Lab 4. Attitude control

Do this before coming to the lab:

- **Review your notes on the dynamic model of the quadcopter.**
- **Make sure your batteries are charged.**

Important notes:

- **Read the "Deliverables" section before starting work, so that you know what information you need to record (and which photos to take) as you are proceeding through the lab.**
- **Wear your safety glasses throughout the lab. You are most likely to be injured by someone else's stupidity – even if you think you're acting safely, don't assume the same of others.**
- **Make sure your batteries are charged, and charge your spare battery while using the other, so that you are not slowed down by low batteries.**

In this lab we will use the state estimate we developed in the lab 3, and the force maps from lab 2, to control the vehicle's attitude. We will do this while restricting the vehicle to a single degree of freedom (its pitch angle), using the rotational rig shown in Figure 4.2.

For this work we will assume that the estimator that you developed in lab 3 is perfect, i.e. that it outputs the true values. We will then do feedback control on these values. This is called *observer-based control*, and is a very typical method for structuring control systems.

Last update: 2017-10-17 at 16:32:16

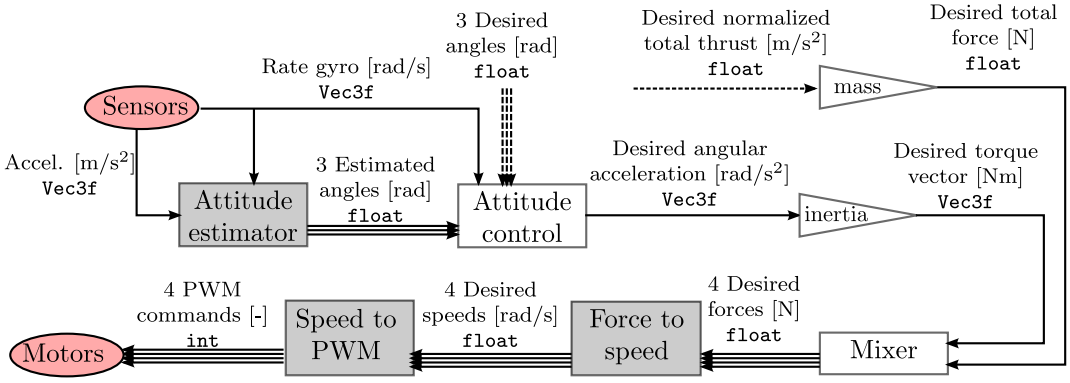


Figure 4.1: System block diagram. The shaded boxes are components that we built in the prior labs, and hardware is shown as shaded ellipses. The triangular elements represent simple multiplications, and unshaded components are the focus of this lab (and are more complex algorithms).

Our goal in this lab is to make the pitch angle of the vehicle behave like a damped second-order system. The input to the system is the torque around the vehicle y-axis, and the output is the vehicle pitch angle.

We will extend our control pipeline from the previous labs, so that it looks like Figure 4.1. Specifically, we will connect the attitude estimator from Lab 3 to the propeller speed control framework of Lab 2.

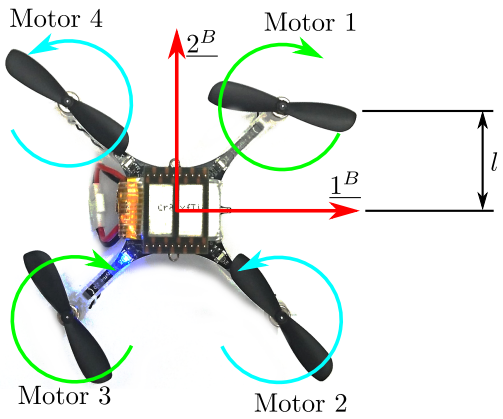
4.1 Physical parameters

For reference, the vehicle's physical parameters are listed here:

$$
\begin{aligned}
l &= 33 \times 10^{-3}\text{m} \\
m^B &= 30 \times 10^{-3}\text{kg} \\
\left[\underline{I_B^B}\right]^B &= \begin{bmatrix} 16 & 0 & 0 \\ 0 & 16 & 0 \\ 0 & 0 & 29 \end{bmatrix} \times 10^{-6}\text{kgm}^2 \\
\kappa &= 0.01\text{m}
\end{aligned}
$$

4.2 Motor force mixer

First, you will implement code that transforms a desired total force (in N), and a desired torque vector (in Nm), and outputs the four individual motor forces. You do not need to write a separate C++ function implementing this; you can just make a clearly marked section in `MainLoop`. The desired forces will be sent to the functions you used in the last lab, to eventually create PWM commands for the motors. To do this, you will require the physical layout of the vehicle:



The propeller distance is $l = 33 \times 10^{-3}$m – note that this is *not* the distance from the center of mass, which would be a factor $\sqrt{2}$ more. The coefficient coupling a propeller's force to the torque about the propeller's axis of rotation is $\kappa = 0.01$m.

