MAE 3260 Final Group Work: Exploring a Quadcopter Report

Title: Functional Fidelity Flow Formulation Foray For Fourfold Fan Flight

Topic of Interest: Quadcopter

Abstract: We are going to attempt to model a quadcopter, and how it reacts to inputs from the controller and disturbances. We decided to study this system as we have a basic understanding of the system from MAE3230, and would like to learn about the system in detail. We will attempt to find ODEs that describe a simplified model of the system, and create block diagrams to model the system and observe how the inputs relate to the outputs (position). Using these, we will try and find out how to achieve smooth movement and steering. We will then attempt to make an animation of how it will behave in MATLAB.

Students/Roles:

Student	Task/Role	Portfolio
Catherine Guo	I contributed to deriving and linearizing ODEs, developing PD controllers, and designing inputs.	https://cornell-mae-ug.github.io/fa25-portfolio-CathGuo/
Joe Dalton	I worked on generating the MATLAB script along with drawing the block diagram for the pitch.	https://cornell-mae-ug.github.io/fa25-portfolio-JoeDa-125/
Kevin Sturm	I worked on researching different types of controllers and state space models and comparing their advantages / disadvantages.	https://cornell-mae-ug.github.io/fa25-portfolio-kevin-sturm/
Nuo Xiang Yan	I worked on identifying how the system parameters affect the resultant rotational motions using the MATLAB script, as well general explanations bridging the dynamics and consequences of them.	https://cornell-mae-ug.github.io/fa25-portfolio-ny227/

List of MAE 3260 concepts or skills used in this group work:

- Models:
  - ODEs
  - TFs
  - State space
  - Block diagrams
- Open-loop system:
  - Parameter estimation
  - Steady state behavior
  - Step response
  - Passive design
- Active control:
  - Feedback control law
  - Command following

Initial Research

We began by looking into existing models of quadcopters, and found a variety of sources, though they all tended to vary slightly in their approaches. Generally, we concluded that it was important for our model to be linear to stay within the scopes of this project. Additionally, as the system of a quadcopter requires 6 Degrees of Freedom (3 spatial and 3 rotational), we determined that our state space would have 12 entries, including every DoF and its derivative.

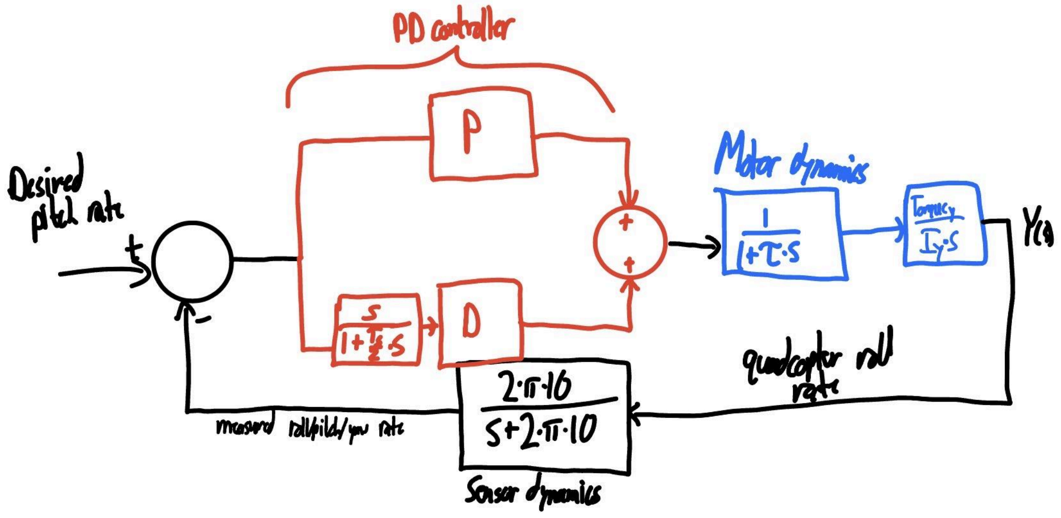
Another key element we determined from our research was the specific scope we would be looking into, and what behavior we wanted to model. We determined that it would be most appropriate to model the behavior of a step input from the controller, where we model how the quadcopter will approach the desired angle of flight, based on the user's input to the transmitter.

