IIT Hyderabad

Gravity Gradient Stabilization of Satellites

Submitted by:
Group: FlatEarthers
ME21BTECH11001 Abhishek Ghosh
ME21BTECH11002 Aditya Verma
ME21BTECH11012 C Bharath
ME21BTECH11033 Areeb Hussain

ME5060: Spacecraft Dynamics and Control
Mechanical Engineering
04.05.2025

Submitted to:
Dr. Vishnu Unni

Contents

Abstract

Satellites are important in various scenarios, including communication, ground navigation, and much more. A crucial part of ensuring a well built system is the control and stabilzation of satelliters. In this project, we explore the use of gravity gradient torque to stabilize a satellite's attitude in low Earth orbit (LEO). The method leverages Earth's non-uniform gravitational field acting on the satellite's extended body to provide a passive restoring torque. We also explore the Dzahnibekov effect in this project. Since onboard sensors provide noisy measurements of the satellite's orientation, we also implement a Kalman filter to estimate the satellite's true state from sensor data. This combination of passive control and active estimation provides a simple yet effective means to achieve Earth-pointing stability.

Aim

To stabilize a satellite in Low Earth Orbit (LEO) such that it maintains an Earth-pointing orientation using gravity gradient torque and accounts for sensor noise using a Kalman filter.

1 Introduction

When a satellite orbits Earth, the gravitational force acting on each of its parts is not equal. This difference, called the gravity gradient, causes a torque on the satellite. If the satellite is designed properly, this torque can help it naturally align itself with the local vertical (toward Earth). This method of stabilization does not use fuel or active actuators, which saves energy and reduces complexity.

Next, in free space, a rotating object can behave unpredictably due to what is known as the **Dzhanibekov Effect**. This effect shows that if a satellite rotates about its intermediate principal axis (the one with the medium moment of inertia), it can start flipping periodically even if no external torque is applied. This is due to the instability of rotation around that axis. This kind of motion can be seen in videos of flipping nuts or tools in microgravity environments.

To stabilise the satellite (in stable equillibrium), we use the gravity gradient torque, which provides a passive restoring force. It encourages rotation about the stable axis (usually the axis with the maximum or minimum moment of inertia) and aligns the satellite with the local vertical. This keeps the satellite Earth-pointing and reduces the chances of chaotic rotation without using complex active control systems.

2 Theory and Control Strategy

2.1 Gravity Gradient Torque – Detailed Derivation

We start with a rigid body (the satellite) orbiting Earth. The gravitational force acting on different parts of the satellite is not uniform due to its size. This results in a torque that can passively stabilize the satellite.

Let:

- $\vec{r}$ be the position vector from Earth's center to the satellite's center of mass (COM)
- $\vec{\rho}$ be the position vector from the satellite's COM to a mass element dm
- $\vec{R} = \vec{r} + \vec{\rho}$ be the position vector from Earth's center to the mass element dm

The gravitational force on dm is:

$$d\vec{F} = -\frac{\mu\, dm}{|\vec{R}|^3}\vec{R}$$

where $\mu = GM$ is the gravitational parameter of Earth.

The torque $\vec{T}$ about the center of mass is:

$$\vec{T} = \int \vec{\rho} \times d\vec{F}$$

Assuming $\|\vec{\rho}\| \ll \|\vec{r}\|$, we expand using a Taylor approximation:

$$\frac{1}{|\vec{R}|^3} \approx \frac{1}{r^3}\left(1 - 3\frac{\vec{r}\cdot\vec{\rho}}{r^2}\right)$$

So,

$$\vec{R} \approx \vec{r} + \vec{\rho} \Rightarrow d\vec{F} \approx -\frac{\mu\, dm}{r^3}\left(\vec{r} + \vec{\rho} - 3\frac{\vec{r}\cdot\vec{\rho}}{r^2}\vec{r}\right)$$

Substitute into the torque:

$$\vec{T} = -\frac{\mu}{r^3}\int \vec{\rho} \times \left(\vec{r} + \vec{\rho} - 3\frac{\vec{r}\cdot\vec{\rho}}{r^2}\vec{r}\right) dm$$

Breaking this down:

- $\int \vec{\rho} \times \vec{r}\, dm = \vec{r} \times \int \vec{\rho}\, dm = 0$ (since $\int \vec{\rho}\, dm = 0$ at COM)
- $\int \vec{\rho} \times \vec{\rho}\, dm = 0$
- The remaining term:

$$\int \vec{\rho} \times \left(-3\frac{\vec{r}\cdot\vec{\rho}}{r^2}\vec{r}\right) dm = -3\frac{1}{r^2}\int (\vec{r}\cdot\vec{\rho})\,(\vec{\rho}\times\vec{r})\, dm$$

This leads to the final expression (in terms of inertia tensor $\mathbf{I}$):

$$\vec{T} = 3\frac{\mu}{r^3}\vec{r} \times (\mathbf{I}\vec{r})$$

Interpretation: The torque aligns the satellite so that one of its principal axes points toward Earth. This method is passive and energy-efficient.

2.2 Sensors

- Gyroscope (IMU) : Measures angular velocity; essential but drifts over time.
- Magnetometer : Measures Earth's magnetic field; used for coarse attitude correction.
- Sun Sensor : Detects Sun's direction; accurate in sunlight.
- Earth Horizon Sensor : Detects Earth's limb for pitch/roll estimation.

2.3 Kalman Filter – Mathematical Overview

To estimate the satellite's attitude from noisy sensor data, we use a Kalman Filter. It works in a predict-correct cycle using system and measurement models.

Let:

- $\vec{x}_k$ be the state vector (e.g., orientation, angular velocity)
- $\vec{u}_k$ be the control input (torque; often zero for passive control)
- $\vec{z}_k$ be the measurement vector (sensor data)

System dynamics:

$$\vec{x}_{k+1} = \mathbf{A}\vec{x}_k + \mathbf{B}\vec{u}_k + \vec{w}_k$$

Measurement model:

$$\vec{z}_k = \mathbf{H}\vec{x}_k + \vec{v}_k$$

Here:

- $\mathbf{A}$ is the state transition matrix
- $\mathbf{B}$ is the control matrix
- $\mathbf{H}$ is the measurement matrix
- $\vec{w}_k$, $\vec{v}_k$ are zero-mean Gaussian noise terms (process and measurement)

Kalman Filter steps:
Prediction Step

$$\vec{x}_{k|k-1} = \mathbf{A}\vec{x}_{k-1|k-1}$$

$$\mathbf{P}_{k|k-1} = \mathbf{A}\mathbf{P}_{k-1|k-1}\mathbf{A}^T + \mathbf{Q}$$

Update Step

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^T \left(\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R}\right)^{-1}$$

$$\vec{x}_{k|k} = \vec{x}_{k|k-1} + \mathbf{K}_k \left(\vec{z}_k - \mathbf{H}\vec{x}_{k|k-1}\right)$$

$$\mathbf{P}_{k|k} = \left(\mathbf{I} - \mathbf{K}_k\mathbf{H}\right)\mathbf{P}_{k|k-1}$$

Where:

- $\mathbf{P}$ is the estimation error covariance
- $\mathbf{Q}$, $\mathbf{R}$ are process and measurement noise covariances
- $\mathbf{K}_k$ is the Kalman gain

Why Kalman Filter? It fuses noisy data from different sensors (e.g., gyroscopes, magnetometers) and gives an accurate estimate of the satellite's attitude. This helps reduce drift and filter out noise for reliable control.

