**EXPERIMENT 8**

**Flight of a Small Quadcopter with Obstacle Avoidance**

**Objective**

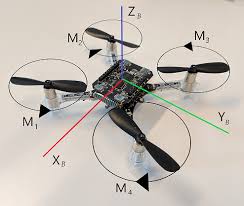
This experiment will introduce students to the flight and control of a small quadcopter, the CrazyFlie 2.1. In Part I, students will adjust the gains of a PID controller in an autonomous waypoint maneuver. In Part II, students will interact with the obstacle avoidance capabilities of the quadcopter.

**Equipment Required**

1. Crazyflie 2.1 quadcopter with flow deck and multi-ranger deck
2. Crazyradio 2.4 GHz
3. Crazyflie Python Client

**Background**

We define the body frame of a quadrotor with the X, Y and Z axes shown below in Figure 8.1, with roll angle about the x-axis, pitch angle about the y-axis, and yaw angle about the z-axis. There are different ways of attaching a reference frame to a quadcopter, Figure 8.1 shows the convention consistent with Crazyflie quadrotor’s source code.



**Figure 8.1 UAV Body Frame Axes**

(source: <http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8991431&fileOId=8991432>)

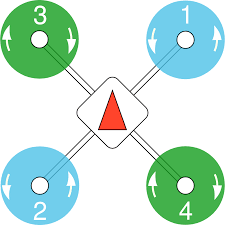
A quadrotor has four sources of thrust and torque with its four rotors. As seen below in Figure 8.2, adjacent rotors rotate in opposite directions.

In Figure 8.2, If the output of motor 1,4 are increased and 2,3 are decreased, the quadrotor can maintain an equal amount of total thrust while creating a roll moment.

If the output of motor 1,3 are increased and 2,4 are decreased, the quadrotor can maintain an equal amount of total thrust while creating a pitch moment.

If the output of motor 1,2 are increased and output of motor 3,4 decreased, the quadcopter can develop a yawing moment.

This clever arrangement of rotors enables control of a quadrotor in 3D space.



**Figure 8.2 Quadrotor Propeller Direction**

**(Note this does NOT represent crazyflie’s configuration)**

**(source:** <https://dev.px4.io/v1.9.0/en/airframes/airframe_reference.html>)

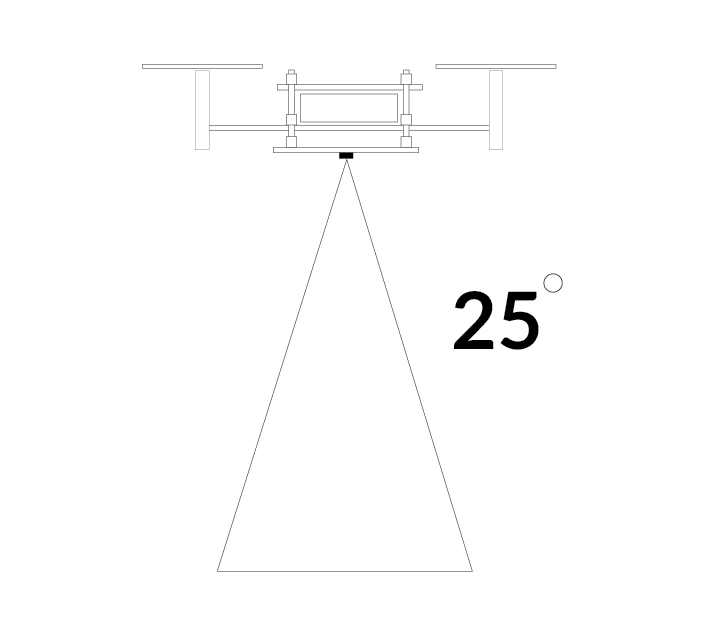
**Flow Deck:**

The Flow Deck consists of a laser distance sensor that measures distance to the ground and a low resolution camera called optical flow sensor which measures movements parallel to the ground. Together these sensors allow the Crazyflie to interpret its movement in horizontal and vertical direction.

Note that the distance sensor will return the minimum distance to any object with its detection range as depicted in Figure 8.3. During the quadcopter’s operation, it may mistake an object (like a box lying on the ground) as the actual ground, try to maintain a certain distance over the false ground, and therefore suddenly and unexpectedly accelerate upwards. So be cautious and clear out the flight area before flight tests.