For simplicity of the analysis, we also chose to go with a PD controller. We determined that, while the PID controller would be better for disturbance rejection or precise motion, the PD controller was a more simple controller that would allow for a sufficient model to demonstrate step response and animation of the desired motion. LQR controllers were another option we looked into, but ultimately achieved the same effect as using a PID controller, as it would give an even better model in control theory, but also further complicate the system.

Steady State

For our model of a Quadcopter, steady state happens any time the Quadcopter is at a fixed angle. This includes when the Quadcopter is hovering, stationary in the air, or when it is positioned at a fixed angle, moving with constant velocity. Importantly, this means that any time our quadcopter has reached our desired angle with no more angular velocity, this means it is in steady state.

Block Diagram (for pic


PD controller
P
D
$\frac{s}{1+\frac{T_f}{2} s}$
+
+
Desired pitch rate
+
-
Motor dynamics
$\frac{1}{1+\tau \cdot s}$
$\frac{Torque_y}{I_y \cdot s}$
$Y(s)$
quadcopter roll rate
$\frac{2 \cdot \pi \cdot 10}{s+2 \cdot \pi \cdot 10}$
measured roll/pitch/yaw rate
Sensor dynamics

Above is a rough sketch of a block diagram for the pitch rate of our quadcopter. We used a PD controller for the design. If we were building out the full model for the quadcopter, we would also need block diagrams for roll, yaw, and vertical acceleration. The only thing that would change between these other block diagrams would be the P and D gains, and the quadcopter dynamics block seen on the far right of the diagram. This diagram also isn't entirely complete in that we have yet to plug in numerical values for things such as the settling time, time constant, and inertial characteristics of the quadcopter. It is also likely in real life that we would receive a disturbance torque to the system that is not currently modeled in this block diagram or in our ODEs.

ODE's

The quadcopter attitude dynamics are derived from two principles of rigid body. The Euler angle kinematics relate the time derivatives of the Euler angles to the body angular velocities through a nonlinear transformation matrix. This relationship is given by

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \frac{\sin\phi}{\cos\theta} & \frac{\cos\phi}{\cos\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \text{[1].}$$

The transformation matrix relates the body frame angular velocity vector to the time derivatives of the Euler angles (inertial frame), accounting for the fact that these two representations of rotational velocity are not identical except in special cases.

The angular acceleration is governed by the Euler's equations of motion for a rigid body, which state that

$$I_x\dot{p} = (I_y - I_z)qr + \tau_x$$
$$I_y\dot{q} = (I_z - I_x)pr + \tau_y$$
$$I_z\dot{r} = (I_x - I_y)pq + \tau_z \text{ [1].}$$

Here, I_x, I_y, and I_z represent the moments of inertia about the x, y, and z axes. The terms τ_x, τ_y, and τ_z represent the applied control torques generated by different motor speeds. The cross coupling terms represent gyroscopic effects that arise from the conservation of angular momentum.

The nonlinear equations described above are difficult to analyze using classical control techniques such as transfer functions. We linearize the dynamics around the equilibrium point for convenience. The equilibrium point is defined by $\phi = \theta = \psi = 0$ and p = q = r = 0. For small deviations from this equilibrium, we can apply small angle approximations where sin(x) ≈ x, cos(x) ≈ 1, tan(x) ≈ x for every angular position ϕ, θ, and ψ. Applying the approximation to the kinematic equations yielding the simplified relationships

$$\dot{\phi} \approx p,\ \dot{\theta} \approx q,\ \dot{\psi} \approx r.$$

Additionally, the gyroscopic coupling terms in Euler's equations involve products of angular rates, which will become negligible (compared to first-order terms) when angular rates are small. Therefore we can neglect these terms in our linearized model

$$\dot{p} \approx \frac{\tau_x}{I_x},\ \dot{q} \approx \frac{\tau_y}{I_y},\ \dot{r} \approx \frac{\tau_z}{I_z}.$$

State Space

Our state space has 12 elements, including 3 positions, 3 angles, and the velocity of each of those terms. This gives us the state space below.

$$X = \begin{bmatrix} X \\ Y \\ Z \\ \theta \\ \varphi \\ \psi \\ \dot{X} \\ \dot{Y} \\ \dot{Z} \\ \dot{\theta} \\ \dot{\varphi} \\ \dot{\psi} \end{bmatrix} \qquad \dot{X} = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \\ \dot{\theta} \\ \dot{\varphi} \\ \dot{\psi} \\ \ddot{X} \\ \ddot{Y} \\ \ddot{Z} \\ \ddot{\theta} \\ \ddot{\varphi} \\ \ddot{\psi} \end{bmatrix}$$

Control

We implemented a PD controller for attitude stabilization. The controller consists of three independent single input single output loops, one for each rotational axis. This decoupled approach is valid within the linearized regime where cross coupling effects are negligible. Each control loop measures the current angle and angular velocity, compares the angle to its commanded value, and generates a corrective torque based on both the error and the rate of change. This feedback loop forms the foundation of the attitude control system.