3 Tasks done in this project

3.1 Gravity Gradient and Dzhanibekov Effect

Methodology

This simulation explores the attitude dynamics of a rigid body in orbit under the influence of gravity gradient torque, along with the aim of reproducing the Dzhanibekov effect—a phenomenon observed in asymmetric bodies where rotation about the intermediate principal axis leads to unstable motion.

- **Orbit Assumption:** A perfectly circular low Earth orbit at an altitude of 500 km is assumed.
- **Inertia Tensor:** The spacecraft is modeled as a rigid body with different moments of inertia along each principal axis: $I_x = 200$, $I_y = 150$, $I_z = 100$ kg $\cdot$ m^2. This asymmetry is essential for observing the Dzhanibekov instability, and also to satisfy the conditions of Gravity Gradient Stabilisation.
- **Initial Conditions:** The body is given a small initial angular velocity, and its orientation is represented by a unit quaternion slightly perturbed from the identity.
- **Equations of Motion:** The rotational dynamics include:
  - Euler's equations: $\dot{\omega} = I^{-1}(T_{\text{gg}} - \omega \times I\omega)$
  - Quaternion kinematics: $\dot{q} = \frac{1}{2} q \otimes \omega_{\text{quat}}$
- **Gravity Gradient Torque:** Computed as:

$$T_{\text{gg}} = 3\omega_o^2 \cdot (\vec{z}_{\text{body}} \times I\vec{z}_{\text{body}})$$

  where $\vec{z}_{\text{body}}$ is the nadir vector transformed into the body frame using quaternion rotation.
- **Integration:** The system is integrated using `scipy`'s `solve_ivp`, with quaternion normalization enforced to prevent drift.
- **Output:** The final quaternion states are converted into Euler angles (roll, pitch, yaw) for intuitive interpretation. These angles are plotted over time to reveal attitude evolution.

3.2 Applying Kalman Filter

In this part of the simulation, we address the problem of estimating the angular velocities (ω_x, ω_y, and ω_z) using noisy measurements. The goal is to filter out the noise in the measurements using a Kalman filter, which provides an optimal estimate of the state (in this case, the angular velocities) over time.

The Kalman filter is a recursive algorithm that estimates the state of a dynamic system by minimizing the mean squared error between the predicted state and the actual measurement. The filter uses two steps in each iteration:

- **Prediction**: It predicts the state of the system based on the previous state estimate.

- **Update**: The filter corrects the prediction by incorporating the new measurement, adjusting the estimated state using the Kalman gain.

In this case, the noisy measurements of angular velocities (ω_x, ω_y, and ω_z) are used as inputs to the Kalman filter. The filter is designed to estimate the true values of the angular velocities by reducing the impact of noise.

Process

The Kalman filter is applied individually to the angular velocities ω_x, ω_y, and ω_z. The measurement noise is assumed to be Gaussian, with a standard deviation defined by `noise_std`. This noise is added to the true values of the angular velocities, simulating real-world measurement errors.

The filter uses the following parameters:

- **Q** (process noise covariance): This value represents the uncertainty in the process model, i.e., how much the system's true state might change between measurements.
- **R** (measurement noise covariance): This value represents the uncertainty in the measurements, which in our case is the standard deviation squared of the noise added to the measurements.

The filter runs for each angular velocity component (ω_x, ω_y, and ω_z) to estimate their values over time. The results are compared with the true values of the angular velocities, allowing us to visualize how well the Kalman filter is able to reduce the noise in the measurements.

Results

The comparison is plotted for each component (ω_x, ω_y, ω_z) showing the true angular velocity, the noisy measured values, and the filtered estimate from the Kalman filter. The filter effectively smooths out the noise, providing an estimate that closely tracks the true values despite the presence of measurement errors.

This part demonstrates the effectiveness of the Kalman filter in estimating dynamic system states with noisy data, and it shows how the filter can be applied to spacecraft attitude dynamics for improving measurement accuracy.

3.3 Introducing Energy Dissipation

In this part of the simulation, we introduce energy dissipation into the system to model damping effects on the satellite's angular velocities and orientation. The presence of damping simulates real-world scenarios where energy is lost due to friction or other resistive forces, such as atmospheric drag or internal resistance in the satellite's control system. If this is not included, then the oscillations or disturbances in the satellite do not die out.

We model the damping forces by adding damping torques to the angular momentum equations of the satellite. The damping torques are proportional to the angular velocities in each of the three axes (x, y, z), with damping coefficients c_x, c_y, and c_z representing the resistance along each axis.

Process

The equations of motion for the satellite are modified to include damping effects. Specifically, the gravitational gradient torques, which arise due to the interaction of the satellite with Earth's gravity field, are combined with damping torques. The damping torque terms are given by:

$$M_x = M_{x_{gg}} - c_x \cdot \omega_x$$
$$M_y = M_{y_{gg}} - c_y \cdot \omega_y$$
$$M_z = M_{z_{gg}} - c_z \cdot \omega_z$$

where: - $M_{x_{gg}}$, $M_{y_{gg}}$, and $M_{z_{gg}}$ are the gravity gradient torques, which depend on the satellite's orientation and the gravitational field of the Earth. - c_x, c_y, and c_z are the damping coefficients along the respective axes. - ω_x, ω_y, and ω_z are the angular velocities along the x, y, and z axes.

These damping torques are subtracted from the gravity gradient torques, reducing the angular velocities over time.

