

CT System Project

ECEN 3300 Linear Systems - Lab Notebook

Name: Ben Mackert

Date: 2/17/2026

Overview

In this lab you will implement — from scratch — analysis tools for both **first-order** and **second-order** LTI systems.

First-order:

$$H(s) = \frac{K}{\tau s + 1}$$

Parameter	Symbol	Meaning
DC gain	K	Steady-state output / input ratio
Time constant	τ	Sets the speed of response (s)

Second-order:

$$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

Parameter	Symbol	Meaning
Natural frequency	ω_n	Sets the speed of response (rad/s)
Damping ratio	ζ	Controls oscillatory behavior (dimensionless)

You will complete five parts:

1. **First-Order Systems** — step response, pole, and Bode plot
2. **Second-Order: Step & Impulse Response** — simulate and annotate time-domain behavior
3. **Second-Order: Pole-Zero Map** — plot poles in the complex s-plane
4. **Second-Order: Bode Plot** — frequency-domain magnitude and phase
5. **Design Challenge** — meet specifications for both system types

Allowed imports: `numpy`, `matplotlib`, `scipy.signal`. Do **not** use high-level wrappers like `control` or `sympy`.

Part 0 - Imports & Setup

(10 Points)

```
In [63]: import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

plt.rcParams.update({'figure.dpi': 110, 'axes.grid': True, 'grid.alpha': 0.3})
print("Setup complete")
```

Setup complete

Part 1 — First-Order Systems

(15 Points)

Background

The standard first-order transfer function is:

$$H(s) = \frac{K}{\tau s + 1}$$

It has a **single real pole** at $s = -1/\tau$ and no finite zeros.

Step response (exact closed-form):

$$y(t) = K \left(1 - e^{-t/\tau} \right) u(t)$$

Key time-domain facts:

Time	Value	Meaning
$t = \tau$	$y = 0.632 K$	One time constant - 63.2% of final value
$t = 2\tau$	$y = 0.865 K$	
$t = 4\tau$	$y = 0.982 K$	Effectively settled (2% criterion: $T_s = 4\tau$)

Frequency response ($s = j\omega$):

$$H(j\omega) = \frac{K}{j\omega\tau + 1}$$

- DC gain: $20 \log_{10}(K)$ dB
- Corner (break) frequency: $\omega_c = 1/\tau$ rad/s
- Magnitude rolls off at **-20 dB/decade** above ω_c
- Phase: $0^\circ \rightarrow -90^\circ$, passing through -45° at ω_c

Task 1.1 - Implement `fo_step_response`

Compute the step response **two ways** and verify they match.

```
In [64]: def fo_step_response(K, tau, t_end=None, n_points=1000):
        """
        Compute the unit step response of  $H(s) = K / (\tau s + 1)$ .

        Parameters
        -----
        K      : float - DC gain
        tau     : float - time constant (s)
        t_end   : float or None - if None, use 6*tau
        n_points : int

        Returns
        -----
        t       : ndarray
        y_exact  : ndarray - closed-form  $K*(1 - \exp(-t/\tau))$ 
        y_scipy  : ndarray - from scipy.signal.step
        """
        if t_end is None:
            t_end = 6 * tau # YOUR CODE (hint: 6*tau gives a clean settling window)

        t = np.linspace(0, t_end, n_points)

        # TODO: Closed-form exact solution
        y_exact = K * (1 - np.exp(-t / tau)) # YOUR CODE

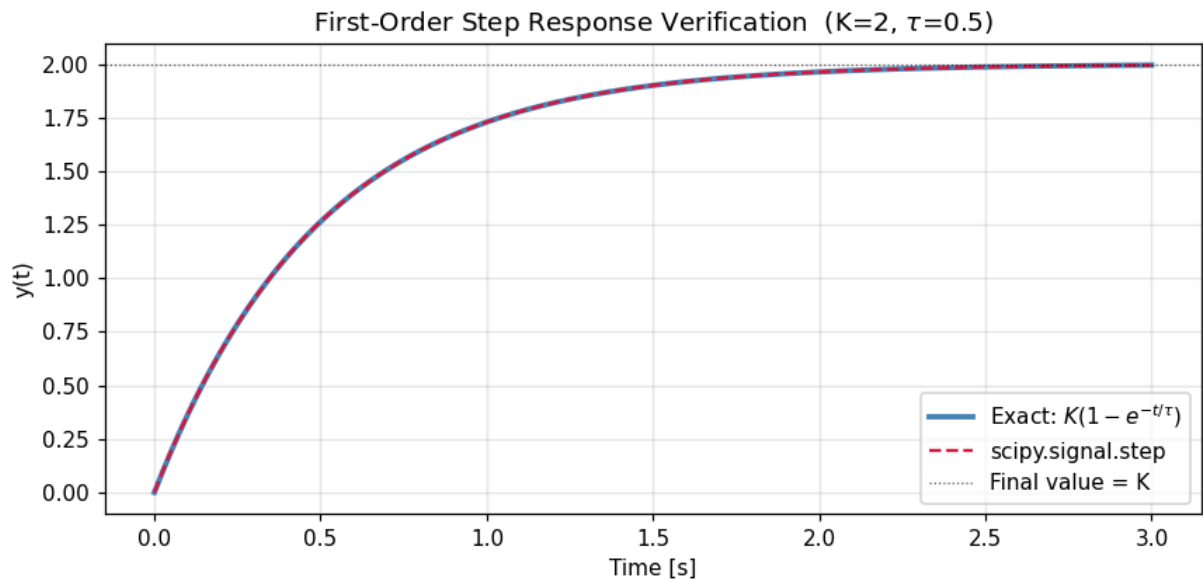
        # TODO: scipy - define num/den for  $K/(\tau s + 1)$ , build TransferFunction
        # Hint: Look up the format for num, den, e.g., under scipy.signal.step
        num = [K] # YOUR CODE
        den = [tau, 1] # YOUR CODE
        sys = signal.TransferFunction(num, den)
        _, y_scipy = signal.step(sys, T=t)

        return t, y_exact, y_scipy

# --- Verification: both curves should overlap perfectly ---
t, y_exact, y_scipy = fo_step_response(K=2.0, tau=0.5)
```

```
# Hint: Familiarize yourself with matplotlib.pyplot commands if necessary
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(t, y_exact, color='steelblue', linewidth=2.5, label='Exact: $K(1-e^{-t/\tau})$')
ax.plot(t, y_scipy, color='crimson', linewidth=1.5, linestyle='--', label='scipy.signal.step')
ax.axhline(2.0, color='black', linestyle=':', linewidth=0.8, alpha=0.6, label='Final value = K')
ax.set_xlabel('Time [s]'); ax.set_ylabel('y(t)')
ax.set_title('First-Order Step Response Verification (K=2, $\tau=0.5$)')
ax.legend(); plt.tight_layout(); plt.show()

max_err = np.max(np.abs(y_exact - y_scipy))
print(f'Max error between exact and scipy: {max_err:.2e} (should be < 1e-4)')
```



Max error between exact and scipy: 1.55e-15 (should be < 1e-4)

Task 1.2 - Annotated Step Response

Implement `fo_plot_step` with annotations for the time constant and settling time.

```
In [65]: def fo_plot_step(K, tau, ax=None, color='steelblue', label=None):
        """
        Plot the annotated first-order step response.

        Required annotations:
        - Horizontal dashed line at y = K (final value)
        - Shaded +-2% settling band around K
        - Vertical dashed line at t = tau with a dot at y = 0.632*K
          and a text label '63.2%'
        - Vertical dashed line at t = 4*tau with 'Ts = 4 tau' label
        """
        if ax is None:
            ax = plt.gca()

        t, y_exact, _ = fo_step_response(K, tau)

        # TODO: Plot the response curve
        # YOUR CODE
        ax.plot(t, y_exact, color=color, linewidth=2, label=label)
```

```

# TODO: Final value line at y = K
# YOUR CODE
ax.axhline(K, color='black', linestyle='--', linewidth=1, label=f'Final')

# TODO: Shade the  $\pm 2\%$  settling band around K
# Hint: ax.fill_between or ax.axhspan
# YOUR CODE
ax.axhspan(0.98*K, 1.02*K, alpha=0.15, color='green', label='±2% band')

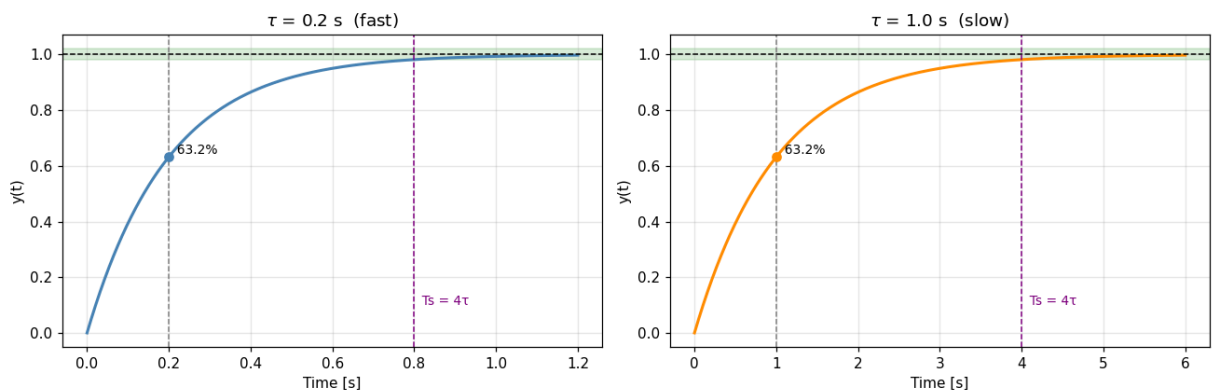
# TODO: Mark  $t = \tau$  - vertical dashed line + dot at 63.2% of K
# YOUR CODE
ax.axvline(tau, color='gray', linestyle='--', linewidth=1)
ax.plot(tau, 0.632*K, 'o', color='gray')
ax.text(tau, 0.632*K, ' 63.2%', fontsize=9, va='bottom')

# TODO: Mark  $T_s = 4\tau$  - vertical dashed line + label
# YOUR CODE
ax.axvline(4*tau, color='purple', linestyle='--', linewidth=1)
ax.text(4*tau, 0.1*K, '   $T_s = 4\tau$ ', fontsize=9, color='purple')

ax.set_xlabel('Time [s]'); ax.set_ylabel('y(t)')
ax.set_title(f'First-Order Step Response (K={K},  $\tau$ = $\tau$ )')
return ax

# --- Test with two different time constants ---
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
fo_plot_step(K=1.0, tau=0.2, ax=axes[0], color='steelblue')
fo_plot_step(K=1.0, tau=1.0, ax=axes[1], color='darkorange')
axes[0].set_title('$\tau$ = 0.2 s (fast)'); axes[1].set_title('$\tau$ = 1.0 s (slow)')
plt.tight_layout(); plt.show()

