Lab Session 6
TurtleBot Class + roslaunch + rosparams

1. Creating the TurtleBot Class

We want to spawn multiple turtles in our environment and move them in a swarm configuration or individually. To do this, we will create our own TurtleBot class that stores all the variables and methods(functions) relating to each individual turtle. Each turtle therefore becomes an object of the TurtleBot class.

Create a new package and add the following node and call it swarm.py :

```
import rospy
from turtlesim.srv import *
from turtlesim.msg import Pose
rospy.init_node('Swarm_Turtles')
class TurtleBot:

        def __init__(self, name, initial_x, initial_y, initial_theta):

                self.name = name
                self.spawnTurtle(name, initial_x, initial_y, initial_theta)

                self.position = Pose()
                self.pose_subscriber = rospy.Subscriber(f'{name}/pose', Pose,
self.update_pose)
                self.rate = rospy.Rate(10)

        def spawnTurtle(self, name, x, y, theta):
                rospy.wait_for_service('spawn')
                server = rospy.ServiceProxy('spawn', Spawn)
                request = server (x, y, theta, name)

        def update_pose(self, data):
                self.position.x = data.x
                self.position.y = data.y
                self.position.theta = data.theta

if __name__ == '__main__':
        t1 = TurtleBot('T1', 1, 1, 0)
        t2 = TurtleBot('T2', 1, 9, 0)
        t3 = TurtleBot('T3', 9, 1, 0)
        t4 = TurtleBot('T4', 9, 9, 0)
        rospy.spin()
```

To to add (spawn) a new turtle in our turtlesim environment, four parameters need to be passed : the name of the turtle (make sure it's unique) and the initial location and configuration of the turtle in the turtlesim window. The node above after running will spawn four turtles in each corner of the window.

Open three terminal tabs and run the following commands:

Tab1:

```
$ roscore
```

Tab2:

```
$ rosrun turtlesim turtlesim_node
```

Tab3:

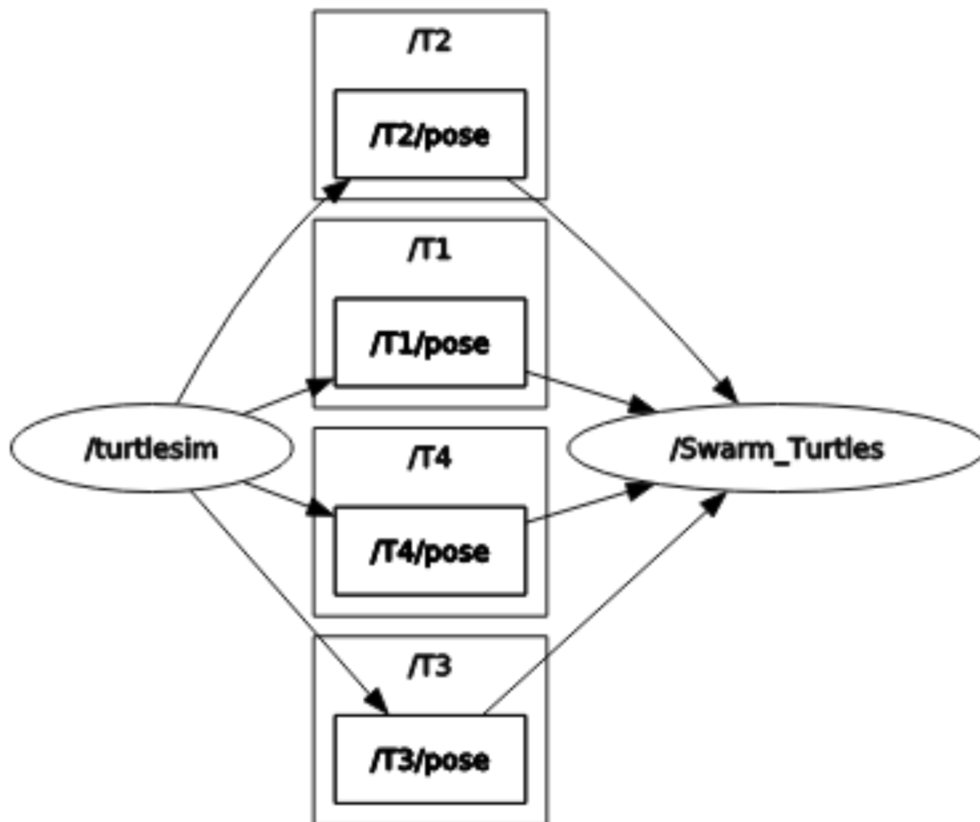
```
$ rosrun <your_package> swarm.py
```

Make sure to add the swarm.py node to your CMAKELISTS.txt file.

After running the code you will get the following output on your turtlesim window:

TurtleSim

And your rqt_graph will look like this:



جامعة المنارة
MANARA UNIVERSITY

2. Rosparam

rosparam contains the rosparam command-line tool for getting and setting ROS Parameters on the Parameter Server using YAML-encoded files. It contains a collection of parameters and variables that our project depends on.

Instead of adding each turtle manually In our swarm.py node, we will modify the node to retrieve all the parameters necessary from our rosparam file and spawn the turtles based on the retrieved data. We will add each class parameter as a list(array) inside our .yaml file. The .yaml file could be stored anywhere in our package but it is common to be placed inside of a /config directory inside our package.