**Multi-ranger deck**

The Multi-ranger deck uses five laser sensors to measure the distance in the front/back/left/right/up direction. This enables the Crazyflie to detect proximity to objects at up to four meters. This allows Crazyflie to avoid obstacles.



**Figure 8.3 Cone Shape of the Crazyflie Height Sensing**

**Part I: Position PID Controller Tuning**

Four PID controllers work together to facilitate position control of Crazyflie.

The outermost position controller, the controller we are tuning today, takes the target position as reference, and calculates a target linear velocity to be sent to the velocity controller.

The velocity controller takes target linear velocity as reference, and calculates a target roll/pitch angle to be sent to the attitude controller, which uses the target roll/pitch angle as reference and calculate target roll/pitch rate (angular velocity).

Finally, the innermost rate controller takes the target roll/pitch rate as reference, and directly controls the motors to generate moment in X/Y axis to achieve desired angular rates.

In this lab, we will vary the PID gains for the outmost position controller and treat the inner controllers as a black box.

We shall use the following PID controller gains as a starting point.

**Table 2 Initial PID Gains**

|  |  |  |  |
| --- | --- | --- | --- |
| Initial PID Gains | X-Axis | Y-Axis | Z-Axis |
| **kp** | 2.0 | 2.0 | 2.0 |
| **kd** | 0.2 | 0.2 | 0.2 |

Remember, the current altitude is measured with the laser distance sensor, who reports the distance to the closest obstacle within the cone shaped detection range. Therefore, if the Crazyflie is too close to a wall or surface, the height reading may not be accurate. The area of the cone increases with altitude.

To see the effect of PID gains have on vehicle performance, we will command the quadrotor to follow a set of waypoints, each time with different PID gains.

We shall use lab8\_part1\_pid.py to control the Crazyflie quadrotor. The program allows user to set PID gains for position and velocity controller. Upon execution, the program would instruct the quadrotor to take off, follow a set of waypoints, and land, meanwhile record quadrotor’s position throughout the experiment.

There are two sections of the program you may want to change

1. Waypoints
2. # Change the sequence according to your setup
3. #             x    y    z  YAW
4. sequence = [
5. # make sure the x,y coordinate of the first waypoint is 0,0
6. (0, 0, 0.4, 0),
7. (0.5, 0, 0.4, 0),
8. (0.5, 0.5, 0.4, 0),
9. (0.5, 0.5, 0.1, 0),
10. ]

The waypoints are in reference to the world coordinate frame, which is aligned with the vehicle body frame when the program is executed. X is aligned with forward direction, Y leftward, and Z upward.

You may change target waypoints to your liking, make sure

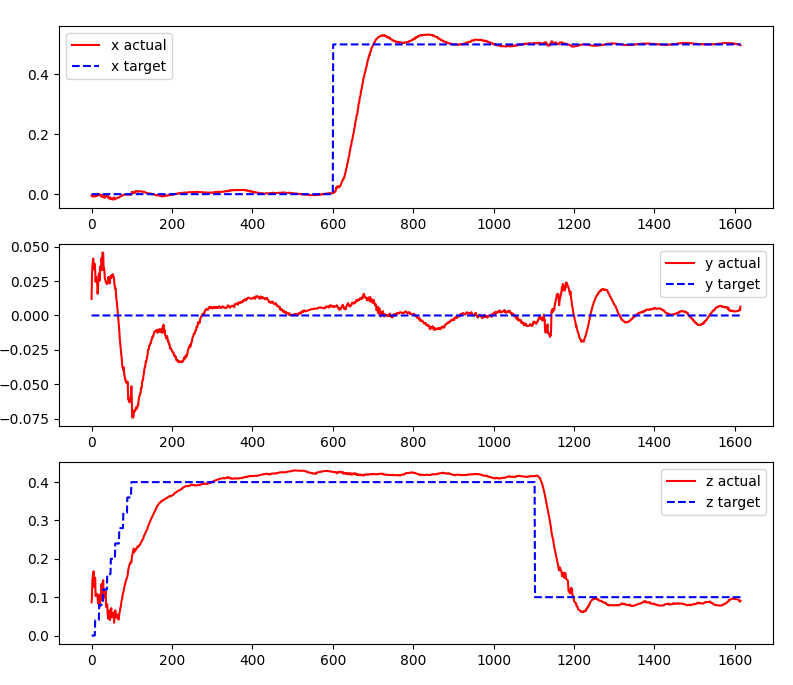
1. You use same waypoints for each experiment
2. The waypoints are safe and achievable
3. The first waypoint has x,y coordinate of 0,0
4. PID Gains
5. # set PID gains
6. def set\_gains(scf):
7. # Default gain
8. # posCtlPid.xKp: 2.0
9. # posCtlPid.xKi: 0.0
10. # posCtlPid.xKd: 0.0
11. # posCtlPid.yKp: 2.0
12. # posCtlPid.yKi: 0.0
13. # posCtlPid.yKd: 0.0
14. # posCtlPid.zKp: 2.0
15. # posCtlPid.zKi: 0.5
16. # posCtlPid.zKd: 0.0
17. # Modify Position Gains
18. scf.cf.param.set\_value('posCtlPid.xKp', 2.0)
19. scf.cf.param.set\_value('posCtlPid.xKd', 0.5)
20. scf.cf.param.set\_value('posCtlPid.yKp', 2.0)
21. scf.cf.param.set\_value('posCtlPid.yKd', 0.5)
22. scf.cf.param.set\_value('posCtlPid.zKp', 2.0)
23. scf.cf.param.set\_value('posCtlPid.zKd', 0.5)
24. # Modify Velocity Gains
25. scf.cf.param.set\_value('velCtlPid.vxKp', 10.0)
26. scf.cf.param.set\_value('velCtlPid.vxKi', 1.0)
27. scf.cf.param.set\_value('velCtlPid.vyKp', 10.0)
28. scf.cf.param.set\_value('velCtlPid.vyKi', 1.0)
29. scf.cf.param.set\_value('velCtlPid.vzKp', 15.0)
30. scf.cf.param.set\_value('velCtlPid.vzKi', 15.0)

