AE483 Lab Manual: Week #4
Parameter Estimation and Pitch Control

T. Bretl

September 22, 2019

Strategy

This lab is going to require a significant amount of work outside of your lab section time. It is important that you be able to use the time you *are* in your lab section efficiently. We suggest that you complete all of Section 2 (Design) *before* coming to your lab section, that you complete all of Section 3 (Implementation) during your lab section, and that you complete all of Section 4 (Test) after your lab section—returning as often as necessary to get a great controller and to collect enough data. Needless to say, it is best to have completed everything you were asked to do last week (e.g., parameter estimation and model derivation) before starting lab this week. If you are behind, then we strongly recommend that you focus on Section 3 during lab—doing so enables you to run experiments with your controller. You can go back later and refine your control design, your parameter estimates, etc.

1 Goal

Last week, you derived equations of motion for the drone when mounted on the pitch test stand, and estimated the unknown physical parameters in these equations of motion—including the parameters that relate motor commands, spin rates, and applied forces and torques due to the rotors. Your goal this week is to design, implement, and test a model-based controller that makes the pitch angle of the drone track a desired value—in particular, a sinusoidal function of time. You will be responsible for writing a report that answers the following two questions:

- How accurate are your parameter estimates?

  At minimum, you will need to clearly explain and justify the way in which you obtained your estimates of k_F, k_M, and so forth, and to characterize the uncertainty in these estimates. You will also need to compare predictions made by your parameterized model in simulation to results in hardware experiments—in terms of trajectories (i.e., states and inputs as functions of time) and in terms of frequency response (i.e., magnitude and phase of θ_{pitch} as they vary with frequency of θ_{desired}).

- How well does your controller work?

  At minimum, you will need to clearly explain and justify the approach you took to control design, implementation, and test. You will also need to find the *bandwidth* of your closed-loop system—the largest frequency of the sinusoidal desired pitch angle that can be tracked with small error by your controller.[1]

[1] http://www.cds.caltech.edu/~murray/amwiki/index.php/Second_Edition

Each lab group will have different answers to these questions—it is inconceivable that two lab groups end up with exactly the same parameter estimates, the same controller, and the same results in experiment. You will be responsible for writing a report with your answers that will be submitted as a group no later than 11:59PM on Friday, October 4, 2019. Details will be posted to slack. You will do the following things in lab today to prepare for writing this report:

- Design a controller (Section 2).
- Implement this controller (Section 3).
- Test this controller (Section 4).

Section 5 has a summary of in-lab deliverables. **Make sure to save your flight data for later analysis.** It is highly recommended that you take video of all flights, in addition to logging data with the ground-station.

2 Design

Last week, you derived equations of motion for the drone on the pitch test stand. Do these things:

- Write the ordinary differential equations that describe motion about the pitch axis in the form

  $$\dot{x} = h(x, u)$$

  where

  $$x = \begin{bmatrix} \theta_2 \\ w_2 \end{bmatrix} \qquad u = \begin{bmatrix} u_2 \end{bmatrix}.$$

  In these equations, please use the value for J_2 that you found last week.

- Find the equilibrium point (x_e, u_e) that corresponds to a pitch angle of θ_{desired}.

- Linearize $\dot{x} = h(x, u)$ about (x_e, u_e) and express your result in the form

  $$\dot{x}_c = Ax_c + Bu_c$$

  where

  $$x_c = x - x_e \qquad u_c = u - u_e.$$

  You should find that A and B do not depend on θ_{desired}. As a consequence, any control policy you design based on this linearized model will work—without modification—for any desired pitch angle (at least in theory), even if this angle is a function of time.

