Attitude Control for Quadcopters: A PD-Based Approach for Stabilization

Akash Ajin, Akhilesh Menon, Paul Choi, Shiv Kanade
MAT292: Ordinary Differential Equations

Fall 2025

Abstract

This project investigates the attitude dynamics of a quadcopter, an inherently unstable system that requires continuous feedback control for stable flight. We derive a 12-state nonlinear rigid-body model based on the Newton–Euler equations of motion and formulate it as a coupled system of ordinary differential equations (ODEs). Using this model, we design a Proportional–Derivative (PD) control architecture for attitude (roll, pitch, yaw) and a Proportional–Integral–Derivative (PID) controller for altitude. Numerical simulations using a Runge–Kutta ODE solver (`scipy.integrate.solve_ivp`) show that the tuned controllers stabilize the vehicle from rest, track step commands in roll and yaw with small overshoot and short settling times, and maintain altitude with negligible steady-state error. These results illustrate how ODE modeling and numerical integration directly enable the analysis and design of feedback controllers for real engineering systems.

1 Introduction

1.1 Motivation

A quadcopter is an unmanned aerial vehicle (UAV) whose flight is controlled by four motors. This UAV's navigational agility has made it popular in fields from photography to logistics. However, quadcopters are inherently unstable. Without a constant stream of adjustments from a control system, minor disturbances would cause them to tumble. The system's dynamics are also highly non-linear and coupled, making it a very fitting problem for an ODE course.

1.2 Project Goal

The primary objective of this project was to model the unstable flight dynamics of a quadcopter as a system of coupled, non-linear ordinary differential equations, and to use this model to design and tune a stabilizing feedback controller. We implement a PD architecture for attitude and a PID controller for altitude, and evaluate their performance in numerical

simulation. The project provides a clear, practical demonstration of how the tools of an ODE course—modeling, linear algebra, and numerical solvers—are used to solve a fundamental problem in modern robotics. In this work we restrict attention to hover and small-angle attitude maneuvers, which allows us to focus on deriving and simulating the underlying ODEs without the added complexity of full 3D trajectory tracking.

2 Mathematical Model and Theoretical Foundation

The quadcopter's motion is modeled as a rigid body in 3D space, using a fixed inertial frame (Earth, E) and a rotating body frame (Body, B) attached to the vehicle [1], [2].

2.1 State Vector

The system is described by a 12-element state vector $\mathbf{x} \in \mathbb{R}^{12}$:

$$\mathbf{x} = [\ \underbrace{x, y, z}_{\substack{\text{position} \\ \text{(Frame E)}}}\ ,\quad \underbrace{\phi, \theta, \psi}_{\substack{\text{attitude (Euler)} \\ \text{(Frame E)}}}\ ,\quad \underbrace{\dot{x}, \dot{y}, \dot{z}}_{\substack{\text{lin. velocity} \\ \text{(Frame E)}}}\ ,\quad \underbrace{p, q, r}_{\substack{\text{ang. velocity} \\ \text{(Frame B)}}}\ ]^T$$

This state is a hybrid, as linear motion is tracked in the inertial frame E while rotational motion is tracked in the body frame B. This requires transformation matrices to couple the dynamics.

2.2 Translational Dynamics

In the inertial frame, Newton's 2nd Law governs translational motion:

$$\mathbf{F}_{\text{net},E} = m\ddot{\mathbf{p}}_E = m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix}$$

The net force is the sum of gravity $\mathbf{F}_{grav,E} = [0, 0, -mg]^T$ and the total thrust $\mathbf{F}_{thrust,B} = [0, 0, T]^T$. Thrust is generated in the body frame (acting along the quadcopter's z_B-axis) and must be rotated into the inertial frame:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \frac{1}{m} \left(\begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R(\phi, \theta, \psi) \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \right)$$

where $T = T_1 + T_2 + T_3 + T_4$ and R is the $Z - Y - X$ body-to-inertial rotation matrix. R is constructed as $R = R_z(\psi)R_y(\theta)R_x(\phi)$:

$$R = \begin{bmatrix} c\psi c\theta & c\psi s\theta s\phi - s\psi c\phi & c\psi s\theta c\phi + s\psi s\phi \\ s\psi c\theta & s\psi s\theta s\phi + c\psi c\phi & s\psi s\theta c\phi - c\psi s\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix}$$

where $c\alpha = \cos(\alpha)$ and $s\alpha = \sin(\alpha)$. The first six ODEs are thus: $\dot{x} = \dot{x}$, $\dot{y} = \dot{y}$, $\dot{z} = \dot{z}$, and the three translational acceleration equations above $(\ddot{x}, \ddot{y}, \ddot{z})$.

