Quadrotor Dynamic Model: Propeller Gyroscopic Effect.

I copied the equations below from the last post, where we see we matched our Bouabdallah paper's equations for rotational motion. We know some propeller input is going to be a part of the applied torque. There is going to be a set of equations for linear motion too, but let's clarify what is going on here regarding torques about our 3 body axes. Let's look at these equations and…

Review Rigid body rotation gyroscopic effect

$$\sum T_x = I_x\dot{\omega}_x + (I_z - Iy)\omega_y\omega_z$$
$$\sum T_y = I_x\dot{\omega}_y + (I_x - Iz)\omega_x\omega_z$$
$$\sum T_z = I_x\dot{\omega}_z + (I_y - Ix)\omega_x\omega_y$$

Remember Newton's Second Law $\sum F_i = m \cdot a_i$? This tells us that the sum of the forces on a mass in the i'th direction must equal the mass of the body times the acceleration in that direction.

If we take one of the equations above and rewrite it we can recognize this familiar form, albeit in rotational nomenclature. The rightmost term is the gyroscopic effect of the entire rigid body. It tells us that if there is rotational rate about y- and z-axes a torque is induced about the X-axis. If we neglect this term you see we'd $\sum T_x = I_x\dot{\omega}_x$t with in simple second-law form.

It's a little confusing, but all we've done is already offer one item to the, "sum-of-torques": the gyroscopic effect resulting from the rigid body rotation. Notice when we moved

the term in parenthesis to the other side it looks like an added torque now but we swapped the subtraction in parenthesis so the math is the same.

$$I_x\dot{\omega}_x = \sum T_x + (I_y - I_z)\omega_y\omega_z$$

Gyroscopic effect of Propellers rotation.

“tail-dragger” airplane example

I made a video below to describe a phenomenon familiar to pilots, particularly pilots of powerful, “tail draggers” like the P-51 mustang(below). A propeller is a large spinning disk. when we, “tip” the plane of rotation of the propeller: pitch the airplane, a torque on the body results due to this gyroscopic effect.

Imagine you are in the cockpit of the P-51, speeding down the runway (lucky you). That monster prop has a healthy angular momentum vector pointing perpendicular to the prop. Your tail is dragging until you get enough lift from your wings (due to streamline curvature, if you recall a recent post!). At some point your tail will lift quickly and you’ll be parallel to the ground before your front wheels leave the ground. This

looks like a quick, “pitch” of say 15-degrees based on the photo above. Let’s say it took a second for the tail to rise as we’re nearing take-off, so the pitch rate is 15-degrees per-second as the tail wheel leaves the ground.

That same pitch is pitching the, “plane” of the spinning propeller forward, or more to the vertical. When you watch the video below and inspect the equations below you will see that this, “pitch forward” of the propeller at the above angular rate of the airplane going from dragging-to-level results in a, “gyroscopic torque” about the airplane’s vertical axis: you feel the plane want to, “yaw” or twist so you’ll need to correct with some rudder.

The effect can be negligible

Our propellers are horizontal in the body-plane, but there is a torque induced on the body that results from propeller rotation and rotation of the propeller plane about the x- and y-axis of our system too…at least in theory. It may be negligible depending on propeller moment of inertia relative to other effects, but let’s model it anyway.

We can choose to include it or neglect it later when we perform a sensitivity analysis. This is where we can look at rough magnitude of various effects and decide to include them in our model or cancel the terms in the equations: ignore them as too small to make a difference. We’ll do this systematically and not guess at this point. Besides, this is a neat phenomenon for us to appreciate in detail.

Video Explaining Equation Derivation

Gyroscopic effect of our propellers

The following PDF derivations arrive at the same terms as the Bouabdallah paper (equation 8). There's a sign convention difference between the paper and my derivations, but the result is the same:

$$T_{x,prop} = I_{prop} \cdot \dot{\phi}(\omega_2 + \omega_4 - \omega_1 - \omega_3)$$
$$T_{y,prop} = I_{prop} \cdot \dot{\theta}(\omega_1 + \omega_3 - \omega_2 - \omega_4)$$

The first equation tells us that there will be a torque about our body X-axis when there is angular rate-of-change to our pitch: $\dot{\phi}$. This means our quadcopter will feel a body roll-inducing torque when it pitches. This is due to the gyroscopic effects of the 4 propellers.

