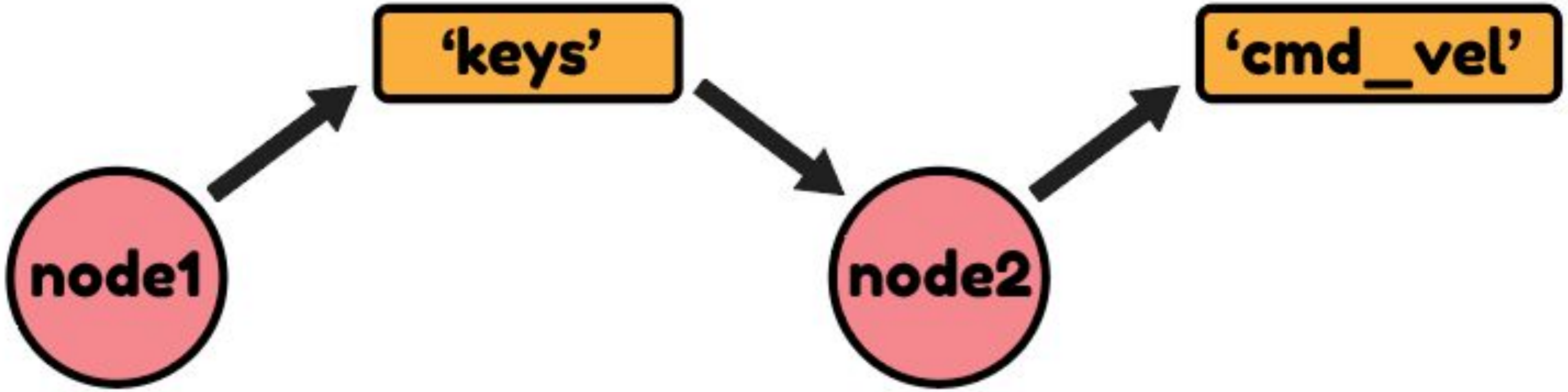
Custom Teleoperation

- We have looked at topics for communication of nodes
- A publisher node sends messages to a topic
- A subscriber node receives messages from a topic and contains a callback function which is called each time it receives a message

- We will use nodes communicating over topics to make our custom teleoperation program

Custom Teleoperation

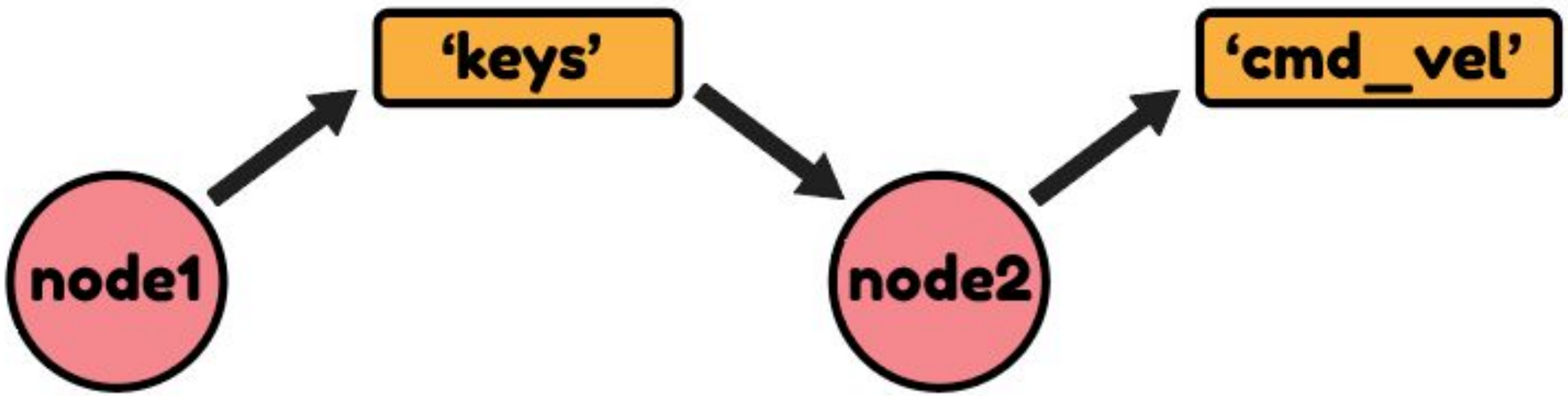
- The custom teleoperation program can be made by the node graph:



- There are 2 nodes and 2 topics
- The first node publishes to a topic
- The second node subscribes to one topic and publishes to another topic

Custom Teleoperation

- The custom teleoperation program can be made by the node graph:



- The first node detects the keystrokes from the user and publishes the character of the pressed key to a topic (‘keys’)
- The second node subscribes to the ‘keys’ topic. It receives the character of the pressed key. Depending on the pressed key, it publishes the appropriate twist messages to the ‘cmd_vel’ topic for moving the robot

Launch Files

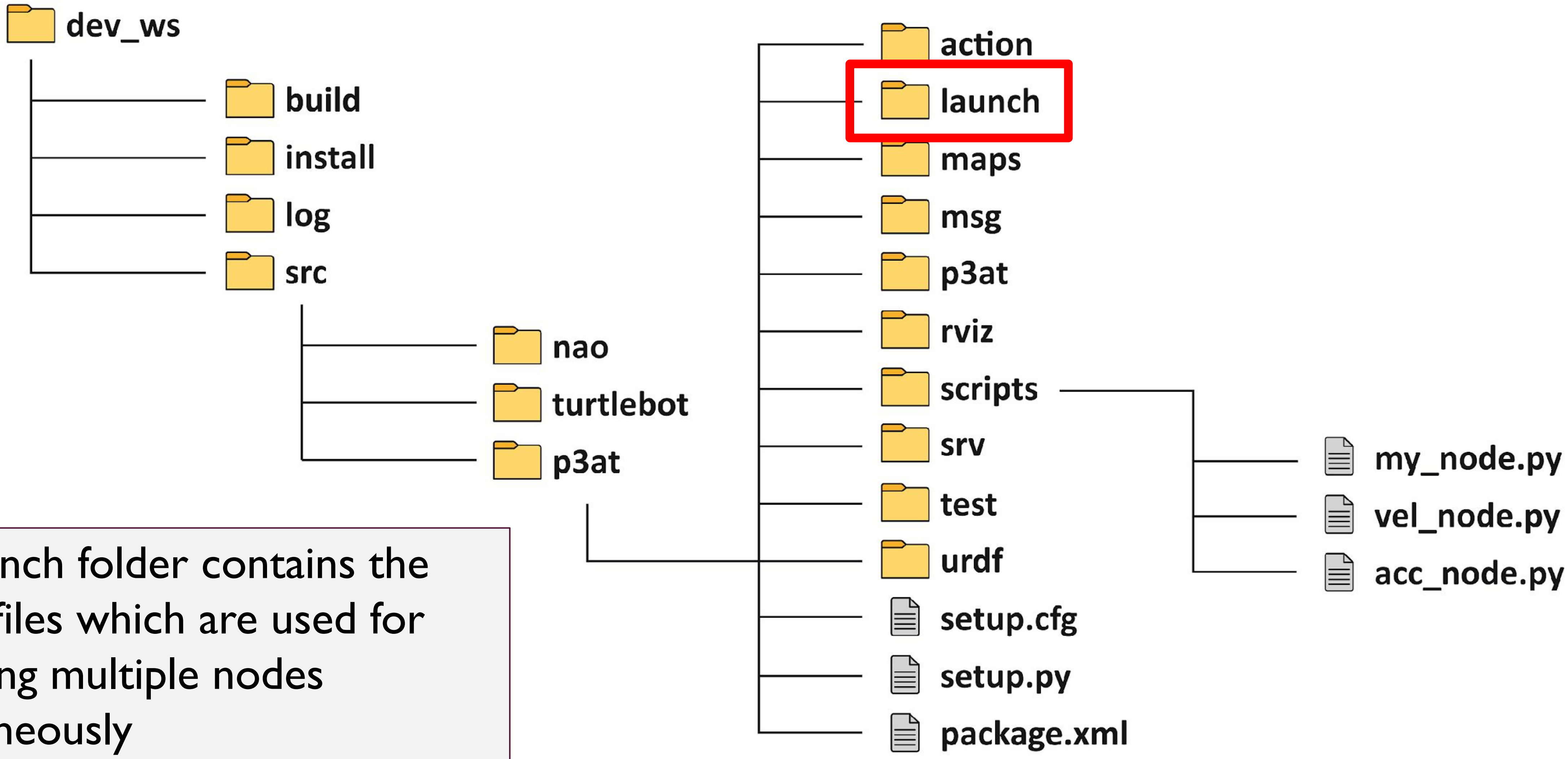
- So far, we have used the **ros2 run** command for execution of nodes:

```
ros2 run <package_name> <node_file>
```

