ROS Lab 3

Vipul Dinesh, 220929024, MTE-A-09

General:

- **P (Proportional) Control:**
  Adjusts the output proportionally to the error. It responds quickly to changes but may leave a steady-state error.
- **PI (Proportional-Integral) Control:**
  Combines proportional control with integral control, which accumulates the error over time. It reduces steady-state error, improving accuracy.
- **PID (Proportional-Integral-Derivative) Control:**
  Adds derivative control to PI, which considers the rate of error change. It improves stability and response, reducing overshoot and oscillations.
- `turtlesim` package has two notable executables – `turtlesim_node` to generate a simple simulator and `turtle_teleop_key` to control the previously mentioned node using a keyboard.

1. Create turtle_control package

```
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_python turtle_control --dependencies
rclpy
```

2. Create an executable/node called `turtle_controller.py` in the directory `~/ros2_ws/src/turtle_control/turtle_control` to achieve proportional control of the turtle to reach a point, say (9,9)

`turtle_controller.py`

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

# Importing necessary libraries and modules from ROS2 and Python
import rclpy
from rclpy.node import Node
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist
import math

# Defining the TurtleControllerNode class that inherits from Node
class TurtleControllerNode(Node):
    # Constructor method for the class
    def __init__(self):
```

```
        # Calling the parent class constructor and naming the node
"turtle_controller"
        super().__init__("turtle_controller")

        # Setting target coordinates for the turtle
        self.target_x = 9.0
        self.target_y = 9.0

        # Initializing the pose variable
        self.pose_ = None

        # Creating a publisher for the cmd_vel topic to send velocity
commands to the turtle
        self.cmd_vel_publisher_ = self.create_publisher(Twist,
"turtle1/cmd_vel", 10)

        # Creating a subscriber for the pose topic to get the turtle's
current pose
        self.pose_subscriber_ = self.create_subscription(Pose,
"turtle1/pose", self.callback_turtle_pose, 10)

        # Creating a timer to call the control loop periodically (every 0.01
seconds)
        self.control_loop_timer_ = self.create_timer(0.01,
self.control_loop)



    # Callback function to update the pose of the turtle
    def callback_turtle_pose(self, msg):
        self.pose_ = msg

    # Control loop function to send velocity commands to the turtle
    def control_loop(self):
        # If the pose is not yet received, do nothing
        if self.pose_ is None:
            return

        # Calculate the distance between the current pose and the target
        dist_x = self.target_x - self.pose_.x
        dist_y = self.target_y - self.pose_.y
        distance = math.sqrt(dist_x * dist_x + dist_y * dist_y)

        # Create a Twist message to send velocity commands
        msg = Twist()
```

```python
        # If the distance is greater than a threshold, set linear and
angular velocities
        if distance > 0.5:
            msg.linear.x = distance

            # Calculate the angle to the goal
            goal_theta = math.atan2(dist_y, dist_x)
            diff = goal_theta - self.pose_.theta

            # Normalize the angle difference
            if diff > math.pi:
                diff -= 2 * math.pi
            elif diff < -math.pi:
                diff += 2 * math.pi

            msg.angular.z = diff
        else:
            # If the turtle is close enough to the target, stop the turtle
            msg.linear.x = 0.0
            msg.angular.z = 0.0

        # Publish the velocity command
        self.cmd_vel_publisher_.publish(msg)

    # Callback function for the move_location service
    def callback_get_distance(self, request, response):
        # Calculate the distance to the requested location
        x = request.loc_x - self.pose_.x
        y = request.loc_y - self.pose_.y
        response.distance = math.sqrt(x * x + y * y)
        return response

# Main function to initialize and spin the ROS2 node
def main(args=None):
    # Initialize the rclpy library
    rclpy.init(args=args)

    # Create an instance of the TurtleControllerNode
    node = TurtleControllerNode()

    # Spin the node so its callbacks are called
    rclpy.spin(node)

    # Shutdown the rclpy library
```

```
    rclpy.shutdown()

# Entry point of the script
if __name__ == "__main__":
    main()
```

3. Make changes in `setup.py` to set `turtle_controller.py` as an executable named `control`

setup.py

