University of Vermont
Autonomy

—

HW 4

Assigned: Thursday October 3, 2024
Due: Thursday Oct. 10, 2024 at 11:59pm.

This HW assignment contains a single [long] problem and will be graded out of 15 points. Start early! Please submit a PDF file of your solutions, and python files as requested in the problems. The figures in your PDF file must be properly labeled. The Python files must be properly commented.

Objective

In this homework, you will design and implement a digital controller to control a drone hovering above the ground. This homework will give you practice around modeling, controller design, and tuning, as well as implementing plant and controllers in Python. The ideas you will employ here parallel those we developed in class to control the differential drive ground robot.

Instructions

Download the files attached to this homework assignment from Brightspace and save them to the same folder. Set up your Python environment by downloading and installing the correct packages, as indicated in `requirements.txt`. In addition to `requirements.txt`, you will see several files: `controller.py`, `main.py`, `PID.py`, `quadcopter.py`, and `utilities.py`. You will modify `quadcopter.py` to implement the drone physics and `controller.py` to code the controller. **Do not modify any other files.** `main.py` is the main file that you will run. It contains an interactive simulation built with Pygame to visualize the drone.

Background

Over the recent years, there has been significant interest in manned and unmanned aerial drones, particularly quadcopters. Quadcopters are widely used in applications such as transportation, photography, surveillance, and search and rescue operations. A local company, Beta Technologies, is developing electric quadrotor drones for passenger use.

A typical quadcopter is equipped with four motors, each driving a propeller that generates thrust along the motor's axis. By adjusting the motor speeds, the thrust levels can be controlled to achieve a desired orientation and altitude. For example, if all motors spin at the same speed, and the total thrust cancels gravity, the quadcopter will hover. If the front two motors spin faster, the quadcopter will pitch up. A quadcopter has 6 degrees of freedom (DOF): 3 for position (x, y, z) and 3 for orientation (roll ϕ, pitch θ, and yaw ψ).

To simplify the problem, we will restrict the quadcopter's motion to a plane (i.e., the computer screen), reducing it to 3 DOF. Specifically, we will only model roll (ϕ) and translational motion in the x-y plane, ignoring pitch and yaw. Given that the motion is restricted as such, we will only model two propellers instead of four. These assumptions will make the problem tractable for this assignment. Generalization to 6 DOF is not difficult and we will visit it later.

Physics-Based Modeling

Consider the free body diagram below, showing a side-view of the quadcopter. The ellipses represent the propellers, the rectangle represents the drone's body, and the red dot represents the center of gravity (CG). The positive translational and angular directions are shown by the arrows. The angle ϕ is the roll angle, the angle of the drone with respect to the x-axis.

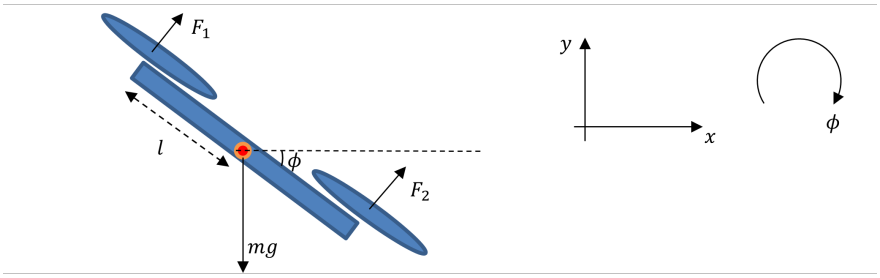

Figure 1: Free body diagram of the quadcopter.

The following forces act on the quadcopter:

- The gravitational force, mg, acting downwards on the CG.
- The thrusts from the two propellers, F_1 and F_2, acting a distance l from the CG. The sum of these forces $(F_1 + F_2)$ produces an overall force in the direction perpendicular to the body. The difference between these forces $(F_1 - F_2)$ produces a moment/torque around the CG that causes the drone to rotate. This moment is given by $(F_1 - F_2)l$.
- A drag/friction force proportional to the velocity in the x and y directions, given by $D\dot{x}$ and $D\dot{y}$, where D is the drag coefficient. These forces are not shown on the diagram to avoid clutter.

Ignoring complex aerodynamic forces and torques, and assuming rigid body dynamics, the equations of motion are:

$$m\ddot{x} = (F_1 + F_2)\sin(\phi) - D\dot{x}, \tag{1}$$
$$m\ddot{y} = (F_1 + F_2)\cos(\phi) - mg - D\dot{y}, \tag{2}$$
$$J\ddot{\phi} = (F_1 - F_2)l, \tag{3}$$

where m is the drone's mass, J is the moment of inertia, l is the arm length, and g is the acceleration due to gravity. The motor dynamics are modeled for simplicity as first-order systems without saturation:

$$\tau\dot{F}_i = -F_i + u_i, \quad i = 1, 2 \tag{4}$$

where τ is the motor time constant and u_i is the control input for motor i. Note that u_i are what your controller will eventually calculate.

Controller

To begin, note that the thrusts F_1 and F_2 affect Eq. (3) through $F_1 - F_2$. To simplify notation, introduce the collective and differential thrusts:

$$F_{\text{col}} = F_1 + F_2 \tag{5}$$
$$F_{\text{dif}} = F_1 - F_2 \tag{6}$$

Similarly, define the "desired" collective and differential thrusts:

$$u_{\text{col}} = u_1 + u_2 \tag{7}$$
$$u_{\text{dif}} = u_1 - u_2 \tag{8}$$

We will assume that the user (i.e. the pilot) has direct control over u_{col} but not u_{dif}. Instead, the user can specify the desired angle, r_ϕ, which your drone needs to track. The user specifies these two quantities (u_{col} and r_ϕ) by moving their mouse cursor on the screen when the pygame simulation is running.

The controller calculates u_{dif} and, in turn u_1 and u_2, to ensure that $\phi(t) \to r_\phi$. The control system diagram is shown in Figure 2. As implied by this figure, we have assumed that both ϕ and $\dot{\phi}$ are perfectly measured. We will thus close two feedback loops in sequence, one to stabilize the $\dot{\phi}$ dynamics and one to stabilize the ϕ dynamics, as shown in the block diagram. This architecture is the cascaded control from class and is very common in control systems design. The feedback loop from r_{in} to $\dot{\phi}$ is referred to as the "inner loop", and the entire system (with the inner loop in place) from r_ϕ to ϕ is referred to as the "outer loop". As we saw in class, with this architecture, controller design is done modularly: we first design the inner loop controller, and then design the outer loop, ensuring that the outer loop is slower (i.e. has longer settling time) as compared to the inner loop. You typically want the settling time of the inner loop to be at least 5 times smaller (i.e., faster) so that modularity is maintained. Thus, in the design of the outer loop, you can assume that the inner loop is at steady-state (i.e., is infinitely fast).

Assignment and deliverables

To complete this assignment, perform the following steps:

1. Write the dynamics (i.e. differential equations) of F_{dif} in terms of u_{dif}. Provide the new differential equations in your solution. Hint: take the time derivative of both sides of (6) and substitute in.

   Solution: $\dot{F}_{\text{dif}} = \frac{1}{\tau}(-F_{\text{dif}} + u_{\text{dif}})$

2. Re-write the dynamics in (3) in terms of F_{dif}. Note: just substitute in. Provide the new equations in your solutions.

   Solution: $J\ddot{\phi} = F_{\text{dif}} l$

3. Using the Laplace transform on the dynamics in Step 2 and substituting the values from the table on the last page of this document, derive the three transfer functions corresponding to the empty blocks in the block diagram shown in Figure 2. These will constitute your "plant". Complete the block diagram with your findings and include it in your submission.

   Solution: The right most block is $\frac{1}{s}$. The second block from the right is $\frac{l}{Js} = \frac{3}{s}$. The middle block is $\frac{1}{\tau s+1} = \frac{1}{0.2s+1}$.

4. Now design C_{in}, followed by C_{out}, such that the system satisfies:

   - Zero steady-state error for step commands despite the presence of disturbances.
   - A settling time less than 2 seconds.
   - No high-frequency oscillations.

   The procedure for designing C_{out} will be similar to HW 2 Problem 4. Your solution must include your design process, your final controllers, the step response of the inner loop, and the step response of the outer loop with the inner loop closed.

   Solution: There are two loops. If the settling time of the outer loop is to be 2 seconds, we are going to design the inner loop for a settling time of less than 0.4 seconds. We use pidTuner in Matlab with the plant $\frac{3}{s(0.2s+1)}$ to design a PD controller (since integral is not necessary for this inner loop). The controller I designed was $20 + \frac{4s}{0.02s+1}$, which yielded an inner loop settling time of around 0.15 seconds. The DC gain of the closed-loop inner loop is 1 so for the design of the outer loop, the inner loop can

Symbol	Description	Value
m	Mass of the drone	2 Kg
J	Moment of inertia of the drone	0.05 Kg·m^2
τ	Time constant of the motors	0.2 s
l	Length of each arm	0.15 m
g	Acceleration due to gravity	9.81 m/s^2
D	Air drag/viscous friction coefficient	0 Kg/s

Table 1: Drone parameters

be effectively ignored. We use pidTuner again, this time with the plant $\frac{1}{s}$ and this time with a PI controller. The controller I designed was $10 + \frac{10}{s}$. The settling time of the outer loop, as calculated from this controller and $\frac{1}{s}$ as the plant, should be around 1.7s.

Let's find the actual (not approximated) closed-loop transfer function to make sure the 1.7s settling time is accurate. This is not strictly needed for your solution, but is a good exercise to do. With the inner loop closed, the transfer function from r_{in} to $\dot{\phi}$ (i.e. the closed inner loop) is given by

$$P_i = \frac{C_{in}\frac{3}{s}\frac{1}{0.2s+1}}{1 + C_{in}\frac{3}{s}\frac{1}{0.2s+1}}.$$

With the inner loop closed, the overall transfer function from r_ϕ to ϕ is given by:

$$\frac{C_{out}P_i\frac{1}{s}}{1 + C_{out}P_i\frac{1}{s}}.$$

Please make sure you understand where these transfer functions come from.

The step response of this transfer function has settling time of around 1.7s (as predicted above) and no high frequency oscillations, as required.

5. Import your PID class (in a separate file `PID.py`) from the previous homework into `controller.py` and implement the control logic from above inside the `update()` function. Fill out the lines of code indicated by "...". Note that the update function should calculate u_{dif}, then calculate u_1 and u_2 (note: u_{col} is given, see variable `desired_thrust`).

6. open `quadcopter.py` and complete the Quadcopter class that implements the drone dynamics. To this end, fill out the code under the comments that begin with "EE 5550: ..." The drone parameters are shown in the table below.

7. Run `main.py` and hit space bar. Apply a step input by moving the mouse quickly to the right and waiting until the angle settles. Keep the vertical position of the mouse cursor close to the middle of the screen. Hit spacebar again to pause and click on "Save to CSV". This creates a .csv file in the local directory. You can open it with Python, Matlab, or Excel and plot the signals. In your solutions, include a plot of the desired angle and the actual angle as a function of time (superimposed on the same plot). Compare the step response with the one computed analytically in step 4 above. Comment on similarities and differences and if there are differences, explain what causes them. Hint: the differences should not be drastic.

   Solution: The step responses are shown in the figure below. Clearly, they are close, but not identical. The differences are due to forward euler discretization of both the controller and the plant.

Submit the above deliverables as a single PDF file. Please also include your `controller.py` and `quadcopter.py` files. Make sure your code is thoroughly commented.

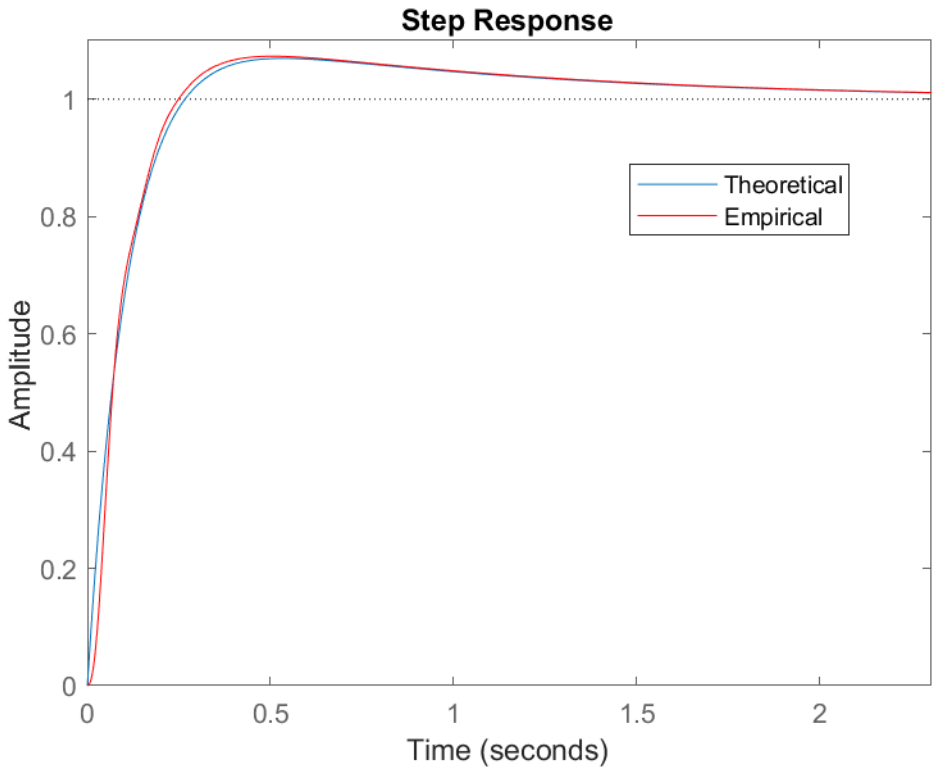

Figure 2: Step responses.

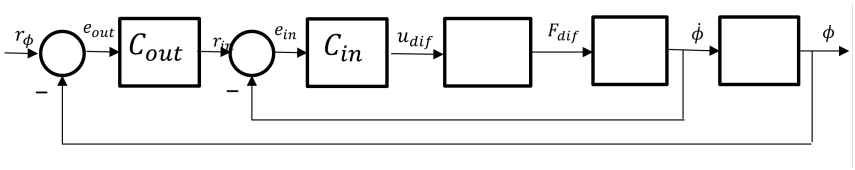

Figure 3: Controller block diagram for the drone.

Solution: the file `controller.py` is the following:

```
# The drone controller
# Copyright 2024, Prof. Hamid Ossareh @ University of Vermont

from PID import PID

class Controller:
    def __init__(self, Ts):
        # Nominal parameters that we can use for the quadcopter.
        self.Ts = Ts
        self.m = 2          # mass in kg
        self.J = 0.05       # mass moment of inertia
        self.motorTC = 0.2  # motor time constant
        self.l = 0.15       # length of each arm
        self.gravity = 9.81 # acceleration due to gravity

        # initialize controllers: angle: PD inner loop, PI outer loop,
        self.angleRateController = PID(Kp=20, Ki=0, Kd=4, N=0.02, Ts=self.Ts, umax=1000, umin=-1000, Kt=0)
        self.angleController = PID(Kp=10, Ki=10, Kd=0, N=0.02, Ts=self.Ts, umax=1000, umin=-1000, Kt=0)

    def update(self, phi, phi_dot, desired_angle, desired_thrust):

        # update the angle controller to compute the desired moment
        e_out = self.angleController.update(desired_angle, phi)
        u_dif = self.angleRateController.update(e_out, phi_dot)

        # compute actuator commands
        motor1_desired_thrust = desired_thrust / 2 + u_dif / 2
        motor2_desired_thrust = desired_thrust / 2 - u_dif / 2

        return motor1_desired_thrust, motor2_desired_thrust
```

The file `quadcopter.py` is as follows.

```
# The drone class
# Copyright 2024, Prof. Hamid Ossareh @ University of Vermont

import math

class Quadcopter:
    def __init__(self, Ts):
        # Physical Properties of the quadcopter.
        self.m = 2                # mass (kg)
        self.J = 0.05             # mass moment of inertia (kg m^2)
        self.motorTC = 0.2        # motor time constant (1/s)
        self.l = 0.15             # length of each arm (m)
        self.gravity = 9.81       # acceleration due to gravity (m/s^2)
        self.fricCoef = 0         # viscous friction of air on the body (damping coefficient)
        self.Ts = Ts

        # Initialize the states
        self.phi = 0              # roll angle
        self.phi_dot = 0          # roll rate
        self.x = 0                # initial horizontal position on the screen (0,0) is the center
        self.y = 0                # initial vertical position on the screen (0,0) is the center
        self.xdot = 0             # horizontal velocity
        self.ydot = 0             # vertical velocity
```

```
        self.motor1Thrust = self.m * self.gravity / 2            # initial motor 1 thrust
        self.motor2Thrust = self.motor1Thrust                    # initial motor 2 thrust

    # Update the dynamics
    def update(self, u1, u2):
        # Update motor dynamics and compute thrusts/moments
        self.motor1Thrust = self.motor1Thrust + self.Ts * (1 / self.motorTC) * (-self.motor1Thrust + u1)
        self.motor2Thrust = self.motor2Thrust + self.Ts * (1 / self.motorTC) * (-self.motor2Thrust + u2)

        # Update roll dynamics
        vehicleMoment = (self.motor1Thrust - self.motor2Thrust) * self.l
        self.phi_dot = self.phi_dot + (1 / self.J) * self.Ts * (vehicleMoment)
        self.phi = self.phi + self.Ts * self.phi_dot

        # Calculate the force in the x and y directions (force applied to center of mass)
        yForce = (self.motor1Thrust + self.motor2Thrust) * math.cos(self.phi) - self.m * self.gravity
        xForce = (self.motor1Thrust + self.motor2Thrust) * math.sin(self.phi)

        # Update position dynamics
        self.xdot = self.xdot + self.Ts * (1 / self.m) * (xForce - self.xdot * self.fricCoef)
        self.x = self.x + self.Ts * self.xdot

        self.ydot = self.ydot + (1 / self.m) * self.Ts * (yForce - self.ydot * self.fricCoef)
        self.y = self.y + self.Ts * self.ydot
```