This is where you change PID gains.

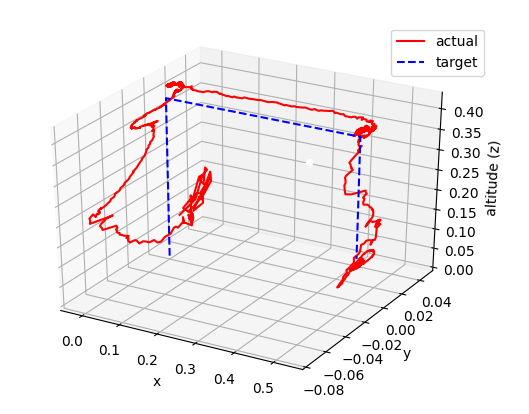
Read the code, make sure you understand what to expect. After the program is finished, a log would be generated with name lab8\_log.npy. Make sure you change the name of this file before running the next experiment, otherwise it would be overwritten.

To view the log, run lab8\_part1\_plot.py. If you have changed the name of the log, edit the following line in lab8\_part1\_plot.py to read the correct log.

1. data = np.load('lab8\_log.npy')

You should see the following

After you close this window, you should see an interactive 3D plot of the vehicle’s trajectory:



Using these graphs, you will need to compare control performance with different gains in your lab report.

Use the following gain for your experiments:

Gain Set #1: Nominal Gain in Table 2

Gain Set #2: Higher

**Table 3 Gain Set #2**

|  |  |  |  |
| --- | --- | --- | --- |
| Initial PID Gains | X-Axis | Y-Axis | Z-Axis |
| **kp** | 3.0 | 3.0 | 3.0 |
| **kd** | 0.2 | 0.2 | 0.2 |

Gain Set #3: Higher

**Table 4 Gain Set #3**

|  |  |  |  |
| --- | --- | --- | --- |
| Initial PID Gains | X-Axis | Y-Axis | Z-Axis |
| **kp** | 2.0 | 2.0 | 2.0 |
| **kd** | 0.5 | 0.5 | 0.5 |

**Procedure**

1. change PID gains to gain set #1
2. run python3 lab8\_part1\_plot.py
3. change log name, view log
4. change PID gains to gain set #2
5. run python3 lab8\_part1\_plot.py
6. change log name, view log
7. change PID gains to gain set #3
8. run python3 lab8\_part1\_plot.py
9. change log name, view log
10. Compare three logs, discuss the difference and explain how each gain affects the outcome

**Part II: Obstacle Avoidance Using a Distance Sensor**

This portion of the experiment will examine the concept of obstacle avoidance technology for a small quadcopter. Obstacle avoidance technology is becoming increasingly important in small drone applications. This lab will use a proximity sensor to detect objects in front of the drone. The concept can be extended to detect objects on all sides.