2.3 Rotational Dynamics

Rotational motion is described by the Newton–Euler equations in the body frame [1], [3]:

$$\tau_B = I\dot{\omega}_B + \omega_B \times (I\omega_B)$$

where $\tau_B = [\tau_\phi, \tau_\theta, \tau_\psi]^T$ are the net torques, I is the 3×3 inertia tensor, and $\omega_B = [p, q, r]^T$. Assuming a diagonal inertia tensor $I = \text{diag}(I_{xx}, I_{yy}, I_{zz})$, the ODEs for the angular rates are:

$$\begin{aligned}
\dot{p} &= (1/I_{xx})\,(\tau_\phi - (I_{zz} - I_{yy})qr) \\
\dot{q} &= (1/I_{yy})\,(\tau_\theta - (I_{xx} - I_{zz})pr) \\
\dot{r} &= (1/I_{zz})\,(\tau_\psi - (I_{yy} - I_{xx})pq)
\end{aligned}$$

The Euler angle derivatives (in frame E) are related to the body rates (in frame B) by the transformation W:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = W(\phi, \theta) \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

These nine equations form the complete system of 12 first-order ODEs ($\dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \ddot{x}, \ddot{y}, \ddot{z}, \dot{p}, \dot{q}, \dot{r}$), which can be written in the form $\dot{\mathbf{x}} = f(t, \mathbf{x}, \mathbf{u})$.

3 Methodology

3.1 Overall Control Architecture

The controller is organized in two layers. An **inner attitude loop** uses three PD controllers to track desired Euler angles $(\phi_d, \theta_d, \psi_d)$ by commanding body-frame torques $(\tau_\phi, \tau_\theta, \tau_\psi)$. An **altitude loop** uses a PID controller to track the desired height z_d by commanding the total thrust T. Together, these four controllers produce the virtual control vector

$$\mathbf{u}_v = [T, \tau_\phi, \tau_\theta, \tau_\psi]^T,$$

which is then mapped to the individual motor thrusts via the inverse mixing matrix M^{-1} described below. In the simulations we chose simple step commands (for example, $z_d = 1.0\,\text{m}$ and attitude steps of $\phi_d = 10^\circ$, $\psi_d = 30^\circ$ over a finite time window), while holding $\theta_d = 0^\circ$ to test cross-coupling.

3.2 Control Strategy: PID Controller

To stabilize the dynamics, we implemented four independent controllers following the standard PID control law [4], [5], which computes a corrective action

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt}.$$

The proportional term $K_p e(t)$ responds to the current error, the integral term accumulates past error to eliminate steady-state drift, and the derivative term anticipates future error to damp oscillations. For the fast attitude axes (ϕ, θ, ψ) we ultimately set $K_i \approx 0$ and used a PD structure, while the altitude controller retained a nonzero K_i to remove steady-state error against gravity.

3.3 Control Allocation

The four PID outputs, known as virtual controls, $\mathbf{u}_v = [T, \tau_\phi, \tau_\theta, \tau_\psi]^T$, must be mapped to the four individual motor thrusts $\mathbf{T}_m = [T_1, T_2, T_3, T_4]^T$. Based on the quadcopter configuration and the final invertible mixing matrix used:

$$\begin{aligned} T &= T_1 + T_2 + T_3 + T_4 \\ \tau_\phi &= L(-T_1 + T_3) \quad \text{(Roll torque)} \\ \tau_\theta &= L(-T_2 + T_4) \quad \text{(Pitch torque)} \\ \tau_\psi &= k_m(-T_1 + T_2 - T_3 + T_4) \quad \text{(Yaw torque)} \end{aligned}$$

This required the inverse mapping $\mathbf{T}_m = M^{-1}\mathbf{u}_v$, where M^{-1} is:

$$M^{-1} = \begin{bmatrix} 1/4 & -1/(2L) & 0 & -1/(4k_m) \\ 1/4 & 0 & -1/(2L) & 1/(4k_m) \\ 1/4 & 1/(2L) & 0 & -1/(4k_m) \\ 1/4 & 0 & 1/(2L) & 1/(4k_m) \end{bmatrix}$$

3.4 Numerical Simulation

The simulation was implemented in Python using the `scipy.integrate.solve_ivp` function [6], which employs a Runge–Kutta 45 method to solve the 12 ODEs [7]. We used a main loop that calls `solve_ivp` at each discrete control step with sampling period $\Delta t = 0.01$ s. At each step k, the controller computes the motor thrusts from the current state $\mathbf{x}(t_k)$, and `solve_ivp` integrates the ODEs from t_k to $t_{k+1} = t_k + \Delta t$ using these thrusts as constant inputs. This mimics a digital flight controller running at 100 Hz interacting with continuous-time vehicle dynamics. The initial condition is a vehicle at rest near the origin, which highlights the need for active stabilization.