- As we have discussed in class, a standard choice of control policy is *linear state feedback*, which has the form

  $$u_c(t) = -Kx_c(t)$$

  for some choice of gain matrix K. In theory, this choice will "work" (i.e., will make x_c go to zero, so will make x go to x_e) whenever all eigenvalues of $A - BK$ have negative real part. There are many different ways of choosing K. As you know from AE353, one approach that can be very effective is by solving the so-called LQR problem—choosing K for which

  $$u_c(t) = -Kx_c(t)$$

is the solution to

$$\begin{aligned} \underset{u_c}{\text{minimize}} \quad & \int_{t_0}^{\infty} \left(x_c(t)^T Q x_c(t) + u_c(t)^T R u_c(t) \right) dt \\ \text{subject to} \quad & \dot{x}_c(t) = A x_c(t) + B u_c(t), \quad x_c(t_0) = x_0, \end{aligned}$$

where Q and R are appropriately-size, positive definite, symmetric matrices called "weights." Luckily, it is possible to solve this problem easily in MATLAB with a call to

```
K = lqr(Ac, Bc, Q, R);
```

You are welcome to take a different approach if you have a good reason—if you don't, then apply LQR to find K.

Congratulations, you have a controller. As you know from AE353, it is good practice to verify that your choice of K makes the closed-loop linearized system stable (in theory). You will likely have to "tune" your controller to get it to work well in experiment—one way to do so is by changing the weights, the diagonal elements of Q and R.

3 Implement

At this point, the file `lab.c` in your on-board code probably looks something like this:

```
#include "lab.h"
#include "main.h"
#include "sdk.h"
#include "math.h"

int counter = 0;

void lab(void) {
    counter++;
}
```

It is probably best to start by replacing this code with the template that is available here (on Box):

https://uofi.box.com/s/oniuvgc7f5eqmomtxth32anlahhclk4p

It is probably easiest to download this template and to copy/paste the text into your existing `lab.c` file. The sections that follow lead you through each thing you'll need to change in the template in order to implement your controller.

3.1 Enable logging the right set of variables

You'll see these lines of code in the template:

```
// VARIABLES TO BE SENT OFF-BOARD
int counter = 0;                // number of times through control loop
float angle_pitch_desired = 0;  // desired pitch angle (rad)
int mu1 = 0;                    // motor command applied to rotor 1
int mu2 = 0;                    // motor command applied to rotor 2
```

The values of these variables will be computed later in the code (see below). Follow the instructions in Appendix D to make sure that the value of both `counter`, `angle_pitch_desired`, `mu1`, and `mu2`, as well as the values of `RO_ALL_Data.angle_pitch` and `RO_ALL_Data.angvel_pitch`, are logged to a data file. Please note that `angle_pitch_desired` is a floating-point number, so has `VARTYPE_SINGLE` and not `VARTYPE_INT32`.

3.2 Enable setting a few key parameters with ACI Tool

You'll see these lines of code in the template:

```
// PARAMETERS (received from off-board)
float k1 = 1;                   // first element of 1x2 matrix K
float k2 = 1;                   // second element of 1x2 matrix K
float period = 1;               // seconds
```

The parameters `k1` and `k2` are the two components of the gain matrix K from Section 2:

$$K = \begin{bmatrix} k_1 & k_2 \end{bmatrix}.$$

You are being asked to find the frequency response of your controller, so the desired value of θ_{pitch} in your experiments will be a sinusoid—the parameter `period` is the period of this sinusoid. You will want to be able to change these parameters—to "tune the gains" and to change the period of the sinusoid—without having to recompile and flash your code. Follow the instructions in Appendix E to enable setting these parameters with the ACI Tool.

3.3 Hard-code all other parameters

You'll see these lines of code in the template:

```
// ALL OTHER PARAMETERS
float kF = 1;                   // motor constant (squared spin rate to force)
float alpha = 1;                // for converting spin rate to motor command
float beta = 1;                 // for converting spin rate to motor command
float l = 1;                    // spar length (m)
float m = 1;                    // mass (kg)
float g = 1;                    // acceleration of gravity (m / s^2)
// - for computing desired pitch angle (sinusoidal with time)
float input_amplitude = 0;      // amplitude of sinusoid (rad)
int sample_rate = 1;            // number of times per second that control loop runs (Hz)
```

The values are all placeholders. Replace them with the correct values.

3.4 Get the desired pitch angle

You'll see these lines of code in the template:

```
// get current time (from counter and sample_rate)
t = 0;

// get desired pitch angle (from input_amplitude, t, and period)
angle_pitch_desired = 0;
```

To find the frequency response of your controller, the desired value of θ_{pitch} must be

$$\sin\left((2\pi/n)t\right)$$

where n is the period (in seconds) of the sinusoid and where t is the current time (also in seconds). The lines of code in the template encourage you to compute this desired value in two steps:

- First, compute the current time (`t`), given knowledge of how many times the control loop has run (`counter`) and of how many times the control loop runs per second (`sample_rate`). Note that `t` is a floating-point number and that both `counter` and `sample_rate` are integers—please read Section 6 before proceeding, and remember to cast both `counter` and `sample_rate` as floating-point numbers before using them.
- Second, compute the desired pitch angle, given the current time `t` and the period—called n in the expression above, and `period` in the code. Again, you'll want to read Section 6 and be careful to cast the integer `period` as a floating-point number before using it..

Replace line 53 and 56 with your own code.

3.5 Get the pitch angle and angular velocity from sensor data

You'll see these lines of code in the template:

```
58 // get pitch angle and angular velocity from sensor data:
59 //      RO_ALL_Data.angle_pitch
60 //      RO_ALL_Data.angvel_pitch
61 angle_pitch = 0;
62 angvel_pitch = 0;
```

Remember that, in the on-board code, `RO_ALL_Data.angle_pitch` contains the IMU's current best estimate of pitch angle, and `RO_ALL_Data.angvel_pitch` contains the IMU's current best estimate of the component of angular velocity about the pitch (body-fixed y) axis. You have worked with these variables before—in the first lab, you used `aciPublishVariable` calls to send these variables from the drone to the ground-station in order to log them in a data file. However, as you know, these variables are integers, and are in weird units. In order to use them for the purpose of control, you'll need to convert them into floating-point numbers in consistent units—radians for the pitch angle, and radians per second for the pitch angular velocity. Replace lines 61 and 62 with these conversions—they should be identical (more or less) to what you implemented in your ground station code for Lab 1.

3.6 Get the total torque and force that you need from all rotors

You'll see these lines of code in the template:

```
64 // get total torque due to rotors about y axis
65 // from k1, k2, angle_pitch, angle_pitch_desired, and angvel_pitch
66 u2 = 0;
67
68 // get total force due to rotors along -z axis
69 // from m, g, and angle_pitch
70 u4 = 0;
```

In Section 2, you designed a controller of the form

$$u_c = -Kx_c$$

where

$$K = \begin{bmatrix} k_1 & k_2 \end{bmatrix}$$

and

$$x_c = x - x_e \qquad u_c = u - u_e$$

and

$$x = \begin{bmatrix} \theta_{\text{pitch}} \\ w_2 \end{bmatrix} \qquad u = \begin{bmatrix} u_2 \end{bmatrix}.$$

Replace line 66 with an expression for u_2, the torque due to the rotors about the body $+y$ axis, that is consistent with your control design. You also found a choice of u_4, the force due to the rotors about the body $-z$ axis, that would—if you were actually flying the drone (not on a pitch test stand)—achieve gravity compensation. Replace line 70 with this expression.

3.7 Get the rotor forces to achieve the given total torque and force

You'll see these lines of code in the template:

```
72 // get forces from u2, u4, and l
73 f1 = 0;
74 f2 = 0;
```

Last week, you found expressions for the force f_1 of the front rotor and the force f_2 of the rear rotor that would achieve a given torque u_2 and a given force u_4. Replace lines 73 and 74 with these expressions.

3.8 Get the spin rates to achieve the given rotor forces

You'll see these lines of code in the template:

```
76 // get spin rates from f1, f2, and kF
77 sigma1 = 0;
78 sigma2 = 0;
```

Last week, you found an expression for the spin rate σ that would achieve a given force f with a rotor. Replace each of lines 77 and 78 with this expression (to compute σ_1 and σ_2, respectively).

