Inspiring Excellence

Student ID:		**Lab Section:**	
Name:		**Lab Group:**	

Software Lab 4

Proportional–Integral–Derivative (PID) Controller

Topic Overview

This lab aims to acquaint students with the fundamentals of Proportional–Integral–Derivative (PID) controller in ROS. It covers the implementation of a PID controller of a drone robot in Python and the visualization of the robot's movements in the *Gazebo* simulator.

Learning Outcome

After this lecture, the students will be able to:

a. Know the concepts of PID controllers.
b. Implement a basic PID controller in Python and visualize in Gazebo.
c. Implement a simple quadrotor & remote-control system in Python.
d. Learn how to build and develop a practical project in ROS.

Prepared by Mir Sayeed Mohammad, and Shayekh Bin Islam

4.1 PID Control

PID systems automatically apply accurate and responsive correction to a control function. An everyday example is the cruise control on a car, where ascending a hill would lower speed if constant engine power were applied. The controller's PID algorithm restores the measured speed to the desired speed with minimal delay and overshoot by increasing the power output of the engine in a controlled manner.

The overall control function is given by:

$$u(t) = K_\text{p} e(t) + K_\text{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\text{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t},$$

where K_p, K_I, and K_D, all non-negative, denote the coefficients for the proportional, integral, and derivative terms respectively (sometimes denoted P, I, and D).

Although a PID controller has three control terms, some applications need only one or two terms to provide appropriate control. This is achieved by setting the unused parameters to zero and is called a PI, PD, P, or I controller in the absence of the other control actions. PI controllers are fairly common in applications where derivative action would be sensitive to measurement noise, but the integral term is often needed for the system to reach its target value.

The derivative term can be approximated as the $D_{t2} = (e_{t2} - e_{t1}) / (t_2 - t_1)$ and integral term can be approximated as $I_{t2} = I_{t1} + e_{t2}(t_2 - t_1)$.

4.2 PID Controller for the Quadrotor in Python

The implementation of of the PID controller is located at the file:
~/lab4_catkin_ws/src/quadrotor/hector_quadrotor/hector_quadrotor_controller/src/set_pid.py

Here the skeleton of the functions are given. For calculating the P, I and D components, we are given all the state of the controller, **`(prev_p, prev_i, prev_d, current_error, dt)`** which are respectively the P, I & D values of the previous time step, currentr error and the time duration (in seconds) for the current state. Finally, combine the three components will the coefficients.

```
def get_p(prev_p, prev_i, prev_d, current_error, dt):
    return 0.0

def get_i(prev_p, prev_i, prev_d, current_error, dt):
    return 0.0

def get_d(prev_p, prev_i, prev_d, current_error, dt):
    return 0.0
```

```
def combine_pid(current_p, current_i, current_d, k_p, k_i, k_d):
    return 0.0
```

Now, for the remote controller, the source code is located in this file:
~/lab4_catkin_ws/src/quadrotor/hector_ui/src/ui_hector_quad.py
Write appropriate Twist messages for controlling the drone to Move Up, Move Down, Move Forward, Move Backward, Move Right, Move Left, Turn Right (Rotate Clockwise) and Turn Left (Rotate Counter-Clockwise). The Move Up control is implemented for you. Others have to be implemented similarly.

```
# Line number 104
def up_fun():
    setText("Up")
    vel_msg = Twist()
    # TODO: Move Up => vel_msg.linear.z = float(1.0)
    vel_pub.publish(vel_msg)

def down_fun():
    setText("Down")
    # TODO: Move Down

def forward_fun():
    setText("Fordward")
    # TODO: Move Forward

def backward_fun():
    setText("Backward")
    # TODO: Move Backward

def right_fun():
    setText("Right")
    # TODO: Move Right

def left_fun():
    setText("Left")
    # TODO: Move Left

def cw_fun():
    setText("Turn Right")
    # TODO: Turn Right
```

```python
def ccw_fun():
    setText("Turn Left")
    # TODO: Move Left
```

Running the Quadrotor

At first we need to build our workspace. We have move the quadrotor folder output the source to build its dependencies first, then move back and build itself. This will take some time (10 minutes).