4 Results

4.1 PID Tuning

The goal of tuning was to achieve a near *critically damped* response for all four axes: fast tracking with minimal overshoot or oscillation.

We initially attempted the Ziegler–Nichols Ultimate Cycle Method [8] to establish a theoretical baseline. However, due to the highly non-linear nature and cross-coupling present in the full ODE model, this approach yielded aggressive gains that resulted in significant

under-damped oscillations and, critically, the integral term K_i consistently triggered integral windup, leading to instability in the attitude axes.

Therefore, the final stable design for the fast-acting attitude controllers (roll, pitch, yaw) uses a Proportional–Derivative (PD) structure ($K_i \approx 0$). The altitude controller retains the integral term to eliminate steady-state error against gravity. The final tuned gains are presented in Table 1. For these gains, the altitude response exhibits a rise time of roughly $t_r \approx 2.0\,\text{s}$ and settles within a small band around the target by about $t_s \approx 3.0\,\text{s}$ with negligible overshoot. The roll and yaw responses have rise times on the order of 1.0 s and settle without sustained oscillations, which is consistent with a nearly critically damped design.

Table 1: Final Controller Gains for Critically Damped Response

Controller	K_p	K_i	K_d
Altitude (z)	0.17	0.21	1.00
Roll (ϕ)	0.08	0.001	0.043
Pitch (θ)	0.08	0.000	0.043
Yaw (ψ)	0.80	0.000	0.16

4.2 Hover Stabilization

The simulation commanded the quadcopter to maintain a hover at $z = 1.0$m. The Altitude controller performance is shown in Figure 1.

4.3 Attitude Disturbance Rejection

The controller's ability to track commands and stabilize against disturbances was tested by commanding a 10-degree Roll and a 30-degree Yaw from $t = 2$s to $t = 8$s. The attitude controller performance is shown in Figure 2.

5 Discussion

The primary objective of stabilizing the quadcopter using an ODE-based model and feedback control was successfully met.

5.1 Interpretation of Results

As shown in Figure 1, the altitude controller (PID) achieves near-perfect tracking of the $z = 1.0\,\text{m}$ setpoint with no visible overshoot and a short settling time. The integral term K_i successfully eliminates the steady-state error that would otherwise be present due to gravity and small modeling offsets in the gravity-compensation feed-forward.

Figure 2 demonstrates the performance of the attitude controllers (PD). The roll and yaw axes track their respective command steps (10° and 30°) with fast rise times and settle without the destructive oscillations observed during the initial tuning phase. When pitch

