Quad-rotor P&S

Exercise 3: Familiarisation with Practical Setup and PID Implementation

Learning objectives relevant for this exercise sheet:

v) Experienced the challenges of tuning a PID/LQR controller for achieving stable hover of a quad-rotor, both in simulation and on the real-world system,

vi) Be able to write C++ code for implementing a PID/LQR controller.

The following steps are an outline for getting familiar with the practical setup, the "D-FaLL-System", and for how to write and debug the code your PID implementation. Any additional information required for each step will be provided in class.

A) Getting your Crazyflie recognised by Vicon

The Vicon motion capture system detects and tracks anything that is round and reflective, called a marker. When three (or more) such markers are attached to an object with a fixed (non-symmetric) configuration, then the Vicon system can recognised this as a specific object and provides the (x, y, z) position and (γ, β, α) attitude information of that object.

The goal of this task is to make three such markers and attach them to your Crazyflie.

B) Familiarisation with the "D-FaLL-System"

A tour of the software system that allows you to control the Crazyflie will be given in class. The goal of this task is that you learn the following:

- How to launch the system,
- How to connect to your Crazyflie and start/stop a flight,
- How to emergency stop your Crazyflie, and when to do this,
- What safety measures are enforced by the system and how to know when one of these safety measures is triggered,
- Where to add your implementation of the outer loop controller,
- Where and how to compile any code changes you make,
- How to use the ".yaml" file for changing parameters "in-flight",
- How to display data of your Crazyflie in real time using the `rqt` plotting tool.
- How to connect a variable from your outer loop controller to the `DebugMsg` that is provided for debugging your controller using the `rqt` plotting tool.

C) Rotation of error (approximately) into body frame

This task is the practical version of the task from exercise sheet 2 with the same name. Provide a reference of 10 degrees for yaw, and then a reference of 60 degrees for yaw. How does the (x, y) tracking performance differ? Did one of the safety measures trigger?

The goal of this task is to implement the rotation of the outer controller errors into the body frame and remedy the (x, y) tracking performance.

In the `StudentControllerService.cpp` file where you implement your outer loop controller, the function `convertIntoBodyFrame` is already defined and called appropriately by the outer loop controller. Implement the simple rotation into body frame that was your solution to the respective task from exercise sheet 2. Then re-compile your code and test the tracking performance of the same yaw references.

Now try a step change of 340 degrees, can you explain the behaviour observed?

D) P Control for altitude

In exercise sheet 2 you implemented and tuned a PID controller for altitude and yaw control. The tuning heuristic suggested was the Ziegler-Nichols (Z&N) oscillation method. This method requires the system to be controlled with a P-only controller and the proportional gain is increased until sustained oscillations are observed, referred to as the "ultimate gain" in the handout and denoted K_u, with the period of these oscillations referred to as the "ultimate period" and denoted P_u.

The goal of this task is to implement a P-only controller for altitude (i.e., inertial frame $z^{(I)}$), and find a value for K_u and P_u on the real system.

- Run your simulations again to check what proportional gain is expected to result in sustained oscillations of altitude.
- Implement a P-only altitude controller in your outer loop controller, i.e., in the function `calculateControlOutput` that is in the `StudentControllerService.cpp` file.
- Make the P gain of your controller a parameter in the ".yaml" file so that you can tune the P gain during flight without needing to recompile for every change.
- Start from half of the K_u gain you found in simulation and increase until (sustained) oscillations are observed. Display the $z^{(I)}$ data using the `rqt` plotting tool and measure the (approximate) period of the oscillations observed.
- Remember that motor saturation may prevent sustained oscillations from being observed, so it may informative to use `rqt` to display control actions of your outer loop controller.
- Check how the error input to your controller is being computed, i.e., reference minus measurement or measurement minus reference, and ensure that you choose the sign of your proportional gain accordingly.
- For guidance on adding a ".yaml" parameter, you can mimic how the mass is defined and used. Specifically, in the `StudentController.yaml` file the relevant line of code is:
  `mass :  28`
  And in the `StudentControllerService.cpp` file the relevant line of code is:
  `yaml_cf_mass_in_grams = getParameterFloat(nodeHandle_for_paramaters,"mass");`
  Where the variable `yaml_cf_mass_in_grams` is a class variable that is defined in the header file `StudentControllerService.h` as:
  `float yaml_cf_mass_in_grams = 25.0;`

E) PID Control for altitude

Using the K_u and P_u for the altitude controller from Task D, you can apply the Z&N rules of thumb to compute a starting set of P, I, and D gains.

The goal of this task is to implement a PID controller for altitude and tune the P, I, and D gains.

- For computing the derivative and integral of the error, the yaml parameter `control_frequency` is useful, and its value is already loaded into the class variable named `yaml_control_frequency`.
- Before testing out your chosen P, I, and D gains on the real system, you should run your simulation and take note of what hand tuning adjustments were required in the respective task of exercise sheet 2 to get desirable tracking performance.

F) PID Control for yaw

Repeat Tasks D and E for the yaw controller, i.e., for the α controller.