The motor numbers in the figure correspond to the PWM outputs in `MainLoop()`.

Implement a mixer for the vehicle, that converts a desired torque vector (in Nm) and total force (in N) to four individual motor forces (in N). A mixer was derived in the lecture – confirm that your vehicle has the same physical layout as the assumptions that were made in the lecture. You will need this mixer for Deliverable 2.

The four forces are then converted to desired speeds by `speedFromForce()`, and from there to PWM commands by `pwmCommandFromSpeed()`.

Next, implement the two multiplication elements shown in Figure 4.1. At the end of this, the last part of your `MainLoop()` should take a desired normalized thrust (in $\mathrm{m/s^2}$) and desired angular acceleration (in $\mathrm{rad\,s^{-2}}$), and convert this to PWM commands which are

sent out to the motors.

Before we actually run the motors, we will verify that we are approximately at the right order of magnitude with our forces. Command a normalized total thrust of $8\mathrm{m\,s^{-2}}$, and an angular acceleration of $(0, 0, 0)\,\mathrm{rad\,s^{-2}}$. *Do not* arm the vehicles, but connect to the quadcopter over USB, and run `quad status` to check what the motor commands are. They should each be approximately 135 – if they are substantially higher or lower, you probably have a bug in your implementation.

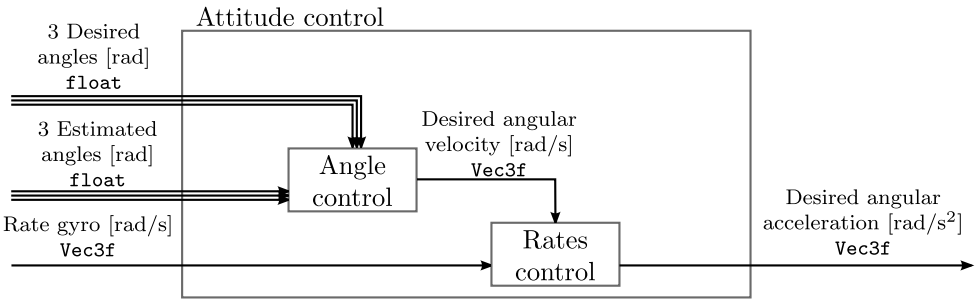
Once you have implemented this, perform the following validation steps:

- Command a normalized total thrust of $8\mathrm{m\,s^{-2}}$, and an angular acceleration of $(0, 0, 0)\,\mathrm{rad\,s^{-2}}$: when holding the vehicle in your hand, it should appear to almost take off.
- Command a normalized total thrust of $8\mathrm{m\,s^{-2}}$, and an angular acceleration of $(0, 0, 5)\,\mathrm{rad\,s^{-2}}$ while the vehicle is standing on a table: the vehicle should start to rotate about its $\underline{3^B}$ axis, in a positive direction. Friction from the surface it is resting on will prevent it from accelerating too much. Check that it's rotating in the correct sense (with positive velocity along $\underline{3^B}$).

4.3 Single axis attitude control

Mount the vehicle on the rotational rig, as shown in Figure 4.2, using two screws as you had in Lab 2. The vehicle now has only a single degree of freedom, the pitch angle, and we can attempt to control it without fear of instability causing damage to the vehicle (or hurting ourselves).