The control torques are computed using the law

$$u(t) = K_P e(t) - K_d y(t),$$

where e represents the angle error vector,

$$\begin{bmatrix} r_\phi - \phi \\ r_\theta - \theta \\ r_\psi - \psi \end{bmatrix},$$

y represents the angular velocity vector,

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix},$$

K_P is a diagonal matrix of proportional gains,

$$\begin{bmatrix} K_{P\phi} & 0 & 0 \\ 0 & K_{P\theta} & 0 \\ 0 & 0 & K_{P\psi} \end{bmatrix},$$

and K_d is a diagonal matrix of derivative gains,

$$\begin{bmatrix} K_{d\phi} & 0 & 0 \\ 0 & K_{d\theta} & 0 \\ 0 & 0 & K_{d\psi} \end{bmatrix}.$$

The proportional term generates torque proportional to the angle error, providing a restoring force that drives the system toward the commanded angle. The larger the error, the stronger the corrective torque. The derivative term generates torque proportional to angular velocity, providing damping by opposing rotational motion. The damping is for reducing overshoot and oscillations that would occur with proportional control alone.

Substituting the control law into the linearized dynamics gives the closed-loop system equations.

$$\ddot{\phi} = (K_{P\phi} e_{\phi} - K_{d\phi} \dot{\phi}) I_x$$
$$\rightarrow I_x \ddot{\phi} + K_{d\phi} \dot{\phi} + K_{P\phi} \phi = K_{P\phi} r_{\phi}$$
$$\ddot{\theta} = (K_{P\theta} e_{\theta} - K_{d\theta} \dot{\theta}) I_y$$
$$\rightarrow I_y \ddot{\theta} + K_{d\theta} \dot{\theta} + K_{P\theta} \theta = K_{P\theta} r_{\theta}$$
$$\ddot{\psi} = (K_{P\psi} e_{\psi} - K_{d\psi} \dot{\psi}) I_z$$
$$\rightarrow I_z \ddot{\psi} + K_{d\psi} \dot{\psi} + K_{P\psi} \psi = K_{P\psi} r_{\psi}$$

$$e = r - y$$

A hard step input, while mathematically simple and providing the fastest theoretical response, creates several practical problems in real-world quadcopter operation. The instantaneous discontinuity produces an infinite jerk (derivative of acceleration), which causes severe mechanical stress on motors. The aggressive response also could produce significant overshoot, which may violate safety constraints in confined spaces. To address the fundamental limitations, we developed and compared exponential command input patterns that provide continuous trajectories with controlled acceleration and jerk characteristics.

The hard step input $r(t) = \phi u(t)$ serves as our baseline for comparison. The exponential approach input looks like $r(t) = \phi(1\text{-}e^{-t/\tau})$, where time constant τ selected such that trajectory reaches 99% of the target at time T ($T=5\tau$).

We used parameters of $\omega_n = 20$ rad/s, $\zeta = 0.5$, $\phi = 30° = 0.524$ rad for the analytical predictions. For hard step input, rise time (10% to 90%) is approximately

$$t_r = \frac{1.8}{\omega_n} = \frac{1.8}{20} = 0.09\ seconds,$$

percent overshoot of

$$M_O = 100e^{\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}} = 100e^{\frac{-\pi 0.5}{\sqrt{1-0.25}}} = 16.3\%,$$

and the settling time of

$$t_s = \frac{4}{\zeta\omega_n} = \frac{4}{0.5*20} = 0.4\ seconds.$$

For exponential input, we select time constant (0.434 seconds) such that the command reaches 99% of target at t = 2 seconds ($0.99=1\text{-}e^{-2/\tau}$). This lead to rise time (10% to 90%) approximately