The system dynamics are governed by the following equations of motion: - The angular accelerations are calculated based on the torques and the satellite's moments of inertia:

$$\dot{\omega}_x = \frac{M_x - (I_z - I_y)\omega_y\omega_z}{I_x}$$
$$\dot{\omega}_y = \frac{M_y - (I_x - I_z)\omega_z\omega_x}{I_y}$$
$$\dot{\omega}_z = \frac{M_z - (I_y - I_x)\omega_x\omega_y}{I_z}$$

- The kinematic equations for the satellite's orientation are given by:

$$\dot{\phi} = \omega_x + \sin(\phi)\tan(\theta)\omega_y + \cos(\phi)\tan(\theta)\omega_z$$
$$\dot{\theta} = \cos(\phi)\omega_y - \sin(\phi)\omega_z$$
$$\dot{\psi} = \frac{\sin(\phi)}{\cos(\theta)}\omega_y + \frac{\cos(\phi)}{\cos(\theta)}\omega_z$$

where ϕ, θ, and ψ represent the roll, pitch, and yaw angles of the satellite, respectively.

Results

The results of the simulation show the evolution of both the angular velocities (ω_x, ω_y, ω_z) and the Euler angles (ϕ, θ, ψ) over time. The presence of damping reduces the angular velocities gradually, stabilizing the satellite's rotation. The angular velocities approach zero as energy is dissipated due to the damping torques.

The plots illustrate the time evolution of the angular velocities and Euler angles. As expected, the damping effect reduces the oscillations, and the system reaches a steady state with minimized rotational energy.

This part demonstrates the role of energy dissipation in satellite dynamics, highlighting how damping forces influence the satellite's attitude and angular velocities over time. The introduction of damping offers a more realistic model of the satellite's behavior in space, where resistive forces contribute to the gradual reduction of angular momentum.

3.4 Apply Kalman Filter with Energy

In this part, we introduce the Extended Kalman Filter (EKF) to estimate the satellite's state in the presence of noisy measurements, while also considering the energy dissipation effects. The satellite's motion is still influenced by the gravitational gradient torque and the damping torque, but now we also include noise in the measurements of the satellite's states.

Satellite Dynamics with Energy Damping

The dynamics of the satellite are modeled similarly to Part 2, with the addition of damping torque modeled as a linear function of angular velocity. The torque components are computed as:

$$M_x = \left(-\frac{3\mu}{R^3}\right)(I_z - I_y)\sin(\phi)\cos(\phi)\cos(\theta)^2$$

$$M_y = \left(-\frac{3\mu}{R^3}\right)(I_z - I_x)\sin(\theta)\cos(\phi)\cos(\theta)$$

$$M_z = \left(-\frac{3\mu}{R^3}\right)(I_y - I_x)\sin(\theta)\sin(\phi)\cos(\theta)$$

$$M_{\text{damping}} = -c \cdot \omega$$

where c is the damping coefficient and ω is the angular velocity. The total torque is the sum of gravitational gradient torque and damping torque. The angular acceleration is then computed as:

$$\omega_{\text{dot}} = I^{-1}(M_{\text{total}} - \omega \times (I \cdot \omega))$$

The kinematic equations for the satellite's Euler angles (ϕ, θ, and ψ) are the same as in Part 2.

Noisy Measurements

To simulate real-world measurements, noise is added to the true states of the satellite. This noise is Gaussian in nature and has different standard deviations for the angles and angular velocities:

$$\text{Noise std} = [0.001, 0.001, 0.001, 0.0005, 0.0005, 0.0005]$$

Extended Kalman Filter (EKF)

The EKF is used to estimate the satellite's state from the noisy measurements. The filter operates in two steps:

- **Prediction Step:** The state is predicted using the satellite's dynamics and the previous state estimate. The Jacobian matrix F is computed to linearize the system around the current state estimate.

- **Update Step:** The predicted state is updated using the Kalman gain K and the measurement residual. The Kalman gain is computed as:

$$K = P_{\text{pred}} H^T (H P_{\text{pred}} H^T + R_{\text{cov}})^{-1}$$

where P_{pred} is the predicted covariance matrix, and H is the measurement matrix.

The updated state estimate is then:

$$x_{\text{est}} = x_{\text{pred}} + K \cdot (z - x_{\text{pred}})$$

Results and Comparison

The true states of the satellite (angular velocities and Euler angles) are plotted alongside the EKF estimates. The comparison shows how well the EKF can track the true state despite the noisy measurements. The estimates converge towards the true values over time, demonstrating the effectiveness of the Kalman filter in state estimation.

Conclusion

The Kalman Filter successfully filters out noise from the measurements and provides accurate estimates of the satellite's state. By including damping and noisy measurements, this approach simulates more realistic satellite dynamics and estimation techniques.

4 Plots and Inference

4.1 Part 1: Gravity Gradient and Dzhanibekov Effect

- **Plot Overview:** The Euler angles (Roll, Pitch, Yaw) are plotted over time. The satellite starts with a small angular impulse and evolves under the influence of gravity gradient torque alone.
- **Key Inferences:**
  - The attitude does not stabilize but instead exhibits large periodic oscillations, especially in the pitch and yaw axes.
  - The behavior reflects the **Dzhanibekov effect**, where bodies with asymmetric inertia undergo flipping or tumbling dynamics.
- **Reason:** The gravity gradient torque is a conservative force and does not dissipate energy. Due to the unequal moments of inertia ($I_x \neq I_y \neq I_z$), the satellite exhibits energy exchange between rotational modes, leading to unstable but bounded oscillations.

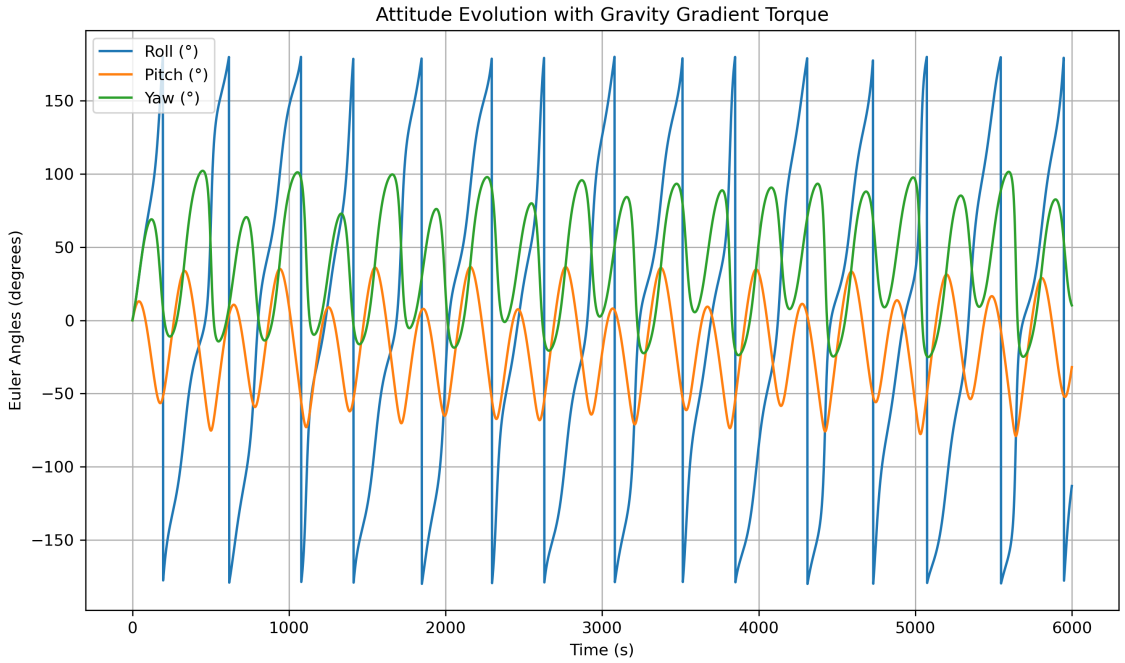

Figure 1:

