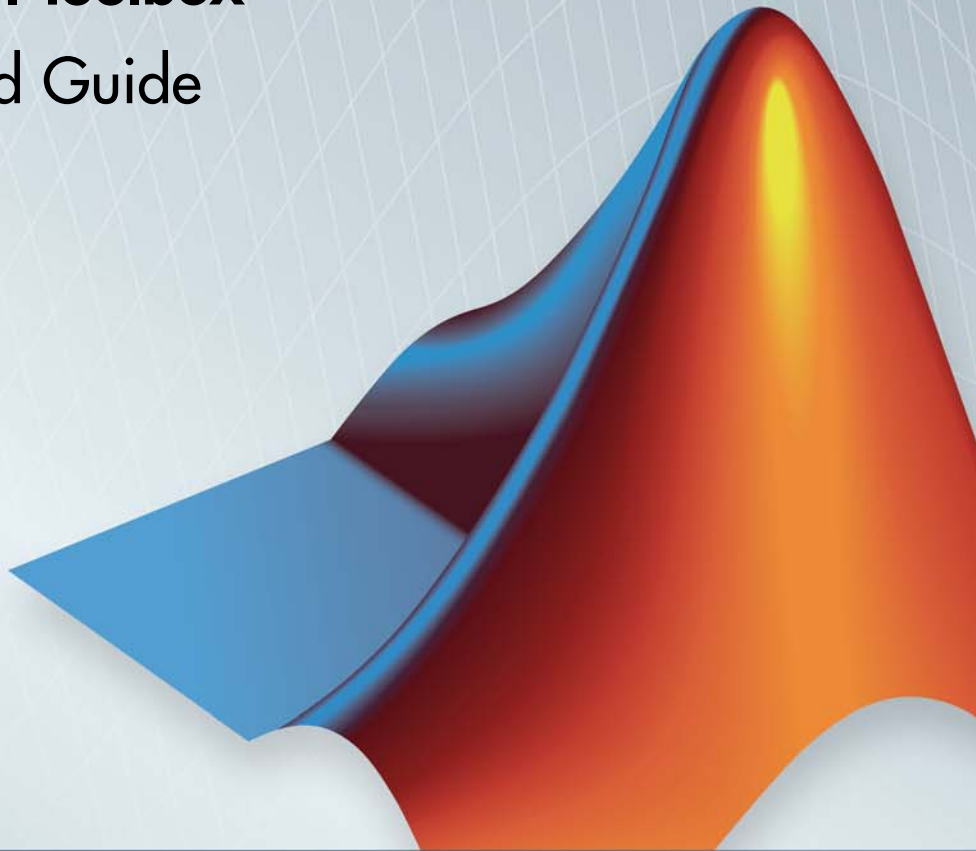


# Robust Control Toolbox™

## Getting Started Guide

**R2013b**

*Gary Balas  
Richard Chiang  
Andy Packard  
Michael Safonov*



# MATLAB®

## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



The MathWorks, Inc.  
3 Apple Hill Drive  
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

*Robust Control Toolbox™ Getting Started Guide*

© COPYRIGHT 2005–2013 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

September 2005	First printing	New for Version 3.0.2 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.1.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.4.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.5 (Release 2010b)
April 2011	Online only	Revised for Version 3.6 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.3 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)



## Introduction

**1**

<b>Robust Control Toolbox Product Description</b> .....	1-2
Key Features .....	1-2
<b>Product Requirements</b> .....	1-3
<b>Modeling Uncertainty</b> .....	1-4
<b>System with Uncertain Parameters</b> .....	1-5
<b>Worst-Case Performance</b> .....	1-9
<b>Worst-Case Performance of Uncertain System</b> .....	1-10
<b>Loop-Shaping Controller Design</b> .....	1-12
<b>Model Reduction and Approximation</b> .....	1-16
<b>Reduce Order of Synthesized Controller</b> .....	1-17
<b>LMI Solvers</b> .....	1-21
<b>Extends Control System Toolbox Capabilities</b> .....	1-22
<b>Acknowledgments</b> .....	1-23
<b>Bibliography</b> .....	1-25

## Multivariable Loop Shaping

### 2

<b>Tradeoff Between Performance and Robustness</b> .....	2-2
<b>Norms and Singular Values</b> .....	2-4
Properties of Singular Values .....	2-4
<b>Typical Loop Shapes, S and T Design</b> .....	2-6
Robustness in Terms of Singular Values .....	2-7
Guaranteed Gain/Phase Margins in MIMO Systems .....	2-12
<b>Using LOOPSYN to Do H-Infinity Loop Shaping</b> .....	2-15
<b>Loop-Shaping Control Design of Aircraft Model</b> .....	2-16
Design Specifications .....	2-18
MATLAB Commands for a LOOPSYN Design .....	2-18
<b>Fine-Tuning the LOOPSYN Target Loop Shape <math>G_d</math> to Meet Design Goals</b> .....	2-21
<b>Mixed-Sensitivity Loop Shaping</b> .....	2-22
<b>Mixed-Sensitivity Loop-Shaping Controller Design</b> ...	2-24
<b>Loop-Shaping Commands</b> .....	2-26

## Model Reduction for Robust Control

### 3

<b>Why Reduce Model Order?</b> .....	3-2
<b>Hankel Singular Values</b> .....	3-3
<b>Model Reduction Techniques</b> .....	3-5

<b>Approximate Plant Model by Additive Error</b>	
<b>Methods</b> .....	<b>3-7</b>
<b>Approximate Plant Model by Multiplicative Error</b>	
<b>Method</b> .....	<b>3-9</b>
<b>Using Modal Algorithms</b> .....	<b>3-12</b>
Rigid Body Dynamics .....	<b>3-12</b>
<b>Reducing Large-Scale Models</b> .....	<b>3-16</b>
<b>Normalized Coprime Factor Reduction</b> .....	<b>3-17</b>
<b>Bibliography</b> .....	<b>3-19</b>

## Robustness Analysis

# 4

<b>Create Models of Uncertain Systems</b> .....	<b>4-2</b>
Creating Uncertain Models of Dynamic Systems .....	<b>4-2</b>
Creating Uncertain Parameters .....	<b>4-3</b>
Quantifying Unmodeled Dynamics .....	<b>4-6</b>
<b>Robust Controller Design</b> .....	<b>4-9</b>
<b>MIMO Robustness Analysis</b> .....	<b>4-14</b>
Adding Independent Input Uncertainty to Each Channel .....	<b>4-15</b>
Closed-Loop Robustness Analysis .....	<b>4-19</b>
Nominal Stability Margins .....	<b>4-21</b>
Robustness of Stability Model Uncertainty .....	<b>4-22</b>
Worst-Case Gain Analysis .....	<b>4-23</b>
<b>Summary of Robustness Analysis Tools</b> .....	<b>4-27</b>

## H-Infinity and Mu Synthesis

### 5

<b>Interpretation of H-Infinity Norm</b> .....	5-2
Norms of Signals and Systems .....	5-2
Using Weighted Norms to Characterize Performance ....	5-3
<b>H-Infinity Performance</b> .....	5-9
Performance as Generalized Disturbance Rejection .....	5-9
Robustness in the H-Infinity Framework .....	5-15
<b>Active Suspension Control Design</b> .....	5-17
<b>Bibliography</b> .....	5-38

## Tuning Fixed Control Architectures

### 6

<b>What Is a Fixed-Structure Control System?</b> .....	6-3
<b>Choosing an Approach to Fixed Control Structure Tuning</b> .....	6-4
<b>Difference Between Fixed-Structure Tuning and Traditional H-Infinity Synthesis</b> .....	6-5
Bibliography .....	6-5
<b>Structure of Control System for Tuning With looptune</b> .....	6-6
<b>Set Up Your Control System for Tuning with looptune</b> .....	6-8
Set Up Your Control System for looptunein MATLAB ....	6-8
Set Up Your Control System for looptune in Simulink ....	6-8



<b>Performance and Robustness Specifications for looptune</b> .....	<b>6-10</b>
<b>Tune MIMO Control System for Specified Bandwidth</b> .....	<b>6-12</b>
<b>What Is hinfstruct?</b> .....	<b>6-19</b>
<b>Formulating Design Requirements as H-Infinity Constraints</b> .....	<b>6-20</b>
<b>Structured H-Infinity Synthesis Workflow</b> .....	<b>6-21</b>
<b>Build Tunable Closed-Loop Model for Tuning with hinfstruct</b> .....	<b>6-22</b>
Constructing the Closed-Loop System Using Control System Toolbox Commands .....	<b>6-22</b>
Constructing the Closed-Loop System Using Simulink Control Design Commands .....	<b>6-26</b>
<b>Tune the Controller Parameters</b> .....	<b>6-29</b>
<b>Interpret the Outputs of hinfstruct</b> .....	<b>6-30</b>
Output Model is Tuned Version of Input Model .....	<b>6-30</b>
Interpreting gamma .....	<b>6-30</b>
<b>Validate the Controller Design</b> .....	<b>6-31</b>
Validating the Design in MATLAB .....	<b>6-31</b>
Validating the Design in Simulink .....	<b>6-32</b>
<b>Set Up Your Control System for Tuning with systune</b> ..	<b>6-35</b>
Set Up Your Control System for systune in MATLAB ....	<b>6-35</b>
Set Up Your Control System for systune in Simulink ....	<b>6-35</b>
<b>Specifying Design Requirements for systune</b> .....	<b>6-37</b>
<b>Tune Controller Against Set of Plant Models</b> .....	<b>6-39</b>

<b>Supported Blocks for Tuning in Simulink</b> .....	<b>6-40</b>
Tuning Other Blocks .....	<b>6-40</b>
<b>Speed Up Tuning with Parallel Computing Toolbox</b>	
Software .....	<b>6-42</b>
<b>Tuning Control Systems with SYSTUNE</b> .....	<b>6-44</b>
<b>Tuning Feedback Loops with LOOPTUNE</b> .....	<b>6-54</b>
<b>Tuning Control Systems in Simulink</b> .....	<b>6-62</b>
<b>Building Tunable Models</b> .....	<b>6-72</b>
<b>Using Design Requirement Objects</b> .....	<b>6-81</b>
<b>Validating Results</b> .....	<b>6-102</b>
<b>Tuning Multi-Loop Control Systems</b> .....	<b>6-112</b>
<b>Using Parallel Computing to Accelerate Tuning</b> .....	<b>6-124</b>
<b>PID Tuning for Setpoint Tracking vs. Disturbance</b>	
Rejection .....	<b>6-129</b>
<b>Decoupling Controller for a Distillation Column</b> .....	<b>6-141</b>
<b>Tuning of a Digital Motion Control System</b> .....	<b>6-154</b>
<b>Control of a Linear Electric Actuator</b> .....	<b>6-175</b>
<b>Multi-Loop PID Control of a Robot Arm</b> .....	<b>6-187</b>
<b>Active Vibration Control in Three-Story Building</b> .....	<b>6-205</b>
<b>Tuning of a Two-Loop Autopilot</b> .....	<b>6-220</b>

<b>Multi-Loop Control of a Helicopter</b> .....	<b>6-238</b>
<b>Fixed-Structure Autopilot for a Passenger Jet</b> .....	<b>6-248</b>
<b>Fault-Tolerant Control of a Passenger Jet</b> .....	<b>6-262</b>
<b>Fixed-Structure H-infinity Synthesis with HINFSTRUCT</b> .....	<b>6-274</b>
<b>MIMO Control of Diesel Engine</b> .....	<b>6-286</b>
<b>Digital Control of Power Stage Voltage</b> .....	<b>6-304</b>

## Gain-Scheduled Controllers

# 7

<b>Gain-Scheduled Control Systems</b> .....	<b>7-2</b>
<b>Plant Models for Gain-Scheduled Control</b> .....	<b>7-4</b>
Gain Scheduling for Linear Parameter-Varying Plants ...	7-4
Gain Scheduling for Nonlinear Plants .....	7-5
<b>Parametric Gain Surfaces</b> .....	<b>7-8</b>
<b>Tuning Gain-Scheduled Controllers</b> .....	<b>7-13</b>
<b>Validating Gain-Scheduled Controllers</b> .....	<b>7-14</b>
<b>Improving Gain-Scheduled Tuning Results</b> .....	<b>7-15</b>
Normalize the Scheduling Variables .....	7-15
Changing Requirements With Operating Condition .....	7-16
<b>Tunable Gain Surface With Two Scheduling Variables</b> .....	<b>7-18</b>

<b>Gain-Scheduled PID Controller</b> .....	<b>7-22</b>
<b>Tuning of Gain-Scheduled Three-Loop Autopilot</b> .....	<b>7-24</b>

## **Index**

---

# Introduction

---

- “Robust Control Toolbox Product Description” on page 1-2
- “Product Requirements” on page 1-3
- “Modeling Uncertainty” on page 1-4
- “System with Uncertain Parameters” on page 1-5
- “Worst-Case Performance” on page 1-9
- “Worst-Case Performance of Uncertain System” on page 1-10
- “Loop-Shaping Controller Design” on page 1-12
- “Model Reduction and Approximation” on page 1-16
- “Reduce Order of Synthesized Controller” on page 1-17
- “LMI Solvers” on page 1-21
- “Extends Control System Toolbox Capabilities” on page 1-22
- “Acknowledgments” on page 1-23
- “Bibliography” on page 1-25

# Robust Control Toolbox Product Description

## Design robust controllers for uncertain plants

Robust Control Toolbox™ provides functions, algorithms, and blocks for analyzing and tuning control systems for performance and robustness. You can create uncertain models by combining nominal dynamics with uncertain elements, such as uncertain parameters or unmodeled dynamics. You can analyze the impact of plant model uncertainty on control system performance and identify worst-case combinations of uncertain elements. H-infinity and mu-synthesis techniques let you design controllers that maximize robust stability and performance.

The toolbox automatically tunes both SISO and MIMO controllers. These can include decentralized, fixed-structure controllers with multiple tunable blocks spanning multiple feedback loops. The toolbox lets you tune one controller against a set of plant models. You can also tune gain-scheduled controllers. You can specify multiple tuning objectives, such as reference tracking, disturbance rejection, stability margins, and closed-loop pole locations.

## Key Features

- Modeling of systems with uncertain parameters or neglected dynamics
- Worst-case analysis of stability margins and sensitivity to disturbances
- Automatic tuning of centralized, decentralized, and multiloop controllers
- Automatic tuning of gain-scheduled controllers
- Robustness analysis and controller tuning in Simulink®
- H-infinity and mu-synthesis algorithms
- General-purpose LMI solvers

## **Product Requirements**

Robust Control Toolbox software requires that you have installed Control System Toolbox™ software.

## Modeling Uncertainty

At the heart of robust control is the concept of an uncertain LTI system. Model uncertainty arises when system gains or other parameters are not precisely known, or can vary over a given range. Examples of real parameter uncertainties include uncertain pole and zero locations and uncertain gains. You can also have unstructured uncertainties, by which is meant complex parameter variations satisfying given magnitude bounds.

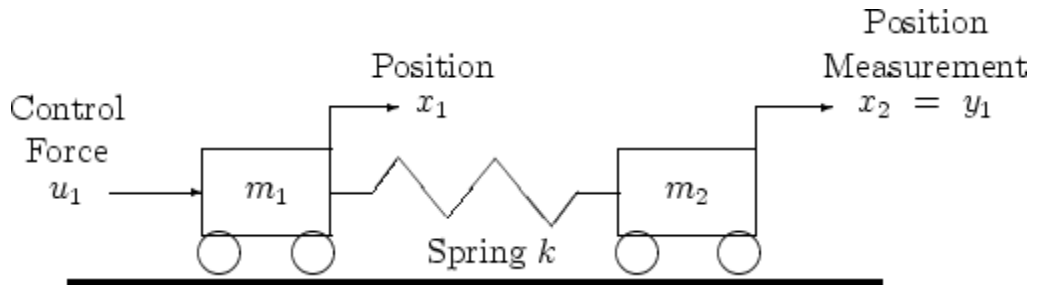
With Robust Control Toolbox software you can create uncertain LTI models as MATLAB® objects specifically designed for robust control applications. You can build models of complex systems by combining models of subsystems using addition, multiplication, and division, as well as with Control System Toolbox commands like `feedback` and `lft`.

For information about LTI model types, see “Linear System Representation”.



## System with Uncertain Parameters

For instance, consider the two-cart "ACC Benchmark" system [13] consisting of two frictionless carts connected by a spring shown as follows.



### ACC Benchmark Problem

The system has the block diagram model shown below, where the individual carts have the respective transfer functions.

$$G_1(s) = \frac{1}{m_1 s^2}$$

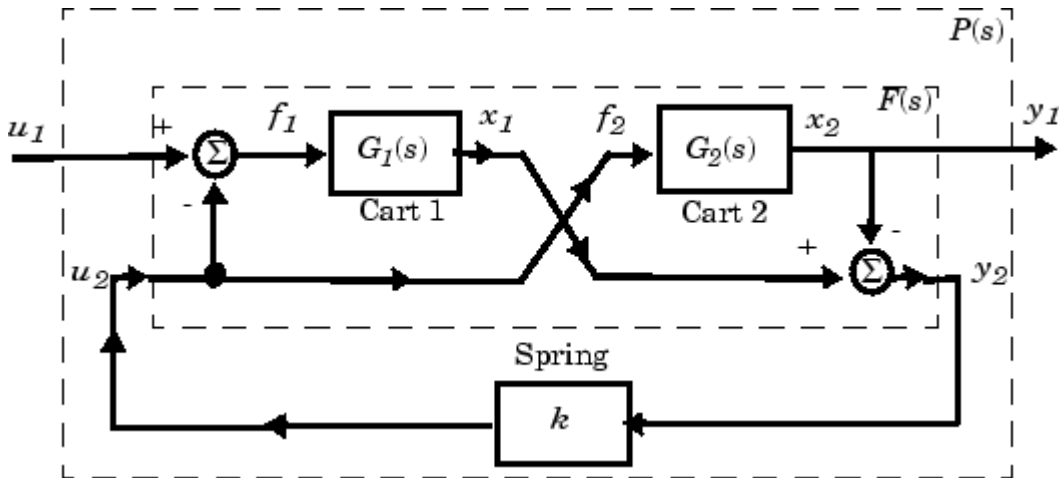
$$G_2(s) = \frac{1}{m_2 s^2}.$$

The parameters  $m_1$ ,  $m_2$ , and  $k$  are uncertain, equal to one plus or minus 20%:

$$m_1 = 1 \pm 0.2$$

$$m_2 = 1 \pm 0.2$$

$$k = 1 \pm 0.2$$



**"ACC Benchmark" Two-Cart System Block Diagram  $y_1 = P(s) u_1$**

The upper dashed-line block has transfer function matrix  $F(s)$ :

$$F(s) = \begin{bmatrix} 0 \\ G_1(s) \end{bmatrix} [1 \quad -1] + \begin{bmatrix} 1 \\ -1 \end{bmatrix} [0 \quad G_2(s)].$$

This code builds the uncertain system model P shown above:

```
% Create the uncertain real parameters m1, m2, & k
m1 = ureal('m1',1,'percent',20);
m2 = ureal('m2',1,'percent',20);
k = ureal('k',1,'percent',20);

s = zpk('s'); % create the Laplace variable s
G1 = ss(1/s^2)/m1; % Cart 1
G2 = ss(1/s^2)/m2; % Cart 2

% Now build F and P
F = [0;G1]*[1 -1]+[1;-1]*[0,G2];
P = lft(F,k) % close the loop with the spring k
```

The variable `P` is a SISO uncertain state-space (USS) object with four states and three uncertain parameters, `m1`, `m2`, and `k`. You can recover the nominal plant with the command

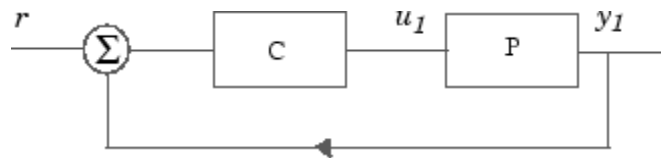
```
zpk(P.nominal)
```

which returns

```
Zero/pole/gain:
      1
-----
s^2 (s^2 + 2)
```

If the uncertain model `P(s)` has LTI negative feedback controller

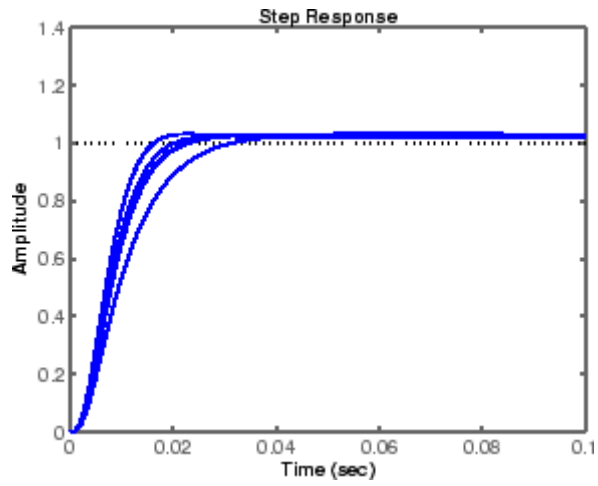
$$C(s) = \frac{100(s+1)^3}{(0.001s+1)^3}$$



then you can form the controller and the closed-loop system  $y_1 = T(s) u_1$  and view the closed-loop system's step response on the time interval from  $t=0$  to  $t=0.1$  for a Monte Carlo random sample of five combinations of the three uncertain parameters `k`, `m1`, and `m2` using this code:

```
C=100*ss((s+1)/(.001*s+1))^3 % LTI controller
T=feedback(P*C,1); % closed-loop uncertain system
step(usample(T,5),.1);
```

The resulting plot is shown below.



**Monte Carlo Sampling of Uncertain System's Step Response**

## Worst-Case Performance

To be robust, your control system should meet your stability and performance requirements for all possible values of uncertain parameters. Monte Carlo parameter sampling via `usample` can be used for this purpose as shown in Monte Carlo Sampling of Uncertain System's Step Response on page 1-8, but Monte Carlo methods are inherently hit or miss. With Monte Carlo methods, you might need to take an impossibly large number of samples before you hit upon or near a worst-case parameter combination.

Robust Control Toolbox software gives you a powerful assortment of *robustness analysis* commands that let you directly calculate upper and lower bounds on worst-case performance without random sampling.

<b>Worst-Case Robustness Analysis Commands</b>	
<code>loopmargin</code>	Comprehensive analysis of feedback loop
<code>loopsens</code>	Sensitivity functions of feedback loop
<code>ncfmargin</code>	Normalized coprime stability margin of feedback loop
<code>robustperf</code>	Robust performance of uncertain systems
<code>robuststab</code>	Stability margins of uncertain systems
<code>wcgain</code>	Worst-case gain of an uncertain system
<code>wcmargin</code>	Worst-case gain/phase margins for feedback loop
<code>wcsens</code>	Worst-case sensitivity functions of feedback loop

## Worst-Case Performance of Uncertain System

Returning to the “System with Uncertain Parameters” on page 1-5, the closed loop system is:

```
T=feedback(P*C,1); % Closed-loop uncertain system
```

This uncertain state-space model  $T$  has three uncertain parameters,  $k$ ,  $m1$ , and  $m2$ , each equal to  $1\pm 20\%$  uncertain variation. To analyze whether the closed-loop system  $T$  is robustly stable for all combinations of values for these three parameters, you can execute the commands:

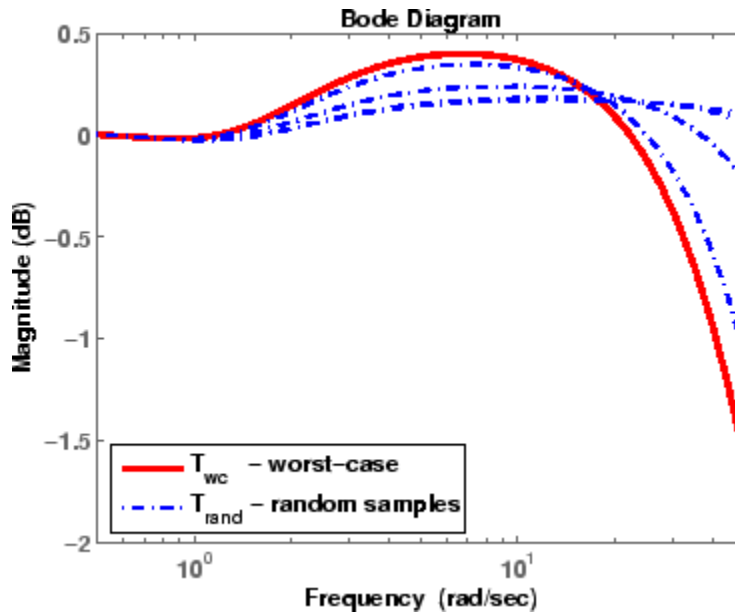
```
[StabilityMargin,Udestab,REPORT] = robuststab(T);  
REPORT
```

This displays the REPORT:

```
Uncertain System is robustly stable to modeled uncertainty.  
-- It can tolerate up to 311% of modeled uncertainty.  
-- A destabilizing combination of 500% the modeled uncertainty exists,  
    causing an instability at 44.3 rad/s.
```

The report tells you that the control system is robust for all parameter variations in the  $\pm 20\%$  range, and that the smallest destabilizing combination of real variations in the values  $k$ ,  $m1$ , and  $m2$  has sizes somewhere between 311% and 500% greater than  $\pm 20\%$ , i.e., between  $\pm 62.2\%$  and  $\pm 100\%$ . The value  $Udestab$  returns an estimate of the 500% destabilizing parameter variation combination:

```
Udestab =  
    k: 1.2174e-005  
   m1: 1.2174e-005  
   m2: 2.0000.
```



### Uncertain System Closed-Loop Bode Plots

You have a comfortable safety margin of between 311% to 500% larger than the anticipated  $\pm 20\%$  parameter variations before the closed loop goes unstable. But how much can closed-loop performance deteriorate for parameter variations constrained to lie strictly within the anticipated  $\pm 20\%$  range? The following code computes worst-case peak gain of  $T$ , and estimates the frequency and parameter values at which the peak gain occurs:

```
[PeakGain,Uwc] = wcgain(T);
Twc=usubs(T,Uwc);
% Worst case closed-loop system T
Trand=usample(T,4);
% 4 random samples of uncertain system T
bodemag(Twc,'r',Trand,'b-.',{.5,50}); % Do bode plot
legend('T_{wc} - worst-case',...
'T_{rand} - random samples',3);
```

The resulting plot is shown in Uncertain System Closed-Loop Bode Plots on page 1-11.

## Loop-Shaping Controller Design

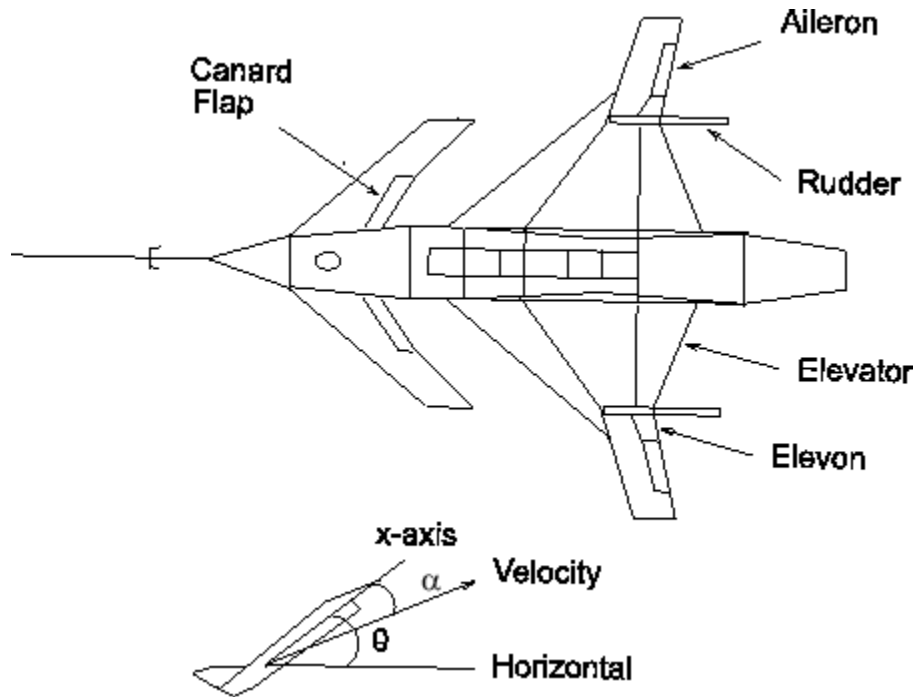
One of the most powerful yet simple controller synthesis tools is `loopsyn`. Given an LTI plant, you specify the shape of the open-loop systems frequency response plot that you want, then `loopsyn` computes a stabilizing controller that best approximates your specified loop shape.

For example, consider the 2-by-2 NASA HiMAT aircraft model (Safonov, Laub, and Hartmann [8]) depicted in the following figure. The control variables are elevon and canard actuators ( $\delta_e$  and  $\delta_c$ ). The output variables are angle of attack ( $\alpha$ ) and attitude angle ( $\theta$ ). The model has six states:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} \dot{\alpha} \\ \alpha \\ \dot{\theta} \\ \theta \\ x_e \\ x_\delta \end{bmatrix}$$

where  $x_e$  and  $x_\delta$  are elevon and canard actuator states.





### Aircraft Configuration and Vertical Plane Geometry

You can enter the state-space matrices for this model with the following code:

```
% NASA HiMAT model G(s)
ag = [ -2.2567e-02  -3.6617e+01  -1.8897e+01  -3.2090e+01  3.2509e+00  -7.6257e-01;
        9.2572e-05  -1.8997e+00  9.8312e-01  -7.2562e-04  -1.7080e-01  -4.9652e-03;
        1.2338e-02  1.1720e+01  -2.6316e+00  8.7582e-04  -3.1604e+01  2.2396e+01;
        0           0  1.0000e+00           0           0           0;
        0           0           0           0  -3.0000e+01           0;
        0           0           0           0           0  -3.0000e+01];

bg = [ 0  0;
       0  0;
       0  0;
       0  0;
       30 0;
       0 30];

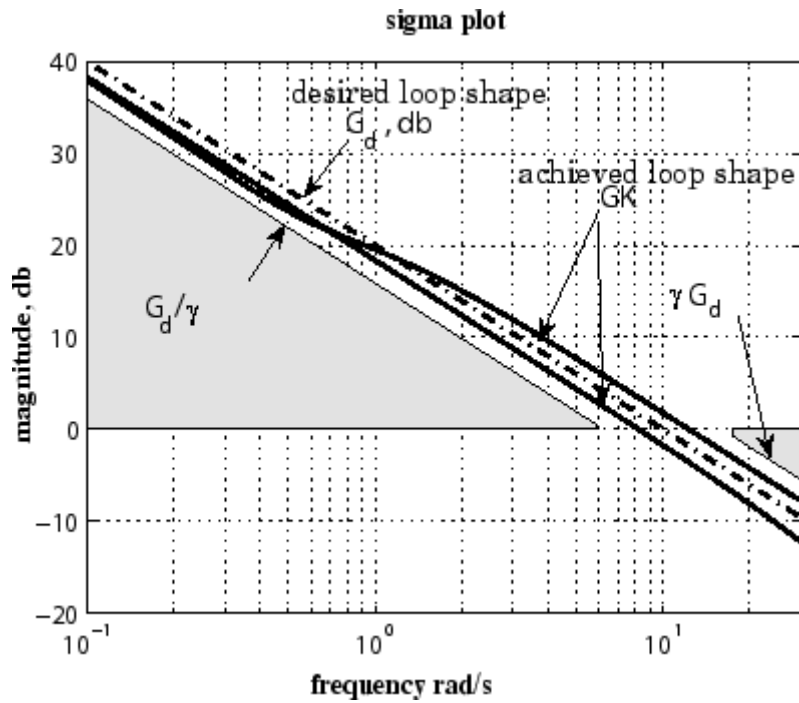
cg = [ 0  1  0  0  0  0];
```

```
        0    0    0    1    0    0];  
dg = [ 0    0;  
      0    0];  
G=ss(ag,bg,cg,dg);
```

To design a controller to shape the frequency response (`sigma`) plot so that the system has approximately a bandwidth of 10 rad/s, you can set as your target desired loop shape  $G_d(s)=10/s$ , then use `loopsyn(G,Gd)` to find a loop-shaping controller for `G` that optimally matches the desired loop shape `Gd` by typing:

```
s=zpk('s'); w0=10; Gd=w0/(s+.001);  
[K,CL,GAM]=loopsyn(G,Gd); % Design a loop-shaping controller K  
  
% Plot the results  
sigma(G*K,'r',Gd,'k-.',Gd/GAM,'k:',Gd*GAM,'k:',{.1,30})  
figure ;T=feedback(G*K,eye(2));  
sigma(T,ss(GAM),'k:',{.1,30});grid
```

The value of  $\gamma = \text{GAM}$  returned is an indicator of the accuracy to which the optimal loop shape matches your desired loop shape and is an upper bound on the resonant peak magnitude of the closed-loop transfer function `T=feedback(G*K,eye(2))`. In this case,  $\gamma = 1.6024 = 4 \text{ dB}$  — see the next figure.



### **MIMO Robust Loop Shaping with `loopsyn(G,Gd)`**

The achieved loop shape matches the desired target  $G_d$  to within about  $\gamma$  dB.

## Model Reduction and Approximation

Complex models are not always required for good control. Unfortunately, however, optimization methods (including methods based on  $H_\infty$ ,  $H_2$ , and  $\mu$ -synthesis optimal control theory) generally tend to produce controllers with at least as many states as the plant model. For this reason, Robust Control Toolbox software offers you an assortment of model-order reduction commands that help you to find less complex low-order approximations to plant and controller models.

Model Reduction Commands	
<code>reduce</code>	Main interface to model approximation algorithms
<code>balancmr</code>	Balanced truncation model reduction
<code>bstmr</code>	Balanced stochastic truncation model reduction
<code>hankelmr</code>	Optimal Hankel norm model approximations
<code>modreal</code>	State-space modal truncation/realization
<code>ncfmr</code>	Balanced normalized coprime factor model reduction
<code>schurmr</code>	Schur balanced truncation model reduction
<code>slowfast</code>	State-space slow-fast decomposition
<code>stabsep</code>	State-space stable/antistable decomposition
<code>imp2ss</code>	Impulse response to state-space approximation

Among the most important types of model reduction methods are minimize bounds methods on additive, multiplicative, and normalized coprime factor (NCF) model error. You can access all three of these methods using the command `reduce`.

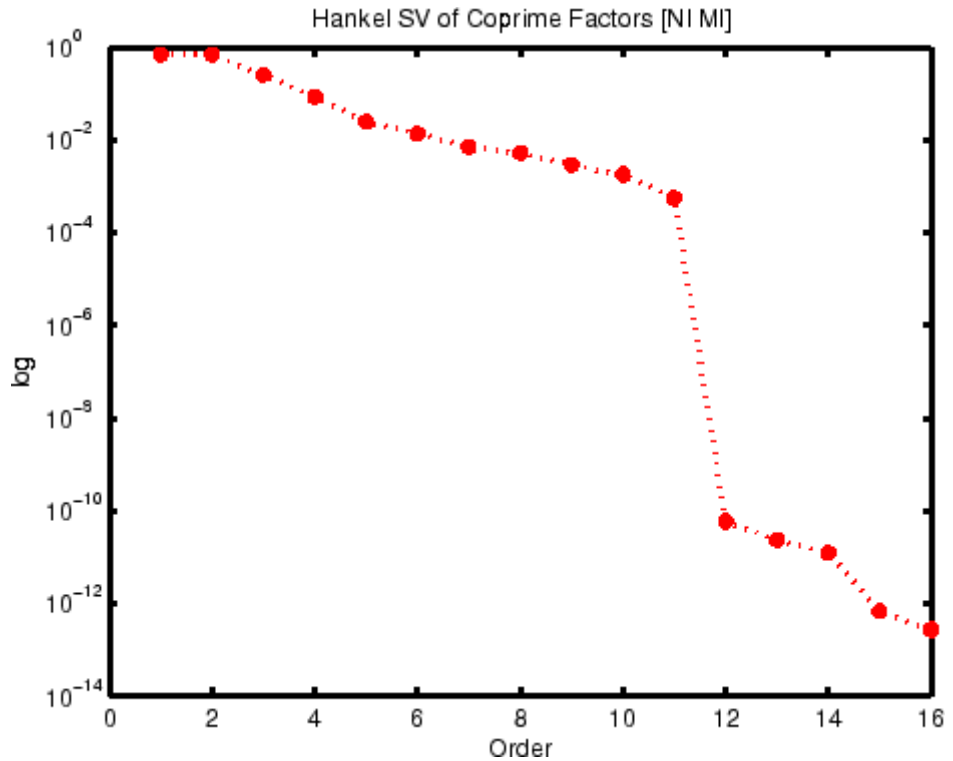
## Reduce Order of Synthesized Controller

For instance, the NASA HiMAT model considered in “Loop-Shaping Controller Design” on page 1-12 has eight states, and the optimal loop-shaping controller turns out to have 16 states. Using model reduction, you can remove at least some of the states without appreciably affecting stability or closed-loop performance. For controller order reduction, the NCF model reduction is particularly useful, and it works equally well with controllers that have poles anywhere in the complex plane.

For the NASA HiMAT design in the last section, you can type

```
hanke1sv(K, 'ncf', 'log');
```

which displays a logarithmic plot of the NCF Hankel singular values — see the following figure.

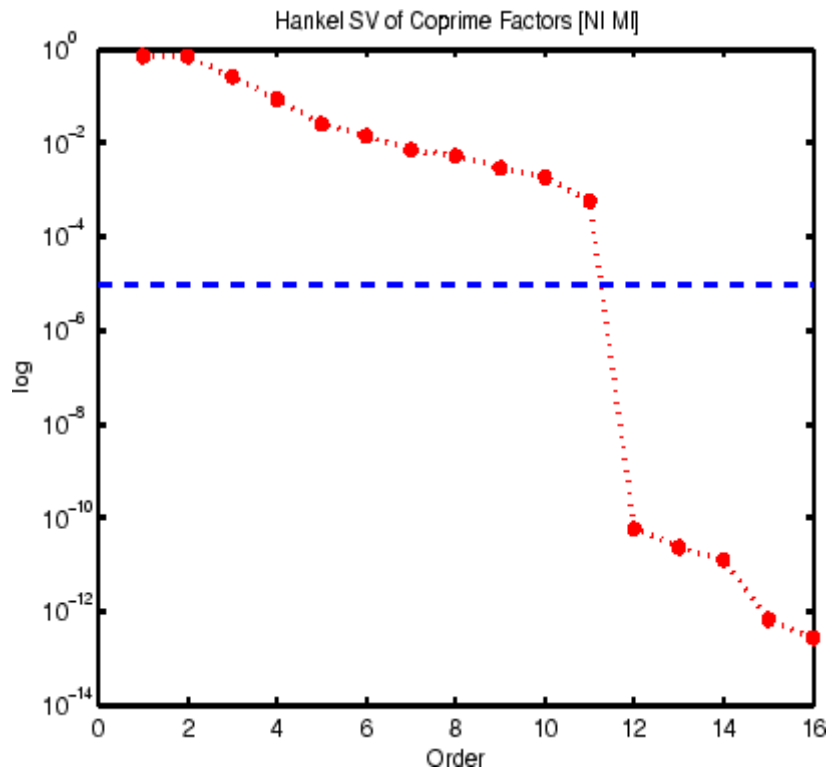


### Hankel Singular Values of Coprime Factorization of K

Theory says that, without danger of inducing instability, you can confidently discard at least those controller states that have NCF Hankel singular values that are much smaller than  $\text{ncfmargin}(G,K)$ .

Compute  $\text{ncfmargin}(G,K)$  and add it to your Hankel singular values plot.

```
hankelsv(K,'ncf','log');v=axis;
hold on; plot(v(1:2), ncfmargin(G,K)*[1 1], '--'); hold off
```



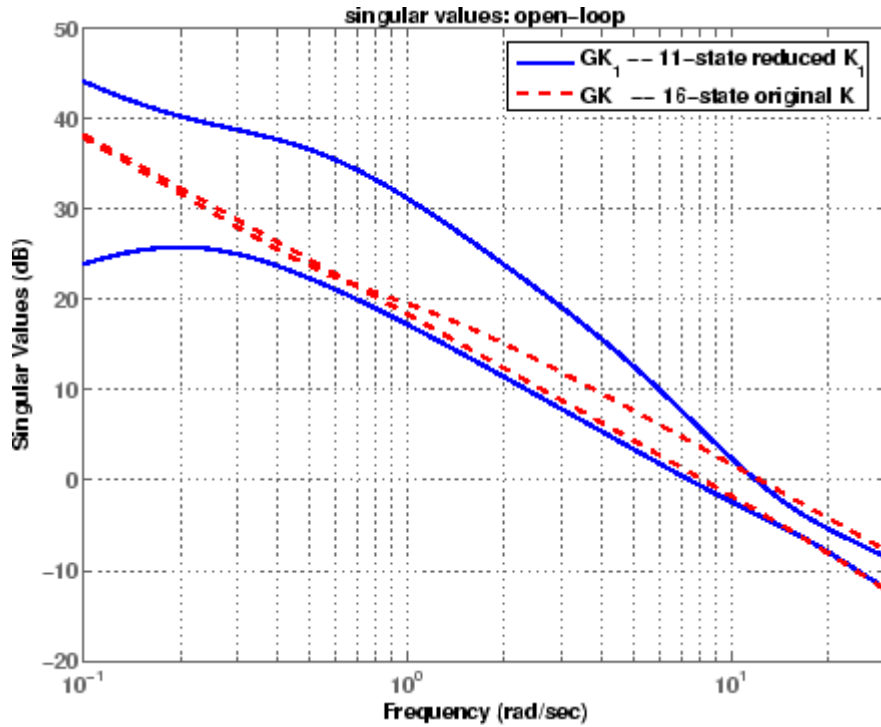
**Five of the 16 NCF Hankel Singular Values of HiMAT Controller K Are Small Compared to  $\text{ncfmargin}(G,K)$**

In this case, you can safely discard 5 of the 16 states of K and compute an 11-state reduced controller by typing:

```
K1=reduce(K,11,'errortype','ncf');
```

The result is plotted in the following figure.

```
sigma(G*K1,'b',G*K,'r--',{.1,30});
```



### HiMAT with 11-State Controller $K_1$ vs. Original 16-State Controller $K$

The picture above shows that low-frequency gain is decreased considerably for inputs in one vector direction. Although this does not affect stability, it affects performance. If you wanted to better preserve low-frequency performance, you would discard fewer than five of the 16 states of  $K$ .



## LMI Solvers

At the core of many emergent robust control analysis and synthesis routines are powerful general-purpose functions for solving a class of convex nonlinear programming problems known as linear matrix inequalities. The LMI capabilities are invoked by Robust Control Toolbox software functions that evaluate worst-case performance, as well as functions like `hinf` and `h2hinf`. Some of the main functions that help you access the LMI capabilities of the toolbox are shown in the following table.

<b>Specification of LMIs</b>	
<code>lmiedit</code>	GUI for LMI specification
<code>setlmis</code>	Initialize the LMI description
<code>lmivar</code>	Define a new matrix variable
<code>lmiterm</code>	Specify the term content of an LMI
<code>newlmi</code>	Attach an identifying tag to new LMIs
<code>getlmis</code>	Get the internal description of the LMI system
<b>LMI Solvers</b>	
<code>feasp</code>	Test feasibility of a system of LMIs
<code>gevp</code>	Minimize generalized eigenvalue with LMI constraints
<code>mincx</code>	Minimize a linear objective with LMI constraints
<code>dec2mat</code>	Convert output of the solvers to values of matrix variables
<b>Evaluation of LMIs/Validation of Results</b>	
<code>evallmi</code>	Evaluate for given values of the decision variables
<code>showlmi</code>	Return the left and right sides of an evaluated LMI

## Extends Control System Toolbox Capabilities

Robust Control Toolbox software is designed to work with Control System Toolbox software. Robust Control Toolbox software extends the capabilities of Control System Toolbox software and leverages the LTI and plotting capabilities of Control System Toolbox software. The major analysis and synthesis commands in Robust Control Toolbox software accept LTI object inputs, e.g., LTI state-space systems produced by commands such as:

```
G=tf(1,[1 2 3])  
G=ss([-1 0; 0 -1], [1;1],[1 1],3)
```

The uncertain system (USS) objects in Robust Control Toolbox software generalize the Control System Toolbox LTI SS objects and help ease the task of analyzing and plotting uncertain systems. You can do many of the same algebraic operations on uncertain systems that are possible for LTI objects (multiply, add, invert), and Robust Control Toolbox software provides USS uncertain system extensions of Control System Toolbox software interconnection and plotting functions like `feedback`, `lft`, and `bode`.

## Acknowledgments

Professor **Andy Packard** is with the Faculty of Mechanical Engineering at the University of California, Berkeley. His research interests include robustness issues in control analysis and design, linear algebra and numerical algorithms in control problems, applications of system theory to aerospace problems, flight control, and control of fluid flow.

Professor **Gary Balas** is with the Faculty of Aerospace Engineering & Mechanics at the University of Minnesota and is president of MUSYN Inc. His research interests include aerospace control systems, both experimental and theoretical.

Dr. **Michael Safonov** is with the Faculty of Electrical Engineering at the University of Southern California. His research interests include control and decision theory.

Dr. **Richard Chiang** is employed by Boeing Satellite Systems, El Segundo, CA. He is a Boeing Technical Fellow and has been working in the aerospace industry over 25 years. In his career, Richard has designed 3 flight control laws, 12 spacecraft attitude control laws, and 3 large space structure vibration controllers, using modern robust control theory and the tools he built in this toolbox. His research interests include robust control theory, model reduction, and in-flight system identification. Working in industry instead of academia, Richard serves a unique role in our team, bridging the gap between theory and reality.

The linear matrix inequality (LMI) portion of Robust Control Toolbox software was developed by these two authors:

Dr. **Pascal Gahinet** is employed by MathWorks. His research interests include robust control theory, linear matrix inequalities, numerical linear algebra, and numerical software for control.

Professor **Arkadi Nemirovski** is with the Faculty of Industrial Engineering and Management at Technion, Haifa, Israel. His research interests include convex optimization, complexity theory, and nonparametric statistics.

The structured  $H_\infty$  synthesis (`hinfstruct`) portion of Robust Control Toolbox software was developed by the following author in collaboration with Pascal Gahinet:

Professor **Pierre Apkarian** is with ONERA (The French Aerospace Lab) and the Institut de Mathématiques at Paul Sabatier University, Toulouse, France. His research interests include robust control, LMIs, mathematical programming, and nonsmooth optimization techniques for control.

## Bibliography

- [1] Boyd, S.P., El Ghaoui, L., Feron, E., and Balakrishnan, V., *Linear Matrix Inequalities in Systems and Control Theory*, Philadelphia, PA, SIAM, 1994.
- [2] Dorato, P. (editor), *Robust Control*, New York, IEEE Press, 1987.
- [3] Dorato, P., and Yedavalli, R.K. (editors), *Recent Advances in Robust Control*, New York, IEEE Press, 1990.
- [4] Doyle, J.C., and Stein, G., "Multivariable Feedback Design: Concepts for a Classical/Modern Synthesis," *IEEE Trans. on Automat. Contr.*, 1981, AC-26(1), pp. 4-16.
- [5] El Ghaoui, L., and Niculescu, S., *Recent Advances in LMI Theory for Control*, Philadelphia, PA, SIAM, 2000.
- [6] Lehtomaki, N.A., Sandell, Jr., N.R., and Athans, M., "Robustness Results in Linear-Quadratic Gaussian Based Multivariable Control Designs," *IEEE Trans. on Automat. Contr.*, Vol. AC-26, No. 1, Feb. 1981, pp. 75-92.
- [7] Safonov, M.G., *Stability and Robustness of Multivariable Feedback Systems*, Cambridge, MA, MIT Press, 1980.
- [8] Safonov, M.G., Laub, A.J., and Hartmann, G., "Feedback Properties of Multivariable Systems: The Role and Use of Return Difference Matrix," *IEEE Trans. of Automat. Contr.*, 1981, AC-26(1), pp. 47-65.
- [9] Safonov, M.G., Chiang, R.Y., and Flashner, H., " $H_\infty$  Control Synthesis for a Large Space Structure," *Proc. of American Contr. Conf.*, Atlanta, GA, June 15-17, 1988.
- [10] Safonov, M.G., and Chiang, R.Y., "CACSD Using the State-Space  $L_\infty$  Theory — A Design Example," *IEEE Trans. on Automatic Control*, 1988, AC-33(5), pp. 477-479.
- [11] Sanchez-Pena, R.S., and Sznaier, M., *Robust Systems Theory and Applications*, New York, Wiley, 1998.

[12] Skogestad, S., and Postlethwaite, I., *Multivariable Feedback Control*, New York, Wiley, 1996.

[13] Wie, B., and Bernstein, D.S., "A Benchmark Problem for Robust Controller Design," *Proc. American Control Conf.*, San Diego, CA, May 23-25, 1990; also Boston, MA, June 26-28, 1991.

[14] Zhou, K., Doyle, J.C., and Glover, K., *Robust and Optimal Control*, Englewood Cliffs, NJ, Prentice Hall, 1996.

# Multivariable Loop Shaping

---

- “Tradeoff Between Performance and Robustness” on page 2-2
- “Norms and Singular Values” on page 2-4
- “Typical Loop Shapes, S and T Design” on page 2-6
- “Using LOOPSYN to Do H-Infinity Loop Shaping” on page 2-15
- “Loop-Shaping Control Design of Aircraft Model” on page 2-16
- “Fine-Tuning the LOOPSYN Target Loop Shape  $G_d$  to Meet Design Goals” on page 2-21
- “Mixed-Sensitivity Loop Shaping” on page 2-22
- “Mixed-Sensitivity Loop-Shaping Controller Design” on page 2-24
- “Loop-Shaping Commands” on page 2-26

## Tradeoff Between Performance and Robustness

When the plant modeling uncertainty is not too big, you can design high-gain, high-performance feedback controllers. High loop gains significantly larger than 1 in magnitude can attenuate the effects of plant model uncertainty and reduce the overall sensitivity of the system to plant noise. But if your plant model uncertainty is so large that you do not even know the sign of your plant gain, then you cannot use large feedback gains without the risk that the system will become unstable. Thus, plant model uncertainty can be a fundamental limiting factor in determining what can be achieved with feedback.

---

**Multiplicative Uncertainty:** Given an approximate model of the plant  $G_0$  of a plant  $G$ , the *multiplicative uncertainty*  $\Delta_M$  of the model  $G_0$  is defined

$$\text{as } \Delta_M = G_0^{-1}(G - G_0)$$

or, equivalently,

$$G = (I + \Delta_M)G_0.$$

---

Plant model uncertainty arises from many sources. There might be small unmodeled time delays or stray electrical capacitance. Imprecisely understood actuator time constants or, in mechanical systems, high-frequency torsional bending modes and similar effects can be responsible for plant model uncertainty. These types of uncertainty are relatively small at lower frequencies and typically increase at higher frequencies.

In the case of single-input/single-output (SISO) plants, the frequency at which there are uncertain variations in your plant of size  $|\Delta_M|=2$  marks a critical threshold beyond which there is insufficient information about the plant to reliably design a feedback controller. With such a 200% model uncertainty, the model provides no indication of the phase angle of the true plant, which means that the only way you can reliably stabilize your plant is to ensure that the loop gain is less than 1. Allowing for an additional factor of 2 margin for error, your control system bandwidth is essentially limited



to the frequency range over which your multiplicative plant uncertainty  $\Delta_M$  has gain magnitude  $|\Delta_M| < 1$ .

## Norms and Singular Values

For MIMO systems the transfer functions are matrices, and relevant measures of gain are determined by singular values,  $H_\infty$ , and  $H_2$  norms, which are defined as follows:

---

**$H_2$  and  $H_\infty$  Norms** The  $H_2$ -norm is the energy of the impulse response of plant  $G$ . The  $H_\infty$ -norm is the peak gain of  $G$  across all frequencies and all input directions.

---

Another important concept is the notion of singular values.

---

**Singular Values:** The *singular values* of a rank  $r$  matrix  $A \in C^{m \times n}$ , denoted  $\sigma_i$ , are the nonnegative square roots of the eigenvalues of  $A^*A$  ordered such that  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p > 0$ ,  $p \leq \min\{m, n\}$ .

If  $r < p$  then there are  $p - r$  zero singular values, i.e.,  $\sigma_{r+1} = \sigma_{r+2} = \dots = \sigma_p = 0$ .

The greatest singular value  $\sigma_1$  is sometimes denoted

$$\bar{\sigma}(A) = \sigma_1.$$

When  $A$  is a square  $n$ -by- $n$  matrix, then the  $n$ th singular value (i.e., the least singular value) is denoted

$$\bar{\sigma}(A) \square \sigma_n.$$


---

## Properties of Singular Values

Some useful properties of singular values are:

$$\bar{\sigma}(A) = \max_{x \in C^h} \frac{\|Ax\|}{\|x\|}$$

$$\underline{\sigma}(A) = \min_{x \in C^h} \frac{\|Ax\|}{\|x\|}$$

These properties are especially important because they establish that the greatest and least singular values of a matrix  $A$  are the maximal and minimal "gains" of the matrix as the input vector  $x$  varies over all possible directions.

For stable continuous-time LTI systems  $G(s)$ , the  $H_2$ -norm and the  $H_\infty$ -norms are defined terms of the frequency-dependent singular values of  $G(j\omega)$ :

$H_2$ -norm:

$$\|G\|_2 \square \left[ \frac{1}{2\pi} \right] \int_{-\infty}^{\infty} \sum_{i=1}^p (\sigma_i(G(j\omega)))^2 d\omega$$

$H_\infty$ -norm:

$$\|G\|_2 \square \sup_{\omega} \bar{\sigma}(G(j\omega))$$

where  $\sup$  denotes the least upper bound.

## Typical Loop Shapes, $S$ and $T$ Design

Consider the multivariable feedback control system shown in the following figure. In order to quantify the multivariable stability margins and performance of such systems, you can use the singular values of the closed-loop transfer function matrices from  $r$  to each of the three outputs  $e$ ,  $u$ , and  $y$ , *viz.*

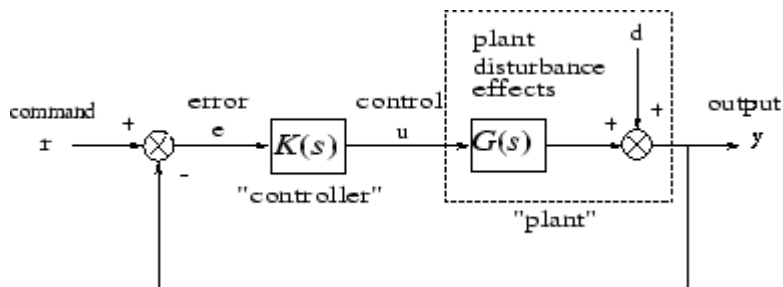
$$S(s) \stackrel{\text{def}}{=} (I + L(s))^{-1}$$

$$R(s) \stackrel{\text{def}}{=} K(s)(I + L(s))^{-1}$$

$$T(s) \stackrel{\text{def}}{=} L(s)(I + L(s))^{-1} = I - S(s)$$

where the  $L(s)$  is the loop transfer function matrix

$$L(s) = G(s)K(s). \quad (2-1)$$



**Block Diagram of the Multivariable Feedback Control System**

The two matrices  $S(s)$  and  $T(s)$  are known as the *sensitivity function* and *complementary sensitivity function*, respectively. The matrix  $R(s)$  has no common name. The singular value Bode plots of each of the three transfer function matrices  $S(s)$ ,  $R(s)$ , and  $T(s)$  play an important role in robust multivariable control system design. The singular values of the loop transfer function matrix  $L(s)$  are important because  $L(s)$  determines the matrices  $S(s)$  and  $T(s)$ .

## Robustness in Terms of Singular Values

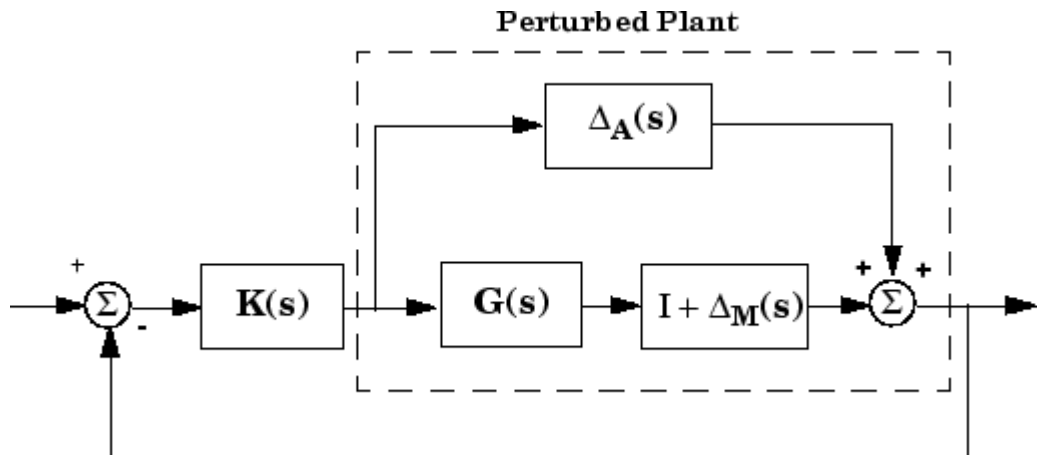
The singular values of  $S(j\omega)$  determine the disturbance attenuation, because  $S(s)$  is in fact the closed-loop transfer function from disturbance  $d$  to plant output  $y$  — see Block Diagram of the Multivariable Feedback Control System on page 2-6. Thus a disturbance attenuation performance specification can be written as

$$\bar{\sigma}(S(j\omega)) \leq |W_1^{-1}(j\omega)| \quad (2-2)$$

where  $|W_1^{-1}(j\omega)|$  is the desired disturbance attenuation factor. Allowing  $|W_1(j\omega)|$  to depend on frequency  $\omega$  enables you to specify a different attenuation factor for each frequency  $\omega$ .

The singular value Bode plots of  $R(s)$  and of  $T(s)$  are used to measure the stability margins of multivariable feedback designs in the face of additive plant perturbations  $\Delta_A$  and multiplicative plant perturbations  $\Delta_M$ , respectively. See the following figure.

Consider how the singular value Bode plot of complementary sensitivity  $T(s)$  determines the stability margin for multiplicative perturbations  $\Delta_M$ . The multiplicative stability margin is, by definition, the "size" of the smallest stable  $\Delta_M(s)$  that destabilizes the system in the figure below when  $\Delta_A = 0$ .



### Additive/Multiplicative Uncertainty

Taking  $\bar{\sigma}(\Delta_M(j\omega))$  to be the definition of the "size" of  $\Delta_M(j\omega)$ , you have the following useful characterization of "multiplicative" stability robustness:

---

**Multiplicative Robustness:** The size of the smallest destabilizing multiplicative uncertainty  $\Delta_M(s)$  is:

$$\bar{\sigma}(\Delta_M(j\omega)) = \frac{1}{\bar{\sigma}(T(j\omega))}.$$


---

The smaller is  $\bar{\sigma}(T(j\omega))$ , the greater will be the size of the smallest destabilizing multiplicative perturbation, and hence the greater will be the stability margin.

A similar result is available for relating the stability margin in the face of additive plant perturbations  $\Delta_A(s)$  to  $R(s)$  if you take  $\bar{\sigma}(\Delta_A(j\omega))$  to be the definition of the "size" of  $\Delta_A(j\omega)$  at frequency  $\omega$ .

**Additive Robustness:** The size of the smallest destabilizing additive uncertainty  $\Delta_A$  is:

$$\bar{\sigma}(\Delta_A(j\omega)) = \frac{1}{\bar{\sigma}(R(j\omega))}.$$

As a consequence of robustness theorems 1 and 2, it is common to specify the stability margins of control systems via singular value inequalities such as

$$\bar{\sigma}(R\{j\omega\}) \leq |W_2^{-1}(j\omega)| \quad (2-3)$$

$$\bar{\sigma}(T\{j\omega\}) \leq |W_3^{-1}(j\omega)| \quad (2-4)$$

where  $|W_2(j\omega)|$  and  $|W_3(j\omega)|$  are the respective sizes of the largest anticipated additive and multiplicative plant perturbations.

It is common practice to lump the effects of all plant uncertainty into a single fictitious multiplicative perturbation  $\Delta_M$ , so that the control design requirements can be written

$$\frac{1}{\sigma_i(S(j\omega))} \geq |W_1(j\omega)|; \quad \bar{\sigma}_i(T[j\omega]) \leq |W_3^{-1}(j\omega)|$$

as shown in Singular Value Specifications on L, S, and T on page 2-12.

It is interesting to note that in the upper half of the figure (above the 0 dB line),

$$\underline{\sigma}(L(j\omega)) \approx \frac{1}{\bar{\sigma}(S(j\omega))}$$

while in the lower half of Singular Value Specifications on L, S, and T on page 2-12 (below the 0 dB line),

$$\underline{\sigma}(L(j\omega)) \approx \bar{\sigma}(T(j\omega)).$$

This results from the fact that

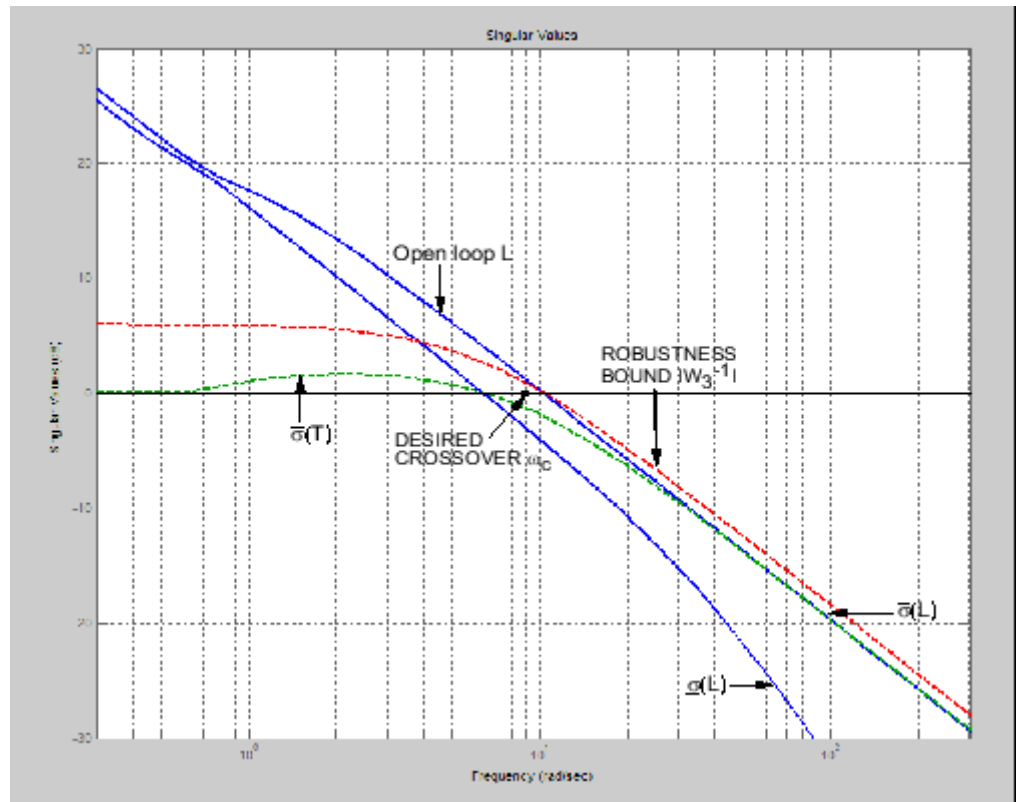
$$S(s) \stackrel{\text{def}}{=} (I + L(s))^{-1} \approx L(s)^{-1}$$

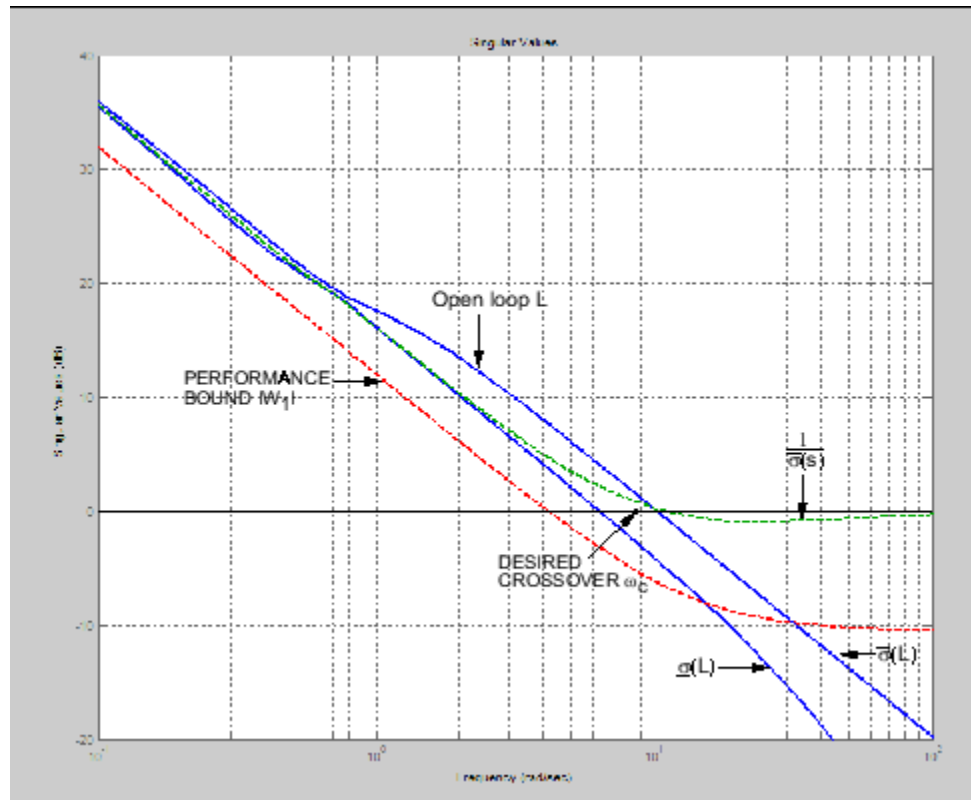
if  $\underline{\sigma}(L(s)) \gg 1$ , and

$$T(s) \stackrel{\text{def}}{=} L(s)(I + L(s))^{-1} \approx L(s)$$

if  $\bar{\sigma}(L(s)) \gg 1$ .







### Singular Value Specifications on L, S, and T

Thus, it is not uncommon to see specifications on disturbance attenuation and multiplicative stability margin expressed directly in terms of forbidden regions for the Bode plots of  $\sigma_i(L(j\omega))$  as "singular value loop shaping" requirements, either as specified upper/lower bounds or as a target desired loop shape — see the preceding figure.

### Guaranteed Gain/Phase Margins in MIMO Systems

For those who are more comfortable with classical single-loop concepts, there are the important connections between the multiplicative stability margins predicted by  $\bar{\sigma}(T)$  and those predicted by classical  $M$ -circles, as found on the Nichols chart. Indeed in the single-input/single-output case,

$$\bar{\sigma}(T(j\omega)) = \left| \frac{L(j\omega)}{1+L(j\omega)} \right|$$

which is precisely the quantity you obtain from Nichols chart  $M$ -circles. Thus,

$\|T\|_\infty$  is a multiloop generalization of the closed-loop resonant peak magnitude which, as classical control experts will recognize, is closely related to the damping ratio of the dominant closed-loop poles. Also, it turns out that you

can relate  $\|T\|_\infty$ ,  $\|S\|_\infty$  to the classical gain margin  $G_M$  and phase margin  $\theta_M$  in each feedback loop of the multivariable feedback system of Block Diagram of the Multivariable Feedback Control System on page 2-6 via the formulas:

$$G_M \geq 1 + \frac{1}{\|T\|_\infty}$$

$$G_M \geq 1 + \frac{1}{1 - \frac{1}{\|S\|_\infty}}$$

$$\theta_M \geq 2 \sin^{-1} \left( \frac{1}{2\|T\|_\infty} \right)$$

$$\theta_M \geq 2 \sin^{-1} \left( \frac{1}{2\|T\|_\infty} \right).$$

(See [6].) These formulas are valid provided  $\|S\|_\infty$  and  $\|T\|_\infty$  are larger than 1, as is normally the case. The margins apply even when the gain perturbations or phase perturbations occur simultaneously in several feedback channels.

The infinity norms of  $S$  and  $T$  also yield gain reduction tolerances. The *gain reduction tolerance*  $g_m$  is defined to be the minimal amount by which the gains in each loop would have to be *decreased* in order to destabilize the system.

Upper bounds on  $g_m$  are as follows:

$$g_M \leq 1 - \frac{1}{\|T\|_\infty}$$
$$g_M \leq \frac{1}{1 + \frac{1}{\|S\|_\infty}}.$$

## Using LOOPSYN to Do H-Infinity Loop Shaping

The command `loopsyn` lets you design a stabilizing feedback controller to optimally shape the open loop frequency response of a MIMO feedback control system to match as closely as possible a desired loop shape  $G_d$  — see the preceding figure. The basic syntax of the `loopsyn` loop-shaping controller synthesis command is:

```
K = loopsyn(G,Gd)
```

Here  $G$  is the LTI transfer function matrix of a MIMO plant model,  $G_d$  is the target desired loop shape for the loop transfer function  $L=G*K$ , and  $K$  is the optimal loop-shaping controller. The LTI controller  $K$  has the property that it shapes the loop  $L=G*K$  so that it matches the frequency response of  $G_d$  as closely as possible, subject to the constraint that the compensator must stabilize the plant model  $G$ .

## Loop-Shaping Control Design of Aircraft Model

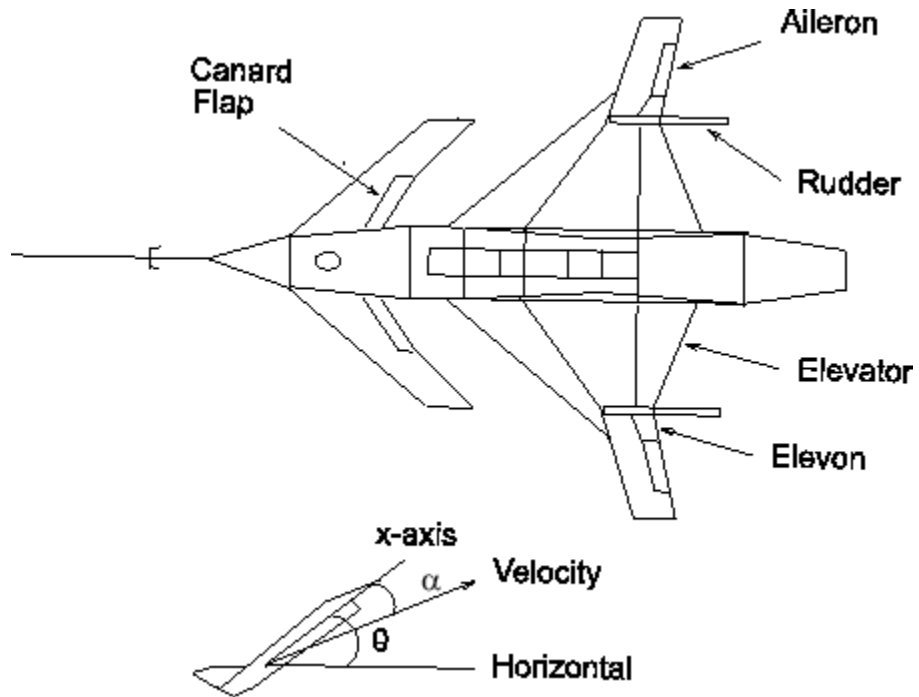
To see how the `loopsyn` command works in practice to address robustness and performance tradeoffs, consider again the NASA HiMAT aircraft model taken from the paper of Safonov, Laub, and Hartmann [8]. The longitudinal dynamics of the HiMAT aircraft trimmed at 25000 ft and 0.9 Mach are unstable and have two right-half-plane phugoid modes. The linear model has state-space realization  $G(s) = C(Is - A)^{-1}B$  with six states, with the first four states representing angle of attack ( $\alpha$ ) and attitude angle ( $\theta$ ) and their rates of change, and the last two representing elevon and canard control actuator dynamics — see Aircraft Configuration and Vertical Plane Geometry on page 2-17.

```

ag = [
-2.2567e-02  -3.6617e+01  -1.8897e+01  -3.2090e+01  3.2509e+00  -7.6257e-01;
9.2572e-05  -1.8997e+00  9.8312e-01  -7.2562e-04  -1.7080e-01  -4.9652e-03;
1.2338e-02  1.1720e+01  -2.6316e+00  8.7582e-04  -3.1604e+01  2.2396e+01;
0           0           1.0000e+00           0           0           0;
0           0           0           0           -3.0000e+01  0;
0           0           0           0           0           -3.0000e+01];
bg = [0  0;
      0  0;
      0  0;
      0  0;
      30 0;
      0 30];
cg = [0  1  0  0  0  0;
      0  0  0  1  0  0];
dg = [0  0;
      0  0];
G=ss(ag,bg,cg,dg);

```

The control variables are elevon and canard actuators ( $\delta_e$  and  $\delta_c$ ). The output variables are angle of attack ( $\alpha$ ) and attitude angle ( $\theta$ ).



### Aircraft Configuration and Vertical Plane Geometry

This model is good at frequencies below 100 rad/s with less than 30% variation between the true aircraft and the model in this frequency range. However as noted in [8], it does not reliably capture very high-frequency behaviors, because it was derived by treating the aircraft as a rigid body and neglecting lightly damped fuselage bending modes that occur at somewhere between 100 and 300 rad/s. These unmodeled bending modes might cause as much as 20 dB deviation (i.e., 1000%) between the frequency response of the model and the actual aircraft for frequency  $\omega > 100$  rad/s. Other effects like control actuator time delays and fuel sloshing also contribute to model inaccuracy at even higher frequencies, but the dominant unmodeled effects are the fuselage bending modes. You can think of these unmodeled bending modes as multiplicative uncertainty of size 20 dB, and design your controller using loopsyn, by making sure that the loop has gain less than  $-20$  dB at, and beyond, the frequency  $\omega > 100$  rad/s.

### Design Specifications

The singular value design specifications are

- **Robustness Spec.:**  $-20$  dB/decade roll-off slope and  $-20$  dB loop gain at  $100$  rad/s
- **Performance Spec.:** Minimize the sensitivity function as much as possible.

Both specs can be accommodated by taking as the desired loop shape

$$G_d(s)=8/s$$

### MATLAB Commands for a LOOPSYN Design

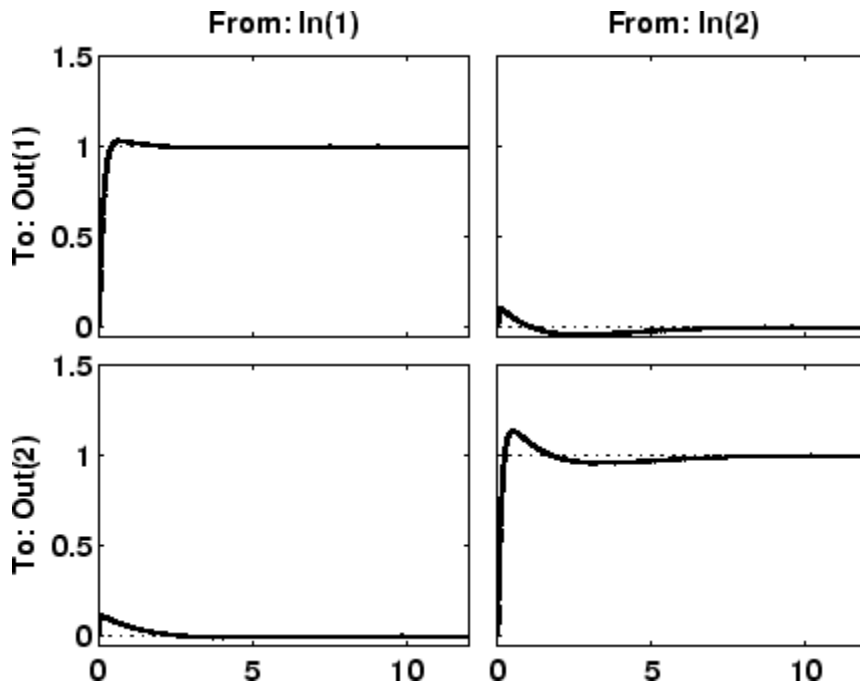
```
%% Enter the desired loop shape Gd
s=zpk('s'); % Laplace variable s
Gd=8/s; % desired loop shape
%% Compute the optimal loop shaping controller K
[K,CL,GAM]=loopsyn(G,Gd);
%% Compute the loop L, sensitivity S and
%% complementary sensitivity T:
L=G*K;
I=eye(size(L));
S=feedback(I,L); % S=inv(I+L);
T=I-S;
%% Plot the results:
% step response plots
step(T);title('\alpha and \theta command step responses');
% frequency response plots
figure;
sigma(I+L,'--',T,':',L,'r--',Gd,'k-',Gd/GAM,'k:',...
    Gd*GAM,'k',{.1,100});grid
legend('1/\sigma(S) performance',...
    '\sigma(T) robustness',...
    '\sigma(L) loopshape',...
    '\sigma(Gd) desired loop',...
    '\sigma(Gd) \pm GAM, dB');
```



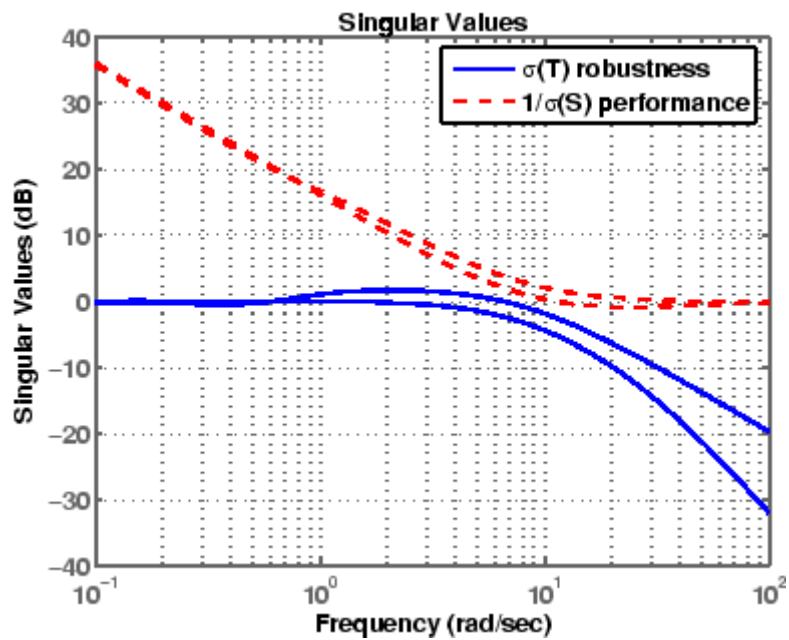
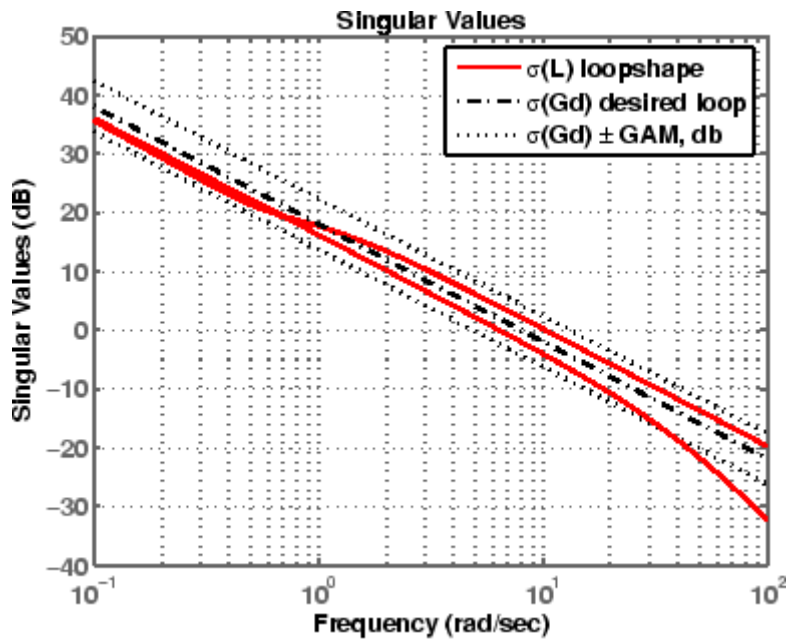
The plots of the resulting step and frequency response for the NASA HiMAT loopsyn loop-shaping controller design are shown in the following figure. The number  $\pm \text{GAM}$ , dB (i.e.,  $20\log_{10}(\text{GAM})$ ) tells you the accuracy with which your loopsyn control design matches the target desired loop:

$$\bar{\sigma}(GK), \text{db} \geq |G_d|, \text{db} - \text{GAM}, \text{db} \quad (\omega < \omega_c)$$

$$\bar{\sigma}(GK), \text{db} \geq |G_d|, \text{db} + \text{GAM}, \text{db} \quad (\omega > \omega_c).$$



HiMAT Closed Loop Step Responses



LOOPSYN Design Results for NASA HiMAT

## Fine-Tuning the LOOPSYN Target Loop Shape Gd to Meet Design Goals

If your first attempt at loopsyn design does not achieve everything you wanted, you will need to readjust your target desired loop shape Gd. Here are some basic design tradeoffs to consider:

- **Stability Robustness.** Your target loop Gd should have low gain (as small as possible) at high frequencies where typically your plant model is so poor that its phase angle is completely inaccurate, with errors approaching  $\pm 180^\circ$  or more.
- **Performance.** Your Gd loop should have high gain (as great as possible) at frequencies where your model is good, in order to ensure good control accuracy and good disturbance attenuation.
- **Crossover and Roll-Off.** Your desired loop shape Gd should have its 0 dB crossover frequency (denoted  $\omega_c$ ) between the above two frequency ranges, and below the crossover frequency  $\omega_c$  it should roll off with a negative slope of between  $-20$  and  $-40$  dB/decade, which helps to keep phase lag to less than  $-180^\circ$  inside the control loop bandwidth ( $0 < \omega < \omega_c$ ).

Other considerations that might affect your choice of Gd are the right-half-plane poles and zeros of the plant G, which impose fundamental limits on your 0 dB crossover frequency  $\omega_c$  [12]. For instance, your 0 dB crossover  $\omega_c$  must be greater than the magnitude of any plant right-half-plane poles and less than the magnitude of any right-half-plane zeros.

$$\max_{\text{Re}(p_i) > 0} |p_i| < \omega_c < \min_{\text{Re}(z_i) > 0} |z_i|.$$

If you do not take care to choose a target loop shape Gd that conforms to these fundamental constraints, then loopsyn will still compute the optimal loop-shaping controller K for your Gd, but you should expect that the optimal loop  $L = G * K$  will have a poor fit to the target loop shape Gd, and consequently it might be impossible to meet your performance goals.

## Mixed-Sensitivity Loop Shaping

A popular alternative approach to `loopsyn` loop shaping is  $H_\infty$  *mixed-sensitivity* loop shaping, which is implemented by the Robust Control Toolbox software command:

```
K=mixsyn(G,W1,[],W3)
```

With `mixsyn` controller synthesis, your performance and stability robustness specifications equations (2-2) and (2-4) are combined into a single infinity norm specification of the form

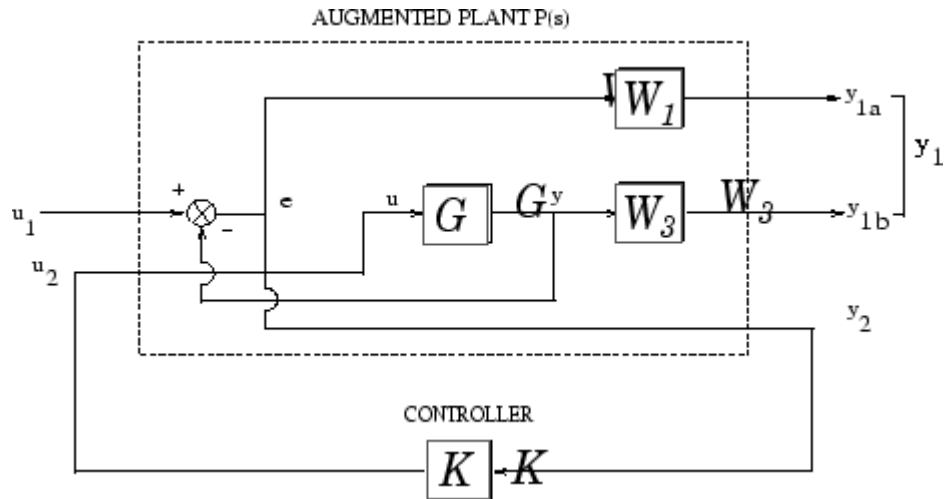
$$\|T_{y_1 u_1}\|_\infty \leq 1$$

where (see MIXSYN  $H_\infty$  Mixed-Sensitivity Loop Shaping  $T_{y_1 u_1}$  on page 2-23):

$$T_{y_1 u_1} \stackrel{\text{def}}{=} \begin{bmatrix} W_1 & S \\ W_3 & T \end{bmatrix}.$$

The term  $\|T_{y_1 u_1}\|_\infty$  is called a *mixed-sensitivity cost function* because it penalizes both sensitivity  $S(s)$  and complementary sensitivity  $T(s)$ . Loop shaping is achieved when you choose  $W_1$  to have the target loop shape for frequencies  $\omega < \omega_c$ , and you choose  $1/W_3$  to be the target for  $\omega > \omega_c$ . In choosing design specifications  $W_1$  and  $W_3$  for a `mixsyn` controller design, you need to ensure that your 0 dB crossover frequency for the Bode plot of  $W_1$  is below the 0 dB crossover frequency of  $1/W_3$ , as shown in Singular Value Specifications on L, S, and T on page 2-12, so that there is a gap for the desired loop shape  $G_d$  to pass between the performance bound  $W_1$  and your robustness bound

$W_3^{-1}$ . Otherwise, your performance and robustness requirements will not be achievable.



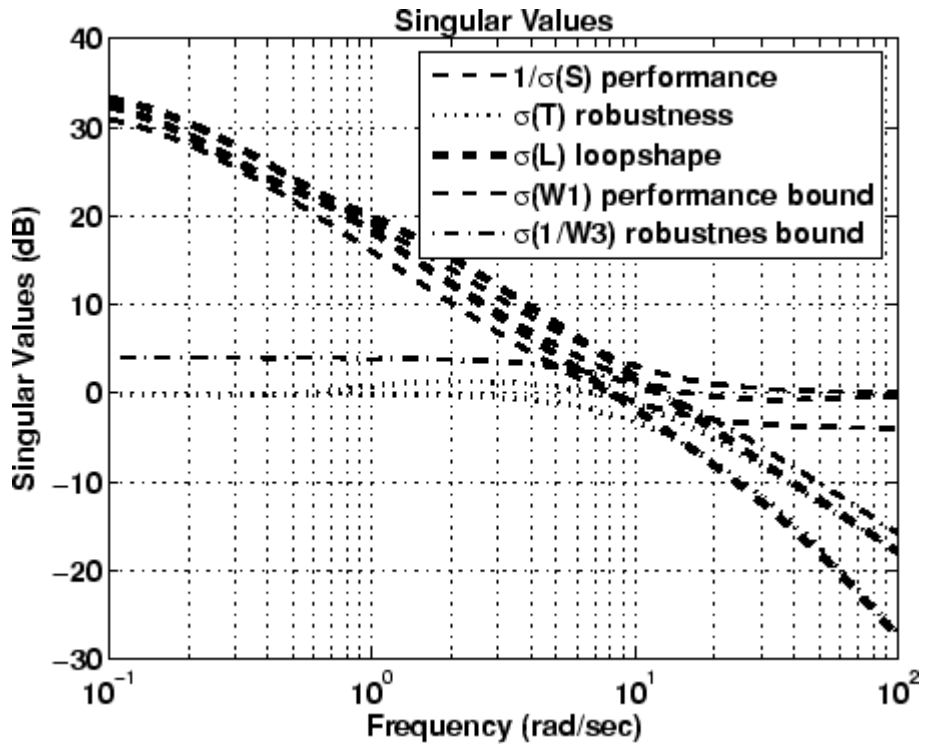
**MIXSYN H. Mixed-Sensitivity Loop Shaping  $T_{y_1 u_1}$**

## Mixed-Sensitivity Loop-Shaping Controller Design

To do a  $\text{mixsyn } H_\infty$  mixed-sensitivity synthesis design on the HiMAT model, start with the plant model  $G$  created in “Mixed-Sensitivity Loop-Shaping Controller Design” on page 2-24 and type the following commands:

```
% Set up the performance and robustness bounds W1 & W3
s=zpk('s'); % Laplace variable s
MS=2;AS=.03;WS=5;
W1=(s/MS+WS)/(s+AS*WS);
MT=2;AT=.05;WT=20;
W3=(s+WT/MT)/(AT*s+WT);
% Compute the H-infinity mixed-sensitivity optimal controller K1
[K1,CL1,GAM1]=mixsyn(G,W1,[],W3);
% Next compute and plot the closed-loop system.
% Compute the loop L1, sensitivity S1, and comp sensitivity T1:
L1=G*K1;
I=eye(size(L1));
S1=feedback(I,L1); % S=inv(I+L1);
T1=I-S1;
% Plot the results:
% step response plots
step(T1,1.5);
title('\alpha and \theta command step responses');
% frequency response plots
figure;
sigma(I+L1,'- -',T1,': ',L1,'r- -',... W1/GAM1,'k- -',GAM1/W3,'k- .',{.1,100});grid
legend('1/\sigma(S) performance',...
'\sigma(T) robustness',...
'\sigma(L) loopshape',...
'\sigma(W1) performance bound',...
'\sigma(1/W3) robustness bound');
```

The resulting  $\text{mixsyn}$  singular value plots for the NASA HiMAT model are shown below.



**MIXSYN Design Results for NASA HiMAT**

## Loop-Shaping Commands

Robust Control Toolbox software gives you several choices for shaping the frequency response properties of multiinput/multioutput (MIMO) feedback control loops. Some of the main commands that you are likely to use for loop-shaping design, and associated utility functions, are listed below:

<b>MIMO Loop-Shaping Commands</b>	
loopsyn	$H_\infty$ loop-shaping controller synthesis
ltrsyn	LQG loop-transfer recovery
mixsyn	$H_\infty$ mixed-sensitivity controller synthesis
ncfsyn	Glover-McFarlane $H_\infty$ normalized coprime factor loop-shaping controller synthesis
<b>MIMO Loop-Shaping Utility Functions</b>	
augw	Augmented plant for weighted $H_2$ and $H_\infty$ mixed-sensitivity control synthesis
makeweight	Weights for $H_\infty$ mixed sensitivity (mixsyn, augw)
sigma	Singular value plots of LTI feedback loops



# Model Reduction for Robust Control

---

- “Why Reduce Model Order?” on page 3-2
- “Hankel Singular Values” on page 3-3
- “Model Reduction Techniques” on page 3-5
- “Approximate Plant Model by Additive Error Methods” on page 3-7
- “Approximate Plant Model by Multiplicative Error Method” on page 3-9
- “Using Modal Algorithms” on page 3-12
- “Reducing Large-Scale Models” on page 3-16
- “Normalized Coprime Factor Reduction” on page 3-17
- “Bibliography” on page 3-19

## Why Reduce Model Order?

In the design of robust controllers for complicated systems, model reduction fits several goals:

- 1** To simplify the best available model in light of the purpose for which the model is to be used—namely, to design a control system to meet certain specifications.
- 2** To speed up the simulation process in the design validation stage, using a smaller size model with most of the important system dynamics preserved.
- 3** Finally, if a modern control method such as LQG or  $H_\infty$  is used for which the complexity of the control law is not explicitly constrained, the order of the resultant controller is likely to be considerably greater than is truly needed. A good model reduction algorithm applied to the control law can sometimes significantly reduce control law complexity with little change in control system performance.

Model reduction routines in this toolbox can be put into two categories:

- **Additive error method** — The reduced-order model has an additive error bounded by an error criterion.
- **Multiplicative error method** — The reduced-order model has a multiplicative or relative error bounded by an error criterion.

The error is measured in terms of peak gain across frequency ( $H_\infty$  norm), and the error bounds are a function of the neglected Hankel singular values.

## Hankel Singular Values

In control theory, eigenvalues define a system stability, whereas *Hankel singular values* define the “energy” of each state in the system. Keeping larger energy states of a system preserves most of its characteristics in terms of stability, frequency, and time responses. Model reduction techniques presented here are all based on the Hankel singular values of a system. They can achieve a reduced-order model that preserves the majority of the system characteristics.

Mathematically, given a *stable* state-space system  $(A,B,C,D)$ , its Hankel singular values are defined as [1]

$$\sigma_H = \sqrt{\lambda_i(PQ)}$$

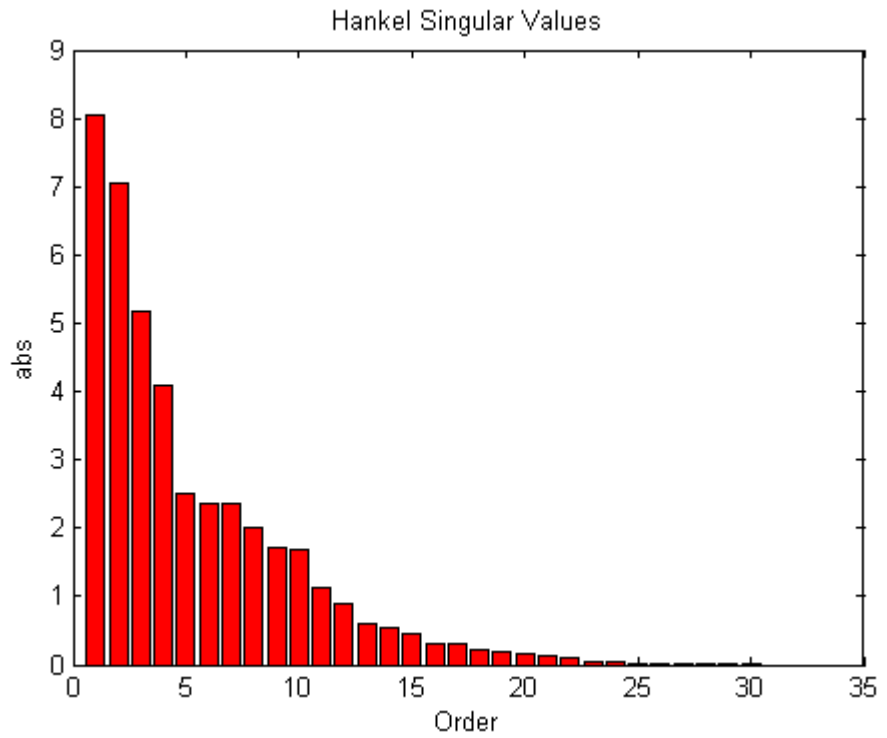
where  $P$  and  $Q$  are *controllability* and *observability grammians* satisfying

$$\begin{aligned}AP + PA^T &= -BB^T \\ A^T Q + QA &= -C^T C.\end{aligned}$$

For example,

```
rng(1234, 'twister');  
G = rss(30,4,3);  
hankelsv(G)
```

returns a Hankel singular value plot as follows:



which shows that system  $G$  has most of its “energy” stored in states 1 through 15 or so. Later, you will see how to use model reduction routines to keep a 15-state reduced model that preserves most of its dynamic response.

## Model Reduction Techniques

Robust Control Toolbox software offers several algorithms for model approximation and order reduction. These algorithms let you control the absolute or relative approximation error, and are all based on the Hankel singular values of the system.

Robust control theory quantifies a system uncertainty as either *additive* or *multiplicative* types. These model reduction routines are also categorized into two groups: *additive error* and *multiplicative error* types. In other words, some model reduction routines produce a reduced-order model  $G_{red}$

of the original model  $G$  with a bound on the error  $\|G - G_{red}\|_{\infty}$ , the peak gain across frequency. Others produce a reduced-order model with a bound on

the relative error  $\|G^{-1}(G - G_{red})\|_{\infty}$ .

These theoretical bounds are based on the “tails” of the Hankel singular values of the model, i.e.,

### Additive Error Bound

$$\|G - G_{red}\|_{\infty} \leq 2 \sum_{k=1}^n \sigma_k \quad (3-1)$$

where  $\sigma_i$  are denoted the  $i$ th Hankel singular value of the original system  $G$ .

### Multiplicative (Relative) Error Bound

$$\|G^{-1}(G - G_{red})\|_{\infty} \leq \prod_{k=1}^n \left( 1 + 2\sigma_k \left( \sqrt{1 + \sigma_k^2} + \sigma_k \right) \right) - 1 \quad (3-2)$$

where  $\sigma_i$  are denoted the  $i$ th Hankel singular value of the phase matrix of the model  $G$  (see the `bstmr` reference page).

**Top-Level Model Reduction Command**

Method	Description
reduce	Main interface to model approximation algorithms

**Normalized Coprime Balanced Model Reduction Command**

Method	Description
ncfmr	Normalized coprime balanced truncation

**Additive Error Model Reduction Commands**

Method	Description
balancmr	Square-root balanced model truncation
schurmr	Schur balanced model truncation
hankelmr	Hankel minimum degree approximation

**Multiplicative Error Model Reduction Command**

Method	Description
bstmr	Balanced stochastic truncation

**Additional Model Reduction Tools**

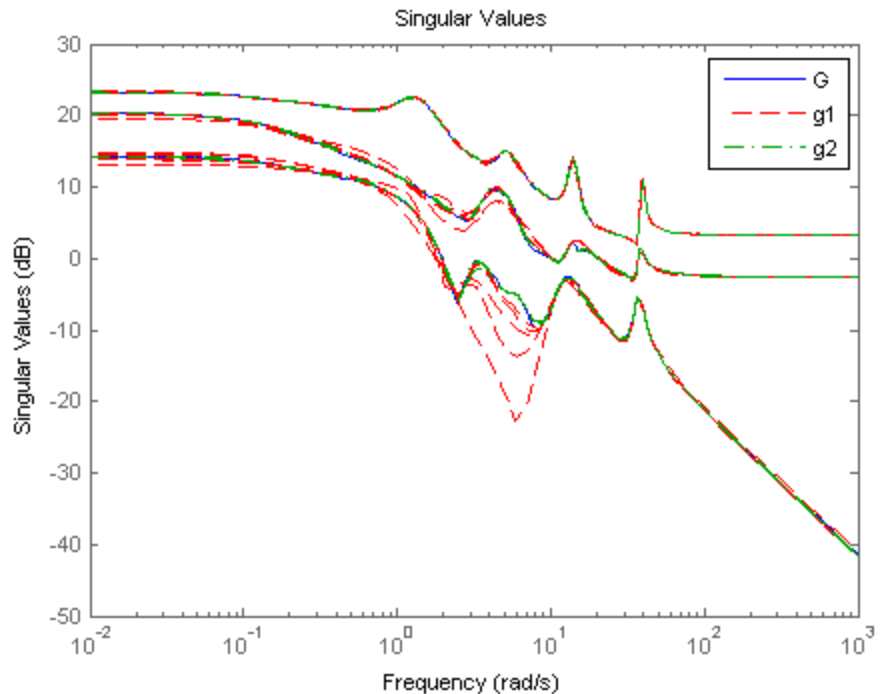
Method	Description
modreal	Modal realization and truncation
slowfast	Slow and fast state decomposition
stabsep	Stable and antistable state projection

## Approximate Plant Model by Additive Error Methods

Given a system in LTI form, the following commands reduce the system to any desired order you specify. The judgment call is based on its Hankel singular values as shown in the previous paragraph.

```
rng(1234, 'twister');
G = rss(30,4,3);
% balanced truncation to models with sizes 12:16
[g1,info1] = balancmr(G,12:16); % or use reduce
% Schur balanced truncation by specifying `MaxError'
[g2,info2] = schurmr(G, 'MaxError', [1,0.8,0.5,0.2]);
sigma(G, 'b-', g1, 'r--', g2, 'g-.')
```

shows a comparison plot of the original model G and reduced models g1 and g2.

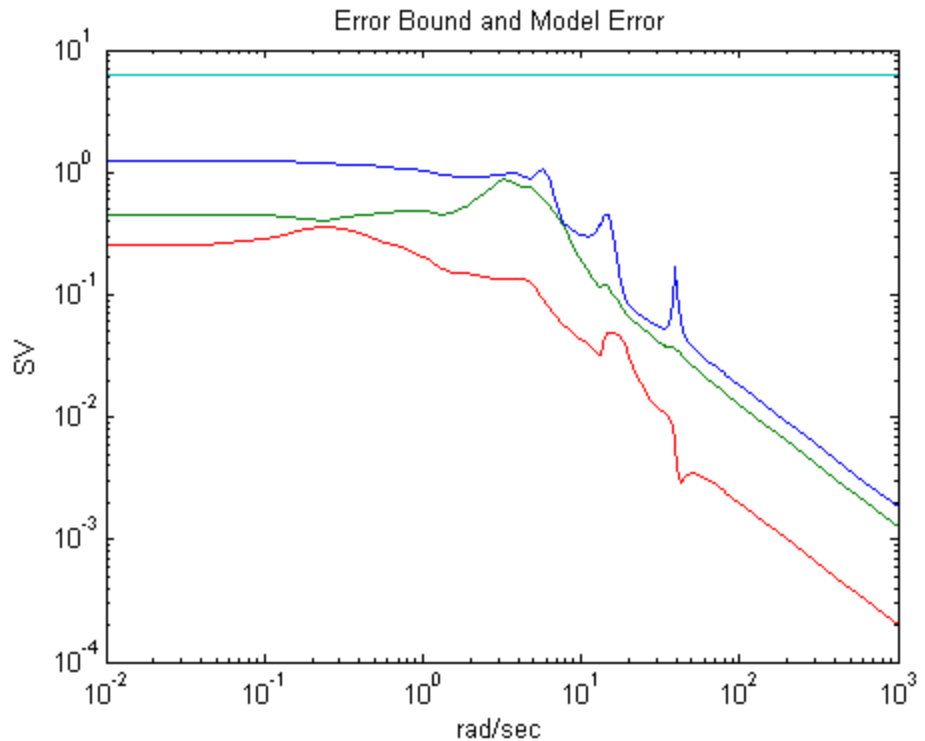


To determine whether the theoretical error bound is satisfied,

```
norm(G-g1(:, :, 1), 'inf') % 1.2556  
info1.ErrorBound(1) % 6.2433
```

or plot the model error vs. error bound via the following commands:

```
[sv,w] = sigma(G-g1(:, :, 1));  
loglog(w,sv,w,info1.ErrorBound(1)*ones(size(w)))  
xlabel('rad/sec');ylabel('SV');  
title('Error Bound and Model Error')
```



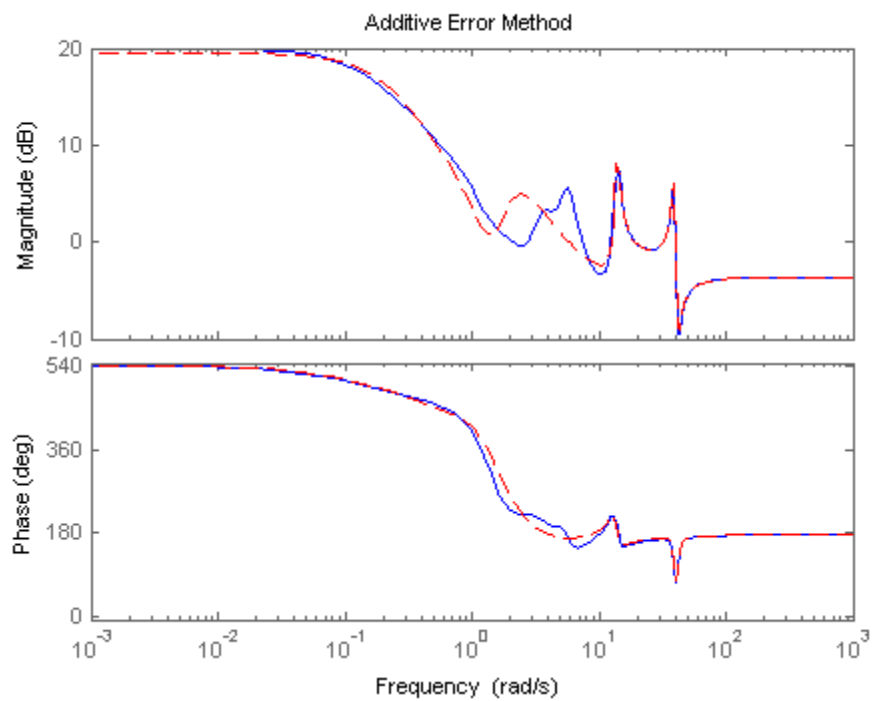


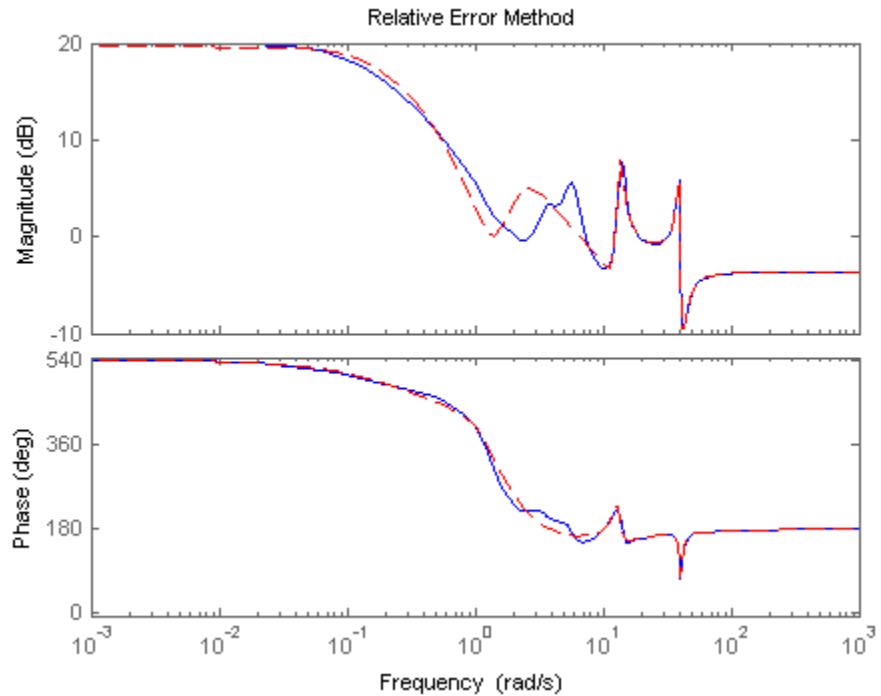
## Approximate Plant Model by Multiplicative Error Method

In most cases, *multiplicative error* model reduction method `bstmr` tends to bound the relative error between the original and reduced-order models across the frequency range of interest, hence producing a more accurate reduced-order model than the *additive error* methods. This characteristic is obvious in system models with low damped poles.

The following commands illustrate the significance of a multiplicative error model reduction method as compared to any additive error type. Clearly, the phase-matching algorithm using `bstmr` provides a better fit in the Bode plot.

```
rng(1234, 'twister');  
G = rss(30,1,1);  
[gr,infor] = reduce(G, 'algo', 'balance', 'order', 7);  
[gs,infos] = reduce(G, 'algo', 'bst', 'order', 7);  
figure(1);bode(G, 'b-', gr, 'r--');  
title('Additive Error Method')  
figure(2);bode(G, 'b-', gs, 'r--');  
title('Relative Error Method')
```





Therefore, for some systems with low damped poles/zeros, the balanced stochastic method (bstmr) produces a better reduced-order model fit in those frequency ranges to make multiplicative error small. Whereas additive error methods such as balancmr, schurmr, or hankelmr only care about minimizing the overall “absolute” peak error, they can produce a reduced-order model missing those low damped poles/zeros frequency regions.

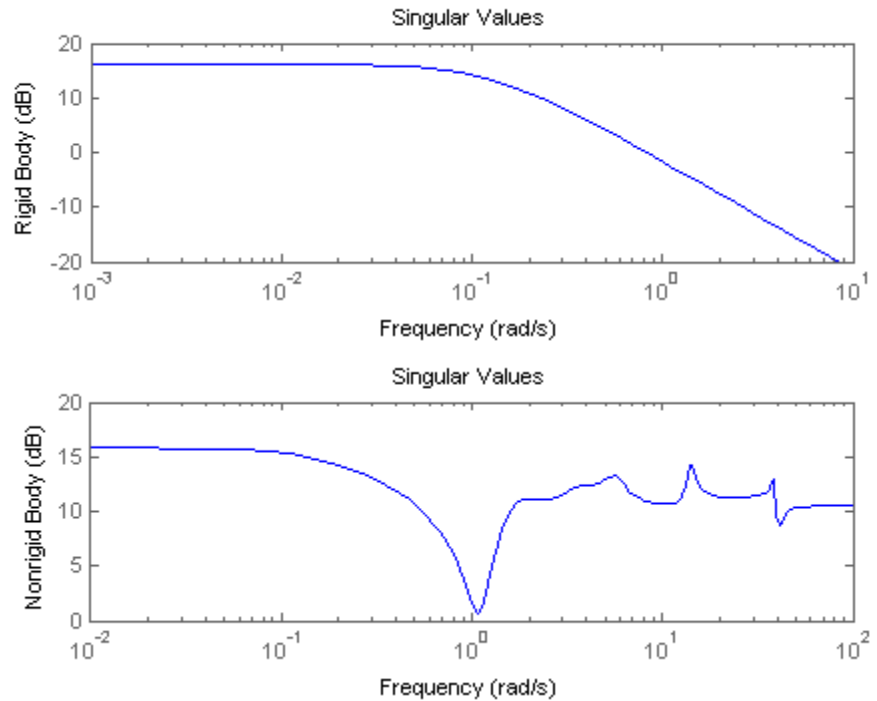
## Using Modal Algorithms

### Rigid Body Dynamics

In many cases, a model's  $j\omega$ -axis poles are important to keep after model reduction, e.g., rigid body dynamics of a flexible structure plant or integrators of a controller. A unique routine, `modreal`, serves the purpose nicely.

`modreal` puts a system into its *modal form*, with eigenvalues appearing on the diagonal of its A-matrix. Real eigenvalues appear in 1-by-1 blocks, and complex eigenvalues appear in 2-by-2 real blocks. All the blocks are ordered in ascending order, based on their eigenvalue magnitudes, by default, or descending order, based on their real parts. Therefore, specifying the number of  $j\omega$ -axis poles splits the model into two systems with one containing only  $j\omega$ -axis dynamics, the other containing the non- $j\omega$  axis dynamics.

```
rng(1234, 'twister');  
G = rss(30,1,1);  
[Gjw,G2] = modreal(G,1); % only one rigid body dynamics  
G2.d = Gjw.d; Gjw.d = 0; % put DC gain of G into G2  
subplot(211);sigma(Gjw);ylabel('Rigid Body')  
subplot(212);sigma(G2);ylabel('Nonrigid Body')
```



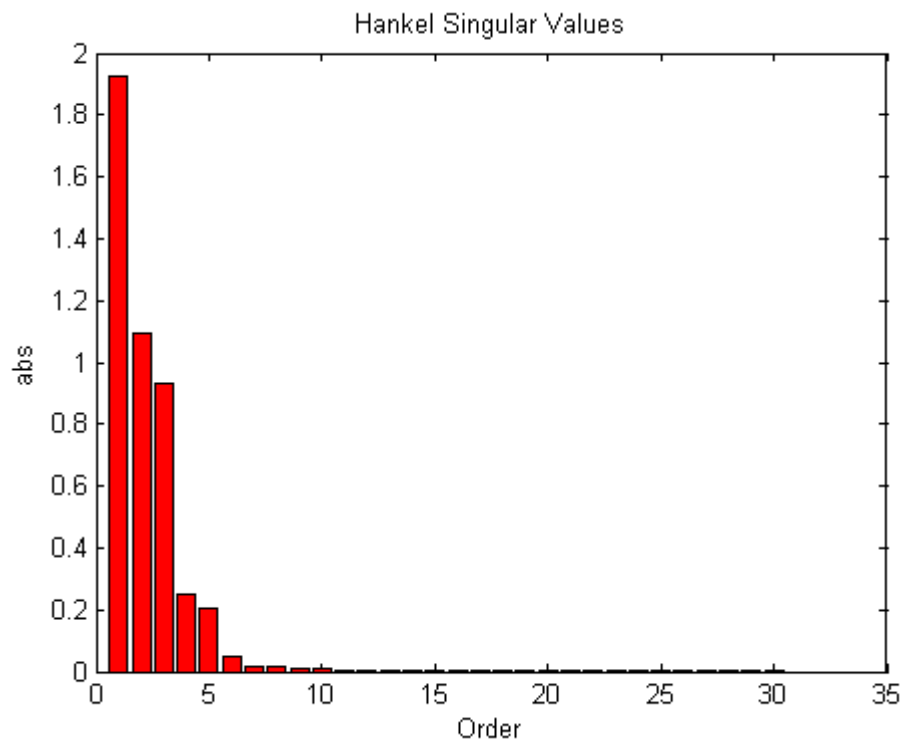
Further model reduction can be done on  $G_2$  without any numerical difficulty. After  $G_2$  is further reduced to  $G_{red}$ , the final approximation of the model is simply  $G_{jw} + G_{red}$ .

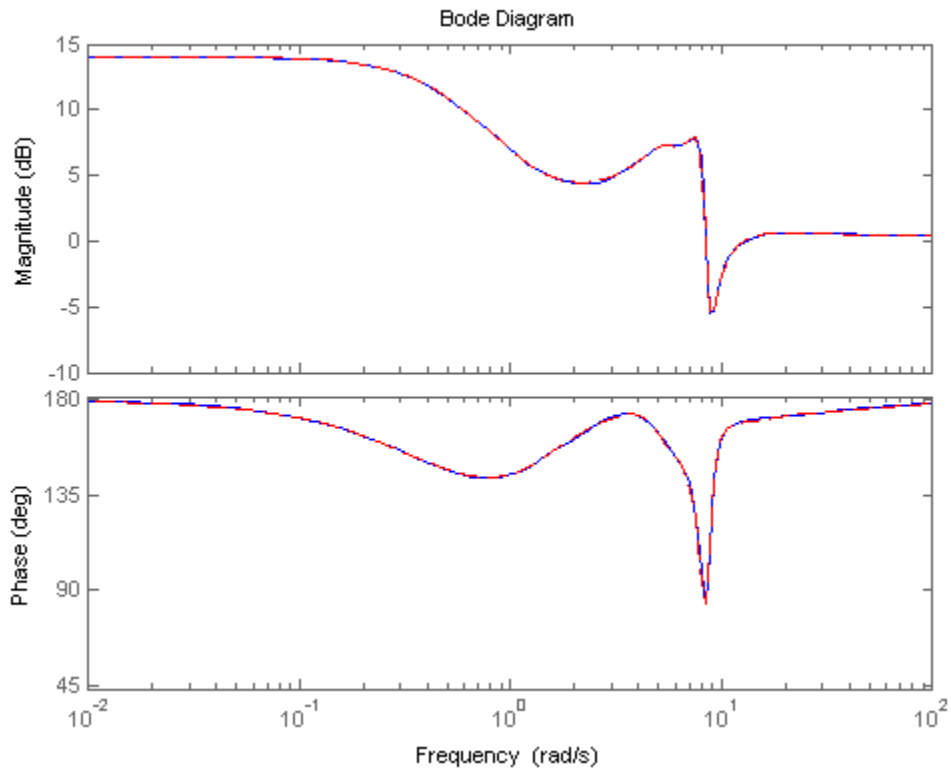
This process of splitting  $j\omega$ -axis poles has been built in and automated in all the model reduction routines (`balancmr`, `schurmr`, `hankelmr`, `bstmr`, `hankelsv`) so that users need not worry about splitting the model.

The following single command creates a size 8 reduced-order model from its original 30-state model:

```
rng(5678, 'twister');
G = rss(30,1,1);
[gr,info] = reduce(G); % choose a size of 8 at prompt
bode(G, 'b-', gr, 'r--')
```

Without specifying the size of the reduced-order model, a Hankel singular value plot is shown below.





The default algorithm `balancmr` of `reduce` has done a great job of approximating a 30-state model with just eight states. Again, the rigid body dynamics are preserved for further controller design.

## Reducing Large-Scale Models

For some really large size problems (states  $> 200$ ), `modreal` turns out to be the only way to start the model reduction process. Because of the size and numerical properties associated with those large size, and low damped dynamics, most Hankel based routines can fail to produce a good reduced-order model.

`modreal` puts the large size dynamics into the modal form, then truncates the dynamic model to an intermediate stage model with a comfortable size of 50 or so states. From this point on, those more sophisticated Hankel singular value based routines can further reduce this intermediate stage model, in a much more accurate fashion, to a smaller size for final controller design.

For a typical 240-state flexible spacecraft model in the spacecraft industry, applying `modreal` and `bstmr` (or any other additive routines) in sequence can reduce the original 240-state plant dynamics to a seven-state three-axis model including rigid body dynamics. Any modern robust control design technique mentioned in this toolbox can then be easily applied to this smaller size plant for a controller design.



## Normalized Coprime Factor Reduction

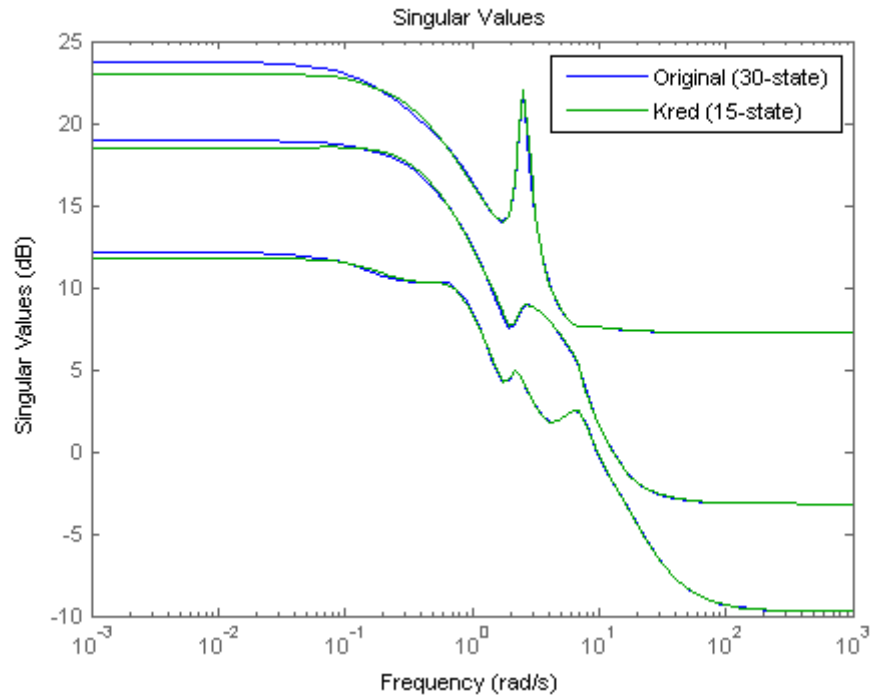
A special model reduction routine `ncfmr` produces a reduced-order model by truncating a balanced coprime set of a given model. It can directly simplify a modern controller with integrators to a smaller size by balanced truncation of the normalized coprime factors. It does not need `modreal` for pre-/postprocessing as the other routines do. However, any integrators in the model will not be preserved.

```
rng(89, 'twister');  
K= rss(30,4,3);  
[Kred,info2] = ncfmr(K);
```

Again, without specifying the size of the reduced-order model, any model reduction routine presented here will plot a Hankel singular value bar chart and prompt you for a reduced model size. In this case, enter 15.

Then, plot the singular values of the original and reduced-order models.

```
sigma(K,Kred)  
legend('Original (30-state)', 'Kred (15-state)')
```



If integral control is important, previously mentioned methods (except `ncfmr`) can nicely preserve the original integrator(s) in the model.

## Bibliography

- [1] Glover, K., "All Optimal Hankel Norm Approximation of Linear Multivariable Systems, and Their  $L^\infty$  - Error Bounds," *Int. J. Control*, Vol. 39, No. 6, 1984, pp. 1145-1193.
- [2] Zhou, K., Doyle, J.C., and Glover, K., *Robust and Optimal Control*, Englewood Cliffs, NJ, Prentice Hall, 1996.
- [3] Safonov, M.G., and Chiang, R.Y., "A Schur Method for Balanced Model Reduction," *IEEE Trans. on Automat. Contr.*, Vol. 34, No. 7, July 1989, pp. 729-733.
- [4] Safonov, M.G., Chiang, R.Y., and Limebeer, D.J.N., "Optimal Hankel Model Reduction for Nonminimal Systems," *IEEE Trans. on Automat. Contr.*, Vol. 35, No. 4, April 1990, pp. 496-502.
- [5] Safonov, M.G., and Chiang, R.Y., "Model Reduction for Robust Control: A Schur Relative Error Method," *International J. of Adaptive Control and Signal Processing*, Vol. 2, 1988, pp. 259-272.
- [6] Obinata, G., and Anderson, B.D.O., *Model Reduction for Control System Design*, London, Springer-Verlag, 2001.



# Robustness Analysis

---

- “Create Models of Uncertain Systems” on page 4-2
- “Robust Controller Design” on page 4-9
- “MIMO Robustness Analysis” on page 4-14
- “Summary of Robustness Analysis Tools” on page 4-27

## Create Models of Uncertain Systems

Dealing with and understanding the effects of uncertainty are important tasks for the control engineer. Reducing the effects of some forms of uncertainty (initial conditions, low-frequency disturbances) without catastrophically increasing the effects of other dominant forms (sensor noise, model uncertainty) is the primary job of the feedback control system.

Closed-loop stability is the way to deal with the (always present) uncertainty in initial conditions or arbitrarily small disturbances.

High-gain feedback in low-frequency ranges is a way to deal with the effects of unknown biases and disturbances acting on the process output. In this case, you are forced to use roll-off filters in high-frequency ranges to deal with high-frequency sensor noise in a feedback system.

Finally, notions such as gain and phase margins (and their generalizations) help quantify the sensitivity of stability and performance in the face of *model uncertainty*, which is the imprecise knowledge of how the control input directly affects the feedback variables.

Robust Control Toolbox software has built-in features allowing you to specify model uncertainty simply and naturally. The primary building blocks, called *uncertain elements* (or uncertain Control Design Blocks) are uncertain real parameters and uncertain linear, time-invariant objects. These can be used to create coarse and simple or detailed and complex descriptions of the model uncertainty present within your process models.

Once formulated, high-level system robustness tools can help you analyze the potential degradation of stability and performance of the closed-loop system brought on by the system model uncertainty.

### **Creating Uncertain Models of Dynamic Systems**

The two dominant forms of model uncertainty are as follows:

- Uncertainty in parameters of the underlying differential equation models

- Frequency-domain uncertainty, which often quantifies model uncertainty by describing absolute or relative uncertainty in the process's frequency response

Using these two basic building blocks, along with conventional system creation commands (such as `ss` and `tf`), you can easily create uncertain system models.

