Mechatronics & Robotics
Mini Project 3*

Two launch files have been written for this project. A launch file for the main part of the project, which includes adding and rotating the turtle, and another including controlling the movement using arrow keys.

1st Alireza Kamali Ardakani
School of Mechanical Engineering
University of Tehran
Tehran, Iran
alirezakamali@ut.ac.ir

2nd Mehdi Tale Masouleh
School of Electrical and Computer Engineering
University of Tehran
Tehran, Iran
m.t.masouleh@ut.ac.ir

***Abstract*—In this project, the topics learned were implemented in the Linux operating system. In addition to that, on the platform Robot Operating System(ROS) in turtleSim Playground, the said tasks were implemented in the form of coding, and its performance can be adjusted and viewed by running the attached file. Said tasks include removing the default turtle on the environment, adding several turtles to the page according to the assigned character, and also rotating the added turtles, and, if desired, controlling their movement using keyboard keys. The ability to encounter and solve errors while writing the program was one of the most important challenges during this project. In addition, the ability to search correctly and communicate with the available tools was one of the most important features of this project.**

***Index Terms*—Robot Operating System, Linux, turtlesim, spawn, kill, rotating, python, Terminal**

I. Draw character with turtles!

At first, the kill service available in turtlesim topics should be called using commands. This command is a service in ROS, like what was said in the class before, and executes the kill command as a response.

Next, by using the spawn service code written in the class, after receiving the input (x coordinate, y coordinate, rotation, name), the desired value should be returned.

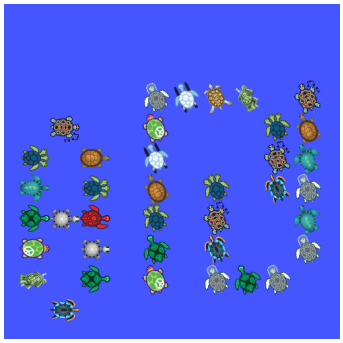

Fig. 1. Turtles in turtlesim playground

It should be noted that in these two services, try and except codes have been written for error handling, so that in case of any error, even if the code is run, it will show us an error in the output. It should be noted that these two items, kill and spawn, are the same service, and if a client node is written for them, they will be displayed as a separate node in the rqt graph.

A. Explanation of the Python Script for Adding turtles in ROS

The provided Python script is used to manage turtles in the Turtlesim simulator, a part of the Robot Operating System (ROS). The script includes functionalities to kill the default turtle and spawn multiple new turtles at specified locations and orientations.

```
#!/usr/bin/python3
import rospy
from turtlesim.srv import Spawn,Kill
```

These lines import the necessary libraries. The `rospy` library is the ROS Python client library. `Spawn` and `Kill` are service definitions from the `turtlesim` package used to create and remove turtles, respectively.

```
def kill_main_turtle(name):
    rospy.wait_for_service('/kill')
    try:
        kill_turtle = rospy.ServiceProxy
        ('/kill' , Kill)

        server_response = kill_turtle(name)
    except rospy.ServiceException as e:
        rospy.loginfo(f"Service call failed: %s"%e)
```

The `kill_main_turtle` function removes a turtle by name. It waits for the `/kill` service to become available and then attempts to call this service. If the service call fails, an exception is caught, and an error message is logged.

```
def spawner(x,y,theta,name):
    rospy.wait_for_service("/spawn")
    try:
```

```
        spawn_client  =rospy.ServiceProxy('/spawn', Spawn)
        resp = spawn_client(x,y,theta, name)
        return resp
    except:
        print("Failed")
```

The `spawner` function creates a new turtle at coordinates `(x, y)` with orientation `theta` and a specified `name`. Similar to the kill function, it waits for the `/spawn` service and then tries to call this service. If the service call fails, it prints "Failed".

```
if __name__ == '__main__':
    try:
        kill_main_turtle('turtle1')
```

The script execution starts here. It first tries to kill the default turtle named `turtle1`.

```
        # Spawn First Parameters Turtle
        spawner(1,2,0,'my_turtle1')
        spawner(1,3,0,'my_turtle2')
        spawner(1,4,0,'my_turtle3')
        spawner(1,5,0,'my_turtle4')
        etc
```

This block spawns multiple turtles with the first set of parameters. Each turtle is placed at a specific coordinate `(x, y)` with orientation `0` and given a unique name such as `my_turtle1`, `my_turtle2`, etc.

```
        # Spawn Second Parameters Turtles
        spawner(5,2,2,'my_turtle14')
        spawner(5,3,2,'my_turtle15')
        spawner(5,4,2,'my_turtle16')
        spawner(5,5,2,'my_turtle17')
        spawner(5,6,2,'my_turtle18')
        etc
```

This block spawns another set of turtles with different parameters. The coordinates `(x, y)` and orientation `theta` are different from the previous set, and each turtle is given a unique name from `my_turtle14` to `my_turtle38`.

```
    except rospy.ROSInternalException:
        pass
```

If there is an internal ROS exception during the script execution, it is caught, and the script is terminated gracefully.