- The ros2 run command is useful for starting individual nodes for testing and debugging purposes
- However, most robot systems end up having tens or hundreds of nodes that run at the same time. Executing all of these nodes one at a time is not practical
- In such cases, we can use the **ros2 launch** command to start all of the nodes simultaneously
- ros2 launch operates on *launch files*

```
ros2 launch <launch_file>
ros2 launch <package_name> <launch_file>
```

Launch Files



The launch folder contains the launch files which are used for executing multiple nodes simultaneously

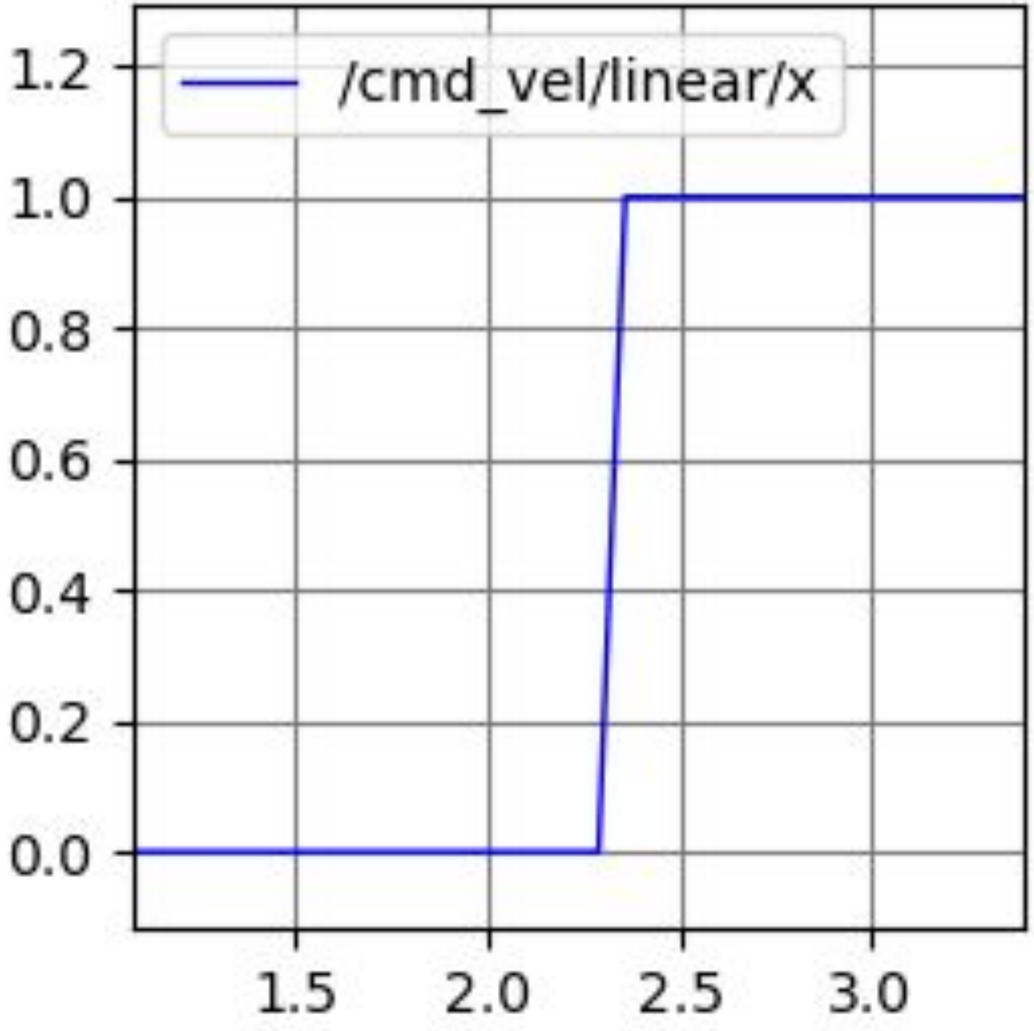
Launch Files

- A launch file is a Python/XML/YAML script that describes a collection of nodes to run (along with their topic remappings and parameters)

- Besides executing nodes, a launch file has other functions:
  - Start programs on other computers across the network via ssh
  - Auto respawn nodes that crash
  - Set parameters for configuring nodes
  - Start simulations with the required world
  - Bring models (sdf, urdf) into the simulation environment
  - Start rviz (a GUI for visualization purposes)

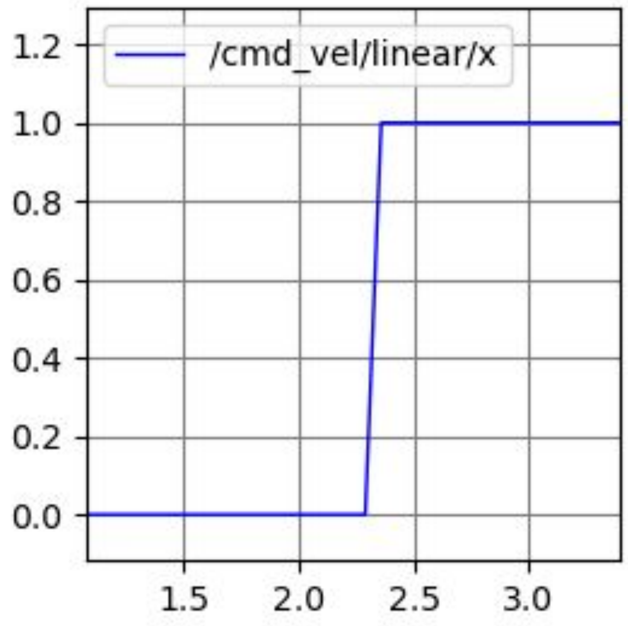
RQt Plots

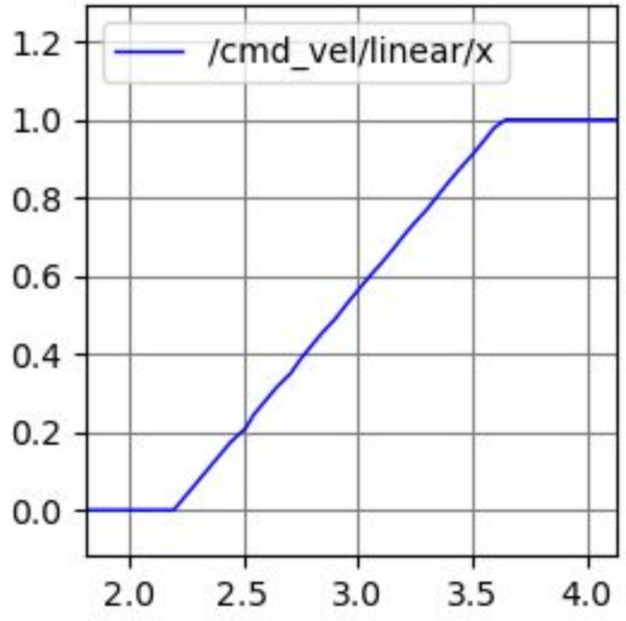
- RQt is a graphic-user interface provided by ROS
- RQt can be used to display the node graph, handle topics, send service requests and so on
- One use of RQt is to make real-time plots of messages on a topic
- The diagram shows the RQt plot for the linear velocity of the robot in its x-direction



Ramped Velocities

- It should be noted that the teleoperation program presented here has some limitations such as tapping the keys (instead of holding)
- One major limitation is that the velocities being given are step velocities. This is not a problem in the simulation because the robot is treated as a mechanical entity
- However, real robots cannot start/stop instantaneously; they undergo a finite acceleration
- When a hardware robot's motors try to suddenly jump to a different velocity, problems such as skidding, belt-slipping or breaking of the driveline can occur
- To avoid such problems, it is a good practice to ramp the velocities in a finite amount of time





Lab Tasks

- Download the manual from LMS
- Perform the Lab Tasks as given in the manual and submit it on LMS
- Remember to execute scripts with the terminal