## Creating Uncertain Parameters

An uncertain parameter has a name (used to identify it within an uncertain system with many uncertain parameters) and a nominal value. Being uncertain, it also has variability, described in one of the following ways:

- An additive deviation from the nominal
- A range about the nominal
- A percentage deviation from the nominal

Create a real parameter, with name 'bw', nominal value 5, and a percentage uncertainty of 10%.

```
bw = ureal('bw',5,'Percentage',10)
```

This creates a `ureal` object. View its properties using the `get` command.

```
Uncertain Real Parameter: Name bw, NominalValue 5, variability = [-10 10]%
get(bw)
```

```
    Name: 'bw'
NominalValue: 5
    Mode: 'Percentage'
    Range: [4.5000 5.5000]
 PlusMinus: [-0.5000 0.5000]
  Percentage: [-10 10]
AutoSimplify: 'basic'
```

Note that the range of variation (`Range` property) and the additive deviation from nominal (the `PlusMinus` property) are consistent with the `Percentage` property value.

You can create state-space and transfer function models with uncertain real coefficients using `ureal` objects. The result is an *uncertain state-space object*, or `uss`. As an example, use the uncertain real parameter `bw` to model a first-order system whose bandwidth is between 4.5 and 5.5 rad/s.

```
H = tf(1,[1/bw 1])
USS: 1 State, 1 Output, 1 Input, Continuous System
    bw: real, nominal = 5, variability = [-10 10]%, 1 occurrence
```

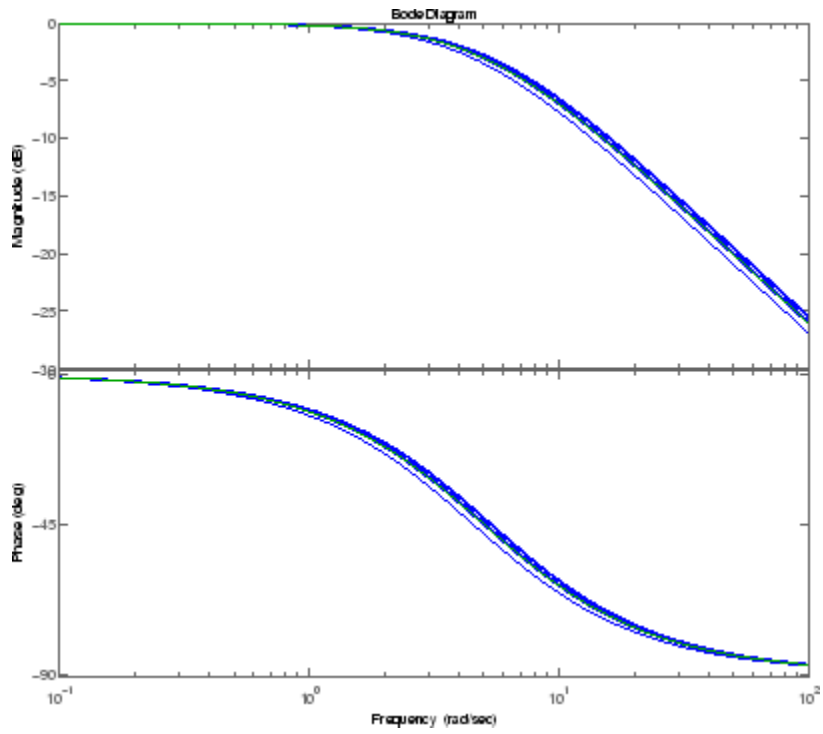
Note that the result `H` is an uncertain system, called a `uss` object. The nominal value of `H` is a state-space object. Verify that the pole is at  $-5$ .

```
pole(H.NominalValue)
ans =
    -5
```

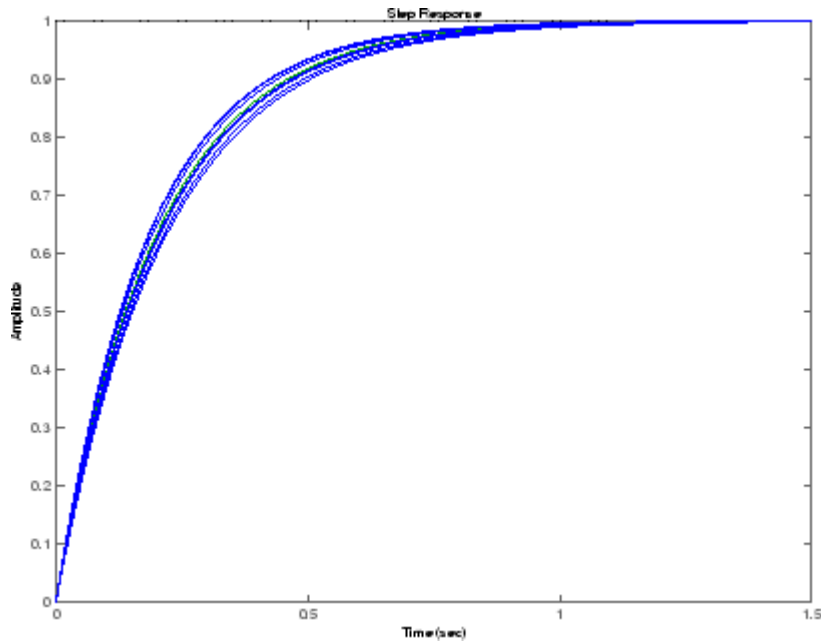
Next, use `bode` and `step` to examine the behavior of `H`.

```
bode(H,{1e-1 1e2});
```





step(H)



While there are variations in the bandwidth and time constant of  $H$ , the high-frequency rolls off at 20 dB/decade regardless of the value of  $bw$ . You can capture the more complicated uncertain behavior that typically occurs at high frequencies using the `ultidyn` uncertain element, which is described next.

## Quantifying Unmodeled Dynamics

An informal way to describe the difference between the model of a process and the actual process behavior is in terms of bandwidth. It is common to hear “The model is good out to 8 radians/second.” The precise meaning is not clear, but it is reasonable to believe that for frequencies lower than, say, 5 rad/s, the model is accurate, and for frequencies beyond, say, 30 rad/s, the model is not necessarily representative of the process behavior. In the frequency range between 5 and 30, the guaranteed accuracy of the model degrades.

The uncertain linear, time-invariant dynamics object `ultidyn` can be used to model this type of knowledge. An `ultidyn` object represents an unknown linear system whose only known attribute is a uniform magnitude bound on its frequency response. When coupled with a nominal model and a

frequency-shaping filter, `ultidyn` objects can be used to capture uncertainty associated with the model dynamics.

Suppose that the behavior of the system modeled by `H` significantly deviates from its first-order behavior beyond 9 rad/s, for example, about 5% potential relative error at low frequency, increasing to 1000% at high frequency where `H` rolls off. In order to model frequency domain uncertainty as described above using `ultidyn` objects, follow these steps:

- 1** Create the nominal system `Gnom`, using `tf`, `ss`, or `zpk`. `Gnom` itself might already have parameter uncertainty. In this case `Gnom` is `H`, the first-order system with an uncertain time constant.
- 2** Create a filter `W`, called the “weight,” whose magnitude represents the relative uncertainty at each frequency. The utility `makeweight` is useful for creating first-order weights with specific low- and high-frequency gains, and specified gain crossover frequency.
- 3** Create an `ultidyn` object `Delta` with magnitude bound equal to 1.

The uncertain model `G` is formed by  $G = Gnom * (1 + W * Delta)$ .

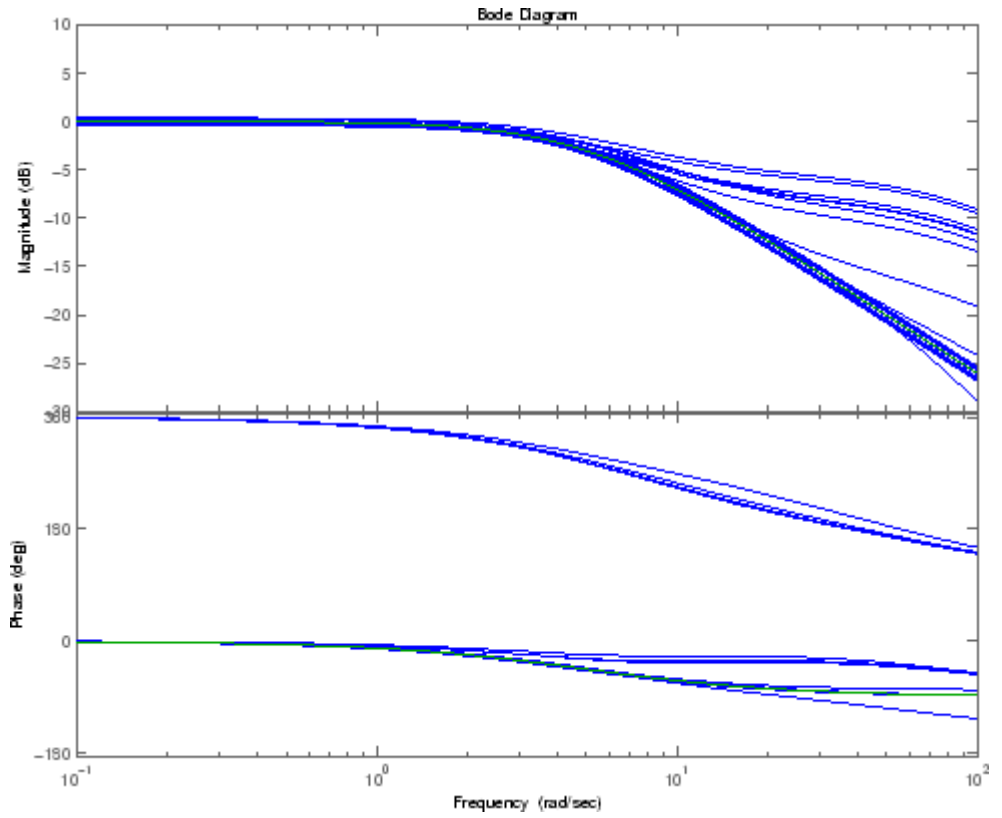
If the magnitude of `W` represents an absolute (rather than relative) uncertainty, use the formula  $G = Gnom + W * Delta$  instead.

The following commands carry out these steps:

```
Gnom = H;
W = makeweight(.05,9,10);
Delta = ultidyn('Delta',[1 1]);
G = Gnom*(1+W*Delta)
USS: 2 States, 1 Output, 1 Input, Continuous System
   Delta: 1x1 LTI, max. gain = 1, 1 occurrence
         bw: real, nominal = 5, variability = [-10 10]%, 1 occurrence
```

Note that the result `G` is also an uncertain system, with dependence on both `Delta` and `bw`. You can use `bode` to make a Bode plot of 20 random samples of `G`'s behavior over the frequency range [0.1 100] rad/s.

```
bode(G,{1e-1 1e2},25)
```



In the next section, you design and compare two feedback controllers for  $G$ .

## Robust Controller Design

In this tutorial, design a feedback controller for  $G$ , the uncertain model created in “Create Models of Uncertain Systems” on page 4-2. The goals of this design are the usual ones: good steady-state tracking and disturbance rejection properties. Because the plant model is nominally a first-order lag, choose a PI control architecture. Given the desired closed-loop damping ratio  $\xi$  and natural frequency  $\omega_n$ , the design equations for  $K_I$  and  $K_P$  (based on the nominal open-loop time constant of 0.2) are

$$K_I = \frac{\omega_n^2}{5}, K_P = \frac{2\xi\omega_n}{5} - 1.$$

Follow these steps to design the controller:

- 1 In order to study how the uncertain behavior of  $G$  affects the achievable closed-loop bandwidth, design two controllers, both achieving  $\xi=0.707$ , with different  $\omega_n$ : 3 and 7.5 respectively.

```
xi = 0.707;
wn = 3;
K1 = tf([(2*xi*wn/5-1) wn*wn/5],[1 0]);
wn = 7.5;
K2 = tf([(2*xi*wn/5-1) wn*wn/5],[1 0]);
```

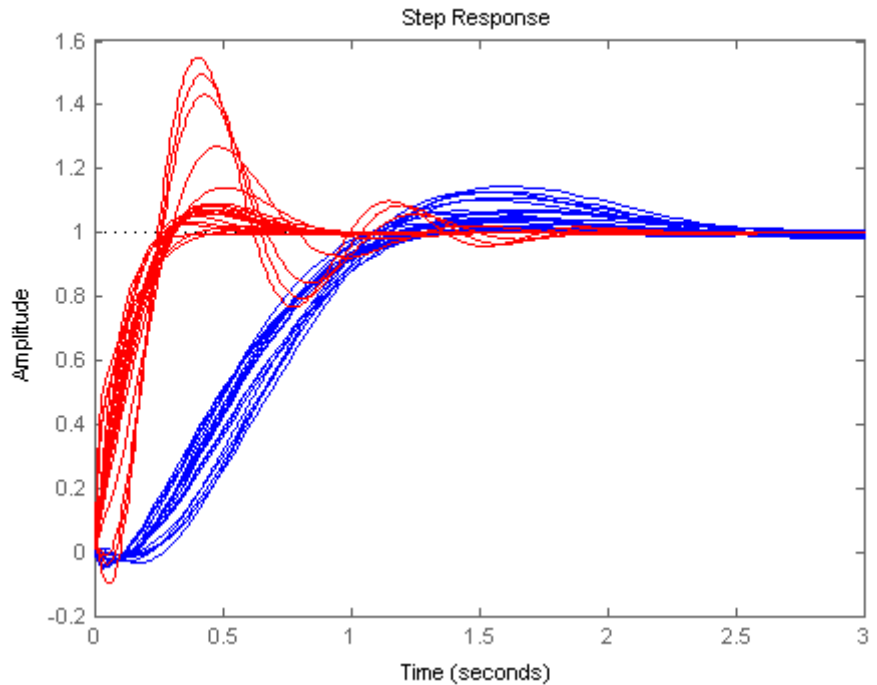
Note that the nominal closed-loop bandwidth achieved by K2 is in a region where  $G$  has significant model uncertainty. It will not be surprising if the model variations lead to significant degradations in the closed-loop performance.

- 2 Form the closed-loop systems using feedback.

```
T1 = feedback(G*K1,1);
T2 = feedback(G*K2,1);
```

- 3 Plot the step responses of 20 samples of each closed-loop system.

```
tfinal = 3;
stepplot(T1,'b',T2,'r',tfinal)
```



The step responses for T2 exhibit a faster rise time because K2 sets a higher closed loop bandwidth. However, the model variations have a greater effect.

You can use `robuststab` to check the robustness of stability to the model variations.

```
[stabmarg1,destabu1,report1] = robuststab(T1);
stabmarg1
stabmarg1 =
    ubound: 4.0241
    lbound: 4.0241
    destabfreq: 3.4959
[stabmarg2,destabu2,report2] = robuststab(T2);
stabmarg2
stabmarg2 =
```

```

ubound: 1.2545
lbound: 1.2544
destabfreq: 10.5249

```

The `stabmarg` variable gives lower and upper bounds on the stability margin. A stability margin greater than 1 means the system is stable for all values of the modeled uncertainty. A stability margin less than 1 means there are allowable values of the uncertain elements that make the system unstable. The `report` variable briefly summarizes the analysis.

```

report1
report1 =
Uncertain System is robustly stable to modeled uncertainty.
-- It can tolerate up to 402% of modeled uncertainty.
-- A destabilizing combination of 402% the modeled uncertainty exists, causing an instability at
3.5 rad/s.
report2
report2 =
Uncertain System is robustly stable to modeled uncertainty.
-- It can tolerate up to 125% of modeled uncertainty.
-- A destabilizing combination of 125% the modeled uncertainty exists, causing an instability at
10.5 rad/s.

```

While both systems are stable for all variations, their performance is clearly affected to different degrees. To determine how the uncertainty affects closed-loop performance, you can use `wcgain` to compute the *worst-case* effect of the uncertainty on the peak magnitude of the closed-loop sensitivity ( $S=1/(1+GK)$ ) function. This peak gain is typically correlated with the amount of overshoot in a step response.

To do this, form the closed-loop sensitivity functions and call `wcgain`.

```

S1 = feedback(1,G*K1);
S2 = feedback(1,G*K2);
[maxgain1,wcu1] = wcgain(S1);
maxgain1 =
    lbound: 1.8684
    ubound: 1.9025
    critfreq: 3.5152

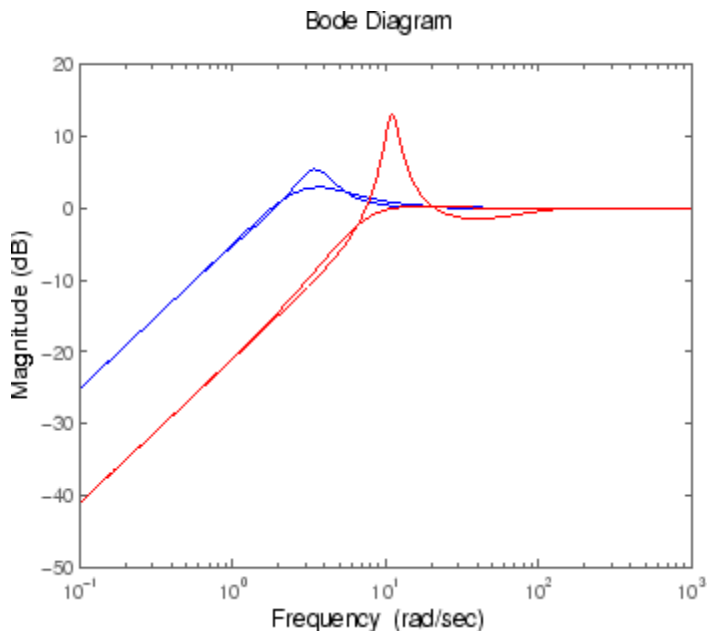
```

```
[maxgain2,wcu2] = wcgain(S2);  
maxgain2  
maxgain2 =  
    lbound: 4.6031  
    ubound: 4.6671  
    critfreq: 11.0231
```

The `maxgain` variable gives lower and upper bounds on the worst-case peak gain of the sensitivity transfer function, as well as the specific frequency where the maximum gain occurs. The `wcu` variable contains specific values of the uncertain elements that achieve this worst-case behavior.

You can use `usubs` to substitute these worst-case values for uncertain elements, and compare the nominal and worst-case behavior. Use `bodemag` and `step` to make the comparison.

```
bodemag(S1.NominalValue,'b',usubs(S1,wcu1),'b');  
hold on  
bodemag(S2.NominalValue,'r',usubs(S2,wcu2),'r');  
hold off
```





Clearly, while  $K_2$  achieves better nominal sensitivity than  $K_1$ , the nominal closed-loop bandwidth extends too far into the frequency range where the process uncertainty is very large. Hence the worst-case performance of  $K_2$  is inferior to  $K_1$  for this particular uncertain model.

The next section explores these robustness analysis tools further on a multiinput, multioutput system.

## MIMO Robustness Analysis

The previous sections focused on simple uncertainty models of single-input and single-output systems, predominantly from a transfer function perspective. You can also create uncertain state-space models made up of uncertain state-space matrices. Moreover, all the analysis tools covered thus far can be applied to these systems as well.

Consider, for example, a two-input, two-output, two-state system whose model has parametric uncertainty in the state-space matrices. First create an uncertain parameter  $p$ . Using the parameter, make uncertain  $A$  and  $C$  matrices. The  $B$  matrix happens to be not-uncertain, although you will add frequency domain input uncertainty to the model in “Adding Independent Input Uncertainty to Each Channel” on page 4-15.

```
p = ureal('p',10,'Percentage',10);
A = [0 p;-p 0];
B = eye(2);
C = [1 p;-p 1];
H = ss(A,B,C,[0 0;0 0]);
```

You can view the properties of the uncertain system  $H$  using the `get` command.

```
get(H)

    a: [2x2 umat]
    b: [2x2 double]
    c: [2x2 umat]
    d: [2x2 double]
    e: []
    StateName: {2x1 cell}
    StateUnit: {2x1 cell}
    NominalValue: [2x2 ss]
    Uncertainty: [1x1 struct]
    InternalDelay: [0x1 double]
    InputDelay: [2x1 double]
    OutputDelay: [2x1 double]
    Ts: 0
    TimeUnit: 'seconds'
    InputName: {2x1 cell}
```

```

    InputUnit: {2x1 cell}
    InputGroup: [1x1 struct]
    OutputName: {2x1 cell}
    OutputUnit: {2x1 cell}
    OutputGroup: [1x1 struct]
    Name: ''
    Notes: {}
    UserData: []

```

The properties `a`, `b`, `c`, `d`, and `StateName` behave in exactly the same manner as `ss` objects. The properties `InputName`, `OutputName`, `InputGroup`, and `OutputGroup` behave in exactly the same manner as all the system objects (`ss`, `zpk`, `tf`, and `frd`). The `NominalValue` is an `ss` object.

## Adding Independent Input Uncertainty to Each Channel

The model for `H` does not include actuator dynamics. Said differently, the actuator models are unity-gain for all frequencies.

Nevertheless, the behavior of the actuator for channel 1 is modestly uncertain (say 10%) at low frequencies, and the high-frequency behavior beyond 20 rad/s is not accurately modeled. Similar statements hold for the actuator in channel 2, with larger modest uncertainty at low frequency (say 20%) but accuracy out to 45 rad/s.

Use `ultidyn` objects `Delta1` and `Delta2` along with shaping filters `W1` and `W2` to add this form of frequency domain uncertainty to the model.

```

W1 = makeweight(.1,20,50);
W2 = makeweight(.2,45,50);
Delta1 = ultidyn('Delta1',[1 1]);
Delta2 = ultidyn('Delta2',[1 1]);
G = H*blkdiag(1+W1*Delta1,1+W2*Delta2)

```

`G =`

Uncertain continuous-time state-space model with 2 outputs, 2 inputs, 4 s  
 The model uncertainty consists of the following blocks:  
 Delta1: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences

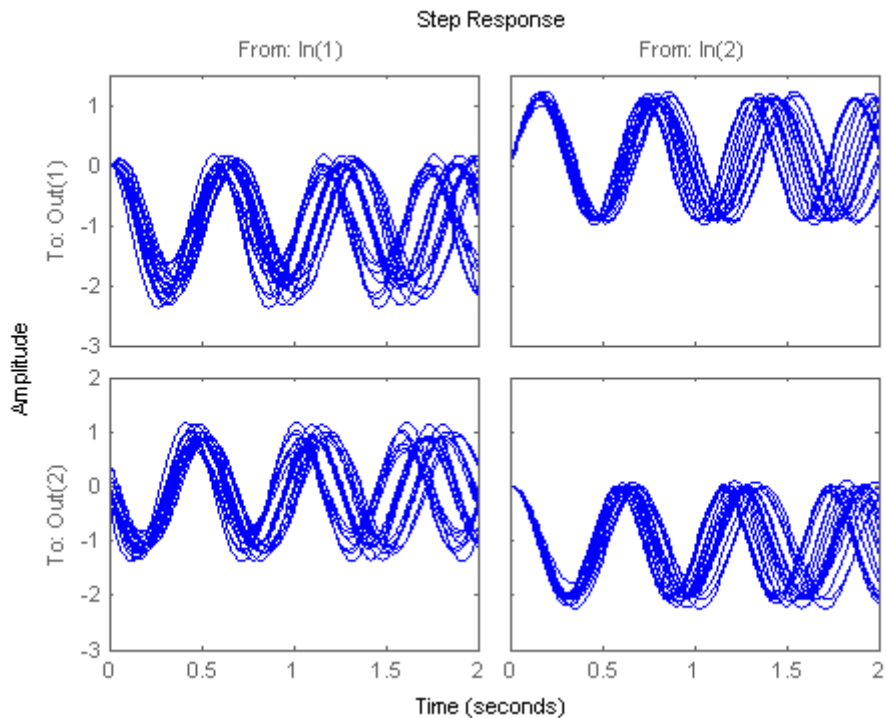
Delta2: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences  
p: Uncertain real, nominal = 10, variability = [-10,10]%, 2 occurrences

Type "G.NominalValue" to see the nominal value, "get(G)" to see all properties

Note that G is a two-input, two-output uncertain system, with dependence on three uncertain elements, Delta1, Delta2, and p. It has four states, two from H and one each from the shaping filters W1 and W2, which are embedded in G.

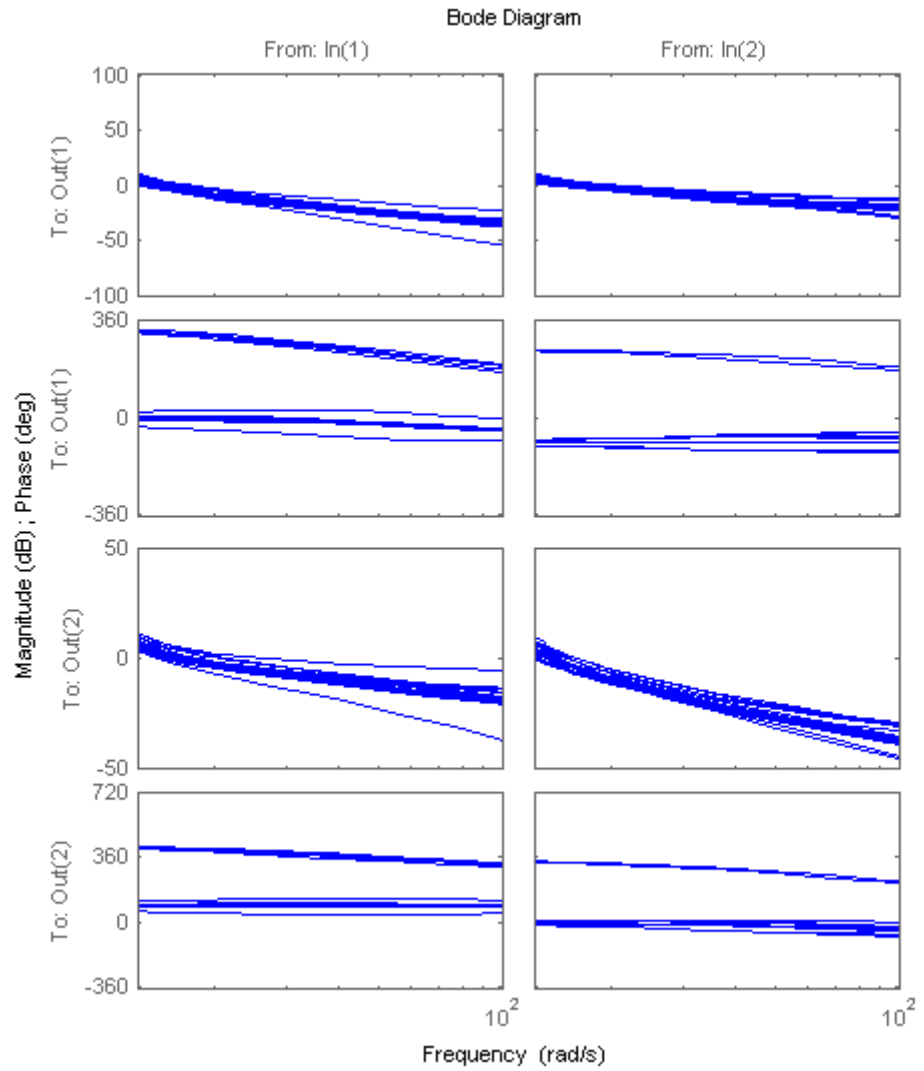
You can plot a 2-second step response of several samples of G. The 10% uncertainty in the natural frequency is obvious.

```
stepplot(G,2)
```



You can create a Bode plot of samples of  $G$ . The high-frequency uncertainty in the model is also obvious. For clarity, start the Bode plot beyond the resonance.

```
bodeplot(G, {13 100})
```



## Closed-Loop Robustness Analysis

Load the controller and verify that it is two-input and two-output.

```
load mimoKexample
size(K)
```

State-space model with 2 outputs, 2 inputs, and 9 states.

You can use the command `loopsens` to form all the standard plant/controller feedback configurations, including sensitivity and complementary sensitivity at both the input and output. Because  $G$  is uncertain, all the closed-loop systems are uncertain as well.

```
F = loopsens(G,K)
```

```
F =
```

```

      Si: [2x2 uss]
      Ti: [2x2 uss]
      Li: [2x2 uss]
      So: [2x2 uss]
      To: [2x2 uss]
      Lo: [2x2 uss]
      PSi: [2x2 uss]
      CSo: [2x2 uss]
      Poles: [13x1 double]
      Stable: 1
```

$F$  is a structure with many fields. The poles of the nominal closed-loop system are in  $F.Poles$ , and  $F.Stable$  is 1 if the nominal closed-loop system is stable. In the remaining 10 fields,  $S$  stands for sensitivity,  $T$  for complementary sensitivity, and  $L$  for open-loop gain. The suffixes  $i$  and  $o$  refer to the input and output of the plant ( $G$ ). Finally,  $P$  and  $C$  refer to the “plant” and “controller.”

Hence  $T_i$  is mathematically the same as

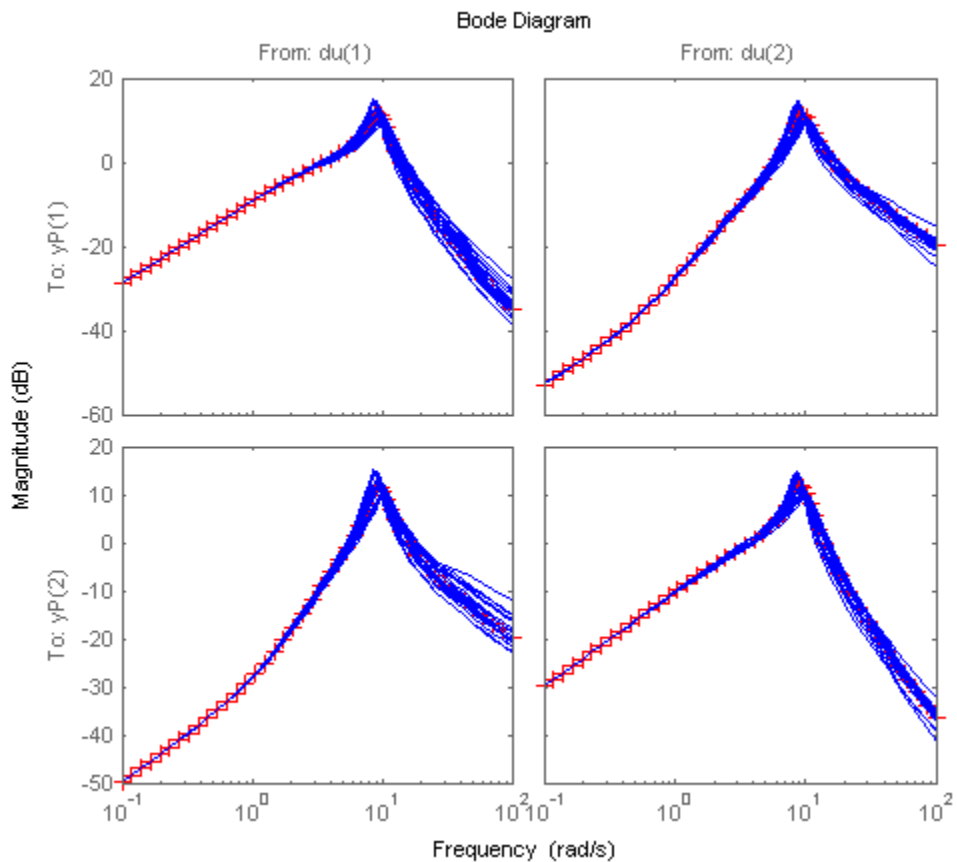
$$K(I + GK)^{-1}G$$

while  $L_o$  is  $G^*K$ , and  $CSo$  is mathematically the same as

$$K(I + GK)^{-1}$$

You can examine the transmission of disturbances at the plant input to the plant output using `bodemag` on `F.PSi`. Graph some samples along with the nominal.

```
bodemag(F.PSi.NominalValue,'r+',F.PSi,'b-',{1e-1 100})
```





## Nominal Stability Margins

You can use `loopmargin` to investigate loop-at-a-time gain and phase margins, loop-at-a-time disk margins, and simultaneous multivariable margins. They are computed for the nominal system and do not reflect the uncertainty models within `G`.

Explore the simultaneous margins individually at the plant input, individually at the plant output, and simultaneously at both input and output.

```
[I,DI,SimI,O,DO,SimO,Sim] = loopmargin(G,K);
```

The third output argument is the *simultaneous* gain and phase variations allowed in all input channels to the plant.

SimI

```
SimI =
```

```
GainMargin: [0.1180 8.4769]
PhaseMargin: [-76.5441 76.5441]
Frequency: 6.2287
```

This information implies that the gain at the plant input can vary in both channels independently by factors between (approximately)  $1/8$  and  $8$ , *as well as* phase variations up to  $76$  degrees.

The sixth output argument is the simultaneous gain and phase variations allowed in all output channels to the plant.

SimO

```
SimO =
```

```
GainMargin: [0.1193 8.3836]
PhaseMargin: [-76.3957 76.3957]
Frequency: 18.3522
```

Note that the simultaneous margins at the plant output are similar to those at the input. This is not always the case in multiloop feedback systems.

The last output argument is the simultaneous gain and phase variations allowed in all input *and* output channels to the plant. As expected, when you consider all such variations simultaneously, the margins are somewhat smaller than those at the input or output alone.

```
Sim
```

```
Sim =
```

```
GainMargin: [0.5671 1.7635]  
PhaseMargin: [-30.8882 30.8882]  
Frequency: 18.3522
```

Nevertheless, these numbers indicate a generally robust closed-loop system, able to tolerate significant gain (more than +/-50% in each channel) and 30 degree phase variations simultaneously in all input and output channels of the plant.

## Robustness of Stability Model Uncertainty

With `loopmargin`, you determined various margins of the nominal, multiloop system. These margins are computed only for the nominal system, and do not reflect the uncertainty explicitly modeled by the `ureal` and `ultidyn` objects. When you work with detailed, complex uncertain system models, the conventional margins computed by `loopmargin` might not always be indicative of the actual stability margins associated with the uncertain elements. You can use `robuststab` to check the stability margin of the system to these specific modeled variations.

In this example, use `robuststab` to compute the stability margin of the closed-loop system represented by `Delta1`, `Delta2`, and `p`.

Use any of the closed-loop systems within `F = loopsens(G,K)`. All of them, `F.Si`, `F.To`, etc., have the same internal dynamics, and hence the stability properties are the same.

```
[stabmarg,desgtabu,report] = robuststab(F.So);  
stabmarg
```

```
stabmarg =
```

```
ubound: 2.2175
lbound: 2.2175
destabfreq: 13.7576
```

```
report
```

```
report =
```

```
Uncertain System is robustly stable to modeled uncertainty.
-- It can tolerate up to 222% of modeled uncertainty.
-- A destabilizing combination of 222% the modeled uncertainty exists, causing an instability at
13.8 rad/s.
```

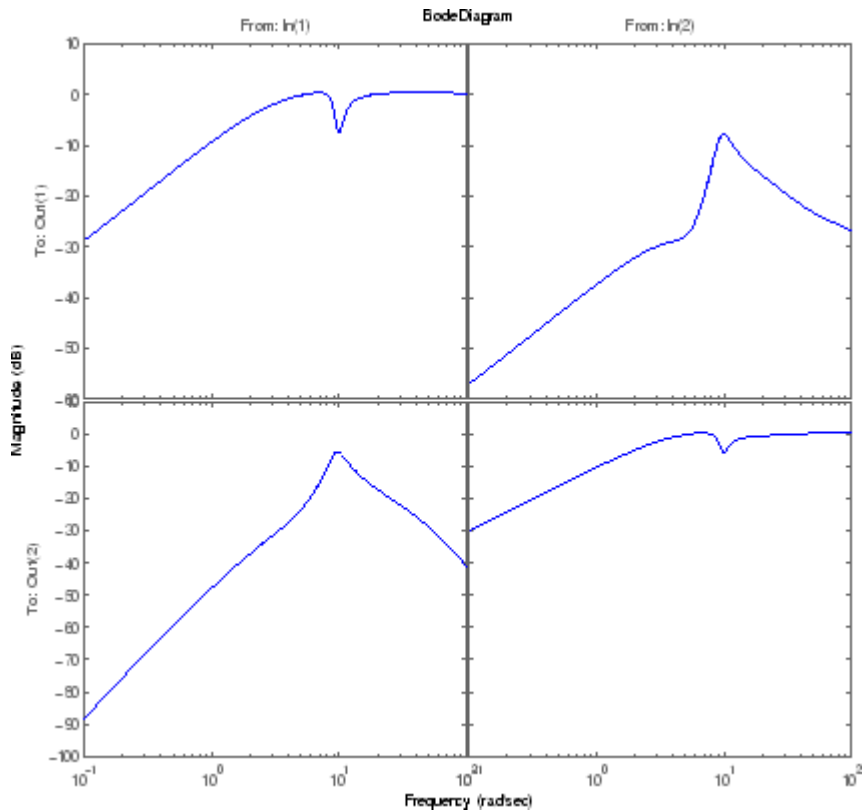
This analysis confirms what the loopmargin analysis suggested. The closed-loop system is quite robust, *in terms of stability*, to the variations modeled by the uncertain parameters Delta1, Delta2, and p. In fact, the system can tolerate more than twice the modeled uncertainty without losing closed-loop stability.

The next section studies the effects of these variations on the closed-loop output sensitivity function.

## **Worst-Case Gain Analysis**

You can plot the Bode magnitude of the nominal output sensitivity function. It clearly shows decent disturbance rejection in all channels at low frequency.

```
bodemag(F.So.NominalValue, {1e-1 100})
```



You can compute the peak value of the maximum singular value of the frequency response matrix using `norm`.

```
[PeakNom, freq] = norm(F.So.NominalValue, 'inf')
```

```
PeakNom =
```

```
1.1317
```

```
freq =
```

```
7.0483
```

The peak is about 1.13, occurring at a frequency of 36 rad/s.

What is the maximum output sensitivity gain that is achieved when the uncertain elements  $\Delta_1$ ,  $\Delta_2$ , and  $p$  vary over their ranges? You can use `wcgain` to answer this.

```
[maxgain,wcu] = wcgain(F.So);  
maxgain
```

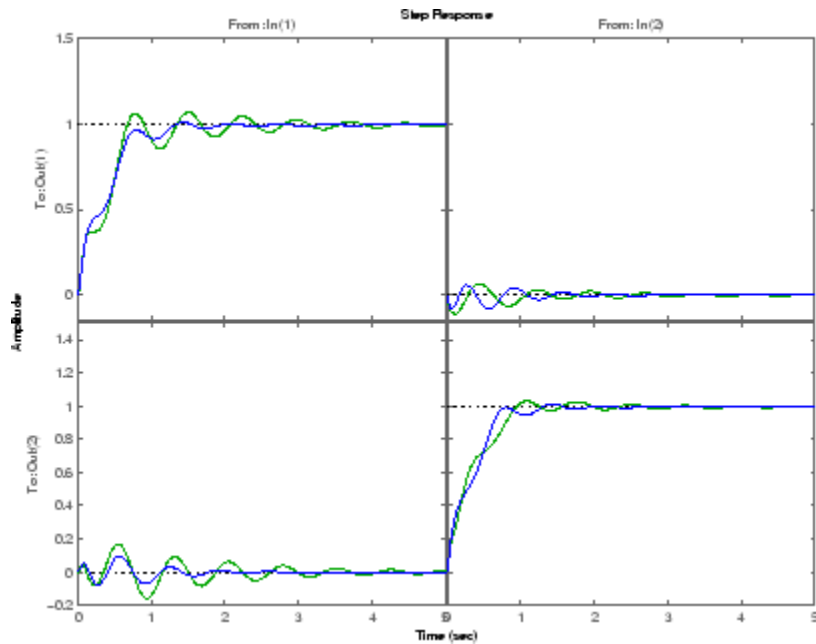
```
maxgain =
```

```
    lbound: 2.1017  
    ubound: 2.1835  
    critfreq: 8.5546
```

The analysis indicates that the worst-case gain is somewhere between 2.1 and 2.2. The frequency where the peak is achieved is about 8.5.

You can replace the values for  $\Delta_1$ ,  $\Delta_2$ , and  $p$  that achieve the gain of 2.1, using `usubs`. Make the substitution in the output complementary sensitivity, and do a step response.

```
step(F.To.NominalValue,usubs(F.To,wcu),5)
```



The perturbed response, which is the worst combination of uncertain values in terms of output sensitivity amplification, does not show significant degradation of the command response. The settling time is increased by about 50%, from 2 to 4, and the off-diagonal coupling is increased by about a factor of about 2, but is still quite small.

## Summary of Robustness Analysis Tools

Function	Description
ureal	Create uncertain real parameter.
ultidyn	Create uncertain, linear, time-invariant dynamics.
uss	Create uncertain state-space object from uncertain state-space matrices.
ufrd	Create uncertain frequency response object.
loopsens	Compute all relevant open and closed-loop quantities for a MIMO feedback connection.
loopmargin	Compute loop-at-a-time as well as MIMO gain and phase margins for a multiloop system, including the simultaneous gain/phase margins.
robustperf	Robustness performance of uncertain systems.
robuststab	Compute the robust stability margin of a nominally stable uncertain system.
wcgain	Compute the worst-case gain of a nominally stable uncertain system.
wcmargin	Compute worst-case (over uncertainty) loop-at-a-time disk-based gain and phase margins.
wcsens	Compute worst-case (over uncertainty) sensitivity of plant-controller feedback loop.





# H-Infinity and Mu Synthesis

---

- “Interpretation of H-Infinity Norm” on page 5-2
- “H-Infinity Performance” on page 5-9
- “Active Suspension Control Design” on page 5-17
- “Bibliography” on page 5-38

## Interpretation of H-Infinity Norm

### Norms of Signals and Systems

There are several ways of defining norms of a scalar signal  $e(t)$  in the time domain. We will often use the 2-norm, ( $L_2$ -norm), for mathematical convenience, which is defined as

$$\|e\|_2 := \left( \int_{-\infty}^{\infty} e(t)^2 dt \right)^{\frac{1}{2}}.$$

If this integral is finite, then the signal  $e$  is *square integrable*, denoted as  $e \in L_2$ . For vector-valued signals

$$e(t) = \begin{bmatrix} e_1(t) \\ e_2(t) \\ \vdots \\ e_n(t) \end{bmatrix},$$

the 2-norm is defined as

$$\begin{aligned} \|e\|_2 &:= \left( \int_{-\infty}^{\infty} \|e(t)\|_2^2 dt \right)^{\frac{1}{2}} \\ &= \left( \int_{-\infty}^{\infty} e^T(t) e(t) dt \right)^{\frac{1}{2}}. \end{aligned}$$

In  $\mu$ -tools the dynamic systems we deal with are exclusively linear, with state-space model

$$\begin{bmatrix} \dot{x} \\ e \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ d \end{bmatrix},$$

or, in the transfer function form,

$$e(s) = T(s)d(s), \quad T(s) := C(sI - A)^{-1}B + D$$

Two mathematically convenient measures of the transfer matrix  $T(s)$  in the frequency domain are the matrix  $H_2$  and  $H_\infty$  norms,

$$\|T\|_2 := \left[ \frac{1}{2\pi} \int_{-\infty}^{\infty} \|T(j\omega)\|_F^2 d\omega \right]^{\frac{1}{2}}$$

$$\|T\|_\infty := \max_{\omega \in R} \bar{\sigma}[T(j\omega)],$$

where the Frobenius norm (see the MATLAB `norm` command) of a complex matrix  $M$  is

$$\|M\|_F := \sqrt{\text{Trace}(M^* M)}.$$

Both of these transfer function norms have input/output time-domain interpretations. If, starting from initial condition  $x(0) = 0$ , two signals  $d$  and  $e$  are related by

$$\begin{bmatrix} \dot{x} \\ e \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ d \end{bmatrix},$$

then

- For  $d$ , a unit intensity, white noise process, the steady-state variance of  $e$  is  $\|T\|_2$ .
- The  $L_2$  (or RMS) gain from  $d \rightarrow e$ ,

$$\max_{d \neq 0} \frac{\|e\|_2}{\|d\|_2}$$

is equal to  $\|T\|_\infty$ . This is discussed in greater detail in the next section.

## Using Weighted Norms to Characterize Performance

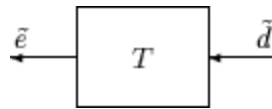
Any performance criterion must also account for

- Relative magnitude of outside influences
- Frequency dependence of signals
- Relative importance of the magnitudes of regulated variables

So, if the performance objective is in the form of a matrix norm, it should actually be a *weighted norm*

$$\|W_L T W_R\|$$

where the weighting function matrices  $W_L$  and  $W_R$  are frequency dependent, to account for bandwidth constraints and spectral content of exogenous signals. The most natural (mathematical) manner to characterize acceptable performance is in terms of the MIMO  $\|\cdot\|_\infty (H_\infty)$  norm. For this reason, this section now discusses some interpretations of the  $H_\infty$  norm.

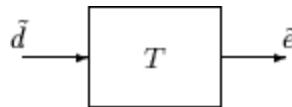


**Unweighted MIMO System**

Suppose  $T$  is a MIMO stable linear system, with transfer function matrix  $T(s)$ .

For a given driving signal  $\tilde{d}(t)$ , define  $\tilde{e}$  as the output, as shown below.

Note that it is more traditional to write the diagram in Unweighted MIMO System: Vectors from Left to Right on page 5-4 with the arrows going from left to right as in Weighted MIMO System on page 5-6.



**Unweighted MIMO System: Vectors from Left to Right**

The two diagrams shown above represent the exact same system. We prefer to write these block diagrams with the arrows going right to left to be consistent with matrix and operator composition.

Assume that the dimensions of  $T$  are  $n_e \times n_d$ . Let  $\beta > 0$  be defined as

$$\beta := \|T\|_\infty := \max_{\omega \in R} \bar{\sigma}[T(j\omega)].$$

Now consider a response, starting from initial condition equal to 0. In that case, Parseval's theorem gives that

$$\frac{\|\tilde{e}\|_2}{\|\tilde{d}\|_2} = \frac{\left[ \int_0^\infty \tilde{e}^T(t) \tilde{e}(t) dt \right]^{\frac{1}{2}}}{\left[ \int_0^\infty \tilde{d}^T(t) \tilde{d}(t) dt \right]^{\frac{1}{2}}} \leq \beta.$$

Moreover, there are specific disturbances  $d$  that result in the ratio  $\|\tilde{e}\|_2 / \|\tilde{d}\|_2$  arbitrarily close to  $\beta$ . Because of this,  $\|T\|_\infty$  is referred to as the  $L_2$  (or RMS) gain of the system.

As you would expect, a sinusoidal, steady-state interpretation of  $\|T\|_\infty$  is also possible: For any frequency  $\bar{\omega} \in R$ , any vector of amplitudes  $a \in R^{n_d}$ , and any vector of phases  $\phi \in R^{n_d}$ , with  $\|a\|_2 \leq 1$ , define a time signal

$$\tilde{d}(t) = \begin{bmatrix} a_1 \sin(\bar{\omega}t + \phi_1) \\ \vdots \\ a_{n_d} \sin(\bar{\omega}t + \phi_{n_d}) \end{bmatrix}.$$

Applying this input to the system  $T$  results in a steady-state response  $\tilde{e}_{ss}$  of the form

$$\tilde{e}_{ss}(t) = \begin{bmatrix} b_1 \sin(\bar{\omega}t + \phi_1) \\ \vdots \\ b_{n_e} \sin(\bar{\omega}t + \phi_{n_e}) \end{bmatrix}.$$

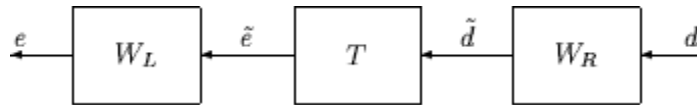
The vector  $b \in \mathbb{R}^{n_e}$  will satisfy  $\|b\|_2 \leq \beta$ . Moreover,  $\beta$ , as defined in Weighted MIMO System on page 5-6, is the smallest number such that this is true for every  $\|a\|_2 \leq 1$ ,  $\bar{\omega}$ , and  $\varphi$ .

Note that in this interpretation, the vectors of the sinusoidal magnitude responses are unweighted, and measured in Euclidean norm. If realistic multivariable performance objectives are to be represented by a single MIMO  $\|\cdot\|_\infty$  objective on a closed-loop transfer function, additional scalings are necessary. Because many different objectives are being lumped into one matrix and the associated cost is the norm of the matrix, it is important to use frequency-dependent weighting functions, so that different requirements can be meaningfully combined into a single cost function. Diagonal weights are most easily interpreted.

Consider the diagram of Weighted MIMO System on page 5-6, along with Unweighted MIMO System: Vectors from Left to Right on page 5-4.

Assume that  $W_L$  and  $W_R$  are diagonal, stable transfer function matrices, with diagonal entries denoted  $L_i$  and  $R_i$ .

$$W_L = \begin{bmatrix} L_1 & 0 & \dots & 0 \\ 0 & L_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & L_{n_e} \end{bmatrix}, \quad W_R = \begin{bmatrix} R_1 & 0 & \dots & 0 \\ 0 & R_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & R_{n_d} \end{bmatrix}.$$



$$e = W_L \tilde{e} = W_L T \tilde{d} = W_L T W_R d$$

### Weighted MIMO System

Bounds on the quantity  $\|W_L T W_R\|_\infty$  will imply bounds about the sinusoidal

steady-state behavior of the signals  $\tilde{d}$  and  $\tilde{e} (= T\tilde{d})$  in the diagram of Unweighted MIMO System: Vectors from Left to Right on page 5-4.

Specifically, for sinusoidal signal  $\tilde{d}$ , the steady-state relationship between

$\tilde{e}(=T\tilde{d})$ ,  $\tilde{d}$  and  $\|W_LTW_R\|_\infty$  is as follows. The steady-state solution  $\tilde{e}_{ss}$ , denoted as

$$\tilde{e}_{ss}(t) = \begin{bmatrix} \tilde{e}_1 \sin(\bar{\omega}t + \phi_1) \\ \vdots \\ \tilde{e}_{n_e} \sin(\bar{\omega}t + \phi_{n_d}) \end{bmatrix} \quad (5-1)$$

satisfies

$$\sum_{i=1}^{n_e} |W_{L_i}(j\bar{\omega})\tilde{e}_i|^2 \leq 1$$

for all sinusoidal input signals  $\tilde{d}$  of the form

$$\tilde{d}(t) = \begin{bmatrix} \tilde{d}_1 \sin(\bar{\omega}t + \phi_1) \\ \vdots \\ \tilde{d}_{n_e} \sin(\bar{\omega}t + \phi_{n_d}) \end{bmatrix} \quad (5-2)$$

satisfying

$$\sum_{i=1}^{n_d} \frac{|\tilde{d}_i|^2}{|W_{R_i}(j\bar{\omega})|^2} \leq 1$$

if and only if  $\|W_LTW_R\|_\infty \leq 1$ .

This approximately (*very* approximately — the next statement is not actually correct) implies that  $\|W_LTW_R\|_\infty \leq 1$  if and only if for every fixed frequency  $\bar{\omega}$ , and all sinusoidal disturbances  $\tilde{d}$  of the form Equation 5-2 satisfying

$$|\tilde{d}_i| \leq |W_{R_i}(j\bar{\omega})|$$

the steady-state error components will satisfy

$$|\tilde{e}_i| \leq \frac{1}{|W_{L_i}(j\bar{\omega})|}.$$

This shows how one could pick performance weights to reflect the desired frequency-dependent performance objective. Use  $W_R$  to represent the relative magnitude of sinusoids disturbances that might be present, and use  $1/W_L$  to represent the desired upper bound on the subsequent errors that are produced.

Remember, however, that the weighted  $H_\infty$  norm does *not* actually give element-by-element bounds on the components of  $\tilde{e}$  based on element-by-element bounds on the components of  $\tilde{d}$ . The precise bound it gives is in terms of Euclidean norms of the components of  $\tilde{e}$  and  $\tilde{d}$  (weighted appropriately by  $W_L(j\bar{\omega})$  and  $W_R(j\bar{\omega})$ ).

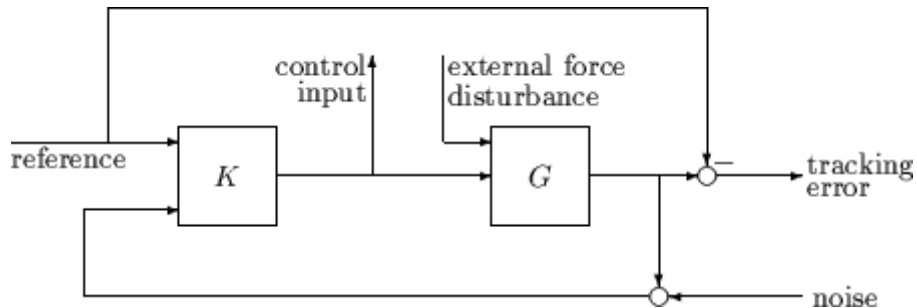


## H-Infinity Performance

### Performance as Generalized Disturbance Rejection

The modern approach to characterizing closed-loop performance objectives is to measure the size of certain closed-loop transfer function matrices using various matrix norms. Matrix norms provide a measure of how large output signals can get for certain classes of input signals. Optimizing these types of performance objectives over the set of stabilizing controllers is the main thrust of recent optimal control theory, such as  $L_1$ ,  $H_2$ ,  $H_\infty$ , and optimal control. Hence, it is important to understand how many types of control objectives can be posed as a minimization of closed-loop transfer functions.

Consider a tracking problem, with disturbance rejection, measurement noise, and control input signal limitations, as shown in Generalized and Weighted Performance Block Diagram on page 5-11.  $K$  is some controller to be designed and  $G$  is the system you want to control.



### Typical Closed-Loop Performance Objective

A reasonable, though not precise, design objective would be to design  $K$  to keep tracking errors and control input signal small for all reasonable reference commands, sensor noises, and external force disturbances.

Hence, a natural performance objective is the closed-loop gain from exogenous influences (reference commands, sensor noise, and external force disturbances) to regulated variables (tracking errors and control input signal). Specifically, let  $T$  denote the closed-loop mapping from the outside influences to the regulated variables:

$$\underbrace{\begin{bmatrix} \text{tracking error} \\ \text{control input} \end{bmatrix}}_{\text{regulated variables}} = T \underbrace{\begin{bmatrix} \text{reference} \\ \text{external force} \\ \text{noise} \end{bmatrix}}_{\text{outside influences}}$$

You can assess performance by measuring the gain from *outside influences* to *regulated variables*. In other words, good performance is associated with  $T$  being small. Because the closed-loop system is a multiinput, multioutput (MIMO) dynamic system, there are two different aspects to the gain of  $T$ :

- Spatial (*vector* disturbances and *vector* errors)
- Temporal (dynamic relationship between input/output signals)

Hence the performance criterion must account for

- Relative magnitude of outside influences
- Frequency dependence of signals
- Relative importance of the magnitudes of regulated variables

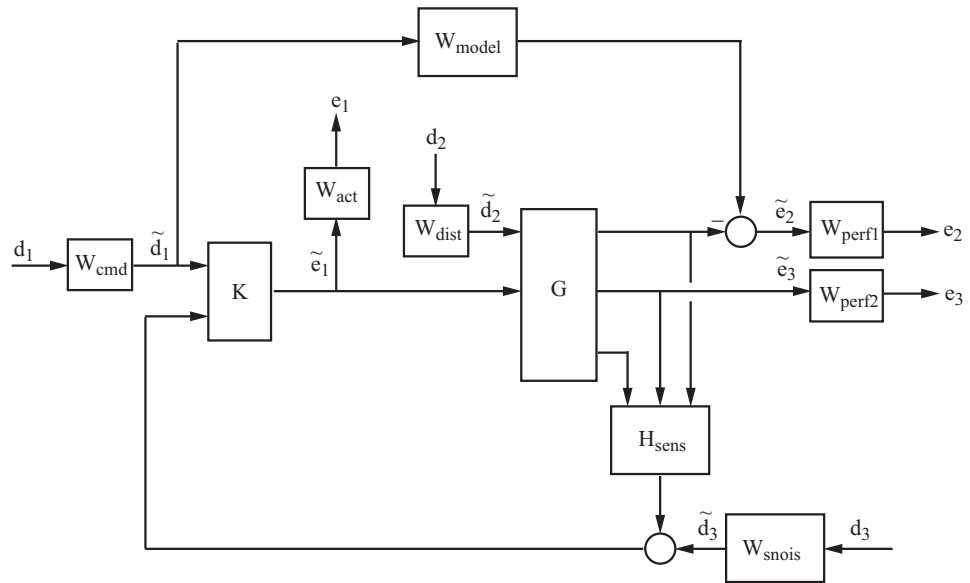
So if the performance objective is in the form of a matrix norm, it should actually be a *weighted norm*

$$\|W_L T W_R\|$$

where the weighting function matrices  $W_L$  and  $W_R$  are frequency dependent, to account for bandwidth constraints and spectral content of exogenous signals. A natural (mathematical) manner to characterize acceptable performance is in terms of the MIMO  $\|\cdot\|_\infty$  ( $H_\infty$ ) norm. See “Interpretation of H-Infinity Norm” on page 5-2 for an interpretation of the  $H_\infty$  norm and signals.

### Interconnection with Typical MIMO Performance Objectives

The closed-loop performance objectives are formulated as weighted closed-loop transfer functions that are to be made small through feedback. A generic example, which includes many relevant terms, is shown in block diagram form in Generalized and Weighted Performance Block Diagram on page 5-11. In the diagram,  $G$  denotes the plant model and  $K$  is the feedback controller.



### Generalized and Weighted Performance Block Diagram

The blocks in this figure might be scalar (SISO) and/or multivariable (MIMO), depending on the specific example. The mathematical objective of  $H_\infty$  control is to make the closed-loop MIMO transfer function  $T_{ed}$  satisfy  $\|T_{ed}\|_\infty < 1$ . The weighting functions are used to scale the input/output transfer functions such that when  $\|T_{ed}\|_\infty < 1$ , the relationship between  $\tilde{d}$  and  $\tilde{e}$  is suitable.

Performance requirements on the closed-loop system are transformed into the  $H_\infty$  framework with the help of *weighting* or *scaling* functions. Weights are selected to account for the relative magnitude of signals, their frequency dependence, and their relative importance. This is captured in the figure above, where the weights or scalings  $[W_{cmd}, W_{dist}, W_{snois}]$  are used to transform and scale the normalized input signals  $[d_1, d_2, d_3]$  into physical units defined as  $[d_1, d_2, d_3]$ . Similarly weights or scalings  $[W_{act}, W_{perf1}, W_{perf2}]$  transform and scale physical units into normalized output signals  $[e_1, e_2, e_3]$ . An interpretation of the signals, weighting functions, and models follows.

<b>Signal</b>	<b>Meaning</b>
$d_1$	Normalized reference command
$\tilde{d}_1$	Typical reference command in physical units
$d_2$	Normalized exogenous disturbances
$\tilde{d}_2$	Typical exogenous disturbances in physical units
$d_3$	Normalized sensor noise
$\tilde{d}_3$	Typical sensor noise in physical units
$e_1$	Weighted control signals
$\tilde{e}_1$	Actual control signals in physical units
$e_2$	Weighted tracking errors
$\tilde{e}_2$	Actual tracking errors in physical units
$e_3$	Weighted plant errors
$\tilde{e}_3$	Actual plant errors in physical units

**$W_{cmd}$**

$W_{cmd}$  is included in  $H_\infty$  control problems that require tracking of a reference command.  $W_{cmd}$  shapes the normalized reference command signals (magnitude and frequency) into the actual (or typical) reference signals that you expect to occur. It describes the magnitude and the frequency dependence of the reference commands generated by the normalized reference signal. Normally  $W_{cmd}$  is flat at low frequency and rolls off at high frequency. For example, in a flight control problem, fighter pilots generate stick input reference commands up to a bandwidth of about 2 Hz. Suppose that the stick has a maximum travel of three inches. Pilot commands could be modeled as normalized signals passed through a first-order filter:

$$W_{cmd} = \frac{3}{\frac{1}{2 \cdot 2\pi} s + 1}.$$

**$W_{model}$** 

$W_{model}$  represents a desired ideal model for the closed-loop system and is often included in problem formulations with tracking requirements. Inclusion of an ideal model for tracking is often called a *model matching* problem, i.e., the objective of the closed-loop system is to match the defined model. For good command tracking response, you might want the closed-loop system to respond like a well-damped second-order system. The ideal model would then be

$$W_{model} = \frac{\omega^2}{s^2 + 2\zeta\omega + \omega^2}$$

for specific desired natural frequency  $\omega$  and desired damping ratio  $\zeta$ . Unit conversions might be necessary to ensure exact correlation between the ideal model and the closed-loop system. In the fighter pilot example, suppose that roll-rate is being commanded and  $10^\circ/\text{second}$  response is desired for each inch of stick motion. Then, in these units, the appropriate model is:

$$W_{model} = 10 \frac{\omega^2}{s^2 + 2\zeta\omega + \omega^2}.$$

 **$W_{dist}$** 

$W_{dist}$  shapes the frequency content and magnitude of the exogenous disturbances affecting the plant. For example, consider an electron microscope as the plant. The dominant performance objective is to mechanically isolate the microscope from outside mechanical disturbances, such as ground excitations, sound (pressure) waves, and air currents. You can capture the spectrum and relative magnitudes of these disturbances with the transfer function weighting matrix  $W_{dist}$ .

 **$W_{perf1}$** 

$W_{perf1}$  weights the difference between the response of the closed-loop system and the ideal model  $W_{model}$ . Often you might want accurate matching of the ideal model at low frequency and require less accurate matching at higher frequency, in which case  $W_{perf1}$  is flat at low frequency, rolls off at first or

second order, and flattens out at a small, nonzero value at high frequency. The inverse of the weight is related to the allowable size of tracking errors, when dealing with the reference commands and disturbances described by  $W_{cmd}$  and  $W_{dist}$ .

### **$W_{perf2}$**

$W_{perf2}$  penalizes variables internal to the process  $G$ , such as actuator states that are internal to  $G$  or other variables that are not part of the tracking objective.

### **$W_{act}$**

$W_{act}$  is used to shape the penalty on control signal use.  $W_{act}$  is a frequency varying weighting function used to penalize limits on the deflection/position, deflection rate/velocity, etc., response of the control signals, when dealing with the tracking and disturbance rejection objectives defined above. Each control signal is usually penalized independently.

### **$W_{snois}$**

$W_{snois}$  represents frequency domain models of sensor noise. Each sensor measurement feedback to the controller has some noise, which is often higher in one frequency range than another. The  $W_{snois}$  weight tries to capture this information, derived from laboratory experiments or based on manufacturer measurements, in the control problem. For example, medium grade accelerometers have substantial noise at low frequency and high frequency. Therefore the corresponding  $W_{snois}$  weight would be larger at low and high frequency and have a smaller magnitude in the mid-frequency range. Displacement or rotation measurement is often quite accurate at low frequency and in steady state, but responds poorly as frequency increases. The weighting function for this sensor would be small at low frequency, gradually increase in magnitude as a first- or second-order system, and level out at high frequency.

### **$H_{sens}$**

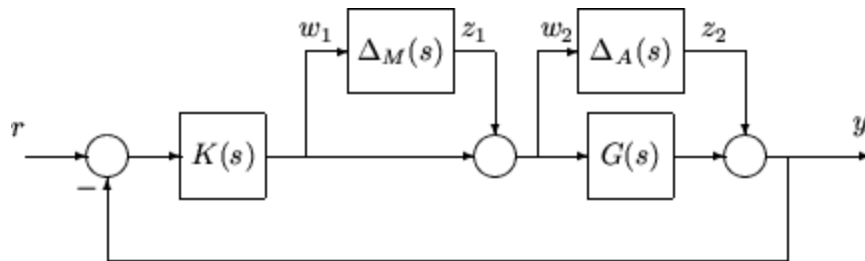
$H_{sens}$  represents a model of the sensor dynamics or an external antialiasing filter. The transfer functions used to describe  $H_{sens}$  are based on physical

characteristics of the individual components. These models might also be lumped into the plant model  $G$ .

This generic block diagram has tremendous flexibility and many control performance objectives can be formulated in the  $H_\infty$  framework using this block diagram description.

## Robustness in the H-Infinity Framework

Performance and robustness tradeoffs in control design were discussed in the context of multivariable loop shaping in “Tradeoff Between Performance and Robustness” on page 2-2. In the  $H_\infty$  control design framework, you can include robustness objectives as additional disturbance to error transfer functions — disturbances to be kept small. Consider the following figure of a closed-loop feedback system with additive and multiplicative uncertainty models.

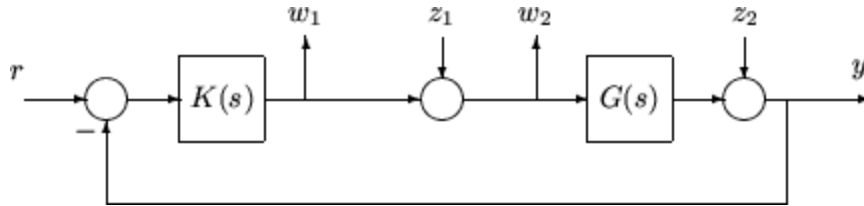


The transfer function matrices are defined as:

$$TF(s)_{z_1 \rightarrow w_1} = T_I(s) = KG(I + GK)^{-1}$$

$$TF(s)_{z_2 \rightarrow w_2} = KS_O(s) = K(I + GK)^{-1}$$

where  $T_I(s)$  denotes the input complementary sensitivity function and  $S_O(s)$  denotes the output sensitivity function. Bounds on the size of the transfer function matrices from  $z_1$  to  $w_1$  and  $z_2$  to  $w_2$  ensure that the closed-loop system is robust to multiplicative uncertainty,  $\Delta_M(s)$ , at the plant input, and additive uncertainty,  $\Delta_A(s)$ , around the plant  $G(s)$ . In the  $H_\infty$  control problem formulation, the robustness objectives enter the synthesis procedure as additional input/output signals to be kept small. The interconnection with the uncertainty blocks removed follows.



The  $H_\infty$  control robustness objective is now in the same format as the performance objectives, that is, to minimize the  $H_\infty$  norm of the transfer matrix from  $z$ ,  $[z_1, z_2]$ , to  $w$ ,  $[w_1, w_2]$ .

Weighting or scaling matrices are often introduced to shape the frequency and magnitude content of the sensitivity and complementary sensitivity transfer function matrices. Let  $W_M$  correspond to the multiplicative uncertainty and  $W_A$  correspond to the additive uncertainty model.  $\Delta_M(s)$  and  $\Delta_A(s)$  are assumed to be a norm bounded by 1, i.e.,  $|\Delta_M(s)| < 1$  and  $|\Delta_A(s)| < 1$ . Hence as a function of frequency,  $|W_M(j\omega)|$  and  $|W_A(j\omega)|$  are the respective sizes of the largest anticipated additive and multiplicative plant perturbations.

The multiplicative weighting or scaling  $W_M$  represents a percentage error in the model and is often small in magnitude at low frequency, between 0.05 and 0.20 (5% to 20% modeling error), and growing larger in magnitude at high frequency, 2 to 5 ((200% to 500% modeling error). The weight will transition by crossing a magnitude value of 1, which corresponds to 100% uncertainty in the model, at a frequency at least twice the bandwidth of the closed-loop system. A typical multiplicative weight is

$$W_M = 0.10 \frac{\frac{1}{5}s + 1}{\frac{1}{200}s + 1}.$$

By contrast, the additive weight or scaling  $W_A$  represents an absolute error that is often small at low frequency and large in magnitude at high frequency. The magnitude of this weight depends directly on the magnitude of the plant model,  $G(s)$ .



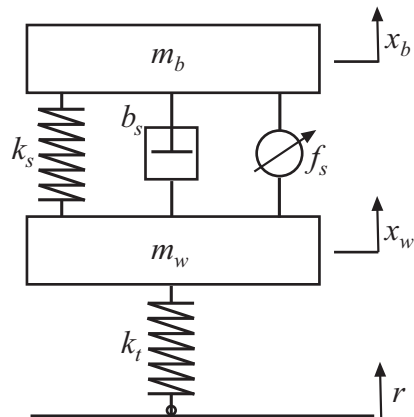
## Active Suspension Control Design

This example shows how to use robust control techniques to design an active suspension system for a quarter car body and wheel assembly model. In this example, you use  $H_\infty$  design techniques to design a controller for a nominal quarter-car model. Then, you use  $\mu$  synthesis to design a robust controller that accounts for uncertainty in the model.

Conventional *passive suspensions* use a spring and damper between the car body and wheel assembly. The spring-damper characteristics are adjusted to emphasize one of several conflicting objectives such as passenger comfort, road holding, and suspension deflection. *Active suspensions* use a feedback-controller hydraulic actuator between the chassis and wheel assembly, which allows the designer to better balance these objectives.

### Quarter-Car Suspension Model

This example uses the quarter-car model of the following illustration to design active suspension control laws.



The mass,  $m_b$ , represents the car chassis (body) and the mass,  $m_w$ , represents the wheel assembly. The spring,  $k_s$ , and damper,  $b_s$ , represent the passive

spring and shock absorber placed between the car body and the wheel assembly. The spring,  $k_t$ , models the compressibility of the pneumatic tire. The variables  $x_b$ ,  $x_w$ , and  $r$  are the car body travel, the wheel travel, and the road disturbance, respectively. The force,  $f_s$ , which is applied between the body and wheel assembly, is controlled by feedback. This force represents the active component of the suspension system.

The following state-space equations describe the quarter-car dynamics.

$$\begin{aligned}\dot{x}_1 &= x_2, \\ \dot{x}_2 &= -\frac{1}{m_b} [k_s(x_1 - x_3) + b_s(x_2 - x_4) - f_s], \\ \dot{x}_3 &= x_4, \\ \dot{x}_4 &= \frac{1}{m_w} [k_s(x_1 - x_3) + b_s(x_2 - x_4) - k_t(x_3 - r) - f_s].\end{aligned}$$

The state variables in the system are defined as  $x_1 := x_b$ ,  $x_2 := \dot{x}_b$ ,  $x_3 := x_w$ , and  $x_4 := \dot{x}_w$ .

Define the physical parameters of the system.

```
mb = 300;      % kg
mw = 60;      % kg
bs = 1000;    % N/m/s
ks = 16000;   % N/m
kt = 190000;  % N/m
```

Use these equations and parameter values to construct a state-space model, `qcar`, of the quarter-car suspension system.

```
A = [ 0 1 0 0; [-ks -bs ks bs]/mb ; ...
      0 0 0 1; [ks bs -ks-kt -bs]/mw];
B = [0 0; 0 10000/mb ; 0 0; [kt -10000]/mw];
C = [1 0 0 0; 1 0 -1 0; A(2,:)];
D = [0 0; 0 0; B(2,:)];

qcar = ss(A,B,C,D);
qcar.StateName = {'body travel xb (m)'; 'body vel (m/s)'; ...
```

```

        'wheel travel xw (m)'; 'wheel vel (m/s)'};
qcar.InputName = {'r'; 'fs'};
qcar.OutputName = {'xb'; 'sd'; 'ab'};

```

The model inputs are the road disturbance,  $r$ , and actuator force,  $f_s$ . The model outputs are the car body travel,  $x_b$ , suspension deflection  $s_d = x_b - x_w$ , and car body acceleration  $a_b = \ddot{x}_s$ .

The transfer function from actuator to body travel and acceleration has an imaginary-axis zero. Examine the zero of this transfer function.

```
tzero(qcar({'xb', 'ab'}, 'fs'))
```

```

ans =

    -0.0000 +56.2731i
    -0.0000 -56.2731i

```

The natural frequency of this zero, 56.27 rad/s, is called the *tire-hop frequency*.

The transfer function from the actuator to suspension deflection also has an imaginary-axis zero. Examine this zero.

```
zero(qcar('sd', 'fs'))
```

```

ans =

    0.0000 +22.9734i
    0.0000 -22.9734i

```

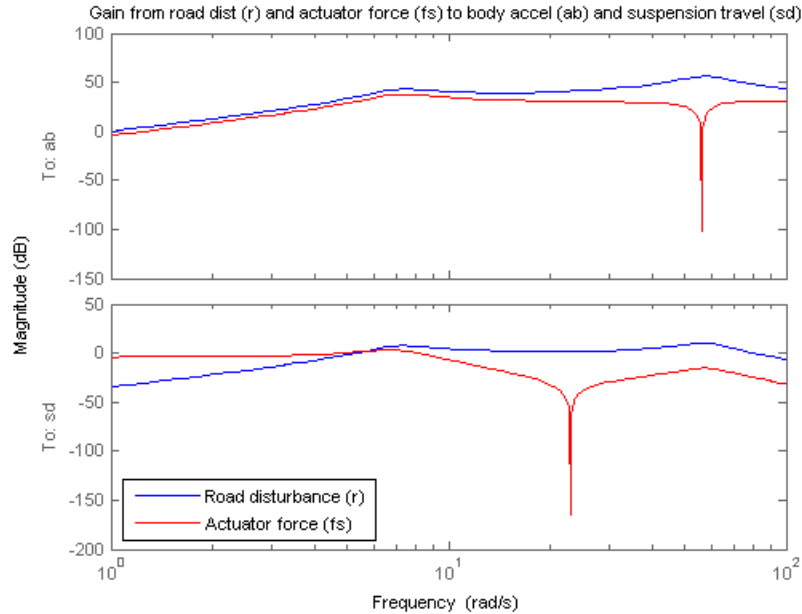
The natural frequency of this zero, 22.97 rad/s, is called the *rattlespace frequency*.

Plot the frequency response of the quarter-car model from inputs,  $(r, f_s)$ , to outputs,  $(a_b, s_d)$ . Both zeros are visible on the Bode plot.

```

bodemag(qcar({'ab', 'sd'}, 'r'), 'b', qcar({'ab', 'sd'}, 'fs'), 'r', {1 100});
legend('Road disturbance (r)', 'Actuator force (fs)', 'location', 'SouthWest')
title(['Gain from road dist (r) and actuator force (fs) '...
        'to body accel (ab) and suspension travel (sd)'])

```



Road disturbances influence the motion of the car and suspension:

- Small body acceleration influences passenger comfort.
- Small suspension travel contributes to good road handling. Further, limits on the actuator displacement constrain the allowable travel.

Because of the imaginary axis zeros, feedback control cannot improve the response from road disturbance ( $r$ ) to body acceleration ( $a_b$ ) at the tire-hop frequency. Similarly, feedback control cannot improve the response from  $r$  to suspension deflection ( $s_d$ ) at the rattlespace frequency. Moreover, there is an inherent trade-off between passenger comfort and suspension deflection. Any reduction of body travel at low frequency results in an increase of suspension deflection. This trade-off arises because of the relationship  $x_w = x_b - s_d$  and the fact that  $x_w$  roughly follows  $r$  at low frequency (less than 5 rad/s).

### Hydraulic Actuator Model

The hydraulic actuator used for active suspension control is connected between the body mass,  $m_b$ , and the wheel assembly mass,  $m_w$ . The nominal actuator dynamics can be represented by the first-order transfer function:

$$ActNom(s) = \frac{1}{\frac{1}{60}s + 1}.$$

The maximum displacement is 0.05 m.

The nominal actuator model approximates the physical actuator dynamics. You can model variations between the actuator model and the physical device as a family of actuator models. You can also use this approach to model variations between the passive quarter-car model and actual vehicle dynamics. The resulting family of models comprises a nominal model with a frequency-dependent amount of uncertainty.

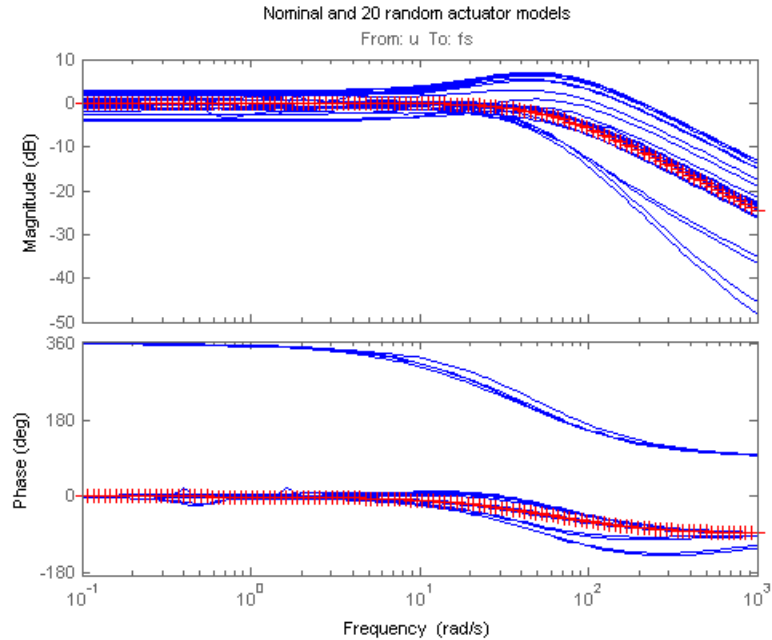
Create an uncertain model that represents this family of models.

```
ActNom = tf(1,[1/60 1]);
Wunc = makeweight(0.40,15,3);
unc = ultidyn('unc',[1 1],'SampleStateDim',5);
Act = ActNom*(1 + Wunc*unc);
Act.InputName = 'u';
Act.OutputName = 'fs';
```

At low frequency, below 3 rad/s, the model can vary up to 40% from its nominal value. Around 3 rad/s, the percentage variation starts to increase. The uncertainty crosses 100% at 15 rad/s, and reaches 2000% at approximately 1000 rad/sec. The weighting function, Wunc, reflects this profile and is used to modulate the amount of uncertainty as a function of frequency. The result Act is an uncertain state-space model of the actuator.

Examine the uncertain actuator model by plotting the frequency response of 20 randomly sampled models from Act.

```
bode(Act,'b',Act.NominalValue,'r+',logspace(-1,3,120))
title('Nominal and 20 random actuator models')
```



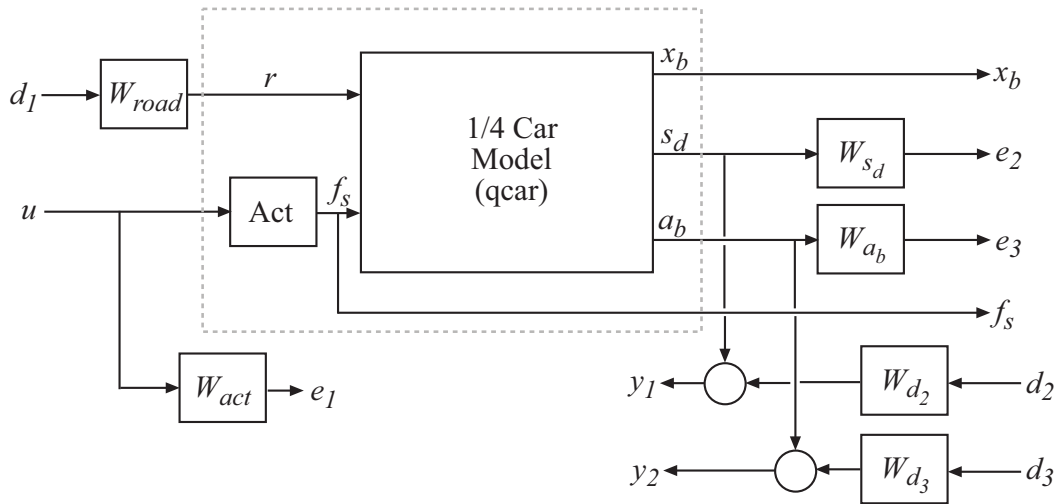
The plus (+) marker denotes the nominal actuator model. The blue solid lines represent the randomly sampled models.

### Design Objectives for H-Infinity Synthesis

To use  $H_\infty$  synthesis algorithms, you must express your design objectives as a single cost function to be minimized. For the quarter-car model, the main control objectives are formulated in terms of passenger comfort and road handling. These objectives relate to body acceleration,  $a_b$ , and suspension travel,  $s_d$ . Other factors that influence the control design include:

- Characteristics of the road disturbance
- Quality of the sensor measurements for feedback
- Limits on the available control force

Use weights to model external disturbances and quantify the design objectives, as shown in the following diagram.



The feedback controller uses the measurements  $y_1$  and  $y_2$  of the suspension travel,  $s_d$ , and body acceleration,  $a_b$ , to compute the control signal,  $u$ . This control signal drives the hydraulic actuator. There are three external sources of disturbance:

- The road disturbance,  $r$ , which is modeled as a normalized signal,  $d_1$ , which is shaped by a weighting function  $W_{road}$ .
- Sensor noise on both measurements. This noise is modeled as normalized signals,  $d_2$  and  $d_3$ , which are shaped by weighting functions  $W_{d_2}$  and  $W_{d_3}$ .

You can reinterpret the control objectives as a disturbance rejection goal. The goal is to minimize the impact of the disturbances,  $d_1$ ,  $d_2$ , and  $d_3$ , on a weighted combination of suspension travel ( $s_d$ ), body acceleration ( $a_b$ ), and control effort ( $u$ ). You can consider the  $H_\infty$  norm (peak gain) as the measure of the effect of the disturbances. Then, you can meet the requirements by designing a controller that minimizes the  $H_\infty$  norm from the disturbance inputs,  $d_1$ ,  $d_2$ , and  $d_3$ , to the error signals,  $e_1$ ,  $e_2$ , and  $e_3$ .

Create the weighting functions that model the design objectives.

```
Wroad = ss(0.07);
Wroad.u = 'd1';
Wroad.y = 'r';
```

```
Wact = 8*tf([1 50],[1 500]);
Wact.u = 'u';
Wact.y = 'e1';

Wd2 = ss(0.01);
Wd2.u = 'd2';
Wd2.y = 'Wd2';

Wd3 = ss(0.5);
Wd3.u = 'd3';
Wd3.y = 'Wd3';
```

The constant weight  $W_{road} = 0.07$  models broadband road deflections of magnitude 7 cm.  $W_{act}$  is a highpass filter. This filter penalizes high-frequency content in the control signal, and thus limits control bandwidth.  $W_{d2}$  and  $W_{d3}$  model broadband sensor noise of intensity 0.01 and 0.5, respectively. In a more realistic design,  $W_{d2}$  and  $W_{d3}$  would be frequency dependent to model the noise spectrum of the displacement and acceleration sensors. The inputs and outputs of all weighting functions are named to facilitate interconnection. The notation  $u$  and  $y$  are shorthand for the `InputName` and `OutputName` properties, respectively.

Specify target functions for the closed-loop response of the system from the road disturbance,  $r$ , to the suspension deflection,  $s_d$ , and body acceleration,  $a_b$ .

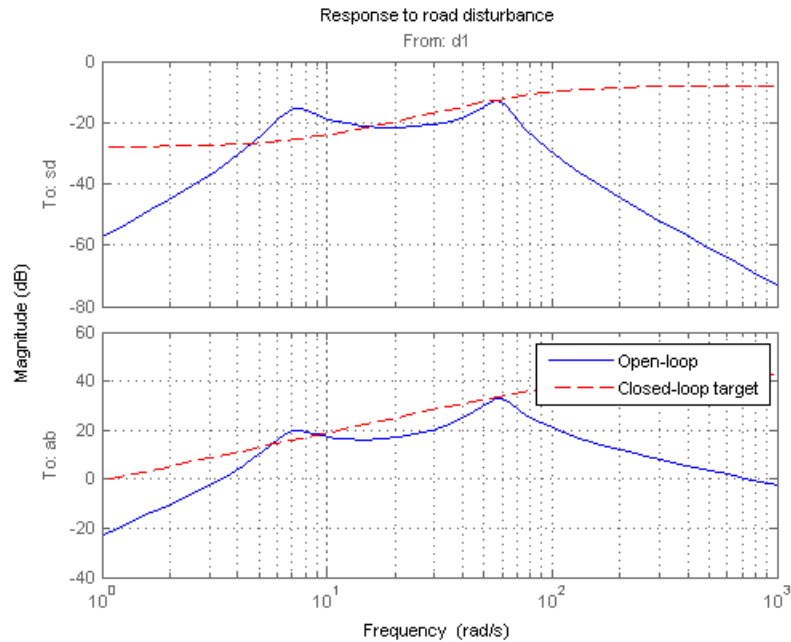
```
HandlingTarget = 0.04 * tf([1/8 1],[1/80 1]);
ComfortTarget = 0.4 * tf([1/0.45 1],[1/150 1]);
Targets = [HandlingTarget;ComfortTarget];
```

Because of the actuator uncertainty and imaginary-axis zeros, the targets attenuate disturbances only below 10 rad/s. These targets represent the goals of passenger comfort (small car body acceleration) and adequate road handling (small suspension deflection).

Plot the closed-loop targets and compare to the open-loop response.

```
bodemag(qcar({'sd','ab'},'r')*Wroad,'b',Targets,'r--',{1,1000})
grid, title('Response to road disturbance')
legend('Open-loop','Closed-loop target')
```





The corresponding performance weights  $W_{sd}$  and  $W_{ab}$  are the reciprocals of the comfort and handling targets. To investigate the trade-off between passenger comfort and road handling, construct three sets of weights,  $(\beta W_{sd}, (1 - \beta) W_{ab})$ . These weights use a blending parameter,  $\beta$ , to modulate the trade-off.

```
beta = reshape([0.01 0.5 0.99],[1 1 3]);
```

```
Wsd = beta/HandlingTarget;  
Wsd.u = 'sd';  
Wsd.y = 'e3';
```

```
Wab = (1-beta)/ComfortTarget;  
Wab.u = 'ab';  
Wab.y = 'e2';
```

$Wsd$  and  $Wab$  are arrays of weighting functions that correspond to three different trade-offs: emphasizing comfort ( $\beta = 0.01$ ), balancing comfort and handling ( $\beta = 0.5$ ), and emphasizing handling ( $\beta = 0.99$ ).

Connect the quarter-car plant model, actuator model, and weighting functions to construct the block diagram of the plant model weighted by the objectives.

```
sdmeas = sumblk('y1 = sd+Wd2');
abmeas = sumblk('y2 = ab+Wd3');
ICinputs = {'d1'; 'd2'; 'd3'; 'u'};
ICoutputs = {'e1'; 'e2'; 'e3'; 'y1'; 'y2'};
qcaric = connect(qcar(2:3,:), Act, Wroad, Wact, Wab, Wsd, Wd2, Wd3, ...
                sdmeas, abmeas, ICinputs, ICoutputs);
```

qcaric is an array of three models, one for each value of the blending parameter,  $\beta$ . Also, the models in qcaric are uncertain, because they contain the uncertain actuator model Act.

### Nominal H-Infinity Synthesis

Use hinfsyn to compute an  $H_\infty$  controller for each value of the blending parameter,  $\beta$ . hinfsyn ignores the uncertainty in the plant models and synthesizes a controller for the nominal value of each model.

```
ncont = 1;
nmeas = 2;
K = ss(zeros(ncont, nmeas, 3));
gamma = zeros(3, 1);
for i=1:3
    [K(:, :, i), ~, gamma(i)] = hinfsyn(qcaric(:, :, i), nmeas, ncont);
end
```

The weighted plant model has one control input (ncont), the hydraulic actuator force. The model also has two measurement outputs (nmeas), which give the suspension deflection and body acceleration.

Examine the resulting closed-loop  $H_\infty$  norms, gamma.

```
gamma

gamma =

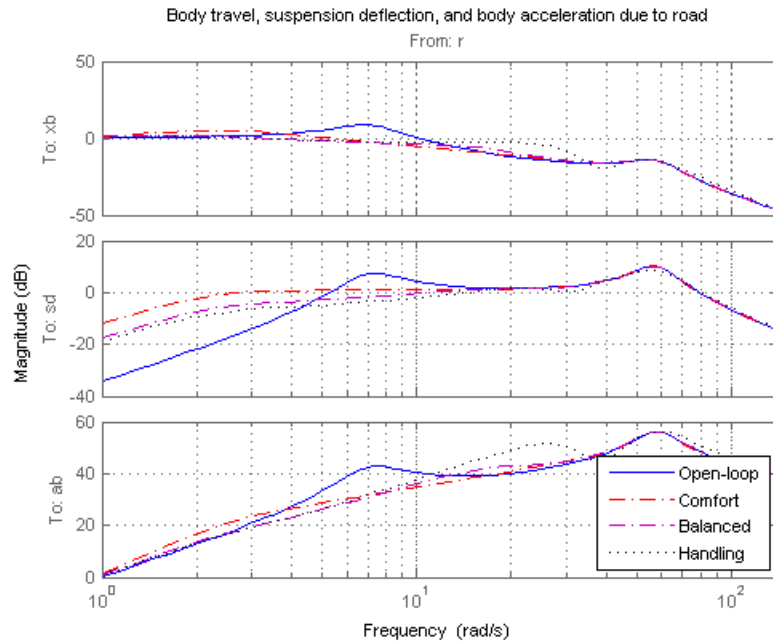
    0.9410
    0.6724
    0.8877
```

The three  $H_\infty$  controllers achieve closed-loop  $H_\infty$  norms of 0.94 (emphasizing comfort), 0.67 (balancing comfort and handling), and 0.89 (emphasizing handling).

Construct closed-loop models of the quarter-car plant with the synthesized controller, corresponding to each of the three blending parameter values. Compare the frequency response from the road disturbance to  $x_b$ ,  $s_d$ , and  $a_b$  for the passive and active suspensions.

```
K.u = {'sd','ab'}; K.y = 'u';
CL = connect(qcar,Act.Nominal,K,'r',{'xb','sd','ab'});

clf
bodemag(qcar(:, 'r'), 'b', CL(:, :, 1), 'r-', ...
        CL(:, :, 2), 'm-', CL(:, :, 3), 'k:', {1,140})
grid
legend('Open-loop', 'Comfort', 'Balanced', 'Handling', 'location', 'SouthEast')
title('Body travel, suspension deflection, and body acceleration due to road')
```



The solid blue line corresponds to the open-loop response. The other lines are the closed-loop frequency responses for the different comfort and handling blends. All three controllers reduce suspension deflection and body acceleration below the rattlespace frequency (23 rad/s).

**Time-Domain Evaluation**

To further evaluate the three designs, perform time-domain simulations using the following road disturbance signal  $r(t)$ :

$$r(t) = \begin{cases} \alpha(1 - \cos 8\pi t), & 0 \leq t \leq 0.25 \\ 0, & \text{otherwise.} \end{cases}$$

This signal corresponds to a road bump of height 5 cm.

Create a vector that represents the road disturbance.

```
t = 0:0.005:1;
roaddist = zeros(size(t));
roaddist(1:51) = 0.025*(1-cos(8*pi*t(1:51)));
```

Build the closed-loop model using the synthesized controller.

```
SIMK = connect(qcar,Act.Nominal,K,'r',{ 'xb'; 'sd'; 'ab'; 'fs' });
```

SIMK is a model array containing three closed-loop models, one for each of the three blending parameter values. Each model in the array represents a closed loop that is built from the original quarter-car plant model, the nominal actuator model, and the corresponding synthesized controller.

Simulate and plot the time-domain response of the closed-loop models to the road disturbance signal.

```
p1 = lsim(qcar(:,1),roaddist,t);
y1 = lsim(SIMK(1:4,1,1),roaddist,t);
y2 = lsim(SIMK(1:4,1,2),roaddist,t);
y3 = lsim(SIMK(1:4,1,3),roaddist,t);

clf
subplot(221)
plot(t,p1(:,1),'b',t,y1(:,1),'r.',t,y2(:,1),'m.',t,y3(:,1),'k.',t,roaddist,'g')
title('Body travel')
ylabel('x_b (m)')

subplot(222)
plot(t,p1(:,3),'b',t,y1(:,3),'r.',t,y2(:,3),'m.',t,y3(:,3),'k.',t,roaddist,'g')
title('Body acceleration')
ylabel('a_b (m/s^2)')

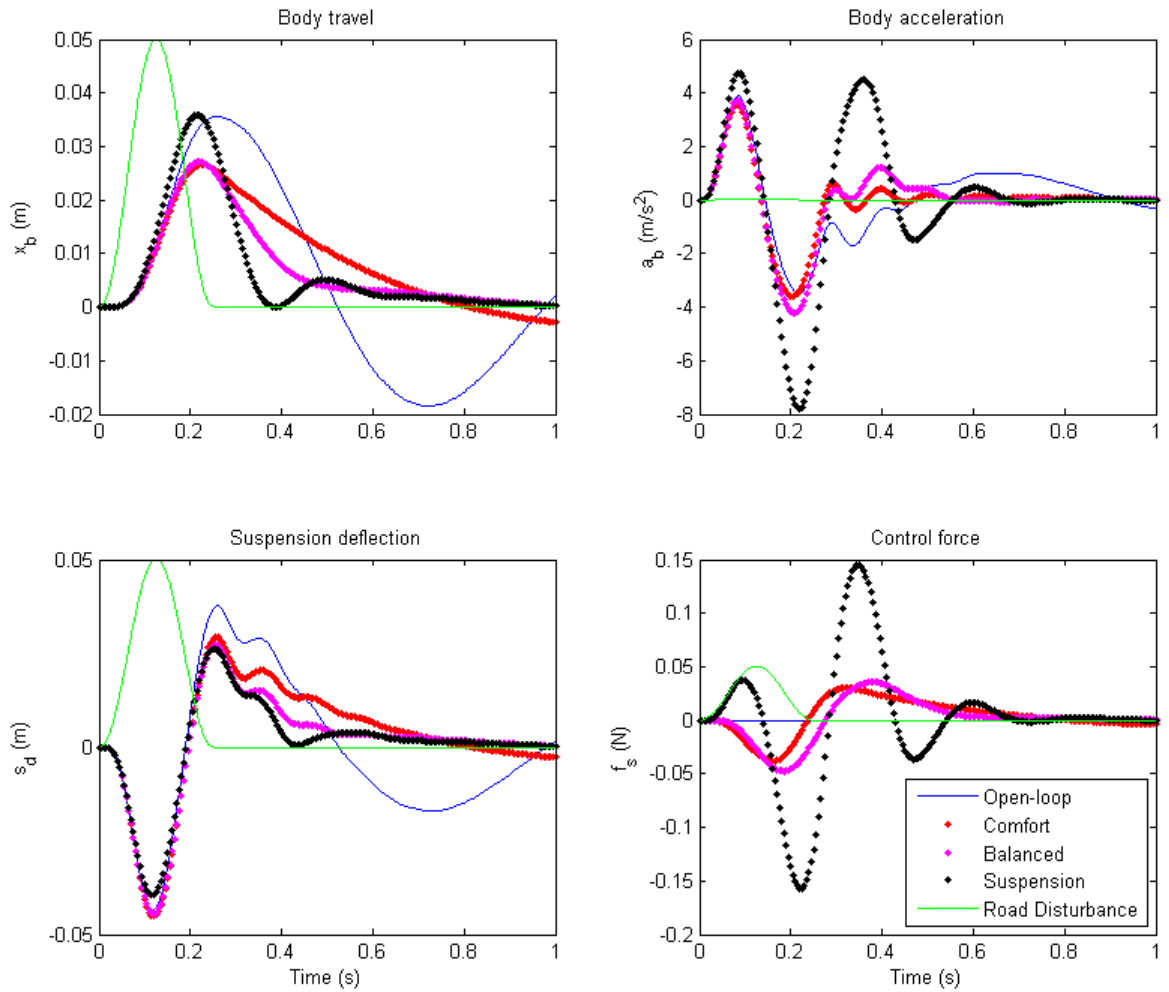
subplot(223)
plot(t,p1(:,2),'b',t,y1(:,2),'r.',t,y2(:,2),'m.',t,y3(:,2),'k.',t,roaddist,'g')
title('Suspension deflection')
xlabel('Time (s)')
ylabel('s_d (m)')

subplot(224)
plot(t,zeros(size(t)),'b',t,y1(:,4),'r.',t,y2(:,4),'m.',t,y3(:,4),'k.',t,roaddist,'g')
title('Control force')
```

```
xlabel('Time (s)')
```

```
ylabel('f_s (N)')
```

```
legend('Open-loop', 'Comfort', 'Balanced', 'Suspension', 'Road Disturbance', 'location', 'SouthEast')
```



The simulations show that the body acceleration is smallest for the controller emphasizing passenger comfort. Body acceleration is largest for the controller emphasizing suspension deflection. The balanced design achieves a good tradeoff between body acceleration and suspension deflection.

### Robust $\mu$ Design

So far you designed  $H_\infty$  controllers that meet the performance objectives for the nominal actuator model. However, this model is only an approximation of the true actuator. To make sure that controller performance is maintained even with model error and uncertainty, you must design the model to have *robust performance*. In this part of the example, you use  $\mu$ -synthesis to design a controller that achieves robust performance for the entire family of actuator models that takes uncertainty into account.

Use D-K iteration to synthesize a controller for the quarter-car model with actuator uncertainty.

```
[Krob,~,RPmuval] = dksyn(qcaric(:, :, 2), nmeas, ncont);
```

The model `qcaric(:, :, 2)` is the weighted quarter-car model for the uncertain model that corresponds to the blending variable  $\beta = 0.5$ .

Examine the resulting  $\mu$ -synthesis controller.

```
size(Krob)
```

State-space model with 1 outputs, 2 inputs, and 11 states.

Build the closed-loop model using the robust controller, `Krob`.

```
Krob.u = {'sd', 'ab'};
Krob.y = 'u';
SIMKrob = connect(qcar, Act.Nominal, Krob, 'r', {'xb'; 'sd'; 'ab'; 'fs'});
```

Simulate and plot the nominal time-domain response to a road bump with the robust controller.

```
p1 = lsim(qcar(:, 1), roaddist, t);
y1 = lsim(SIMKrob(1:4, 1), roaddist, t);
```

```
clf
```

```
subplot(221)
plot(t,p1(:,1),'b',t,y1(:,1),'r',t,roaddist,'g')
title('Body travel'), ylabel('x_b (m)')

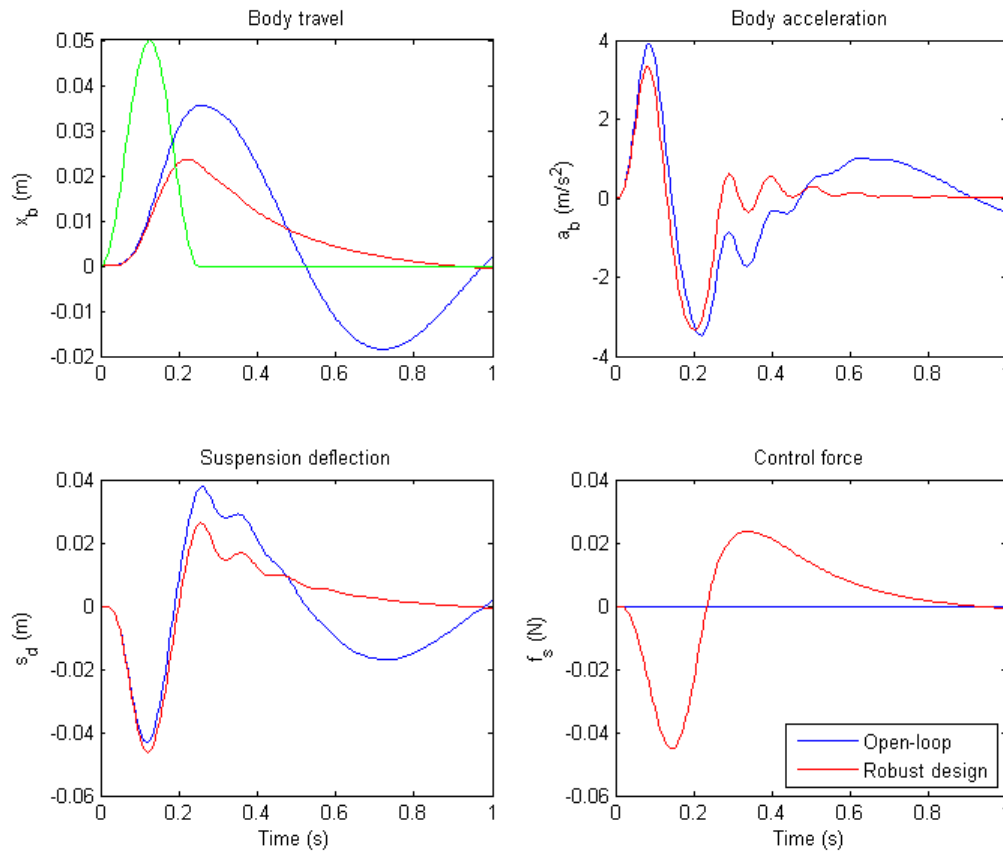
subplot(222)
plot(t,p1(:,3),'b',t,y1(:,3),'r')
title('Body acceleration'), ylabel('a_b (m/s^2)')

subplot(223)
plot(t,p1(:,2),'b',t,y1(:,2),'r')
title('Suspension deflection'), xlabel('Time (s)'), ylabel('s_d (m)')

subplot(224)
plot(t,zeros(size(t)),'b',t,y1(:,4),'r')
title('Control force'), xlabel('Time (s)'), ylabel('f_s (N)')

legend('Open-loop','Robust design','location','SouthEast')
```





These responses are similar to those obtained with the balanced  $H_\infty$  controller.

Examine the effect of the robust controller on variability caused by model uncertainty. To do so, simulate the response to a road bump for 120 actuator models randomly sampled from the uncertain model, Act. Perform this simulation for both the  $H_\infty$  and the robust controllers, to compare the results.

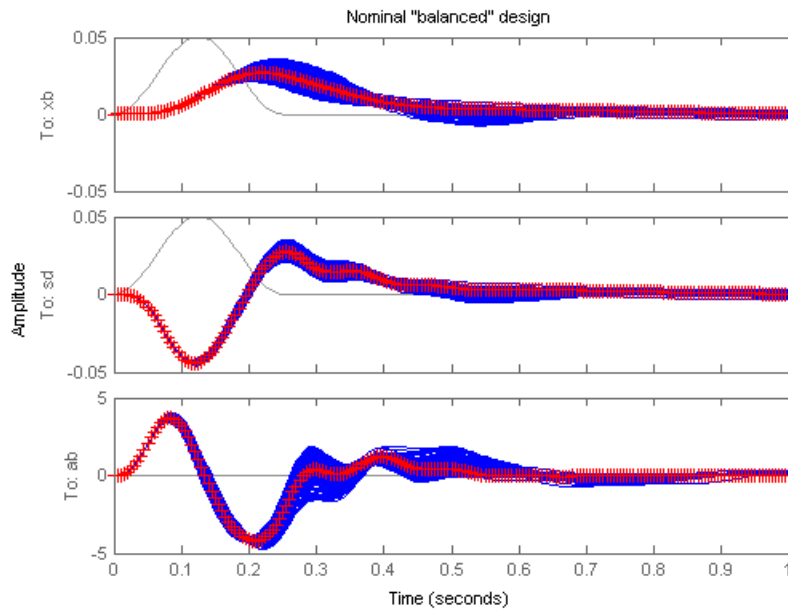
Compute an uncertain closed-loop model with the balanced  $H_\infty$  controller, K. Sample this model, simulate the sampled models, and plot the results.

```

CLU = connect(qcar,Act,K(:, :, 2), 'r', {'xb', 'sd', 'ab'});

nsamp = 120;
rng('default');
figure(1)
clf
lsim(usample(CLU,nsamp), 'b', CLU.Nominal, 'r+', roaddist, t)
title('Nominal "balanced" design')

```

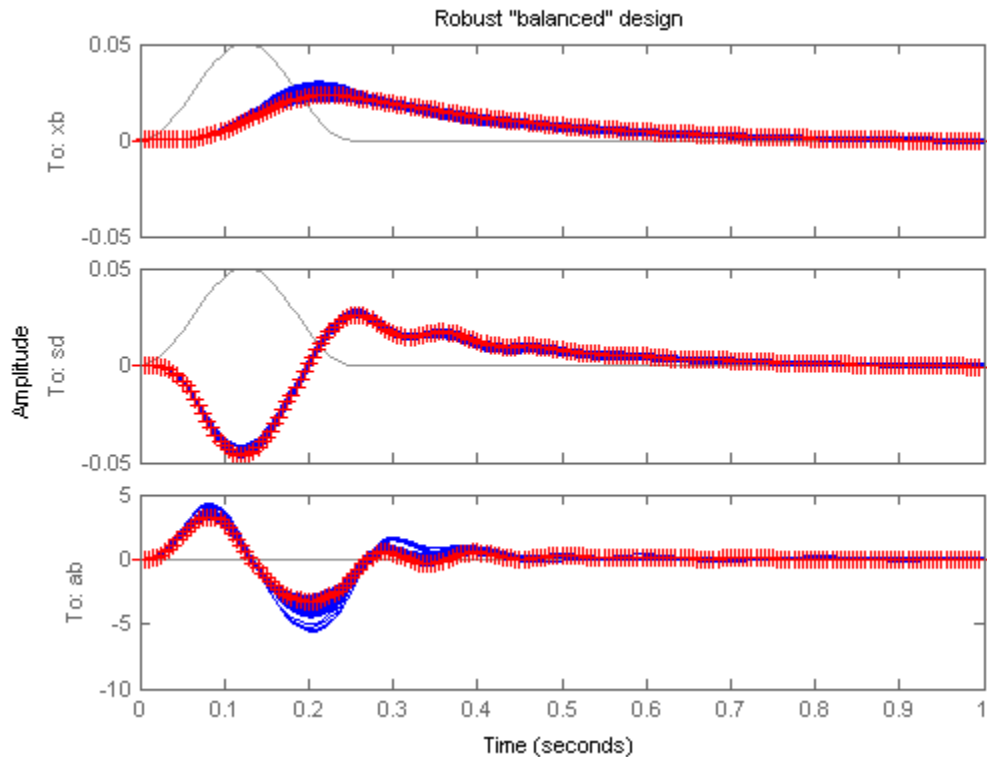


Compute an uncertain closed-loop model with the balanced robust controller, Krob. Sample this model, simulate the sampled models, and plot the results.

```

CLU = connect(qcar,Act,Krob, 'r', {'xb', 'sd', 'ab'});
figure(2)
clf
lsim(usample(CLU,nsamp), 'b', CLU.Nominal, 'r+', roaddist, t)
title('Robust "balanced" design')

```



The robust controller reduces variability caused by model uncertainty, and delivers more consistent performance.

### Controller Simplification

The robust controller `Krob` has eleven states. It is often the case that controllers synthesized with `dksyn` have high order. You can use the model reduction functions to find a lower-order controller that achieves the same level of robust performance. Use `reduce` to generate approximations of various orders. Then, use `robustperf` to compute the robust performance margin for each reduced-order approximation.

Create an array of reduced-order controllers.

```

NS = order(Krob);
StateOrders = 1:NS;
Kred = reduce(Krob,StateOrders);

```

Krob is a model array containing a reduced-order controller of every order from 1 up to the original 11 states.

Compute the robust performance margin for each reduced controller.

```

CLP = lft(qcaric(:,:,2),Kred);
ropt = robustperfOptions('Sensitivity','off','Display','off','Mussv','a');
PM = robustperf(CLP,ropt);

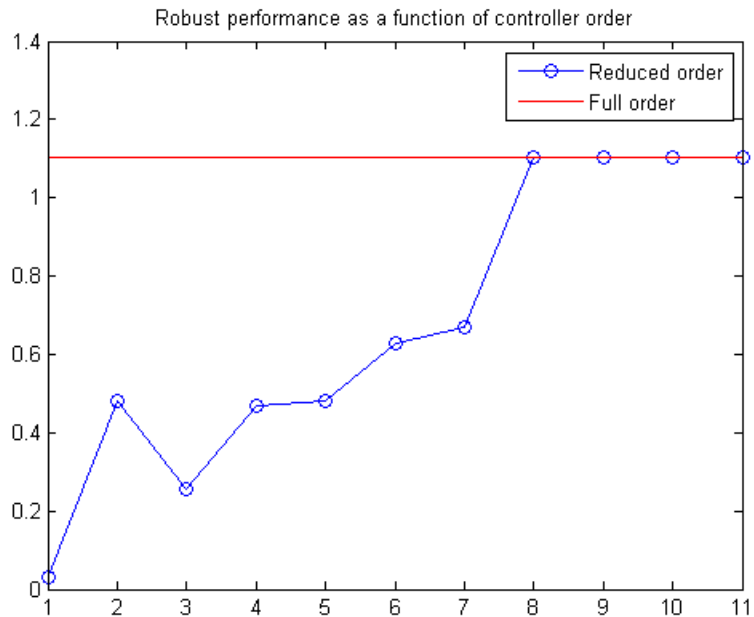
```

Compare the robust performance of the reduced- and full-order controllers.

```

plot(StateOrders,[PM.LowerBound],'b-o',...
      StateOrders, repmat(1/RPmuval,[1 NS]),'r');
title('Robust performance as a function of controller order')
legend('Reduced order','Full order')

```



There is no significant difference in robust performance between the 8th- and 11th-order controllers. Therefore, you can safely replace `Krob` by its 8th-order approximation.

```
Krob8 = Kred(:, :, 8);
```

You now have a simplified controller, `Krob8`, that provides robust control with a balance between passenger comfort and handling.

## Bibliography

- [1] Balas, G.J., and A.K. Packard, "The structured singular value  $\mu$ -framework," CRC Controls Handbook, Section 2.3.6, January, 1996, pp. 671-688.
- [2] Ball, J.A., and N. Cohen, "Sensitivity minimization in an  $H_\infty$  norm: Parametrization of all suboptimal solutions," *International Journal of Control*, Vol. 46, 1987, pp. 785-816.
- [3] Bamieh, B.A., and Pearson, J.B., "A general framework for linear periodic systems with applications to  $H_\infty$  sampled-data control," *IEEE Transactions on Automatic Control*, Vol. AC-37, 1992, pp. 418-435.
- [4] Doyle, J.C., Glover, K., Khargonekar, P., and Francis, B., "State-space solutions to standard  $H_2$  and  $H_\infty$  control problems," *IEEE Transactions on Automatic Control*, Vol. AC-34, No. 8, August 1989, pp. 831-847.
- [5] Fialho, I., and Balas, G.J., "Design of nonlinear controllers for active vehicle suspensions using parameter-varying control synthesis," *Vehicle Systems Dynamics*, Vol. 33, No. 5, May 2000, pp. 351-370.
- [6] Francis, B.A., *A course in  $H_\infty$  control theory*, Lecture Notes in Control and Information Sciences, Vol. 88, Springer-Verlag, Berlin, 1987.
- [7] Glover, K., and Doyle, J.C., "State-space formulae for all stabilizing controllers that satisfy an  $H_\infty$  norm bound and relations to risk sensitivity," *Systems and Control Letters*, Vol. 11, pp. 167-172, August 1989. *International Journal of Control*, Vol. 39, 1984, pp. 1115-1193.
- [8] Hedrick, J.K., and Batsuen, T., "Invariant Properties of Automotive Suspensions," *Proceedings of The Institution of Mechanical Engineers*, 204 (1990), pp. 21-27.
- [9] Lin, J., and Kanellakopoulos, I., "Road Adaptive Nonlinear Design of Active Suspensions," *Proceedings of the American Control Conference*, (1997), pp. 714-718.

- [10] Packard, A.K., Doyle, J.C., and Balas, G.J., "Linear, multivariable robust control with a  $\mu$  perspective," *ASME Journal of Dynamics, Measurements and Control: Special Edition on Control*, Vol. 115, No. 2b, June, 1993, pp. 426-438.
- [11] Skogestad, S., and Postlethwaite, I., *Multivariable Feedback Control: Analysis & Design*, John Wiley & Sons, 1996.
- [12] Stein, G., and Doyle, J., "Beyond singular values and loopshapes," *AIAA Journal of Guidance and Control*, Vol. 14, Num. 1, January, 1991, pp. 5-16.
- [13] Zames, G., "Feedback and optimal sensitivity: model reference transformations, multiplicative seminorms, and approximate inverses," *IEEE Transactions on Automatic Control*, Vol. AC-26, 1981, pp. 301-320.





# Tuning Fixed Control Architectures

---

- “What Is a Fixed-Structure Control System?” on page 6-3
- “Choosing an Approach to Fixed Control Structure Tuning” on page 6-4
- “Difference Between Fixed-Structure Tuning and Traditional H-Infinity Synthesis” on page 6-5
- “Structure of Control System for Tuning With looptune” on page 6-6
- “Set Up Your Control System for Tuning with looptune” on page 6-8
- “Performance and Robustness Specifications for looptune” on page 6-10
- “Tune MIMO Control System for Specified Bandwidth” on page 6-12
- “What Is hinfstruct?” on page 6-19
- “Formulating Design Requirements as H-Infinity Constraints” on page 6-20
- “Structured H-Infinity Synthesis Workflow” on page 6-21
- “Build Tunable Closed-Loop Model for Tuning with hinfstruct” on page 6-22
- “Tune the Controller Parameters” on page 6-29
- “Interpret the Outputs of hinfstruct” on page 6-30
- “Validate the Controller Design” on page 6-31
- “Set Up Your Control System for Tuning with systune” on page 6-35
- “Specifying Design Requirements for systune” on page 6-37
- “Tune Controller Against Set of Plant Models” on page 6-39
- “Supported Blocks for Tuning in Simulink” on page 6-40

- “Speed Up Tuning with Parallel Computing Toolbox Software” on page 6-42
- “Tuning Control Systems with SYSTUNE” on page 6-44
- “Tuning Feedback Loops with LOOPTUNE” on page 6-54
- “Tuning Control Systems in Simulink” on page 6-62
- “Building Tunable Models” on page 6-72
- “Using Design Requirement Objects” on page 6-81
- “Validating Results” on page 6-102
- “Tuning Multi-Loop Control Systems” on page 6-112
- “Using Parallel Computing to Accelerate Tuning” on page 6-124
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection” on page 6-129
- “Decoupling Controller for a Distillation Column” on page 6-141
- “Tuning of a Digital Motion Control System” on page 6-154
- “Control of a Linear Electric Actuator” on page 6-175
- “Multi-Loop PID Control of a Robot Arm” on page 6-187
- “Active Vibration Control in Three-Story Building” on page 6-205
- “Tuning of a Two-Loop Autopilot” on page 6-220
- “Multi-Loop Control of a Helicopter” on page 6-238
- “Fixed-Structure Autopilot for a Passenger Jet” on page 6-248
- “Fault-Tolerant Control of a Passenger Jet” on page 6-262
- “Fixed-Structure H-infinity Synthesis with HINFSTRUCT” on page 6-274
- “MIMO Control of Diesel Engine” on page 6-286
- “Digital Control of Power Stage Voltage” on page 6-304

## What Is a Fixed-Structure Control System?

Fixed-structure control systems are control systems that have predefined architectures and controller structures. For example,

- A single-loop SISO control architecture where the controller is a fixed-order transfer function, a PID controller, or a PID controller plus a filter.
- A MIMO control architecture where the controller has fixed order and structure. For example, a 2-by-2 decoupling matrix plus two PI controllers is a MIMO controller of fixed order and structure.
- A multiple-loop SISO or MIMO control architecture, including nested or cascaded loops, with multiple gains and dynamic components to tune.

You can use `systemtune`, `looptune` or `hinfstruct` for frequency-domain tuning of virtually any SISO or MIMO feedback architecture to meet your design requirements. You can use both approaches to tune fixed structure control systems in either MATLAB or Simulink (requires Simulink Control Design™).

## Choosing an Approach to Fixed Control Structure Tuning

Robust Control Toolbox includes several commands for frequency-domain tuning of control systems with fixed control structure.

- `systemtune` (or `s1Tunable.systemtune`) — Automatically tunes control systems from high-level design requirements you specify, such as reference tracking, disturbance rejection, and stability margins. `systemtune` jointly tunes all the free parameters of your control system regardless of its architecture and the number of feedback loops it contains.

`systemtune` can also robustly tune control systems for multiple plant models or multiple operating conditions of the plant.

For more information, see “Set Up Your Control System for Tuning with `systemtune`” on page 6-35.

- `looptune` (or `s1Tunable.looptune`) — Automatic tuning of SISO or MIMO feedback loops. `looptune` tunes the a SISO or MIMO feedback loop to high-level design requirements such as loop bandwidth, reference tracking, or stability margins. For more information, see “Set Up Your Control System for Tuning with `looptune`” on page 6-8.
- `hinfstruct` —  $H_\infty$  synthesis with fixed-structure controllers. Using `hinfstruct` requires you to formulate the  $H_\infty$  synthesis problem and choose weighting functions yourself. For more information, see “What Is `hinfstruct`?” on page 6-19.

## Difference Between Fixed-Structure Tuning and Traditional H-Infinity Synthesis

All of the tuning commands `systemtune`, `looptune`, and `hinfstruct` tune the controller parameters by optimizing the  $H_\infty$  norm across a closed-loop system (see [1]). However, these functions differ in important ways from traditional  $H_\infty$  methods.

Traditional  $H_\infty$  synthesis (performed using the `hifsyn` or `loopsyn` commands) designs a full-order, centralized controller. Traditional  $H_\infty$  synthesis provides no way to impose structure on the controller and often results in a controller that has high-order dynamics. Thus, the results can be difficult to map to your specific real-world control architecture. Additionally, traditional  $H_\infty$  synthesis requires you to express all design requirements in terms of a single weighted MIMO transfer function.

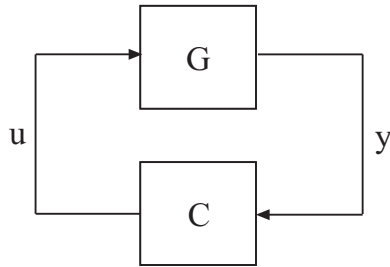
In contrast, structured  $H_\infty$  synthesis allows you to describe and tune the specific control system with which you are working. You can specify your control architecture, including the number and configuration of feedback loops. You can also specify the complexity, structure, and parametrization of each tunable component in your control system, such as PID controllers, gains, and fixed-order transfer functions. Additionally, you can easily combine requirements on separate closed-loop transfer functions.

### Bibliography

- [1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis," *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71-86.

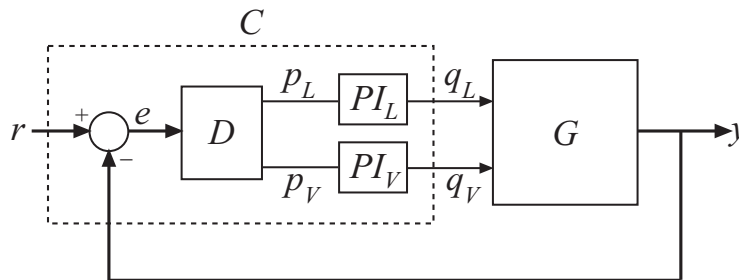
## Structure of Control System for Tuning With looptune

looptune tunes the feedback loop illustrated below to meet default requirements or requirements that you specify.



C represents the controller and G represents the plant. The sensor outputs (*measurement signals*) and actuator outputs  $u$  (*control signals*) define the boundary between plant and controller. The controller is the portion of your control system whose inputs are measurements, and whose outputs are control signals. Conversely, the plant is the remainder—the portion of your control system that receives control signals as inputs, and produces measurements as outputs.

For example, in the control system of the following illustration, the controller C receives the measurement  $y$ , and the reference signal  $r$ . The controller produces the controls  $q_L$  and  $q_V$  as outputs.



The controller C has a fixed internal structure. C includes a gain matrix  $D$ , the PI controllers  $PI_L$  and  $PI_V$ , and a summing junction. The looptune command tunes free parameters of C such as the gains in  $D$  and the

proportional and integral gains of PI\_L and PI\_V. You can also use looptune to co-tune free parameters in both C and G.

## Set Up Your Control System for Tuning with looptune

### In this section...

“Set Up Your Control System for looptune in MATLAB” on page 6-8

“Set Up Your Control System for looptune in Simulink” on page 6-8

### Set Up Your Control System for looptune in MATLAB

To set up your control system in MATLAB for tuning with looptune:

- 1 Parameterize the tunable elements of your controller. You can use predefined structures such as `ltiblock.pid`, `ltiblock.gain`, and `ltiblock.tf`. Or, you can create your own structure from elementary tunable parameters (`realp`).
- 2 Use model interconnection commands such as `series` and `connect` to build a tunable `genss` model representing the controller `C0`.
- 3 Create a Numeric LTI model representing the plant `G`. For co-tuning the plant and controller, represent the plant as a tunable `genss` model.

### Set Up Your Control System for looptune in Simulink

To set up your control system in Simulink for tuning with `sITunable.looptune` (requires Simulink Control Design software):

- 1 Use `sITunable` to create an interface to the Simulink model of your control system. When you create the interface, you specify which blocks to tune in your model.
- 2 Use `sITunable.addControl` and `sITunable.addMeasurement` to specify the control and measurement signals that define the boundaries between plant and controller. Use `sITunable.addSwitch` to mark optional loop-opening or signal injection sites for specifying and assessing open-loop requirements.

The `sITunable` interface automatically linearizes your Simulink model. The `sITunable` interface also automatically parametrizes the blocks that you



specify as tunable blocks. For more information, see the `sITunable` reference page and “Supported Blocks for Tuning in Simulink” on page 6-40.

## **Related Examples**

- “Tune MIMO Control System for Specified Bandwidth” on page 6-12
- “Tuning Feedback Loops with LOOPTUNE” on page 6-54

## **Concepts**

- “Structure of Control System for Tuning With looptune” on page 6-6

## Performance and Robustness Specifications for `looptune`

When you use `looptune`, you specify a target control bandwidth as a target open-loop gain crossover, the `wc` input to `looptune`. By default, `looptune` automatically tunes your control system to a loop shape of  $wc/s$  to enforce the following requirements:

- Performance—integral action at low frequency
- Roll-off—gain roll-off at frequencies above the crossover frequency
- Stability margins—adequate robustness at frequencies above the crossover frequency. You can override the default stability margins using the `looptuneOptions` command to create an options set to pass to `looptune`.

`looptune` is a wrapper for the `systemtune` command. `looptune` simplifies the setup for less complex problems by automatically tuning your feedback loop to meet the default performance requirements. You can specify additional design requirements using `TuningGoal` requirements objects. The following table summarizes the types of design requirements you can specify.