II. Rotating the turtles!

To rotate the turtles added to the environment, the code of a turtle must be used first. This code is available by default on the ros wiki site, which is a powerful community in ROS, but the way to implement it in this project was different according to the demands.

A. Explanation of the Python Script for Rotating Turtles in Turtlesim

The provided Python script is used to rotate multiple turtles in the Turtlesim simulator, a part of the Robot Operating System (ROS). Below is a line-by-line explanation of the script.

```
#!/usr/bin/python3
import rospy
from geometry_msgs.msg import Twist
```

These lines import the necessary libraries. The `rospy` library is the ROS Python client library, and `Twist` is a message type from the `geometry_msgs` package used to specify linear and angular velocity in free space.

```
def rotate_turtle(turtle_names):

    rospy.init_node('rotate_turtle' , anonymous=True)
    publishers = {}
    rate= rospy.Rate(10)
```

The `rotate_turtle` function begins by initializing a new ROS node named `rotate_turtle`. The `anonymous=True` flag ensures that the node name is unique by appending random numbers to the node name. A dictionary `publishers` is created to hold publishers for each turtle, and a `rate` object is created to control the loop execution rate at 10 Hz.

```
    # Define the rotation speed
    vel_msg = Twist()
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 1
```

A `Twist` message, `vel_msg`, is defined to specify the velocity commands. The linear velocities in the x, y, and z directions are set to 0, while the angular velocity around the z-axis is set to 1, which means the turtle will rotate around the z-axis.

```
    # create publishers
    for turtle_name in turtle_names:
        publishers[turtle_name] =
        rospy.Publisher(...)
```

For each turtle in `turtle_names`, a publisher is created and stored in the `publishers` dictionary. The publisher sends `Twist` messages to the topic `/turtle_name/cmd_vel`, which controls the turtle's velocity.

```
    rospy.loginfo(...)
```

A log message is generated to indicate the start of the rotation process for the specified turtles.

```
    while not rospy.is_shutdown():
        for turtle_name ,
        pub in publishers.items():
            try:
                pub.publish(vel_msg)
        rate.sleep()
```

The main loop runs until ROS is shut down. Within this loop, for each turtle and its corresponding publisher, the `vel_msg` is published to the topic to command the turtle to rotate. If a `ROSInterruptException` occurs, it is logged as an error

specific to the turtle. Any other exceptions are also logged with an error message. The loop sleeps for the time specified by `rate` to maintain the 10 Hz execution rate.

```
def build_turtle_list(n):
    return [f"my_turtle{i}" for i in range(1,n+1)]
```

The `build_turtle_list` function generates a list of turtle names in the format `my_turtle1`, `my_turtle2`, ..., up to `my_turtlen`.

```
if __name__ == '__main__':
    try:
        turtles_list = build_turtle_list(38)
        rotate_turtle(turtles_list)
    except rospy.ROSInternalException:
        pass
```

The script execution starts here. The `build_turtle_list` function is called to create a list of 38 turtles. The `rotate_turtle` function is then called with this list. If a `ROSInternalException` occurs, it is caught, and the script is terminated gracefully.

III. Moving with arrow keys - BONOUS

A. Code Explanation

```
import sys, select, termios, tty
```

Importing necessary libraries and modules:

- `sys`, `select`, `termios`, `tty`: Standard Python modules for handling terminal input.

The `get_key` function captures key presses from the terminal. It configures the terminal to raw mode, reads the key press, and then restores the terminal settings.

The `move_turtles` function creates a `Twist` message based on the key pressed and publishes it to all turtles. The arrow keys map to specific linear velocities.

Initial setup for the script:

- `settings`: Save the current terminal settings.
- `rospy.init_node('turtle_controller')`: Initialize the ROS node.
- `turtle_names`: List of turtle names to be used.

```
publishers = {name: rospy.Publisher
(f'/{name}/cmd_vel',
Twist,
queue_size=10)
for name in
turtle_names}
```

Create a dictionary of publishers for each turtle to publish `Twist` messages to their `cmd_vel` topic.