$$t_r =- \tau \ln(1 - 0.9) =- 0.434 \ln(1 - 0.9) = 1\ second,$$

percent overshoot of

$$M_O = 0\%,$$

and the settling time of

$$t_{98\%} =- \tau \ln(1 - 0.98) =- 0.434 \ln(1 - 0.98) = 1.7\ seconds.$$

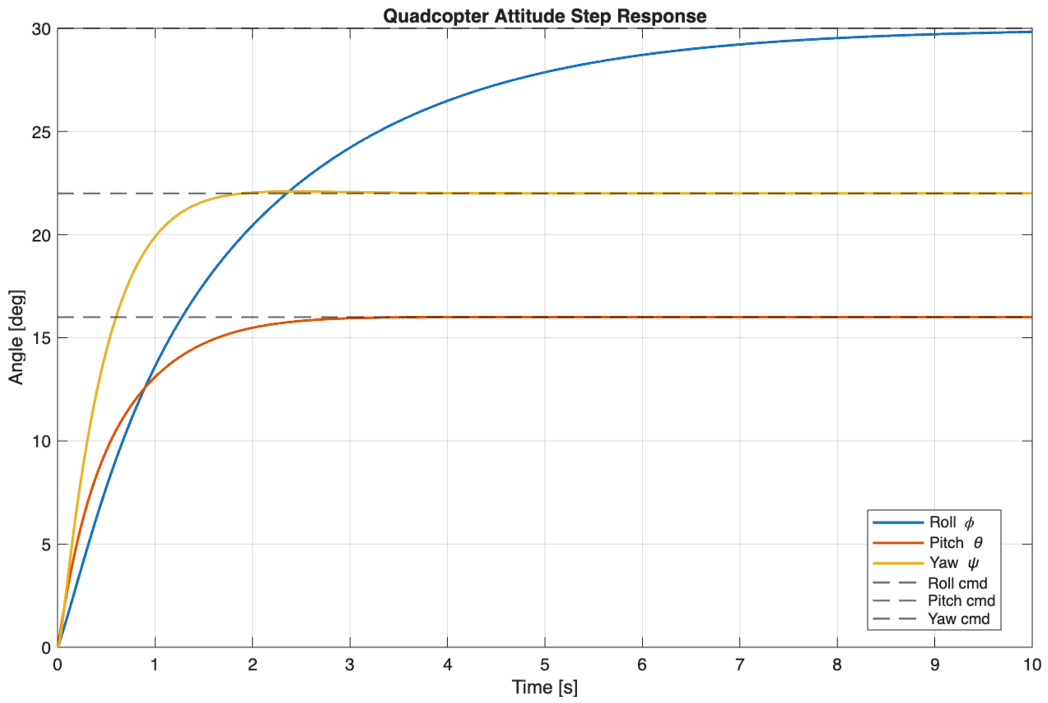
The predictions reveal clear trade-offs between the two approaches. The hard step input provides an extremely faster response with a rise time 11.5 times faster than the exponential approach. However, this speed comes at the cost of 16.8% overshoot, which could violate safety constraints. The settling time advantage is less dramatic, with the step settling only ~4 times faster. The exponential input sacrifices speed for smoothness and predictability. The complete elimination of overshoot makes it suitable for applications with strict safety margins. The slower rise time is acceptable for most nonemergency tasks.

