

HW4 ME5501 – Robotics and Unmanned Systems

Teddy Krulewich

September 28, 2022

Problem 1

Complete the following ROS tutorials at <https://docs.ros.org/en/foxy/Tutorials.html>

- Understanding ROS Nodes
- Understanding ROS Topics

Save a screenshot of the Turtlebot simulator and show/print the screenshot.



Screenshot of Turtlebot Simulator

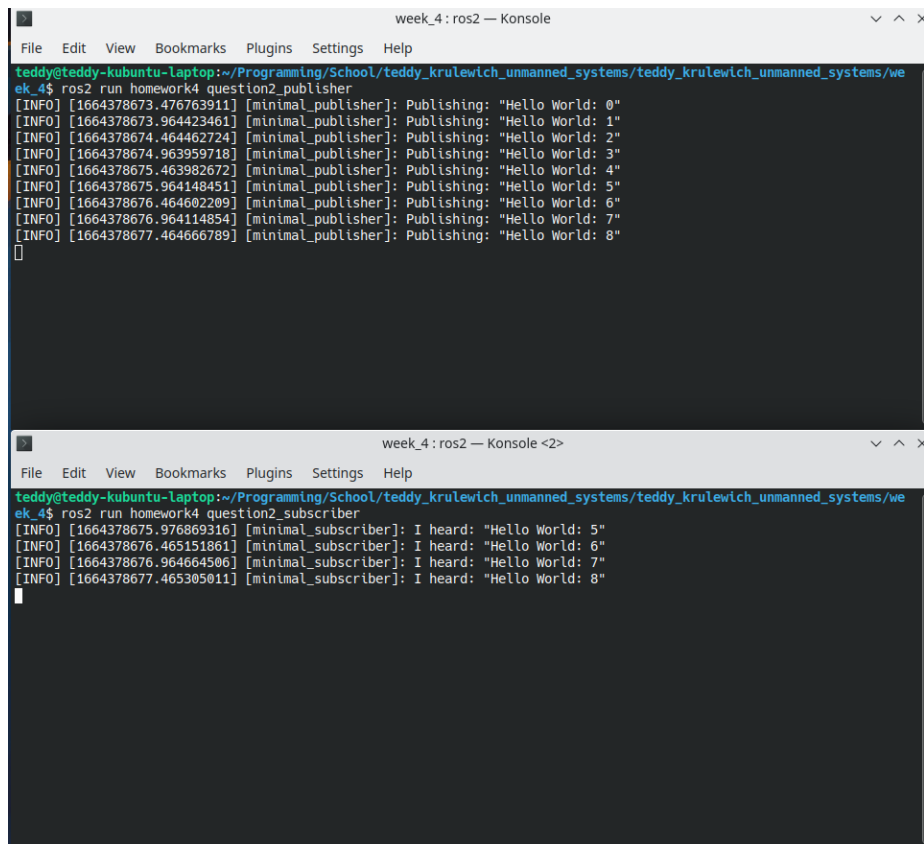
Problem 2

Complete the following ROS tutorials at

<http://wiki.ros.org/ROS/Tutorials>

- 12. Writing a Simple Publisher and Subscriber (Python)
- 13. Examining the Simple Publisher and Subscriber

Save a screenshot of the running code in tutorials 13, and print these screenshots to turn in.



