THE UNIVERSITY OF HONG KONG

Department of Data and Systems Engineering

DASE7505 Intelligent Unmanned Systems

Lab: Closed loop control of mobile robots

Objectives:

- Explain the concept of feedback and the role of a controller in a closed-loop robotic system.
- Understand basic controllers (P, PD, PI, PID) in Python nodes under ROS2.
- Use /odom feedback to compute pose error and generate /cmd_vel commands.
- Understand the architecture of multi-node control systems (decision → localization → controller).
- Evaluate controller performance using plots of pose/velocity/error, if applicable.

Remark:

This lab will not require any operations on VMware. However, bonus points maybe given to students who try to implement PID on their ROS and demonstrated some interesting results.

Recap of PID:

Have a recap of some basic priors about PID

- DASE7505 lecture notes.
- A shorter video from Prof. Kevin Lynch: https://youtu.be/taSlxgvvrBM?si=fawRXMvDJiSl6Vp-

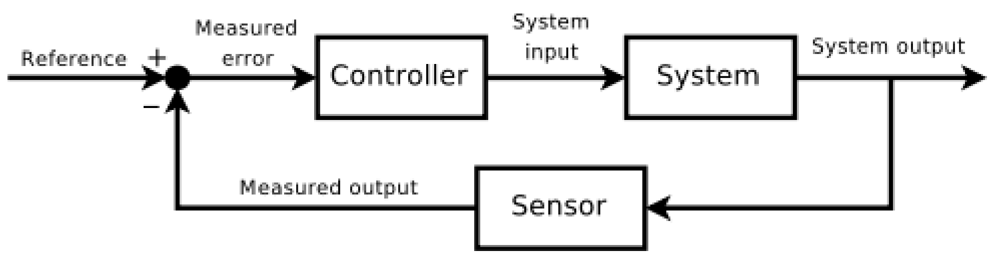
A three-term proportional-integral-derivative (PID) controller has the transfer function:

$$G_c(s) = k_P + \frac{k_I}{s} + sk_D$$

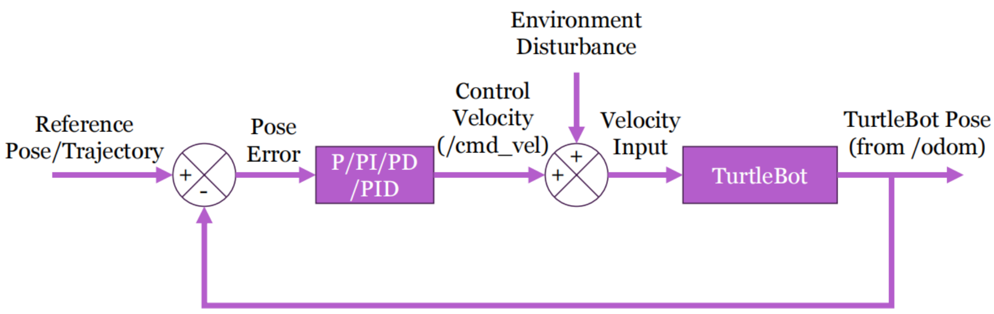
- *Proportional* term k_P to close the loop
- *Integral* term k_I to assure zero error to constant reference and disturbance inputs
- ***Derivative*** term k_D to improve stability and good dynamic response

Step-by-Step Instructions:

A typical feedback control could be illustrated as below:



For our TurtleBot lab, we also use P/PI/PID controller to perform localization and trajectories tracking.