The addition of obstacle avoidance technology is essential for indoor UAV flight, resulting in less broken equipment. While the piloting of the UAV is the inner-loop control, the obstacle avoidance can be considered the outer-loop control, overriding the pilot’s inputs to prevent a collision.

The proximity sensor must be used in conjunction with an appropriate obstacle avoidance algorithm that is relevant to the flight conditions and scenario. In the case of this lab, the UAV will be not be allowed to come within 40 cm of the object in front of it. A minimum distance of 40 cm will always be maintained. The UAV will set a velocity in the direction opposite to the incoming object. As another example, an algorithm could also be created to maintain a proximity to a nearby object, i.e. maintaining distance 40 cm away from a target. This type of algorithm would be useful in swarm or following applications.

For this experiment, we try to mimic static obstacle and incoming obstacle. To mimic static obstacle, you will place an obstacle within the safety margin and see how the vehicle reacts to it. For incoming obstacle, we move the obstacle to the vehicle and see how the vehicle reacts to it.

The following codes are inside lab8\_part2\_push.py.

1. is\_close
2. **def** is\_close(range):
3. MIN\_DISTANCE = 0.4  # m
5. **if** range **is** None:
6. **return** False
7. **else**:
8. **return** range < MIN\_DISTANCE

This function determines whether there is an obstacle within the minimum distance.

1. main
2. **if** \_\_name\_\_ == '\_\_main\_\_':
3. # Initialize the low-level drivers (don't list the debug drivers)
4. cflib.crtp.init\_drivers(enable\_debug\_driver=False)
6. cf = Crazyflie(rw\_cache='./cache')
7. with SyncCrazyflie(URI, cf=cf) as scf:
8. start\_position\_printing(scf)
9. with MotionCommander(scf) as motion\_commander:
10. with Multiranger(scf) as multiranger:
11. keep\_flying = True
13. **while** keep\_flying:
14. VELOCITY = 0.5
15. velocity\_x = 0.0
16. velocity\_y = 0.0
18. **if** is\_close(multiranger.front):
19. velocity\_x -= VELOCITY
20. **if** is\_close(multiranger.back):
21. velocity\_x += VELOCITY
23. **if** is\_close(multiranger.left):
24. velocity\_y -= VELOCITY
25. **if** is\_close(multiranger.right):
26. velocity\_y += VELOCITY
28. **if** is\_close(multiranger.up):
29. keep\_flying = False
31. motion\_commander.start\_linear\_motion(
32. velocity\_x, velocity\_y, 0)
34. time.sleep(0.1)
36. **print**('Demo terminated!')
38. # Process the data before logging
39. init\_time = logged\_data[0][0]
40. **for** i **in** range(len(logged\_data)):
41. # Log time is in ms
42. logged\_data[i][0] = (logged\_data[i][0] - init\_time)\*0.001
44. with open('Experiment2.csv', 'w') as f:
45. writer = csv.writer(f, delimiter =',')
46. writer.writerows(logged\_data)

Read carefully the code listed above, you’ll need them in your discussion in your report. To land the crazyflie put your hand on top of the vehicle.

**Procedure**

1. Place the CrazyFlie in the center of the floor mat and away from any other objects.
2. Open lab8\_part2\_push.py
3. Make sure the MIN\_DISTANCE is set to 0.4 meters.
4. Make sure the VELOCITY is set to 0.5 m/s.
5. Run lab8\_part2\_push.py  
   python3 lab8\_part2\_push.py
6. The CrazyFlie should reach a steady hover before interfering. Place your hand or a flat object in front of the push sensor and observe as the CrazyFlie maintains a minimum distance of 0.2 m from the incoming object. Make observations regarding the return to steady state hover.
7. Try to imitate a static obstacle with your hand or object, commanding the CrazyFlie to maintain the minimum distance.
8. Save the position data.
9. Try to imitate an incoming obstacle with an object, slower than 0.5m/s.
10. Save the position data.
11. Try to imitate an incoming obstacle with an object, faster than 0.5m/s. Do not hit the vehicle!
12. Save the position data.
13. Close everything. **DO NOT SAVE THE CHANGES.**

**Analysis and Lab Report**

1. Explain what lab8\_part2\_push.py accomplishes and try to explain what the algorithm is doing.
2. Plot all data sets and discuss
3. Discuss what you learned from lab8\_part2\_push.py. Discuss the system response to the given types of obstacles: stationary and incoming.