The image shows two terminal windows from the ROS2 environment. The top window, titled 'week_4: ros2 — Konsole', shows the execution of 'ros2 run homework4 question2_publisher'. It outputs nine lines of log messages, each indicating a publish event from 'minimal_publisher' with the message 'Hello World: 0' through 'Hello World: 8'. The bottom window, titled 'week_4: ros2 — Konsole <2>', shows the execution of 'ros2 run homework4 question2_subscriber'. It outputs four lines of log messages, each indicating a received message from 'minimal_subscriber' with the message 'Hello World: 5' through 'Hello World: 8'. Both windows show the user's prompt as 'teddy@teddy-kubuntu-laptop:~/Programming/School/teddy_kruewlch_unmanned_systems/teddy_kruewlch_unmanned_systems/we'.

```
teddy@teddy-kubuntu-laptop:~/Programming/School/teddy_kruewlch_unmanned_systems/teddy_kruewlch_unmanned_systems/we
ek_4$ ros2 run homework4 question2_publisher
[INFO] [1664378673.476763911] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [1664378673.964423461] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [1664378674.464462724] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [1664378674.963959718] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [1664378675.463982672] [minimal_publisher]: Publishing: "Hello World: 4"
[INFO] [1664378675.964148451] [minimal_publisher]: Publishing: "Hello World: 5"
[INFO] [1664378676.464602209] [minimal_publisher]: Publishing: "Hello World: 6"
[INFO] [1664378676.964114854] [minimal_publisher]: Publishing: "Hello World: 7"
[INFO] [1664378677.464666789] [minimal_publisher]: Publishing: "Hello World: 8"
^

teddy@teddy-kubuntu-laptop:~/Programming/School/teddy_kruewlch_unmanned_systems/teddy_kruewlch_unmanned_systems/we
ek_4$ ros2 run homework4 question2_subscriber
[INFO] [1664378675.976869316] [minimal_subscriber]: I heard: "Hello World: 5"
[INFO] [1664378676.465151861] [minimal_subscriber]: I heard: "Hello World: 6"
[INFO] [1664378676.964664506] [minimal_subscriber]: I heard: "Hello World: 7"
[INFO] [1664378677.465305011] [minimal_subscriber]: I heard: "Hello World: 8"
^
```

Screenshot of subscriber and publisher running

Problem 3

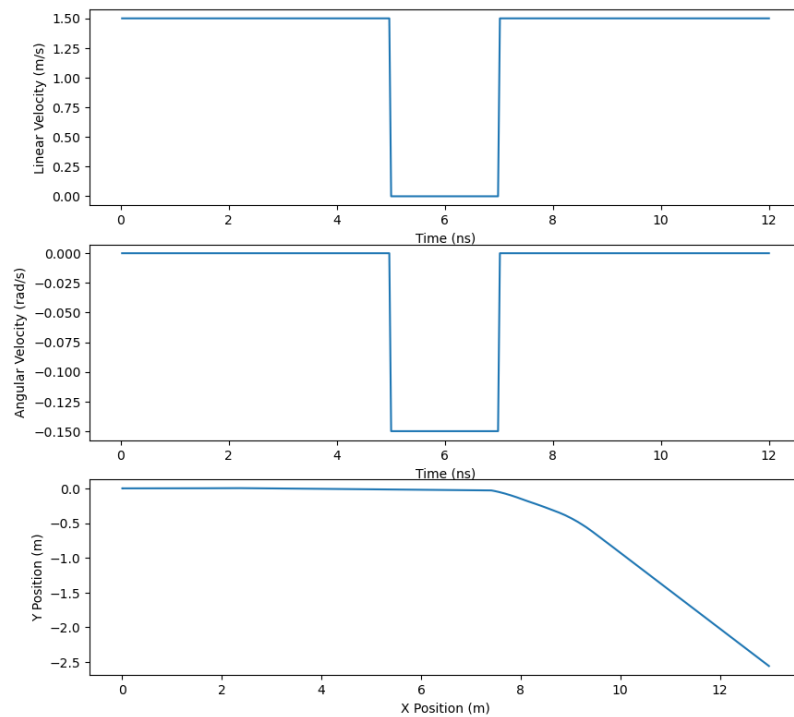
For this problem you will be using ROS2 and Gazebo to simulate the Turtlebot3 Burger platform. Help in loading the simulation can be found at <http://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/>.

Make sure to go to the Gazebo part of the E-manual. When setting the model type, replace “`{TB3_MODEL}`” with “burger.”

Create a ROS2 node that makes the Turtlebot3 travel forward at 1.5 m/s (remember this speed is much faster than it can do in real life) for 5 seconds, then turns to the right at 0.15 rad/s for 2 seconds, then continues forward at 1.5 m/s for another 5 seconds before stopping. Log the position data, and velocity & angular velocity commands.

Create a subplot of the x & y position data, velocity & angular velocity commands versus time.

Submit your Python code.