4.2 Part 2: Kalman Filter Estimation of Angular Velocities

- **Plot Overview:** Three subplots are shown for the angular velocity components ω_x, ω_y, and ω_z. Each subplot compares:

  1. The true angular velocity (from simulation),
  2. The noisy measured signal (with Gaussian noise of standard deviation 0.002 rad/s),
  3. The estimate produced by a scalar 1D Kalman filter.

- **Key Inferences:**

  – The Kalman filter smooths out high-frequency noise from the measured signal and closely follows the true angular velocity.

  – The filtered estimate avoids large deviations and maintains a trajectory that is more accurate than the noisy measurements.

- **Reason:** The Kalman filter operates as an optimal recursive estimator in the presence of Gaussian noise. It minimizes the mean squared error by balancing confidence between the current estimate and the incoming measurement, controlled via the process noise covariance Q and the measurement noise covariance R. In this simulation, a small $Q = 10^{-6}$ and $R = (0.002)^2$ allowed the filter to suppress noise while maintaining good responsiveness.

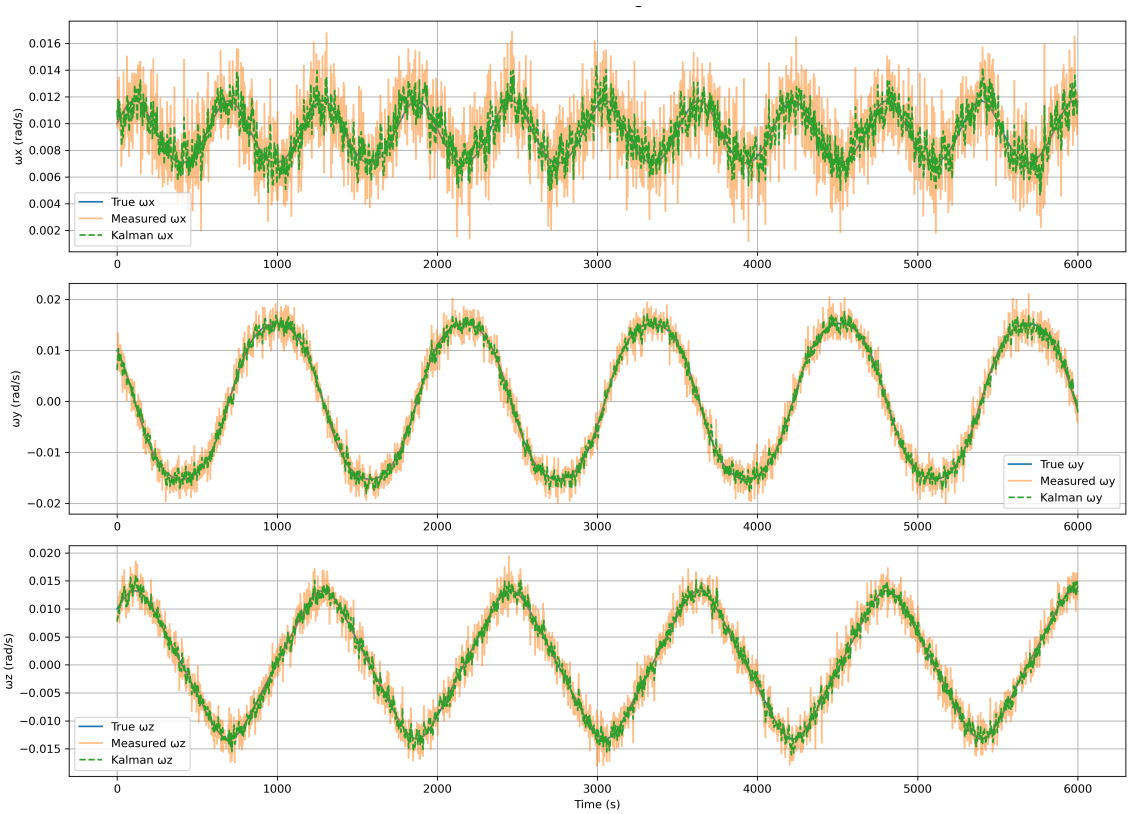

Figure 2:

4.3 Part 3: Introducing Energy Dissipation

- **Plot Overview:** Two subplots are shown. The first displays the time evolution of angular velocities ω_x, ω_y, and ω_z under the influence of gravity gradient torque and damping. The second plot shows the corresponding evolution of the Euler angles (Roll ϕ, Pitch θ, Yaw ψ) in degrees.
- **Key Inferences:**
  - All angular velocity components decay toward zero over time, indicating the dissipation of rotational energy.
  - Euler angles settle to steady values, showing that the satellite reaches a stable orientation.
  - The attitude does not oscillate indefinitely as in the undamped case, but instead stabilizes due to damping.
- **Reason:** The inclusion of damping torques proportional to angular velocity simulates the effect of internal energy loss mechanisms (e.g., magnetic hysteresis, structural friction). These damping torques reduce the kinetic energy over time, allowing the satellite to naturally align with the local gravity gradient field. This leads to a stable configuration with minimal rotational motion.