```



Task 1.3 - Varying τ and K

On a single axes, overlay step responses for multiple (K, τ) pairs.

```

In [66]: fo_cases = [
    (1.0, 0.1, 'K=1,  $\tau$ =0.1', '#E74C3C'),
    (1.0, 0.3, 'K=1,  $\tau$ =0.3', '#3498DB'),
    (1.0, 1.0, 'K=1,  $\tau$ =1.0', '#2ECC71'),
    (2.0, 0.3, 'K=2,  $\tau$ =0.3', '#9B59B6'), # same tau as case 2, double

```

```

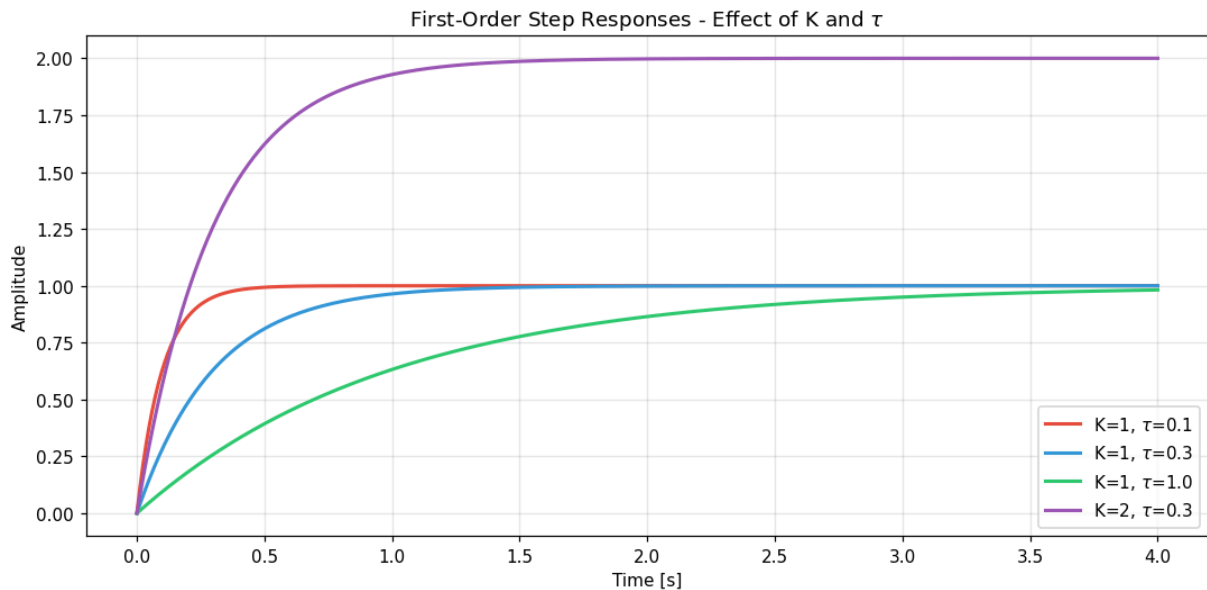
]

fig, ax = plt.subplots(figsize=(10, 5))

for K, tau, label, color in fo_cases:
    t, y_exact, _ = fo_step_response(K, tau, t_end=4.0)
    # TODO: plot each curve with its label and color
    # YOUR CODE
    ax.plot(t, y_exact, color=color, linewidth=2, label=label)

ax.set_xlabel('Time [s]'); ax.set_ylabel('Amplitude')
ax.set_title('First-Order Step Responses - Effect of K and  $\tau$ ')
ax.legend(); plt.tight_layout(); plt.show()

```



Task 1.4 - First-Order Pole Location

Implement `fo_plot_pole` to show the single real pole $s = -1/\tau$ in the s-plane.

```

In [67]: def fo_plot_pole(tau_values, colors, ax=None):
    """
    Plot the pole  $s = -1/\tau$  for each tau value on the real axis.

    Parameters
    -----
    tau_values : list of float
    colors      : list of color strings
    ax         : matplotlib Axes or None
    """
    if ax is None:
        fig, ax = plt.subplots(figsize=(8, 3))

    # TODO: Draw the real (sigma) axis as a horizontal line
    # YOUR CODE
    ax.axhline(0, color='black', linewidth=1)

    # TODO: Draw a faint vertical imaginary axis at sigma=0 for reference

```

```

# YOUR CODE
ax.axvline(0, color='gray', linewidth=0.8, linestyle='--', alpha=0.5)

for tau, color in zip(tau_values, colors):
    # YOUR CODE (s = -1/tau)
    pole = -1 / tau

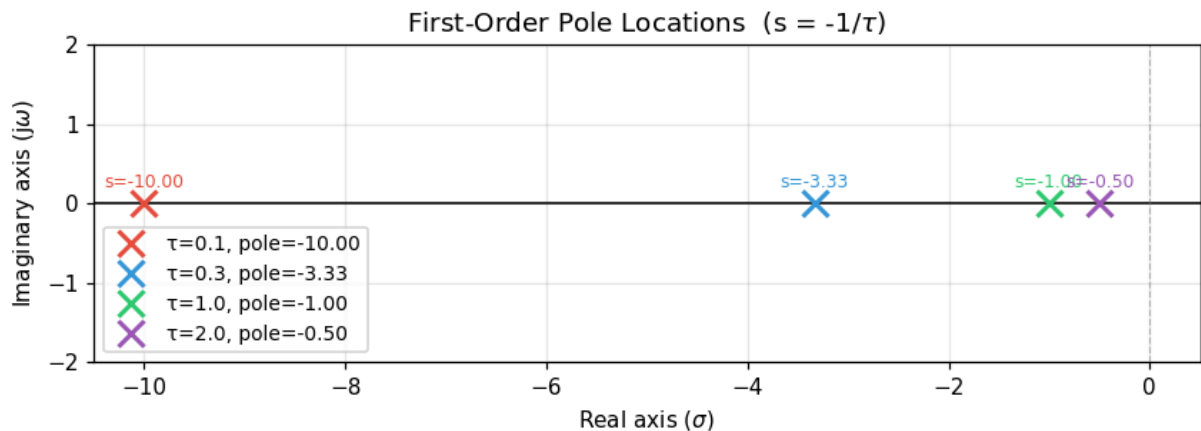
    # TODO: Plot pole as 'x' marker on the real axis (imaginary part = 0)
    # YOUR CODE
    ax.plot(pole, 0, 'x', color=color, markersize=12, markeredgewidth=2,
            label=f'tau={tau}, pole={pole:.2f}')

    # TODO: Annotate with the pole value
    # YOUR CODE
    ax.text(pole, 0.2, f's={pole:.2f}', fontsize=8, ha='center', color=c

ax.set_xlabel('Real axis ($\sigma$)'); ax.set_ylabel('Imaginary axis (j$\omega$)')
ax.set_title('First-Order Pole Locations (s = -1/$\tau$)')
ax.set_ylim(-2, 2)
ax.legend(fontsize=9)
return ax

tau_vals = [0.1, 0.3, 1.0, 2.0]
cols      = ['#E74C3C', '#3498DB', '#2ECC71', '#9B59B6']
fo_plot_pole(tau_vals, cols)
plt.tight_layout(); plt.show()

```



Task 1.5 - First-Order Bode Plot

Implement `fo_bode` for $H(s) = K/(\tau s + 1)$.

Asymptotes to draw:

- Magnitude: flat at $20 \log_{10}(K)$ dB for $\omega \ll 1/\tau$, then -20 dB/decade above ω_c
- Phase: 0° at low freq, -45° at $\omega_c = 1/\tau$, -90° at high freq

```

In [68]: def fo_bode(K, tau, ax_mag=None, ax_phase=None, color='steelblue', label=None)
        ....

```

Bode magnitude and phase for $H(s) = K / (\tau s + 1)$.

Required plot elements:

- Magnitude and phase curves (semilogx)
- Vertical dashed line at corner frequency $\omega_c = 1/\tau$ on both subplots
- -20 dB/decade asymptote on magnitude plot
- Horizontal reference at $(20 \cdot \log_{10}(K) - 3)$ dB
- Dot/marker at the -45 deg phase point

.....

```

wc = 1/tau # YOUR CODE (corner frequency)
w = np.logspace(np.log10(wc / 100), np.log10(wc * 100), 2000)

# TODO: Build num/den and call signal.freqs
num = [K] # YOUR CODE
den = [tau, 1] # YOUR CODE
_, H = signal.freqs(num, den, worN=w)

# TODO: Compute mag_dB and unwrapped phase_deg
mag_dB = 20 * np.log10(np.abs(H)) # YOUR CODE
phase_deg = np.degrees(np.unwrap(np.angle(H))) # YOUR CODE

if ax_mag is None or ax_phase is None:
    fig, (ax_mag, ax_phase) = plt.subplots(2, 1, figsize=(9, 6), sharex=

# TODO: Plot magnitude and phase
# YOUR CODE
ax_mag.semilogx(w, mag_dB, color=color, linewidth=2, label=label)
ax_phase.semilogx(w, phase_deg, color=color, linewidth=2, label=label)

# TODO: Vertical dashed line at wc on both subplots
# YOUR CODE
ax_mag.axvline(wc, color=color, linestyle='--', linewidth=0.8, alpha=0.7)
ax_phase.axvline(wc, color=color, linestyle='--', linewidth=0.8, alpha=0.7)

# TODO: -20 dB/decade asymptote for w > wc
# Hint: mag_asy = 20*log10(K) - 20*log10(w_asy / wc)
# YOUR CODE
w_asy = w[w > wc]
mag_asy = 20*np.log10(K) - 20*np.log10(w_asy / wc)
ax_mag.semilogx(w_asy, mag_asy, 'k--', linewidth=1, alpha=0.4, label='-20 dB/decade')

# TODO: Horizontal reference at the -3 dB point (20*log10(K) - 3)
# YOUR CODE
ax_mag.axhline(20*np.log10(K) - 3, color='gray', linestyle=':', linewidth=1)

ax_mag.set_ylabel('Magnitude [dB]')
ax_phase.set_ylabel('Phase [deg]')
ax_phase.set_xlabel('Frequency $\omega$ (rad/s)')
ax_phase.set_yticks([0, -15, -30, -45, -60, -75, -90])
return ax_mag, ax_phase

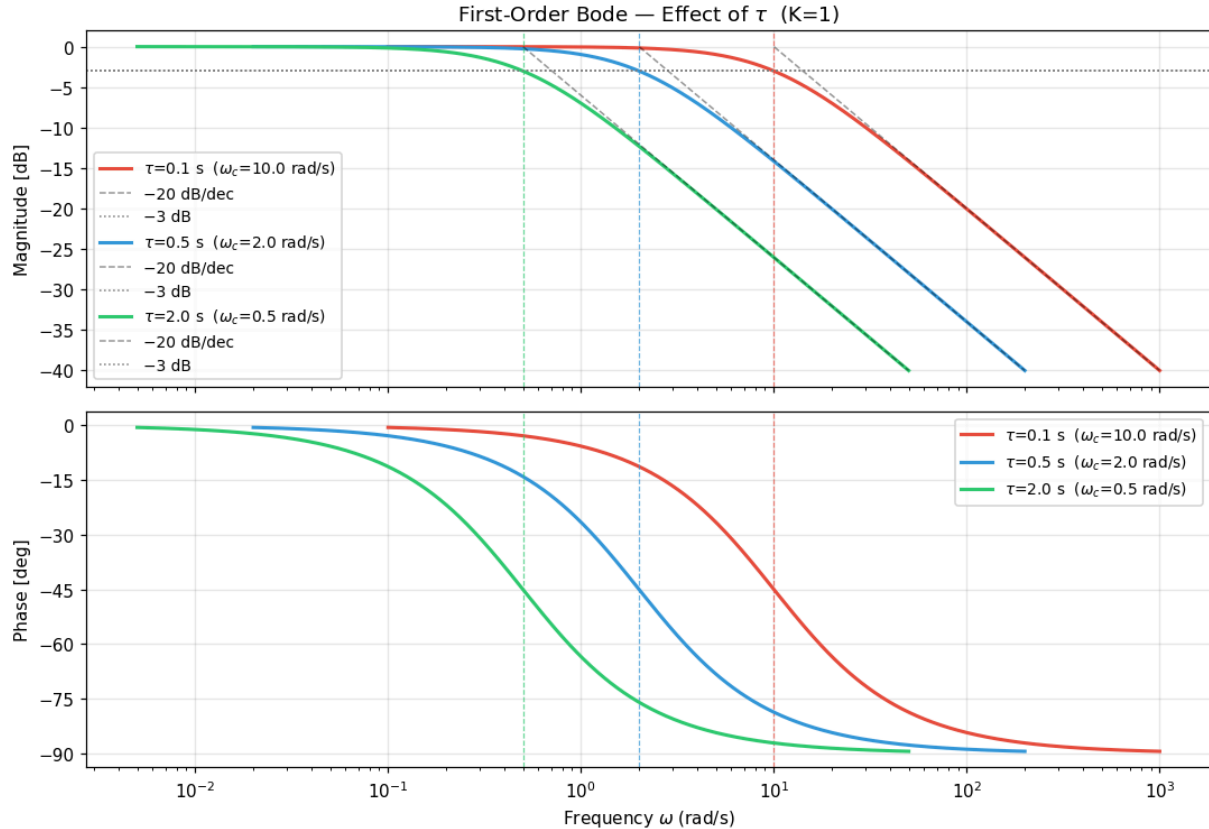
# --- Compare different tau values (K=1 fixed) ---
fig, (ax_m, ax_p) = plt.subplots(2, 1, figsize=(10, 7), sharex=True)
for tau, color in zip([0.1, 0.5, 2.0], ['#E74C3C', '#3498DB', '#2ECC71']):
    fo_bode(K=1.0, tau=tau, ax_mag=ax_m, ax_phase=ax_p,