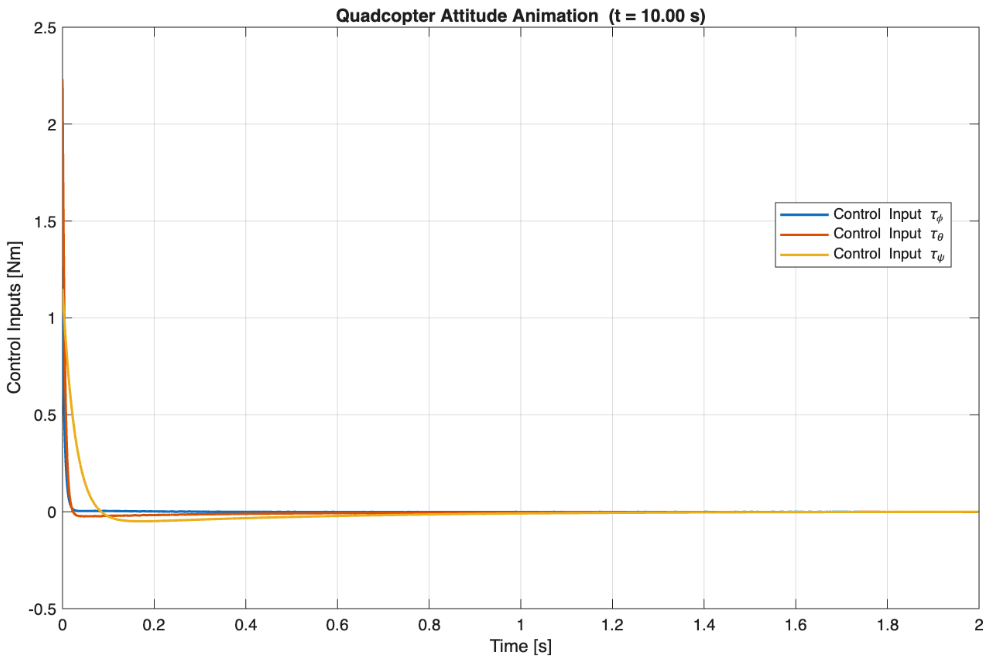
Matlab Script and Animation

We also decided that we wanted to make an animation of how our model behaves. In order to assist in creating the animation, we used ChatGPT to help us write the code. In order to do this, we began by giving it ODEs and explaining that we wanted an animation of the dynamics given input angles and a PD controller. From there, we iterated the code until we had a model that gave us what we were looking for. You can find an exact copy of our script used on the last page of this document.

Sample animation: https://youtu.be/I4UrNOWYjr8

Below, you can find the plot generated by our Matlab script that plots the behavior of each angle over time. This was a good way we could check the logic of our code, as it allowed us to clearly see a step response and behavior that we expected.