Navigate to your package and add a /config directory inside

```
$ roscd <your_package>
$ mkdir config
```

Add a params.yaml folder inside the /config directory

```
$ cd config
$ gedit params.yaml
```

Add the following lines inside and save the file

```
names : ['T1', 'T2', ‘T3', ‘T4']
initial_x : [1, 1, 9, 9]
initial_y: [1, 9, 1, 9]
```

Please note these parameters need to be loaded onto our environment using the $rosparam command to use them:

```
$ rosparam list
$ roscd <your_package>
$ rosparam load config/params.yaml
$ rosparam list
```

Now your parameters should be loaded onto your environment and you can view them inside your terminal using:

```
$ rosparam get names
```

Now, modify the (if) block to load the parameters inside the node and spawn the turtles accordingly:

```
if __name__ == '__main__':
     names = rospy.get_param('names')
     x = rospy.get_param('initial_x')
     y = rospy.get_param('initial_y')

     turtles = list(zip(names, x, y))
     print(turtles)
     objects = []
     for i in turtles:
          objects.append(TurtleBot(i[0], i[1], i[2], 0))

     rospy.spin()
```

Now to run our node we will follow the same steps as before but we need to add the command that loads our parameters.

Tab1:

```
$ roscore
```

Tab2:

```
$ roscd <your_package>
$ rosparam load config/params.yaml
```

Tab3:

```
$ rosrun turtlesim turtlesim_node
```

Tab4:

```
$ rosrun <your_package> swarm.py
```

(You can combine Tab2 with Tab3 or Tab4)
Now we will get the same output as before, the difference is the turtles spawned were based on the information provided in our params.yaml file.

3. Roslaunch

roslaunch is a tool for easily launching multiple ROS nodes locally and remotely. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration (with the .launch extension) files that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

To demonstrate its use let's look at what we previously had to execute to launch our project. We opened four different tabs and executed different commands in each. If we were to add any other nodes to our package we

would have to also open another separate tab. This is where roslaunch comes to play, it will allow us to run all our nodes as well as run our parameters file using the $roslaunch command. To create a launch file, create a /launch directory inside your package and add a swarm.launch file

```
$ roscd <your_package>
$ mkdir launch
$ gedit swarm.launch
```

Add the following lines inside and save:

```
<launch>

<rosparam command="load" file="$(find <your_package>)/config/params.yaml” />
<node pkg="turtlesim" type="turtlesim_node" name="turtle" respawn="true" />
<node pkg="<your_package>" type="swarm.py" name="swarm” />


</launch>
```

The $roslaunch file has the following syntax:

```
$ roslaunch <package_name> <launch_file.launch>
```

Open a single terminal window (without launching roscore) and run the following:

```
$ roslaunch <your_package> swarm.launch
```

Please not that $roslaunch also launches $roscore if it’s not already running.

You should see the same output as before but now we have launched all our nodes and parameters With a single command.

4. Controlling our Turtles

We will add a few methods inside our class that allows us to control the turtle to navigate it to a desired (x, y) coordinate

Modify the TurtleBot class to include these methods:

```
class TurtleBot:
      def __init__(self, name, initial_x, initial_y, initial_theta):

            self.name = name
            self.spawnTurtle(name, initial_x, initial_y, initial_theta)

            self.position = Pose()
            self.pose_subscriber = rospy.Subscriber(f'{name}/pose', Pose,
self.update_pose)
            self.velocity_publisher = rospy.Publisher(f'{name}/cmd_vel', Twist,
queue_size=10)
            self.rate = rospy.Rate(10)

      def spawnTurtle(self, name, x, y, theta):
            rospy.wait_for_service('spawn')
            server = rospy.ServiceProxy('spawn', Spawn)
            request = server (x, y, theta, name)

      def update_pose(self, data):
            self.position.x = data.x
            self.position.y = data.y
            self.position.theta = data.theta


      def steering_angle(self, goal_pose):
            return math.atan2(goal_pose.y - self.position.y, goal_pose.x -
self.position.x)

      def angular_velocity(self, goal_pose, constant=6):
            return constant * (self.steering_angle(goal_pose)-self.position.theta)


      def linear_velocity(self, goal_pose, constant=0.5):
            return constant * self.euclidean_distance(goal_pose)


      def euclidean_distance(self, goal_pose):
            return math.sqrt((goal_pose.x - self.position.x)**2 + (goal_pose.y -
self.position.y)**2)
```

جَـامعة
المَـنارة
MANARA UNIVERSITY

```
    def move_to_goal(self, x, y):
        goal_pose = Pose()
        goal_pose.x = x
        goal_pose.y = y
        tolerance = 0.5
        velocity = Twist()
        while self.euclidean_distance(goal_pose) >= tolerance:

            velocity.linear.x = self.linear_velocity(goal_pose)
            velocity.angular.z = self.angular_velocity(goal_pose)

            self.velocity_publisher.publish(velocity)
            self.rate.sleep()

        velocity.linear.x = 0
        velocity.angular.z = 0
        self.velocity_publisher.publish(velocity)

if __name__ == '__main__':
    names = rospy.get_param('names')
    x = rospy.get_param('initial_x')
    y = rospy.get_param('initial_y')
    turtles = list(zip(names, x, y))
    print(turtles)
    objects = []
    for i in turtles:
        objects.append(TurtleBot(i[0], i[1], i[2], 0))

    objects[2].move_to_goal(9, 9)
    objects[2].move_to_goal(1, 1)
    rospy.spin()
```

After adding the necessary functions as well as a velocity publisher to our constructor to control our robot we will add two extra lines in the (if) block that will allow the third the turtle ‘T3’ to move towards ‘T2’ then turn back around towards ‘T1’. After running \$roslaunch You should see the four turtles spawning and then right afterwards the third turtle moving as explained earlier:

TurtleSim