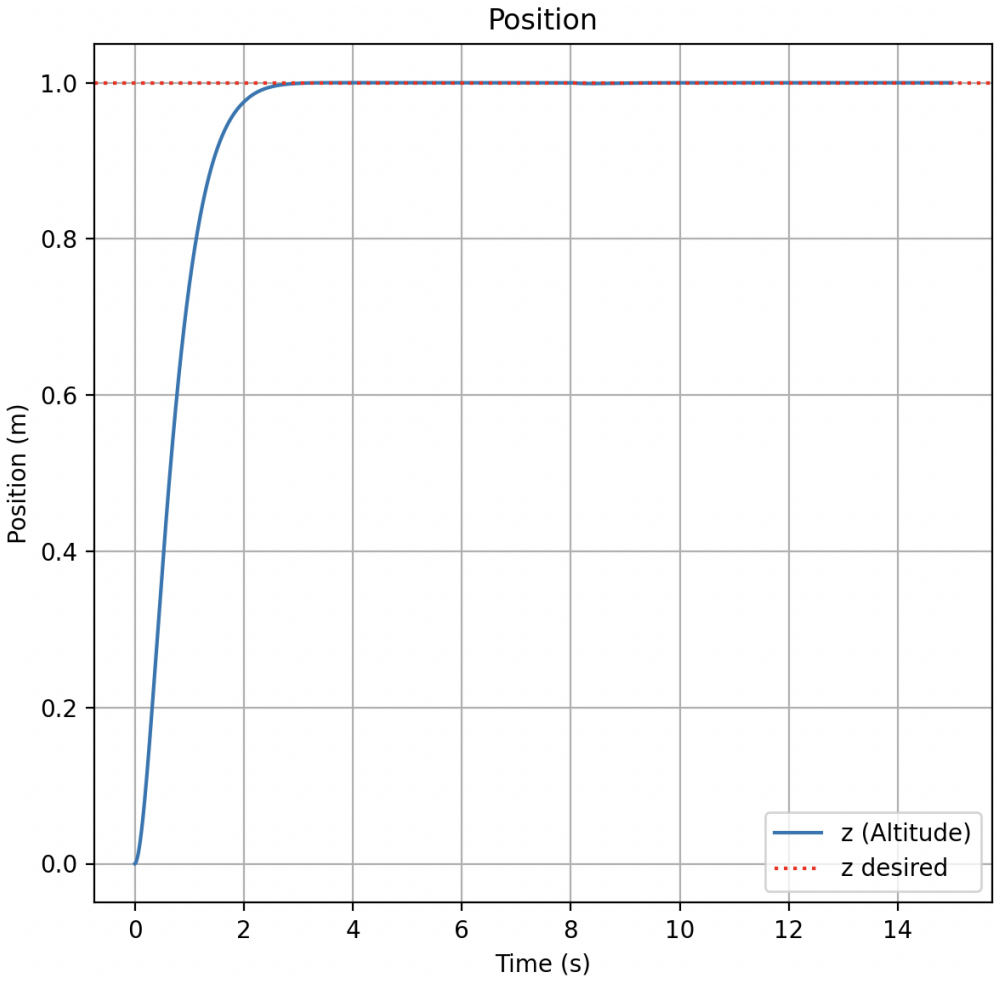

Figure 1: Altitude (z-axis) response to a 1.0m step command. The PID controller brings the quadcopter to the desired altitude in approximately 3 seconds with zero overshoot.

is held at 0°, the response remains close to zero throughout the maneuver, confirming the robustness of the mixing matrix $\mathbf{M}^{-1}$ and the effectiveness of the decoupled control design. Overall, the time-domain behavior observed in simulation is consistent with the design goal of a nearly critically damped response.

5.2 Limitations and Future Work

The model currently has several simplifying assumptions:

- **No aerodynamic drag:** We assume drag on the body is negligible. This is accurate for low speeds but, at higher speeds, drag forces would need to be added to the translational ODEs in $\mathbf{F}_{\text{net},E}$, potentially changing the optimal gains.
- **Perfect control inputs:** We assume motors react instantly and linearly, ignoring actuator saturation and time delays. Including motor dynamics would require additional states and ODEs and would make the closed-loop system higher dimensional.