We will now create a feedback controller to track a desired pitch angle. This will take the following form:



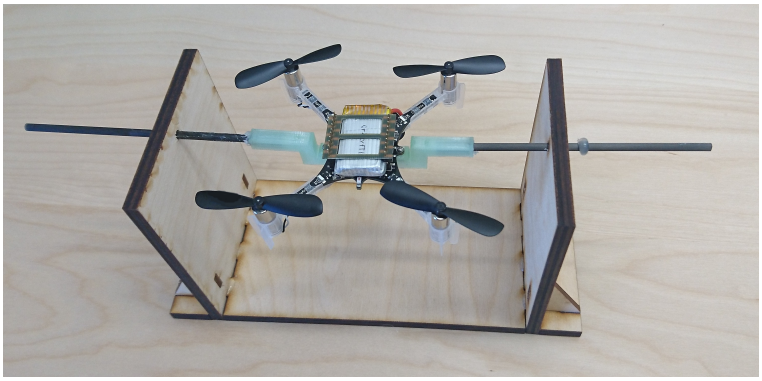


Figure 4.2: Rotational rig that restricts the vehicle to a single degree of freedom.

Note the cascade configuration, with the outer controller generating inputs for the inner controller.

4.3.1 Rates control

As a first step, we will implement a rates controller, that tracks a desired angular velocity by commanding an angular acceleration. This angular acceleration is then fed through the pipeline of functions we have created, and eventually converted to PWM commands for the motors.

We will design three separate controllers, one for each axis of the body. Per controller, there will be one parameter to choose, the time constant. Because of symmetry, we will choose the same time constant for the pitch and roll axes, and a different time constant for the yaw axis.

Recall the body angular velocity, expressed in the body-fixed frame, is $\left[\underline{\omega^{BI}}\right]^B = (p, q, r)$. For each axis, we assume the existence of a desired angular velocity, which we'll write as p_{des}, q_{des}, and r_{des}. We generate an angular acceleration command, $\dot{p}_{\text{cmd}}$ as follows:

$$\dot{p}_{\text{cmd}} = -\frac{1}{t_p}\left(p - p_{\text{des}}\right) \tag{4.1}$$

with t_p the time constant (our controller gain), which has units seconds. We will use the bias-corrected rate gyroscope measurement for p.

Introduce three new constants, as follows:

```
//in MainLoop():
float const timeConstant_rollRate = 0.04f; //[s]
float const timeConstant_pitchRate = timeConstant_rollRate;
float const timeConstant_yawRate = 0.5f; //[s]
```

and then use these constants to convert a desired angular velocity to a commanded angular acceleration. For now, just set the desired angular velocity to zero. Notice that the code above includes a first set of controller gains – we will now try out the performance of the control.

Now that you have implemented a controller, try it out! Set a constant desired normalized thrust at $8\,\mathrm{m\,s^{-2}}$, and set the desired angular velocity to zero. Send the commanded angular acceleration back over telemetry, by using the 'telemetry outputs' in the outputs variable. You can do this as follows, where we assume that your command angular acceleration is a `Vec3f` named `cmdAngAcc`:

```
//after you've computed the command angular acc:
outVals.telemetryOutputs_plusMinus100[3] = cmdAngAcc.x;
outVals.telemetryOutputs_plusMinus100[4] = cmdAngAcc.y;
outVals.telemetryOutputs_plusMinus100[5] = cmdAngAcc.z;
```

Flash the code onto the vehicle (which you mounted on the rotational rig). Remove the USB cable, so that the vehicle is free to rotate, and then run the controller. When you run the controller, you should see the propellers turning, and the vehicle should act to slow down any rotation. Try nudging the vehicle with your fingers, and confirm that the controller performs as you expect. At this point, the vehicle should have no preference for any specific orientation (since the controller isn't using the angle information yet). Note, however, that manufacturing imperfections in the rotational rig may cause a center of mass error, which would cause the vehicle to settle into some specific orientation.

As you move on to the next part, plug the USB cable in again, so that the battery can charge.

4.3.2 Angle control

Next, we will close the loop on the angles as well. We will use a cascaded controller, where we assume that the angles are controlled by directly setting the angular velocity – this requires that our angular velocity control is 'sufficiently faster' than the angle control.

We will use the following form:

$$p_{\text{cmd}} = -\frac{1}{t_\theta}\left(\hat{\theta} - \theta_{\text{des}}\right) \tag{4.2}$$

where t_θ is the time constant, $\hat{\theta}$ is our estimate of the angle, and θ_{des} is the desired angle.