The second equation tells us that there will be a torque about our body Y-axis when there is angular rate-of-change to our roll: $\dot{\theta}$. This means our quadcopter will feel a body pitch-inducing torque when it rolls. Again, this is due to the gyroscopic effects of the 4 propellers.

The hand calculations below arrive at the above equations following the process in the video. The video above explains how to go about the hand calculations for a single propeller.

Hand Calculation PDF Notes

RB 2018-07-11 17.27.01

Conclusion

Above we have one more term for our sum-of-torques about the body X- and Y-axes. The plus-minus signs represent our set-up with two clockwise spinning propellers and two counter-clockwise spinning propellers. If you expand these equations you'll see I just performed the steps in the above video Eight times and combined the equations. We need eight steps because we have 4 propellers and we need to account for the plane-of-rotation rate-of-change about both the x- and y-axis. Rotation about the body z-axis contributes no gyroscopic effect from the propellers because it is just like twisting the axis of the propeller. The plane-of-rotation of the propeller does not change at any rate in this case so no torque is induced on the body.

$$T_{x,prop} = I_{prop} \cdot \dot{\phi}(\omega_2 + \omega_4 - \omega_1 - \omega_3)$$
$$T_{y,prop} = I_{prop} \cdot \dot{\theta}(\omega_1 + \omega_3 - \omega_2 - \omega_4)$$

We've now accounted for gyroscopic effect of the rigid body rotations and the propeller rotations.

Next we'll add the propeller thrust components. These will be our inputs to counter-act the above effects and unknown disturbances (wind) in order to maintain the attitude of our quadcoptor and move it where we want it to go.

Simulation Methods: Double Integrator Example

In the last post I focused on placing the lead zero for the roll and pitch axes based on the limit imposed by a second

double-pole our plant introduces via the motor-propeller, ‘A’ term. I neglected to calculate the proportional gain required for unity-gain crossover at the frequency of maximum phase margin. I also did not design the yaw-axis controller. The first short video completes these steps from last time. The rest of this post will cover some simulation tools and techniques we’ll be using.

Simulation tools and techniques

In the video you can see in the last post I mention the, “satellite attitude” control problem you can find in nearly all control system textbooks. This is basically the same, “plant” model as our roll, pitch, and yaw axes. However, by stripping away our extra poles and parameter values we can first make sense of the simulation tools and methods before applying the tools to our problem.

Satellite Attitude Control: classic double-integrator plant stabilization

Is space absent gravity or any impeding forces lateral thrust a moment-arms distance from the mass center on opposite of a body is a, “couple”. It produces angular acceleration of the body when the moment-arm from each thruster is the same length from the mass center. The simplified model models the rotational thrusters as the only external torques on the body.

This torque results in angular acceleration that double-integrates to angular position. Hence the term, “double

integrator” I suppose, but I don’t know who coined the phrase. In any case, this is a classic, “unstable plant” control problem. Let’s use this simple problem to set-up some simulation tools we will use for the quad-copter control simulations.

Textbook Problem Statement

The following PDF sets-up the basic problem as described in Digital Control of Dynamics Systems by Franklen, Powell, and Workman but you can likely find it in any control systems textbook.

fpw_satellite_attitude

Matlab Simulation Techniques

Matlab software is the go-to tool for control system design and simulation. You can see I use a neat math software tool called Maple as well, but this is fairly expensive and less common. Maple is awesome for creating mathematical, “documents” with in-line calculations as you have seen to this point.

I could probably push Maple to do a bunch more, but then I’d be producing example code that most of you couldn’t use. I’ll be using Matlab quite a bit now, so I can share M-files (the Matlab scripts) with you so you can play with them on your own.

A companion to Matlab is Simulink, which is useful for simulations. I chose the scripted ordinary differential equation (ODE) solver method in a loop instead. It is less intuitive than Simulink, but will offer more flexibility as we get into the Quadrotor simulations. Under-the-hood Simulink uses the same ODE solver, but the, “graphical programming” can become a burden for all but the simplest simulations.