Design Requirement	Description	How To Specify
Time-domain response specification	Reference tracking and overshoot specification	<ul style="list-style-type: none"> <li>• <code>TuningGoal.Tracking</code></li> <li>• <code>TuningGoal.Overshoot</code></li> </ul>
Disturbance rejection	Disturbance rejection and sensitivity to disturbances	<ul style="list-style-type: none"> <li>• <code>TuningGoal.Rejection</code></li> <li>• <code>TuningGoal.Sensitivity</code></li> </ul>
Stability margins	Gain and phase margins, defined as multi-loop disk margins for MIMO systems (see <code>loopmargin</code> )	<code>TuningGoal.Margins</code>

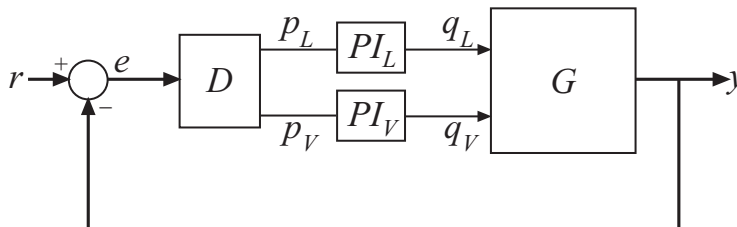
<b>Design Requirement</b>	<b>Description</b>	<b>How To Specify</b>
<b>Custom Loop Shape</b>	Constraints on the open-loop gain profile of a specified SISO or MIMO feedback loop	<p>When desired loop shape near crossover is simple or well understood:</p> <ul style="list-style-type: none"> <li>• <code>TuningGoal.LoopShape</code></li> </ul> <p>When you want to specify only high gain or low gain constraints in certain frequency bands:</p> <ul style="list-style-type: none"> <li>• <code>TuningGoal.MinLoopGain</code></li> <li>• <code>TuningGoal.MaxLoopGain</code></li> </ul>
<b>Other Gain Constraints</b>	Limit on the maximum gain and the gain profile across specified I/Os, such as frequency-weighted peak gain limit or noise amplification limit.	<ul style="list-style-type: none"> <li>• <code>TuningGoal.Gain</code></li> <li>• <code>TuningGoal.WeightedGain</code></li> <li>• <code>TuningGoal.Variance</code></li> <li>• <code>TuningGoal.WeightedVariance</code></li> </ul>
<b>Pole Placement</b>	Constraints on location of tunable closed-loop system poles and controller poles	<ul style="list-style-type: none"> <li>• <code>TuningGoal.Poles</code></li> <li>• <code>TuningGoal.StableController</code></li> </ul>

### Related Examples

- “Using Design Requirement Objects” on page 6-81

## Tune MIMO Control System for Specified Bandwidth

This example shows how to tune the following control system to achieve a loop crossover frequency between 0.1 and 1 rad/s, using `looptune`.



The plant,  $G$ , is a two-input, two-output model ( $y$  is a two-element vector signal). For this example, the transfer function of  $G$  is given by:

$$G(s) = \frac{1}{75s+1} \begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

This sample plant is based on the distillation column described in more detail in the example `Decoupling Controller for a Distillation Column`.

To tune this control system, you first create a numeric model of the plant. Then you create tunable models of the controller elements and interconnect them to build a controller model. Then you use `looptune` to tune the free parameters of the controller model. Finally, examine the performance of the tuned system to confirm that the tuned controller yields desirable performance.

Create a model of the plant.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL', 'qV'};
G.OutputName = 'y';
```

When you tune the control system, `looptune` uses the channel names `G.InputName` and `G.OutputName` to interconnect the plant and controller. Therefore, assign these channel names to match the illustration. When you

set `G.OutputName = 'y'`, the `G.OutputName` is automatically expanded to `{'y(1)'; 'y(2)'}.` This expansion occurs because `G` is a two-output system.

Represent the components of the controller.

```
D = ltiblock.gain('Decoupler', eye(2));
D.InputName = 'e';
D.OutputName = {'pL', 'pV'};
```

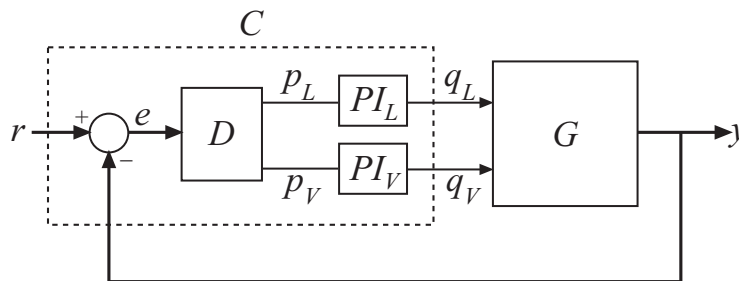
```
PI_L = ltiblock.pid('PI_L', 'pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';
```

```
PI_V = ltiblock.pid('PI_V', 'pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';
```

```
sum1 = sumblk('e = r - y', 2);
```

The control system includes several tunable control elements. `PI_L` and `PI_V` are tunable PI controllers. These elements represented by `ltiblock.pid` models. The fixed control structure also includes a decoupling gain matrix `D`, represented by a tunable `ltiblock.gain` model. When the control system is tuned, `D` ensures that each output of `G` tracks the corresponding reference signal `r` with minimal crosstalk.

Assigning `InputName` and `OutputName` values to these control elements allows you to interconnect them to create a tunable model of the entire controller `C` as shown.



When you tune the control system, `looptune` uses these channel names to interconnect C and G. The controller C also includes the summing junction `sum1`. This is a two-channel summing junction, because `r` and `y` are vector-valued signals of dimension 2.

Connect the controller components.

```
C0 = connect(PI_L,PI_V,D,sum1,{'r','y'},{'qL','qV'});
```

C0 is a tunable `genss` model that represents the entire controller structure. C0 stores the tunable controller parameters and contains the initial values of those parameters.

Tune the control system.

The inputs to `looptune` are G and C0, the plant and initial controller models that you created. The input `wc = [0.1,1]` sets the target range for the loop bandwidth. This input specifies that the crossover frequency of each loop in the tuned system fall between 0.1 and 1 rad/min.

```
wc = [0.1,1];  
[G,C,gam,Info] = looptune(G,C0,wc);
```

```
Final: Peak gain = 0.949, Iterations = 27
```

The displayed **Peak Gain = 0.949** indicates that `looptune` has found parameter values that achieve the target loop bandwidth. `looptune` displays the final peak gain value of the optimization run, which is also the output `gam`. If `gam` is less than 1, all tuning requirements are satisfied. A value greater than 1 indicates failure to meet some requirement. If `gam` exceeds 1, you can increase the target bandwidth range or relax another tuning requirement.

`looptune` also returns the tuned controller model C. This model is the tuned version of C0. It contains the PI coefficients and the decoupling matrix gain values that yield the optimized peak gain value.

Display the tuned controller parameters.

```
showTunable(C)
```

```
Decoupler =
```

$$d = \begin{array}{cc} & \begin{array}{cc} u1 & u2 \end{array} \\ \begin{array}{c} y1 \\ y2 \end{array} & \begin{bmatrix} 1.075 & -0.8273 \\ -1.581 & 1.328 \end{bmatrix} \end{array}$$

Name: Decoupler

Static gain.

-----  
 PI\_L =

$$K_p + K_i * \frac{1}{s}$$

with Kp = 2.82, Ki = 0.0398

Name: PI\_L

Continuous-time PI controller in parallel form.

-----  
 PI\_V =

$$K_p + K_i * \frac{1}{s}$$

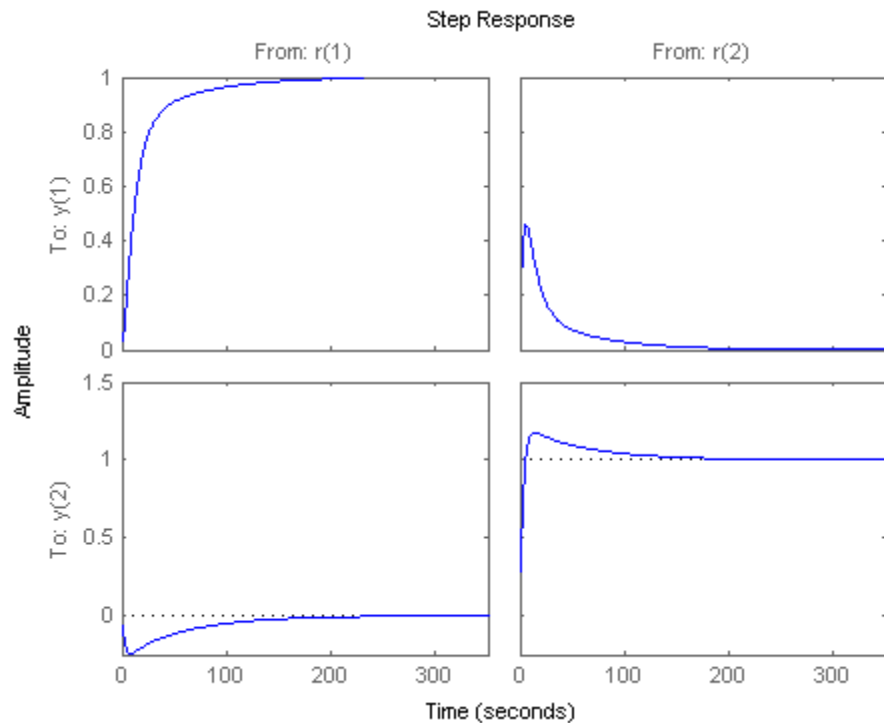
with Kp = -1.94, Ki = -0.0303

Name: PI\_V

Continuous-time PI controller in parallel form.

Check the time-domain response for the control system with the tuned coefficients. To produce a plot, construct a closed-loop model of the tuned control system. Plot the step response from reference to output.

```
T = connect(G,C,'r','y');
step(T)
```

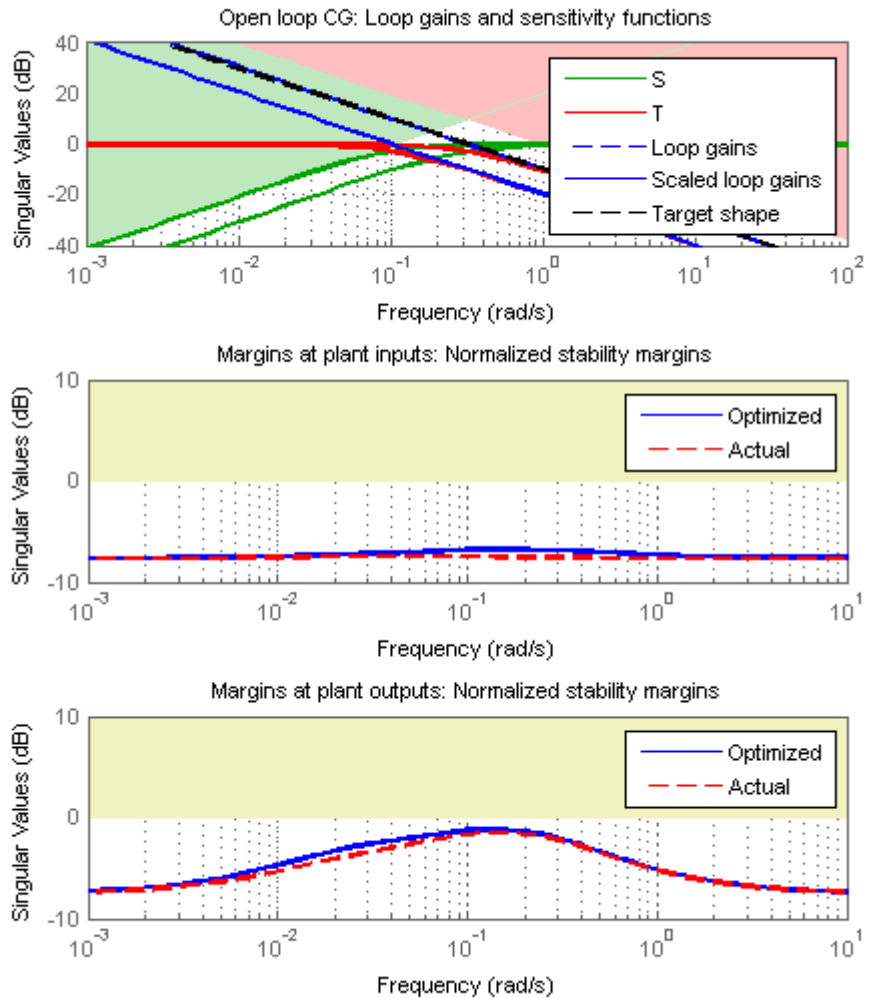


The decoupling matrix in the controller permits each channel of the two-channel output signal  $y$  to track the corresponding channel of the reference signal  $r$ , with minimal crosstalk. From the plot, you can see how well this requirement is achieved when you tune the control system for bandwidth alone. If the crosstalk still exceeds your design requirements, you can use a `TuningGoal.Gain` requirement object to impose further restrictions on tuning.

Examine the frequency-domain response of the tuned result as an alternative method for validating the tuned controller.

```
loopview(G,C,Info)
```





The first plot shows that the open-loop gain crossovers fall within the specified interval  $[0.1, 1]$ . This plot also includes the maximum and tuned values of the sensitivity function  $S = (I - GC)^{-1}$  and complementary sensitivity

$T = I - S$ . The second and third plots show that the MIMO stability margins of the tuned system (blue curve) do not exceed the upper limit (yellow curve).

### **Related Examples**

- “Decoupling Controller for a Distillation Column” on page 6-141
- “Using Design Requirement Objects” on page 6-81

### **Concepts**

- “Structure of Control System for Tuning With looptune” on page 6-6

## What Is hinfstruct?

`hinfstruct` lets you use the frequency-domain methods of  $H_\infty$  synthesis to tune control systems that have predefined architectures and controller structures.

To use `hinfstruct`, you describe your control system as a Generalized LTI model that keeps track of the tunable components of your system. `hinfstruct` tunes those parameters by minimizing the closed-loop gain from the system inputs to the system outputs (the  $H_\infty$  norm).

`hinfstruct` is the counterpart of `hinfsyn` for fixed-structure controllers. The methodology and algorithm behind `hinfstruct` are described in [1].

## Formulating Design Requirements as H-Infinity Constraints

Control design requirements are typically performance measures such as response speed, control bandwidth, roll-off, and steady-state error. To use `hinfstruct`, first express the design requirements as constraints on the closed-loop gain.

You can formulate design requirements in terms of the closed-loop gain using loop shaping. *Loop shaping* is a common systematic technique for defining control design requirements for  $H_\infty$  synthesis. In loop shaping, you first express design requirements as open-loop gain requirements.

For example, a requirement of good reference tracking and disturbance rejection is equivalent to high ( $>1$ ) open-loop gain at low frequency. A requirement of insensitivity to measurement noise or modeling error is equivalent to a low ( $<1$ ) open-loop gain at high frequency. You can then convert these open-loop requirements to constraints on the closed-loop gain using weighting functions.

This formulation of design requirements results in a  $H_\infty$  constraint of the form:

$$\|H(s)\|_\infty < 1,$$

where  $H(s)$  is a closed-loop transfer function that aggregates and normalizes the various requirements.

For an example of how to formulate design requirements for  $H_\infty$  synthesis using loop shaping, see “Fixed-Structure H-infinity Synthesis with HINFSTRUCT” on page 6-274.

For more information about constructing weighting functions from design requirements, see “H-Infinity Performance” on page 5-9.

## Structured H-Infinity Synthesis Workflow

Performing structured  $H_\infty$  synthesis requires the following steps:

- 1** Formulate your design requirements as  $H_\infty$  constraints, which are constraints on the closed-loop gains from specific system inputs to specific system outputs.
- 2** Build tunable models of the closed-loop transfer functions of Step 1.
- 3** Tune the control system using `hinfstruct`.
- 4** Validate the tuned control system.

## Build Tunable Closed-Loop Model for Tuning with `hinfstruct`

In the previous step you expressed your design requirements as a constraint on the  $H_\infty$  norm of a closed-loop transfer function  $H(s)$ .

The next step is to create a Generalized LTI model of  $H(s)$  that includes all of the fixed and tunable elements of the control system. The model also includes any weighting functions that represent your design requirements. There are two ways to obtain this tunable model of your control system:

- Construct the model using Control System Toolbox commands.
- Obtain the model from a Simulink model using Simulink Control Design commands.

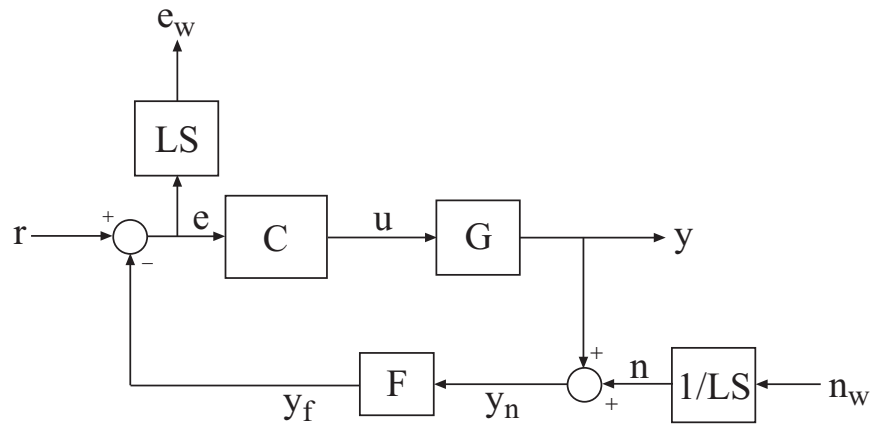
### Constructing the Closed-Loop System Using Control System Toolbox Commands

To construct the tunable generalized linear model of your closed-loop control system in MATLAB:

- 1** Use commands such as `tf`, `zpk`, and `ss` to create numeric linear models that represent the fixed elements of your control system and any weighting functions that represent your design requirements.
- 2** Use tunable models (either Control Design Blocks or Generalized LTI models) to model the tunable elements of your control system. For more information about tunable models, see “Models with Tunable Coefficients” in the *Control System Toolbox User’s Guide*.
- 3** Use model-interconnection commands such as `series`, `parallel`, and `connect` to construct your closed-loop system from the numeric and tunable models.

### Example: Modeling a Control System With a Tunable PI Controller and Tunable Filter

This example shows how to construct a tunable generalized linear model of the following control system for tuning with `hinfstruct`.



This block diagram represents a head-disk assembly (HDA) in a hard disk drive. The architecture includes the plant  $G$  in a feedback loop with a PI controller  $C$  and a low-pass filter,  $F = a/(s+a)$ . The tunable parameters are the PI gains of  $C$  and the filter parameter  $a$ .

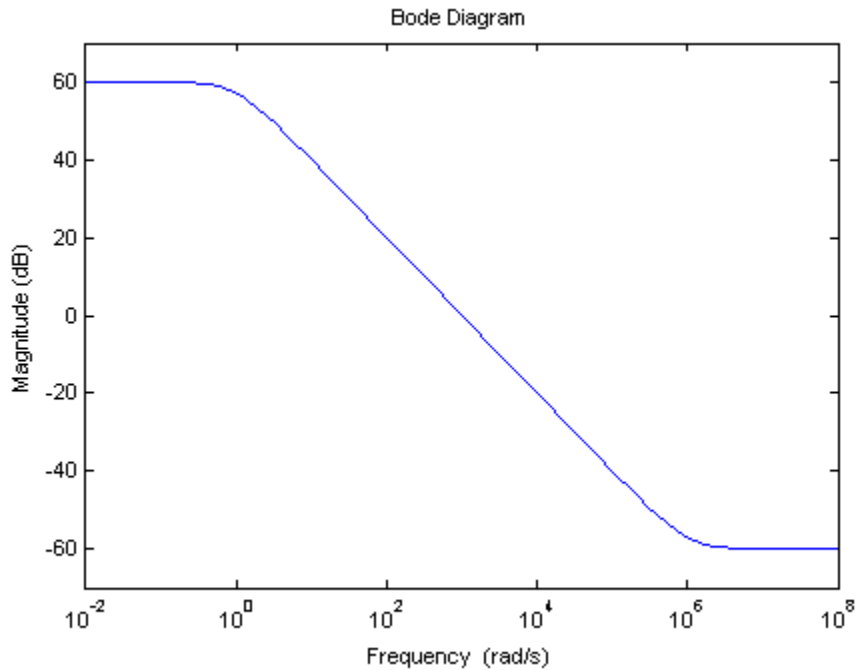
The block diagram also includes the weighting functions  $LS$  and  $1/LS$ , which express the loop-shaping requirements. Let  $T(s)$  denote the closed-loop transfer function from inputs  $(r, n_w)$  to outputs  $(y, e_w)$ . Then, the  $H_\infty$  constraint:

$$\|T(s)\|_\infty < 1$$

approximately enforces the target open-loop response shape  $LS$ . For this example, the target loop shape is

$$LS = \frac{1 + 0.001 \frac{s}{\omega_c}}{0.001 + \frac{s}{\omega_c}}$$

This value of  $LS$  corresponds to the following open-loop response shape.



To tune the HDA control system with `hinfstruct`, construct a tunable model of the closed-loop system  $T(s)$ , including the weighting functions, as follows.

- 1 Load the plant  $G$  from a saved file.

```
load hinfstruct_demo G
```

$G$  is a 9th-order SISO state-space (ss) model.

- 2 Create a tunable model of the PI controller.

You can use the predefined Control Design Block `ltiblock.pid` to represent a tunable PI controller.

```
C = ltiblock.pid('C', 'pi');
```



**3** Create a tunable model of the low-pass filter.

Because there is no predefined Control Design Block for the filter  $F = a/(s+a)$ , use `realp` to represent the tunable filter parameter `a`. Then create a tunable `genss` model representing the filter.

```
a = realp('a',1);
F = tf(a,[1 a]);
```

**4** Specify the target loop shape LC.

```
wc = 1000;
s = tf('s');
LS = (1+0.001*s/wc)/(0.001+s/wc);
```

**5** Label the inputs and outputs of all the components of the control system.

Labeling the I/Os allows you to connect the elements to build the closed-loop system  $T(s)$ .

```
Wn = 1/LS; Wn.InputName = 'nw'; Wn.OutputName = 'n';
We = LS; We.InputName = 'e'; We.OutputName = 'ew';
C.InputName = 'e'; C.OutputName = 'u';
F.InputName = 'yn'; F.OutputName = 'yf';
```

**6** Specify the summing junctions in terms of the I/O labels of the other components of the control system.

```
Sum1 = sumblk('e = r - yf');
Sum2 = sumblk('yn = y + n');
```

**7** Use `connect` to combine all the elements into a complete model of the closed-loop system  $T(s)$ .

```
T0 = connect(G,Wn,We,C,F,Sum1,Sum2,{'r','nw'},{'y','ew'});
```

`T0` is a `genss` object, which is a Generalized LTI model representing the closed-loop control system with weighting functions. The `Blocks` property of `T0` contains the tunable blocks `C` and `a`.

```
T0.Blocks
```

```
ans =
```

```
C: [1x1 ltiblock.pid]
a: [1x1 realp]
```

For more information about generalized models of control systems that include both numeric and tunable components, see “Models with Tunable Coefficients” in the Control System Toolbox documentation.

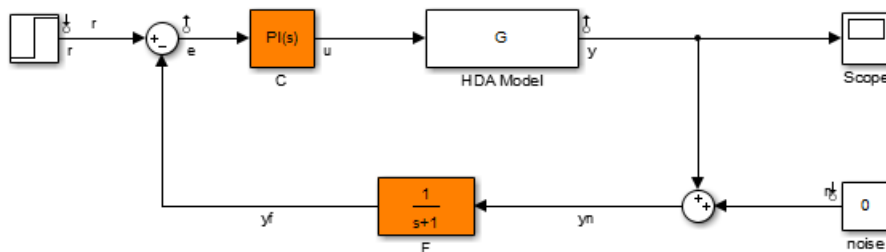
You can now use `hinfstruct` to tune the parameters of this control system. See “Tune the Controller Parameters” on page 6-29.

## Constructing the Closed-Loop System Using Simulink Control Design Commands

If you have a Simulink model of your control system and Simulink Control Design software, use `sITunable` to create an interface to the Simulink model of your control system. When you create the interface, you specify which blocks to tune in your model. The `sITunable` interface allows you to extract a closed-loop model for tuning with `hinfstruct`.

### Example: Creating a Weighted Tunable Model of Control System Starting From a Simulink Model

This example shows how to construct a tunable generalized linear model of the control system in the Simulink model `rct_diskdrive`.



To create a generalized linear model of this control system (including loop-shaping weighting functions):

- 1 Open the model.

```
open('rct_diskdrive');
```

- 2** Create an `sITunable` interface to the model. The interface allows you to specify the tunable blocks and extract linearized open-loop and closed-loop responses. (For more information about the interface, see the `sITunable` reference page.)

```
ST0 = sITunable('rct_diskdrive',{'C','F'});
```

This command specifies that `C` and `F` are the tunable blocks in the model. The `sITunable` interface automatically parametrizes these blocks. The default parametrization of the transfer function block `F` is a transfer function with two free parameters. Because `F` is a low-pass filter, you must constrain its coefficients. To do so, specify a custom parameterization of `F`.

```
a = realp('a',1);    % filter coefficient
setBlockParam(ST0,'F',tf(a,[1 a]));
```

- 3** Extract a tunable model of the closed-loop transfer function you want to tune.

```
T0 = getIOTransfer(ST0,{'r','n'},{'y','e'});
```

This command returns a `genss` model of the linearized closed-loop transfer function from the reference and noise inputs `r,n` to the measurement and error outputs `y,e`. The error output is needed for the loop-shaping weighting function.

- 4** Define the loop-shaping weighting functions and append them to `T0`.

```
wc = 1000;
s = tf('s');
LS = (1+0.001*s/wc)/(0.001+s/wc);
```

```
T0 = blkdiag(1,LS) * T0 * blkdiag(1,1/LS);
```

The generalized linear model `T0` is a tunable model of the closed-loop transfer function  $T(s)$ , discussed in “Example: Modeling a Control System With a Tunable PI Controller and Tunable Filter” on page 6-22.  $T(s)$  is a weighted closed-loop model of the control system of `rct_diskdrive`. Tuning `T0` to enforce the  $H_\infty$  constraint

$$\|T(s)\|_{\infty} < 1$$

approximately enforces the target loop shape LS.

You can now use `hinfstruct` to tune the parameters of this control system. See “Tune the Controller Parameters” on page 6-29.

## Tune the Controller Parameters

After you obtain the `genss` model representing your control system, use `hinfstruct` to tune the tunable parameters in the `genss` model .

`hinfstruct` takes a tunable linear model as its input.

For example, you can tune controller parameters for the example discussed in “Build Tunable Closed-Loop Model for Tuning with `hinfstruct`” on page 6-22 using the following command:

```
[T,gamma,info] = hinfstruct(T0);
```

This command returns the following outputs:

- `T`, a `genss` model object containing the tuned values of `C` and `a`.
- `gamma`, the minimum peak closed-loop gain of `T` achieved by `hinfstruct`.
- `info`, a structure containing additional information about the minimization runs.

## Interpret the Outputs of `hinfstruct`

### Output Model is Tuned Version of Input Model

T contains the same tunable components as the input closed-loop model T0. However, the parameter values of T are now tuned to minimize the  $H_\infty$  norm of this transfer function.

### Interpreting `gamma`

`gamma` is the smallest  $H_\infty$  norm achieved by the optimizer. Examine `gamma` to determine how close the tuned system is to meeting your design constraints. If you normalize your  $H_\infty$  constraints, a final `gamma` value of 1 or less indicates that the constraints are met. A final `gamma` value exceeding 1 by a small amount indicates that the constraints are nearly met.

The value of `gamma` that `hinfstruct` returns is a local minimum of the gain minimization problem. For best results, use the `RandomStart` option to `hinfstruct` to obtain several minimization runs. Setting `RandomStart` to an integer  $N > 0$  causes `hinfstruct` to run the optimization  $N$  additional times, beginning from parameter values it chooses randomly. For example:

```
opts = hinfstructOptions('RandomStart',5);  
[T,gamma,info] = hinfstruct(T0,opts);
```

You can examine `gamma` for each run to identify an optimization result that meets your design requirements.

For more details about `hinfstruct`, its options, and its outputs, see the `hinfstruct` and `hinfstructOptions` reference pages.

## Validate the Controller Design

To validate the `hinfstruct` control design, analyze the tuned output models described in “Interpret the Outputs of `hinfstruct`” on page 6-30. Use these tuned models to examine the performance of the tuned system.

### Validating the Design in MATLAB

This example shows how to obtain the closed-loop step response of a system tuned with `hinfstruct` in MATLAB.

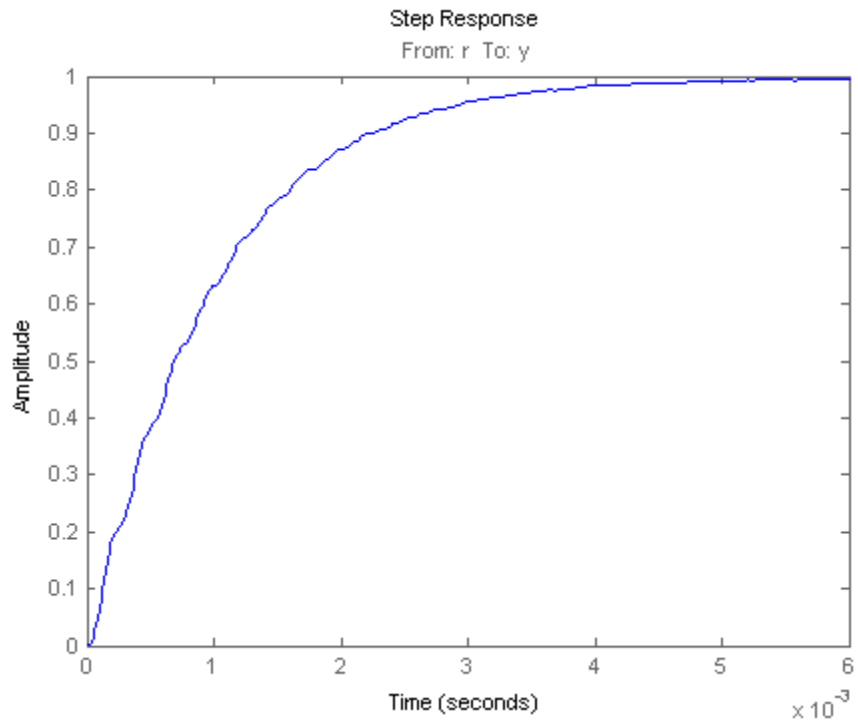
You can use the tuned versions of the tunable components of your system to build closed-loop or open-loop numeric LTI models of the tuned control system. You can then analyze open-loop or closed-loop performance using other Control System Toolbox tools.

In this example, create and analyze a closed-loop model of the HDA system tuned in “Tune the Controller Parameters” on page 6-29. To do so, extract the tuned controller values from the tuned closed-loop model. Use `getBlockValue` to get the tuned value of a tunable Control Design Block. Use `getValue` to evaluate a `genss` model using parameter values from another `genss` model.

```
Ctuned = getBlockValue(T, 'C');  
Ftuned = getValue(F,T);
```

Then construct the tuned closed-loop response of the control system from `r` to `y`.

```
Try = connect(G,Ctuned,Ftuned,Sum1,Sum2, 'r', 'y');  
step(Try)
```



### Validating the Design in Simulink

This example shows how to write tuned values to your Simulink model for validation.

The `sITunable` interface linearizes your Simulink model. As a best practice, validate the tuned parameters in your nonlinear model. You can use the `sITunable` interface to do so.

In this example, write tuned parameters to the `rct_diskdrive` system tuned in “Tune the Controller Parameters” on page 6-29.

Make a copy of the `sITunable` description of the control system, to preserve the original parameter values. Then propagate the tuned parameter values to the copy.

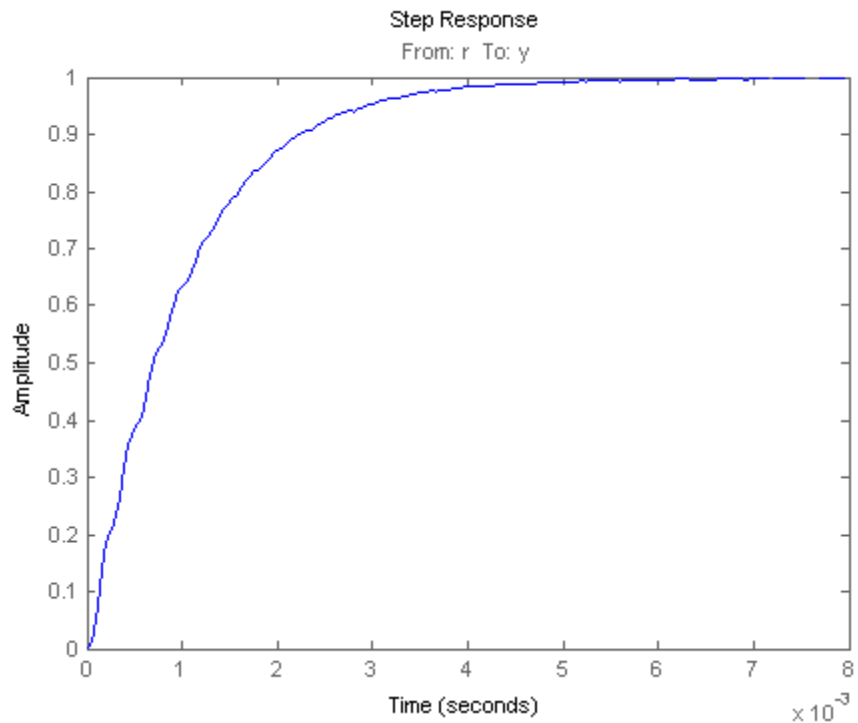


```
ST = copy(ST0);  
setBlockValue(ST,T);
```

This command writes the parameter values from the tuned, weighted closed-loop model T to the corresponding parameters in the interface ST.

You can examine the closed-loop responses of the linearized version of the control system represented by ST. For example:

```
Try = getIOTransfer(ST,'r','y');
```



Since `hinfstruct` tunes a linearized version of your system, you should also validate the tuned controller in the full nonlinear Simulink model. To do so, write the parameter values from the `sITunable` interface to the Simulink model.

```
writeBlockValue(ST)
```

You can now simulate the model using the tuned parameter values to validate the controller design.

## Set Up Your Control System for Tuning with `systeme`

`systeme` tunes the free parameters of your control system to meet tuning requirements that you specify. To use `systeme`, you create a tunable model of your control system in either MATLAB or Simulink.

### Set Up Your Control System for `systeme` in MATLAB

To set up your control system in MATLAB for tuning with `systeme`:

- 1 Parameterize the tunable elements of your control system. You can use predefined structures such as `ltiblock.pid`, `ltiblock.gain`, and `ltiblock.tf`. Or, you can create your own structure from elementary tunable parameters (`realp`).
- 2 Use model interconnection commands such as `feedback` and `connect` to build a closed-loop model of the overall control system as an interconnection of fixed and tunable components. Use `loopswitch` blocks to mark optional loop-opening or signal injection sites for specifying and assessing open-loop requirements. `systeme` tunes the resulting `genss` model of the control system.

### Set Up Your Control System for `systeme` in Simulink

To set up your control system in Simulink for tuning with `s1Tunable.systeme` (requires Simulink Control Design software):

- 1 Use `s1Tunable` to create an interface to the Simulink model of your control system. When you create the interface, you specify which blocks to tune in your model.
- 2 Use `s1Tunable.addIO`, `s1Tunable.addControl`, and `s1Tunable.addMeasurement` to mark the input and output signals you need to specify and assess closed-loop requirements.
- 3 Use `s1Tunable.addSwitch`, `s1Tunable.addControl`, and `s1Tunable.addMeasurement` to mark optional loop-opening sites for specifying and assessing open-loop requirements.

The `s1Tunable` interface automatically linearizes your Simulink model. The `s1Tunable` interface also automatically parametrizes the blocks that you

specify as tunable blocks. For more information, see the `sITunable` reference page and “Supported Blocks for Tuning in Simulink” on page 6-40.

After setting up your control system, define your design requirements using `TuningGoal` objects as described in “Specifying Design Requirements for `systune`” on page 6-37

.

### **Related Examples**

- “Using Design Requirement Objects” on page 6-81
- “Tuning Control Systems with `SYSTUNE`” on page 6-44

## Specifying Design Requirements for systune

When you tune a control system with `systune`, you specify tuning requirements such as reference tracking, maximum gain from specified inputs to specified outputs, or stability margins. You can designate tuning requirements as either soft constraints or hard constraints. `systune` tunes the free parameters of your control system to come as close as possible to satisfying the soft constraints, subject to satisfying the hard constraints.

Specify your tuning requirements using `TuningGoal` requirements objects. The following table summarizes the types of design requirements you can specify for `systune`.

Design Requirement	Description	How To Specify
<b>Time-domain response specification</b>	Reference tracking and overshoot specification	<ul style="list-style-type: none"> <li>• <code>TuningGoal.Tracking</code></li> <li>• <code>TuningGoal.Overshoot</code></li> </ul>
<b>Disturbance rejection</b>	Disturbance rejection and sensitivity to disturbances	<ul style="list-style-type: none"> <li>• <code>TuningGoal.Rejection</code></li> <li>• <code>TuningGoal.Sensitivity</code></li> </ul>
<b>Stability margins</b>	Gain and phase margins, defined as multi-loop disk margins for MIMO systems (see <code>loopmargin</code> )	<code>TuningGoal.Margins</code>
<b>Custom Loop Shape</b>	Constraints on the open-loop gain profile of a specified SISO or MIMO feedback loop	<p>When desired loop shape near crossover is simple or well understood:</p> <ul style="list-style-type: none"> <li>• <code>TuningGoal.LoopShape</code></li> </ul> <p>When you want to specify only high gain or low gain constraints in certain frequency bands:</p> <ul style="list-style-type: none"> <li>• <code>TuningGoal.MinLoopGain</code></li> <li>• <code>TuningGoal.MaxLoopGain</code></li> </ul>

<b>Design Requirement</b>	<b>Description</b>	<b>How To Specify</b>
<b>Other Gain Constraints</b>	Limit on the maximum gain and the gain profile across specified I/Os, such as frequency-weighted peak gain limit or noise amplification limit.	<ul style="list-style-type: none"> <li>• TuningGoal.Gain</li> <li>• TuningGoal.WeightedGain</li> <li>• TuningGoal.Variance</li> <li>• TuningGoal.WeightedVariance</li> </ul>
<b>Pole Placement</b>	Constraints on location of tunable closed-loop system poles and controller poles	<ul style="list-style-type: none"> <li>• TuningGoal.Poles</li> <li>• TuningGoal.StableController</li> </ul>

**Related Examples**

- “Using Design Requirement Objects” on page 6-81
- “Tuning Control Systems with SYSTUNE” on page 6-44

## Tune Controller Against Set of Plant Models

`system` can simultaneously tune the parameters of multiple models or control configurations. For example you can:

- Tune a single controller against a range of plant models, to help ensure that the tuned control system is robust against parameter variations
- Tune for reliable control by simultaneously to multiple plant configurations that represent different failure modes of a system

In either case, `system` finds values for tunable parameters that best satisfy the specified tuning objectives for all models.

To tune a controller parameters against a set of plant models:

- 1** Create an array of `genss` models that represent the control systems or configurations to tune against.
- 2** Specify your tuning objectives using `TuningGoal` requirements objects such as `TuningGoal.Tracking`, `TuningGoal.Gain`, or `TuningGoal.Margins`. If you want to limit a tuning requirement to a subset of the models in the model array, set the `Models` property of the `TuningGoal` requirement.

For example, suppose you have a model array whose first entry represents your control system under normal operating conditions, and whose second entry represents the system in a failure mode. Suppose further that `Req` is a `TuningGoal.Tracking` requirement that only applies to the normally operating system. To enforce the requirement only on the first entry, use the following command:

```
Req.Models = [1];
```

- 3** Provide the model array and tuning requirements as input argument to `system`.

`system` jointly tunes the tunable parameters for all models in the array to best meet the tuning requirements as you specify them. `system` returns an array containing the corresponding tuned models. You can use `getBlockValue` or `showBlockValue` to access the tuned values of the control elements.

## Supported Blocks for Tuning in Simulink

sITunable automatically assigns a parameterization to certain Simulink blocks. You can write tuned parameter values back to the Simulink model using `sITunable.writeBlockValue`.

Supported blocks include the following:

Automatically Parametrized Blocks	Simulink Library
Gain	Math Operations
LTI System	Control System Toolbox
Discrete Filter	Discrete
PID Controller (one-degree-of-freedom only)	<ul style="list-style-type: none"> <li>• Continuous</li> <li>• Discrete</li> </ul>
State-Space blocks	<ul style="list-style-type: none"> <li>• Continuous</li> <li>• Discrete</li> <li>• Simulink Extras Additional Linear</li> </ul>
Zero-pole blocks	<ul style="list-style-type: none"> <li>• Continuous</li> <li>• Discrete</li> </ul>
Transfer function blocks	<ul style="list-style-type: none"> <li>• Continuous</li> <li>• Discrete</li> </ul>

LTI blocks, state-space blocks, transfer function blocks, and zero-pole blocks discretized using the Model Discretizer are also supported.

### Tuning Other Blocks

You can specify tuned blocks in the `sITunable` interface other than the automatically parametrized blocks. `sITunable` assigns a state-space parametrization to such blocks based upon the block linearization. Use



`slTunable.getBlockParam` to examine the parametrization of a block. Use `slTunable.setBlockParam` to override the default parametrization.

For blocks that are not automatically parametrized, you cannot write parameter values back to the block using `slTunable.writeBlockValue`.

## Speed Up Tuning with Parallel Computing Toolbox Software

This example shows how to speed up the tuning of fixed-structure control systems if you have the Parallel Computing Toolbox™ software installed. When you run multiple randomized optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

To distribute randomized optimization runs among workers:

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `systuneOptions`, `looptuneOptions`, or `hinfstructOptions` set that specifies multiple random starts. For example, the following options set specifies 20 random restarts to run in parallel for tuning with `looptune`:

```
options = systuneOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the options set when you call `systune`, `looptune` or `hinfstruct`. For example, if you have already created a tunable control system model, `CL0`, and tunable controller, and tuning requirement vectors `SoftReqs` and `HardReqs`, the following command uses parallel computing to tune the control system of `CL0` with `systune`.

```
[CL,fSoft,gHard,info] = systune(CL0,SoftReq,Hardreq,options);
```

To learn more about configuring a parallel pool, see the Parallel Computing Toolbox documentation.

**See Also** `parpool`

**Related  
Examples**

- “Using Parallel Computing to Accelerate Tuning” on page 6-124

**Concepts**

- “Parallel Preferences”

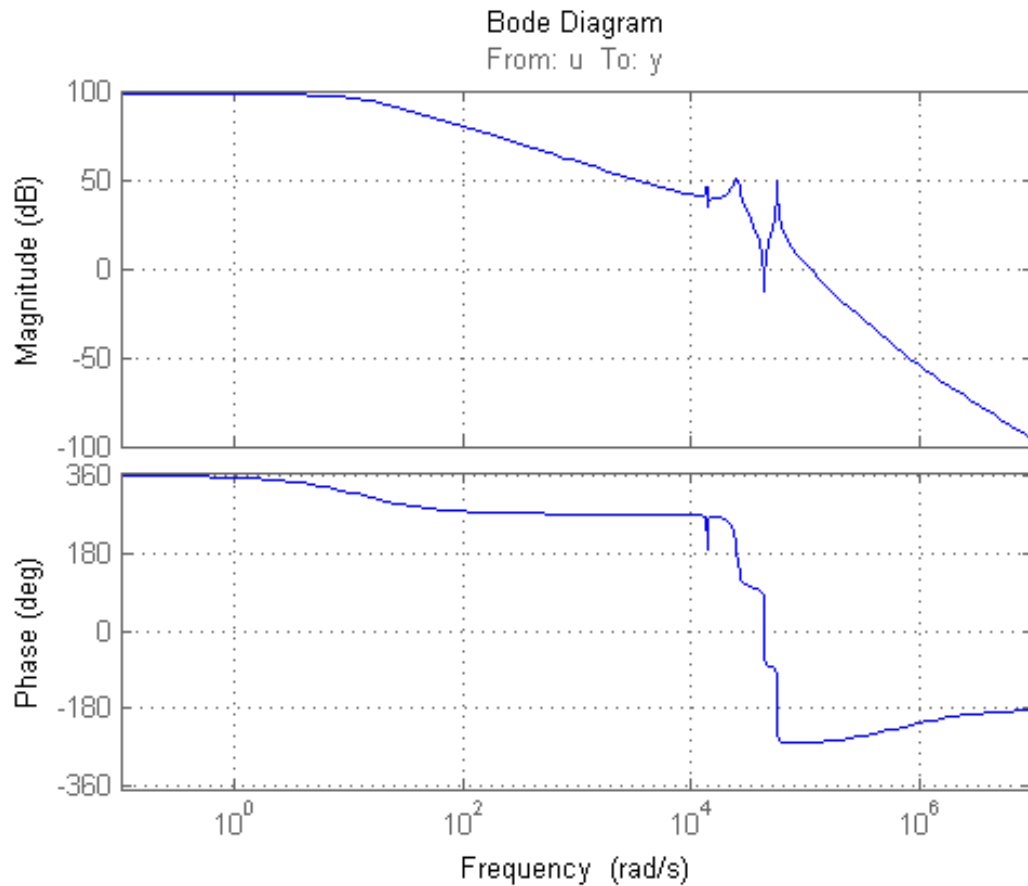
# Tuning Control Systems with SYSTUNE

The `system` command can jointly tune the gains of your control system regardless of its architecture and number of feedback loops. This example outlines the `system` workflow on a simple application.

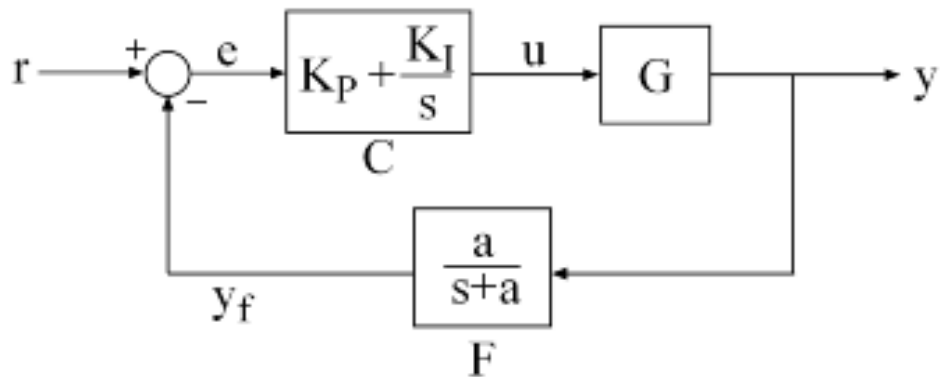
### Head-Disk Assembly Control

This example uses a 9th-order model of the head-disk assembly (HDA) in a hard-disk drive. This model captures the first few flexible modes in the HDA.

```
load rctExamples G
bode(G), grid
```



We use the feedback loop shown below to position the head on the correct track. This control structure consists of a PI controller and a low-pass filter in the return path. The head position  $y$  should track a step change  $r$  with a response time of about one millisecond, little or no overshoot, and no steady-state error.



**Figure 1: Control Structure**

You can use `systemtune` to directly tune the PI gains and filter coefficient  $a$  subject to a variety of time- and frequency-domain requirements.

#### **Specifying the Tunable Elements**

There are two tunable elements in the control structure of Figure 1: the PI controller  $C(s)$  and the low-pass filter

$$F(s) = \frac{a}{s+a}.$$

You can use the `ltiblock.pid` class to parameterize the PI block:

```
C0 = ltiblock.pid('C','pi'); % tunable PI
```

To parameterize the lowpass filter  $F(s)$ , create a tunable real parameter  $a$  and construct a first-order transfer function with numerator  $a$  and denominator  $s + a$ :

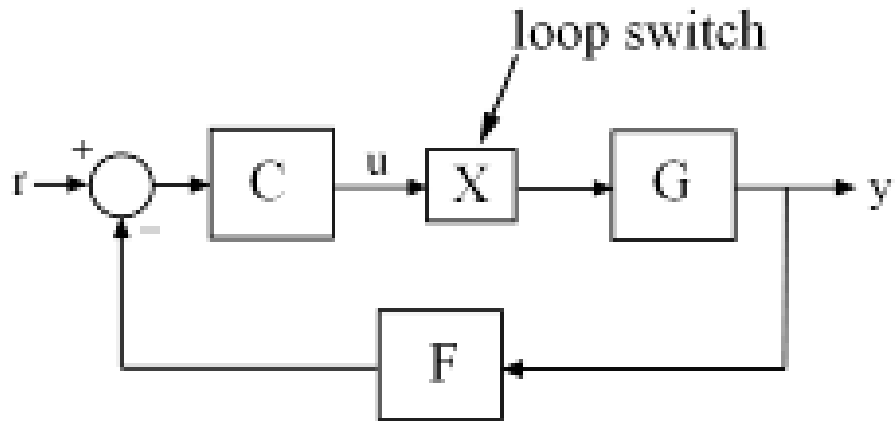
```
a = realp('a',1); % filter coefficient
FO = tf(a,[1 a]); % filter parameterized by a
```

See the *"Building Tunable Models"* example for an overview of available tunable elements.

### Building a Tunable Closed-Loop Model

Next build a closed-loop model of the feedback loop in Figure 1. To facilitate open-loop analysis and specify open-loop requirements such as desired stability margins, add a loop opening switch at the plant input  $u$ :

```
LSU = loopswitch('u');
```



**Figure 2: Loop Switch Block**

Use `feedback` to build a model of the closed-loop transfer from reference  $r$  to head position  $y$ :

```
T0 = feedback(G*LSU*C0,F0); % closed-loop transfer from r to y
T0.InputName = 'r';
T0.OutputName = 'y';
```

The result  $T0$  is a generalized state-space model (`genss`) that depends on the tunable elements  $C$  and  $F$ .

### Specifying the Design Requirements

The `TuningGoal` package contains a variety of control design requirements for specifying the desired behavior of the control system. These include requirements on the response time, deterministic and stochastic gains, loop shape, stability margins, and pole locations. Here we use two requirements to capture the control objectives:

- **Tracking requirement** : The position  $y$  should track the reference  $r$  with a 1 millisecond response time
- **Stability margin requirement** : The feedback loop should have 6dB of gain margin and 45 degrees of phase margin

Use the `TuningGoal.Tracking` and `TuningGoal.Margins` objects to capture these requirements. Note that the margins requirement applies to the open-loop response measured at the plant input  $u$  (location marked by the loopswitch block `LSU`).

```
Req1 = TuningGoal.Tracking('r','y',0.001);
Req2 = TuningGoal.Margins('u',6,45);
```

### Tuning the Controller Parameters

You can now use `sysstune` to tune the PI gain and filter coefficient  $\alpha$ . This function takes the tunable closed-loop model `T0` and the requirements `Req1,Req2`. Use a few randomized starting points to improve the chances of getting a globally optimal design.

```
rng('default')
Options = sysstuneOptions('RandomStart',3);
[T,fSoft,~,Info] = sysstune(T0,[Req1,Req2],Options);
```

```
Final: Soft = 1.35, Hard = -Inf, Iterations = 152
Final: Soft = 2.61, Hard = -Inf, Iterations = 116
Final: Soft = 2.78e+03, Hard = -Inf, Iterations = 184
      Min decay rate 1e-07 is close to lower bound 1e-07
Final: Soft = 1.35, Hard = -Inf, Iterations = 59
```

All requirements are normalized so a requirement is satisfied when its value is less than 1. Here the final value is slightly greater than 1, indicating that the requirements are nearly satisfied. Use the output `fSoft` to see the tuned



value of each requirement. Here we see that the first requirement (tracking) is slightly violated while the second requirement (margins) is satisfied.

`fSoft`

```
fSoft =
    1.3461    0.6326
```

The first output `T` of `systune` is the "tuned" closed-loop model. Use `showTunable` or `getBlockValue` to access the tuned values of the PI gains and filter coefficient:

```
getBlockValue(T,'C') % tuned value of PI controller
```

```
ans =
    Kp + Ki * ---
                s
with Kp = 0.00104, Ki = 0.0122
```

```
Name: C
Continuous-time PI controller in parallel form.
```

```
showTunable(T) % tuned values of all tunable elements
```

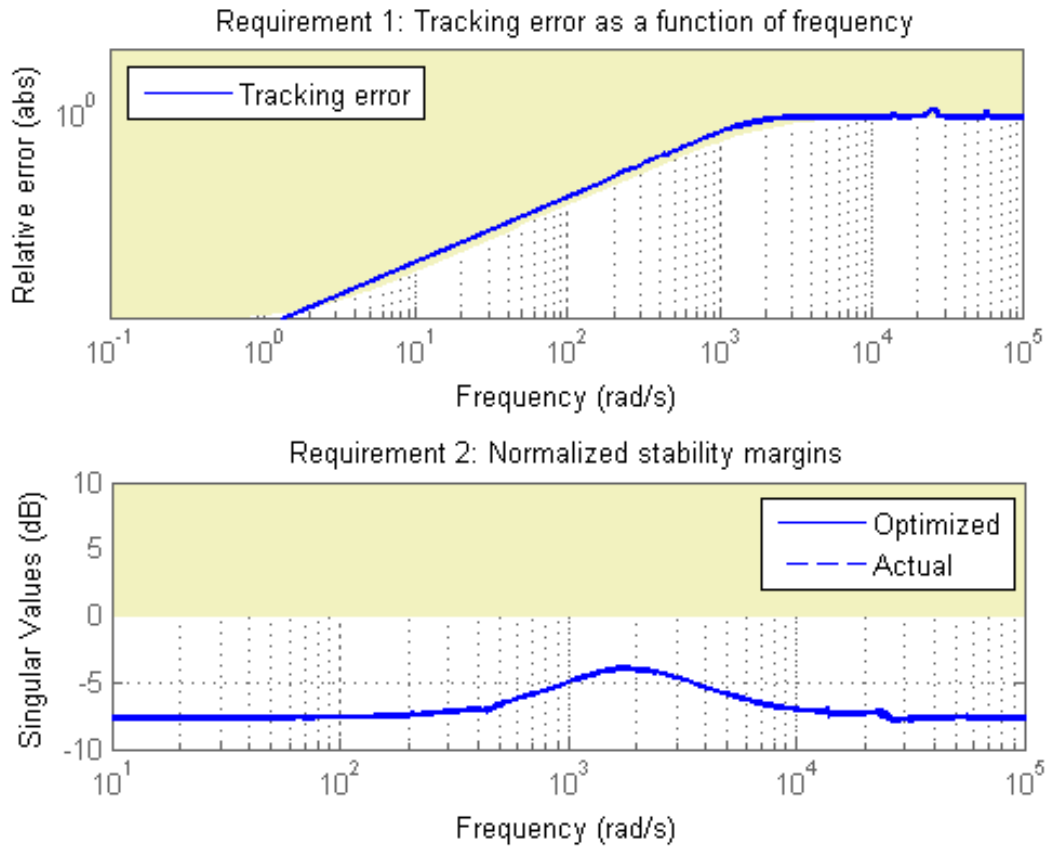
```
C =
    Kp + Ki * ---
                s
with Kp = 0.00104, Ki = 0.0122
```

```
Name: C
Continuous-time PI controller in parallel form.
-----
a = 3.19e+03
```

### **Validating Results**

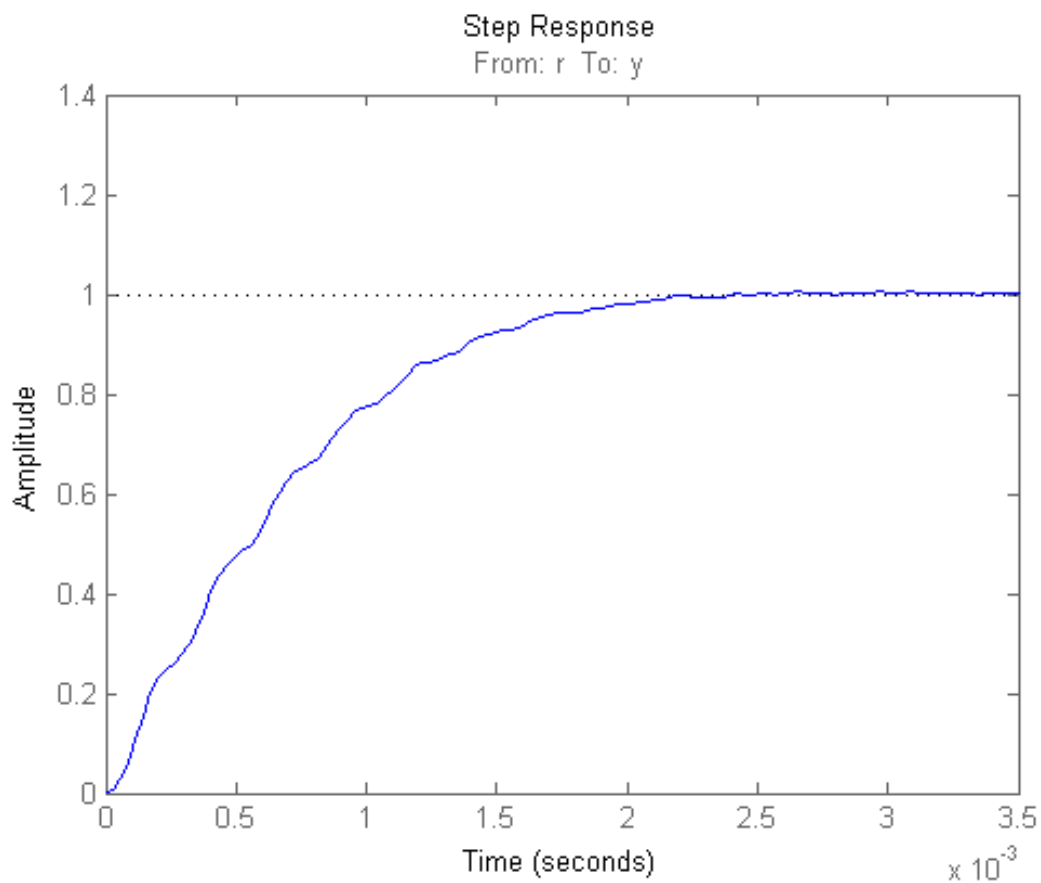
First use `viewSpec` to inspect how the tuned system does against each requirement. The first plot shows the tracking error as a function of frequency, and the second plot shows the normalized disk margins as a function of frequency (see `loopmargin`). See the "*Creating Design Requirements*" example for details.

```
clf, viewSpec([Req1 Req2],T,Info)
```



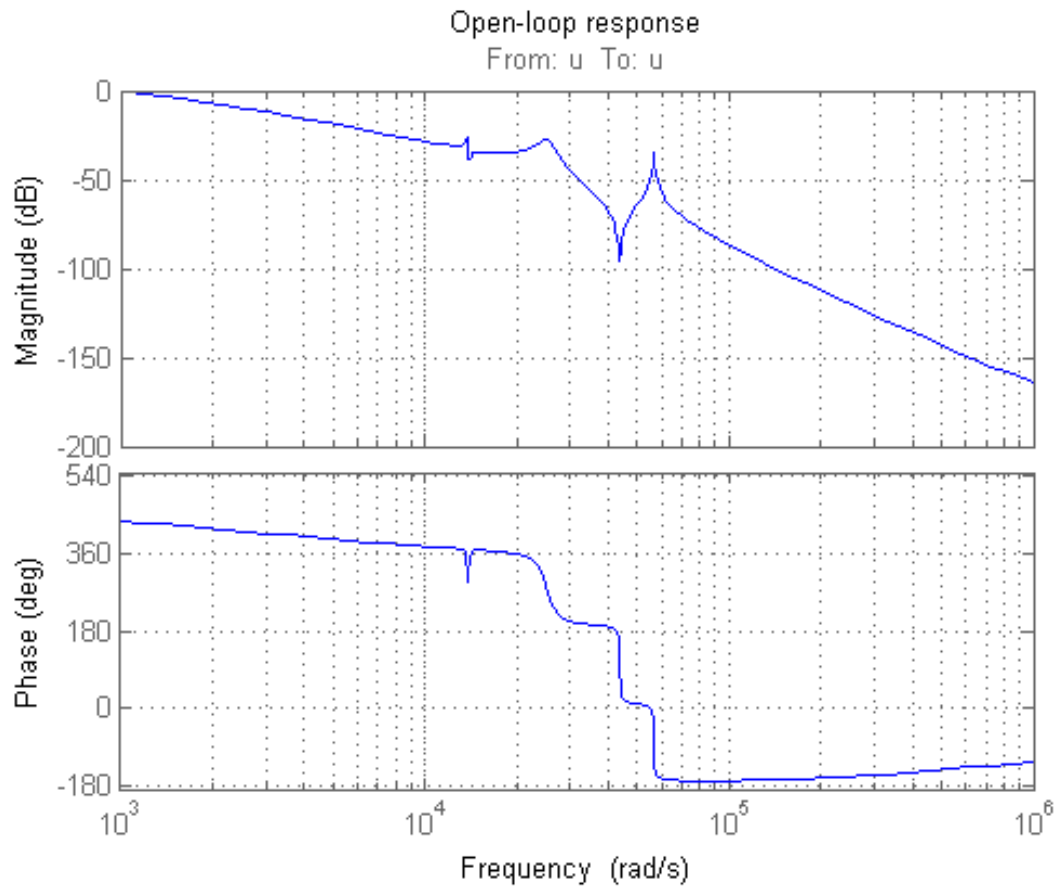
Next plot the closed-loop step response from reference  $r$  to head position  $y$ . The response has no overshoot but wobbles a little.

```
clf, step(T)
```



To investigate further, use `getLoopTransfer` to get the open-loop response at the plant input.

```
L = getLoopTransfer(T, 'u');  
bode(L, {1e3, 1e6}), grid  
title('Open-loop response')
```



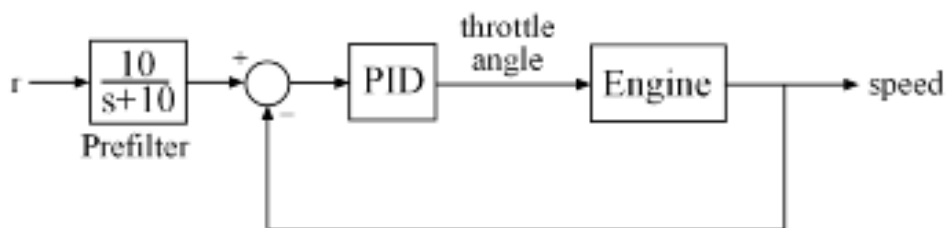
The wobble is due to the first resonance after the gain crossover. To eliminate it, you could add a notch filter to the feedback loop and tune its coefficients along with the lowpass coefficient and PI gains using `systemtune`.

## Tuning Feedback Loops with LOOPTUNE

This example shows the basic workflow of tuning feedback loops with the `looptune` command. `looptune` is similar to `sys­tune` and meant to facilitate loop shaping design by automatically generating the tuning requirements.

### Engine Speed Control

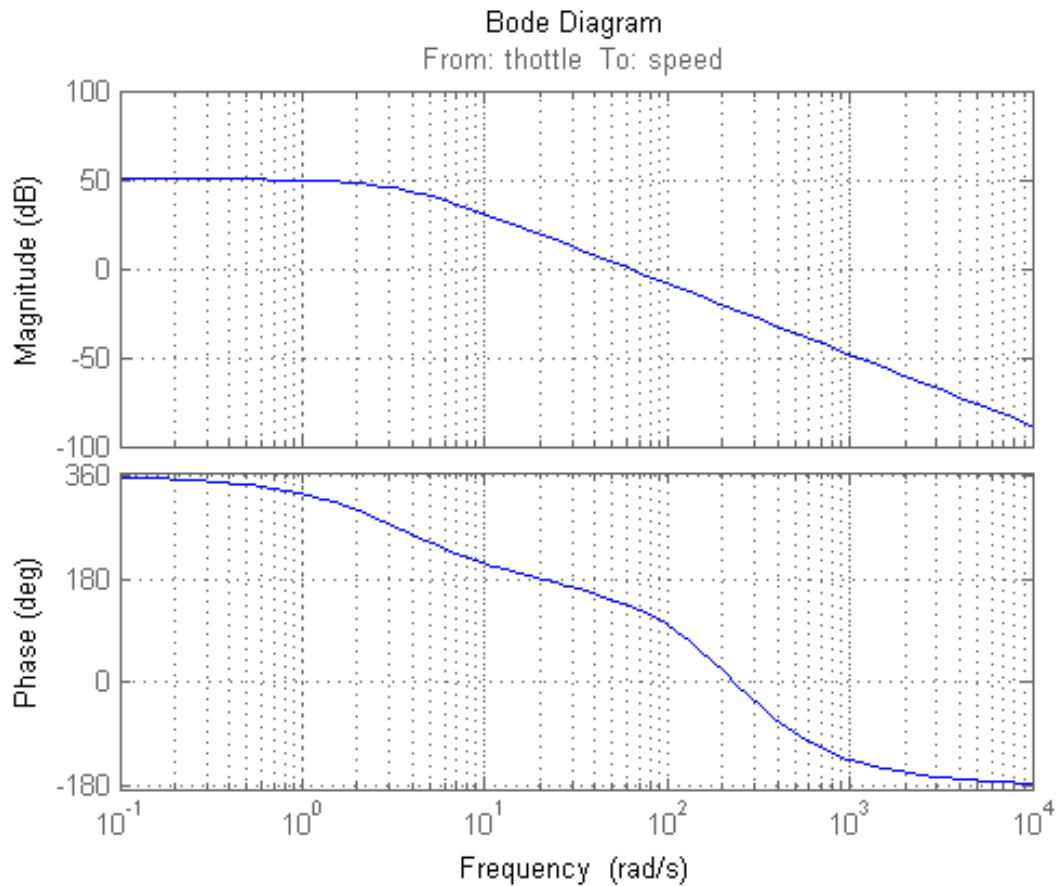
This example uses a simple engine speed control application as illustration. The control system consists of a single PID loop and the PID controller gains must be tuned to adequately respond to step changes in the desired speed. Specifically, we want the response to settle in less than 5 seconds with little or no overshoot.



**Figure 1: Engine Speed Control Loop**

We use the following fourth-order model of the engine dynamics.

```
load rctExamples Engine
bode(Engine), grid
```



### Specifying the Tunable Elements

We need to tune the four PID gains to achieve the desired performance. Use the `ltiblock.pid` class to parameterize the PID controller.

```
PIDO = ltiblock.pid('SpeedController','pid')
```

```
PIDO =
```

```
Parametric continuous-time PID controller "SpeedController" with formula:
```

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

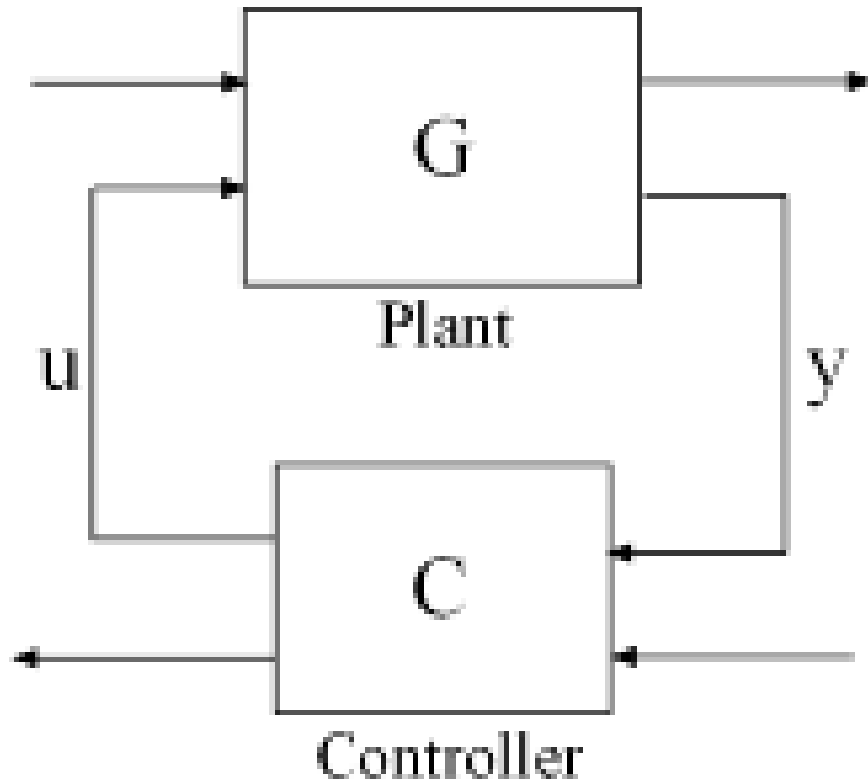
and tunable parameters  $K_p$ ,  $K_i$ ,  $K_d$ ,  $T_f$ .

Type "pid(PID0)" to see the current value and "get(PID0)" to see all properties.

### **Building a Tunable Model of the Feedback Loop**

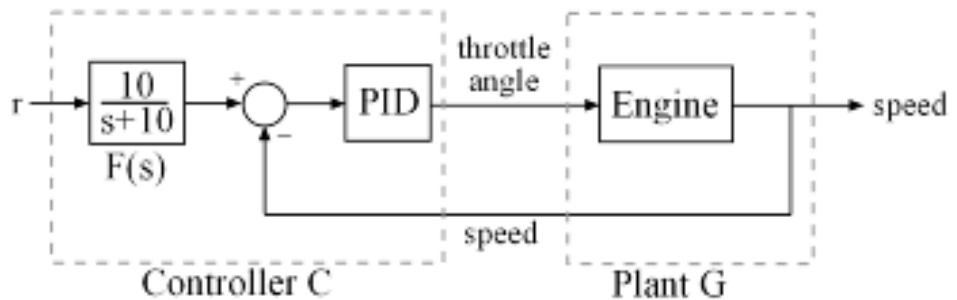
`looptune` tunes the generic SISO or MIMO feedback loop of Figure 2. This feedback loop models the interaction between the plant and the controller. Note that this is a *positive* feedback interconnection.





**Figure 2: Generic Feedback Loop**

For the speed control loop, the plant  $G$  is the engine model and the controller  $C$  consists of the PID and the prefilter  $F$ .



**Figure 3: Feedback Loop for Engine Speed Control**

To use `looptune`, create models for  $G$  and  $C$  in Figure 3. Assign names to the inputs and outputs of each model to specify the feedback paths between plant and controller. Note that the controller  $C$  has two inputs: the speed reference "r" and the speed measurement "speed".

```
F = tf(10,[1 10]); % prefilter

G = Engine;
G.InputName = 'throttle';
G.OutputName = 'speed';

C0 = PID0 * [F , -1];
C0.InputName = {'r','speed'};
C0.OutputName = 'throttle';
```

Here `C0` is a generalized state-space model (`genss`) that depends on the tunable PID block `PID0`.

### Tuning the Controller Parameters

You can now use `looptune` to tune the PID gains subject to a simple control bandwidth requirement. To achieve the 5-second settling time, the gain crossover frequency of the open-loop response should be approximately 1 rad/s. Given this basic requirement, `looptune` automatically shapes the open-loop response to provide integral action, high-frequency roll-off, and adequate stability margins. Note that you could specify additional

requirements to further constrain the design, see *"Decoupling Controller for a Distillation Column"* for an example.

```
wc = 1; % target gain crossover frequency
```

```
[~,C,~,Info] = looptune(G,C0,wc);
```

```
Final: Peak gain = 0.92, Iterations = 3
```

The final value is less than 1, indicating that the desired bandwidth was achieved with adequate roll-off and stability margins. `looptune` returns the tuned controller `C`. Use `getBlockValue` to retrieve the tuned value of the PID block.

```
PIDT = getBlockValue(C,'SpeedController')
```

```
PIDT =
```

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f s + 1}$$

```
with Kp = 0.000484, Ki = 0.00324, Kd = 0.000835, Tf = 1
```

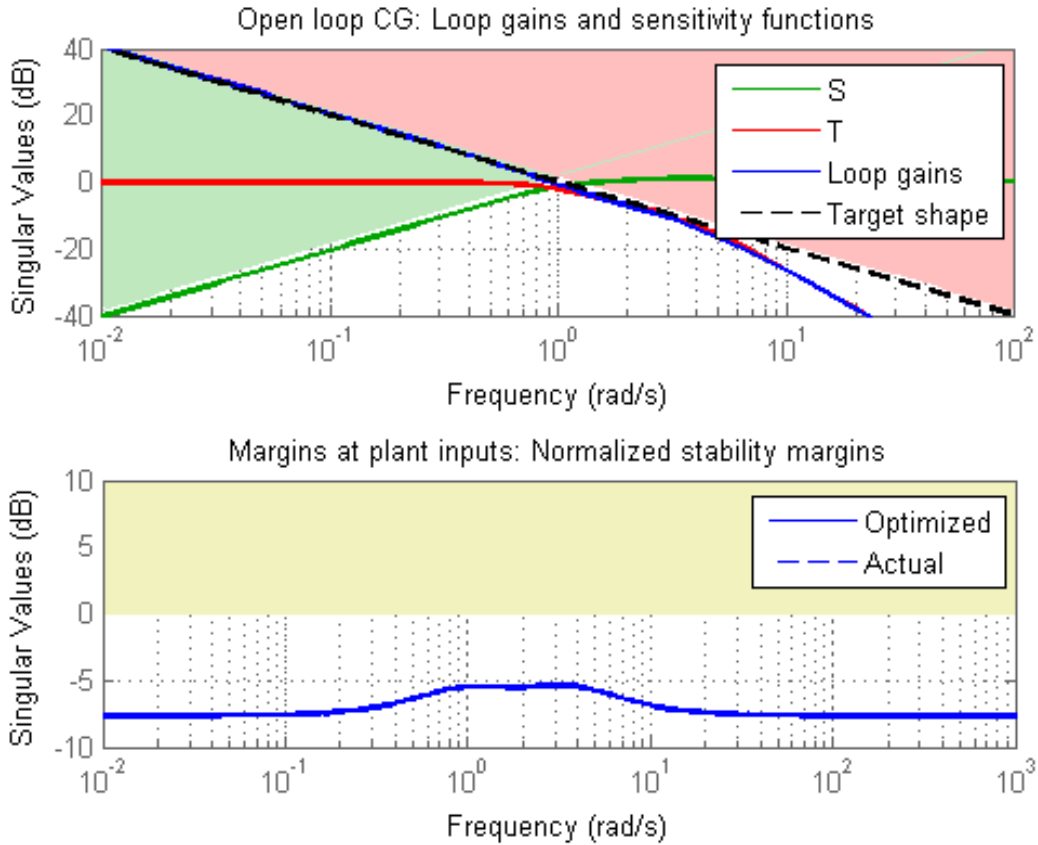
```
Name: SpeedController
```

```
Continuous-time PIDF controller in parallel form.
```

### Validating Results

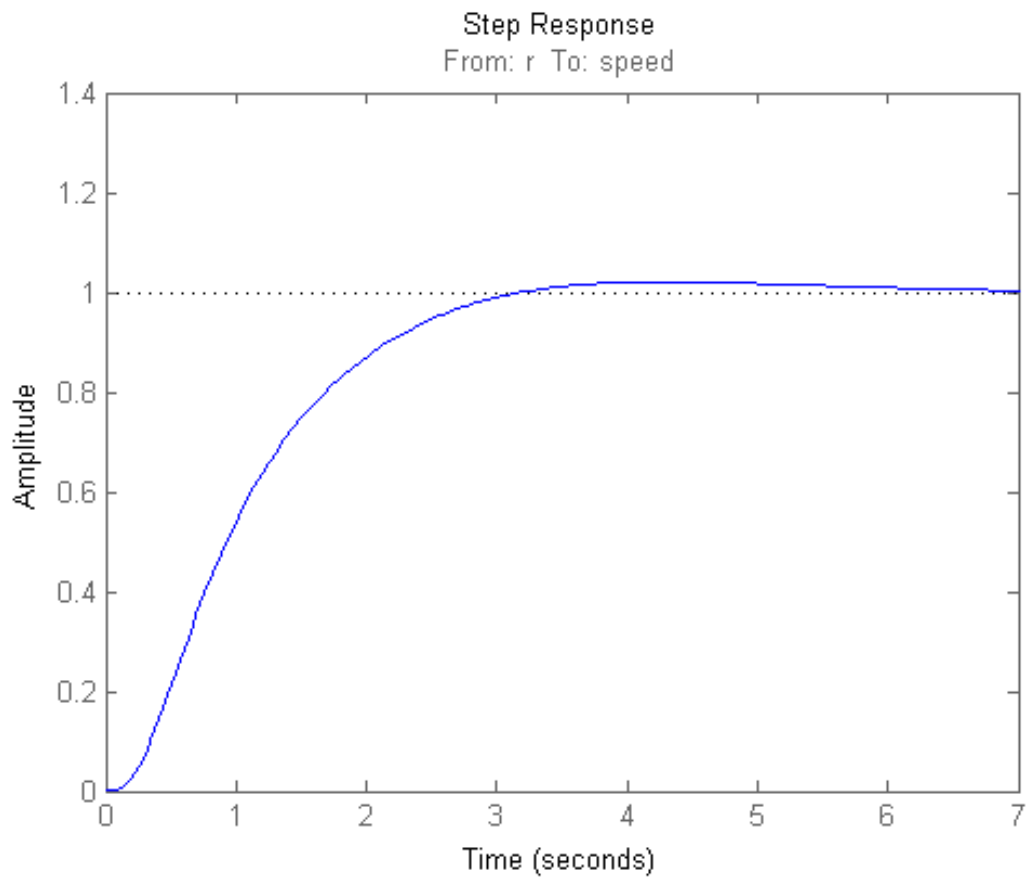
Use `loopview` to validate the design and visualize the loop shaping requirements implicitly enforced by `looptune`.

```
clf, loopview(G,C,Info)
```



Next plot the closed-loop response to a step command in engine speed. The tuned response satisfies our requirements.

```
T = connect(G,C,'r','speed'); % closed-loop transfer from r to speed
clf, step(T)
```



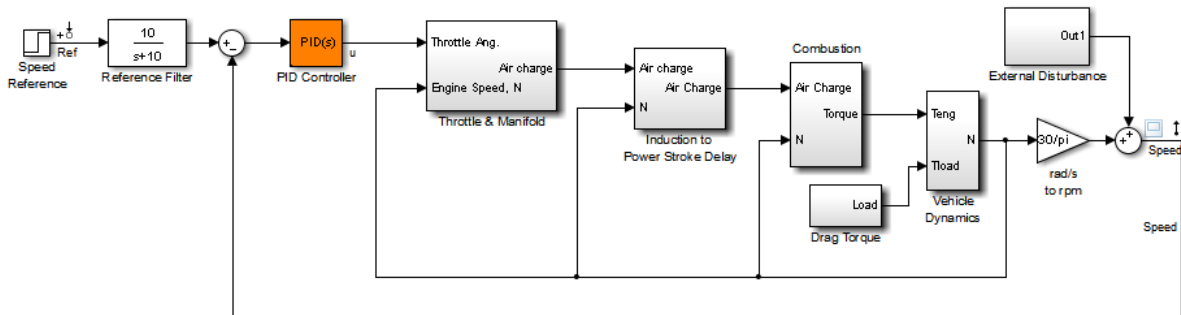
## Tuning Control Systems in Simulink

This example shows how to use `systemtune` or `looptune` to automatically tune control systems modeled in Simulink.

### Engine Speed Control

For this example we use the following model of an engine speed control system:

```
open_system('rct_engine_speed')
```



Copyright 2004-2010 The MathWorks, Inc.

The control system consists of a single PID loop and the PID controller gains must be tuned to adequately respond to step changes in the desired speed. Specifically, we want the response to settle in less than 5 seconds with little or no overshoot. While `pidtune` is a faster alternative for tuning a single PID controller, this simple example is well suited for an introduction to the `systemtune` and `looptune` workflows in Simulink.

### Controller Tuning with SYSTUNE

The `sITunable` interface provides a convenient gateway to `systemtune` for control systems modeled in Simulink. This interface lets you specify which blocks in the Simulink model are tunable and what signals are of interest for open- or closed-loop validation. Create an `sITunable` instance for the `rct_engine_speed` model and list the "PID Controller" block (highlighted in orange) as tunable. Note that all linearization points in the model (signals "Ref" and "Speed" here) are automatically available as analysis I/Os.

```
ST0 = sITunable('rct_engine_speed', 'PID Controller');
```

The PID block is initialized with its value in the Simulink model, which you can access using `getBlockValue`. Note that the proportional and derivative gains are initialized to zero.

```
ST0.getBlockValue('PID Controller')
```

```
ans =
```

$$Ki * \frac{1}{s}$$

```
with Ki = 0.01
```

```
Name: PID_Controller
```

```
Continuous-time I-only controller.
```

Next create a reference tracking requirement to capture the target settling time. Use the signal names in the Simulink model to refer to the reference and output signals, and use a two-second response time target to ensure settling in less than 5 seconds.

```
TrackReq = TuningGoal.Tracking('Ref', 'Speed', 2);
```

You can now tune the control system `ST0` subject to the requirement `TrackReq`.

```
ST1 = systune(ST0, TrackReq);
```

```
Final: Soft = 1.07, Hard = -Inf, Iterations = 121
```

The final value is close to 1 indicating that the tracking requirement is met. `systune` returns a "tuned" version `ST1` of the control system described by `ST0`. Again use `getBlockValue` to access the tuned values of the PID gains:

```
ST1.getBlockValue('PID Controller')
```

ans =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f * s + 1}$$

with Kp = 0.00188, Ki = 0.00336, Kd = 0.000475, Tf = 7.38e-05

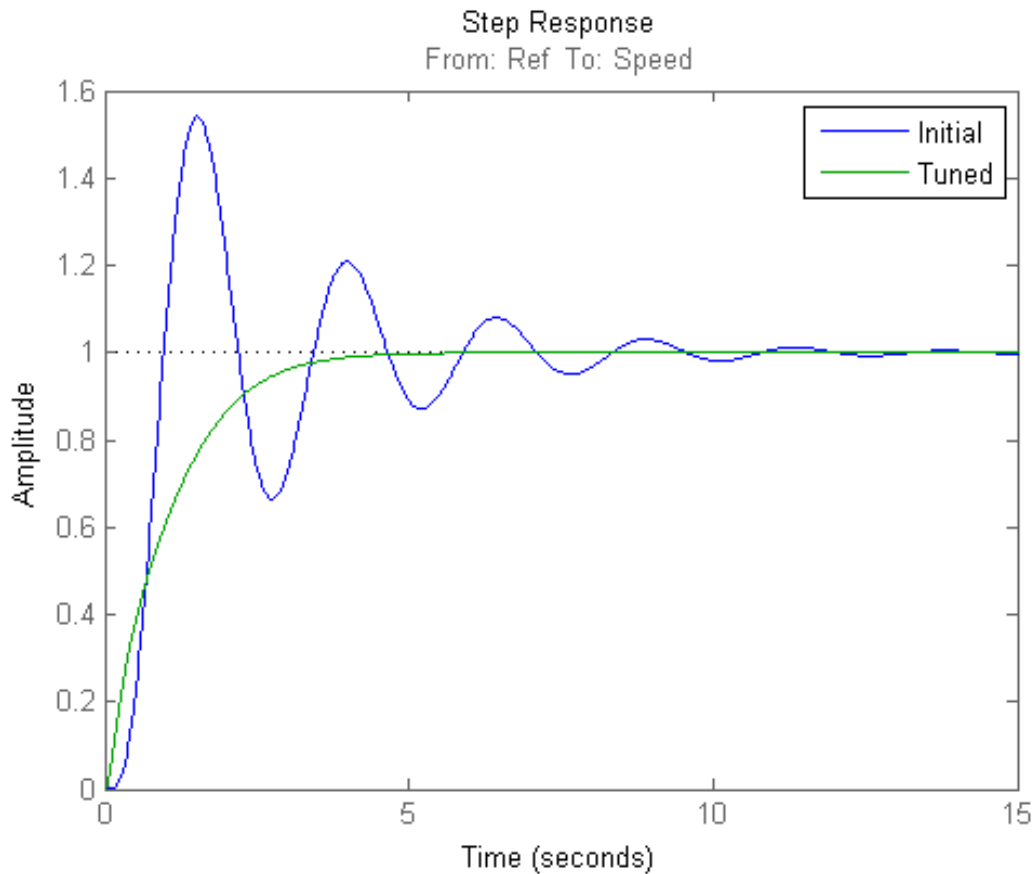
Name: PID\_Controller

Continuous-time PIDF controller in parallel form.

To simulate the closed-loop response to a step command in speed, get the initial and tuned transfer functions from speed command "Ref" to "Speed" output and plot their step responses:

```
T0 = ST0.getIOTransfer('Ref','Speed');
T1 = ST1.getIOTransfer('Ref','Speed');
step(T0,T1)
legend('Initial','Tuned')
```





### Controller Tuning with LOOPTUNE

You can also use `looptune` to tune control systems modeled in Simulink. The `looptune` workflow is very similar to the `systemtune` workflow. One difference is that `looptune` needs to know the boundary between the plant and controller, which is specified in terms of *controls* and *measurements* signals.

```
STO = slTunable('rct_engine_speed','PID Controller');  
  
% Specify plant/controller boundary  
STO.addControl('u');
```

```
ST0.addMeasurement('Speed');
```

```
% Add the "Ref" signal to the list of analysis I/Os  
ST0.addIO('Ref','input')
```

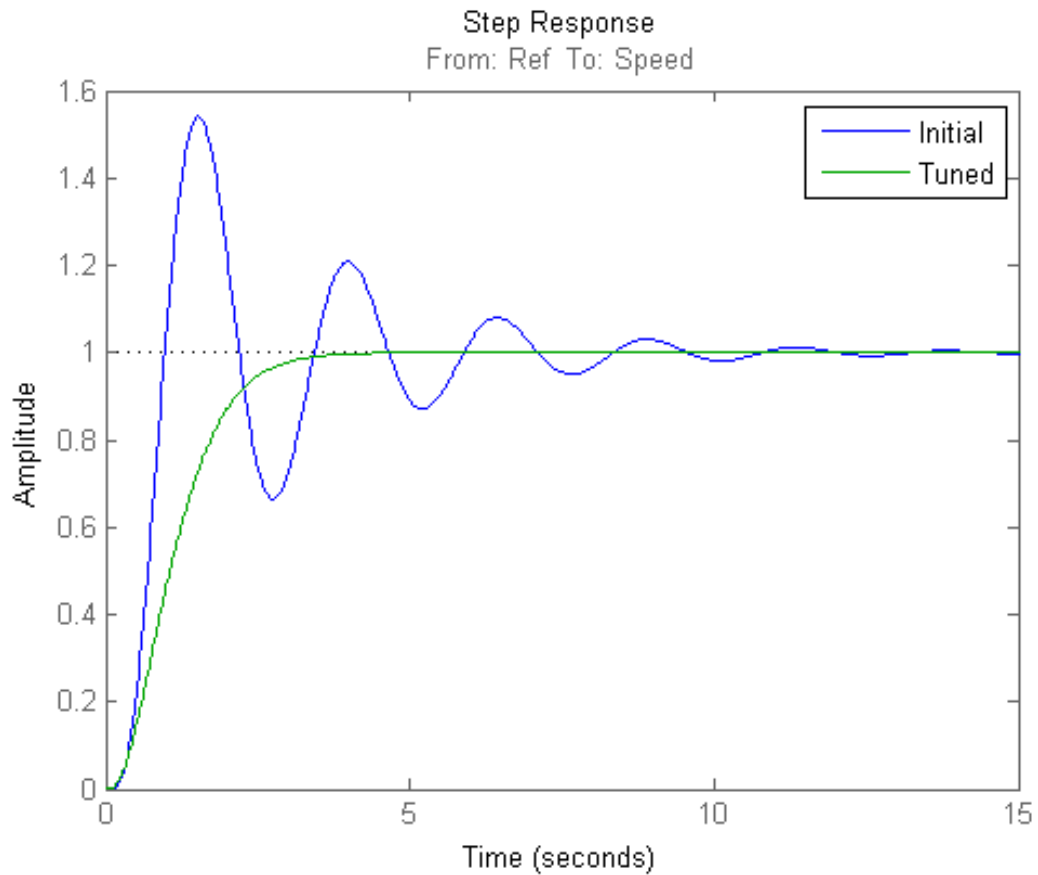
For a single loop the performance is essentially captured by the response time, or equivalently by the open-loop crossover frequency. Based on first-order characteristics the crossover frequency should exceed 1 rad/s for the closed-loop response to settle in less than 5 seconds. You can therefore tune the PID loop using 1 rad/s as target 0-dB crossover frequency.

```
wc = 1;  
ST1 = ST0.looptune(wc);
```

```
Final: Peak gain = 0.961, Iterations = 10
```

Again the final value is close to 1, indicating that the target control bandwidth was achieved. As with `systemtune`, use `getIOTransfer` to compute and plot the closed-loop response from speed command to actual speed. The result is very similar to that obtained with `systemtune`.

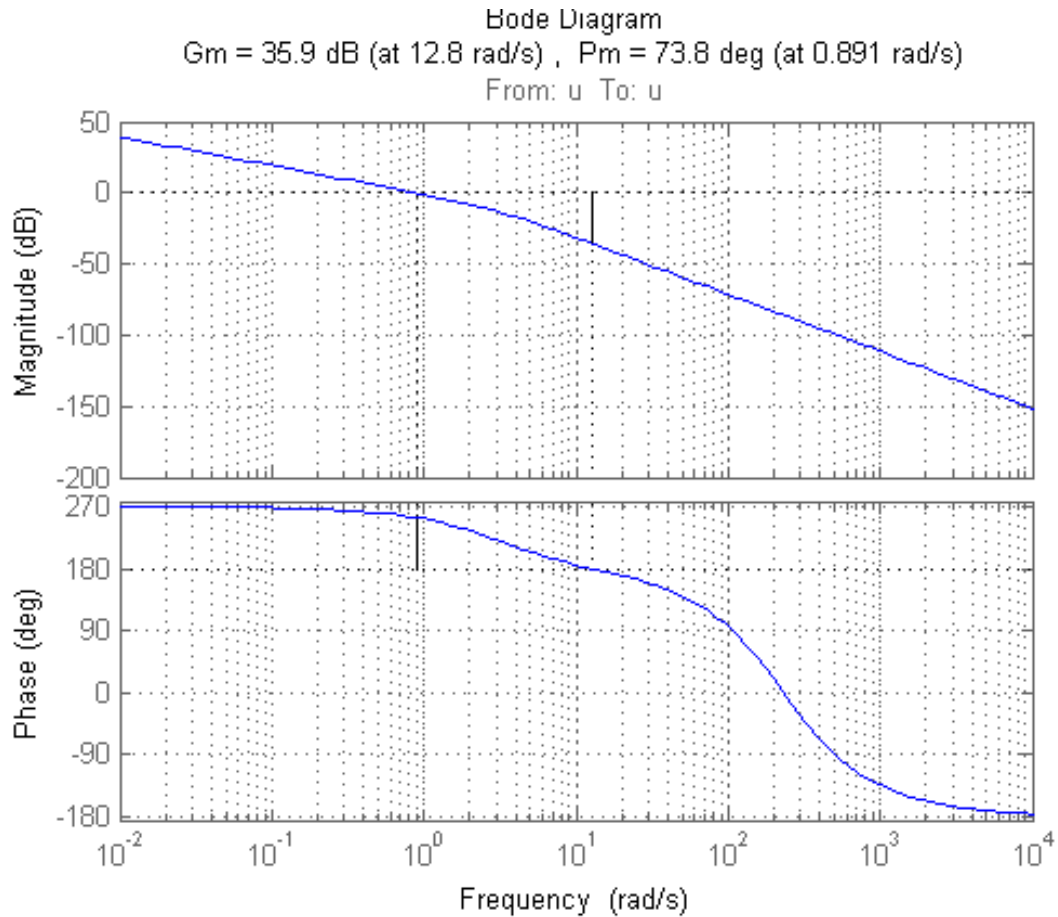
```
T0 = ST0.getIOTransfer('Ref','Speed');  
T1 = ST1.getIOTransfer('Ref','Speed');  
step(T0,T1)  
legend('Initial','Tuned')
```



You can also perform open-loop analysis, for example, compute the gain and phase margins at the plant input  $u$ .

```
% Note: -1 because |margin| expects the negative-feedback loop transfer  
L = ST1.getLoopTransfer('u',-1);
```

```
margin(L), grid
```

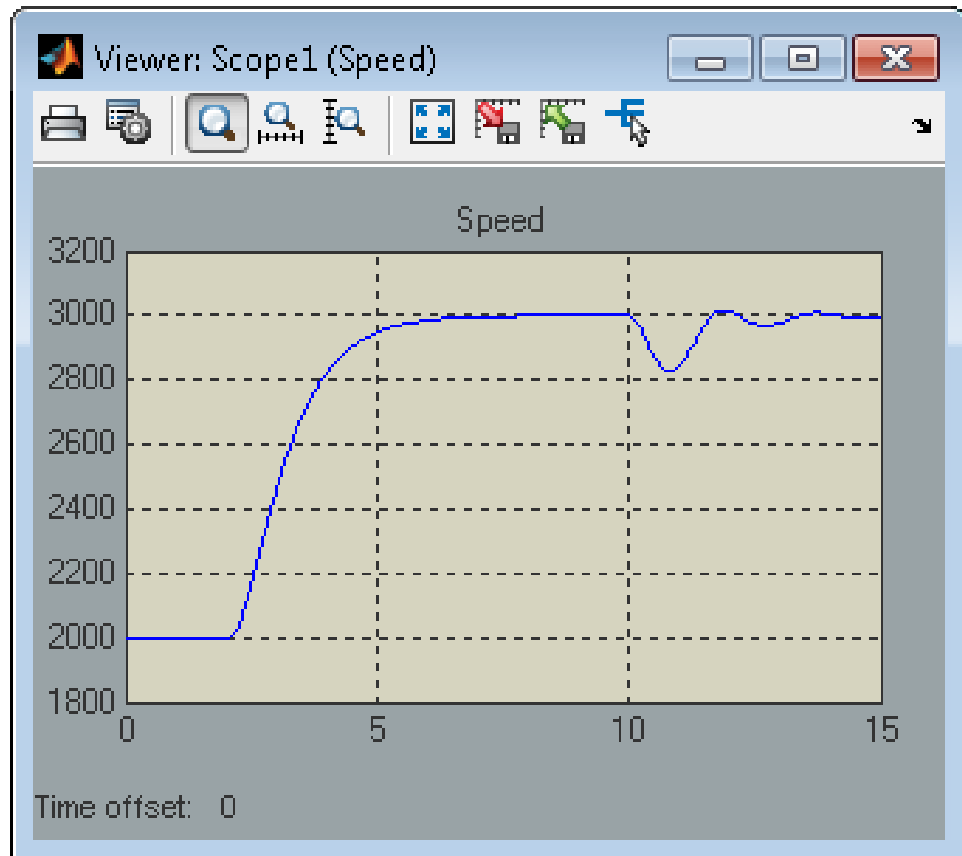


**Validation in Simulink**

Once you are satisfied with the systune or looptune results, you can upload the tuned controller parameters to Simulink for further validation with the nonlinear model.

```
ST1.writeBlockValue()
```

You can now simulate the engine response with the tuned PID controller.



The nonlinear simulation results closely match the linear responses obtained in MATLAB.

### Comparison of PI and PID Controllers

Closer inspection of the tuned PID gains suggests that the derivative term contributes little because of the large value of the  $T_f$  coefficient.

```
ST1.showTunable()
```

Block "rct\_engine\_speed/PID Controller" =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f * s + 1}$$

with  $K_p = 0.000653$ ,  $K_i = 0.00282$ ,  $K_d = 0.0021$ ,  $T_f = 46.7$

Name: PID\_Controller

Continuous-time PIDF controller in parallel form.

This suggests using a simpler PI controller instead. To do this, you need to override the default parameterization for the "PID Controller" block:

```
ST0.setBlockParam('PID Controller',ltiblock.pid('C','pi'))
```

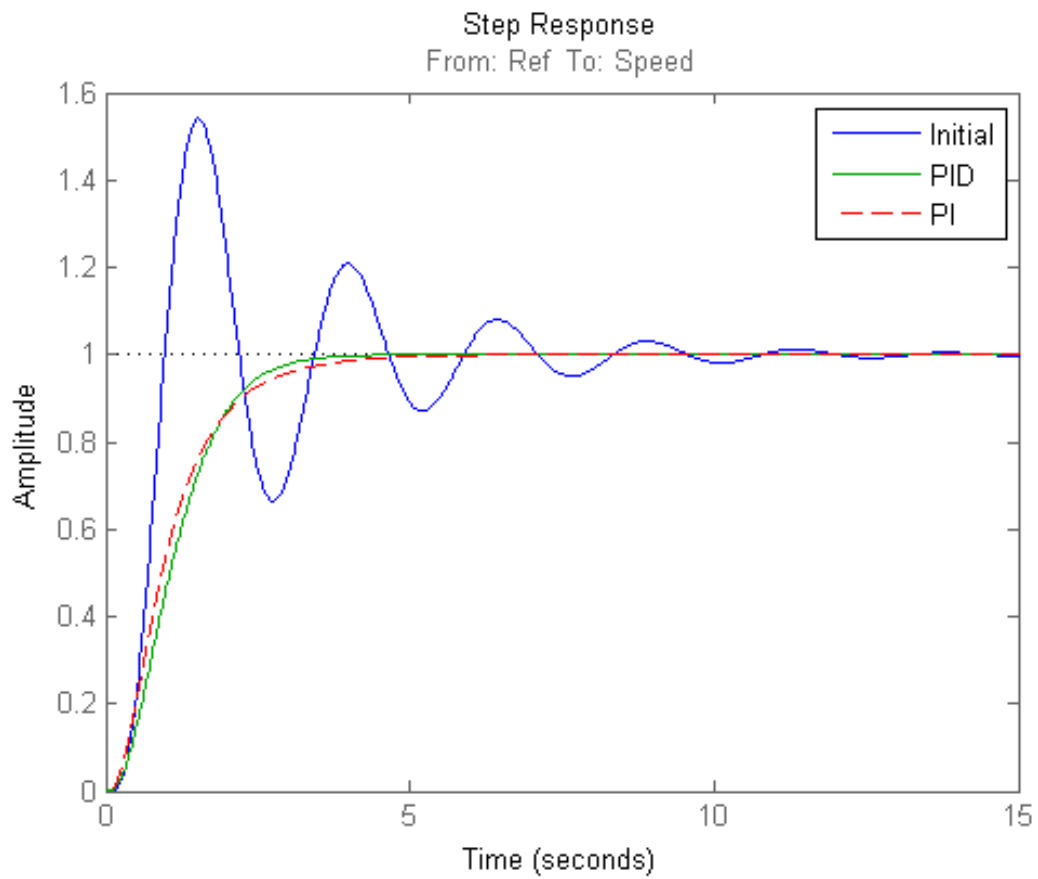
This specifies that the "PID Controller" block should now be parameterized as a mere PI controller. Next re-tune the control system for this simpler controller:

```
ST2 = ST0.looptune(wc);
```

Final: Peak gain = 0.915, Iterations = 5

Again the final value is less than one indicating success. Compare the closed-loop response with the previous ones:

```
T2 = ST2.getIOTransfer('Ref','Speed');
step(T0,T1,T2,'r--')
legend('Initial','PID','PI')
```



Clearly a PI controller is sufficient for this application.

## Building Tunable Models

This example shows how to create tunable models of control systems for use with `sysune` or `looptune`.

### Background

You can tune the gains and parameters of your control system with `sysune` or `looptune`. To use these commands, you need to construct a tunable model of the control system that identifies and parameterizes its tunable elements. This is done by combining numeric LTI models of the fixed elements with parametric models of the tunable elements.

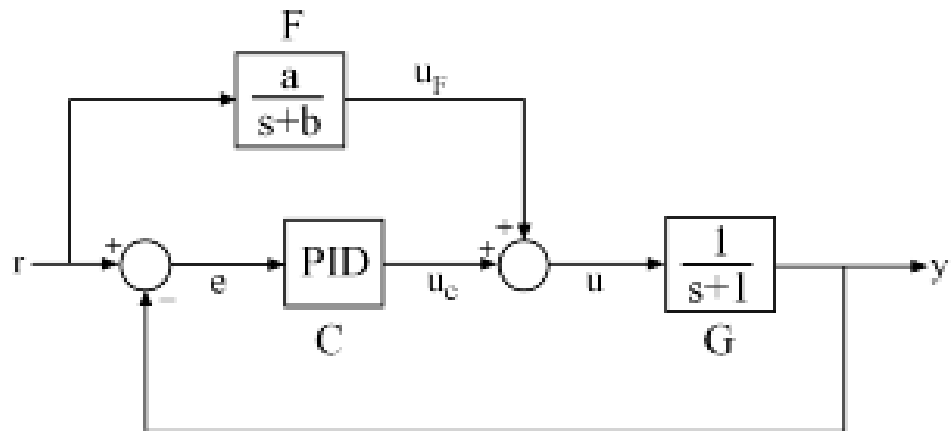
### Using Pre-Defined Tunable Elements

You can use one of the following "parametric" blocks to model commonly encountered tunable elements:

- **ltiblock.gain**: Tunable gain
- **ltiblock.pid**: Tunable PID controller
- **ltiblock.pid2**: Tunable two-degree-of-freedom PID controller
- **ltiblock.tf**: Tunable transfer function
- **ltiblock.ss**: Tunable state-space model.

For example, create a tunable model of the feedforward/feedback configuration of Figure 1 where  $C$  is a tunable PID controller and  $F$  is a tunable first-order transfer function.





**Figure 1: Control System with Feedforward and Feedback Paths**

First model each block in the block diagram, using suitable parametric blocks for  $C$  and  $F$ .

```
G = tf(1,[1 1]);
C = ltiblock.pid('C','pid'); % tunable PID block
F = ltiblock.tf('F',0,1);    % tunable first-order transfer function
```

Then use `connect` to build a model of the overall block diagram. To specify how the blocks are connected, label the inputs and outputs of each block and model the summing junctions using `sumblk`.

```
G.u = 'u'; G.y = 'y';
C.u = 'e'; C.y = 'uC';
F.u = 'r'; F.y = 'uF';

% Summing junctions
S1 = sumblk('e = r-y');
S2 = sumblk('u = uF + uC');

T = connect(G,C,F,S1,S2,'r','y')
```

T =

Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states.  
 C: Parametric PID controller, 1 occurrences.  
 F: Parametric SISO transfer function, 0 zeros, 1 poles, 1 occurrences.

Type "ss(T)" to see the current value, "get(T)" to see all properties, and "set(T)" to change the current value.

This creates a generalized state-space model T of the closed-loop transfer function from r to y. This model depends on the tunable blocks C and F and you can use `systemtune` to automatically tune the PID gains and feedforward coefficients a,b subject to your performance requirements. Use `showTunable` to see the current value of the tunable blocks.

`showTunable(T)`

C =

$$K_i * \frac{1}{s}$$

with  $K_i = 0.001$

Name: C

Continuous-time I-only controller.

F =

$$\frac{10}{s + 10}$$

Name: F

Continuous-time transfer function.

### Interacting with the Tunable Parameters

You can adjust the parameterization of the tunable elements  $C$  and  $F$  by interacting with the objects `C` and `F`. Use `get` to see their list of properties.

```
get(C)
```

```

        Kp: [1x1 param.Continuous]
        Ki: [1x1 param.Continuous]
        Kd: [1x1 param.Continuous]
        Tf: [1x1 param.Continuous]
    IFormula: ''
    DFormula: ''
        Ts: 0
    TimeUnit: 'seconds'
    InputName: {'e'}
    InputUnit: {''}
    InputGroup: [1x1 struct]
    OutputName: {'uC'}
    OutputUnit: {''}
    OutputGroup: [1x1 struct]
        Name: 'C'
        Notes: {}
    UserData: []

```

A PID controller has four tunable parameters  $K_p$ ,  $K_i$ ,  $K_d$ ,  $T_f$ , and `C` contains a description of each of these parameters. Parameter attributes include current value, minimum and maximum values, and whether the parameter is free or fixed.

```
C.Kp
```

```
ans =
```

```

        Name: 'Kp'
        Value: 0
    Minimum: -Inf
    Maximum: Inf
        Free: 1
        Scale: 1

```

```
Info: [1x1 struct]
```

```
1x1 param.Continuous
```

Set the corresponding attributes to override defaults. For example, you can fix the time constant `Tf` to the value 0.1 by

```
C.Tf.Value = 0.1;  
C.Tf.Free = false;
```

### Creating Custom Tunable Elements

For tunable elements not covered by the pre-defined blocks listed above, you can create your own parameterization in terms of elementary real parameters (`realp`). Consider the low-pass filter

$$F(s) = \frac{a}{s + a}$$

where the coefficient  $a$  is tunable. To model this tunable element, create a real parameter  $a$  and define  $F$  as a transfer function whose numerator and denominator are functions of  $a$ . This creates a generalized state-space model  $F$  of the low-pass filter parameterized by the tunable scalar  $a$ .

```
a = realp('a',1); % real tunable parameter, initial value 1  
F = tf(a,[1 a])
```

```
F =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 1  
a: Scalar parameter, 2 occurrences.
```

```
Type "ss(F)" to see the current value, "get(F)" to see all properties, and
```

Similarly, you can use real parameters to model the notch filter

$$N(s) = \frac{s^2 + 2\zeta_1\omega_n s + \omega_n^2}{s^2 + 2\zeta_2\omega_n s + \omega_n^2}$$

with tunable coefficients  $\omega_n, \zeta_1, \zeta_2$ .

```
wn = realp('wn',100);
zeta1 = realp('zeta1',1);    zeta1.Maximum = 1;    % zeta1 <= 1
zeta2 = realp('zeta2',1);    zeta2.Maximum = 1;    % zeta2 <= 1
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]); % tunable notch filter
```

You can also create tunable elements with matrix-valued parameters. For example, model the observer-based controller  $C(s)$  with equations

$$\frac{dx}{dt} = Ax + Bu + L(y - Cx), \quad u = -Kx$$

and tunable gain matrices  $K$  and  $L$ .

```
% Plant with 6 states, 2 controls, 3 measurements
[A,B,C] = ssdata(rss(6,3,2));
```

```
K = realp('K',zeros(2,6));
L = realp('L',zeros(6,3));
```

```
C = ss(A-B*K-L*C,L,-K,0)
```

```
C =
```

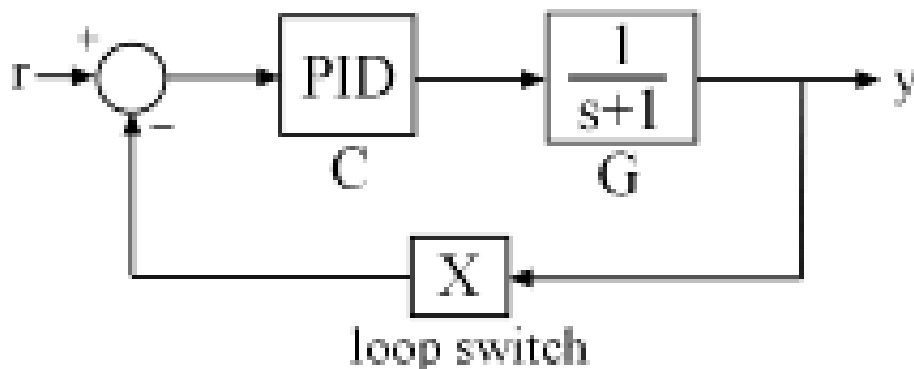
```
Generalized continuous-time state-space model with 2 outputs, 3 inputs, 6
  K: Parametric 2x6 matrix, 2 occurrences.
  L: Parametric 6x3 matrix, 2 occurrences.
```

Type "ss(C)" to see the current value, "get(C)" to see all properties, and

### Enabling Open-Loop Requirements

The `system` command takes a closed-loop model of the overall control system, like the tunable model `T` built at the beginning of this example. Such models

do not readily support open-loop analysis or open-loop specifications such as loop shapes and stability margins. To gain access to open-loop responses, insert a loopswitch block as shown in Figure 2.



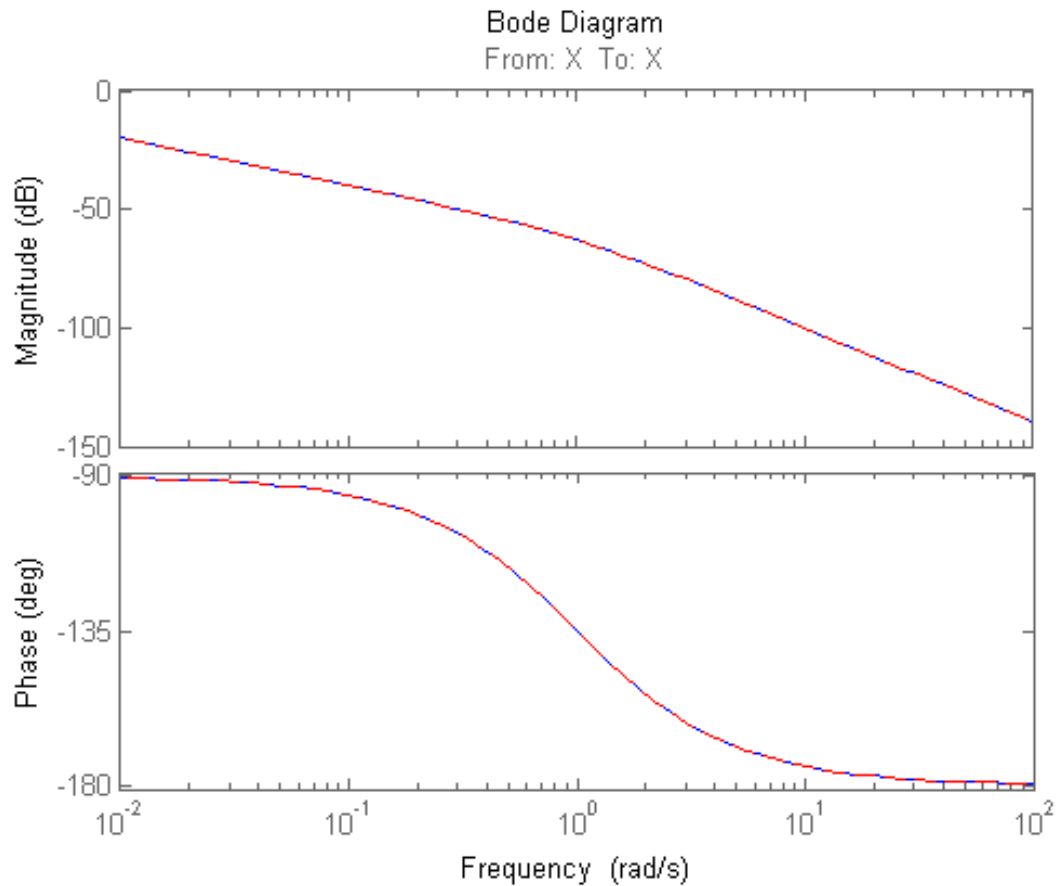
**Figure 2: Loop Switch Block**

The loopswitch block is an open/closed switch that can be used to open feedback loops or to measure open-loop responses. For example, construct a closed-loop model  $T$  of the feedback loop of Figure 2 where  $C$  is a tunable PID.

```
G = tf(1,[1 1]);
C = ltiblock.pid('C','pid');
LS = loopswitch('X');
T = feedback(G*C,LS);
```

By default the loop switch "X" is closed and  $T$  models the closed-loop transfer from  $r$  to  $y$ . However, you can now use `getLoopTransfer` to compute the (negative-feedback) loop transfer function measured at the location "X". Note that this loop transfer function is  $L = GC$  for the feedback loop of Figure 2.

```
L = getLoopTransfer(T,'X',-1); % loop transfer at "X"
clf, bode(L,'b',G*C,'r--')
```



You can also refer to the location "X" when specifying target loop shapes or stability margins for `system`. The requirement then applies to the loop transfer measured at this location.

```
% Target loop shape for loop transfer at "X"
Req1 = TuningGoal.LoopShape('X',tf(5,[1 0]));
```

```
% Target stability margins for loop transfer at "X"
Req2 = TuningGoal.Margins('X',6,40);
```

In general, the loop opening locations are specified in the `Location` property of `loopswitch` blocks. For single-loop switches, the default location is the block name. For multi-loop switches, the default locations are the block name followed by an index.

```
LS = loopswitch('Y',2); % two-channel switch
LS.Location
```

```
ans =
```

```
    'Y(1)'  
    'Y(2)'
```

You can override these default location names by modifying the `Location` property.

```
% Rename loop opening locations to "Inner" and "Outer".
LS.Location = {'Inner' ; 'Outer'};
LS.Location
```

```
ans =
```

```
    'Inner'  
    'Outer'
```



## Using Design Requirement Objects

This example gives a tour of available design requirements for control system tuning with `systune` or `looptune`.

### Background

The `systune` and `looptune` commands let you tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

### Tracking and Overshoot Requirements

The `TuningGoal.Tracking` requirement enforces setpoint tracking and loop decoupling objectives. For example

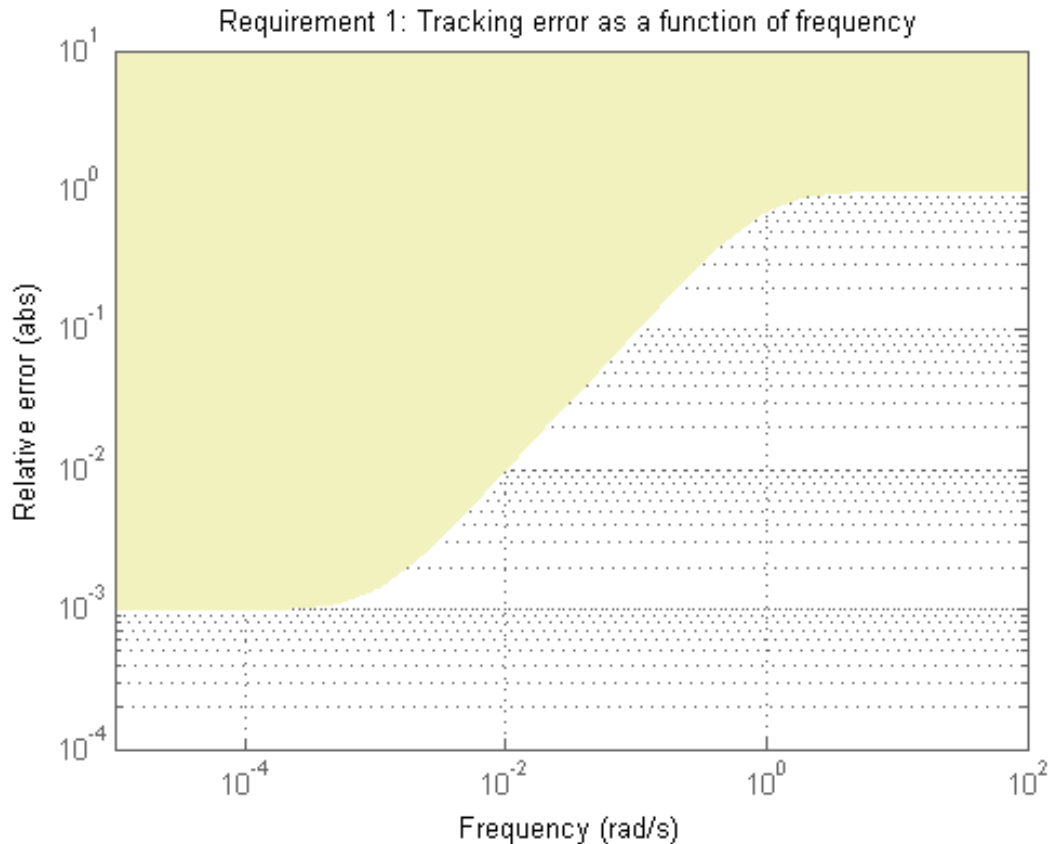
```
R1 = TuningGoal.Tracking('r','y',2);
```

specifies that the output `y` should track the reference `r` with a two-second response time. Similarly

```
R2 = TuningGoal.Tracking({'Vsp','wsp'},{'V','w'},2);
```

specifies that `V` should track `Vsp` and `w` should track `wsp` with minimum cross-coupling between the two responses. Tracking requirements are converted into frequency-domain constraints on the tracking error as a function of frequency. For the first requirement `R1`, for example, the gain from `r` to the tracking error  $e = r - y$  should be small at low frequency and approach 1 (100%) at frequencies greater than 1 rad/s (bandwidth for a two-second response time). You can use `viewSpec` to visualize this frequency-domain constraint. Note that the yellow region indicates where the requirement is violated.

```
viewSpec(R1)
```



If the step response has excessive overshoot, use the `TuningGoal.Overshoot` requirement in conjunction with the tracking requirement. For example, you can limit the overshoot from `r` to `y` to 5% using

```
R3 = TuningGoal.Overshoot('r','y',5);
```

### Disturbance Rejection Requirements

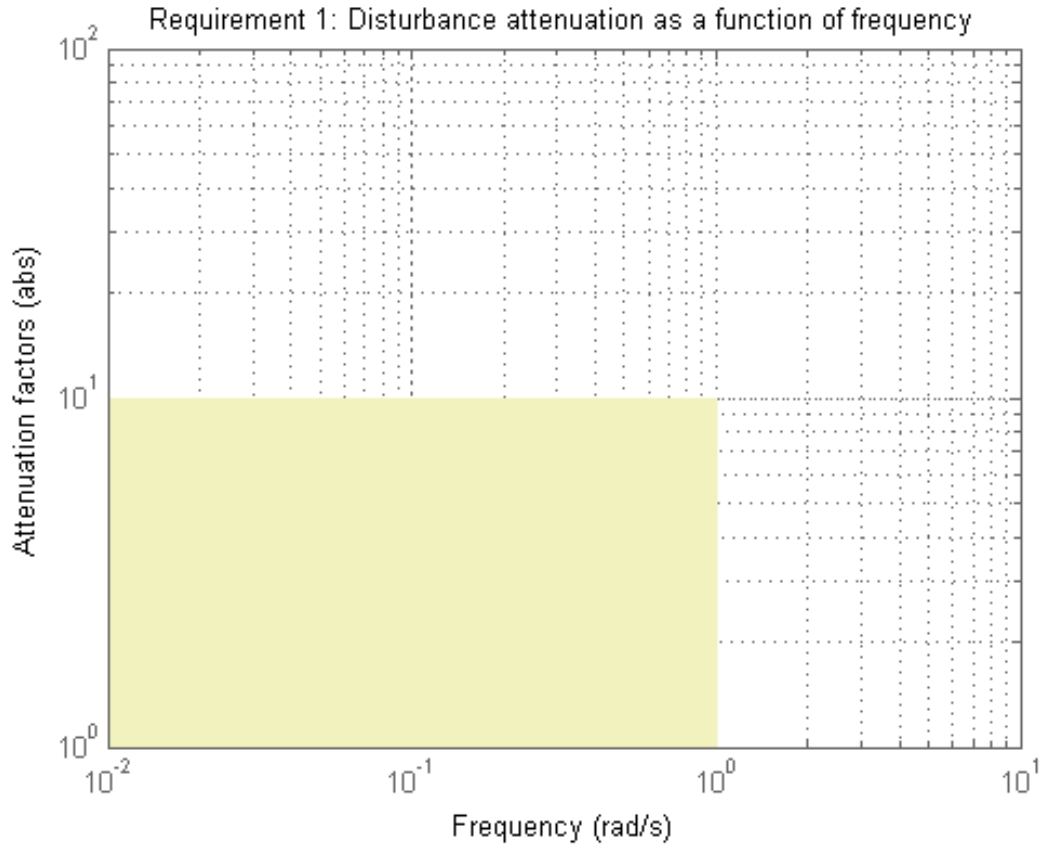
The `TuningGoal.Rejection` requirement enforces disturbance rejection objectives. This requirement specifies the disturbance attenuation as a

function of frequency. The attenuation factor is the ratio between the open- and closed-loop sensitivities to the disturbance (typically greater than one since feedback control reduces the impact of disturbances). As a rule of thumb, a 10-times-larger attenuation requires a 10-times-larger loop gain. For example

```
R1 = TuningGoal.Rejection('u',10);  
R1.Focus = [0 1];
```

specifies that a disturbance entering at the plant input "u" should be attenuated by a factor 10 in the frequency band from 0 to 1 rad/s.

```
viewSpec(R1)
```

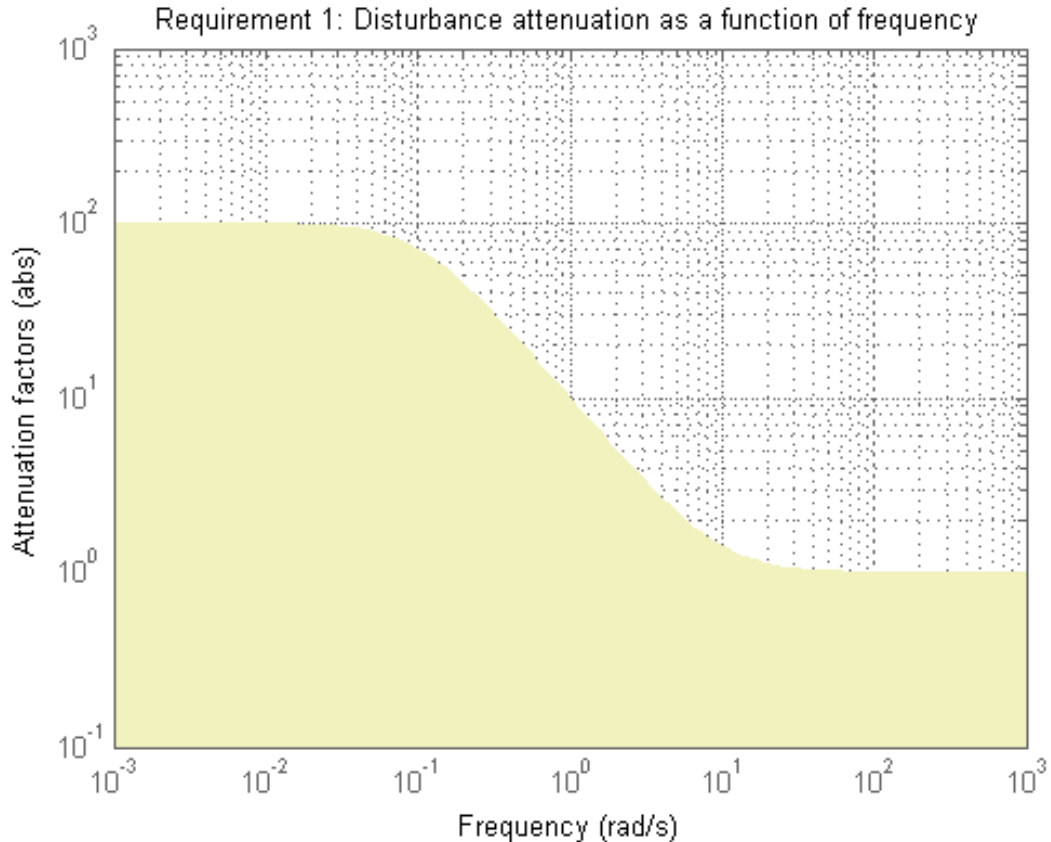


More generally, you can specify a frequency-dependent attenuation profile, for example

```
s = tf('s');  
R2 = TuningGoal.Rejection('u', (s+10)/(s+0.1));
```

specifies an attenuation factor of 100 below 0.1 rad/s gradually decreasing to 1 (no attenuation) after 10 rad/s.

```
viewSpec(R2)
```



In a feedback loop (see, for example, Figure 1), the open- and closed-loop responses from disturbance to output are related by

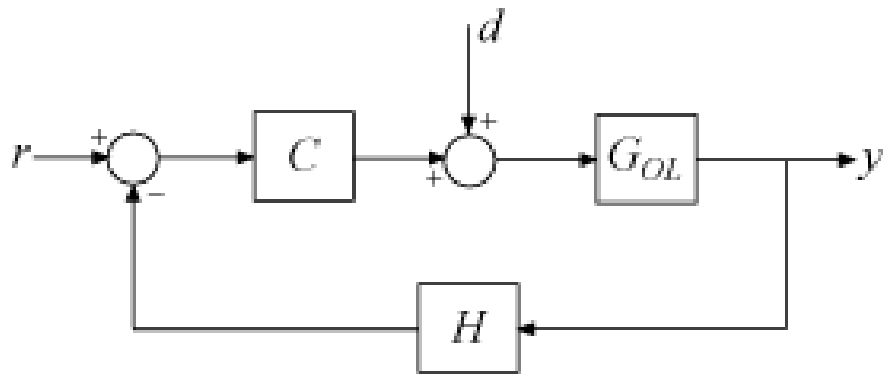
$$G_{CL}(s) = \frac{G_{OL}(s)}{1 + L(s)}$$

where  $L(s)$  is the loop transfer function measured at the disturbance entry point. The gain of  $1 + L$  is the disturbance attenuation factor and its reciprocal  $S = 1/(1 + L)$  is the sensitivity at the disturbance input.