Environment Disturbance
Reference Pose/Trajectory
Pose Error
P/PI/PD /PID
Control Velocity (/cmd_vel)
Velocity Input
TurtleBot
TurtleBot Pose (from /odom)
+
-
+
+

In this lab, you will learn how to implement a closed-loop controller to drive the robot by different coding: the controller, the planner, and the localization. You are encouraged to draw an architecture map to show how they are interconnected with each other after going through these programs. (Any plotting app or website is fine)

- The controller is implemented using different control laws. You will start with a proportional controller (P) and then extend to proportional, derivative, integral (PID).
- The planner generates the desired path/destination for the robot to follow, this can be as simple as a point planner, or a trajectory. At this stage, you will only implement predefined points/trajectories assuming the environment is perfect. You will learn in optional lab about how to implement optimal paths considering the environment (e.g. obstacles, walls).
- The localization tells you where the robot is so that you can move the robot along the desired paths and read the data to feed back to your controller. At this stage, you will simply use the odometry to determine the position of the robot.

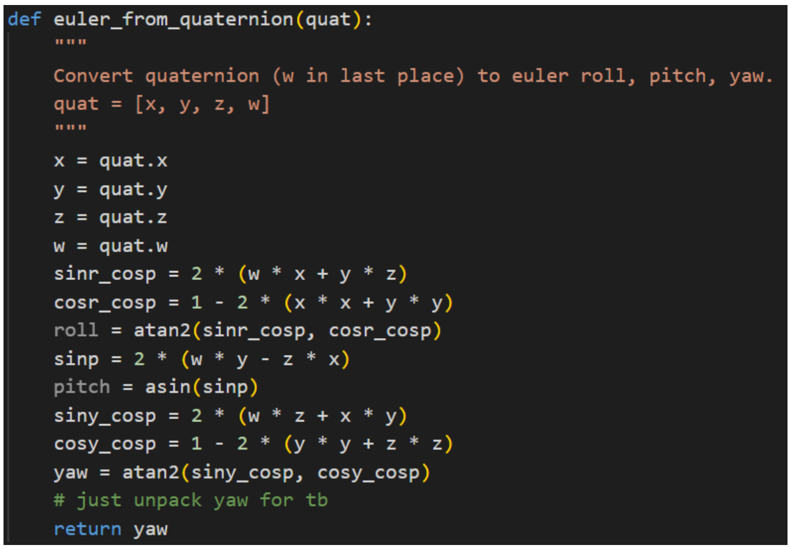
For this lab, you can either access the programs from the lab zip file and edit them in your own operating systems or use the following command to touch them in Ubuntu terminal:

```
cd DASE7505_student
git fetch
git checkout labTwo
```

Then, follow and understand utilities.py, decisions.py, and controller.py.

1. utilities.py:

In the previous lab, you are required to fill in this part:



```python
def euler_from_quaternion(quat):
    """
    Convert quaternion (w in last place) to euler roll, pitch, yaw.
    quat = [x, y, z, w]
    """
    x = quat.x
    y = quat.y
    z = quat.z
    w = quat.w
    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = atan2(sinr_cosp, cosr_cosp)
    sinp = 2 * (w * y - z * x)
    pitch = asin(sinp)
    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = atan2(siny_cosp, cosy_cosp)
    # just unpack yaw for tb
    return yaw
```

In ROS (and robotics generally), a robot's orientation is often represented as a quaternion

$$q = (x, y, z, \omega)$$

which encodes a 3D rotation without suffering from gimbal lock.
However, for mobile robots moving on a plane, we usually only need yaw (θ) — the rotation around the z-axis.

Derivation

Given a unit quaternion $q = (x, y, z, \omega)$, the Euler angles (roll φ, pitch θ, yaw ψ) are obtained as:

$$roll = \text{arctan2}\big(2(\omega x + yz),\, 1 - 2(y^2 + z^2)\big)$$

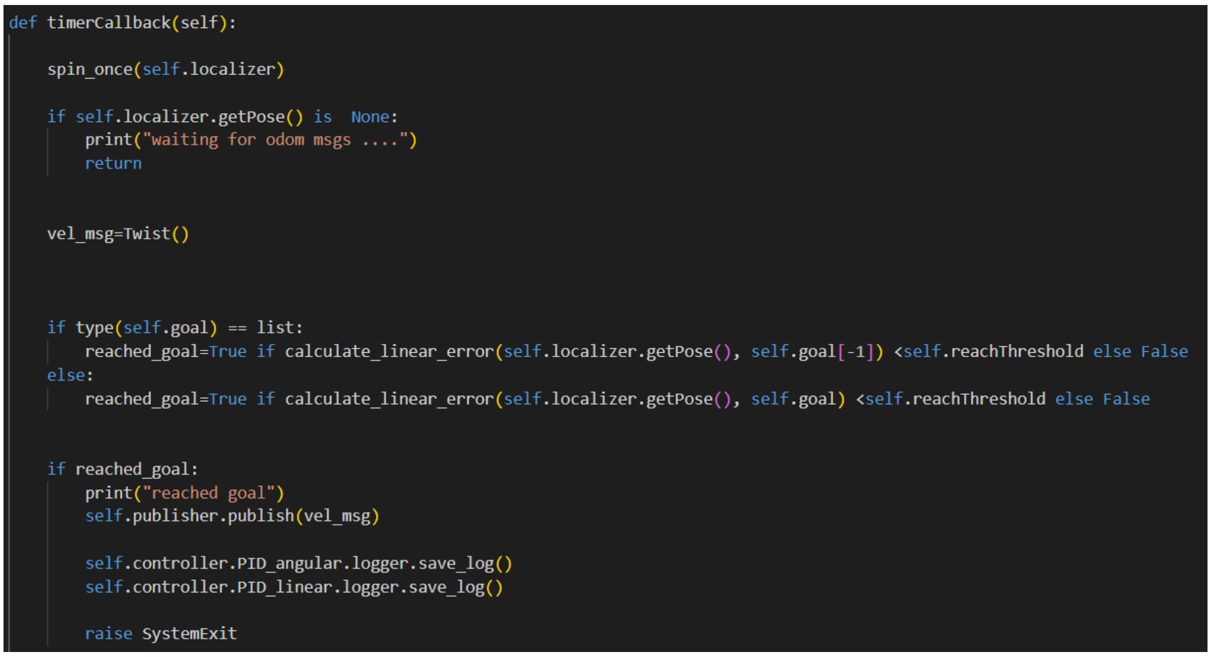
$$pitch = \text{arcsin2}(2(\omega y - zx))$$

$$yaw = arctan2\big(2(\omega z + xy),\, 1 - 2(y^2 + z^2)\big)$$

This function computes all three internally, but since the TurtleBot is planar, only yaw is returned:

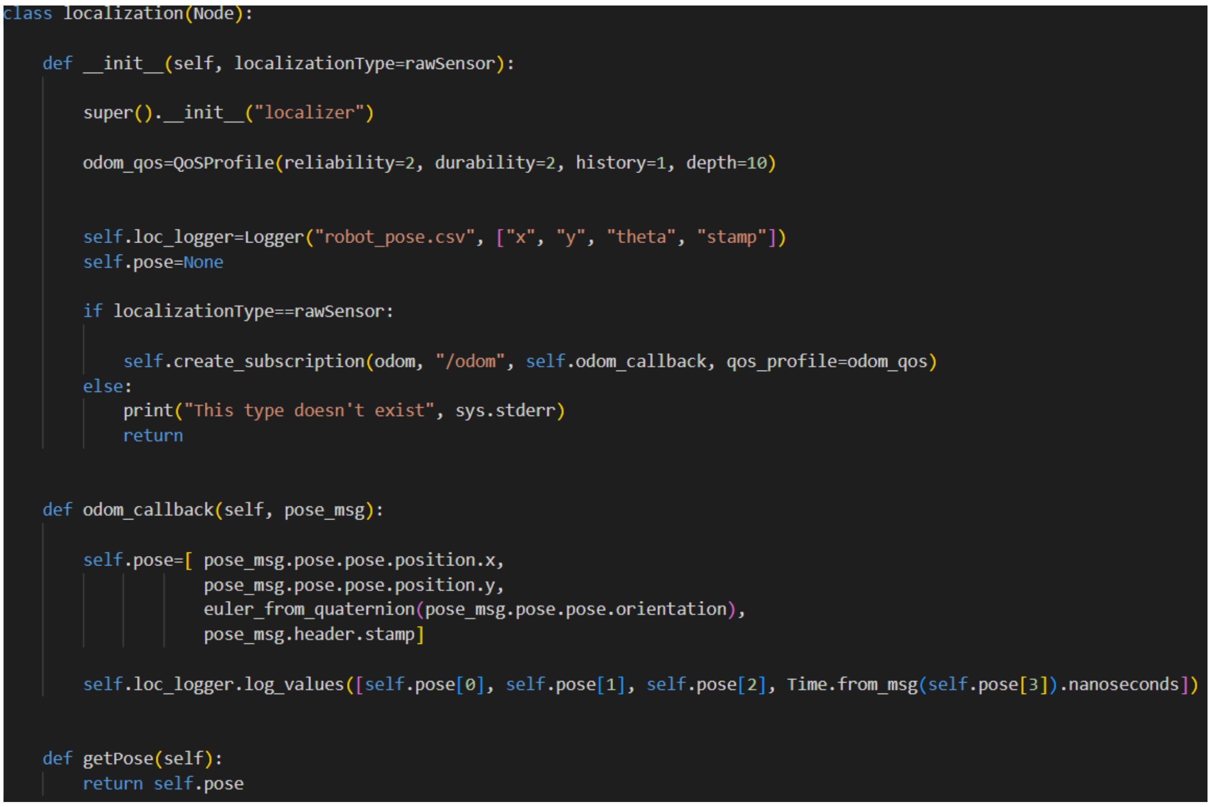
$$\psi = \text{arctan2}(2(\omega z + xy),\, 1 - 2(y^2 + z^2))$$

2. decisions.py

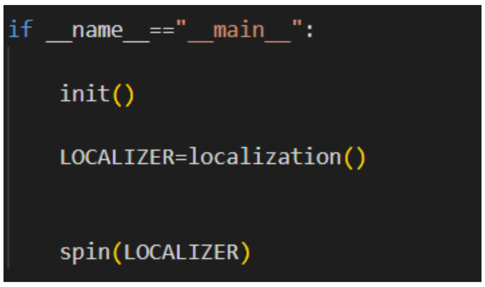


```python
def timerCallback(self):

    spin_once(self.localizer)

    if self.localizer.getPose() is  None:
        print("waiting for odom msgs ....")
        return


    vel_msg=Twist()



    if type(self.goal) == list:
        reached_goal=True if calculate_linear_error(self.localizer.getPose(), self.goal[-1]) <self.reachThreshold else False
    else:
        reached_goal=True if calculate_linear_error(self.localizer.getPose(), self.goal) <self.reachThreshold else False


    if reached_goal:
        print("reached goal")
        self.publisher.publish(vel_msg)

        self.controller.PID_angular.logger.save_log()
        self.controller.PID_linear.logger.save_log()

        raise SystemExit
```

3. localization.py



```python
class localization(Node):

    def __init__(self, localizationType=rawSensor):

        super().__init__("localizer")

        odom_qos=QoSProfile(reliability=2, durability=2, history=1, depth=10)


        self.loc_logger=Logger("robot_pose.csv", ["x", "y", "theta", "stamp"])
        self.pose=None

        if localizationType==rawSensor:

            self.create_subscription(odom, "/odom", self.odom_callback, qos_profile=odom_qos)
        else:
            print("This type doesn't exist", sys.stderr)
            return


    def odom_callback(self, pose_msg):

        self.pose=[ pose_msg.pose.pose.position.x,
                    pose_msg.pose.pose.position.y,
                    euler_from_quaternion(pose_msg.pose.pose.orientation),
                    pose_msg.header.stamp]

        self.loc_logger.log_values([self.pose[0], self.pose[1], self.pose[2], Time.from_msg(self.pose[3]).nanoseconds])


    def getPose(self):
        return self.pose
```