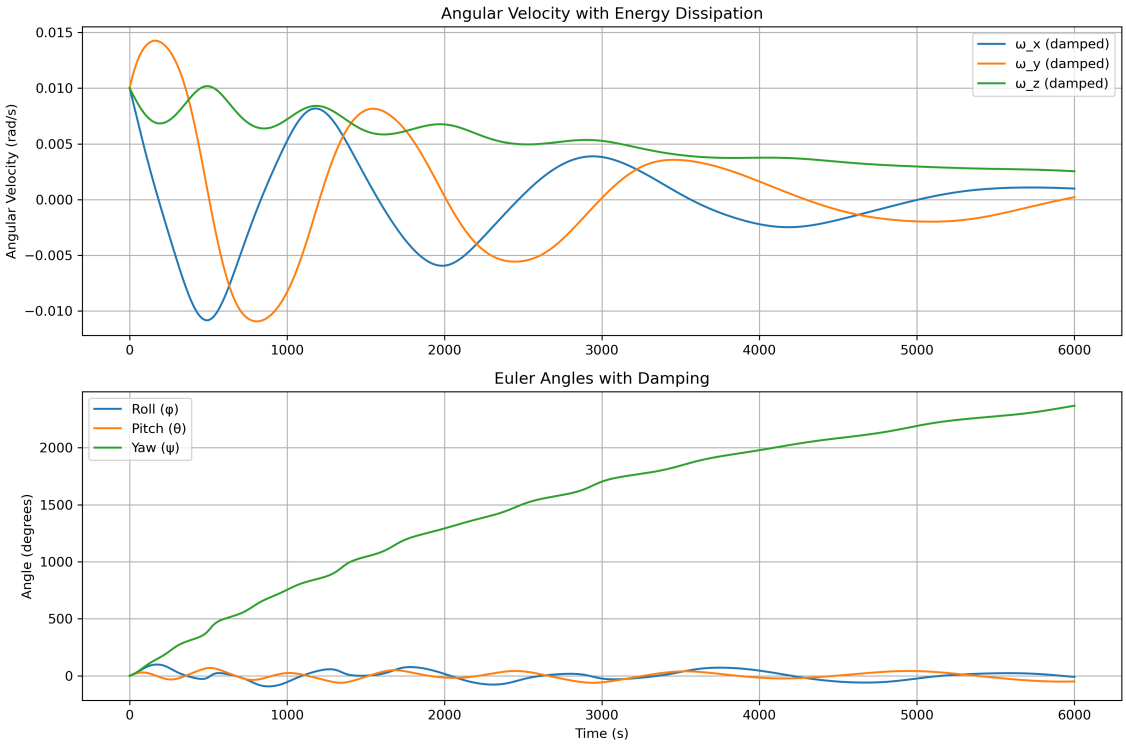

Figure 3:

4.4 Part 4: Introducing Kalman Filter After Energy Dissipation

- **Plot Overview:** The figure includes two subplots. The top subplot shows the true and estimated angular velocities using the Extended Kalman Filter (EKF), while the bottom subplot compares the true and EKF-estimated Euler angles (roll ϕ, pitch θ, and yaw ψ), all plotted over time.
- **Key Inferences:**
  - EKF estimates track the true angular velocities and Euler angles closely, even in the presence of measurement noise.
  - The use of EKF significantly reduces the effect of noise seen in raw measurements and improves attitude estimation accuracy.
  - Damping still ensures energy dissipation and convergence, but EKF enhances state awareness despite noisy observations.
- **Reason:** The EKF uses a prediction-correction mechanism, where the satellite dynamics propagate the state forward and incoming noisy measurements are used to correct the prediction. Even with nonlinear dynamics and additive Gaussian noise, the EKF provides near-optimal state estimates. This is particularly useful for spacecraft attitude determination, where direct measurement of the true state is typically infeasible and must be inferred from sensor data.

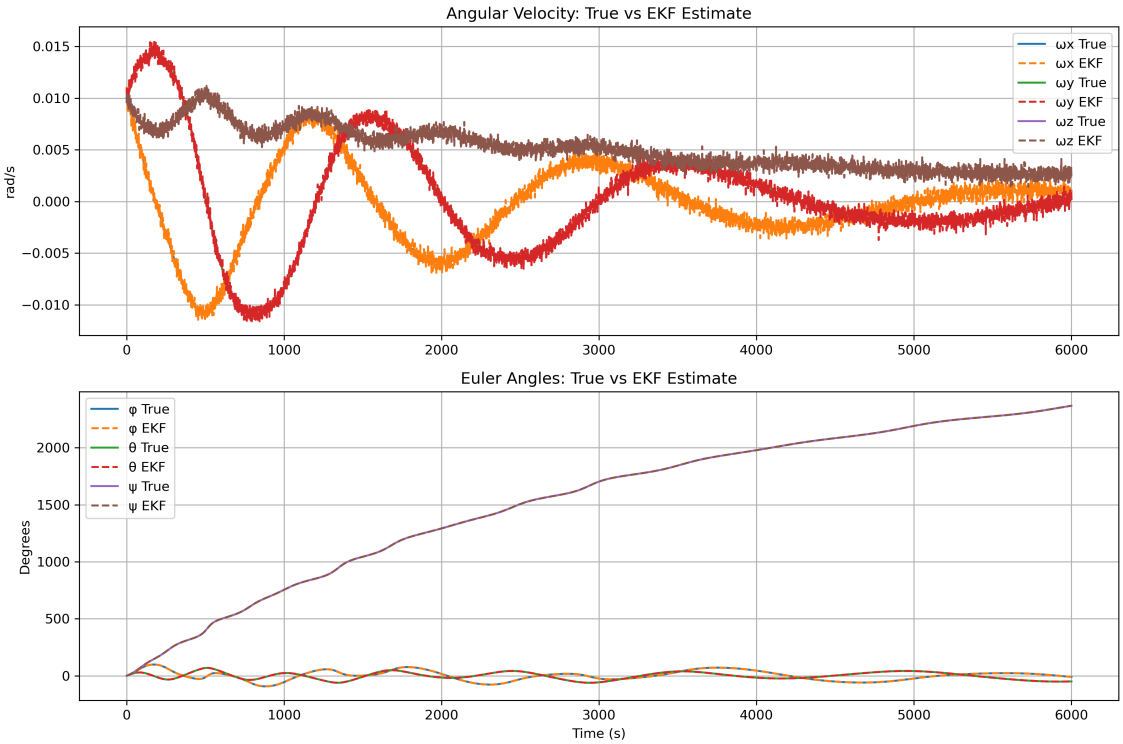

Figure 4:

References

1 The code used to generate plots is in the zip file.

Appendix

Part 1: Gravity Gradient and Dzhanibekov Effect

```
%reset -f
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# === Constants ===
mu = 3.986e14          # Earth's gravitational parameter (m^3/s^2)
Re = 6371e3            # Earth's radius (m)
h = 500e3              # Orbit altitude (m)
r = Re + h             # Orbital radius (m)
omega_orbit = np.sqrt(mu / r**3)  # Mean motion (rad/s)

# === Inertia matrix ===
Ix, Iy, Iz = 200, 150, 100  # kg·m^2
I = np.diag([Ix, Iy, Iz])
I_inv = np.linalg.inv(I)

# === Initial Conditions ===
w0 = np.array([0.01, 0.01, 0.01])      # Small impulse about (rad/s)
q0 = np.array([1.0, 0.001, 0.0, 0.0])   # Initial quaternion (identity)
state0 = np.concatenate((w0, q0))     # Initial state vector

# === Quaternion Utilities ===
def quat_mult(q, r):
    w0, x0, y0, z0 = q
    w1, x1, y1, z1 = r
    return np.array([
        w0*w1 - x0*x1 - y0*y1 - z0*z1,
        w0*x1 + x0*w1 + y0*z1 - z0*y1,
        w0*y1 - x0*z1 + y0*w1 + z0*x1,
        w0*z1 + x0*y1 - y0*x1 + z0*w1
    ])

def quat_derivative(q, w):
    w_quat = np.concatenate(([0], w))
    return 0.5 * quat_mult(q, w_quat)

# === Gravity Gradient Torque ===
def gravity_gradient_torque(I, q):
    z_eci = np.array([0, 0, -1])
    q_conj = np.array([q[0], -q[1], -q[2], -q[3]])
    z_body = quat_mult(quat_mult(q_conj, np.concatenate(([0], z_eci))),
        q)[1:]
    return 3 * omega_orbit**2 * np.cross(z_body, I @ z_body)

# === Dynamics ===
def dynamics(t, state):
    w = state[0:3]
    q = state[3:7]

    Tgg = gravity_gradient_torque(I, q)
    w_dot = I_inv @ (Tgg - np.cross(w, I @ w))
    q_dot = quat_derivative(q, w)
```

```
53
54      return np.concatenate((w_dot, q_dot))
55
56  # === Normalized Integration Wrapper ===
57  def integrate_with_quat_normalization(fun, t_span, y0, t_eval):
58      def wrapped_fun(t, y):
59          dydt = fun(t, y)
60          dydt[3:7] -= (np.dot(y[3:7], dydt[3:7]) / np.dot(y[3:7], y
                [3:7])) * y[3:7]  # keep q unit norm
61          return dydt
62
63      sol = solve_ivp(wrapped_fun, t_span, y0, t_eval=t_eval, rtol=1e-8,
            atol=1e-10)
64
65      # Normalize quaternion after each step
66      for i in range(sol.y.shape[1]):
67          q = sol.y[3:7, i]
68          sol.y[3:7, i] = q / np.linalg.norm(q)
69      return sol
70
71  # === Run Simulation ===
72  t_span = (0, 6000)                              # Simulate for ~5.5 hours
73  t_eval = np.linspace(*t_span, 3000)              # 4000 time points
74  sol = integrate_with_quat_normalization(dynamics, t_span, state0,
        t_eval)
75
76  # === Convert to Euler Angles ===
77  def quaternion_to_euler(q):
78      w, x, y, z = q
79      roll = np.arctan2(2*(w*x + y*z), 1 - 2*(x**2 + y**2))
80      pitch = np.arcsin(2*(w*y - z*x))
81      yaw = np.arctan2(2*(w*z + x*y), 1 - 2*(y**2 + z**2))
82      return np.degrees([roll, pitch, yaw])
83
84  euler_angles = np.array([quaternion_to_euler(sol.y[3:7, i]) for i in
        range(sol.y.shape[1])])
85
86  # === Plot ===
87  plt.figure(figsize=(10, 6))
88  plt.plot(sol.t, euler_angles[:, 0], label='Roll (°)')
89  plt.plot(sol.t, euler_angles[:, 1], label='Pitch (°)')
90  plt.plot(sol.t, euler_angles[:, 2], label='Yaw (°)')
91  plt.xlabel("Time (s)")
92  plt.ylabel("Euler Angles (degrees)")
93  plt.title("Attitude Evolution with Gravity Gradient Torque")
94  plt.legend()
95  plt.grid(True)
96  plt.tight_layout()
97  plt.show()
```

