Introduction to Control in MATLAB

Wyatt Cross

February 2026

Contents

1 Introduction

This document outlines the fundamental MATLAB functions used in control systems. Partial fraction decomposition is best left to computers, so we can earn the big buck$. In this document, valid MATLAB commands are written in monospace like this: `tf()`. You need the MATLAB Control Systems Toolbox to use many of these functions. I use a couple systems in this document, if you are following along in MATLAB, `sys1`, `sys2`, etc. refer to different transfer functions.

2 Creating Transfer Functions

A transfer function relates the output of a **linear, time-invariant (LTI)** system to its input in the Laplace domain. You can create a continuous-time transfer function in MATLAB with one of two ways.

2.1 Numerator and Denominator Arrays

The most common way of defining a transfer function is by providing arrays of the polynomial coefficients for the numerator and denominator, ordered from highest power of s to lowest. Consider the following transfer

function:

$$G(s) = \frac{s+10}{s^3+2s+10}$$

Remember to include the zero coefficient in the denominator

```
num = [1, 10];          % Numerator coefficients
den = [1, 0, 2, 10];    % Denominator coefficients
sys1 = tf(num, den);    % Creates the transfer function object
```

2.2 s as a Transfer Function Variable

Another method that works well for complex equations is explicitly telling MATLAB that `s` is the Laplace variable, and then you just write the math out exactly as it is given.

```
s = tf('s');
sys1 = (s + 10) / (s^3 + 2*s + 10);
```

3 Poles, Zeros, and Damping

System behavior can be automatically found using a couple really simple to use functions.

3.1 Poles and Zeros

Poles determine stability and transient response, zeros affect overshoot (Among other things). The `pzmap` function can generate a map of the pole locations on the complex plane 1

```
p = pole(sys1) % Returns a vector of the system's poles
%ans =
%   0.9237 + 2.1353i
%   0.9237 - 2.1353i
%  -1.8474 + 0.0000i

z = zero(sys1) % Returns a vector of the system's zeros
%ans =
%   -10
pzmap(sys1)    % Creates a pole-zero plot on the complex plane
               % (You can click on the poles to get info about them)
```

3.2 Damping and Frequency

Damping is how quickly oscillations decay back to equilibrium. The `damp()` function prints a table showing the eigenvalues (poles), damping ratio (ζ), natural frequency (ω_n), and time constant (τ).

```
damp(sys1) % Outputs a table

%          Pole               Damping       Frequency      Time Constant
%                                          (rad/seconds)      (seconds)
%
% -1.85e+00                   1.00e+00       1.85e+00          5.41e-01
%  9.24e-01 + 2.14e+00i      -3.97e-01       2.33e+00         -1.08e+00
%  9.24e-01 - 2.14e+00i      -3.97e-01       2.33e+00         -1.08e+00
```

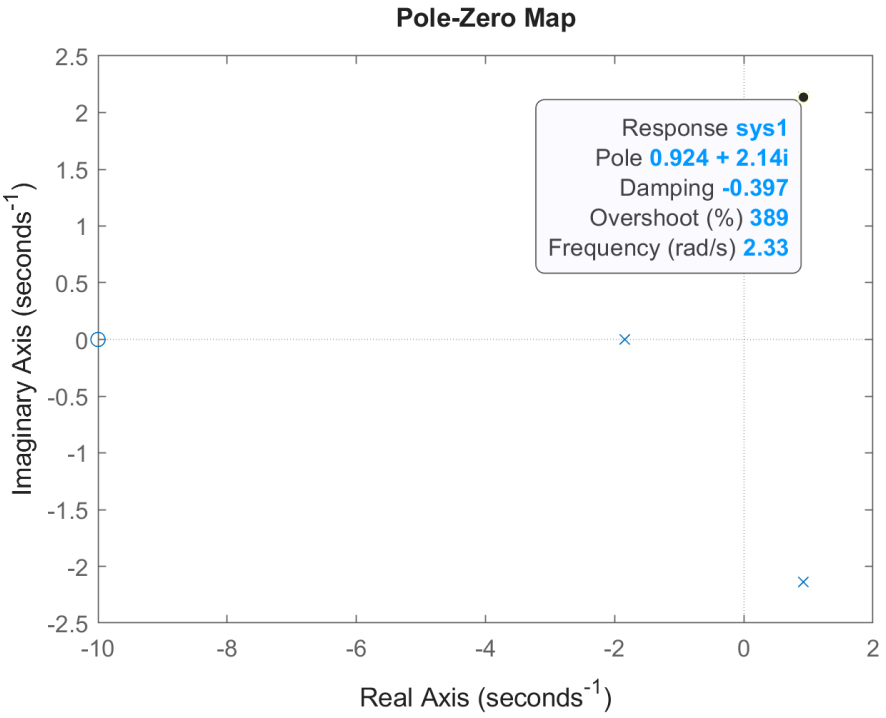

Figure 1: Output of `pzmap(sys1)`. Poles are denoted by crosses and zeros by circles.