```
from setuptools import find_packages, setup

package_name = 'turtle_control'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='vipul',
    maintainer_email='vipul@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
        "control=turtle_control.turtle_controller:main"
        ],
    },
)
```

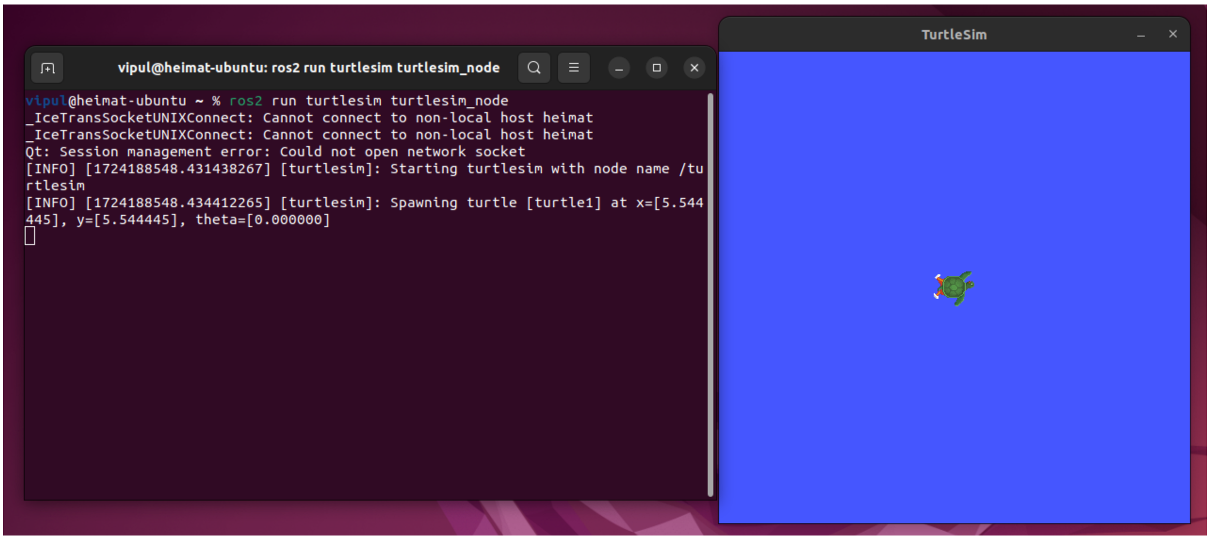
4. Rebuild the package and source it again

```
cd ~/ros2_ws/
colcon build --packages-select turtle_control
source ~/ros2_ws/install/setup.zsh
```

5. Run `turtlesim:turtlesim_node` in one terminal and `turtle_control:control` in another to move it to the desired position

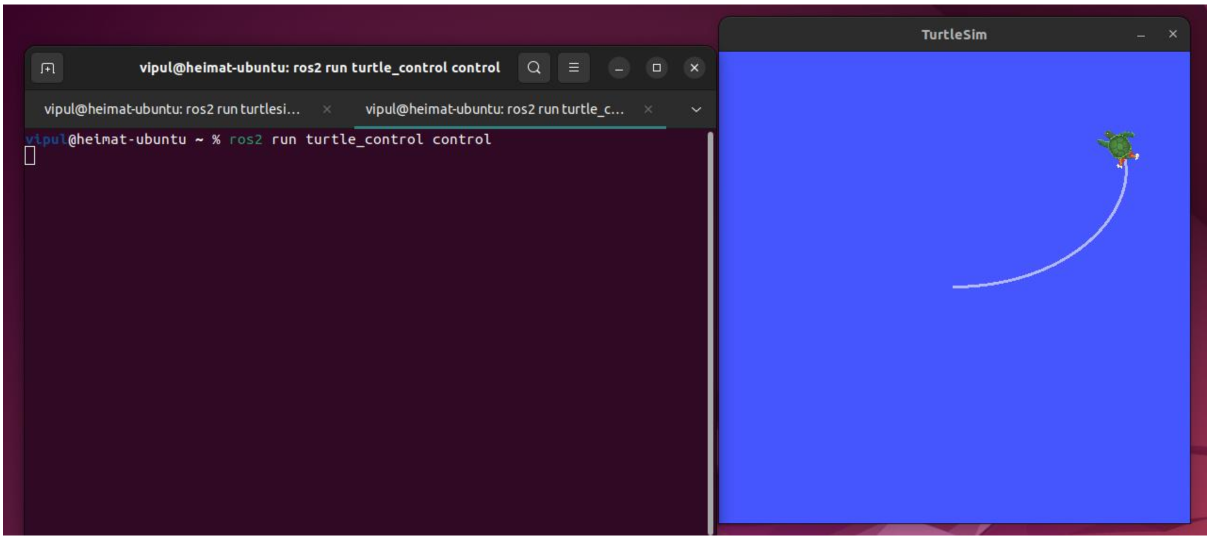
Terminal #1