Plot of position data, velocity, and angular velocity commands versus time

```

1  import rclpy
2  from rclpy.node import Node
3
4  from geometry_msgs.msg import Twist
5  from nav_msgs.msg import Odometry
6
7  import matplotlib.pyplot as plt
8  import numpy as np
9  import math
10
11 def euler_from_quaternion(x:float, y:float, z:float, w:float) -> tuple:
12     """

```

```

13     Convert a quaternion into euler angles (roll, pitch, yaw)
14     roll is rotation around x in radians (counterclockwise)
15     pitch is rotation around y in radians (counterclockwise)
16     yaw is rotation around z in radians (counterclockwise)
17     """
18     t0 = +2.0 * (w * x + y * z)
19     t1 = +1.0 - 2.0 * (x * x + y * y)
20     roll_x = math.atan2(t0, t1)
21
22     t2 = +2.0 * (w * y - z * x)
23     t2 = +1.0 if t2 > +1.0 else t2
24     t2 = -1.0 if t2 < -1.0 else t2
25     pitch_y = math.asin(t2)
26
27     t3 = +2.0 * (w * z + x * y)
28     t4 = +1.0 - 2.0 * (y * y + z * z)
29     yaw_z = math.atan2(t3, t4)
30
31     return roll_x, pitch_y, yaw_z # in radians
32
33
34 class TurtleBotController(Node):
35     # A commanded forward and rotation velocity for the turtlebot with
36     # a specified duration
37     class VelocityCommand:
38     def __init__(self, linear, angular, duration):
39         self.twist = Twist()
40         self.twist.linear.x = linear
41         self.twist.angular.z = angular
42         self.duration = duration
43
44     def __init__(self):
45         super().__init__('turtlebot_controller')
46
47     # create a publisher to send velocity commands to the turtlebot
48     self.cmd_vel_publisher = self.create_publisher(Twist, 'cmd_vel', 10)
49     # create a subscriber to read sensor data
50     self.odom_subscriber = self.create_subscription(Odometry, 'odom',
51     self.odom_callback, 10)
52
53     # will store a list of commands to execute
54     self.velocity_commands = []
55
56     # store the time the node was started
57     self.start_time = self.get_clock().now().nanoseconds
58     self.time_command_started = self.start_time
59
60     # when all commands are executed will be true
61     self.done = False
62
63     # tracks the state of commands, position, and angle over time
64     self.state_records = { 'cmd_vel_linear': [], 'cmd_vel_angular': [],
65     'x': [], 'y': [], 'theta': [] }
66
67
68
69     def add_move_command(self, linear, angular, duration):

```

```

70     self.velocity_commands.append(
71         TurtleBotController.VelocityCommand(linear, angular, duration))
72
73     def odom_callback(self, msg):
74         time = self.get_clock().now().nanoseconds
75         # the total time elapsed since the node started
76         time_elapsed_total = time - self.start_time
77
78         # the time elapsed while executing current command
79         time_elapsed_command = time - self.time_command_started
80
81         # store current sensor data for logging and plotting
82         self.state_records['x'].append(
83             (time_elapsed_total, msg.pose.pose.position.x))
84
85         self.state_records['y'].append(
86             (time_elapsed_total, msg.pose.pose.position.y))
87
88         theta = euler_from_quaternion(
89             msg.pose.pose.orientation.x,
90             msg.pose.pose.orientation.y,
91             msg.pose.pose.orientation.z,
92             msg.pose.pose.orientation.w)[2]
93
94         self.state_records['theta'].append((time_elapsed_total, theta))
95
96         # if we have not finished executing commands
97         if len(self.velocity_commands) > 0:
98             # if the current command has executed for the specified duration
99             # then move on to the next command
100            if time_elapsed_command >= self.velocity_commands[0].duration:
101                self.velocity_commands.pop(0)
102                self.time_command_started = self.get_clock().now().nanoseconds
103
104            # if there are no remaining commands we are done!
105            if len(self.velocity_commands) == 0:
106                # publishes an empty Twist velocity causing the
107                # turtlebot to stop moving
108                self.cmd_vel_publisher.publish(Twist())
109                self.get_logger().info('No more commands')
110
111            self.done = True
112            return
113
114            # store the current command velocities for logging and plotting
115            self.state_records['cmd_vel_linear'].append(
116                (time_elapsed_total, self.velocity_commands[0].twist.linear.x))
117
118            self.state_records['cmd_vel_angular'].append(
119                (time_elapsed_total, self.velocity_commands[0].twist.angular.z))
120
121            # publish the velocity command to the turtle bot
122            self.cmd_vel_publisher.publish(self.velocity_commands[0].twist)
123
124 def main(args=None):
125     rclpy.init(args=args)
126