Part 2: Applying Kalman Filter

```
1  %reset -f
2  import numpy as np
3  from scipy.integrate import solve_ivp
4  import matplotlib.pyplot as plt
```

```
5
6  # === Constants ===
7  mu = 3.986e14          # Earth's gravitational parameter (m^3/s^2)
8  Re = 6371e3            # Earth's radius (m)
9  h = 500e3              # Orbit altitude (m)
10 r = Re + h             # Orbital radius (m)
11 omega_orbit = np.sqrt(mu / r**3)  # Mean motion (rad/s)
12
13 # === Inertia matrix ===
14 Ix, Iy, Iz = 200, 150, 100  # kg·m^2
15 I = np.diag([Ix, Iy, Iz])
16 I_inv = np.linalg.inv(I)
17
18 # === Initial Conditions ===
19 w0 = np.array([0.01, 0.01, 0.01])      # Small impulse (rad/s)
20 q0 = np.array([1.0, 0.001, 0.0, 0.0])   # Initial quaternion (identity)
21 state0 = np.concatenate((w0, q0))     # Initial state vector
22
23 # === Quaternion Utilities ===
24 def quat_mult(q, r):
25     w0, x0, y0, z0 = q
26     w1, x1, y1, z1 = r
27     return np.array([
28         w0*w1 - x0*x1 - y0*y1 - z0*z1,
29         w0*x1 + x0*w1 + y0*z1 - z0*y1,
30         w0*y1 - x0*z1 + y0*w1 + z0*x1,
31         w0*z1 + x0*y1 - y0*x1 + z0*w1
32     ])
33
34 def quat_derivative(q, w):
35     w_quat = np.concatenate(([0], w))
36     return 0.5 * quat_mult(q, w_quat)
37
38 # === Gravity Gradient Torque ===
39 def gravity_gradient_torque(I, q):
40     z_eci = np.array([0, 0, -1])
41     q_conj = np.array([q[0], -q[1], -q[2], -q[3]])
42     z_body = quat_mult(quat_mult(q_conj, np.concatenate(([0], z_eci))),
           q)[1:]
43     return 3 * omega_orbit**2 * np.cross(z_body, I @ z_body)
44
45 # === Dynamics ===
46 def dynamics(t, state):
47     w = state[0:3]
48     q = state[3:7]
49
50     Tgg = gravity_gradient_torque(I, q)
51     w_dot = I_inv @ (Tgg - np.cross(w, I @ w))
52     q_dot = quat_derivative(q, w)
53
54     return np.concatenate((w_dot, q_dot))
55
56 # === Normalized Integration Wrapper ===
57 def integrate_with_quat_normalization(fun, t_span, y0, t_eval):
58     def wrapped_fun(t, y):
59         dydt = fun(t, y)
60         dydt[3:7] -= (np.dot(y[3:7], dydt[3:7]) / np.dot(y[3:7], y
              [3:7])) * y[3:7]  # keep q unit norm
```

```
        return dydt

    sol = solve_ivp(wrapped_fun, t_span, y0, t_eval=t_eval, rtol=1e-8,
        atol=1e-10)

    # Normalize quaternion after each step
    for i in range(sol.y.shape[1]):
        q = sol.y[3:7, i]
        sol.y[3:7, i] = q / np.linalg.norm(q)
    return sol

# === Run Simulation ===
t_span = (0, 6000)                              # Simulate for ~5.5 hours
t_eval = np.linspace(*t_span, 3000)              # 3000 time points
sol = integrate_with_quat_normalization(dynamics, t_span, state0,
    t_eval)

# Extract true angular velocities
t = sol.t
 x  = sol.y[0, :]
 y  = sol.y[1, :]
 z  = sol.y[2, :]

# === Convert to Euler Angles ===
def quaternion_to_euler(q):
    w, x, y, z = q
    roll = np.arctan2(2*(w*x + y*z), 1 - 2*(x**2 + y**2))
    pitch = np.arcsin(2*(w*y - z*x))
    yaw = np.arctan2(2*(w*z + x*y), 1 - 2*(y**2 + z**2))
    return np.degrees([roll, pitch, yaw])

euler_angles = np.array([quaternion_to_euler(sol.y[3:7, i]) for i in
    range(sol.y.shape[1])])

# === Plot ===
plt.figure(figsize=(10, 6))
plt.plot(sol.t, euler_angles[:, 0], label='Roll (°)')
plt.plot(sol.t, euler_angles[:, 1], label='Pitch (°)')
plt.plot(sol.t, euler_angles[:, 2], label='Yaw (°)')
plt.xlabel("Time (s)")
plt.ylabel("Euler Angles (degrees)")
plt.title("Attitude Evolution with Gravity Gradient Torque")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()


# Simulate noisy measurements
np.random.seed(42)
noise_std = 0.002  # rad/s
#  x ,  y ,  z  = np.zeros(steps),np.zeros(steps),np.zeros(steps)
 x_meas  =  x  + np.random.normal(0, noise_std, len( x ))
 y_meas  =  y  + np.random.normal(0, noise_std, len( y ))
 z_meas  =  z  + np.random.normal(0, noise_std, len( z ))

def kalman_filter(z, Q=1e-6, R=noise_std**2):
    """Simple Kalman Filter for 1D signals"""
```

```
    n = len(z)
    x_est = np.zeros(n)
    P = 1.0  # initial estimation error covariance
    x = z[0]  # initial estimate

    for i in range(n):
        # Prediction
        P = P + Q

        # Kalman Gain
        K = P / (P + R)

        # Update
        x = x + K * (z[i] - x)
        P = (1 - K) * P

        x_est[i] = x

    return x_est

# Apply Kalman Filter
 x_kal   = kalman_filter( x_meas  )
 y_kal   = kalman_filter( y_meas  )
 z_kal   = kalman_filter( z_meas  )

# Plot comparison
plt.figure(figsize=(14, 10))

#  x
plt.subplot(3, 1, 1)
plt.plot(t,  x , label='True  x ', linewidth=1.5)
plt.plot(t,  x_meas  , label='Measured  x ', alpha=0.5)
plt.plot(t,  x_kal  , label='Kalman  x ', linestyle='--')
plt.ylabel(' x  (rad/s)')
plt.legend()
plt.grid()

#  y
plt.subplot(3, 1, 2)
plt.plot(t,  y , label='True  y ', linewidth=1.5)
plt.plot(t,  y_meas  , label='Measured  y ', alpha=0.5)
plt.plot(t,  y_kal  , label='Kalman  y ', linestyle='--')
plt.ylabel(' y  (rad/s)')
plt.legend()
plt.grid()

#  z
plt.subplot(3, 1, 3)
plt.plot(t,  z , label='True  z ', linewidth=1.5)
plt.plot(t,  z_meas  , label='Measured  z ', alpha=0.5)
plt.plot(t,  z_kal  , label='Kalman  z ', linestyle='--')
plt.xlabel('Time (s)')
plt.ylabel(' z  (rad/s)')
plt.legend()
plt.grid()

plt.tight_layout()
```

```
plt.suptitle('Kalman Filter Estimation of Angular Velocities', fontsize
    =16, y=1.02)
plt.show()
```

Part 3: Introducing Energy Dissipation

```
%reset -f
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Constants and parameters
mu = 3.986004418e14
R_earth = 6371e3
altitude = 500e3
R = R_earth + altitude

I_x, I_y, I_z = 100.0, 150.0, 200.0

# Damping coefficients
c_x, c_y, c_z = 0.05, 0.05, 0.05

# Initial state [  ,   ,   , x ,  y ,  z ]
initial_angles = np.array([0.0, 0.0, 0.0])
initial_omega = np.array([0.01, 0.01, 0.01])
state0 = np.concatenate((initial_angles, initial_omega))

# Time array
t = np.linspace(0, 6000, 6001)

def satellite_dynamics_damped(state, t):
    # Unpack state
      ,   ,   , x ,  y ,  z  = state

    # Gravity gradient torque
    Mx_gg = (-3 * mu / R**3) * (I_z - I_y) * np.sin(  )*np.cos(  )*np.
        cos(  )**2
    My_gg = (-3 * mu / R**3) * (I_z - I_x) * np.sin(  )*np.cos(  )*np.
        cos(  )
    Mz_gg = (-3 * mu / R**3) * (I_y - I_x) * np.sin(  )*np.sin(  )*np.
        cos(  )

    # Damping torque
    Mx = Mx_gg - c_x *  x
    My = My_gg - c_y *  y
    Mz = Mz_gg - c_z *  z

    # Angular acceleration
    d  x  = (Mx - (I_z - I_y)* y * z ) / I_x
    d  y  = (My - (I_x - I_z)* z * x ) / I_y
    d  z  = (Mz - (I_y - I_x)* x * y ) / I_z

    # Kinematic equations
    d   =  x  + np.sin(  )*np.tan(  )* y  + np.cos(  )*np.tan(  )* z
    d   = np.cos(  )* y  - np.sin(  )* z
    d   = (np.sin(  )/np.cos(  ))* y  + (np.cos(  )/np.cos(  ))* z
```

```
48
49      return [d  , d  , d  , d x , d y , d z]
50
51  # Solve ODE
52  solution_damped = odeint(satellite_dynamics_damped, state0, t)
53
54  # Extract components
55   _d ,  _d ,  _d  = solution_damped[:, 0], solution_damped[:, 1],
       solution_damped[:, 2]
56   x_d ,  y_d ,  z_d  = solution_damped[:, 3], solution_damped[:, 4],
       solution_damped[:, 5]
57
58  # Plot
59  plt.figure(figsize=(12, 8))
60
61  plt.subplot(2, 1, 1)
62  plt.plot(t,  x_d , label=' _x  (damped)')
63  plt.plot(t,  y_d , label=' _y  (damped)')
64  plt.plot(t,  z_d , label=' _z  (damped)')
65  plt.title('Angular Velocity with Energy Dissipation')
66  plt.ylabel('Angular Velocity (rad/s)')
67  plt.legend()
68  plt.grid()
69
70  plt.subplot(2, 1, 2)
71  plt.plot(t, np.degrees( _d ), label='Roll (  )')
72  plt.plot(t, np.degrees( _d ), label='Pitch (  )')
73  plt.plot(t, (np.degrees( _d )), label='Yaw (  )')
74  plt.title('Euler Angles with Damping')
75  plt.xlabel('Time (s)')
76  plt.ylabel('Angle (degrees)')
77  plt.legend()
78  plt.grid()
79
80  plt.tight_layout()
81  plt.show()
```