Instead of specifying the minimum attenuation, you can specify the maximum sensitivity using the `TuningGoal.Sensitivity` requirement. For example,

```
R3 = TuningGoal.Sensitivity('u', (s+0.1)/(s+10));
```

is equivalent to the rejection requirement R2 above.



**Figure 1: Sample feedback loop.**

### Gain Requirement

The `TuningGoal.Gain` requirement enforces gain limits on SISO or MIMO closed-loop transfer functions. This requirement is useful to enforce adequate disturbance rejection and roll off, limit sensitivity and control effort, and prevent saturation. For MIMO transfer functions, "gain" refers to the largest singular value of the frequency response matrix. The gain limit can be frequency dependent. For example

```
R1 = TuningGoal.Gain('d', 'y', s/(s+1)^2);
```

specifies that the gain from  $d$  to  $y$  should not exceed the magnitude of the transfer function  $s/(s+1)^2$ .

```
viewSpec(R1)
```

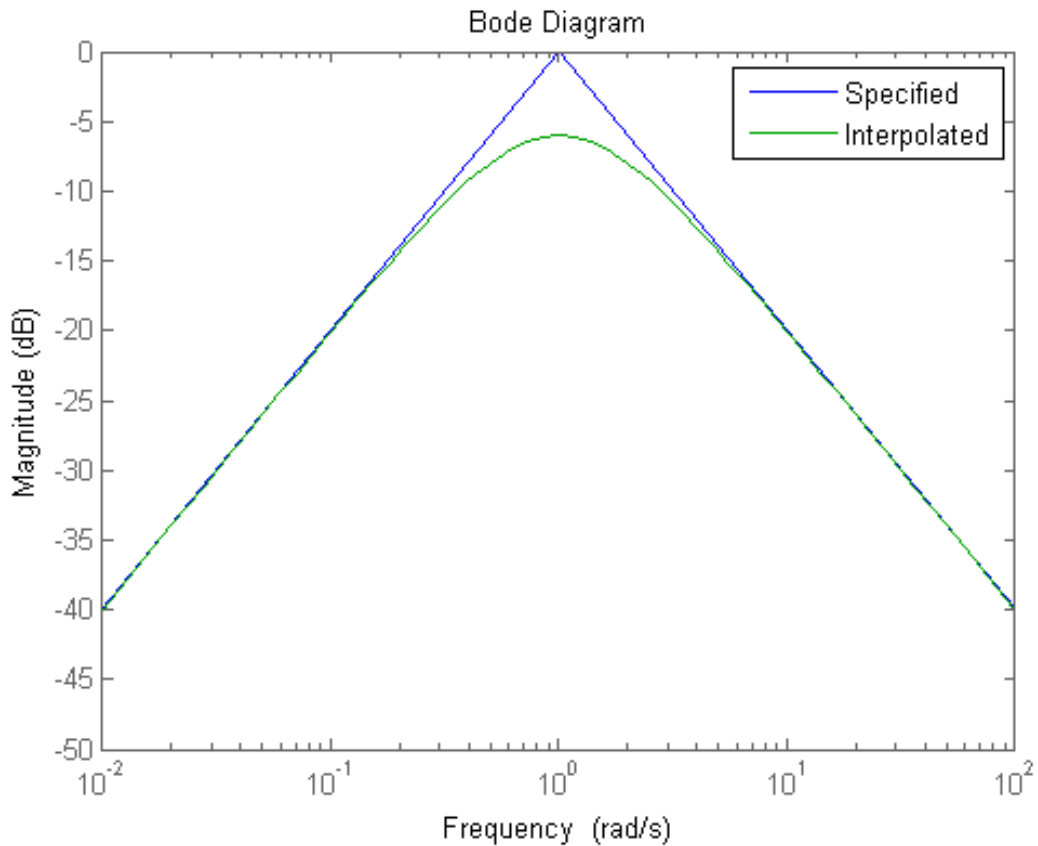


It is often convenient to just sketch the asymptotes of the desired gain profile. For example, instead of the transfer function  $s/(s+1)^2$ , we could just specify gain values of 0.01,1,0.01 at the frequencies 0.01,1,100, the point (1,1) being the breakpoint of the two asymptotes  $s$  and  $1/s$ .

```
Asymptotes = frd([0.01,1,0.01],[0.01,1,100]);
R2 = TuningGoal.Gain('d','y',Asymptotes);
```

The requirement object automatically turns this discrete gain profile into a gain limit defined at all frequencies.

```
bodemag(Asymptotes,R2.MaxGain)
legend('Specified','Interpolated')
```



### Variance Requirement

The `TuningGoal.Variance` requirement limits the noise variance amplification from specified inputs to specified outputs. In technical terms, this requirement constrains the  $H_2$  norm of a closed-loop transfer function.



This requirement is preferable to `TuningGoal.Gain` when the input signals are random processes and the average gain matters more than the peak gain. For example,

```
R = TuningGoal.Variance('n','y',0.1);
```

limits the output variance of `y` to  $0.1^2$  for a unit-variance white-noise input `n`.

### Weighted Gain and Weighted Variance Requirements

The `TuningGoal.WeightedGain` and `TuningGoal.WeightedVariance` requirements are generalizations of the `TuningGoal.Gain` and `TuningGoal.Variance` requirements. These requirements constrain the  $H_\infty$  or  $H_2$  norm of a frequency-weighted closed-loop transfer function  $W_L(s)T(s)W_R(s)$ , where  $W_L(s)$  and  $W_R(s)$  are user-defined weighting functions. For example

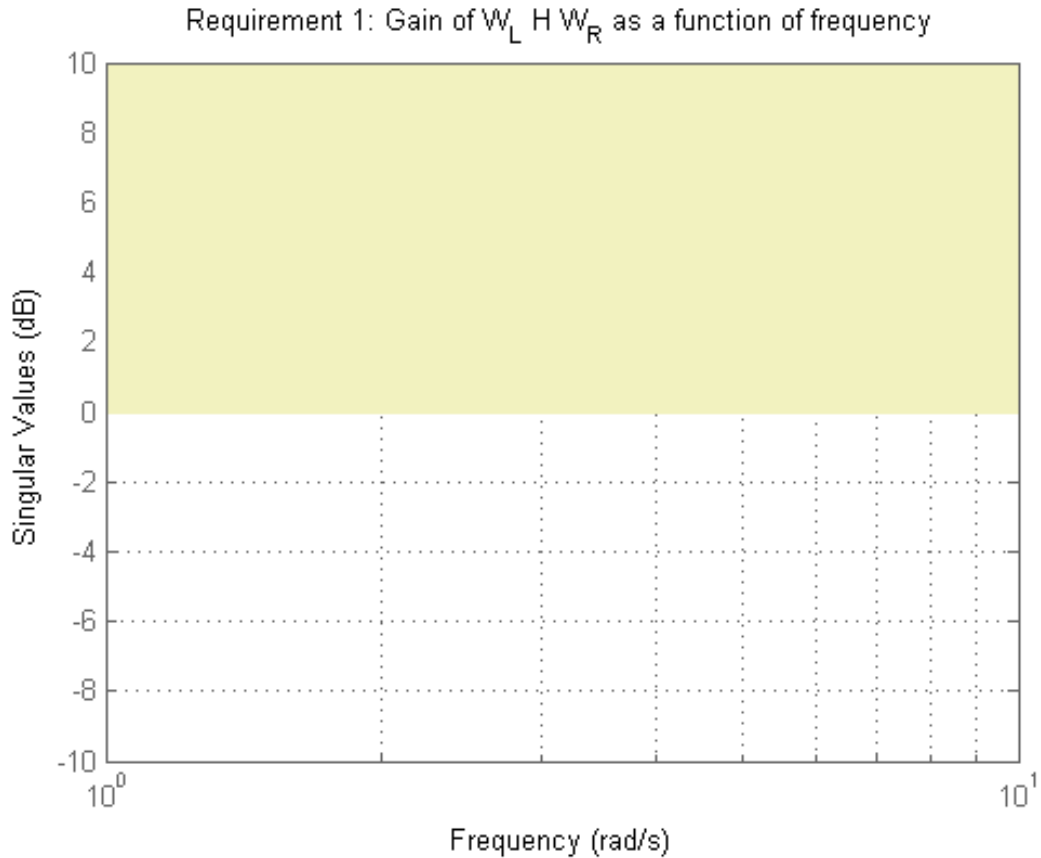
```
WL = blkdiag(1/(s+0.001),s/(0.001*s+1));
WR = [];
R = TuningGoal.WeightedGain('r',{'e','y'},WL,[]);
```

specifies the constraint

$$\left\| \begin{pmatrix} \frac{1}{s+0.001} T_{re} \\ \frac{s}{0.001s+1} T_{ry} \end{pmatrix} \right\|_\infty < 1$$

Note that this is a normalized gain constraint (unit bound across frequency).

```
viewSpec(R)
```



The `TuningGoal.WeightedVariance` requirement is useful to specify LQG-like performance objectives. For example, the LQG cost

$$J = \lim_{T \rightarrow \infty} E \left( \frac{1}{T} \int_0^T (x^T Q x + u^T R u) dt \right)$$

can be expressed as

$$J = \left\| \begin{pmatrix} W_z & 0 \\ 0 & W_u \end{pmatrix} T(s) \right\|_2^2$$

where  $T(s)$  is the closed-loop transfer from process and measurement noise to the states  $x$  and controls  $u$ , and  $W_z, W_u$  are Cholesky factors of  $Q$  and  $R$  (see `chol`).

### Loop Shape Requirement

The `TuningGoal.LoopShape` requirement is used to shape the open-loop response gain(s), a design approach known as *loop shaping*. For example,

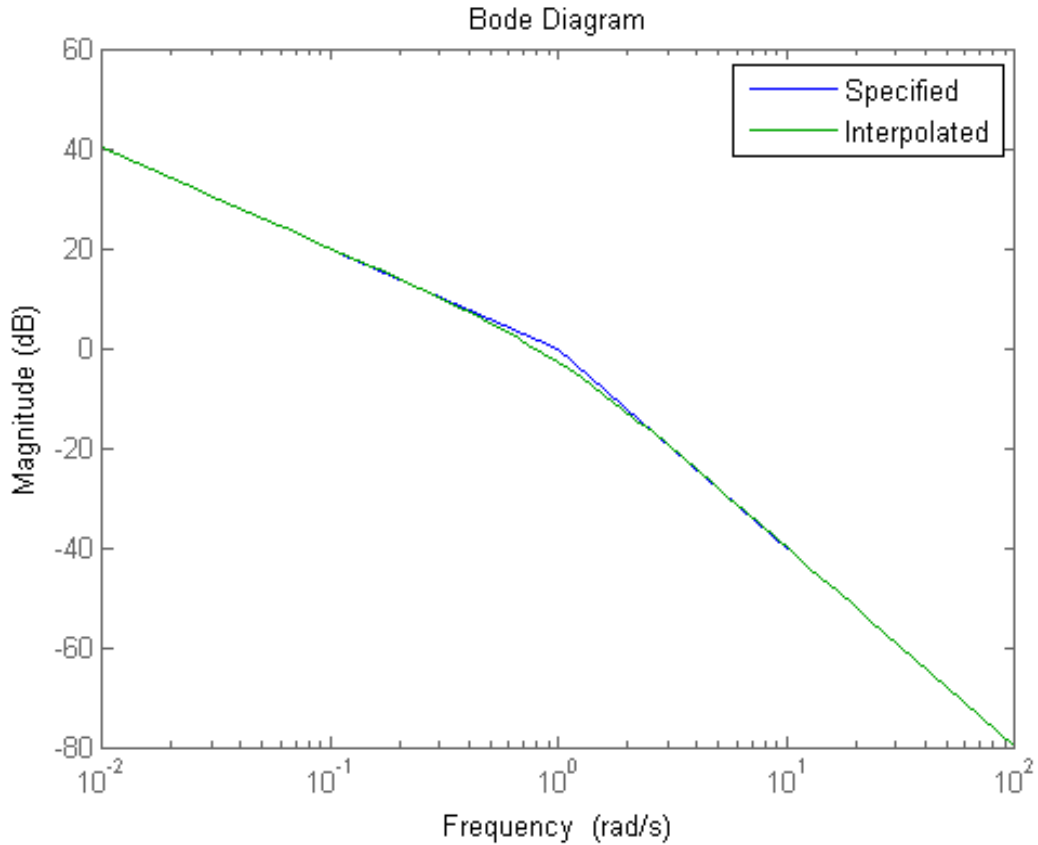
```
R1 = TuningGoal.LoopShape('u',1/s);
```

specifies that the open-loop response measured at the location "u" should look like a pure integrator (as far as its gain is concerned). In MATLAB, the location "u" should be marked using a `loopswitch` block, see the *"Building Tunable Models"* example for details. In Simulink, this location should be registered as control, measurement, or switch using the `addControl`, `addMeasurement`, `addSwitch` methods of the `sITunable` tuning interface.

As with other gain specifications, you can just specify the asymptotes of the desired loop shape using a few frequency points. For example, to specify a loop shape with gain crossover at 1 rad/s, -20 dB/decade slope before 1 rad/s, and -40 dB/decade slope after 1 rad/s, just specify that the gain at the frequencies 0.1,1,10 should be 10,1,0.01, respectively.

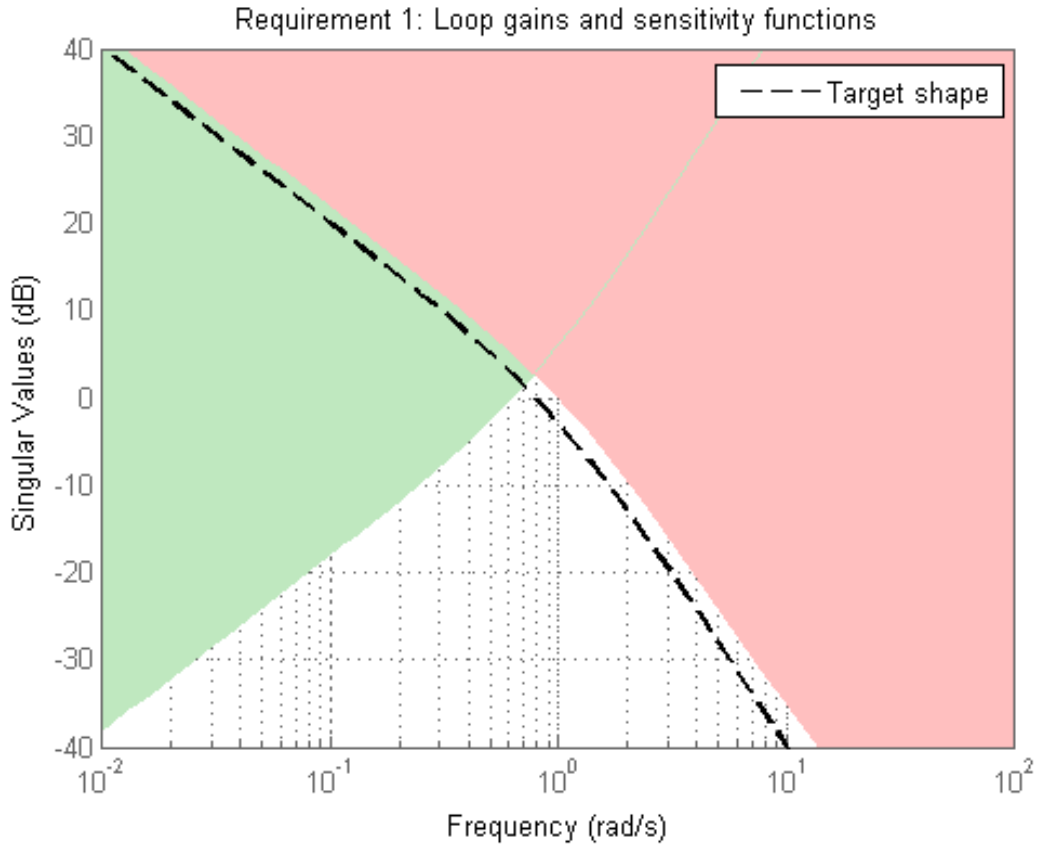
```
LS = frd([10,1,0.01],[0.1,1,10]);
R2 = TuningGoal.LoopShape('u',LS);
```

```
bodemag(LS,R2.LoopGain)
legend('Specified','Interpolated')
```



Loop shape requirements are constraints on the open-loop response  $L$ . For tuning purposes, they are converted into closed-loop gain constraints on the sensitivity function  $S = 1/(1 + L)$  and complementary sensitivity function  $T = L/(1 + L)$ . Use `viewSpec` to visualize the target loop shape and corresponding gain bounds on  $S$  (green) and  $T$  (red).

```
viewSpec(R2)
```



Instead of `TuningGoal.LoopShape`, you can use `TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain` to specify minimum or maximum values for the loop gain in a particular frequency band. This is useful when the actual loop shape near crossover is best left to the tuning algorithm to figure out. For example, the following requirements specify the minimum loop gain inside the bandwidth and the roll-off characteristics outside the bandwidth, but do not specify the actual crossover frequency nor the loop shape near crossover.

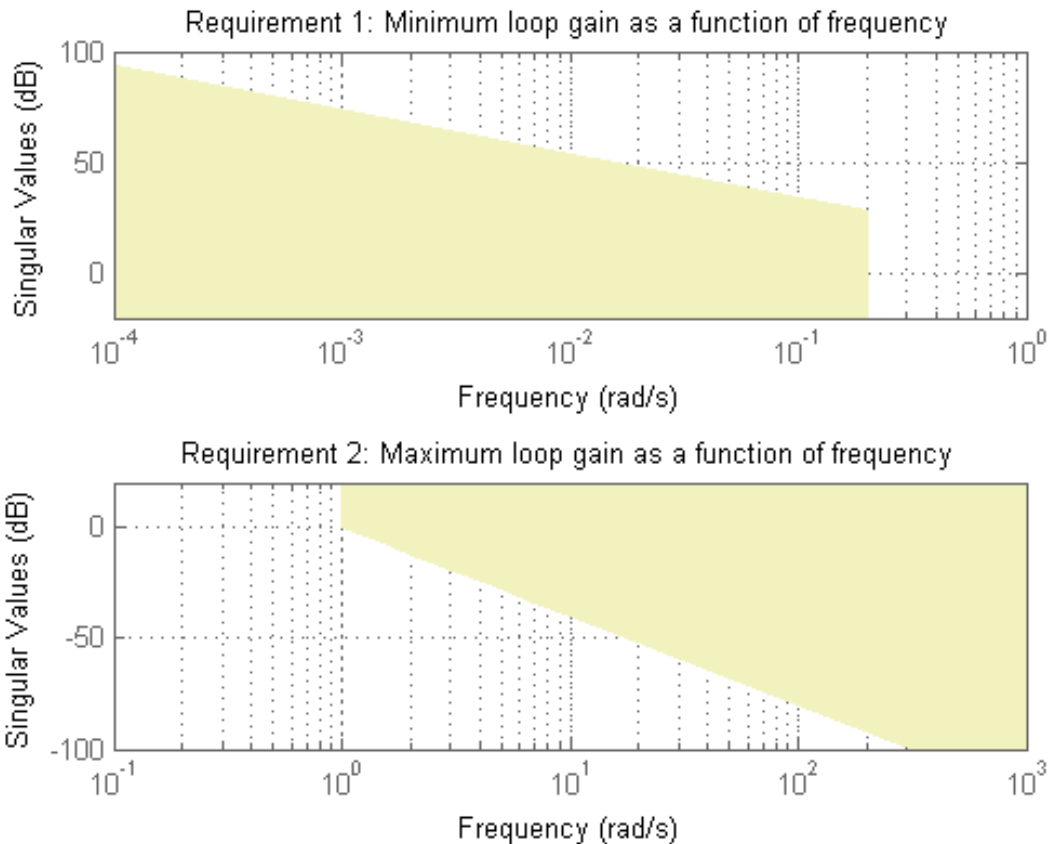
```
MinLG = TuningGoal.MinLoopGain('u',5/s); % integral action
MinLG.Focus = [0 0.2];
```

```

MaxLG = TuningGoal.MaxLoopGain('u',1/s^2); % -40dB/decade roll off
MaxLG.Focus = [1 Inf];

viewSpec([MinLG MaxLG])

```



### Stability Margins Requirement

The `TuningGoal.Margins` requirement enforces minimum amounts of gain and phase margins at the specified loop opening site(s). For MIMO feedback

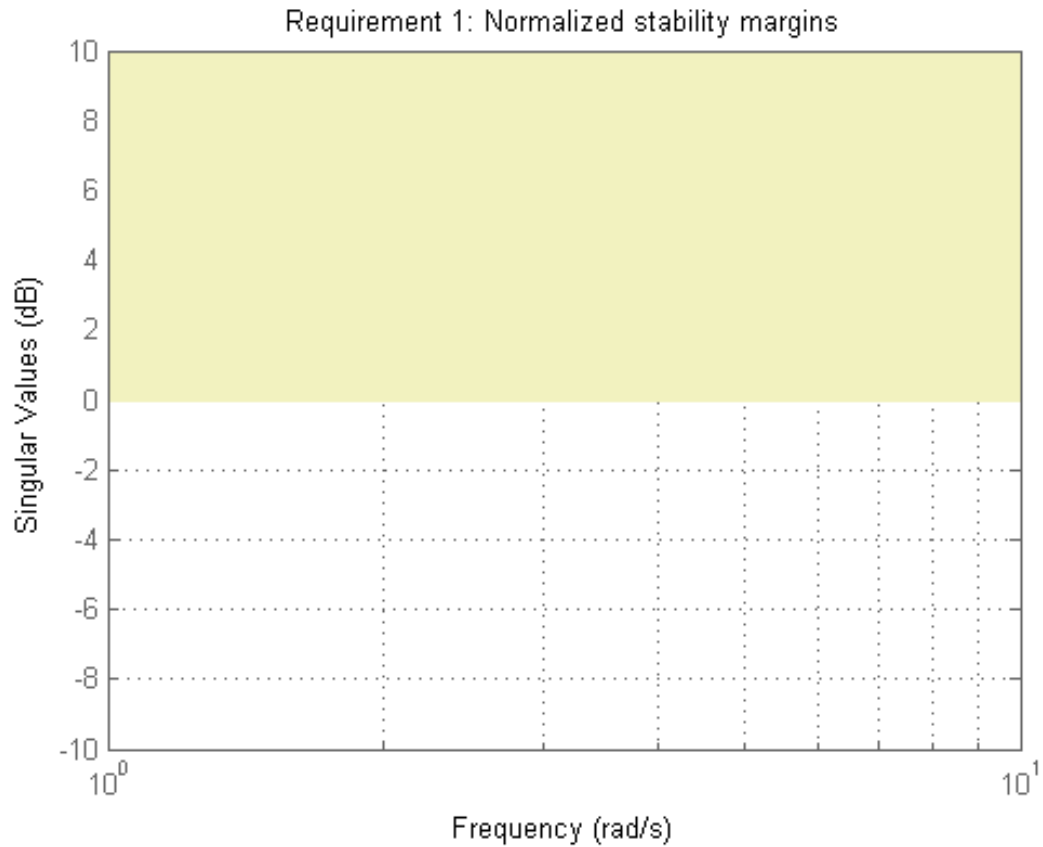
loops, this requirement uses the notion of *disk margins*, which guarantee stability for concurrent gain and phase variations of the specified amount in all feedback channels (see `loopmargin` for details). For example,

```
R = TuningGoal.Margins('u',6,45);
```

enforces  $\pm 6$  dB of gain margin and 45 degrees of phase margin at the location "u". In MATLAB, this location should be marked with a `loopswitch` block in the overall model of the control system. In Simulink, it should be registered as control, measurement, or switch using the `addControl`, `addMeasurement`, `addSwitch` methods of the `sITunable` tuning interface. Stability margins are typically measured at the plant inputs or plant outputs or both.

The target gain and phase margin values are converted into a normalized gain constraint on some appropriate closed-loop transfer function. The desired margins are achieved at frequencies where the gain is less than 1.

```
viewSpec(R)
```



#### Closed-Loop Pole Requirement

The `TuningGoal.Poles` requirement constrains the location of the closed-loop poles. You can enforce some minimum decay rate

$$\operatorname{Re}(s) < -\alpha,$$

impose some minimum damping ratio

$$\operatorname{Re}(s) < -\zeta|s|,$$



or constrain the pole magnitude to

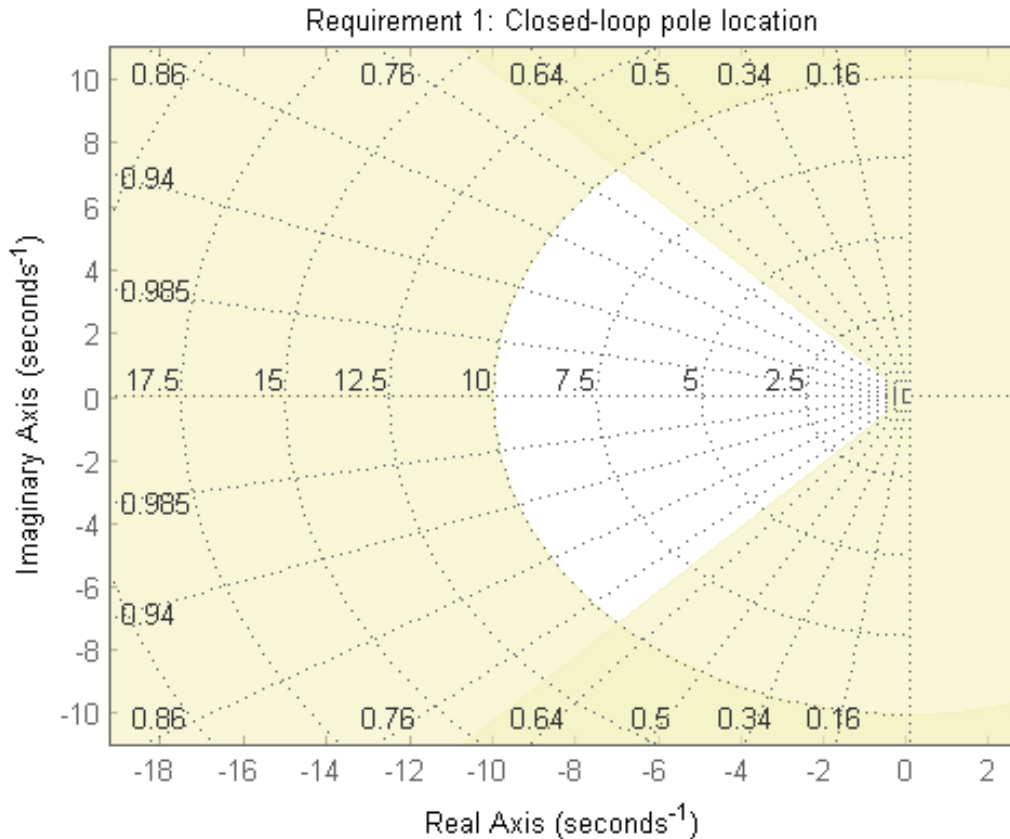
$$|s| < \omega_{\max}.$$

For example

```
R = TuningGoal.Poles();  
R.MinDecay = 0.5;  
R.MinDamping = 0.7;  
R.MaxFrequency = 10;
```

constrains the closed-loop poles to lie in the white region below.

```
viewSpec(R)
```



Increasing the MinDecay value results in faster transients. Increasing the MinDamping value results in better damped transients. Decreasing the MaxFrequency value prevents fast dynamics.

#### Stable Controller Requirement

The `TuningGoal.StableController` requirement enforces stability of a given control element (compensator). The tuning algorithm may produce unstable compensators for unstable plants. You can prevent this by constraining

the location of the compensator poles. For example, if your compensator is parameterized as a second-order transfer function,

```
C = ltiblock.tf('C',1,2);
```

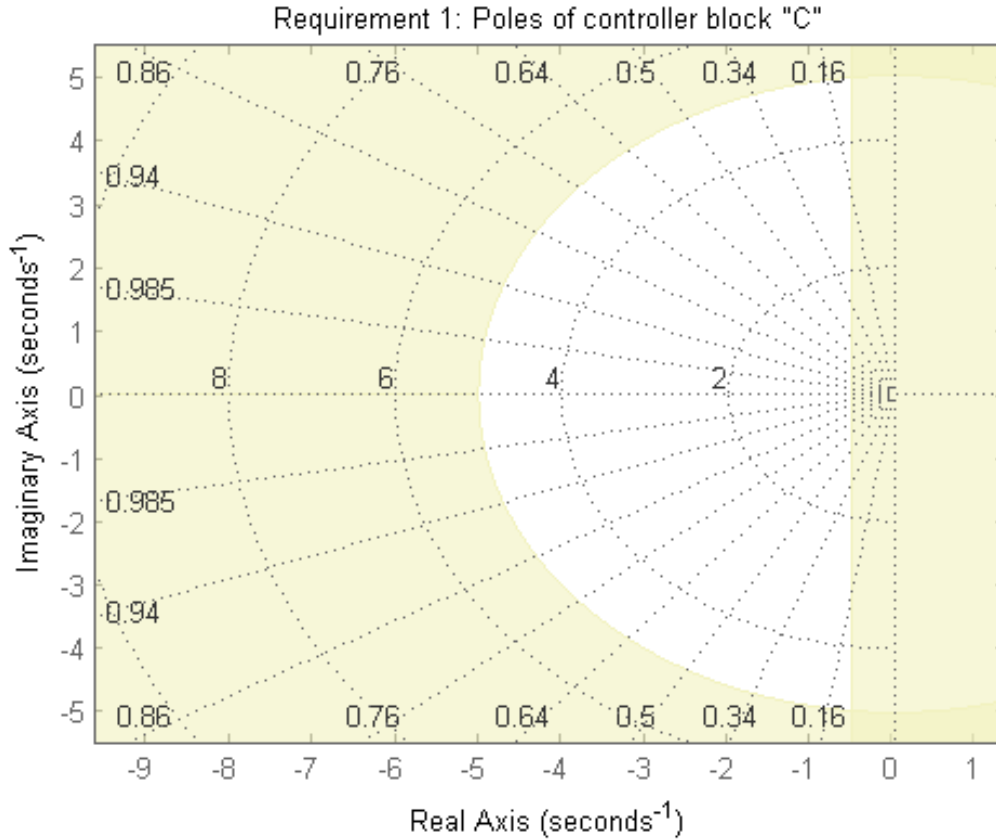
you can force it to have stable dynamics by adding the requirement

```
R = TuningGoal.StableController('C');
```

You can further constrain the minimum decay rate and maximum natural frequency of its poles, for example,

```
R.MinDecay = 0.5;  
R.MaxFrequency = 5;
```

```
viewSpec(R)
```



**Configuring Requirements**

All requirements discussed above are objects that can be further configured by modifying their default attributes. The display shows the list of such attributes. For example

```
R = TuningGoal.Gain('d','y',1)
```

```
R =
```

Gain with properties:

```
Input: {'d'}
Output: {'y'}
MaxGain: [1x1 zpk]
Focus: [0 Inf]
Models: NaN
Openings: {0x1 cell}
Name: ''
```

Three attributes are shared by multiple requirements. The `Focus` property specifies the frequency band in which the requirement is active. For example,

```
R.Focus = [1 20];
```

limits the gain from `d` to `y` in the frequency interval [1,20] only. The `Models` property specifies which models the requirement applies to (in the context of tuning for multiple plant models). For example,

```
R.Models = [2 3 5];
```

indicates that the requirement only applies to the second, third, and fifth model in the model array supplied to `systemtune`. Finally, the `Openings` property lets you specify additional loop openings. For example

```
R = TuningGoal.Margins('Inner',6,45);
R.Openings = 'Outer';
```

specifies stability margins for the inner loop with the outer loop open. All loop opening sites should be registered with `loopswitch` blocks in MATLAB and with the `addControl`, `addMeasurement`, `addSwitch` methods of the `slTunable` tuning interface in Simulink.

# Validating Results

This example shows how to interpret and validate tuning results from `systemtune`.

### Background

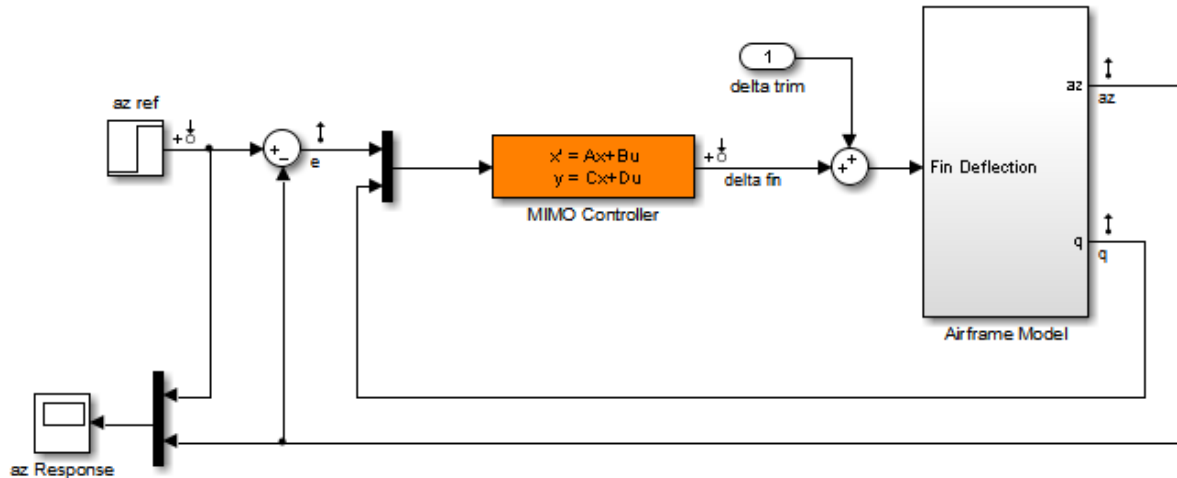
You can tune the parameters of your control system with `systemtune` or `looptune`. The design specifications are captured using `TuningGoal` requirement objects. This example shows how to interpret the results from `systemtune`, graphically verify the design requirements, and perform additional open- and closed-loop analysis.

### Controller Tuning with `SYSTEMTUNE`

We use an autopilot tuning application as illustration, see the *"Tuning of a Two-Loop Autopilot"* example for details. The tuned compensator is the "MIMO Controller" block highlighted in orange in the model below.

```
open_system('rct_airframe2')
```

## Two-loop autopilot for controlling the vertical acceleration of an airframe



See `AutopilotExample` to see how you can tune this MIMO controller with the `SYSTUNE` command.

The setup and tuning steps are repeated below for completeness.

```
STO = slTunable('rct_airframe2','MIMO Controller');
STO.addControl('delta fin');
```

```
% Compensator parameterization
C0 = ltiblock.ss('C',2,1,2);
C0.d.Value(1) = 0; C0.d.Free(1) = false;
STO.setBlockParam('MIMO Controller',C0)
```

```
% Requirements
Req1 = TuningGoal.Tracking('az ref','az',1); % tracking
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0])); % roll-off
Req3 = TuningGoal.Margins('delta fin',7,45); % margins
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain); % disturbance rejection
```

```
% Tuning
Opt = systuneOptions('RandomStart',3);
```

```
rng('default')
[ST1,fSoft,~,Info] = ST0.systune([Req1,Req2,Req3,Req4],Opt);
```

```
Final: Soft = 1.5, Hard = -Inf, Iterations = 60
Final: Soft = 1.15, Hard = -Inf, Iterations = 188
Final: Soft = 1.15, Hard = -Inf, Iterations = 64
Final: Soft = 1.15, Hard = -Inf, Iterations = 97
```

### Interpreting Results

`systune` run three optimizations from three different starting points and returned the best overall result. The first output `ST` is an `sITunable` interface representing the tuned control system. The second output `fSoft` contains the final values of the four requirements for the best design.

```
fSoft
```

```
fSoft =

    1.1477    1.1477    0.5458    1.1477
```

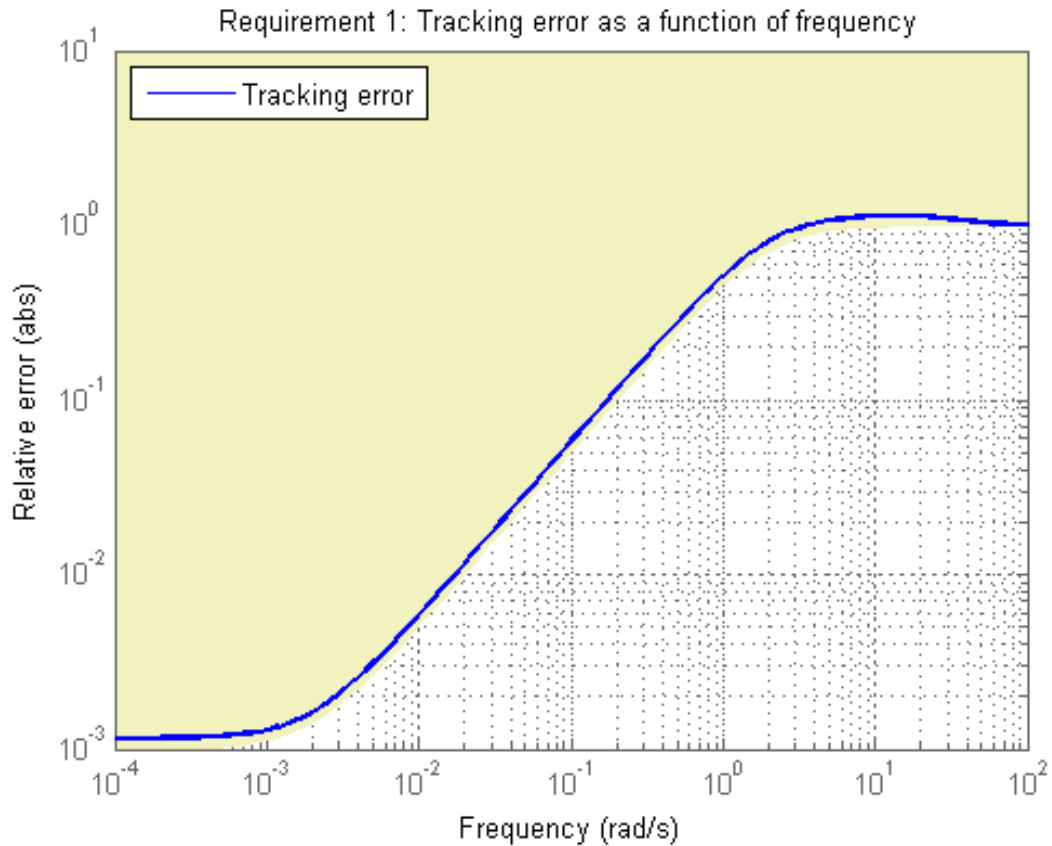
Requirements are normalized so a requirement is satisfied if and only if its value is less than 1. Inspection of `fSoft` reveals that Requirements 1,2,4 are active and slightly violated while Requirement 3 (stability margins) is satisfied.

### Verifying Requirements

Use `viewSpec` to graphically inspect each requirement. This is useful to understand whether small violations are acceptable or what causes large violations. Make sure to provide the structure `Info` returned by `systune` to properly account for scalings and other parameters computed by the optimization algorithms. First verify the tracking requirement.

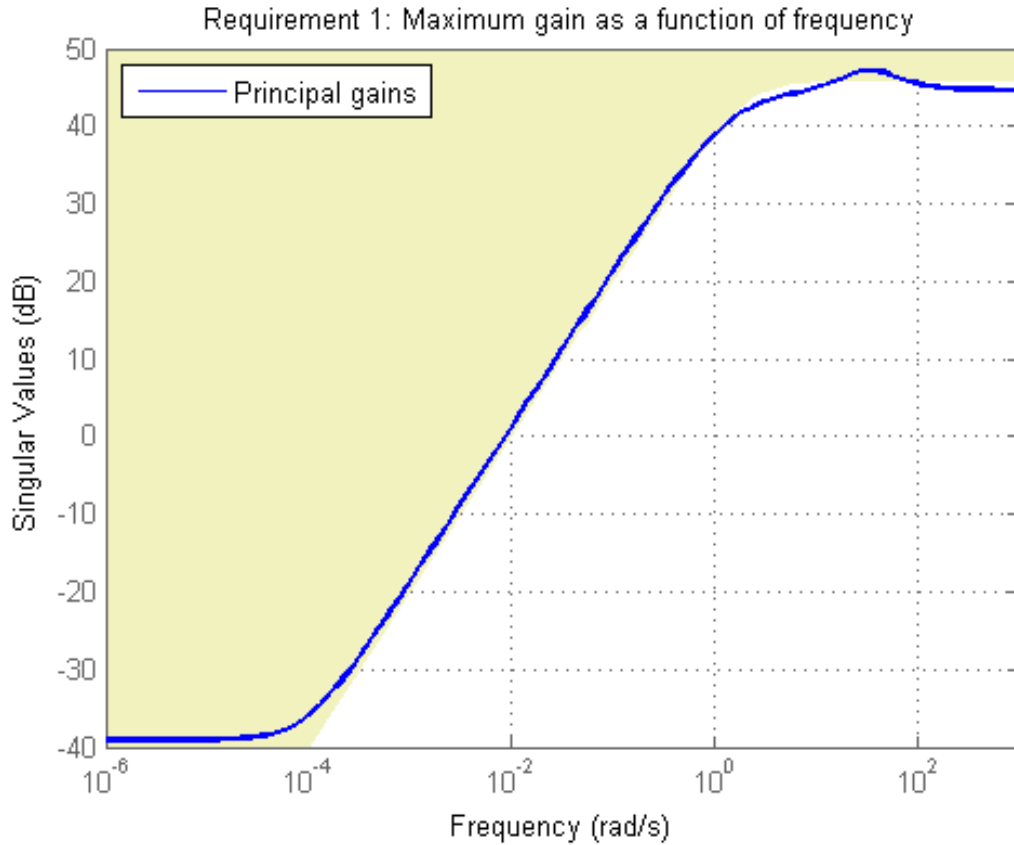
```
viewSpec(Req1,ST1,Info)
```





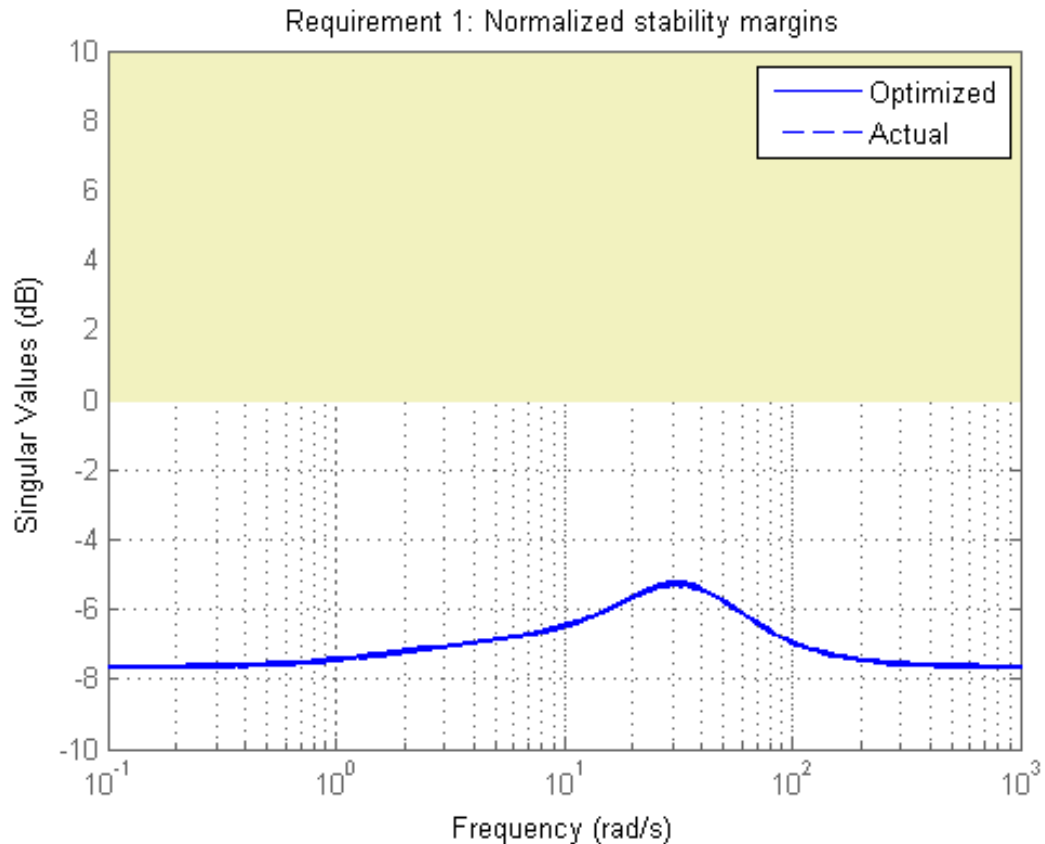
We observe a slight violation across frequency, suggesting that setpoint tracking will perform close to expectations. Similarly, verify the disturbance rejection requirement.

```
viewSpec(Req4,ST1,Info)  
legend('location','NorthWest')
```



Most of the violation is at low frequency with a small bump near 35 rad/s, suggesting possible damped oscillations at this frequency. Finally, verify the stability margin requirement.

`viewSpec(Req3,ST1,Info)`



This requirement is satisfied at all frequencies, with the smallest margins achieved near the crossover frequency as expected. To see the actual margin values at a given frequency, click on the red curve and read the values from the data tip.

### Evaluating Requirements

You can also use `evalSpec` to evaluate each requirement, that is, compute its contribution to the soft and hard constraints. For example

```
[H1,f1] = evalSpec(Req1,ST1,Info);
```

returns the value `f1` of the requirement and the underlying frequency-weighted transfer function `H1` used to compute it. You can verify that `f1` matches the first entry of `fSoft` and coincides with the peak gain of `H1`.

```
[f1 fSoft(1) getPeakGain(H1,1e-6)]
```

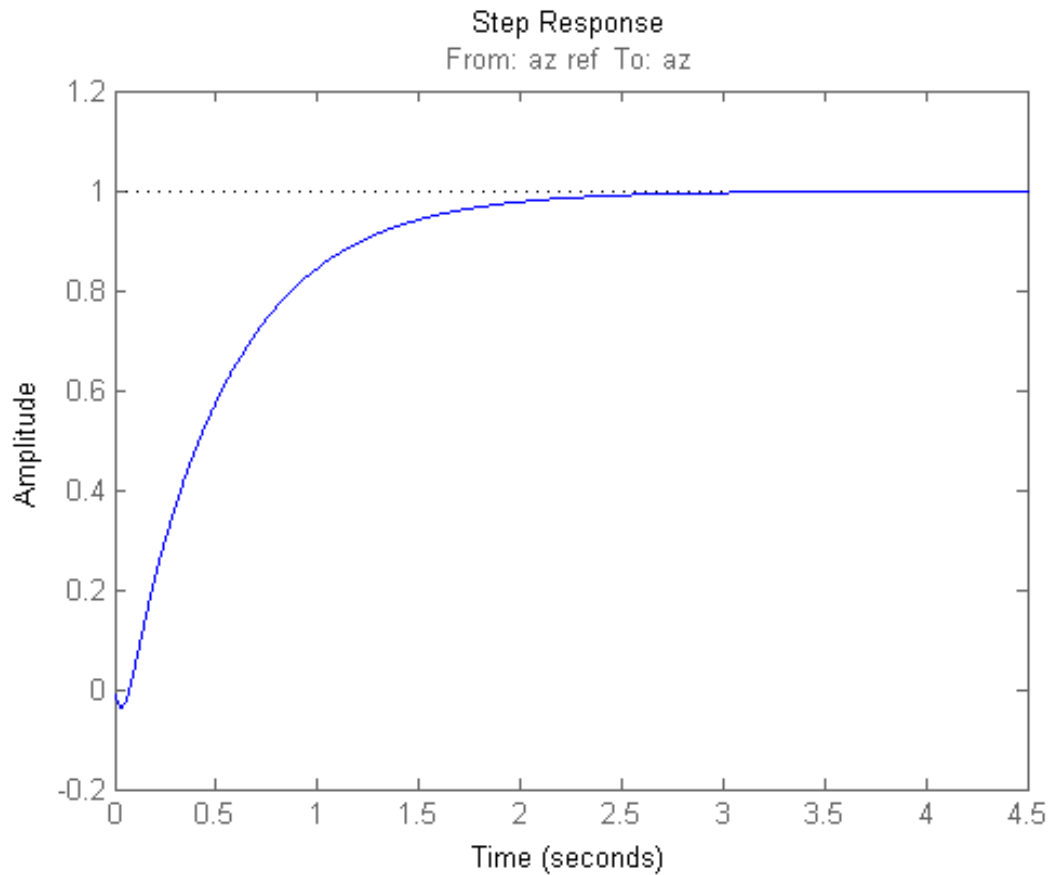
```
ans =
```

```
1.1477    1.1477    1.1477
```

### Analyzing System Responses

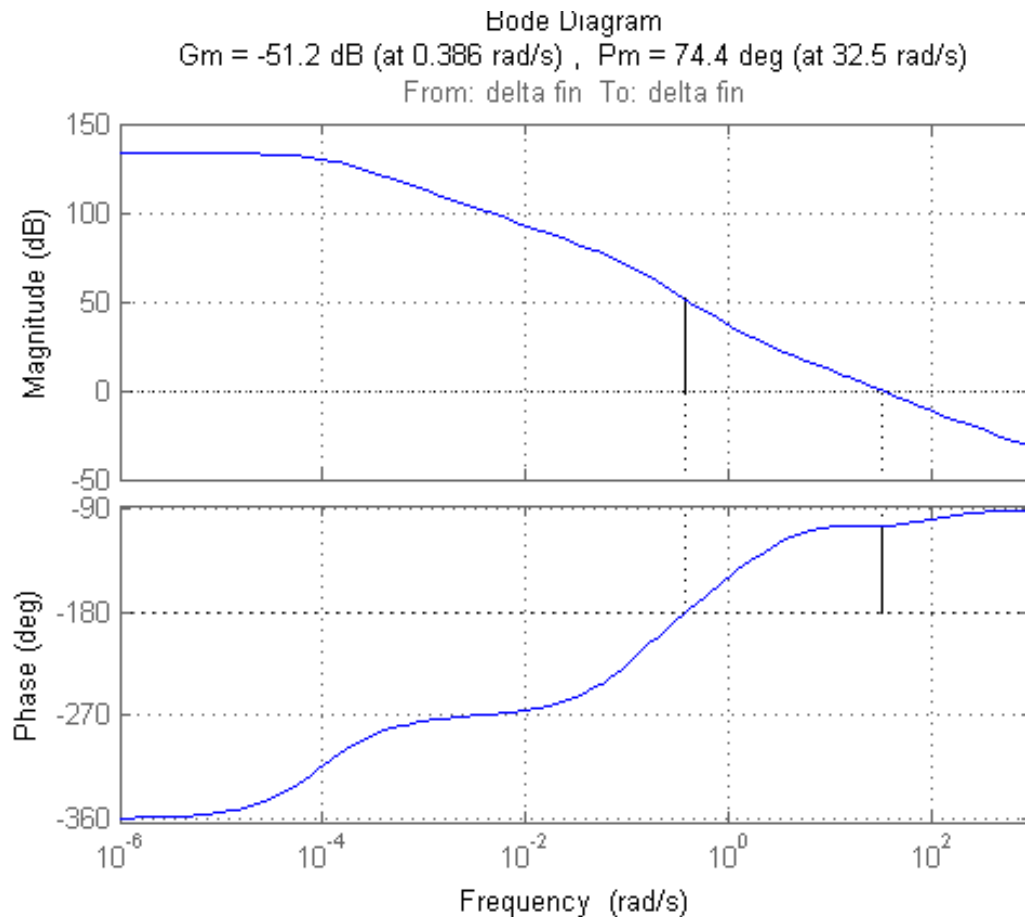
In addition to verifying requirements, you can perform basic open- and closed-loop analysis using `getIOTransfer` and `getLoopTransfer`. For example, verify tracking performance in the time domain by plotting the response `az` to a step command `azref` for the tuned system `ST1`.

```
T = ST1.getIOTransfer('az ref','az');  
step(T)
```



Also plot the open-loop response measured at the plant input `delta fin`. You can use this plot to assess the classical gain and phase margins at the plant input.

```
L = ST1.getLoopTransfer('delta fin',-1); % negative-feedback loop transfer  
margin(L), grid
```



### Soft vs Hard Requirements

So far we have treated all four requirements equally in the objective function. Alternatively, you can use a mix of soft and hard constraints to differentiate between must-have and nice-to-have requirements. For example, you could treat Requirements 3,4 as hard constraints and optimize the first two requirements subject to these constraints. For best results, do this only after obtaining a reasonable design with all requirements treated equally.

```
[ST2, fSoft, gHard, Info] = ST1.systune([Req1 Req2], [Req3 Req4]);
```

---

```
Final: Soft = 1.31, Hard = 0.99998, Iterations = 137
```

```
fSoft
```

```
fSoft =
```

```
    1.2259    1.3068
```

```
gHard
```

```
gHard =
```

```
    0.4762    1.0000
```

Here `fSoft` contains the final values of the first two requirements (soft constraints) and `gHard` contains the the final values of the last two requirements (hard constraints). The hard constraints are satisfied since all entries of `gHard` are less than 1. As expected, the best value of the first two requirements went up as the optimizer strived to strictly enforce the fourth requirement.

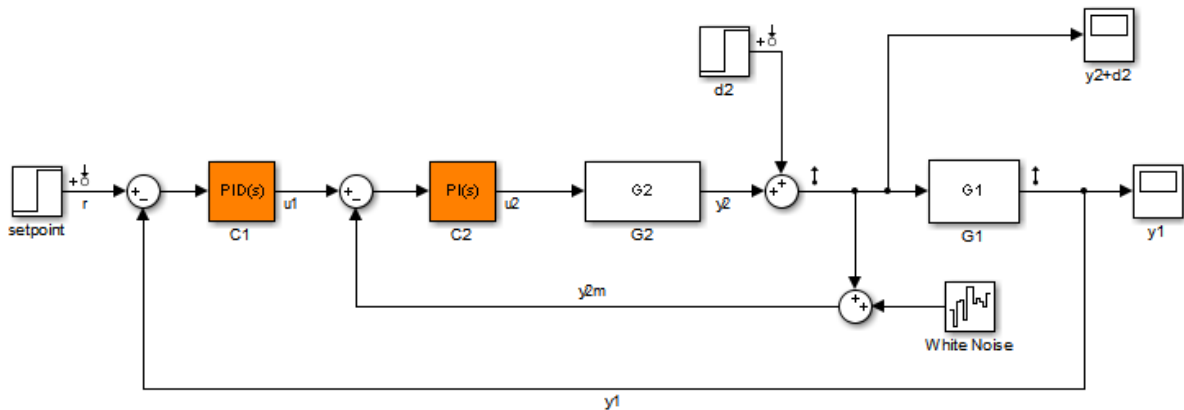
## Tuning Multi-Loop Control Systems

This example shows how to jointly tune the inner and outer loops of a cascade architecture with the `system` command.

### Cascaded PID Loops

Cascade control is often used to achieve smooth tracking with fast disturbance rejection. The simplest cascade architecture involves two control loops (inner and outer) as shown in the block diagram below. The inner loop is typically faster than the outer loop to reject disturbances before they propagate to the outer loop.

```
open_system('rct_cascade')
```



### Plant Models and Bandwidth Requirements

In this example, the inner loop plant  $G_2$  is

$$G_2(s) = \frac{3}{s + 2}$$

and the outer loop plant  $G_1$  is



$$G_1(s) = \frac{10}{(s+1)^3}$$

```
G2 = zpk([], -2, 3);
G1 = zpk([], [-1 -1 -1], 10);
```

We use a PI controller in the inner loop and a PID controller in the outer loop. The outer loop must have a bandwidth of at least 0.2 rad/s and the inner loop bandwidth should be ten times larger for adequate disturbance rejection.

### Tuning the PID Controllers with SYSTUNE

When the control system is modeled in Simulink, use the `sITunable` interface in Simulink Control Design™ to set up the tuning task. List the tunable blocks and mark the signals `y1` and `y2` as loop opening locations where to measure open-loop transfers and specify loop shapes.

```
STO = sITunable('rct_cascade', {'C1', 'C2'});
STO.addSwitch({'y1', 'y2'})
```

You can query the current values of `C1` and `C2` in the Simulink model using `showTunable`. The control system is unstable for these initial values as confirmed by simulating the Simulink model.

```
STO.showTunable()
```

```
Block "rct_cascade/C1" =
```

$$K_p + K_i * \frac{1}{s}$$

```
with Kp = 0.1, Ki = 0.1
```

```
Name: C1
```

```
Continuous-time PI controller in parallel form.
```

```
-----
```

```
Block "rct_cascade/C2" =
```

$$K_p + K_i * \frac{1}{s}$$

```
with Kp = 0.1, Ki = 0.1
```

```
Name: C2
```

```
Continuous-time PI controller in parallel form.
```

Next use "LoopShape" requirements to specify the desired bandwidths for the the inner and outer loops. Use  $0.2/s$  as target loop shape for the outer loop to enforce integral action with a gain crossover frequency at 0.2 rad/s:

```
% Outer loop bandwidth = 0.2
s = tf('s');
Req1 = TuningGoal.LoopShape('y1',0.2/s); % loop transfer measured at y1
Req1.Name = 'Outer Loop';
```

Use  $2/s$  for the inner loop to make it ten times faster (higher bandwidth) than the outer loop. To constrain the inner loop transfer, make sure to open the outer loop by specifying y1 as a loop opening:

```
% Inner loop bandwidth = 2
Req2 = TuningGoal.LoopShape('y2',2/s); % loop transfer measured at y2
Req2.Openings = 'y1'; % with outer loop opened at y1
Req2.Name = 'Inner Loop';
```

You can now tune the PID gains in C1 and C2 with systune:

```
[ST,fSoft,~,Info] = ST0.systune([Req1,Req2]);
```

```
Final: Soft = 0.945, Hard = -Inf, Iterations = 60
      Min decay rate 1.01e-07 is close to lower bound 1e-07
```

Use showTunable to see the tuned PID gains.

```
ST.showTunable()
```

Block "rct\_cascade/C1" =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

with  $K_p = 0.0333$ ,  $K_i = 0.0168$ ,  $K_d = -0.0368$ ,  $T_f = 9.88e+06$

Name: C1

Continuous-time PIDF controller in parallel form.

-----

Block "rct\_cascade/C2" =

$$K_p + K_i * \frac{1}{s}$$

with  $K_p = 0.76$ ,  $K_i = 2$

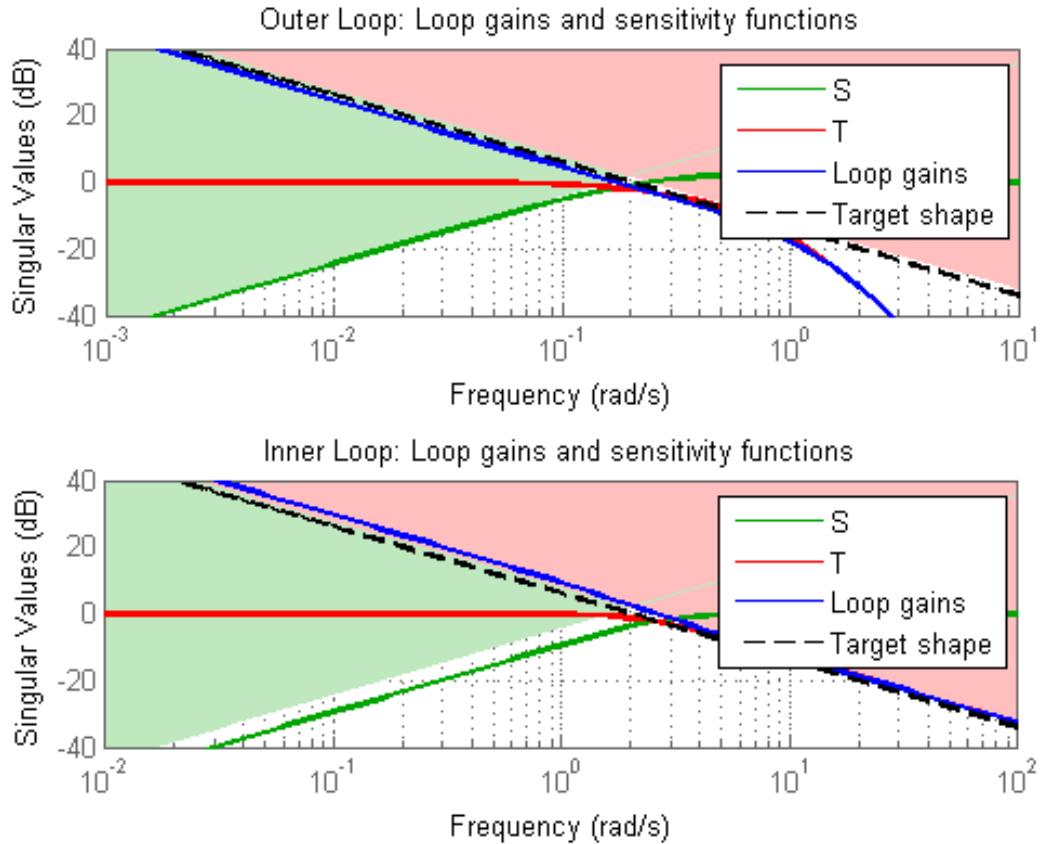
Name: C2

Continuous-time PI controller in parallel form.

### Validating the Design

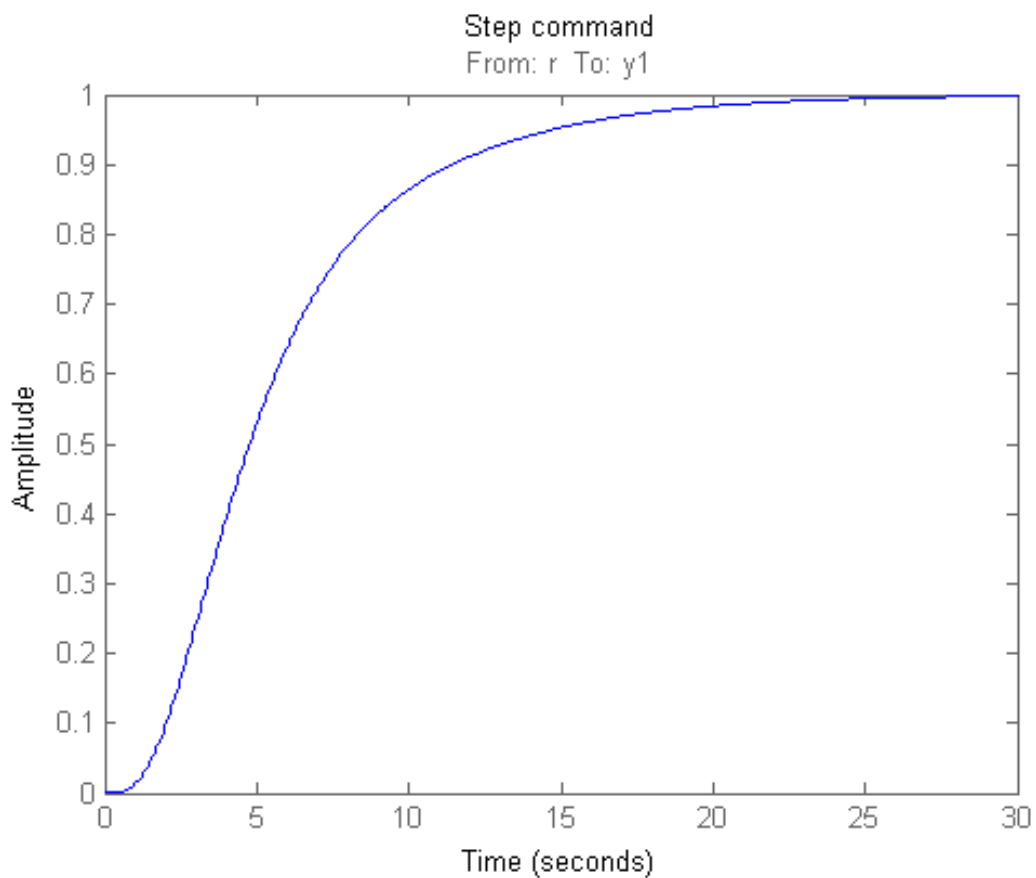
The final value is less than 1 which means that `sysTune` successfully met both loop shape requirements. Confirm this by inspecting the tuned control system `ST` with `viewSpec`

```
viewSpec([Req1,Req2],ST,Info)
```

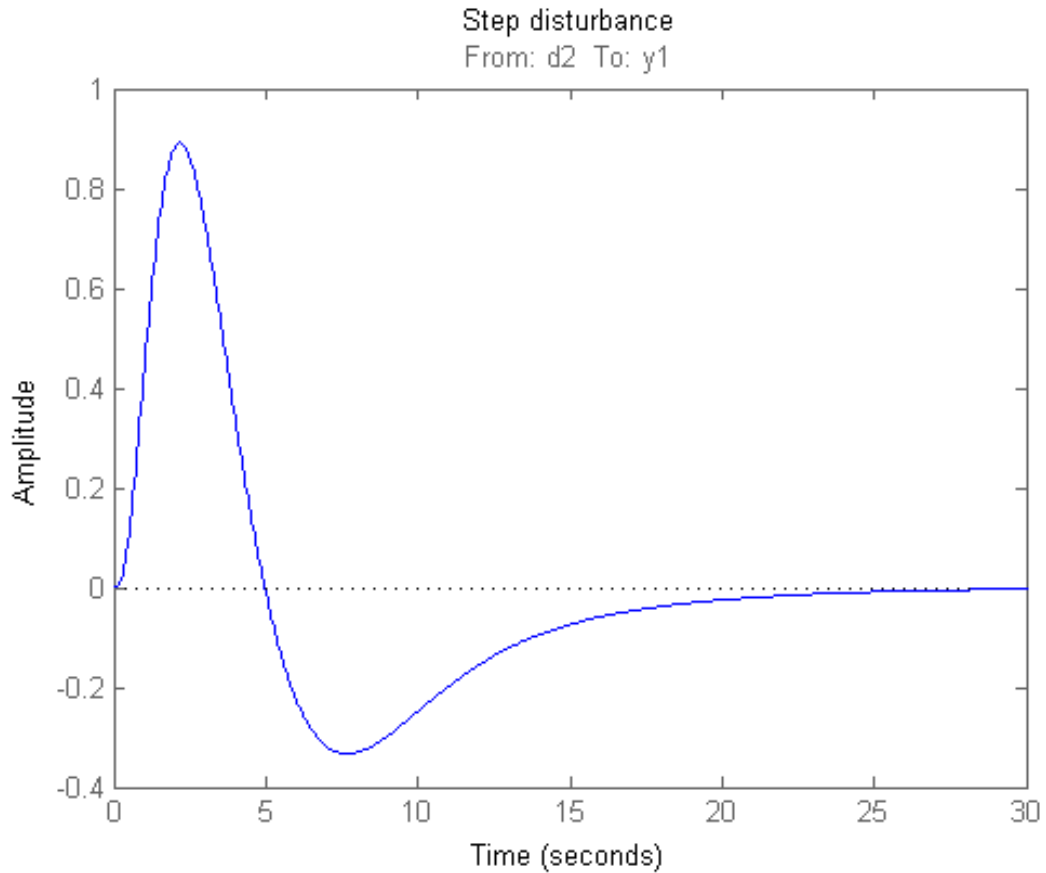


Note that the inner and outer loops have the desired gain crossover frequencies. To further validate the design, plot the tuned responses to a step command  $r$  and step disturbance  $d_2$ :

```
% Response to a step command
H = ST.getIOTransfer('r','y1');
clf, step(H,30), title('Step command')
```



```
% Response to a step disturbance  
H = ST.getIOTransfer('d2','y1');  
step(H,30), title('Step disturbance')
```

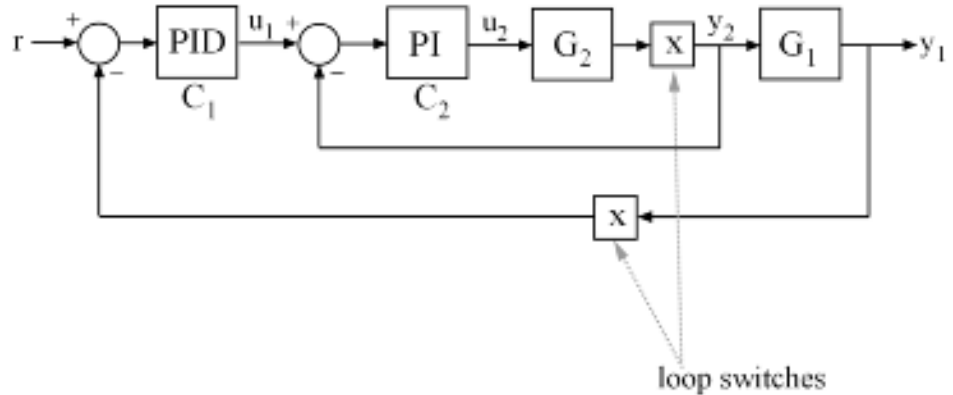


Once you are satisfied with the linear analysis results, use `writeBlockValue` to write the tuned PID gains back to the Simulink blocks. You can then conduct a more thorough validation in Simulink.

`ST.writeBlockValue`

**Equivalent Workflow in MATLAB**

If you do not have a Simulink model of the control system, you can perform the same steps using LTI models of the plant and Control Design blocks to model the tunable elements.



**Figure 1: Cascade Architecture**

First create parametric models of the tunable PI and PID controllers.

```
C1 = ltiblock.pid('C1','pid');
C2 = ltiblock.pid('C2','pi');
```

Then use loopswitch blocks to mark the loop opening locations  $y_1$  and  $y_2$ .

```
LS1 = loopswitch('y1');
LS2 = loopswitch('y2');
```

Finally, create a closed-loop model  $T_0$  of the overall control system by closing each feedback loop. The result is a generalized state-space model depending on the tunable elements  $C_1$  and  $C_2$ .

```
InnerCL = feedback(LS2*G2*C2,1);
T0 = feedback(G1*InnerCL*C1,LS1);
T0.InputName = 'r';
T0.OutputName = 'y1';
```

You can now tune the PID gains in C1 and C2 with `sysstune`.

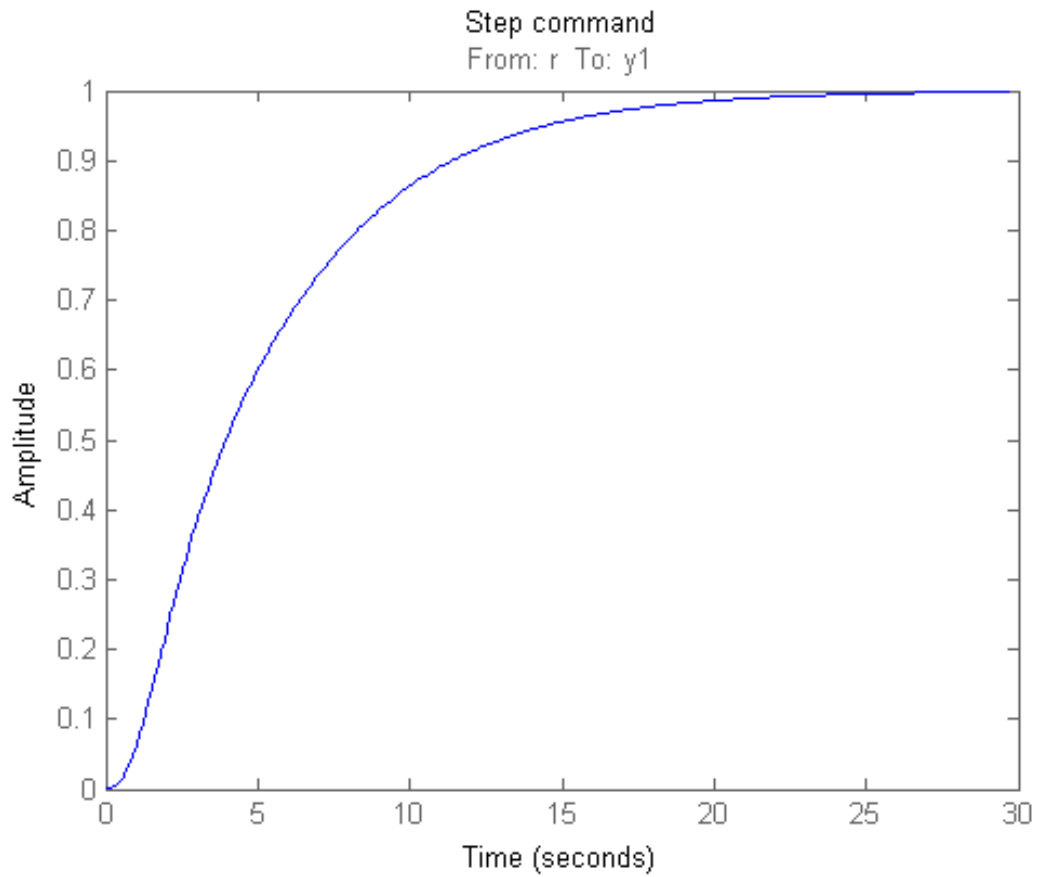
```
[T,fSoft,~,Info] = sysstune(T0,[Req1,Req2]);
```

```
Final: Soft = 0.858, Hard = -Inf, Iterations = 97
```

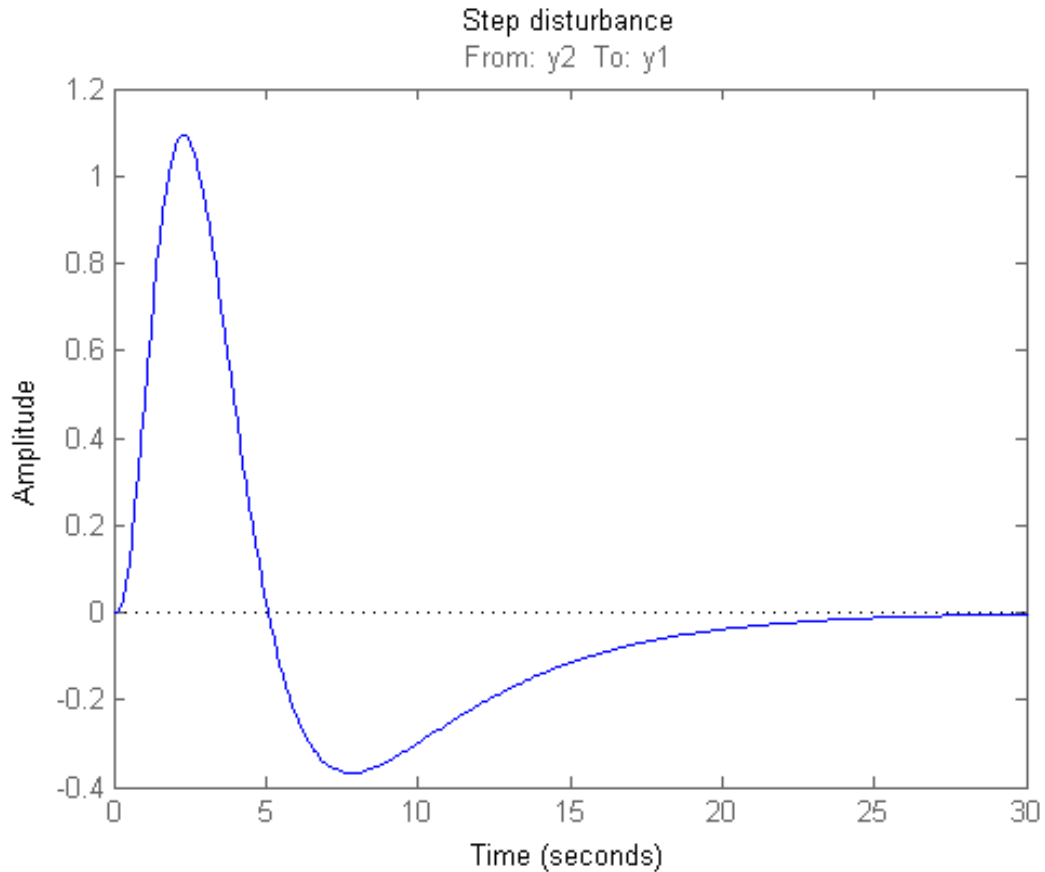
As before, use `getIOTransfer` to compute and plot the tuned responses to a step command `r` and step disturbance entering at the location `y2`:

```
% Response to a step command  
H = getIOTransfer(T,'r','y1');  
clf, step(H,30), title('Step command')
```



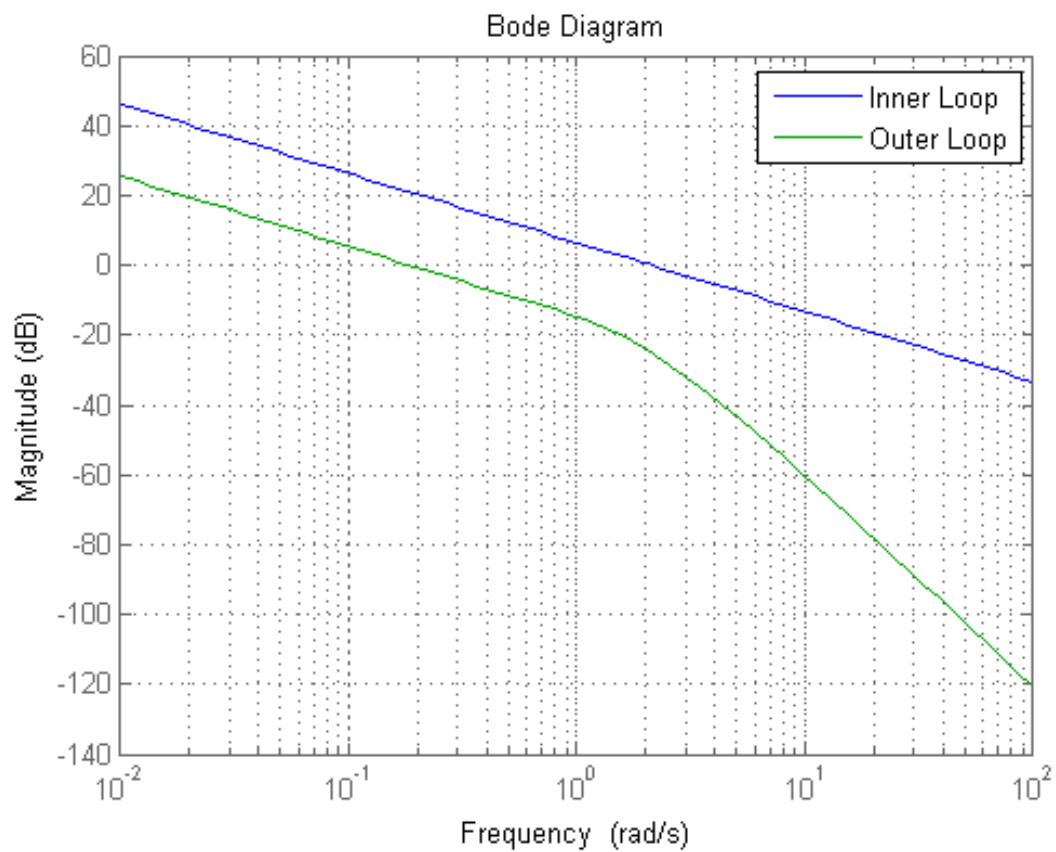


```
% Response to a step disturbance  
H = getIOTransfer(T,'y2','y1');  
step(H,30), title('Step disturbance')
```



You can also plot the open-loop gains for the inner and outer loops to validate the bandwidth requirements. Note the -1 sign to compute the negative-feedback open-loop transfer:

```
L1 = getLoopTransfer(T,'y1',-1);      % crossover should be at .2
L2 = getLoopTransfer(T,'y2',-1,'y1'); % crossover should be at 2
bodemag(L2,L1,{1e-2,1e2}), grid
legend('Inner Loop','Outer Loop')
```



## Using Parallel Computing to Accelerate Tuning

This example shows how to leverage the Parallel Computing Toolbox™ to accelerate multi-start strategies for tuning fixed-structure control systems.

### Background

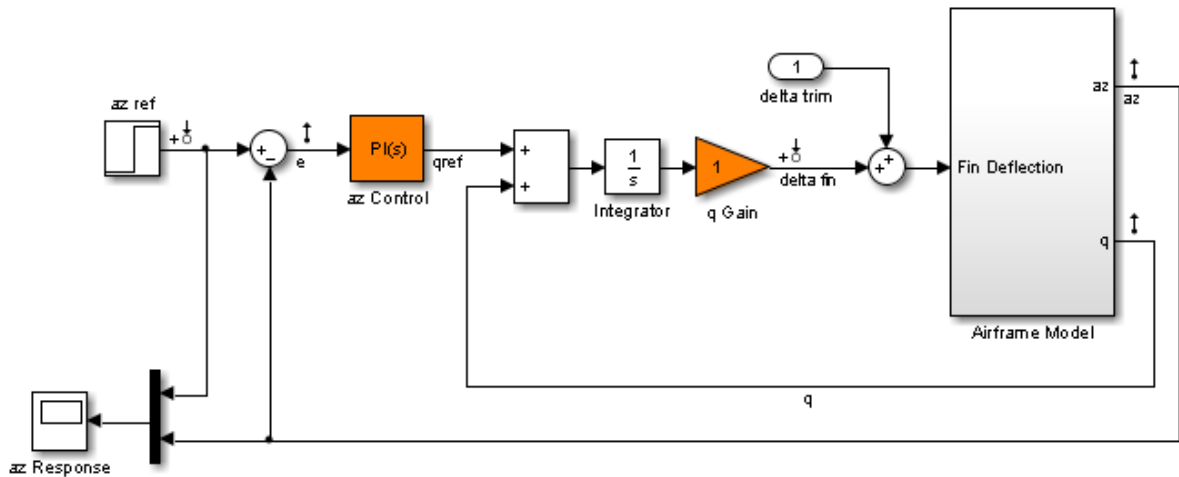
Both `systune` and `looptune` use local optimization methods for tuning the control architecture at hand. To mitigate the risk of ending up with a locally optimal but globally poor design, it is recommended to run several optimizations starting from different randomly generated initial points. If you have a multi-core machine or have access to distributed computing resources, you can significantly speed up this process using the Parallel Computing Toolbox.

This example shows how to parallelize the tuning of an airframe autopilot with `looptune`. See the example "Tuning of a Two-Loop Autopilot" for more details about this application of `looptune`.

### Autopilot Tuning

The airframe dynamics and autopilot are modeled in Simulink.

```
open_system('rct_airframe1')
```

**Two-loop autopilot for controlling the vertical acceleration of an airframe**

See `AutopilotExample` to see how you can tune the autopilot gains with the `LOOPTUNE` command.

The autopilot consists of two cascaded loops whose tunable elements include two PI controller gains ("az Control" block) and one gain in the pitch-rate loop ("q Gain" block). The vertical acceleration  $az$  should track the command  $az_{ref}$  with a 1 second response time. Use `sITunable` to configure this tuning task (see "Tuning of a Two-Loop Autopilot" example for details):

```
STO = sITunable('rct_airframe1',{ 'az Control', 'q Gain'});
STO.addControl('delta fin');
STO.addMeasurement({'az', 'q'});

% Design requirements
wc = [3,12]; % bandwidth
TrackReq = TuningGoal.Tracking('az ref','az',1); % tracking
```

**Parallel Tuning with LOOPTUNE**

We are ready to tune the autopilot gains with `looptune`. To minimize the risk of getting a poor-quality local minimum, run 30 optimizations starting from

30 randomly generated values of the three gains. Configure the `looptune` options to enable parallel processing of these 30 runs:

```
rng('default')
Options = looptuneOptions('RandomStart',30,'UseParallel',true);
```

Next call `looptune` to launch the tuning algorithm. The 30 runs are automatically distributed across available computing resources:

```
[ST,gam,Info] = ST0.looptune(wc,TrackReq,Options);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to
Final: Failed to enforce closed-loop stability (max Re(s) = 0.042)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.051)
Final: Peak gain = 1.23, Iterations = 46
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.04)
Final: Peak gain = 61.9, Iterations = 72
Final: Peak gain = 1.23, Iterations = 42
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.039)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 90
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.042)
Final: Peak gain = 1.23, Iterations = 46
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.039)
```

```
Final: Peak gain = 1.23, Iterations = 103
Final: Peak gain = 1.23, Iterations = 57
Final: Peak gain = 1.23, Iterations = 120
```

Most runs return 1.23 as optimal gain value, suggesting that this local minimum has a wide region of attraction and is likely to be the global optimum. Use `showBlockValue` to see the corresponding gain values:

```
showBlockValue(ST)
```

```
Block "rct_airframe1/az Control" =
```

$$K_p + K_i * \frac{1}{s}$$

```
with Kp = 0.00165, Ki = 0.00166
```

```
Name: az_Control
```

```
Continuous-time PI controller in parallel form.
```

```
-----
Block "rct_airframe1/q Gain" =
```

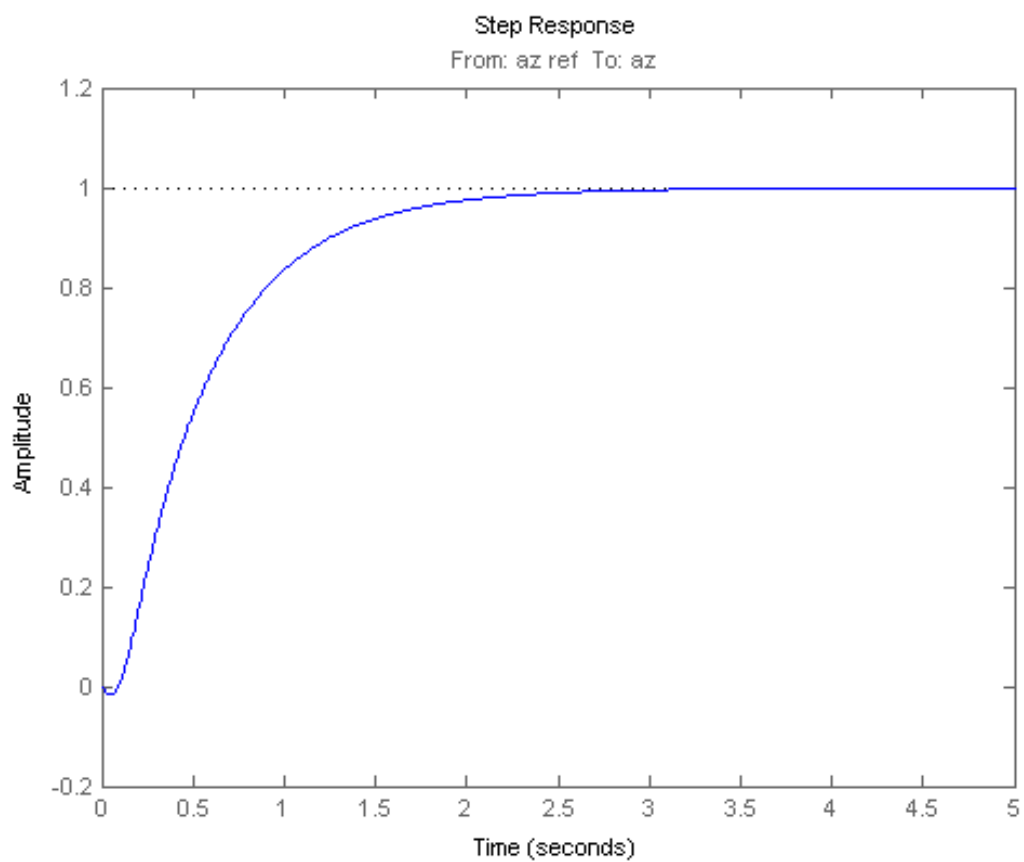
$$d = \frac{u1}{y1 \ 1.985}$$

```
Name: q_Gain
```

```
Static gain.
```

Plot the closed-loop response for this set of gains:

```
T = ST.getIOTransfer('az ref','az');
step(T,5)
```



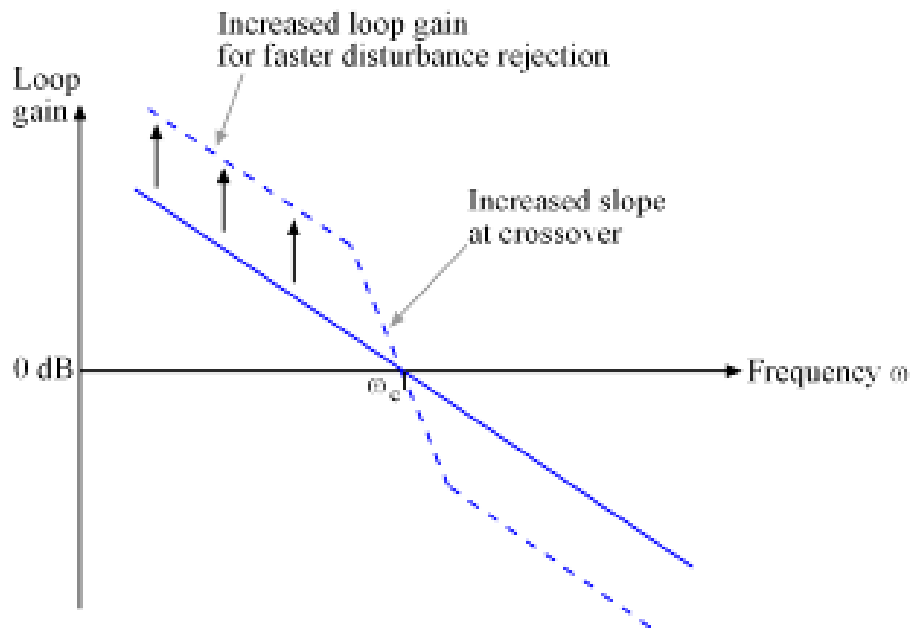


## PID Tuning for Setpoint Tracking vs. Disturbance Rejection

This example uses `systemtune` to explore trade-offs between setpoint tracking and disturbance rejection when tuning PID controllers.

### PID Tuning Trade-Offs

When tuning 1-DOF PID controllers, it is often impossible to achieve good tracking and fast disturbance rejection at the same time. Assuming the control bandwidth is fixed, faster disturbance rejection requires more gain inside the bandwidth, which can only be achieved by increasing the slope at the crossover frequency. Because a larger slope means a smaller phase margin, this typically comes at the expense of more overshoot in the response to setpoint changes.

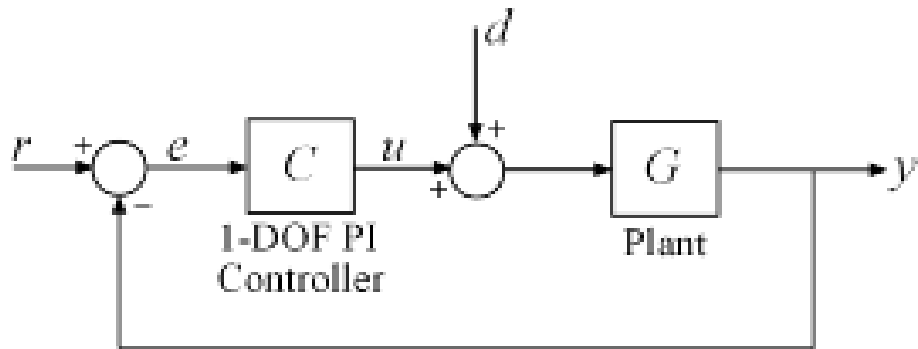


**Figure 1: Trade-off in 1-DOF PID Tuning.**

This example uses `systemtune` to explore this trade-off and find the right compromise for your application. See also `pidtool` for a more direct and interactive way to make such trade-off (**Transient behavior** slider).

**Tuning Setup**

Consider the PI loop of Figure 2 with a load disturbance at the plant input.

**Figure 2: PI Control Loop.**

For this example we use the plant model

$$G(s) = \frac{10(s+5)}{(s+1)(s+2)(s+10)}.$$

The PI controller  $C$  has two parameters to tune: the proportional and integral gains. The target control bandwidth is 1 rad/s.

Construct a tunable model `T0` of the closed-loop transfer from  $r$  to  $y$ . Use a `loopswitch` block to mark the location  $u$  where the disturbance enters.

```
G = zpk(-5, [-1 -2 -10], 10);
C = ltiblock.pid('C', 'pi');
```

```
LS = loopswitch('u');

T0 = feedback(G*LS*C,1);
T0.u = 'r'; T0.y = 'y';
```

The gain of the open-loop response  $L = GC$  is a key indicator of the feedback loop behavior. The open-loop gain should be high (greater than one) inside the control bandwidth to ensure good disturbance rejection, and should be low (less than one) outside the control bandwidth to be insensitive to measurement noise and unmodeled plant dynamics. Accordingly, use three requirements to express the control objectives:

- "Tracking" requirement to specify a response time of about 2 seconds to step changes in  $r$ .
- "MinLoopGain" requirement to keep the loop gain high before 0.5 rad/s
- "MaxLoopGain" requirement to limit the control bandwidth and force a roll-off of -20 dB/decade past 4 rad/s

```
s = tf('s');
wc = 1; % target crossover frequency

% Tracking
R1 = TuningGoal.Tracking('r','y',2);

% Disturbance rejection
R2 = TuningGoal.MinLoopGain('u',wc/s);
R2.Focus = [0 0.5];

% Bandwidth and roll-off
R3 = TuningGoal.MaxLoopGain('u',4/s);
```

### Tuning of 1-DOF PI Controller

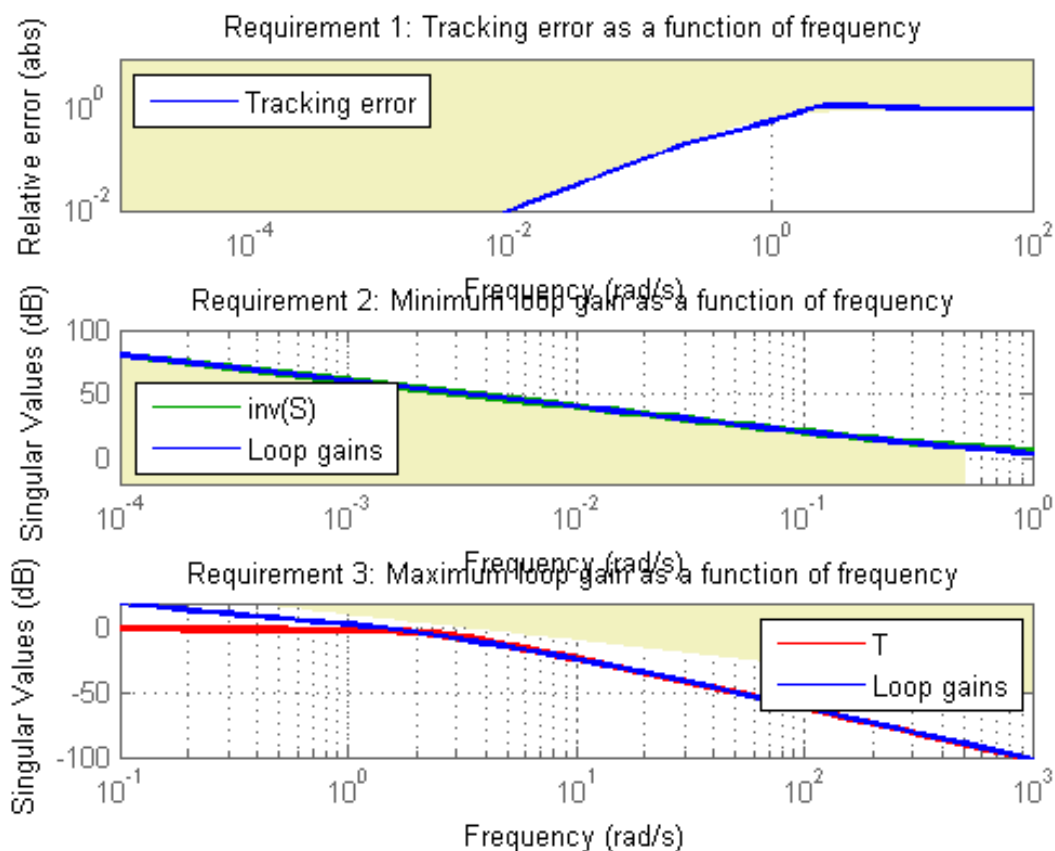
Using `systemtune`, you can now tune the PI gains to meet these requirements. Treat the bandwidth and disturbance rejection goals as hard constraints and optimize tracking subject to these constraints.

```
T1 = systemtune(T0,R1,[R2 R3]);
```

Final: Soft = 1.24, Hard = 0.99998, Iterations = 126

Verify that all three requirements are nearly met. The blue curves are the achieved values and the yellow patches highlight regions where the requirements are violated.

viewSpec([R1 R2 R3],T1)



**Tracking vs. Rejection Trade-Off**

To gain insight into this trade-off, increase the required loop gain by a factor  $\alpha$  inside the control bandwidth (frequency band  $[0,0.5]$  rad/s). Re-tune the PI gains for the values  $\alpha = 2, 5$ .

```
% Increase loop gain by factor 2
alpha = 2;
R2.MinGain = alpha*wc/s;
T2 = systune(T0,R1,[R2 R3]);
```

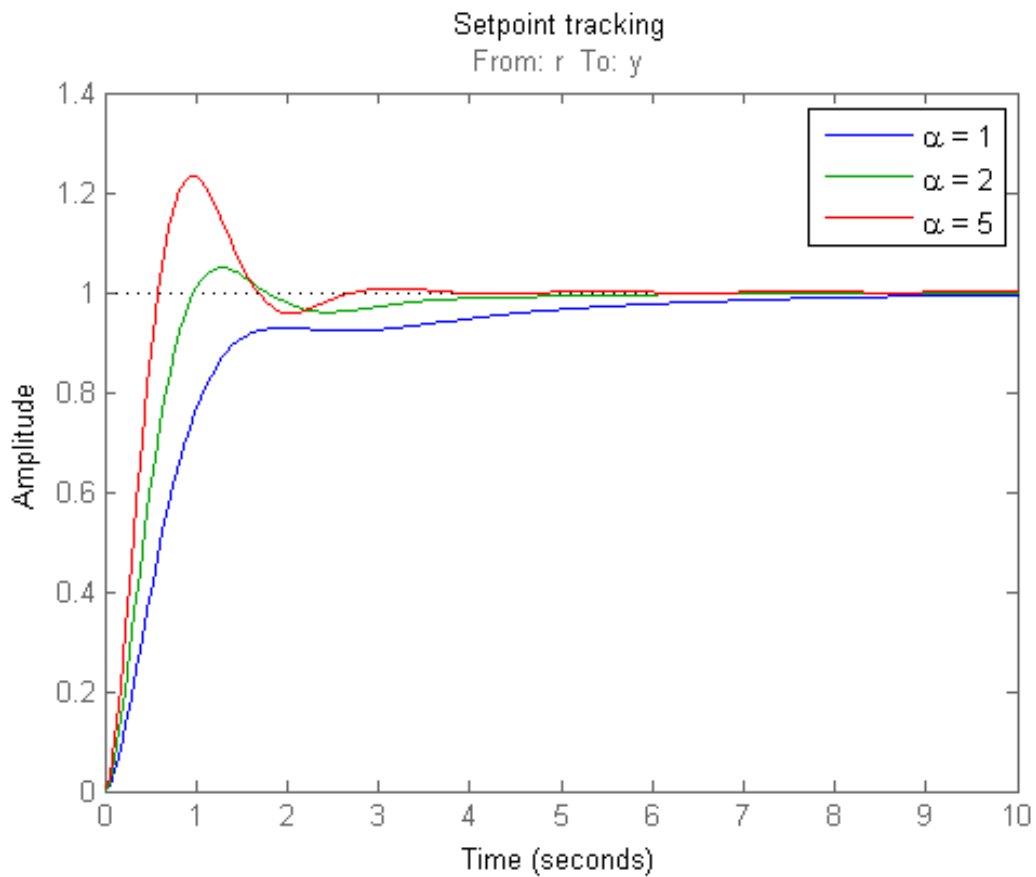
```
Final: Soft = 1.32, Hard = 1, Iterations = 105
```

```
% Increase loop gain by factor 5
alpha = 5;
R2.MinGain = alpha*wc/s;
T3 = systune(T0,R1,[R2 R3]);
```

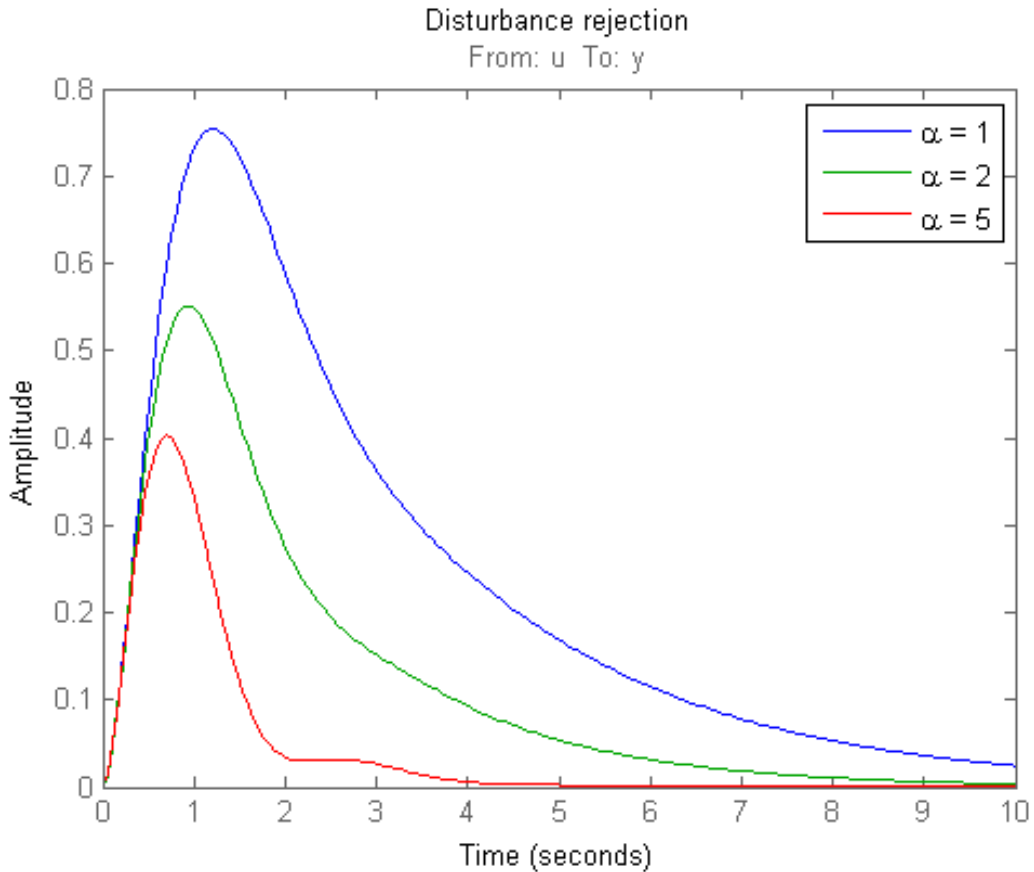
```
Final: Soft = 1.52, Hard = 0.99991, Iterations = 140
```

Now compare the responses to a step command  $r$  and to a step disturbance  $d$  entering at the plant input  $u$ .

```
clf, step(T1,T2,T3,10)
title('Setpoint tracking')
legend('\alpha = 1', '\alpha = 2', '\alpha = 5')
```



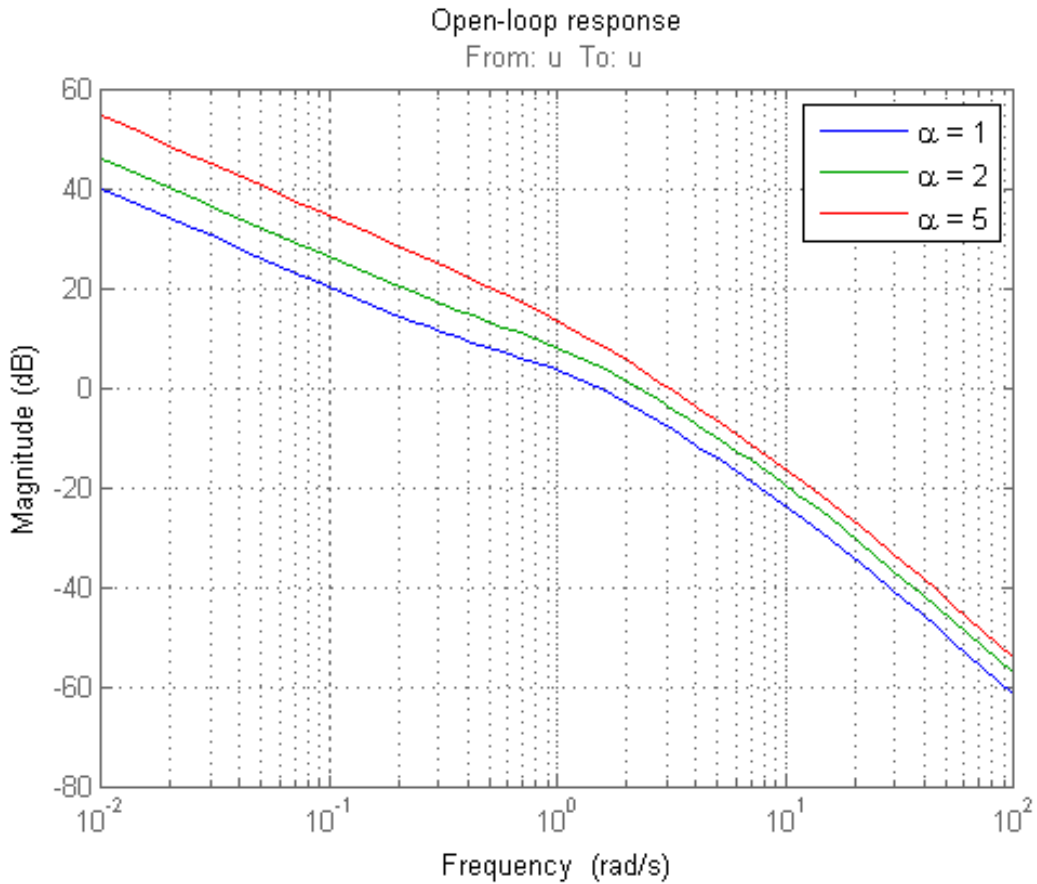
```
% Compute closed-loop transfer from u to y
D1 = getIOTransfer(T1,'u','y');
D2 = getIOTransfer(T2,'u','y');
D3 = getIOTransfer(T3,'u','y');
step(D1,D2,D3,10)
title('Disturbance rejection')
legend('\alpha = 1','\alpha = 2','\alpha = 5')
```



Note how disturbance rejection improves as alpha increases, but only at the expense of increased overshoot and oscillations in setpoint tracking. Plot the open-loop responses for the three designs, and note how the slope at crossover (0dB) increases with alpha.

```
L1 = getLoopTransfer(T1, 'u');
L2 = getLoopTransfer(T2, 'u');
L3 = getLoopTransfer(T3, 'u');
bodemag(L1,L2,L3,{1e-2,1e2}), grid
title('Open-loop response')
```

legend('\alpha = 1', '\alpha = 2', '\alpha = 5')



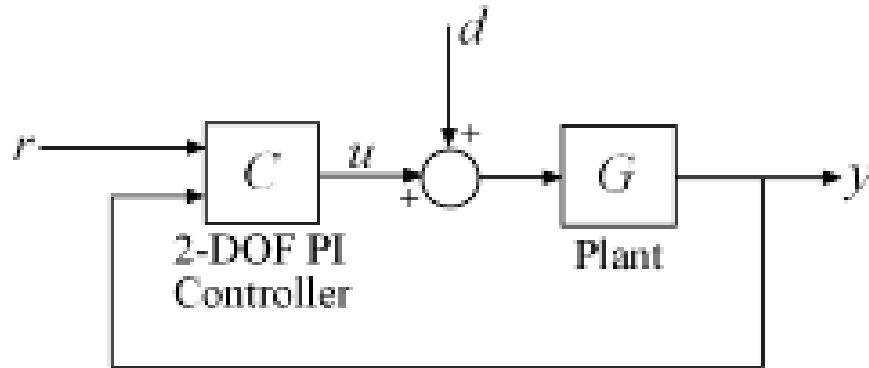
Which design is most suitable depends on the primary purpose of the feedback loop you are tuning.

#### Tuning of 2-DOF PI Controller

If you cannot compromise tracking to improve disturbance rejection, consider using a 2-DOF architecture instead. A 2-DOF PI controller is capable of fast



disturbance rejection without significant increase of overshoot in setpoint tracking.



**Figure 3: 2-DOF PI Control Loop.**

Use the `ltiblock.pid2` object to parameterize the 2-DOF PI controller and construct a tunable model `T0` of the closed-loop system in Figure 3.

```
C = ltiblock.pid2('C', 'pi');
T0 = feedback(G*LS*C,1,2,1,+1);
T0 = T0(:,1);
T0.u = 'r'; T0.y = 'y';
```

Next tune the 2-DOF PI controller for the largest loop gain tried earlier ( $\alpha = 5$ ).

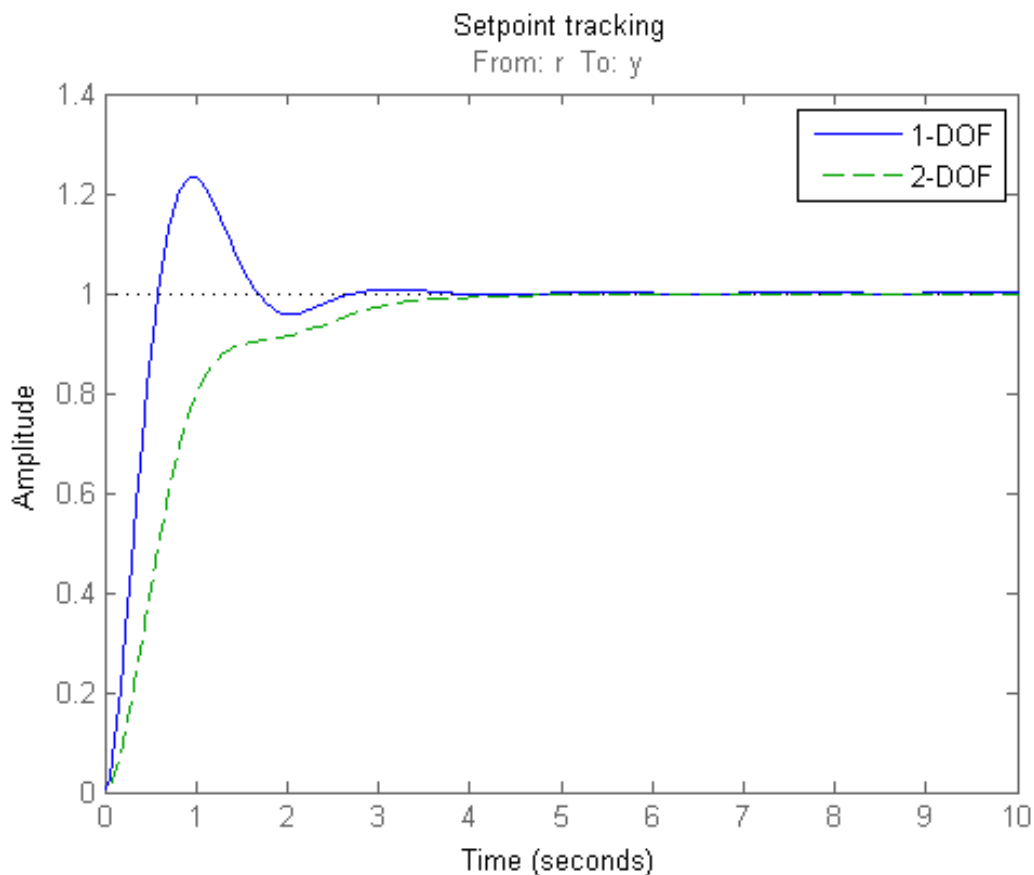
```
% Minimum loop gain inside bandwidth (for disturbance rejection)
alpha = 5;
R2.MinGain = alpha*wc/s;

% Tune 2-DOF PI controller
T4 = systune(T0,R1,[R2 R3]);
```

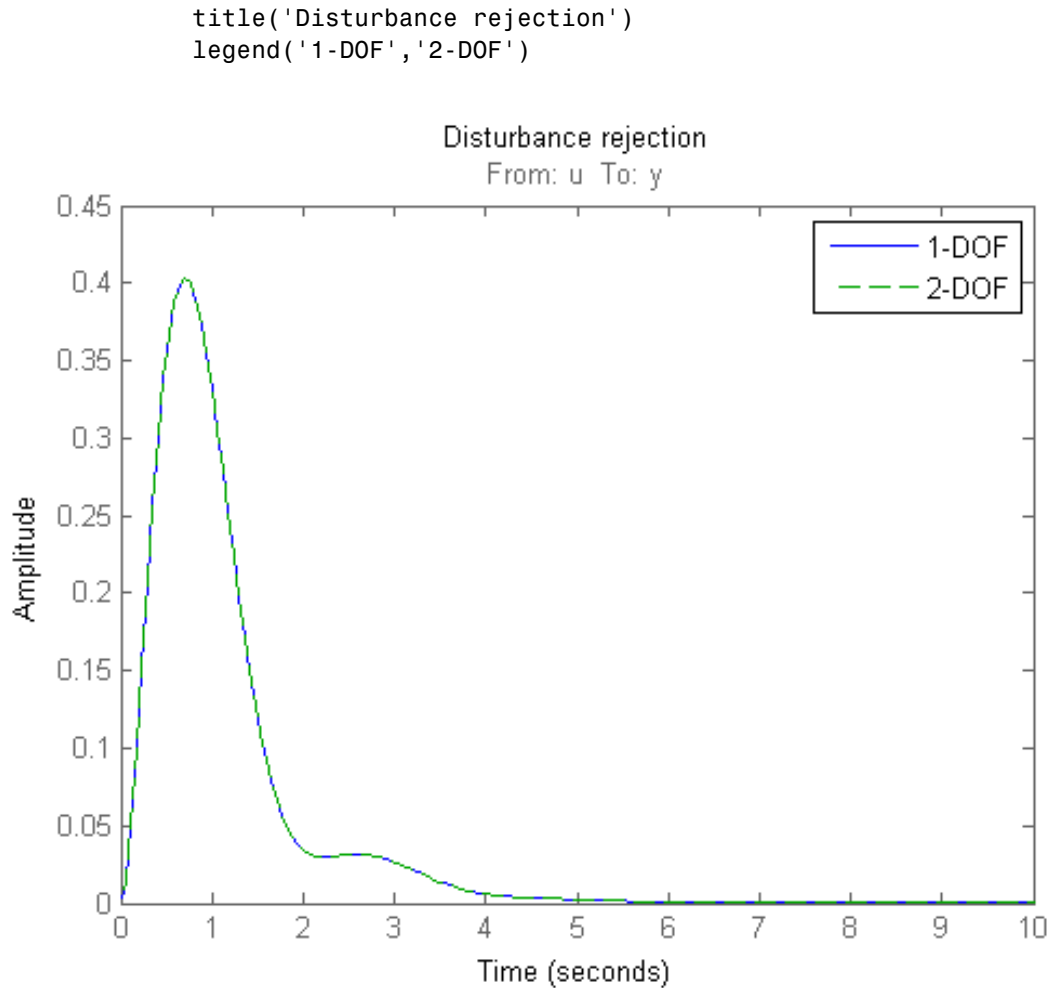
Final: Soft = 1.31, Hard = 0.99992, Iterations = 119

Compare the setpoint tracking and disturbance rejection properties of the 1-DOF and 2-DOF designs for  $\alpha = 5$ .

```
clf, step(T3, 'b', T4, 'g--', 10)
title('Setpoint tracking')
legend('1-DOF', '2-DOF')
```



```
D4 = getIOTransfer(T4, 'u', 'y');
step(D3, 'b', D4, 'g--', 10)
```



The responses to a step disturbance are identical but the 2-DOF controller eliminates the overshoot in the response to a setpoint change. Compare the tuned 1-DOF and 2-DOF PI controllers and observe how the proportional and integral gains are nearly the same, the main difference coming from the setpoint weight  $b$ .

```
showTunable(T3) % 1-DOF PI
```

C =

$$K_p + K_i * \frac{1}{s}$$

with  $K_p = 1.94$ ,  $K_i = 2.13$

Name: C

Continuous-time PI controller in parallel form.

`showTunable(T4) % 2-DOF PI`

C =

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 1.9426$ ,  $K_i = 2.1266$ ,  $K_d = 0$ ,  $T_f = 1$ ,  $b = 0.32537$ ,  $c = 1$ .

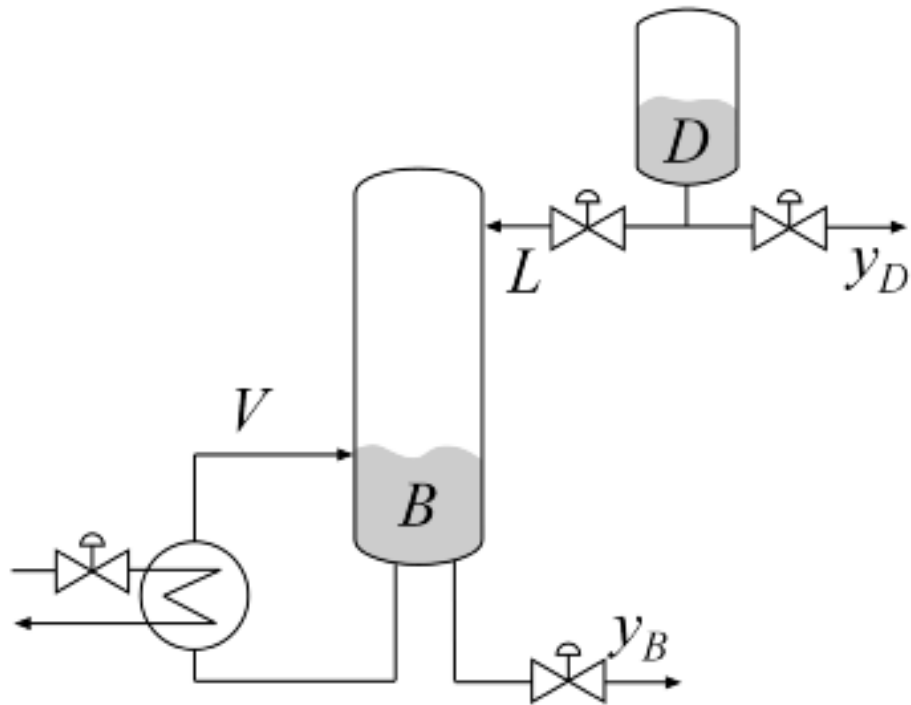
Continuous-time 2-DOF PID controller.

## Decoupling Controller for a Distillation Column

This example shows how to use Robust Control Toolbox™ to decouple the two main feedback loops in a distillation column.

### Distillation Column Model

This example uses a simple model of the distillation column shown below.



**Figure 1: Distillation Column**

In the so-called LV configuration, the controlled variables are the concentrations  $y_D$  and  $y_B$  of the chemicals D (tops) and B (bottoms), and

the manipulated variables are the reflux L and boilup V. This process exhibits strong coupling and large variations in steady-state gain for some combinations of L and V. For more details, see Skogestad and Postlethwaite, *Multivariable Feedback Control*.

The plant is modeled as a first-order transfer function with inputs L, V and outputs yD, yB:

$$G(s) = \frac{1}{75s + 1} \begin{pmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{pmatrix}$$

```
s = tf('s', 'TimeUnit', 'minutes');
G = [87.8 -86.4 ; 108.2 -109.6]/(75*s+1);
G.InputName = {'L', 'V'};
G.OutputName = {'yD', 'yB'};
```

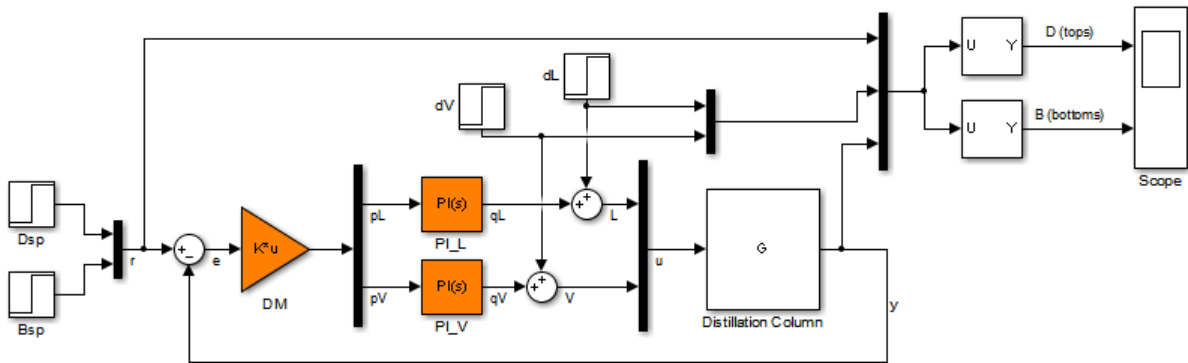
### Control Architecture

The control objectives are as follows:

- Independent control of the tops and bottoms concentrations by ensuring that a change in the tops setpoint Dsp has little impact on the bottoms concentration B and vice versa
- Response time of about 15 minutes
- Fast rejection of input disturbances affecting the effective reflux L and boilup V

To achieve these objectives we use the control architecture shown below. This architecture consists of a static decoupling matrix DM in series with two PI controllers for the reflux L and boilup V.

```
open_system('rct_distillation')
```



Decoupling controller for a distillation column

### Controller Tuning in Simulink with LOOPTUNE

The looptune command provides a quick way to tune MIMO feedback loops. When the control system is modeled in Simulink, you just specify the tuned blocks, the control and measurement signals, and the desired bandwidth, and looptune automatically sets up the problem and tunes the controller parameters. looptune shapes the open-loop response to provide integral action, roll-off, and adequate MIMO stability margins.

Use the sITunable interface to specify the tuned blocks, plant/controller boundary, and signals for closed-loop validation.

```
% Tuned blocks (specified by their name in the model)
STO = sITunable('rct_distillation',{'PI_L','PI_V','DM'});

% Control and measurement signals
STO.addControl({'L','V'})
STO.addMeasurement('y')

% Analysis I/Os
STO.addIO('r','input')           % reference inputs
STO.addIO({'dL','dV'},'input')   % disturbances
```

Use `TuningGoal` objects to express the design requirements. Specify that each loop should respond to a step command in about 15 minutes with minimum cross-coupling.

```
TR = TuningGoal.Tracking('r','y',15);
```

Set the control bandwidth by using `[0.1,0.5]` as gain crossover band for the open-loop response.