```

```

127 # create a turtlebot
128 turtlebot_controller = TurtleBotController()
129
130 # command the turtlebot to move forward at 1.5 m/s for 5 seconds
131 turtlebot_controller.add_move_command(1.5, 0.0, 5000000000)
132 # then comand the turtlebot to turn at 0.15 rad/s for 2 seconds
133 turtlebot_controller.add_move_command(0.0, -0.15, 2000000000)
134 # then command the turtle bot to move forward at 1.5 m/s for 5 seconds
135 turtlebot_controller.add_move_command(1.5, 0.0, 5000000000)
136
137 # while the turtle bot hasn't finsihed executing its commands, update the node
138 while not turtlebot_controller.done:
139     rclpy.spin_once(turtlebot_controller)
140
141
142 fig, ax = plt.subplots(3, 1)
143
144 # plot the linear velocity command vs time
145 ax[0].plot([x[0] / 1000000000 for x in
146 turtlebot_controller.state_records['cmd_vel_linear']],
147 [x[1] for x in turtlebot_controller.state_records['cmd_vel_linear']],
148 label='linear')
149
150 ax[0].set_xlabel('Time (ns)')
151 ax[0].set_ylabel('Linear Velocity (m/s)')
152
153 # plot the angular velocity command vs time
154 ax[1].plot([x[0] / 1000000000 for x in
155 turtlebot_controller.state_records['cmd_vel_angular']],
156 [x[1] for x in turtlebot_controller.state_records['cmd_vel_angular']],
157 label='angular')
158
159 ax[1].set_xlabel('Time (ns)')
160 ax[1].set_ylabel('Angular Velocity (rad/s)')
161
162 # plot the x and y position of the turtlebot, showing the path
163 ax[2].plot([x[1] for x in turtlebot_controller.state_records['x']],
164 [y[1] for y in turtlebot_controller.state_records['y']], label='y')
165
166 ax[2].set_xlabel('X Position (m)')
167 ax[2].set_ylabel('Y Position (m)')
168
169
170 plt.show()
171
172
173 turtlebot_controller.destroy_node()
174 rclpy.shutdown()
175
176
177 if __name__ == '__main__':
178     main()
179

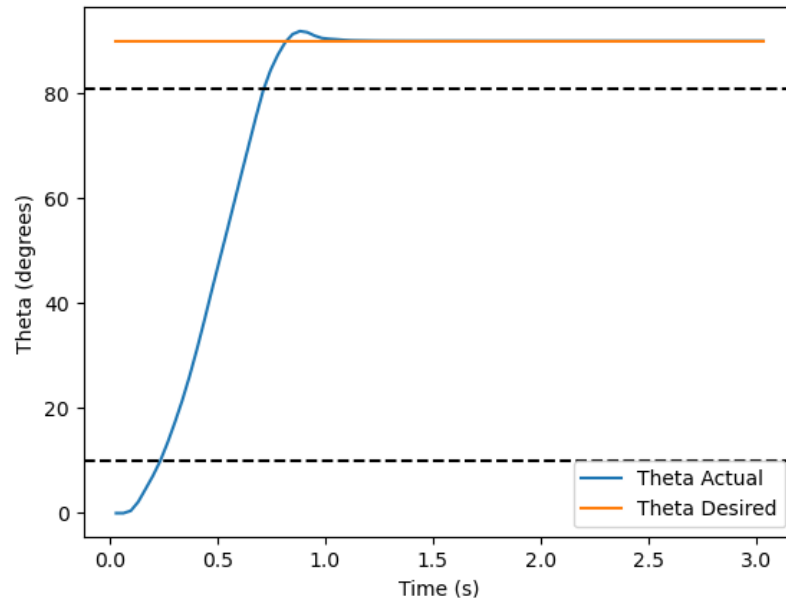
```

Problem 4

Create a ROS2 node/script that uses a feedback controller to control the heading of the Turtlebot3. Once you have adequately tuned the controller, collect the data (by writing to a log file) from a 90 degree step input (use a forward speed of 0.15 m/s).