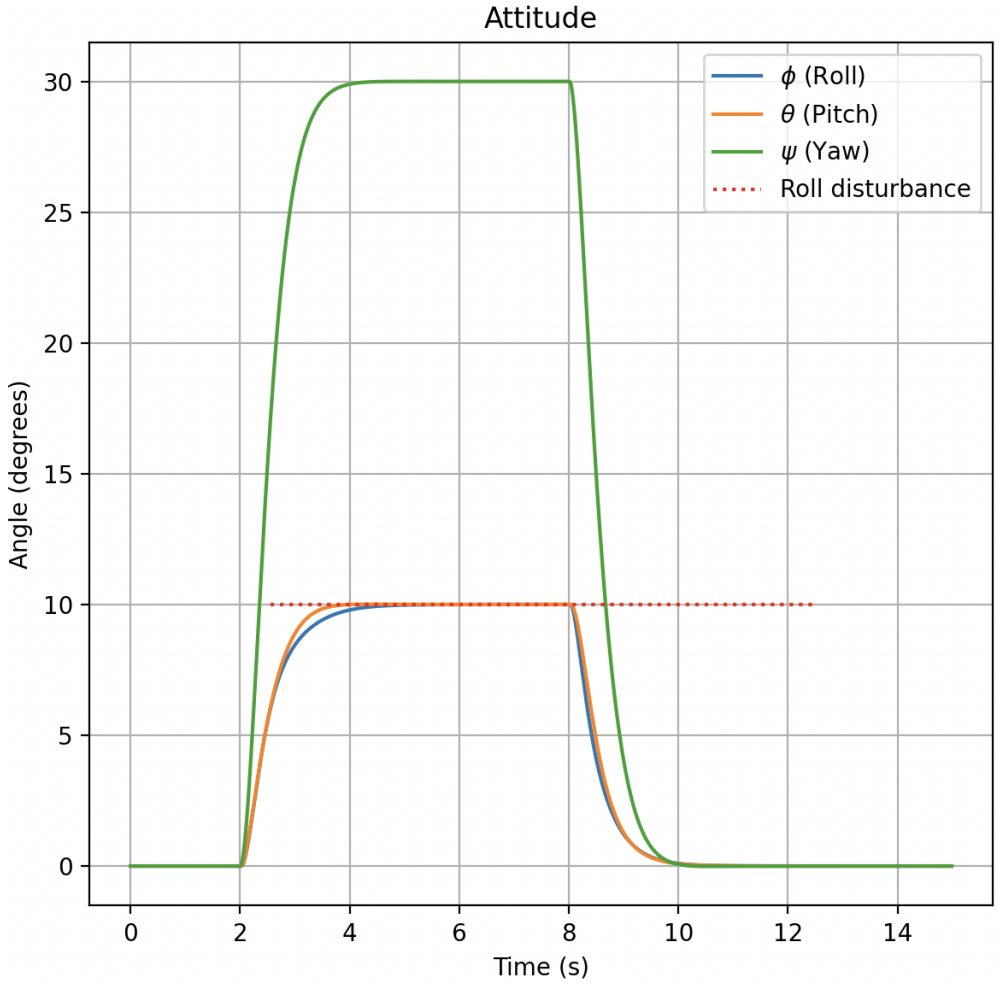

Figure 2: Attitude (roll, pitch, yaw) response to simultaneous step commands (Roll=10°, Yaw=30°). The PD controller achieves fast rise time with minimal overshoot, and the Pitch axis remains stable at 0°, demonstrating excellent **cross-coupling rejection.**

- **Attitude vs. position control:** The current controller regulates altitude and attitude but not horizontal position. A natural extension is a cascaded architecture where an outer-loop PID generates desired roll and pitch angles from (x, y) position errors, enabling full 3D trajectory tracking.

These limitations suggest clear directions for future work: extend the ODE model to include additional physics and actuators, and design corresponding controllers that preserve stability and performance.

6 Conclusion

We derived a 12-DOF non-linear ODE system governing quadcopter dynamics and implemented a stabilizing PD/PID control architecture solved numerically with Runge–Kutta methods (`scipy.integrate.solve_ivp`). The simulations show that the inherently unsta-

ble open-loop system can be stabilized, with step responses that exhibit small overshoot and short settling times in both altitude and attitude. This project demonstrates how the core ideas from an ODE course—formulating physical laws as differential equations and solving them numerically—translate directly into the analysis and design of practical control systems [5], [7].

References

[1] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor robots," *IEEE Robotics and Automation Magazine*, vol. 19, no. 3, pp. 20–32, 2012. DOI: 10.1109/MRA.2012.2206474.

[2] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, "Quadrotor helicopter flight dynamics and control: Theory and experiment," in *AIAA Guidance, Navigation and Control Conference*, AIAA, 2007.

[3] D. T. Greenwood, *Classical Dynamics*. Dover Publications, 2003.

[4] K. J. Åstr"om and T. H"agglund, *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, 1995.

[5] K. Ogata, *Modern Control Engineering*, 5th ed. Prentice Hall, 2010.

[6] P. Virtanen, R. Gommers, T. E. Oliphant, et al., "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[7] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, 1998.

[8] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *Transactions of the ASME*, vol. 64, no. 8, pp. 759–768, 1942.

A Simulation Code

The final Python code used for the simulation and data generation is provided below.

```
"""Quadcopter attitude/altitude simulator.

How to run
~~~~~~~~~~
1. Install dependencies once: ``pip install numpy scipy matplotlib``.
2. Execute ``python Code\\ simulator.py`` from this folder.  Two plots
     will appear when the run finishes (position/attitude and motor thrusts).

How to experiment
~~~~~~~~~~~~~~~~~
- Tune the PID gains in the "Initialize Controllers" section: start with
    only Kp, then add Kd for damping, finally Ki for steady-state trim.
```

```
- Modify the step disturbance (currently a 10° roll request from 2{8 s)
    or replace it with your own trajectory generator.
- Adjust physical parameters (mass, arm length, inertia tensor, etc.) to
    match the vehicle you want to emulate.

Testing ideas
~~~~~~~~~~~~~
- Add assertions/prints inside the loop (e.g., ensure motor thrusts stay
    within [T_min, T_max]).
- Compare different gain sets by running the file multiple times and
    inspecting the plots or exporting ``x_values``/``motor_thrusts``.
"""

import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

class PIDController:
    """Minimal PID helper used by each axis controller.