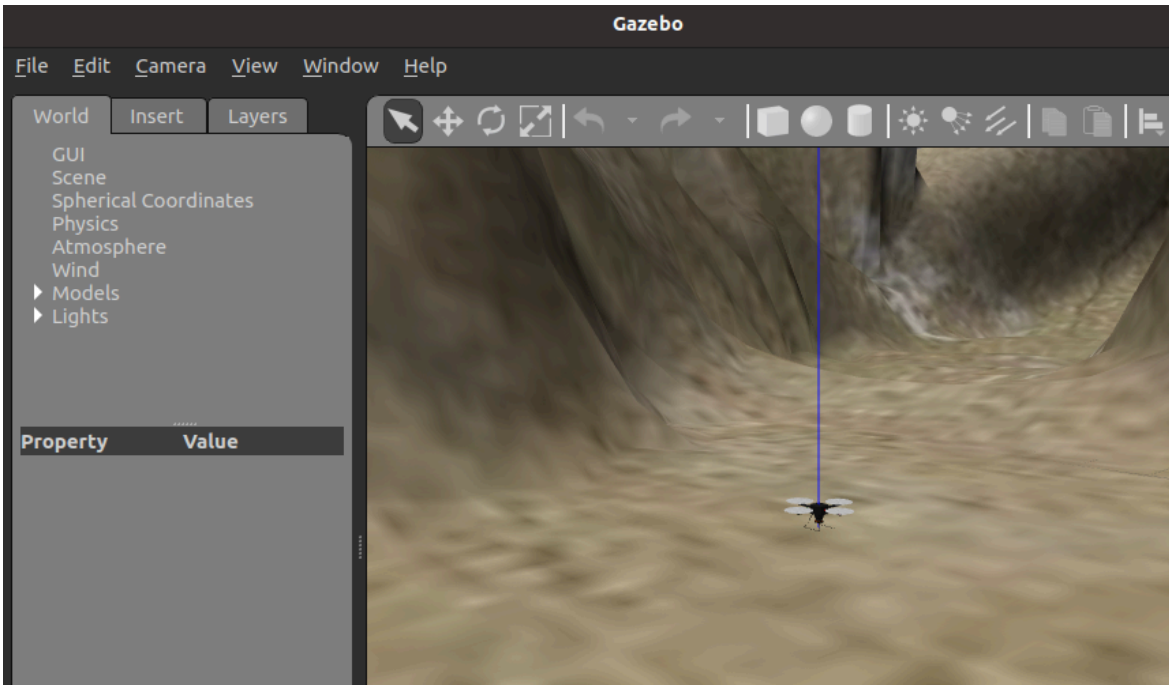
```
ubuntu@ubuntu2004:~/lab4_catkin_ws$ mv src/quadrotor/ .
ubuntu@ubuntu2004:~/lab4_catkin_ws$ catkin_make
ubuntu@ubuntu2004:~/lab4_catkin_ws$ mv quadrotor/ src/
ubuntu@ubuntu2004:~/lab4_catkin_ws$ catkin_make
```

Now, we can register this workspace to our environment.

```
ubuntu@ubuntu2004:~/lab4_catkin_ws$ source devel/setup.bash
ubuntu@ubuntu2004:~/lab4_catkin_ws$ echo "source $PWD/devel/setup.bash" >> $HOME/.bashrc
```