Create a plot of the desired and actual heading versus time. What is the rise time, settling time, and percent overshoot of your controller?

Submit your Python code.



Plot of desired and actual heading versus time

The rise time was $0.70 - 0.2191 = \mathbf{0.4809}$ seconds.

The settling time was $1.3 - 0.8042 = \mathbf{0.4958}$ seconds.

The percent overshoot was $100 * (91.855 / 90.0 - 1.0) = \mathbf{2.06\%}$.

I updated the previous code in Problem 2 by adding a PID controller

```
1 class PID:
2     """
3     Simple PID controller for a single variable
4     """
```

```

5     def __init__(self, kp, ki, kd):
6         self.kp = kp
7         self.ki = ki
8         self.kd = kd
9
10        self.last_error = 0
11        self.integral = 0
12
13        self.output = 0
14
15        def update(self, error, dt):
16            """
17            Update the PID controller using new error value
18            """
19            self.integral += error * dt
20            derivative = (error - self.last_error) / dt
21
22            self.output = self.kp * error + self.ki * self.integral + self.kd * derivative
23
24            self.last_error = error

```

I modified the constructor adding the following

```

1     class TurtleBotController(Node):
2         def __init__(self):
3             super().__init__('turtlebot_controller')
4             ...
5             ...
6             # used to calculate dt between updates
7             self.last_update = self.start_time
8
9             # desired heading
10            self.desired_theta = None
11            self.current_theta = None
12
13            self.theta_controller = PID(1, 0.5, 0.0)

```

I modified the callback function as follows

```

1     def odom_callback(self, msg):
2         # if we are done dont execute anything else
3         if self.done:
4             return
5
6         # if we have not set a desired heading
7         if self.desired_theta is None:
8             self.done = True
9
10        # get current time and time elapsed since the node was started
11        time = self.get_clock().now().nanoseconds
12        time_elapsed = time - self.start_time
13
14        # get the dt in seconds since the last update
15        dt = (time - self.last_update)
16
17

```



```

18     # read sensor data to get position and heading
19     self.current_x = msg.pose.pose.position.x
20     self.current_y = msg.pose.pose.position.y
21     self.current_theta = euler_from_quaternion(
22         msg.pose.pose.orientation.x,
23         msg.pose.pose.orientation.y,
24         msg.pose.pose.orientation.z,
25         msg.pose.pose.orientation.w)[2]
26
27     # store sensor data for logging and plotting
28     self.state_records['x'].append((time_elapsed, self.current_x))
29     self.state_records['y'].append((time_elapsed, self.current_y))
30     self.state_records['theta'].append((time_elapsed, self.current_theta))
31
32     # create a Twist for the velocity command
33     twist = Twist()
34     # move forward at 0.15 m/s
35     twist.linear.x = 0.15
36
37
38     # if more than 3 seconds have elapsed, stop the turtlebot
39     if time_elapsed > 3000000000:
40         twist.angular.z = 0.0
41         self.cmd_vel_publisher.publish(twist)
42         self.done = True
43         return
44
45     # run a PID controller using the error in heading
46     self.theta_controller.update(self.desired_theta - self.current_theta, dt)
47
48     # set the angular velocity to the output of the PID controller
49     twist.angular.z = self.theta_controller.output
50
51     twist.angular.z = clamp(twist.angular.z, -2.84, 2.84)
52
53     # publish the velocity command to the turtlebot
54     self.cmd_vel_publisher.publish(twist)
55
56     # store the velocity command for logging and plotting
57     self.state_records['cmd_vel_linear'].append((time, twist.linear.x))
58     self.state_records['cmd_vel_angular'].append((time, twist.angular.z))
59
60     self.last_update = time

