Lab 3

Start Assignment

- Due Feb 4 by 11:59pm
- Points 100
- Submitting a file upload
- Available after Jan 28 at 8pm

Overview

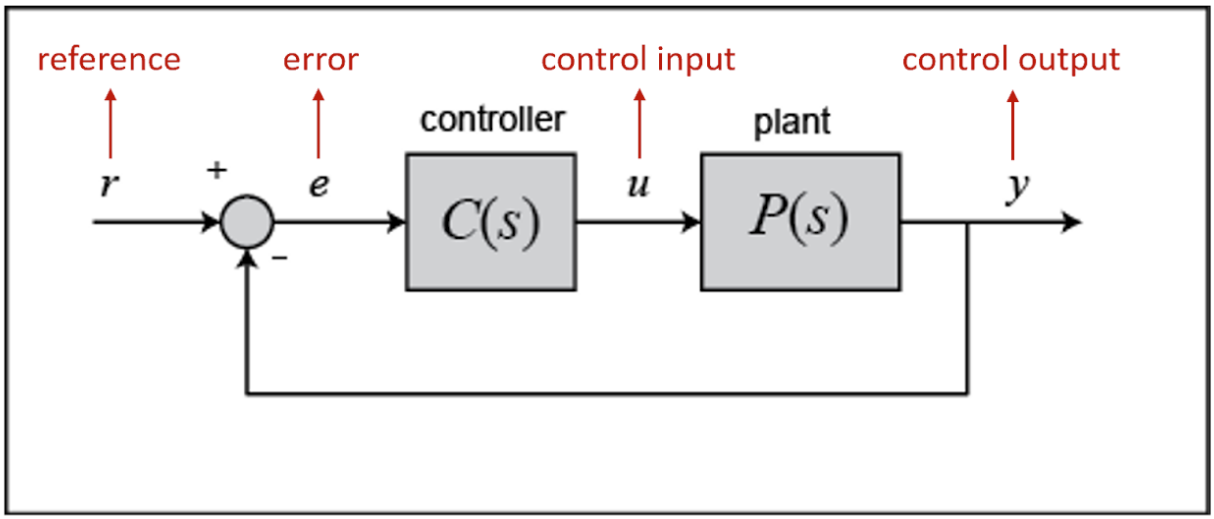
In this lab, we are going to apply closed-loop control to trajectory tracking. A ROS Subscriber is included in the script to receive real-time odometry feedback from Gazebo simulator. With the odom/pose updates, the robot can adjust its moving direction accordingly and check if the desired waypoint has been reached.

Specifically, the task is to implement a PD controller (a variant/subset of PID controller) to track the diamond shape trajectory. Same as in Lab 2, the waypoints are (0, 0), (3, -4), (6, 0), and (3, 4), and the sequence does matter. After task completion, the robot should stop at the origin and the Python script should exit gracefully. Please plot the trajectory (using another provided Python script) and discuss your results in the lab report.

To help you complete the lab, please review the **Classes section** (https://docs.python.org/3/tutorial/classes.html) in Python Docs

PID Control

Consider the following feedback control diagram.



- The plant is the system we would like to control. In our case, the control input u is the velocity command we send to the robot, the control output y is the current state (2D pose: x, y, theta) of the robot, and the reference r is the desired state we would like the robot to reach.
- The controller is what we need to design. It takes in the tracking error e, which is the difference between the desired reference r and the actual output y, and **computes the control input** u **according to the following equation**. (The control input to the plant is the output of the controller).

$$u(t) = k_p e(t) + k_i \int e(t)dt + k_d \frac{de(t)}{dt}$$

- This equation reveals the name of the controller: Proportional–Integral–Derivative (PID) controller, because it has three terms: proportional term, integral term and derivative term. In each term, we have a coefficient k multiplies the (integral/derivative of the) error. The `Kp`, `Ki`, and `Kd` are the coefficients/parameters we need to tune.
- In math, it has rigorous analysis to show the stability and convergence of the system, which can be used to calculate the optimal parameters `Kp`, `Ki` and `Kd`. (This should be covered in EE132 Automatic Control course. See this **PID Controller Design (http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID)** tutorial for more information.) However, in this lab manually tuned non-optimal parameters are sufficient to complete the task. You may start with some value close to 1 for `Kp` and a smaller value for `Kd`.
- As for the integral and the derivative of the error, in discrete systems, we can replace integral with summation and replace derivative with subtraction.

$$u(t) = k_p e(t) + k_i \sum e(t) + k_d \frac{e(t) - e(t - \Delta t)}{\Delta t}$$