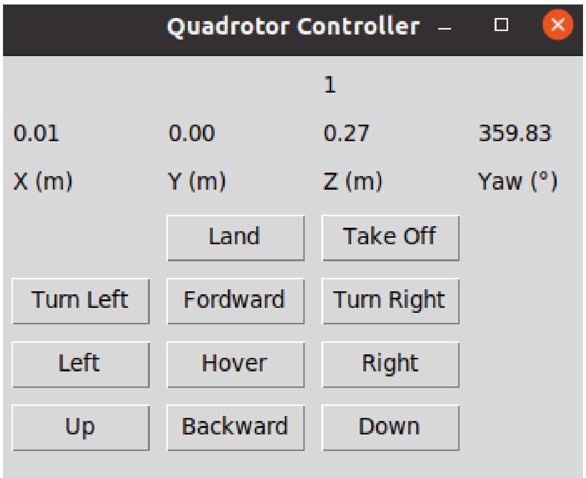
Now, we bring our Quadrotor environment to start the Gazebo simulator.

```
ubuntu@ubuntu2004:~/lab4_catkin_ws$ roslaunch hector_quadrotor_demo outdoor_flight_gazebo.launch
```



To control the drone, we launch:

```
ubuntu@ubuntu2004:~/lab4_catkin_ws$ rosrun hector_ui ui_hector_quad.py
```



To make the drone fly, press the **Up** button first. If the sources codes are correct, the drone will start to fly up slowly. To stop the drone flying up press **Hover**. Similarly, other controls should work. In case the quadrotor crashes, turn off the simulator and re-run.

Tuning the PID Co-Efficients

The PID co-efficients for the quadrotor are mentioned in this file:
~/lab4_catkin_ws_soln/src/quadrotor/hector_quadrotor/hector_quadrotor_controller/params/controller.yaml

```yaml
controller:
   pose:
       type: hector_quadrotor_controller/PoseController
       xy:
           k_p: 2.0
           k_i: 0.0
           k_d: 0.0
           limit_output: 5.0
       z:
           k_p: 2.0
           k_i: 0.0
           k_d: 0.0
           limit_output: 5.0
       yaw:
           k_p: 2.0
           k_i: 0.0
           k_d: 0.0
           limit_output: 1.0
   twist:
       type: hector_quadrotor_controller/TwistController
       linear/xy:
```

```
        k_p: 5.0
        k_i: 1.0
        k_d: 0.0
        limit_output: 10.0
        time_constant: 0.05
    linear/z:
        k_p: 5.0
        k_i: 1.0
        k_d: 0.0
        limit_output: 10.0
        time_constant: 0.05
    angular/xy:
        k_p: 10.0
        k_i: 5.0
        k_d: 5.0
        time_constant: 0.01
    angular/z:
        k_p: 5.0
        k_i: 2.5
        k_d: 0.0
        limit_output: 3.0
        time_constant: 0.1
    limits:
        load_factor: 1.5
        force/z: 30.0
        torque/xy: 10.0
        torque/z: 1.0
motor:
    type: hector_quadrotor_controller/MotorController
```

Answer the following Questions:

1. The `controller.pose.xy, controller.pose.z and controller.pose.yaw` uses `k_p = 2.0` . If we change `controller.pose.xy.k_p = 10.0 or 20.0`. Then, try to fly the quadrotor again. Do you notice any change in the drone control during its flying? [When **controller.yaml** is modified to change PID co-efficients, we need to close all and restart the simulator and remote controller again to take this change effect.]
2. Change `controller.pose.xy.k_p = 10.0` and `controller.pose.xy.k_i = 5.0.` Do you notice any change in the drone control during its flying?
3. List down the P, PI and PID controllers from the **controller.yaml** file. For a true PID controller, k_p, k_i and k_d - all must be non-zero. For a PI controller, k_d is zero.
4. Could you find a true PID controller? If yes, explain why is it important to use PID there and using P/PI controller is not enough in that case.

Submission

1. **set_pid.py** `and` **ui_hector_quad.py** Python files.

2. Answer the given questions in the report.
3. Draw the tentative node-topic communication graph for the quadrotor and remote-control system.

Resources

1. https://app.theconstruct.ai/Desktop Free virtual machine online hosted by The Construct. They also have an extensive list of tutorials and courses on ROS. https://app.theconstruct.ai/courses/
2. ROS official tutorials: https://wiki.ros.org/ROS/Tutorials