

CENG 483 – BEHAVIORAL ROBOTICS

Homework Set #4

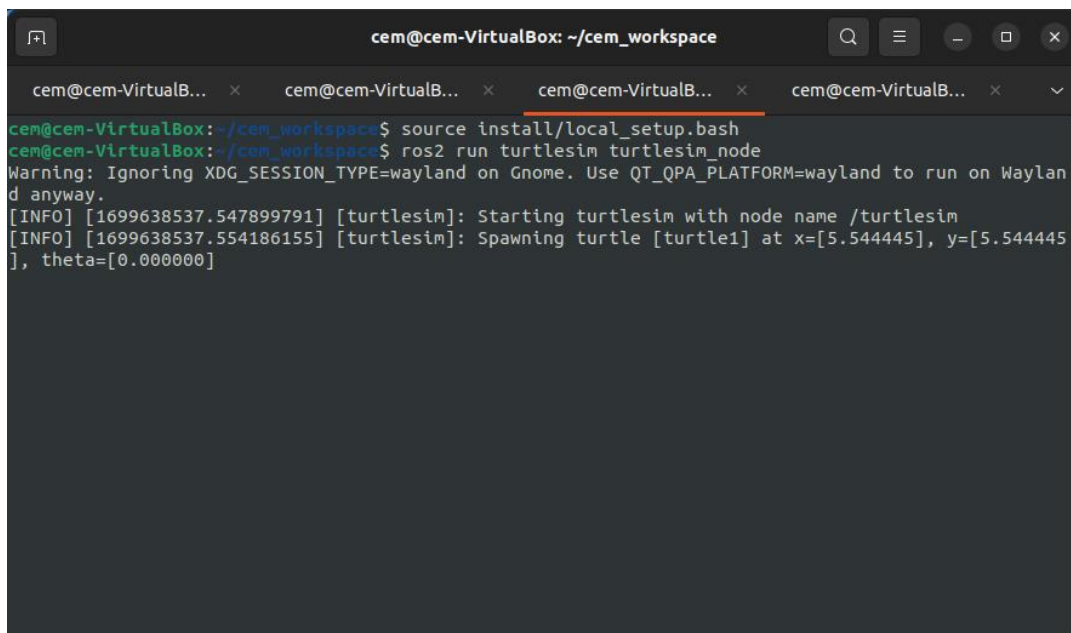
Due : 07.11.2023

Name : Cem Tolga MÜNYAS

Student ID : 270206042

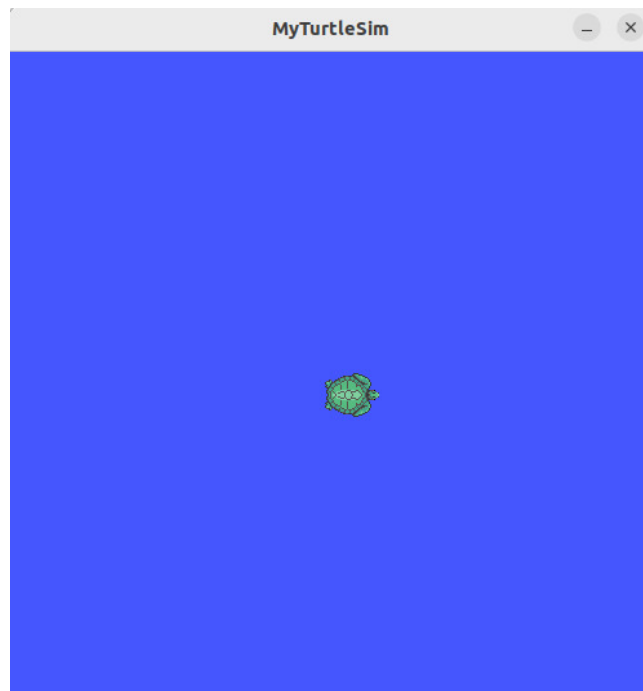
1. For the first question, we were asked to develop / write a controller for the turtle in the turtlesim simulation by teleoperating it where the turtle demonstrates a command in a completely opposite way as asked by the user / operator.

I started with sourcing the ROS2 in order to communicate with the node I will create. Then I created a node in turtlesim.

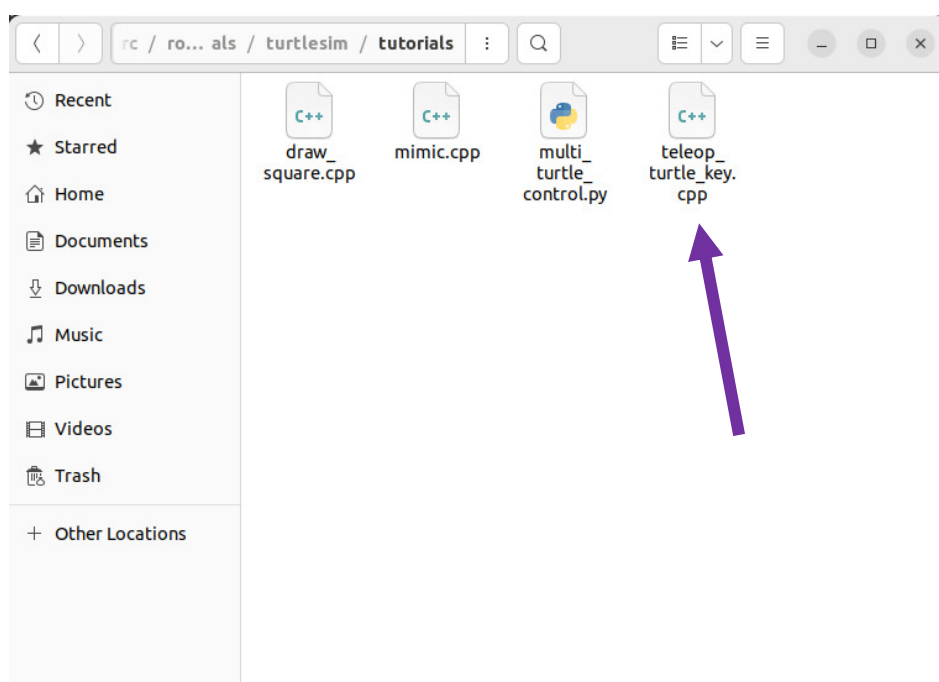
A terminal window titled 'cem@cem-VirtualBox: ~/cem_workspace' with four tabs. The terminal shows the following commands and output:

```
cem@cem-VirtualBox:~/cem_workspace$ source install/local_setup.bash
cem@cem-VirtualBox:~/cem_workspace$ ros2 run turtlesim turtlesim_node
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
[INFO] [1699638537.547899791] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [1699638537.554186155] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

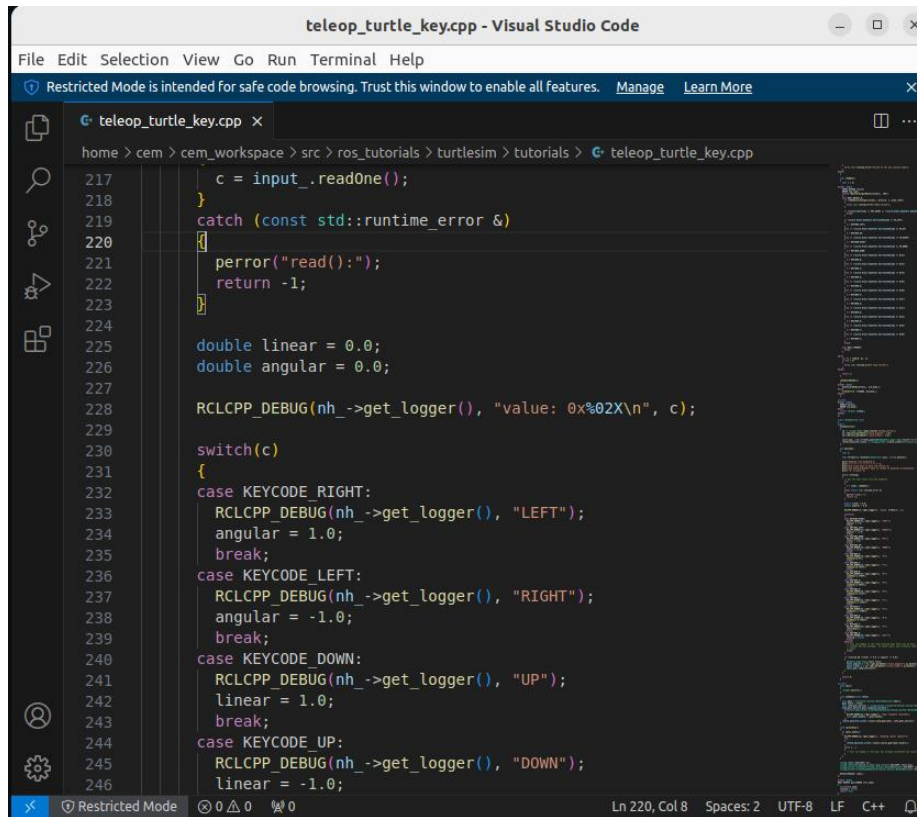
I applied `ros2 run turtlesim turtlesim_node` command so that a turtle appeared in the simulator window.



The question asks us to change some features so that our turtle could be able to move exactly the opposite way which we control. We need to find the file where the turtle takes orders and reorganize it according to our new requests. I know that the turtle takes command of movements with the feature teleoperation. After quick research on the internet, I found out that the teleoperation function file in ROS2 is *teleop_turtle_key.cpp*. So if we can find the code blocks which give command turtle to move, we can simply edit it according to our request.



In another terminal, I could be able to open the file (*teleop_turtle_key.cpp*) which I mentioned above.



```
217     c = input_.readOne();
218 }
219 catch (const std::runtime_error &)
220 {
221     perror("read():");
222     return -1;
223 }
224
225 double linear = 0.0;
226 double angular = 0.0;
227
228 RCLCPP_DEBUG(nh_ ->get_logger(), "value: 0x%02X\n", c);
229
230 switch(c)
231 {
232 case KEYCODE_RIGHT:
233     RCLCPP_DEBUG(nh_ ->get_logger(), "LEFT");
234     angular = 1.0;
235     break;
236 case KEYCODE_LEFT:
237     RCLCPP_DEBUG(nh_ ->get_logger(), "RIGHT");
238     angular = -1.0;
239     break;
240 case KEYCODE_DOWN:
241     RCLCPP_DEBUG(nh_ ->get_logger(), "UP");
242     linear = 1.0;
243     break;
244 case KEYCODE_UP:
245     RCLCPP_DEBUG(nh_ ->get_logger(), "DOWN");
246     linear = -1.0;
```

I could be able to find the code block which gives linear and angular directions to the turtle in the turtlesim. Since we want the turtle to move exactly the opposite direction with our commands, I changed the operands “Left to Right”, “Right to Left”, “Up to Down” and “Down to Up”. I saved the file with new version and exit.

```
switch(c)
{
case KEYCODE_RIGHT:
    RCLCPP_DEBUG(nh_ ->get_logger(), "LEFT");
    angular = 1.0;
    break;
case KEYCODE_LEFT:
    RCLCPP_DEBUG(nh_ ->get_logger(), "RIGHT");
    angular = -1.0;
    break;
case KEYCODE_DOWN:
    RCLCPP_DEBUG(nh_ ->get_logger(), "UP");
    linear = 1.0;
    break;
case KEYCODE_UP:
    RCLCPP_DEBUG(nh_ ->get_logger(), "DOWN");
    linear = -1.0;
    break;
```

So, we want to operate our script as newly edited. In order to work properly, we should use *colcon build* command in the underlay terminal to affect our workspace and the turtlesim simulator window.

```
cem@cem-VirtualBox: ~/cem_workspace
cem@cem-VirtualBox:~/cem_workspace/src$ cd ..
cem@cem-VirtualBox:~/cem_workspace$ rosdep install -i --from-path src --rosdist ro iron -y
#All required rosdeps installed successfully
cem@cem-VirtualBox:~/cem_workspace$ colcon build
[1.405s] WARNING:colcon.colcon_core.package_selection:Some selected packages are already built in one or more underlay workspaces:
'turtlesim' is in: /opt/ros/iron
If a package in a merged underlay workspace is overridden and it installs headers, then all packages in the overlay must sort their include directories by workspace order. Failure to do so may result in build failures or undefined behavior at run time.
If the overridden package is used by another package in any underlay, then the overriding package in the overlay must be API and ABI compatible or undefined behavior at run time may occur.

If you understand the risks and want to override a package anyways, add the following to the command line:
--allow-overriding turtlesim

This may be promoted to an error in a future release of colcon-override-check.
Starting >>> turtlesim
[Processing: turtlesim]
Finished <<< turtlesim [57.2s]
Summary: 1 package finished [58.1s]
```

In the final process, we are ready to teleoperate our turtle with the keys as we specified before. We are expecting it to move exactly the opposite direction as we command. In another terminal, I ran *turtle_teleop_key*, to manipulate our turtle in the turtlesim window.

We can also validate the opposite movements of our turtle through the screen recorder video.

```
cem@cem-VirtualBox: ~/cem_workspace
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
```

2. In the second question, we were asked to control turtles in two turtlesim simulators using keyboard, where turtles move in exactly the opposite way to each other. My method for this question was to edit the file *turtle_teleop_key.cpp* for the requirements. In this way, I will be able to launch the turtlesim simulator from one terminal using the commands *ros2 run turtlesim multisim_launch.py*.

While I was editing the *turtle_teleop_key.cpp* file, the first thing that I have done is creating two different publishers in order to send messages. These publishers were called *twist_pub_turtle1_* and *twist_pub_turtle2_*.

```
//Creating the publishers
twist_pub_ = nh_>create_publisher<geometry_msgs::msg::Twist>("turtle1/cmd_vel", 1);
twist_pub_turtle1_ = nh_>create_publisher<geometry_msgs::msg::Twist>("turtlesim1/turtle1/cmd_vel", 1);
twist_pub_turtle2_ = nh_>create_publisher<geometry_msgs::msg::Twist>("turtlesim2/turtle1/cmd_vel", 1);
```

Then I have redesigned the angular and linear movements for forward and inverse motion in the turtlesim. In this way, once we press the arrow keys to teleoperate the turtles, we will observe that one of them will make the forward motion and the other one will make the inverse motion.

```
if (running && (linear != 0.0 || angular != 0.0))
{
    geometry_msgs::msg::Twist twist;
    geometry_msgs::msg::Twist twist_inverse;
    // Calculating the angular movements for forward and inverse
    twist.angular.z = nh_>get_parameter("scale_angular").as_double() * angular;
    twist_inverse.angular.z = nh_>get_parameter("scale_angular").as_double() * -angular;
    // Calculating the linear movements for forward and inverse
    twist.linear.x = nh_>get_parameter("scale_linear").as_double() * linear;
    twist_inverse.linear.x = nh_>get_parameter("scale_linear").as_double() * -linear;

    twist_pub_>publish(twist);
    twist_pub_turtle1_>publish(twist);
    twist_pub_turtle2_>publish(twist_inverse);
}
}
```

In the last three rows of the code, I used pointers to publish the right messages to the *turtlesim1* and *turtlesim2* nodes.

```

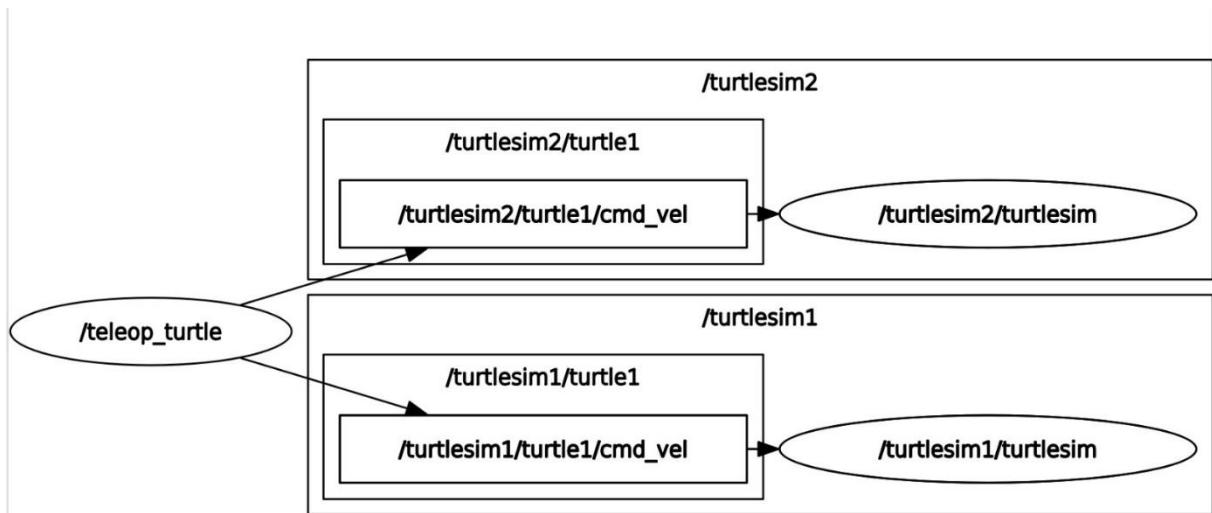
//Constructors
rclcpp::Node::SharedPtr nh_;
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr twist_pub_;
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr twist_pub_turtle1_;
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr twist_pub_turtle2_;

rclcpp_action::Client<turtlesim::action::RotateAbsolute>::SharedPtr rotate_absolute_client_;
rclcpp_action::ClientGoalHandle<turtlesim::action::RotateAbsolute>::SharedPtr goal_handle_;

KeyboardReader input_;
};

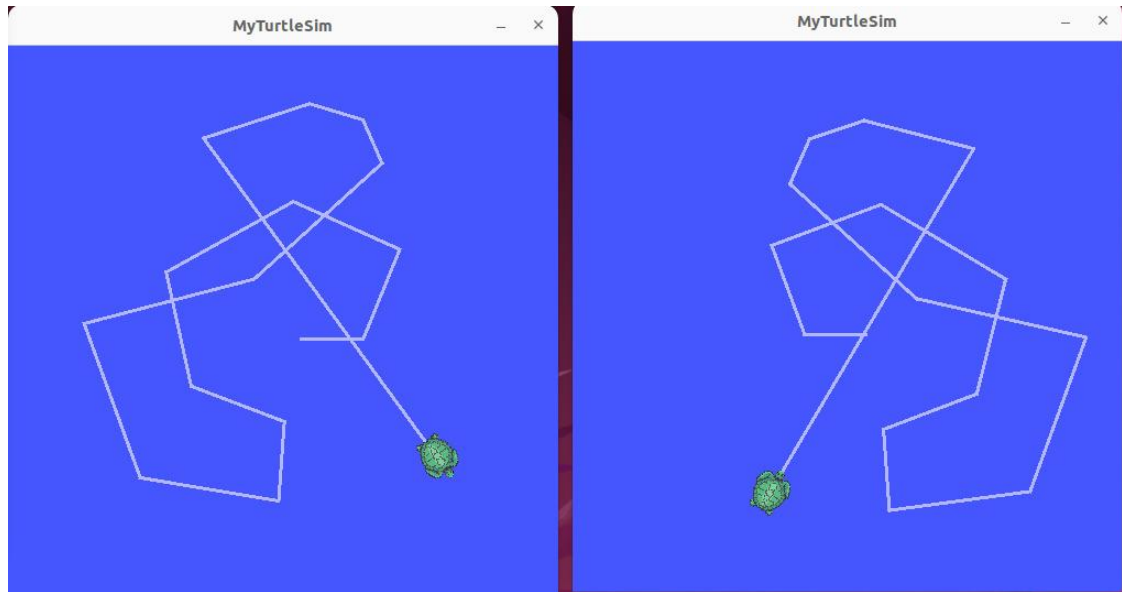
```

At the end of the `turtle_teleop_key.cpp` file, I have created two different constructors to operate the simulations. Using `rqt_graph`, I was able to visualize the nodes and the connections between the nodes and the teleoperation node.



Once editing the source code has finished, I used `colcon build` command in the underlay so that it could affect the overlay. When the building process is done, I was ready to run the turtlesim simulation in the terminal and use arrow keys to operate the turtles in two different simulations. Since we want to launch them from one terminal, we use `ros2 run turtlesim multisim_launch.py` command.

In the below figure, we can clearly see that while the first turtle in the left window was moving forward, the second turtle in the right window was moving reverse. (we can clearly observe the opposite motion in the screen recorder video.)



REFERENCES

- <https://docs.ros.org/en/iron/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html>
- <https://docs.ros.org/en/iron/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html>
- <https://docs.ros.org/en/iron/Tutorials/Beginner-CLI-Tools/Using-Rqt-Console/Using-Rqt-Console.html>
- Ozan Gülbaş