- Due to the negative effect from integral component, we will drop this term and only focus on PD controller. Also, the time interval Δt can be merged into parameter `Kd`. (If we run at 10Hz, the time interval will be 0.1 second.) Therefore, we have the following equation ready, **which is what you need to implement in this lab**.

$$u(t) = k_p e(t) + k_d (e(t) - e(t - \Delta t))$$

- A discrete PD controller implementation in Python is provided for your information. (You may or may not use it in your own implementation). To make it work, you need to understand the PD control algorithm and complete the three lines of code under the update function.

```
class Controller:
    """
    Controller class for implementing the PD Controller
    """

    def __init__(self, P=0.0, D=0.0, set_point=0):
        self.Kp = P
```

```
        self.Kd = D
        self.set_point = set_point # reference (desired value)
        self.previous_error = 0

    def update(self, current_value):

        # TODO: Calculate error, P_term and D_term
        error = 0.0
        P_term = 0.0
        D_term = 0.0
        self.previous_error = error
        return P_term + D_term

    def setPoint(self, set_point):
        self.set_point = set_point
        self.previous_error = 0

    def setPD(self, P=0.0, D=0.0):
        self.Kp = P
        self.Kd = D
```

Launch Files

- So far, we've been using the default launch packages provided by the authors of TurtleBot4. However, the to also spawn the dock along with the robot. While it is convenient and the default standard, in this lab, we're going to remove the dock so that the TurtleBot4 and the world frame completely align from the very beginning.
- Launch files help execute packages and nodes automatically. Here are some **tutorials (https://docs.ros.org/en/jazzy/Tutorials/Intermediate/Launch/Creating-Launch-Files.html)** for creating launch files for your own projects. In the interest of time, we're going to provide you with the launch files for this lab so that you can begin writing the closed-loop control. First let's create a directory where the launch files can be placed. We'll follow the standard convention here.

```
mkdir -p ~/ros2_ws/src/eecs256aw25/launch
```

- Please go ahead and download the files `lab3_gz.launch.py` **(https://elearn.ucr.edu/courses/215657/files/23670535?wrap=1)** **(https://elearn.ucr.edu/courses/215657/files/23670535/download?download_frd=1)** and `lab3_turtlebot4_spawn.py` **(https://elearn.ucr.edu/courses/215657/files/22990627?wrap=1)** **(https://elearn.ucr.edu/courses/215657/files/22990627/download?download_frd=1)** and place them in the folder we created above. Typically, the files are downloaded to the `Downloads` directory. However, if you've configured the files to be downloaded somewhere else, please place them in the launch folder.

```
mv ~/Downloads/lab3_gz.launch.py ~/ros2_ws/src/eecs256aw25/launch/
mv ~/Downloads/lab3_turtlebot4_spawn.launch.py ~/ros2_ws/src/eecs256aw25/launch/
```

- Now that the launch files have been placed, we need to let ROS know where we've placed them. In order to do so, we add the following lines to `setup.py` :

```
import os
from glob import glob
```

```
# Other imports ...

package_name = 'eecs256aw25'

setup(
    # Other parameters ...
    data_files=[
        # ... Other data files
        # Include all launch files.
        (os.path.join('share', package_name, 'launch'), glob('launch/*'))
    ]
)
```

- Finally, we build the package. If everything is successful, we should be good to go!

```
cd ~/ros2_ws/
colcon build
```

Programming Tips

- Note that the orientation of the robot (theta) ranges from -pi to pi on 2D plane. When the robot turns in CCW direction and passes the direction of negative x axis, the value of theta will jump from pi to -pi. This needs to be handled properly, otherwise the robot will keep turning in place and cannot move forward. It is recommended that you bring up a robot, turn in place using keyboard teleoperation, and see how the value changes for theta. (You can see it from ROS logging messages if using the provided closed_loop.py file).
- In general, PID controller is used to track a certain target value (called **setpoint**), and make sure the system can converge to this target value. For example, to control the temperature in a boiler system. (Note that the setpoint is a scalar, set to a certain target value.)
- In our case, we have three variables (x, y, theta) to describe the 2D pose of the robot. At one time only one of them can be set as the desired value to track in a PID controller. If you are willing to track x, y, and theta at the same time, you will need three PID controllers. (To comlpete the task in this lab, out of three controllers, **the one to track theta is required**, and the other two for x and y are optional. See below an example algorithm.)
- The following is an example of how to apply feedback control algorithm to waypoint navigation problem. You may follow this algorithm to start your implementation.