Video of simulation output

This video shows a space capsule initially pointing Up, but it’s assumed to be in outer space in a zero-gravity environment. When the simulation begins the capsule thrusters (illustrated in red) respond to a step-change in desired capsule orientation of 180-degrees. The simulated control system thrusts the capsule to the new desired orientation.

An actual, “satellite orientation” controller would be much more optimal relative to fuel burn. For example, fuel wouldn’t be wasted allowing the overshoot you see here. The burn trajectory would be pre-determined somewhat like an, ‘S’ to minimize the burn, and the servo method illustrated here would be left to operate in a very small window for final adjustment most likely. Perhaps we will return to this simple model if we explore optimal control topics later.

For now, this video shows what the Matlab code below does. If you run the Matlab M-file you should see what you see in this video, and from there you can modify the code as you learn.

Matlab M-Files

Download the following two files, place them in the same folder, and run the, “satellite” script. You’ll see the above plot output, and you can experiment from there by reading the comments in the script.

Main simulation script: satellite.m

The ODE function: ssodefun.m

Closing Remarks

The goal here was to frame-up some simulation and animation tools using Matlab. I used a simple, single-axis double-integrator, “textbook problem” to get the tools set-up.

From here I can proceed to introduce the multi-axis complexity of the Quadrotor with it’s extra poles and parameters. That’s next!

Quadrotor Control: State-Space Model

I covered, “PID” (Proportional-Integral-Differential) or, “classical” controller designs for the quadrotor platform in a post last fall…time flies! We really only employ the P and the D elements. The, ‘D’ is the, “lead compensator”. The proportional gain P is the last step and you can see how this design technique is performed in that post.

This is a fine method and I’m guessing it is the extent of the controller design for most quadrotor platforms you buy, and code bases you might use if you buy a controller unit like a Pixhawk and run Ardupilot on it. An old colleague of mine is a leader on Ardupilot project. He walked me through the code a few months ago.

If I were anxious to fly I’d be pulling parts together and droning around my neighborhood with the Pixhawk, an off-the-

shelf quadcopter, and Arupilot all tuned-up as they have designed a user-friendly set of tools: PID tuning, etc. I could implement my controller based on my recent lead compensator design.

A lead compensator design (“PID tuned”) in Ardupilot will get me a stable quad. I want to push on to the challenge Bouabdallah presents in his paper: control of the platform over a wider, “flight envelope”. This means when we can’t ignore the gyroscopic effects of the rigid body in particular, and perhaps the propellers. In this case we can’t simplify the equations into a friendly transfer function and use, “classical” design techniques. We need to use, “state-space” techniques.

State-Space Design Techniques

A tendency here is to just form some matrices instead of transfer functions, assume full-state feedback or implement an, “observer”, and use Matlab commands to calculate some gains. We could do that, and it’s what you do in a control systems class quite often.

I never got deeply engaged in the technique largely because it is perhaps overkill for many practical designs for land vehicles and industrial process control. I learned the design steps, but I took a lot of the math for granted as I used Matlab. I never studied or pursued multi-axis, multi-input control design for aircraft and such. Even there often times good modelling and design simplification can get you to transfer functions and classical design methods.