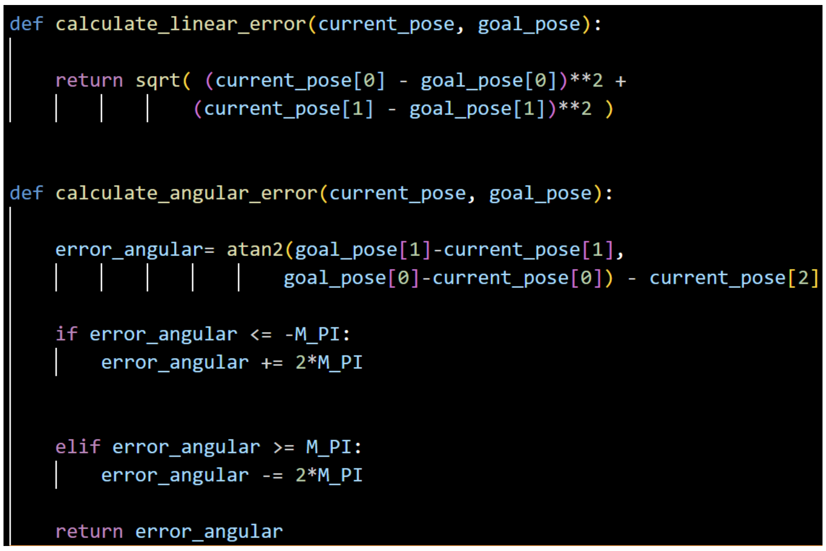
```python
if __name__=="__main__":

    init()

    LOCALIZER=localization()


    spin(LOCALIZER)
```

Why we have these three lines of code here and using the “if __name__=="__main__"?

PID Control

Start with a simple case and write a P controller only.

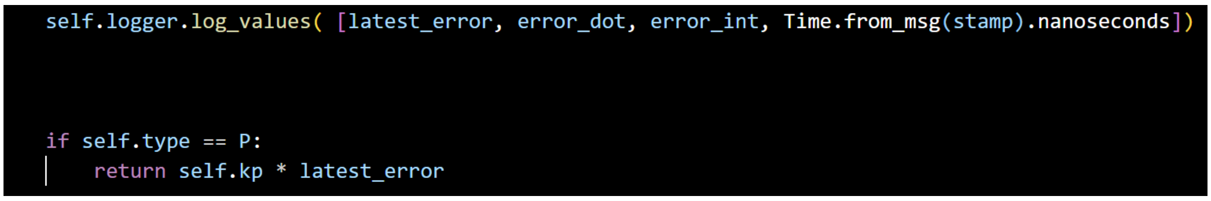
1. Compute the linear and angular error:



```python
def calculate_linear_error(current_pose, goal_pose):

    return sqrt( (current_pose[0] - goal_pose[0])**2 +
                 (current_pose[1] - goal_pose[1])**2 )


def calculate_angular_error(current_pose, goal_pose):

    error_angular= atan2(goal_pose[1]-current_pose[1],
                         goal_pose[0]-current_pose[0]) - current_pose[2]

    if error_angular <= -M_PI:
        error_angular += 2*M_PI


    elif error_angular >= M_PI:
        error_angular -= 2*M_PI

    return error_angular
```

Try to write the step-by-step math in a professional latex/mathtype format similar to the previous elaborations