```

I modified the main function as follows

```

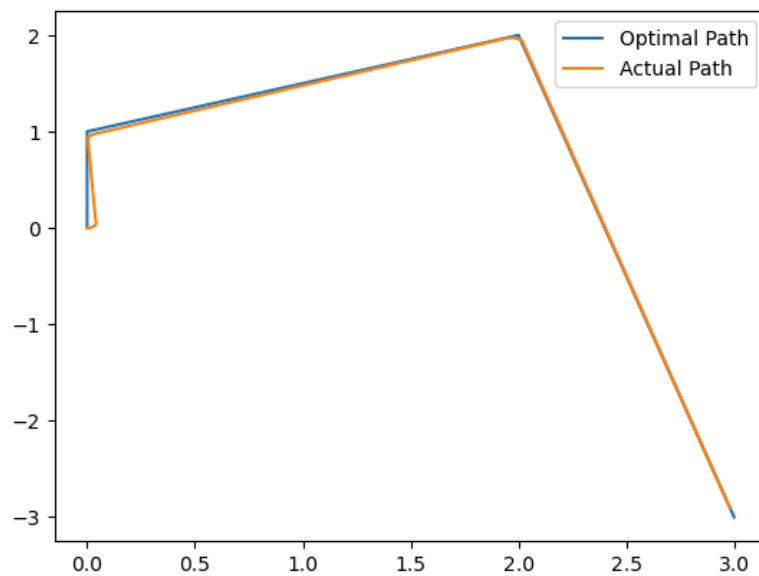
1  def main(args=None):
2      rclpy.init(args=args)
3
4      turtlebot_controller = TurtleBotController()
5
6      turtlebot_controller.desired_theta = math.pi / 2
7
8      while not turtlebot_controller.done:
9          rclpy.spin_once(turtlebot_controller)
10         ...
11         ...

```

Problem 5

Create a ROS2 node that follows the given set of prescribed waypoints: $[0,0]$, $[0,1]$, $[2,2]$, $[3,-3]$. Start your robot at $[0,0]$. Create a plot of the X, Y coordinates to show how well your robot follows the desired path. Use a maximum translational speed of 0.15 m/s.

Submit your Python code.



Plot of desired and actual X and Y coordinates versus time

I created a `Waypoint` class and modified the constructor for the node as follows

```
1 class TurtleBotController(Node):
2     class Waypoint:
3         """
4         A waypoint that the turtlebot will try to reach
5         """
6     def __init__(self, x, y):
7         self.x = x
8         self.y = y
9
10    def __init__(self):
11        super().__init__('turtlebot_controller')
12
```

```

13     # create a publisher to send velocity commands to the turtlebot
14     self.cmd_vel_publisher = self.create_publisher(Twist, 'cmd_vel', 10)
15     # create a subscriber to read sensor data
16     self.odom_subscriber = self.create_subscription(Odometry, 'odom',
17     self.odom_callback, 10)
18
19
20     # store the time the node was started
21     self.start_time = self.get_clock().now().nanoseconds
22     self.last_update = self.start_time
23
24     # the turtle bot will continue updating until this is set to true
25     self.done = False
26
27     # store the state for logging and plotting
28     self.state_records = { 'cmd_vel_linear': [], 'cmd_vel_angular': [],
29     'x': [], 'y': [], 'theta': [], 'theta_des': [] }
30
31     # current position and location of the turtlebot
32     self.current_theta = None
33     self.current_x = None
34     self.current_y = None
35
36     # create a PID controller for angular velocity
37     self.theta_controller = PID(7, 0.0, 1.0)
38
39     # the list of waypoints in order that the turtlebot will try to reach
40     self.waypoints = []
41

```

I modified the callback function as follows

```

1  def odom_callback(self, msg):
2      # get current time, time elapsed, and delta time
3      time = self.get_clock().now().nanoseconds
4      time_elapsed = time - self.start_time
5      dt = time - self.last_update
6
7
8      self.last_update = time
9
10     # if there are no more waypoints, stop the turtlebot
11     if len(self.waypoints) == 0:
12         self.done = True
13         twist = Twist()
14         twist.linear.x = 0.0
15         twist.angular.z = 0.0
16         self.cmd_vel_publisher.publish(twist)
17         return
18
19     # read sensor data to get position and heading
20     self.current_x = msg.pose.pose.position.x
21     self.current_y = msg.pose.pose.position.y
22     self.current_theta = euler_from_quaternion(
23     msg.pose.pose.orientation.x,
24     msg.pose.pose.orientation.y,
25     msg.pose.pose.orientation.z,