Create, again, new constants for this controller:

```
//in MainLoop():
float const timeConstant_rollAngle = 0.4f; //[s]
float const timeConstant_pitchAngle = timeConstant_rollAngle;
float const timeConstant_yawAngle = 1.0f; //[s]
```

and then implement it to create the desired angular velocity that your rates controller used. Add the resulting commanded angular velocity to the telemetry outputs as well:

```
//after you've computed the command angular velocity:
outVals.telemetryOutputs_plusMinus100[6] = cmdAngVel.x;
outVals.telemetryOutputs_plusMinus100[7] = cmdAngVel.y;
outVals.telemetryOutputs_plusMinus100[8] = cmdAngVel.z;
```

Set the desired angles all to zero, and the commanded normalized thrust to $8\,\mathrm{m\,s^{-2}}$, as before. Flash the code onto the vehicle.

Turn on the vehicle, and observe what happens. Does the vehicle level itself out? Try nudging the vehicle a little – do you see the vehicle return to zero angle?

Note that, again, manufacturing imperfections may cause a center of mass error when using the rig. This would cause a torque, which will cause the vehicle to settle at some non-zero orientation.

4.4 Characterization experiments

You will now conduct a series of experiments to characterize the system. You will be required to analyze log files for each of these, so make sure to label your log files clearly, and take notes for each experiment. We recommend that you start "RadioAndJoystick" afresh for each experiment, so that you have one log file per experiment. Immediately after completing the experiment, kill "RadioAndJoystick" (by hitting `<ctrl>+c` twice), and rename the log file to something you will recognize later (and ideally, copy it into a separate folder so you don't accidentally delete it).

For these experiments, we want to see how well the vehicle can track commands. Specifically, we will have the vehicle execute an attitude maneuver where it switches from zero pitch angle, to an angle of 30° (≈ 0.5236rad). Specifically, implement in your MainLoop code that sets the desired pitch angle to 0.5236rad if the blue joystick button is pushed, and zero otherwise. Store the angle setpoint in `outVals.telemetryOutputs_plusMinus100[9]`, so that you can evaluate your tracking using the log file. Make sure that your attitude estimate is being output as well:

```
//send our attitude estimate back
outVals.telemetryOutputs_plusMinus100[0] = estAtt_roll;
outVals.telemetryOutputs_plusMinus100[1] = estAtt_pitch;
outVals.telemetryOutputs_plusMinus100[2] = estAtt_yaw;
```

4.4.1 Response to a step input

(See also Deliverable 3.) Set the desired normalized thrust at 8m/s^2, and flash the vehicle. Remove the USB cable, so that it can rotate freely. Let the vehicle run, and let it reach zero pitch angle. Now, hold down the blue button for approximately 2 seconds, so that the vehicle goes to the new setpoint, and then release the button to let the vehicle return. Do this a few times, so that you have a representative set of dynamic responses. Observe what happens – how did the response look?

Open the log file, and plot the estimated pitch angle against time. On the same plot, show the commanded angle, so you can compare the ability to track commands. Can you see where the command changed from 0 to 30°? Compute the time that the system requires to go from 0% to 90% of its final value. Because of friction and a likely center of mass error in the rig, the system is unlikely to start at 0 and end at the desired 30° – compute the rise time by looking at when the system has covered 90% of the distance from its initial value when the setpoint changed to the final value at which it settles. How do you think the additional friction and disturbance torques influence the results here, compared to free flight?

4.4.2 Purposefully unstable rates control

(See also Deliverable 4.) We know that a stable controller can track the attitude. Now, we will see what a purposefully unstable control does. We will do this by flipping the sign of the rates controller, by replacing the time constant with a *negative* number:

```
float const timeConstant_rollRate = -0.04f; //NOTE SIGN!
```

Keep the normalized thrust at 8m/s^2. Flash this code onto the vehicle, observe what happens, and store the log file somewhere.

Note: the vehicle will disarm itself (and thus turn off the motors) if it detects excessive angular velocities.

4.4.3 Control at low thrusts

(See also Deliverable 5.) Restore the correct sign for the controllers. We will now again look at the response to a step input, but we will reduce the normalized thrust to $2\text{m}\,\text{s}^{-2}$.

Let the vehicle run again, and again let it change between 0 and 30° a few times, (with enough time to settle in between). Can you see a difference in behavior from before? Again compute the time that the system requires to go from 0% to 90% of its final settling value. How do these numbers compare to those that you got at 8m/s^2? Why is the response different?