```
ros2 run turtlesim turtlesim_node
```



Terminal #2

```
ros2 run turtle_control control
```



6. Create a launch file to run `turtlesim:turtlesim_node` and `turtle_control:control` using a single command

6.1. Create a `launch` folder in `turtle_control` folder and add `turtle.launch.py` in it

```
cd ~/ros2_ws/src/turtle_control
mkdir launch
cd launch
```

```
touch turtle.launch.py
chmod +x turtle.launch.py
```

turtle.launch.py

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
    Node(
        package='turtlesim',
        executable='turtlesim_node',
        output='screen'),
    Node(
        package='turtle_control',
        executable='control',
        output='screen'),
    ])
```

6.2. Modify `setup.py` file to recognise launch files upon rebuilding

setup.py

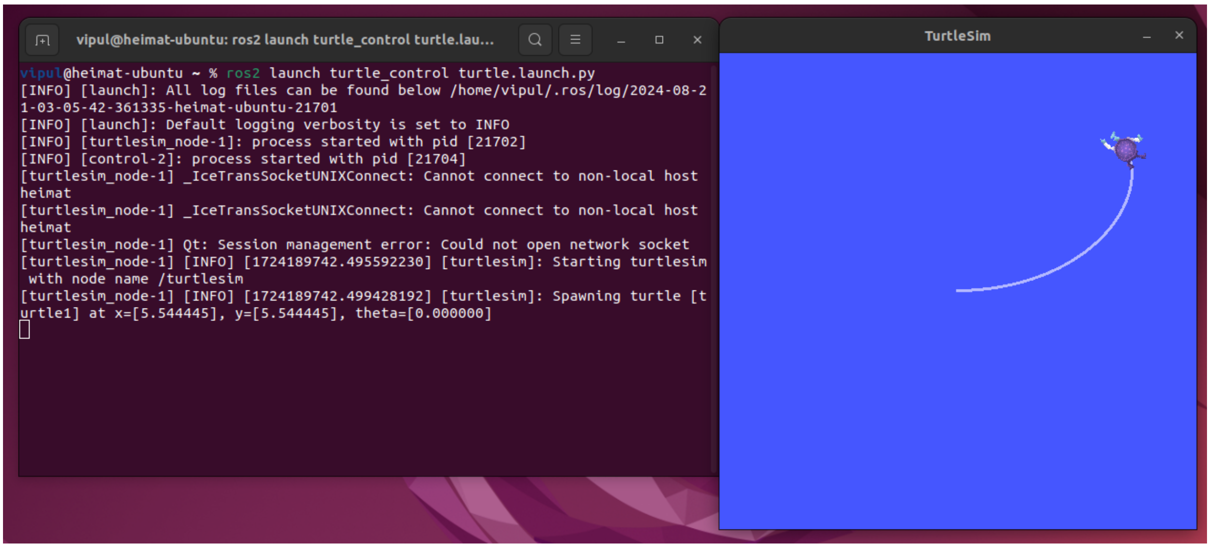
```python
from setuptools import find_packages, setup
from glob import glob
import os

package_name = 'turtle_control'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*')),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='vipul',
    maintainer_email='vipul@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
```

```
        tests_require=['pytest'],
        entry_points={
            'console_scripts': [
            "control=turtle_control.turtle_controller:main"
            ],
        },
    )
```

6.3. Rebuild the package and source it again (Repeat [Step 4](#))

7. Launch the newly created `turtle.launch.py`

```
ros2 launch turtle_control turtle.launch.py
```



8. Create an executable/node called `turtle_controller_pid.py` in the directory `~/ros2_ws/src/turtle_control/turtle_control` to achieve PID control of the turtle to reach a point, say (9,9)