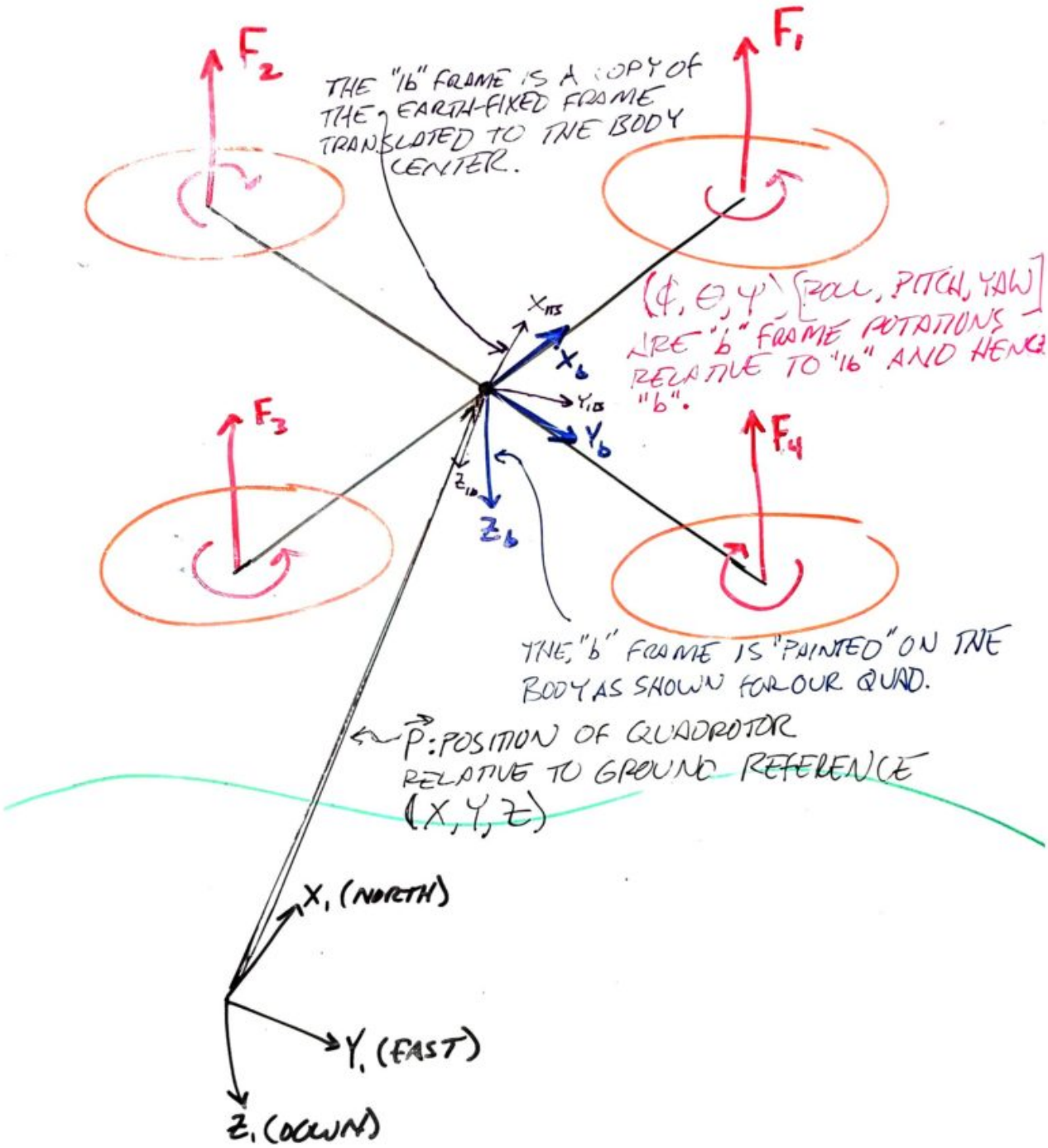
Blakelock’s book provides many examples of classical design to aircraft and missile control problems. “State-Space” techniques have also been coined with the term, “modern” which implies the old tried-and-true classical methods are old-fashioned, but this is a misnomer. The best solution for any

problem is the simplest model you can form, and the method you are most comfortable with to achieve confidence in the results that meet your requirements.

In Bouabdallah's paper we see a reason for state-space design when we don't ignore body moment-of-inertia and propeller gyroscopic effects. We model the system we'll successively linearize instead of a transfer function that's valid near a hovering, horizontal state.

We're going to attempt controller design that we might use to track a moving target. We're building a bird of prey to hunt on-the-wing, not just a bird that wants to hover around looking for a mouse! Let's do this with a drive for deep understanding though, and not just jump to Matlab's, "LQR" command for some gains we used other geniuses' math to calculate. I won't approach their level of genius, but I want to learn as much as I can so here we go…

Platform Sketch



Equations of Motion, Revisited

Refer to an earlier post to review how body moment-of-inertia term appears in the equations of motion for the platform

Propeller Gyroscopic Effect

Refer to an earlier post to see how this term is derived.