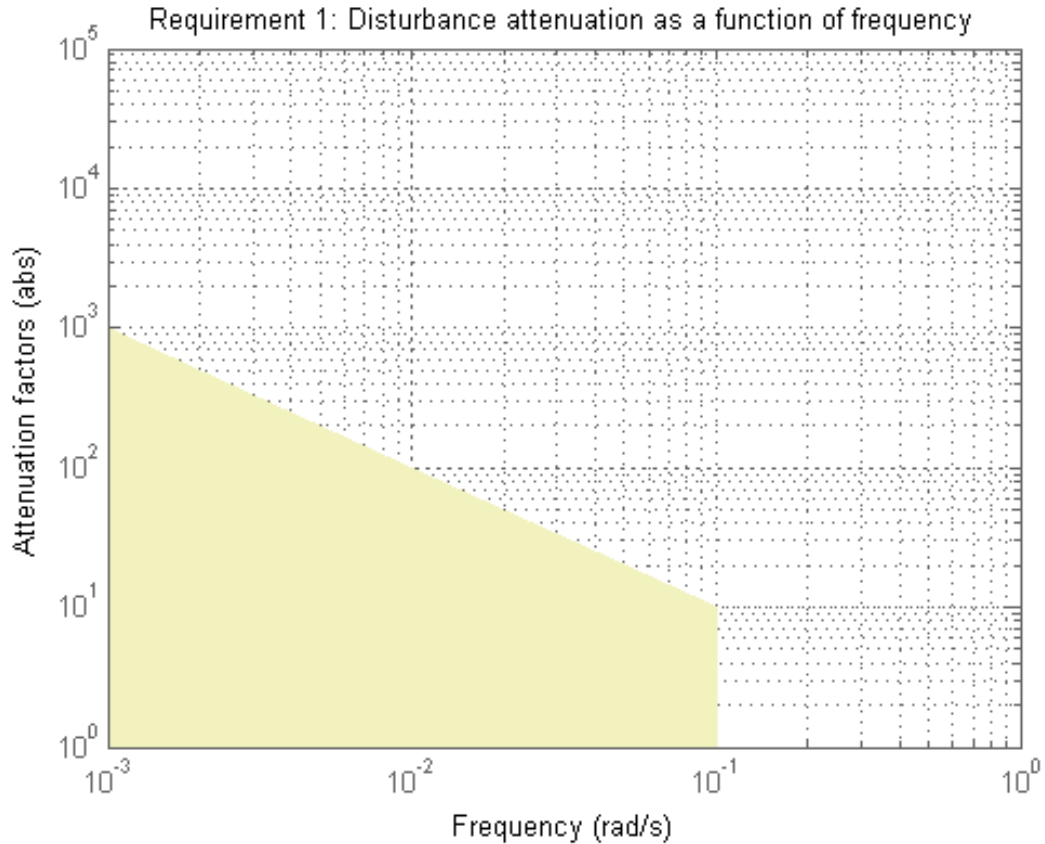
```
wc = [0.1,0.5];
```

To enforce fast disturbance rejection, specify that the disturbance attenuation factor should be at least 20 dB at 0.1 rad/s, increasing to infinity at 0 rad/s since the controller has integral action.

```
DR = TuningGoal.Rejection({'L','V'},1/s);  
DR.Focus = [0 0.1]; % enforced between 0 and 0.1 rad/s
```

```
viewSpec(DR)
```





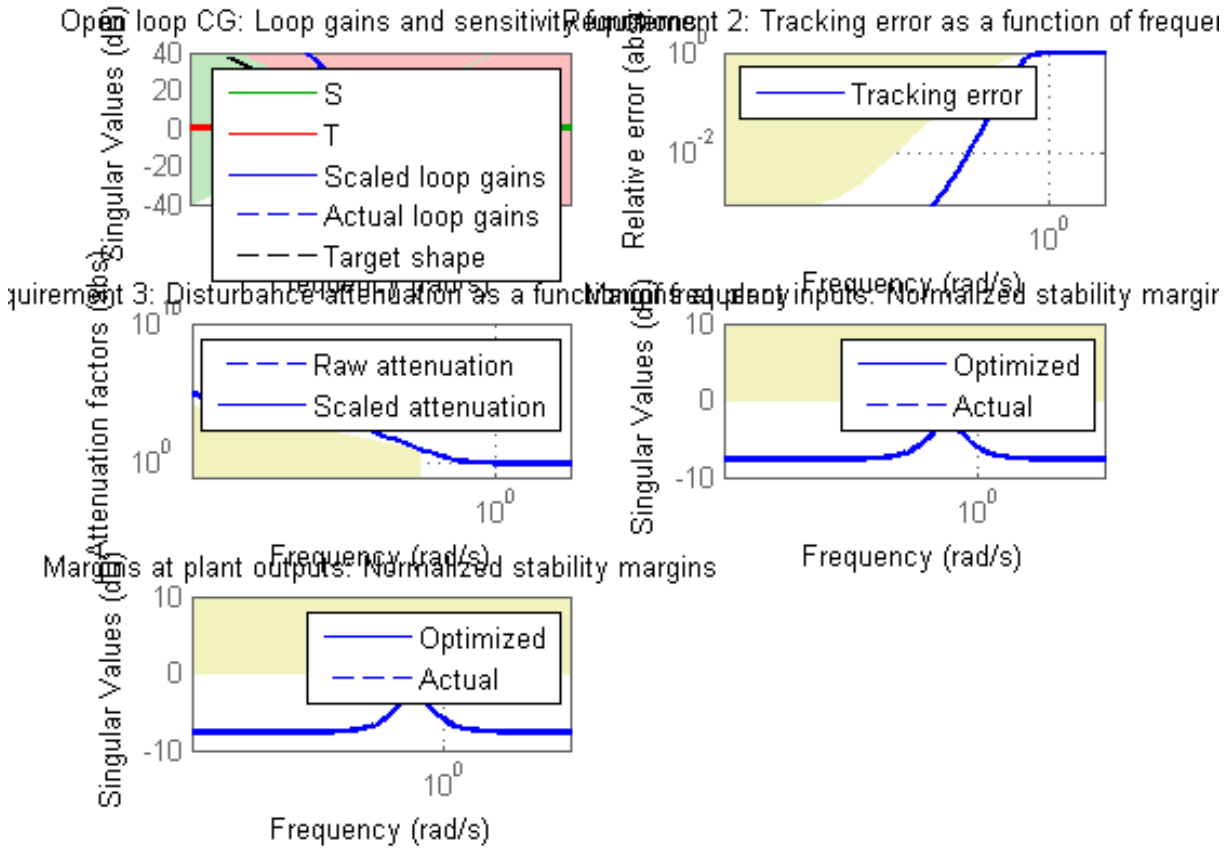
Next use `looptune` to tune the controller blocks `PI_L`, `PI_V`, and `DM` subject to these requirements.

```
[ST,gam,Info] = ST0.looptune(wc,TR,DR);
```

```
Final: Peak gain = 1, Iterations = 46
```

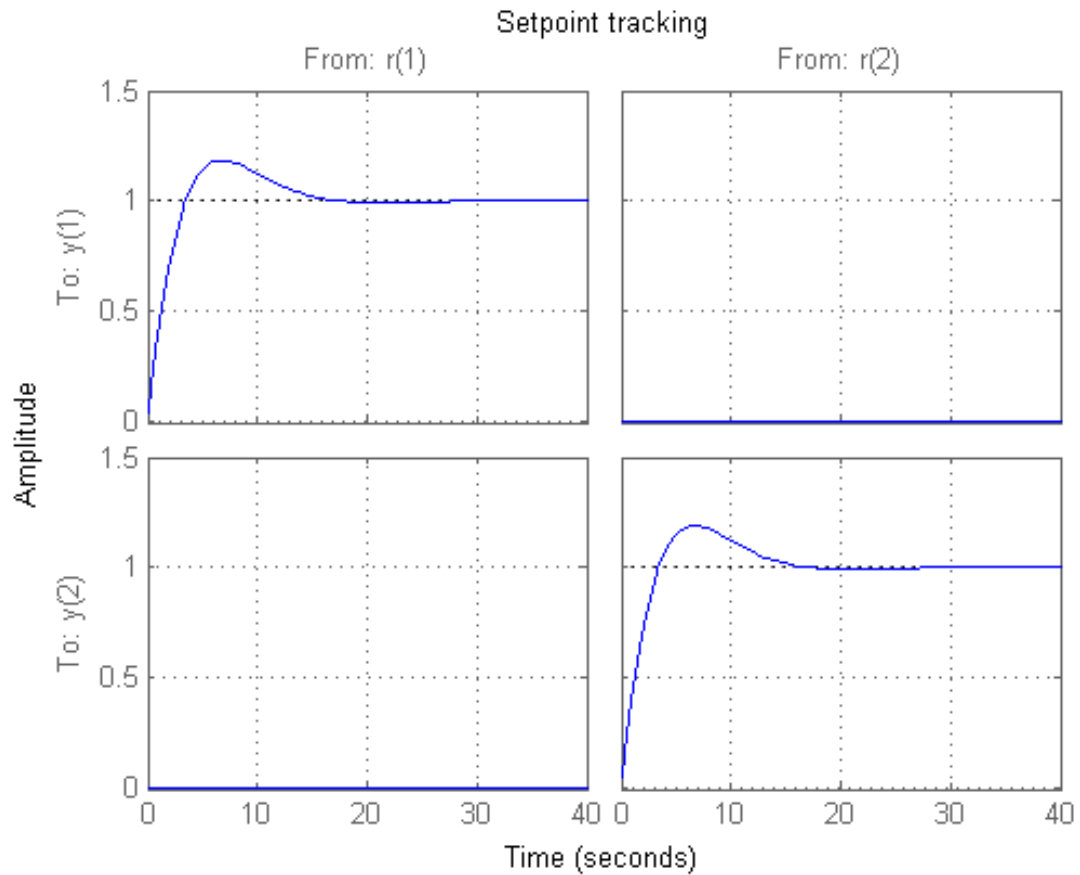
The final value is near 1 which indicates that all requirements were met. Use `loopview` to check the resulting design. The responses should stay outside the shaded areas.

ST.loopview(Info)

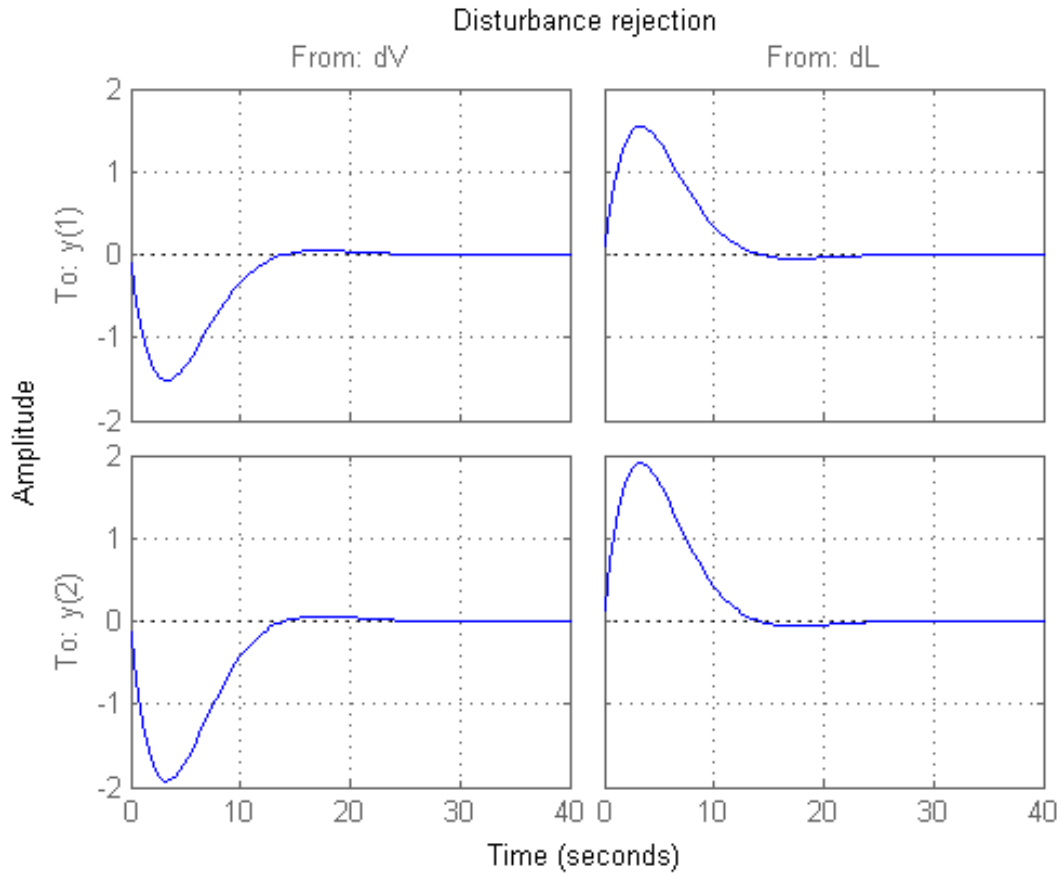


Use `getIOTransfer` to access and plot the closed-loop responses from reference and disturbance to the tops and bottoms concentrations. The tuned responses show a good compromise between tracking and disturbance rejection.

```
clf
Ttrack = ST.getIOTransfer('r','y');
step(Ttrack,40), grid, title('Setpoint tracking')
```

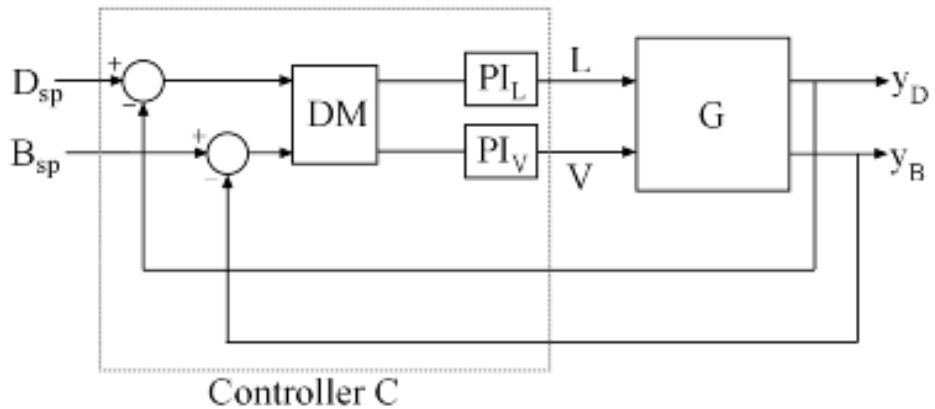


```
Treject = ST.getIOTransfer({'dV','dL'},'y');  
step(Treject,40), grid, title('Disturbance rejection')
```



**Equivalent Workflow in MATLAB**

If you do not have a Simulink model of the control system, you can use LTI objects and Control Design blocks to create a MATLAB representation of the following block diagram.



**Figure 2: Block Diagram of Control System**

First parameterize the tunable elements using Control Design blocks. Use the `ltiblock.gain` block to parameterize DM:

```
DM = ltiblock.gain('Decoupler', eye(2));
```

This creates a 2x2 static gain with four tunable parameters. Similarly, use the `ltiblock.pid` block to parameterize the two PI controllers:

```
PI_L = ltiblock.pid('PI_L', 'pi'); PI_L.TimeUnit = 'minutes';
PI_V = ltiblock.pid('PI_V', 'pi'); PI_V.TimeUnit = 'minutes';
```

At this point, the tunable elements are over-parameterized because multiplying DM by two and dividing the PI coefficients by two does not change the overall controller. To eliminate redundant parameters, normalize the PI controllers by fixing their proportional gain Kp to 1:

```
PI_L.Kp.Value = 1; PI_L.Kp.Free = false;
PI_V.Kp.Value = 1; PI_V.Kp.Free = false;
```

Next construct a model C0 of the controller C in Figure 2.

```
C0 = blkdiag(PI_L, PI_V) * DM * [eye(2) -eye(2)];
```

```
% Note: I/O names should be consistent with those of G
CO.InputName = {'Dsp', 'Bsp', 'yD', 'yB'};
CO.OutputName = {'L', 'V'};
```

Now tune the controller parameters with `looptune` as done previously.

```
% Tracking requirement
TR = TuningGoal.Tracking({'Dsp', 'Bsp'}, {'yD', 'yB'}, 15);

% Disturbance rejection requirement
DR = TuningGoal.Rejection({'L', 'V'}, 1/s);
DR.Focus = [0 0.1];

% Crossover band
wc = [0.1, 0.5];

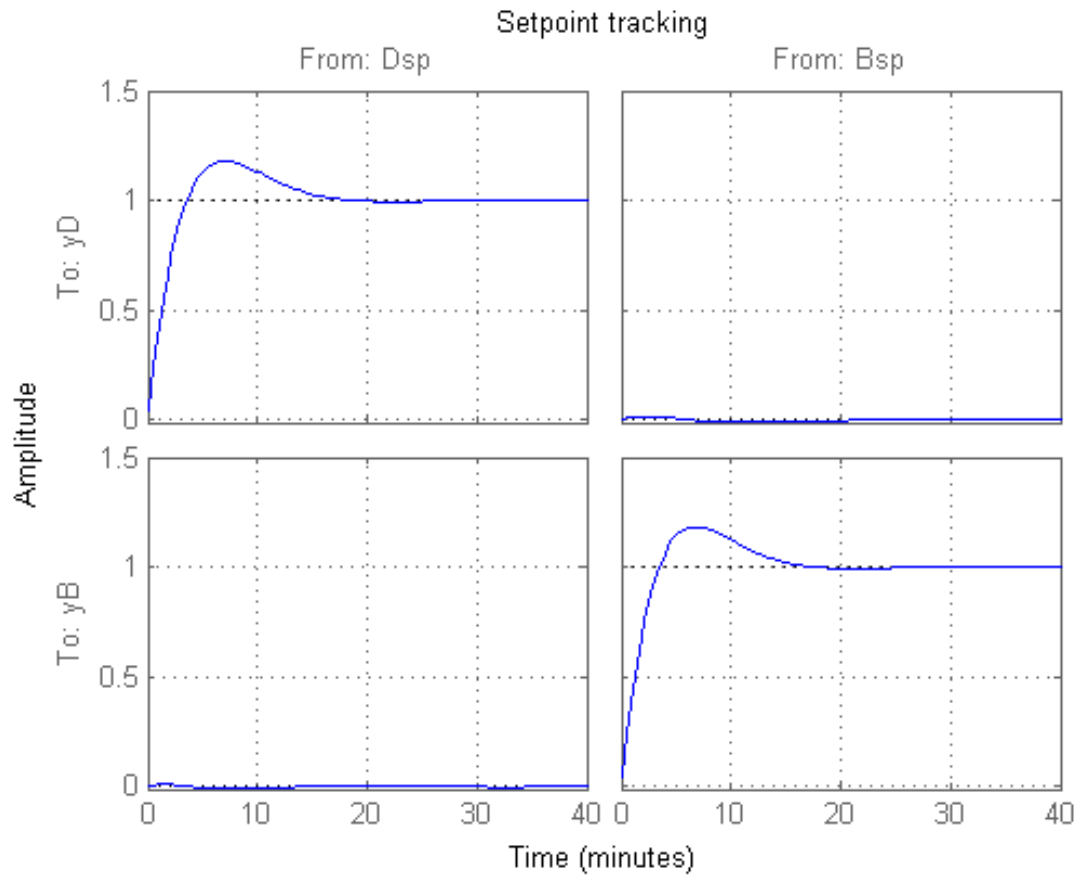
[~, C] = looptune(G, CO, wc, TR, DR);
```

Final: Peak gain = 1, Iterations = 74

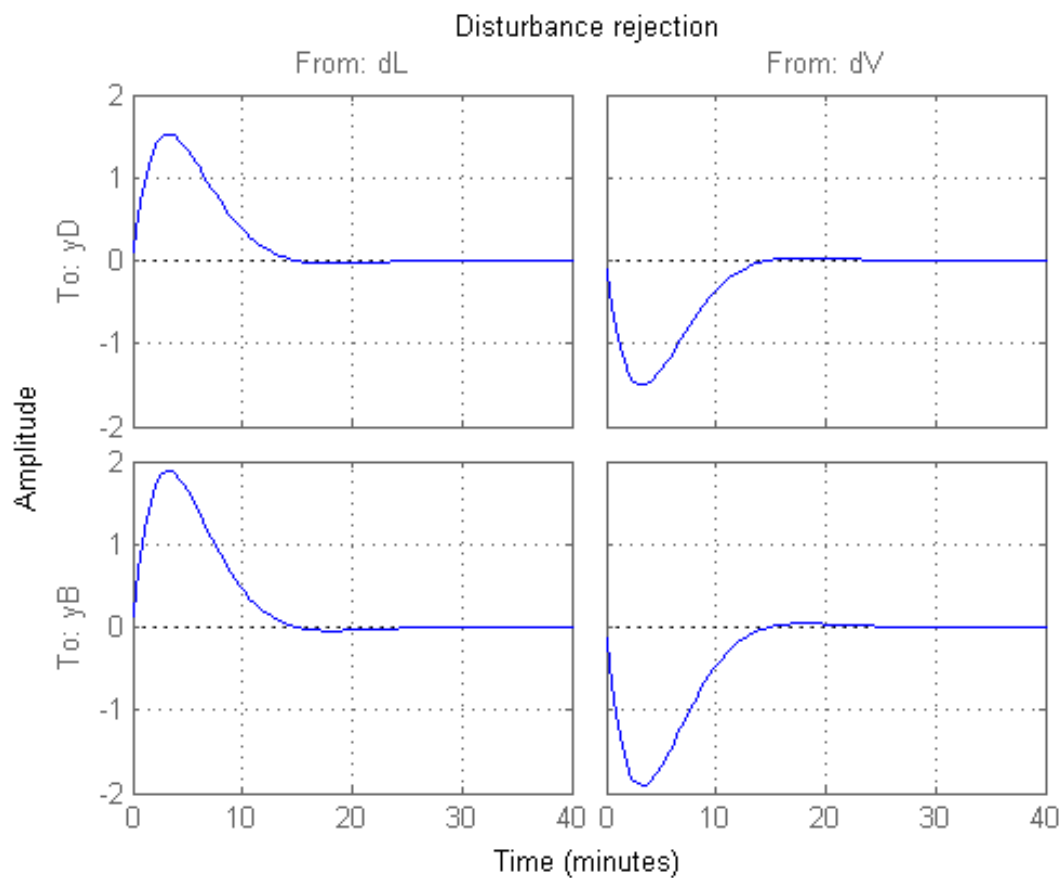
To validate the design, close the loop with the tuned compensator **C** and simulate the step responses for setpoint tracking and disturbance rejection. Also compare the open- and closed-loop disturbance rejection characteristics in the frequency domain.

```
Tcl = connect(G, C, {'Dsp', 'Bsp', 'L', 'V'}, {'yD', 'yB'});

Ttrack = Tcl(:, [1 2]);
step(Ttrack, 40), grid, title('Setpoint tracking')
```



```
Treject = Tc1(:,[3 4]);
Treject.InputName = {'dL','dV'};
step(Treject,40), grid, title('Disturbance rejection')
```

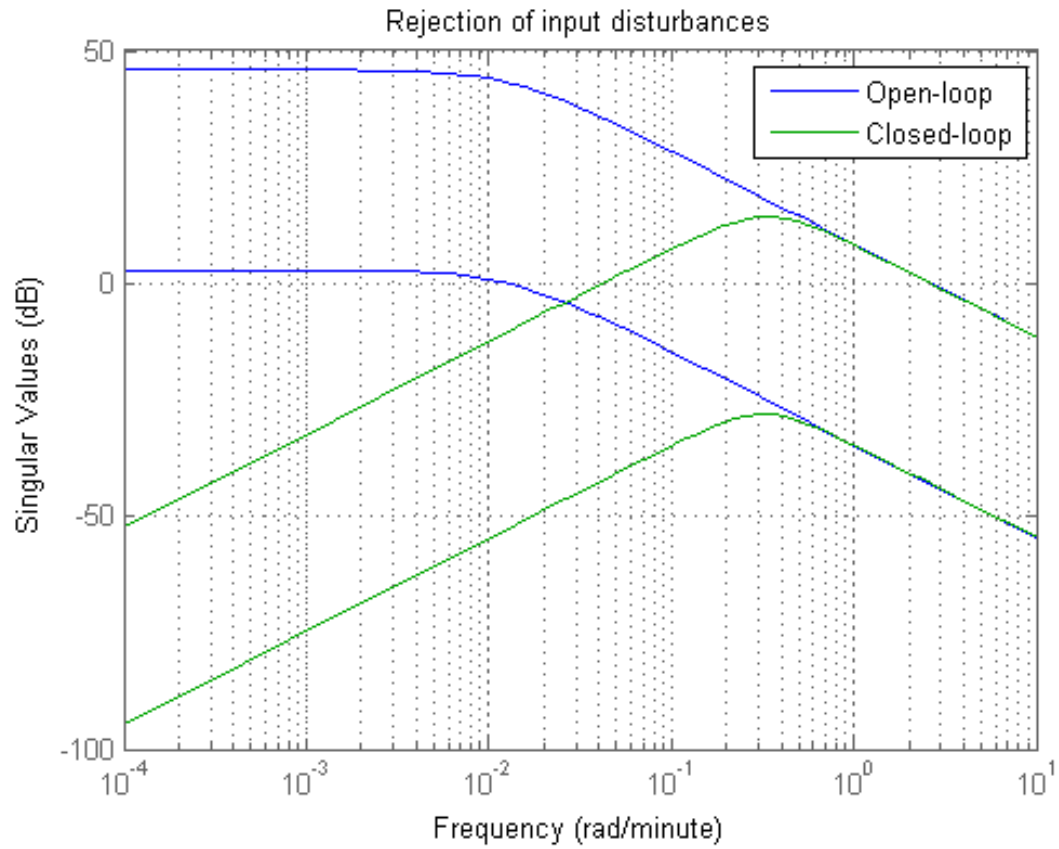


```

clf, sigma(G,Treject), grid
title('Rejection of input disturbances')
legend('Open-loop', 'Closed-loop')

```





The results are similar to those obtained in Simulink.

## Tuning of a Digital Motion Control System

This example shows how to use Robust Control Toolbox™ to tune a digital motion control system.

### Motion Control System

The motion system under consideration is shown below.

### Digital Motion Control Setup

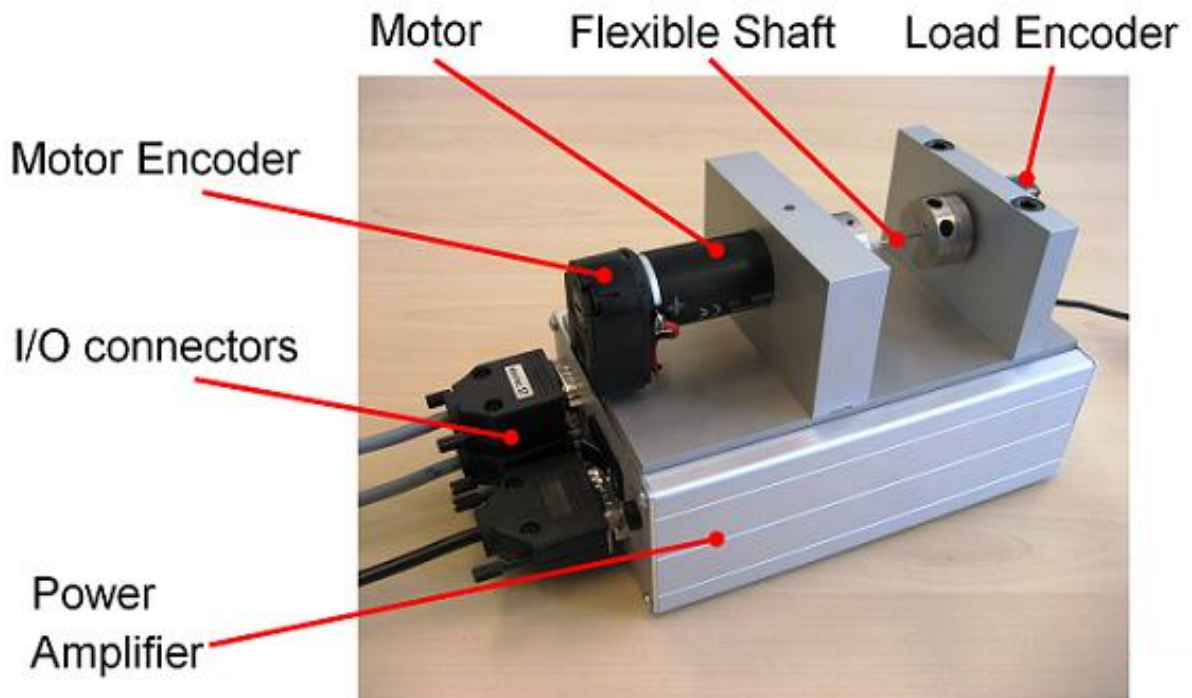


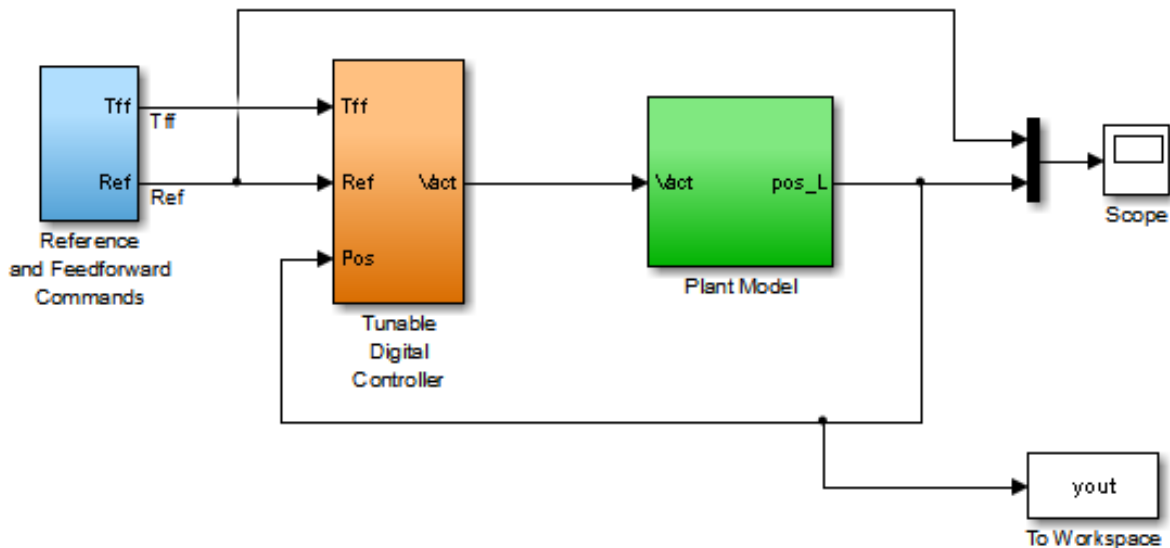
Figure 1: Digital motion control hardware

This device could be part of some production machine and is intended to move some load (a gripper, a tool, a nozzle, or anything else that you can imagine) from one angular position to another and back again. This task is part of the "production cycle" that has to be completed to create each product or batch of products.

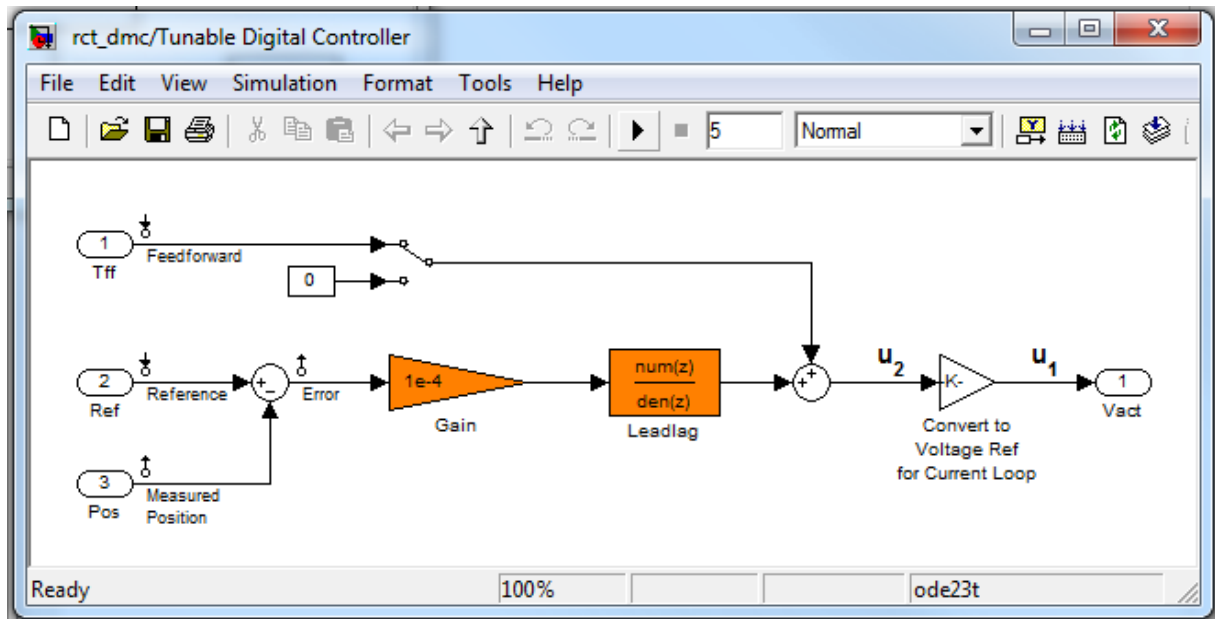
The digital controller must be tuned to maximize the production speed of the machine without compromising accuracy and product quality. To do this, we first model the control system in Simulink using a 4th-order model of the inertia and flexible shaft:

```
open_system('rct_dmc')
```

### Digital Motion Control System



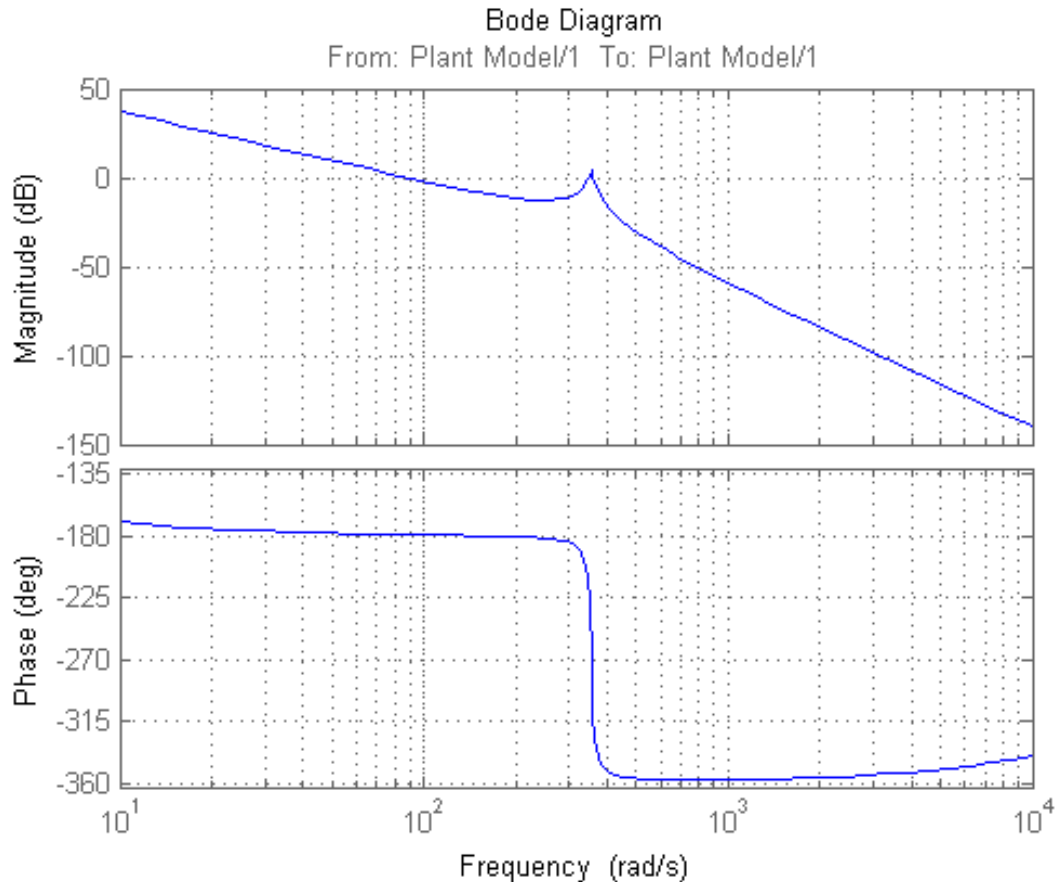
The "Tunable Digital Controller" consists of a gain in series with a lead/lag controller.



**Figure 2: Digital controller**

Tuning is complicated by the presence of a flexible mode near 350 rad/s in the plant:

```
G = linearize('rct_dmc','rct_dmc/Plant Model');
bode(G,{10,1e4}), grid
```



### Compensator Tuning

We are seeking a 0.5 second response time to a step command in angular position with minimum overshoot. This corresponds to a target bandwidth of approximately 5 rad/s. The `looptune` command offers a convenient way to tune fixed-structure compensators like the one in this application. To use `looptune`, first instantiate the `sITunable` interface to automatically acquire the control structure from Simulink. This requires specifying the tuned blocks and the control and measurement signals (to delimit the controller).

```
STO = sITunable('rct_dmc',{'Gain','Leadlag'});
```

```
ST0.addControl('Leadlag');
ST0.addMeasurement('Measured Position');
```

Next use `looptune` to tune the compensator parameters for the target gain crossover frequency of 5 rad/s:

```
ST1 = ST0.looptune(5);
```

```
Final: Peak gain = 0.975, Iterations = 21
```

A final value below or near 1 indicates success. Inspect the tuned values of the gain and lead/lag filter:

```
ST1.showBlockValue
```

```
Block "rct_dmc/Tunable Digital Controller/Gain" =
```

```

d =
      u1
y1  2.753e-06
```

```
Name: Gain
Static gain.
```

```
-----
Block "rct_dmc/Tunable Digital Controller/Leadlag" =
```

```

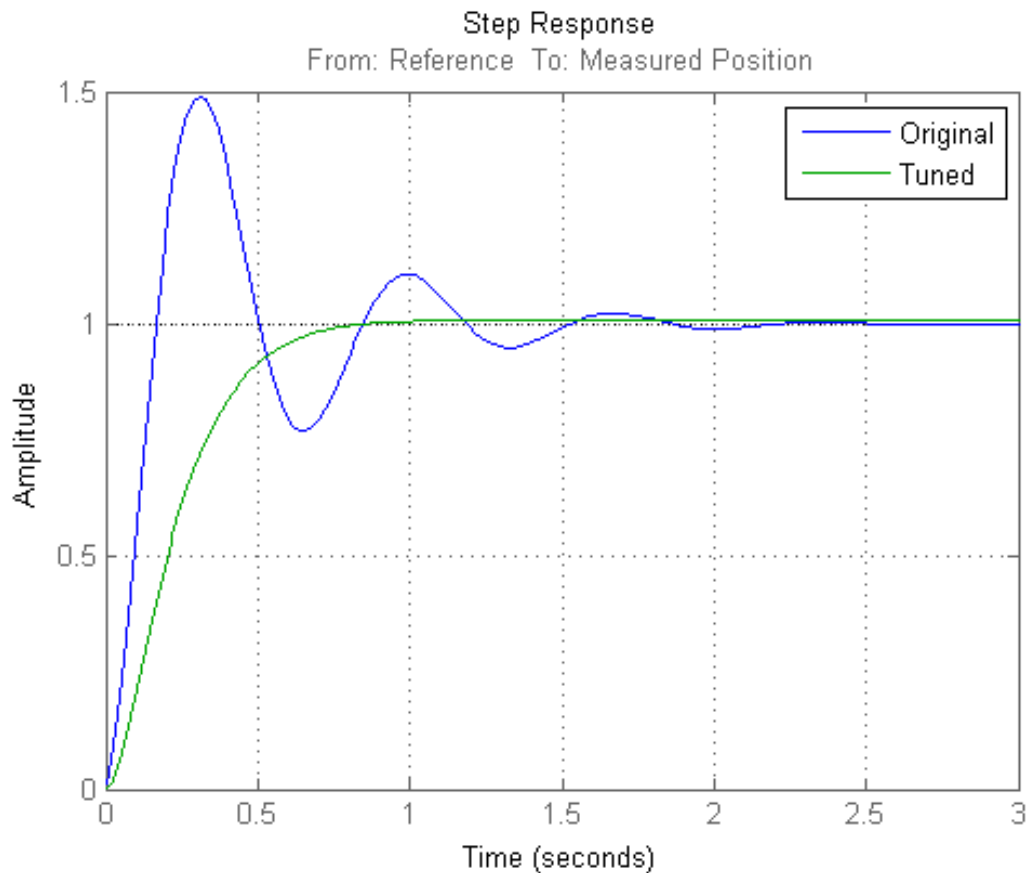
30.54 s + 59.01
-----
      s + 18.94
```

```
Name: Leadlag
Continuous-time transfer function.
```

### Design Validation

To validate the design, use the `slTunable` interface to quickly access the closed-loop transfer functions of interest and compare the responses before and after tuning.

```
T0 = ST0.getIOTransfer('Reference','Measured Position');  
T1 = ST1.getIOTransfer('Reference','Measured Position');  
step(T0,T1), grid  
legend('Original','Tuned')
```



The tuned response has significantly less overshoot and satisfies the response time requirement. However these simulations are obtained using

a continuous-time lead/lag compensator (`looptune` operates in continuous time) so we need to further validate the design in Simulink using a digital implementation of the lead/lag compensator. Use `writeBlockValue` to apply the tuned values to the Simulink model and automatically discretize the lead/lag compensator to the rate specified in Simulink.

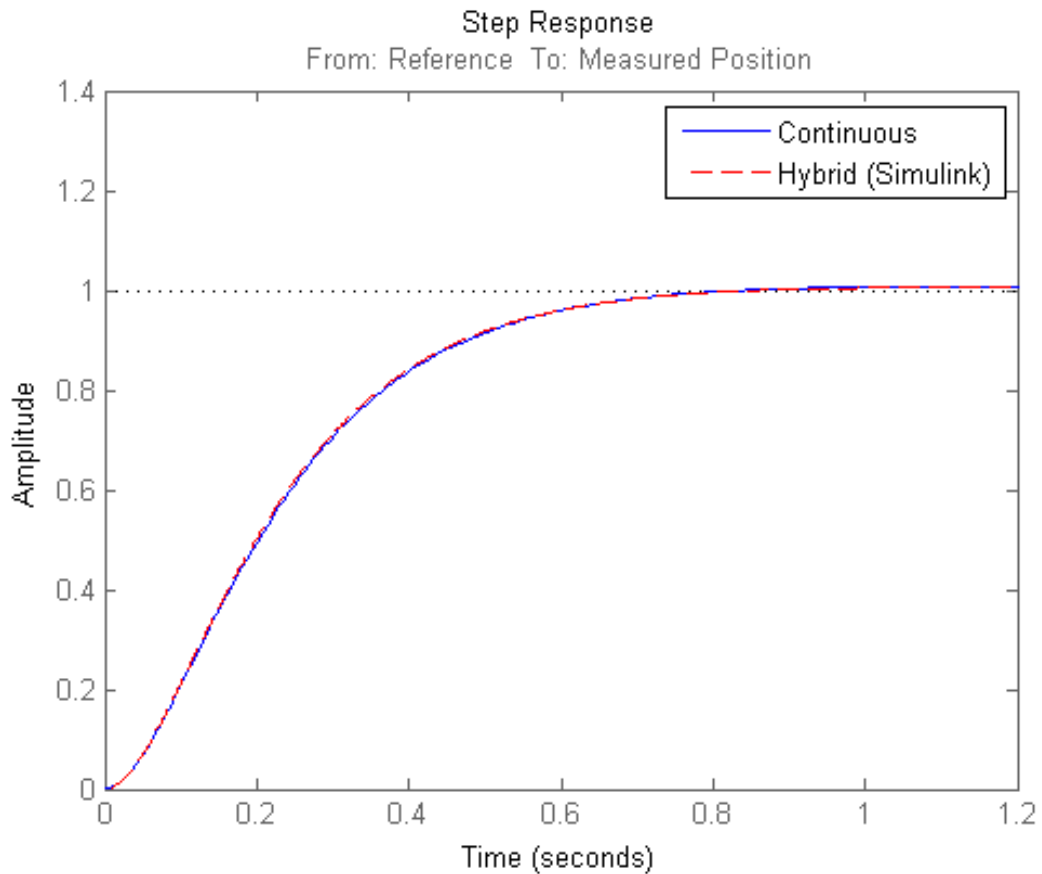
```
ST1.writeBlockValue
```

You can now simulate the response of the continuous-time plant with the digital controller:

```
sim('rct_dmc'); % angular position logged in "yout" variable
t = yout.time;
y = yout.signals.values;
step(T1), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')
```

```
Current plot held
```

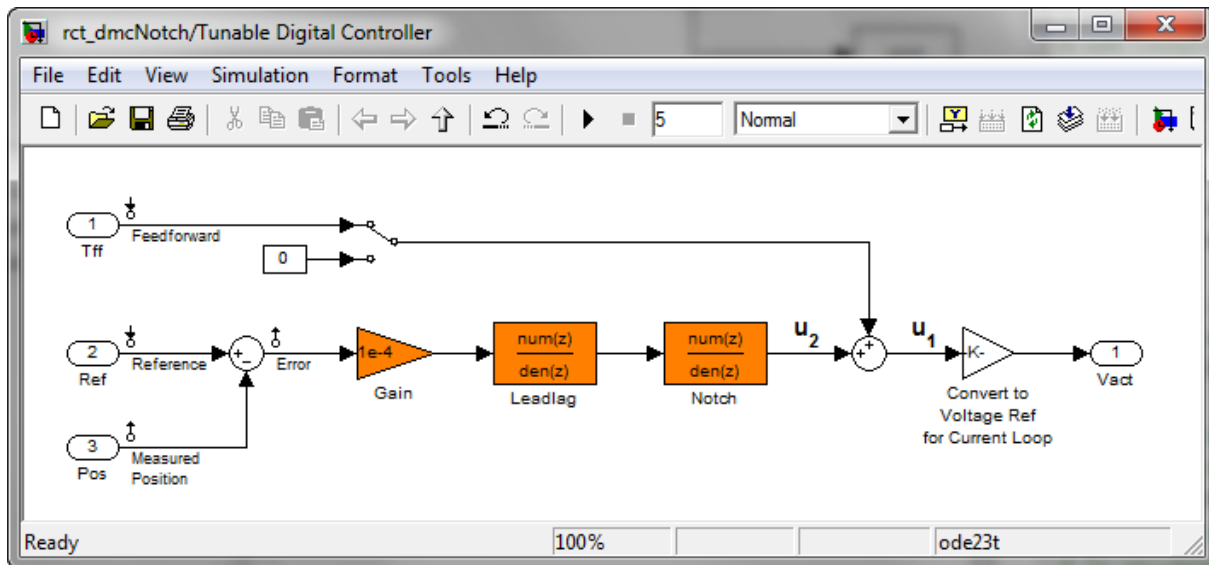




The simulations closely match and the coefficients of the digital lead/lag can be read from the "Leadlag" block in Simulink.

#### **Tuning an Additional Notch Filter**

Next try to increase the control bandwidth from 5 to 50 rad/s. Because of the plant resonance near 350 rad/s, the lead/lag compensator is no longer sufficient to get adequate stability margins and small overshoot. One remedy is to add a notch filter as shown in Figure 3.



**Figure 3: Digital Controller with Notch Filter**

To tune this modified control architecture, specify the tunable blocks and controller I/Os as before:

```
STO = sITunable('rct_dmcNotch',{'Gain','Leadlag','Notch'});
STO.addControl('Notch');
STO.addMeasurement('Measured Position');
```

By default the "Notch" block is parameterized as any second-order transfer function. To retain the notch structure

$$N(s) = \frac{s^2 + 2\zeta_1\omega_n s + \omega_n^2}{s^2 + 2\zeta_2\omega_n s + \omega_n^2},$$

specify the coefficients  $\omega_n$ ,  $\zeta_1$ ,  $\zeta_2$  as real parameters and create a parametric model N of the transfer function shown above:

```
wn = realp('wn',300);
zeta1 = realp('zeta1',1); zeta1.Maximum = 1; % zeta1 <= 1
```

```
zeta2 = realp('zeta2',1); zeta2.Maximum = 1; % zeta2 <= 1
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]); % tunable notch filter
```

Then associate this parametric notch model with the "Notch" block in the Simulink model. Because the control system is tuned in the continuous time, you can use a continuous-time parameterization of the notch filter even though the "Notch" block itself is discrete.

```
ST0.setBlockParam('Notch',N);
```

Next use `looptune` to jointly tune the "Gain", "Leadlag", and "Notch" blocks with a 50 rad/s target crossover frequency. To eliminate residual oscillations from the plant resonance, specify a target loop shape with a -40 dB/decade roll-off past 50 rad/s.

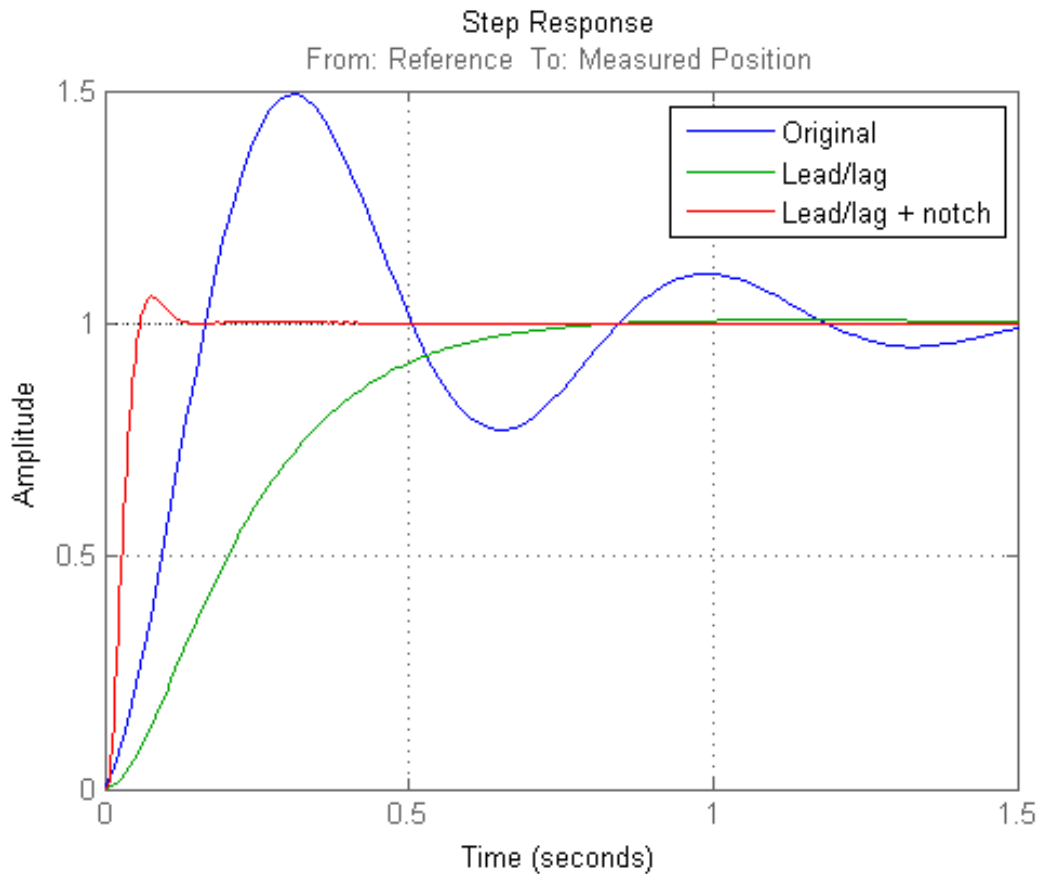
```
% Specify target loop shape with a few frequency points
Freqs = [5 50 500];
Gains = [10 1 0.01];
TLS = TuningGoal.LoopShape('Notch',frd(Gains,Freqs));
```

```
ST2 = ST0.looptune(TLS);
```

```
Final: Peak gain = 1.05, Iterations = 67
```

The final gain is close to 1, indicating that all requirements are met. Compare the closed-loop step response with the previous designs.

```
T2 = ST2.getIOTransfer('Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')
```

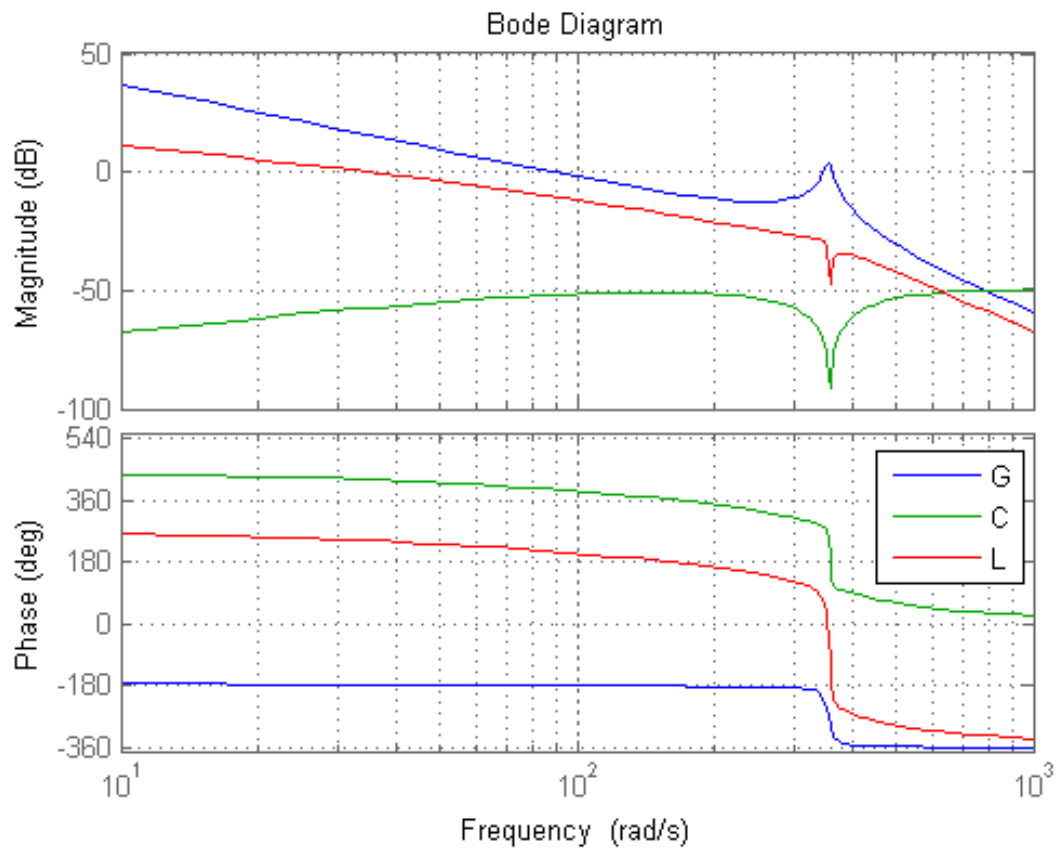


To verify that the notch filter performs as expected, evaluate the total compensator  $C$  and the open-loop response  $L$  and compare the Bode responses of  $G$ ,  $C$ ,  $L$ :

```
% Get tuned block values (in the order blocks are listed in ST2.TunedBlocks)
[g,LL,N] = ST2.getBlockValue();
C = N * LL * g;

L = ST2.getLoopTransfer('Notch',-1);
```

```
bode(G,C,L,{1e1,1e3}), grid
legend('G','C','L')
```



This Bode plot confirms that the plant resonance has been correctly "notched out."

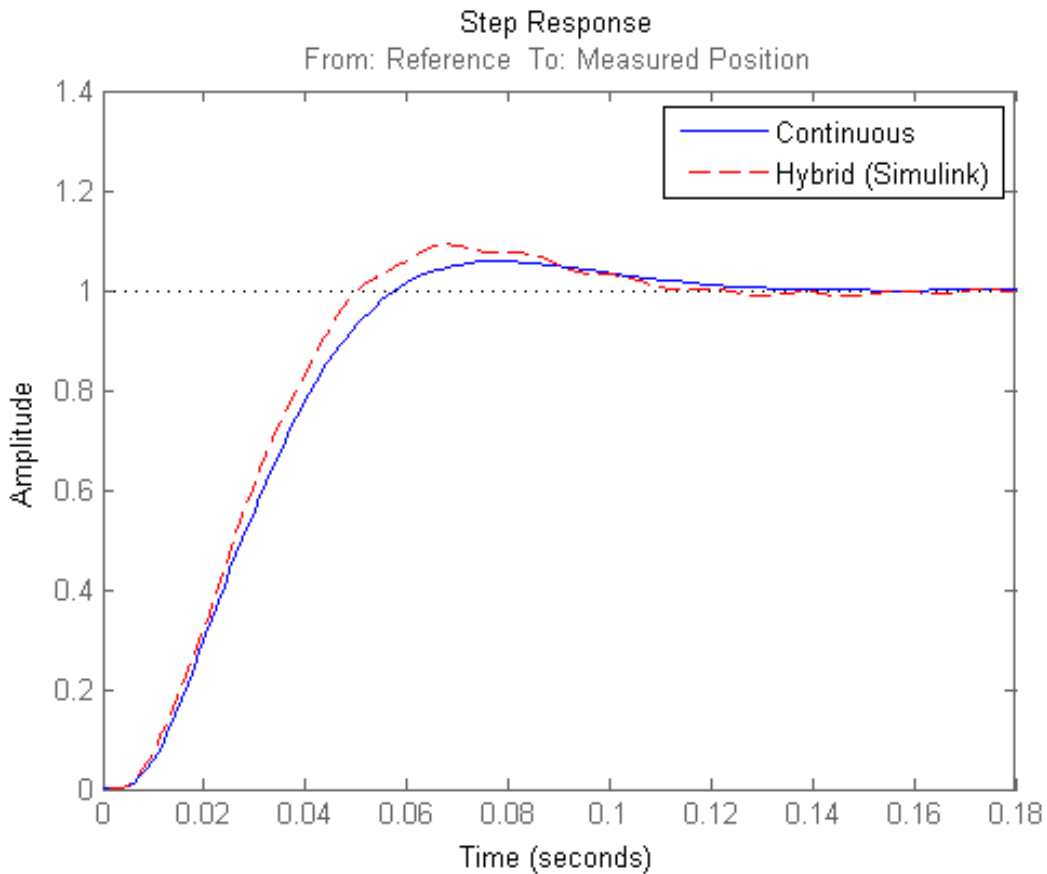
#### Discretizing the Notch Filter

Again use `writeBlockValue` to discretize the tuned lead/lag and notch filters and write their values back to Simulink. Compare the MATLAB and Simulink responses:

```
ST2.writeBlockValue
```

```
sim('rct_dmcNotch');  
t = yout.time;  
y = yout.signals.values;  
step(T2), hold, plot(t,y,'r--')  
legend('Continuous','Hybrid (Simulink)')
```

```
Current plot held
```



The Simulink response exhibits small residual oscillations. The notch filter discretization is the likely culprit because the notch frequency is close to the Nyquist frequency  $\pi/0.002=1570$  rad/s. By default the notch is discretized using the ZOH method. Compare this with the Tustin method prewarped at the notch frequency:

```

wn = damp(N); % natural frequency of the notch filter
Ts = 0.002; % sample time of discrete notch filter

Nd1 = c2d(N,Ts,'zoh');

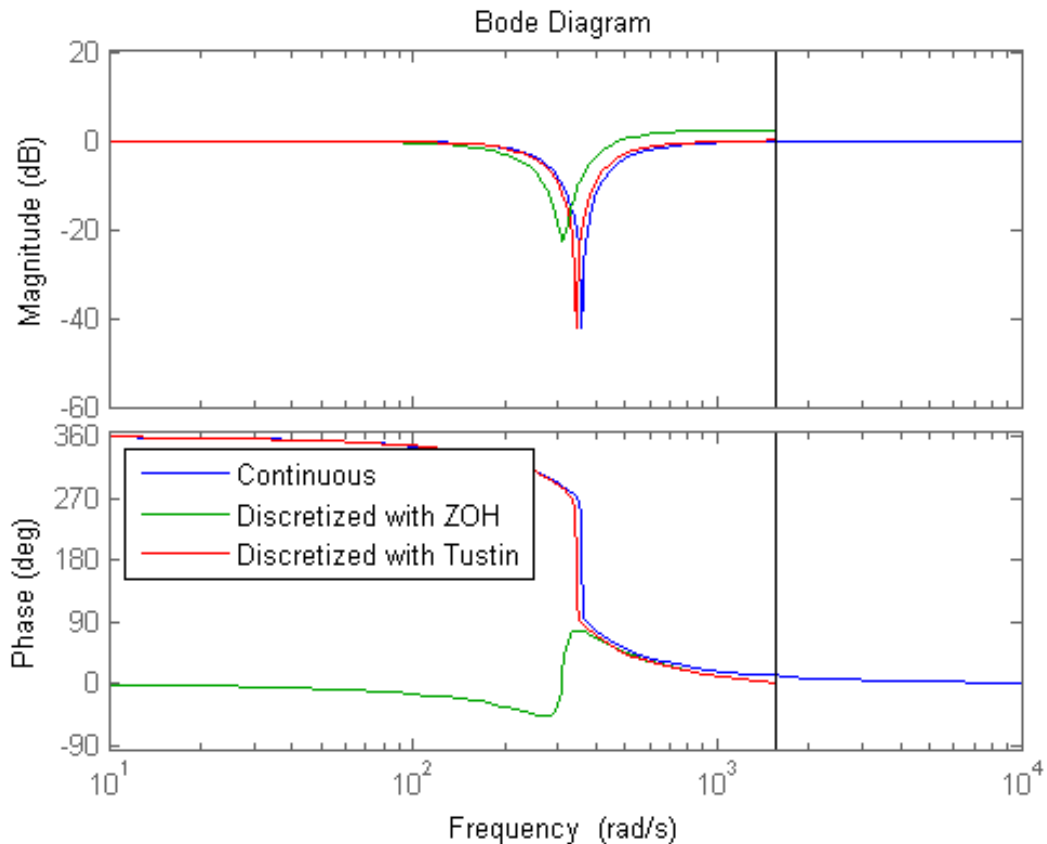
```

```

Nd2 = c2d(N,Ts,'tustin',c2dOptions('PrewarpFrequency',wn(1)));

clf, bode(N,Nd1,Nd2)
legend('Continuous','Discretized with ZOH','Discretized with Tustin',...
      'Location','NorthWest')

```



The ZOH method has significant distortion and prewarped Tustin should be used instead. To do this, specify the desired rate conversion method for the notch filter block:

```
ST2.setBlockRateConversion('Notch','tustin',wn(1))
```

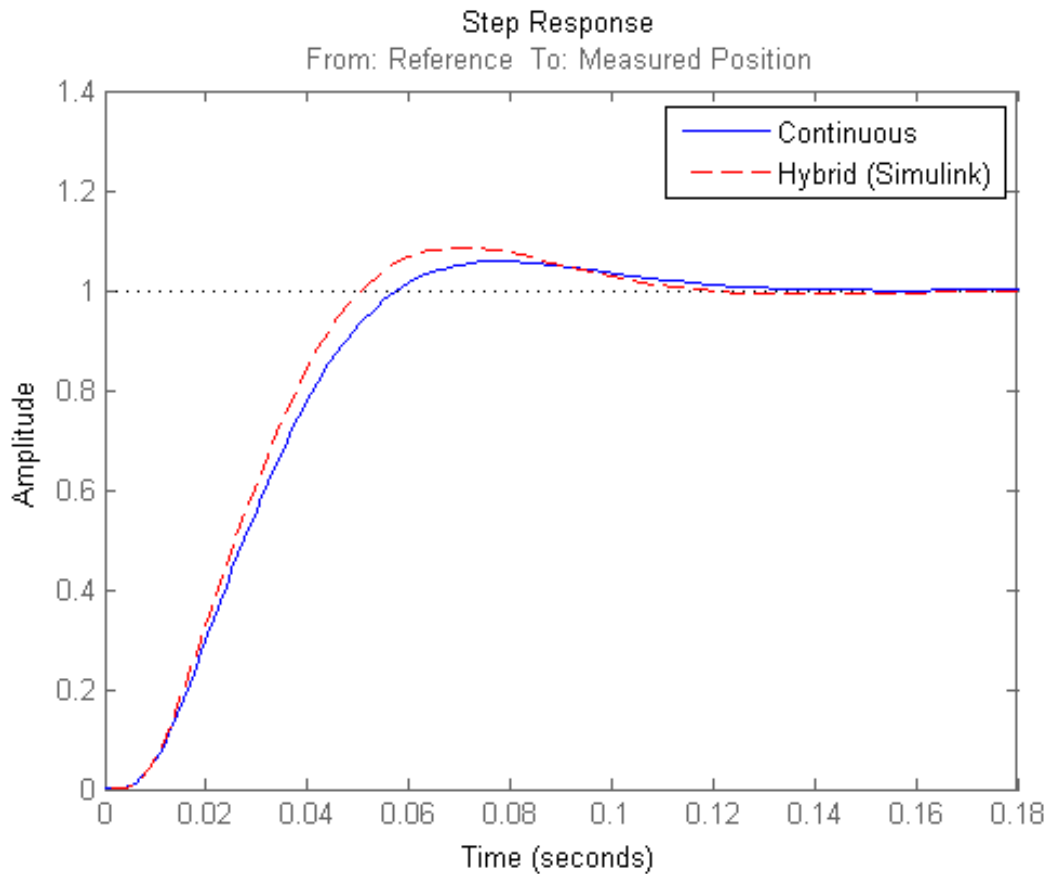


ST2.writeBlockValue

writeBlockValue now uses Tustin prewarped at the notch frequency to discretize the notch filter and write it back to Simulink. Verify that this gets rid of the oscillations.

```
sim('rct_dmcNotch');  
t = yout.time;  
y = yout.signals.values;  
step(T2), hold, plot(t,y,'r--')  
legend('Continuous','Hybrid (Simulink)')
```

Current plot held



### Discrete-Time Tuning

Alternatively, you can tune the controller directly in discrete time to avoid discretization issues with the notch filter. To do this, specify that the Simulink model should be linearized and tuned at the controller sample time of 0.002 seconds:

```
STO.Ts = 0.002;
```

To prevent high-gain control and saturations, add a requirement that limits the gain from reference to control signal (output of Notch block).

```
GL = TuningGoal.Gain('Reference', 'Notch', 0.01);
```

Now retune the controller at the specified sampling rate and verify the tuned open- and closed-loop responses.

```
ST2 = ST0.looptune(TLS, GL);
```

```
% Closed-loop responses
```

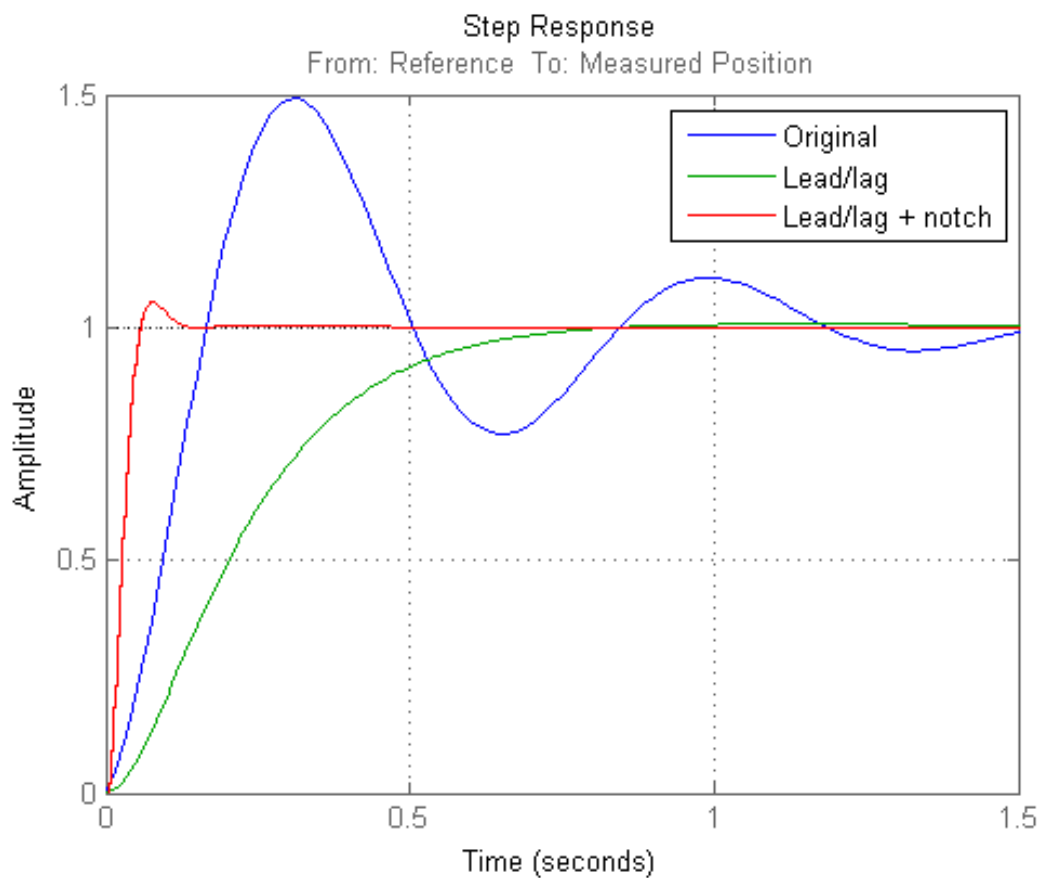
```
T2 = ST2.getIOTransfer('Reference', 'Measured Position');
```

```
clf
```

```
step(T0, T1, T2, 1.5), grid
```

```
legend('Original', 'Lead/lag', 'Lead/lag + notch')
```

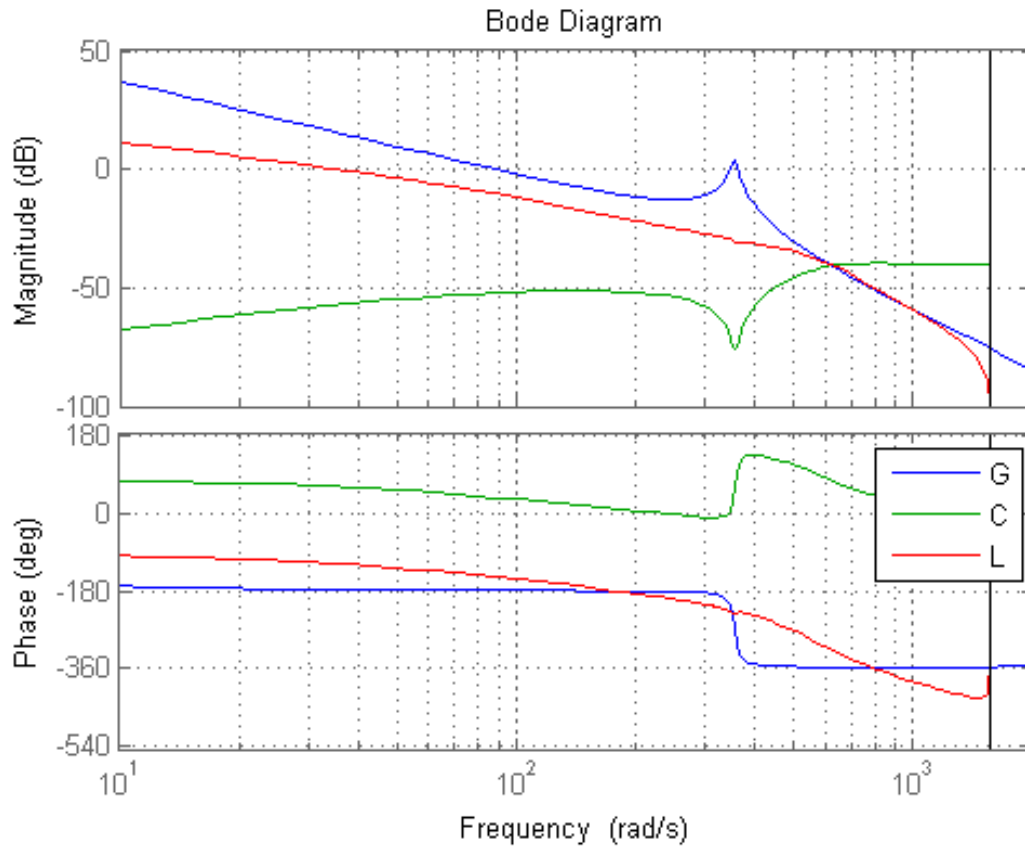
```
Final: Peak gain = 1.04, Iterations = 50
```



```

% Open-loop responses
[g,LL,N] = ST2.getBlockValue();
C = N * LL * g;
L = ST2.getLoopTransfer('Notch',-1);
bode(G,C,L,{1e1,2e3}), grid
legend('G','C','L')

```



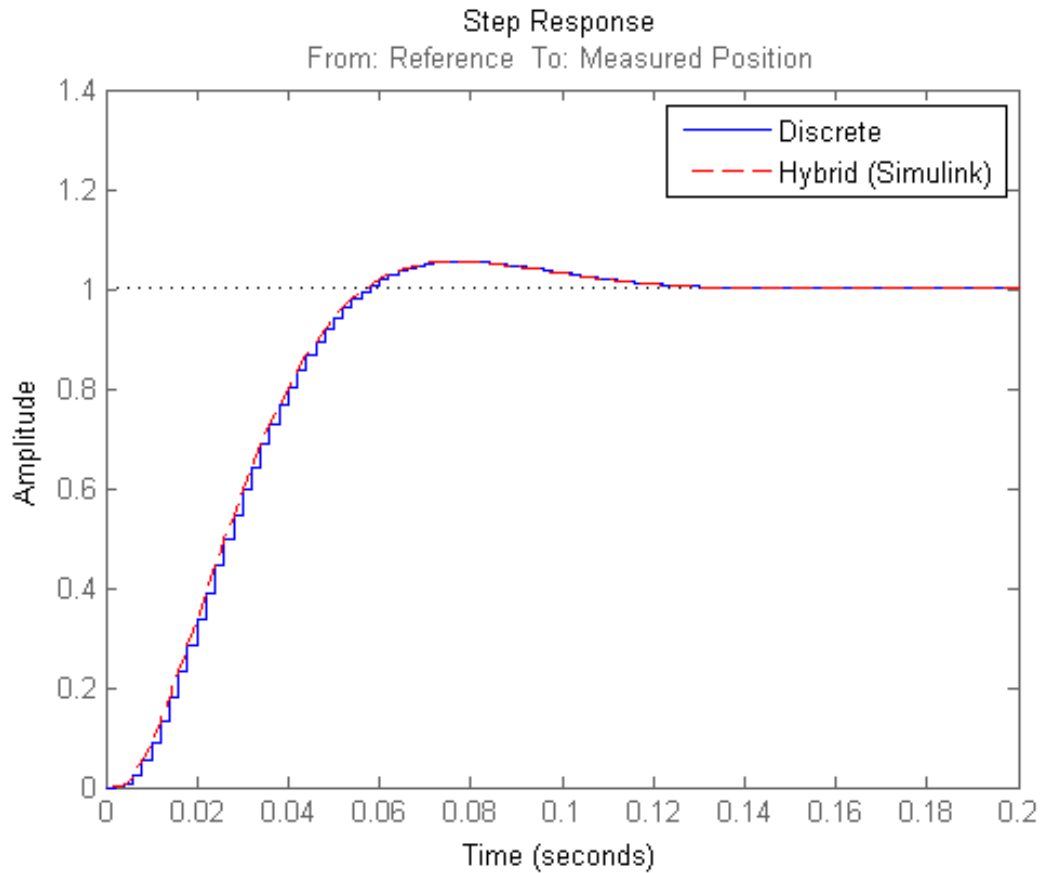
The results are similar to those obtained when tuning the controller in continuous time. Now validate the digital controller against the continuous-time plant in Simulink.

```
ST2.writeBlockValue
```

```
sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
```

```
legend('Discrete','Hybrid (Simulink)')
```

Current plot held



This time, the hybrid response closely matches its discrete-time approximation and no further adjustment of the notch filter is required.

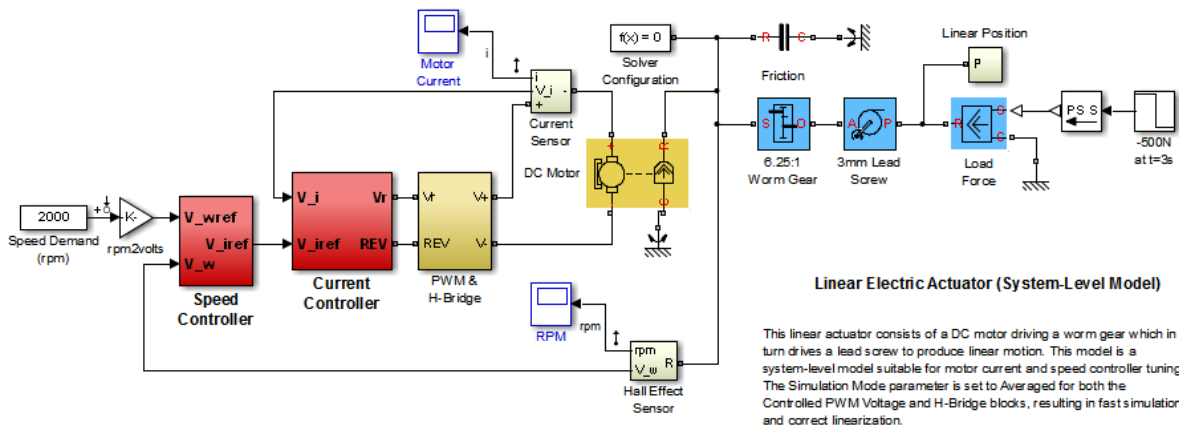
## Control of a Linear Electric Actuator

This example shows how to use Robust Control Toolbox™ to tune the current and velocity loops in a linear electric actuator with saturation limits.

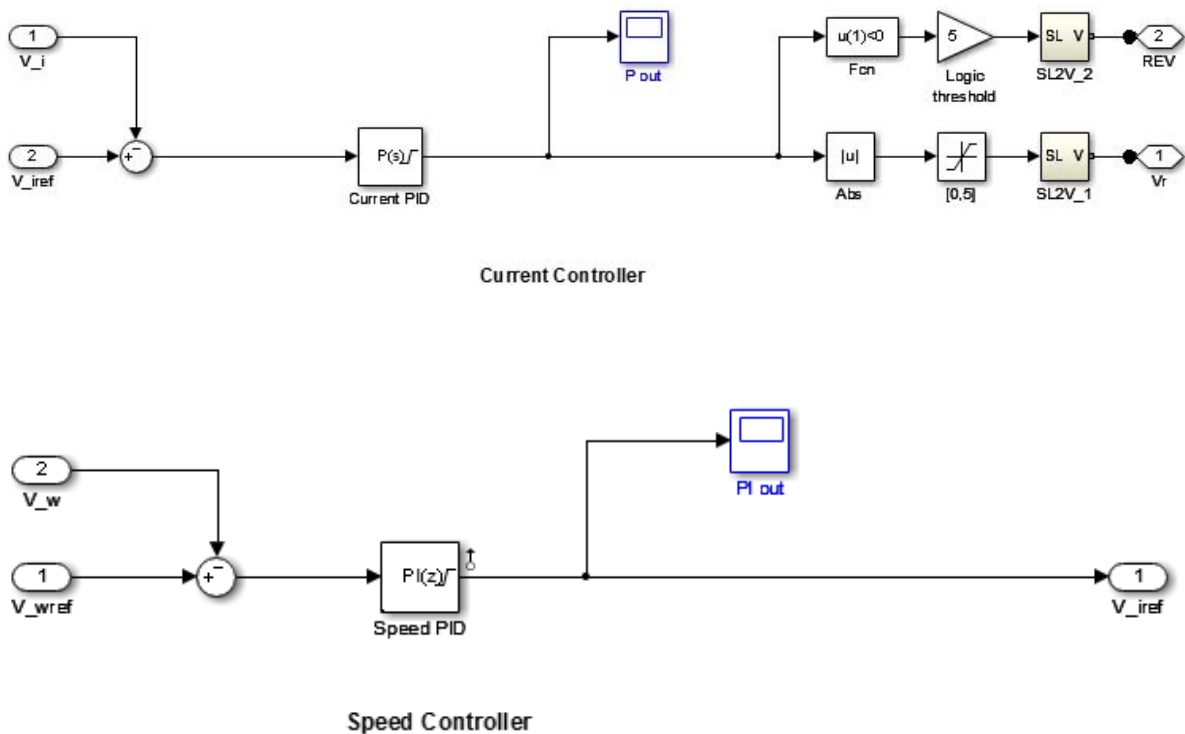
### Linear Electric Actuator Model

Open the Simulink model of the linear electric actuator:

```
open_system('rct_linact')
```



The electrical and mechanical components are modeled using SimElectronics and SimMechanics. The control system consists of two cascaded feedback loops controlling the driving current and angular speed of the DC motor.



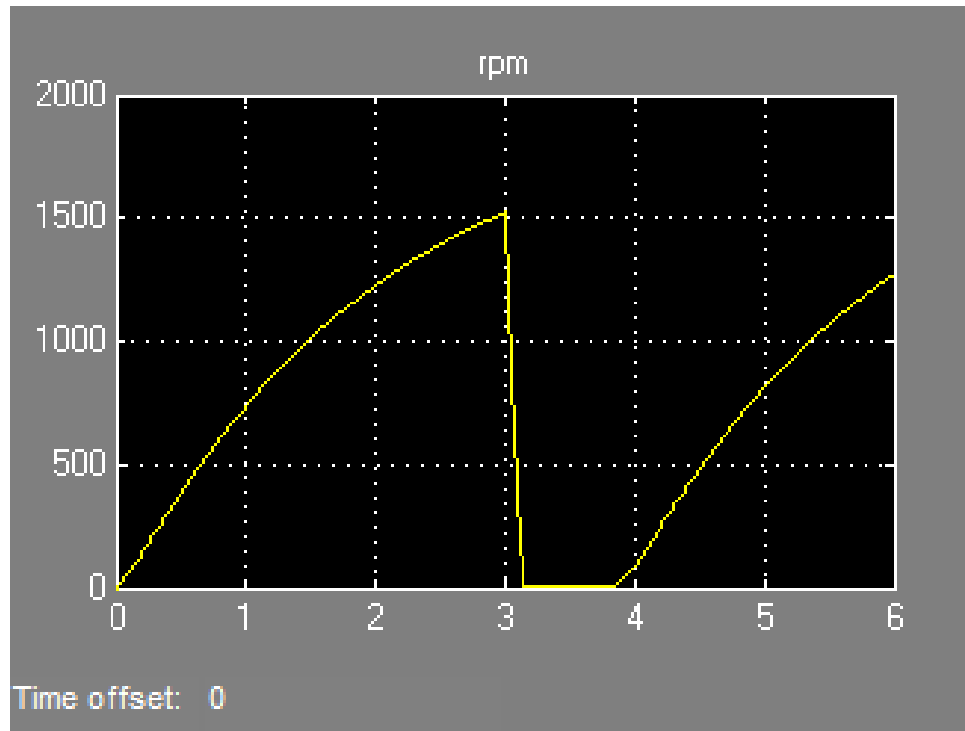
**Figure 1: Current and Speed Controllers.**

Note that the inner-loop (current) controller is a proportional gain while the outer-loop (speed) controller has proportional and integral actions. The output of both controllers is limited to plus/minus 5.

**Design Specifications**

We need to tune the proportional and integral gains to respond to a 2000 rpm speed demand in about 0.1 seconds with minimum overshoot. The initial gain settings in the model are  $P=50$  and  $PI(s)=0.2+0.1/s$  and the corresponding response is shown in Figure 2. This response is too slow and too sensitive to load disturbances.





**Figure 2: Untuned Response.**

### Control System Tuning

You can use `systemtune` to jointly tune both feedback loops. To set up the design, create an instance of the `sITunable` interface with the list of tuned blocks. All blocks and signals are specified by their names in the model. The model is linearized at  $t=0.5$  to avoid discontinuities in some derivatives at  $t=0$ .

```
TunedBlocks = {'Current PID','Speed PID'};
op = findop('rct_linact',0.5); % linearize at t=0.5

% Create tuning interface
STO = sITunable('rct_linact',TunedBlocks,op);
```

The data structure `ST0` contains a description of the control system and its tunable elements. Next specify that the DC motor should follow a 2000 rpm speed demand in 0.1 seconds:

```
TR = TuningGoal.Tracking('Speed Demand (rpm)', 'rpm', 0.1);
```

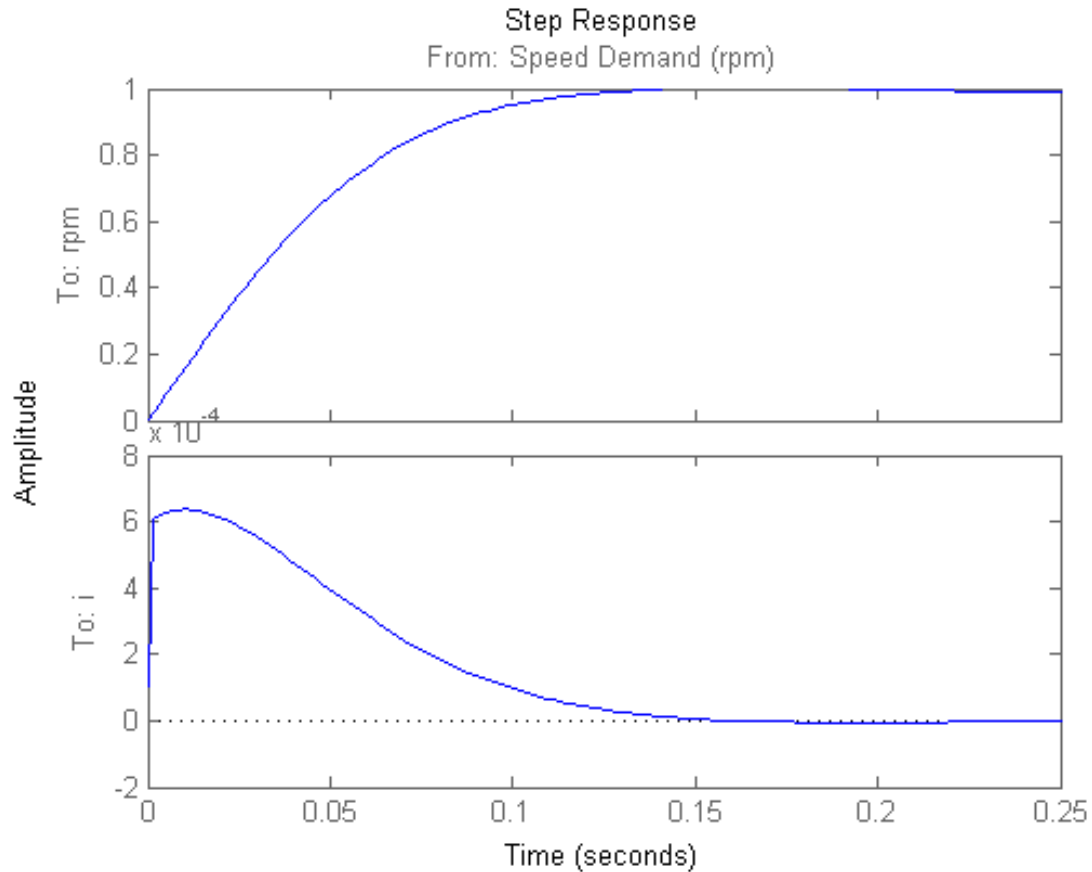
You can now tune the proportional and integral gains with `looptune`:

```
ST1 = systune(ST0, TR);
```

```
Final: Soft = 1.12, Hard = -Inf, Iterations = 34
```

This returns an updated description `ST1` containing the tuned gain values. To validate this design, plot the closed-loop response from speed demand to speed:

```
T1 = ST1.getIOTransfer('Speed Demand (rpm)', {'rpm', 'i'});  
step(T1)
```



The response looks good in the linear domain so push the tuned gain values to Simulink and further validate the design in the nonlinear model.

```
ST1.writeBlockValue()
```

The nonlinear simulation results appear in Figure 3. The nonlinear behavior is far worse than the linear approximation, a discrepancy that can be traced to saturations in the inner loop (see Figure 4).

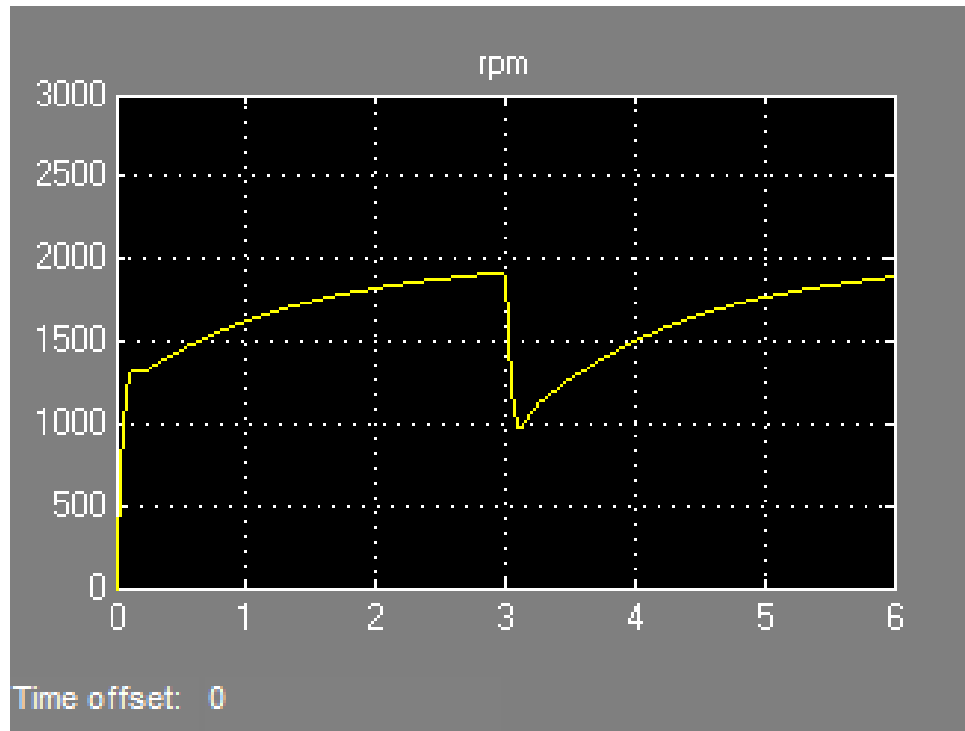
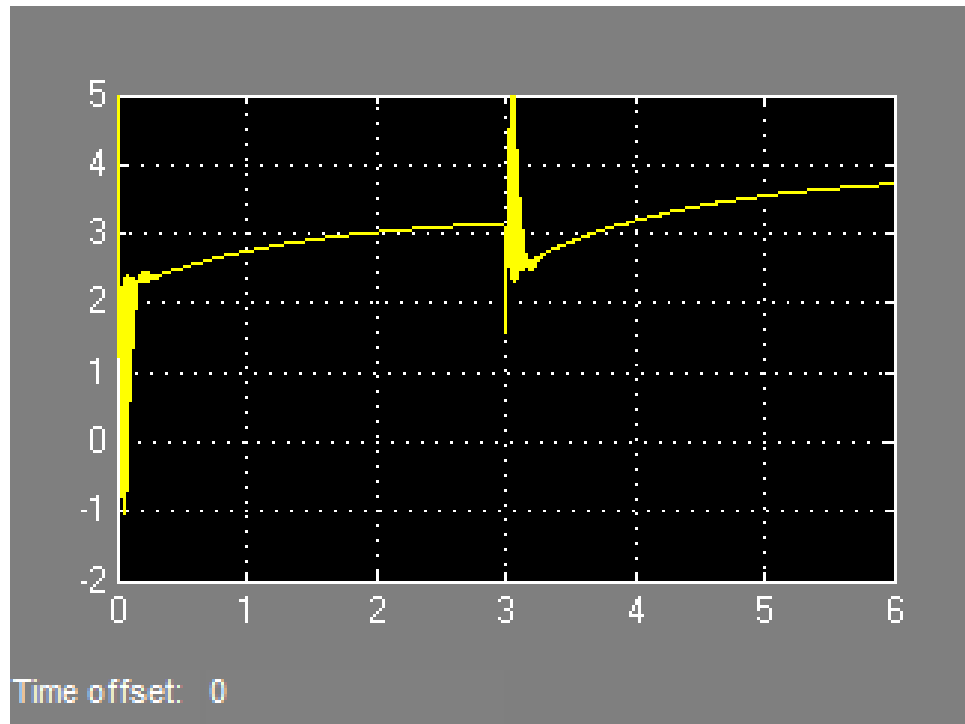


Figure 3: Nonlinear Simulation of Tuned Controller.



**Figure 4: Current Controller Output (limited to plus/minus 5).**

```
% Close model to discard this first set of tuned gains
close_system('rct_linact',0)
```

### Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves `system` free to allocate the control effort between the inner and outer loops. Saturations in the inner loop suggest that the proportional gain is too high and that some rebalancing is needed. One possible remedy is to explicitly limit the gain from the speed command to the outputs of the P and PI controllers. For a speed reference of 2000 rpm and saturation limits of plus/minus 5, the average gain should not exceed  $5/2000 = 0.0025$ . To be conservative, we can try to keep the gain from speed reference to controller

outputs below 0.001. To do this, add two gain requirements and retune the controller gains with all three requirements in place.

```
% Mark the outputs of the "Current PID" and "Speed PID" blocks as control
% signals so that they can be referenced in the gain requirements
ST0 = slTunable('rct_linact',TunedBlocks,op);
ST0.addControl({'Current PID','Speed PID'})
```

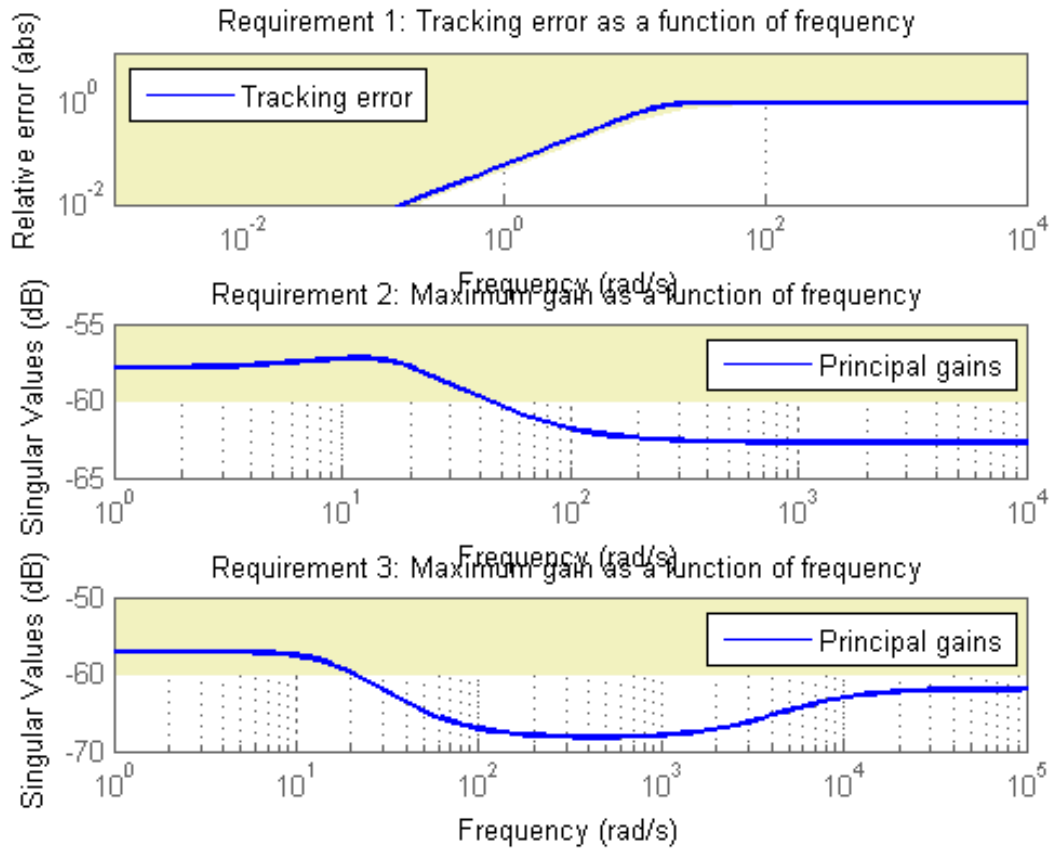
```
% Limit gain from speed demand to control signals to avoid saturation
MG1 = TuningGoal.Gain('Speed Demand (rpm)','Speed PID',0.001);
MG2 = TuningGoal.Gain('Speed Demand (rpm)','Current PID',0.001);
```

```
% Retune with these additional requirements
[ST2,~,~,info] = systune(ST0,[TR,MG1,MG2]);
```

```
Final: Soft = 1.39, Hard = -Inf, Iterations = 42
```

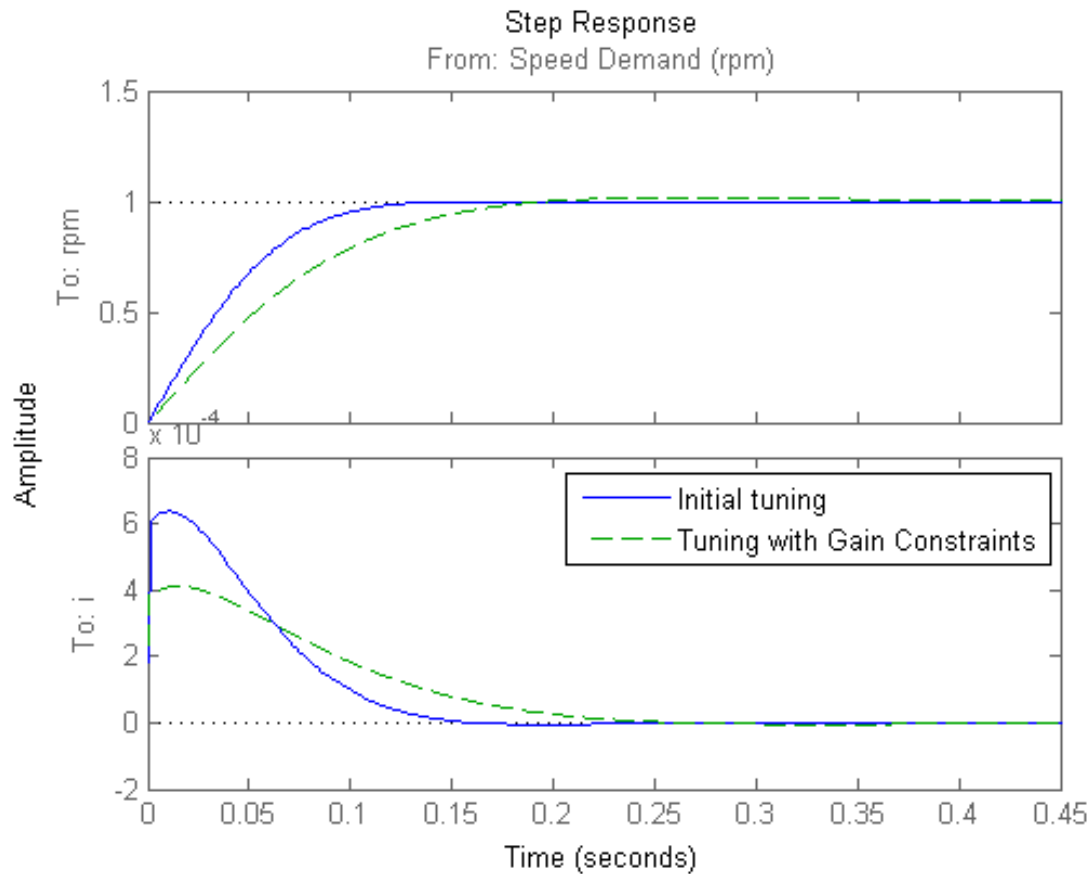
The final gain 1.39 indicates that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1). Use `viewSpec` to inspect how the tuned controllers fare against each requirement.

```
viewSpec([TR,MG1,MG2],ST2,info)
```



Next compare the two designs in the linear domain.

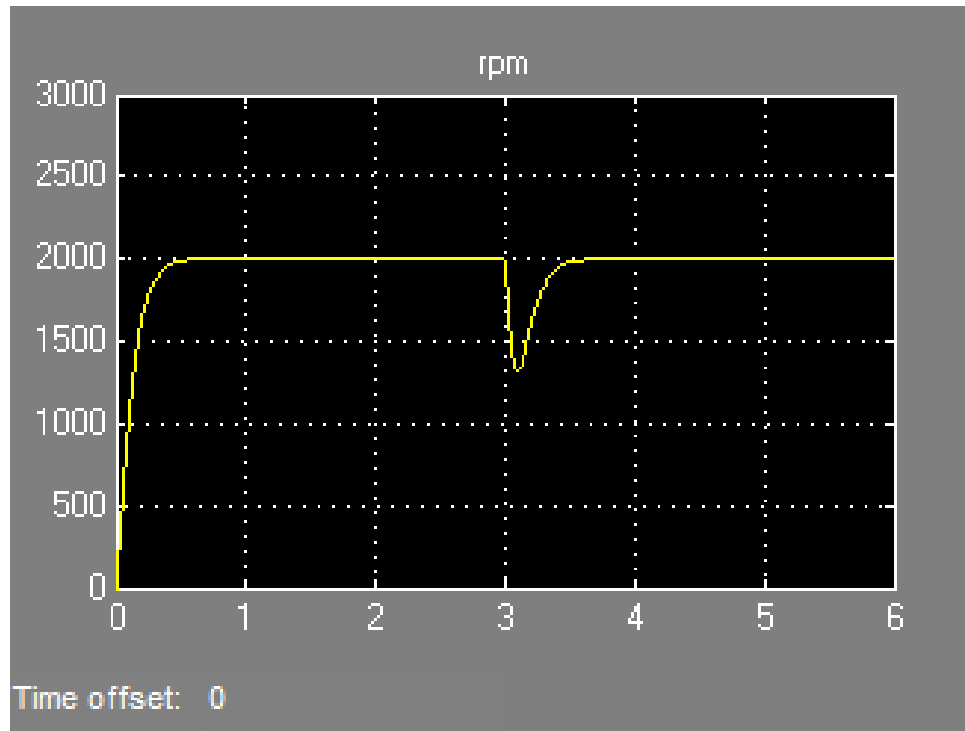
```
T2 = ST2.getIOTransfer('Speed Demand (rpm)', {'rpm', 'i'});
clf, step(T1, 'b', T2, 'g--')
legend('Initial tuning', 'Tuning with Gain Constraints')
```



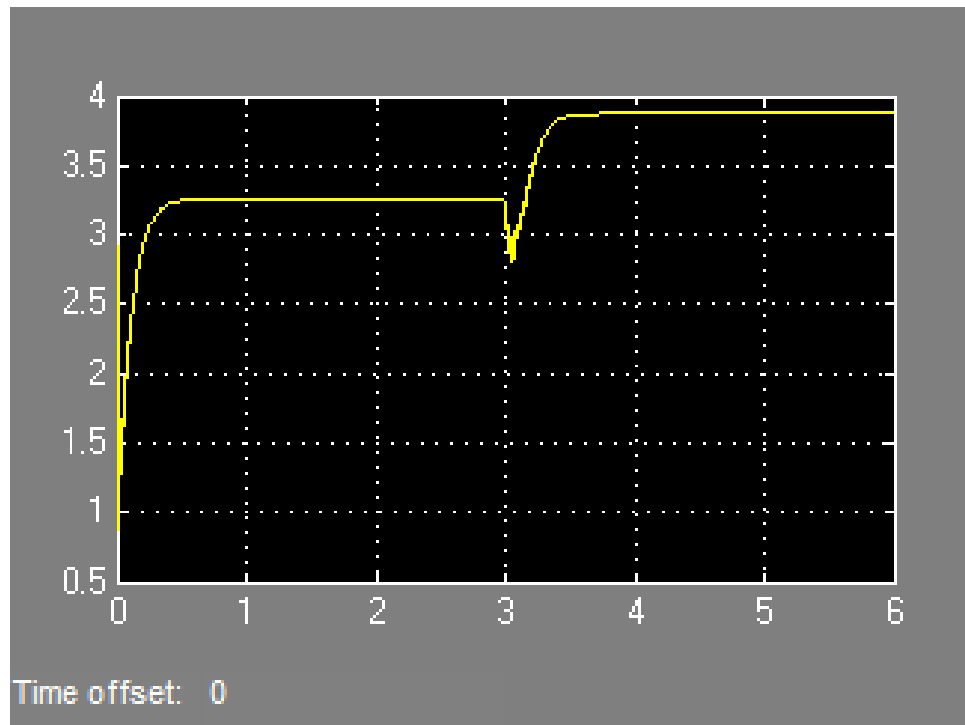
The second design is less aggressive but still meets the response time requirement. Finally, push the new tuned gain values to the Simulink model and simulate the response to a 2000 rpm speed demand and 500 N load disturbance. The simulation results appear in Figure 5 and the current controller output is shown in Figure 6.

```
ST2.writeBlockValue()
```





**Figure 5: Nonlinear Response of Tuning with Gain Constraints.**



**Figure 6: Current Controller Output.**

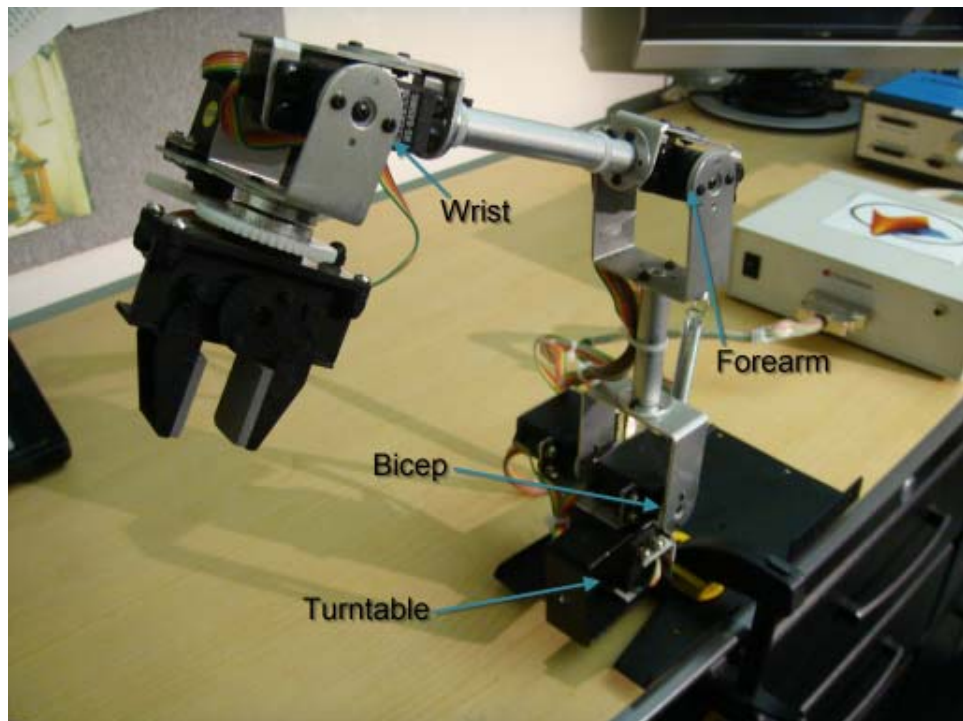
The nonlinear responses are now satisfactory and the current loop is no longer saturating. The additional gain constraints have forced systune to re-distribute the control effort between the inner and outer loops so as to avoid saturation.

## Multi-Loop PID Control of a Robot Arm

This example shows how to use looptune to tune a multi-loop controller for a 4-DOF robotic arm manipulator.

### Robotic Arm Model and Controller

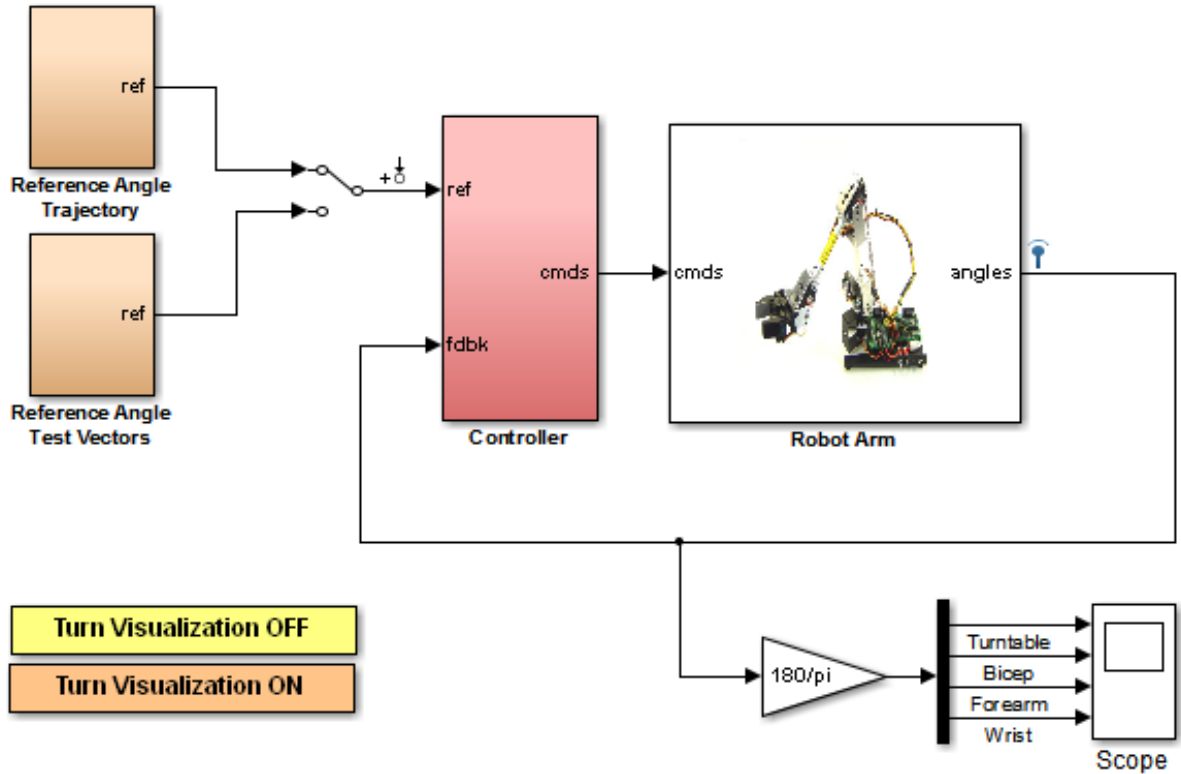
This example uses the four degree-of-freedom robotic arm shown below. This arm consists of four joints labeled from base to tip: "Turntable", "Bicep", "Forearm", and "Wrist". Each joint is actuated by a DC motor except for the Bicep joint which uses two DC motors in tandem.



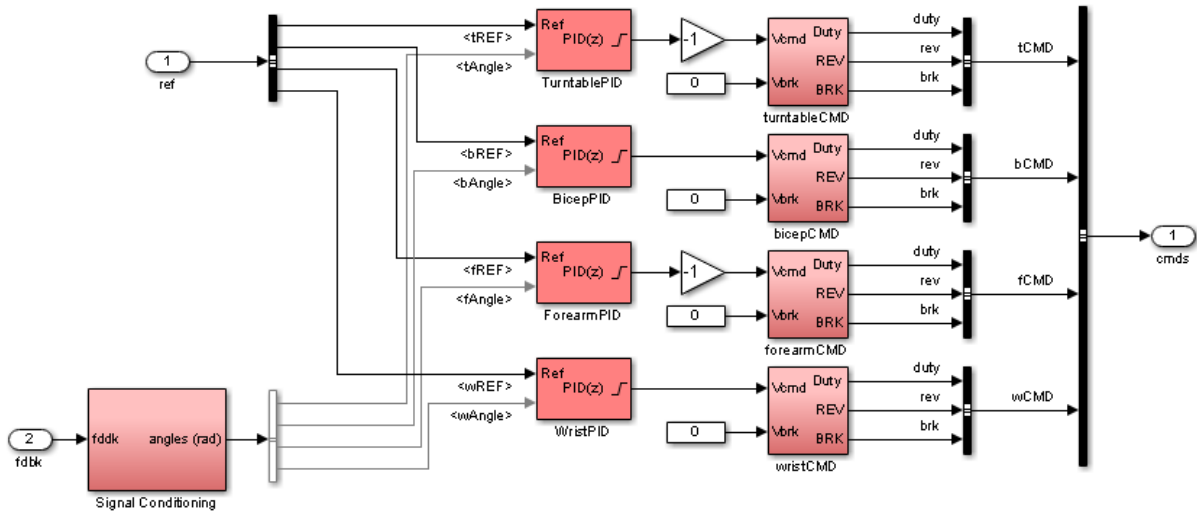
**Figure 1: Robotic arm manipulator.**

Open the Simulink model of the robot arm.

open\_system('rct\_robotarm')



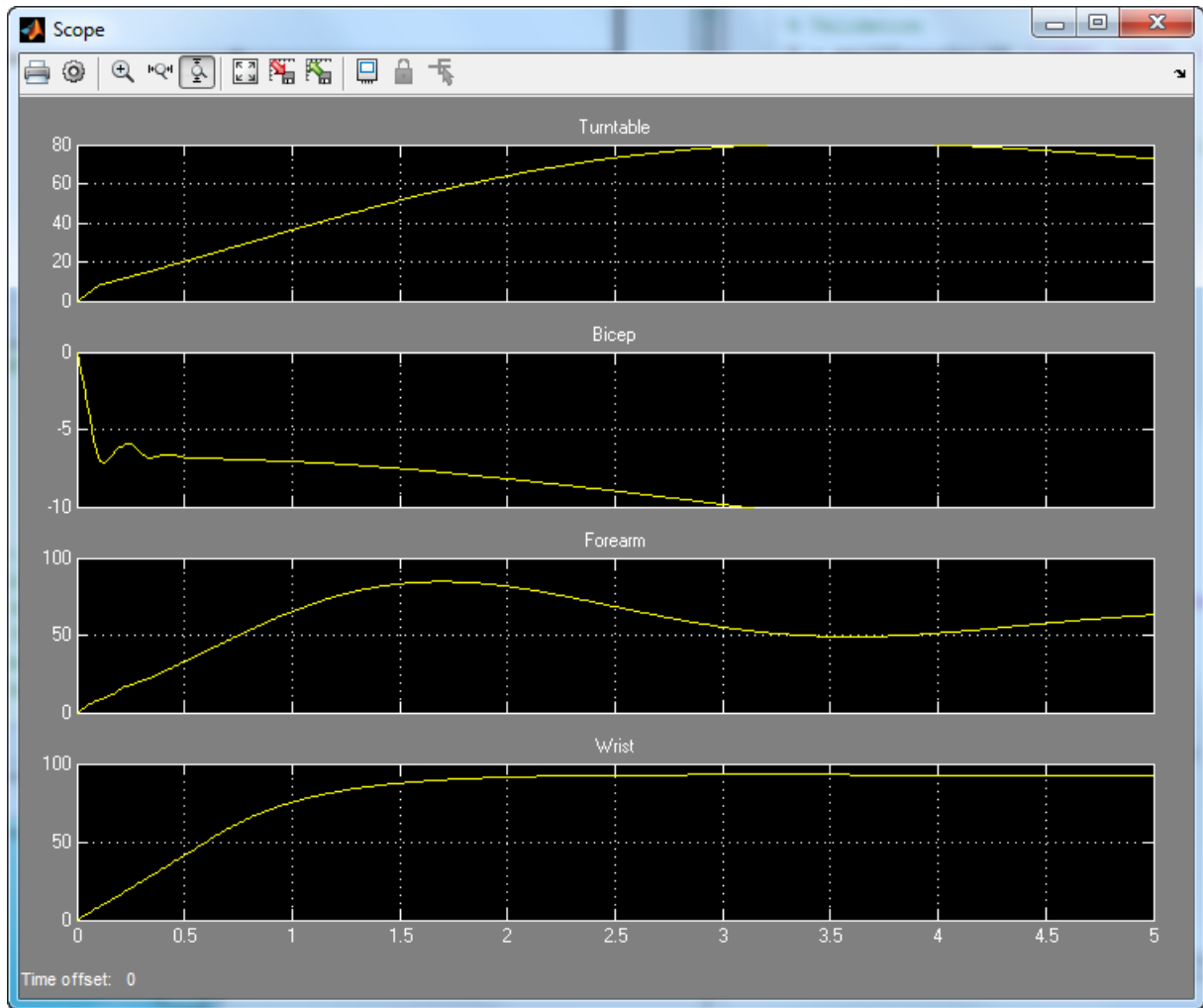
The controller consists of four PID controllers (one per joint). Each PID controller is implemented using the "2-DOF PID Controller" block from the Simulink library (see *PID Tuning for Setpoint Tracking vs. Disturbance Rejection* example for motivation).



**Figure 2: Controller structure.**

Typically, such multi-loop controllers are tuned sequentially by tuning one PID loop at a time and cycling through the loops until the overall behavior is satisfactory. This process can be time consuming and is not guaranteed to converge to the best overall tuning. Alternatively, you can use `systune` or `looptune` to jointly tune all four PID loops subject to system-level requirements such as response time and minimum cross-coupling.

In this example, the arm must move to a particular configuration in about 1 second with smooth angular motion at each joint. The arm starts in a fully extended vertical position with all joint angles at zero. The end configuration is specified by the angular positions: Turntable = 60 deg, Bicep = -10 deg, Forearm = 60 deg, Wrist = 90 deg. The angular trajectories for the original PID settings are shown below. Clearly the response is too sluggish and the forearm is wobbling.



**Figure 3: Untuned angular response.**

### **Linearizing the Plant**

The robot arm dynamics are nonlinear. To understand whether the arm can be controlled with one set of PID gains, linearize the plant at various

points (snapshot times) along the trajectory of interest. Here "plant" refers to the dynamics between the control signals (outputs of PID blocks) and the measurement signals (output of "Robot Arm" block).

```
SnapshotTimes = 0:1:5;
% Plant is from PID outputs to Robot Arm outputs
LinIOs = [...
    linio('rct_robotarm/Controller/TurntablePID',1,'openinput'),...
    linio('rct_robotarm/Controller/BicepPID',1,'openinput'),...
    linio('rct_robotarm/Controller/ForearmPID',1,'openinput'),...
    linio('rct_robotarm/Controller/WristPID',1,'openinput'),...
    linio('rct_robotarm/Robot Arm',1,'output')];
LinOpt = linearizeOptions('SampleTime',0); % seek continuous-time model
G = linearize('rct_robotarm',LinIOs,SnapshotTimes,LinOpt);
```

```
size(G)
```

```
6x1 array of state-space models.
Each model has 4 outputs, 4 inputs, and between 0 and 13 states.
```

The robot arm model linearizes to zero at  $t=0$  due to the Bicep and Forearm joints hitting their mechanical limits:

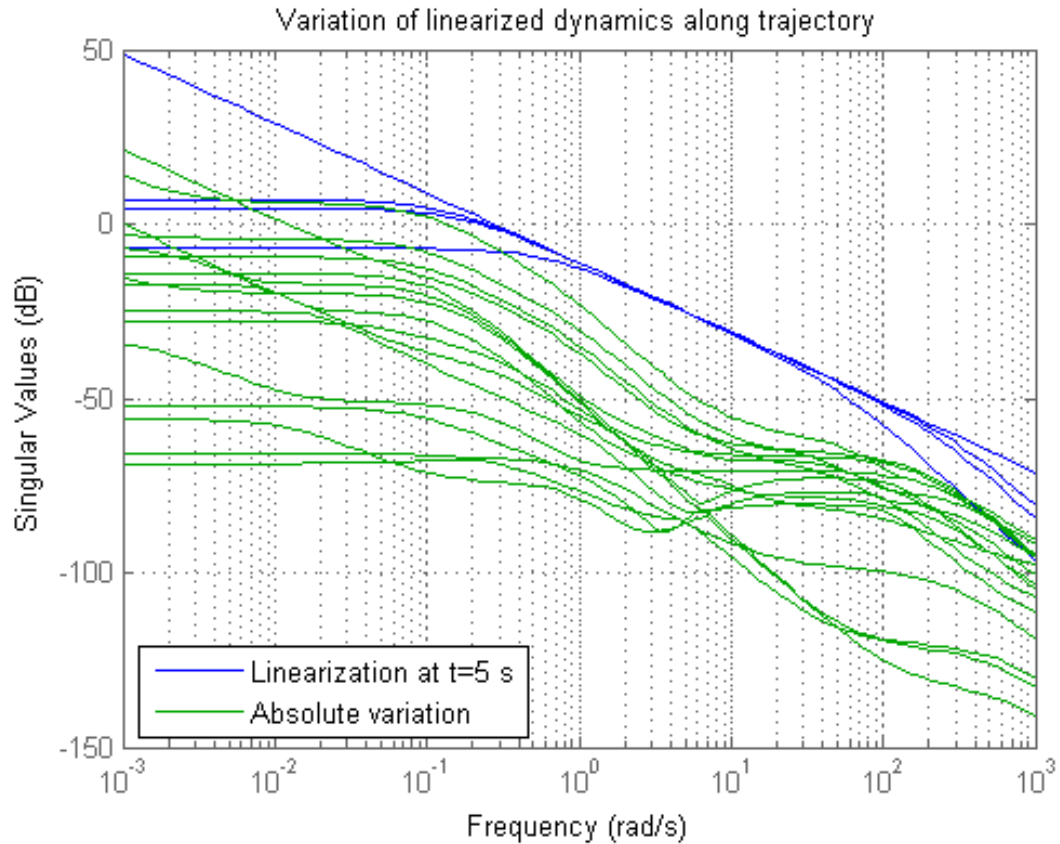
```
getPeakGain(G(:, :, 1))
```

```
ans =
```

```
0
```

Plot the gap between the linearized models at  $t=1,2,3,4$  seconds and the final model at  $t=5$  seconds.

```
G5 = G(:, :, end); % t=5
G5.SamplingGrid = [];
sigma(G5,G(:, :, 2:5)-G5,{1e-3,1e3}), grid
title('Variation of linearized dynamics along trajectory')
legend('Linearization at t=5 s','Absolute variation',...
    'location','SouthWest')
```



While the dynamics vary significantly at low and high frequency, the variation drops to less than 10% near 10 rad/s, which is roughly the desired control bandwidth. Small plant variations near the target gain crossover frequency suggest that we can control the arm with a single set of PID gains and need not resort to gain scheduling.