```

```

26 msg.pose.pose.orientation.w)[2]
27
28 # get the current waypoing (top of list)
29 waypoint = self.waypoints[0]
30 # find its distance from the turtlebot
31 distance = math.sqrt(
32 (waypoint.x - self.current_x)**2 + (waypoint.y - self.current_y)**2)
33
34 # if the turtlebot is close enough to the waypoint, move on
35 if distance < 0.1:
36     self.waypoints.pop(0)
37     return
38
39
40 # find the angle between current position and waypoint
41 self.desired_theta = math.atan2(
42     waypoint.y - self.current_y,
43     waypoint.x - self.current_x)
44
45 # adjust desired angle so that it turns the shortest way
46 if self.desired_theta - self.current_theta > math.pi:
47     self.desired_theta = self.desired_theta - 2 * math.pi
48 elif self.desired_theta - self.current_theta < -math.pi:
49     self.desired_theta = self.desired_theta + 2 * math.pi
50
51 # store the sensor data and desired heading for logging and plotting
52 self.state_records['x'].append((time_elapsed, self.current_x))
53 self.state_records['y'].append((time_elapsed, self.current_y))
54 self.state_records['theta'].append((time_elapsed, self.current_theta))
55 self.state_records['theta_des'].append((time_elapsed, self.desired_theta))
56
57 # set the translational velocity to 0.15 m/s
58 twist = Twist()
59 twist.linear.x = 0.15
60
61 # use PID controller to set the angular velocity
62 self.theta_controller.update(self.desired_theta - self.current_theta, dt)
63 twist.angular.z = self.theta_controller.output
64
65 # cap the angular velocity at 2.84 rad/s
66 twist.angular.z = clamp(twist.angular.z, -2.84, 2.84)
67
68 # publish the velocity command to the turtlebot
69 self.cmd_vel_publisher.publish(twist)
70
71 # store the velocity command for logging and plotting
72 self.state_records['cmd_vel_linear'].append((time_elapsed, twist.linear.x))
73 self.state_records['cmd_vel_angular'].append((time_elapsed, twist.angular.z))
74

```

I modified the main function as follows

```

1 def main(args=None):
2     rclpy.init(args=args)
3
4     turtlebot_controller = TurtleBotController()
5

```

```

6 turtlebot_controller.done = False
7 turtlebot_controller.add_waypoint(0, 0)
8 turtlebot_controller.add_waypoint(0, 1)
9 turtlebot_controller.add_waypoint(2, 2)
10 turtlebot_controller.add_waypoint(3, -3)
11
12 while not turtlebot_controller.done:
13     rclpy.spin_once(turtlebot_controller)
14     ...
15     ...

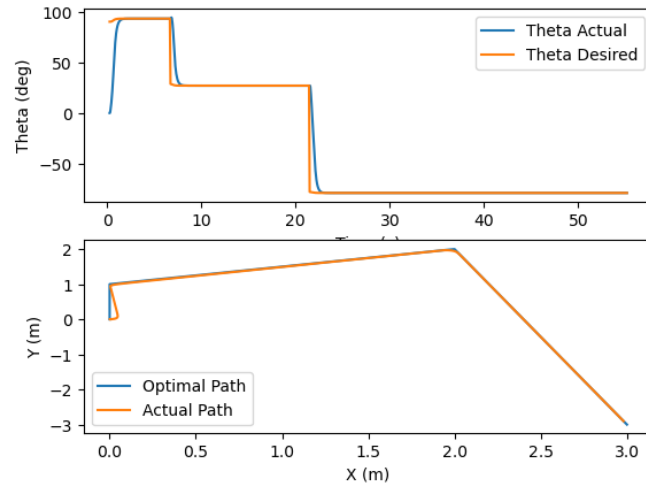
```