```

```

color=color, label=f'$\\tau$={tau} s ($\\omega_c$={1/tau:.1f} r
ax_m.set_title('First-Order Bode - Effect of $\\tau$ (K=1)')
ax_m.legend(fontsize=9); ax_p.legend(fontsize=9)
plt.tight_layout(); plt.show()

```



Task 1.6 - Effect of DC Gain K

Fix $\tau = 0.5$ s and vary $K \in \{0.5, 1, 2, 5\}$. Plot the Bode magnitude and phase.

What shifts in the magnitude plot? Does K affect the phase? Explain.

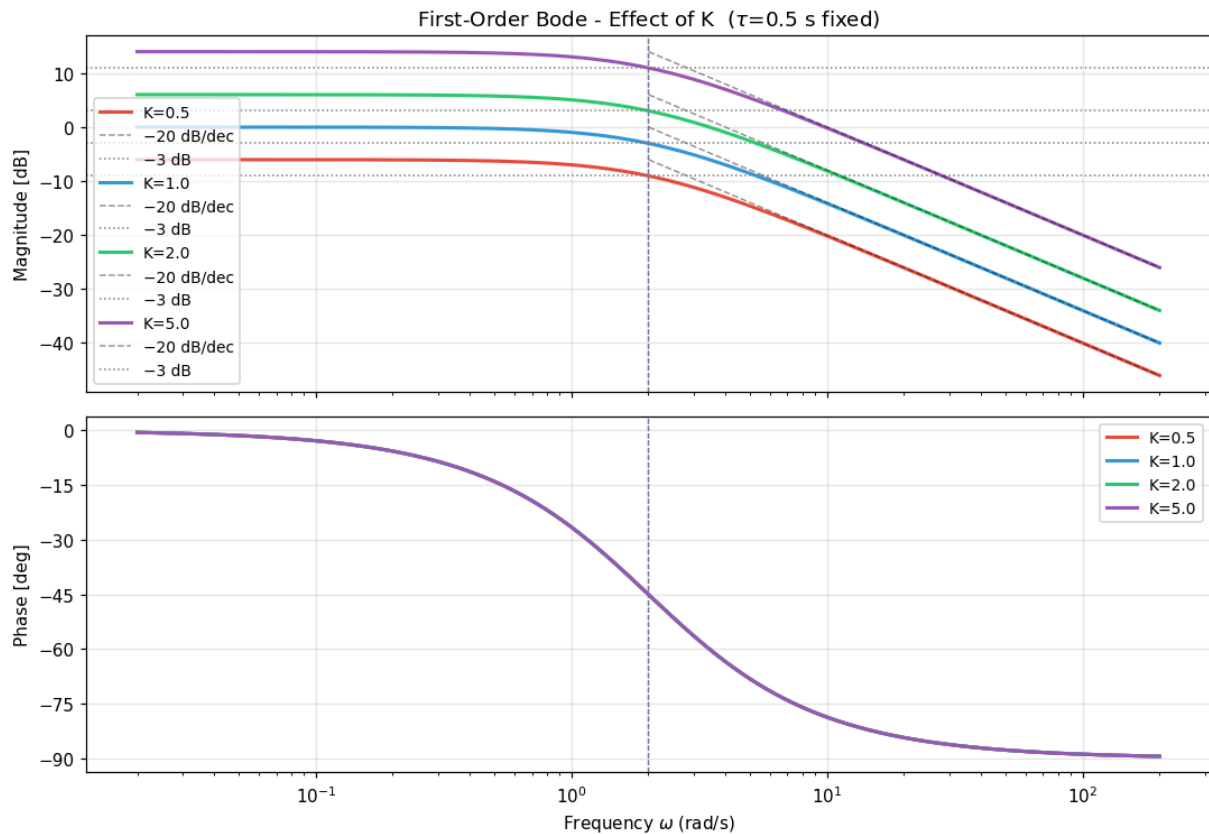
```

In [69]: fig, (ax_m, ax_p) = plt.subplots(2, 1, figsize=(10, 7), sharex=True)

tau_fixed = 0.5
for K, color in zip([0.5, 1.0, 2.0, 5.0], ['#E74C3C', '#3498DB', '#2ECC71',
# TODO: call fo_bode for each K, overlay on ax_m and ax_p
# YOUR CODE
fo_bode(K=K, tau=tau_fixed, ax_mag=ax_m, ax_phase=ax_p,
color=color, label=f'K={K}'))
pass

ax_m.set_title(f'First-Order Bode - Effect of K ($\\tau$={tau_fixed} s fixe
ax_m.legend(fontsize=9); ax_p.legend(fontsize=9)
plt.tight_layout(); plt.show()

```



Questions - Part 1

1. From Task 1.2: read $y(\tau)$ from your plot. What fraction of the final value is this?
2. In Task 1.3, $K=1$ $\tau=0.3$ vs. $K=2$ $\tau=0.3$ have the same shape but different amplitude. Why doesn't K change the time scale?
3. From Task 1.4: if you double τ , where does the pole move, and how does the response change?
4. In Task 1.6, varying K shifts the magnitude **vertically** but leaves phase **unchanged**. Prove this using $\angle H(j\omega)$.
5. A first-order system can **never** overshoot. Prove this from $y(t) = K(1 - e^{-t/\tau})$.

Your answers:

1. From task 1.2, it is 63.2/100 of the final value.
2. K does not change the time scale because it is not in the exponent which is where we get the time constant τ , it is instead a multiplier on the output.
3. If we double τ , the pole will move closer to the origin. A pole that is closer to the origin means a slower exponential decay and the system will take longer to settle (the settling time doubles).
4. Substituting $s = j\omega$ into $H(s) = K / (\tau s + 1)$, the phase is: $\angle H(j\omega) = \angle K - \angle(j\omega\tau + 1) = 0^\circ - \arctan(\omega\tau) = -\arctan(\omega\tau)$. Since K is a positive real constant, its angle is 0° and

cancels out entirely. The phase depends only on ω and τ , so changing K has no effect on the phase.

5. For all $t \geq 0$, the exponential term $e^{-(t/\tau)}$ is always between 0 and 1, so $(1 - e^{-(t/\tau)})$ is always between 0 and 1. Therefore $y(t)$ is always between 0 and K — it approaches K from below but never exceeds it. Overshooting would require $y(t) > K$, which is impossible since the exponential never becomes negative.

Part 2 - Second-Order: Step & Impulse Response

(15 Points)

Background

For an **underdamped** system ($0 < \zeta < 1$), the step response is:

$$y(t) = \left[1 - \frac{e^{-\zeta\omega_n t}}{\sqrt{1-\zeta^2}} \sin(\omega_d t + \phi) \right] u(t), \quad \omega_d = \omega_n \sqrt{1-\zeta^2}, \quad \phi = \arccos(\zeta)$$

Key time-domain metrics:

Metric	Formula
OS%	
%	$= 100$
Overshoot	$\frac{e^{-\pi\zeta}}{\sqrt{1-\zeta^2}}$
Peak time	$T_p = \frac{\pi}{\omega_d}$
Settling time (2%)	$T_s \approx \frac{4}{\zeta\omega_n}$

Task 2.1 - Implement `compute_step_response`

Complete the function below. Use `scipy.signal.TransferFunction` and `scipy.signal.step`.

```
In [70]: def compute_step_response(wn, zeta, t_end=None, n_points=2000):
        """
        Compute the unit step response of  $H(s) = \frac{wn^2}{s^2 + 2*zeta*wn*s + wn^2}$ 

        Parameters
        -----
        wn      : float - natural frequency (rad/s)
        zeta     : float - damping ratio
```

```

t_end    : float or None – end time; if None, choose automatically
n_points : int    – number of time samples

Returns
-----
t : ndarray – time vector
y : ndarray – step response
"""
# TODO: Define numerator and denominator coefficient arrays
num = [wn**2]# YOUR CODE
den = [1.2*zeta*wn, wn**2]# YOUR CODE

# TODO: If t_end is None, set it to 5 * settling time (4 / (zeta*wn))
if t_end is None:
    t_end = 5 * (4 / (zeta * wn)) # YOUR CODE

t = np.linspace(0, t_end, n_points)

# TODO: Build a Transfer Function and compute step response
sys = signal.TransferFunction(num, den)# YOUR CODE
t_out, y_out = signal.step(sys, T=t) # YOUR CODE

return t_out, y_out

# Quick sanity check – steady state should be 1.0
t, y = compute_step_response(wn=10, zeta=0.5)
print(f"Steady-state value: {y[-1]:.4f} (expected $\approx$ 1.0)")