4.4.4 Accidentally unstable control

(See also Deliverable 6.) Restore the normalized thrust to $8\text{m}\,\text{s}^{-2}$. We will now try to cause an instability in the system by making the angle controller more aggressive. Halve the angle control time constant, as below:

```
float const timeConstant_rollAngle = 0.4f*0.5f; //halved!
```

Flash the vehicle, and run the experiment. Keep halving the time constant and running the experiment – at what value of the time constant is the vehicle incapable of tracking a commanded attitude? Does the instability differ, qualitatively, from that observed in Sec. 4.4.2?

4.4.5 Clean up, and 3D control

Once you are done, restore the attitude gain so that the system is stable again:

```
float const timeConstant_rollAngle = 0.4f;
```

Remove the code that switches the pitch angle command, and set this to a constant zero (so that all three desired angles are zero). Ensure that the desired normalized thrust is $8\text{m}\,\text{s}^{-2}$.

Flash the vehicle, and test that it works on the rig. Next, remove the vehicle from the rig, and hold it in your hand. Let the motors run, and convince yourself that the attitude control is working – the vehicle should try to remain level as you tilt it in any direction. With the motors running, drop the vehicle from approx. 30cm onto the table – the attitude should remain flat throughout, and the vehicle should fall slowly (compared to free fall).

4.5 Deliverables

You must submit electronically, via email, a PDF report describing what you did in this lab exercise. The report should be typed up, e.g. using Latex or a word processor; it should be complete, i.e. each deliverable should be a separate section, which is self-contained. Comment on your results: what was surprising, what is unexplained, etc.

Besides being technically correct, your report must be clear and concise. We will apply a "signal-to-noise ratio" penalty, if we feel that your report is not clearly written, neat, and concise. Any code snippets that you submit must be neat and legible, with comments describing the code.

There may be some homework questions posted with the laboratory exercises. You should complete these, and append their solutions to the report. You do not need to type up the solutions to the homework questions, you can simply scan them in and append them to your report.

Lab reports are due electronically at 14:00 on the Wednesday two weeks after your group performed the lab, that is

- Groups 1-8: **14:00 on Wednesday 1 November**
- Groups 9-17: **14:00 on Wednesday 8 November**

Submit the reports with homework as a single PDF for your group, via email, making sure that you complete this checklist:

- Attach: single PDF (which contains group ID, and all member names)
- To: `me136_gsi@berkeley.edu`
- Subject (replace "XX" with your group ID): `[ME136 lab report] Group XX`
- CC: Put all group members in CC.

Late submissions: you will lose 20% if your submission is late by less than eight hours. For each additional hour, you lose an additional 10%.

For your report:

1. Describe your experimental set-up (with photographs as necessary). [5%]
2. Provide the equations you used for the mixer, and give the code snippet where it is implemented. [5%]
3. For the step input response experiment: [10%]
   (a) Plot the time history of the vehicle's angle and desired angle against time.
   (b) Show a detail plot of a single step from 0 to 30°. On this plot, indicate the rise time.
4. For the purposefully unstable rates controller: [10%]
   (a) Plot the time history of the vehicle's angular velocity, and desired angular velocity, against time.
   (b) Compute the "size" of the instability, by estimating how long it takes the angular velocity to double (provide the time to go from 2 to 4, then 4 to 8, then 8 to 16, and then 16 to 32 $\mathrm{rad\,s^{-1}}$).
   (c) How does the time-to-double compare to what you would predict, using your knowledge of the implementation?
5. For the low thrusts experiment: [10%]
   (a) Plot the time history of the vehicle's angle and desired angle against time.
   (b) Show a detail plot of a the angles for a single step from 0 to 30°. On this plot, indicate the overshoot and rise time.
   (c) Show a detail plot of the motor commands for the same time sequence as the above angles.
   (d) Describe how, and why, the response is different than when we used $8\mathrm{m\,s^{-2}}$.
6. For the accidentally unstable control: [10%]
   (a) Plot the time history of the vehicle's angle and desired angle against time.
   (b) At what value of the time constant did the instability occur? Why would the system become unstable as you make the controller more aggressive? What lesson can we take away from this?
   (c) What do you expect the influence of the friction in the rig was, for the instability? Does it make the system more, or less, prone to instability?
   (d) In terms of the behaviour of the system, how does this instability differ from the one you purposefully triggered?

7. Provide the full listing of your `MainLoop` function after you restore the code to the stable controller. Also provide any other functions you introduced. Make sure that everything is well commented.

8. Complete the Lab 4 homework questions, posted separately on the class webpage [50%]