4 Time-Domain Response

Consider the following transfer function:

$$G(s) = \frac{5}{s^2 + 5s + 25}$$

4.1 Step Response (`step`)

The step response simulates how the system reacts to a sudden and sustained input. It is very useful for seeing if part of your system will blow up when disturbed, and how it responds in 2.

```
sys2 = tf(5,[1 5 25]);    % Define an open-loop system
step(sys2)                % Step response

% If you want to plot for a specified amount of time, do
t = 0:0.01:30;      % Create a time domain
y = step(sys2, t);
plot(t, y)
```

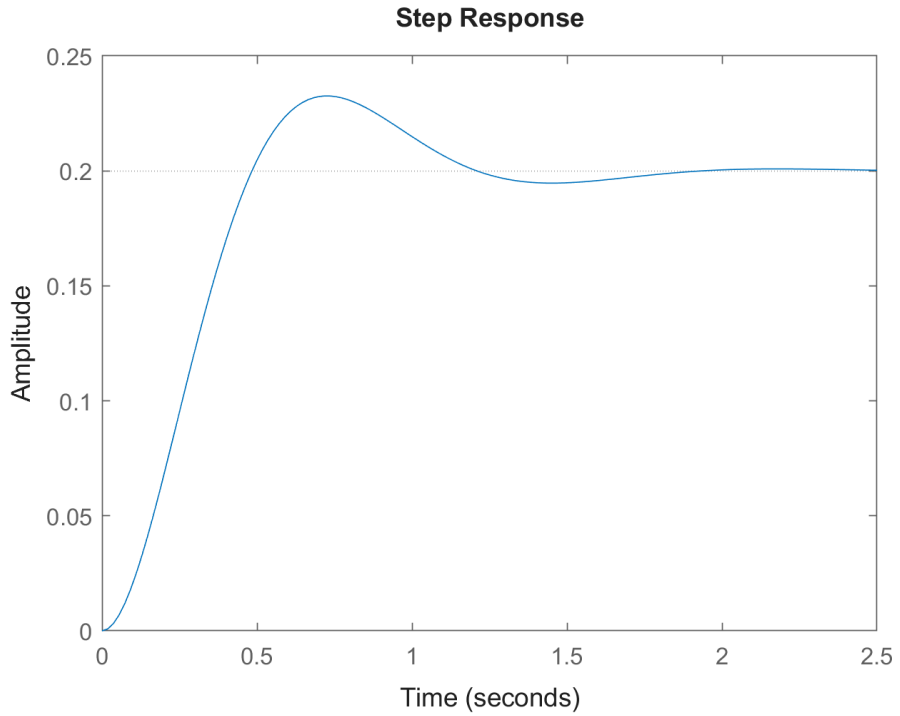

Figure 2: Plot generated by `step(sys2)`. Note, since the poles are $-2.5 \pm 4.3301j$, the system is stable and will eventually converge, but will be oscillatory from the complex component.

4.2 Step Characteristics (`stepinfo`)

Instead of staring at a graph to find the exact peak time or settling time, MATLAB has a function that calculates standard performance metrics for you.

```
stepinfo(sys2) % Returns a struct with performance parameters
%ans =
%          RiseTime: 0.3278
%     TransientTime: 1.6152
%      SettlingTime: 1.6152
%       SettlingMin: 0.1863
%       SettlingMax: 0.2326
%         Overshoot: 16.2929
%        Undershoot: 0
%              Peak: 0.2326
%          PeakTime: 0.7184
```

Note, the default rise time setting in `lsim` is not 63% (It is from 10% to 90%).

Call `stepinfo(sys2,RiseTimeThreshold=[0,0.63])` to specify that you want the time the system takes to go from 0% to 63% of its final value.

4.3 Simulating Arbitrary Inputs (`lsim`)

The `step()` function assumes a unit step input which rarely happens in reality. The `lsim()` function allows you to simulate the time response of a system to **any arbitrary input signal** 3.

```
t = 0:0.01:10;    % Time vector. 0 to 10 seconds with enough resolution between
```

```
u = zeros(length(t),1); u(100:400) = 4; % Define arbitrary input
y = lsim(sys2, u, t);  % Simulate response
figure; hold on;
plot(t,u); plot(t,y)
legend('Command','Actual');
```