```

Steady-state value: 1.0000 (expected ≈ 1.0)

Task 2.2 - Implement `step_response_metrics`

Compute the analytical metrics for an underdamped system.

```

In [71]: def step_response_metrics(wn, zeta):
        """
        Return a dict of step-response metrics for an underdamped system.
        Returns None for metrics that don't apply when zeta >= 1.

        Keys: 'wd', 'OS_pct', 'Tp', 'Ts'
        """
        metrics = {}

        # TODO: Compute damped natural frequency wd

        if zeta < 1:
            metrics['wd'] = wn * np.sqrt(1 - zeta**2)
            metrics['OS_pct'] = 100 * np.exp(-np.pi * zeta / np.sqrt(1 - zeta**2))
            metrics['Tp'] = np.pi / metrics['wd']
        else:
            metrics['wd'] = 0.0
            metrics['OS_pct'] = 0.0
            metrics['Tp'] = None

```

```

metrics['Ts'] = 4 / (zeta * wn)

return metrics

```

```

# Test
m = step_response_metrics(wn=10, zeta=0.3)
print("Metrics for wn=10, zeta=0.3:")
for k, v in m.items():
    print(f" {k:8s} = {v}")

```

```

Metrics for wn=10, zeta=0.3:
wd      = 9.539392014169456
OS_pct  = 37.232610492658644
Tp      = 0.32932839419151544
Ts      = 1.3333333333333333

```

Task 2.3 - Implement `plot_step_response`

Plot the step response **with annotations** marking T_p , OS%, and T_s .

```

In [72]: def plot_step_response(wn, zeta, ax=None, color='steelblue', label=None):
        """
        Plot the annotated step response on `ax` (or current axes if None).

        Required annotations:
        - Horizontal dashed line at y=1.0 (final value)
        - Shaded  $\pm 2\%$  settling band
        - Vertical dashed line at  $T_p$  with OS% label (underdamped only)
        - Vertical dashed line at  $T_s$  with  $T_s$  label
        """
        if ax is None:
            ax = plt.gca()

        t, y = compute_step_response(wn, zeta)
        m = step_response_metrics(wn, zeta)

        # TODO: Plot the step response curve
        # YOUR CODE
        ax.plot(t, y, color=color, linewidth=2, label=label)

        # TODO: Draw a dashed horizontal line at y = 1.0
        # YOUR CODE
        ax.axhline(1.0, color='black', linestyle='--', linewidth=1, label='Final')

        # TODO: Shade the  $\pm 2\%$  settling band (y = 0.98 to 1.02)
        # Hint: ax.fill_between or ax.axhspan
        # YOUR CODE
        ax.axhspan(0.98, 1.02, alpha=0.15, color='green', label='±2% band')

        # TODO: If underdamped, mark  $T_p$  with a vertical line and annotate OS%
        if zeta < 1 and m['Tp'] is not None:
            ax.axvline(m['Tp'], color='red', linestyle='--', linewidth=1)
            ax.text(m['Tp'], max(y)*1.01, f" OS={m['OS_pct']:.1f}%", fontsize=9)

```

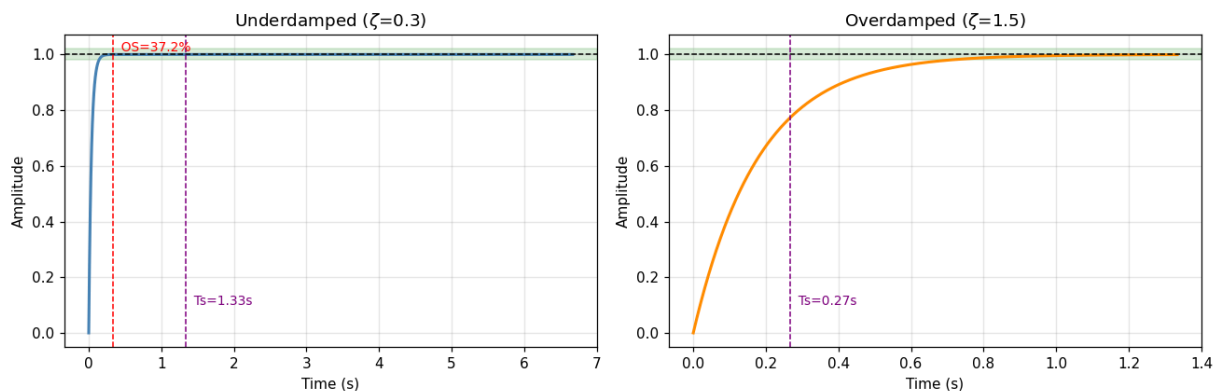
```

# TODO: Mark Ts with a vertical line and label
# YOUR CODE
ax.axvline(m['Ts'], color='purple', linestyle='--', linewidth=1)
ax.text(m['Ts'], 0.1, f"Ts={m['Ts']:.2f}s", fontsize=9, color='purple')

ax.set_xlabel("Time (s)")
ax.set_ylabel("Amplitude")
ax.set_title(f"Step Response ($\\omega_n$={wn}, $\\zeta$={zeta})")
return ax

# --- Test your implementation ---
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_step_response(wn=10, zeta=0.3, ax=axes[0], color='steelblue', label='$\\omega_n=10, \\zeta=0.3$')
plot_step_response(wn=10, zeta=1.5, ax=axes[1], color='darkorange', label='$\\omega_n=10, \\zeta=1.5$')
axes[0].set_title("Underdamped ($\\zeta=0.3$)")
axes[1].set_title("Overdamped ($\\zeta=1.5$)")
plt.tight_layout()
plt.show()

```



Task 2.4 - Impulse Response

Implement `compute_impulse_response`. Recall that $h(t) = \mathcal{L}^{-1}\{H(s)\}$, and use `scipy.signal.impulse`.

Relationship to remember: $y_{step}(t) = \int_0^t h(\tau) d\tau$

```

In [73]: def compute_impulse_response(wn, zeta, t_end=None, n_points=2000):
    """
    Compute the impulse response h(t) of the second-order system.

    Returns
    -----
    t : ndarray
    h : ndarray
    """
    # TODO: Mirror the structure of compute_step_response but use signal.imp
    # YOUR CODE

```

```

num = [wn**2]
den = [1, 2*zeta*wn, wn**2]

if t_end is None:
    t_end = 5 * (4 / (zeta * wn))

t = np.linspace(0, t_end, n_points)

sys = signal.TransferFunction(num, den)
t_out, h_out = signal.impulse(sys, T=t)

return t_out, h_out

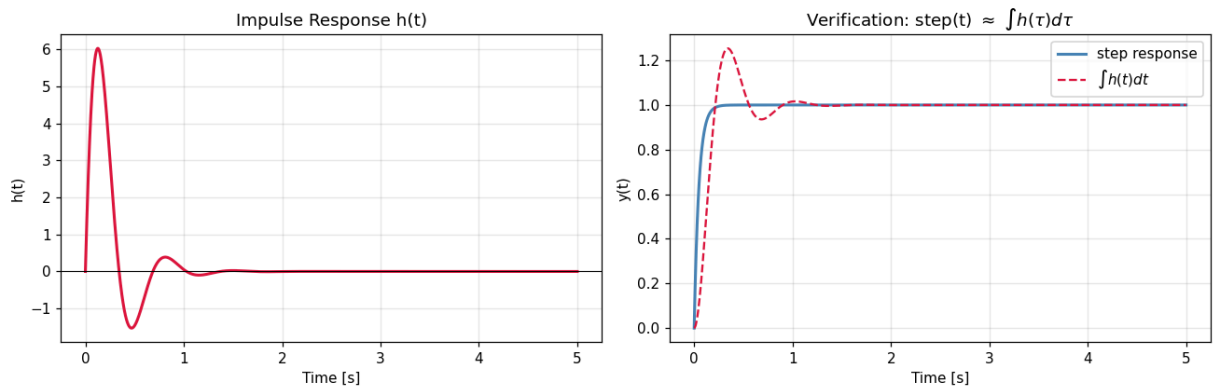
# --- Verification: numerically integrate h(t) and compare to step response
wn_test, zeta_test = 10, 0.4
t_h, h = compute_impulse_response(wn_test, zeta_test)
t_s, y = compute_step_response(wn_test, zeta_test)

# Numerical integration of h(t)
dt = t_h[1] - t_h[0]
y_from_h = np.cumsum(h) * dt

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].plot(t_h, h, color='crimson', linewidth=2)
axes[0].axhline(0, color='black', linewidth=0.7)
axes[0].set_xlabel("Time [s]"); axes[0].set_ylabel("h(t)")
axes[0].set_title("Impulse Response h(t)")

axes[1].plot(t_s, y, color='steelblue', linewidth=2, label='step r
axes[1].plot(t_h, y_from_h, color='crimson', linewidth=1.5, linestyle='--
axes[1].set_xlabel("Time [s]"); axes[1].set_ylabel("y(t)")
axes[1].set_title(r"Verification: step(t) $\approx$ $\int h(\tau)d\tau$") #t
axes[1].legend()
plt.tight_layout(); plt.show()