Propeller Thrust Inputs

Refer to an earlier post for details.

Quadrotor State Model

The following model copies the Bouabdallah paper. It took some time and effort to arrive at his model through our step-by-step derivations and learning.

state_model

Up Next…

“Linear Quadratic Methods”: throughout the, “flight envelope” of roll- and pitch-rates that are needed for the, ‘A’ and, ‘B’ matrices we’ll need a, “control law”. that is, we’ll need a feedback gain matrix to either..

- Regulate to, “zero” state values (steady, horizontal).
    - This means we’ll implement a, “Linear Quadratic Regulator” (LQR).
- “Track” to a desired platform attitude, away from horizontal
    - This is a linear tracking problem…a twist on the LQR problem.

The next post will get into these details!

Quadrotor Linear Quadratic Regulator (LQR)

Big gap since the last post where we finally got the state-space model laid down. It got us to the plant model derived by Bouabdallah and others in his paper that we've used as a guide from the start.

The goal all along has been not only to analyze and design candidate controllers for a Quadrotor platform, but to take forays into applied math, mechanics, and advanced techniques that we might not need for a camera hover drone but which give us a chance to stretch our skills.

A typical quadrotor controller decouples the roll, pitch, and yaw axes and implements PID controllers for each axis as described in our post from awhile back. We know this works, and we could jump into Ardupilot source code for say a PixHawk Cube and see exactly where and how PID controllers are implemented in source code.

Instead of getting into an implementation let's push on to learning Linear Quadratic control techniques for a Multi-Input, Multi-Output (MIMO) system.

Why State-Space Model and LQ Control?

The axis decoupling created Single-Input, Single-output (SISO) systems out of the Roll, Pitch, and Yaw axes and expects each axis to work independent of the other via PID control, or more specifically via a lead compensator as illustrated earlier. This is an adequate, in most cases a good, robust design so why am I making this more complicated? It might be bad engineering practice for a simple camera drone, but as we are not designing a specific drone just yet, we're keeping some extra terms in our design. We're working on the general case still: a wide flight envelope away from small-signal hover response.

We're keeping body inertial effects and propeller gyroscopic effects in the model. These null-out for roll-, pitch- and yaw-rates near zero, at which point the state-space A and B matrices simplify to what are basically 3 independent axes of control, a situation where our Lead compensator applies. By not assuming these terms negligible we retain cross-axis coupling and can't treat the platform as a set of three independent axes controlled separately from one another.

The classical design method is a bit more intuitive compared to an LQ optimal approach where we pick relative weights for state convergence cost and control cost (more on this later). Recall the insight we gained about plant dynamics and dominant poles that limit plant response: body and prop moments of inertia. This is intuitively obvious: large, heavy rig and slow prop response limits ability to accelerate body rotations. We could see these factors in Bode plots and scale our crossover accordingly. The math and our intuition were well aligned.

When we compute a linear quadratic gain matrix we'll be

pumping our intent through the calculus of variations and some formulae that seem like magic. We'll compute, "optimal" gains according to a cost function. We'll gain some insight by varying our design parameters but it will be an iterative process relying on comparative analysis of simulations. It won't be as prescriptive as frequency design methods or pole placement in general.

Also, "optimal" is a bit of a misnomer. An, "optimal" gain is only so relative to a particular combination of state and control weights. It's not, "optimal" in an engineering sense, only in a mathematical sense for the weighting parameters we supply.

Starting Point: Full-State Feedback Regulator

We'll eventually add roll, pitch, and yaw reference inputs that a flight controller outer loop will command. These will be the reference inputs to this inner platform attitude control loop that we are familiar with from our classical control designs. However, to get started let's implement a simple full-state feedback regulator. This will get our design and simulation tools in order before we introduce reference inputs.

Below is the system model: full-state feedback to gain K with state transition matrix A, and input matrix B. Another simplification here is the availability of state measurements for feedback. We can expect to have measurements from gyros and accelerometers but these will feed a state estimator that will in-turn supply the full-state feedback in our loop. However, state estimation is another topic, and it can be decoupled from the controller design. We'll cover it later.

When we ultimately add reference inputs and state estimation to this diagram it will represent a design for implementation,

but in diagram form it will look more intimidating than this picture. Let’s learn from this first, and build on it.

Our state vector, ‘x’ contains roll, roll rate, pitch, pitch rate, yaw, and yaw rate, respectively:

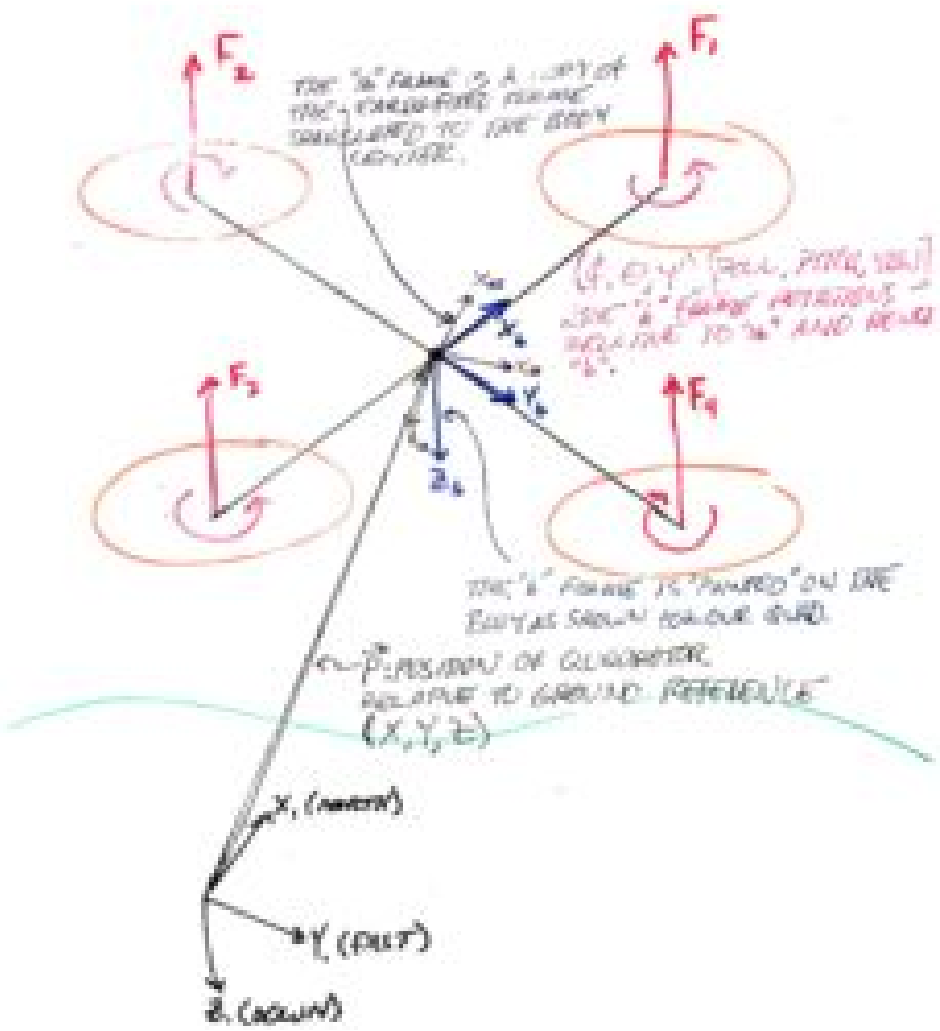
$$x = [\phi \ \dot{\phi} \ \theta \ \dot{\theta} \ \psi \ \dot{\psi}]^T$$

With…

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\dot{\psi}}{2} \cdot \left(\frac{I_y - I_z}{I_x} \right) & 0 & \frac{\dot{\theta}}{2} \cdot \left(\frac{I_y - I_z}{I_x} \right) \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{\dot{\psi}}{2} \cdot \left(\frac{I_z - I_x}{I_y} \right) & 0 & 0 & 0 & \frac{\dot{\phi}}{2} \cdot \left(\frac{I_z - I_x}{I_y} \right) \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & \frac{\dot{\theta}}{2} \cdot \left(\frac{I_x - I_y}{I_z} \right) & 0 & \frac{\dot{\phi}}{2} \cdot \left(\frac{I_x - I_y}{I_z} \right) & 0 & 0 \end{bmatrix}$$

$$B=\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{l}{I_x} & 0 & 0 & \frac{J}{I_x}\cdot\dot{\theta} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{l}{I_y} & 0 & \frac{J}{I_y}\cdot\dot{\phi} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{I_z} & 0 \end{bmatrix}$$

and referring to our body model sketch once again…



Feedback Gain ‘K’

This is will be an optimal control solution. In this post we will rely on Matlab’s LQR() function. You’ll see the call in the Matlab script later but first,

There are States in the Transition and Control Matrices, Why?

As we recall from the last post and can see above, the A and B matrices have the three rate states within. Had we not employed Bouabdallah’s trick to add the rates as states themselves we’d have a product of rates in elements of A. We eliminated this non-linearity, but we still have states in A. This is not invariant throughout flight.

Options…

1. Model small-signal near-hover control: zero the rate states in the A matrix
    1. This is basically what we did for the classical design methods. It permits decoupling the axes, at which point this LQR method is less intuitive that 3 decoupled control loops designed as described previously
2. Recalculate Control Law K over the flight envelope: successive linearization.
    1. To the furthest extent this involves taking output rate states as inputs to A and B which are then inputs to a solution of the Matrix Ricatti equation resulting in gain matrix K.

Given we’ve already pursued Step #1 in our classical design approach, let’s push ahead with approach #2

Successive Linearization of the Plant Model: A and B Matrices

It would be impractical to recompute feedback gain K as a function of revised A and B matrices for each new sample (estimate) for $[\dot{\phi}\ \dot{\theta}\ \dot{\psi}]$ in our control loop calculations. Our platform control loop would be solving the matrix Ricatti equation in real-time.

A practical design could be a three-index look-up table that covers the range of these body axis rotational rates with a reasonably granular step size. This look-up table could contain pre-computed gain matrix K corresponding to the lookup value indexed by current $[\dot{\phi}\ \dot{\theta}\ \dot{\psi}]$. Let's assume something along these lines as a design direction later.

For the simulation that follows we employ Matlab's LQR function in a loop and we successively compute control gain matrix K. This allows us to visualize the regulator process using the power of the simulation tools before adding complications: reference inputs, gain scheduling from look-up table, and other practical details.

The Control Law: Computing K

Our objective in this exercise is to regulate to zero state-vector X governed by differential equation

$$\dot{x} = Ax + Bu$$

To minimize performance measure

$$J = \frac{1}{2}\int_{t_0}^{t_f}[x^T(t)\,Q\,x^T(t) + u^T\,R\,u^T]dt$$

The performance measure employs two matrices Q and R to weight the importance of state convergence and control effort,

respectively. Derivation of the performance measure and how the plant differential equation and this performance measure combine to determine an, “optimal” controller gain K given A, B, Q, and R is a topic for a lengthy foray into optimal control theory and the calculus of variations.

We’re going to skip this for now and simply use Matlabs lqr() implementation to meet the stated objective and supply us with a gain matrix K. By first visualizing through simulations what we are achieving with these methods we’ll have perhaps a greater appreciation for the details when we dive into all the math behind Matlab’s “lqr()” function.

We will see the relative effects of Q and R on regulator performance. We’ll then move on to the tracking problem when we are not regulating states to zero, but attempting to track reference inputs for these states. An outer flight control loop will supply these references for tracking targets in flight, for example.

For now let’s accept the performance measure as a result of something called the, “second variation” in the calculus of variations, employed for neighboring-optimal solutions. The final time in our definite integral is also, “free” but we’ll cover that later

Regulator Simulation

This basic regulator simulation is the foundation for complexities we’ll add as we go: tracking reference inputs, state estimation, and introduction of actuator limits, plant uncertainties, and disturbances.