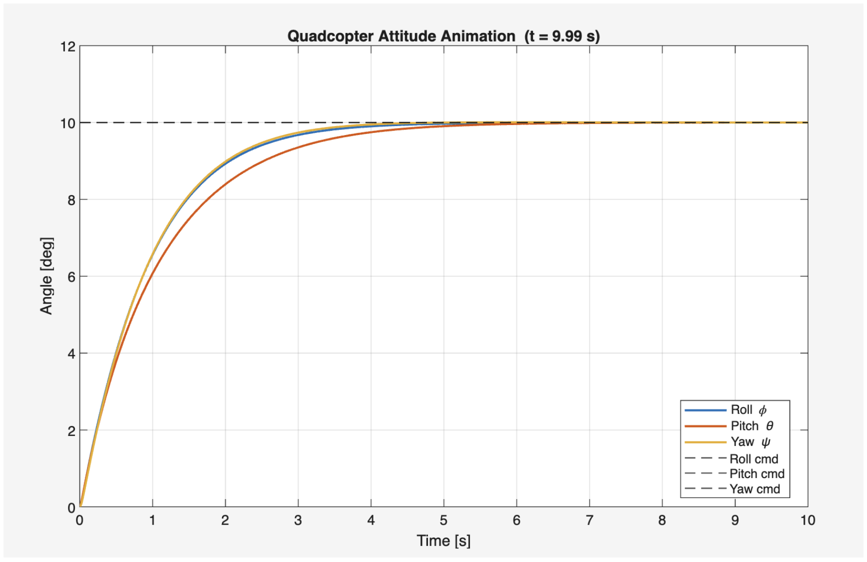




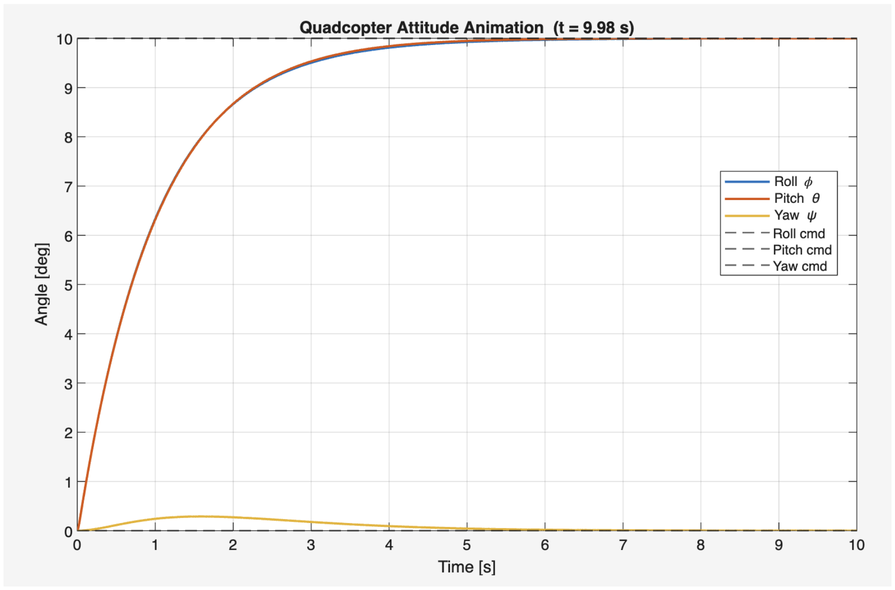
Reflecting on our animation, we can see that the quadcopter quickly responds to the input, and behaves how we expect it to. The angular acceleration begins quite high, but as the angle approaches the input angle, the proportional control has a smaller input, leading to the behavior displayed in the animation of step response graph above. Because of this, we believe that our code gives an accurate and reasonable model of the behavior of a quadcopter modeled with a PD controller. We can also see that our initial control inputs are quite high due to the high proportional gain of the pitch controller. If this was actually being implemented into a quadcopter, we would have to check to make sure the high effort wouldn't saturate the motor. If it did, this could be fixed by lowering the proportional gain of the pitch controller.

Gain Values and Rotational Behavior

Through modifying the chosen gain values of the system and identifying trends, we can further analyze the system and break down how choices within the system can affect the results. While the differences between angles like the roll (rotates about x axis), pitch (rotates about y axis), and yaw (rotates about z) of the quadcopter are simply a matter of orientation, the differences in rotational inertia cause these orientations to behave differently. Specifically, the quadcopter in our script is modeled with the rotational inertias of 0.02 kg*m^2 in the x and y directions, and 0.04 kg*m^2 along the z axis. To study the effects of this nonuniform inertia, we first set Kp and Kd gains for all dimensions to 2 and the command degree to be 10 (low angle input to maintain small angle approximations) for uniformity across the board, which produced the graph below.



This result showcases that the differences in rotational inertia causes the damping ratio of the yaw and roll to be approximately the same with pitch being lower. For the motion of a quadcopter, the most important angles to control a quadcopter's motion is the roll and pitch, as yaw isn't particularly important when it comes to piloting a quadcopter except for more advanced maneuvers. In order to control the system smoothly, we ideally want these two dimensions to behave identically so that translations in the x or y axis do not get influenced differently with the same degree of input. Noticeably however, once we set the yaw command to be 0, roll and pitch dimensions behave nearly identically, so this discrepancy is likely due to the gyroscopic coupling not being symmetric, which causes unequal bleed overs into the different rotational axes when a yaw command is inputted. These flaws are inherent within the Euler angle representatives used to model the system, so simply modifying the gains would only reduce the problem, not eliminate it. In reality, we could increase the pitch gains such that the discrepancies in it are smaller, or use a cascade attitude rate controller instead of PD as most quadcopters do.



References

[1]: A. Sulficar, H. Suresh, A. Varma, and A. Radhakrishnan, “MODELING, SIMULATION AND Complete Control of a Quadcopter,” National Institute of Technology Karnatka, Surathkal, Mangalore, May, 2017.

[2]: Carbon Aeronautics, “Manual-Quadcopter-Drone”, *github.com*, August, 2022. [Online]. Available “https://github.com/CarbonAeronautics/Manual-Quadcopter-Drone/tree/main”.