```



Task 2.5 - Damping Comparison Plot

On a single axes, plot the step response for all four damping regimes. Use the colors provided.

```

In [74]: damping_cases = [
    (0.15, "$\\zeta=0.15 (underdamped)", "#E74C3C"),
    (0.5, "$\\zeta=0.5 (underdamped)", "#3498DB"),
    (1.0, "$\\zeta=1.0 (critically damped)", "#2ECC71"),
    (2.0, "$\\zeta=2.0 (overdamped)", "#9B59B6"),
]

wn = 10.0

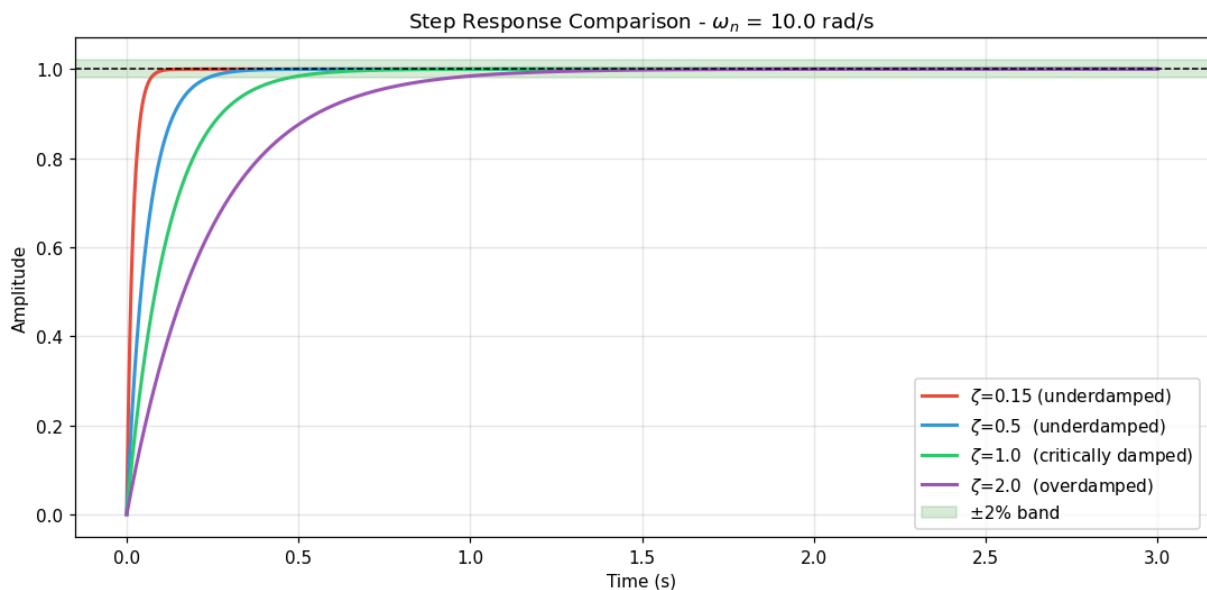
fig, ax = plt.subplots(figsize=(10, 5))

for zeta, label, color in damping_cases:
    t, y = compute_step_response(wn, zeta, t_end=3.0)
    # TODO: Plot each curve with its label and color
    # YOUR CODE
    ax.plot(t, y, color=color, linewidth=2, label=label)

# TODO: Add reference lines: y=1.0, y=1.02, y=0.98, shaded band
# YOUR CODE
ax.axhline(1.0, color='black', linestyle='--', linewidth=1)
ax.axhspan(0.98, 1.02, alpha=0.15, color='green', label='±2% band')

ax.set_xlabel("Time (s)")
ax.set_ylabel("Amplitude")
ax.set_title(f"Step Response Comparison - $\\omega_n$ = {wn} rad/s")
ax.legend()
plt.tight_layout(); plt.show()

```



Questions - Part 2

Answer in the cell below.

1. For $\zeta = 0.15$: read OS% from your plot. How well does it match the analytical formula?
2. Which value of ζ reaches steady-state fastest? Is it exactly $\zeta = 1.0$? Explain.

3. Double ω_n to 20 rad/s (keep $\zeta = 0.5$). What happens to T_p and T_s ? Derive the scaling relationship.
4. Why does numerically integrating $h(t)$ recover the step response?

Your answers:

1. From the plot, the peak occurs at around $y=1.63$ giving $OS = 62.8\%$. This matches the analytical formula for OS very closely confirming that the formula is accurate for underdamped systems
2. The value that reaches the tolerance band the fastest is actually $\zeta=0.7$ which is slightly underdamped. This is because critically damped has no overshoot but also reaches steady state slowly where as a slight underdamping may have a little overshoot but reach steady state much faster.
3. Both T_p and T_s are halved when ω_n is doubled. This follows directly from the formulas: $T_p = \pi / \omega_d = \pi / (\omega_n * \sqrt{1-\zeta^2})$ doubles ω_n , halves T_p . $T_s = 4 / (\zeta * \omega_n)$ doubles ω_n , halves T_s . In general, all time domain metrics scale as $1/\omega_n$, so increasing ω_n speeds up the response proportionally.
4. The step response is the integral of the impulse response: $y_{\text{step}}(t) = \int h(\tau) d\tau$. Computing $\text{np.cumsum}(h)*dt$ approximates this integral, recovering the step response.

Part 3 - Pole-Zero Map

(15 Points)

Background

The poles of $H(s)$ are the roots of the denominator $s^2 + 2\zeta\omega_n s + \omega_n^2 = 0$:

$$s_{1,2} = -\zeta\omega_n \pm \omega_n \sqrt{\zeta^2 - 1}$$

Geometrically:

- Both poles lie on a **circle of radius ω_n** centered at the origin
- The angle from the negative real axis is $\theta = \arccos(\zeta)$
- Real part = $-\zeta\omega_n \rightarrow$ exponential decay rate
- Imaginary part = $\pm\omega_d \rightarrow$ oscillation frequency

Task 3.1 — Implement `compute_poles`

```
In [75]: def compute_poles(wn, zeta):
        """
        Compute the two poles of  $H(s) = \frac{wn^2}{s^2 + 2*zeta*wn*s + wn^2}$ .

        Returns
        -----
        poles : ndarray of shape (2,) with complex dtype

        Hint: Use np.roots() or the newer numpy.polynomial.polynomial.polyroots()
        Hint: Check the roots/polyroots format (coefficients in increasing or decreasing order)
        """
        den = [1, 2*zeta*wn, wn**2] # YOUR CODE
        return np.roots(den) # YOUR CODE

# Test cases
print("wn=10, zeta=0.3:", compute_poles(10, 0.3))
print("wn=10, zeta=1.0:", compute_poles(10, 1.0)) # should be repeated real poles
print("wn=10, zeta=2.0:", compute_poles(10, 2.0)) # should be two real poles
```

```
wn=10, zeta=0.3: [-3.+9.53939201j -3.-9.53939201j]
wn=10, zeta=1.0: [-10.00000008 -9.99999992]
wn=10, zeta=2.0: [-37.32050808 -2.67949192]
```

Task 3.2 - Implement `plot_pole_zero_map`

Requirements:

- Mark poles with 'x' markers
- Draw the ω_n **circle** (dashed) so students can verify $|s_{1,2}| = \omega_n$
- Draw dashed **constant- ζ lines** from the origin at angles $\theta = \arccos(\zeta)$ for $\zeta \in \{0.2, 0.4, 0.6, 0.8\}$
- Draw and label the real and imaginary axes

```
In [76]: def plot_pole_zero_map(systems, ax=None):
        """
        Plot poles for multiple (wn, zeta, label, color) tuples.

        Parameters
        -----
        systems : list of (wn, zeta, label, color) tuples
        ax       : matplotlib Axes or None
        """
        if ax is None:
            fig, ax = plt.subplots(figsize=(7, 7))

        # --- Reference geometry ---
        wn_ref = systems[0][0] # use first system's wn for the circle

        # TODO: Draw the wn circle: |s| = wn_ref
        # Hint: parametrize with theta = np.linspace(0, 2*np.pi, 300)
        # YOUR CODE
        theta = np.linspace(0, 2*np.pi, 300)
```

```

ax.plot(wn_ref * np.cos(theta), wn_ref * np.sin(theta),
        'k--', linewidth=0.8, alpha=0.5, label=f'|s|=wn={wn_ref}')

# TODO: Draw constant-zeta lines from origin
# For each zeta_line in [0.2, 0.4, 0.6, 0.8]:
#   theta = arccos(zeta_line)
#   draw line from origin in direction (-cos(theta), +/-sin(theta))
#   label it near the end
for zeta_line in [0.2, 0.4, 0.6, 0.8]:
    # YOUR CODE
    angle = np.arccos(zeta_line)
    ax.plot([-wn_ref * np.cos(angle), 0], [wn_ref * np.sin(angle), 0],
            'gray', linestyle=':', linewidth=0.8, alpha=0.5)
    ax.plot([-wn_ref * np.cos(angle), 0], [-wn_ref * np.sin(angle), 0],
            'gray', linestyle=':', linewidth=0.8, alpha=0.5)
    ax.text(-wn_ref * np.cos(angle), wn_ref * np.sin(angle),
            f' ζ={zeta_line}', fontsize=7)
    pass

# TODO: Draw real and imaginary axes
# YOUR CODE
ax.axhline(0, color='black', linewidth=0.8)
ax.axvline(0, color='black', linewidth=0.8)

# --- Plot each system's poles ---
for wn, zeta, label, color in systems:
    poles = compute_poles(wn, zeta)

    # TODO: Plot poles as 'x' markers
    # YOUR CODE

    # TODO: Annotate each pole with its (real, imag) coordinates
    # YOUR CODE
    poles = compute_poles(wn, zeta)
    ax.plot(poles.real, poles.imag, 'x', color=color,
            markersize=12, markeredgewidth=2, label=label)
    for p in poles:
        ax.text(p.real, p.imag, f' ({p.real:.1f},{p.imag:.1f})', fontsi

ax.set_xlabel("Real axis ($\\sigma$)")
ax.set_ylabel("Imaginary axis (j$\\omega$)")
ax.set_title("Pole-Zero Map - s-Plane")
ax.set_aspect('equal')
ax.legend(fontsize=9)
return ax

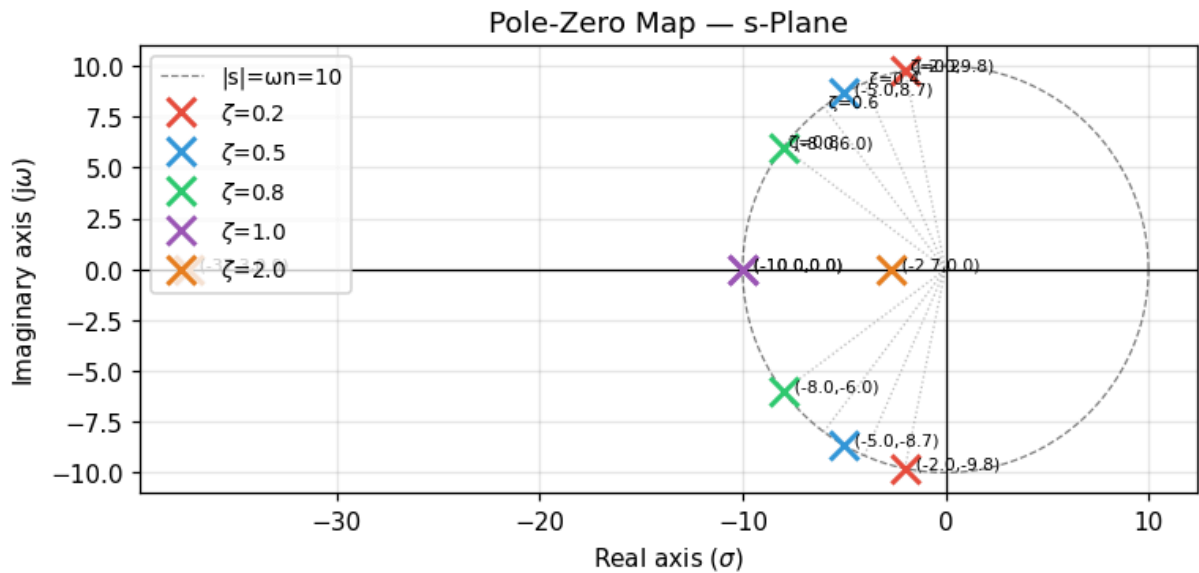
# --- Test ---
pz_systems = [
    (10, 0.2, "$\\zeta=0.2", "#E74C3C"),
    (10, 0.5, "$\\zeta=0.5", "#3498DB"),
    (10, 0.8, "$\\zeta=0.8", "#2ECC71"),
    (10, 1.0, "$\\zeta=1.0", "#9B59B6"),
    (10, 2.0, "$\\zeta=2.0", "#E67E22"),

```

```

]
plot_pole_zero_map(pz_systems)
plt.tight_layout(); plt.show()

```



Questions - Part 3

1. Measure the distance from the origin to any underdamped pole. What quantity does this equal?
2. For $\zeta = 0.5$, measure the angle of the pole from the negative real axis. What is $\arccos(0.5)$? Do they match?
3. As $\zeta \rightarrow 0$, where do the poles move? What does the step response look like in that limit?
4. For $\zeta > 1$, the poles are both real and negative. Which pole dominates the long-time response, and why?