    Parameters
    ----------
    Kp, Ki, Kd : float
        Proportional / integral / derivative gains.  Tuning tip:
        raise Kp until you see crisp response, use Kd to damp oscillations,
        then add a small Ki only if you need zero steady-state error.
    """
    def __init__(self, Kp, Ki, Kd):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.integral_error = 0.0
        self.previous_error = 0.0

    def update(self, error, dt):
        """Calculate the PID control output."""
        # Proportional term
        p_term = self.Kp * error

        # Integral term
        self.integral_error += error * dt
        i_term = self.Ki * self.integral_error

        # Derivative term
        derivative_error = (error - self.previous_error) / dt
```

```
        d_term = self.Kd * derivative_error

        # Update previous error for next iteration
        self.previous_error = error

        return p_term + i_term + d_term

    def reset(self):
        """Resets the integral and previous error."""
        self.integral_error = 0.0
        self.previous_error = 0.0

def quadcopter_dynamics(t, x, m, g, I, L, k_m, T1, T2, T3, T4):
    """Continuous-time rigid-body model passed to ``solve_ivp``.

    Parameters
    ----------
    t : float
        Current integration time (unused, but required by ``solve_ivp``).
    x : ndarray shape (12,)
        State vector ``[x, y, z, phi, theta, psi, x_dot, y_dot, z_dot, p, q, r]``.
    m, g, I, L, k_m : floats/array
        Physical constants: mass, gravity, inertia matrix, arm length, motor torque coef
    T1..T4 : float
        Individual motor thrusts (already saturated and non-negative).

    Returns
    -------
    dot_x : ndarray shape (12,)
        Time derivative of the state used by the numerical integrator.
    """

    # --- 1. Unpack State Vector ---
    pos = x[0:3]       # [x, y, z]
    angles = x[3:6]    # [phi, theta, psi]
    vel = x[6:9]       # [x_dot, y_dot, z_dot]
    rates = x[9:12]    # [p, q, r]

    phi, theta, psi = angles
    p, q, r = rates

    # --- 2. Calculate Forces and Torques ---
    # Total thrust in Body frame
    T_total = T1 + T2 + T3 + T4
    F_thrust_B = np.array([0, 0, T_total])
```

```
# Torques in Body frame (based on '+' configuration)
tau_phi   = L * (-T1 + T3)               # roll
tau_theta = L * (-T2 + T4)               # pitch
tau_psi   = k_m * (-T1 + T2 - T3 + T4)   # yaw
tau_B = np.array([tau_phi, tau_theta, tau_psi])

# --- 3. Precompute Trig Functions ---
c_phi = np.cos(phi)
s_phi = np.sin(phi)
c_theta = np.cos(theta)
s_theta = np.sin(theta)
c_psi = np.cos(psi)
s_psi = np.sin(psi)

# --- 4. Calculate State Derivatives ---
dot_x = np.zeros(12)

# $\dot{p} = v$ (Derivative of position is velocity)
dot_x[0:3] = vel

# $\dot{\eta} = W \omega$ (Derivative of Euler angles)
# Transformation matrix W
W = np.array([
    [1, s_phi * np.tan(theta), c_phi * np.tan(theta)],
    [0, c_phi, -s_phi],
    [0, s_phi / c_theta, c_phi / c_theta]
])
dot_x[3:6] = W @ rates

# $\dot{v} = \ddot{p}$ (Derivative of velocity is acceleration)
# Rotation matrix R (Body to Inertial)
    # State and actuator histories.  Having these arrays makes it easy to
    # compute custom metrics (settling time, overshoot, RMS error) after the
    # run by simply operating on the columns you care about.
R = np.array([
    [c_psi * c_theta, c_psi * s_theta * s_phi - s_psi * c_phi, c_psi * s_theta * c_p
    [s_psi * c_theta, s_psi * s_theta * s_phi + c_psi * c_phi, s_psi * s_theta * c_p
    [-s_theta, c_theta * s_phi, c_theta * c_phi]
])