Part 4: Apply Kalman Filter

```
1  %reset -f
2  import numpy as np
3  import matplotlib.pyplot as plt
4  from scipy.integrate import odeint
5
6  # Constants
7  mu = 3.986004418e14  # Earth's gravitational parameter (m^3/s^2)
8  R_earth = 6371e3     # Earth's radius (m)
9  altitude = 500e3     # Satellite altitude (m)
10 R = R_earth + altitude
11
12 # Inertia (kg·m²)
13 I_x, I_y, I_z = 100, 150, 200
14 I = np.diag([I_x, I_y, I_z])
15
16 # Initial state: [  ,  ,  , x , y , z ]
17 state0 = np.array([0, 0, 0, 0.01, 0.01, 0.01])
```

```
18
19 # Time vector
20 t = np.linspace(0, 6000, 6001)  # 100 minutes at 1s interval
21 dt = t[1] - t[0]
22
23 # Satellite dynamics with damping
24 def satellite_dynamics_damped(state, t):
25       ,   ,   ,  x ,  y ,  z  = state
26        = np.array([ x ,  y ,  z ])
27
28     # Gravity gradient torque
29     Mx = (-3 * mu / R**3) * (I_z - I_y) * np.sin(  )*np.cos(  )*np.cos(
           )**2
30     My = (-3 * mu / R**3) * (I_z - I_x) * np.sin(  )*np.cos(  )*np.cos(
           )
31     Mz = (-3 * mu / R**3) * (I_y - I_x) * np.sin(  )*np.sin(  )*np.cos(
           )
32     M = np.array([Mx, My, Mz])
33
34     # Damping torque (simple linear model)
35     damping_coeff = 0.05
36     M_damping = -damping_coeff *
37
38     # Total torque
39     M_total = M + M_damping
40
41     # Angular acceleration
42      _dot  = np.linalg.inv(I) @ (M_total - np.cross(  , I @  ))
43
44     # Kinematic equations
45     d  =  x  + np.sin(  )*np.tan(  )* y  + np.cos(  )*np.tan(  )* z
46     d  = np.cos(  )* y  - np.sin(  )* z
47     d  = (np.sin(  )/np.cos(  ))* y  + (np.cos(  )/np.cos(  ))* z
48
49     return [d  , d  , d  , * _dot ]
50
51 # Simulate true dynamics
52 true_states = odeint(satellite_dynamics_damped, state0, t)
53  _true  ,  _true  ,  _true  = true_states[:, 0], true_states[:, 1],
      true_states[:, 2]
54  x_true  ,  y_true  ,  z_true  = true_states[:, 3], true_states[:, 4],
      true_states[:, 5]
55
56 # Simulate noisy measurements
57 np.random.seed(0)
58 noise_std = np.array([0.001]*3 + [0.0005]*3)  # angles + angular
      velocity
59 measurements = true_states + np.random.normal(0, noise_std, true_states
      .shape)
60
61 # EKF initialization
62 n = 6
63 x_est = np.copy(state0)
64 P = np.eye(n) * 0.01
65 Q = np.eye(n) * 1e-6
66 R_cov = np.diag(noise_std**2)
67 H = np.eye(n)
68
```

```
estimates = []

def dynamics_jacobian():
    F = np.eye(n)
    F[0, 3] = 1
    F[1, 4] = 1
    F[2, 5] = 1
    return F

for k in range(len(t)):
    # Predict step
    dx = np.array(satellite_dynamics_damped(x_est, t[k]))
    x_pred = x_est + dx * dt
    F = dynamics_jacobian()
    P_pred = F @ P @ F.T + Q

    # Update step
    z = measurements[k]
    y = z - x_pred
    S = H @ P_pred @ H.T + R_cov
    K = P_pred @ H.T @ np.linalg.inv(S)
    x_est = x_pred + K @ y
    P = (np.eye(n) - K @ H) @ P_pred

    estimates.append(x_est)

estimates = np.array(estimates)
 _kf ,  _kf ,  _kf  = estimates[:, 0], estimates[:, 1], estimates[:, 2]
 x_kf ,  y_kf ,  z_kf  = estimates[:, 3], estimates[:, 4], estimates[:,
    5]

# --- Plotting ---
plt.figure(figsize=(12, 8))

plt.subplot(2, 1, 1)
plt.plot(t,  x_true , label=' x  True')
plt.plot(t,  x_kf , '--', label=' x  EKF')
plt.plot(t,  y_true , label=' y  True')
plt.plot(t,  y_kf , '--', label=' y  EKF')
plt.plot(t,  z_true , label=' z  True')
plt.plot(t,  z_kf , '--', label=' z  EKF')
plt.title('Angular Velocity: True vs EKF Estimate')
plt.ylabel('rad/s')
plt.legend()
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(t, np.degrees( _true ), label='  True')
plt.plot(t, np.degrees( _kf ), '--', label='  EKF')
plt.plot(t, np.degrees( _true ), label='  True')
plt.plot(t, np.degrees( _kf ), '--', label='  EKF')
plt.plot(t, np.degrees( _true ), label='  True')
plt.plot(t, np.degrees( _kf ), '--', label='  EKF')
plt.title('Euler Angles: True vs EKF Estimate')
plt.xlabel('Time (s)')
plt.ylabel('Degrees')
plt.legend()
plt.grid()
```

```
126
127 plt.tight_layout()
128 plt.show()
```