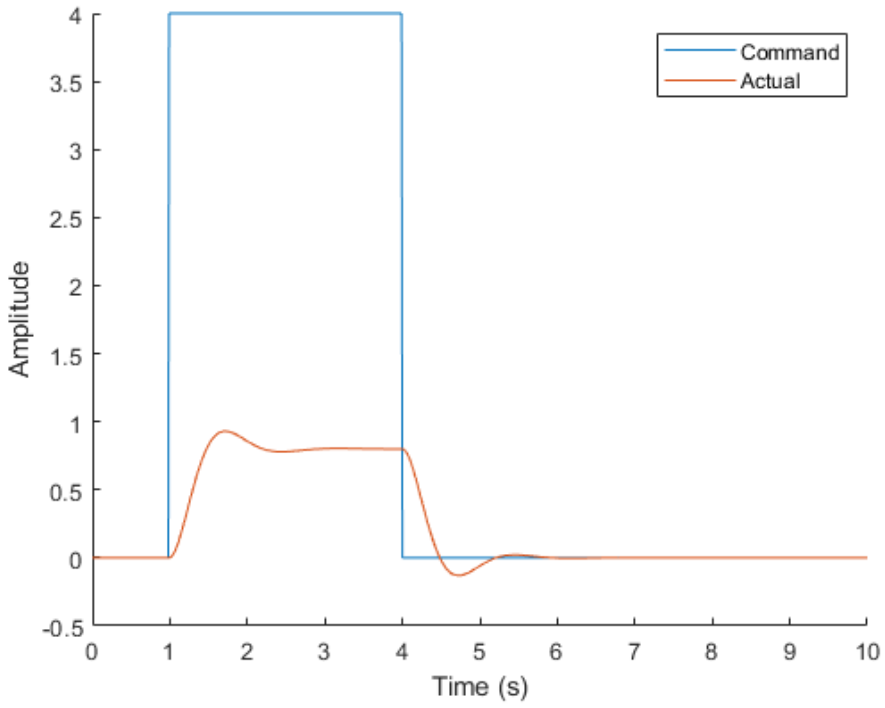

Figure 3: Plot of output of `lsim`. Note how the output stabilizes around 0.8 for this magnitude 4 step input. This is a result of the DC gain, since $G(0) = \frac{5}{0^2+5*0+25} = 0.2$ for a unit step. Since the step here is magnitude 4, the open loop system ends up at 0.8 (Try `dcgain(sys2)` to check that the unit step DC gain is 0.2). The fact that it restores to zero is a function of the disturbance ceasing and the system being asymptotically stable.

The `u` variable in this snippet can be any signal you want to evaluate (in this case, the open loop system) against. Here, I used a magnitude 4 step from 1 to 4 seconds. Other common options include using `u=SOME_CONSTANT*ones(size(t))` to see how the system reaches a desired value.

5 Frequency Response

5.1 Bode Plots (`bode`)

The Bode plot consists of two graphs: magnitude and phase, both plotted against frequency 4. It tells you how much your system amplifies or attenuates an oscillating signal, and how much the signal's timing is shifted.

```
bode(sys2) % Plot of magnitude (dB) and phase (degrees) vs frequency (rad/s)
```

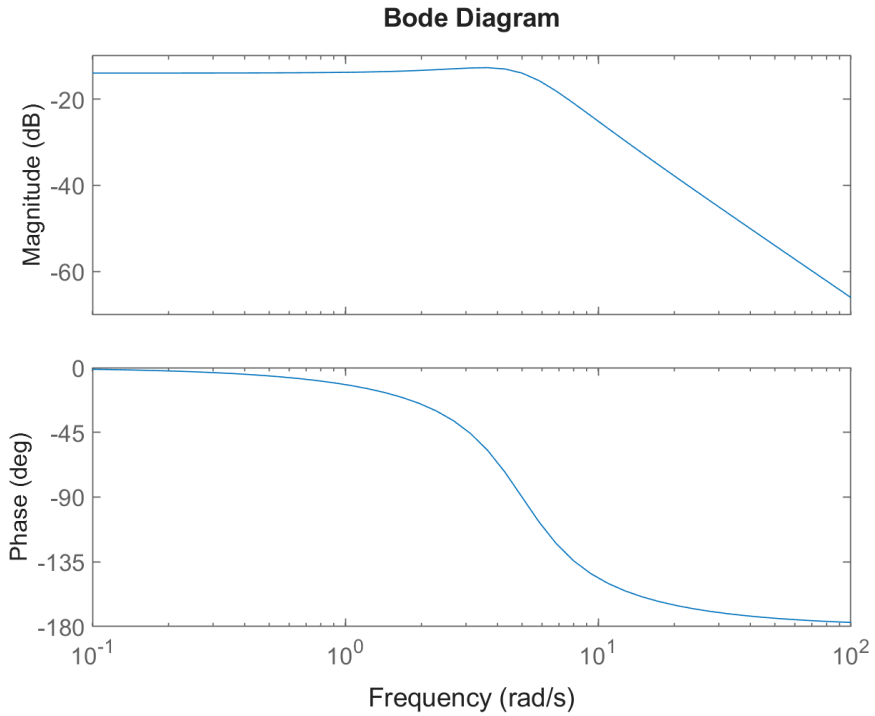

Figure 4: Output of `bode(sys2)`.

5.2 Stability Margins (`margin`)

If a feedback loop has too much delay (phase lag) or reacts too aggressively (high gain), it becomes unstable, which can cause undesirable oscillations. The `margin()` function calculates exactly how close the system is to instability.

```
margin(sys1) % Plots the Bode response with lables for gain and phase margin
```

Calling `[Gm, Pm, Wcg, Wcp] = margin(sys1)` will save the Gain margin (`Gm`), Phase margin (`Pm`), and their corresponding crossover frequencies to variables instead of plotting.

6 Root Locus

The Root Locus method shows how the closed-loop poles of a system move as a single parameter (usually the controller gain, K) varies from zero to infinity 5. It is very helpful to predict system stability.

```
rlocus(sys1) % Generates the root locus plot
```

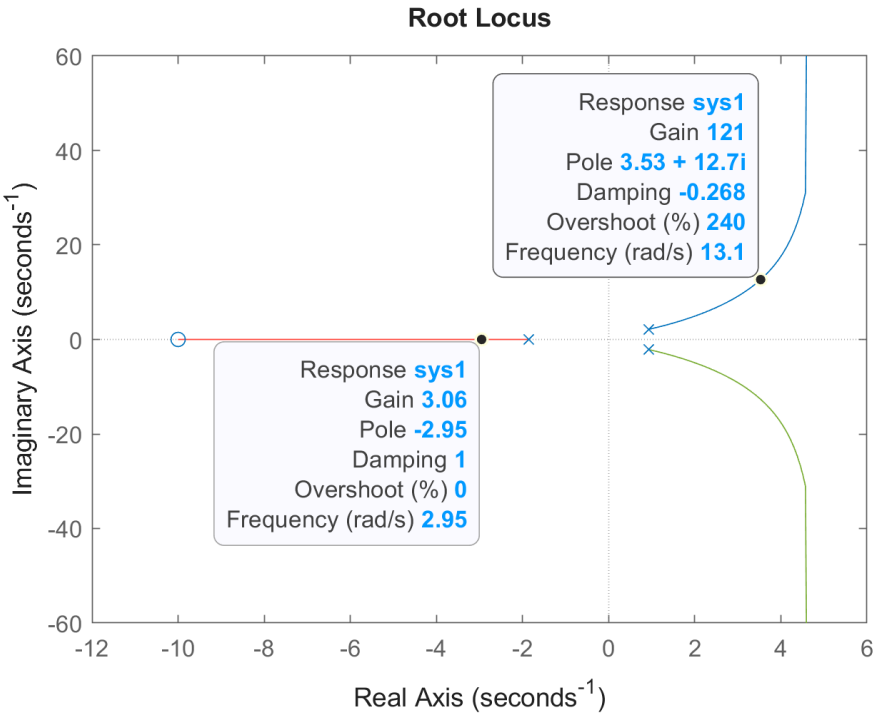

Figure 5: Output of `rlocus(sys1)`.

7 State-Space Representations

Real systems usually have multiple inputs affecting multiple outputs (MIMO). State-space models track internal system states over time with matrix differential equations:

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

$$\mathbf{y} = C\mathbf{x} + D\mathbf{u}$$

Creating a State-Space Model

You can define a system directly using the A, B, C, and D matrices.

```
% State space matrices for a 2-state system
A = [-0.5, 0; 1, -2];  % State matrix (comes from system dynamics)
B = [1; 0];            % Control matrix (determines influence on internal states)
C = [0, 1];            % Output matrix (maps internal states to outputs)
D = 0;                 % Feedthrough Matrix (usually zero for our problems.
    coupling that bypasses the system)
sys_ss = ss(A, B, C, D); % Creates the state-space object
```

Converting Between Representations

MATLAB makes it trivial to convert a transfer function into a state-space model, and vice versa. This is incredibly useful if you are given a mathematical model in one format but need to analyze it using tools better suited for the other.

```
ss_to_tf = tf(sys_ss); % Converts a state-space model to a transfer function
tf_to_ss = ss(sys1);   % Converts a transfer function to a state-space model
```

8 Connecting Systems

Real-world models are rarely a single block. They consist of cascade and feedback loops. The `series` function connects systems in cascade (multiplication of transfer functions), while the `feedback` function closes a feedback loop.

```
sys_series = series(sys1, sys2); % Equivalent to sys1 * sys2

sys_closed = feedback(sys_series, 1); % Close the loop with unity negative
    feedback
```

By default, `feedback(G,H)` computes the following, which corresponds to unity negative feedback when $H = 1$.

$$\frac{G}{1 + GH}$$