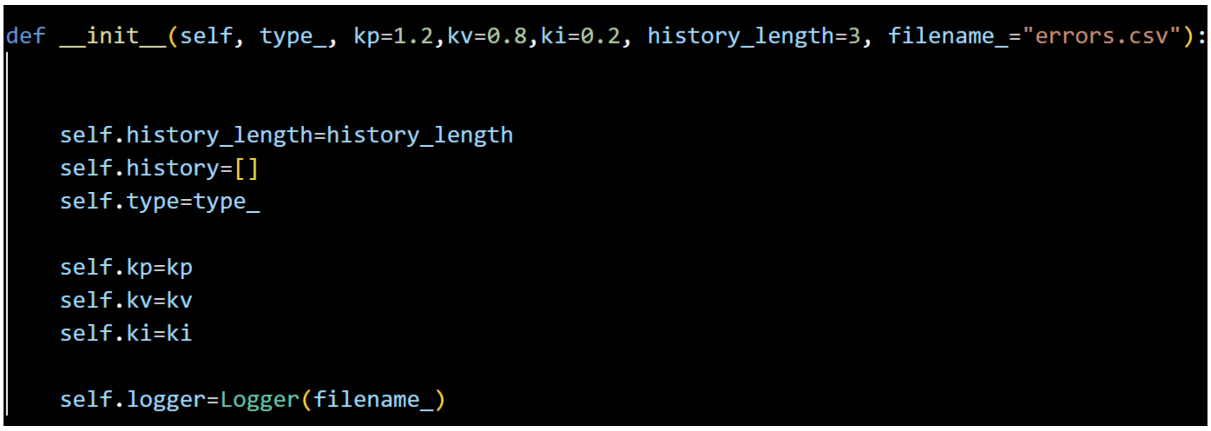
2. Use the error to implement the control law of the P-controller and follow the comments in pid.py: No integral/derivative terms yet, just a gain times the error.



```python
    self.logger.log_values( [latest_error, error_dot, error_int, Time.from_msg(stamp).nanoseconds])



    if self.type == P:
        return self.kp * latest_error
```

In the pid.py, how history of error is handled?

3. Log the errors to evaluate the performance of the controller and to tune the gains:

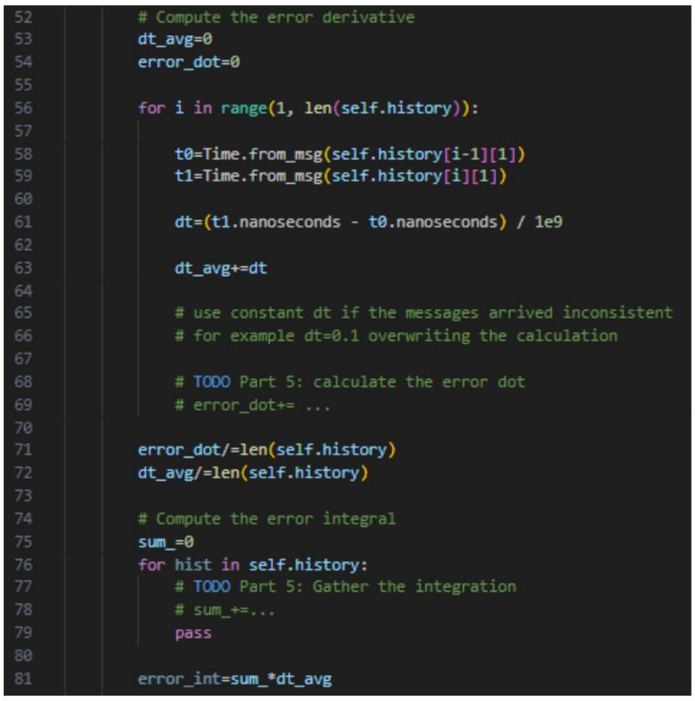


```python
def __init__(self, type_, kp=1.2,kv=0.8,ki=0.2, history_length=3, filename_="errors.csv"):


    self.history_length=history_length
    self.history=[]
    self.type=type_

    self.kp=kp
    self.kv=kv
    self.ki=ki

    self.logger=Logger(filename_)
```

Our predefined utilities.Logger is already designed to write CSV headers and rows → how errors.csv is obtained and logged?