# Gravity vector in Inertial frame
F_grav_I = np.array([0, 0, -m * g])

# Thrust in Inertial frame
```

```
    F_thrust_I = R @ F_thrust_B

    # Net force in Inertial frame
    F_net_I = F_grav_I + F_thrust_I

    # Translational acceleration
    dot_x[6:9] = F_net_I / m

    # $\dot{\omega} = \ddot{\eta}$ (Derivative of angular rates)
    # Newton-Euler equations
    I_inv = np.linalg.inv(I)
    # Gyroscopic terms (omega_B x (I @ omega_B))
    gyro_terms = np.cross(rates, I @ rates)

    dot_x[9:12] = I_inv @ (tau_B - gyro_terms)

    return dot_x

# --- Main Simulation ---
if __name__ == "__main__":

    # --- Physical Constants ---
    # Feel free to swap in your vehicle numbers here.  Keeping these in
    # one place makes it easy to create "what-if" scenarios (heavier mass,
    # longer arms, different inertia tensor, etc.).
    g = 9.81      # Gravitational acceleration (m/s^2)
    m = 0.5       # Mass of quadcopter (kg)
    L = 0.225     # Arm length (m)
    k_m = 0.01    # Yaw torque coefficient (N*m / N)
    # Inertia tensor (assuming a symmetrical frame)
    I_xx = 0.005
    I_yy = 0.005
    I_zz = 0.01
    I = np.diag([I_xx, I_yy, I_zz])

    # --- Simulation Setup ---
    # ``dt`` controls controller rate; lowering it increases fidelity but
    # also simulation time.  ``total_time`` lets you test longer missions.
    dt = 0.01             # Control loop time step (s) -> 100 Hz
    total_time = 15.0     # Total simulation time (s)
    num_steps = int(total_time / dt)

    # Initial state vector [x, y, z, phi, theta, psi, x_dot, y_dot, z_dot, p, q, r]
    # Start on the ground, slightly tilted
    x_initial = np.zeros(12)
```

```
x_initial[3] = np.deg2rad(0.0) # 1 degree roll
x_initial[4] = np.deg2rad(0.0) # -1 degree pitch

# Store results
t_values = np.zeros(num_steps)
# State history makes it easy to compute settling time/overshoot metrics
# or export data to CSV for offline analysis.
x_values = np.zeros((12, num_steps))
t_values[0] = 0.0
x_values[:, 0] = x_initial

# Store motor thrusts for plotting/logging
motor_thrusts = np.zeros((4, num_steps))

# --- Initialize Controllers ---
# Each axis gets its own PID.  Start with just Kp (Ki=Kd=0), then bring
# in Kd for damping, finally Ki for steady-state trim.  The numbers below
# are simply baseline values|swap in your own when experimenting.
pid_z = PIDController(Kp=3, Ki=0, Kd=2.28)          # Altitude loop
pid_roll = PIDController(Kp=0.08, Ki=0.00, Kd=0.043)  # Primary tuning axis
pid_pitch = PIDController(Kp=0.08, Ki=0.0, Kd=0.043)   # Gentle hold
pid_yaw = PIDController(Kp=0.1, Ki=0.0, Kd=0.057)       # Yaw hold

# --- Control Allocation Matrix (Inverse) ---
# T_m = M_inv @ u_v
M_inv = np.array([
    [0.25,        -1.0/(2.0*L),  0.0,           -1.0/(4.0*k_m)],
    [0.25,         0.0,          -1.0/(2.0*L),   1.0/(4.0*k_m)],
    [0.25,         1.0/(2.0*L),  0.0,           -1.0/(4.0*k_m)],
    [0.25,         0.0,          1.0/(2.0*L),    1.0/(4.0*k_m)],
])

# Max/Min motor thrust (for saturation)
T_max = (m * g) / 2.0 # Max thrust per motor (e.g., 2x gravity)
T_min = 0.0

print("Starting simulation...")

# --- Main Loop ---
for i in range(1, num_steps):
    t_start = t_values[i-1]
    t_end = t_start + dt
    x_current = x_values[:, i-1]

    # --- 1. Define Setpoints (Desired State) ---
```

```
# Swap this section for your maneuver generator.  Steps, ramps,
# chirps, or data-driven trajectories all go here.
z_desired = 1.0
roll_desired = 0.0
pitch_desired = 0.0
yaw_desired = 0.0

# Simple step disturbance for attitude.  Use this block to mimic
# wind gusts or pilot stick inputs by forcing temporary setpoint jumps.
if 2.0 < t_start < 8.0:
    roll_desired = np.deg2rad(10.0)
    yaw_desired = np.deg2rad(30.0)
    pitch_desired = np.deg2rad(10.0)

# --- 2. Calculate Errors ---
# Note: We control z-position, but attitude angles.
# For a full controller, you'd control x,y position which would
# generate desired roll/pitch, but for this project this is simpler.