3.9 Get the motor commands to achieve the given spin rates

You'll see these lines of code in the template:

```
80 // get motor commands from alpha, beta, sigma1, and sigma2
81 mu1 = 0;
82 mu2 = 0;
```

Last week, you found an expression for the motor command that would achieve a given spin rate. Replace each of lines 81 and 82 with this expression (to compute μ_1 and μ_2, respectively). Note that these motor commands are subsequently bounded (with the `if` statement on lines 89-98) before they are actually applied.

3.10 Test

Do this before proceeding:

- Compile and flash your code to a drone that is *not* on the pitch test stand.
- Confirm that you can set parameters with ACI Tool.
- Confirm that you can collect the right data with the ground station.

Show both your code (in particular, `lab.c`) and the results to a TA.

4 Test

4.1 Experiment

Once you have finished with control design (Section 2) and implementation (Section 3) on your own drone, you are ready to move to the drone that is on the pitch test stand. Work with your TA to login to the appropriate computer, to compile and flash your code to the drone on the test stand, to use ACI Tool to set parameter values, and to run your ground station. Make sure that you turn off (disarm) the drone immediately if anything goes wrong. You will want to collect data from at least five different experiments—hopefully many more—with a range of frequencies (i.e., a range of choices for the `period` of the desired pitch angle). You should be sure to choose a frequency low enough—and to have tuned your controller well enough—so that tracking performance is very good. You should also be sure to choose a frequency high enough so that tracking performance becomes very bad. The "bandwidth" of your closed-loop system in somewhere in between these two extremes—find it!

4.2 Simulation

You will also need to test your controller in simulation (e.g., in MATLAB) and compare your results to experiment in some way. Remember that it is easy to compute the frequency response of a closed-loop system in MATLAB. Additional guidance on how to simulate the closed-loop system will be provided in class and on slack (this should be review for many of you). Does simulation match experiment? If not, is there something wrong with your simulation? Or, is there something wrong with your experiment, e.g., with your controller implementation? Is there something that you are not modeling? (For example, are you modeling the fact that motor commands can't be higher than 200 or lower than 1?) These are questions for you to consider.

5 Summary of in-lab deliverables

You should have done the following things in lab:

1. Show the TA your choice of K (Section 2).
2. Show the TA that your on-board and off-board code works—that you can set the value of all parameters with ACI Tool, and that you can log all of the required data with your ground station—with a drone that is not on the pitch motion test stand (Section 3).
3. Show the TA one set of data that was collected with your controller running on the drone that is on the pitch motion test stand (Section 4).

Details of report requirements and of the submission process will be posted to Slack.

6 Help on "typecasting" in C

C is very strict when it comes to variable types. Consider the following code (in a file that I called `demo.c`):

```
#include <stdio.h>
#include <math.h>

int main(void) {
    int a = 4;
    int b = 10;
    float c = a / b;                        // <---- BAD
    printf("c = %f\n", c);
    return 0;
}
```

This code is supposed to take the quotient 4/10 and display the result—naturally, I would expect to see 0.4. If I compile and run this code, however, I get the following:

```
timothybretl$ gcc -o demo demo.c
timothybretl$ ./demo
c = 0.000000
```

The reason for this is that `a` and `b` are both integers, so when I take `a / b`, C thinks I'm working with integers and so gets rid of everything after the decimal. In order to fix this problem, I need to "typecast" all my integers as floating-point numbers:

```
#include <stdio.h>
#include <math.h>

int main(void) {
    int a = 4;
    int b = 10;
    float c = ((float) a) / ((float) b);    // <---- GOOD
    printf("c = %f\n", c);
    return 0;
}
```

Now, I get what I expected:

```
timothybretl$ gcc -o demo demo.c
timothybretl$ ./demo
c = 0.400000
```