Matlab Script

```
function quadcopter_attitude_step

      % Simple quadcopter attitude step-response with 3D animation
      % State x = [phi; theta; psi; p; q; r]
      % phi,theta,psi = roll, pitch, yaw (rad)
      % p,q,r = body angular rates about x,y,z (rad/s)

      close all; clc;

      %% Parameters (rough but reasonable for a small quad)
      params.Ix = 0.02;   % kg*m^2
      params.Iy = 0.02;   % kg*m^2
      params.Iz = 0.04;   % kg*m^2

      % PD gains on [roll, pitch, yaw]
      params.Kp = diag([2, 8, 3]);     % proportional
      params.Kd = diag([8, 4, 1.5]);   % derivative

      %% Command: step in roll, pitch, yaw (degrees -> radians)
      phi_cmd_deg   = 30;   % roll command
      theta_cmd_deg =  16;   % pitch command
      psi_cmd_deg   = 22;   % yaw command

      cmd = deg2rad([phi_cmd_deg; theta_cmd_deg; psi_cmd_deg]);   % 3x1

      %% Initial conditions and sim time
      x0 = zeros(6,1);      % start level, no rotation
      tspan = [0 10];        % seconds

      %% Simulate nonlinear attitude dynamics
      [T, X] = ode45(@(t,x) quad_attitude_dynamics(t,x,cmd,params), tspan, x0);

      %% Plot attitude response (in degrees)
      figure;
      plot(T, X(:,1)*180/pi, 'LineWidth', 1.5); hold on;
      plot(T, X(:,2)*180/pi, 'LineWidth', 1.5);
      plot(T, X(:,3)*180/pi, 'LineWidth', 1.5);
      yline(phi_cmd_deg,   '--', 'LineWidth', 1);
      yline(theta_cmd_deg, '--', 'LineWidth', 1);
      yline(psi_cmd_deg,   '--', 'LineWidth', 1);
      grid on;
      xlabel('Time [s]');
      ylabel('Angle [deg]');
      legend({'Roll \phi','Pitch \theta','Yaw \psi', ...
              'Roll cmd','Pitch cmd','Yaw cmd'}, ...
              'Location','best');
      title('Quadcopter Attitude Step Response');

      %% Animate quadcopter attitude
      animate_quadcopter(T, X, cmd);

      end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
function dx = quad_attitude_dynamics(~, x, cmd, params)
% Nonlinear attitude dynamics for a rigid-body quadcopter (rotation only)

% Unpack state
phi   = x(1);   % roll
theta = x(2);   % pitch
psi   = x(3);   % yaw
p     = x(4);   % roll rate
q     = x(5);   % pitch rate
r     = x(6);   % yaw rate

angles = [phi; theta; psi];
rates  = [p; q; r];

% PD control torques about body axes
angle_error = cmd - angles;      % desired - current
tau = params.Kp * angle_error - params.Kd * rates;  % 3x1 [tau_phi;
tau_theta; tau_psi]

Ix = params.Ix;
Iy = params.Iy;
Iz = params.Iz;

% Rigid-body rotational dynamics: I * w_dot + w x (I w) = tau
p_dot = ((Iy - Iz)/Ix)*q*r + tau(1)/Ix;
q_dot = ((Iz - Ix)/Iy)*p*r + tau(2)/Iy;
r_dot = ((Ix - Iy)/Iz)*p*q + tau(3)/Iz;

% Euler kinematics (ZYX convention: yaw-pitch-roll)
sphi   = sin(phi);
cphi   = cos(phi);
ttheta = tan(theta);
ctheta = cos(theta);

% Map body rates [p;q;r] -> Euler angle rates [phi_dot;theta_dot;psi_dot]
E = [1,          sphi*ttheta,  cphi*ttheta;
     0,          cphi,        -sphi;
     0,          sphi/ctheta,  cphi/ctheta];

ang_dot = E * rates;   % [phi_dot; theta_dot; psi_dot]

dx = [ang_dot;
      p_dot;
      q_dot;
      r_dot];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
function animate_quadcopter(T, X, cmd)
% Simple 3D animation of quadcopter roll/pitch/yaw

armLength = 0.3;   % length of each arm (m)

% Define 4 arm endpoint points in body frame (X forward, Y right, Z down)
```

```
bodyPts = [ armLength, -armLength,    0,          0;
                 0,         0,    armLength, -armLength;
                 0,         0,       0,          0      ];  % 3x4

figure;
axis equal;
xlim([-0.5 0.5]);
ylim([-0.5 0.5]);
zlim([-0.5 0.5]);
grid on;
xlabel('X');
ylabel('Y');
zlabel('Z');
view(35, 25);
title('Quadcopter Attitude Animation');