Upgrade to a PID controller

1. Implement the error derivative and integral:



```python
        # Compute the error derivative
        dt_avg=0
        error_dot=0

        for i in range(1, len(self.history)):

            t0=Time.from_msg(self.history[i-1][1])
            t1=Time.from_msg(self.history[i][1])

            dt=(t1.nanoseconds - t0.nanoseconds) / 1e9

            dt_avg+=dt

            # use constant dt if the messages arrived inconsistent
            # for example dt=0.1 overwriting the calculation

            # TODO Part 5: calculate the error dot
            # error_dot+= ...

        error_dot/=len(self.history)
        dt_avg/=len(self.history)

        # Compute the error integral
        sum_=0
        for hist in self.history:
            # TODO Part 5: Gather the integration
            # sum_+=...
            pass

        error_int=sum_*dt_avg
```

Calculate the derivative error

Calculate the integral error

Complete this part of the code with math elaborations

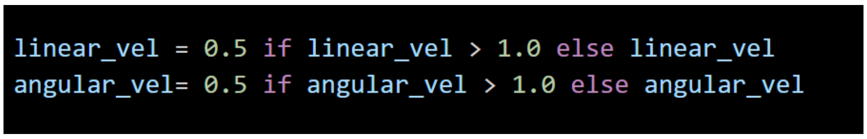
2. Implement the control laws for PD, PI, PID by combinations of the previously calculate error values to create feedback:

```
if self.type == P:
    return #(.....)

elif self.type == PD:
    return #(.....)

elif self.type == PI:
    return #(.....)

elif self.type == PID:
    return #(.....)
```

Complete this part of the code

The proportional term of the PID controller is $K_p e(t)$. What physical meaning does $e(t)$ have in this context?

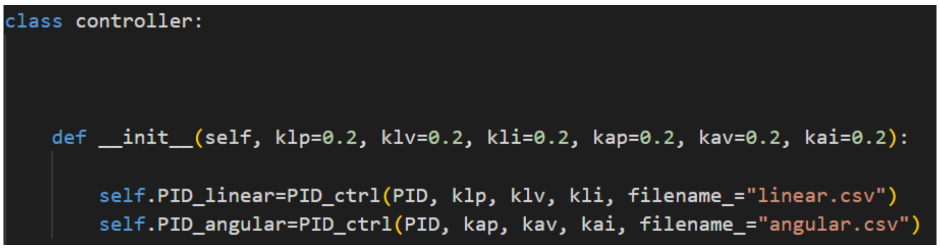
3. Understand the saturation limits for the robot's linear and angular velocity (in controller.py)



```
linear_vel = 0.5 if linear_vel > 1.0 else linear_vel
angular_vel= 0.5 if angular_vel > 1.0 else angular_vel
```

Why we need the saturations limits here? What might happen if were removed?

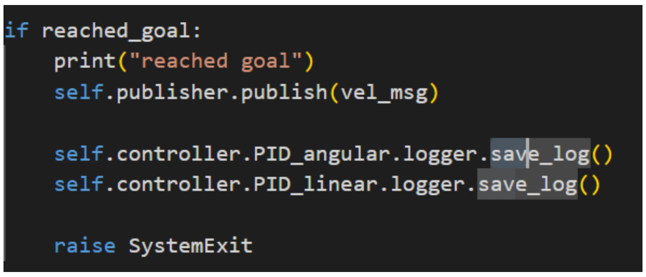
4. In controller.py, the controller class creates two PID controllers: one for linear motion and one for angular motion.



```
class controller:



    def __init__(self, klp=0.2, klv=0.2, kli=0.2, kap=0.2, kav=0.2, kai=0.2):

        self.PID_linear=PID_ctrl(PID, klp, klv, kli, filename_="linear.csv")
        self.PID_angular=PID_ctrl(PID, kap, kav, kai, filename_="angular.csv")
```

Why are separate controllers used instead of a single PID for both?