For now we want to get the simulation correct by observing ideal condition regulation to zero from initial conditions on roll, pitch, and yaw. The video below represents simulation

output from the Matlab script explained below.

http://www.mtwallets.com/wp-content/uploads/2020/10/quad_regulator_sim-1.mp4

Method

The plant is modelled in continuous time. An outer simulation loop is equivalent to sampling time of a physical system: the interval on which states would be sensed and/or estimated and updated commands actuated on an actual system.

In the simulation the system is the plant model A and B as coefficients of the state differential equations. On each iteration of the sampling loop the time interval is further divided into time steps over which Matlab's ode45 integrates the state differential equations.

Each invocation of ode45 takes as initial conditions for the states the last state output from the last invocation. Ode45 solves the system below on each invocation for a time vector that represents the sampling time interval further divided, computing a state estimate x for each step of that finer interval.

Command input, 'u' is constant for each invocation. It represents operating on feedback from the previous loop, applying the controller gain K, and issuing an update command, 'u' to the plant.

```
function xd = ssodefunc(t,x,a,b,c,u)
      xd = a*x + b*u;
end
```

After each invocation the full simulation time vector is appended with the new timesteps from 't' and the state estimates are appended to the full simulation state estimates.

Think of it as though we are kicking a continuous (analog) system on a time interval with updated commands, ‘u’. The analog nature of the plant is represented by the finer time steps for which Ode45 produces state estimates. All of these between-sampling-interval state estimates represent the actual motion of a physical system.

By taking the last Ode45 state solutions from each interval as initial conditions for which to use as feedback to control gain K and as initial conditions for the next iteration we are simulating as though we are, “sampling” (with sensors) on this interval, as if we only know the last solution from each invocation of Ode45.

By plotting every finer interval it is as though we are observing the behavior of the continuous-time, analog system under the control of our sampling-and-control system.

Simulation Details

The Matlab M-file below implements the simulation described above. I use a, “home use” license of MATLAB R2019b with the Control Systems toolbox and the Signal Processing toolbox although you likely need only the former. Mathworks, “home use” licensing is a great recent offer. For a couple hundred dollars you can add this to your tool set. In the past I benefitted from commercial licenses I had access to for work. It’s great to have a personal license. I highly recommend it.

Matlab Script

The plant parameter values are as derived in an earlier post where we implemented the classical lead compensators for roll, pitch, and yaw.

“tstep” is our control loop sampling interval. It has a basis

in the reality of our band-limited mechanical plant. Recall we closed the classical loops with unity gain crossover at 10Hz. If we imagine tracking a disturbance in this range, how fast do we need to sample? I remember some old design rules of 5 or greater times sampling frequency relative to loop crossover, so this implies we want to be in the neighborhood of 50-100Hz sampling frequency. This sounds about right.

If you play with this simulation with slow tsteps of ~0.1+ seconds you'll see the system go unstable. This is because we're sampling the plant too slow relative to the inherent dynamics. On the fast side we would run into computational and sensor limits, but it's reasonable to assume a 50-100Hz sampling rate for this platform control loop will be achievable in hardware.

Below you will see tstep=0.02s representing a 50Hz control loop. Imagine this would be the rate at which we would sample sensors, update state estimates, and issue revised commands to our motors. This might be a bit slow but it's fine for now. 50-100Hz is likely what a hardware platform will support. There will be other sampling rates within the system to, for example, acquire inertial sensor data and filter estimates for the platform control loop. These may run considerably faster, and we will, "sample" the output of these subsystems for estimated at our control loop sampling rate.

Download the M-file

Or review it here…

Conclusion

We introduced a simple multi-input, multi-output (MIMO) Linear Quadratic Regulator design and simulation here with full-state feedback. State feedback will ultimately come from a state estimator: a Kalman filter that will perform sensor

fusion from our gyroscopes, accelerometers, GNSS (GPS), compass, and barometric sensor on a real flight platform.

This state estimation will be a topic all it’s own. We’ll also later introduce reference inputs for pitch, roll, and yaw so we can control to setpoints and not simply regulate to zero values.

Until then, study and play with the m-file above if you can find yourself a Matlab seat!