Problem 6

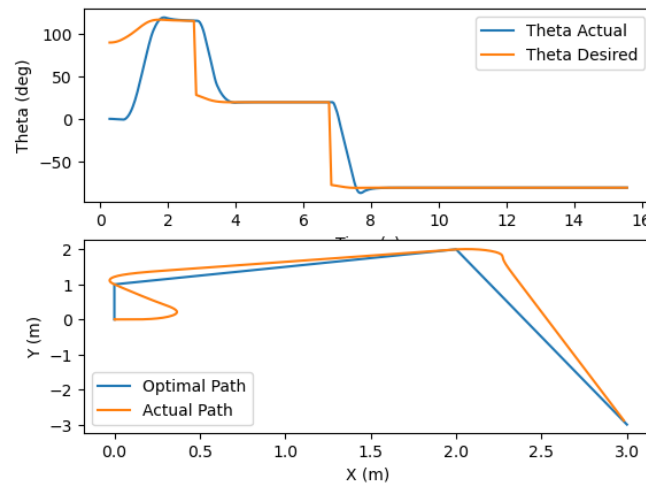
Redo Problem 5 with speeds ranging from 0.15 – 1.5 m/s (use at least 4 speeds) and provide the results regarding the ability of the turtlebot to follow the desired path/waypoints. A single plot showing the deviation from the desired path for each speed is the minimum acceptable information. Discuss the results, what is the max speed we should use for the simulation environment of the turtlebot (if we want to go as fast as possible)?

I found a speed of 1.05 m/s to be the fastest in reaching the waypoints, however, the path taken was far from optimal. In an empty map that path might work, but if there were obstacles, it likely would have collided with something.

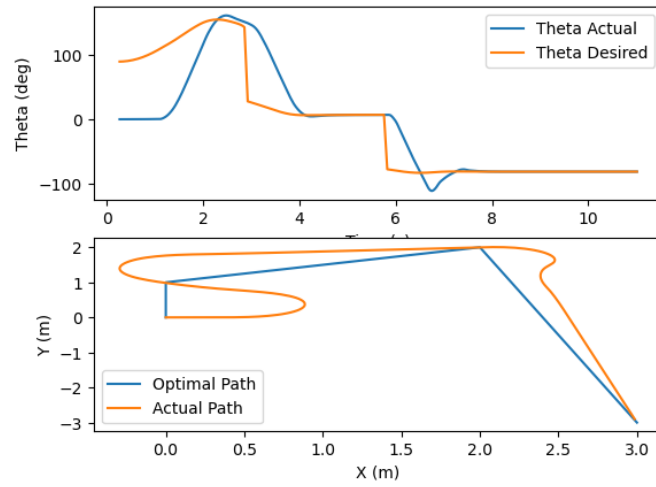
A speed of 0.6 m/s was reasonably close to the optimal path and reasonably quick.



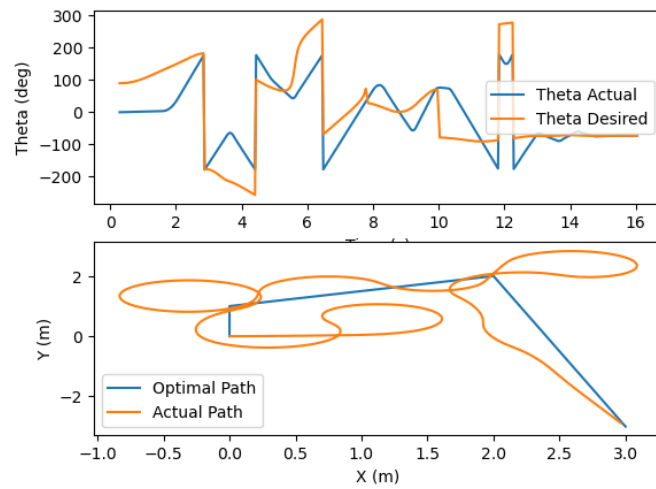
Speed = 0.15 m/s, Time to complete = 55.23 seconds



Speed = 0.6 m/s, Time to complete = 15.63 seconds



Speed = 1.05 m/s, Time to complete = 11.08 seconds



Speed = 1.5 m/s, Time to complete = 16.61 seconds