Other code explanation is written in python attached script!

IV. Wall Avoidance!

After the code for the movement of the turtles was written, it was time to add something interesting to the other tasks. A modification was made to prevent the turtles from hitting the wall and making errors.

In this modification, I added a check in the `move_turtles` function to detect if the turtles are too close to the walls. If so, their linear velocity is set to zero to prevent them from smashing into the wall. Additionally, I added a loop to initialize the poses of all turtles before starting the control loop, ensuring that we have up-to-date information about their positions.

And finally i got this result :)

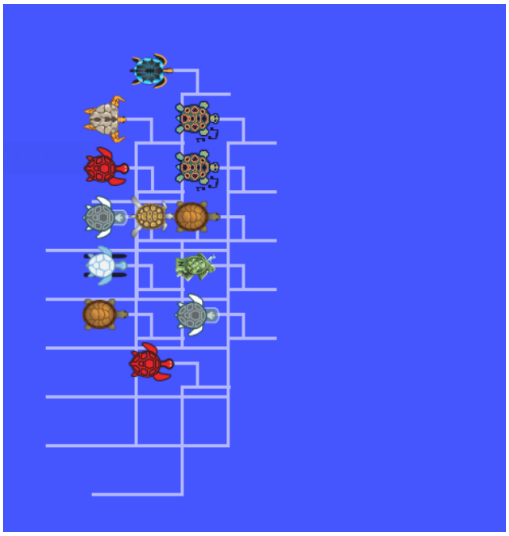

Fig. 2. keyboard Control Result

V. Errors!

To overcome errors during the project, LLMs like chatGPT were sometimes used, but many questions and uncertainties were answered on the ROS-Answers site. Questions such as how to use services and topics and messages.

A. Rotate Multi Turtle at the same Time

One of these challenges was how to rotate the turtles together at the same time, and for this, a list of created turtles was prepared using the following code.

```
def build_turtle_list(n):
    return [f"my_turtle{i}" for i in range(1,n+1)]
```

In the Rotate Function, A dictionary `publishers` is created to hold ROS publishers for each turtle. Each publisher is responsible for sending velocity commands (`Twist` messages) to the corresponding turtle. and in the loop continuously publishes the rotation command to each turtle's topic.

B. typeError

Another small challenge that we faced was an error that occurred during the execution of the rotate function there was a problem with how to import data into the loop, and it was nothing but converting string to float32. For this, I noticed that in the example code available on the ROS wiki site, the value of the speed and rotation of the turtles is accepted as input

```
Command 'pip3' not found, but can be installed with:

apt install python3-pip
Please ask your administrator.
```

```
python3 get-pip.py --user
```

from the user, but by simplifying the code and applying fixed input in the code, this problem was solved.

```
Everything Fine
Let's rotate yout robot
Input your speed (degree/sec):30
type your distance(degree):30
Clockwise: 30
Traceback (most recent call last):
  File "/home/alireza/catkin_ws/src/kamali_rotate/src/turtle_rotate.py", line 5
2, in <module>
    rotate()
  File "/home/alireza/catkin_ws/src/kamali_rotate/src/turtle_rotate.py", line 1
9, in rotate
    angular_speed = speed*2.0*PI/360.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Fig. 3. typeError encountered in the project

C. Sourcing the Run Files

Another error that I sometimes encountered due to negligence was not sourcing the project files so as not to run into problems while running the codes. This series of steps starts with the execution of the master node named roscore, continues with the execution of the turtlesim environment, and finally ends by writing a command similar to

(rosrun , package-name , python-script.py)

in the Linux terminal. It was remarkable in the way it was executed.

Sometimes it took me a few minutes to find the file, but finally, I noticed this minor error in the execution of the files. Of course, the right solution was to write a launch file at the beginning of the project and run that file every time to avoid wasting time.

D. Installing packages on Linux system

Many problems were encountered in the implementation and solution of the last task, which was control using arrow keys.

Problems such as admin's lack of access to library installation in Linux can be solved with the following code.

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

E. Out of My sight!

Writing the following code must be written in the first line of the Python script.