1. Suppose the robot's current orientation is θ, the desired orientation is θ^*, the current position on X-Y plane is (x,y) and the desired position on X-Y plane is (x^*, y^*).
2. Calculate the moving direction from the difference between (x,y) and (x^*, y^*); set it as the desired orientation θ^*.
3. Initialize a PID controller with the setpoint θ^* and a set of parameters (`Kp`, `Ki` and `Kd`). Adjust the angle according to the angular velocity computed by the PID controller.
4. Once $\theta \to \theta^*$, start moving forward at a constant speed (or using a linear velocity generated by another PID controller), and keep adjusting the angle and checking the remaining distance toward desired position (x^*, y^*).

5. Once $(x,y) \rightarrow (x*, y*)$, stop and repeat the process for the next waypoint.

Additional Packages

- The `tf_transformations` package is required for conversions between Quaternions and Euler Angles. Install the `tf_transformations` package using the following command:

```
sudo apt-get update
sudo apt-get install ros-jazzy-tf-transformations
```

- We may also need to install `numpy` and `matplotlib` for saving and visualizing the trajectories. We can do this using the following:

```
sudo apt-get install python3-numpy python3-matplotlib
```

Sample Code

A sample code is provided as the starting point for your implementation. Please read carefully the provided code, and understand its functionality.

- Open a new terminal and go to the eecs256aw25 package. We will start from a new python script.

```
import rclpy
import threading
import time

import numpy as np

from rclpy.action import ActionClient
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from geometry_msgs.msg import TwistStamped, Pose2D
from nav_msgs.msg import Odometry

from tf_transformations import euler_from_quaternion

class Turtlebot(Node):

    def __init__(self):

        # Create and initialize the Node - this is required
        super().__init__('turtlebot_move')

        self.get_logger().info("Press Ctrl + C to terminate")

        # Current pose of the robot
        self.pose = Pose2D()

        # Logging counter
        self.logging_counter = 0

        # Store the robot trajectory
        self.trajectory = []

        # Create publisher for publishing the velocity commands
        self.vel_pub = self.create_publisher(TwistStamped, "cmd_vel", 10)
```

```
        # Create a subscriber for the ground truth
        self.odom_sub = self.create_subscription(Odometry,
            "odom",
            self.odom_callback,
            100)

    def odom_callback(self, msg):
        """
        Callback function to get the robot odometry.

        @param msg - Message passed by the odom topic.
        """

        # get pose = (x, y, theta) from odometry topic
        quarternion = [msg.pose.pose.orientation.x,msg.pose.pose.orientation.y,\
                    msg.pose.pose.orientation.z, msg.pose.pose.orientation.w]

        (roll, pitch, yaw) = euler_from_quaternion(quarternion)
        self.pose.theta = yaw
        self.pose.x = msg.pose.pose.position.x
        self.pose.y = msg.pose.pose.position.y

        # logging once every 100 times
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            self.trajectory.append([self.pose.x, self.pose.y])  # save trajectory

            # display (x, y, theta) on the terminal
            self.get_logger().info("odom: x=" + str(self.pose.x) +\
                ";  y=" + str(self.pose.y) + ";  theta=" + str(yaw))

    def run(self):
        """
        Run function.
        """

        # Create a Twist object
        # Essentially it's going to publish the velocity
        vel = TwistStamped()

        # Populate the fields of the message
        vel.twist.linear.x = 0.5
        vel.twist.angular.z = 0.0

        self.get_logger().info('Running closed-loop control...')

        # TODO: Add Closed-loop control code over here
        # HINT: Use the pose provided to the subscriber
        for _ in range(500):
            self.vel_pub.publish(vel)
            time.sleep(0.1)

    def save_trajectory(self):
        """
        Save trajectory to disk.
        """
        np.savetxt("trajectory.csv", np.array(self.trajectory), fmt='%f', delimiter=',')

def closed_loop_entry_point_function(args=None):
    """
    Entry point function for the Closed Loop control node.

    @param args - Arguments that need to be passed to ROS.
    """
```

```
    # Initialize the client library - this is required
    rclpy.init(args=args)

    # Create the publisher node
    turtlebot_node = Turtlebot()

    # Spin so that the node is busy processing
    # while no exceptions are encountered and ROS
    # is executing
    # For this lab - spin rclpy on separate thread
    thread = threading.Thread(target=rclpy.spin, args=(turtlebot_node,), daemon=True)
    thread.start()

    # Allow time for other nodes to start
    time.sleep(5)

    try:
        turtlebot_node.run()
    except KeyboardInterrupt:
        turtlebot_node.get_logger().info("Keyboard Interrupt.")
    finally:
        turtlebot_node.save_trajectory()

    # Destroy the node explicitly
    turtlebot_node.destroy_node()

    # Shut down this context
    rclpy.shutdown()


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

Sample Code Explained

- Odometry works in the way that it counts how many rounds the wheel rotates. Therefore, if you lift and place a robot from one place to another, it will still "think" that it is at the original place.
- As we mentioned in the **tutorial (https://elearn.ucr.edu/courses/215657/assignments/996346)** provided in Lab 1, we can use ROS Publisher to send a message out. ROS Subscriber is the one on the other side to receive and process the messages. The required arguments are the topic name `odom`, the message type `Odometry`, a pointer to the callback function and an integer indicating the Quality of Service profile. The Quality of Service profile lets users tune the communication settings between different nodes. For more information on this, the **ROS wiki is available (https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Quality-of-Service-Settings.html)**. The callback function will be executed whenever a new message is received (asynchronously; in another thread). We leverage the shared variables in TurtleBot class to store the latest pose of the robot. Note that the msg argument in the callback function is of the type `Odometry`. The definition is specified in **the ROS Wiki documentation (https://docs.ros.org/en/jazzy/p/nav_msgs/_downloads/805cd3c972470c0b38c699bb4ef064cb/Odometry.msg)**.