`turtle_controller_pid.py`

```
#!/usr/bin/env python3
import rclpy
import math
from rclpy.node import Node
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose

class Controller_Node(Node):
    def __init__(self):
        super().__init__('turtle_control')
        self.get_logger().info("Node Started")
```

```
        self.desired_x = 9.0  # Adjust as needed
        self.desired_y = 9.0  # Adjust as needed




        # Publisher and Subscriber
        self.my_pose_sub = self.create_subscription(Pose,
"/turtle1/pose", self.pose_callback, 10)
        self.my_vel_command = self.create_publisher(Twist,
"/turtle1/cmd_vel", 10)

    def pose_callback(self, msg: Pose):
        #self.get_logger().info(f"Current x={msg.x} current
y={msg.y} and current angle = {msg.theta}")

        integral_dist = 0.0
        previous_err_dist = 0.0
        integral_theta = 0.0
        previous_err_theta = 0.0
        # Calculate errors in position
        err_x = self.desired_x - msg.x
        err_y = self.desired_y - msg.y
        err_dist = (err_x**2+err_y**2)**0.5

        # Distance error (magnitude of the error vector)

        #self.get_logger().info(f"Error in x {err_x} and error
in y {err_y}")

        # Desired heading based on the position error
        desired_theta = math.atan2(err_y, err_x)

        # Error in heading
        err_theta = desired_theta - msg.theta

        # Handle wrap-around issues (e.g., if error jumps from
+pi to -pi)
        while err_theta > math.pi:
            err_theta -= 2.0 * math.pi
        while err_theta < -math.pi:
            err_theta += 2.0 * math.pi
        #self.get_logger().info(f"Desired Angle =
{desired_theta} current angle {msg.theta} Error angle {err_theta}")
```

```
            # P (ID not required) for linear velocity (distance
control)

            Kp_dist = 0.4
            Ki_dist = 0.1
            Kd_dist = 0.08
            Kp_theta = 2
            Ki_theta = 0.1
            Kd_theta = 0.01



            integral_dist += err_dist
            derivative_dist = err_dist - previous_err_dist
            integral_theta += err_theta
            derivative_theta = err_theta - previous_err_theta


            # TODO: Add integral and derivative calculations for
complete PID

            # PID control for linear velocity
            #l_v = Kp_dist * abs(err_x) # + Ki_dist * integral_dist
+ Kd_dist * derivative_dist
            if err_dist >= 0.1: #checking whether error distance
within tolerence
                l_v = Kp_dist * abs(err_dist) + Ki_dist *
integral_dist + Kd_dist * derivative_dist
                previous_err_dist = err_dist
            else:
                self.get_logger().info(f"Turtlesim  stopping goal
distance within tolerence")
                l_v = 0.0


            # PID control for angular velocity
            if err_theta >=0.08: #checking whether heading angle
error within tolerence
                a_v = Kp_theta * err_theta + Ki_theta *
integral_theta + Kd_theta * derivative_theta
                previous_err_theta = err_theta

            else:
                self.get_logger().info(f"Turtlesim  stopping goal
heading within tolerence")
```

```
                a_v = 0.0

            # Send the velocities
            self.my_velocity_cont(l_v, a_v)

        def my_velocity_cont(self, l_v, a_v):
            #self.get_logger().info(f"Commanding liner ={l_v} and
angular ={a_v}")
            my_msg = Twist()
            my_msg.linear.x = l_v
            my_msg.angular.z = a_v
            self.my_vel_command.publish(my_msg)

    def main(args=None):
        rclpy.init(args=args)
        node = Controller_Node()
        rclpy.spin(node)
        node.destroy_node()
        rclpy.shutdown()

    if __name__ == '__main__':
        main()
```

9. Modify `setup.py` to make `turtle_controller_pid.py` as an executable called `control_pid` (similar to Step 3)

10. Generate a launch file called `turtle_pid.launch.py` to run both `turtlesim:turtlesim_node` and `turtle_control:control_pid` using a single command (similar to Step 6 - includes rebuild)

11. Launch the newly created `turtle_pid.launch.py`

```
ros2 launch turtle_control turtle_pid.launch.py
```