#### **Tuning the PID Controllers with LOOPTUNE**



With `looptune`, you can directly tune all four PID loops to achieve the desired response time with minimal loop interaction and adequate MIMO stability margins. The controller is tuned in continuous time and automatically discretized when writing the PID gains back to Simulink. Use the `sITunable` interface to specify which blocks must be tuned and where the plant/controller boundary is.

```
% Use linearization at t=3s for plant model
op = findop('rct_robotarm',3);
TunedBlocks = {'TurntablePID','BicepPID','ForearmPID','WristPID'};
ST0 = sITunable('rct_robotarm',TunedBlocks,op);

% Control signals are the PID outputs
ST0.addControl(TunedBlocks);
% Measurement signals at the joint angles
ST0.addMeasurement('Robot Arm');
```

In its simplest use, `looptune` only needs to know the target control bandwidth, which should be at least twice the reciprocal of the desired response time. Here the desired response time is 1 second so try a target bandwidth of 5 rad/s (bearing in mind that the plant dynamics vary least near 10 rad/s).

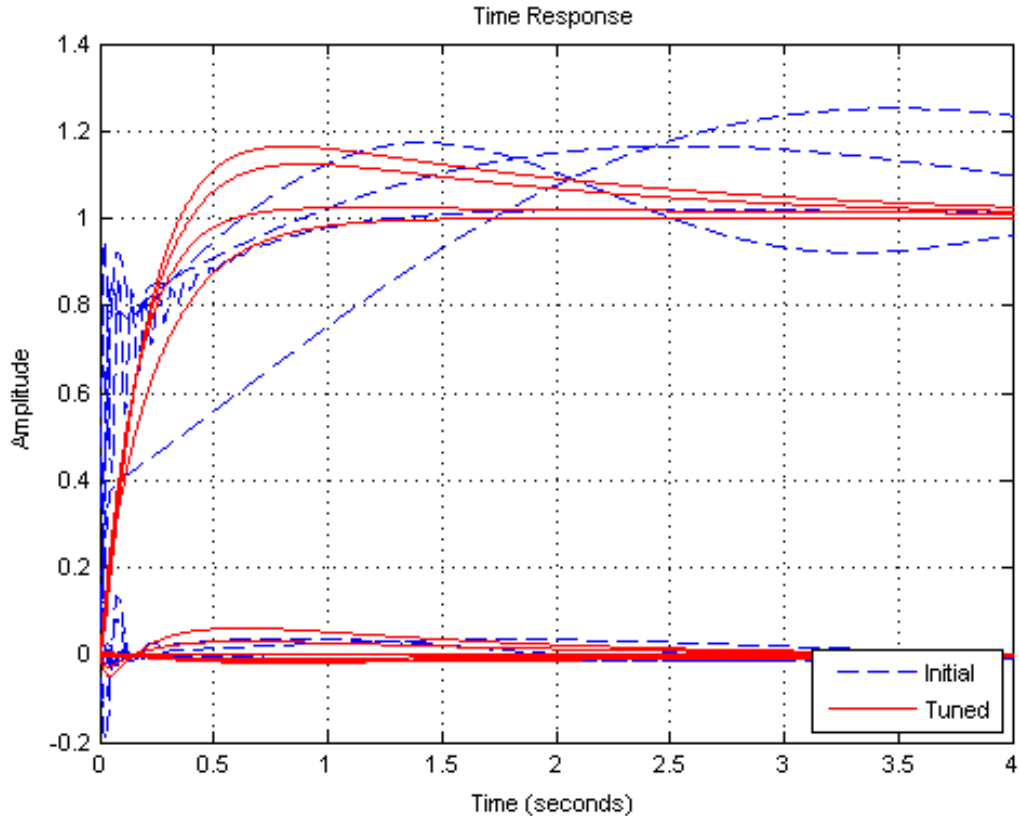
```
wc = 5; % target gain crossover frequency
ST1 = looptune(ST0,wc);
```

```
Final: Peak gain = 1, Iterations = 54
      Min decay rate 1e-07 is close to lower bound 1e-07
```

A final value near or below 1 indicates that `looptune` achieved the requested bandwidth. Compare the responses to a step command in angular position for the initial and tuned controllers.

```
RefSignals = {'tREF','bREF','fREF','wREF'};
T0 = getIOTransfer(ST0,RefSignals,'Robot Arm');
T1 = getIOTransfer(ST1,RefSignals,'Robot Arm');

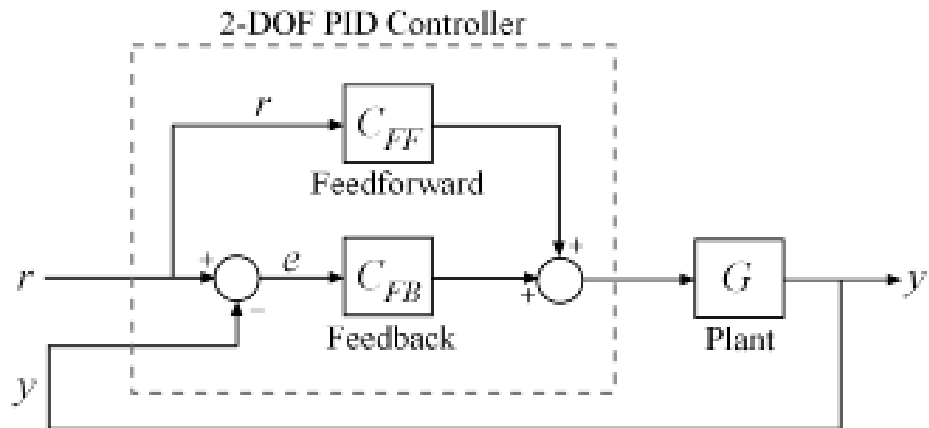
opt = timeoptions; opt.IOGrouping = 'all'; opt.Grid = 'on';
stepplot(T0,'b--',T1,'r',4,opt)
legend('Initial','Tuned','location','SouthEast')
```



The four curves settling near  $y=1$  represent the step responses of each joint, and the curves settling near  $y=0$  represent the cross-coupling terms. The tuned controller is a clear improvement but should ideally settle faster with less overshoot.

### Exploiting the Second Degree of Freedom

The 2-DOF PID controllers have a feedforward and a feedback component.



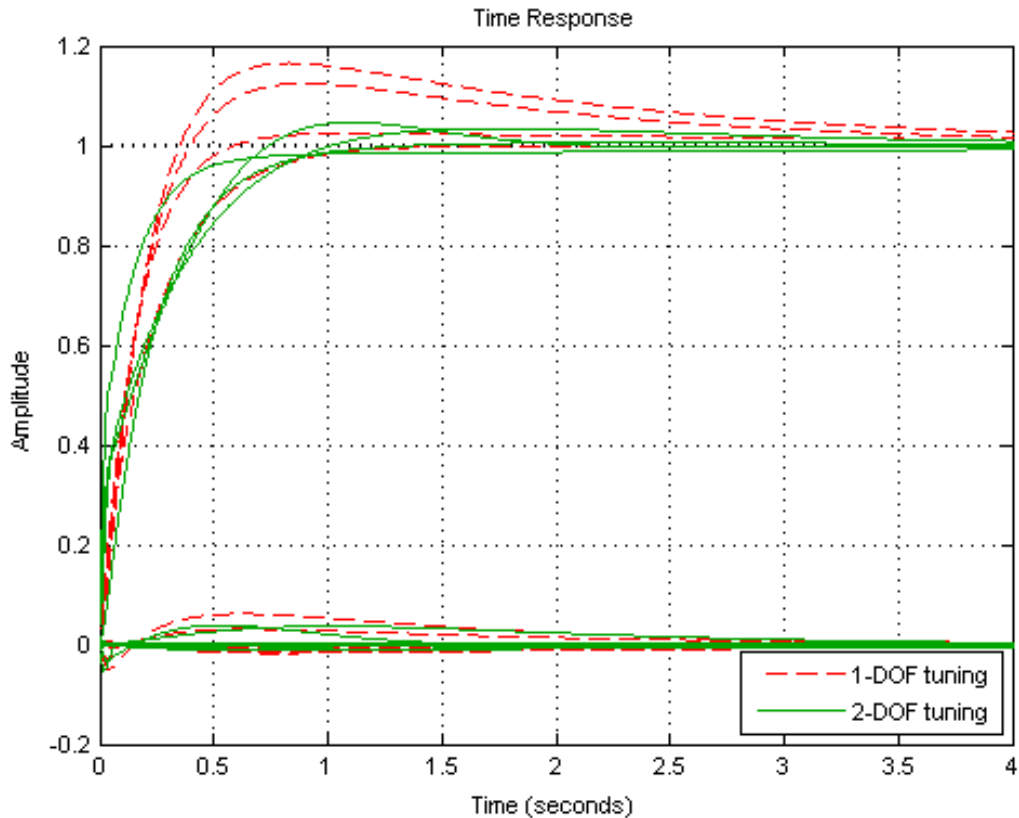
**Figure 4: Two degree-of-freedom PID controllers.**

By default, `looptune` only tunes the feedback loop and does not "see" the feedforward component. This can be confirmed by verifying that the  $b$  and  $c$  parameters of the PID controllers remain set to their initial value  $b = c = 1$  (use `showTunable` for this purpose). To take advantage of the feedforward action and reduce overshoot, replace the bandwidth target by an explicit tracking requirement from reference angles to joint angles.

```
TR = TuningGoal.Tracking(RefSignals,'Robot Arm',0.5);
ST2 = looptune(ST0,TR);
```

Final: Peak gain = 1.06, Iterations = 72

```
T2 = getIOTransfer(ST2,RefSignals,'Robot Arm');
stepplot(T1,'r--',T2,'g',4,opt)
legend('1-DOF tuning','2-DOF tuning','location','SouthEast')
```



The 2-DOF tuning reduces overshoot and takes advantage of the  $b$  and  $c$  parameters as confirmed by inspecting the tuned PID gains:

```
showTunable(ST2)
```

```
Block "rct_robotarm/Controller/TurntablePID" =
```

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{s} (c*r-y)$$

$$s \quad Tf*s+1$$

with Kp = 12.427, Ki = 7.3436, Kd = 0.65591, Tf = 0.024911, b = 0.90521,

Continuous-time 2-DOF PID controller.

-----

Block "rct\_robotarm/Controller/BicepPID" =

$$u = Kp (b*r-y) + Ki \frac{1}{s} (r-y) + Kd \frac{s}{Tf*s+1} (c*r-y)$$

with Kp = 12.3814, Ki = 7.1887, Kd = 1.5736, Tf = 0.48625, b = 0.7019, c

Continuous-time 2-DOF PID controller.

-----

Block "rct\_robotarm/Controller/ForearmPID" =

$$u = Kp (b*r-y) + Ki \frac{1}{s} (r-y) + Kd \frac{s}{Tf*s+1} (c*r-y)$$

with Kp = 20.2349, Ki = 40.4771, Kd = 1.0118, Tf = 0.013114, b = 0.57636,

Continuous-time 2-DOF PID controller.

-----

Block "rct\_robotarm/Controller/WristPID" =

$$u = Kp (b*r-y) + Ki \frac{1}{s} (r-y) + Kd \frac{s}{Tf*s+1} (c*r-y)$$

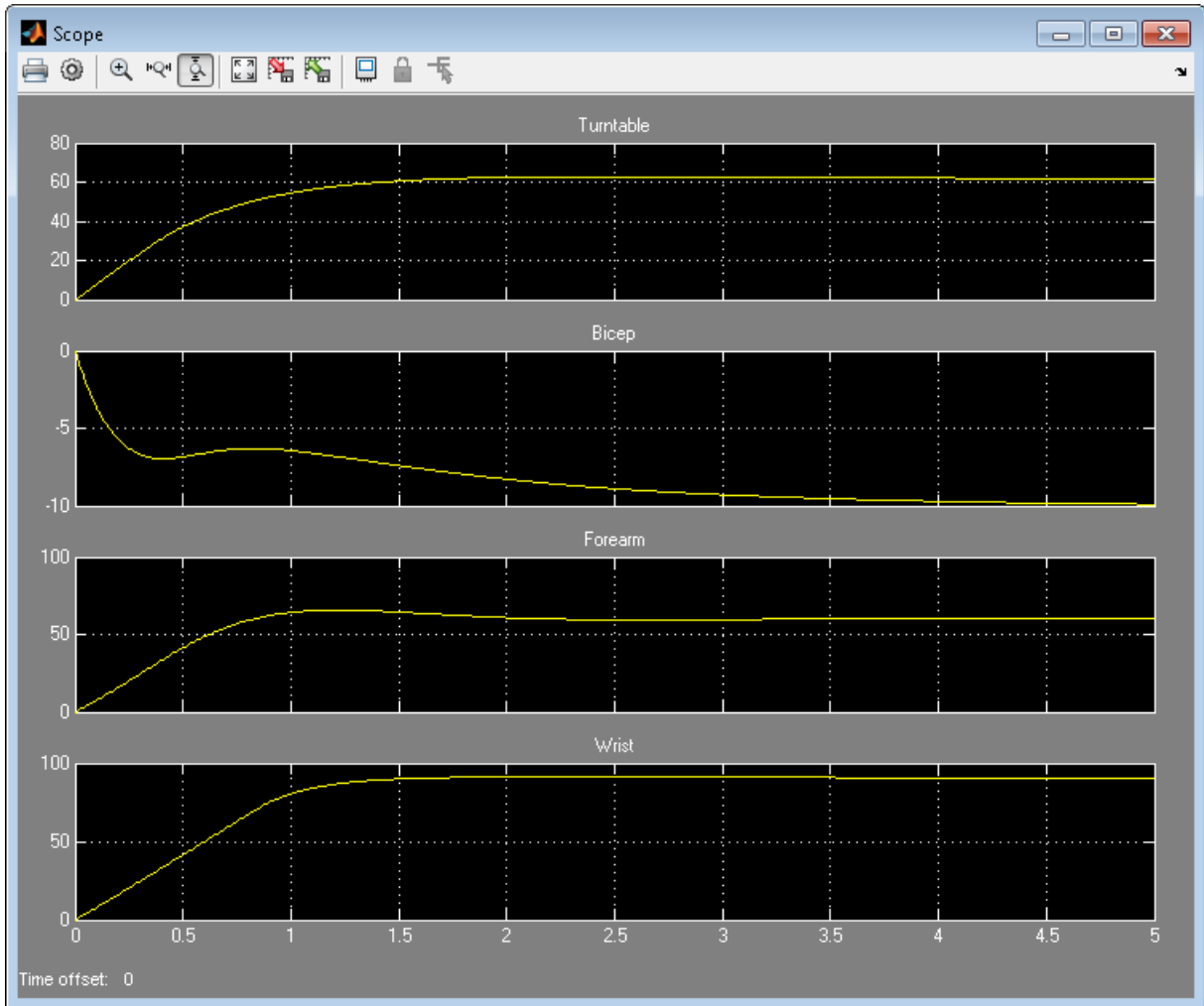
with Kp = 28.8414, Ki = 6.4949, Kd = 0.83132, Tf = 0.013636, b = 0.9661,

Continuous-time 2-DOF PID controller.

### **Validating the Tuned Controller**

The tuned linear responses look satisfactory so write the tuned values of the PID gains back to the Simulink blocks and simulate the overall maneuver. The simulation results appear in Figure 5.

```
writeBlockValue(ST2)
```

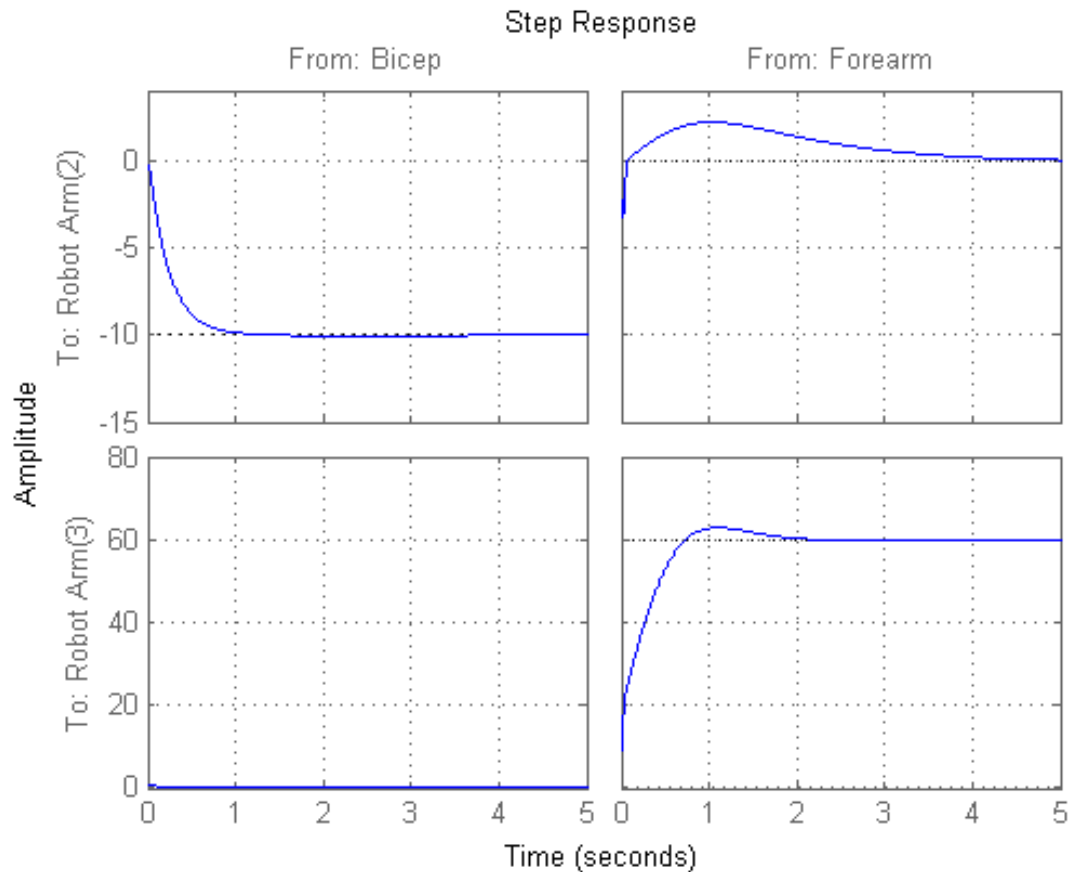


**Figure 5: Tuned angular response.**

The responses look good except for the Bicep joint whose response is somewhat sluggish and jerky. It is tempting to blame this discrepancy on nonlinear effects, but this turns out to be just cross-coupling effects between the Forearm and Bicep joints. To see this, plot the step response of these two

joints for the **actual** step changes occurring during the maneuver (-10 deg for the Bicep joint and 60 deg for the Forearm joint).

```
H2 = T2(2:3,2:3) * diag([-10 60]); % scale by step amplitude
H2.u = {'Bicep', 'Forearm'};
step(H2,5), grid
```



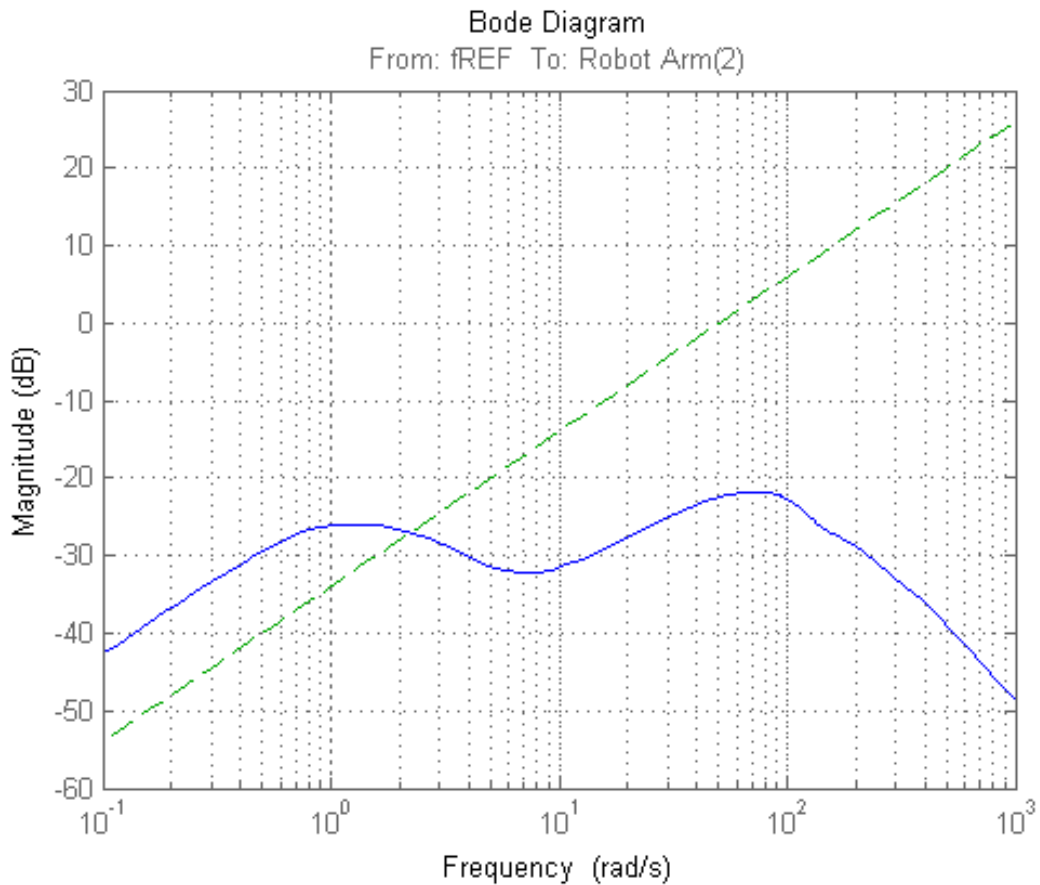
When brought to scale, the first row of plots show that a 60-degree step change in Forearm position has a sizeable and lasting impact on the Bicep position. This explains the sluggish Bicep response observed when simultaneously moving all four joints.



### Eliminating Undesirable Cross-Couplings

Seen from the Bicep joint, a step change in Forearm position acts as an external disturbance. Better rejection of this disturbance is the key to improving the Bicep response. Plot the magnitude of the closed-loop transfer function from `fREF` (Forearm reference angle) to the Bicep angle.

```
s = tf('s');  
bodemag(T2('Robot Arm(2)', 'fREF'), 0.02*s, 'g--', {1e-1, 1e3}), grid
```



Notice the hump near 1 rad/s. This is responsible for the lasting (low frequency) offset in the Bicep position. To mitigate this, you need to lower the gain below 1 rad/s and ideally keep it below the dashed green line. Add a requirement to enforce this gain limit and keep overshoot in check by also limiting the peak gain from reference to actual angles.

```
% Reduce cross-coupling forearm -> bicep
MG = TuningGoal.Gain('fREF','Robot Arm(2)',0.02*s);
MG.Focus = [0 1];
```

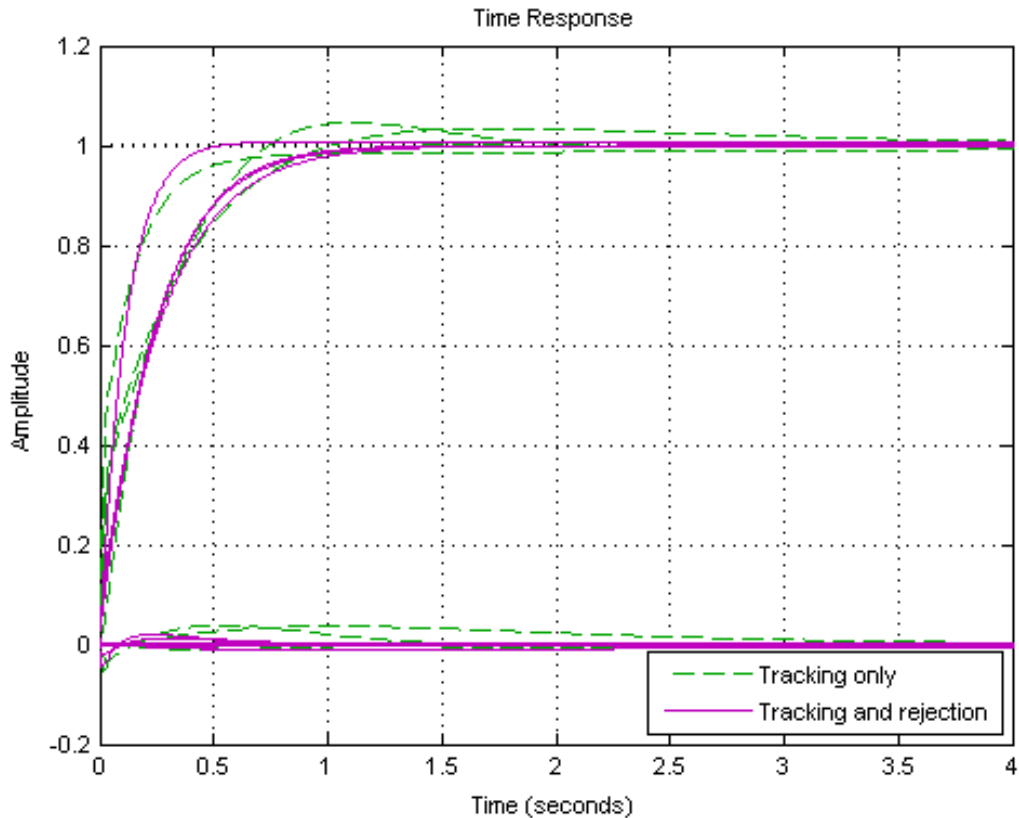
```
% Limit overshoot to 5%
OS = TuningGoal.Overshoot(RefSignals,'Robot Arm',5);
```

```
% Retune controller with all three requirements in force
ST3 = looptune(ST0,TR,MG,OS);
```

```
Final: Peak gain = 1.01, Iterations = 115
```

Compare results with the previous design.

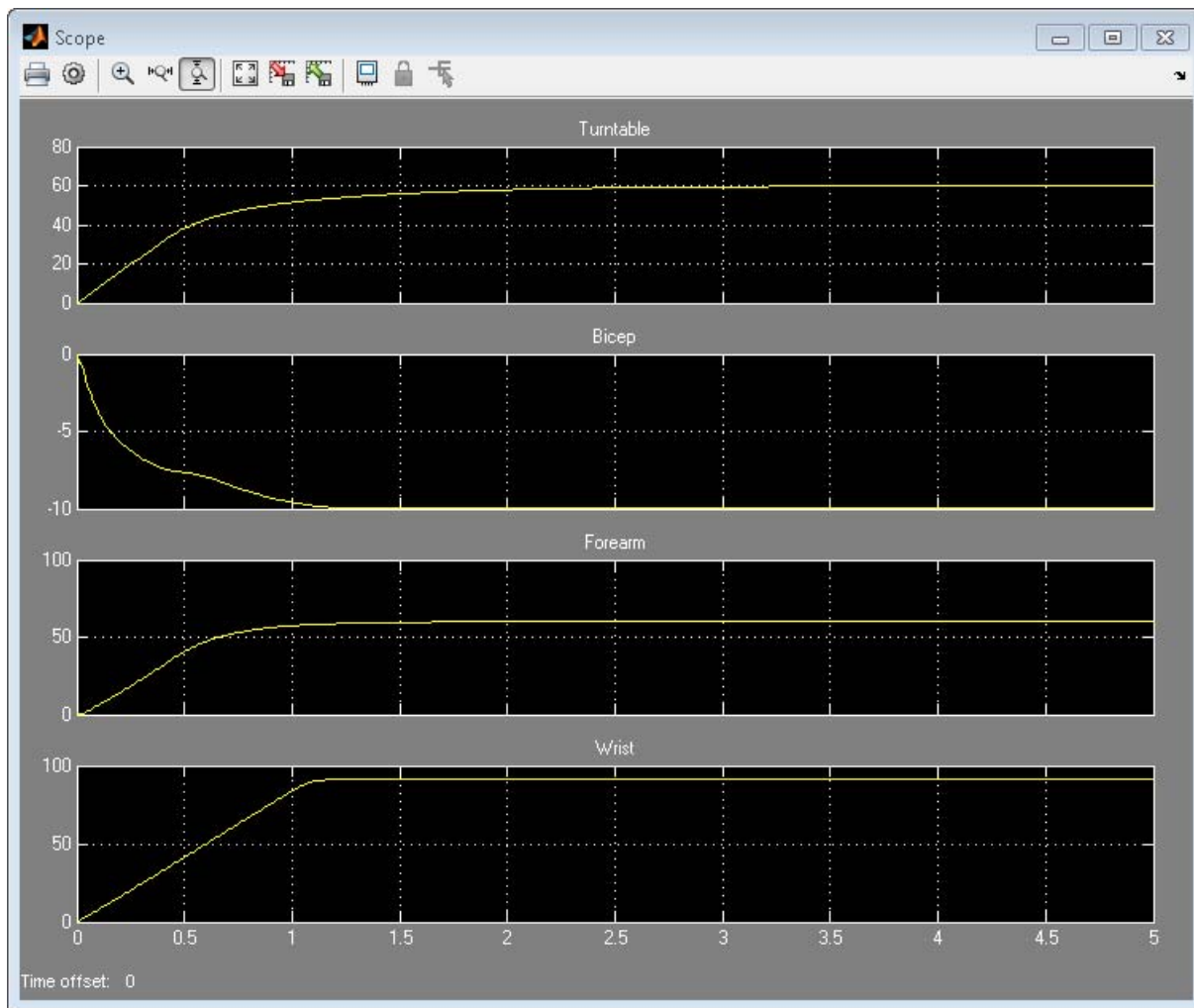
```
T3 = getIOTransfer(ST3,RefSignals,'Robot Arm');
stepplot(T2,'g--',T3,'m',4,opt)
legend('Tracking only','Tracking and rejection','location','SouthEast')
```



The cross-coupling terms settle faster now. Push the retuned values to Simulink for further validation.

```
writeBlockValue(ST3)
```

The simulation results appear in Figure 6. The Bicep response is now on par with the other joints in terms of settling time and smoothness.



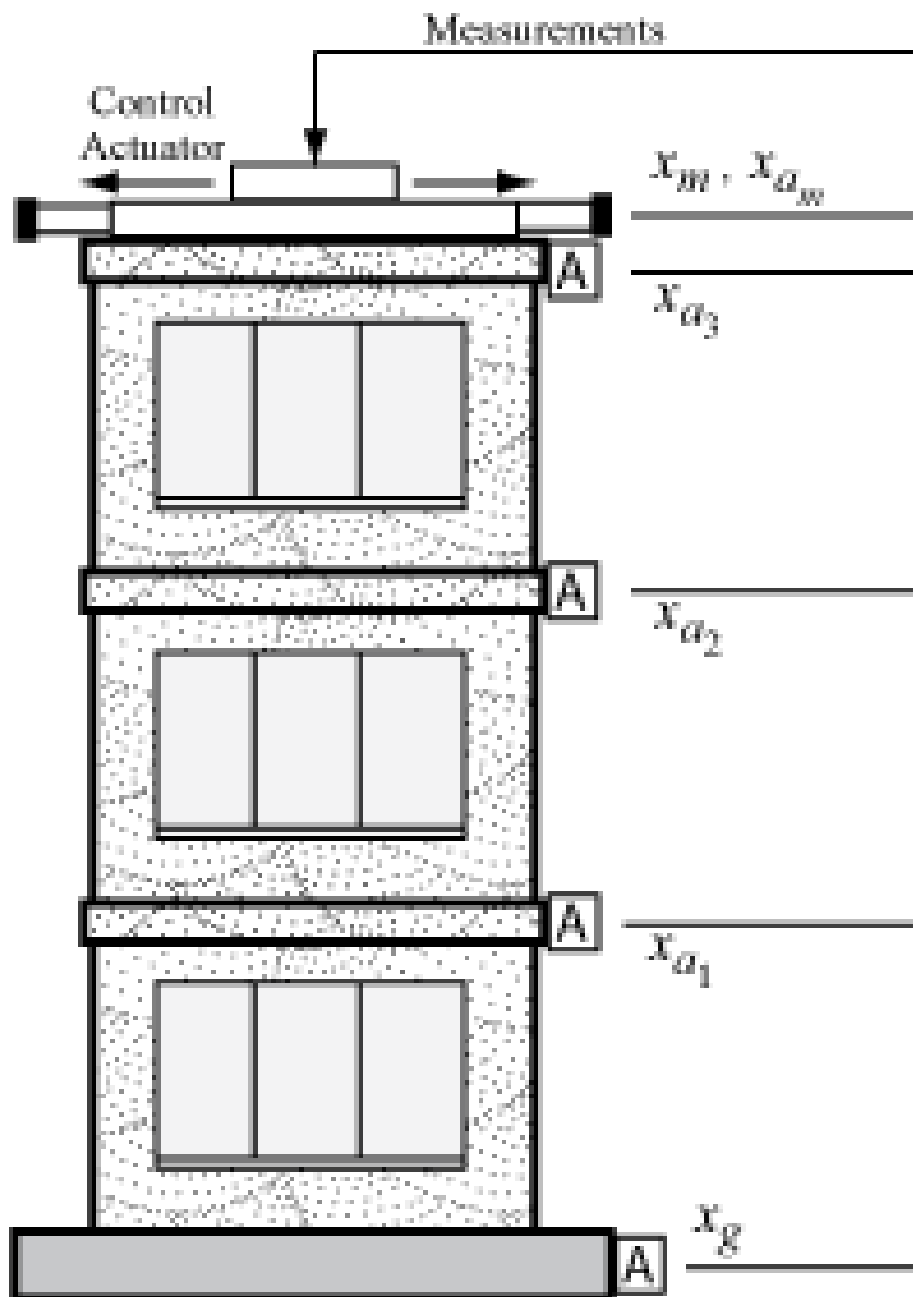
**Figure 6: Tuned angular response with reduced Bicep-Forearm coupling.**

## Active Vibration Control in Three-Story Building

This example uses systune to control seismic vibrations in a three-story building.

### Background

This example considers an Active Mass Driver (AMD) control system for vibration isolation in a three-story experimental structure. This setup is used to assess control design techniques for increasing safety of civil engineering structures during earthquakes. The structure consists of three stories with an active mass driver on the top floor which is used to attenuate ground disturbances. This application is borrowed from "*Benchmark Problems in Structural Control: Part I - Active Mass Driver System*," B.F. Spencer Jr., S.J. Dyke, and H.S. Deoskar, *Earthquake Engineering and Structural Dynamics*, 27(11), 1998, pp. 1127-1139.



### Figure 1: Active Mass Driver Control System

The plant P is a 28-state model with the following state variables:

- $x(i)$ : displacement of  $i$ -th floor relative to the ground (cm)
- $x_m$ : displacement of AMD relative to 3rd floor (cm)
- $x_v(i)$ : velocity of  $i$ -th floor relative to the ground (cm/s)
- $x_{vm}$ : velocity of AMD relative to the ground (cm/s)
- $x_a(i)$ : acceleration of  $i$ -th floor relative to the ground (g)
- $x_{am}$ : acceleration of AMD relative to the ground (g)
- $d(1)=x(1)$ ,  $d(2)=x(2) - x(1)$ ,  $d(3)=x(3) - x(2)$ : inter-story drifts

The inputs are the ground acceleration  $x_{ag}$  (in g) and the control signal  $u$ . We use  $1\text{ g} = 981\text{ cm/s}^2$ .

```
load ThreeStoryData
size(P)
```

State-space model with 20 outputs, 2 inputs, and 28 states.

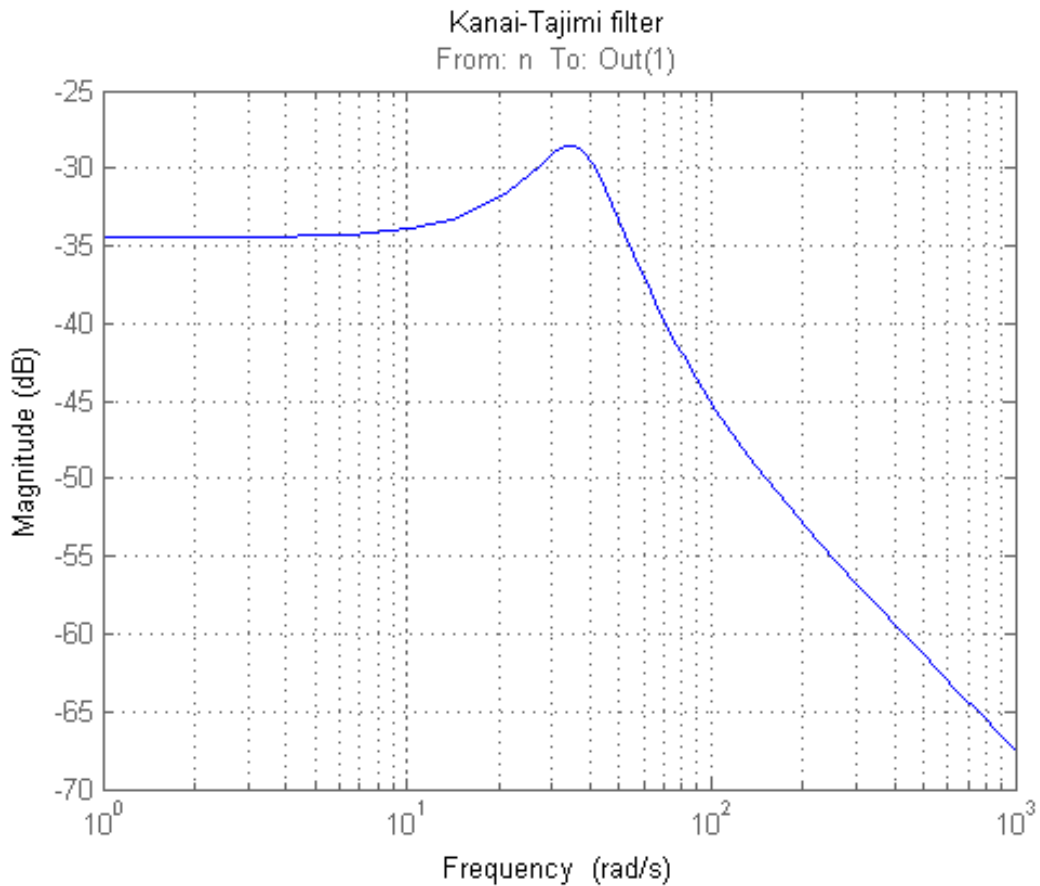
### Model of Earthquake Acceleration

The earthquake acceleration is modeled as a white noise process filtered through a Kanai-Tajimi filter.

```
zg = 0.3;    wg = 37.3;
S0 = 0.03*zg/(pi*wg*(4*zg^2+1));
num = sqrt(S0)*[2*zg*wg wg^2];
den = [1 2*zg*wg wg^2];

F = sqrt(2*pi)*tf(num,den);
F.InputName = 'n';    % white noise input

bodemag(F), grid, title('Kanai-Tajimi filter')
```



### Open-Loop Characteristics

The effect of an earthquake on the uncontrolled structure can be simulated by injecting a white noise input  $n$  into the plant-filter combination. You can also use `covar` to directly compute the standard deviations of the resulting inter-story drifts and accelerations.

```
% Add Kanai-Tajimi filter to the plant
PF = P*append(F , 1);
```

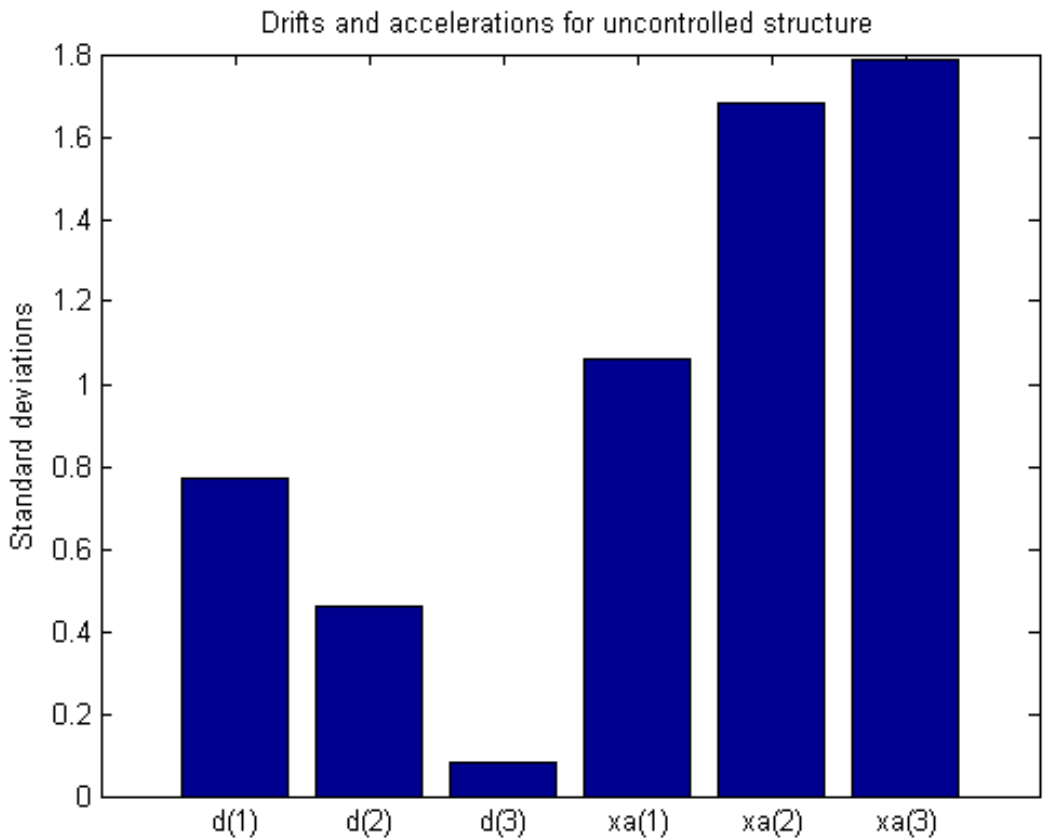
```
% Standard deviations of open-loop drifts
```



```
CV = covar(PF('d','n'),1);
d0 = sqrt(diag(CV));

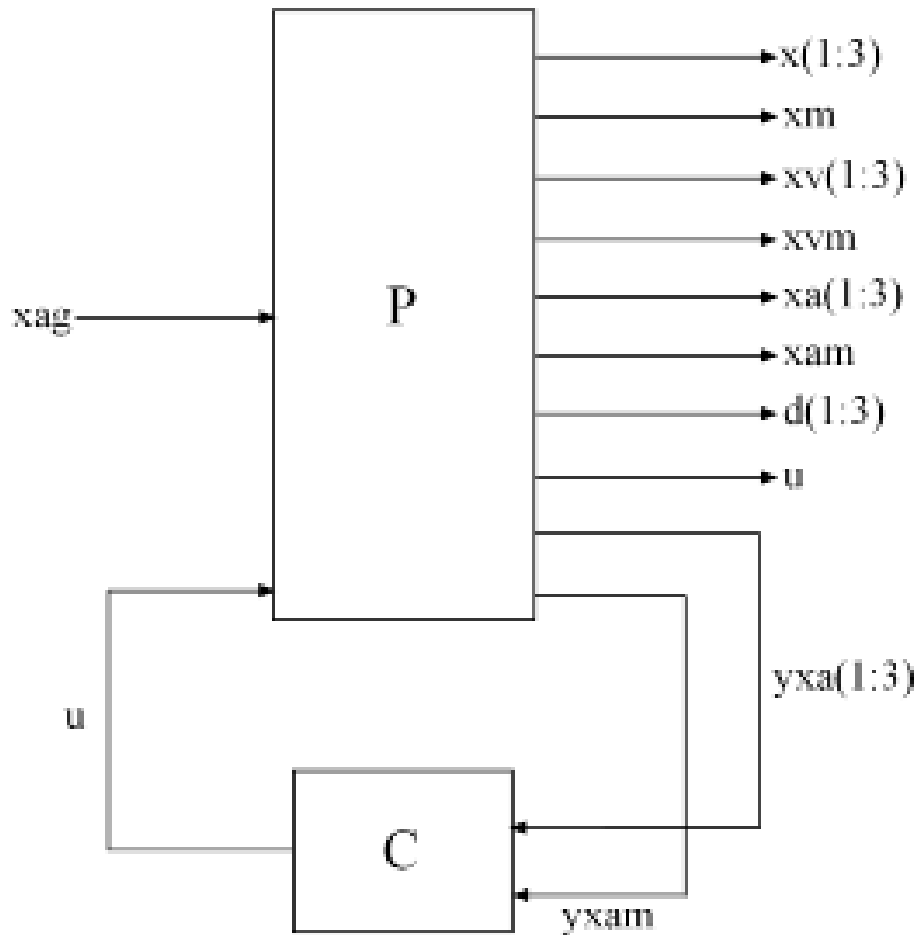
% Standard deviations of open-loop acceleration
CV = covar(PF('xa','n'),1);
xa0 = sqrt(diag(CV));

% Plot open-loop RMS values
clf, bar([d0 ; xa0])
title('Drifts and accelerations for uncontrolled structure')
ylabel('Standard deviations')
set(gca,'XTickLabel',{'d(1)', 'd(2)', 'd(3)', 'xa(1)', 'xa(2)', 'xa(3)'})
```



**Control Structure and Design Requirements**

The control structure is depicted in Figure 2.



**Figure 2: Control Structure**

The controller  $C$  uses measurements  $y_{xa}$  and  $y_{xam}$  of  $x_a$  and  $x_{am}$  to generate the control signal  $u$ . Physically, the control  $u$  is an electrical current driving an hydraulic actuator that moves the masses of the AMD. The design requirements involve:

- Minimization of the inter-story drifts  $d(i)$  and accelerations  $x_a(i)$
- Hard constraints on control effort in terms of mass displacement  $x_m$ , mass acceleration  $x_{am}$ , and control effort  $u$

All design requirements are assessed in terms of standard deviations of the corresponding signals. Use `TuningGoal.Variance` to express these requirements and scale each variable by its open-loop standard deviation to seek uniform relative improvement in all variables.

```
% Soft requirements on drifts and accelerations
Soft = [...
    TuningGoal.Variance('n','d(1)', d0(1)) ; ...
    TuningGoal.Variance('n','d(2)', d0(2)) ; ...
    TuningGoal.Variance('n','d(3)', d0(3)) ; ...
    TuningGoal.Variance('n','xa(1)', xa0(1)) ; ...
    TuningGoal.Variance('n','xa(2)', xa0(2)) ; ...
    TuningGoal.Variance('n','xa(3)', xa0(3))];

% Hard requirements on control effort
Hard = [...
    TuningGoal.Variance('n','xm', 3) ; ...
    TuningGoal.Variance('n','xam', 2) ; ...
    TuningGoal.Variance('n','u', 1)];
```

### Controller Tuning

`systemtune` lets you tune virtually any controller structure subject to these requirements. The controller complexity can be adjusted by trial-and-error, starting with sufficiently high order to gauge the limits of performance, then reducing the order until you observe a noticeable performance degradation. For this example, start with a 5th-order controller with no feedthrough term.

```
C = ltiblock.ss('C',5,1,4);
C.d.Value = 0;
C.d.Free = false; % Fix feedthrough to zero
```

Construct a tunable model `T0` of the closed-loop system of Figure 2 and tune the controller parameters with `systemtune`.

```
% Build tunable closed-loop model
```

```
T0 = lft(PF,C);

% Tune controller parameters
[T,fSoft,gHard] = systune(T0,Soft,Hard);

Final: Soft = 0.601, Hard = 0.99806, Iterations = 163
```

The summary indicates that we achieved an overall reduction of 40% in standard deviations (Soft = 0.6) while meeting all hard constraints (Hard < 1).

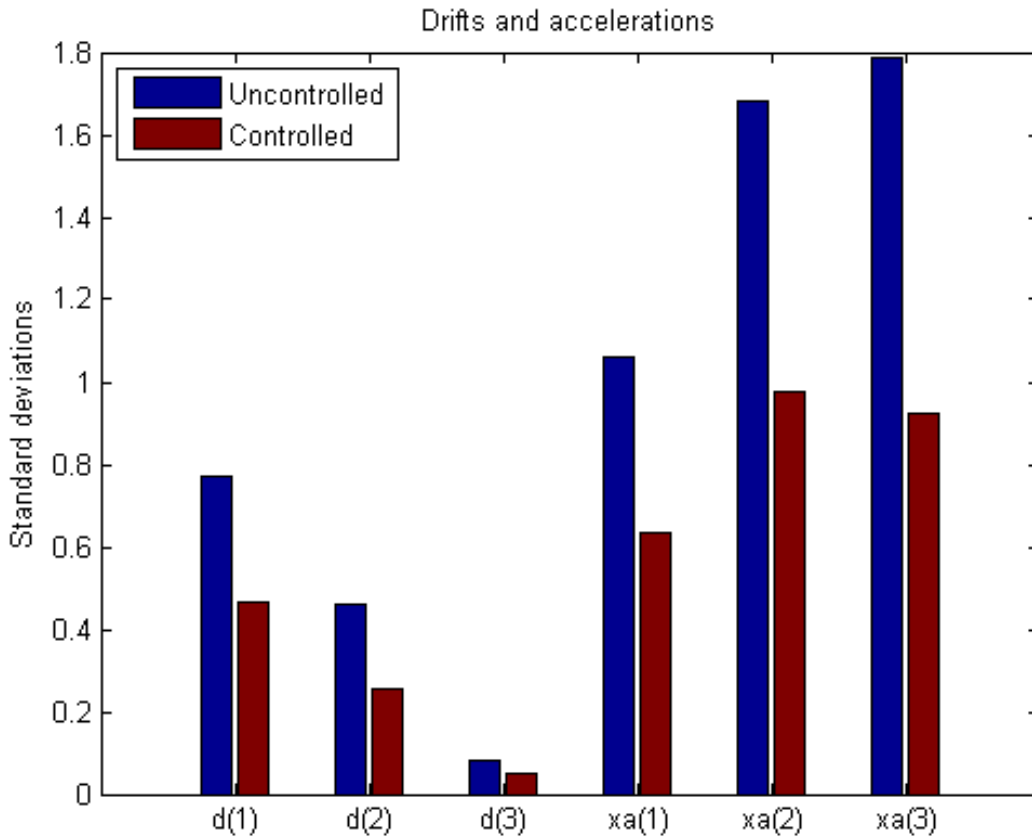
### Validation

Compute the standard deviations of the drifts and accelerations for the controlled structure and compare with the uncontrolled results. The AMD control system yields significant reduction of both drift and acceleration.

```
% Standard deviations of closed-loop drifts
CV = covar(T('d','n'),1);
d = sqrt(diag(CV));

% Standard deviations of closed-loop acceleration
CV = covar(T('xa','n'),1);
xa = sqrt(diag(CV));

% Compare open- and closed-loop values
clf, bar([d0 d; xa0 xa])
title('Drifts and accelerations')
ylabel('Standard deviations')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
legend('Uncontrolled','Controlled','location','NorthWest')
```



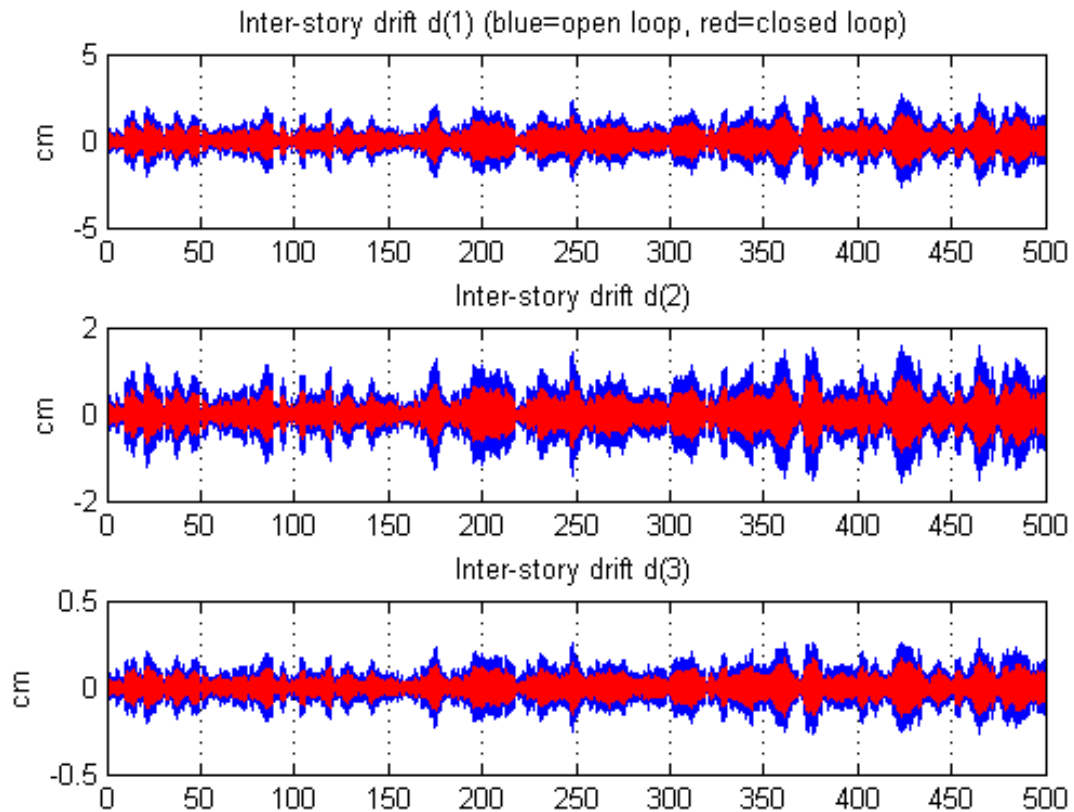
Simulate the response of the 3-story structure to an earthquake-like excitation in both open and closed loop. The earthquake acceleration is modeled as a white noise process colored by the Kanai-Tajimi filter.

```
% Sampled white noise process
rng('default')
dt = 1e-3;
t = 0:dt:500;
n = randn(1,length(t))/sqrt(dt); % white noise signal
```

```
% Open-loop simulation
ysimOL = lsim(PF(:,1), n , t);

% Closed-loop simulation
ysimCL = lsim(T, n , t);

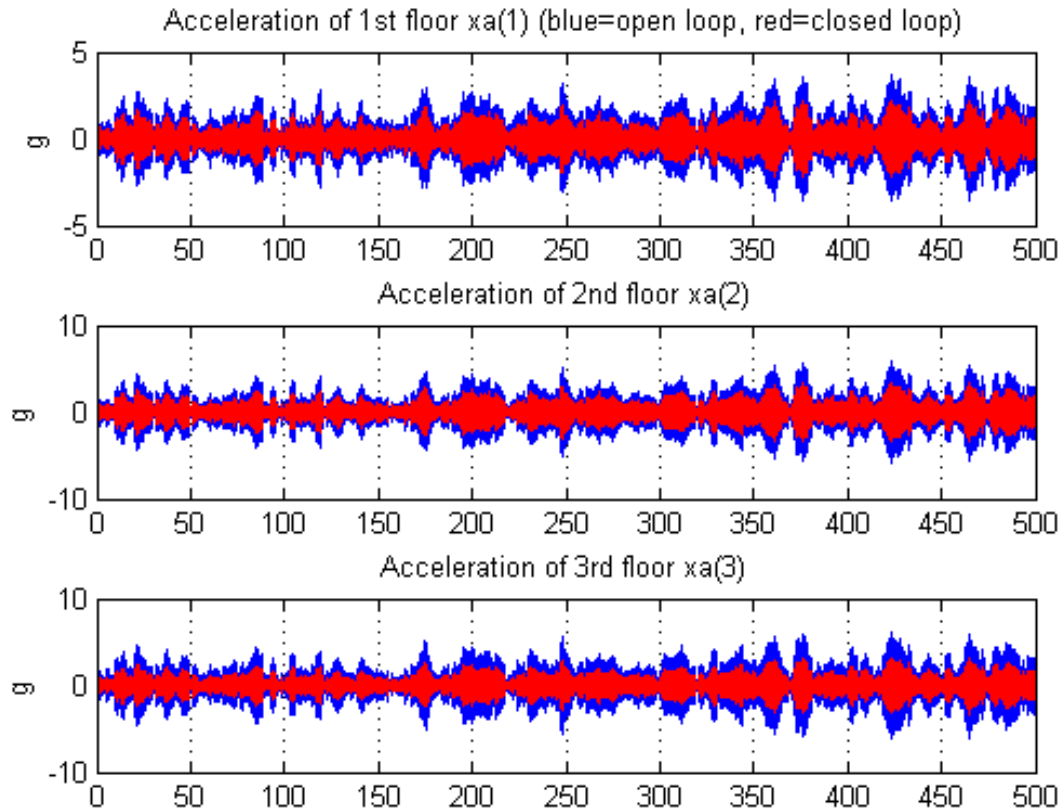
% Drifts
clf, subplot(311); plot(t,ysimOL(:,13),'b',t,ysimCL(:,13),'r'); grid;
title('Inter-story drift d(1) (blue=open loop, red=closed loop)'); ylabel('cm');
subplot(312); plot(t,ysimOL(:,14),'b',t,ysimCL(:,14),'r'); grid;
title('Inter-story drift d(2)'); ylabel('cm');
subplot(313); plot(t,ysimOL(:,15),'b',t,ysimCL(:,15),'r'); grid;
title('Inter-story drift d(3)'); ylabel('cm');
```



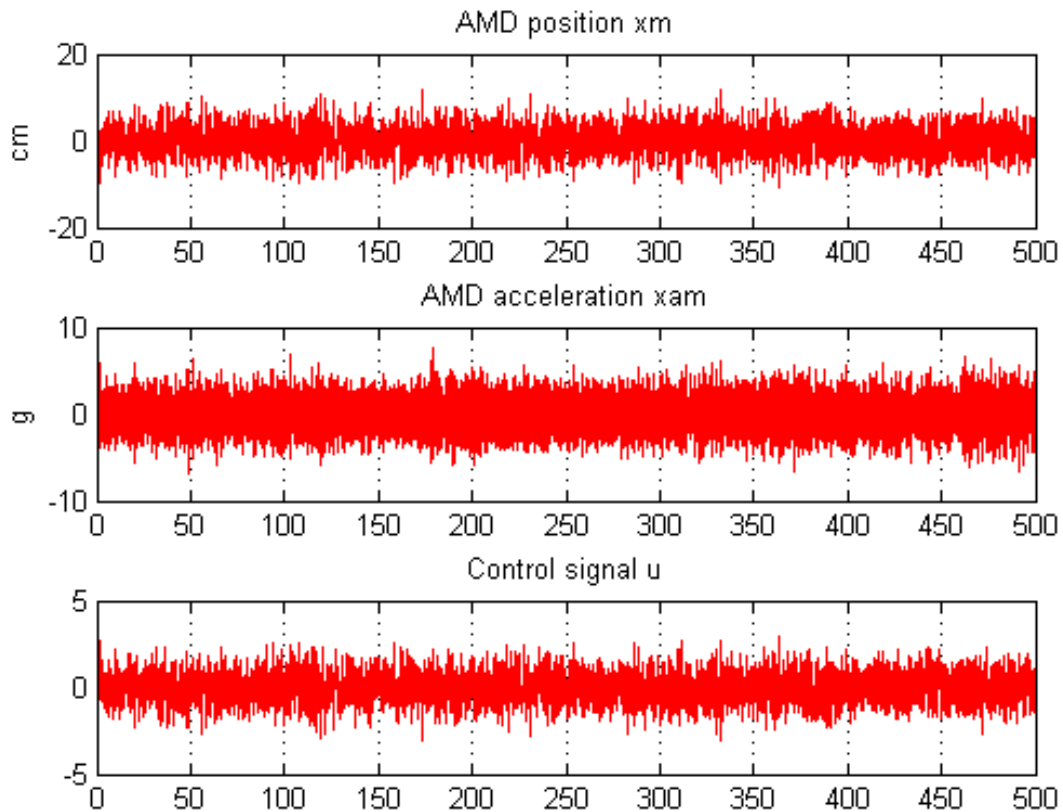
```
% Accelerations
```

```
clf, subplot(311); plot(t,ysimOL(:,9),'b',t,ysimCL(:,9),'r'); grid;
title('Acceleration of 1st floor xa(1) (blue=open loop, red=closed loop)');
subplot(312); plot(t,ysimOL(:,10),'b',t,ysimCL(:,10),'r'); grid;
title('Acceleration of 2nd floor xa(2)'); ylabel('g');
subplot(313); plot(t,ysimOL(:,11),'b',t,ysimCL(:,11),'r'); grid;
title('Acceleration of 3rd floor xa(3)'); ylabel('g');
```





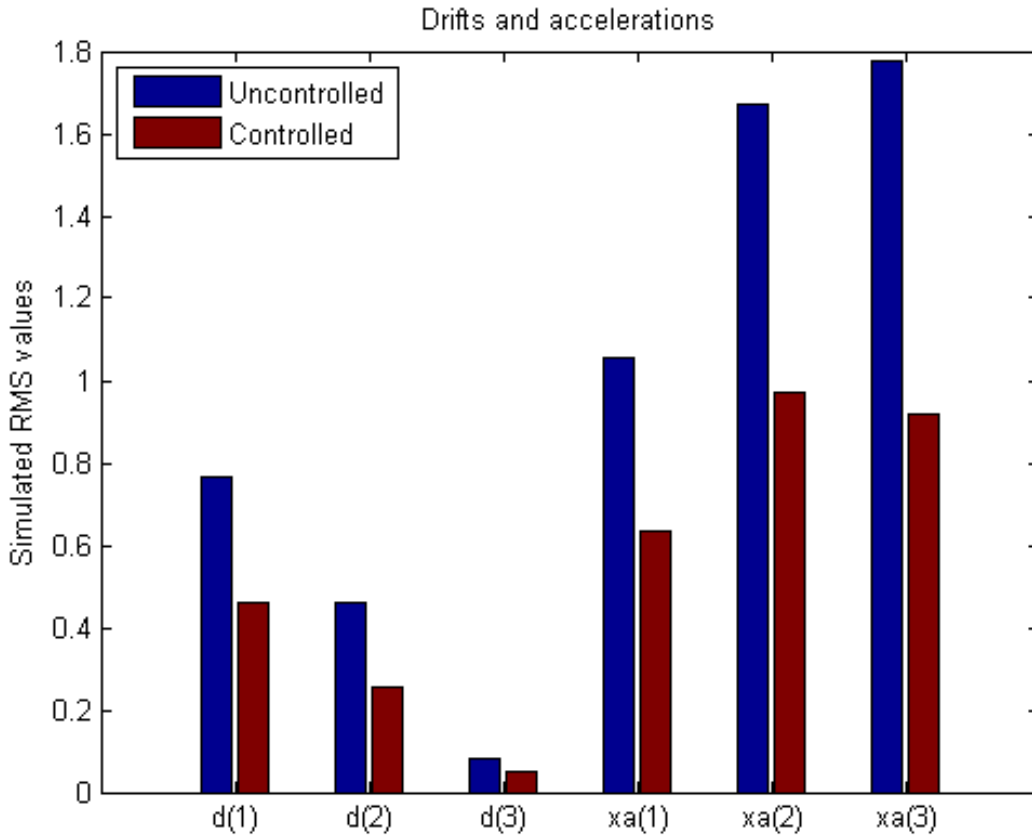
```
% Control variables
clf, subplot(311); plot(t,ysimCL(:,4),'r'); grid;
title('AMD position  $x_m$ '); ylabel('cm');
subplot(312); plot(t,ysimCL(:,12),'r'); grid;
title('AMD acceleration  $x_{am}$ '); ylabel('g');
subplot(313); plot(t,ysimCL(:,16),'r'); grid;
title('Control signal  $u$ ');
```



Plot the root-mean-square (RMS) of the simulated signals for both the controlled and uncontrolled scenarios. Assuming ergodicity, the RMS performance can be estimated from a single sufficiently long simulation of the process and coincides with the standard deviations computed earlier. Indeed the RMS plot closely matches the standard deviation plot obtained earlier.

```
clf, bar([std(ysim0L(:,13:15)) std(ysim0L(:,9:11)) ; ...
         std(ysimCL(:,13:15)) std(ysimCL(:,9:11))])
title('Drifts and accelerations')
ylabel('Simulated RMS values')
```

```
set(gca,'XTickLabel',{'d(1)', 'd(2)', 'd(3)', 'xa(1)', 'xa(2)', 'xa(3)'})  
legend('Uncontrolled', 'Controlled', 'location', 'NorthWest')
```



Overall, the controller achieves significant reduction of ground vibration both in terms of drift and acceleration for all stories while meeting the hard constraints on control effort and mass displacement.

## Tuning of a Two-Loop Autopilot

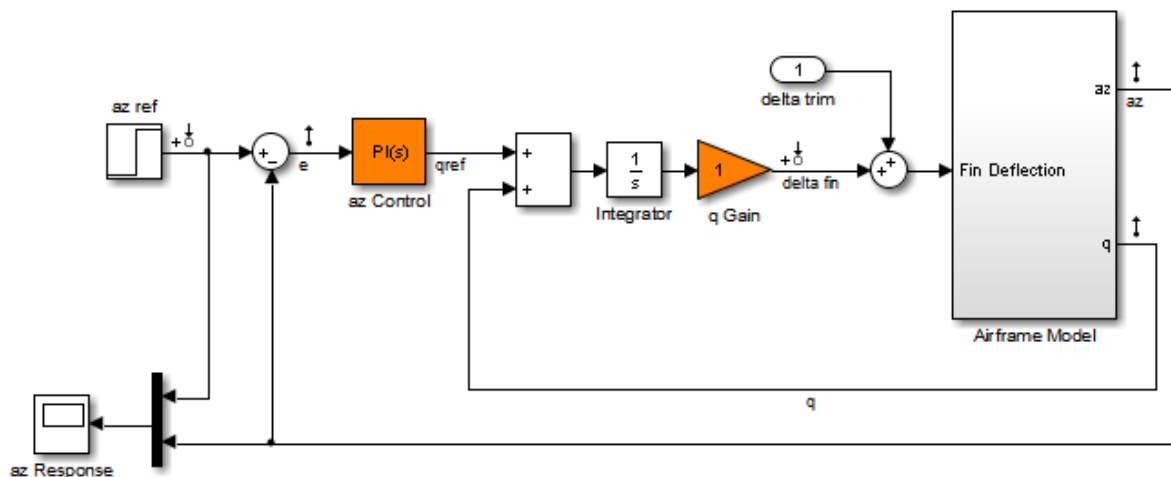
This example shows how to use Robust Control Toolbox™ to tune a two-loop autopilot controlling the pitch rate and vertical acceleration of an airframe.

### Model of Airframe Autopilot

The airframe dynamics and the autopilot are modeled in Simulink.

```
open_system('rct_airframe1')
```

### Two-loop autopilot for controlling the vertical acceleration of an airframe



See `AutopilotExample` to see how you can tune the autopilot gains with the `LOOPTUNE` command.

The autopilot consists of two cascaded loops. The inner loop controls the pitch rate  $q$ , and the outer loop controls the vertical acceleration  $az$  in response to the pilot stick command  $az_{ref}$ . In this architecture, the tunable elements include the PI controller gains ("az Control" block) and the pitch-rate gain ("q Gain" block). The autopilot must be tuned to respond to a step command  $az_{ref}$  in about 1 second with minimal overshoot. In this example, we tune the autopilot gains for one flight condition corresponding to zero incidence and a speed of 984 m/s.

To analyze the airframe dynamics, trim the airframe for  $\alpha = 0$  and  $V = 984\text{m/s}$ . The trim condition corresponds to zero normal acceleration and pitching moment ( $w$  and  $q$  steady). Use `findop` to compute the corresponding closed-loop operating condition. Note that we added a "delta trim" input port so that `findop` can adjust the fin deflection to produce the desired equilibrium of forces and moments.

```
opspec = operspec('rct_airframe1');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
% integrator states unknown, not steady
opspec.States(5).SteadyState = 0;
opspec.States(6).SteadyState = 0;

op = findop('rct_airframe1',opspec);
```

Operating Point Search Report:

-----

Operating Report for the Model rct\_airframe1.  
(Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.

States:

-----

(1.) rct\_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equat  
       x:                  0          dx:                  984

```

      x:    -3.05e+03    dx:         0
(2.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:         0      dx:    -0.00972
(3.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:        984     dx:        22.7
      x:         0      dx:    -1.44e-11 (0)
(4.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:    -0.00972    dx:    1.72e-16 (0)
(5.) rct_airframe1/Integrator
      x:    0.000708    dx:    -0.00972
(6.) rct_airframe1/az Control/Integrator
      x:         0      dx:    0.000242

```

Inputs:

-----

```

(1.) rct_airframe1/delta trim
      u:    0.000708    [-Inf Inf]

```

Outputs: None

-----

Linearize the "Airframe Model" block for the computed trim condition `op` and plot the gains from the fin deflection `delta` to `az` and `q`:

```

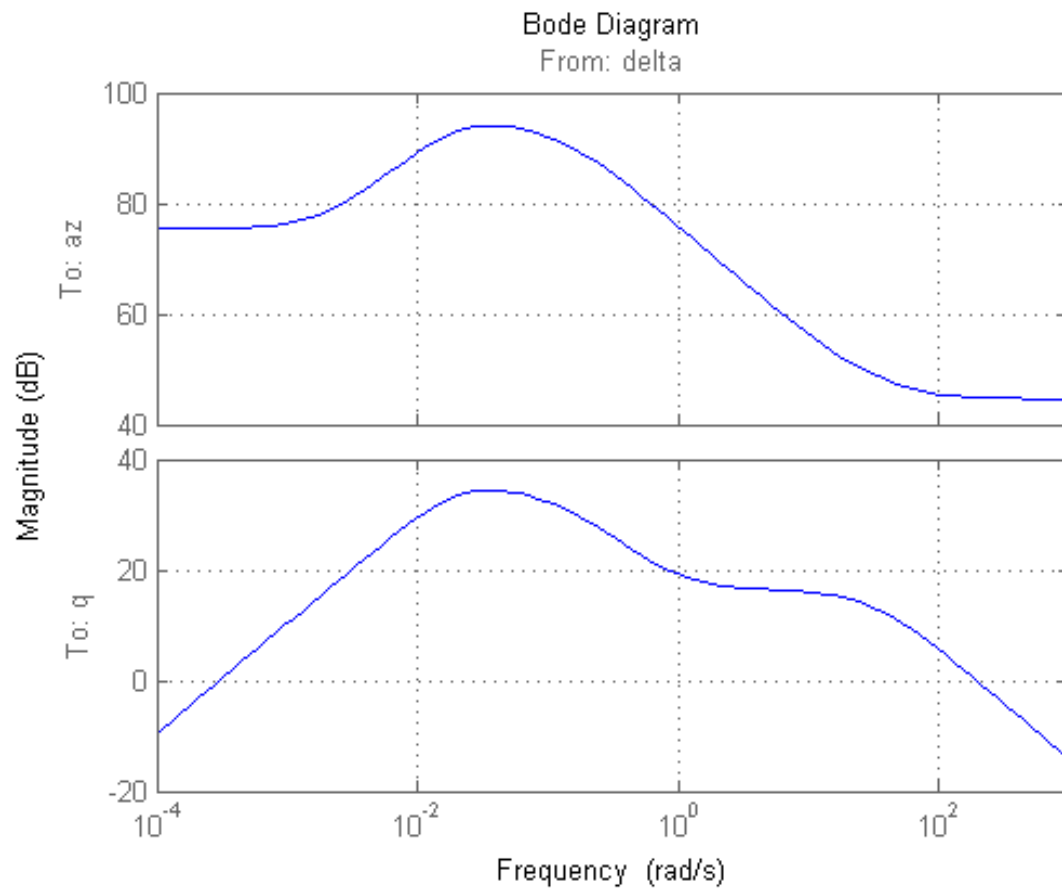
G = linearize('rct_airframe1','rct_airframe1/Airframe Model',op);
G.InputName = 'delta';
G.OutputName = {'az','q'};

```

```

bodemag(G), grid

```



Note that the airframe model has an unstable pole:

`pole(G)`

`ans =`

```
-0.0320
-0.0255
 0.1253
```

-29.4685

### Frequency-Domain Tuning with LOOPTUNE

You can use the `looptune` function to automatically tune multi-loop control systems subject to basic requirements such as integral action, adequate stability margins, and desired bandwidth. To apply `looptune` to the autopilot model, create an instance of the `sITunable` interface and designate the Simulink blocks `az Control` and `q Gain` as tunable. Also specify the trim condition `op` to correctly linearize the airframe dynamics.

```
STO = sITunable('rct_airframe1',{'az Control','q Gain'},op);
```

Next outline the plant/controller boundary by specifying the control and measurement signals:

```
STO.addControl('delta fin');  
STO.addMeasurement({'az','q'});
```

Finally, tune the control system parameters to meet the 1 second response time requirement. In the frequency domain, this roughly corresponds to a gain crossover frequency  $\omega_c = 5$  rad/s for the open-loop response at the plant input "delta fin".

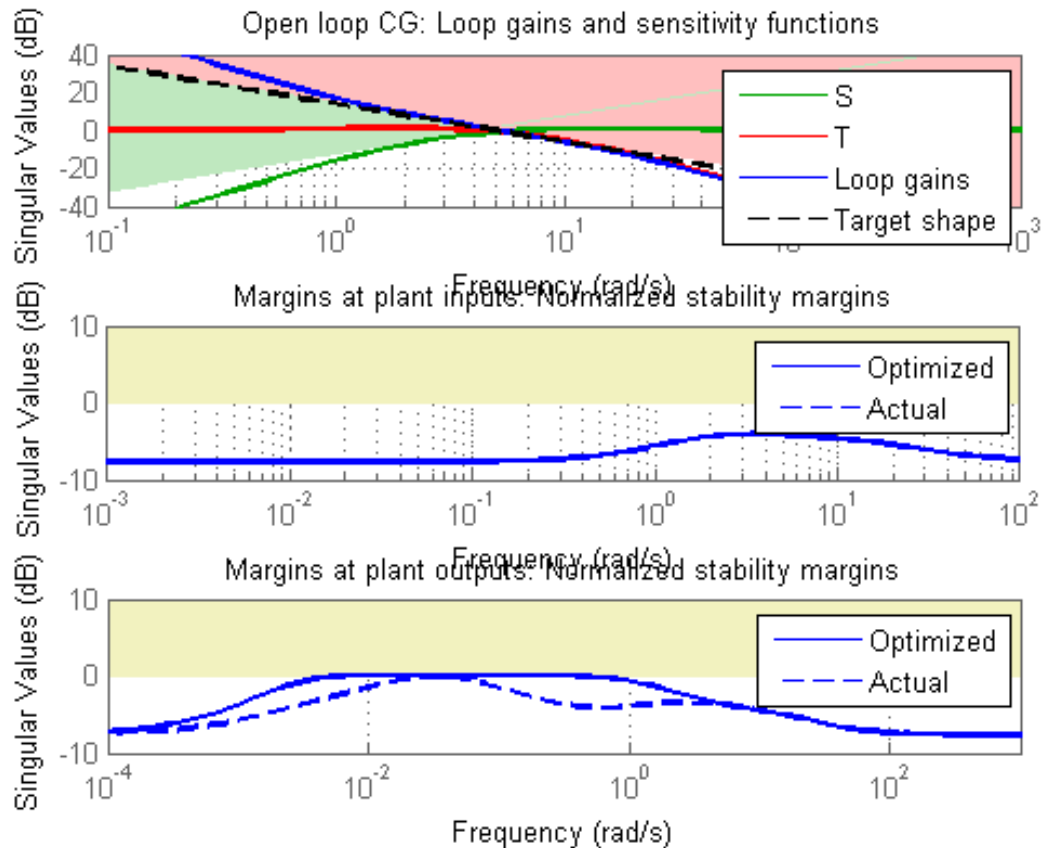
```
wc = 5;  
[ST,gam,Info] = STO.looptune(wc);
```

```
Final: Peak gain = 1.01, Iterations = 52
```

The requirements are normalized so a final value near 1 means that all requirements are met. Confirm this by graphically validating the design.

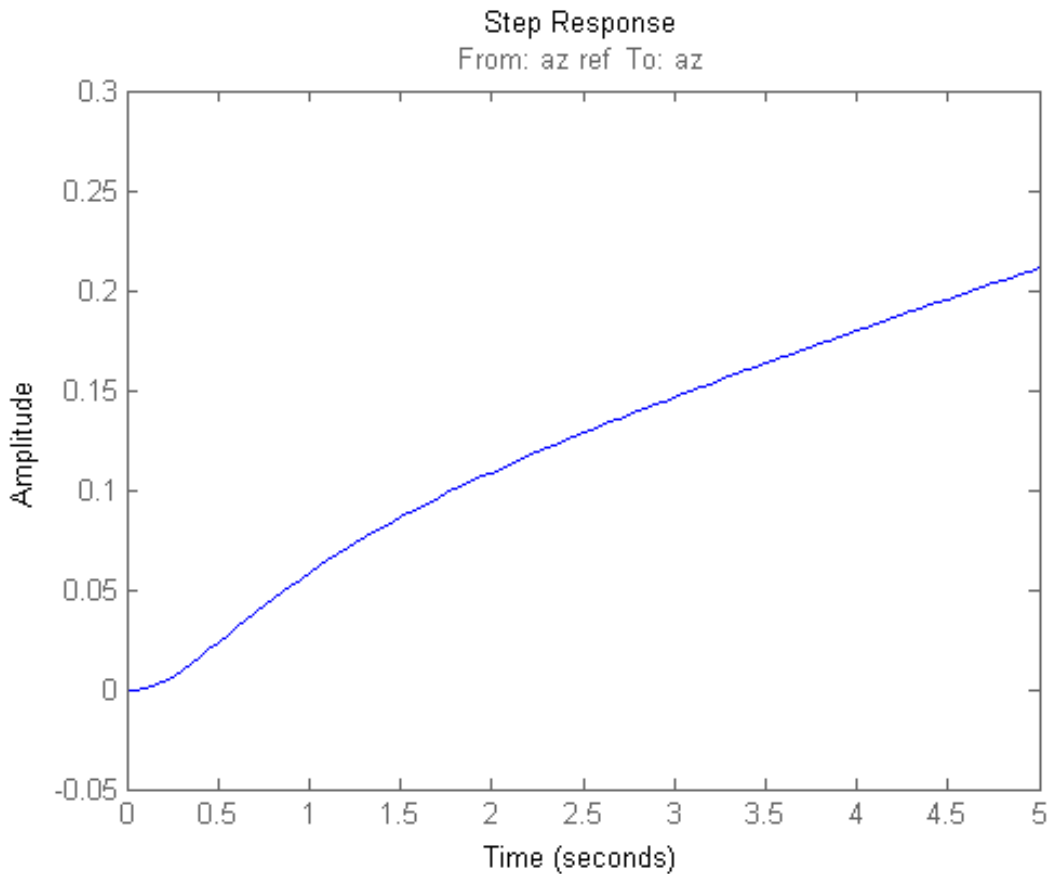
```
ST.loopview(Info)
```





The first plot confirms that the open-loop response has integral action and the desired gain crossover frequency while the second plot shows that the MIMO stability margins are satisfactory (the blue curve should remain below the yellow bound). Next check the response from the step command `azref` to the vertical acceleration `az`:

```
T = ST.getIOTransfer('az ref','az');
clf, step(T,5)
```



The acceleration  $az$  does not track  $az_{ref}$  despite the presence of an integrator in the loop. This is because the feedback loop acts on the two variables  $az$  and  $q$  and we have not specified which one should track  $az_{ref}$ .

#### **Adding a Tracking Requirement**

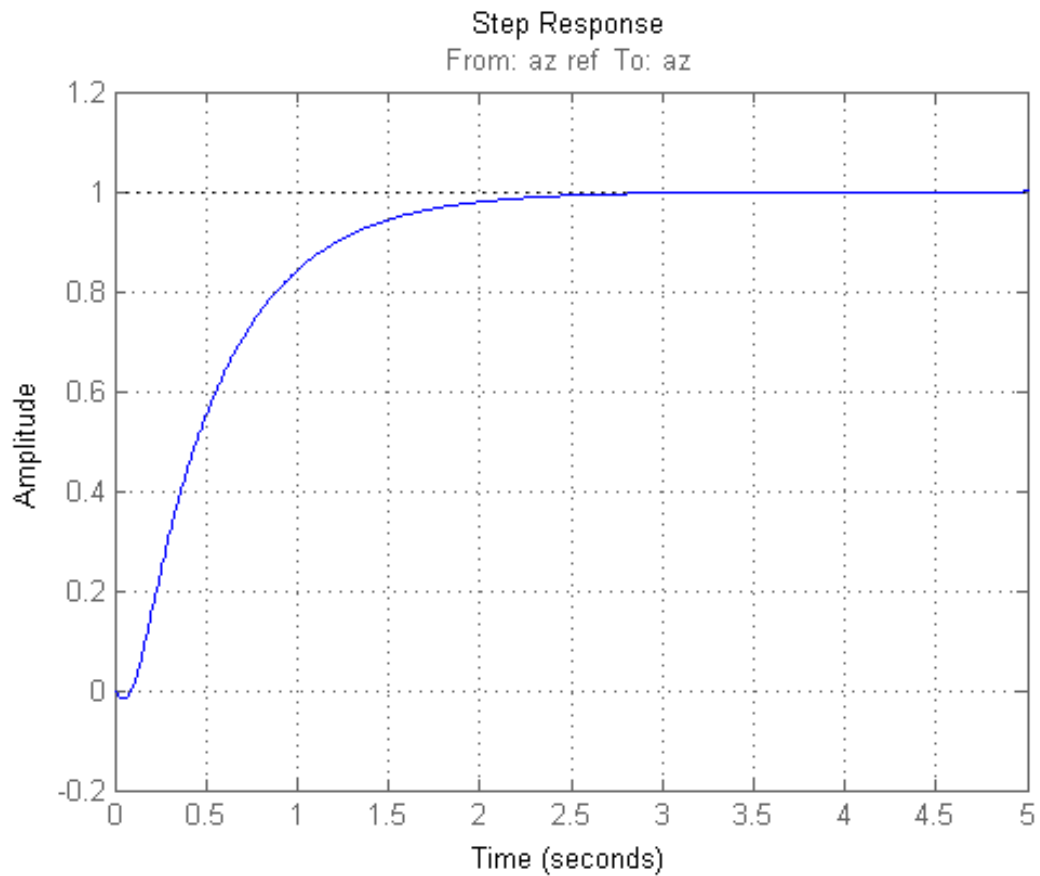
To remedy this issue, add an explicit requirement that  $az$  should follow the step command  $az_{ref}$  with a 1 second response time. Also relax the gain crossover requirement to the interval  $[3,12]$  to let the tuner find the appropriate gain crossover frequency.

```
TrackReq = TuningGoal.Tracking('az ref','az',1);  
ST = ST0.looptune([3,12],TrackReq);
```

Final: Peak gain = 1.23, Iterations = 47

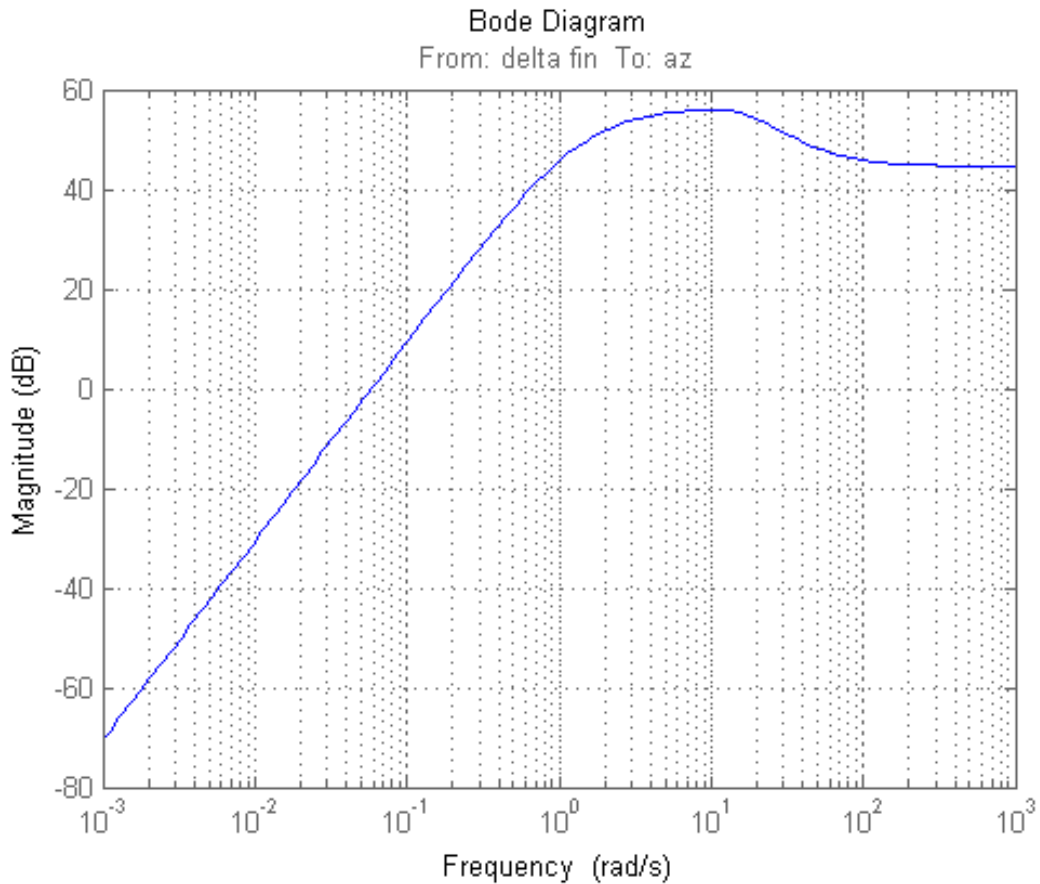
The step response from azref to az is now satisfactory:

```
Tr1 = ST.getIOTransfer('az ref','az');  
step(Tr1,5), grid
```

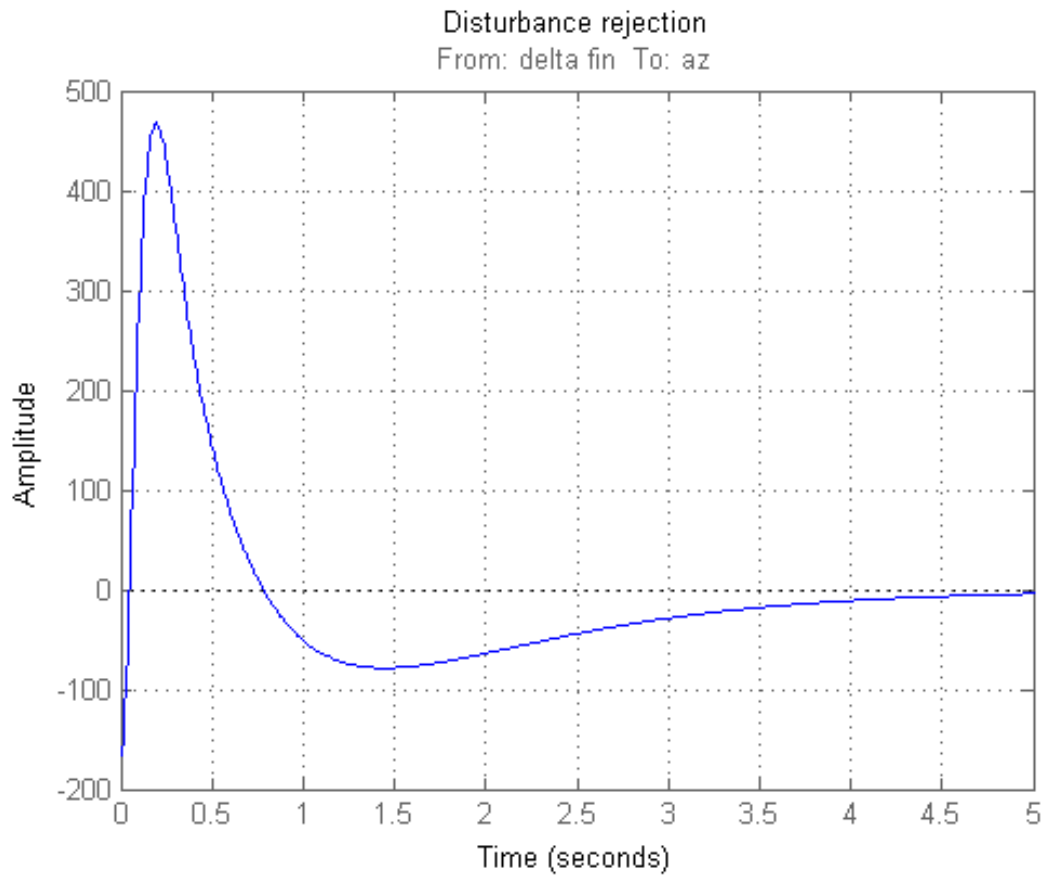


Also check the disturbance rejection characteristics by looking at the responses from a disturbance entering at the plant input

```
Td1 = ST.getIOTransfer('delta fin','az');  
bodemag(Td1), grid
```



```
step(Td1,5), grid, title('Disturbance rejection')
```



Use `showBlockValue` to see the tuned values of the PI controller and inner-loop gain

```
ST.showBlockValue()
```

```
Block "rct_airframe1/az Control" =
```

$$K_p + K_i * \frac{1}{s}$$

```

                                s

    with Kp = 0.00166, Ki = 0.0017

Name: az_Control
Continuous-time PI controller in parallel form.

-----

Block "rct_airframe1/q Gain" =

    d =
        u1
    y1  1.987

Name: q_Gain
Static gain.
```

If this design is satisfactory, use `writeBlockValue` to apply the tuned values to the Simulink model and simulate the tuned controller in Simulink.

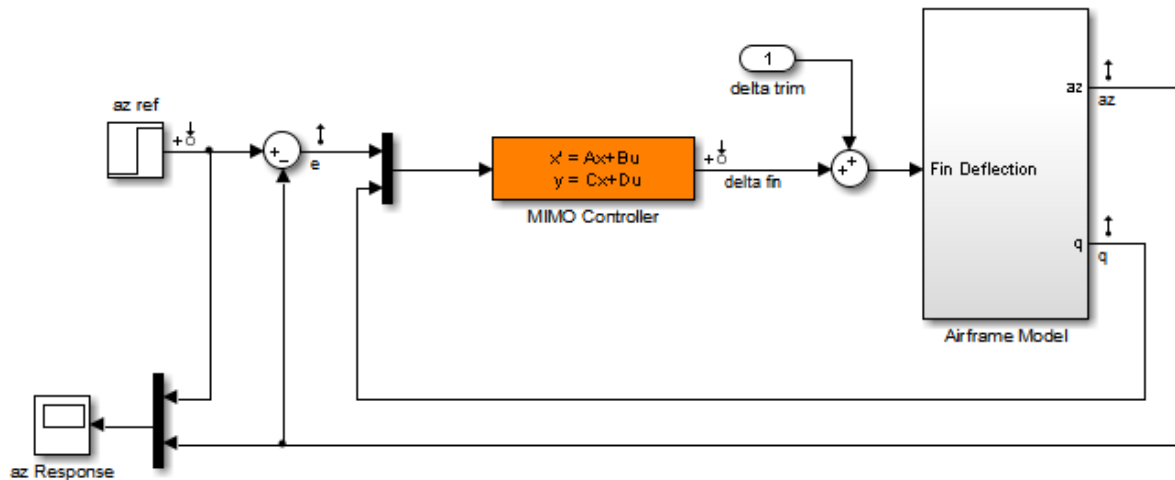
```
ST.writeBlockValue()
```

### **MIMO Design with SYSTUNE**

Cascaded loops are commonly used for autopilots. Yet one may wonder how a single MIMO controller that uses both `az` and `q` to generate the actuator command `delta fin` would compare with the two-loop architecture. Trying new control architectures is easy with `systemtune` or `looptune`. For variety, we now use `systemtune` to tune the following MIMO architecture.

```
open_system('rct_airframe2')
```

## Two-loop autopilot for controlling the vertical acceleration of an airframe



See `AutopilotExample` to see how you can tune this MIMO controller with the `SYSTUNE` command.

As before, compute the trim condition for  $\alpha = 0$  and  $V = 984\text{m/s}$ .

```
opspec =operspec('rct_airframe2');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
% controller states unknown, not steady
opspec.States(5).SteadyState = [0;0];
```

```
op = findop('rct_airframe2',opspec);
```

```
Operating Point Search Report:
```

```
-----
```

```
Operating Report for the Model rct_airframe2.  
(Time-Varying Components Evaluated at time t=0)
```

```
Operating point specifications were successfully met.
```

```
States:
```

```
-----
```

```
(1.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:          0          dx:          984
      x:   -3.05e+03      dx:           0
(2.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:          0          dx:   -0.00972
(3.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:          984          dx:          22.7
      x:          0          dx:   2.46e-11 (0)
(4.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equat
      x:   -0.00972      dx:   -4.02e-16 (0)
(5.) rct_airframe2/MIMO Controller
      x:   0.000654      dx:   -0.009
      x:   1.06e-18      dx:   0.0303
```

```
Inputs:
```

```
-----
```

```
(1.) rct_airframe2/delta trim
      u:   0.000436      [-Inf Inf]
```

```
Outputs: None
```

```
-----
```

As with looptune, use the s1Tunable interface to configure the Simulink model for tuning.

```
STO = s1Tunable('rct_airframe2','MIMO Controller',op);
```



```
STO.addControl('delta fin');
```

Try a second-order MIMO controller with zero feedthrough from  $e$  to  $\delta a$ . To do this, create the desired controller parameterization and associate it with the "MIMO Controller" block using `setBlockParam`:

```
C0 = ltiblock.ss('C',2,1,2); % Second-order controller
C0.d.Value(1) = 0; C0.d.Free(1) = false; % Fix D(1) to zero
STO.setBlockParam('MIMO Controller',C0)
```

Next create the tuning requirements. Here we use the following four requirements:

- 1 Tracking:**  $az$  should respond in about 1 second to the  $az_{ref}$  command
- 2 Bandwidth and roll-off:** The loop gain at  $\delta a$  should roll off after 25 rad/s with a -20 dB/decade slope
- 3 Stability margins:** The margins at  $\delta a$  should exceed 7 dB and 45 degrees
- 4 Disturbance rejection:** The attenuation factor for input disturbances should be 40 dB at 1 rad/s increasing to 100 dB at 0.001 rad/s.

```
% Tracking
Req1 = TuningGoal.Tracking('az ref','az',1);

% Bandwidth and roll-off
Req2 = TuningGoal.MaxLoopGain('delta fin',tf(25,[1 0]));

% Margins
Req3 = TuningGoal.Margins('delta fin',7,45);

% Disturbance rejection
% Use an FRD model to sketch the desired attenuation profile with a few poi
Freqs = [0 0.001 1];
MinAtt = [100 100 40]; % in dB
Req4 = TuningGoal.Rejection('delta fin',frd(db2mag(MinAtt),Freqs));
Req4.Focus = [0 1];
```

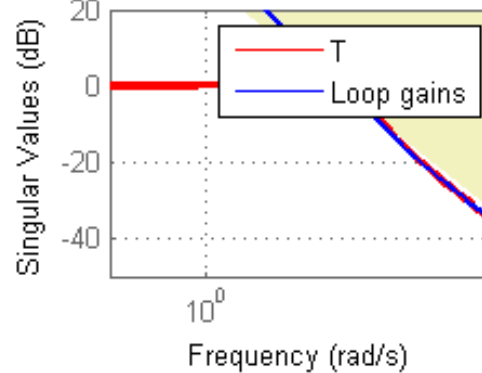
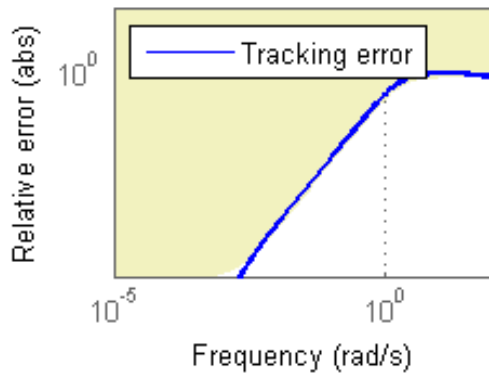
You can now use `systemtune` to tune the controller parameters subject to these requirements.

```
AllReqs = [Req1,Req2,Req3 Req4];  
Opt = systemtuneOptions('RandomStart',3);  
  
rng(0)  
[ST,fSoft,~,Info] = ST0.systemtune(AllReqs,Opt);  
  
Final: Soft = 1.42, Hard = -Inf, Iterations = 40  
Final: Soft = 1.14, Hard = -Inf, Iterations = 96  
Final: Soft = 1.14, Hard = -Inf, Iterations = 51  
Final: Soft = 1.14, Hard = -Inf, Iterations = 155
```

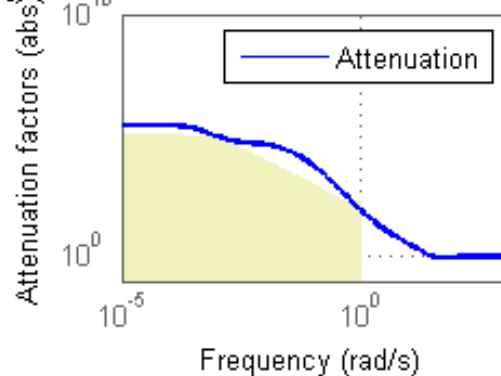
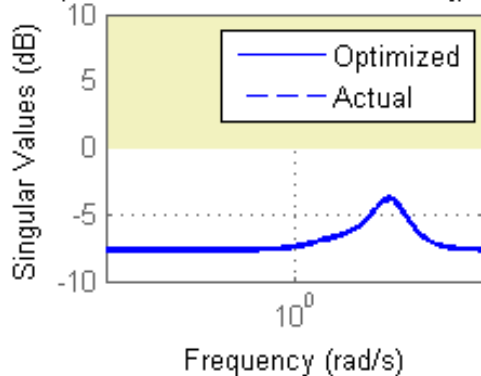
The best design has an overall objective value close to 1, indicating that all four requirements are nearly met. Use `viewSpec` to inspect each requirement for the best design.

```
viewSpec(AllReqs,ST,Info)
```

Requirement 1: Tracking error as a function of frequency. Requirement 2: Maximum loop gain as a function of frequency.



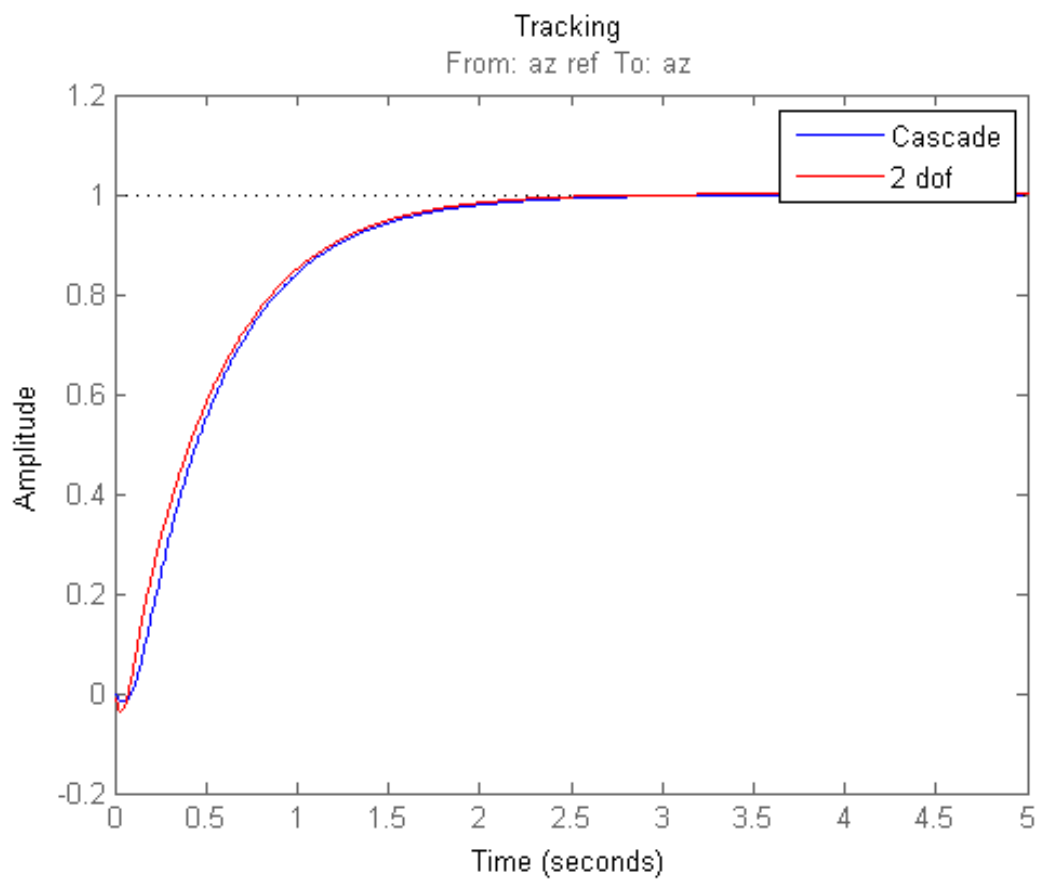
Requirement 3: Normalized stability margin. Requirement 4: Disturbance attenuation as a function of frequency.



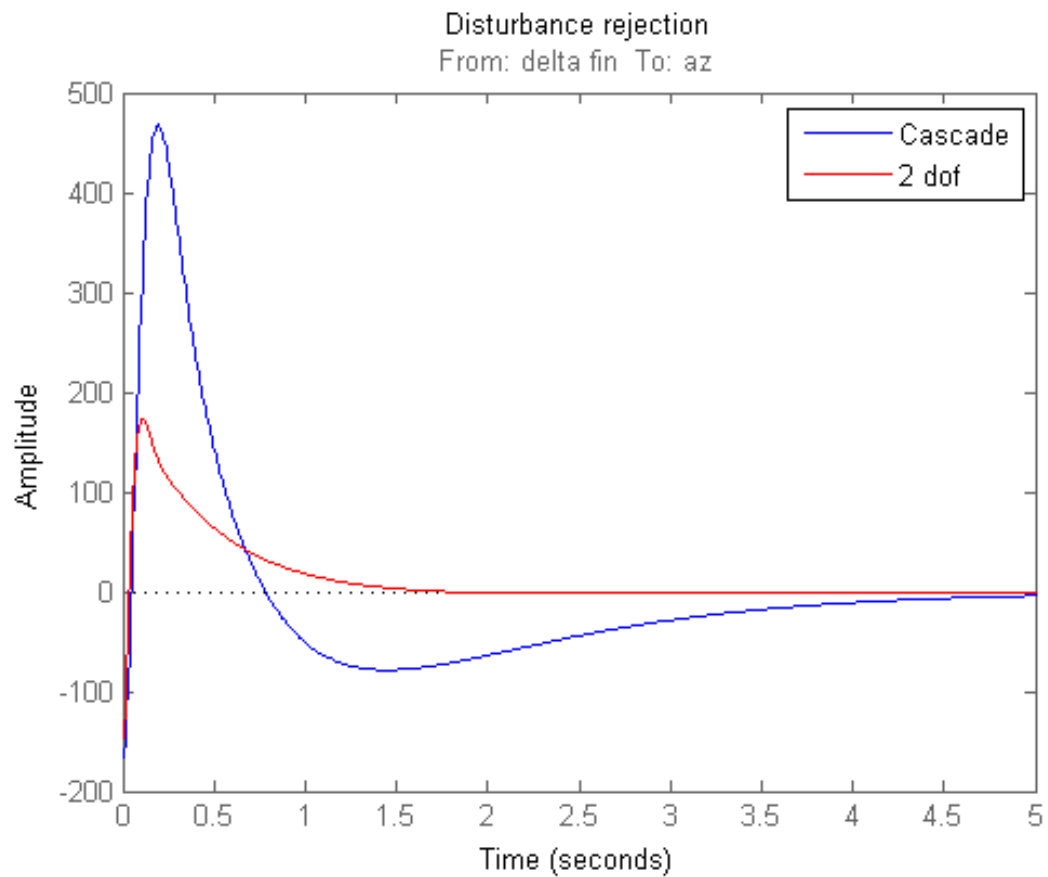
Compute the closed-loop responses and compare with the two-loop design.

```
T = ST.getIOTransfer({'az ref','delta fin'},'az');

clf
step(Tr1,'b',T(1),'r',5)
title('Tracking'), legend('Cascade','2 dof')
```



```
step(Td1,'b',T(2),'r',5)  
title('Disturbance rejection'), legend('Cascade','2 dof')
```



The tracking performance is similar but the second design has better disturbance rejection properties.

## Multi-Loop Control of a Helicopter

This example shows how to use Robust Control Toolbox™ to tune a multi-loop controller for a rotorcraft.

### Helicopter Model

This example uses an 8-state helicopter model at the hovering trim condition. The state vector  $x = [u, w, q, \text{theta}, v, p, \text{phi}, r]$  consists of

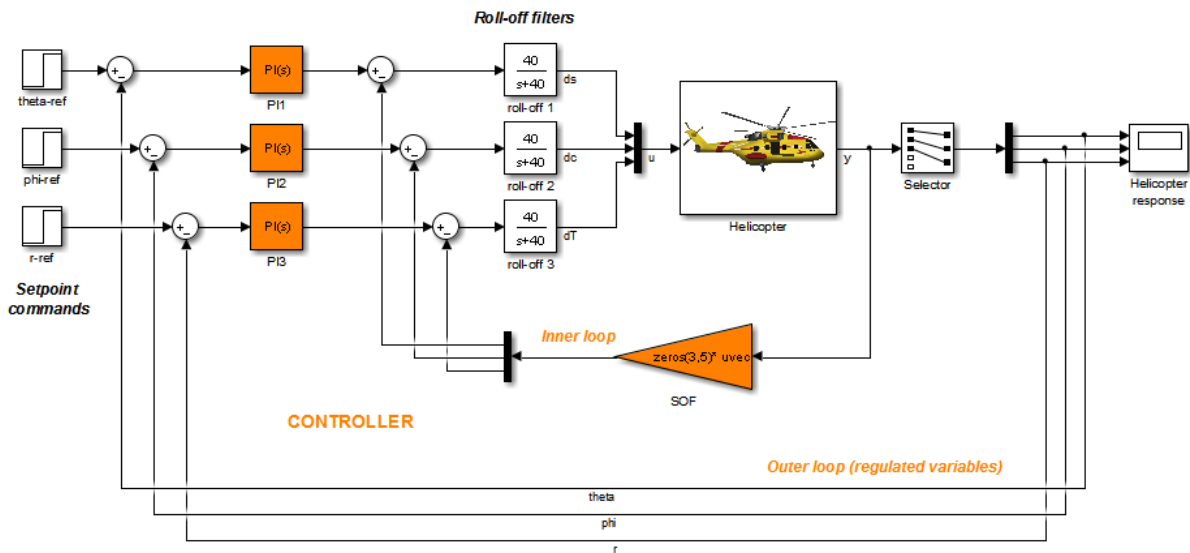
- Longitudinal velocity  $u$  (m/s)
- Lateral velocity  $v$  (m/s)
- Normal velocity  $w$  (m/s)
- Pitch angle  $\text{theta}$  (deg)
- Roll angle  $\text{phi}$  (deg)
- Roll rate  $p$  (deg/s)
- Pitch rate  $q$  (deg/s)
- Yaw rate  $r$  (deg/s).

The controller generates commands  $d_s, d_c, d_T$  in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of  $\text{theta}, \text{phi}, p, q,$  and  $r$ .

### Control Architecture

The following Simulink model depicts the control architecture:

```
open_system('rct_helico')
```



The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance. The main control objectives are as follows:

- Track setpoint changes in  $\theta$ ,  $\phi$ , and  $r$  with zero steady-state error, settling times of about 2 seconds, minimal overshoot, and minimal cross-coupling
- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise
- Provide strong multivariable gain and phase margins (robustness to simultaneous gain/phase variations at the plant inputs and outputs, see `loopmargin` for details).

We use lowpass filters with cutoff at 40 rad/s to partially enforce the second objective.

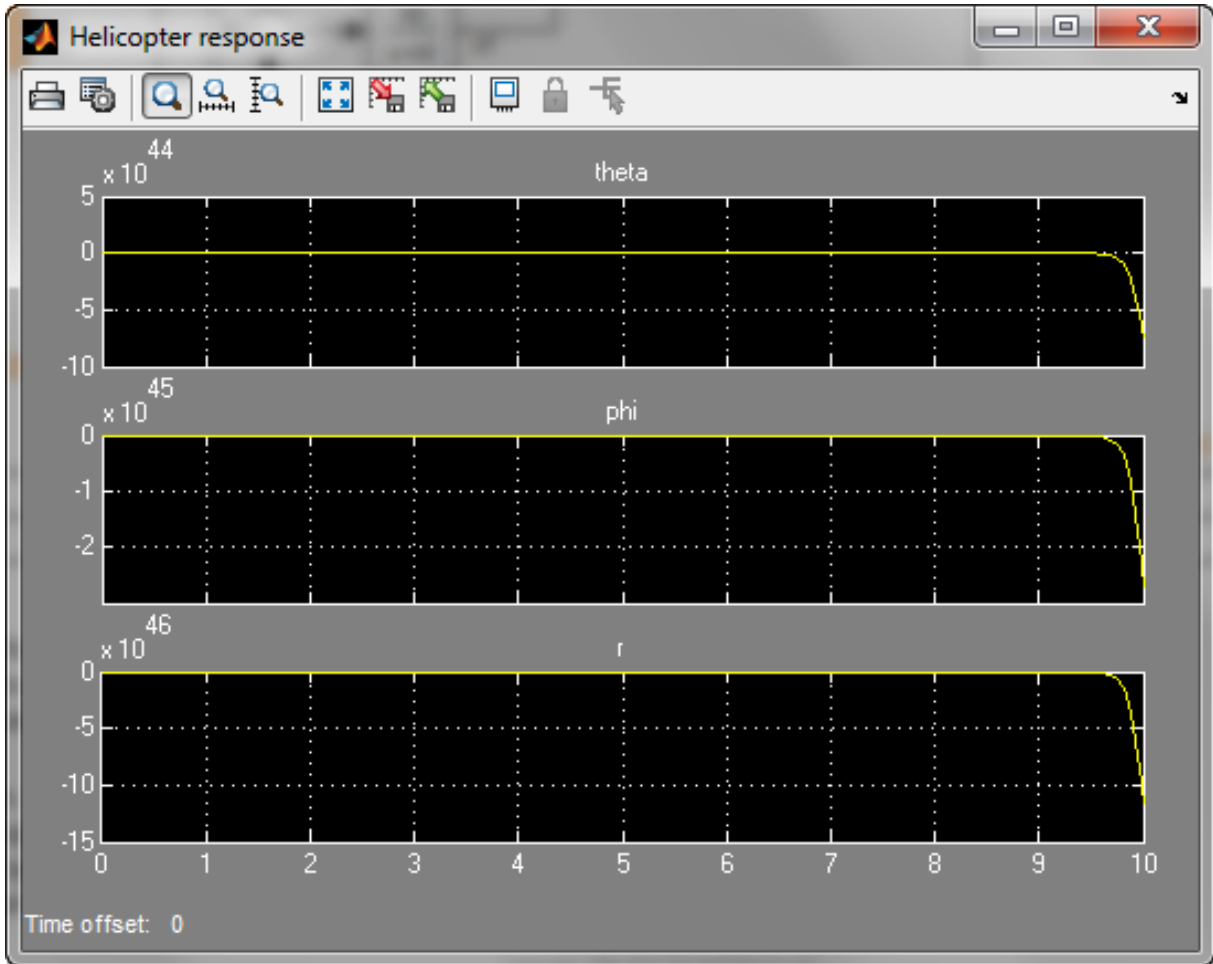
### Controller Tuning

You can jointly tune the inner and outer loops with the `sysTune` command. This command only requires models of the plant and controller along with the desired bandwidth (which is function of the desired response time). When the control system is modeled in Simulink, you can use the `sITunable` interface to quickly set up the tuning task. Create an instance of this interface with the list of blocks to be tuned.

```
ST0 = sITunable('rct_helico',{'PI1','PI2','PI3','SOF'});
```

Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model ( $1 + 1/s$  for the PI controllers and zero for the static output-feedback gain). Simulating the model shows that the control system is unstable for these initial values:





Specify the I/O signals of interest for setpoint tracking.

```
STO.addIO({'theta-ref','phi-ref','r-ref'},'input') % setpoint commands
STO.addIO({'theta','phi','r'},'output') % corresponding output
```

Also identify the plant inputs and outputs (control and measurement signals) where the stability margin are measured.

```
ST0.addControl('u');  
ST0.addMeasurement('y');
```

Finally, capture the design requirements using `TuningGoal` objects. We use the following requirements for this example:

- **Tracking requirement:** The outputs `theta`, `phi`, `r` must track the commands `theta_ref`, `phi_ref`, `r_ref` with a 2-second response time
- **Stability margins:** The multivariable gain and phase margins at the plant inputs `u` and plant outputs `y` must be at least 5 dB and 40 degrees
- **Fast dynamics:** The magnitude of the closed-loop poles must not exceed 20 to prevent fast dynamics and jerky transients

```
% Tracking  
TrackReq = TuningGoal.Tracking({'theta-ref','phi-ref','r-ref'},...  
    {'theta','phi','r'},2,1e-2);  
  
% Gain and phase margins at plant inputs and outputs  
MarginReq1 = TuningGoal.Margins('u',5,40);  
MarginReq2 = TuningGoal.Margins('y',5,40);  
  
% Limit on fast dynamics  
PoleReq = TuningGoal.Poles();  
PoleReq.MaxFrequency = 20;
```

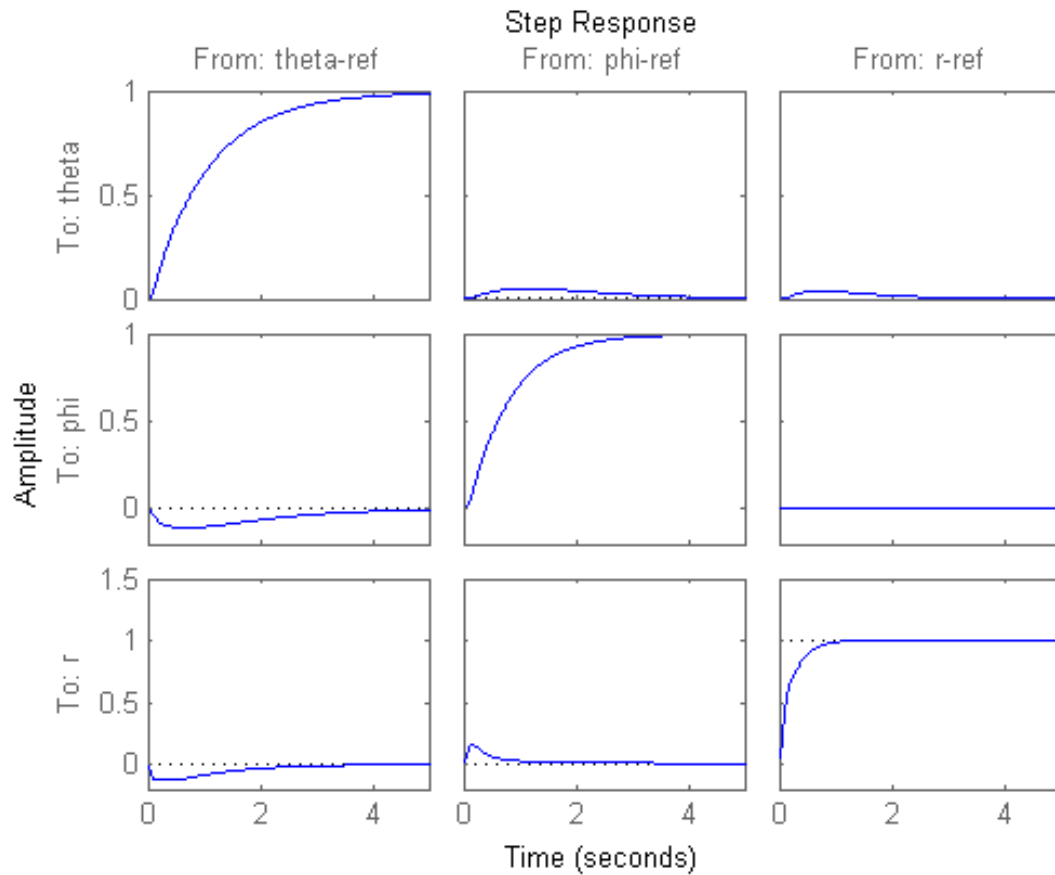
You can now use `systemtune` to jointly tune all controller parameters. This returns the tuned version `ST1` of the control system `ST0`.

```
AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];  
[ST1,fSoft,~,Info] = ST0.systemtune(AllReqs);
```

```
Final: Soft = 1.15, Hard = -Inf, Iterations = 60
```

The final value is close to 1 so the requirements are nearly met. Plot the tuned responses to step commands in `theta`, `phi`, `r`:

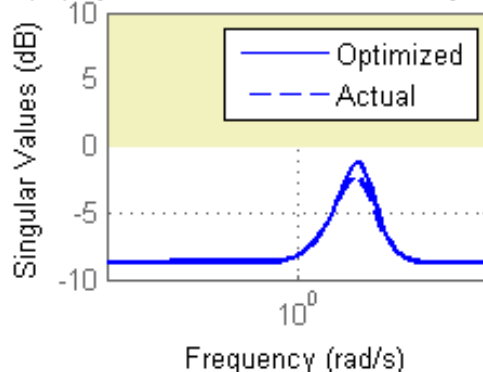
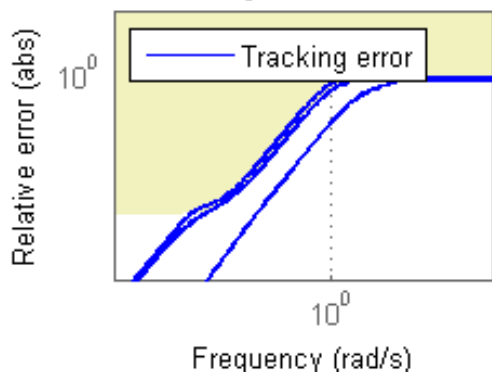
```
T1 = ST1.getIOTransfer({'theta-ref','phi-ref','r-ref'},{'theta','phi','r'})  
step(T1,5)
```



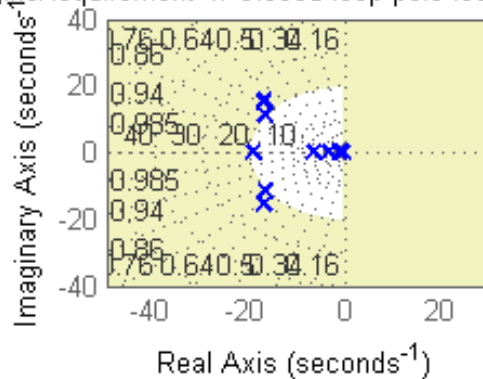
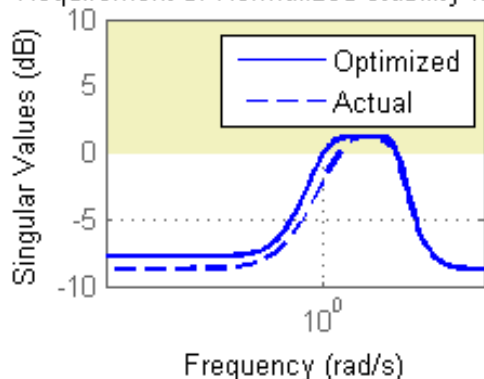
The response time is about two seconds with no overshoot and little cross-coupling. You can use `viewSpec` for a more thorough validation of each requirement, including a visual assessment of the multivariable stability margins (see `loopmargin` for details):

```
viewSpec(AllReqs,ST1,Info)
```

Requirement 1: Tracking error as a function of frequency Requirement 2: Normalized stability margins



Requirement 3: Normalized stability margins Requirement 4: Closed-loop pole location



Inspect the tuned values of the PI controllers and static output-feedback gain.

```
ST1.showTunable()
```

```
Block "rct_helico/PI1" =
```

$$K_p + K_i * \frac{1}{s}$$

with  $K_p = 0.501$ ,  $K_i = 1.89$

Name: PI1

Continuous-time PI controller in parallel form.

-----

Block "rct\_helico/PI2" =

$$K_p + K_i * \frac{1}{s}$$

with  $K_p = -0.0565$ ,  $K_i = -1.86$

Name: PI2

Continuous-time PI controller in parallel form.

-----

Block "rct\_helico/PI3" =

$$K_p + K_i * \frac{1}{s}$$

with  $K_p = -0.482$ ,  $K_i = -4.84$

Name: PI3

Continuous-time PI controller in parallel form.

-----

Block "rct\_helico/SOF" =

d =

	u1	u2	u3	u4	u5
y1	2.035	-0.243	-0.1026	0.5102	0.0064
y2	-0.4642	-1.511	0.01247	-0.07461	-0.1088

```
y3 -0.9591 0.4915 -0.9768 -0.0835 0.2357
```

```
Name: SOF  
Static gain.
```

### **Benefit of the Inner Loop**

You may wonder whether the static output feedback is necessary and whether PID controllers aren't enough to control the helicopter. This question is easily answered by re-tuning the controller with the inner loop open. First break the inner loop by adding a loop opening after the SOF block:

```
ST0.addOpening('SOF')
```

Then remove the SOF block from the tunable block list and re-parameterize the PI blocks as full-blown PIDs with the correct loop signs (as inferred from the first design).

```
PID = pid(0,0.001,0.001,.01); % initial guess for PID controllers
```

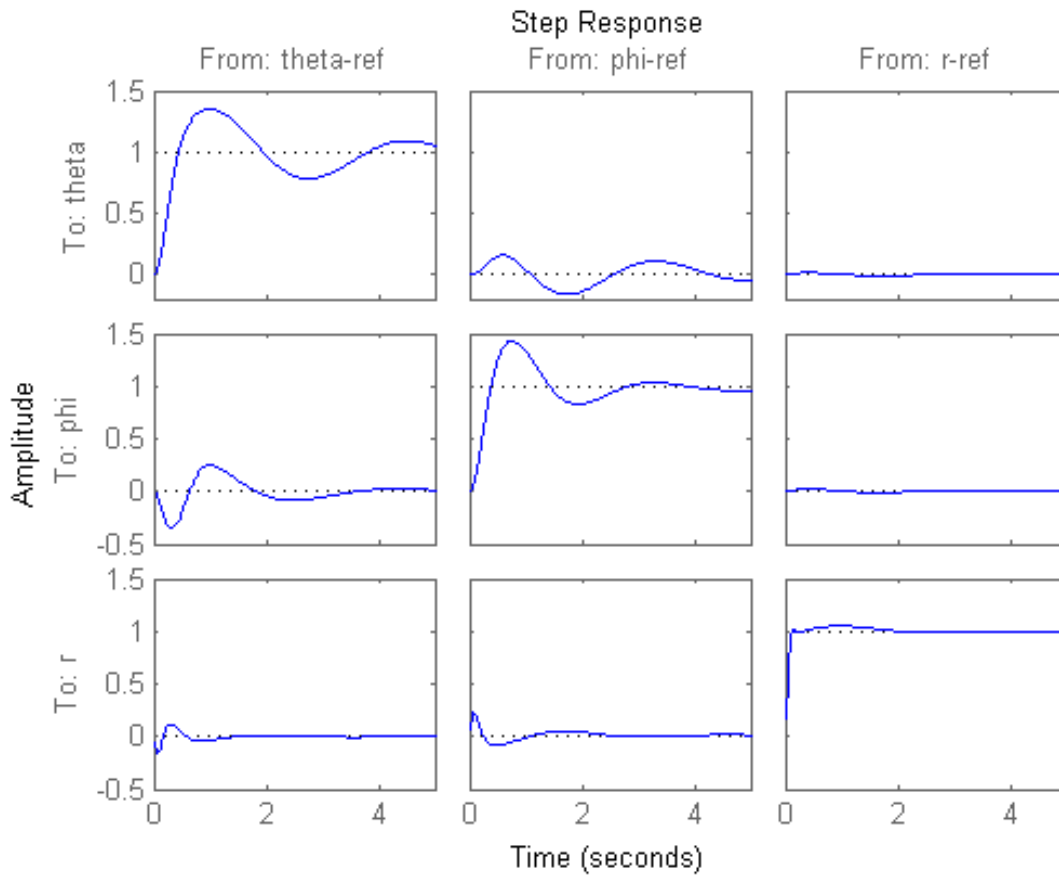
```
ST0.TunedBlocks = ST0.TunedBlocks(1:3);  
ST0.setBlockParam('PI1',ltiblock.pid('C1',PID));  
ST0.setBlockParam('PI2',ltiblock.pid('C2',-PID));  
ST0.setBlockParam('PI3',ltiblock.pid('C3',-PID));
```

Re-tune the three PID controllers and plot the closed-loop step responses.

```
[ST2,fSoft,~,Info] = ST0.systune(AllReqs);
```

```
Final: Soft = 2.07, Hard = -Inf, Iterations = 102
```

```
T2 = ST2.getIOTransfer({'theta-ref','phi-ref','r-ref'},{'theta','phi','r'})  
clf, step(T2,5)
```



The final value is no longer close to 1 and the step responses confirm some performance degradation with regard to settling time, overshoot, and decoupling. This suggests that the inner loop has an important stabilizing effect that should be preserved.

## Fixed-Structure Autopilot for a Passenger Jet

This example shows how to use Robust Control Toolbox™ to tune the standard configuration of a longitudinal autopilot. We thank Professor D. Alazard from Institut Supérieur de l'Aéronautique et de l'Espace for providing the aircraft model and Professor Pierre Apkarian from ONERA for developing the example.

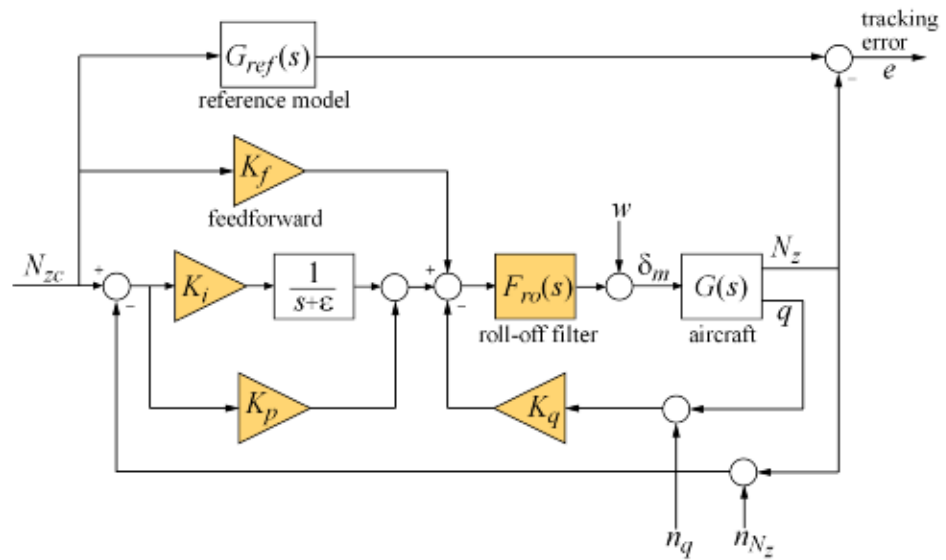
### Aircraft Model and Autopilot Configuration

The longitudinal autopilot for a supersonic passenger jet flying at Mach 0.7 and 5000 ft is depicted in Figure 1. The autopilot main purpose is to follow vertical acceleration commands  $N_{ze}$  issued by the pilot. The feedback structure consists of an inner loop controlling the pitch rate  $q$  and an outer loop controlling the vertical acceleration  $N_z$ . The autopilot also includes a feedforward component and a reference model  $G_{ref}(s)$  that specifies the desired response to a step command  $N_{ze}$ . Finally, the second-order roll-off filter

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

is used to attenuate noise and limit the control bandwidth as a safeguard against unmodeled dynamics. The tunable components are highlighted in orange.

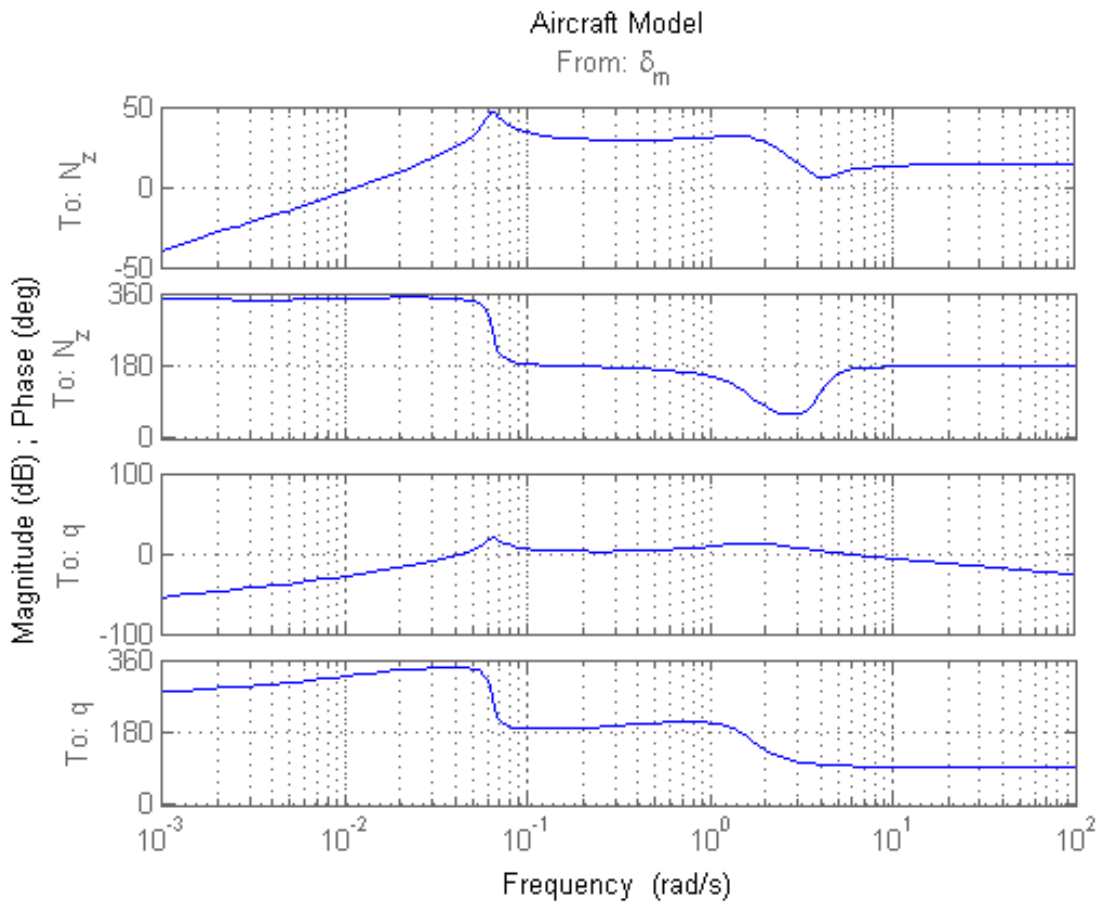




**Figure 1: Longitudinal Autopilot Configuration.**

The aircraft model  $G(s)$  is a 5-state model, the state variables being the aerodynamic speed  $V$  (m/s), the climb angle  $\gamma$  (rad), the angle of attack  $\alpha$  (rad), the pitch rate  $q$  (rad/s), and the altitude  $H$  (m). The elevator deflection  $\delta_m$  (rad) is used to control the vertical load factor  $N_z$ . The open-loop dynamics include the  $\alpha$  oscillation with frequency and damping ratio  $\omega_n = 1.7$  (rad/s) and  $\zeta = 0.33$ , the phugoid mode  $\omega_n = 0.64$  (rad/s) and  $\zeta = 0.06$ , and the slow altitude mode  $\lambda = -0.0026$ .

```
load ConcordeData G
bode(G,{1e-3,1e2}), grid
title('Aircraft Model')
```

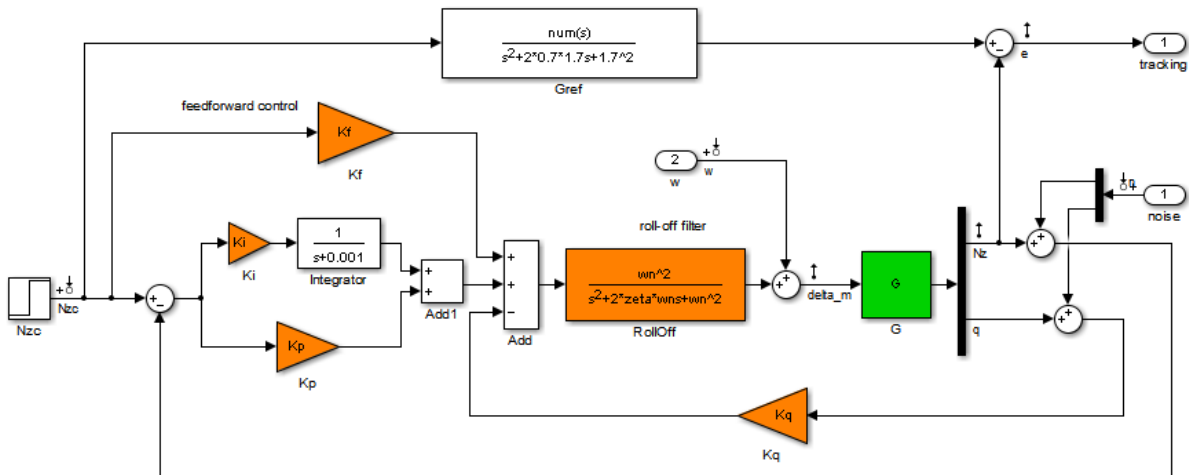


Note the zero at the origin in  $G(s)$ . Because of this zero, we cannot achieve zero steady-state error and must instead focus on the transient response to acceleration commands. Note that acceleration commands are transient in nature so steady-state behavior is not a concern. This zero at the origin also precludes pure integral action so we use a pseudo-integrator  $1/(s + \epsilon)$  with  $\epsilon = 0.001$ .

**Tuning Setup**

When the control system is modeled in Simulink, you can use the `sITunable` interface to quickly set up the tuning task. Open the Simulink model of the autopilot.

```
open_system('rct_concorde')
```



Longitudinal autopilot for a passenger jet

You can tune this autopilot with the `SYSTUNE` command, see `concorde_demo` for details

Copyright 2004-2012 The MathWorks, Inc.

Configure the `sITunable` interface by listing the tuned blocks in the Simulink model (highlighted in orange).

```
STO = sITunable('rct_concorde',{'Ki','Kp','Kq','Kf','RollOff'});
```

This automatically picks all linearization points in the model as analysis I/Os.

```
STO.IOs
```

```
ans =
```

```
'rct_concorde/Nzc/1[Nzc]' 'input'
```

```

'rct_concorde/noise/1[n]'           'input'
'rct_concorde/w/1[w]'             'input'
'rct_concorde/Demux/1[Nz]'        'output'
'rct_concorde/Sum2/1[e]'          'output'
'rct_concorde/Sum5/1[delta_m]'    'output'

```

This also parameterizes each tuned block and initializes the block parameters based on their values in the Simulink model. Note that the four gains  $K_i, K_p, K_q, K_f$  are initialized to zero in this example. By default the roll-off filter  $F_{ro}(s)$  is parameterized as a generic second-order transfer function. To parameterize it as

$$F_{ro}(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2},$$

create real parameters  $\zeta, \omega_n$ , build the transfer function shown above, and associate it with the Roll10ff block.

```

wn = realp('wn', 3);           % natural frequency
zeta = realp('zeta', 0.8);     % damping
Fro = tf(wn^2, [1 2*zeta*wn wn^2]); % parametric transfer function

```

```
STO.setBlockParam('Roll10ff', Fro) % use Fro to parameterize "Roll10ff" block
```

### Design Requirements

The autopilot must be tuned to satisfy three main design requirements:

1. **Setpoint tracking:** The response  $N_z$  to the command  $N_{ze}$  should closely match the response of the reference model:

$$G_{ref}(s) = \frac{1.7^2}{s^2 + 2 \times 0.7 \times 1.7s + 1.7^2}.$$

This reference model specifies a well-damped response with a 2 second settling time.

2. **High-frequency roll-off:** The closed-loop response from the noise signals to  $\delta_m$  should roll off past 8 rad/s with a slope of at least -40 dB/decade.

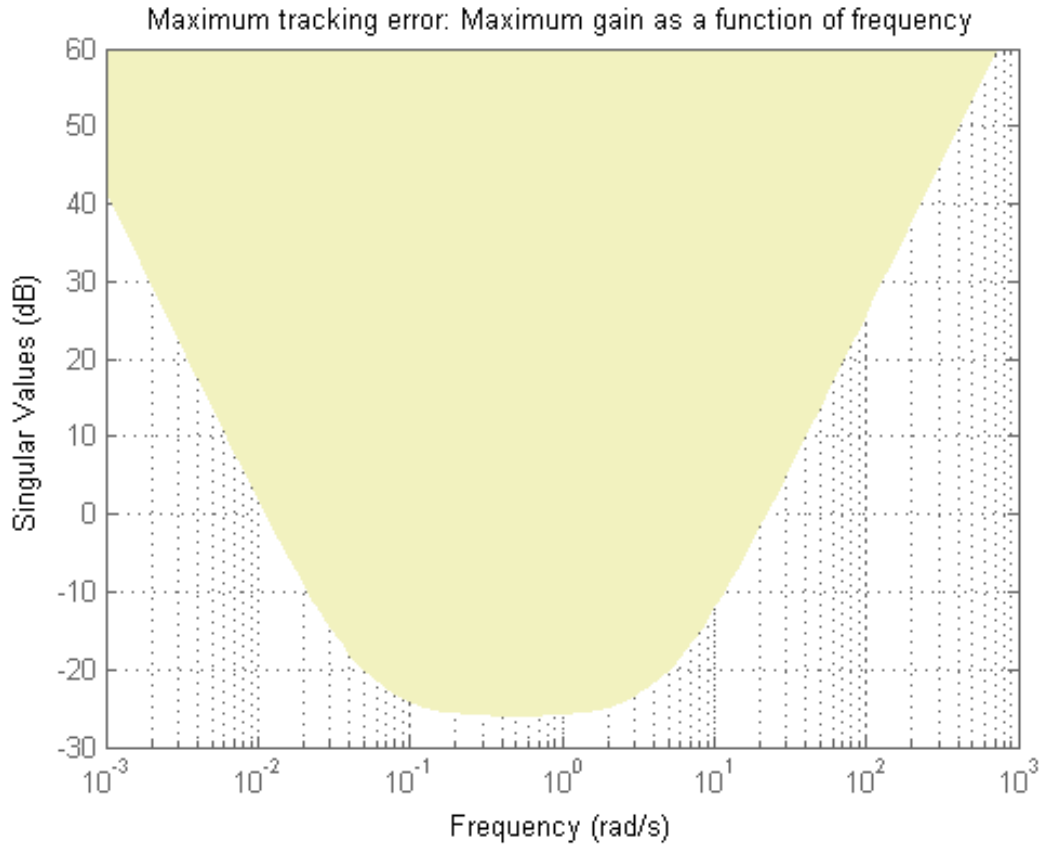
3. **Stability margins:** The stability margins at the plant input  $\delta_m$  should be at least 7 dB and 45 degrees.

For setpoint tracking, we require that the gain of the closed-loop transfer from the command  $N_{zc}$  to the tracking error  $e$  be small in the frequency band [0.05,5] rad/s (recall that we cannot drive the steady-state error to zero because of the plant zero at  $s=0$ ). Using a few frequency points, sketch the maximum tracking error as a function of frequency and use it to limit the gain from  $N_{zc}$  to  $e$ .

```
Freqs = [0.005 0.05 5 50];
Gains = [5 0.05 0.05 5];
Req1 = TuningGoal.Gain('Nzc','e',frd(Gains,Freqs));
Req1.Name = 'Maximum tracking error';
```

The `TuningGoal.Gain` constructor automatically turns the maximum error sketch into a smooth weighting function. Use `viewSpec` to graphically verify the desired error profile.

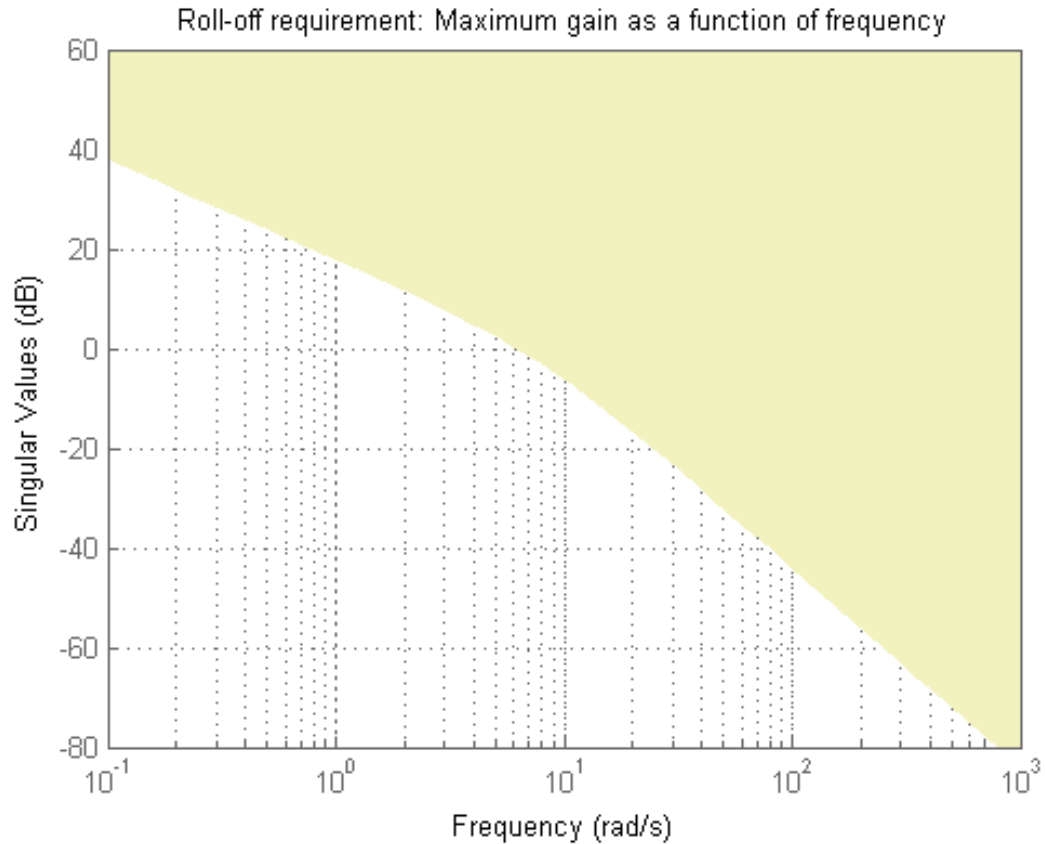
```
viewSpec(Req1)
```



Repeat the same process to limit the high-frequency gain from the noise inputs to  $\delta_m$  and enforce a -40 dB/decade slope in the frequency band from 8 to 800 rad/s

```
Freqs = [0.8 8 800];
Gains = [10 1 1e-4];
Req2 = TuningGoal.Gain('n', 'delta_m', frd(Gains, Freqs));
Req2.Name = 'Roll-off requirement';
```

```
viewSpec(Req2)
```



Finally, register the plant input  $\delta_m$  as a site for open-loop analysis and use `TuningGoal.Margins` to capture the stability margin requirement.

```
STO.addSwitch('delta_m')
```

```
Req3 = TuningGoal.Margins('delta_m',7,45);
```

### Autopilot Tuning

We are now ready to tune the autopilot parameters with `sysune`. This command takes the untuned configuration `ST0` and the three design requirements and returns the tuned version `ST` of `ST0`. All requirements are satisfied when the final value is less than one.

```
[ST,fSoft,~,Info] = sysune(ST0,[Req1 Req2 Req3]);
```

```
Final: Soft = 0.965, Hard = -Inf, Iterations = 57
```

Use `showTunable` to see the tuned block values.

```
ST.showTunable()
```

```
Block "rct_concorde/Ki" =
```

```
  d =
      u1
  y1 -0.03004
```

```
Name: Ki
Static gain.
```

```
-----
```

```
Block "rct_concorde/Kp" =
```

```
  d =
      u1
  y1 -0.009625
```

```
Name: Kp
Static gain.
```

```
-----
```

```
Block "rct_concorde/Kq" =
```

```
  d =
      u1
```



```

        y1 -0.2871

Name: Kq
Static gain.

-----

Block "rct_concorde/Kf" =

    d =
           u1
    y1 -0.02279

Name: Kf
Static gain.

-----

wn = 4.82
-----
zeta = 0.514

To get the tuned value of  $F_{ro}(s)$ , use getBlockValue to evaluate Fro for
the tuned parameter values in ST:

Fro = ST.getBlockValue('RollOff');
tf(Fro)

ans =

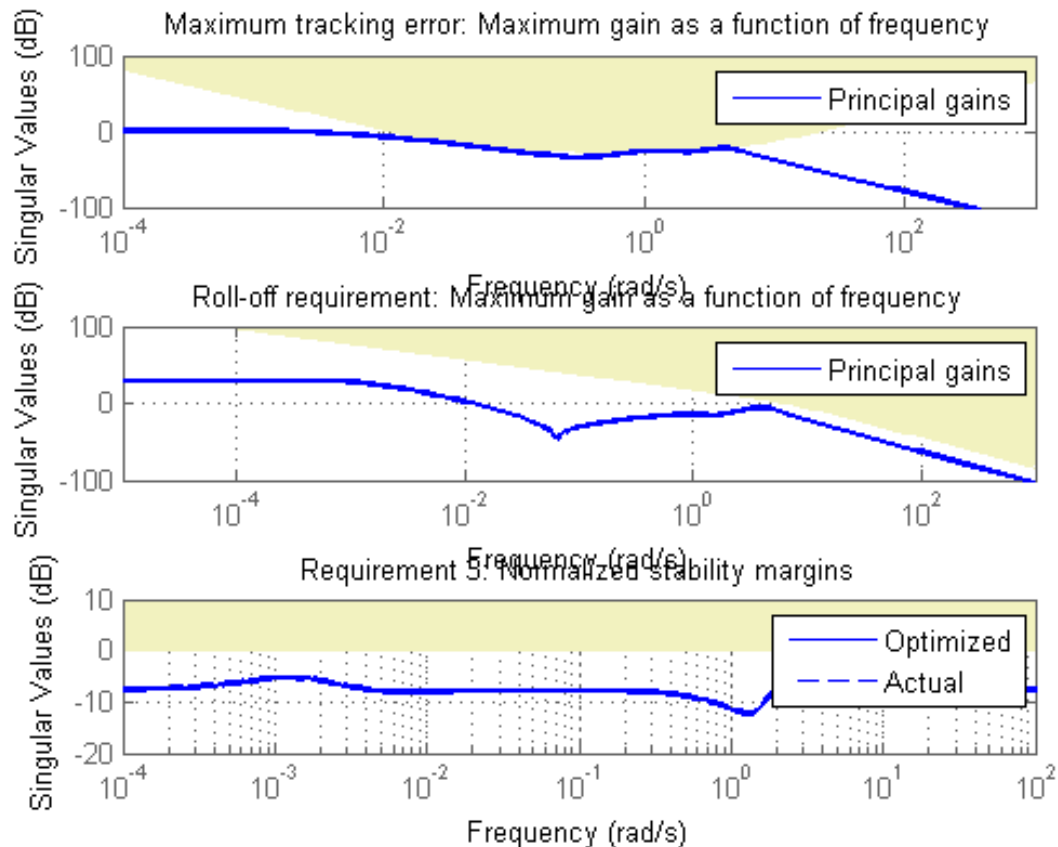
           23.23
-----
    s^2 + 4.952 s + 23.23

Continuous-time transfer function.

```

Finally, use viewSpec to graphically verify that all requirements are satisfied.

```
viewSpec([Req1 Req2 Req3],ST)
```



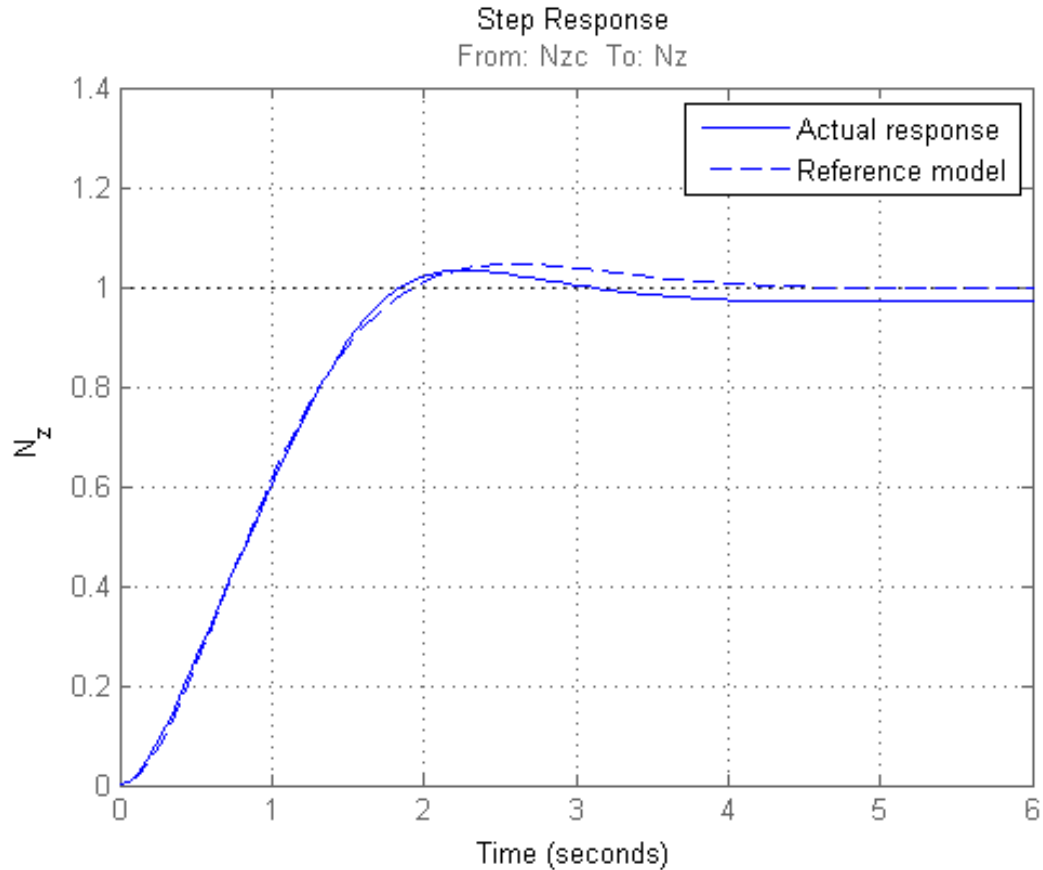
### Closed-Loop Simulations

We now verify that the tuned autopilot satisfies the design requirements. First compare the step response of  $N_z$  with the step response of the reference model  $G_{ref}(s)$ . Again use `getIOTransfer` to compute the tuned closed-loop transfer from  $N_{zc}$  to  $N_z$ :

```
Gref = tf(1.7^2,[1 2*0.7*1.7 1.7^2]); % reference model
```

```
T = ST.getIOTransfer('Nzc','Nz'); % transfer Nzc -> Nz

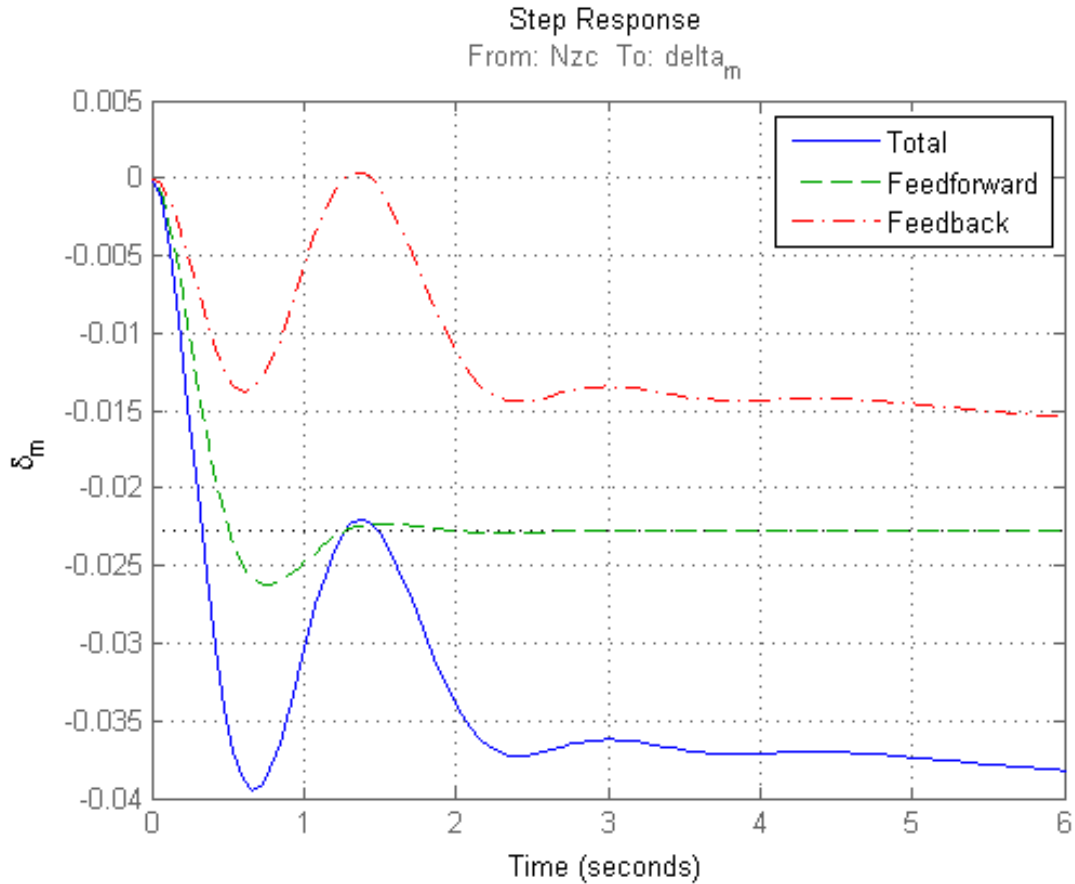
clf, step(T,'b',Gref,'b--',6), grid,
ylabel('N_z'), legend('Actual response','Reference model')
```



Also plot the deflection  $\delta_m$  and the respective contributions of the feedforward and feedback paths:

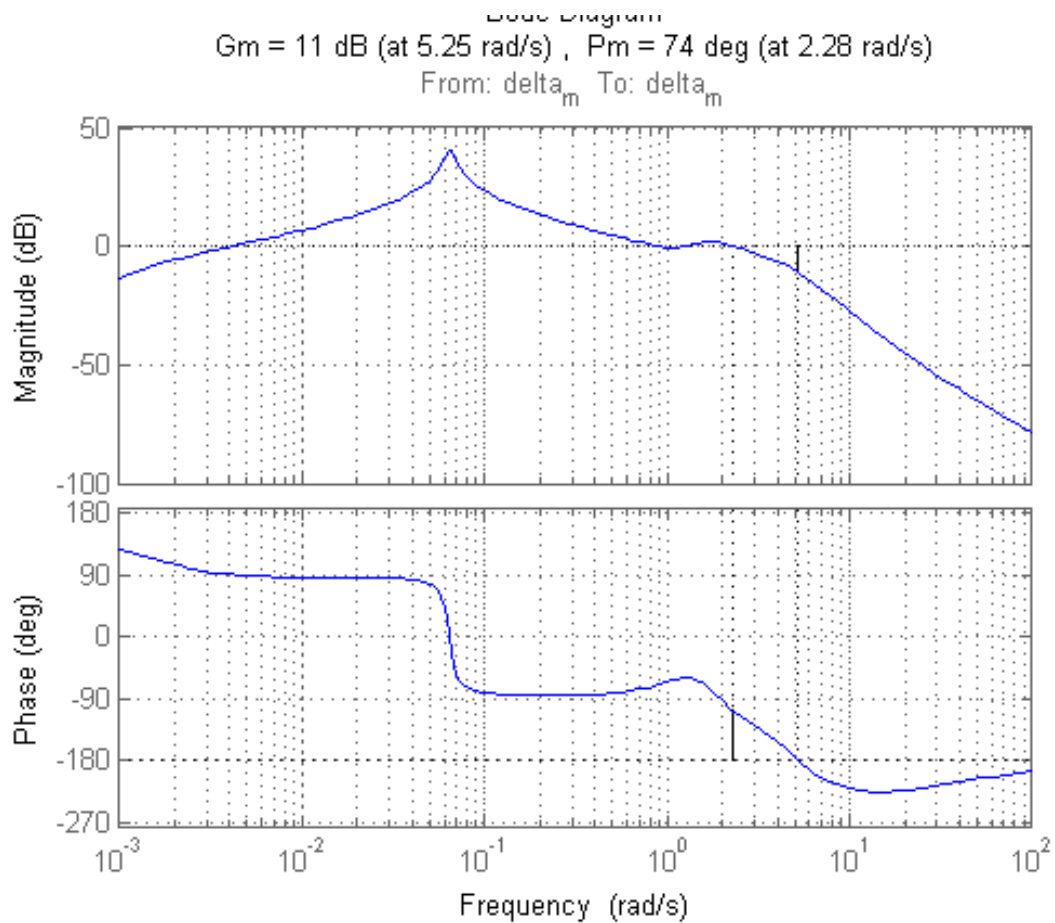
```
T = ST.getIOTransfer('Nzc','delta_m'); % transfer Nzc -> delta_m
Kf = ST.getBlockValue('Kf');           % tuned value of Kf
Tff = Fro*Kf;                          % feedforward contribution to delta_m
```

```
step(T,'b',Tff,'g--',T-Tff,'r-.',6), grid
ylabel('\delta_m'), legend('Total','Feedforward','Feedback')
```



Finally, check the roll-off and stability margin requirements by computing the open-loop response at  $\delta_m$ .

```
OL = ST.getLoopTransfer('delta_m',-1); % negative-feedback loop transfer
margin(OL); grid; set(gca,'XLim',[1e-3,1e2])
```



The Bode plot confirms a roll-off of -40 dB/decade past 8 rad/s and indicates gain and phase margins in excess of 10 dB and 70 degrees.

## Fault-Tolerant Control of a Passenger Jet

This example shows how to tune a fixed-structure controller for multiple operating modes of the plant.

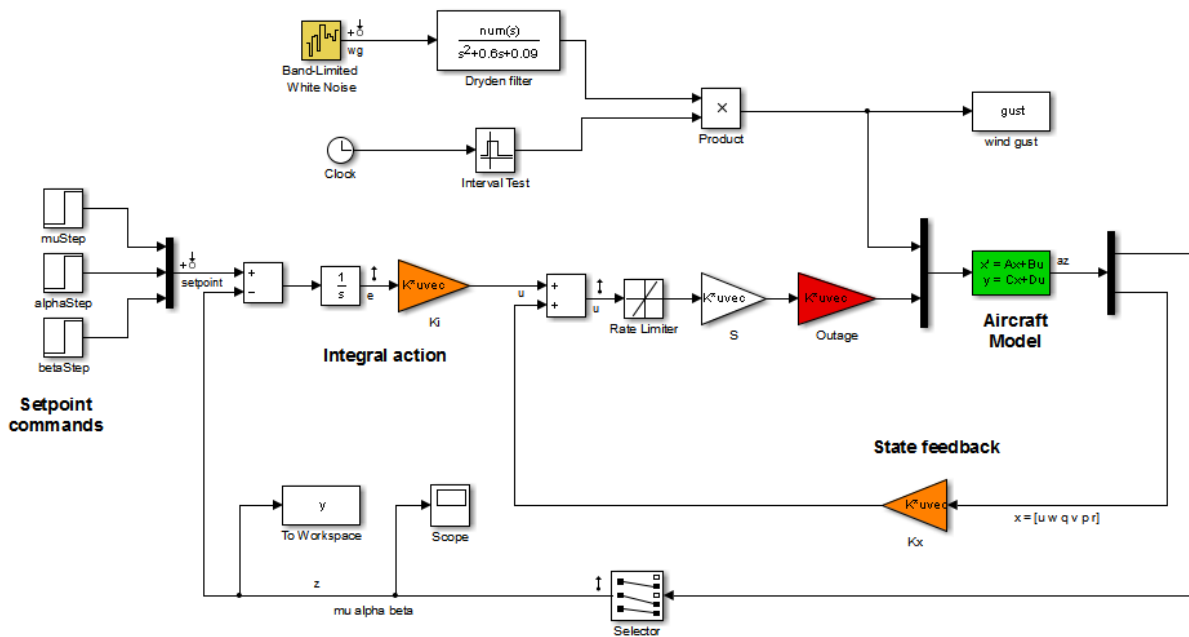
### Background

This example deals with fault-tolerant flight control of passenger jet undergoing outages in the elevator and aileron actuators. The flight control system must maintain stability and meet performance and comfort requirements in both nominal operation and degraded conditions where some actuators are no longer effective due to control surface impairment. Wind gusts must be alleviated in all conditions. This application is sometimes called *reliable control* as aircraft safety must be maintained in extreme flight conditions.

### Aircraft Model

The control system is modeled in Simulink.

```
open_system('faultTolerantAircraft')
```



The aircraft is modeled as a rigid 6th-order state-space system with the following state variables (units are mph for velocities and deg/s for angular rates):

- $u$ : x-body axis velocity
- $w$ : z-body axis velocity
- $q$ : pitch rate
- $v$ : y-body axis velocity
- $p$ : roll rate
- $r$ : yaw rate

The state vector is available for control as well as the flight-path bank angle rate  $\mu$  (deg/s), the angle of attack  $\alpha$  (deg), and the sideslip angle  $\beta$  (deg). The control inputs are the deflections of the right elevator, left elevator, right aileron, left aileron, and rudder. All deflections are in degrees. Elevators are grouped symmetrically to generate the angle of attack. Ailerons are

grouped anti-symmetrically to generate roll motion. This leads to 3 control actions as shown in the Simulink model.

The controller consists of state-feedback control in the inner loop and MIMO integral action in the outer loop. The gain matrices  $K_i$  and  $K_x$  are 3-by-3 and 3-by-6, respectively, so the controller has 27 tunable parameters.

### **Actuator Failures**

We use a 9x5 matrix to encode the nominal mode and various actuator failure modes. Each row corresponds to one flight condition, a zero indicating outage of the corresponding deflection surface.

```
OutageCases = [...  
    1 1 1 1 1; ... % nominal operational mode  
    0 1 1 1 1; ... % right elevator outage  
    1 0 1 1 1; ... % left elevator outage  
    1 1 0 1 1; ... % right aileron outage  
    1 1 1 0 1; ... % left aileron outage  
    1 0 0 1 1; ... % left elevator and right aileron outage  
    0 1 0 1 1; ... % right elevator and right aileron outage  
    0 1 1 0 1; ... % right elevator and left aileron outage  
    1 0 1 0 1; ... % left elevator and left aileron outage  
];
```

### **Design Requirements**

The controller should:

- 1** Provide good tracking performance in  $\mu$ ,  $\alpha$ , and  $\beta$  in nominal operating mode with adequate decoupling of the three axes
- 2** Maintain performance in the presence of wind gust of 10 mph
- 3** Limit stability and performance degradation in the face of actuator outage.

To express the first requirement, you can use an LQG-like cost function that penalizes the integrated tracking error  $e$  and the control effort  $u$ :



$$J = \lim_{T \rightarrow \infty} E \left( \frac{1}{T} \int_0^T \|W_e e\|^2 + \|W_u u\|^2 dt \right).$$

The diagonal weights  $W_e$  and  $W_u$  are the main tuning knobs for trading responsiveness and control effort and emphasizing some channels over others. Use the `WeightedVariance` requirement to express this cost function, and relax the performance weight  $W_e$  by a factor 2 for the outage scenarios.

```
We = diag([10 20 15]); Wu = eye(3);
```

```
% Nominal tracking requirement
```

```
SoftNom = TuningGoal.WeightedVariance('setpoint',{ 'e','u'}, blkdiag(We,Wu),  
SoftNom.Models = 1; % nominal model
```

```
% Tracking requirement for outage conditions
```

```
SoftOut = TuningGoal.WeightedVariance('setpoint',{ 'e','u'}, blkdiag(We/2,Wu),  
SoftOut.Models = 2:9; % outage scenarios
```

For wind gust alleviation, limit the variance of the error signal  $e$  due to the white noise  $wg$  driving the wind gust model. Again use a less stringent requirement for the outage scenarios.

```
% Nominal gust alleviation requirement
```

```
HardNom = TuningGoal.Variance('wg','e',0.02);  
HardNom.Models = 1;
```

```
% Gust alleviation requirement for outage conditions
```

```
HardOut = TuningGoal.Variance('wg','e',0.1);  
HardOut.Models = 2:9;
```

### **Controller Tuning for Nominal Flight**

Set the wind gust speed to 10 mph and initialize the tunable state-feedback and integrators gains of the controller.

```
GustSpeed = 10;  
Ki = eye(3);  
Kx = zeros(3,6);
```

Use the `slTunable` interface to acquire a closed-loop model for each of the nine flight conditions. The resulting generalized state-space array `T0` contains nine tunable models parameterized by the gains `Ki`, `Kx`.

```
clear T0
for k = 9:-1:1
    outage = OutageCases(k,:) ;
    ST = slTunable('faultTolerantAircraft',{'Ki','Kx'}) ;
    T0(:,:,k) = ST.getIOTransfer({'setpoint';'wg'},{'e';'u'}) ;
end
```

`T0`

`T0 =`

```
9x1 array of generalized continuous-time state-space models.
Each model has 6 outputs, 4 inputs, 11 states, and the following blocks:
  Ki: Parametric 3x3 gain, 1 occurrences.
  Kx: Parametric 3x6 gain, 1 occurrences.
```

Type "`ss(T0)`" to see the current value, "`get(T0)`" to see all properties, and

Use `system` to tune the controller gains for the first model (nominal flight condition) subject to the nominal requirements. Treat the wind gust alleviation as a hard constraint.

```
[T,fSoft,gHard] = systune(T0(:,:,1),SoftNom,HardNom);
```

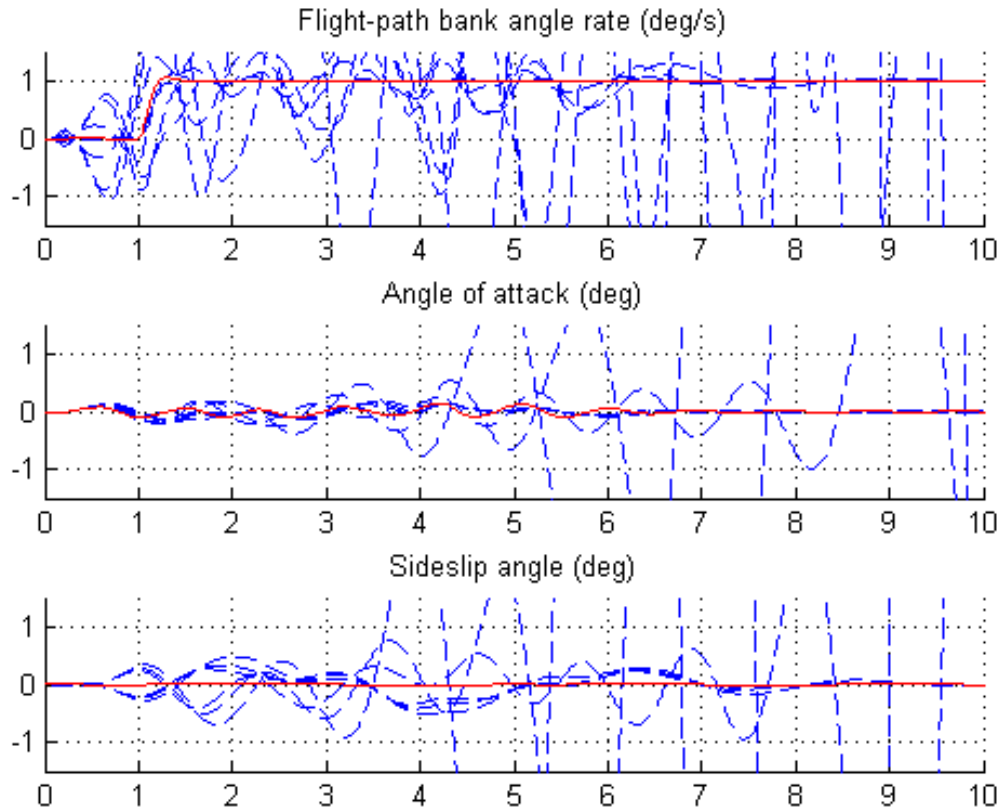
```
Final: Soft = 22.6, Hard = 0.99884, Iterations = 275
```

Retrieve the gain values and simulate the responses to step commands in `mu`, `alpha`, `beta` for the nominal and degraded flight conditions. All simulations include wind gust effects, and the red curve is the nominal response.

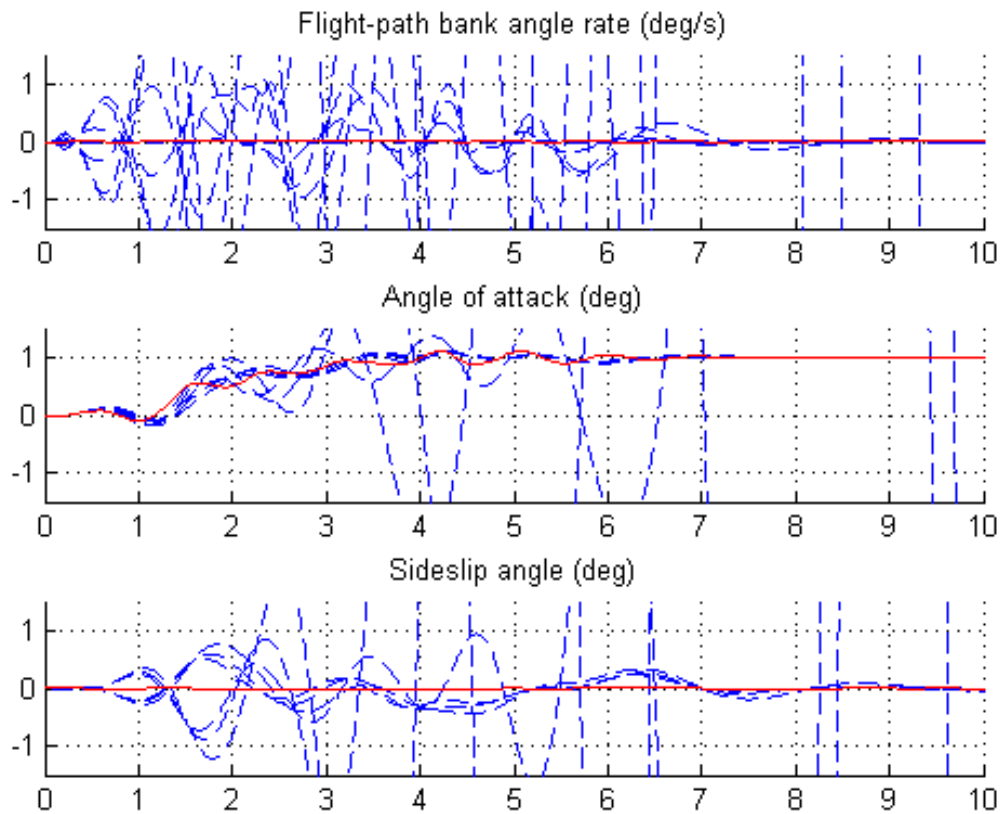
```
Ki = getBlockValue(T, 'Ki'); Ki = Ki.d;
Kx = getBlockValue(T, 'Kx'); Kx = Kx.d;
```

```
% Bank-angle setpoint simulation
```

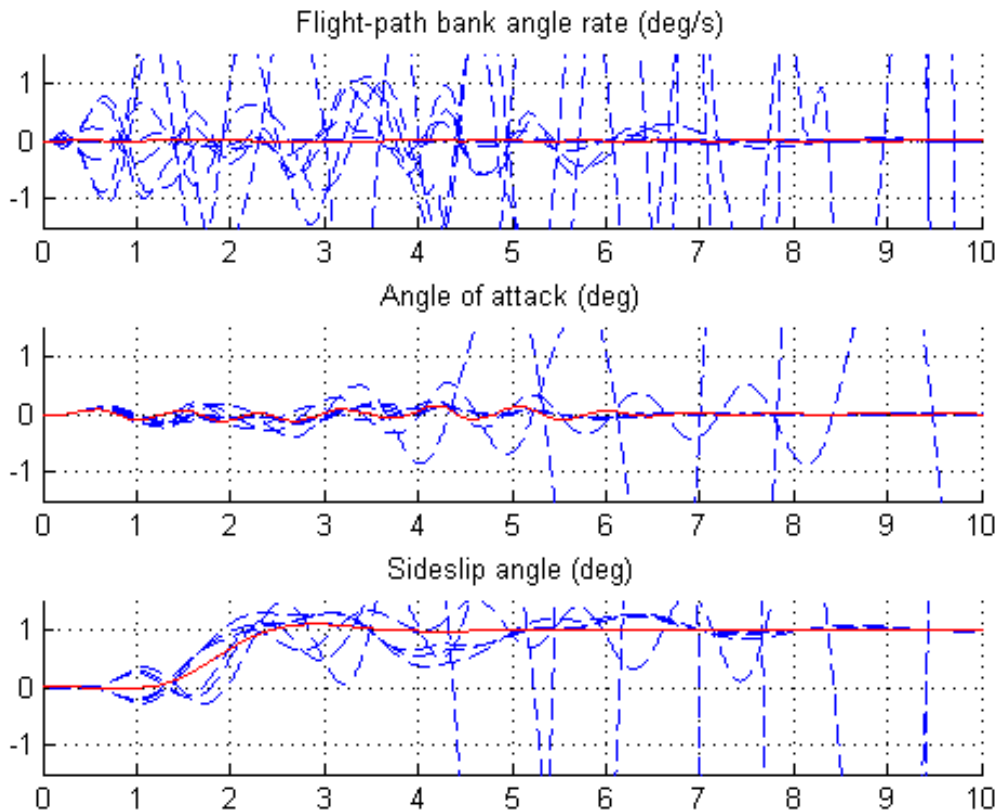
```
plotResponses(OutageCases,1,0,0);
```



```
% Angle-of-attack setpoint simulation  
plotResponses(OutageCases,0,1,0);
```



```
% Sideslip-angle setpoint simulation
plotResponses(OutageCases,0,0,1);
```



The nominal responses are good but the deterioration in performance is unacceptable when faced with actuator outage.

### Controller Tuning for Impaired Flight

To improve reliability, retune the controller gains to meet the nominal requirement for the nominal plant (first model) as well as the relaxed requirements for all eight outage scenarios. Note that `sys tune` now takes an array of tunable models and tunes the gains against this set of plant models.

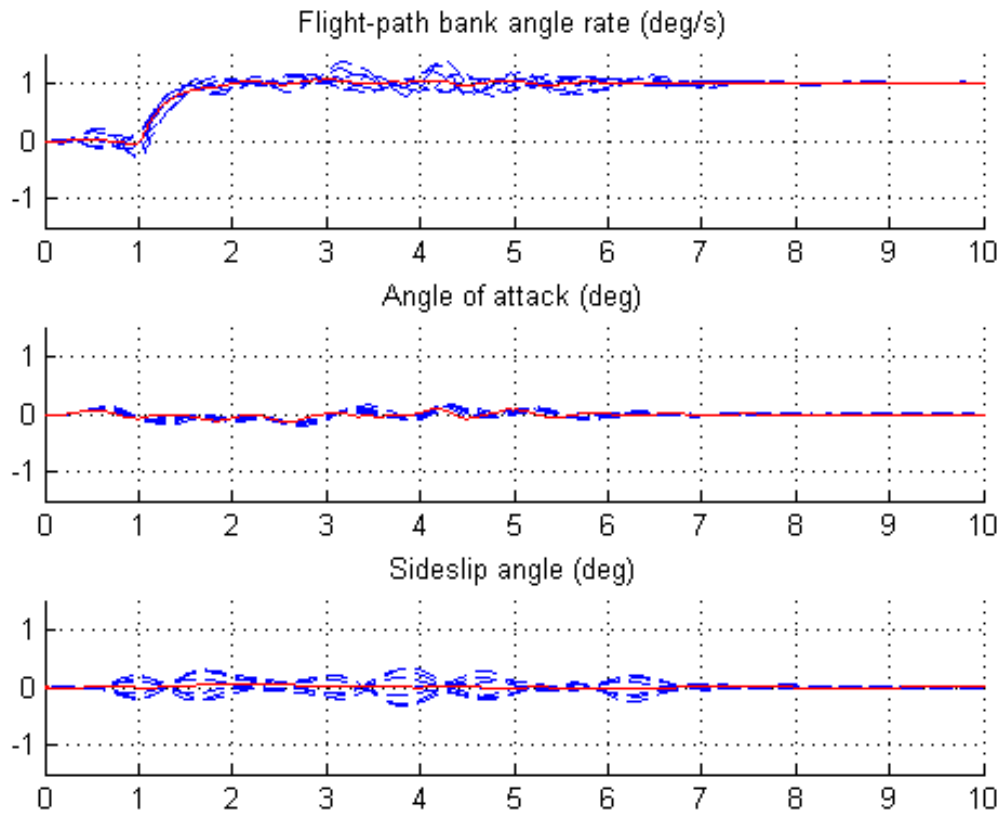
```
[T,fSoft,gHard] = systune(T0,[SoftNom;SoftOut],[HardNom;HardOut]);
```

```
Final: Soft = 25.8, Hard = 0.99946, Iterations = 492
```

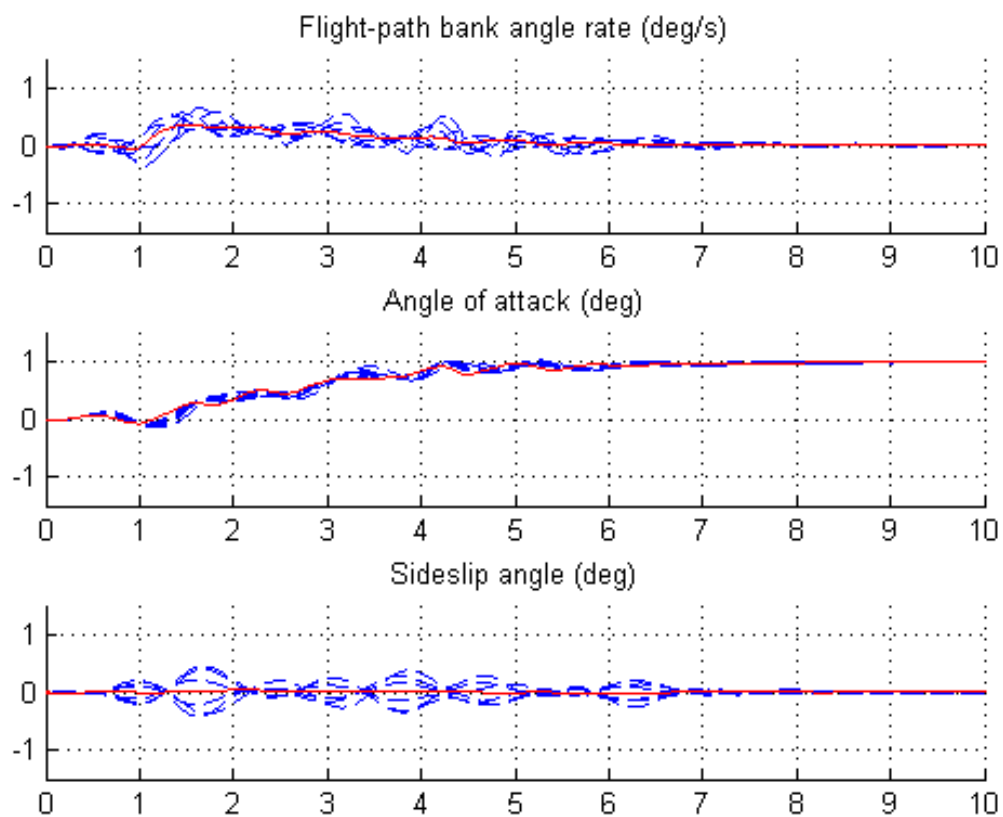
The optimal performance (square root of LQG cost  $J$ ) is only slightly worse than for the nominal tuning (26 vs. 23). Retrieve the gain values and rerun the simulations (red curve is the nominal response).

```
Ki = getBlockValue(T, 'Ki'); Ki = Ki.d;  
Kx = getBlockValue(T, 'Kx'); Kx = Kx.d;
```

```
% Bank-angle setpoint simulation  
plotResponses(OutageCases,1,0,0);
```

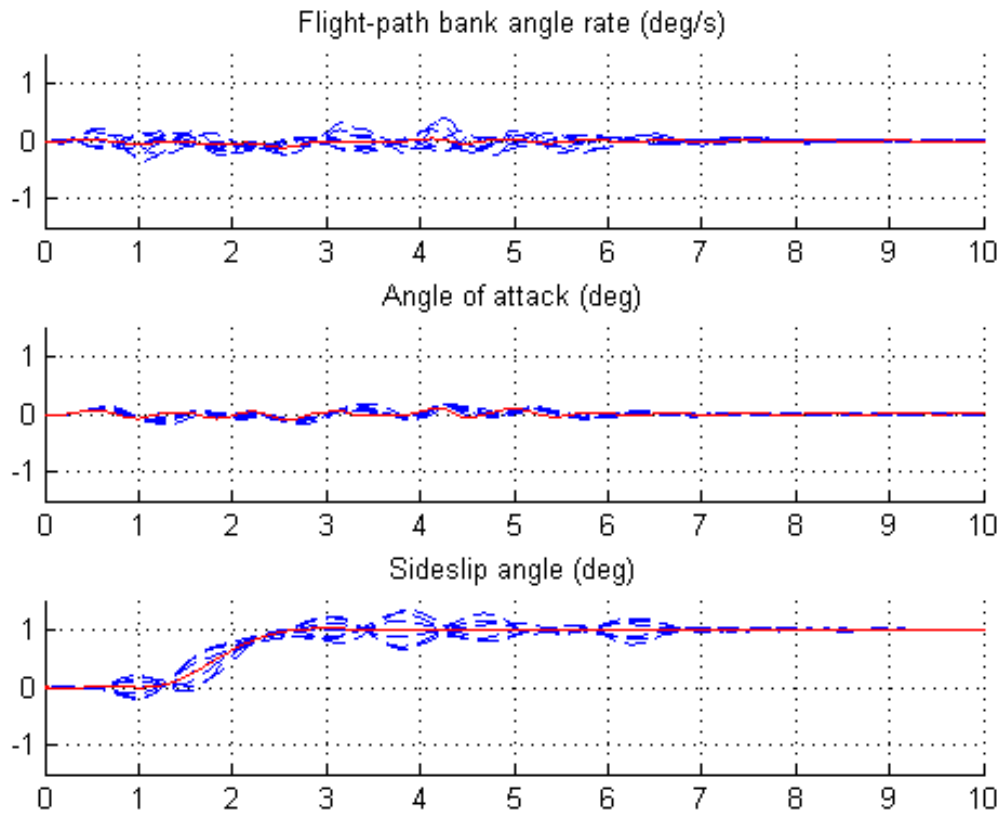


```
% Angle-of-attack setpoint simulation  
plotResponses(OutageCases,0,1,0);
```



```
% Sideslip-angle setpoint simulation  
plotResponses(OutageCases,0,0,1);
```





The controller now provides acceptable performance for all outage scenarios considered in this example. The design could be further refined by adding specifications such as minimum stability margins and gain limits to avoid actuator rate saturation.

## Fixed-Structure H-infinity Synthesis with HINFSTRUCT

This example uses the `hinfstruct` command to tune a fixed-structure controller subject to  $H_\infty$  constraints.

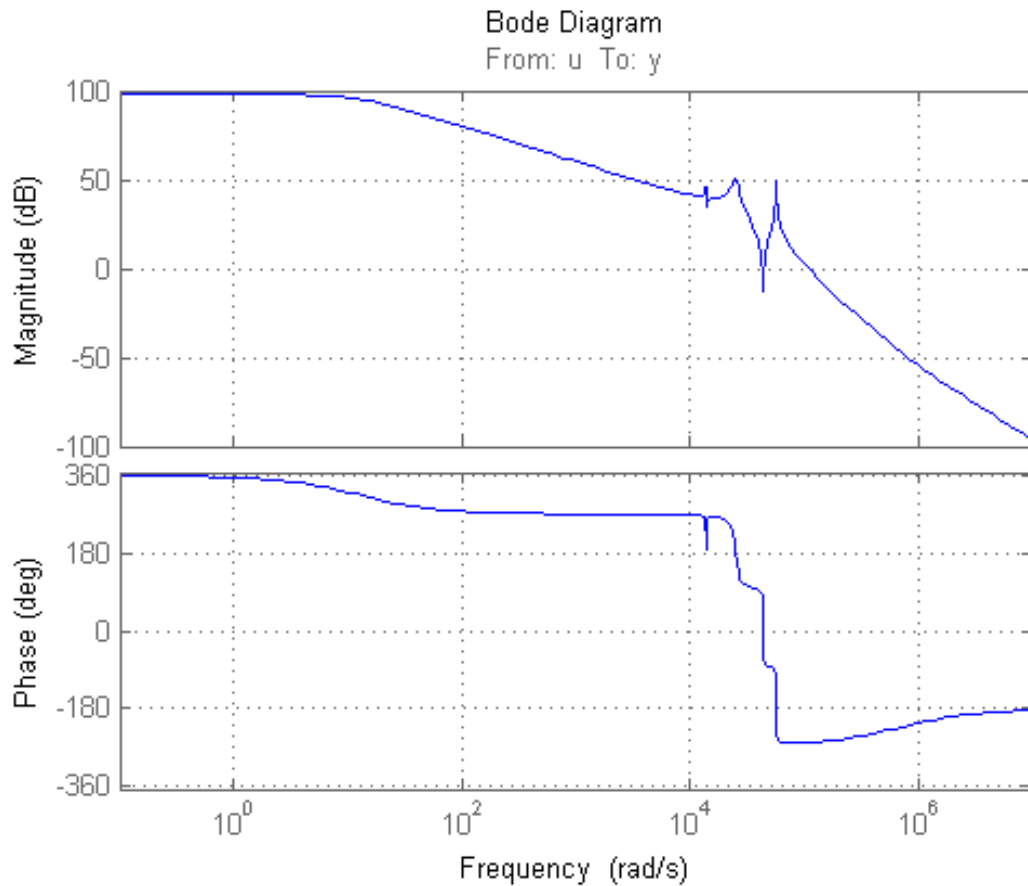
### Introduction

The `hinfstruct` command extends classical  $H_\infty$  synthesis (see `hifsyn`) to fixed-structure control systems. This command is meant for users already comfortable with the `hifsyn` workflow. If you are unfamiliar with  $H_\infty$  synthesis or find augmented plants and weighting functions intimidating, use `sysstune` and `looptune` instead. See "*Tuning Control Systems with SYSTUNE*" for the `sysstune` counterpart of this example.

### Plant Model

This example uses a 9th-order model of the head-disk assembly (HDA) in a hard-disk drive. This model captures the first few flexible modes in the HDA.

```
load hinfstruct_demo G
bode(G), grid
```



We use the feedback loop shown below to position the head on the correct track. This control structure consists of a PI controller and a low-pass filter in the return path. The head position  $y$  should track a step change  $r$  with a response time of about one millisecond, little or no overshoot, and no steady-state error.

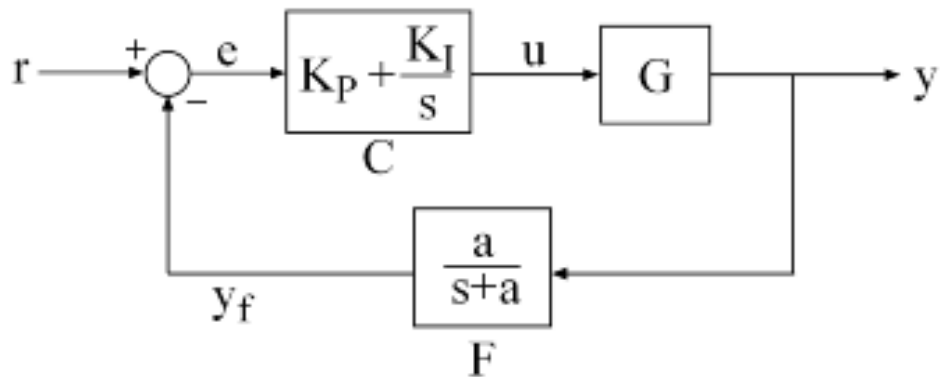


Figure 1: Control Structure

#### Tunable Elements

There are two tunable elements in the control structure of Figure 1: the PI controller  $C(s)$  and the low-pass filter

$$F(s) = \frac{a}{s+a}.$$

Use the `ltiblock.pid` class to parameterize the PI block and specify the filter  $F(s)$  as a transfer function depending on a tunable real parameter  $a$ .

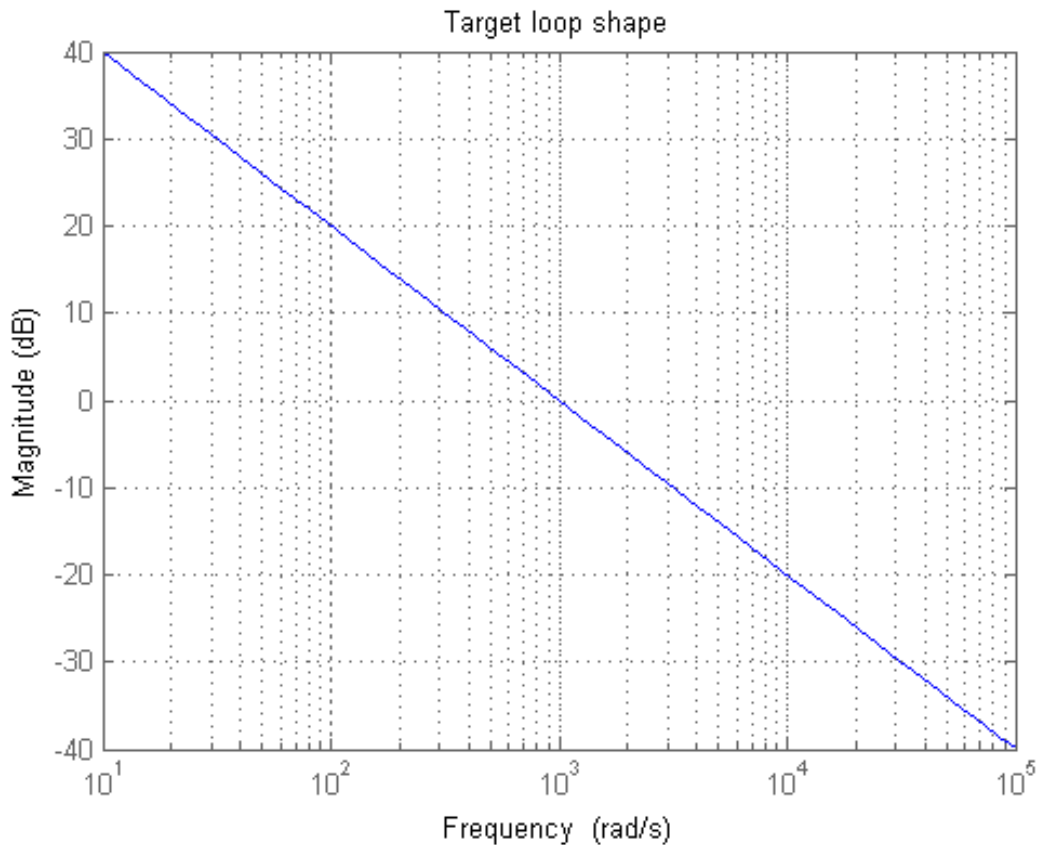
```
C0 = ltiblock.pid('C','pi'); % tunable PI
a = realp('a',1); % filter coefficient
F0 = tf(a,[1 a]); % filter parameterized by a
```

#### Loop Shaping Design

Loop shaping is a frequency-domain technique for enforcing requirements on response speed, control bandwidth, roll-off, and steady state error. The idea is to specify a target gain profile or "loop shape" for the open-loop response  $L(s) = F(s)G(s)C(s)$ . A reasonable loop shape for this application should

have integral action and a crossover frequency of about 1000 rad/s (the reciprocal of the desired response time of 0.001 seconds). This suggests the following loop shape:

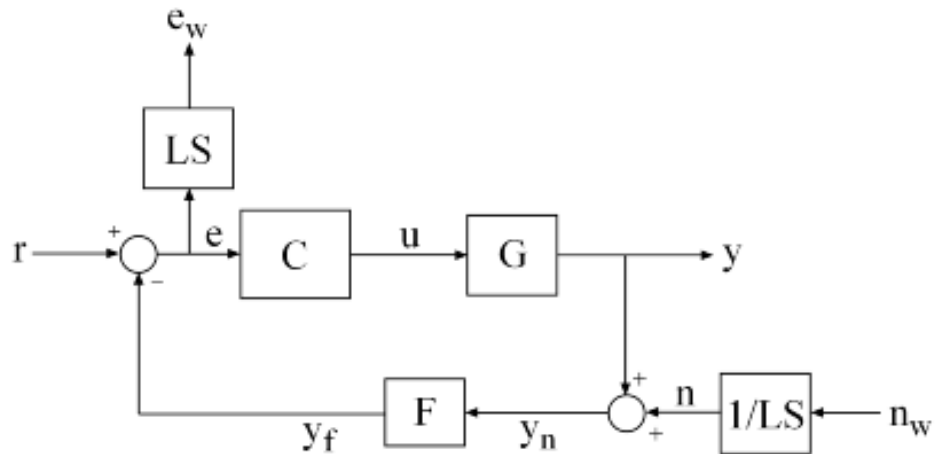
```
wc = 1000; % target crossover
s = tf('s');
LS = (1+0.001*s/wc)/(0.001+s/wc);
bodemag(LS,{1e1,1e5}), grid, title('Target loop shape')
```



Note that we chose a bi-proper, bi-stable realization to avoid technical difficulties with marginally stable poles and improper inverses. In order to

tune  $C(s)$  and  $F(s)$  with `hinfstruct`, we must turn this target loop shape into constraints on the closed-loop gains. A systematic way to go about this is to instrument the feedback loop as follows:

- Add a measurement noise signal  $n$
- Use the target loop shape  $LS$  and its reciprocal to filter the error signal  $e$  and the white noise source  $n_w$ .



**Figure 2: Closed-Loop Formulation**

If  $T(s)$  denotes the closed-loop transfer function from  $(r, n_w)$  to  $(y, e_w)$ , the gain constraint

$$\|T\|_{\infty} < 1$$

secures the following desirable properties:

- At low frequency ( $\omega < \omega_c$ ), the open-loop gain stays above the gain specified by the target loop shape LS
- At high frequency ( $\omega > \omega_c$ ), the open-loop gain stays below the gain specified by LS
- The closed-loop system has adequate stability margins
- The closed-loop step response has small overshoot.

We can therefore focus on tuning  $C(s)$  and  $F(s)$  to enforce  $\|T\|_\infty < 1$ .

### Specifying the Control Structure in MATLAB

In MATLAB, you can use the connect command to model  $T(s)$  by connecting the fixed and tunable components according to the block diagram of Figure 2:

```
% Label the block I/Os
Wn = 1/LS;  Wn.u = 'nw';  Wn.y = 'n';
We = LS;    We.u = 'e';   We.y = 'ew';
CO.u = 'e';  CO.y = 'u';
FO.u = 'yn'; FO.y = 'yf';

% Specify summing junctions
Sum1 = sumblk('e = r - yf');
Sum2 = sumblk('yn = y + n');

% Connect the blocks together
T0 = connect(G,Wn,We,CO,FO,Sum1,Sum2,{'r','nw'},{'y','ew'});
```

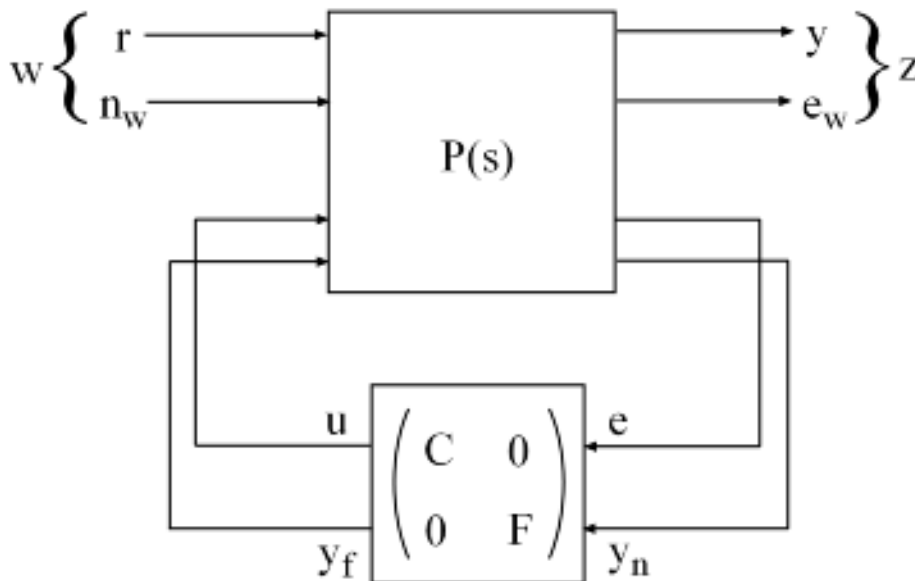
These commands construct a generalized state-space model T0 of  $T(s)$ . This model depends on the tunable blocks C and a:

T0.Blocks

```
ans =

      C: [1x1 ltiblock.pid]
      a: [1x1 realp]
```

Note that  $T_0$  captures the following "Standard Form" of the block diagram of Figure 2 where the tunable components  $C, F$  are separated from the fixed dynamics.



**Figure 3: Standard Form for Disk-Drive Loop Shaping**

#### **Tuning the Controller Gains**

We are now ready to use `hinfstruct` to tune the PI controller  $C$  and filter  $F$  for the control architecture of Figure 1. To mitigate the risk of local minima, run three optimizations, two of which are started from randomized initial values for  $C_0$  and  $F_0$ :

```
rng('default')
opt = hinfstructOptions('Display','final','RandomStart',5);
T = hinfstruct(T0,opt);
```

Final: Peak gain = 3.91, Iterations = 125



```

Final: Peak gain = 1.56, Iterations = 96
Final: Peak gain = 597, Iterations = 192
      Min decay rate 1.05e-07 is close to lower bound 1e-07
Final: Peak gain = 1.56, Iterations = 146
Final: Peak gain = 1.56, Iterations = 99
Final: Peak gain = 3.88, Iterations = 63

```

The best closed-loop gain is 1.56, so the constraint  $\|T\|_{\infty} < 1$  is nearly satisfied. The `hinfstruct` command returns the tuned closed-loop transfer  $T(s)$ . Use `showTunable` to see the tuned values of  $C$  and the filter coefficient  $a$ :

```
showTunable(T)
```

```
C =
```

$$K_p + K_i * \frac{1}{s}$$

```
with Kp = 0.000846, Ki = 0.0103
```

```
Name: C
```

```
Continuous-time PI controller in parallel form.
```

```
-----
```

```
a = 5.49e+03
```

Use `getBlockValue` to get the tuned value of  $C(s)$  and use `getValue` to evaluate the filter  $F(s)$  for the tuned value of  $a$ :

```
C = getBlockValue(T,'C');
```

```
F = getValue(F0,T.Blocks); % propagate tuned parameters from T to F
```

```
tf(F)
```

```
ans =
```

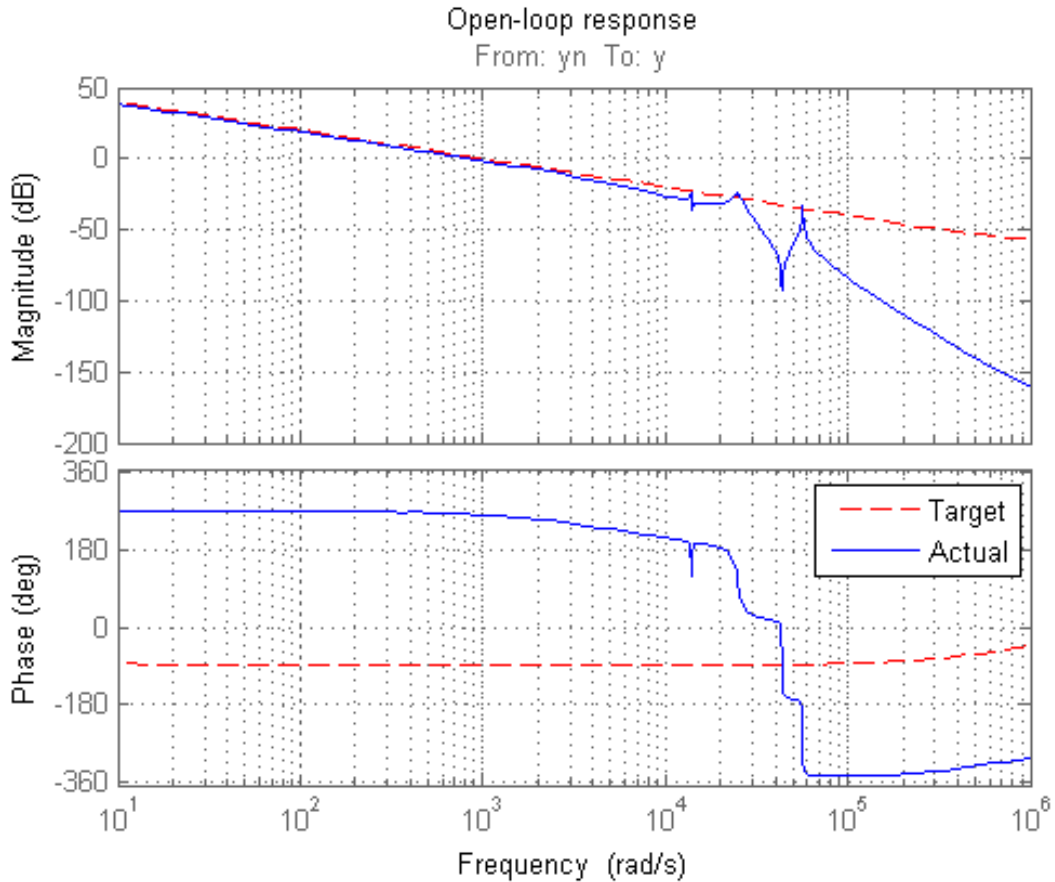
```
From input "yn" to output "yf":
5486
```

```
-----
s + 5486
```

Continuous-time transfer function.

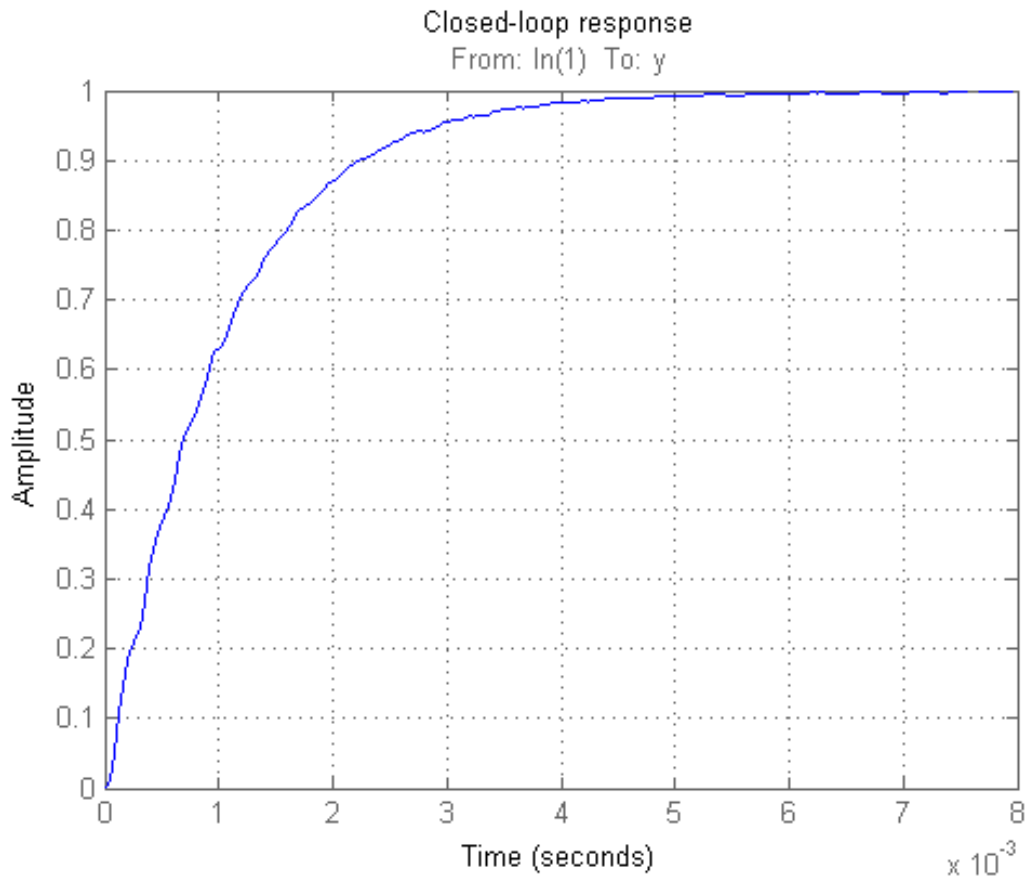
To validate the design, plot the open-loop response  $L=F*G*C$  and compare with the target loop shape LS:

```
bode(LS,'r--',G*C*F,'b',{1e1,1e6}), grid,
title('Open-loop response'), legend('Target','Actual')
```



The 0dB crossover frequency and overall loop shape are as expected. The stability margins can be read off the plot by right-clicking and selecting the **Characteristics** menu. This design has 24dB gain margin and 81 degrees phase margin. Plot the closed-loop step response from reference  $r$  to position  $y$ :

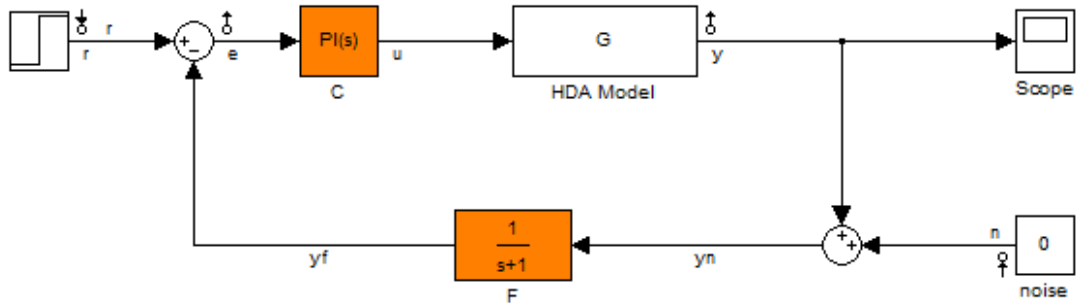
```
step(feedback(G*C,F)), grid, title('Closed-loop response')
```



While the response has no overshoot, there is some residual wobble due to the first resonant peaks in  $G$ . You might consider adding a notch filter in the forward path to remove the influence of these modes.

### Tuning the Controller Gains from Simulink

Suppose you used this Simulink model to represent the control structure. If you have Simulink Control Design installed, you can tune the controller gains from this Simulink model as follows. First mark the signals  $r$ ,  $e$ ,  $y$ ,  $n$  as linearization I/Os:



See `hinfstruct_demo` to see how you can tune the PI gains and the filter coefficient with the `HINFSTRUCT` command.

Copyright 2004-2010 The MathWorks, Inc.

Then create an instance of the `sITunable` interface and mark the Simulink blocks `C` and `F` as tunable:

```
ST = sITunable('rct_diskdrive',{ 'C', 'F' });
```

Since the filter  $F(s)$  has a special structure, explicitly specify how to parameterize the `F` block:

```
a = realp('a',1); % filter coefficient
ST.setBlockParam('F',tf(a,[1 a]));
```

Finally, use the `getIOTransfer` method to derive a tunable model of the closed-loop transfer function  $T(s)$  (see Figure 2)

```
% Tunable model of closed-loop transfer (r,n) -> (y,e)
slT0 = ST.getIOTransfer({'r','n'},{'y','e'});
```

```
% Add weighting functions in n and e channels
slT0 = blkdiag(1,LS) * slT0 * blkdiag(1,1/LS);
```

You are now ready to tune the controller gains with `hinfstruct`:

```
rng(0)
opt = hinfstructOptions('Display','final','RandomStart',5);
slT = hinfstruct(slT0,opt);
```

```
Final: Peak gain = 3.92, Iterations = 119
Final: Peak gain = 1.56, Iterations = 99
Final: Peak gain = 597, Iterations = 188
      Min decay rate 1e-07 is close to lower bound 1e-07
Final: Peak gain = 3.88, Iterations = 99
Final: Peak gain = 1.56, Iterations = 100
Final: Peak gain = 3.88, Iterations = 58
```

Verify that you obtain the same tuned values as with the MATLAB approach:

```
showTunable(slT)
```

```
C =
```

$$K_p + K_i * \frac{1}{s}$$

```
with Kp = 0.000846, Ki = 0.0103
```

```
Name: C
```

```
Continuous-time PI controller in parallel form.
```

```
-----
```

```
a = 5.49e+03
```

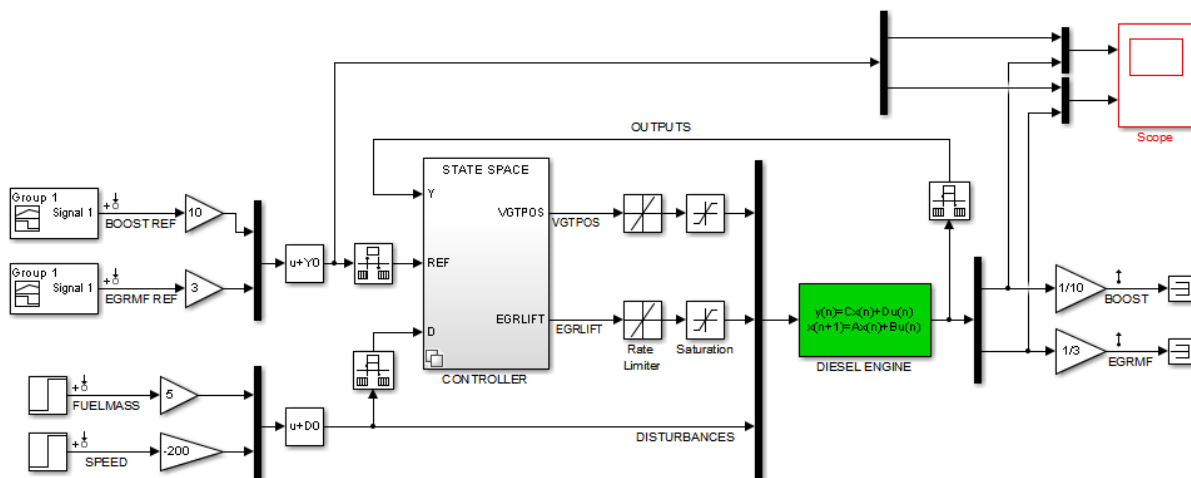
## MIMO Control of Diesel Engine

This example uses `system` to design and tune a MIMO controller for a Diesel engine. The controller is tuned in discrete time for a single operating condition.

### Diesel Engine Model

Modern Diesel engines use a variable geometry turbocharger (VGT) and exhaust gas recirculation (EGR) to reduce emissions. Tight control of the VGT boost pressure and EGR massflow is necessary to meet strict emission targets. This example shows how to design and tune a MIMO controller that regulates these two variables when the engine operates at 2100 rpm with a fuel mass of 12 mg per injection-cylinder.

```
open_system('rct_diesel')
```

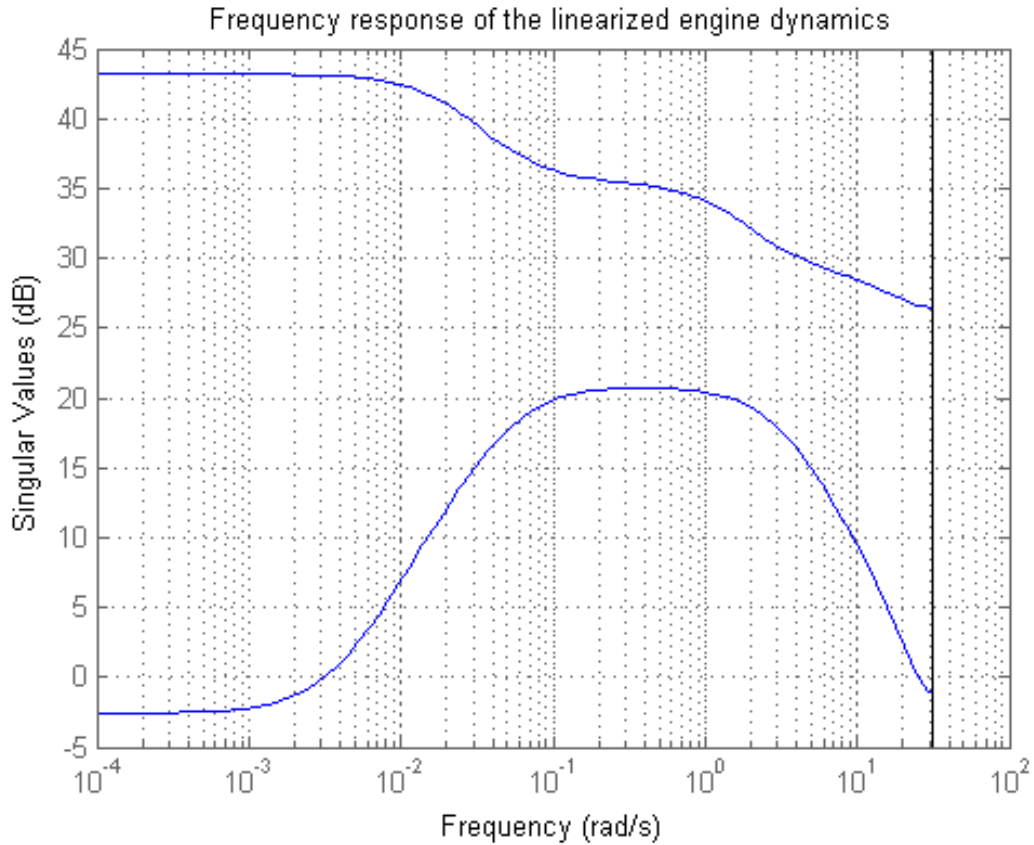


The VGT/EGR control system is modeled in Simulink. The controller adjusts the positions `EGRLIFT` and `VGTPPOS` of the EGR and VGT valves. It has access to the boost pressure and EGR massflow targets and measured values, as well as fuel mass and engine speed measurements. Both valves have rate and saturation limits. The plant model is sampled every 0.1 seconds and the control signals `EGRLIFT` and `VGTPPOS` are refreshed every 0.2 seconds. This

example considers step changes of +10 KPa in boost pressure and +3 g/s in EGR massflow. To better gauge cross-coupling effects, we scale each variable by its absolute variation and **BOOST** and **EGRMF** are therefore normalized boost pressure and EGR massflow responses.

For the operating condition under consideration, we used System Identification to derive a linear model of the engine from experimental data. The frequency response from the manipulated variables **EGRLIFT** and **VGTPPOS** to the controlled variables **BOOST** and **EGR MF** appears below. Note that the plant is ill conditioned at low frequency which makes independent control of boost pressure and EGR massflow difficult.

```
sigma(Plant(:,1:2)), grid  
title('Frequency response of the linearized engine dynamics')
```



**Control Objectives**

There are two main control objectives:

- 1 Respond to step changes in boost pressure and EGR massflow in about 5 seconds with minimum cross coupling
- 2 Be insensitive to (small) variations in speed and fuel mass.

Use a tracking requirement for the first objective.

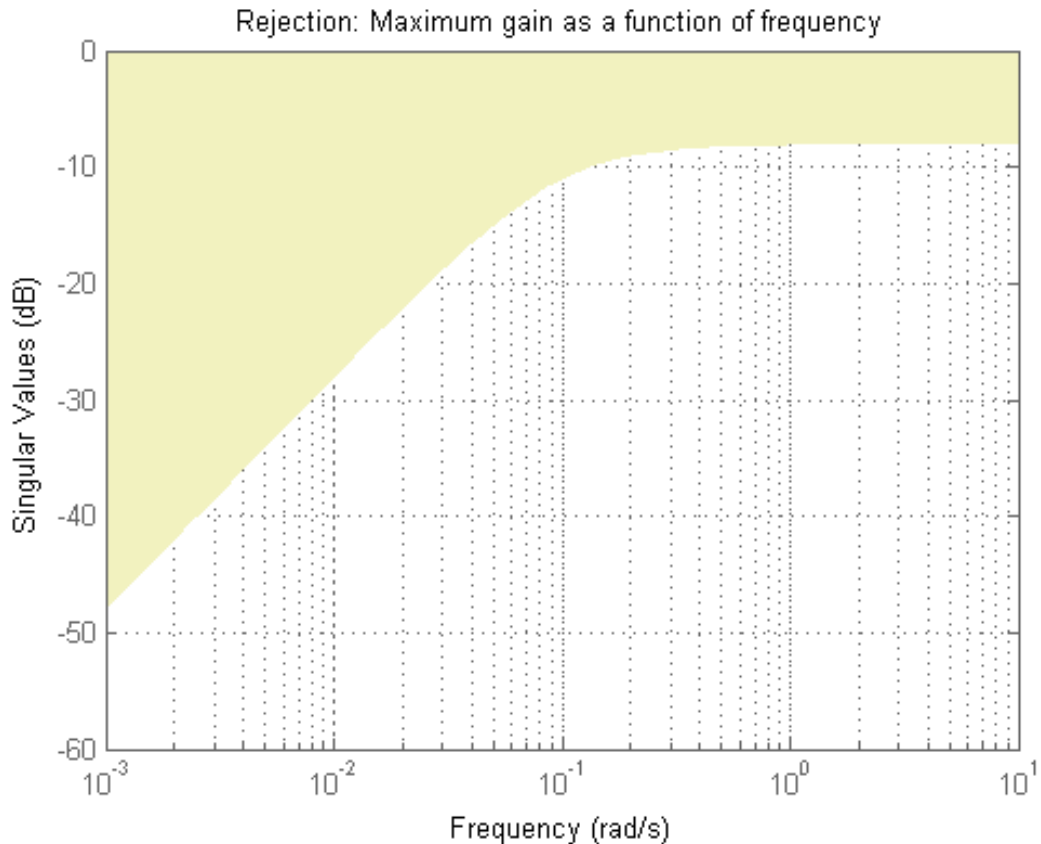


```
% 5-second response time, steady-state error less than 3%  
TR = TuningGoal.Tracking({'BOOST REF'; 'EGRMF REF'}, {'BOOST'; 'EGRMF'}, 5, 3e-2);  
TR.Name = 'Setpoint tracking';
```

For the second objective, treat speed and fuel mass as disturbances and specify that the sensitivity to these disturbances should be small at low frequency and gradually increase as we approach the control bandwidth.

```
s = tf('s');  
DR = TuningGoal.Gain({'FUELMASS'; 'SPEED'}, {'BOOST'; 'EGRMF'}, 0.4*s/(s+0.1));  
DR.Name = 'Rejection';
```

```
viewSpec(DR)
```



To provide adequate robustness to unmodeled dynamics and aliasing, limit the control bandwidth and impose sufficient stability margins at both the plant inputs and outputs. Because we are dealing with a 2-by-2 MIMO feedback loops, these stability margins are interpreted as disk margins (see `loopmargin` and `TuningGoal.Margins` for details).

```
% Roll off of -20 dB/dec past 1 rad/s
RO = TuningGoal.MaxLoopGain({'EGRLIFT', 'VGTPOS'}, 1, 1);
RO.LoopScaling = 'off';
RO.Name = 'Rolloff';
```

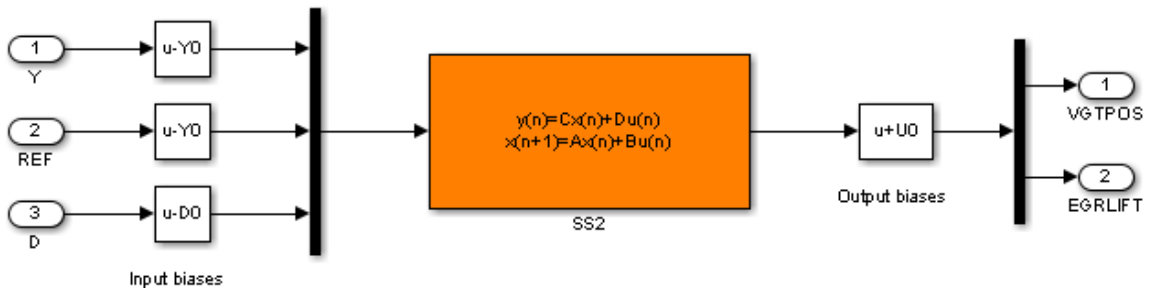
```

% 7 dB of gain margin and 45 degrees of phase margin
M1 = TuningGoal.Margins({'EGRLIFT', 'VGTPPOS'},7,45);
M1.Name = 'Plant input';
M2 = TuningGoal.Margins('DIESEL ENGINE',7,45);
M2.Name = 'Plant output';

```

### Tuning of Blackbox MIMO Controller

Without a-priori knowledge of a suitable control structure, first try "blackbox" state-space controllers of various orders. The plant model has four states, so try a controller of order four or less. Here we tune a second-order controller since the "SS2" block in the Simulink model has two states.



2-DOF MIMO PID with disturbance feedforward

### Figure 1: Second-order blackbox controller.

Use the `sITunable` interface to configure the Simulink model for tuning. Mark the block "SS2" as tunable, register the locations where to assess margins and loop shapes, and specify that linearization and tuning should be performed at the controller sampling rate.

```

ST0 = sITunable('rct_diesel','SS2');
ST0.addSwitch({'CONTROLLER/EGRLIFT','CONTROLLER/VGTPOS','DIESEL ENGINE'})

```

```
ST0.Ts = 0.2;
```

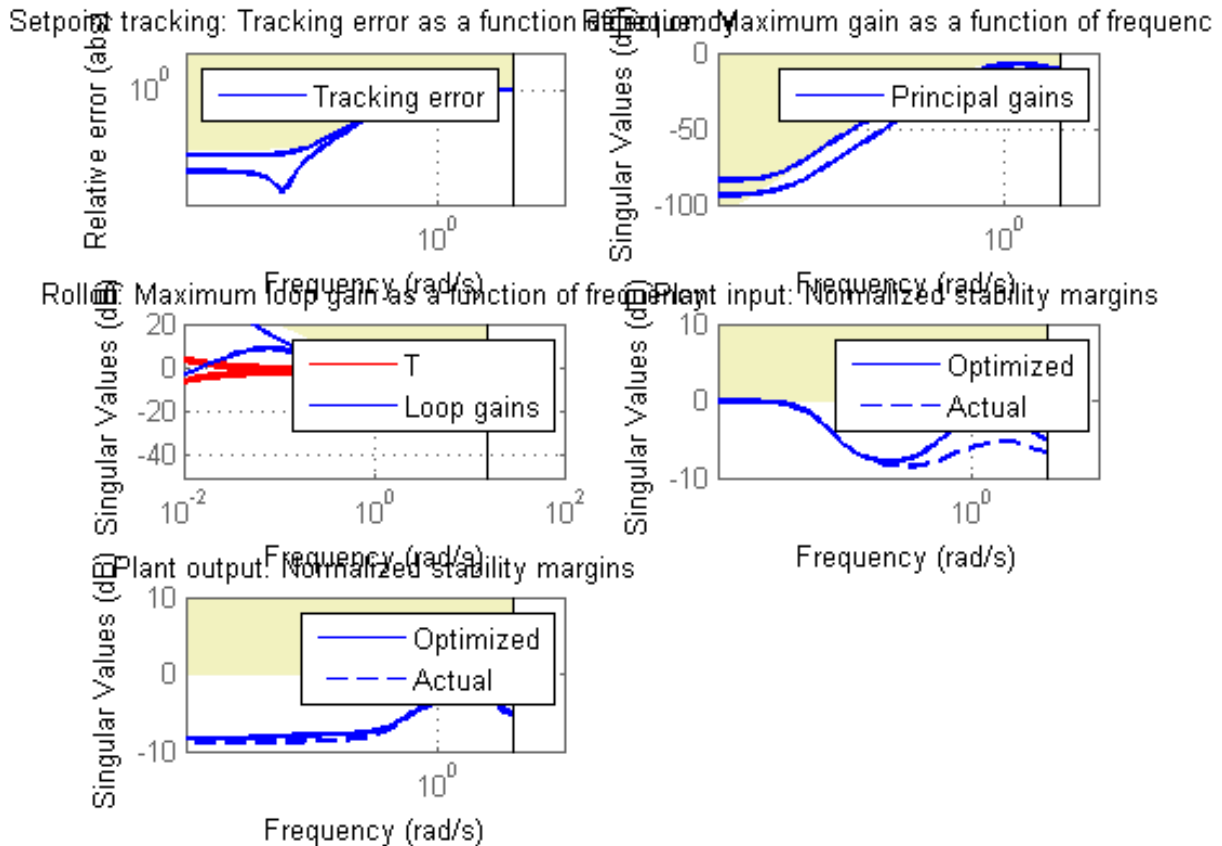
Now use `systemtune` to tune the state-space controller subject to our control objectives. Treat the stability margins and roll-off target as hard constraints and try to best meet the remaining objectives (soft constraints).

```
[ST1,~,~,Info1] = systemtune(ST0,[TR DR],[M1 M2 RO]);
```

```
Final: Soft = 1.05, Hard = 0.99244, Iterations = 649
```

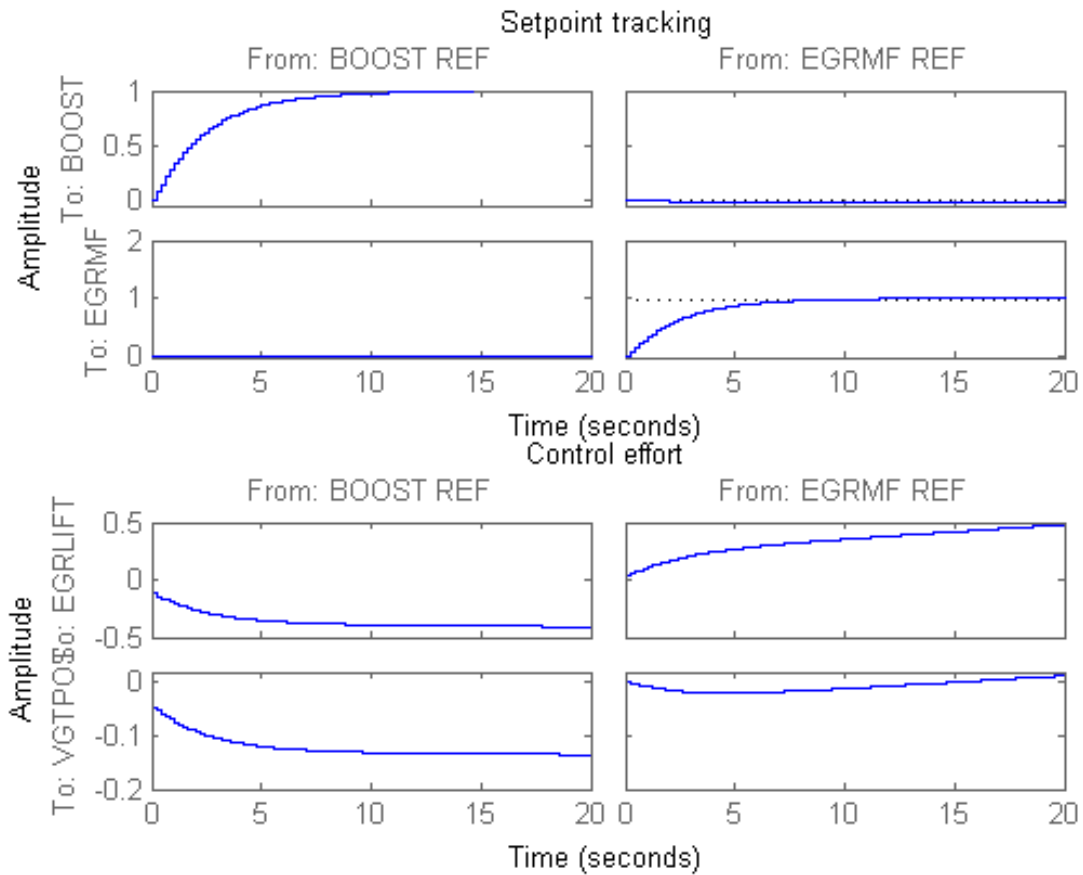
All requirements are nearly met (a requirement is satisfied when its normalized value is less than 1). Verify this graphically.

```
viewSpec([TR DR RO M1 M2],ST1,Info1)
```

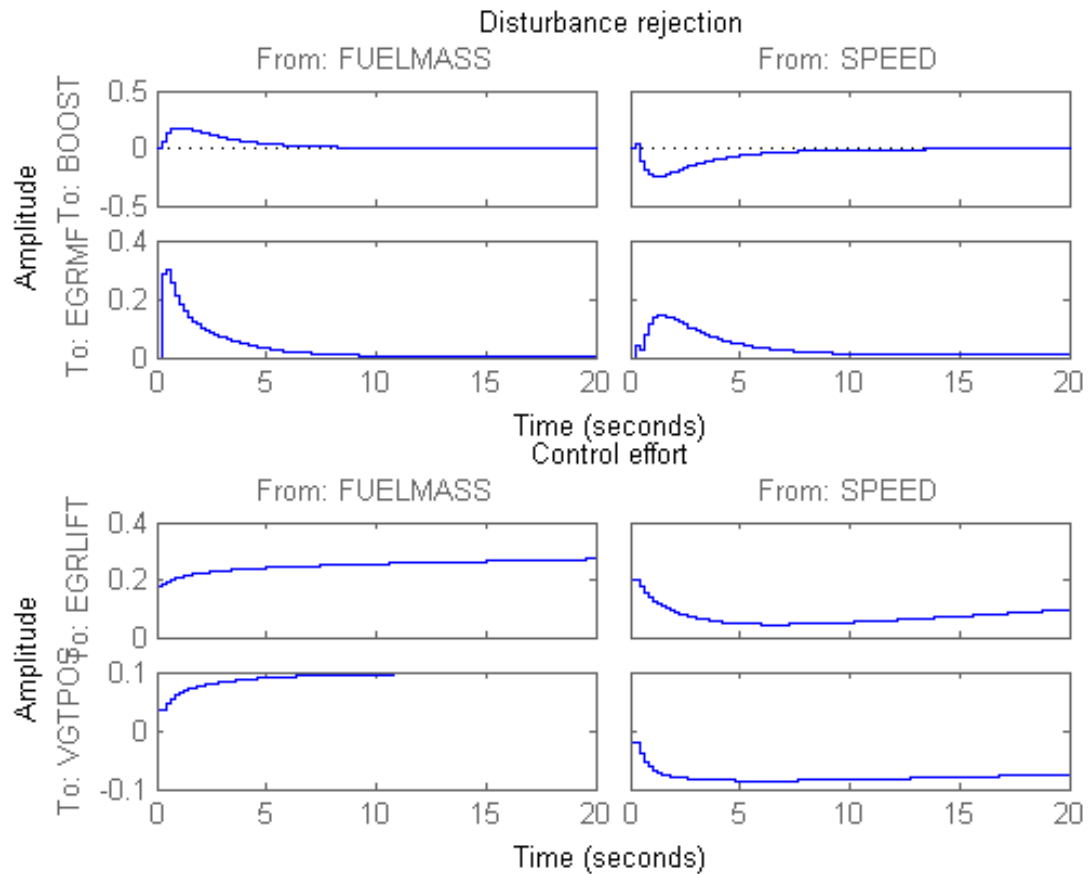


Plot the setpoint tracking and disturbance rejection responses.

```
clf
T1 = getIOTransfer(ST1,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT'})
subplot(211), step(T1(1:2,:),20), title('Setpoint tracking')
subplot(212), step(T1(3:4,:),20), title('Control effort')
```



```
D1 = getIOTransfer(ST1,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRIFT','VGT'});
subplot(211), step(D1(1:2,:),20), title('Disturbance rejection')
subplot(212), step(D1(3:4,:),20), title('Control effort')
```



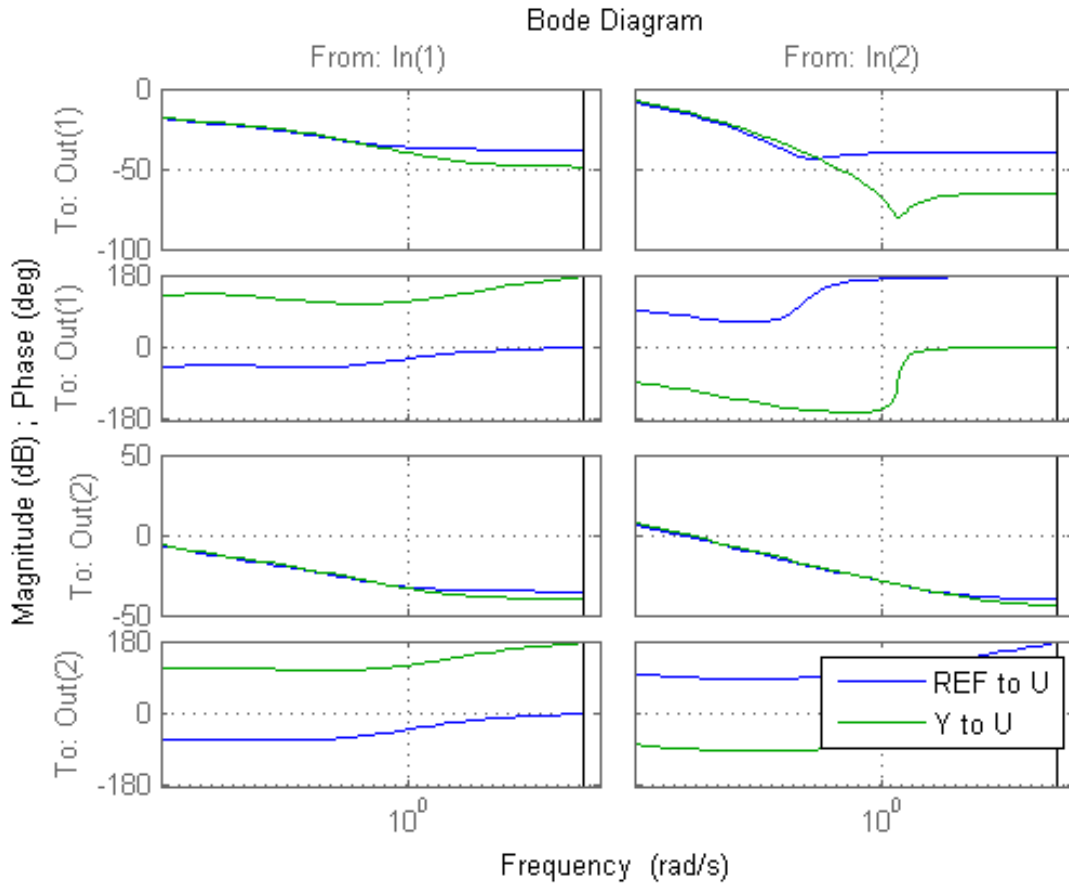
The controller responds in less than 5 seconds with minimum cross-coupling between the BOOST and EGRMF variables.

### Tuning of Simplified Control Structure

The state-space controller could be implemented as is, but it is often desirable to boil it down to a simpler, more familiar structure. To do this, get the tuned controller and inspect its frequency response

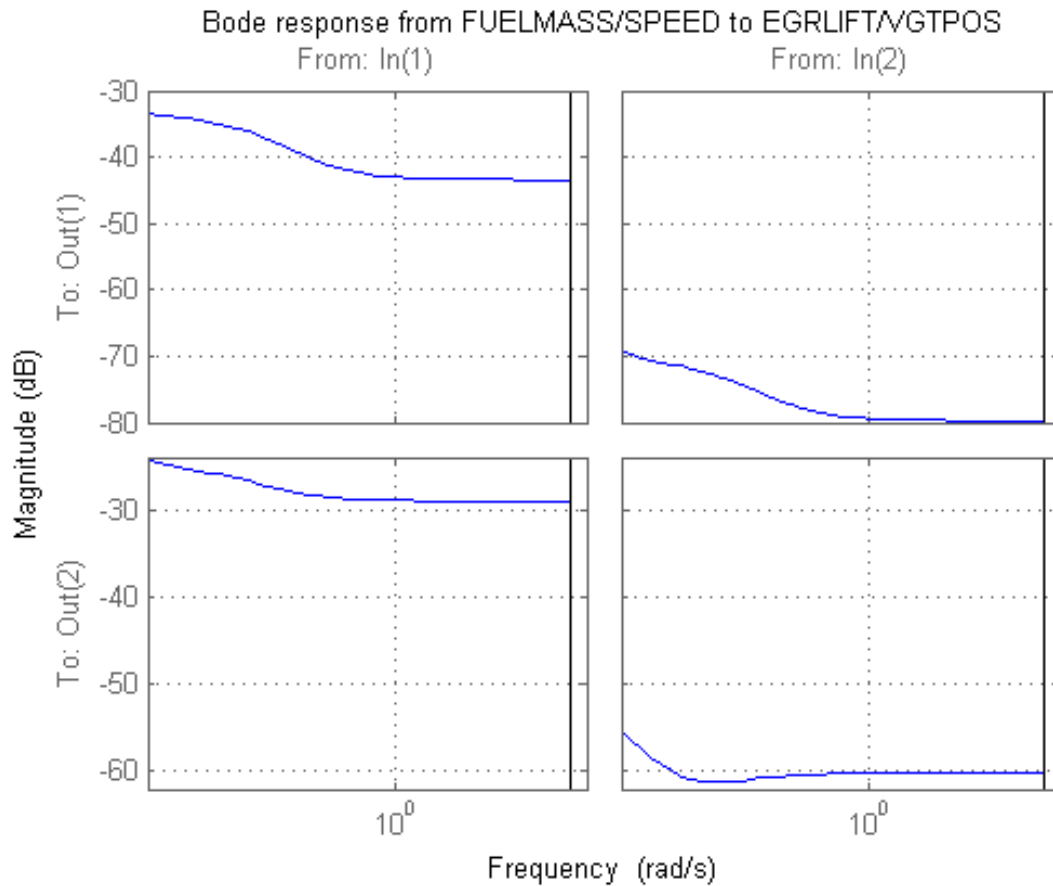
```
C = getBlockValue(ST1, 'SS2');
```

```
clf
bode(C(:,1:2),C(:,3:4),{.02 20}), grid
legend('REF to U','Y to U')
```

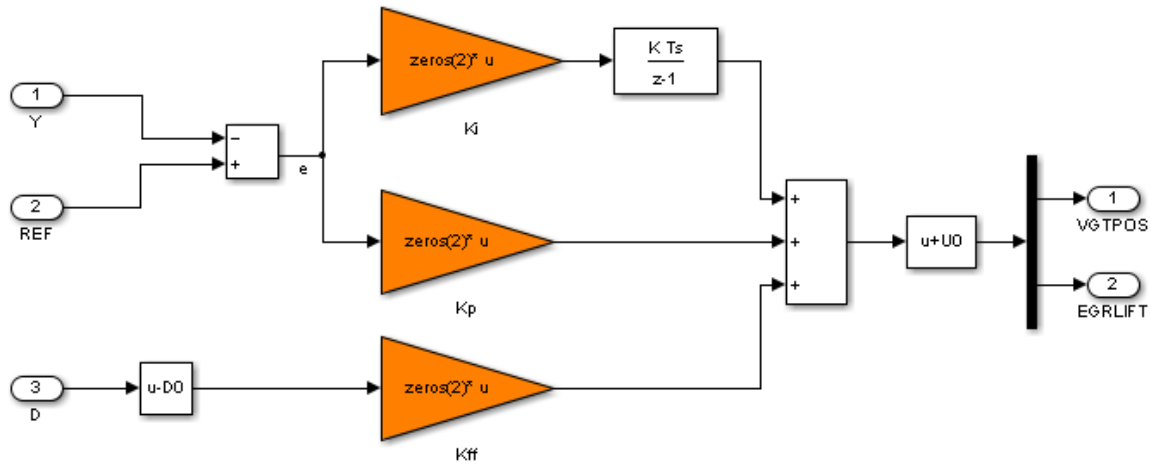


```
bodemag(C(:,5:6),{.02 20}), grid
title('Bode response from FUELMASS/SPEED to EGRLIFT/VGTPOS')
```





The first plot suggests that the controller essentially behaves like a PI controller acting on REF-Y (the difference between the target and actual values of the controlled variables). The second plot suggests that the transfer from measured disturbance to manipulated variables could be replaced by a simple gain. Altogether this suggests the following simplified control structure consisting of a MIMO PI controller with a disturbance feedforward gain.



MIMO PID with disturbance feedforward

**Figure 2: Simplified control structure.**

Using variant subsystems, you can implement both control structures in the same Simulink model and use a variable to switch between them. Here setting `MODE=2` selects the MIMO PI structure. Again use `systune` to tune the three 2-by-2 gain matrices `Kp`, `Ki`, `Kff` in the simplified control structure.

```
% Select "MIMO PI" variant in "CONTROLLER" block
MODE = 2;
```

```
% Configure tuning interface
```

```
ST0 = slTunable('rct_diesel',{'Kp','Ki','Kff'});
```

```
ST0.addSwitch({'CONTROLLER/EGRLIFT','CONTROLLER/VGTPOS','DIESEL ENGINE'})
```

```
ST0.Ts = 0.2;
```

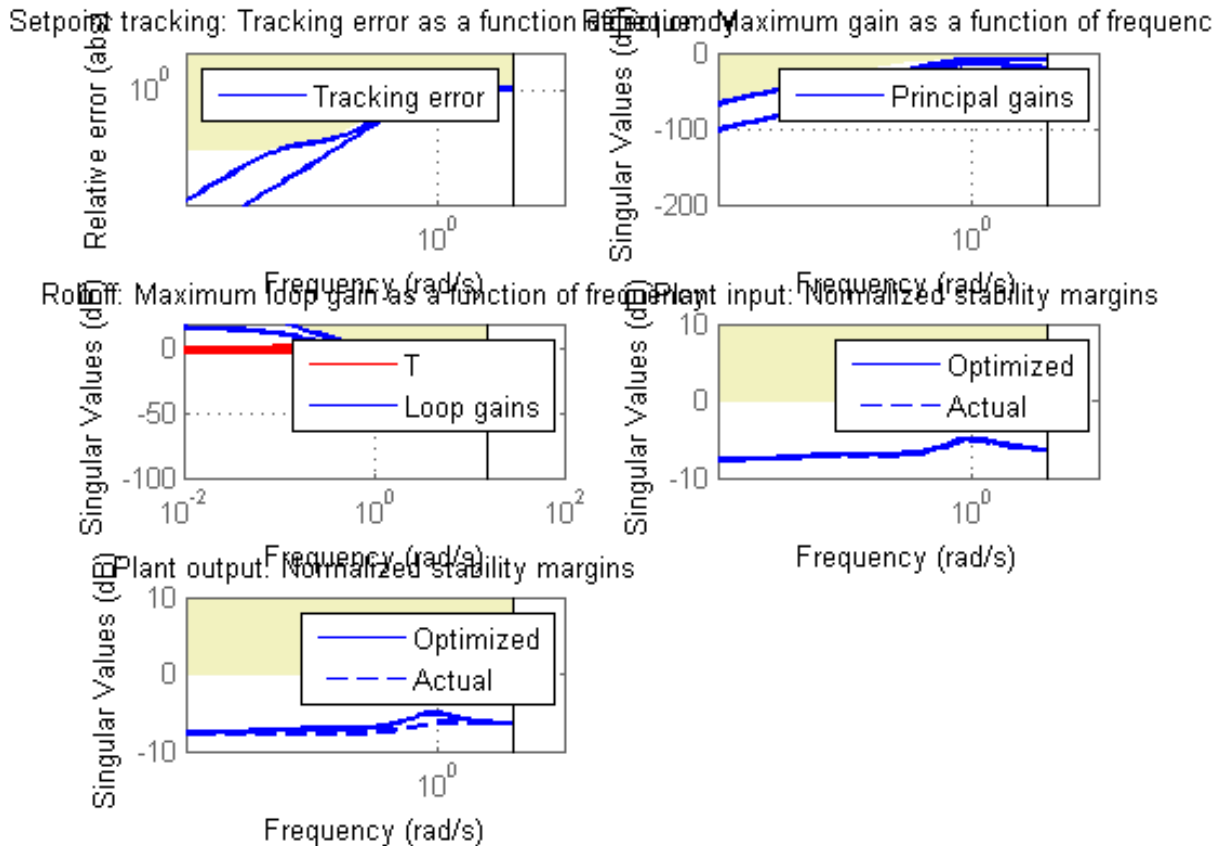
```
% Tune MIMO PI controller.
```

```
[ST2,~,~,Info2] = systune(ST0,[TR DR],[M1 M2 RO]);
```

```
Final: Soft = 1.08, Hard = 0.99923, Iterations = 174
```

Again all requirements are nearly met.

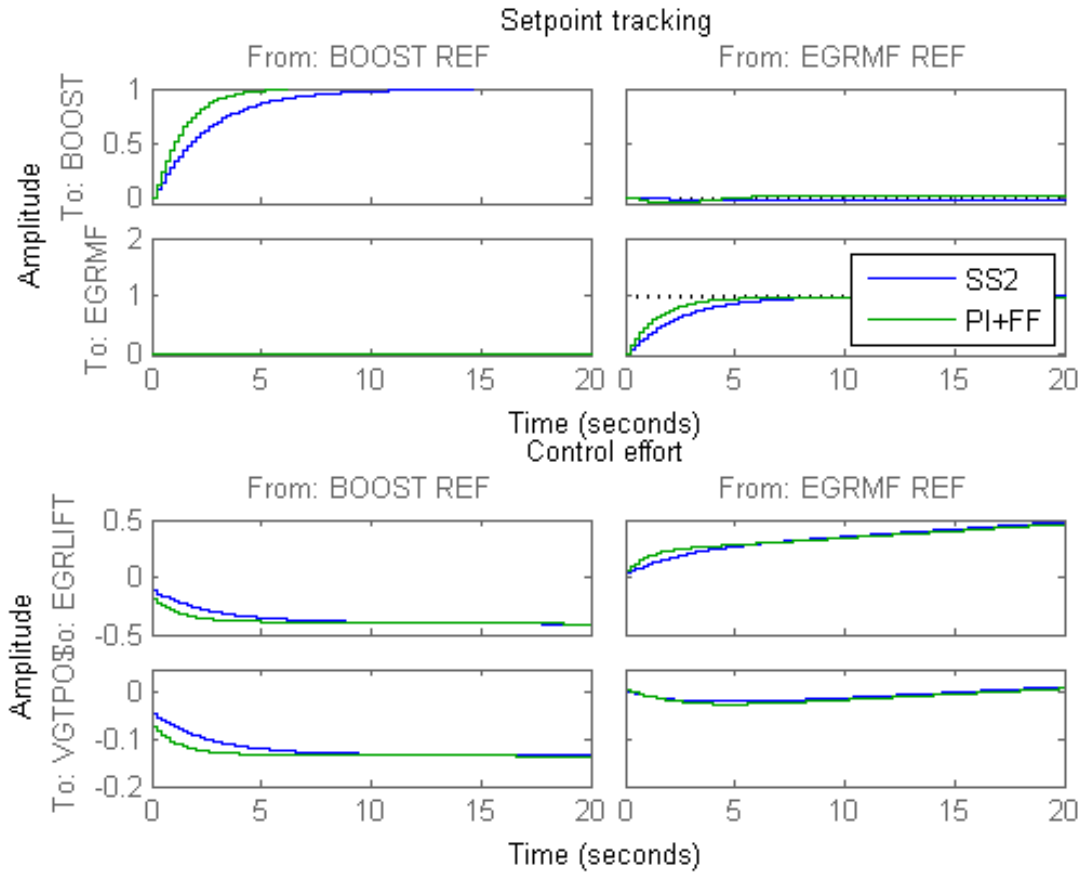
```
viewSpec([TR DR R0 M1 M2],ST2,Info2)
```



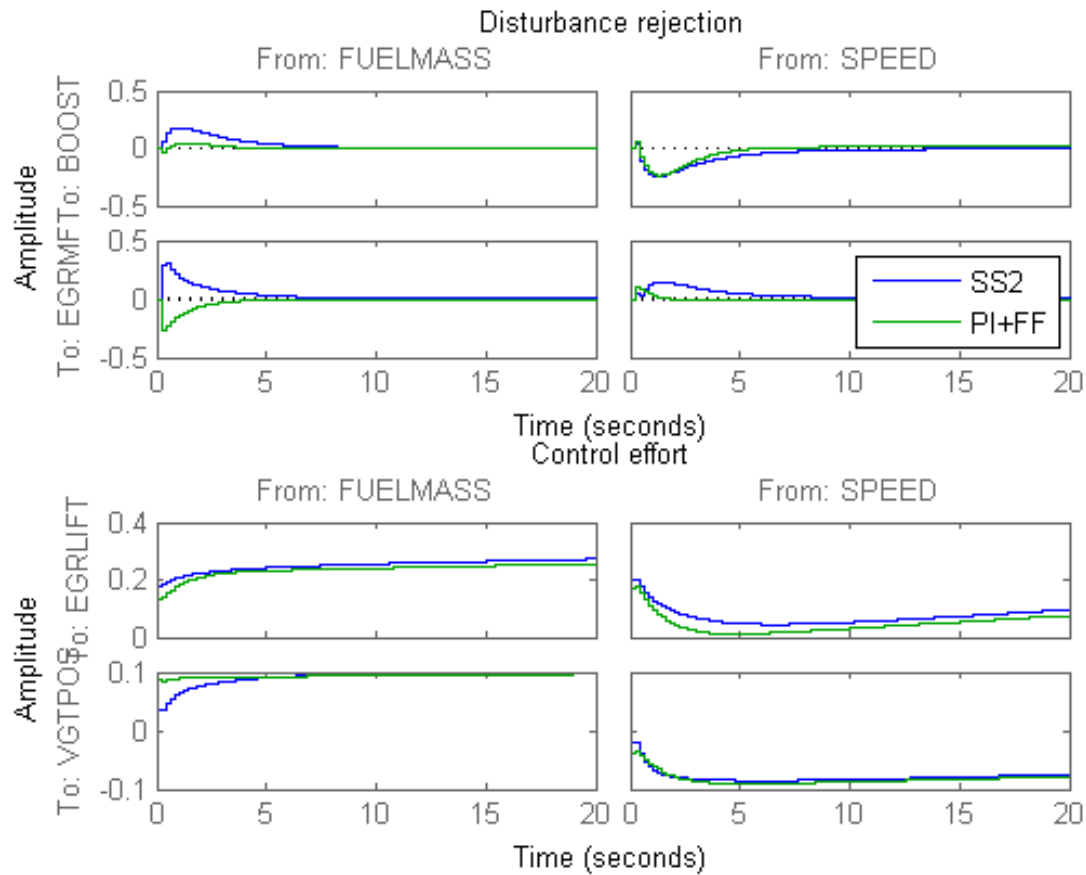
Plot the closed-loop responses and compare with the state-space design.

```
clf
T2 = getIOTransfer(ST2,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRIFT'})
subplot(211), step(T1(1:2,:),T2(1:2,:),20), title('Setpoint tracking')
legend('SS2','PI+FF')
```

```
subplot(212), step(T1(3:4,:),T2(3:4,:),20), title('Control effort')
```



```
D2 = getIOTransfer(ST2,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGT'});
subplot(211), step(D1(1:2,:),D2(1:2,:),20), title('Disturbance rejection')
legend('SS2','PI+FF')
subplot(212), step(D1(3:4,:),D2(3:4,:),20), title('Control effort')
```



The blackbox and simplified control structures deliver similar performance. Inspect the tuned values of the PI and feedforward gains.

```
showTunable(ST2)
```

```
Block "rct_diesel/CONTROLLER/MIMO PID/Kp" =
```

```
d =
```

```
u1
```

```
u2
```

```
y1 -0.007385 0.001252
y2 -0.01949 0.01933
```

Name: Kp  
Static gain.

-----  
Block "rct\_diesel/CONTROLLER/MIMO PID/Ki" =

```
d =
      u1      u2
y1 -0.009271 -0.002877
y2 -0.02677 0.07174
```

Name: Ki  
Static gain.

-----  
Block "rct\_diesel/CONTROLLER/MIMO PID/Kff" =

```
d =
      u1      u2
y1 0.01741 0.0001931
y2 0.026 -0.0008658
```

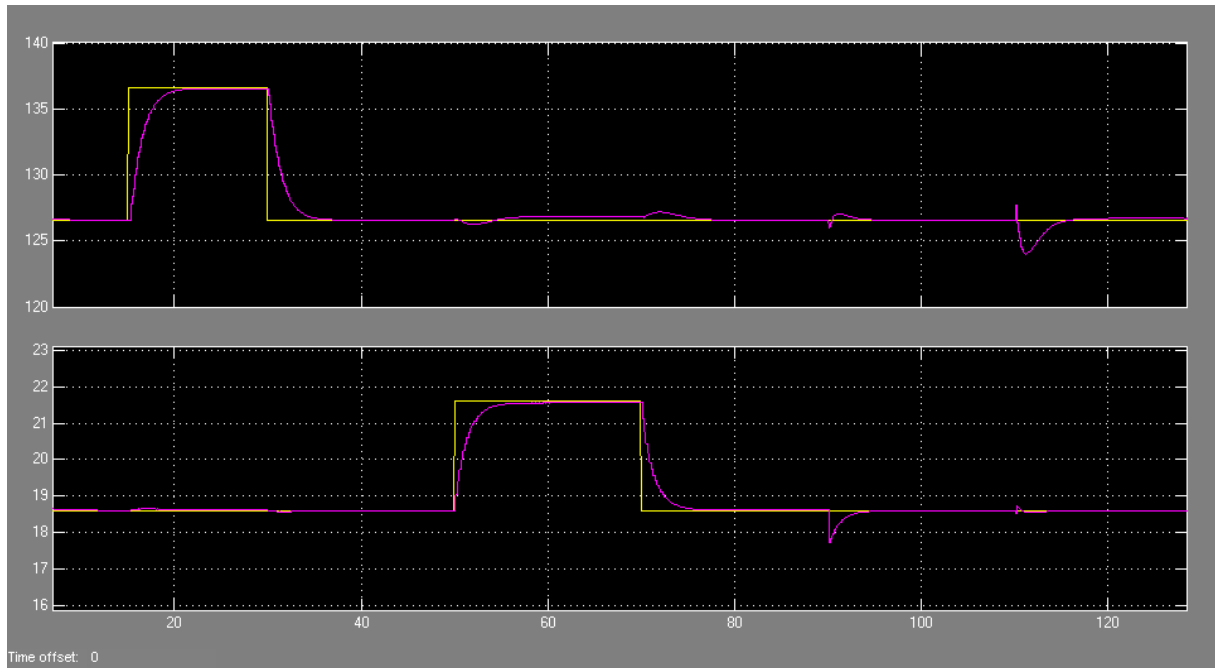
Name: Kff  
Static gain.

### Nonlinear Validation

To validate the MIMO PI controller in the Simulink model, push the tuned controller parameters to Simulink and run the simulation.

```
writeBlockValue(ST2)
```

The simulation results are shown below and confirm that the controller adequately tracks setpoint changes in boost pressure and EGR massflow and quickly rejects changes in fuel mass (at  $t=90$ ) and in speed (at  $t=110$ ).



**Figure 3: Simulation results with simplified controller.**

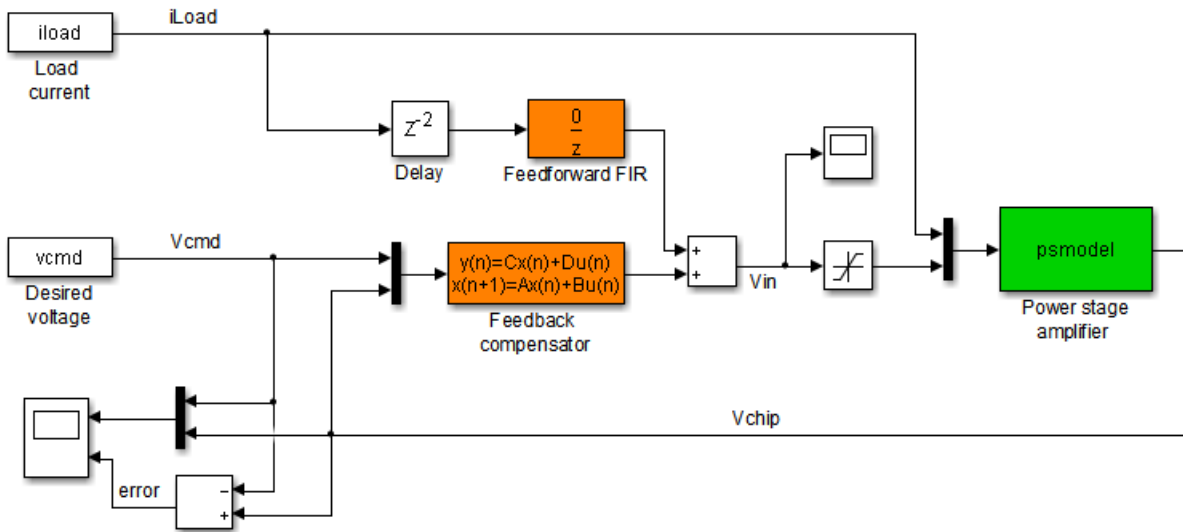
## Digital Control of Power Stage Voltage

This example shows how to tune a high-performance digital controller with bandwidth close to the sampling frequency.

### Voltage Regulation in Power Stage

We use Simulink to model the voltage controller in the power stage for an electronic device:

```
open_system('rct_powerstage')
```

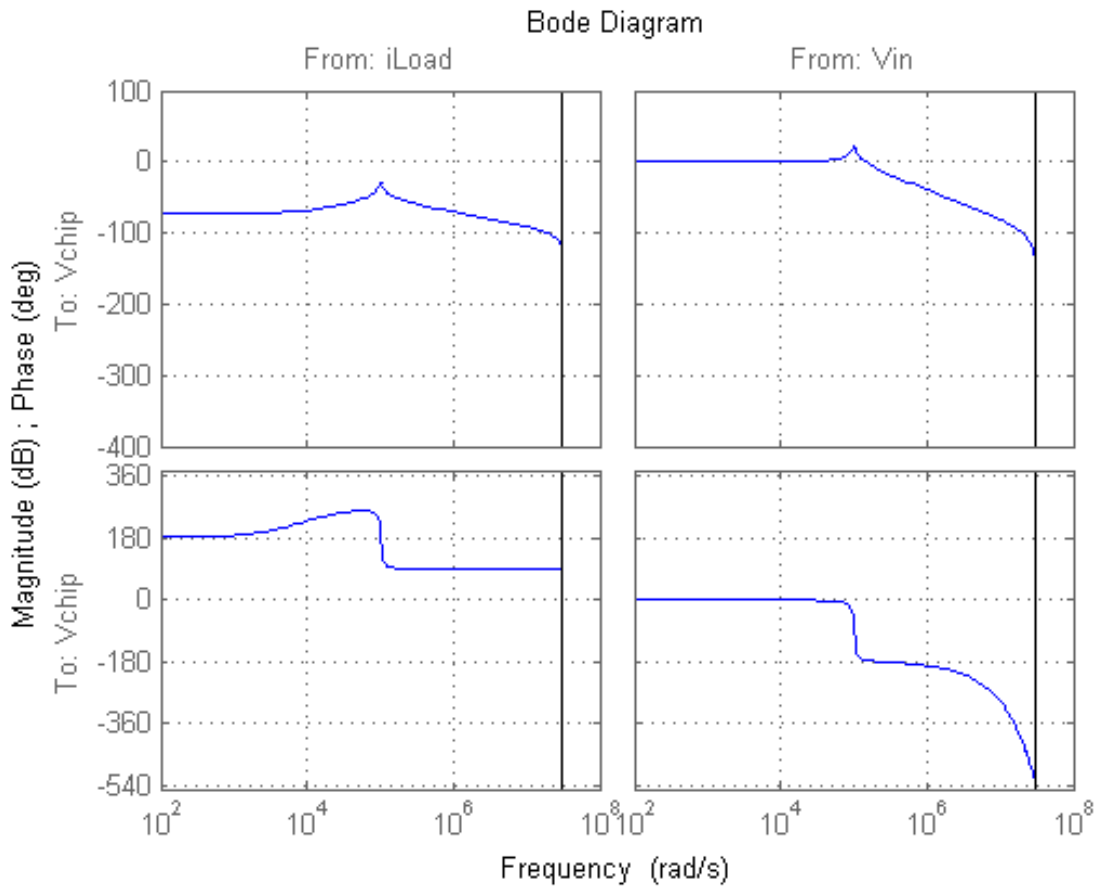


Voltage control in power stage

The power stage amplifier is modeled as a second-order linear system with the following frequency response:

```
bode(psmodel), grid
```





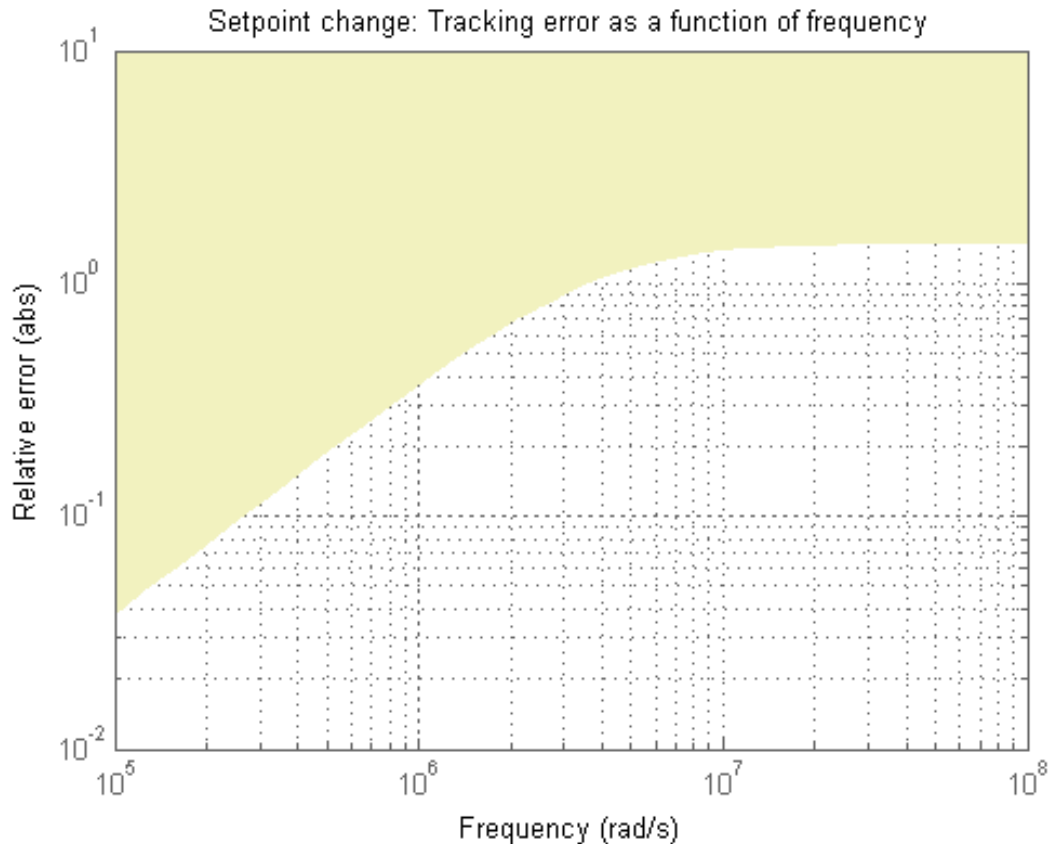
The controller must regulate the voltage  $V_{chip}$  delivered to the device to track the setpoint  $V_{cmd}$  and be insensitive to variations in load current  $i_{Load}$ . The control structure consists of a feedback compensator and a disturbance feedforward compensator. The voltage  $V_{in}$  going into the amplifier is limited to  $V_{max} = 12V$ . The controller sampling rate is 10 MHz (sample time  $T_m$  is  $1e-7$  seconds).

### Performance Requirements

This application is challenging because the controller bandwidth must approach the Nyquist frequency  $\pi/T_m = 31.4$  MHz. To avoid aliasing troubles when discretizing continuous-time controllers, it is preferable to tune the controller directly in discrete time.

The power stage should respond to a setpoint change in desired voltage  $V_{cmd}$  in about 10 sampling periods. Use a tracking requirement to capture this objective.

```
s = tf('s');  
nf = pi/Tm; % Nyquist frequency  
  
Req1 = TuningGoal.Tracking('Vcmd','Vchip',1.5*s/(s+nf/8));  
Req1.Name = 'Setpoint change';  
  
viewSpec(Req1)
```



The power stage should also quickly reject load disturbances  $i_{Load}$ . Express this requirement in terms of gain from  $i_{Load}$  to  $V_{chip}$ . This gain should be low at low frequency for good disturbance rejection.

```
Req2 = TuningGoal.Gain('iLoad','Vchip',1.5e-3 * s/nf);
Req2.Focus = [nf/1e4, nf];
Req2.Name = 'Load disturbance';
```

High-performance demands may lead to high control effort and saturation. For the ramp profile  $v_{cmd}$  specified in the Simulink model (from 0 to 1

in about 250 sampling periods), we want to avoid hitting the saturation constraint  $V_{in} \leq V_{max}$ . Use a rate-limiting filter to model the ramp command, and require that the gain from the rate-limiter input to  $V_{in}$  be less than  $V_{max}$ .

```
RateLimiter = 1/(250*Tm*s); % models ramp command in Simulink

% |RateLimiter * (Vcmd->Vin)| < Vmax
Req3 = TuningGoal.Gain('Vcmd','Vin',Vmax/RateLimiter);
Req3.Focus = [nf/1000, nf];
Req3.Name = 'Saturation';
```

To ensure adequate robustness, require at least 7 dB gain margin and 45 degrees phase margin at the plant input.

```
Req4 = TuningGoal.Margins('Vin',7,45);
Req4.Name = 'Margins';
```

Finally, the feedback compensator has a tendency to cancel the plant resonance by notching it out. Such plant inversion may lead to poor results when the resonant frequency is not exactly known or subject to variations. To prevent this, impose a minimum closed-loop damping of 0.5 to actively damp of the plant's resonant mode.

```
Req5 = TuningGoal.Poles;
Req5.MinDamping = 0.5;
Req5.MaxFrequency = 3*nf;
Req5.Name = 'Damping';
```

### Tuning

Next use `systemtune` to tune the controller parameters subject to the requirements defined above. First use the `s1Tunable` interface to configure the Simulink model for tuning. In particular, specify that there are two tunable blocks and that the model should be linearized and tuned at the sample time  $T_m$ .

```
TunedBlocks = {'compensator','FIR'};
STO = s1Tunable('rct_powerstage',TunedBlocks,'Ts',Tm);

% Register signals of interest and open-loop analysis points
STO.addIO({'Vcmd','iLoad'},'input')
```

```
ST0.addSwitch({'Vchip','Vin'});
```

We want to use an FIR filter as feedforward compensator. To do this, create a parameterization of a first-order FIR filter and assign it to the "Feedforward FIR" block in Simulink.

```
FIR = ltiblock.tf('FIR',1,1,Tm);
% Fix denominator to z^n
FIR.den.Value = [1 0];
FIR.den.Free = false;
ST0.setBlockParam('FIR',FIR);
```

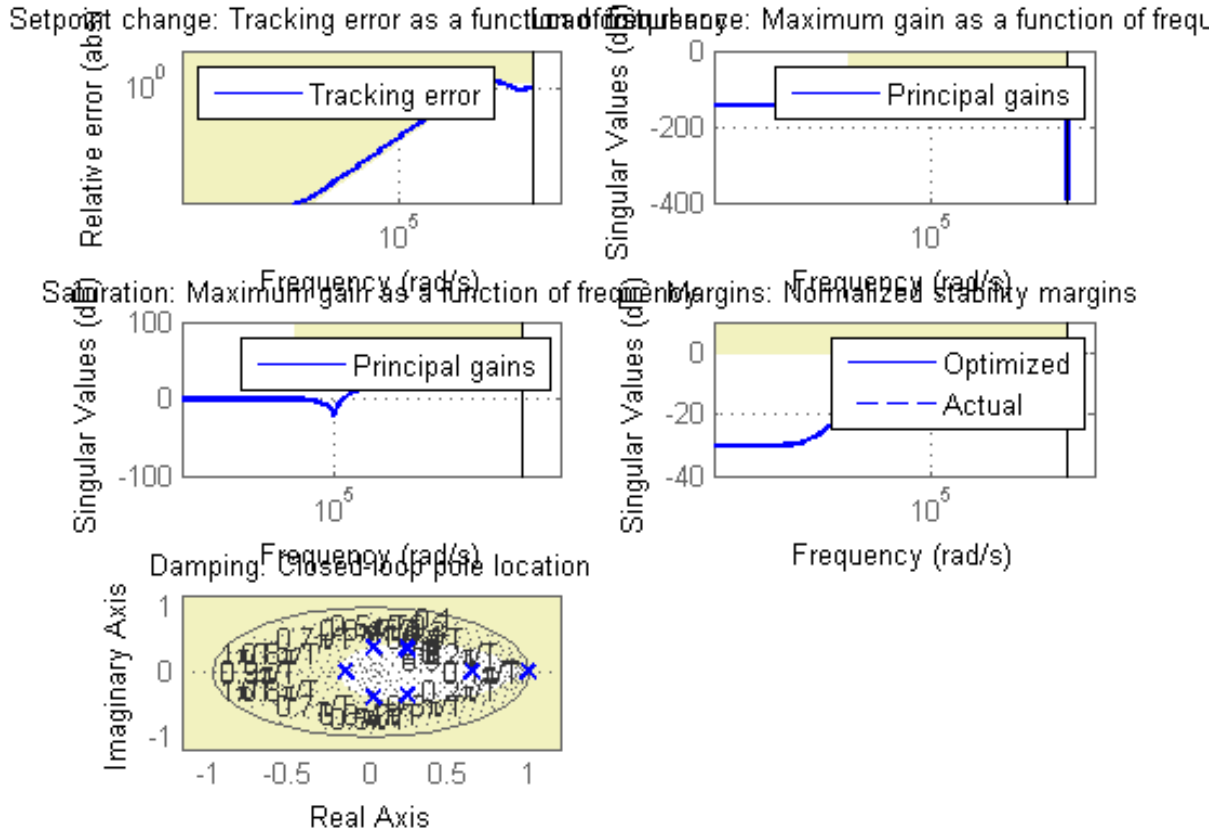
Note that `sITunable` automatically parameterizes the feedback compensator as a third-order state-space model (the order specified in the Simulink block). Next tune the feedforward and feedback compensators with `systune`. Treat the damping and margin requirements as hard constraints and try to best meet the the remaining requirements.

```
rng(0)
topt = systuneOptions('RandomStart',3);
ST = systune(ST0,[Req1 Req2 Req3],[Req4 Req5],topt);
```

```
Final: Soft = 1.29, Hard = 0.96001, Iterations = 342
Final: Soft = 1.46, Hard = 0.99996, Iterations = 478
Final: Soft = 1.61, Hard = 0.99859, Iterations = 352
Final: Soft = 1.27, Hard = 0.99612, Iterations = 358
```

The best design satisfies the hard constraints (`Hard` less than 1) and nearly satisfies the other constraints (`Soft` close to 1). Verify this graphically by plotting the tuned responses for each requirement.

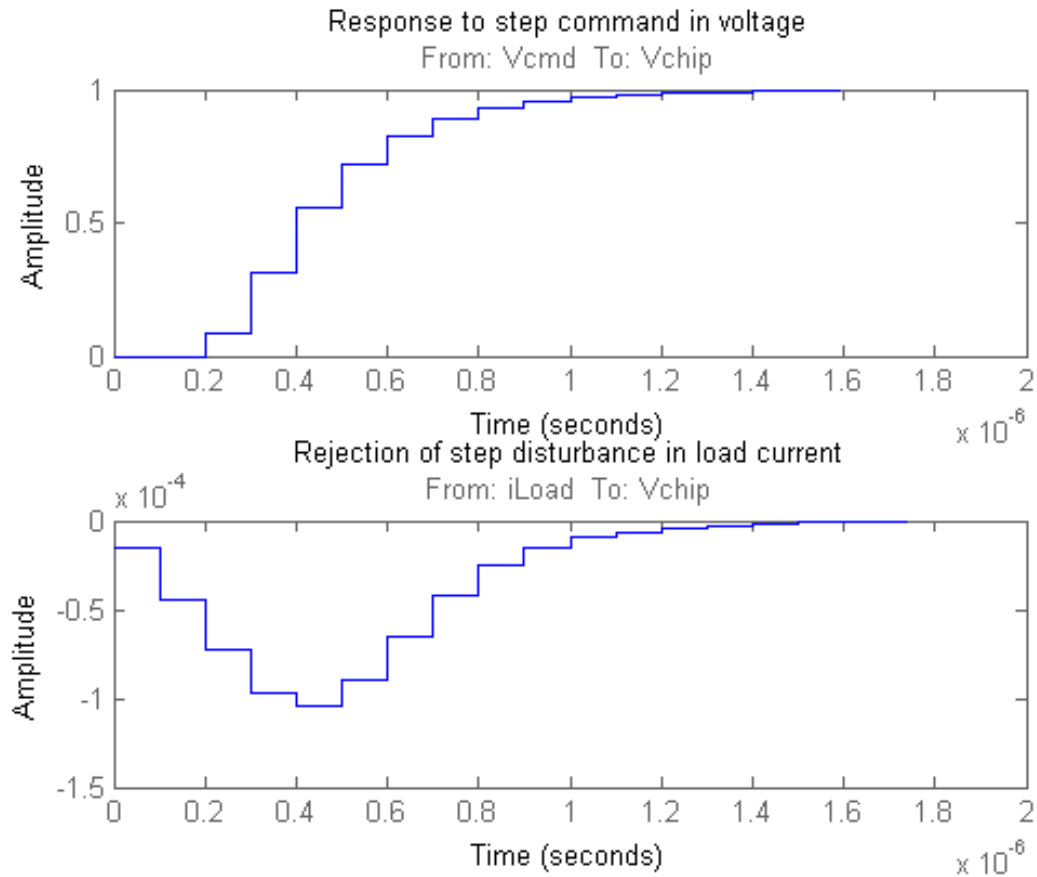
```
viewSpec([Req1 Req2 Req3 Req4 Req5],ST)
```



### Validation

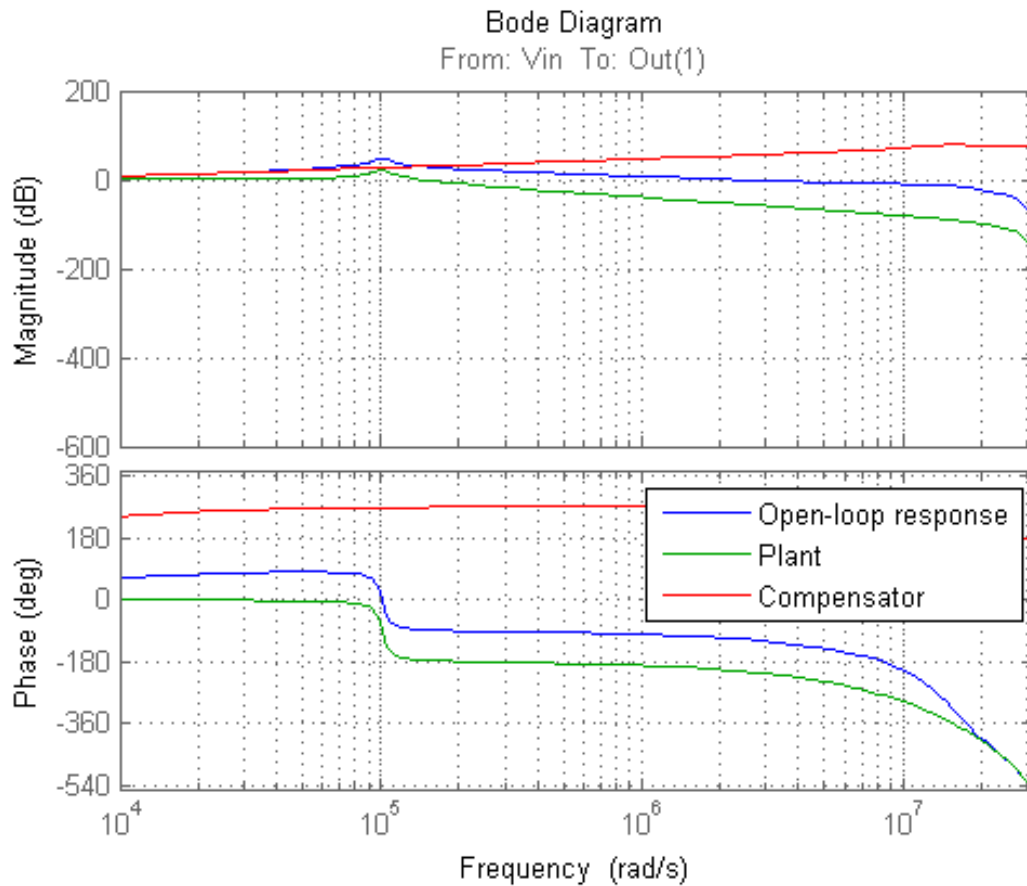
First validate the design in the linear domain using the `sITunable` interface. Plot the closed-loop response to a step command `Vcmd` and a step disturbance `iLoad`.

```
clf
subplot(211), step(ST.getIOTransfer('Vcmd','Vchip'),20*Tm)
title('Response to step command in voltage')
subplot(212), step(ST.getIOTransfer('iLoad','Vchip'),20*Tm)
title('Rejection of step disturbance in load current')
```



Use `getLoopTransfer` to compute the open-loop response at the plant input and superimpose the plant and feedback compensator responses.

```
clf
L = ST.getLoopTransfer('Vin', -1);
C = getBlockValue(ST, 'compensator');
bodeplot(L, psmode1(2), C(2), {1e-3/Tm pi/Tm}), grid
legend('Open-loop response', 'Plant', 'Compensator')
```



The controller achieves the desired bandwidth and the responses are fast enough. Apply the tuned parameter values to the Simulink model and simulate the tuned responses.

```
ST.writeBlockValue()
```

The results from the nonlinear simulation appear below. Note that the control signal  $V_{in}$  remains approximately within  $\pm 12V$  saturation bounds for the setpoint tracking portion of the simulation.



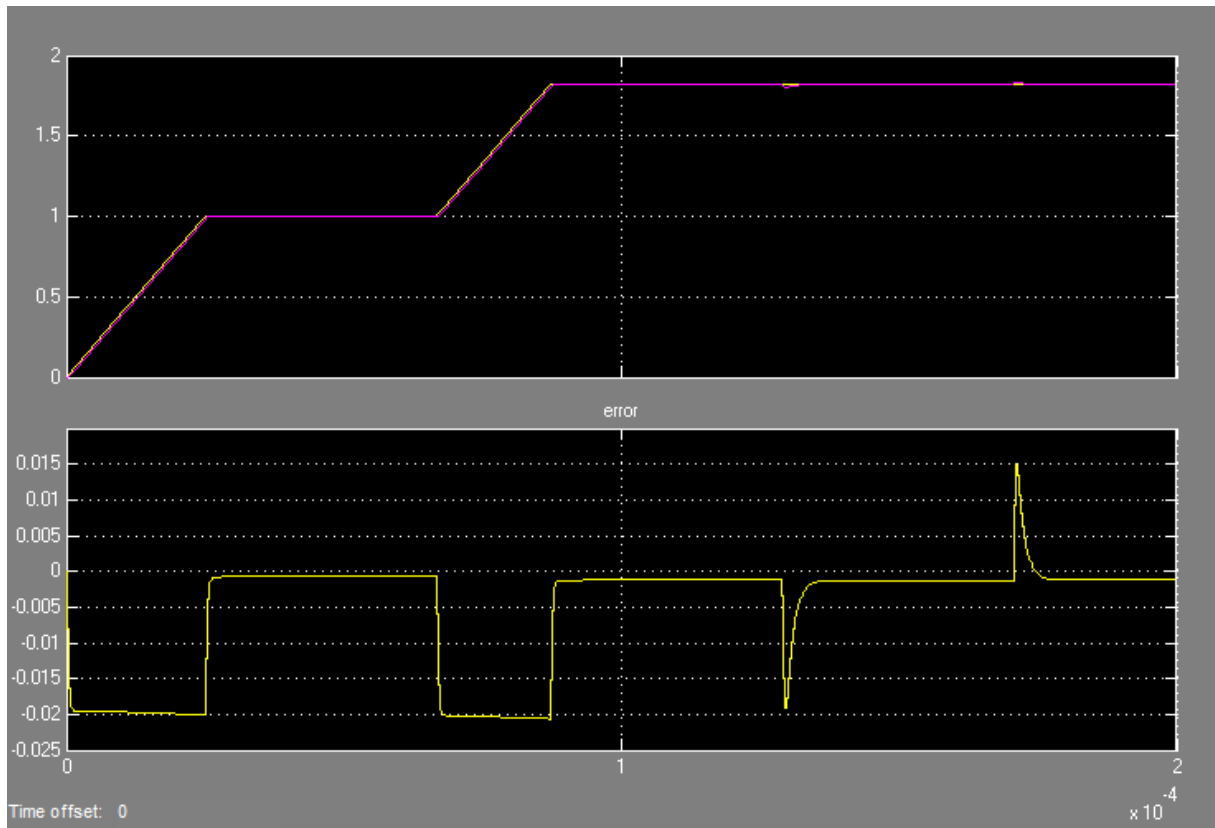
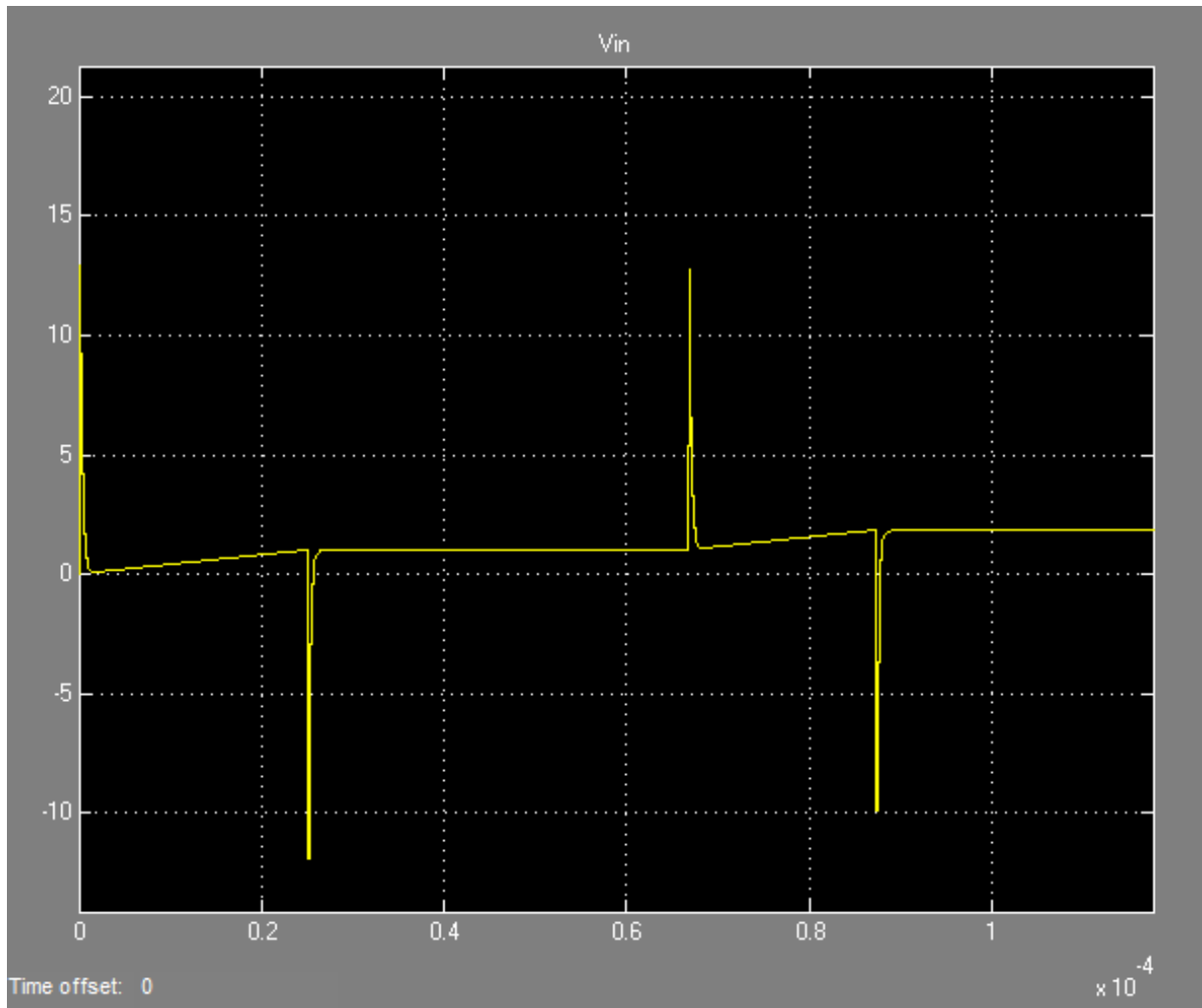


Figure 1: Response to ramp command and step load disturbances.



**Figure 2:** Amplitude of input voltage  $V_{in}$  during setpoint tracking phase.

# Gain-Scheduled Controllers

---

- “Gain-Scheduled Control Systems” on page 7-2
- “Plant Models for Gain-Scheduled Control” on page 7-4
- “Parametric Gain Surfaces” on page 7-8
- “Tuning Gain-Scheduled Controllers” on page 7-13
- “Validating Gain-Scheduled Controllers” on page 7-14
- “Improving Gain-Scheduled Tuning Results” on page 7-15
- “Tunable Gain Surface With Two Scheduling Variables” on page 7-18
- “Gain-Scheduled PID Controller” on page 7-22
- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 7-24

## Gain-Scheduled Control Systems

A *gain-scheduled controller* is a controller whose gains are automatically adjusted as a function of time, operating condition, or plant parameters. Gain scheduling is a common strategy for controlling systems whose dynamics change with time or operating condition. Such systems include linear parameter-varying (LPV) systems and large classes of nonlinear systems. Gain scheduling is most suitable when the scheduling variables are external parameters that vary slowly compared to the control bandwidth, such as ambient temperature of a reaction or speed of a cruising aircraft. Gain scheduling is most challenging when the scheduling variables depend on fast-varying states of the system. Because local linear performance near operating points is no guarantee of global performance in nonlinear systems, extensive simulation-based validation is usually required. See [1] for an overview of gain scheduling and its challenges.

Typically, gain-scheduled controllers are fixed single-loop or multi-loop control structures that use lookup tables to specify gain values as a function of the scheduling variables. For tuning purposes, it is convenient to replace lookup tables by parametric gain surfaces. A *parametric gain surface* is a basis-function expansion whose coefficients are tunable. For example, you can model a time-varying gain  $k(t)$  as a cubic polynomial in  $t$ :

$$k(t) = k_0 + k_1t + k_2t^2 + k_3t^3.$$

Here,  $k_0, \dots, k_3$  are tunable coefficients. For applications where gains vary smoothly with the scheduling variables, this approach drastically reduces the number of tunable parameters. This approach also provides explicit formulas for the gains, and ensures smooth transitions between operating points. In addition, you can use `sys tune` to automatically tune the gain surface coefficients to meet your control objectives at a representative set of operating conditions.

## References

- [1] Rugh, W.J., and J.S. Shamma, “Research on Gain Scheduling”, *Automatica*, 36 (2000), pp. 1401-1425.

**Concepts**

- “Plant Models for Gain-Scheduled Control” on page 7-4

## Plant Models for Gain-Scheduled Control

**In this section...**

“Gain Scheduling for Linear Parameter-Varying Plants” on page 7-4

“Gain Scheduling for Nonlinear Plants” on page 7-5

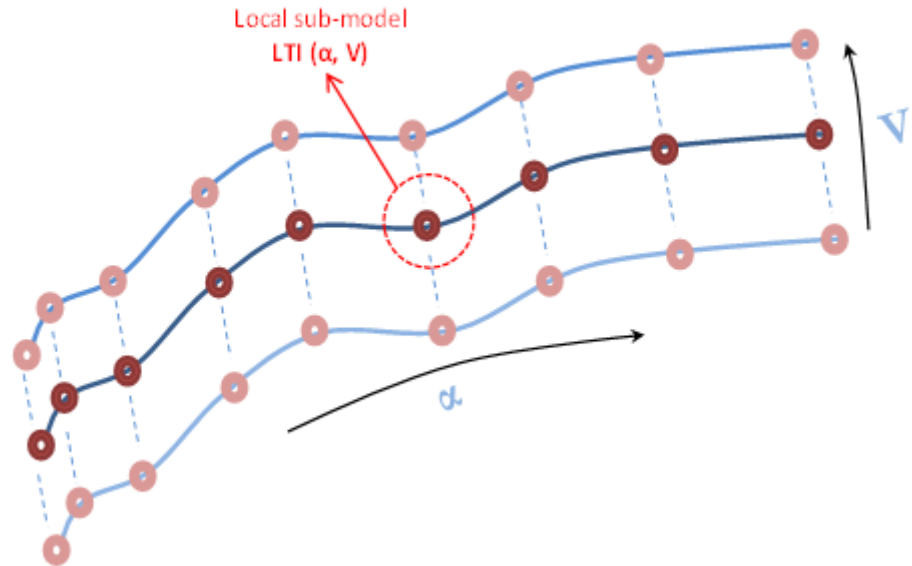
### Gain Scheduling for Linear Parameter-Varying Plants

Gain-scheduled controllers are usually designed and tuned using a family of linear models that depend on the scheduling variables,  $\sigma$ :

$$\begin{aligned}\frac{dx}{dt} &= A(\sigma)x + B(\sigma)u \\ y &= C(\sigma)\dot{x} + D(\sigma)u.\end{aligned}$$

This family of models is called a *linear parameter-varying* (LPV) model. The LPV model describes how the (linearized) plant dynamics vary with time, operating condition, or any other scheduling variable. For example, the pitch axis dynamics of an aircraft can be approximated by an LPV model that depends on incidence angle,  $\alpha$ , air speed,  $V$ , and altitude,  $h$ .

In practice, this continuum of plant models is often replaced by a finite set of linear models obtained for a suitable grid of  $\sigma$  values. This amounts to sampling the LPV dynamics over the operating range and selecting a set of representative design points.



For best results with a gain-scheduled controller, the plant dynamics should vary smoothly between design points.

When you are tuning a gain-scheduled controller for an LPV plant, you can use a sampled array of LTI plant models. Use the `SamplingGrid` property of the LTI model array to associate each linear models in the set with the underlying design points,  $\sigma$ . For example, see “Sample a Tunable (Parametric) Model for Parameter Studies”.

## Gain Scheduling for Nonlinear Plants

In most applications, the plant is not given as an LPV model or a collection of linear models. Instead, the plant dynamics are described by nonlinear differential equations of the form:

$$\begin{aligned}\dot{x} &= f(x, u) \\ y &= g(x, u).\end{aligned}$$

$x$  is the state vector,  $u$  is the plant input, and  $y$  is the plant output. These equations can be specified explicitly. Or, they can be specified implicitly, such as by a Simulink model. For nonlinear plants, the linearized dynamics describe the local behavior of the plant around a family of operating points  $(x(\sigma), u(\sigma))$  parameterized by the scheduling variables,  $\sigma$ . Specifically, the deviations from nominal operating condition are defined as:

$$\delta x = x - x(\sigma), \quad \delta u = u - u(\sigma).$$

These deviations are governed, to first order, by the LPV dynamics:

$$\dot{\delta x} = A(\sigma)\delta x + B(\sigma)\delta u, \quad \delta y = C(\sigma)\delta x + D(\sigma)\delta u,$$

$$\begin{aligned} A(\sigma) &= \frac{\partial f}{\partial x}(x(\sigma), u(\sigma)) & B(\sigma) &= \frac{\partial f}{\partial u}(x(\sigma), u(\sigma)) \\ C(\sigma) &= \frac{\partial g}{\partial x}(x(\sigma), u(\sigma)) & D(\sigma) &= \frac{\partial g}{\partial u}(x(\sigma), u(\sigma)). \end{aligned}$$

When repeated for a finite set of design points,  $\sigma$ , this local linearization produces the type of sampled LPV model described in “Gain Scheduling for Linear Parameter-Varying Plants” on page 7-4.

When your nonlinear plant is modeled in Simulink, you can use Simulink Control Design linearization tools to compute  $A(\sigma)$ ,  $B(\sigma)$ ,  $C(\sigma)$ , and  $D(\sigma)$  for specific values of  $\sigma$ . If the operating points are equilibrium points, you might need to first use `findop` to compute  $(x(\sigma), u(\sigma))$  as a function of  $\sigma$ . If you are controlling the system around a reference trajectory  $(x(\sigma), u(\sigma))$ , you can use snapshot linearization to acquire  $A(\sigma)$ ,  $B(\sigma)$ ,  $C(\sigma)$ , and  $D(\sigma)$  at various points along the  $\sigma$  trajectory. This second scenario includes time-varying systems where the scheduling variable is time. In either case, the result is a collection of linear models sampled at values of  $\sigma$ .

Typically, the gain-scheduled controller’s main task is to keep the closed-loop system close to equilibrium or close to the nominal trajectory,  $(x(\sigma), u(\sigma))$ . The controller provides a corrective action  $\delta u$  which must be added to the nominal command  $u(\sigma)$  to determine the total actuator command.



## **Concepts**

- “Gain-Scheduled Control Systems” on page 7-2
- “Parametric Gain Surfaces” on page 7-8

## Parametric Gain Surfaces

In a gain-scheduled controller, the gains, which are the tunable parameters, are functions of the scheduling variables,  $\sigma$ . For example, a gain-scheduled PI controller is of the form:

$$C(s, \sigma) = K_p(\sigma) + \frac{K_i(\sigma)}{s}.$$

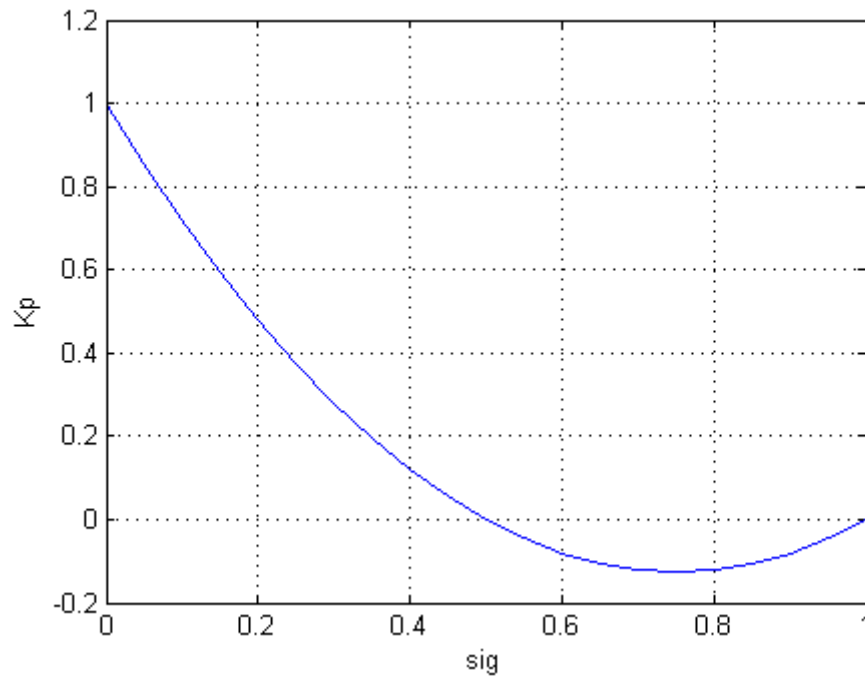
Tuning arbitrary functions is difficult. Therefore, it is necessary either to discretize the operating range, or restrict the generality of the functions themselves.

In the first approach, a collection of design points,  $\sigma$ , is chosen. Then, the gains,  $K_p$  and  $K_i$ , are tuned independently at each design point. The resulting set of gain values is stored in a lookup table driven by the scheduling variables,  $\sigma$ . A drawback of this approach is that the tuning method may yield substantially different values for neighboring design points, leading to undesirable jumps when transitioning from one operating point to another.

Alternatively, you can model the gains as smooth functions of  $\sigma$ , but restrict the generality of such functions by using specific basis function expansions. For example, suppose  $\sigma$  is a scalar variable. You can model  $K_p(\sigma)$  as a quadratic function of  $\sigma$ :

$$K_p(\sigma) = k_0 + k_1\sigma + k_2\sigma^2.$$

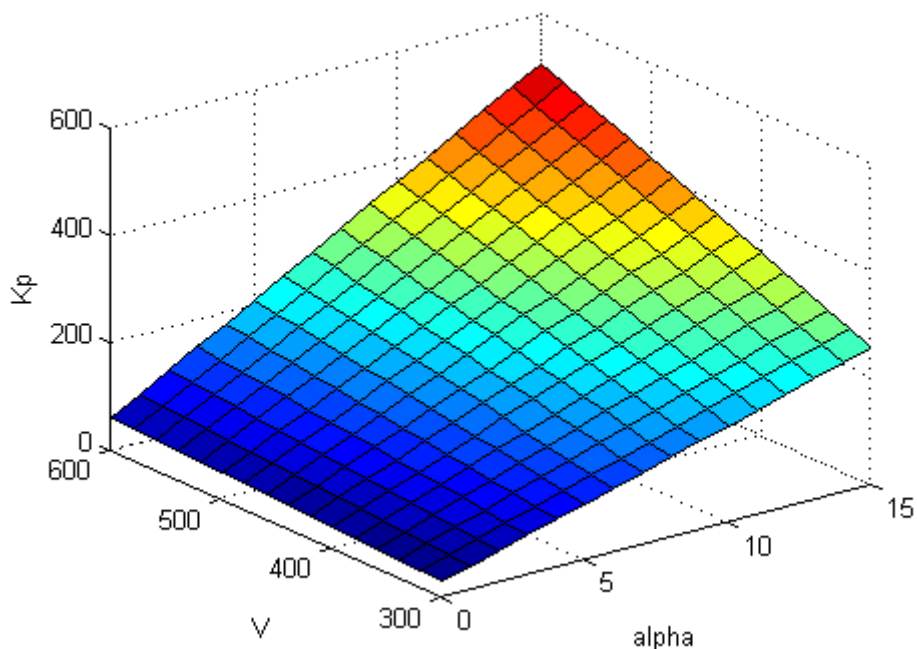
After tuning, this parametric gain might have a profile such as the following (the specific shape of the curve depends on the tuned coefficient values and range of  $\sigma$ ):



Or, suppose that  $\sigma$  consists of two scheduling variables,  $\alpha$ , and  $V$ . Then, you can model  $K_p(\sigma)$  as a bilinear function of  $\alpha$  and  $V$ :

$$K_p(\alpha, V) = k_0 + k_1\alpha + k_2V + k_3\alpha V.$$

After tuning, this parametric gain might have a profile such as the following (again, the specific shape of the curve depends on the tuned coefficient values and ranges of  $\sigma$  values):



In general, a *parametric gain surface* is a particular expansion of the gain in basis functions of  $\sigma$ :

$$K(\sigma) = K_0 + K_1 f_1(\sigma) + \dots + K_M f_M(\sigma).$$

The basis functions  $f_1, \dots, f_M$  are user-selected and fixed. The coefficients of the expansion,  $K_0, \dots, K_M$ , are the tunable parameters of the gain surface.  $K_0, \dots, K_M$  can be scalar or matrix-valued, depending on the size of the gain  $K(\sigma)$ . The choice of basis function is problem-dependent, but it is generally good to try low-order polynomial expansions first.

To tune a gain-scheduled controller with `systemtune`, use the `gainsurf` command to construct a tunable model of a gain surface sampled over a grid of design points ( $\sigma$  values). For example, consider the gain with bilinear dependence on two scheduling variables,  $\alpha$ , and  $V$ :

$$K_p(\alpha, V) = k_0 + k_1\alpha + k_2V + k_3\alpha V.$$

Suppose that  $\alpha$  is an angle of incidence that ranges from 0 to 15 degrees, and  $V$  is a speed that ranges from 300 to 600 m/s. Create a grid of design points that span these ranges.

```
[alpha,V] = ndgrid(0:5:15,300:100:600);
```

Specify the values of the basis functions  $f_1 = \alpha$ ,  $f_2 = V$ ,  $f_3 = \alpha V$  over the grid of design points.

```
F1 = alpha;
```

```
F2 = V;
```

```
F3 = alpha.*V;
```

```
Kp = gainsurf('Kp',1,F1,F2,F3);
```

$K_p$  is a tunable model of the gain surface,  $K_p(\alpha, V)$ . This model contains the tunable coefficients  $k_0$ ,  $k_1$ ,  $k_2$ , and  $k_3$ .

You can use such gain surface models to construct more sophisticated gain-scheduled control elements, such as gain-scheduled PID controllers, filters, or state-space controllers. For example, suppose you create two gain surfaces  $K_p$  and  $K_i$  sampled over the same  $\sigma$  grid. The following command constructs a gain-scheduled tunable PI controller.

```
C = pid(Kp,Ki);
```

Similarly, suppose you create four matrix-valued gain surfaces  $A$ ,  $B$ ,  $C$ ,  $D$  sampled over the same  $\sigma$ . The following command constructs a gain-scheduled state-space controller:

```
C = ss(A,B,C,D);
```

While gain surfaces are tuned at a finite set of design points, they are guaranteed to vary as smoothly as the underlying basis functions from one design point to another.

**See Also** `gainsurf`

### **Related Examples**

- “Tunable Gain Surface With Two Scheduling Variables” on page 7-18
- “Gain-Scheduled PID Controller” on page 7-22

### **Concepts**

- “Plant Models for Gain-Scheduled Control” on page 7-4
- “Tuning Gain-Scheduled Controllers” on page 7-13

## Tuning Gain-Scheduled Controllers

You can use `systemtune` to automatically tune gain-scheduled controllers modeled with parametric gain surfaces. The tuning workflow is as follows:

- 1** Select a set of design points,  $\sigma$ , that adequately covers the operating range. The set can be a regular grid of  $\sigma$  values or a scattered set of values. Typically, start with a small number of design points. If the performance your tuned system achieved at the design points is not maintained in between design points, you can add more design points and retune.
- 2** Build a collection of linear models describing the linearized plant dynamics at the selected design points. You can do so by:
  - Linearizing a Simulink model at each operating condition,  $\sigma$  (requires Simulink Control Design).
  - Sampling an LPV model of the plant at the design points.
- 3** Use gain surfaces to model the gain-scheduled controller. Sample these gain surfaces at the same design points,  $\sigma$ .
- 4** Combine the plant and controller models to build a closed-loop model. The resulting model array covers all design points and depends on the tunable coefficients of the gain surfaces.
- 5** Use `systemtune` to tune the gain surface coefficients subject to suitable requirements at each design point. From the `systemtune` perspective, doing so amounts to tuning one set of parameters against many plant models (multi-model tuning).

### Related Examples

- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 7-24
- Gain Scheduled Control Of a Chemical Reactor

### Concepts

- “Plant Models for Gain-Scheduled Control” on page 7-4
- “Parametric Gain Surfaces” on page 7-8
- “Validating Gain-Scheduled Controllers” on page 7-14

## Validating Gain-Scheduled Controllers

Gain-scheduled controllers require careful validation. The tuning process only guarantees suitable performance in the neighborhood of each design point. In addition, the tuning ignores dynamic couplings between the plant state variables and the scheduling variables (see Section 4.3, “Hidden Coupling”, in [1]). Recommended validation includes:

- Check linear performance on a denser grid of  $\sigma$  values than you used for design. If adequate linear performance is not maintained between design points, you can add more design points and retune.
- Perform nonlinear simulations that drive the closed-loop system through its entire operating range. Pay special attention to maneuvers that cause rapid variations of the scheduling variables.

## References

[1] Rugh, W.J., and J.S. Shamma, “Research on Gain Scheduling”, *Automatica*, 36 (2000), pp. 1401-1425.

## Related Examples

- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 7-24
- Gain Scheduled Control Of a Chemical Reactor

## Concepts

- “Improving Gain-Scheduled Tuning Results” on page 7-15



## Improving Gain-Scheduled Tuning Results

### In this section...

“Normalize the Scheduling Variables” on page 7-15

“Changing Requirements With Operating Condition” on page 7-16

### Normalize the Scheduling Variables

Suppose you are scheduling with respect to a variable  $h$  (altitude) that ranges between 5,000 and 10,000 feet. Further, suppose you use a cubic polynomial model for the gain  $k(h)$ :

$$k(h) = k_0 + k_1h + k_2h^2 + k_3h^3.$$

The terms  $h^2$  and  $h^3$  are of order  $10^4$  and  $10^8$ , respectively. You can therefore expect  $k_2$  to be roughly  $10^4$  times smaller than  $k_1$ , and  $k_3$  to be  $10^8$  times smaller than  $k_1$ , to compensate. This means that the entries of the vector of tunable variables vary by as many as 8 orders of magnitude. Such wide variation may be detrimental to the performance and accuracy of the solvers used by systune.

To avoid such scaling issues, it is good practice to normalize the scheduling variables so that they vary in the range  $[-1,1]$ . For a variable  $x$  taking values in  $[x_{min}, x_{max}]$ , this normalization is a simple change of variable of the form:

$$x_N = \frac{x - x_{mean}}{dx},$$

$$x_{mean} = \frac{x_{min} + x_{max}}{2},$$

$$dx = \frac{x_{max} - x_{min}}{2}.$$

You then create gain surfaces that depend on the normalized scheduling variables, rather than the original scheduling variables. You must perform the same normalization when implementing the tuned controller.

For an example of a gain surface using normalized scheduling variables, see “Tunable Gain Surface With Two Scheduling Variables” on page 7-18.

## Changing Requirements With Operating Condition

It is not uncommon to have the control objectives themselves vary with the operating condition. In this case, the control objectives also depend on the scheduling variables. For example, your system might require a reduced control bandwidth if transmission delays increase or temperature drops. The software offers two approaches to tuning to with such scheduled requirements.

- Create a separate instance of the requirement for each design point. For example, suppose you want to enforce a  $1/s$  loop shape with a crossover frequency that depends on the scheduling variables. Suppose also that you have created a table, `wc`, that contains the target bandwidth for each design point,  $\sigma$ . Then you can construct one `TuningGoal.LoopShape` requirement for each design point. Associate each `TuningGoal.LoopShape` requirement with the corresponding design point using the `Model` property.

```
for ct=1:num_design_points
    R(ct) = TuningGoal.LoopShape('u',wc(ct));
    R(ct).Model = ct;
end
```

- Incorporate the varying portion of the requirement into the closed-loop model of the control system. For example, suppose you want to limit the gain from `d` to `y` to a quantity,  $\gamma$ , that depends on the scheduling variables. Suppose that `T0` is an array of models of the closed-loop system at each design point, and suppose you have created a table, `gmax`, of  $\gamma$  values for each design point,  $\sigma$ . Then you can add a new output `ys = y/\gamma` to the closed-loop model, as follows:

```
% Create array of scalar gains 1/gamma
yScaling = reshape(1./gmax,[1 1 size(gmax)]);
yScaling = ss(yScaling,'InputName','y','OutputName','ys');

% Add these gains in series to y output of T0
T0 = connect(T0,yScaling,T0.InputName,[T0.OutputName ; {'ys'}]);
```

The value of  $\gamma$  changes at each design point according to the table `gmax`. You can then use a single requirement that limits to 1 the gain from `d` to the scaled output `ys`.

```
R = TuningGoal.Gain('d','ys',1);
```

Such effective normalization of requirements moves the requirement variability from the requirement object, `R`, to the closed-loop model, `T0`.

In Simulink, you can use a similar approach by feeding the relevant model inputs and outputs through a gain block. Then, when you linearize the model, change the gain value of the block with the operating condition. For example, set the gain to a MATLAB variable, and use the `Parameters` property in `sLinearizer` to change the variable's value with each linearization condition.

## Tunable Gain Surface With Two Scheduling Variables

This example shows how to model a scalar gain  $K$  with a bilinear dependence on two scheduling variables,  $\alpha$  and  $V$ , as follows:

$$K(\alpha_N, V_N) = K_0 + K_1\alpha_N + K_2V_N + K_3\alpha_NV_N.$$

For this example,  $\alpha$  is an angle of incidence that ranges from 0 to 15 degrees, and  $V$  is a speed that ranges from 300 to 600 m/s. The coefficients  $K_0, \dots, K_3$  are the tunable parameters of this variable gain.

Create a grid of design points,  $(\alpha, V)$ , that are linearly spaced in  $\alpha$  and  $V$ . These design points are where you will tune the gain surface coefficients.

```
[alpha,V] = ndgrid(0:5:15,300:100:600);
```

These arrays, `alpha` and `V`, represent the independent variation of the two scheduling variables, each across its full range.

When you tune the gain surface coefficients with `systemtune`, you might obtain better solver performance by normalizing the scheduling variables to fall within the interval  $[-1,1]$ . Scale the  $\alpha$  and  $V$  grid to fall within this range.

```
alphaN = alpha/15;  
VN = (V-450)/150;
```

Create the tunable gain surface sampled at the grid of  $(\alpha_N, V_N)$  values:

$$K(\alpha_N, V_N) = K_0 + K_1\alpha_N + K_2V_N + K_3\alpha_NV_N.$$

In this expansion, the basis functions are:

$$\begin{aligned} F_1(\alpha_N, V_N) &= \alpha_N \\ F_2(\alpha_N, V_N) &= V_N \\ F_3(\alpha_N, V_N) &= \alpha_NV_N. \end{aligned}$$

Specify the values of the basis functions over the  $(\alpha_N, V_N)$ .

```
F1 = alphaN;
F2 = VN;
F3 = alphaN.*VN;
```

```
K = gainsurf('K',1,F1,F2,F3)
```

```
K =
```

```
4x4 array of generalized matrices with 1 rows, 1 columns, and the following
  K_0: Scalar parameter, 1 occurrences.
  K_1: Scalar parameter, 1 occurrences.
  K_2: Scalar parameter, 1 occurrences.
  K_3: Scalar parameter, 1 occurrences.
```

Type "double(K)" to see the current value, "get(K)" to see all properties,

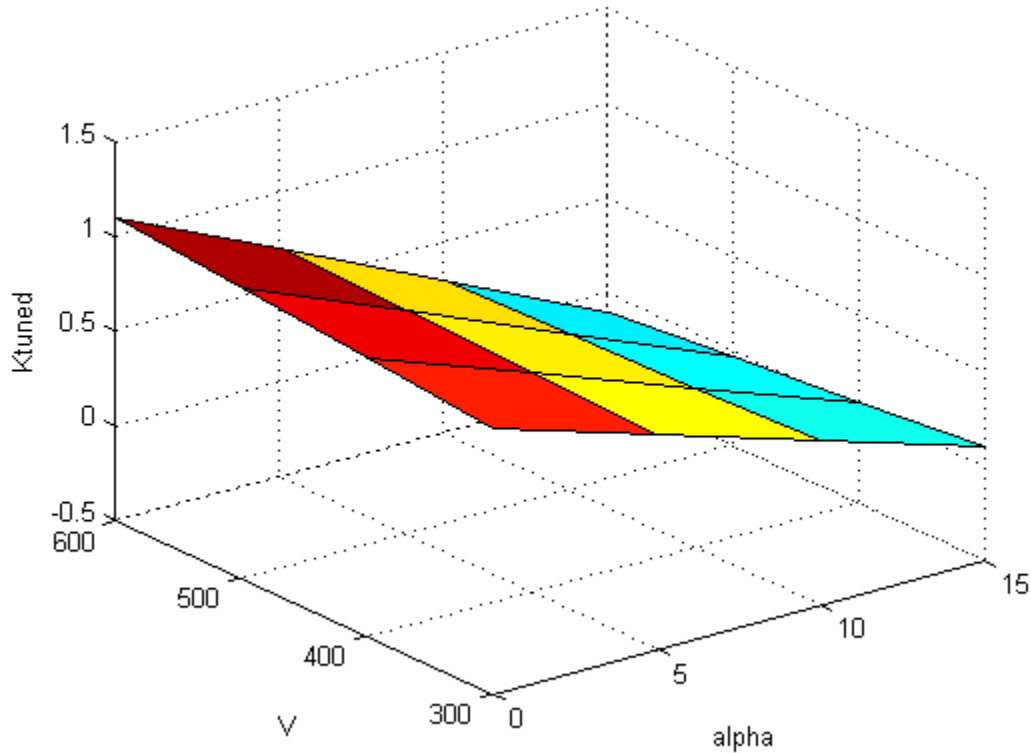
K is an array of generalized matrices. Each element in K corresponds to  $K(a_N, V_N)$  for a particular  $(a_N, V_N)$  pair, and depends on the tunable coefficients  $K_0, \dots, K_3$ .

Associate the independent variable values with the corresponding values of K.

```
K.SamplingGrid = struct('alpha',alpha,'V',V);
```

The `SamplingGrid` property keeps track of the scheduling variable values associated with each entry in K. This association is convenient for tracing results back to independent variable values. For instance, you can use `view(K)` to inspect the tuned values of the gain surface after tuning. When you do so, `view` takes the axis range and labels from the entries in `SamplingGrid`. For this example, instead of tuning, manually set the values of the tunable blocks to non-zero values. View the resulting gain surface as a function of the scheduling variables.

```
values = struct('K_0',1,'K_1',-1,'K_2',0.1,'K_3',-0.2);
Ktuned = setBlockValue(K,values);
view(Ktuned)
```



The variable names and values that you specified in the `SamplingGrid` property are used to scale and label the axes.

You can use `K` as a tunable gain to build a control system with gain-scheduled tunable components. For example, use `K` to create a gain-scheduled low-pass filter.

```
F = tf(K,[1 K]);
```

You can use gain surfaces as arguments to model creation commands like `tf` the same way you would use numeric arguments. The resulting filter is a generalized state-space (`genss`) model array that depends on the four coefficients of the gain surface.

Use model interconnection commands (such as `connect` and `feedback`) to combine `F` with an array of plant models sampled at the same values of  $\alpha$  and  $V$ . You can then use `systemtune` to tune the gain-scheduled controller to meet your design requirements. Because you normalized the scheduling variables to model the tunable gain, you must adjust the coefficient values in the implementation of your tuned controller.

## **Related Examples**

- “Gain-Scheduled PID Controller” on page 7-22
- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 7-24

## **Concepts**

- “Parametric Gain Surfaces” on page 7-8
- “Plant Models for Gain-Scheduled Control” on page 7-4

## Gain-Scheduled PID Controller

This example shows how to create a gain-scheduled PID controller. The controller's proportional, integral, and derivative gains have linear, quadratic, and constant dependence on a scheduling variable,  $\alpha$  as follows:

$$k_p(\alpha) = k_{p0} + k_{p1}\alpha$$

$$k_i(\alpha) = k_{i0} + k_{i1}\alpha + k_{i2}\alpha^2$$

$$k_d(\alpha) = k_{d0}.$$

For tuning purposes, sample these gain formulas for a number of values of  $\alpha$ . For example, sample the gain formulas for five values of  $\alpha$  (five design points) in  $[-1,1]$ .

Create a vector of  $\alpha$  values.

```
alpha = -1:0.5:1;
```

The PID gains depend on  $\alpha$  and  $\alpha^2$ . Evaluate these functions.

```
F1 = alpha;  
F2 = alpha.^2;
```

Create the tunable gain surfaces that represent the proportional, integral, and derivative gains over the grid of  $\alpha$  values. Initialize all the constant coefficients to 1.

```
Kp = gainsurf('Kp',1,F1);  
Ki = gainsurf('Ki',1,F1,F2);  
Kd = gainsurf('Kd',1);
```

Each of  $K_p$ ,  $K_i$ , and  $K_d$  is an array of generalized matrices. The array element  $K_p(:, :, 1)$  (for example) corresponds to  $k_p(\alpha)$  for  $\alpha = -1$ . The element  $K_p(:, :, 2)$  corresponds to  $k_p(\alpha)$  for  $\alpha = -0.5$ , and so on. These values depend on tunable coefficients  $K_{p\_0}$ , and  $K_{p\_1}$ . Thus, each of  $K_p$ ,  $K_i$ , and  $K_d$  represents a gain curve in  $\alpha$  with the appropriate number of tunable coefficients.

Combine the tunable gains into a PID controller. Set the derivative filter time constant to a fixed value of 0.1.



```
Tf = 0.1;
C = pid(Kp,Ki,Kd,Tf)
```

```
C =
```

```
1x5 array of generalized continuous-time state-space models.
Each model has 1 outputs, 1 inputs, 2 states, and the following blocks:
  Kd: Scalar parameter, 1 occurrences.
  Ki_0: Scalar parameter, 1 occurrences.
  Ki_1: Scalar parameter, 1 occurrences.
  Ki_2: Scalar parameter, 1 occurrences.
  Kp_0: Scalar parameter, 1 occurrences.
  Kp_1: Scalar parameter, 1 occurrences.
```

Type "ss(C)" to see the current value, "get(C)" to see all properties, and

You can use generalized matrices as arguments to model creation commands like `pid` and `tf` the same way you would use numeric arguments. The resulting controller is a generalized state-space (`genss`) model array that depends on the six coefficients of the gain curves. Each entry in the array is a PID controller evaluated at a particular value of  $a$ .

Associate the  $a$  values with the corresponding entries of `C`.

```
SG = struct('alpha',alpha);
C.SamplingGrid = SG;
```

Use model interconnection commands (such as `connect` and `feedback`) to combine `C` with an array of plant models sampled at the same values of  $a$ . You can then use `system` to tune the gain-scheduled controller to meet your design requirements.

## Related Examples

- “Tunable Gain Surface With Two Scheduling Variables” on page 7-18
- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 7-24

## Concepts

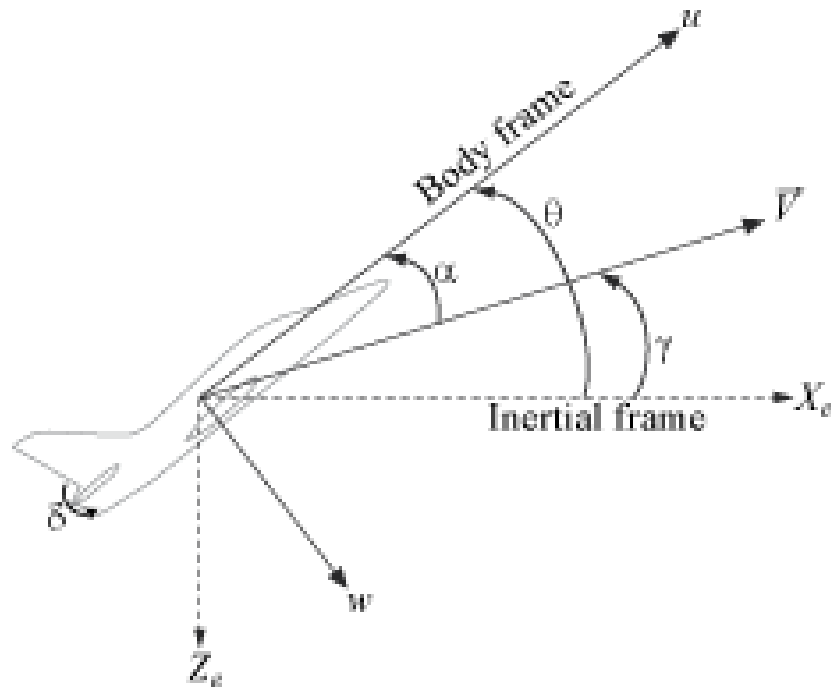
- “Parametric Gain Surfaces” on page 7-8
- “Plant Models for Gain-Scheduled Control” on page 7-4

## Tuning of Gain-Scheduled Three-Loop Autopilot

This example uses `systune` to generate smooth gain schedules for a three-loop autopilot.

### Airframe Model and Three-Loop Autopilot

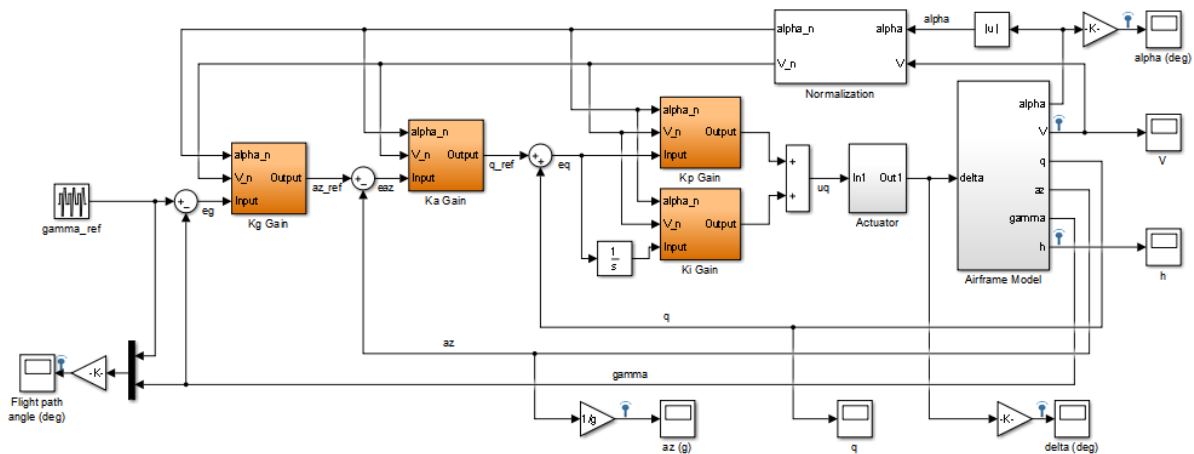
This example uses a three-degree-of-freedom model of the pitch axis dynamics of an airframe. The states are the Earth coordinates  $(X_e, Z_e)$ , the body coordinates  $(u, w)$ , the pitch angle  $\theta$ , and the pitch rate  $q = \dot{\theta}$ . Figure 1 summarizes the relationship between the inertial and body frames, the flight path angle  $\gamma$ , the incidence angle  $\alpha$ , and the pitch angle  $\theta$ .



**Figure 1: Airframe dynamics.**

We use a classic three-loop autopilot structure to control the flight path angle  $\gamma$ . This autopilot adjusts the flight path by delivering adequate bursts of normal acceleration  $a_z$  (acceleration along  $w$ ). In turn, normal acceleration is produced by adjusting the elevator deflection  $\delta$  to cause pitching and vary the amount of lift. The autopilot uses Proportional-Integral (PI) control in the pitch rate loop  $q$  and proportional control in the  $a_z$  and  $\gamma$  loops. The closed-loop system (airframe and autopilot) are modeled in Simulink.

```
open_system('rct_airframeGS')
```



### Autopilot Gain Scheduling

The airframe dynamics are nonlinear and the aerodynamic forces and moments depend on speed  $V$  and incidence  $\alpha$ . To obtain suitable performance throughout the  $(\alpha, V)$  flight envelope, the autopilot gains must be adjusted as a function of  $\alpha$  and  $V$  to compensate for changes in plant dynamics. This adjustment process is called "gain scheduling" and  $\alpha, V$  are called the scheduling variables.

Gain scheduling is a linear technique for controlling nonlinear or time-varying plants. The idea is to compute linear approximations of the plant at various operating conditions, tune the controller gains at each operating condition, and swap gains as a function of operating condition during operation. Conventional gain scheduling involves three major steps:

- 1 Trim and linearize the plant at each operating condition
- 2 Tune the controller gains for the linearized dynamics at each operating condition
- 3 Reconcile the gain values to provide smooth transition between operating conditions.

In this example, we combine Steps 2. and 3. by parameterizing the autopilot gains as first-order polynomials in  $\alpha, V$  and directly tuning the polynomial coefficients for the entire flight envelope. This approach eliminates Step 3. and guarantees smooth gain variations as a function of  $\alpha$  and  $V$ . Moreover, the gain schedule coefficients can be automatically tuned with `systune`.

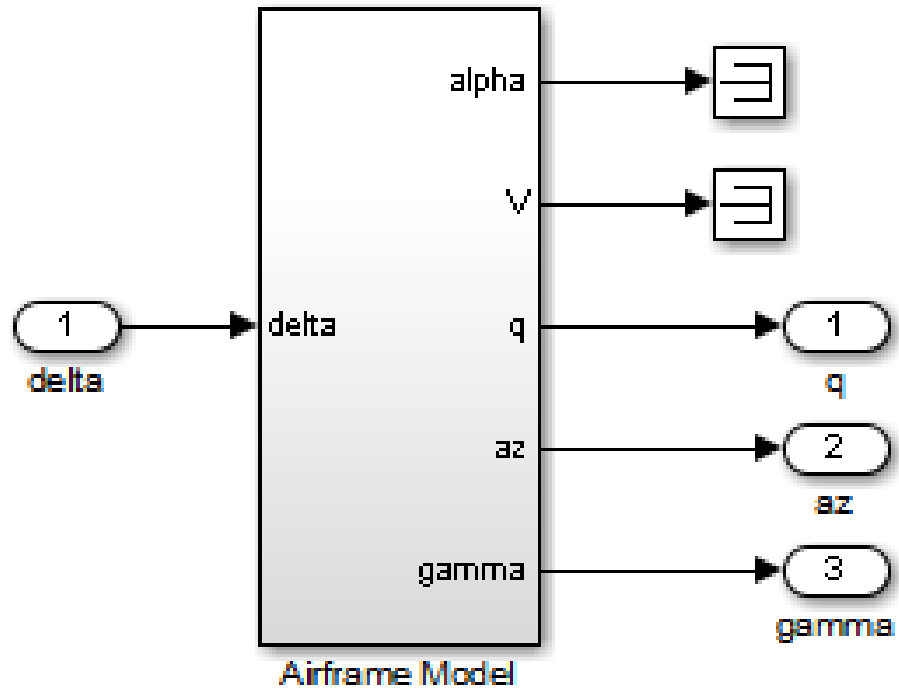
### Trimming and Linearization

Assume that the incidence  $\alpha$  varies between -20 and 20 degrees and that the speed  $V$  varies between 700 and 1400 m/s. When neglecting gravity, the airframe dynamics are symmetric in  $\alpha$  so consider only positive values of  $\alpha$ . Use a 5-by-9 grid of linearly spaced  $(\alpha, V)$  pairs to cover the flight envelope:

```
nA = 5; % number of alpha values
nV = 9; % number of V values
[alpha,V] = ndgrid(linspace(0,20,nA)*pi/180,linspace(700,1400,nV));
```

For each flight condition  $(\alpha, V)$ , linearize the airframe dynamics at trim (zero normal acceleration and pitching moment). This requires computing the elevator deflection  $\delta$  and pitch rate  $q$  that result in steady  $w$  and  $q$ . To do this, first isolate the airframe model in a separate Simulink model.

```
open_system('rct_airframeTRIM')
```



Use `operspec` to specify the trim condition, use `findop` to compute the trim values of  $\delta$  and  $q$ , and linearize the airframe dynamics for the resulting operating point. See the "Trimming and Linearizing an Airframe" example in Simulink Control Design for details. Repeat these steps for the 45 flight conditions  $(\alpha, V)$ .

```
% Compute trim condition for each (alpha,V) pair
clear op
for ct=1:nA*nV
    alpha_ini = alpha(ct);    % Incidence [rad]
    v_ini = V(ct);           % Speed [m/s]

    % Specify trim condition
    opspec = operspec('rct_airframeTRIM');
    % Xe,Ze: known, not steady
```

```

    opspec.States(1).Known = [1;1];
    opspec.States(1).SteadyState = [0;0];
    % u,w: known, w steady
    opspec.States(3).Known = [1 1];
    opspec.States(3).SteadyState = [0 1];
    % theta: known, not steady
    opspec.States(2).Known = 1;
    opspec.States(2).SteadyState = 0;
    % q: unknown, steady
    opspec.States(4).Known = 0;
    opspec.States(4).SteadyState = 1;

    % TRIM
    Options = findopOptions('DisplayReport','off');
    op(ct) = findop('rct_airframeTRIM',opspec,Options);
end

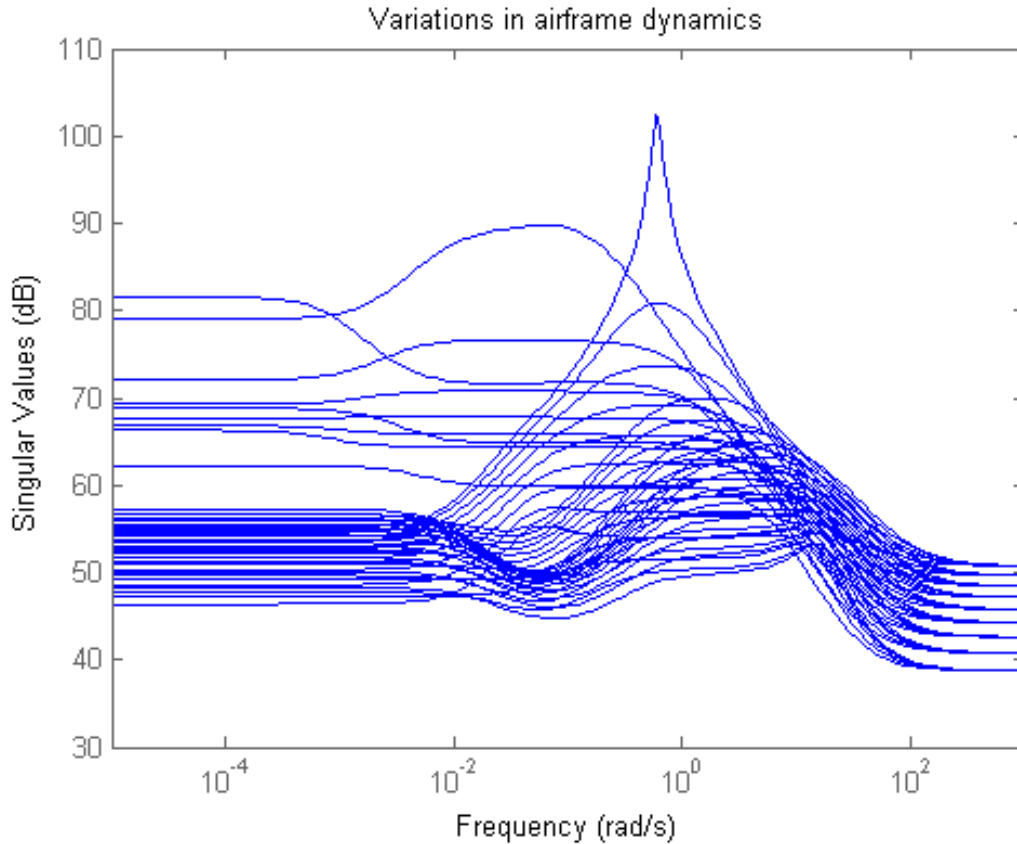
% Linearization I/Os
io = [linio('rct_airframeTRIM/delta',1,'in');... % delta
      linio('rct_airframeTRIM/Airframe Model',3,'out');... % q
      linio('rct_airframeTRIM/Airframe Model',4,'out');... % az
      linio('rct_airframeTRIM/Airframe Model',5,'out')]; % gamma

% Linearize at trim conditions
G = linearize('rct_airframeTRIM',op,io);
G = reshape(G,[nA nV]);
G.u = 'delta';
G.y = {'q' 'az' 'gamma'};

```

This produces a 5-by-9 array of linearized plant models at the 45 flight conditions  $(\alpha, V)$ . The plant dynamics vary substantially across the flight envelope.

```
sigma(G), title('Variations in airframe dynamics')
```



### Autopilot Modeling

The autopilot consists of four gains that must be "scheduled" (adjusted) as a function of  $\alpha$  and  $V$ . Each "gain" is therefore a two-dimensional gain surface. As a first cut, seek gain surfaces with a simple multi-linear dependence on  $\alpha$  and  $V$ :

$$K(\alpha, V) = K_0 + K_1\alpha + K_2V + K_3\alpha V.$$

These gain surfaces are modeled in Simulink as follows:

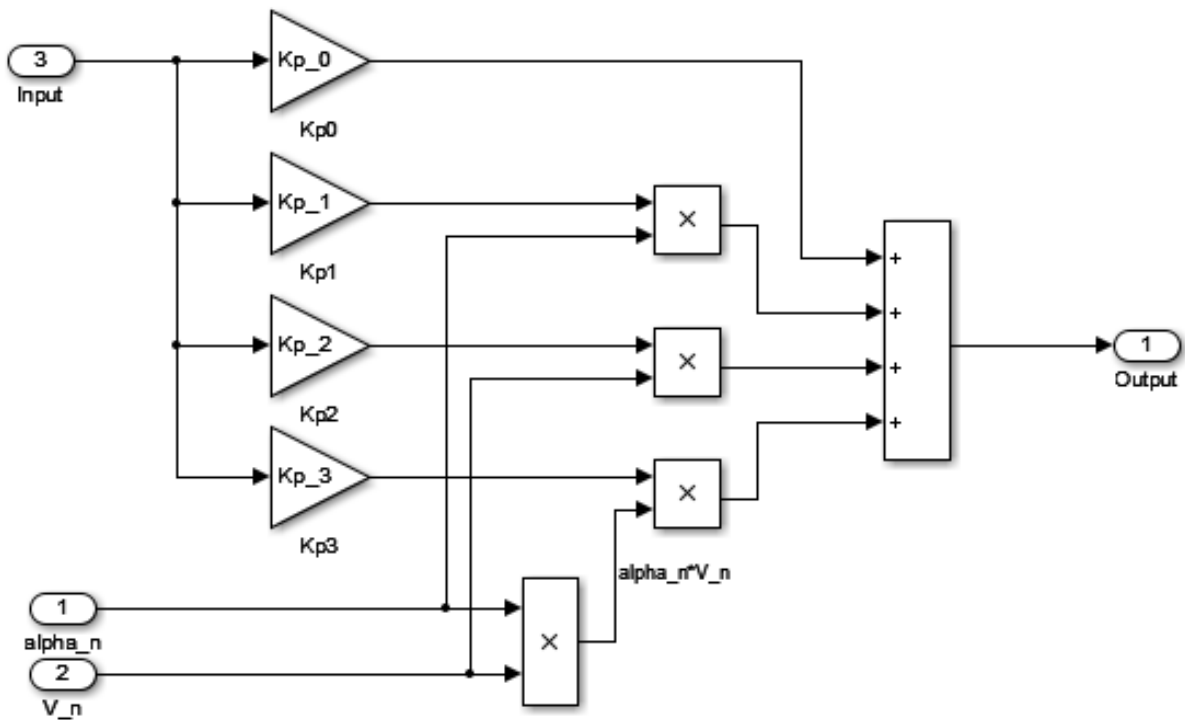


Figure 2: Simulink model for the gain surface  $K_p(\alpha, V)$ .

Use `gainsurf` to parameterize these gain surfaces in terms of the tunable coefficients  $K_0, K_1, K_2, K_3$ . To improve convergence of the tuning algorithm, it is recommended to replace  $\alpha, V$  by normalized values ranging in  $[-1,1]$ :

```
% Subtract the mean value of alpha and divide by its half range
Mean = 10*pi/180; % mean = 10 degrees
HalfRange = Mean; % half range = 10 degrees
alpha_n = (alpha - Mean)/HalfRange; % varies in [-1,1]

% Subtract the mean value of V and divide by its half range
Mean = (700+1400)/2;
HalfRange = (1400-700)/2;
```



```

V_n = (V - Mean)/HalfRange;    % varies in [-1,1]

% Create gain surfaces
Kp = gainsurf('Kp', 0.1, alpha_n, V_n, alpha_n.*V_n);
Ki = gainsurf('Ki', 2, alpha_n, V_n, alpha_n.*V_n);
Ka = gainsurf('Ka', 0.001, alpha_n, V_n, alpha_n.*V_n);
Kg = gainsurf('Kg', -1000, alpha_n, V_n, alpha_n.*V_n);

```

The initial values for the constant coefficient  $K_0$  are based on tuning results for  $\alpha = 10$  deg and  $V = 1000$  m/s (mid-range design). Note that Kp, Ki,... are arrays of matrices (one per flight condition) parameterized by the tunable coefficients  $K_j$ . To facilitate visualization, use the `SamplingGrid` property to keep track of the dependence on  $\alpha, V$ :

```

% Grid of alpha,V values
SG = struct('alpha',alpha,'V',V);

Kp.SamplingGrid = SG;
Ki.SamplingGrid = SG;
Ka.SamplingGrid = SG;
Kg.SamplingGrid = SG;

```

Next construct a closed-loop model T0 by connecting the linearized airframe model G, the second-order actuator model, and the autopilot gains Kp, Ki, Ka, Kg according to the block diagram `rct_airframeGS`. The software lets you manipulate the model arrays G, Kp, Ki,... as single entities (one block in the block diagram).

```

% Actuator model
Act = tf(150^2,[1 2*0.7*150 150^2]);
Act.u = 'u'; Act.y = 'delta_d';

% Controller elements
Cq = pid(Kp,Ki);
Cq.u = 'eq'; Cq.y = 'u';
Ca = ss(Ka);
Ca.u = 'eaz'; Ca.y = 'q_ref';
Cg = ss(Kg);
Cg.u = 'eg'; Cg.y = 'az_ref';

```

```
% Summing junctions
S1 = sumblk('delta = delta_d + d');
S2 = sumblk('eq = q_ref + q');
S3 = sumblk('eaz = az_ref - az');
S4 = sumblk('eg = gamma_ref - gamma');

% Connect the blocks together
T0 = connect(G,Cq,Ca,Cg,S1,S2,S3,S4,Act,...
    {'gamma_ref','az_ref','d'},{'gamma','az'})

T0 =

5x9 array of generalized continuous-time state-space models.
Each model has 2 outputs, 3 inputs, 7 states, and between 4 and 16 blocks

Type "ss(T0)" to see the current value, "get(T0)" to see all properties, and
```

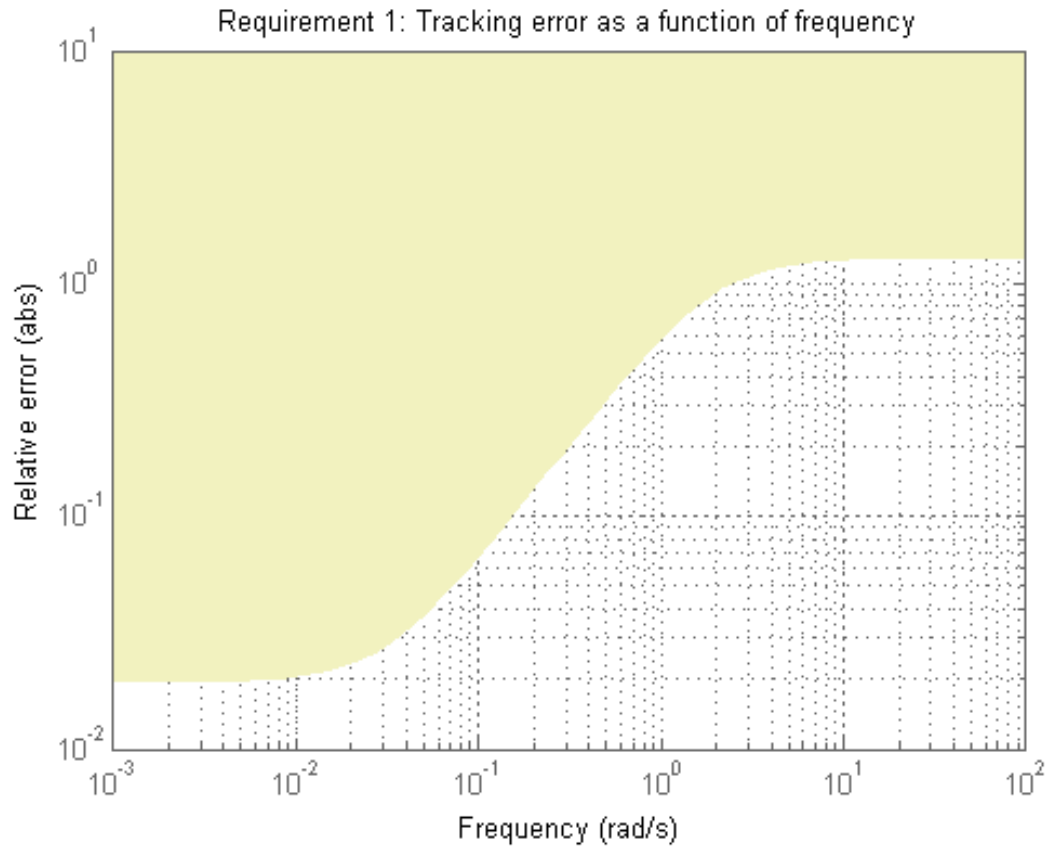
This creates a 5-by-9 array of closed-loop models parameterized by the tunable coefficients of the four gain surfaces.

### Autopilot Tuning

`systeme` can automatically tune the gain surface coefficients for the entire flight envelope. Use `TuningGoal` objects to specify the performance objectives:

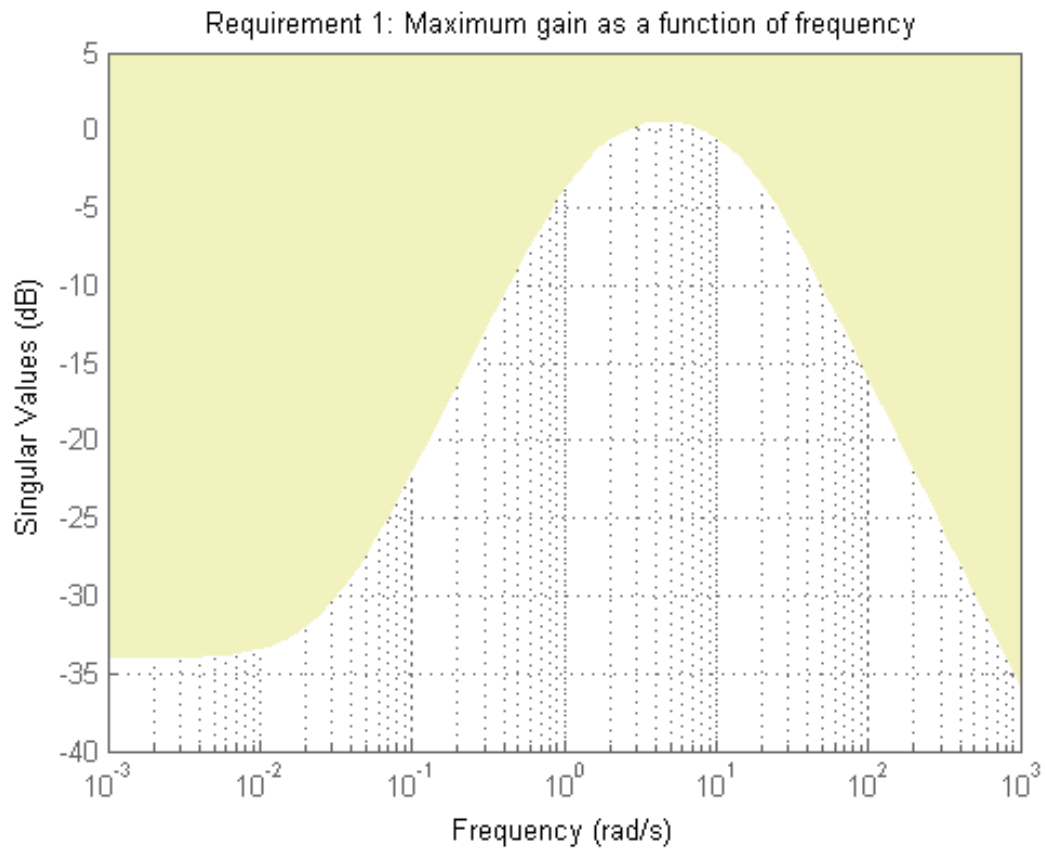
- $\gamma$  loop: Ensure that the tracking error is less than 2% at DC and remains small up to 1 rad/s for a response time of about 1 second.

```
Req1 = TuningGoal.Tracking('gamma_ref','gamma',1.3*tf([1 0.03],[1 2]));
viewSpec(Req1)
```



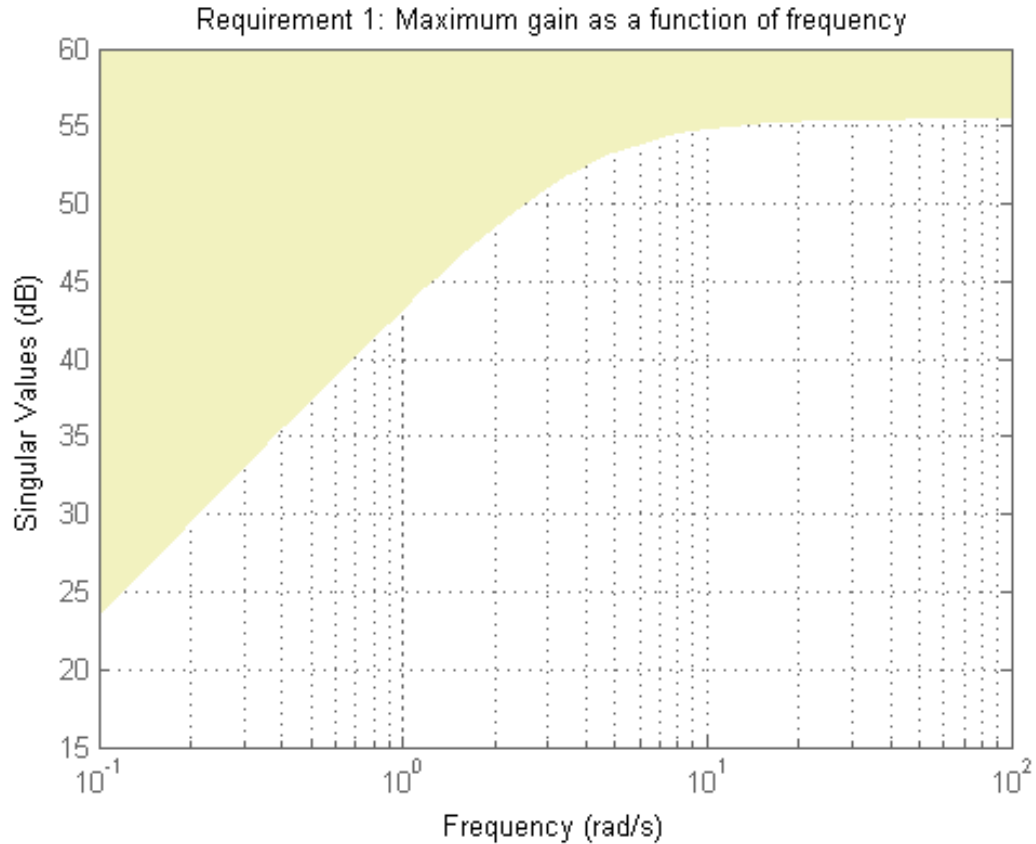
- $a_z$  loop: Ensure good disturbance rejection at low frequency (to track acceleration demands) and past 10 rad/s (to be insensitive to measurement noise).

```
% Note: The disturbance is injected at the az_ref location
RejectionProfile = frd([0.02 0.02 1.2 1.2 0.1],[0 0.02 2 15 150]);
Req2 = TuningGoal.Gain('az_ref','az',RejectionProfile);
viewSpec(Req2)
```



- $q$  loop: Ensure good disturbance rejection up to 10 rad/s

% Note: The disturbance  $d$  enters at the plant input  
`Req3 = TuningGoal.Gain('d','az',600*tf([0.25 0],[0.25 1]));`  
`viewSpec(Req3)`



- Transients: Ensure a minimum damping ratio of 0.35 for oscillation-free transients

```
Req4 = TuningGoal.Poles;
Req4.MinDamping = 0.35;
```

Using `systemtune`, tune the 16 gain surface coefficients to best meet these performance requirements at all 45 flight conditions.

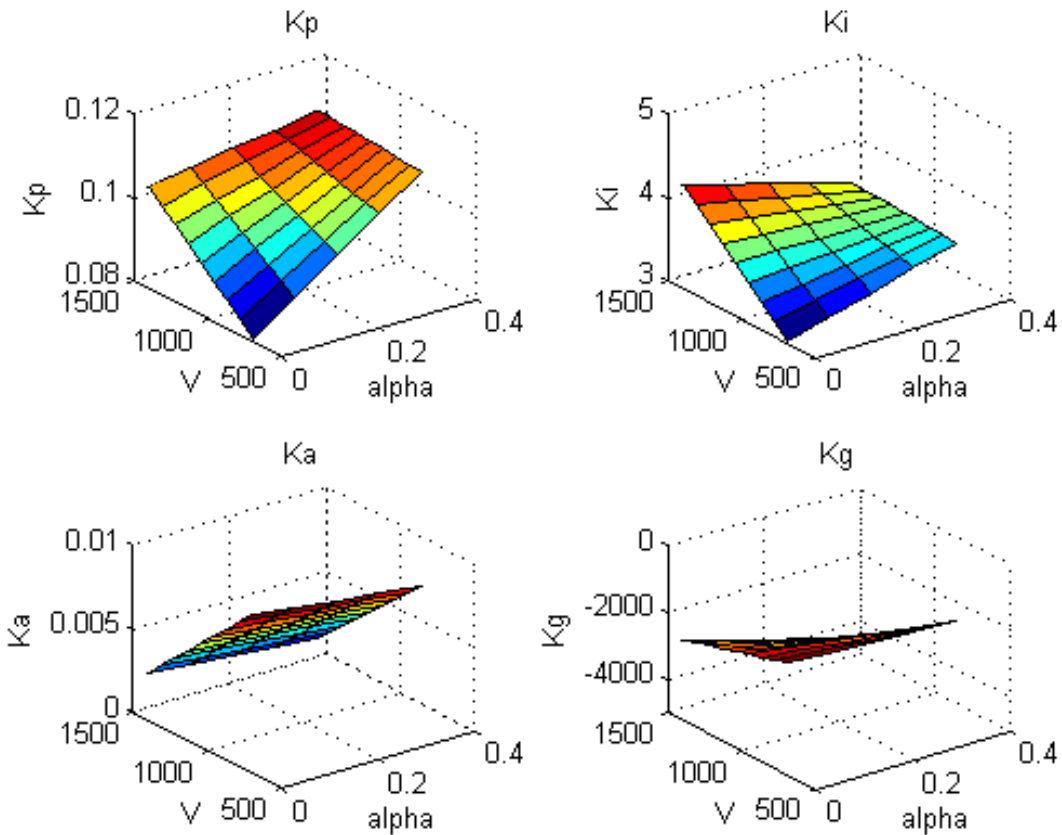
```
T = systemtune(T0,[Req1 Req2 Req3 Req4]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 66
```

The final value of the combined objective is close to 1, indicating that all requirements are nearly met. Visualize the resulting gain surfaces.

```
% Use the tuned values of the gain surface coefficients
Kp = setBlockValue(Kp,T);
Ki = setBlockValue(Ki,T);
Ka = setBlockValue(Ka,T);
Kg = setBlockValue(Kg,T);

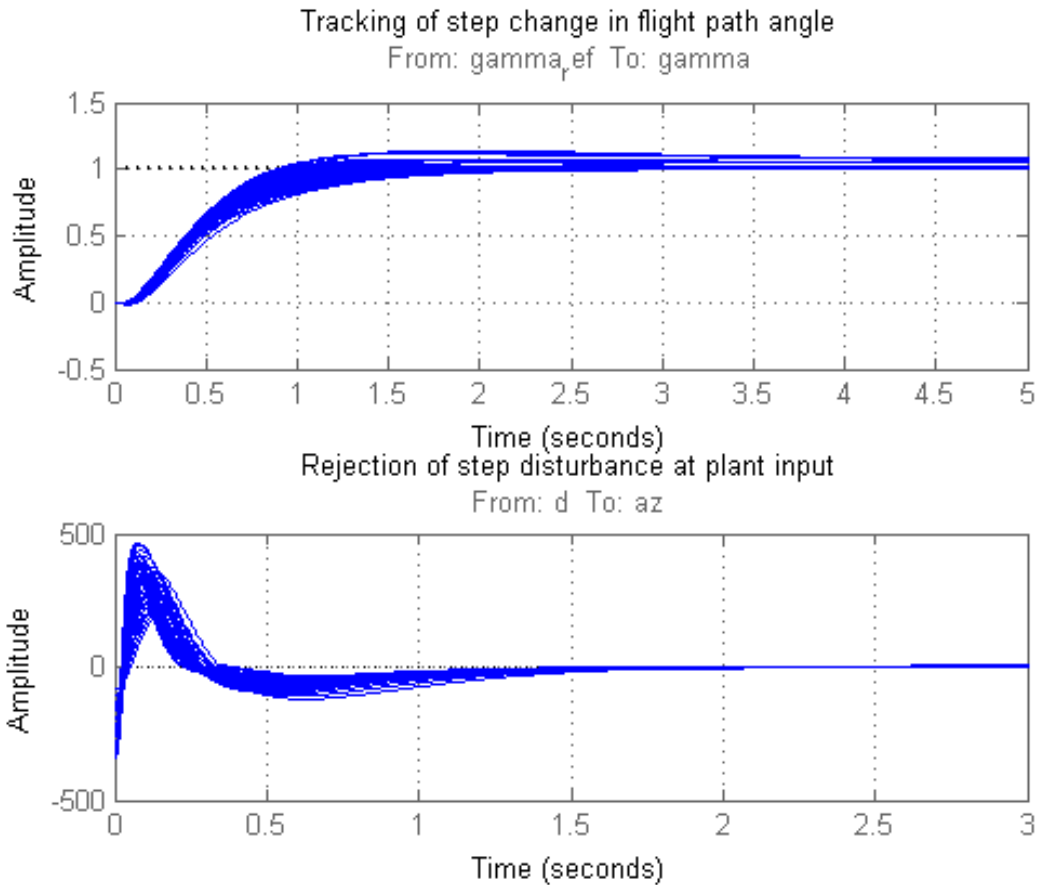
% Plot gain surfaces
clf
subplot(221), view(Kp), title('Kp')
subplot(222), view(Ki), title('Ki')
subplot(223), view(Ka), title('Ka')
subplot(224), view(Kg), title('Kg')
```



### Validation

First validate the tuned autopilot at the 45 flight conditions considered above. Plot the response to a step change in flight path angle and the response to a step disturbance in elevator deflection.

```
clf
subplot(211), step(T('gamma','gamma_ref'),5), grid
title('Tracking of step change in flight path angle')
subplot(212), step(T('az','d'),3), grid
title('Rejection of step disturbance at plant input')
```



The responses are satisfactory at all flight conditions. Next validate the autopilot against the nonlinear airframe model. First push the tuned gain surface coefficients to the Simulink model. The coefficients for the "Kp Gain" block are named "Kp\_0", "Kp\_1", "Kp\_2", "Kp\_3", and similarly for the other gain blocks.

```
[Kp_0,Kp_1,Kp_2,Kp_3] = gainsurfddata(Kp);
[Ki_0,Ki_1,Ki_2,Ki_3] = gainsurfddata(Ki);
[Ka_0,Ka_1,Ka_2,Ka_3] = gainsurfddata(Ka);
[Kg_0,Kg_1,Kg_2,Kg_3] = gainsurfddata(Kg);
```



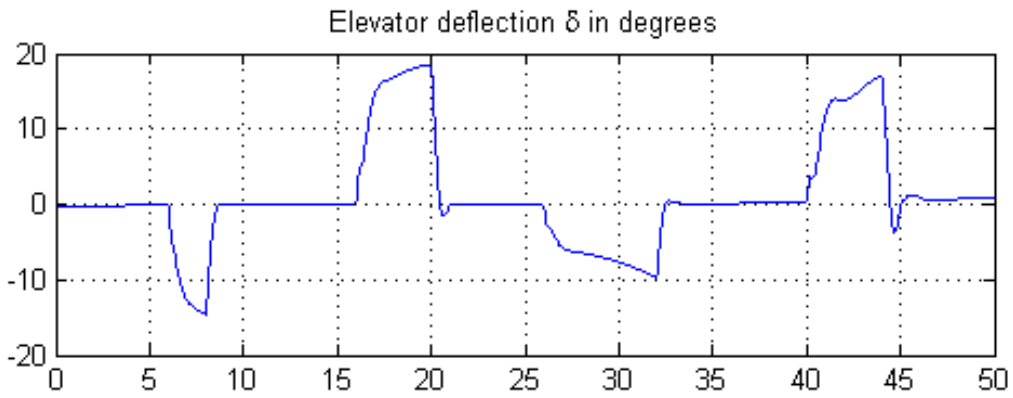
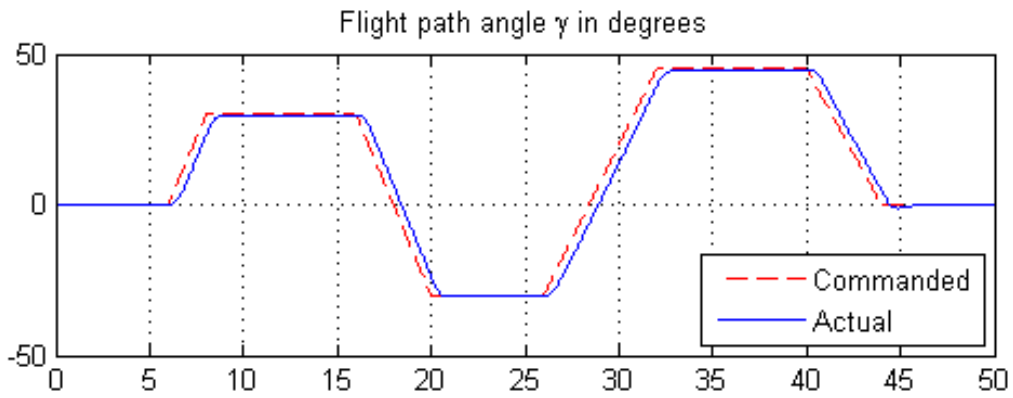
Now simulate the autopilot performance for a maneuver that takes the airframe through a large portion of its flight envelope.

```
% Initial conditions
h_ini = 1000;
alpha_ini = 0;
v_ini = 700;

% Simulate
SimOut = sim('rct_airframeGS', 'ReturnWorkspaceOutputs', 'on');

% Extract simulation data
SimData = get(SimOut, 'sigsOut');
Sim_gamma = getElement(SimData, 'gamma');
Sim_alpha = getElement(SimData, 'alpha');
Sim_V = getElement(SimData, 'V');
Sim_delta = getElement(SimData, 'delta');
Sim_h = getElement(SimData, 'h');
Sim_az = getElement(SimData, 'az');
t = Sim_gamma.Values.Time;

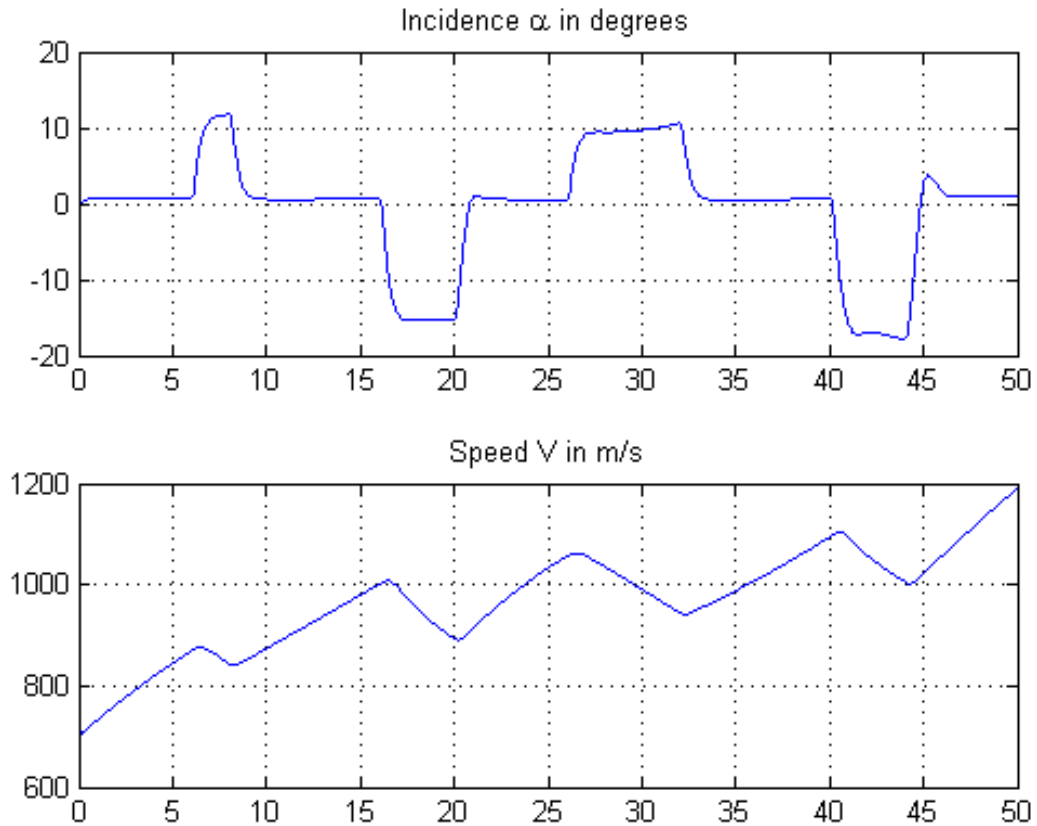
% Plot the main flight variables
clf
subplot(211)
plot(t, Sim_gamma.Values.Data(:,1), 'r--', t, Sim_gamma.Values.Data(:,2), 'b'),
legend('Commanded', 'Actual', 'location', 'SouthEast')
title('Flight path angle \gamma in degrees')
subplot(212)
plot(t, Sim_delta.Values.Data), grid
title('Elevator deflection \delta in degrees')
```



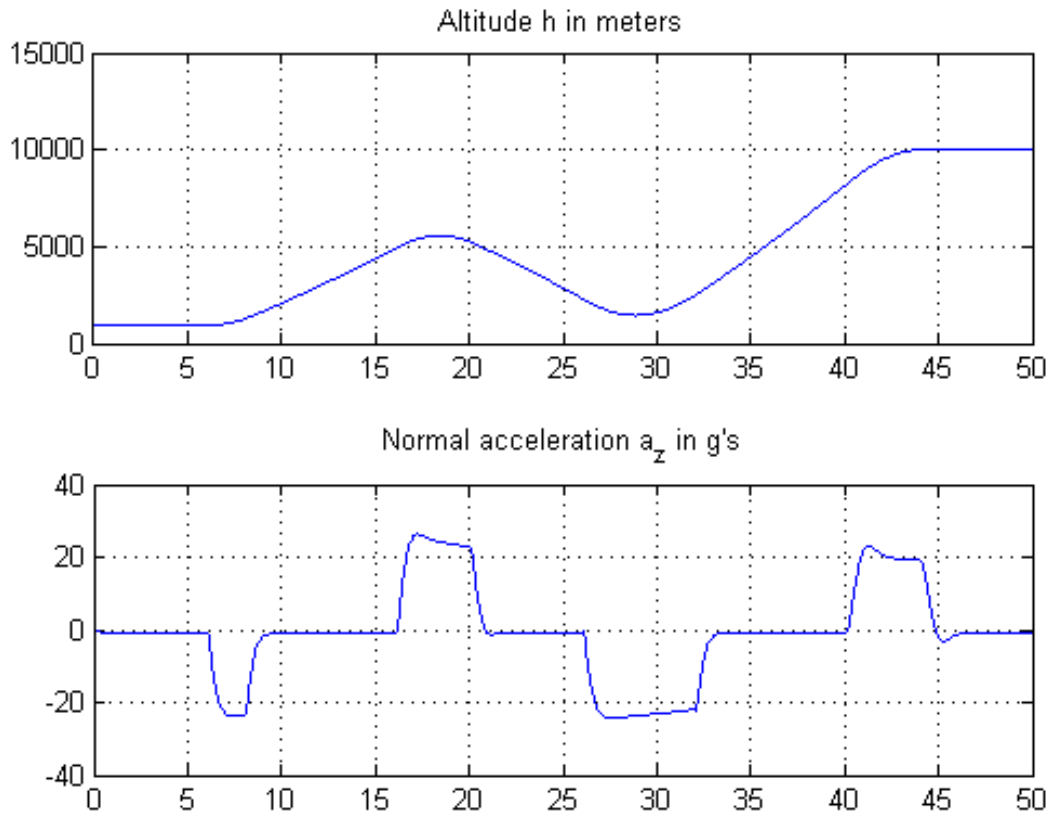
```

subplot(211)
plot(t,Sim_alpha.Values.Data), grid
title('Incidence \alpha in degrees')
subplot(212)
plot(t,Sim_V.Values.Data), grid
title('Speed V in m/s')

```



```
subplot(211)
plot(t,Sim_h.Values.Data), grid
title('Altitude h in meters')
subplot(212)
plot(t,Sim_az.Values.Data), grid
title('Normal acceleration a_z in g's')
```



Tracking of the flight path angle profile remains good throughout the maneuver. Note that the variations in incidence  $\alpha$  and speed  $V$  cover most of the flight envelope considered here ( $[-20,20]$  degrees for  $\alpha$  and  $[700,1400]$  for  $V$ ). And while the autopilot was tuned for a nominal altitude of 3000 m, it fares well despite altitude changing from 1,000 to 10,000 m.

The nonlinear simulation results confirm that the gain-scheduled autopilot delivers consistently high performance throughout the flight envelope. The simple explicit formulas for the gain dependence on the scheduling variables is

amenable to efficient hardware implementation. Alternatively, these formulas can be readily converted into 2D lookup tables for further adjustment.

### **Related Examples**

- Gain Scheduled Control Of a Chemical Reactor

### **Concepts**

- “Gain-Scheduled Control Systems” on page 7-2
- “Parametric Gain Surfaces” on page 7-8



## Symbols and Numerics

2-norm  
 definition 5-2

### A

additive error bound 3-5

### B

block diagrams  
 direction of arrows 5-4  
 bstmr 3-11

### C

complementary sensitivity 2-6  
 crossover frequency  $w_c$  2-21

### D

Delta1 object 4-15  
 Delta2 object 4-15  
 design goals  
 crossover 2-21  
 performance 2-21  
 roll-off 2-21  
 stability robustness 2-21  
 disturbance attenuation 2-7

### E

Euclidean norms 5-8

### F

feedback 4-9  
 forbidden regions 2-12  
 frequency domain uncertainty  
 adding to the model 4-15  
 fundamental limits

right-half-plane poles 2-21  
 right-half-plane zeros 2-21

### G

gain reduction tolerance 2-13  
 gain/phase margins  
 MIMO system 2-12  
 get  
 viewing the properties of the uncertain  
 system 4-14

### H

$H_\infty$   
 mixsyn 2-22  
 norm 2-5  
 $H_\infty$  control  
 performance objectives 5-10  
 $H_2$   
 norm 2-5  
 Hankel singular value  
 NCF 1-18  
 HiMAT aircraft model 1-12

### I

InputGroup property 4-15  
 InputName property 4-15

### L

$L_2$ -norm 5-2  
 linear matrix inequalities  
 LMI solvers 1-21  
 LMI solvers 1-21  
 loop shaping  
 loopsyn 2-6  
 loop transfer function matrix 2-6  
 loopmargin 4-21  
 loopsens 4-19

**M**

makeweight utility 4-7  
maxgain variable 4-12  
mixed-sensitivity cost function 2-22  
mixed-sensitivity loop shaping 2-22  
model reduction 1-16

- additive error methods 3-7
- balanced stochastic method 3-11
- large-scale models 3-16
- multiplicative error method 3-9
- normalized coprime factor truncation 3-17

model reduction routines 3-5  
models

- with uncertain real coefficients 4-4

modreal 3-16  
Monte Carlo random sample 1-7  
multiplicative (relative) error bound 3-5  
multiplicative uncertainty 2-2

**N**

ncfmr 3-17  
NominalValue property 4-15  
norm 4-24  
norms 5-2

- $H_\infty$  2-4
- $H_2$  2-4
- performance 5-3

**O**

OutputGroup property 4-15  
OutputName property 4-15

**P**

Percentage property 4-3  
performance weights 5-8  
perturbation

- additive 2-7

multiplicative 2-7

PlusMinus property 4-3

**R**

Range property 4-3  
reduce 3-7  
report variable 4-11  
robustness

- of stability 2-21

robustness analysis

- Monte Carlo 1-9
- worst case 1-9

robustperf 4-27  
robuststab 4-10  
roll-off 2-21

**S**

schurmr 3-7  
sensitivity 2-6  
singular values 2-4

- properties of 2-4

stabmarg variable 4-11

**U**

ultidyn 4-6  
uncertain elements 4-2  
uncertain LTI system 1-4  
uncertain parameters 4-3  
uncertain state-space object. *See* USS object  
uncertainty

- capturing 4-7

ureal 4-3  
USS object 1-7  
usubs

- substituting worst-case values for uncertain elements 4-12



**W**

W1 and W2 shaping filters 4-15

wcgain

    computing worst-case peak gain 4-11

wcsens 4-27

wcu variable 4-12

weighted norms 5-4

worst-case

    peak gain 1-11