```
#!/usr/bin/python3
```

```
setting /run_id to 9805f288-265b-11ef-a2b9-ed674a9b7a4
process[rosout-1]: started with pid [12734]
started core service [/rosout]
process[turtlesim-2]: started with pid [12737]
RLException: Unable to launch [rotating__node-3].
If it is a script, you may be missing a '#!' declaration at the top.
```

VI. Conclusion

In this project, we successfully implemented various tasks using the Robot Operating System (ROS) within the Turtlesim simulator on a Linux platform. We demonstrated the ability to manipulate the Turtlesim environment by removing and spawning turtles, as well as rotating and controlling their movement through keyboard inputs. The project also highlighted the importance of error handling, efficient search techniques, and effective communication with available tools and resources.

Key accomplishments include:

- Implementing services to kill and spawn turtles dynamically.
- Developing Python scripts to manage and rotate turtles using ROS services.
- Creating a user-friendly interface to control turtle movements with keyboard inputs.
- Adding a wall avoidance feature to enhance the robustness of the turtle control system.
- Overcoming various challenges, such as handling ROS exceptions, type conversion errors, and Linux system administration issues.

The project reinforced our understanding of ROS and its practical applications in robotics. Additionally, it provided valuable experience in debugging and problem-solving, particularly in a Linux environment. The successful execution of the tasks demonstrated the effectiveness of our approach and the potential for further exploration and enhancement of robotic systems using ROS.

Overall, this project was a comprehensive exercise in applying theoretical knowledge to practical scenarios, and it underscored the significance of continuous learning and adaptation in the field of robotics and automation.

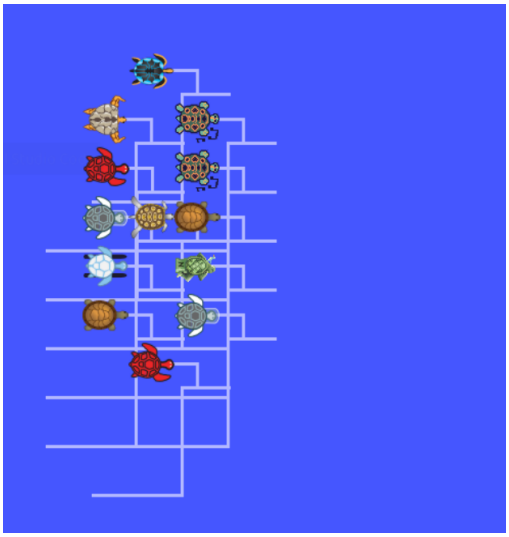

Fig. 4. Keyboard Control with Arrow Keys Results

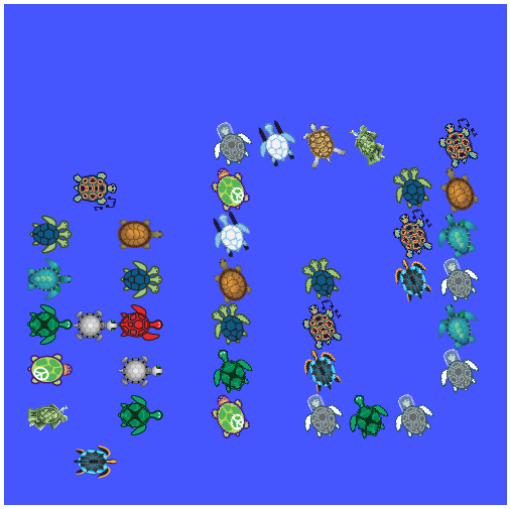

Fig. 5. Adding and Rotating Results

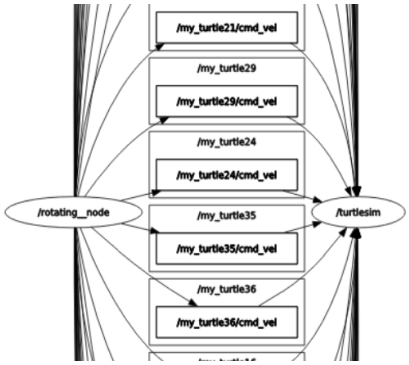

Fig. 6. rqt Graph of nodes

REFERENCES

[1] J. Angeles, Fundamentals of Robotic Mechanical Systems — Theory, Methods, and Algorithms, 01 2007, vol. 124.

[2] ”Turtlesim Tutorials: Rotating Left and Right,” ROS Wiki. [Online]. Available: https://wiki.ros.org/turtlesim/Tutorials/Rotating

[3] ”Turtlesim,” ROS Wiki. [Online]. Available: https://wiki.ros.org/turtlesim. [Accessed: Jun. 9, 2024].

[4] R. N. Jones, ”How to install software on Linux from the command line,” OpenSource.com, Aug. 10, 2018. [Online]. Available: https://opensource.com/article/18/8/how-install-software-linux-command-line. [Accessed: Jun. 9, 2024].