16.30/31 Feedback Control Systems

Lab 2: Multi-Axis Control for Autonomous Navigation

Issued: 10/22/2025 Due: 11/05/2025

Mission Briefing

Congratulations! AeroTech Dynamics crushed Challenge 1 of the Navy competition. Your altitude control system was flawless, precise, and everything those expensive consultants said was impossible with "just feedback control theory." The Navy was impressed, so impressed that they immediately advanced you to Challenge 2.

But success breeds imitation. Word just reached your team that somehow, your magnificent PID design was leaked to the competition. Their army of consultants is now scrambling to reverse-engineer your approach for multi-axis control, confident they can copy their way to victory. Little do they know, you've got modern control theory techniques that will demonstrate the power of systematic design.

Challenge 2 requirements are clear: precise multi-axis trajectory tracking for autonomous navigation. The Navy wants drones that can maintain formation flight, execute complex maneuvers, and track precise 3D paths for supply delivery. Your competitor plans to use multiple independent PID controllers. Time to show them what happens when you apply proper state-space design methods.

The engineering challenge is now multi-dimensional. You need to design, implement, and validate a multi-axis control system that can handle coordinate frame transformations, real-time trajectory tracking, and demonstrate quantifiable performance advantages. Your reputation, the company's future, and bragging rights over consultants depend on it.

Mission Objectives

Your two-week sprint has four critical phases:

1. Characterize visual sensor capabilities and multi-axis system dynamics
2. Implement decentralized PID approach to establish baseline and understand limitations
3. Deploy full state feedback with pole placement for systematic performance guarantees
4. Demonstrate superior trajectory tracking through quantitative comparison

You're provided with `Lab2-Starter-Code.zip` from Canvas. The package includes starter templates with `TODO` markers for your implementation.
Included templates:

- `phase1_sensor_recon.py` - AprilTag characterization and coordinate transformations

- `phase2_classical_multipid.py` - Multi-axis PID implementation
- `phase3_pole_placement.py` - Full state feedback design
- `phase4_trajectory_tracking.py` - Helix trajectory performance comparison

What you will implement:

- Phase 1: Coordinate transformation functions between AprilTag and drone frames
- Phase 2: Extension of Lab 1 PID to three independent controllers
- Phase 3: Pole selection logic and gain calculation from requirements
- Phase 4: Controller testing and performance analysis

Navy Performance Requirements

The Navy has specified clear performance requirements that your control system must achieve:

Performance Metric	**Requirement**
Initial Position Acquisition	
- Settling time	≤ 4 seconds
- Overshoot	$< 15\%$ per axis
- Steady-state error	< 10 cm position
Trajectory Tracking	
- Mean tracking error	< 10 cm
- Maximum deviation	< 20 cm
- Multi-cycle stability	Error growth $< 5\%$ over 5 cycles
Multi-Axis Coordination	
- Axis synchronization	Settling within 1.5 seconds of each other

These requirements will be used to evaluate both control approaches and determine which design methodology provides superior performance.

Control System Architecture

The Tello drone employs a hierarchical control architecture where your controller sends velocity commands to the Tello's internal firmware. For this lab, we focus on 3-DOF position control (x, y, z) with fixed yaw orientation.

We use the simplified kinematic model:

$$\dot{x} = v_x \tag{1}$$
$$\dot{y} = v_y \tag{2}$$
$$\dot{z} = v_z \tag{3}$$

where $[v_x, v_y, v_z]^T$ are the commanded velocity inputs, and all positions are relative to the AprilTag coordinate frame.

Performance Metrics

Building on Lab 1's metrics, we evaluate multi-axis performance:

- **Multi-axis settling time** [s]: Time for all axes to reach and stay within $\pm 5\%$ of setpoint
- **Coordination spread** [s]: Difference between fastest and slowest axis settling times
- **Mean tracking error** [m]: Average Euclidean distance from reference during trajectory

$$e_{track} = \frac{1}{N} \sum_{k=1}^{N} \sqrt{(x_k - x_{ref,k})^2 + (y_k - y_{ref,k})^2 + (z_k - z_{ref,k})^2}$$

- **Error growth ratio**: Comparison of tracking error between first and last trajectory cycles

$$\text{Growth} = \frac{e_{track,cycle5} - e_{track,cycle1}}{e_{track,cycle1}} \times 100\%$$

- **Control effort** [m/s]: RMS of commanded velocities

Phase 1: Foundations of Feedback 15 points

Before engaging in multi-axis control design, you need to understand coordinate frame transformations and Tello drone visual sensor capabilities.

Mission Objective: Establish coordinate frame understanding and characterize AprilTag detection capabilities for downstream controller design.

Task 1.1: Coordinate Frame Transformation

Understanding coordinate transformations is fundamental to multi-axis control with visual feedback.

AprilTag vs Drone Coordinate Frames

- **AprilTag Frame**: Position measurements $[x_{tag}, y_{tag}, z_{tag}]$ relative to tag, where:
  - x_{tag} is lateral displacement (positive = drone is to the right of tag)
  - y_{tag} is vertical displacement (positive = drone is above tag)
  - z_{tag} is distance from tag (positive = drone is away from tag)
- **Drone Body Frame**: Control commands $[v_x, v_y, v_z]$ in drone coordinates, where:
  - v_x is forward/backward velocity (positive = forward)
  - v_y is left/right velocity (positive = right)
  - v_z is up/down velocity (positive = up)

Coordinate Transformation Implementation

Complete the coordinate transformation functions:

- Transform AprilTag measurements to control commands
- For this lab, maintain constant yaw (drone always faces tag)
- Implement safety bounds on velocity commands ($|v_i| \leq 0.3$ m/s)

Task 1.2: AprilTag Detection Characterization

Apply your coordinate frame understanding to characterize sensor performance.

Setup

1. Mount an AprilTag on a wall at approximately 1.5 m height
2. Use your Lab 1 altitude controller to hover at tag height
3. Position drone directly facing the AprilTag (yaw = 0)

Procedure

For each test distance $d \in \{1.0, 1.5, 2.0, 2.5, 3.0\}$ m:

1. Use simple proportional control to achieve target distance
2. Once settled, record 30 s of AprilTag detection data
3. Log for each distance:
    - Detection success rate [%]
    - Position noise: std(x_{tag}), std(y_{tag}), std(z_{tag}) [m]
    - Maximum consecutive dropout duration [s]

Deliverables

- Plots showing detection success rate and pose noise vs distance
- Identify optimal operating range (typically 1.0-2.5m for Tello)
- Document noise characteristics for controller design

Deliverable: Complete coordinate frame implementation and sensor characterization saved in folder `Lab2-Phase1`.

Phase 2: Multi-Axis PID Implementation 25 points

Implement what your competition is attempting: independent PID controllers for each axis. This establishes a performance baseline and reveals fundamental limitations.

Mission Objective: Implement decentralized multi-axis control and document performance limitations that motivate centralized control approaches.

Task 2.1: Multi-Axis PID Implementation

Building on your Lab 1 PID controller, implement three-axis control using `phase2_classical_multipid.py`

Implementation Requirements

- Extend your Lab 1 PID class to create three independent controllers
- Each controller operates independently:
  - X-axis PID: controls v_x based on x_{tag} error
  - Y-axis PID: controls v_y based on y_{tag} error
  - Z-axis PID: controls v_z based on z_{tag} error
- Maintain constant yaw facing AprilTag
- Implement velocity limiting for safety: $|v_i| \leq 0.3$ m/s

Task 2.2: Decentralized PID Tuning

Individual Axis Tuning

For each axis (x, y, z), follow the procedure:

1. **P-only control**:
   - Test $K_p \in \{0.3, 0.5, 0.8, 1.0, 1.5\}$
   - Command 0.3 m step reference
   - Select K_p for fastest response without excessive oscillation
2. **Add integral control**:
   - Test $K_i \in \{0.01, 0.05, 0.1, 0.2\}$
   - Look for steady-state error elimination
3. **Add derivative control**:
   - Test $K_d \in \{0.01, 0.05, 0.1\}$
   - Look for reduced overshoot

Multi-Axis Integration Testing

1. Activate all axes simultaneously with individually tuned gains
2. Test diagonal step command (all axes move together)
3. If unstable, reduce all K_p gains by 20% and retry
4. Document any performance degradation from single-axis testing

Task 2.3: Performance Against Requirements

Evaluate your PID controller against Navy requirements:

- Command step from [1.5, 0, tag_height] to [2.0, 0.3, tag_height+0.3]
- Measure settling time, overshoot, steady-state error for each axis
- Calculate coordination spread
- Document which requirements are met/unmet

Key observations to document:

- Time spent tuning (log actual time)
- Difficulty achieving simultaneous requirements on all axes
- Trade-offs encountered (e.g., fast response vs. low overshoot)

Deliverable: PID implementation, tuning data, and requirement evaluation saved in folder `Lab2-Phase2`.

Phase 3: Full State Feedback with Pole Placement 30 points

Deploy modern control theory to systematically achieve performance requirements. While competitors struggle with trial-and-error, you'll directly specify desired system behavior.

Mission Objective: Design and implement pole placement controller that meets Navy requirements through systematic design.

Task 3.1: State-Space Model Development

Formulate the 3-DOF system in state-space form using `phase3_pole_placement.py`:

System Model

We assume the Tello's internal velocity controller is fast enough that we can model the system as pure integrators from velocity command to position. This is a reasonable assumption given the Tello's sophisticated onboard control.

Based on this integrator model:

- State vector: $\mathbf{x} = [x, y, z]^T$ (positions relative to AprilTag)
- Input vector: $\mathbf{u} = [v_x, v_y, v_z]^T$ (velocity commands)
- System dynamics: $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$

With direct velocity control of position states:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note: With this integrator model and state feedback $\mathbf{u} = -\mathbf{K}(\mathbf{x} - \mathbf{x}_{ref})$, the closed-loop poles are placed at $-K_i$ for each axis. This allows us to directly specify desired dynamics through gain selection.

Task 3.2: Pole Placement Design

Design controller gains to meet Navy requirements.

From Requirements to Poles

In the starter code `phase3_pole_placement.py`, you'll implement a function that converts performance requirements to pole locations. Consider these design principles:

- Settling time (2% criterion): $t_s \approx 4/|\text{Re(pole)}|$
- Overshoot $< 15\%$ requires damping $\zeta > 0.5$ for complex poles
- Real poles give no overshoot but potentially slower response
- With our integrator model, closed-loop poles are placed at $-K_i$ for each axis

Your task: Determine appropriate pole locations based on the Navy requirements and document your reasoning.

Implementation Steps

1. **Initial Design**:
   - Requirement: 4 second settling → poles at $\text{Re}(\lambda) \leq -1$
   - Start conservative with real poles: $\lambda = [-1.5, -1.5, -1.5]$
   - This gives gains: $\mathbf{K} = \text{diag}(1.5, 1.5, 1.5)$
   - Implement control law: $\mathbf{u} = -\mathbf{K}(\mathbf{x} - \mathbf{x}_{ref})$
2. **Gain Calculation**:
   - For our integrator system, gains directly equal pole magnitudes
   - $K = \text{diag}(|p_1|, |p_2|, |p_3|)$ where p_i are desired poles
   - Can verify using `scipy.signal.place_poles()` if desired
3. **Iterative Refinement**:
   - Test step response with initial poles
   - If requirements not met, adjust specific poles
   - Example: Make z-axis faster if height tracking lags: $\lambda = [-1.5, -1.5, -2.0]$
   - Document rationale for final pole selection

Task 3.3: Performance Validation

Test your pole placement controller:

1. **Same test as PID**: Diagonal step command from [1.5, 0, tag_height] to [2.0, 0.3, tag_height+0.3]
2. **Measure**: Settling time, overshoot, steady-state error per axis
3. **Verify**: All requirements met simultaneously
4. **Compare**: Time to design (minutes) vs PID tuning time

What You'll Implement

In the starter code, you will:

- Calculate desired pole locations from requirements
- Compute gain matrix K based on pole choices
- Implement the state feedback control law
- Document your design rationale

Deliverable: Pole placement implementation with design rationale and performance validation saved in folder `Lab2-Phase3`.

Phase 4: Trajectory Tracking Comparison 30 points

Demonstrate superiority through challenging trajectory tracking that reveals accumulating errors and coordination issues.

Mission Objective: Execute helix trajectory with increasing complexity and provide quantitative evidence of control advantages.

Task 4.1: Helix Trajectory Implementation

Implement vertical helix trajectory in `phase4_trajectory_tracking.py`. The trajectory should:

- Center at 1.5m from tag at tag height
- Have radius of 0.3m in y-direction
- Have amplitude of 0.4m in z-direction
- Keep x-position constant (no forward/backward motion)
- Complete each cycle in 20 seconds

The starter code provides the trajectory generation function - your task is to implement the tracking control for both PID and pole placement controllers.

Task 4.2: Progressive Complexity Testing

For both PID and pole placement controllers:

Test Protocol

1. **Initialization**:
    - Start at helix center: [1.5, 0, tag_height]
    - Wait for position stabilization
2. **1-Cycle Test** (20 seconds):
    - Execute single helix cycle
    - Record position data at 10 Hz
    - Calculate mean tracking error
3. **3-Cycle Test** (60 seconds):
    - Execute three continuous cycles
    - Measure error for each cycle separately
    - Check for error accumulation

4. **5-Cycle Test** (100 seconds):
    - Execute five continuous cycles
    - Evaluate long-term stability
    - Calculate error growth ratio

Metrics to Calculate

For each test, compute:

- Mean tracking error per cycle [cm]
- Maximum deviation from reference [cm]
- Error growth: $\frac{\text{error}_{last} - \text{error}_{first}}{\text{error}_{first}} \times 100\%$
- Control effort (RMS of velocity commands)

Task 4.3: Performance Analysis

Create comprehensive comparison:

Metric	**PID Controller**	**Pole Placement**
1-Cycle Performance (20s)		
Mean tracking error [cm] Max deviation [cm] Control effort [m/s]		
3-Cycle Performance (60s)		
Mean tracking error [cm] Max deviation [cm] Error growth from cycle 1 [%]		
5-Cycle Performance (100s)		
Mean tracking error [cm] Max deviation [cm] Error growth from cycle 1 [%]		
Design Process		
Design/tuning time [min] Met all requirements? Number of iterations		

Task 4.4: Executive Summary

Write 2-3 paragraphs addressing:

- Which controller met Navy requirements more completely?
- Quantify the advantage in tracking accuracy and stability

- Compare design time: trial-and-error tuning vs. systematic pole selection
- Why does error accumulate more with PID over multiple cycles?
- Final recommendation: Which approach should AeroTech use for the Navy contract?

Include plots showing:

- 3D trajectory plots (reference vs. actual) for both controllers
- Tracking error vs. time for 5-cycle test
- Bar chart comparing key metrics

Deliverable: Complete trajectory tracking data, performance comparison, plots, and executive summary saved in folder `Lab2-Phase4`.

Mission Debrief and Submission

Outstanding work! You've demonstrated that systematic control design outperforms trial-and-error approaches. The Navy contract is secured, and those consultants learned that modern control theory beats PowerPoint engineering.

Final Deliverable Structure

- Create main folder: `Lab2-[name1]-[name2]`
- Include all phase folders: `Lab2-Phase1` through `Lab2-Phase4`
- Each folder contains: scripts, data files, plots, analysis documents
- Include `README.txt` summarizing key findings
- Add `Executive-Summary.pdf` with final comparison

Key Takeaways You've Demonstrated

- PID: Works but requires extensive tuning, no performance guarantees
- Pole Placement: Systematic design directly from requirements
- Multi-axis coordination naturally handled by state-space methods
- Design time dramatically reduced with modern control theory
- Performance requirements achieved systematically, not by luck

Reflection Questions (Not Graded)

1. How did having clear requirements change your design approach?
2. What was most challenging about tuning multiple PID controllers?
3. How did pole selection directly address the requirements?
4. Why does the tracking error accumulate differently between controllers?
5. If you added disturbances (wind), which controller would be more robust?

Compress everything into `Lab2-[name1]-[name2].zip` for submission. Excellent engineering work!