hold on;

% Draw world axes
line([-0.4 0.4],[0 0],[0 0],'Color',[0.7 0.7 0.7],'LineStyle','-');
line([0 0],[-0.4 0.4],[0 0],'Color',[0.7 0.7 0.7],'LineStyle','-');
line([0 0],[0 0],[-0.4 0.4],'Color',[0.7 0.7 0.7],'LineStyle','-');
text(0.42,0,0,'X');
text(0,0.42,0,'Y');
text(0,0,0.42,'Z');

% Initial attitude
phi   = X(1,1);
theta = X(1,2);
psi   = X(1,3);

R = rotz(psi) * roty(theta) * rotx(phi);  % body -> inertial
pts = R * bodyPts;

% Create graphics objects for arms
h1 = plot3(pts(1,1:2), pts(2,1:2), pts(3,1:2), 'LineWidth', 3); %
front-back arm
h2 = plot3(pts(1,3:4), pts(2,3:4), pts(3,3:4), 'LineWidth', 3); %
left-right arm

% Center marker
hc = plot3(0,0,0,'ko','MarkerSize',6,'MarkerFaceColor','k');

% Desired attitude in legend (optional text)
cmd_deg = cmd * 180/pi;
legend([h1, h2], ...
       {sprintf('Arms (cmd: \\phi=%.1f°, \\theta=%.1f°, \\psi=%.1f°)',
...
        cmd_deg(1), cmd_deg(2), cmd_deg(3)), ...
        'Perpendicular arms'}, ...
       'Location','southoutside');

numSteps  = length(T);
frameStep = max(1, floor(numSteps/400));  % skip points for speed

idxList = 1:frameStep:numSteps;
```

```
for i = 1:length(idxList)
    k = idxList(i);

    phi   = X(k,1);
    theta = X(k,2);
    psi   = X(k,3);

    R = rotz(psi) * roty(theta) * rotx(phi);
    pts = R * bodyPts;

    % Update arm lines
    set(h1, 'XData', pts(1,1:2), 'YData', pts(2,1:2), 'ZData',
pts(3,1:2));
    set(h2, 'XData', pts(1,3:4), 'YData', pts(2,3:4), 'ZData',
pts(3,3:4));

    % Center stays at origin
    set(hc, 'XData', 0, 'YData', 0, 'ZData', 0);

    title(sprintf('Quadcopter Attitude Animation  (t = %.2f s)', T(k)));
    drawnow;

    % --- Make animation run in (approximately) real time ---
    if i < length(idxList)
        t_now   = T(idxList(i));
        t_next  = T(idxList(i+1));
        dt_real = t_next - t_now;   % seconds of simulation time
        pause(max(dt_real, 0));     % wait that long in real time
    end
end
% Initialize GIF
gifFilename = 'quadcopter_animation.gif';
frameDelay = 0.1; % Delay between frames in seconds

for i = 1:length(idxList)
    k = idxList(i);

    phi   = X(k,1);
    theta = X(k,2);
    psi   = X(k,3);

    R = rotz(psi) * roty(theta) * rotx(phi);
    pts = R * bodyPts;

    % Update arm lines
    set(h1, 'XData', pts(1,1:2), 'YData', pts(2,1:2), 'ZData',
pts(3,1:2));
    set(h2, 'XData', pts(1,3:4), 'YData', pts(2,3:4), 'ZData',
pts(3,3:4));

    % Center stays at origin
    set(hc, 'XData', 0, 'YData', 0, 'ZData', 0);

    title(sprintf('Quadcopter Attitude Animation  (t = %.2f s)', T(k)));
    drawnow;

    % Capture the current frame
    frame = getframe(gcf);
    im = frame2im(frame);
    [imind, cm] = rgb2ind(im, 256);

    % Write to the GIF file
    if i == 1
        imwrite(imind, cm, gifFilename, 'gif', 'LoopCount', inf,
'DelayTime', frameDelay);
    else
        imwrite(imind, cm, gifFilename, 'gif', 'WriteMode', 'append',
'DelayTime', frameDelay);
    end

    % --- Make animation run in (approximately) real time ---
    if i < length(idxList)
        t_now   = T(idxList(i));
        t_next  = T(idxList(i+1));
        dt_real = t_next - t_now;   % seconds of simulation time
        pause(max(dt_real, 0));     % wait that long in real time
    end
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
function R = rotx(a)
% Rotation matrix about x-axis
ca = cos(a);
sa = sin(a);
R = [1   0    0;
     0  ca  -sa;
     0  sa   ca];
end

function R = roty(a)
% Rotation matrix about y-axis
ca = cos(a);
sa = sin(a);
R = [ ca  0  sa;
       0  1   0;
     -sa  0  ca];
end

function R = rotz(a)
% Rotation matrix about z-axis
ca = cos(a);
sa = sin(a);
R = [ca  -sa  0;
     sa   ca  0;
      0    0  1];
end
```