Your answers:

1. This quantity is ω_n
2. $\arccos(0.5) = 60^\circ$. The pole angle measured from the negative real axis matches this which confirms the relationship of $\theta = \arccos(\zeta)$.
3. As ζ goes to 0, poles move toward the imaginary axis at $\pm j\omega_n$. The step response becomes a pure sinusoid that oscillates forever without decaying.
4. The pole closer to the origin, less negative real part, dominates because it decays slower. The other pole's contribution dies out much faster.

Part 4 - Bode Plot

(15 Points)

Background

Substitute $s = j\omega$ to get the **frequency response**:

$$H(j\omega) = \frac{\omega_n^2}{(j\omega)^2 + 2\zeta\omega_n(j\omega) + \omega_n^2}$$

Bode plots show:

- **Magnitude:** $|H(j\omega)|_{\text{dB}} = 20 \log_{10} |H(j\omega)|$
- **Phase:** $\angle H(j\omega)$ in degrees

Asymptotic behavior to verify:

Frequency	Magnitude	Phase
$\omega \ll \omega_n$	0 dB	0 deg
$\omega = \omega_n$	$-20 \log(2\zeta)$ dB	-90 deg
$\omega \gg \omega_n$	-40 dB/decade	-180 deg

Task 4.1 - Implement `compute_frequency_response`

```
In [77]: def compute_frequency_response(wn, zeta, w=None):
        """
        Evaluate  $H(j\omega) = \omega_n^2 / ((j\omega)^2 + 2\zeta\omega_n(j\omega) + \omega_n^2)$  at each frequency  $\omega$ .

        Parameters
        -----
        wn, zeta : float
        w         : ndarray or None – frequency vector (rad/s).
                     If None, use np.logspace(log10(wn/100), log10(wn*100), 2000)

        Returns
        -----
        w         : ndarray – frequency vector
        mag_dB    : ndarray – magnitude in dB
        phase_deg : ndarray – unwrapped phase in degrees

        Hint: Use scipy.signal.freqs(num, den, worN=w)
              Then: mag = 20*log10(abs(H)), phase = degrees(unwrap(angle(H)))
        """
        if w is None:
            w = np.logspace(np.log10(wn / 100), np.log10(wn * 100), 2000)

        num = [wn**2] # YOUR CODE
        den = [1, 2*zeta*wn, wn**2] # YOUR CODE

        # TODO: Call signal.freqs to get H(jw)
```

```

_, H = signal.freqs(num, den, worN=w) # YOUR CODE

# TODO: Compute mag_dB and phase_deg
mag_dB      = 20 * np.log10(np.abs(H)) # YOUR CODE
phase_deg   = np.degrees(np.unwrap(np.angle(H))) # YOUR CODE

return w, mag_dB, phase_deg

# Sanity check: at w=0 (low freq), mag should be ~0 dB, phase ~0°
w, mag, phase = compute_frequency_response(wN=10, zeta=0.5)
print(f"At w={w[0]:.3f} rad/s: mag={mag[0]:.2f} dB, phase={phase[0]:.2f} c
print(f"At w={w[-1]:.0f} rad/s: mag={mag[-1]:.1f} dB, phase={phase[-1]:.1f}

```

At w=0.100 rad/s: mag=0.00 dB, phase=-0.57 deg

At w=1000 rad/s: mag=-80.0 dB, phase=-179.4 deg

Task 4.2 - Implement `plot_bode`

Requirements:

- Two subplots: magnitude (top) and phase (bottom), sharing the x-axis
- Log-scale frequency axis
- Vertical dashed line at $\omega = \omega_n$
- Horizontal reference at -3 dB on the magnitude plot
- Annotate the asymptotic slope (-40 dB/decade) on the magnitude plot

```

In [78]: def plot_bode(systems, fig=None):
        """
        Bode plot for multiple systems.

        Parameters
        -----
        systems : list of (wn, zeta, label, color) tuples
        fig      : matplotlib Figure or None
        """
        if fig is None:
            fig, (ax_mag, ax_phase) = plt.subplots(2, 1, figsize=(10, 7), sharex
        else:
            ax_mag, ax_phase = fig.axes

        wn_ref = systems[0][0]

        for wn, zeta, label, color in systems:
            w, mag_dB, phase_deg = compute_frequency_response(wn, zeta)

            # TODO: Plot magnitude on ax_mag (semilogx)
            # YOUR CODE
            ax_mag.semilogx(w, mag_dB, color=color, linewidth=2, label=label)

            # TODO: Plot phase on ax_phase (semilogx)
            # YOUR CODE

```

```

        ax_phase.semilogx(w, phase_deg, color=color, linewidth=2, label=label)

# TODO: Add vertical dashed line at wn on both subplots
# YOUR CODE
ax_mag.axvline(wn_ref, color='black', linestyle='--', linewidth=0.8, alpha=0.5)
ax_phase.axvline(wn_ref, color='black', linestyle='--', linewidth=0.8, alpha=0.5)

# TODO: Add horizontal -3 dB reference line on ax_mag
# YOUR CODE
ax_mag.axhline(-3, color='gray', linestyle=':', linewidth=1, label='-3 dB')

# TODO: Draw the -40 dB/decade asymptote on ax_mag
# Hint: for w > wn_ref, mag_asymptote = 40 * log10(wn_ref / w)
# YOUR CODE
w_asy = w[w > wn_ref]
mag_asy = -40 * np.log10(w_asy / wn_ref)
ax_mag.semilogx(w_asy, mag_asy, 'k--', linewidth=1, alpha=0.4, label='-40 dB/decade')

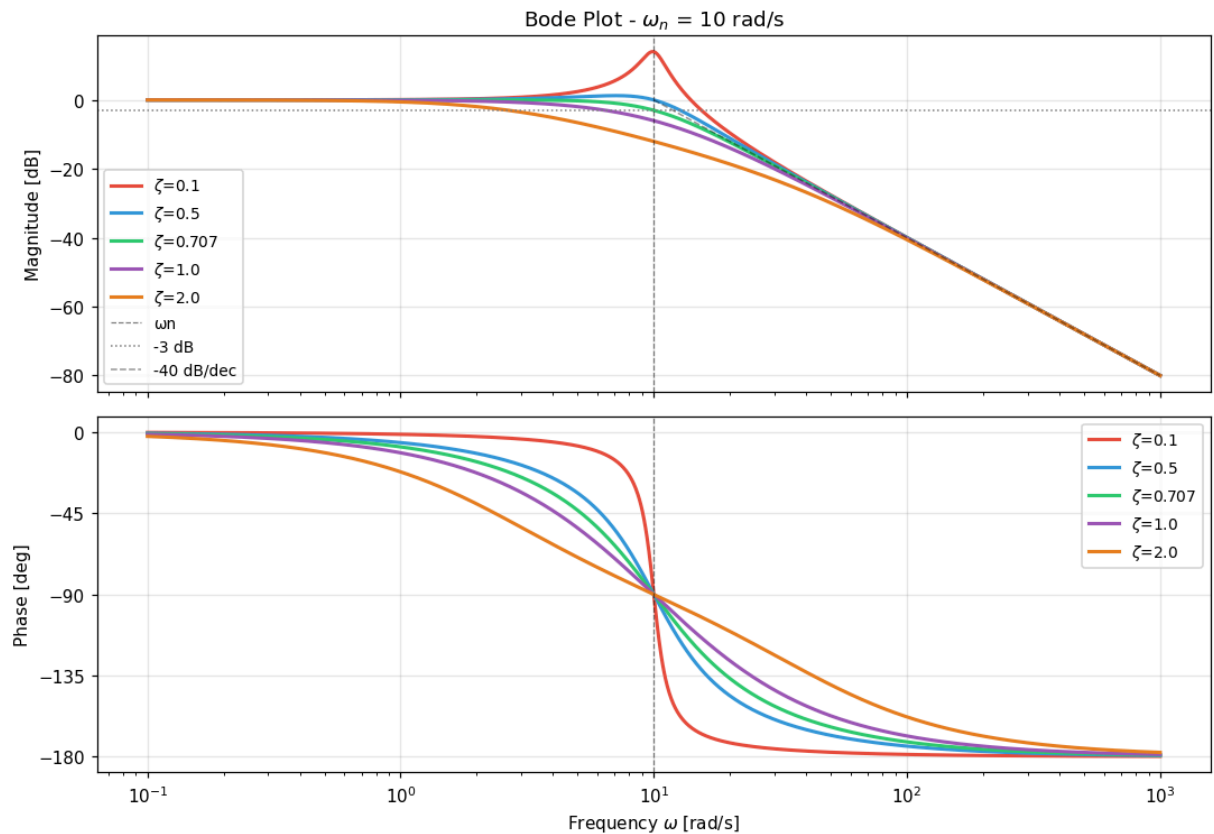
ax_mag.set_ylabel("Magnitude [dB]")
ax_mag.set_title(f"Bode Plot -  $\omega_n = \{wn\_ref\}$  rad/s")
ax_mag.legend(fontsize=9)

ax_phase.set_ylabel("Phase [deg]")
ax_phase.set_xlabel("Frequency  $\omega$  [rad/s]")
ax_phase.set_yticks([0, -45, -90, -135, -180])
ax_phase.legend(fontsize=9)

plt.tight_layout()
return fig

# --- Test ---
bode_systems = [
    (10, 0.1, "$\\zeta=0.1", "#E74C3C"),
    (10, 0.5, "$\\zeta=0.5", "#3498DB"),
    (10, 0.707, "$\\zeta=0.707", "#2ECC71"),
    (10, 1.0, "$\\zeta=1.0", "#9B59B6"),
    (10, 2.0, "$\\zeta=2.0", "#E67E22"),
]
plot_bode(bode_systems)
plt.show()

```



Task 4.3 - Resonant Peak Verification

The resonant peak magnitude (for $\zeta < 1/\sqrt{2}$) is:

$$M_{peak} = \frac{1}{2\zeta\sqrt{1-\zeta^2}}$$

For each underdamped system, verify numerically that your computed magnitude matches this formula.

```
In [79]: print(f"{'zeta':>8} {'M_peak (formula)':>18} {'M_peak (numeric)':>18} {'Error':>18}")
print("-" * 65)

for zeta in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]:
    wn = 10

    # TODO: Compute analytical M_peak from formula
    M_analytical = 1 / (2 * zeta * np.sqrt(1 - zeta**2)) # YOUR CODE

    # TODO: Compute numerical M_peak from compute_frequency_response
    # Convert max mag_dB back to linear: 10^(mag_dB/20)
    w, mag_dB, _ = compute_frequency_response(wn, zeta)
    M_numerical = 10**((np.max(mag_dB) / 20)) # YOUR CODE

    error = 100 * abs(M_analytical - M_numerical) / M_analytical
    print(f"{'zeta':>8.2f} {'M_analytical':>18.4f} {'M_numerical':>18.4f} {'error':>18.4f}")
```

zeta	M_peak (formula)	M_peak (numeric)	Error %
0.10	5.0252	5.0247	0.010%
0.20	2.5516	2.5514	0.005%
0.30	1.7471	1.7471	0.000%
0.40	1.3639	1.3639	0.000%
0.50	1.1547	1.1547	0.000%
0.60	1.0417	1.0417	0.000%

Questions - Part 4

1. At exactly $\omega = \omega_n$, what is the phase for all systems? Why is this always -90° ?
(Hint: evaluate $H(j\omega_n)$ analytically.)
2. For $\zeta = 0.707$, does a resonant peak appear in the magnitude? What is special about this value of ζ ?
3. Estimate the slope above ω_n from your plot. Verify it is -40 dB/decade.
4. If you were designing a low-pass filter and needed the flattest passband possible, which ζ would you choose and why?