5. In decisions.py, the robot reaches the goal, the node calls save_log() and stops the loop



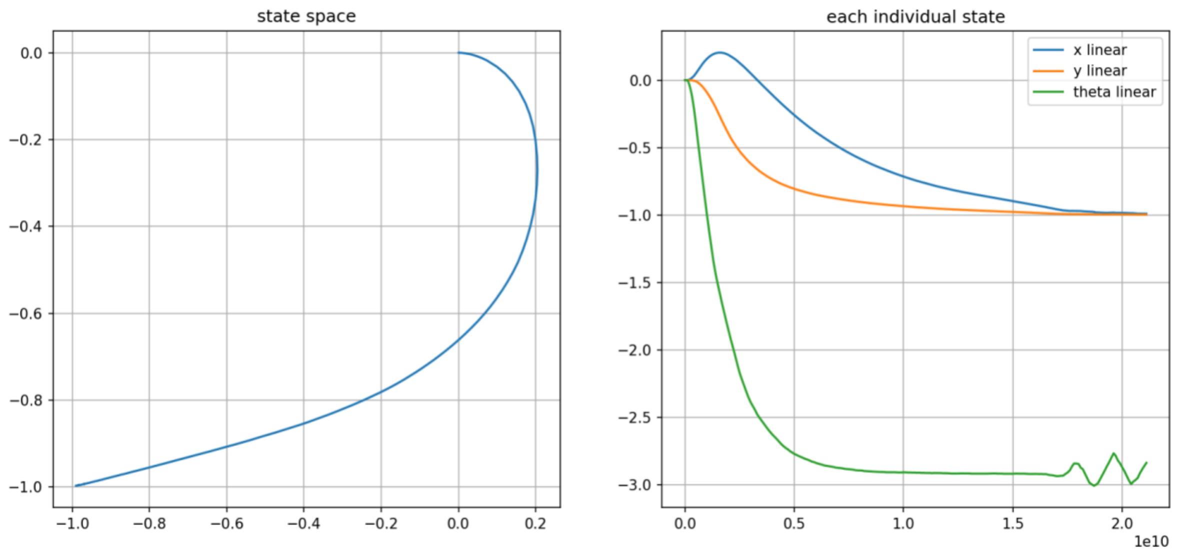
```
if reached_goal:
    print("reached goal")
    self.publisher.publish(vel_msg)

    self.controller.PID_angular.logger.save_log()
    self.controller.PID_linear.logger.save_log()

    raise SystemExit
```

Why is it important to save the log **before** termination?

Suggest one performance metric that could be computed from the log to evaluate control quality.

(Optional) Log your errors to evaluate the performance of the controllers (focus on the tuning for P and PID) and to plot them.

Example:



Submission and Assessment:

Please prepare a written report in pdf format containing personal details in the front page:

- Name (Family Name, First Name)
- UID

Report the following:

- Your filled-in python codes with explanations (i.e., how the parameters change the performance accordingly?)
- A brief discussion to show your understanding of the closed loop control and PID for robotics (e.g., architecture map/flowchart of different python programs provided in this lab repo; questions highlighted in step-by-step instructions) Hint: you may leverage on the course material and the online documentation of the messages to better interpret your data.

There are no minimum page limit or format requirements, but a more professional report could contribute to a higher mark for this lab exercise.

Deadline: check the announcements in HKU Moodle.

Questions:

Any technical issues may be contacted via emails to our TA team (email addresses enclosed in lecture note).

Thanks for your cooperations in our lab sessions and support!

We welcome any constructive feedback to better improve our course

Prepared by Peter WANG
Teaching Assistant for DASE7505
Updated October 2025