error_z = z_desired - x_current[2]       # z
error_roll = roll_desired - x_current[3]   # phi
error_pitch = pitch_desired - x_current[4] # theta
error_yaw = yaw_desired - x_current[5]   # psi

# PID controllers also need to dampen rates (D-term)
# A better way is a "PD" controller on angle and a "P" on rate.
# For simplicity, we'll use the PID as-is, but use angle *error*
# and also pass in the *negative of the current rate* to the D-term
# This is a common trick called "derivative on measurement"
#
# Here we'll stick to the textbook PID:
# We need to compute the derivative of the error.
# de/dt = (error - prev_error) / dt
# For angles: d(phi_d - phi)/dt = -d(phi)/dt = -p (if phi_d is constant)
# Let's modify the PID update to accept current_rate for the D-term

# A standard PID's D-term is Kd * (error - prev_error) / dt
# A PID with "Derivative on Measurement" is:
# u(t) = Kp*e(t) + Ki*integral(e(t)) - Kd * (y(t) - y(t-1))/dt
# For angles, this is approx: -Kd * (rate)

# Let's just use the simple PID from the class for now.
u_z = pid_z.update(error_z, dt)
u_phi = pid_roll.update(error_roll, dt)
u_theta = pid_pitch.update(error_pitch, dt)
```

```
    u_psi = pid_yaw.update(error_yaw, dt)

    # --- 3. Control Allocation ---

    # Add gravity compensation (feed-forward)
    # We must divide by (c_phi * c_theta) to account for tilt
    c_phi, c_theta = np.cos(x_current[3]), np.cos(x_current[4])
    T_hover = (m * g) / (c_phi * c_theta + 1e-6) # Add epsilon to avoid division by

    # Total thrust = hover thrust + altitude correction
    T = T_hover + u_z

    # Combine into virtual command vector [T, tau_phi, tau_theta, tau_psi]
    u_virtual = np.array([T, u_phi, u_theta, u_psi])

    # Calculate actual motor thrusts
    T_motors = M_inv @ u_virtual

    # Saturate motors
    T_motors_sat = np.clip(T_motors, T_min, T_max)
    T1, T2, T3, T4 = T_motors_sat
    motor_thrusts[:, i] = T_motors_sat

    # --- 4. Dynamics Step ---
    # Pass motor thrusts and constants as *constant arguments* for this step
    sol = solve_ivp(
        quadcopter_dynamics,
        [t_start, t_end],
        x_current,
        method='RK45',
        t_eval=[t_end], # Only get the final point
        args=(m, g, I, L, k_m, T1, T2, T3, T4)
    )

    # --- 5. Store Results ---
    t_values[i] = sol.t[-1]
    x_values[:, i] = sol.y[:, -1]

print("Simulation complete.")

# --- 6. Plot Results ---

# Plot Altitude (z) and Position (x, y)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
```

```
plt.plot(t_values, x_values[2, :], label='z (Altitude)')
plt.axhline(y=1.0, color='r', linestyle=':', label='z desired')
plt.title('Position')
plt.xlabel('Time (s)')
plt.ylabel('Position (m)')
plt.legend()
plt.grid(True)

# Plot Attitude (Roll, Pitch, Yaw)
plt.subplot(1, 2, 2)
plt.plot(t_values, np.rad2deg(x_values[3, :]), label=r'$\phi$ (Roll)')
plt.plot(t_values, np.rad2deg(x_values[4, :]), label=r'$\theta$ (Pitch)')
plt.plot(t_values, np.rad2deg(x_values[5, :]), label=r'$\psi$ (Yaw)')
plt.axhline(y=10.0, xmin=0.2, xmax=0.8, color='r', linestyle=':', label='Roll distur
plt.title('Attitude')
plt.xlabel('Time (s)')
plt.ylabel('Angle (degrees)')
plt.legend()
plt.grid(True)
plt.tight_layout()

# Plot Motor Thrusts
plt.figure(figsize=(10, 5))
plt.plot(t_values, motor_thrusts[0, :], label='Motor 1 (Rear)')
plt.plot(t_values, motor_thrusts[1, :], label='Motor 2 (Right)')
plt.plot(t_values, motor_thrusts[2, :], label='Motor 3 (Left)')
plt.plot(t_values, motor_thrusts[3, :], label='Motor 4 (Front)')
plt.title('Motor Thrusts')
plt.xlabel('Time (s)')
plt.ylabel('Thrust (N)')
plt.legend()
plt.grid(True)

plt.show()
```