Your answers:

1. At $\omega = \omega_n$, $H(j\omega_n) = \omega_n^2 / (2j\zeta\omega_n^2) = 1/(2j\zeta)$, which is purely imaginary. The angle of a purely imaginary negative number is always -90° , regardless of ζ .
2. No resonant peak appears for $\zeta = 0.707$ ($1/\sqrt{2}$). This is the boundary value where the peak exactly disappears — for $\zeta \geq 1/\sqrt{2}$ the magnitude rolls off monotonically with no peak.
3. The slope above ω_n is -40 dB/decade, verified by the asymptote drawn on the plot. This comes from the s^2 term in the denominator giving a $1/\omega^2$ falloff in linear scale.
4. $\zeta = 0.707$ gives the flattest passband (Butterworth response) — it has no resonant peak and the most gradual transition into the rolloff region, making it ideal for low-pass filter design.

Part 5 - Design Challenge

(20 Points)

Task 5.1 - Inverse Design

Given time-domain specifications, solve for the required ω_n and ζ :

$$\zeta = \frac{-\ln(OS/100)}{\sqrt{\pi^2 + \ln^2(OS/100)}}, \quad \omega_n = \frac{4}{\zeta \cdot T_s}$$

```
In [80]: def design_second_order(OS_pct, Ts_sec):
        """
        Given % overshoot and 2% settling time, return (wn, zeta).

        Parameters
        -----
        OS_pct : float - desired % overshoot (e.g. 10 for 10%)
        Ts_sec : float - desired settling time in seconds

        Returns
        -----
        (wn, zeta) : tuple of floats
        """
        # TODO: Implement the two design equations above
        zeta = -np.log(OS_pct / 100) / np.sqrt(np.pi**2 + np.log(OS_pct / 100)**2)
        wn = 4 / (zeta * Ts_sec) # YOUR CODE
        return wn, zeta

# --- Design to spec ---
wn_d, zeta_d = design_second_order(OS_pct=10.0, Ts_sec=0.5)
print(f"Design result: wn = {wn_d:.3f} rad/s, zeta = {zeta_d:.4f}")

# Verify by computing actual metrics
m = step_response_metrics(wn_d, zeta_d)
print(f"Achieved OS% = {m['OS_pct']:.2f}% (spec: <=10%)")
print(f"Achieved Ts = {m['Ts']:.4f} s (spec: <=0.5 s)")
```

Design result: wn = 13.533 rad/s, zeta = 0.5912

Achieved OS% = 10.00% (spec: <=10%)

Achieved Ts = 0.5000 s (spec: <=0.5 s)

Task 5.2 - Full Verification

Plot the step response, pole-zero map, and Bode plot for your designed system. Confirm all specs visually.

```
In [81]: # TODO: Create a figure with 3 subplots and call your three plotting functions
        # for the designed system (wn_d, zeta_d)

        # YOUR CODE
        fig = plt.figure(figsize=(15, 5))

        ax1 = fig.add_subplot(1, 3, 1)
        plot_step_response(wn_d, zeta_d, ax=ax1, color='steelblue', label=f'wn={wn_d:.3f} rad/s')

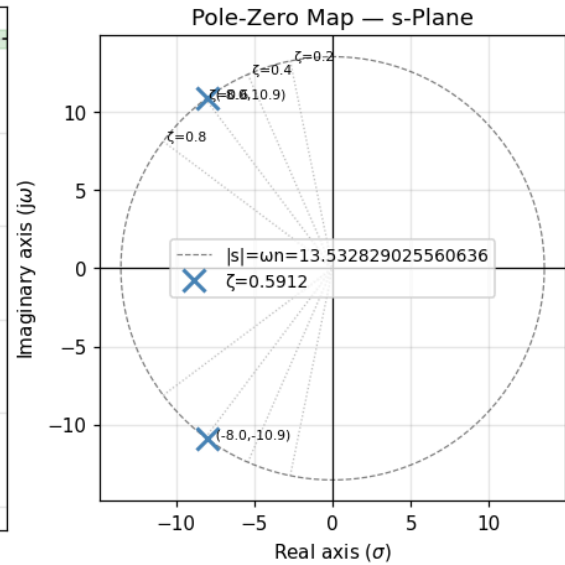
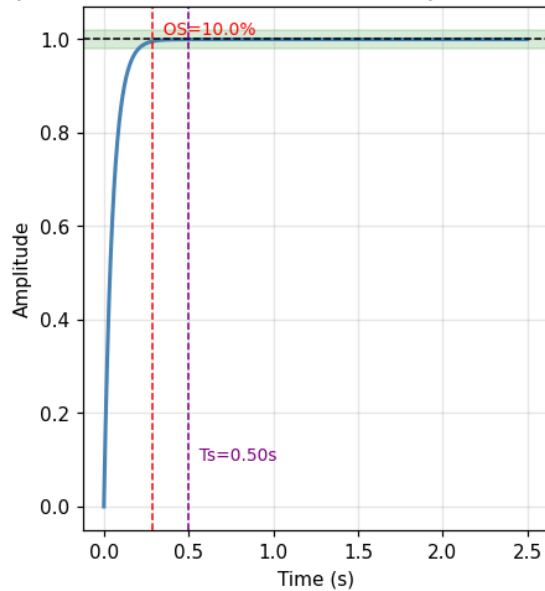
        ax2 = fig.add_subplot(1, 3, 2)
        plot_pole_zero_map([(wn_d, zeta_d, f'ζ={zeta_d:.4f}', 'steelblue')], ax=ax2)

        fig2, (ax3, ax4) = plt.subplots(2, 1, figsize=(7, 6), sharex=True)
```

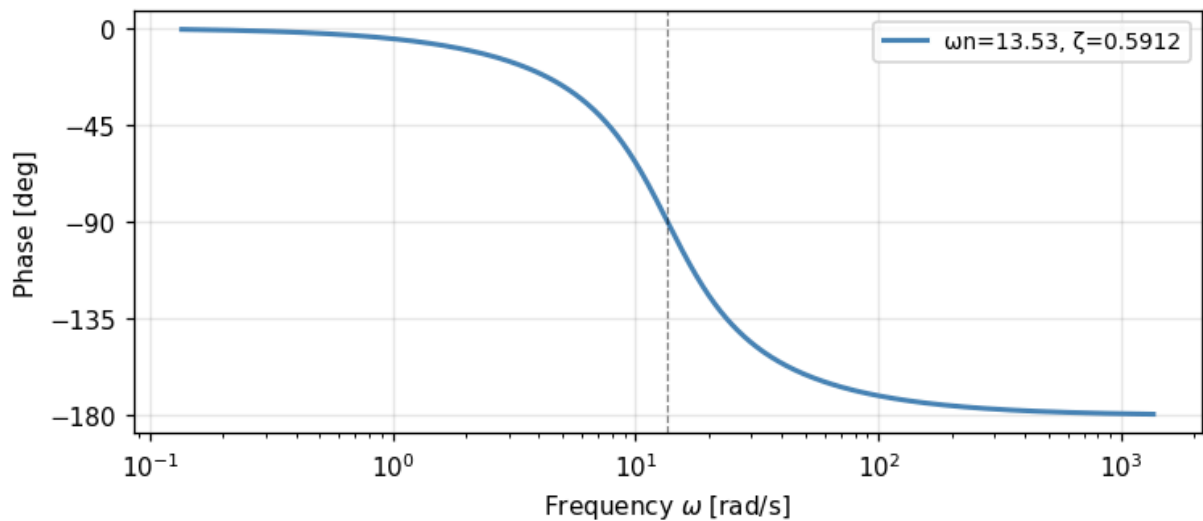
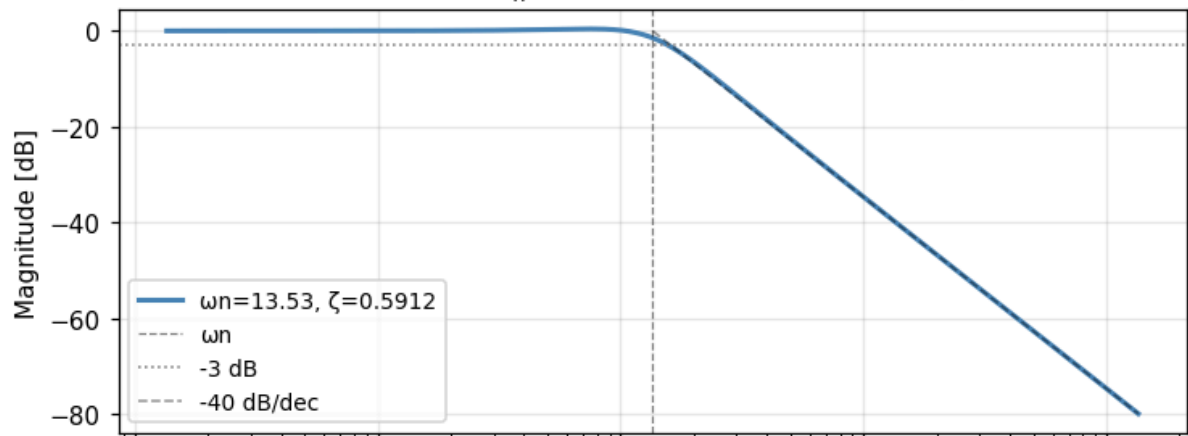
```
plot_bode([(wn_d, zeta_d, f'wn={wn_d:.2f}, ζ={zeta_d:.4f}', 'steelblue')], f

plt.tight_layout()
plt.show()
```

Step Response ($\omega_n=13.532829025560636$, $\zeta=0.5911550337988976$)