```
self.odom_sub = self.create_subscription(Odometry,
            "odom",
            self.odom_callback,
```

```
            100)

def odom_callback(self, msg):
        pass
```

- In the try-except structure, finally is the keyword to indicate that the following code block will be executed regardless if the try block raises an error or not. In this case, we want to save the trajectory even when the robot stops halfway.

```
finally:
    # save trajectory into csv file
    turtlebot_node.save_trajectory()
```

Visualization

We provide a separate Python script to help visualize the trajectory from the saved `csv` file. Please do not submit this file and do not include it in the script you plan to submit. You can place this code in `~/ros2_ws/src/eecs256aw25/eecs256aw25/` under the file name `visualize.py`.

```
import numpy as np
import matplotlib.pyplot as plt

def visualization():

    # load csv file and plot trajectory
    _, ax = plt.subplots(1)
    ax.set_aspect('equal')

    trajectory = np.loadtxt("trajectory.csv", delimiter=',')
    plt.plot(trajectory[:, 0], trajectory[:, 1], linewidth=2)

    plt.xlim(-1, 7)
    plt.ylim(-5, 5)
    plt.grid(True)
    plt.show()

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

Build & Run!

- First, compile the package:

```
cd ~/ros2_ws/
colcon build
source install/setup.bash
```

- Open up the simulator in one window with the TurtleBot 4 Lite model:

```
ros2 launch eecs256aw25 lab3_gz.launch.py model:=lite world:=empty
```

- In another terminal window, navigate to your package and execute the python script like so:

```
cd ~/ros2_ws/src/eecs256aw25/eecs256aw25/
python3 closed_loop.py
```

- Visualize the saved trajectory. It should be saved as `trajectory.csv`:

```
python3 visualize.py
```

Deliverable 1

- As mentioned at the beginning of the lab handout, the goal is send velocity commands to the robot and make it go through the 4 waypoints, which should end up making a diamond pattern. At the end the robot should stop at the origin. The waypoints to visit are (0, 0), (3, -4), (6, 0), and (3, 4) - **in order**.

Submission

1. Submission: individual submission via Canvas.
2. Due time: 11:59pm, Feb 4, Wednesday.
3. Files to submit: (please use exactly the same filename; case sensitive).
    - lab3_report.pdf
    - closed_loop.py
4. Grading rubric:
        - 30% Clearly describe your approach and explain your code in the lab report.
        - 20% Plot the trajectory and discuss the results of different values of Kp and Kd.
        - 40% Implement PD controller; visit four vertices of the diamond trajectory with error < 0.1m. Partial credits will be given according to the number of vertices visited.
        - 10% The script can complete the task on time and exit gracefully.
    - 10% Penalty applies for each late day. No submission will be accepted beyond four (4) days from the due date.

Note: It is required to use closed-loop control (i.e. PD controller) to track the trajectory. Finely tuned open-loop control script may also pass the tests (by adding angular velocity as in Lab 2) but is not allowed. All scripts will be double checked when grading manually. Penalty will apply if such script is found.