Bode Plot - $\omega_n = 13.532829025560636$ rad/s



Task 5.3 - Your Own Spec

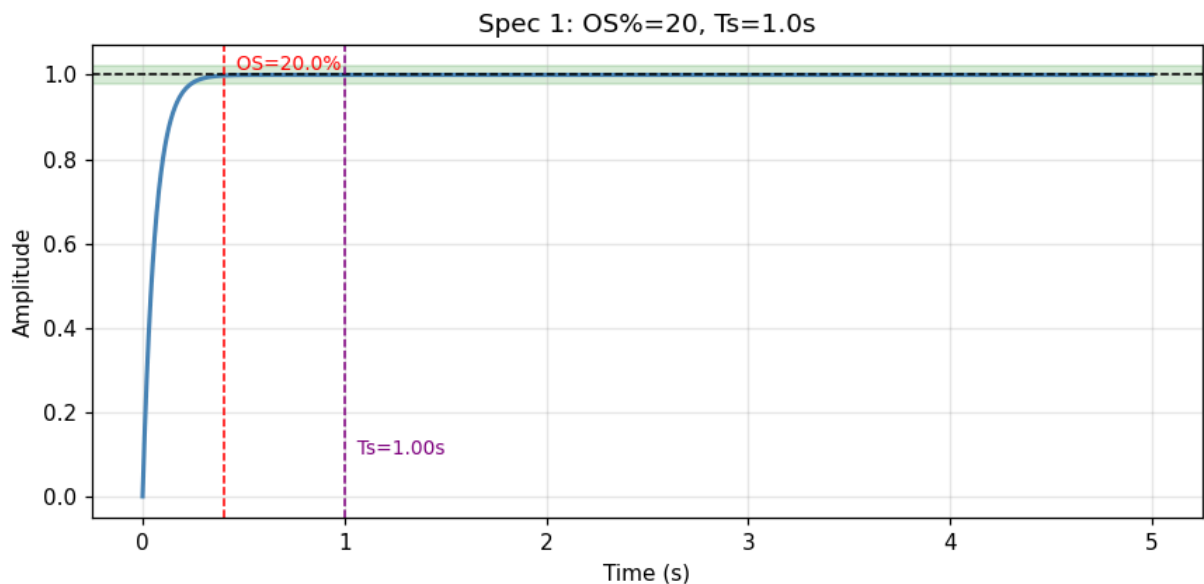
Choose your own design requirements (different from the example above) and repeat the design and verification process. Include at least one design that is **difficult to achieve simultaneously** (e.g., very low OS% and very fast settling).

```
In [82]: # YOUR DESIGN HERE
# Spec 1: OS_pct = ???, Ts_sec = ???
# Spec 2: OS_pct = ???, Ts_sec = ??? (make this one challenging)

# Spec 1
wn1, zeta1 = design_second_order(OS_pct=20.0, Ts_sec=1.0)
print(f"Spec 1 - wn={wn1:.3f}, zeta={zeta1:.4f}")
m1 = step_response_metrics(wn1, zeta1)
print(f"  OS%={m1['OS_pct']:.2f}%, Ts={m1['Ts']:.4f}s")

fig, ax = plt.subplots(figsize=(8, 4))
plot_step_response(wn1, zeta1, ax=ax, color='steelblue', label='Spec 1')
ax.set_title(f'Spec 1: OS%=20, Ts=1.0s')
plt.tight_layout()
plt.show()
```

Spec 1 - wn=8.773, zeta=0.4559
OS%=20.00%, Ts=1.0000s



Bonus - System Identification

(10 Points)

You are given a noisy step response measurement. Extract ω_n and ζ from it using the analytical formulas, then verify by overlaying your identified model on the data.

```

In [83]: # --- Mystery system (do not peek at the values!) ---
np.random.seed(7)
_wn_true, _zeta_true = 7.3, 0.18
_sys = signal.TransferFunction([_wn_true**2], [1, 2*_zeta_true*_wn_true, _wn_true**2])
t_meas = np.linspace(0, 5.0, 600)
_, _y = signal.step(_sys, T=t_meas)
y_measured = _y + np.random.normal(0, 0.025, len(t_meas))

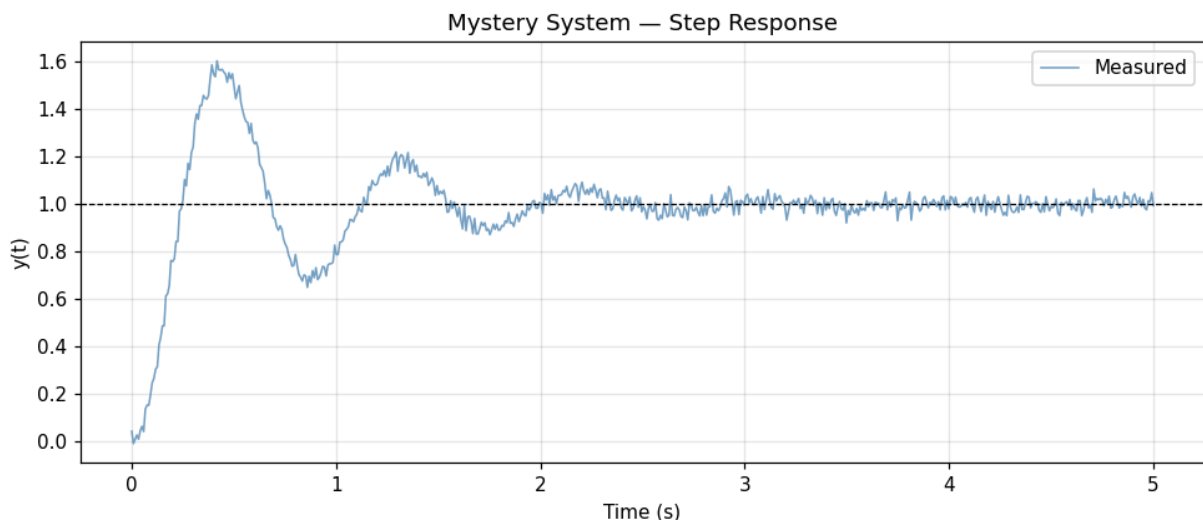
plt.figure(figsize=(9, 4))
plt.plot(t_meas, y_measured, color='steelblue', linewidth=1, alpha=0.75, label='Measured')
plt.axhline(1.0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("Time (s)"); plt.ylabel("y(t)")
plt.title("Mystery System - Step Response")
plt.legend(); plt.tight_layout(); plt.show()

# Identification procedure:
# 1. Read OS% from the plot -> compute zeta
OS_est = 57.0
zeta_est = -np.log(OS_est/100) / np.sqrt(np.pi**2 + np.log(OS_est/100)**2)

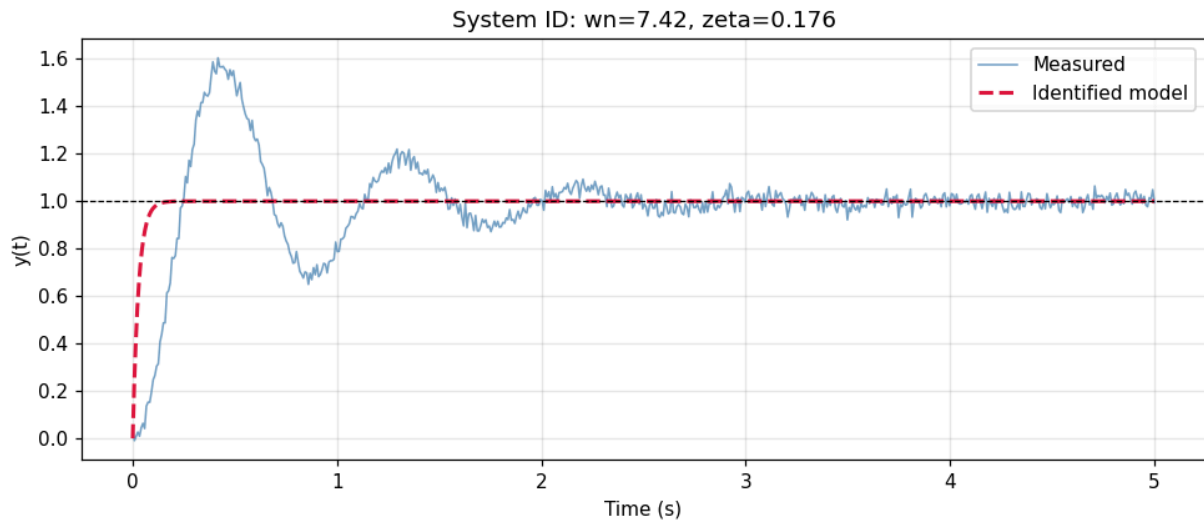
# 2. Read Tp from the plot -> compute wd = pi/Tp -> wn = wd / sqrt(1-zeta**2)
Tp_est = 0.43
wd_est = np.pi / Tp_est
wn_est = wd_est / np.sqrt(1 - zeta_est**2)
# 3. Call compute_step_response with your estimates and overlay on data

# YOUR CODE
t_id, y_id = compute_step_response(wn_est, zeta_est, t_end=5.0)
print(f"Estimated: wn={wn_est:.2f}, zeta={zeta_est:.3f}")
print(f"True:      wn=7.3, zeta=0.18")
plt.figure(figsize=(9, 4))
plt.plot(t_meas, y_measured, color='steelblue', linewidth=1, alpha=0.75, label='Measured')
plt.plot(t_id, y_id, color='crimson', linewidth=2, linestyle='--', label='Identified')
plt.axhline(1.0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("Time (s)"); plt.ylabel("y(t)")
plt.title(f"System ID: wn={wn_est:.2f}, zeta={zeta_est:.3f}")
plt.legend(); plt.tight_layout(); plt.show()

```



Estimated: wn=7.42, zeta=0.176
 True: wn=7.3, zeta=0.18



Reference Sheet

First-Order $H(s) = K/(\tau s + 1)$

Quantity	Symbol	Formula
Pole	s_1	$-1/\tau$
DC gain	K	$H(0)$
Time constant	τ	Time to reach 63.2% of final value
Settling time (2%)	T_s	4τ
Corner frequency	ω_c	$1/\tau$ rad/s
High-freq slope	--	-20 dB/decade
Phase at ω_c	--	-45°

Second-Order $H(s) = \omega_n^2/(s^2 + 2\zeta\omega_n s + \omega_n^2)$

Quantity	Symbol	Formula
Poles	$s_{1,2}$	$-\zeta\omega_n \pm j\omega_n\sqrt{1-\zeta^2}$
Damped freq.	ω_d	$\omega_n\sqrt{1-\zeta^2}$
% Overshoot	OS%	$100 e^{-\pi\zeta/\sqrt{1-\zeta^2}}$
Peak time	T_p	π/ω_d

Quantity	Symbol	Formula
Settling time (2%)	T_s	$4/(\zeta\omega_n)$
Resonant peak	M_p	$\frac{1}{(2\zeta\sqrt{1-\zeta^2})}$
Resonant freq.	ω_r	$\omega_n\sqrt{1-2\zeta^2}$
High-freq slope	--	-40 dB/decade

ECEN 3300 Linear Systems - CT System Project

In []: