Robust Control Toolbox™
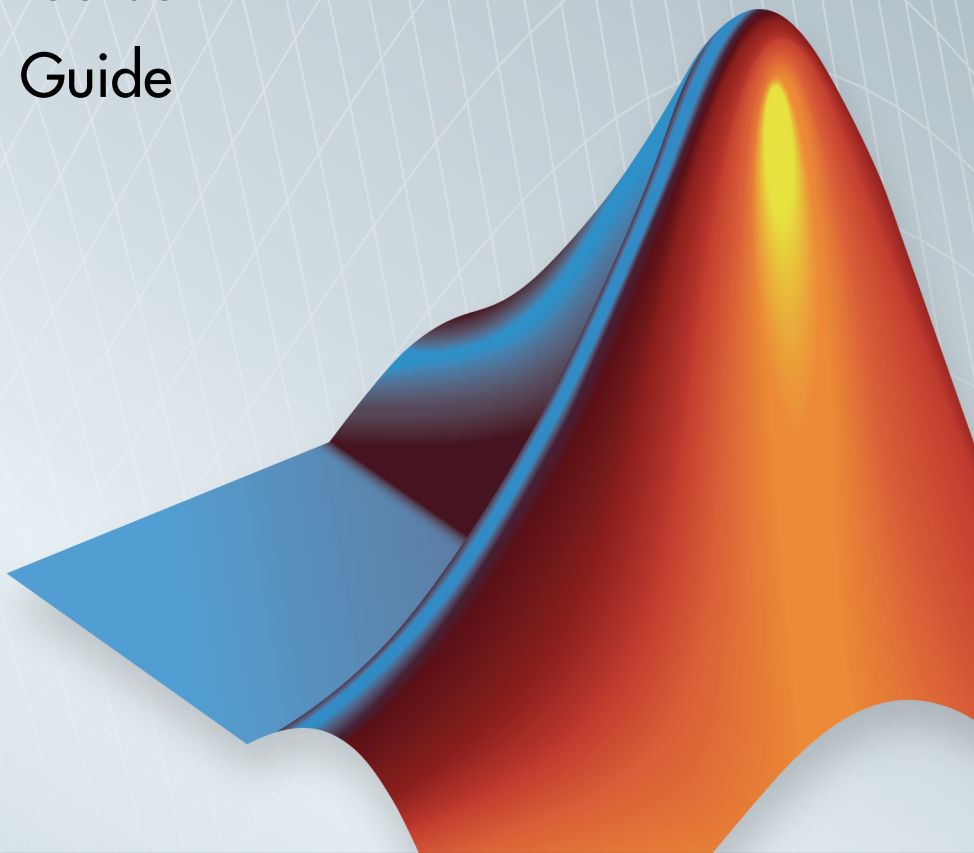
Getting Started Guide

Gary Balas
Richard Chiang
Andy Packard
Michael Safonov

MATLAB®
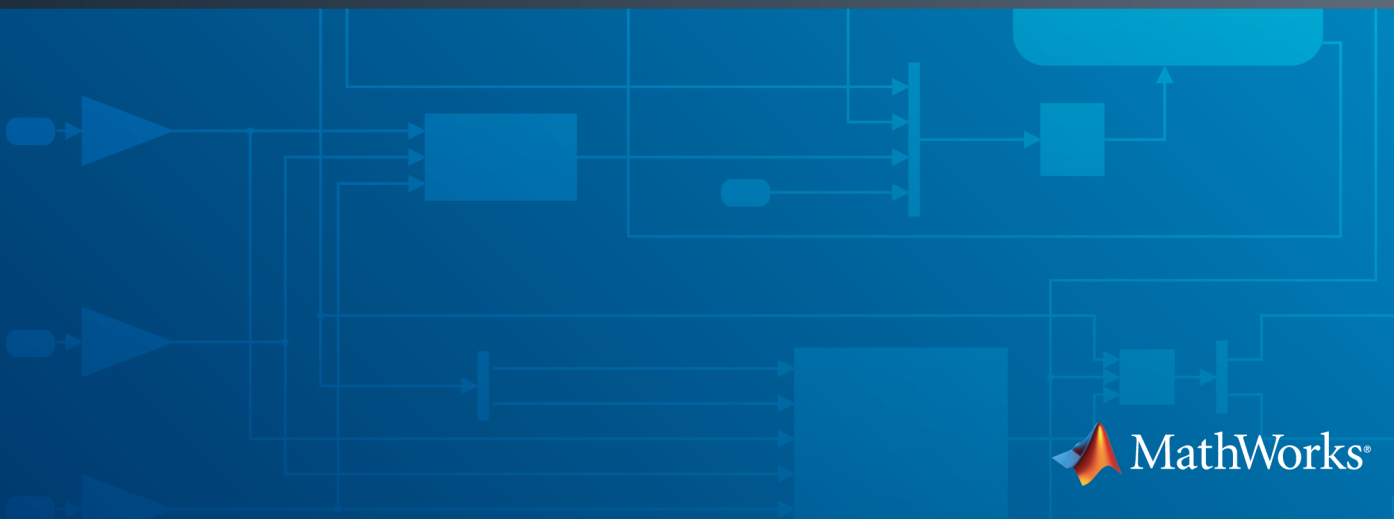
MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

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)
March 2014	Online only	Revised for Version 5.1 (Release 2014a)
October 2014	Online only	Revised for Version 5.2 (Release 2014b)

Contents

Introduction

1

Robustness Analysis

4

H-Infinity and Mu Synthesis

5

6

Tuning Fixed Control Architectures

7

Gain-Scheduled Controllers

8

1

Introduction

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.
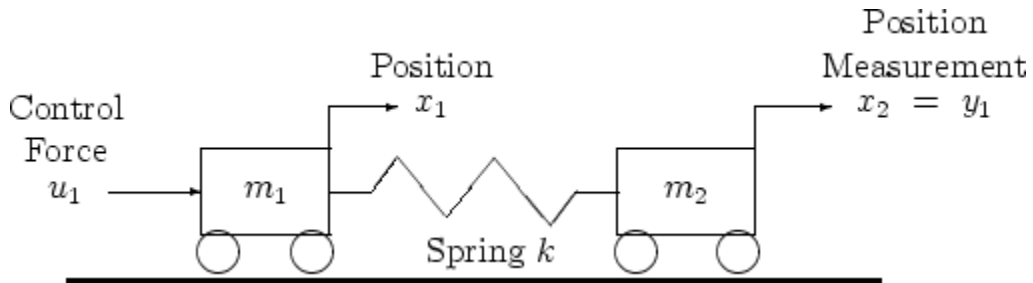
ACC Benchmark Problem

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%:

```
m₁ = 1 ± 0.2
m₂ = 1 ± 0.2
k = 1 ± 0.2
```
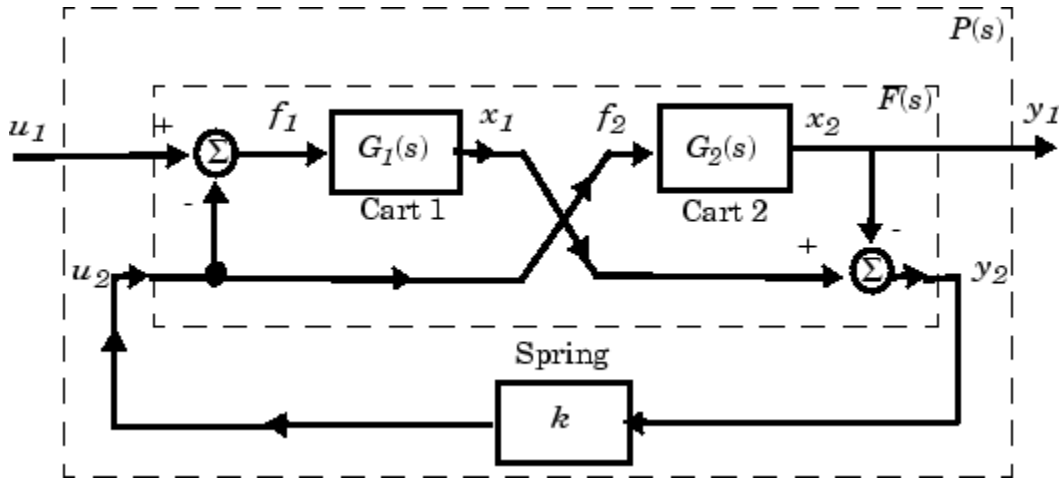
"ACC Benchmark" Two-Cart System Block Diagram $y_1 = P(s) \, u_1$

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

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

This code builds the uncertain system model P shown above:

```
m1 = ureal('m1',1,'percent',20);
m2 = ureal('m2',1,'percent',20);
k  = ureal('k',1,'percent',20);

s = zpk('s');
G1 = ss(1/s^2)/m1;
G2 = ss(1/s^2)/m2;


F = [0;G1]*[1 -1]+[1;-1]*[0,G2];
P = lft(F,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)
```

```
ans =

              1
  --------------------------
  (s^2 + 5.995e-16) (s^2 + 2)
```

```
Continuous-time zero/pole/gain model.
```

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

$$C(s) = \frac{100(s+1)^3}{(0.001s+1)^3}$$
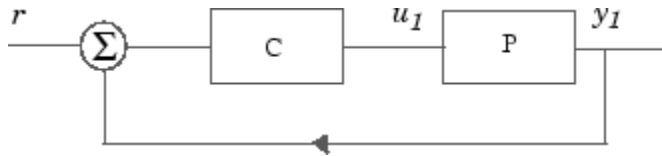
then you can form the controller and the closed-loop system $y_1 = T(s)\, u_1$ and view the closed-loop system's step response on the time interval from t=0 to t=0.1 for a Monte Carlo random sample of five combinations of the three uncertain parameters k, m1, and m2 using this code:

```
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);
```
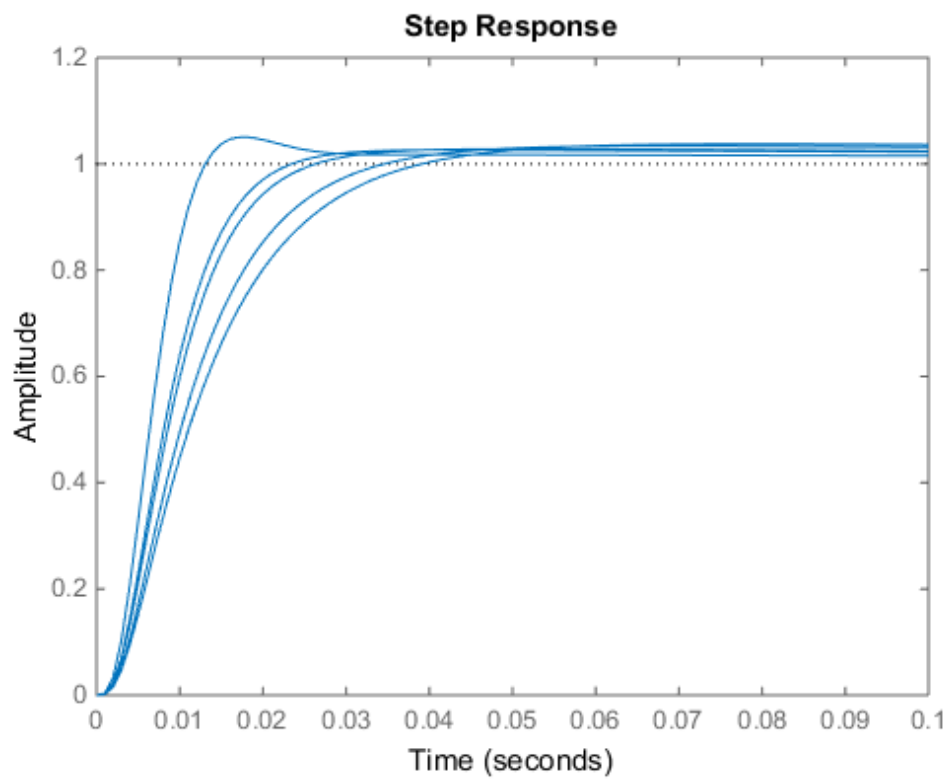
Worst-Case Performance

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 "System with Uncertain Parameters" on page 1-5, 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.

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

Worst-Case Performance of Uncertain System

This example shows how to calculate the worst-case performance of the closed-loop system described in "System with Uncertain Parameters" on page 1-5. The following commands construct that system.

```
m1 = ureal('m1',1,'percent',20);
m2 = ureal('m2',1,'percent',20);
k  = ureal('k',1,'percent',20);

s = zpk('s');
G1 = ss(1/s^2)/m1;
G2 = ss(1/s^2)/m2;


F = [0;G1]*[1 -1]+[1;-1]*[0,G2];
P = lft(F,k);

C = 100*ss((s+1)/(.001*s+1))^3;

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±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


REPORT =

Uncertain system is robustly stable to modeled uncertainty.
 -- It can tolerate up to 301% of the modeled uncertainty.
 -- A destabilizing combination of 500% of the modeled uncertainty was found.
 -- This combination causes an instability at 155 rad/seconds.
 -- Sensitivity with respect to the uncertain elements are:
     'k' is 20%.  Increasing 'k' by 25% leads to a 5% decrease in the margin.
     'm1' is 60%.  Increasing 'm1' by 25% leads to a 15% decrease in the margin.
     'm2' is 58%.  Increasing 'm2' by 25% leads to a 14% decrease in the margin.
```

The report tells you that the control system is robust for all parameter variations in the ±20% range, and that the smallest destabilizing combination of real variations in the values k, m1, and m2 has sizes somewhere between 301% and 500% greater than ±20%, i.e., between ±62.2% and ±100%. The value Udestab returns an estimate of the 500% destabilizing parameter variation combination:

```
Udestab
```

```
Udestab =

     k: 5.5122e-07
    m1: 7.3749e-07
    m2: 0.2151
```

You have a comfortable safety margin of between 311% to 500% larger than the anticipated ±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 ±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});
legend('T_{wc}    - worst-case','T_{rand} - random samples',...
       'Location','SouthWest');
```
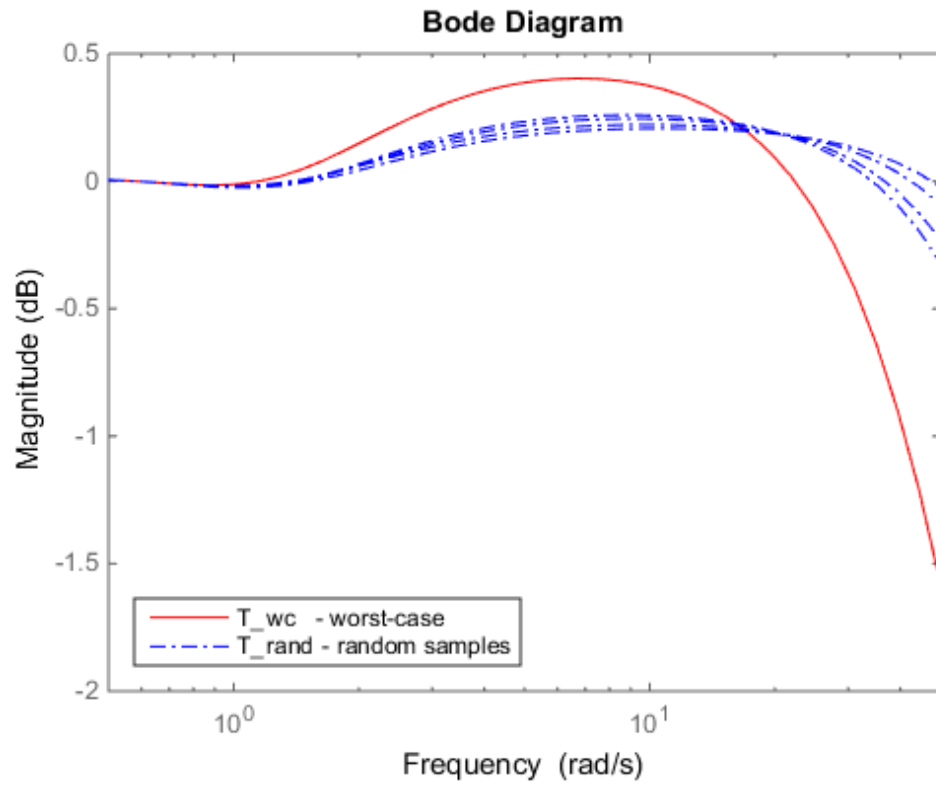
Loop-Shaping Controller Design

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 (δ_e and δ_c). The output variables are angle of attack (α) and attitude angle (θ). 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_δ are elevator and canard actuator states.
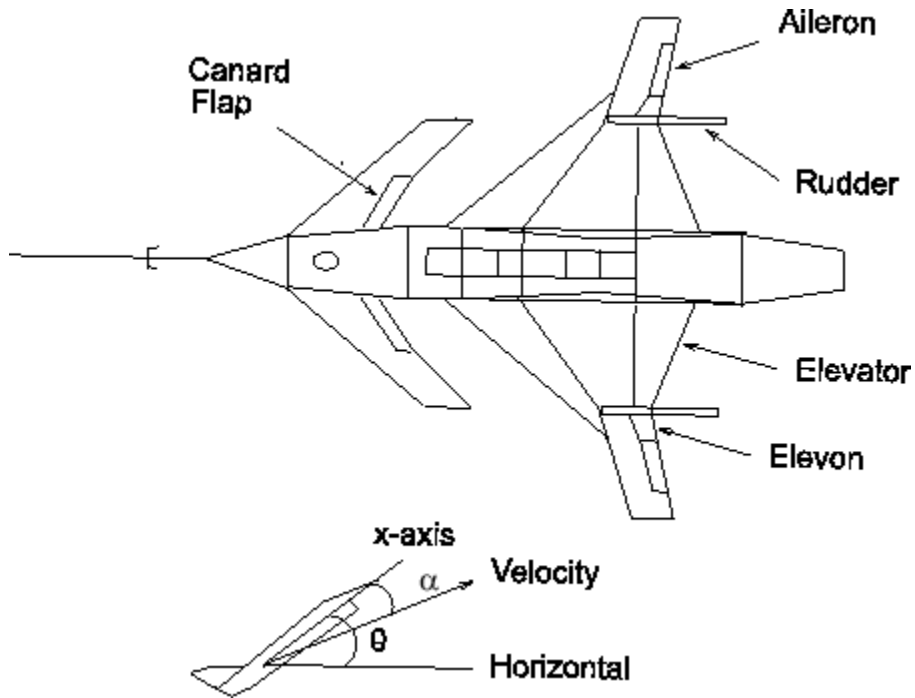
Aircraft Configuration and Vertical Plane Geometry

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

```
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);
% NASA HiMAT model G(s)
```

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})
legend('Achieved Loop Shape','Target Loop Shape','Gd/GAM','Gd*GAM')
figure
T = feedback(G*K,eye(2));
sigma(T,ss(GAM),'r*',{.1,30});
legend('Closed loop','GAM')
grid
```
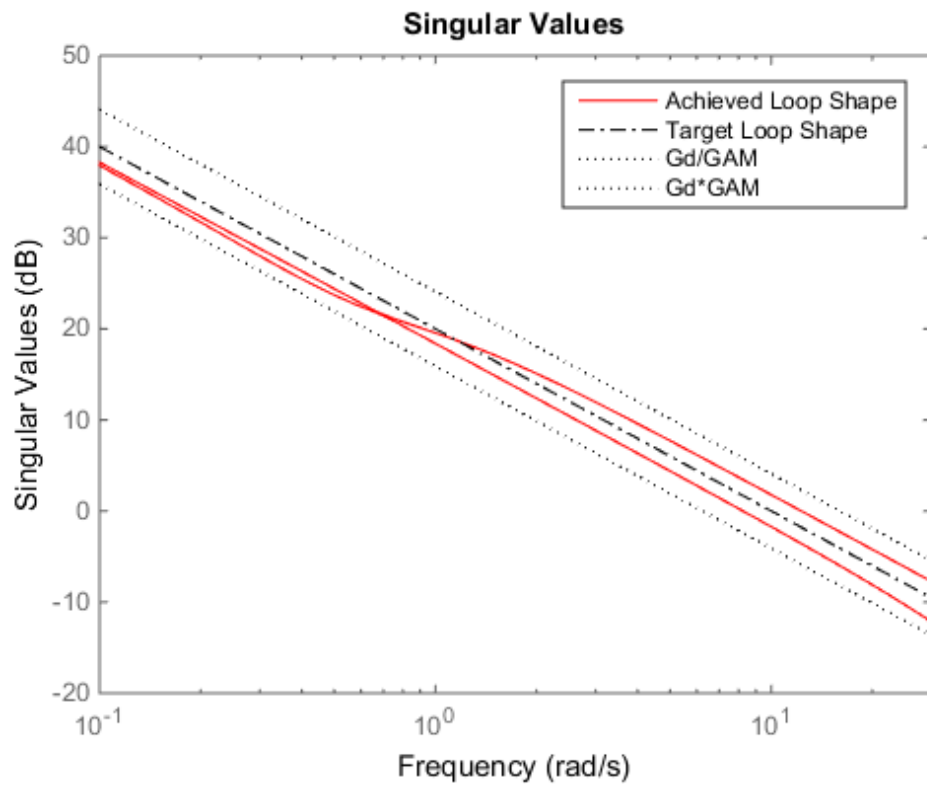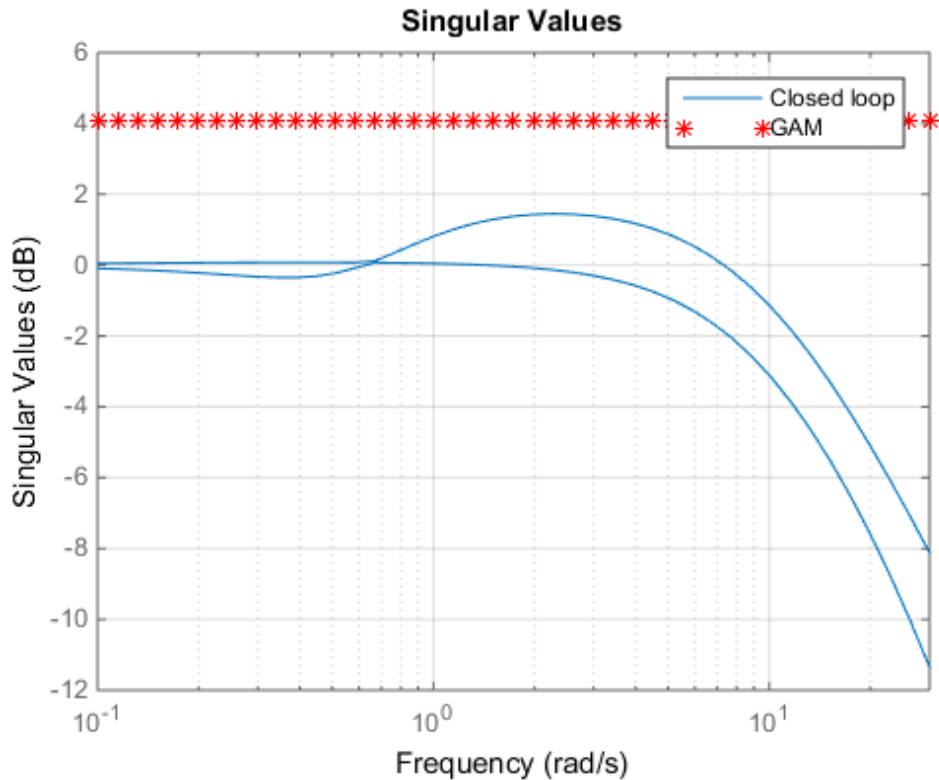
The value of γ= GAM returned is an indicator of the accuracy to which the optimal loop shape matches your desired loop shape and is an upper bound on the resonant peak magnitude of the closed-loop transfer function T = feedback(G*K,eye(2)). In this case, γ = 1.6024 = 4 dB, as the singular value plots show. The plots also show that the achieved loop shape matches the desired target Gd to within about γ dB.

Model Reduction and Approximation

Complex models are not always required for good control. Unfortunately, however, optimization methods (including methods based on H_∞, H_2, and μ-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	
reduce	Main interface to model approximation algorithms
balancmr	Balanced truncation model reduction
bstmr	Balanced stochastic truncation model reduction
hankelmr	Optimal Hankel norm model approximations
modreal	State-space modal truncation/realization
ncfmr	Balanced normalized coprime factor model reduction
schurmr	Schur balanced truncation model reduction
slowfast	State-space slow-fast decomposition
"stabsep"	State-space stable/antistable decomposition
imp2ss	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.

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 `hinfsyn` and `h2hinfsyn`. Some of the main functions that help you access the LMI capabilities of the toolbox are shown in the following table.

Specification of LMIs	
lmiedit	GUI for LMI specification
setlmis	Initialize the LMI description
lmivar	Define a new matrix variable
lmiterm	Specify the term content of an LMI
newlmi	Attach an identifying tag to new LMIs
getlmis	Get the internal description of the LMI system

LMI Solvers	
feasp	Test feasibility of a system of LMIs
gevp	Minimize generalized eigenvalue with LMI constraints
mincx	Minimize a linear objective with LMI constraints
dec2mat	Convert output of the solvers to values of matrix variables

Evaluation of LMIs/Validation of Results	
evallmi	Evaluate for given values of the decision variables
showlmi	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_∞ 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_∞ 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_∞ 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

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* Δ_M of the model G_0 is defined 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 Δ_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_∞, and H_2 norms, which are defined as follows:

H_2 and $H_\#$ Norms The H_2-norm is the energy of the impulse response of plant G. The H_∞-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 σ_i, are the nonnegative square roots of the eigenvalues of A^*A ordered such that $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p > 0$, $p \leq \min\{m, n\}$.

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

The greatest singular value σ_1 is sometimes denoted

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

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

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

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_∞-norms are defined terms of the frequency-dependent singular values of $G(j\omega)$:

H_2-norm:

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

H_∞-norm:

$$\|G\|_2 \triangleq \sup_{\omega}\bar{\sigma}\left(G(j\omega)\right)$$

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.*

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

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

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

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

$$L(s) = G(s)K(s).$$
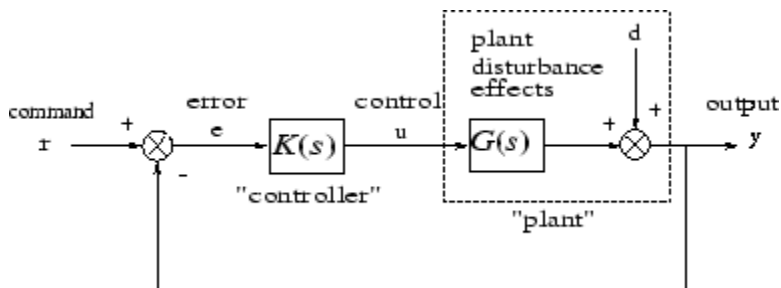
Block Diagram of the Multivariable Feedback Control System

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. Thus a disturbance attenuation performance specification can be written as

$$\bar{\sigma}\left(S(j\omega)\right) \le \left|W_1^{-1}(j\omega)\right|$$

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

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 Δ_A and multiplicative plant perturbations Δ_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 Δ_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 $\Delta_A = 0$.
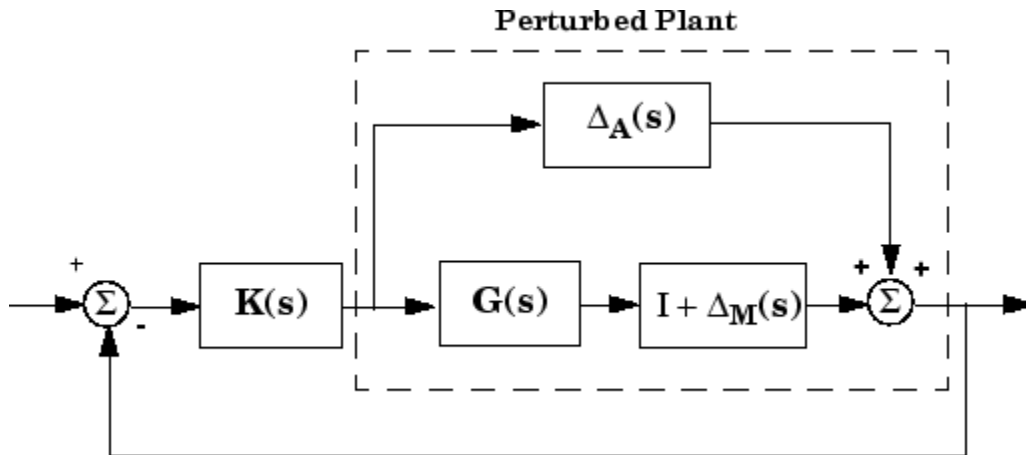
Additive/Multiplicative Uncertainty

Taking $\bar{\sigma}\left(\Delta_M\left(j\omega\right)\right)$ 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:

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

The smaller is $\bar{\sigma}\left(T\left(j\omega\right)\right)$, 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 $\bar{\sigma}\left(\Delta_A\left(j\omega\right)\right)$ to be the definition of the "size" of $\Delta_A(j\omega)$ at frequency ω.

Additive Robustness: The size of the smallest destabilizing additive uncertainty Δ_A is:

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

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

$$\bar{\sigma}\left(R\{j\omega\}\right) \le \left|W_2^{-1}\left(j\omega\right)\right|$$

$$\bar{\sigma}\left(T\{j\omega\}\right) \le \left|W_3^{-1}\left(j\omega\right)\right|$$

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 Δ_M, so that the control design requirements can be written

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

as shown in Singular Value Specifications on L, S, and T.

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 (below the 0 dB line),

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

This results from the fact that

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

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

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

if $\bar{\sigma}(L(s)) \ll 1$.
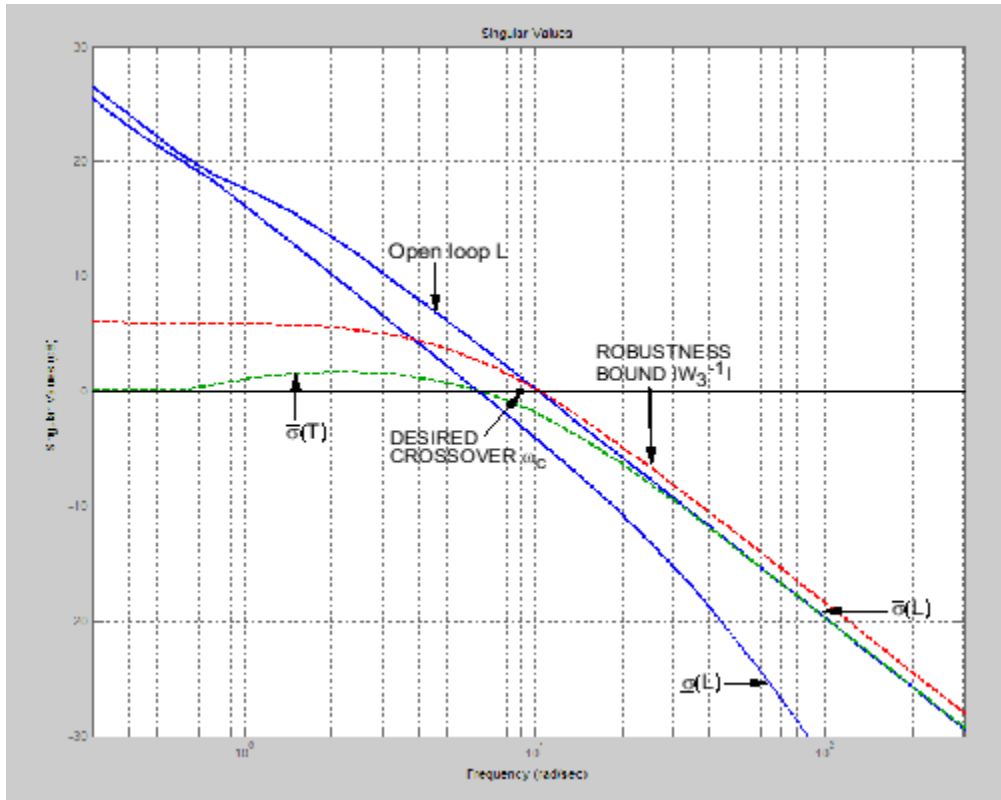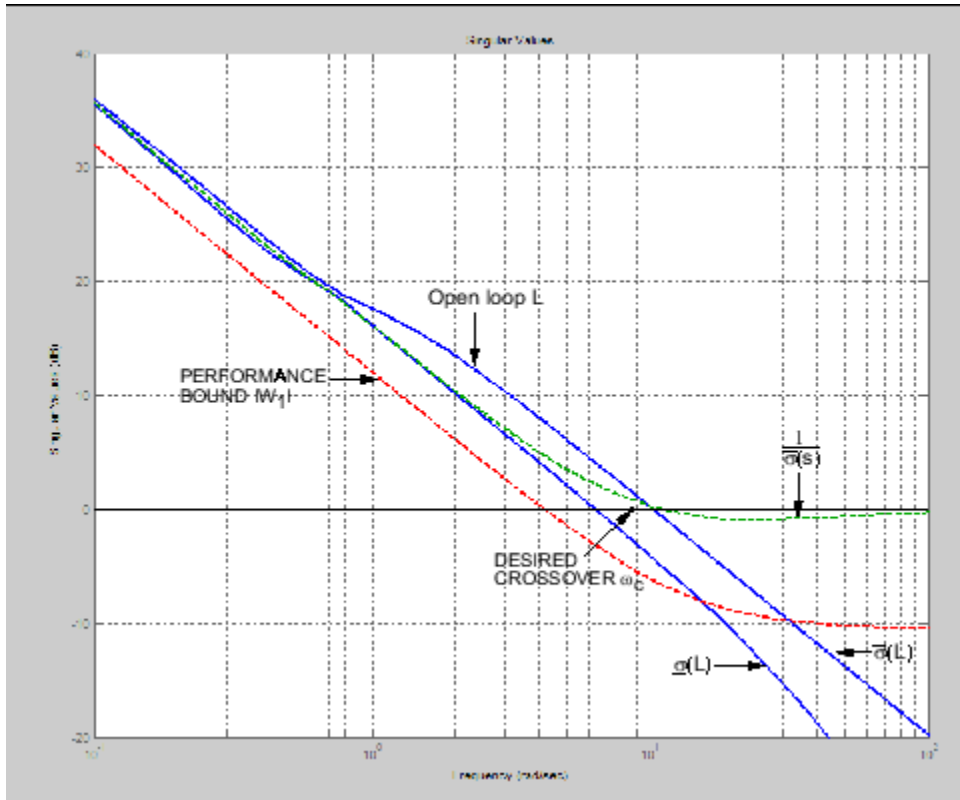
Singular Value Specifications on L, S, and T

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,

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

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 θ_M in each feedback loop of the multivariable feedback system of Block Diagram of the Multivariable Feedback Control System 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:

$$g_M \le 1 - \frac{1}{\|T\|_\infty}$$

$$g_M \le \frac{1}{1 + \dfrac{1}{\|S\|_\infty}}.$$

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 Gd — 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, `Gd` 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 `Gd` 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 (α) and attitude angle (θ) and their rates of change, and the last two representing elevon and canard control actuator dynamics — see Aircraft Configuration and Vertical Plane Geometry.

```
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 (δ_e and δ_c). The output variables are angle of attack (α) and attitude angle (θ).
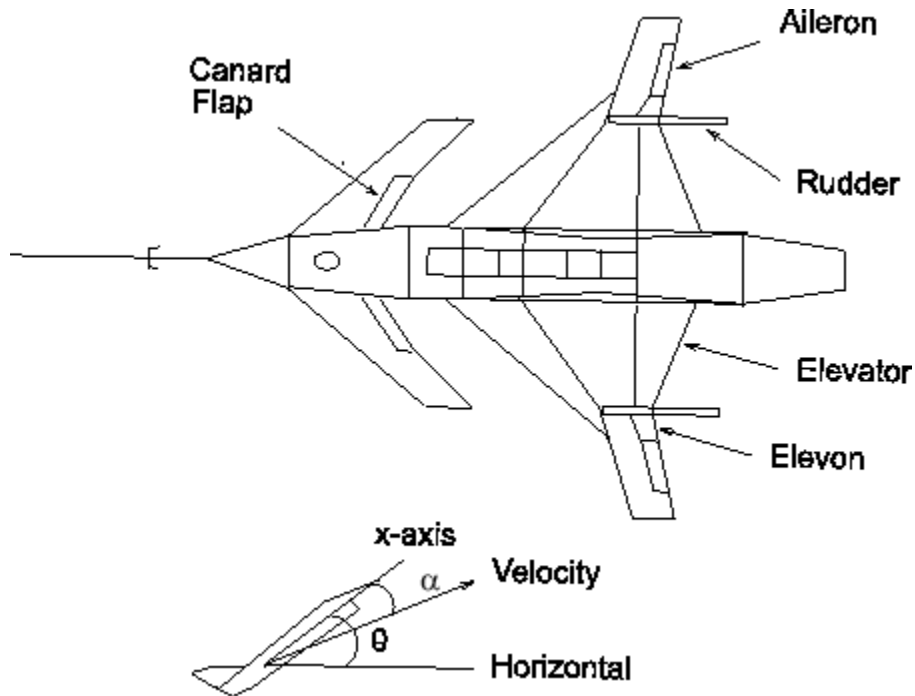
Aircraft Configuration and Vertical Plane Geometry

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

```
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(L,'r--',Gd,'k-.',Gd/GAM,'k:',Gd*GAM,'k:',{.1,100})
legend('\sigma(L) loopshape',...
 '\sigma(Gd) desired loop',...
 '\sigma(Gd) \pm GAM, dB');
figure;
sigma(T,I+L,'r--',{.1,100})
legend('\sigma(T) robustness','1/\sigma(S) performance')
```

α and θ command step responses
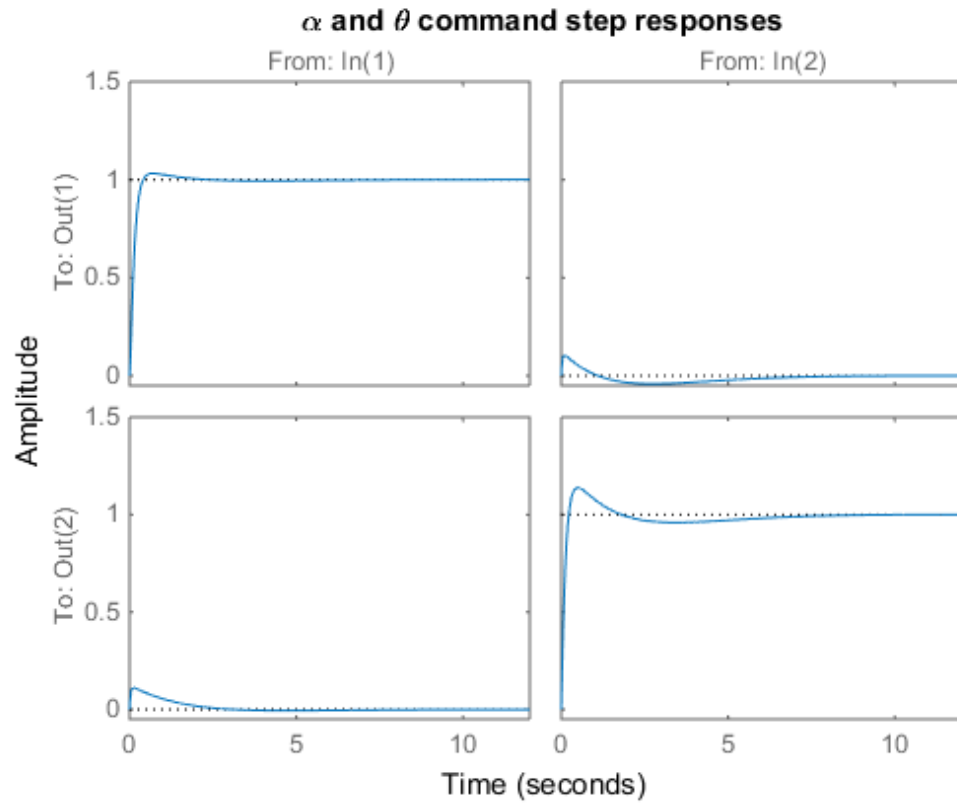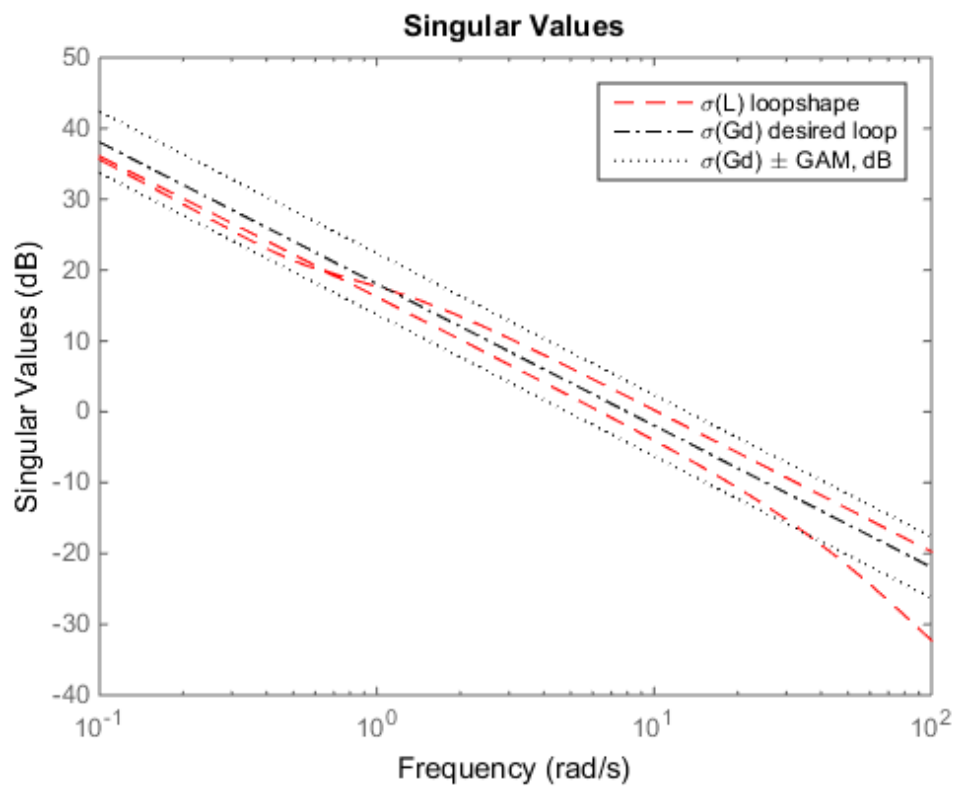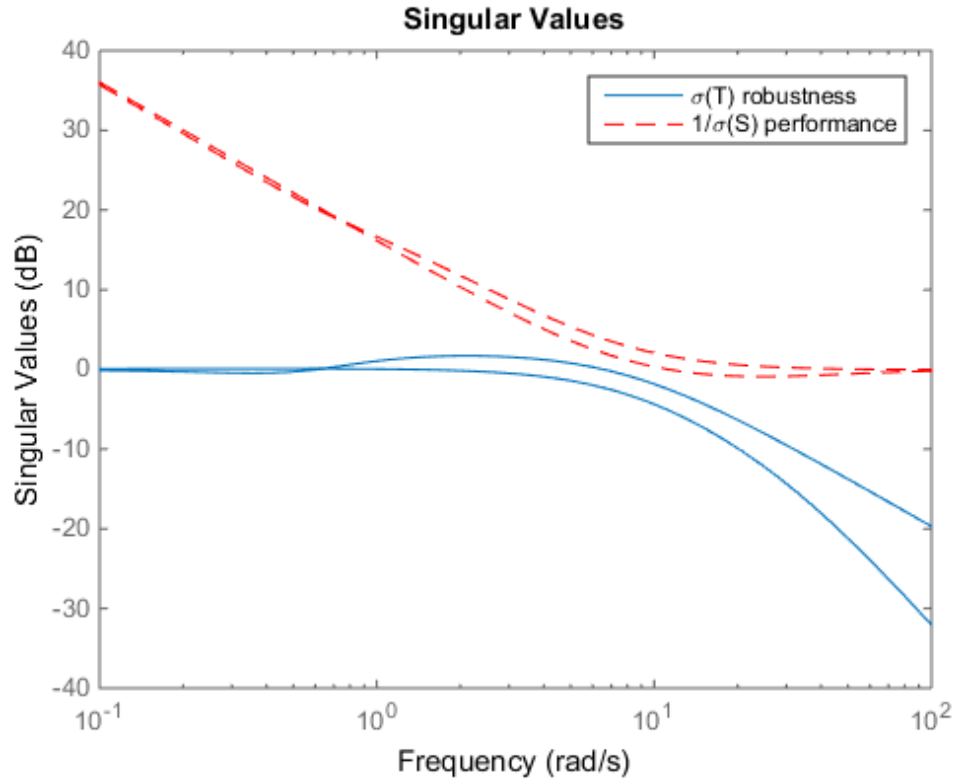
The number ±GAM, dB (i.e., 20log10(GAM)) tells you the accuracy with which your loopsyn control design matches the target desired loop:

$$\bar{\sigma}(GK), \mathrm{db} \geq |G_\mathrm{d}|, \mathrm{db} - \mathrm{GAM}, \mathrm{db} \quad (\omega < \omega_c)$$
$$\bar{\sigma}(GK), \mathrm{db} \geq |G_\mathrm{d}|, \mathrm{db} + \mathrm{GAM}, \mathrm{db} \quad (\omega > \omega_c).$$

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 ±180° 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 ω_c) between the above two frequency ranges, and below the crossover frequency ω_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° 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 ffundamental limits on your 0 dB crossover frequency ω_c [12]. For instance, your 0 dB crossover ω_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.

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

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_∞ *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

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

where (see MIXSYN H_∞ Mixed-Sensitivity Loop Shaping $T_{y1\ u1}$):

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

The term $\left\| T_{y_1 u_1} \right\|_\infty$ 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, so that there is a gap for the desired loop shape `Gd` 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.
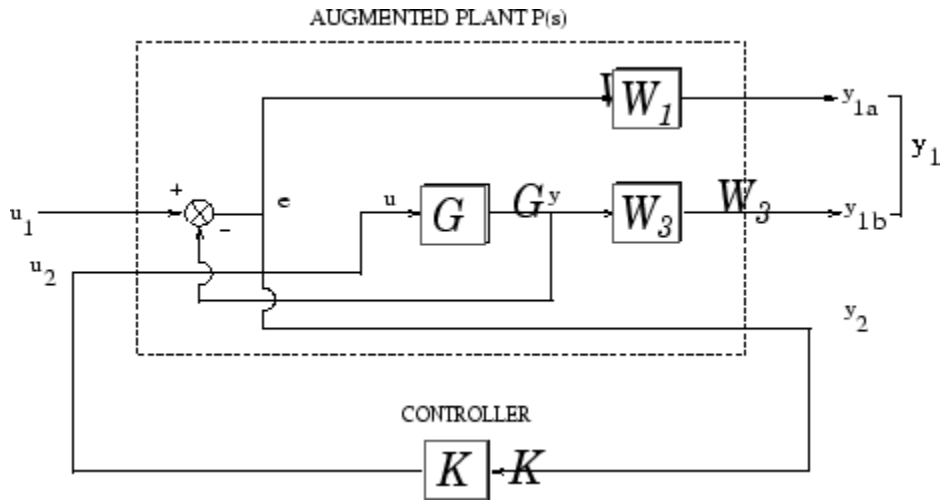
MIXSYN H$_\#$ Mixed-Sensitivity Loop Shaping T$_{y1\ u1}$

Mixed-Sensitivity Loop-Shaping Controller Design

To do a `mixsyn` H_∞ mixed-sensitivity synthesis design on the HiMAT model, start with the plant model `G` discussed in "Mixed-Sensitivity Loop-Shaping Controller Design" on page 2-23. The following code recreates that plant model.

```
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);
```

Set up the performance and robustness bounds, `W1` and `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` using `mixsyn`.

```
[K1,CL1,GAM1] = mixsyn(G,W1,[],W3);
```

Next compute responses of the closed-loop system. Compute the loop `L1`, sensitivity `S1`, and complementary sensitivity `T1`.

```
L1 = G*K1;
I = eye(size(L1));
S1 = feedback(I,L1); % S=inv(I+L1);
T1 = I-S1;
```

Finally, plot time-domain and frequency-domain responses.

```
step(T1,1.5);
title('\alpha and \theta command step responses');

figure;
sigma(I+L1,'--',T1,':',L1,'r--',W1/GAM1,'k--',GAM1/W3,'k-.',{.1,100})
legend('1/\sigma(S) performance','\sigma(T) robustness','\sigma(L) loopshape',...
       '\sigma(W1) performance bound','\sigma(1/W3) robustness bound')
```

3
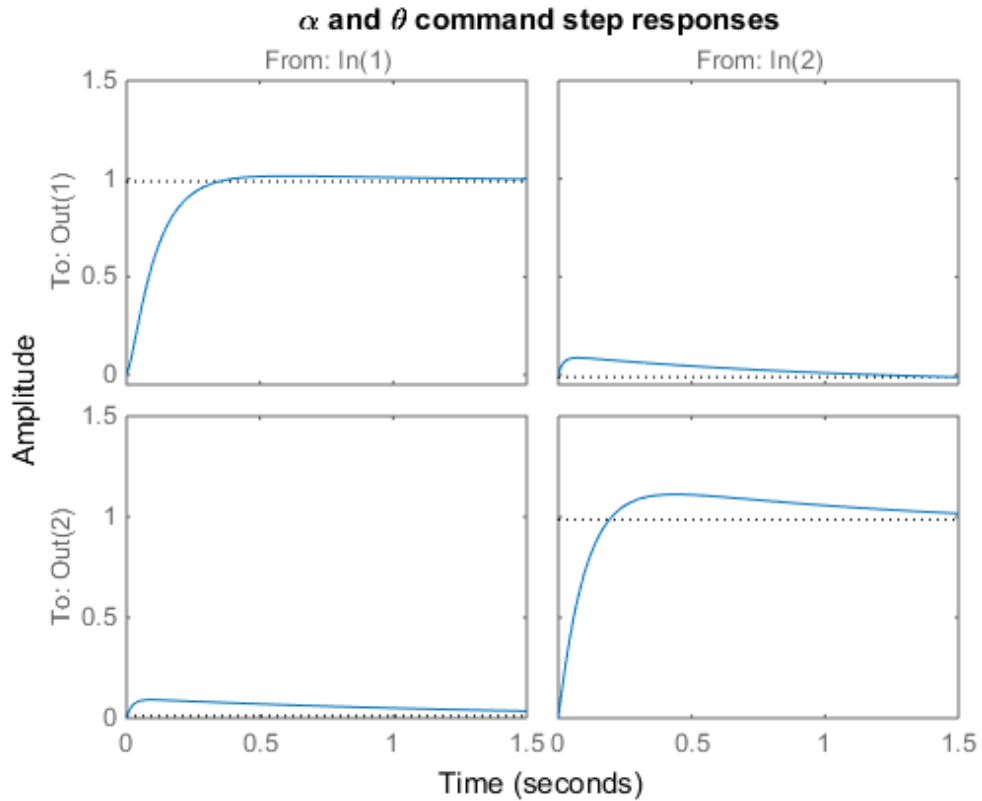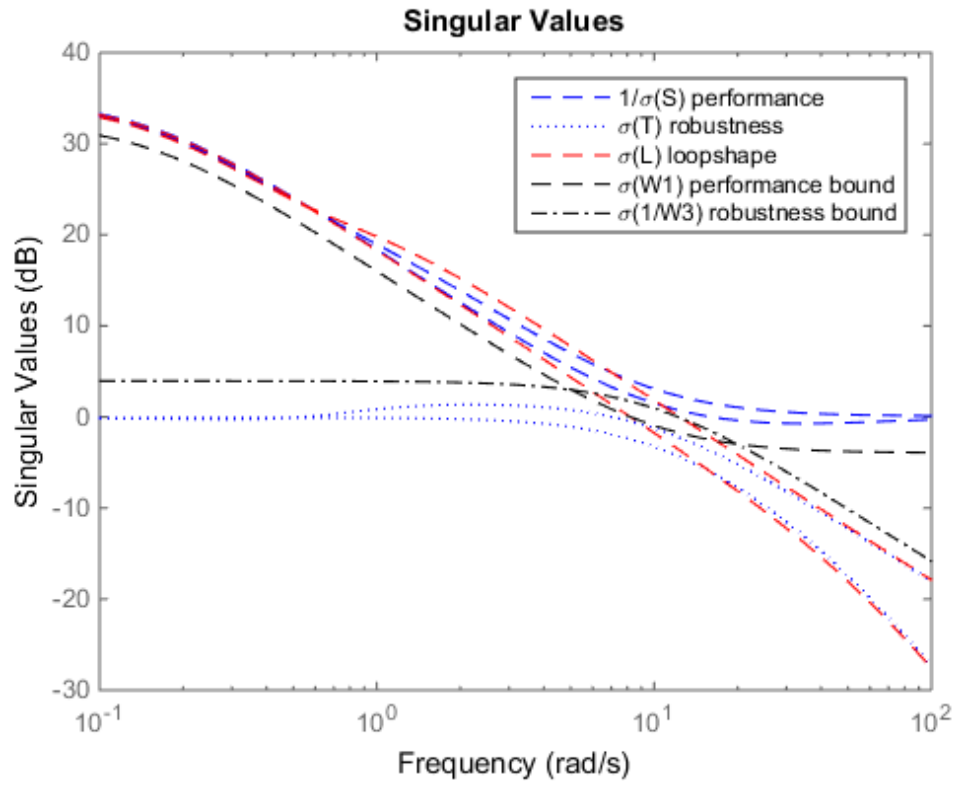
Model Reduction for Robust Control

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_∞ 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_∞ 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]

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

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

$$AP + PA^T = -BB^T$$
$$A^TQ + QA = -C^TC.$$

For example, generate a random 30-state system and plot its Hankel singular values.

```
rng(1234,'twister');
G = rss(30,4,3);
hankelsv(G)
```
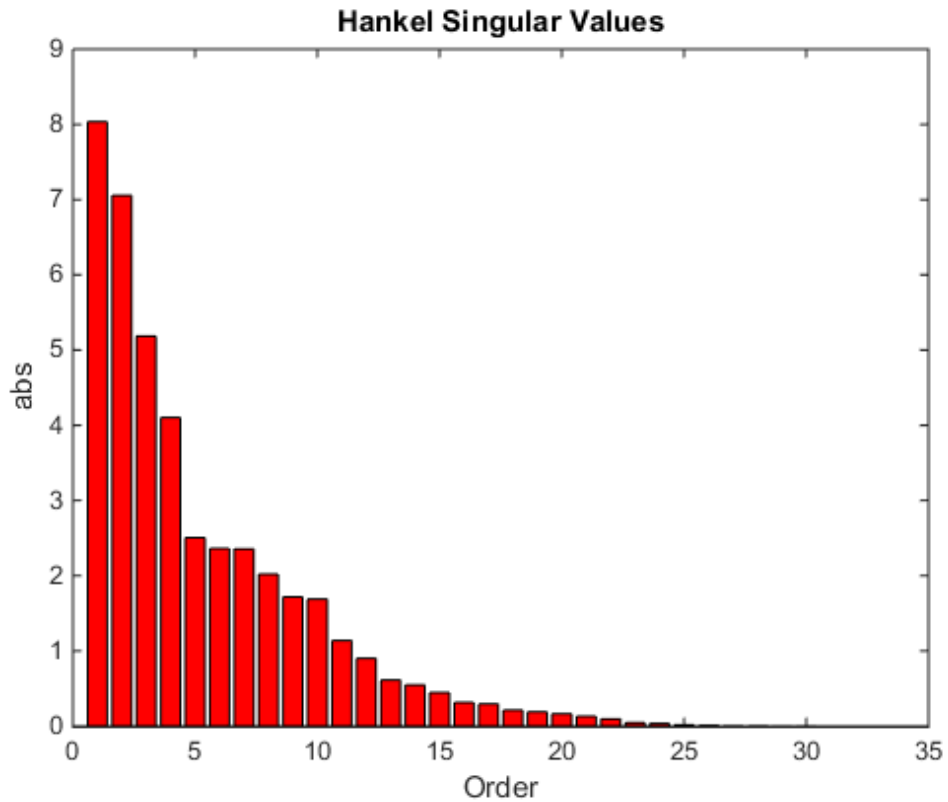
The plot shows shows that system G has most of its "energy" stored in states 1 through 15 or so. Later, you will see how to use model reduction routines to keep a 15-state reduced model that preserves most of its dynamic response.

Related Examples

- "Approximate Plant Model by Additive Error Methods" on page 3-7
- "Approximate Plant Model by Multiplicative Error Method" on page 3-10

More About

- "Model Reduction Techniques" on page 3-5

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 `Gred` of the original model `G` with a bound on the error $\left\| G - Gred \right\|_\infty$, the peak gain across frequency. Others produce a reduced-order model with a bound on the relative error $\left\| G^{-1}\left(G - Gred \right) \right\|_\infty$.

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

Additive Error Bound

$$\left\| G - Gred \right\|_\infty \le 2 \sum_{k+1}^{n} \sigma_i$$

where σ_i are denoted the ith Hankel singular value of the original system `G`.

Multiplicative (Relative) Error Bound

$$\left\| G^{-1}\left(G - Gred \right) \right\|_\infty \le \prod_{k+1}^{n}\left(1 + 2\sigma_i\left(\sqrt{1 + \sigma_i^2} + \sigma_i \right) \right) - 1$$

where σ_i are denoted the ith 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 `G` 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.

```
rng(1234,'twister');
G = rss(30,4,3); % random 30-state model
% balanced truncation to models with sizes 12:16
[G1,info1] = balancmr(G,12:16);
% 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-.')
legend('G','G1','G2')
```
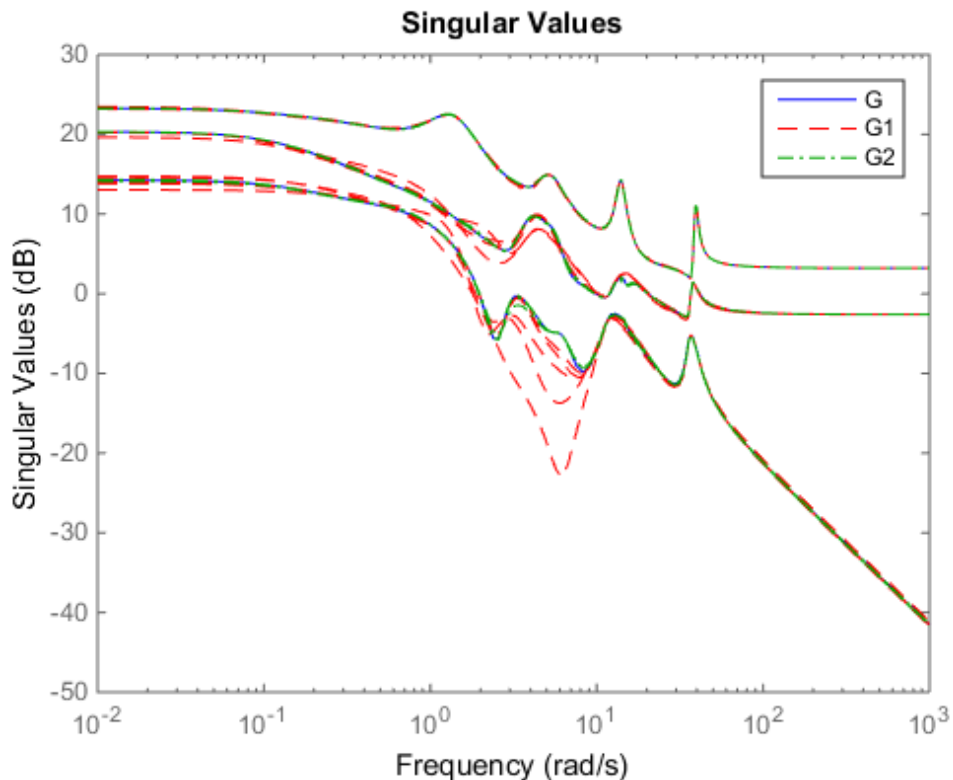
The plot compares the original model `G` with the reduced models `G1` and `G2`.

To determine whether the theoretical error bound is satisfied, calculate the peak difference across frequencies between the gain of the original system and the reduced system. Compare that to the error bound stored in the `info` structure.

```
norm(G-G1(:,:,1),'inf')
info1.ErrorBound(1)
```

```
ans =

    1.2556


ans =

    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')
```

Error Bound and Model Error
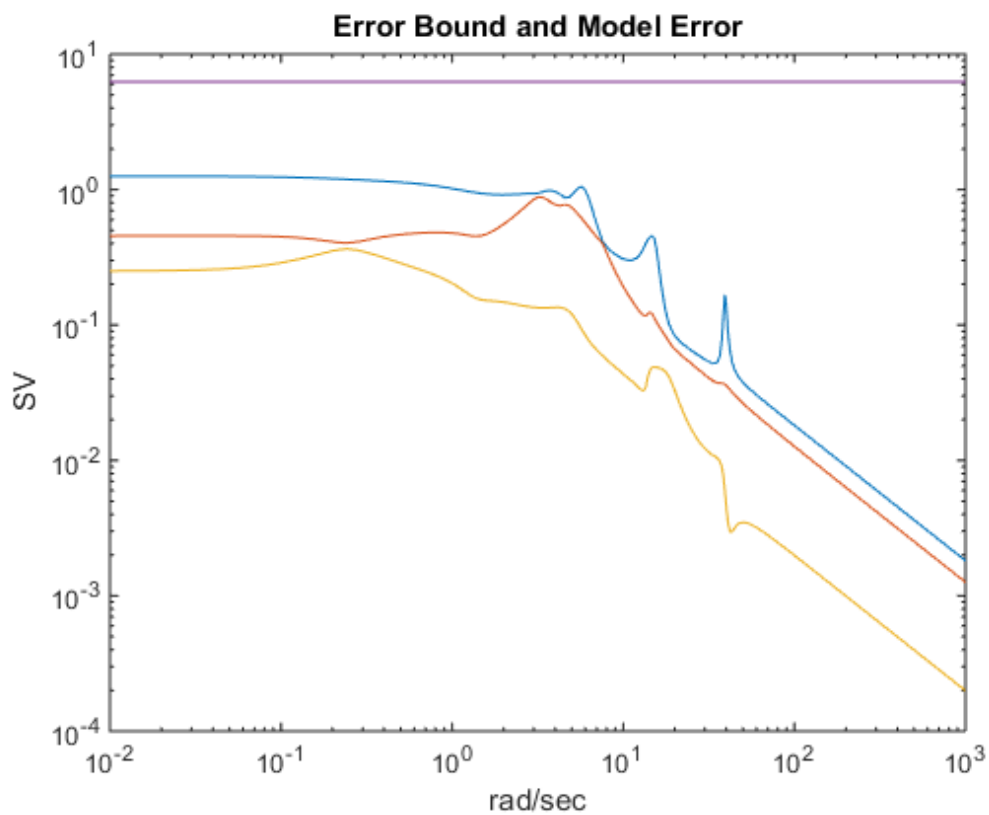
Approximate Plant Model by Multiplicative Error Method

In most cases, the 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(123456);
G = rss(30,1,1);    % random 30-state model

[gr,infor] = reduce(G,'Algorithm','balance','order',7);
[gs,infos] = reduce(G,'Algorithm','bst','order',7);

figure(1)
bode(G,'b-',gr,'r--')
title('Additive Error Method')
legend('Original','Reduced')

figure(2)
bode(G,'b-',gs,'r--')
title('Relative Error Method')
legend('Original','Reduced')
```
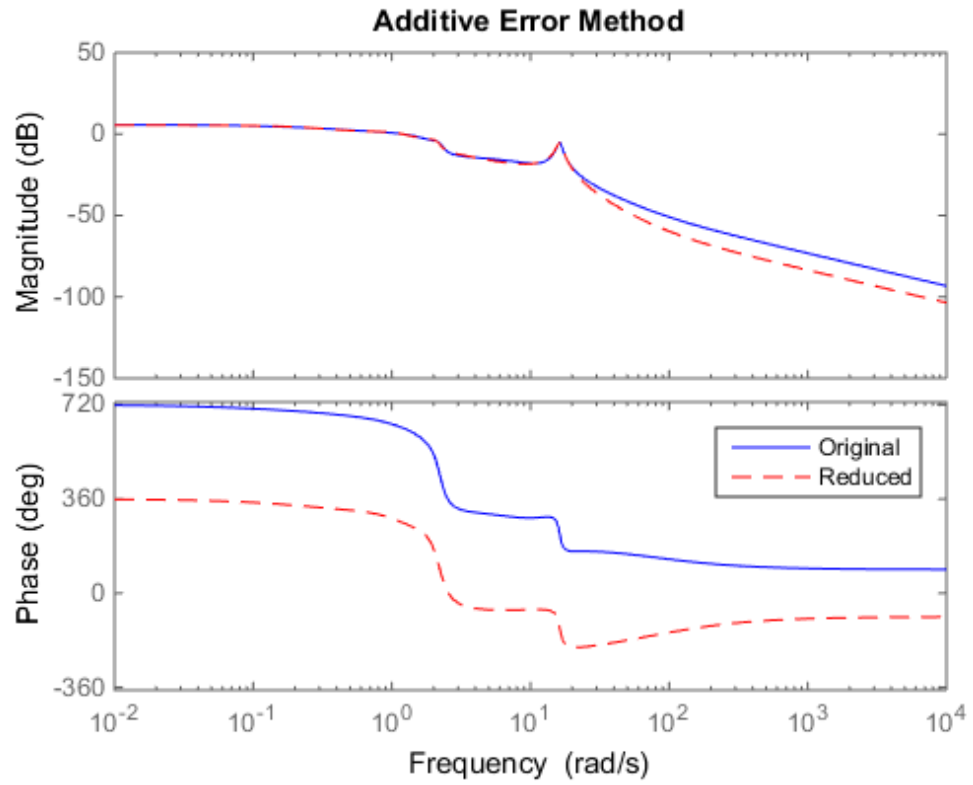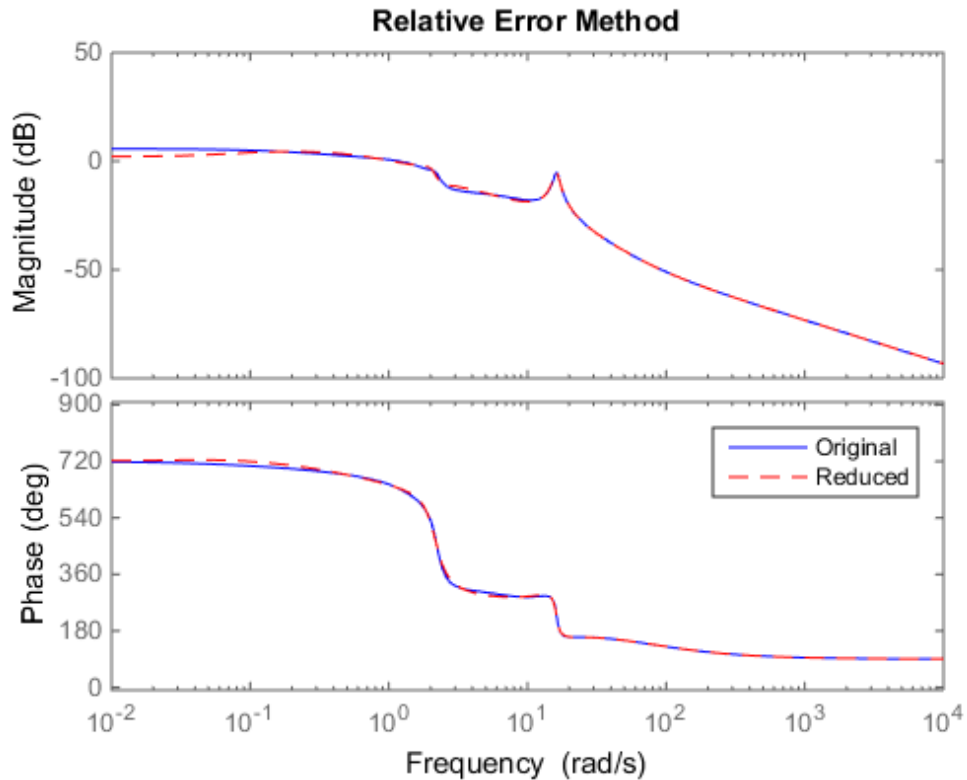
Therefore, for some systems with low damped poles or zeros, the balanced stochastic method (`bstmr`) produces a better reduced-order model fit in those frequency ranges to make multiplicative error small. Whereas additive error methods such as `balancmr`, `schurmr`, or `hankelmr` only care about minimizing the overall "absolute" peak error, they can produce a reduced-order model missing those low damped poles/zeros frequency regions.

See Also

`balancmr` | `bstmr` | `hankelmr` | `schurmr`

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 remaining dynamics.

```
rng(5678,'twister');
G = rss(30,1,1);         % random 30-state model
[Gjw,G2] = modreal(G,1); % only one rigid body dynamics
G2.d = Gjw.d; Gjw.d = O; % put DC gain of G into G2
subplot(211);sigma(Gjw);ylabel('Rigid Body')
subplot(212);sigma(G2);ylabel('Nonrigid Body')
```
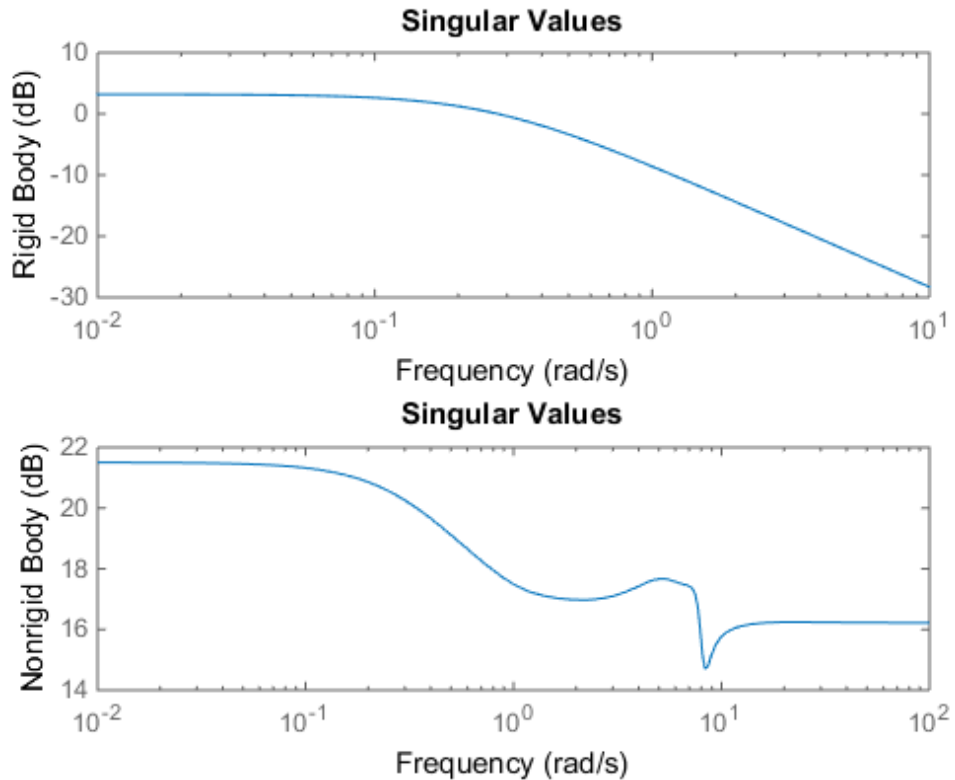
Further model reduction can be done on G2 without any numerical difficulty. After G2 is further reduced to Gred, the final approximation of the model is simply Gjw+Gred.

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

Examine the Hankel singular value plot.

```
hankelsv(G)
```
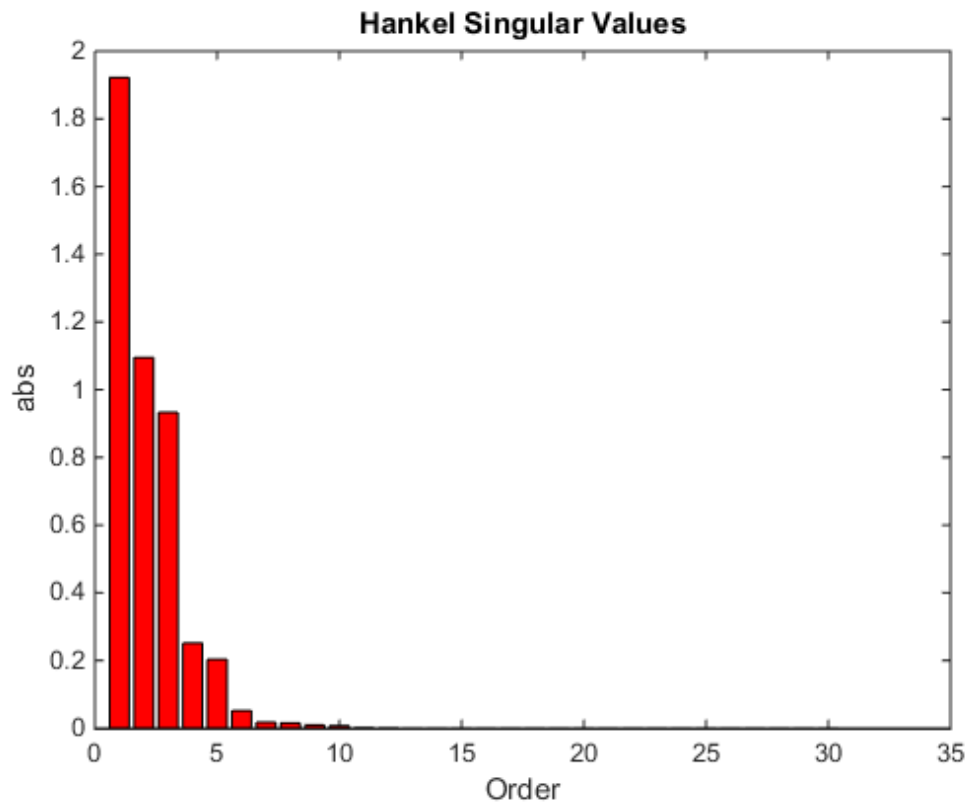
Calculate an eighth-order reduced model.

```
[gr,info] = reduce(G,8);
figure
bode(G,'b-',gr,'r--')
legend('Original','Reduced');
```
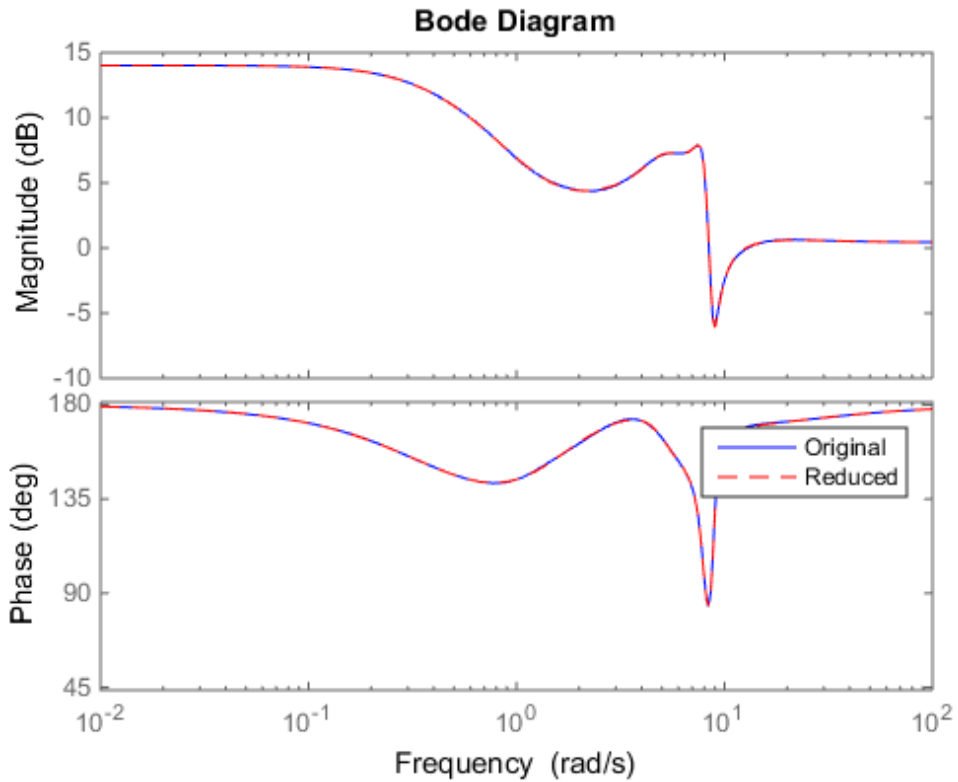
The default algorithm `balancmr` of `reduce` has done a great job of approximating a 30-state model with just eight states. Again, the rigid body dynamics are preserved for further controller design.

See Also

balancmr | bstmr | hankelmr | hankelsv | modreal | schurmr

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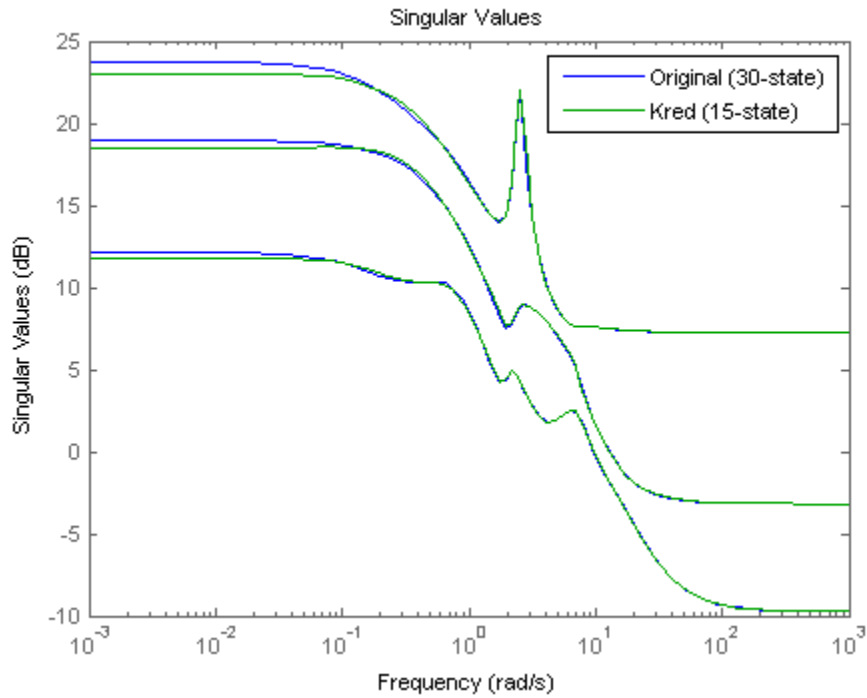
If integral control is important, previously mentioned methods (except `ncfmr`) can nicely preserve the original integrator(s) in the model.

See Also
modreal | ncfmr | ncfmr

Bibliography

[1] Glover, K., "All Optimal Hankel Norm Approximation of Linear Multivariable Systems, and Their L$^{\#}$ - 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.

4

Robustness Analysis

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)
```

```
bw =

  Uncertain real parameter "bw" with nominal value 5 and variability [-10,10]%.
```

This command creates a `ureal` object that stores a number of parameters in its properties. View the properties of `bw`.

```
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 (`uss`) object. 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])


H =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 1 states.
  The model uncertainty consists of the following blocks:
    bw: Uncertain real, nominal = 5, variability = [-10,10]%, 1 occurrences

Type "H.NominalValue" to see the nominal value, "get(H)" to see all properties, and "H.
```

Note that the result H is an uncertain system, called a uss model. The nominal value of H is a state-space (ss) model. Verify that the pole is at -5, as expected from the uncertain parameter's nominal value of 5.

```
pole(H.NominalValue)


ans =

    -5
```

Next, use bodeplot and stepplot to examine the behavior of H. These commands plot the responses of the nominal system and a number of random samples of the uncertain system.

```
bodeplot(H,{1e-1 1e2});
```
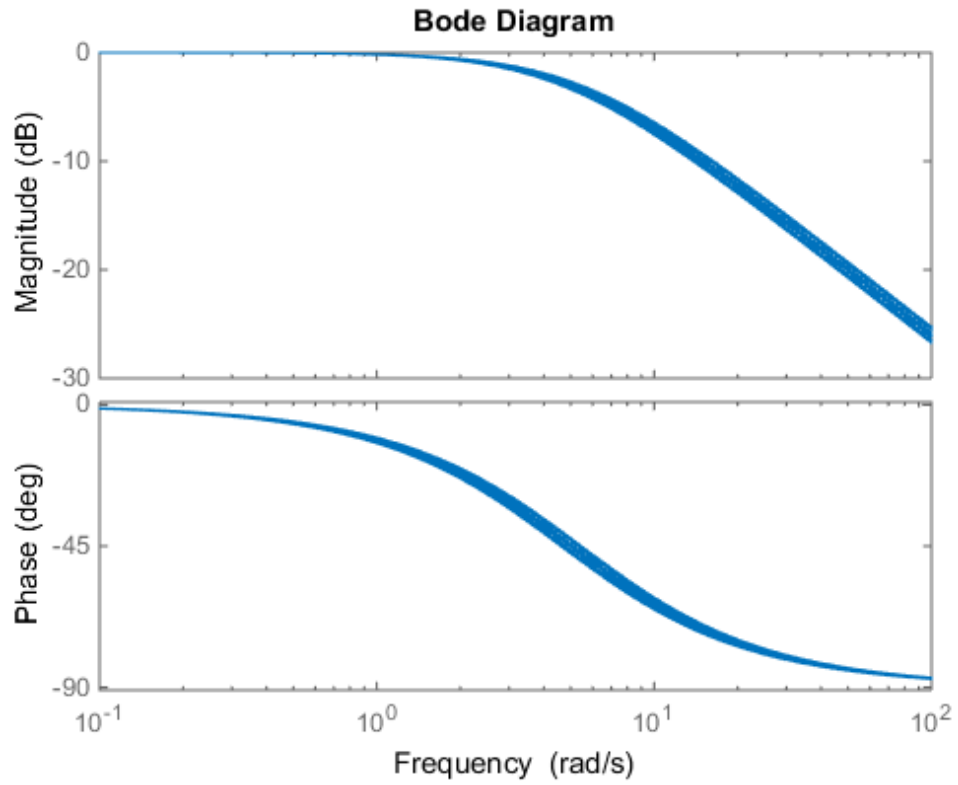
```
stepplot(H)
```
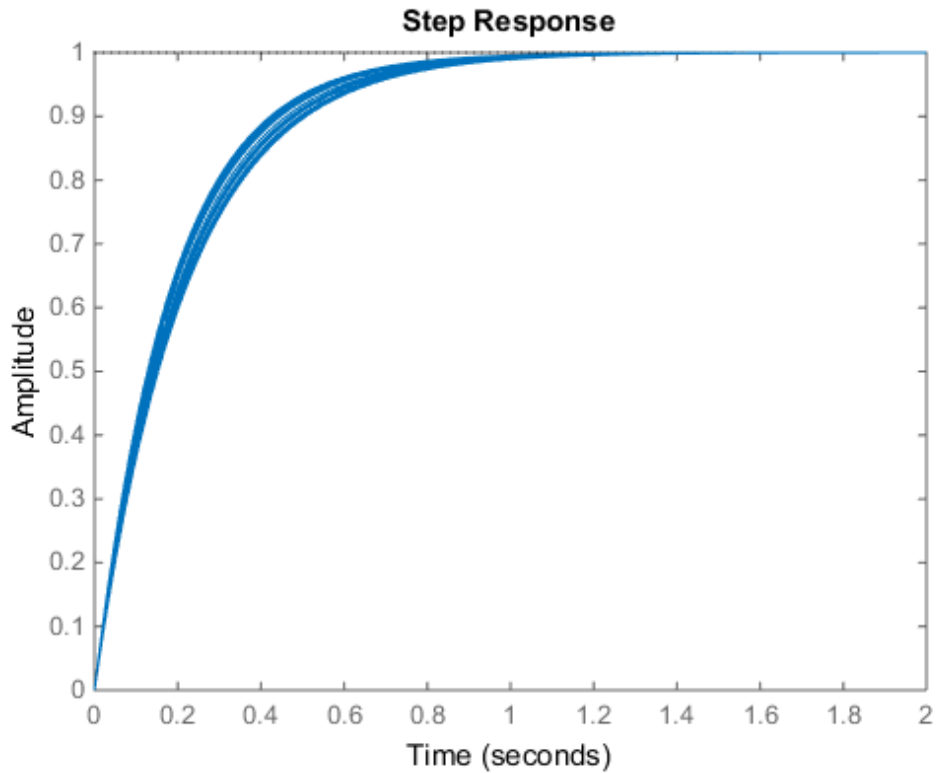
While there are variations in the bandwidth and time constant of H, the high-frequency rolls off at 20 dB/decade regardless of the value of bw. You can capture the more complicated uncertain behavior that typically occurs at high frequencies using the ultidyn uncertain element.

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:

```
bw = ureal('bw',5,'Percentage',10);
H = tf(1,[1/bw 1]);

Gnom = H;
W = makeweight(.05,9,10);
Delta = ultidyn('Delta',[1 1]);
G = Gnom*(1+W*Delta)


G =
```

```
  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    Delta: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences
    bw: Uncertain real, nominal = 5, variability = [-10,10]%, 1 occurrences

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

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})
```
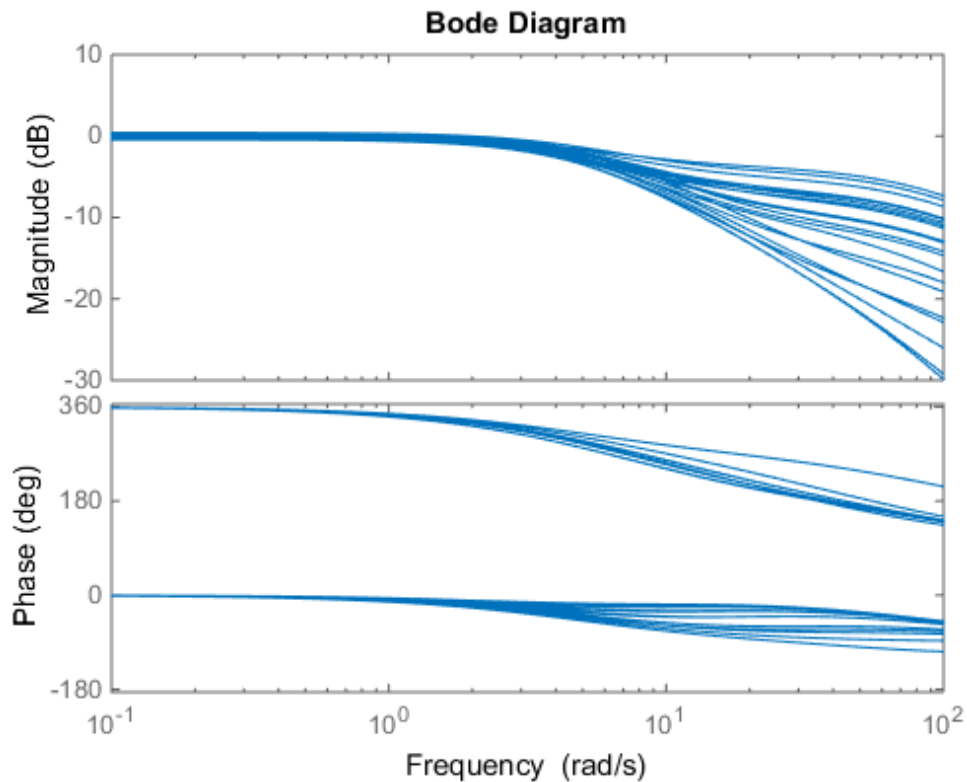
In the next section, you design and compare two feedback controllers for `G`.

Related Examples

*   "Robust Controller Design" on page 4-10

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 ξ and natural frequency ω_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 ξ=0.707, with different ω_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)
```
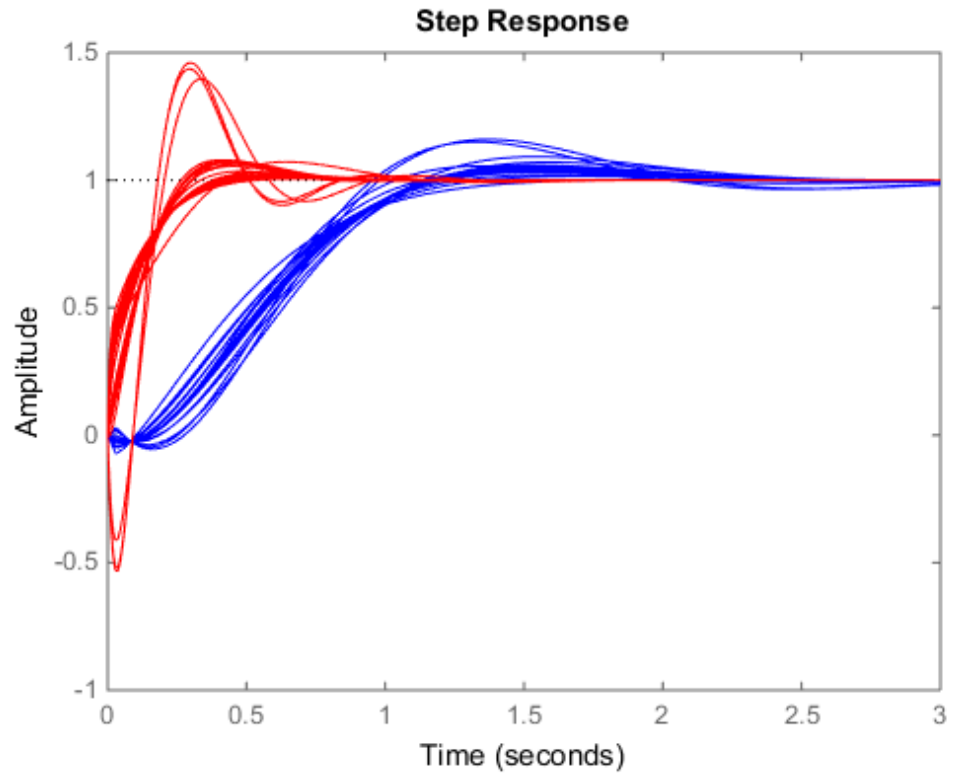
Step Response

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
[stabmarg2,destabu2,report2] = robuststab(T2);
stabmarg2


stabmarg1 =

            LowerBound: 4.0137
```

```
                  UpperBound: 4.0137
     DestabilizingFrequency: 3.6934


stabmarg2 =

                  LowerBound: 1.2530
                  UpperBound: 1.2530
     DestabilizingFrequency: 10.8831
```

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
report2


report1 =

Uncertain system is robustly stable to modeled uncertainty.
 -- It can tolerate up to 401% of the modeled uncertainty.
 -- A destabilizing combination of 401% of the modeled uncertainty was found.
 -- This combination causes an instability at 3.69 rad/seconds.
 -- Sensitivity with respect to the uncertain elements are:
     'Delta' is 100%.  Increasing 'Delta' by 25% leads to a 25% decrease in the margin.
     'bw' is 21%.  Increasing 'bw' by 25% leads to a 5% decrease in the margin.


report2 =

Uncertain system is robustly stable to modeled uncertainty.
 -- It can tolerate up to 125% of the modeled uncertainty.
 -- A destabilizing combination of 125% of the modeled uncertainty was found.
 -- This combination causes an instability at 10.9 rad/seconds.
 -- Sensitivity with respect to the uncertain elements are:
     'Delta' is 100%.  Increasing 'Delta' by 25% leads to a 25% decrease in the margin.
     'bw' is 11%.  Increasing 'bw' by 25% leads to a 3% decrease in the margin.
```

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

[maxgain2,wcu2] = wcgain(S2);
maxgain2


maxgain1 =

            LowerBound: 1.8831
            UpperBound: 1.8835
     CriticalFrequency: 3.2651


maxgain2 =

            LowerBound: 4.6037
            UpperBound: 4.6120
     CriticalFrequency: 11.1286
```

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
```
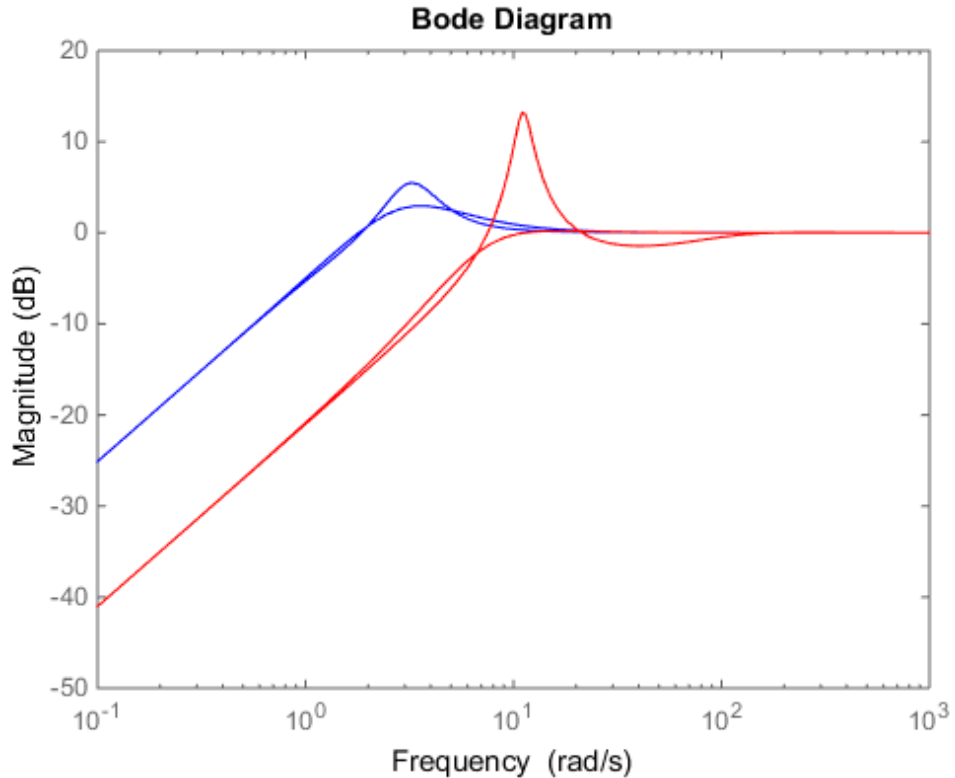
Clearly, while K2 achieves better nominal sensitivity than K1, the nominal closed-loop bandwidth extends too far into the frequency range where the process uncertainty is very large. Hence the worst-case performance of K2 is inferior to K1 for this particular uncertain model.

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 later.

```
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])


H =

  Uncertain continuous-time state-space model with 2 outputs, 2 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    p: Uncertain real, nominal = 10, variability = [-10,10]%, 2 occurrences

Type "H.NominalValue" to see the nominal value, "get(H)" to see all properties, and "H.
```

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: []
     SamplingGrid: [1x1 struct]
```

Most properties behave in the same way as the corresponding properties of `ss` objects.
The property `NominalValue` is itself 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 states.
  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, and "G.
```

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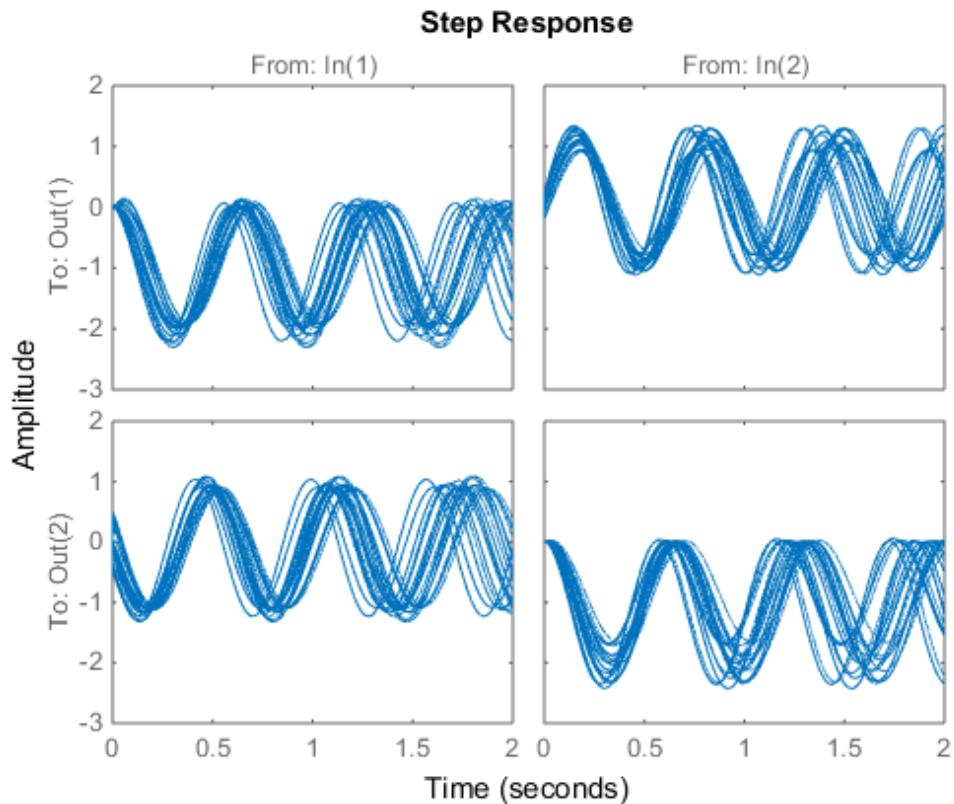
You can create a Bode plot of samples of `G`. The high-frequency uncertainty in the model is also obvious. For clarity, start the Bode plot beyond the resonance.

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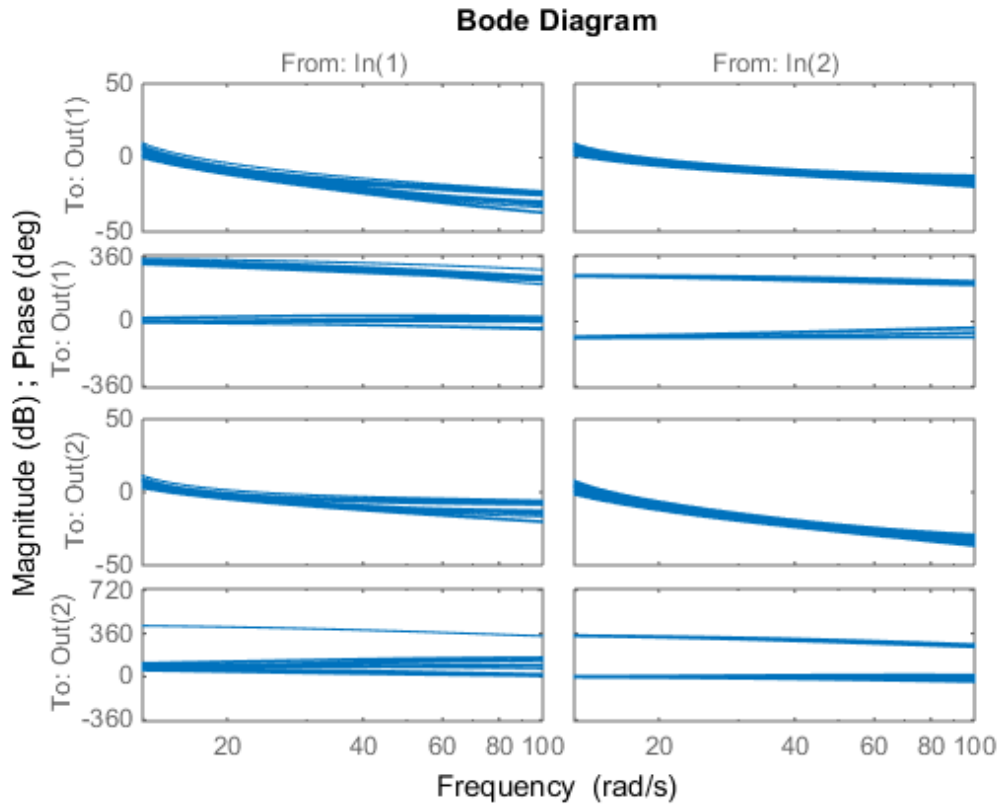

Closed-Loop Robustness Analysis

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

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

```
Warning: Updating objects saved with previous MATLAB version...
Resave your MAT files to improve loading speed.
```

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` or complementary sensitivity, and `L` for open-loop gain. The suffixes `i` and `o` refer to the input and output of the plant. Finally, `P` and `C` refer to the plant and controller.

Hence, `Ti` is mathematically the same as:

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

`Lo` is `G*K`, and `CSo` is mathematically the same as

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

Examine the transmission of disturbances at the plant input to the plant output by plotting responses of `F.PSi`. Graph some samples along with the nominal.

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

Bode Diagram
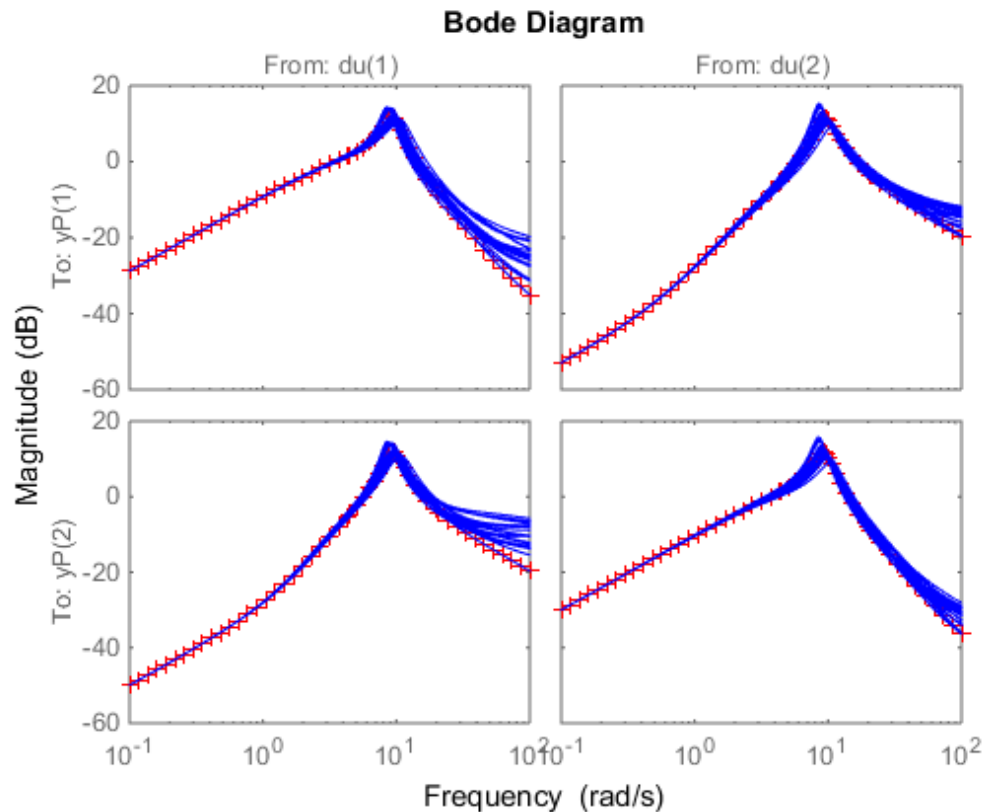
Nominal Stability Margins

You can use `loopmargin` o 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: [0.1179 8.4796]
    PhaseMargin: [-76.5484 76.5484]
      Frequency: 6.3496
```

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: [0.1190 8.4013]
    PhaseMargin: [-76.4242 76.4242]
      Frequency: 19.5393
```

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: [0.5660 1.7667]
    PhaseMargin: [-30.9788 30.9788]
      Frequency: 9.2914
```

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 =

                LowerBound: 2.2174
                UpperBound: 2.2175
    DestabilizingFrequency: 13.5963
```

```
report
```

```
report =

Uncertain system is robustly stable to modeled uncertainty.
 -- It can tolerate up to 222% of the modeled uncertainty.
 -- A destabilizing combination of 222% of the modeled uncertainty was found.
 -- This combination causes an instability at 13.6 rad/seconds.
 -- Sensitivity with respect to the uncertain elements are:
     'Delta1' is 55%.  Increasing 'Delta1' by 25% leads to a 14% decrease in the margin
     'Delta2' is 54%.  Increasing 'Delta2' by 25% leads to a 14% decrease in the margin
```

```
'p' is 39%.  Increasing 'p' by 25% leads to a 10% decrease in the margin.
```

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.

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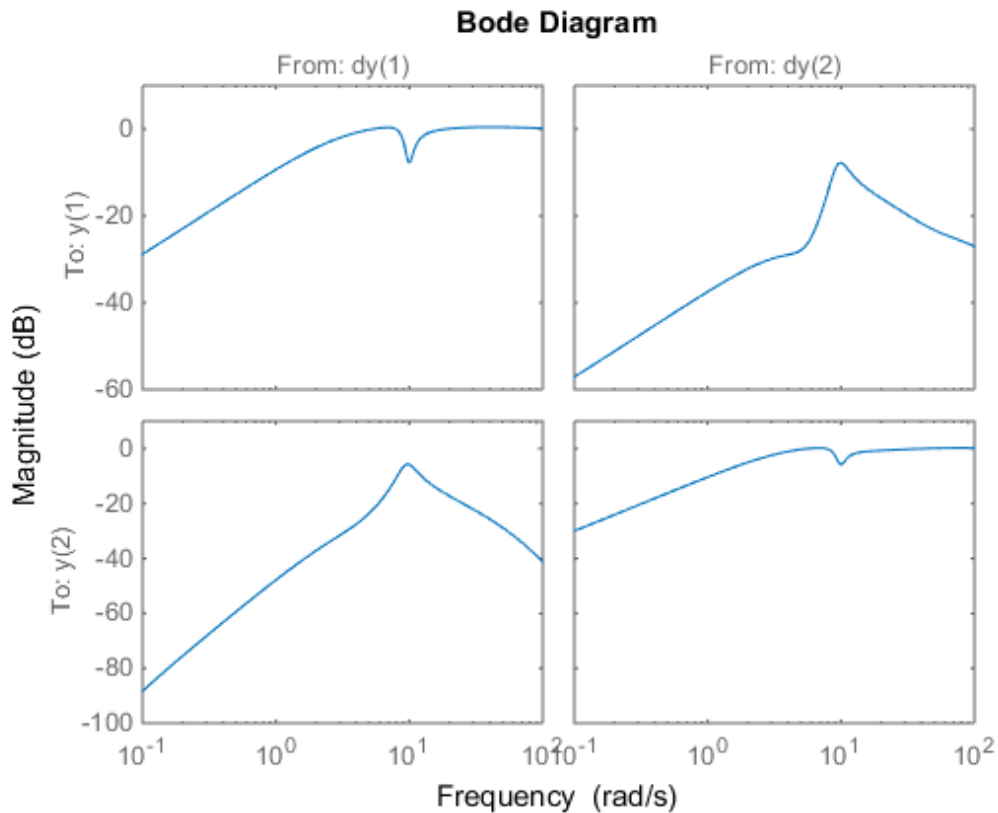
You can compute the peak value of the maximum singular value of the frequency response matrix using `norm`.

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

```
PeakNom =

    1.1288
```

```
freq =

    6.7969
```

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 `Delta1`, `Delta2`, and `p` vary over their ranges? You can use `wcgain` to answer this.

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

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 `Delta1`, `Delta2`, 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)
```
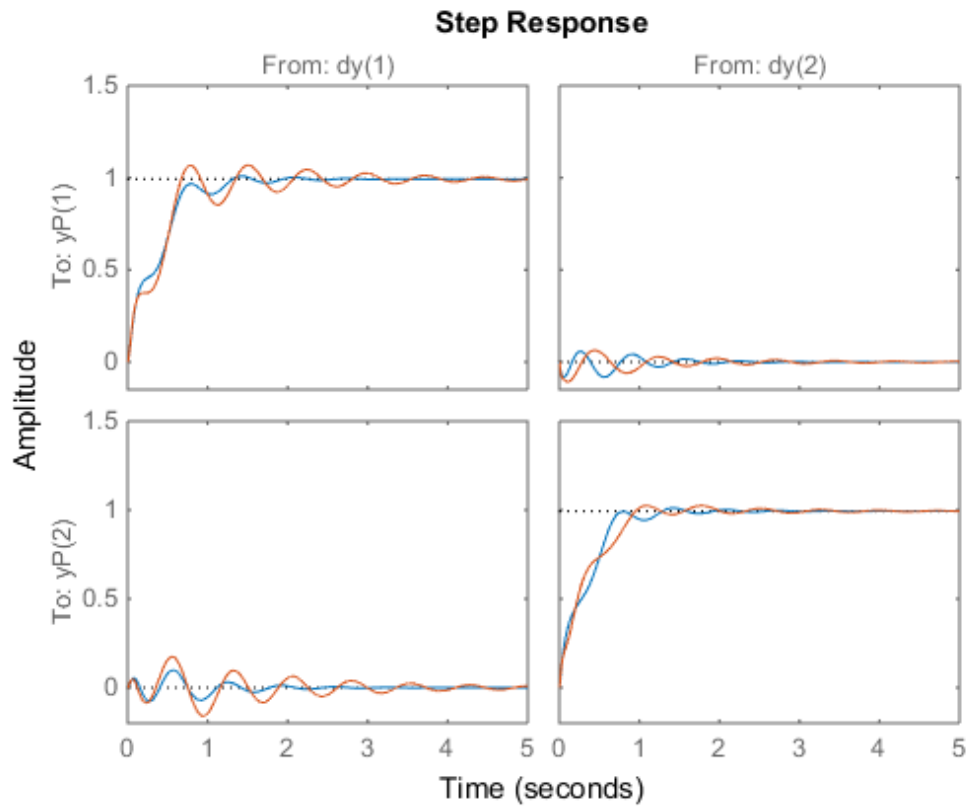
The perturbed response, which is the worst combination of uncertain values in terms of output sensitivity amplification, does not show significant degradation of the command response. The settling time is increased by about 50%, from 2 to 4, and the off-diagonal coupling is increased by about a factor of about 2, but is still quite small.

You can also examine the worst-case frequency response alongside the nominal and sampled systems using `wcgainplot`.

```
wcgainplot(F.To,{1e-1,100})
```
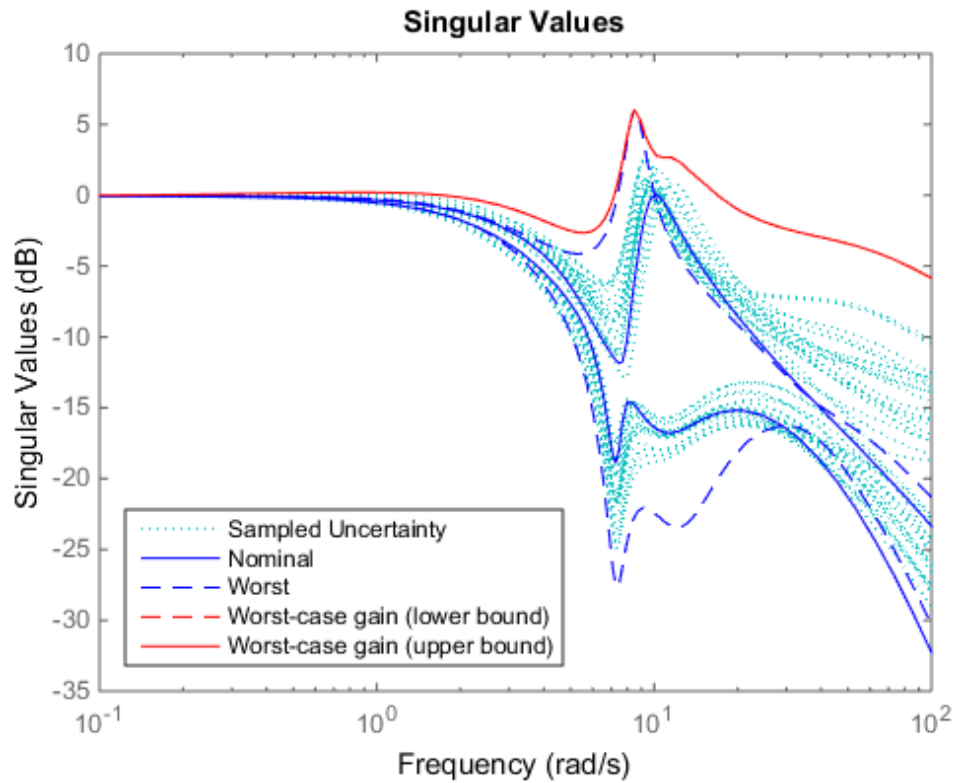
See Also
loopmargin | loopsens | robuststab | ultidyn | usubs | wcgain | wcgainplot

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.

5

H-Infinity and Mu Synthesis

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 $e \# L_2$. 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

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

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

In µ-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,

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

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

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

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

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

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

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 $\#\#_\infty$ (H_∞) norm. For this reason, this section now discusses some interpretations of the H_∞ norm.
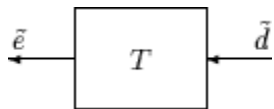
Unweighted MIMO System

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 with the arrows going from left to right as in Weighted MIMO System.
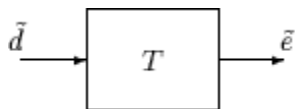
Unweighted MIMO System: Vectors from Left to Right

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

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

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 $\|\tilde{e}\|_2 \big/ \|\tilde{d}\|_2$ arbitrarily close to ß. 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 $b \in R^{n_e}$ will satisfy #b#$_2 \leq$ ß. Moreover, ß, as defined in Weighted MIMO System, is the smallest number such that this is true for every #a#$_2 \leq 1$, $\bar{\omega}$, and #.

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 # #$_\infty$ 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, along with Unweighted MIMO System: Vectors from Left to Right.

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

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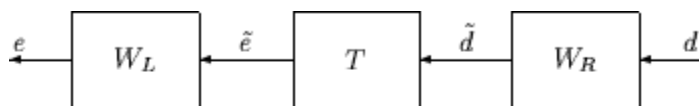
$$e = W_L \tilde{e} = W_L T \tilde{d} = W_L T W_R d$$

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. Specifically, for sinusoidal signal $\tilde{d}$, the steady-state relationship between $\tilde{e}(=T\tilde{d})$, $\tilde{d}$ and $\# W_L T W_R \#_\infty$ is as follows. The steady-state solution $\tilde{e}_{ss}$, denoted as

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

satisfies

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

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

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

satisfying

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

if and only if $\#W_L T W_R\#_\infty \leq 1$.

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

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

the steady-state error components will satisfy

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

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_∞ 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_∞, 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. K is some controller to be designed and G is the system you want to control.
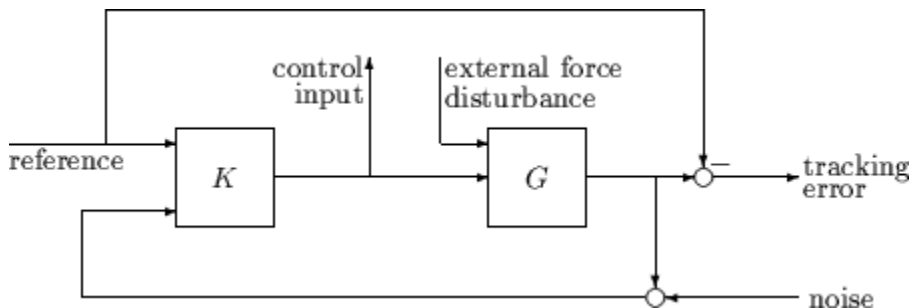
Typical Closed-Loop Performance Objective

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_∞) norm. See "Interpretation of H-Infinity Norm" on page 5-2 for an interpretation of the H_∞ 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. In the diagram, G denotes the plant model and K is the feedback controller.
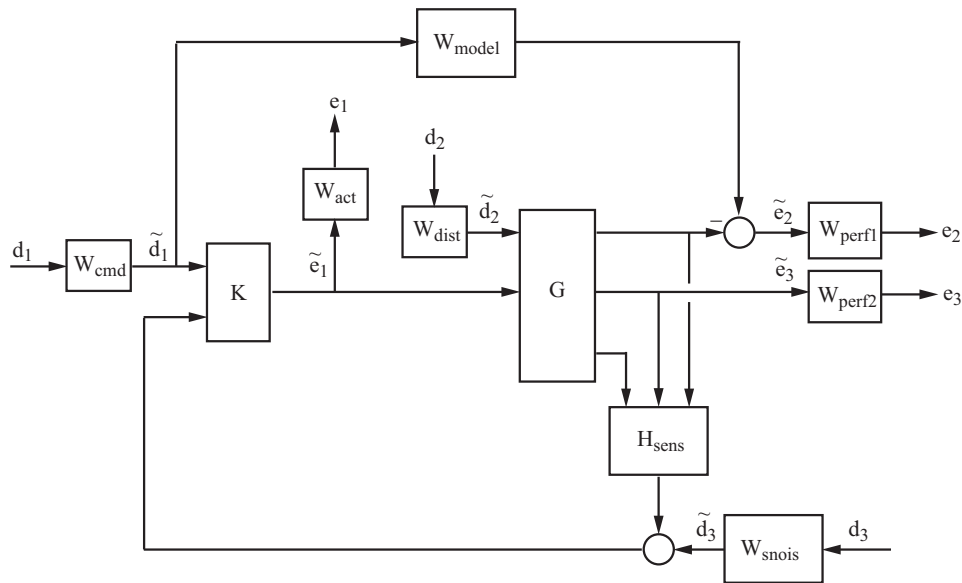
Generalized and Weighted Performance Block Diagram

The blocks in this figure might be scalar (SISO) and/or multivariable (MIMO), depending on the specific example. The mathematical objective of H_∞ 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_∞ 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.

Signal	Meaning
d_1	Normalized reference command

Signal	Meaning
$\tilde{d}_1$	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_∞ 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:

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

W$_{model}$

W_{model} represents a desired ideal model for the closed-looped 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 ω and desired damping ratio ζ. 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°/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_∞ 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_∞ 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.
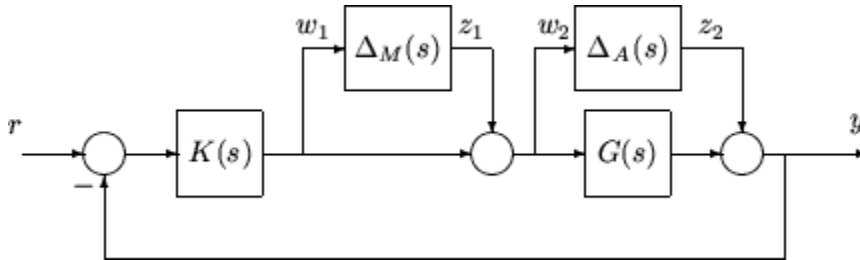
The transfer function matrices are defined as:

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

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

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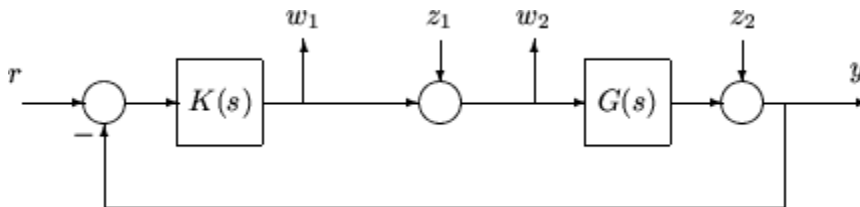
The H_∞ control robustness objective is now in the same format as the performance objectives, that is, to minimize the H_∞ norm of the transfer matrix from z, $[z_1, z_2]$, to w, $[w_1, w_2]$.

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_∞ design techniques to design a controller for a nominal quarter-car model. Then, you use μ 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.
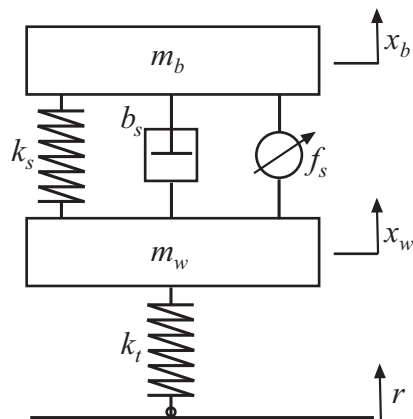
The mass, m_b, represents the car chassis (body) and the mass, m_w, represents the wheel assembly. The spring, k_s, and damper, b_s, represent the passive spring and shock absorber placed between the car body and the wheel assembly. The spring, k_t, models the compressibility of the pneumatic tire. The 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.

$$\dot{x}_1 = x_2,$$

$$\dot{x}_2 = -\frac{1}{m_b}\Big[k_s(x_1 - x_3) + b_s(x_2 - x_4) - f_s \Big],$$

$$\dot{x}_3 = x_4,$$

$$\dot{x}_4 = \frac{1}{m_w}\Big[k_s(x_1 - x_3) + b_s(x_2 - x_4) - k_t(x_3 - r) - f_s \Big].$$

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)'})
```

Gain from road dist (r) and actuator force (fs)
to body accel (ab) and suspension travel (sd)
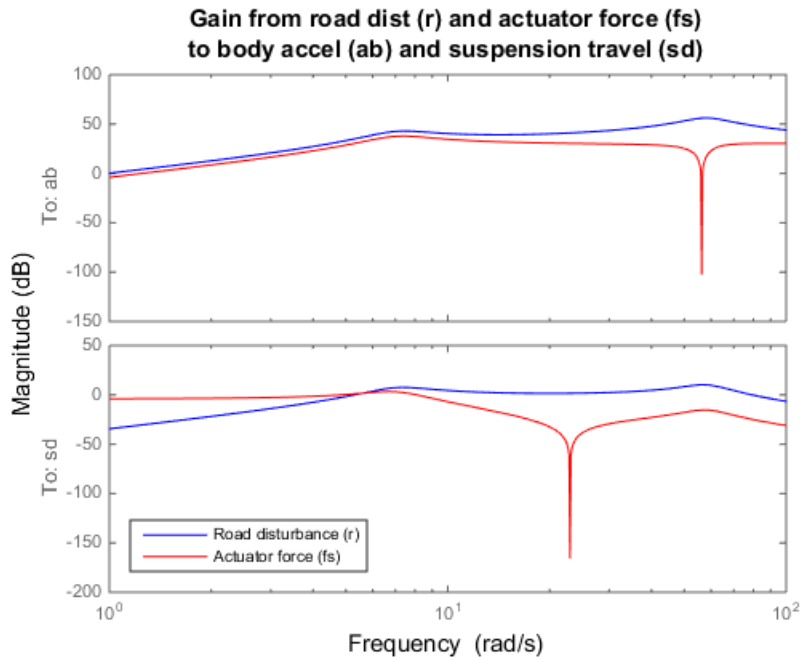
Road disturbances influence the motion of the car and suspension:

- 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:

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

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.

```
bodeplot(Act,'b',Act.NominalValue,'r+',logspace(-1,3,120))
title('Nominal and 20 random actuator models')
```
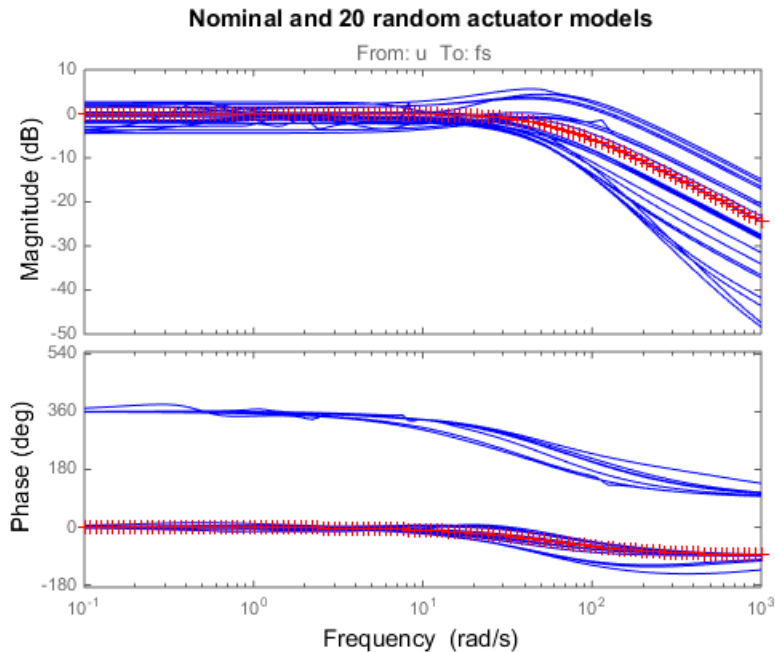
The plus (+) marker denotes the nominal actuator model. The blue solid lines represent the randomly sampled models.

Design Objectives for H-Infinity Synthesis

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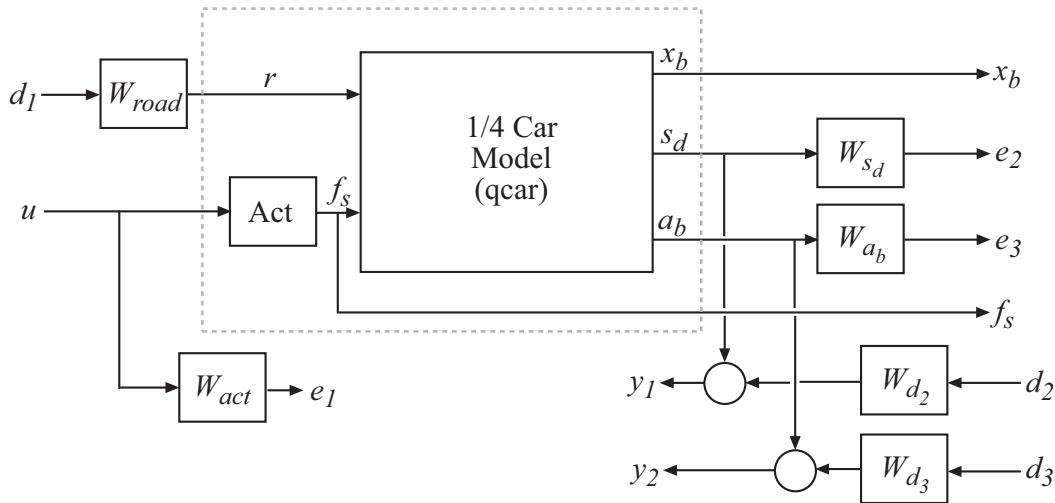
The feedback controller uses the measurements y_1 and y_2 of the suspension travel, s_d, and body acceleration, a_b, to compute the control signal, u. This control signal drives the hydraulic actuator. There are three external sources of disturbance:

- 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_{d2} and W_{d3}.

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_∞ 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_∞ 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 `Wroad = 0.07` models broadband road deflections of magnitude 7 cm. `Wact` is a highpass filter. This filter penalizes high-frequency content in the control signal, and thus limits control bandwidth. `Wd2` and `Wd3` model broadband sensor noise of intensity 0.01 and 0.5, respectively. In a more realistic design, `Wd2` and `Wd3` 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')
```

Response to road disturbance
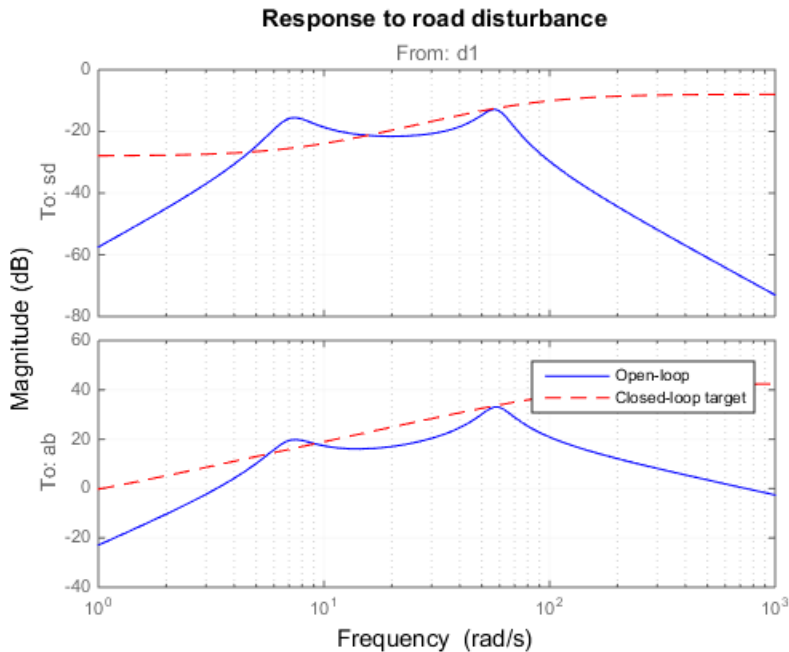
The corresponding performance weights W_{sd} and W_{ab} are the reciprocals of the comfort and handling targets. To investigate the trade-off between passenger comfort and road handling, construct three sets of weights, $(\beta W_{sd}, (1-\beta)W_{ab})$. These weights use a blending parameter, β, to 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, β. 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_∞ controller for each value of the blending parameter, β. 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_∞ norms, gamma.

gamma

gamma =

```
    0.9410
    0.6724
    0.8877
```

The three H_∞ controllers achieve closed-loop H_∞ 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')
```


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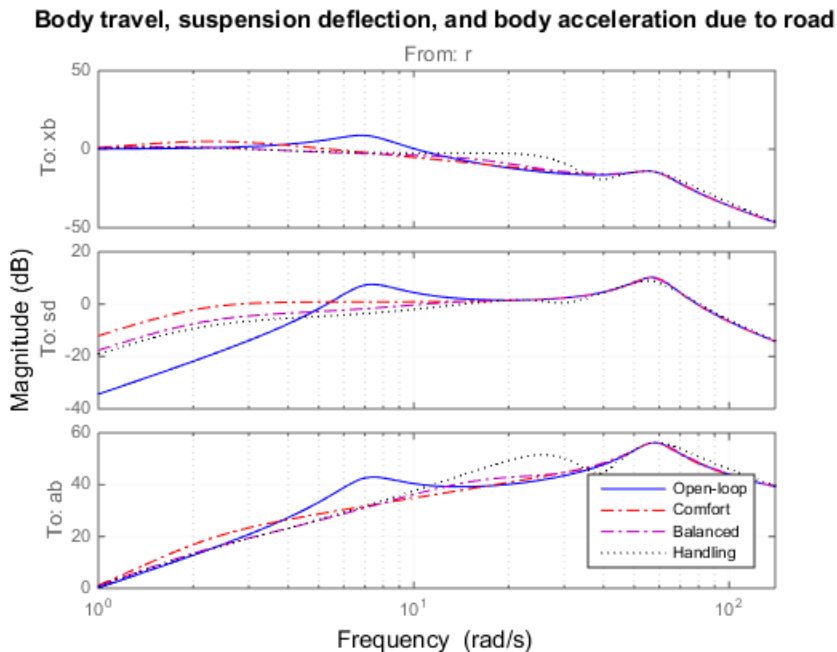
The solid blue line corresponds to the open-loop response. The other lines are the closed-loop frequency responses for the different comfort and handling blends. All three controllers reduce suspension deflection and body acceleration below the rattlespace frequency (23 rad/s).

Time-Domain Evaluation

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

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

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')
```
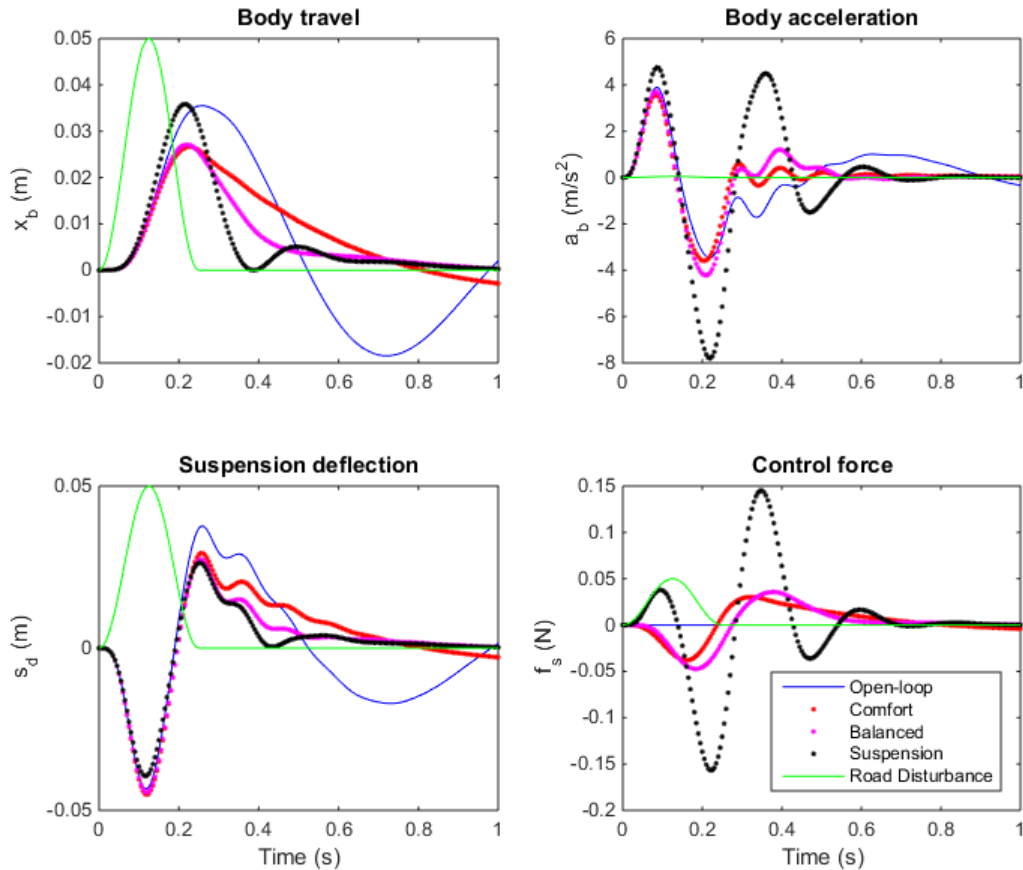

The simulations show that the body acceleration is smallest for the controller emphasizing passenger comfort. Body acceleration is largest for the controller emphasizing suspension deflection. The balanced design achieves a good tradeoff between body acceleration and suspension deflection.

Robust μ Design

So far you designed H_∞ 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 *μ*-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 *μ*-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')
```
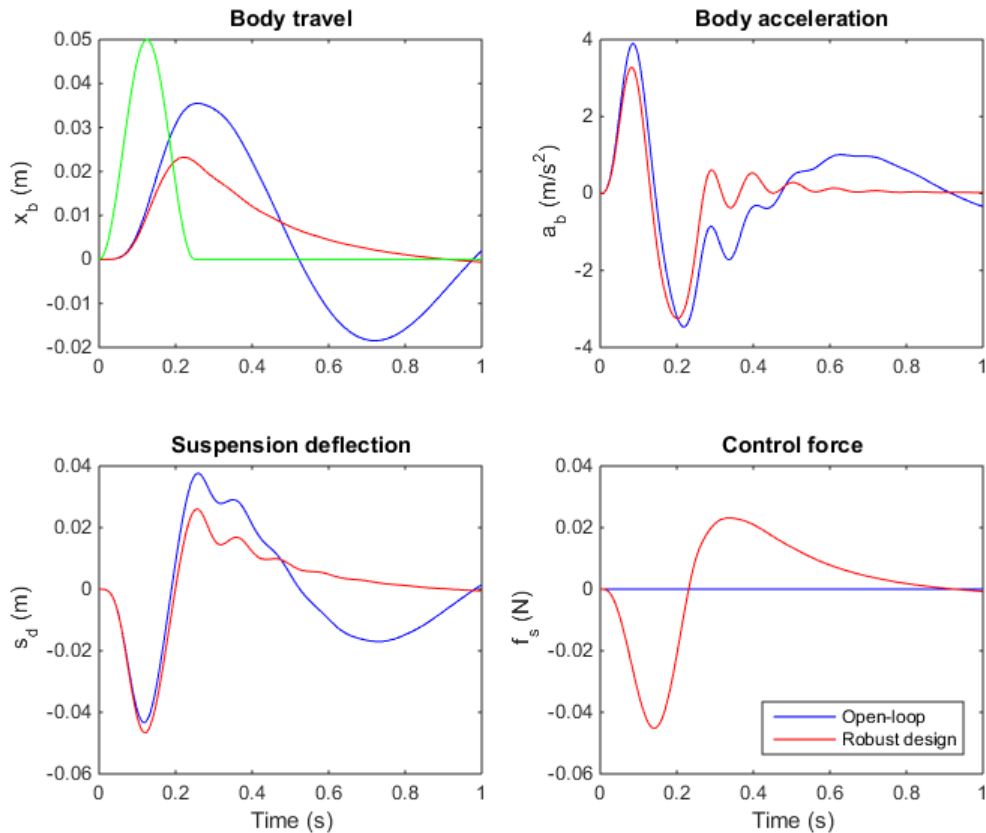

These responses are similar to those obtained with the balanced H_∞ controller.

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_∞ and the robust controllers, to compare the results.

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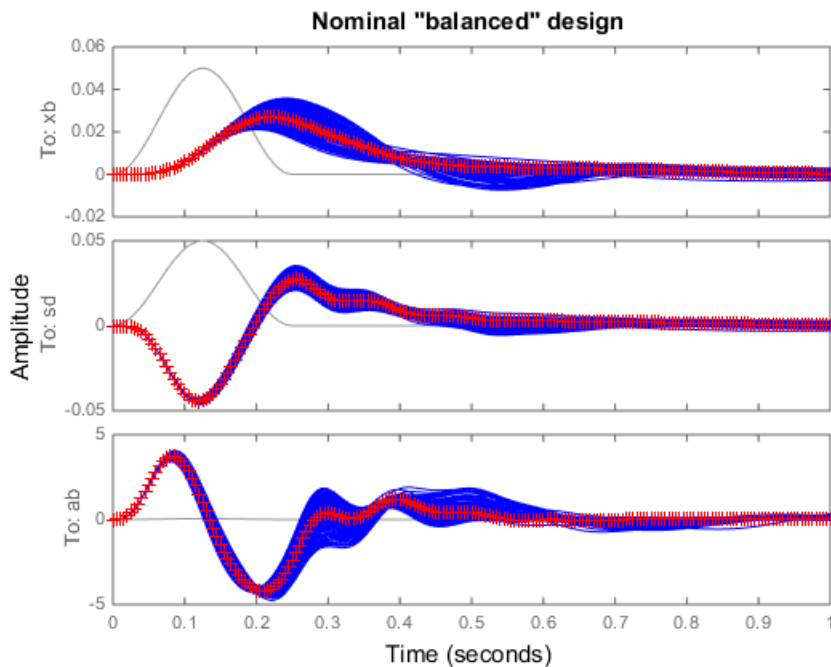
Compute an uncertain closed-loop model with the balanced robust controller, Krob. Sample this model, simulate the sampled models, and plot the results.

```
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')
```
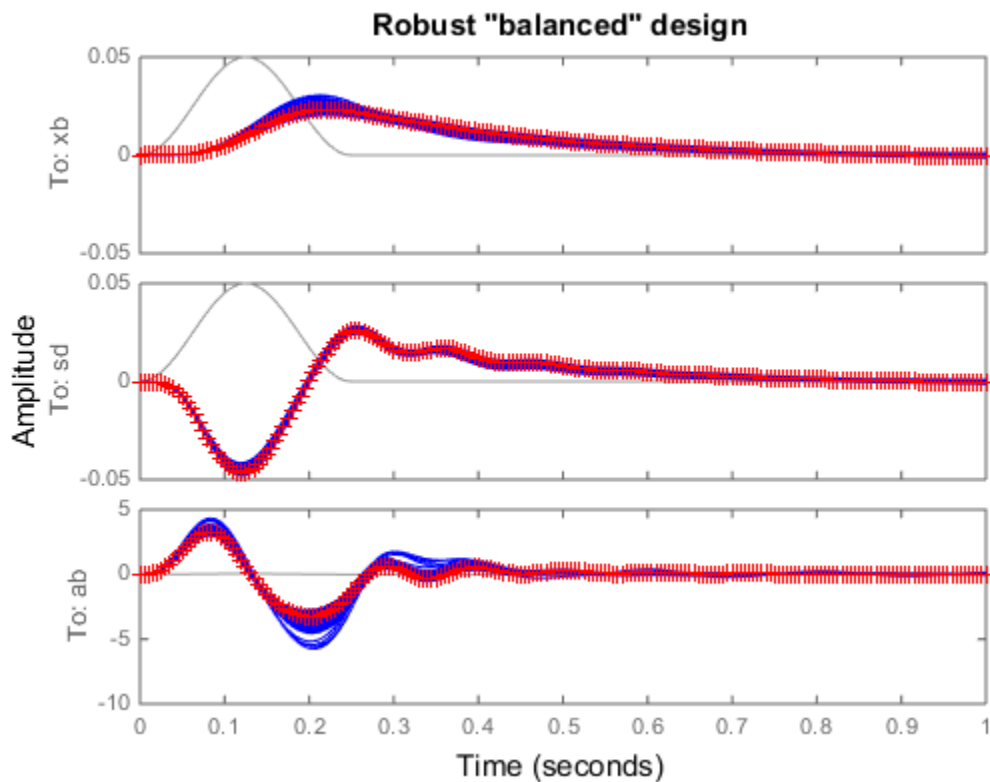

The robust controller reduces variability caused by model uncertainty, and delivers more consistent performance.

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','Location','SouthEast')
```

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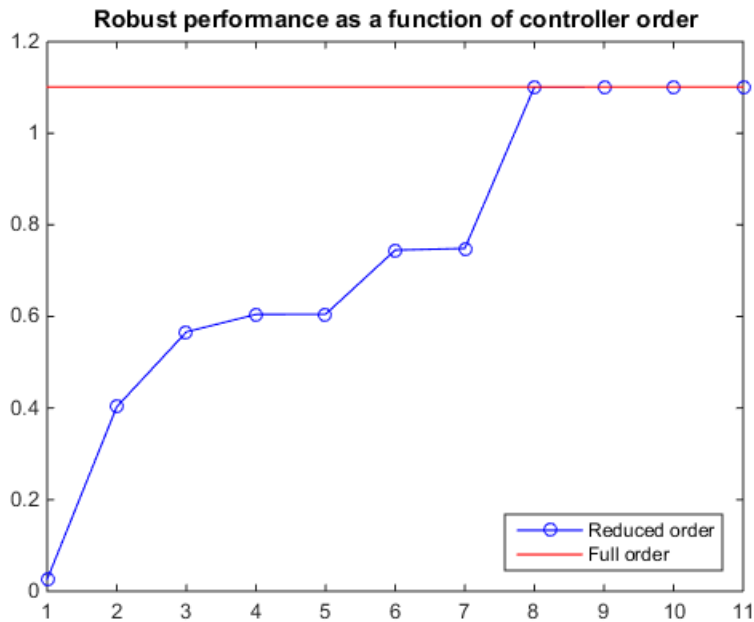
There is no significant difference in robust performance between the 8th- and 11th-order controllers. Therefore, you can safely replace Krob by its 8th-order approximation.

```
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 µ-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_∞ 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_∞ 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_∞ 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_∞ 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_∞ 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 µ 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.

6

Control System Tuning

Automated Tuning of Control Systems

In this section...

Automated Tuning Overview

The control system tuning tools of Robust Control Toolbox automatically tune control systems from high-level tuning goals you specify, such as reference tracking, disturbance rejection, and stability margins. The software jointly tunes all the free parameters of your control system regardless of control system architecture or the number of feedback loops it contains. For example, the model of the following illustration represents a multiloop control system for a helicopter.
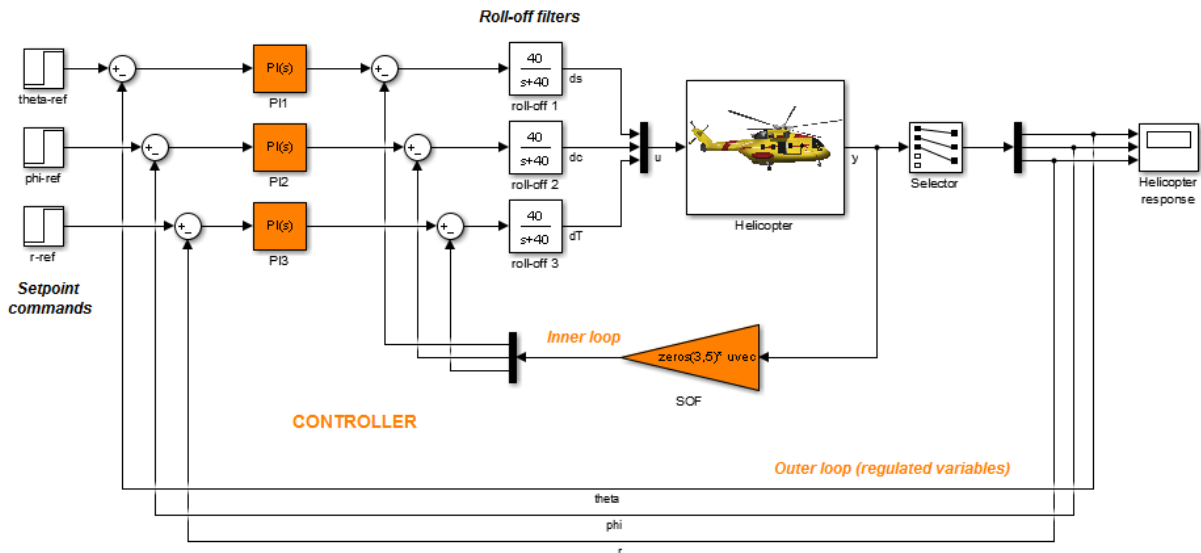
This control system includes a number of fixed elements, such as the helicopter model itself and the roll-off filters. The inner control loop provides static output feedback for

decoupling. The outer loop includes PI controllers for setpoint tracking. The Robust Control Toolbox tuning tools jointly optimize the gains in the `SOF` and `PI` blocks to meet setpoint tracking, stability margin, and other requirements that you specify. These tools allow you to specify any control structure and designate which blocks in your system are tunable.

Control systems are tuned to meet your specific performance and robustness goals subject to feasibility constraints such as actuator limits, sensor accuracy, computing power, or energy consumption. The library of *tuning goals* lets you capture these objectives in a form suitable for fast automated tuning. This library includes standard control objectives such as reference tracking, disturbance rejection, loop shapes, closed-loop damping, and stability margins. Using these tools, you can perform multi-objective tuning of control systems having any structure.

Choosing an Automated Tuning Approach

You can tune control systems at the MATLAB command line or using the Control System Tuner App.

Control System Tuner provides an interactive graphical interface for specifying your tuning goals and validating the performance of the tuned control system.
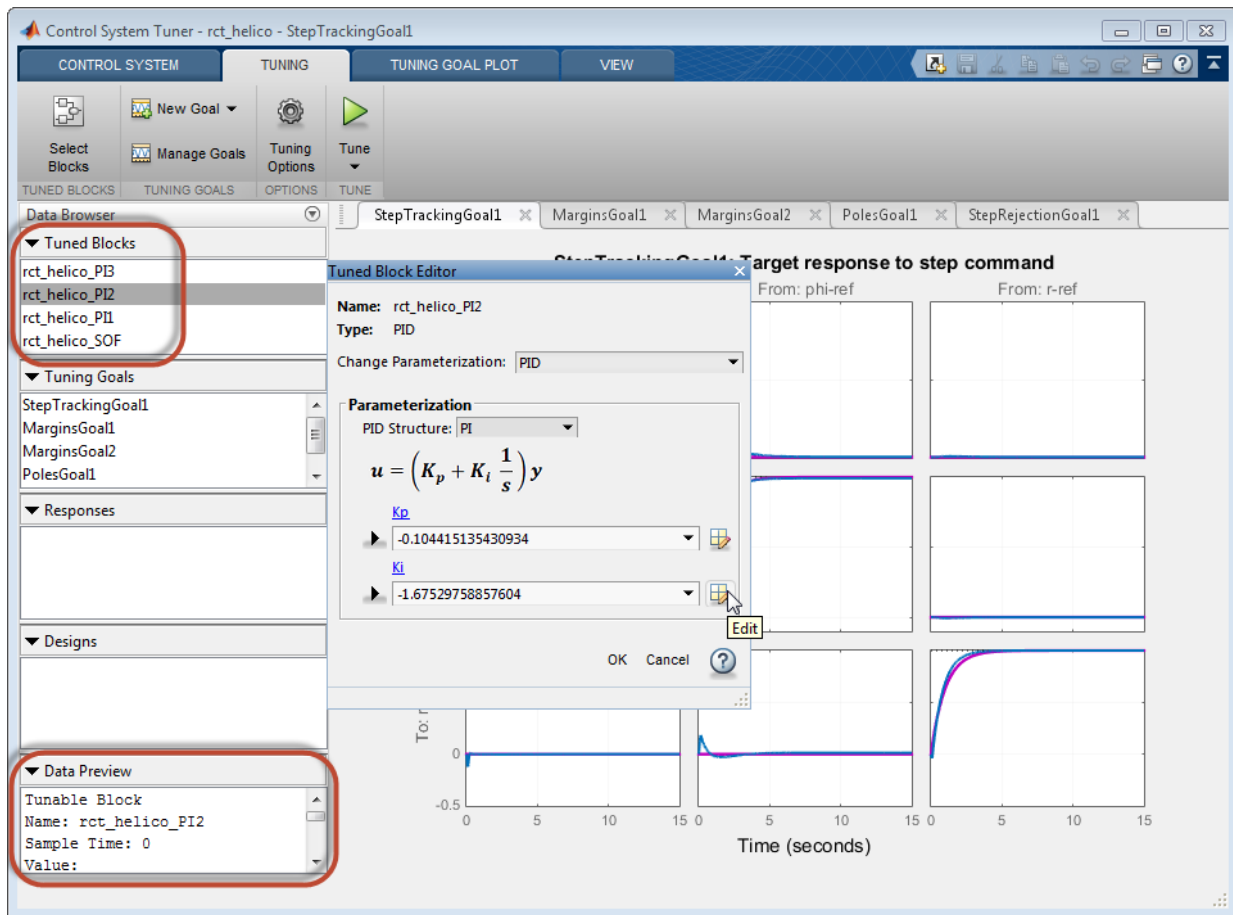
Use Control System Tuner to tune control systems consisting of any number of feedback loops, with tunable components having any structure (such as PID, gain block, or state-space). You can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design™ software, you can tune a control system represented by a Simulink model. Use the graphical interface to configure your tuning goals, examine response plots, and validate your controller design.

The `systune` command can perform all the same tuning tasks as Control System Tuner. Tuning at the command line allows you to write scripts for repeated tuning tasks. `systune` also provides advanced techniques such as tuning a controller for multiple

plants, or designing gain-scheduled controllers. To use the command-line tuning tools, you can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design software, you can tune a control system represented by a Simulink model using an `slTuner` interface. Use the `TuningGoal` requirement objects to configure your tuning goals. Analysis commands such as `getIOTransfer` and `viewSpec` let you examine and validate the performance of your tuned system.

Related Examples
- "Specify Control Architecture in Control System Tuner" on page 6-17
- "Tuning Control Systems with CONTROL SYSTEM TUNER" on page 6-263

More About
- "Automated Tuning Workflow" on page 6-8

Automated Tuning Workflow

Whether you are tuning a control system at the command line or using Control System Tuner, the basic workflow includes the following steps:

1 Define your control architecture, by building a model of your control system from fixed-value blocks and blocks with tunable parameters. You can do so in one of several ways:

- Create a Simulink model of your control system. (Tuning a Simulink model requires Simulink Control Design software.)
- Use a predefined control architecture available in Control System Tuner.
- At the command line, build a tunable `genss` model of your control system out of numeric LTI models and tunable control design blocks.

For more information, see "Specify Control Architecture in Control System Tuner" on page 6-17.

2 Set up your model for tuning.

- In Control System Tuner, identify which blocks of the model you want to tune. See "Model Setup for Control System Tuner".
- If tuning a Simulink model at the command line, create and configure the `slTuner` interface to the model. See "Setup for Tuning Simulink Models at the Command Line".

3 Specify your tuning goals. Use the library of tuning goals to capture requirements such as reference tracking, disturbance rejection, stability margins, and more.

- In Control System Tuner, use the graphical interface to specify tuning goals. See "Tuning Goals".
- At the command-line, use the `TuningGoal` requirement objects to specify your tuning goals. See " at the Command Line".

4 Tune the model. Use the `systune` command or Control System Tuner to optimize the tunable parameters of your control system to best meet your specified tuning goals.

- For tuning in Control System Tuner, see "Parameter Tuning".
- For tuning at the command line, see "Parameter Tuning".

5 Analyze system response and validate the design. Whether at the command line or in Control System Tuner, you can plot system responses to examine any aspects of system performance you need to validate your design.

- For validation in Control System Tuner, see "Analysis and Validation".
- For validating a tuned Simulink model at the command line, see "Validation of Tuned Simulink Models".
- For validating a tuned `genss` model at the command line, see "Validation of Tuned MATLAB Models".

Control System Tuner

Add Signal From the Model

Use this dialog box to add signal locations from a Simulink model to the specifications for tuning goals or response plots. For example, When you create a new response plot using

New Plot, the software prompts you to specify input signals, output signals, and loop-opening locations. Similarly, when you create a new tuning goal, the software prompts you for signal locations at which to apply the tuning goal. Adding signals to these lists opens the **Add Signal From the Model** dialog box.

To add a signal from the Simulink model:

1   In the Simulink model, select the signal you want to add. Shift-click to select multiple signals at once.

The **Add Signal From the Model** dialog box displays the signals that are currently selected in the model.
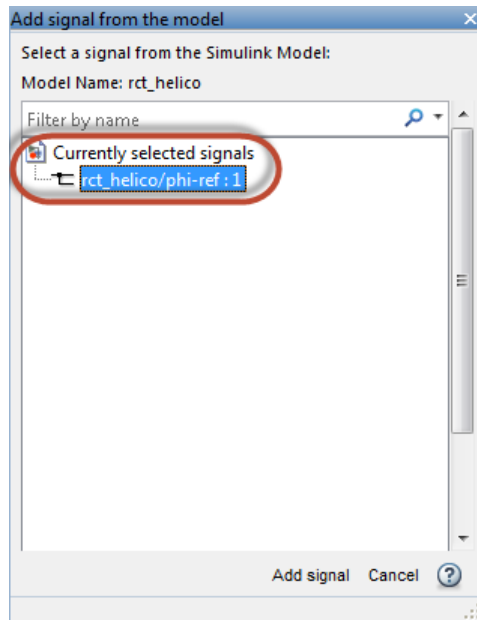
2 In the **Add Signal From the Model** dialog box, click **Add Signal**. The **Add Signal From the Model** dialog box closes, returning you to the plot or new tuning goal specification. The signals you selected are added to the signal list.

Tip When multiple signals are selected, start typing in the **Filter by name** field to filter the list.

Add Bus Element

To select an individual element of a bus signal:

1 In the Simulink model, select the bus signal that contains the signal you want to add.

The **Add Signal From the Model** dialog box displays the name of the currently selected bus. The display also includes signals and nodes within the bus.
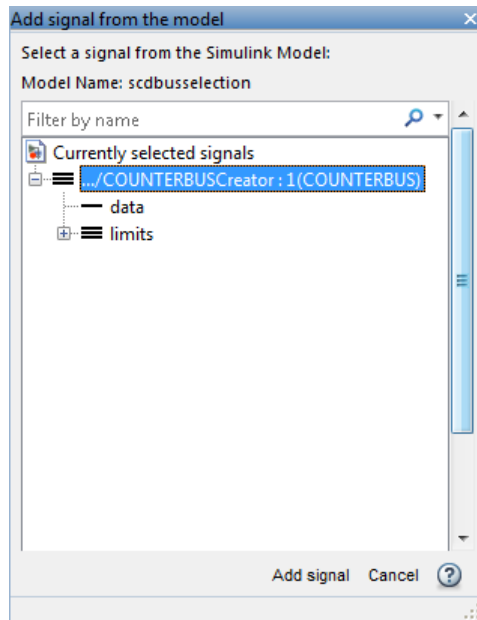
2   In the **Add Signal From the Model** dialog box, find the signal that you want to select. If necessary, expand nodes within the bus.

Tip For large buses, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter a MATLAB "regular expression".

To modify the filtering options, click ▾ adjacent to the **Filter by name** edit box.

Filtering Options

- **Enable regular expression**
  MATLAB "regular expression" for filtering signal names. For example, entering t$ displays all signals whose names end with a lowercase t (and their immediate parents).

- **Show filtered results as a flat list**

  Flat list format to display the list of filtered signals.
  By default, filtered signals are displayed using a tree format. The flat list format uses dot notation to reflect the hierarchy of bus signals.

---
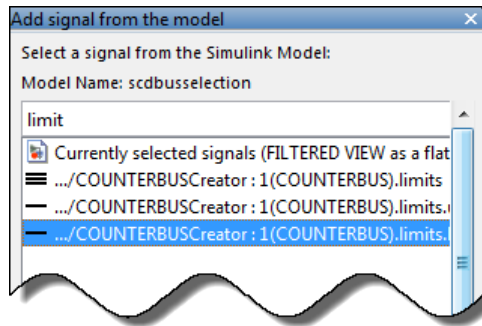
3 When you find the signal you want to add, click to select it. Then click **Add Signal**. The **Add Signal From the Model** dialog box closes, returning you to the plot or new tuning goal specification. The signal you selected is added to the signal list.

Specify Multiple Operating Points

This dialog box allows you to tune your controller for a Simulink model linearized at multiple operating points. Doing so helps to ensure that your tuned controller meets your design requirements at a variety of operating conditions.
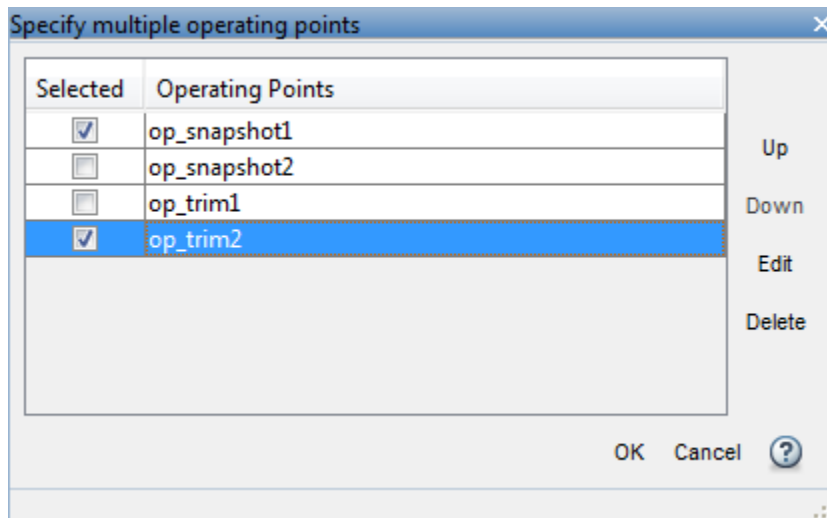
The **Operating Points** list contains all the operating points you have created for the model in Control System Tuner. Check the **Selected** box for an operating point to activate it for tuning.

To reorder operating points in the list, use the **Up** and **Down** buttons. Control System Tuner places linearized models in an array in the order of operating points in this list. When you use the **Apply goal to** option to restrict a tuning goal to a subset of linearized models, enter the array indices corresponding to the operating points you want to include.

Related Examples

- "Specify Operating Points for Tuning in Control System Tuner" on page 6-24

Linearization Options

Use this dialog box to set options for the linearization that Control System Tuner performs on your Simulink model.

Working Domain

Specify whether to compute linearization in continuous time or discrete time. If in discrete time, specify a sample time.

Rate Conversions

Specify the method used for rate conversion when linearizing a multirate system.

- `'Zero-Order Hold'` — Yields best match in the time domain for staircase inputs.
- `Tustin` — Yields best match in the frequency domain.
- `Tustin with Prewarping` — Tustin method with best match at a particular frequency. Enter the prewarping frequency in the text box.
- `Upsample when possible, Zero-Order Hold otherwise` — Upsamples discrete states when possible, and uses zero-order hold otherwise.
- `Upsample when possible, Tustin otherwise` — Upsamples discrete states when possible, and uses Tustin method otherwise.
- `Upsample when possible, Tustin otherwise` — Upsamples discrete states when possible, and uses Tustin method with prewarping otherwise. Enter the prewarping frequency in the text box.

Standard feedback configuration

Use this dialog box to specify values for the fixed blocks and structures for the tuned blocks in the standard feedback configuration:
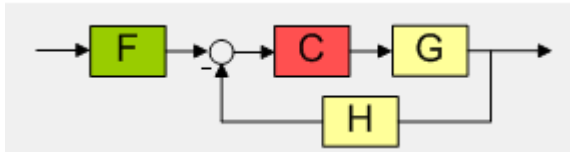
- **Fixed components** — The plant model *G* and the sensor dynamics *H* are fixed components. Each of these components defaults to a SISO unity-gain transfer function.

   To change the value of one of these components, check **Specify new value** and enter the new value in the text box. The new value can be any numeric LTI model, such as a `tf`, `zpk`, or `ss` model. If the LTI model is in the MATLAB workspace, enter the variable name in the text box.

- **Tunable components** — The controller block *C* and the filter *F* are tunable components. When you tune the control system, Control System Tuner adjusts the parameters of these components to meet your tuning goals. Each of the tunable components defaults to a SISO tunable gain (an `ltiblock.gain` block).

To specify a different structure for these blocks, use Control Design Blocks to create tunable components. For example, to make *C* a PID controller, use `ltiblock.pid` to create a tunable PID block in the MATLAB workspace. Then check **Specify new value** for *C* and enter the variable name in the text box.

(See "Tunable Models" for more information about creating tunable components.)

All the components can be MIMO, provided the dimensions are compatible. For example, if the plant has two inputs and two outputs, then all the components must have two inputs and two outputs.

Generalized feedback configuration

View and change the custom control architecture and controller structure represented by a `genss` model.

The display box lists a summary of the `genss` model specifying the current control architecture. The display lists the tunable blocks and their current values.

To specify a different control architecture that is represented by a `genss` model in the MATLAB workspace, enter model's variable name in the **Enter expression or variable for custom parameterization** text box.

For more information, see "Arbitrary Feedback Control Architecture" on page 6-19.

Specify Control Architecture in Control System Tuner

About Control Architecture

Control System Tuner lets you tune a control system having any architecture. *Control system architecture* defines how your controllers interact with the system under control. The architecture comprises the tunable control elements of your system, additional filter and sensor components, the system under control, and the interconnections between all these elements. For example, a common control system architecture is the single-loop feedback configuration of the following illustration:
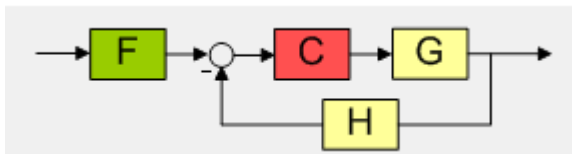
G is the plant model, and H the sensor dynamics. These are usually the fixed components of the control system. The prefilter F and feedback controller C are the tunable elements. Because control systems are so conveniently expressed in this block diagram form, these elements are referred to as fixed blocks and tunable blocks.

Control System Tuner gives you several ways to define your control system architecture:

- Use the predefined feedback structure of the illustration.
- Model any control system architecture in MATLAB by building a generalized state-space (`genss`) model from fixed LTI components and tunable control design blocks.
- Model your control system in Simulink and specify the blocks to tune in Control System Tuner (requires Simulink Control Design software).

Predefined Feedback Architecture

If your control system has the single-loop feedback configuration of the following illustration, use the predefined feedback structure built into Control System Tuner.
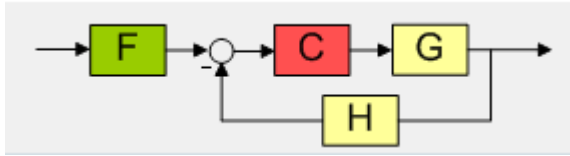
For example, suppose you have a DC motor for which you want to tune a PID controller. The response of the motor is modeled as $G(s) = 1/(s + 1)^2$. Create a fixed LTI model representing the plant, and a tunable PID controller model.

```
Gmot = zpk([],[-1,-1],1);
Cmot = ltiblock.pid('Cmot','PID');
```

Open Control System Tuner.

```
controlSystemTuner
```

Control System Tuner opens, set to tune this default architecture. Next, specify the values of the blocks in the architecture. Click ⊡ to open the **Standard feedback configuration** dialog box.
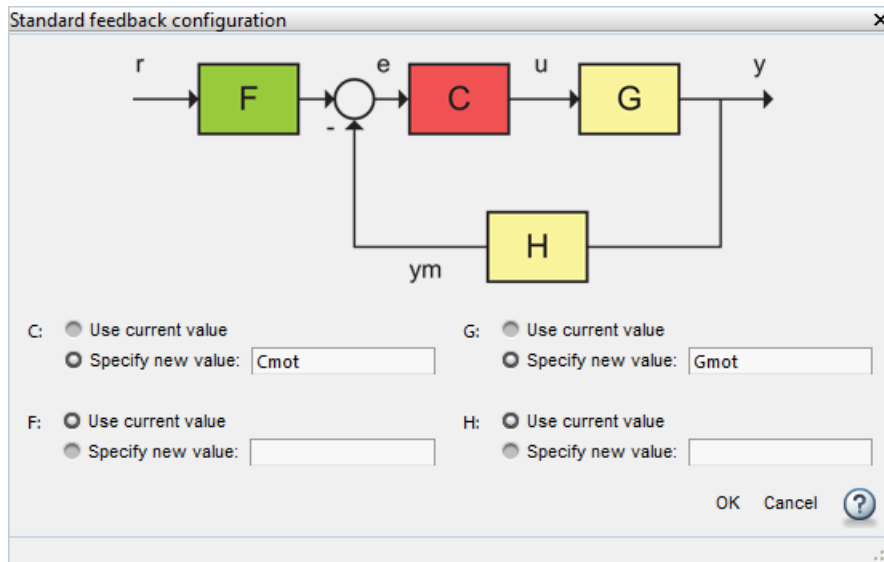
Enter the values for *C* and *G* that you created. Control System Tuner reads these values from the MATLAB workspace. Click **OK**.

The default value for the sensor dynamics is a fixed unity-gain transfer function. The default value for the filter *F* is a tunable gain block.

You can now select blocks to tune, create tuning goals, and tune the control system.

Arbitrary Feedback Control Architecture

If your control architecture does not match Control System Tuner's predefined control architecture, you can create a generalized state-space (`genss`) model with tunable components representing your controller elements. For example, suppose you want to tune the cascaded control system of the following illustration, that includes two tunable PID controllers.

.
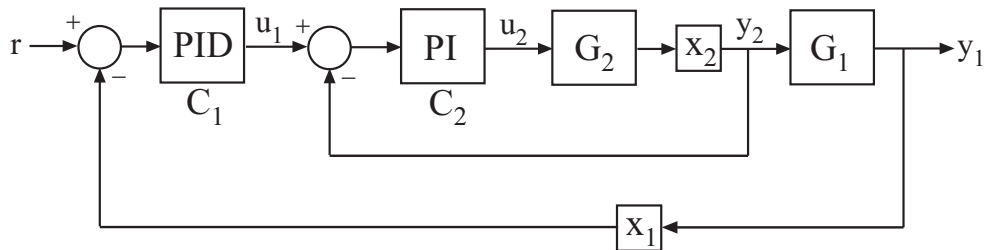
Create tunable control design blocks for the controllers, and fixed LTI models for the plant components, G_1 and G_2. Also include optional loop-opening locations x_1 and x_2. These locations indicate where you can open loops or inject signals for the purpose of specifying requirements for tuning the system.

```
G2 = zpk([],-2,3);
G1 = zpk([],[-1 -1 -1],10);

C20 = ltiblock.pid('C2','pi');
C10 = ltiblock.pid('C1','pid');

X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20,1);
CL0 = feedback(G1*InnerLoop*C10,X1);
CL0.InputName = 'r';
CL0.OutputName = 'y';
```

CL0 is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Open Control System Tuner to tune this model.

```
controlSystemTuner(CL0)
```

You can now select blocks to tune, create tuning goals, and tune the control system.

Related Examples

- "Building Tunable Models"

- "Specify Blocks to Tune in Control System Tuner" on page 6-31

• "Specify Goals for Interactive Tuning" on page 6-44

Control System Architecture in Simulink

If you have Simulink Control Design software, you can model an arbitrary control system architecture in a Simulink model and tune the model in Control System Tuner.

See "Open Control System Tuner for Tuning Simulink Model" on page 6-22.

Open Control System Tuner for Tuning Simulink Model

To open Control System Tuner for tuning a Simulink model, open the model. In the Simulink Editor, select **Analysis** > **Control Design** > **Control System Tuner**.
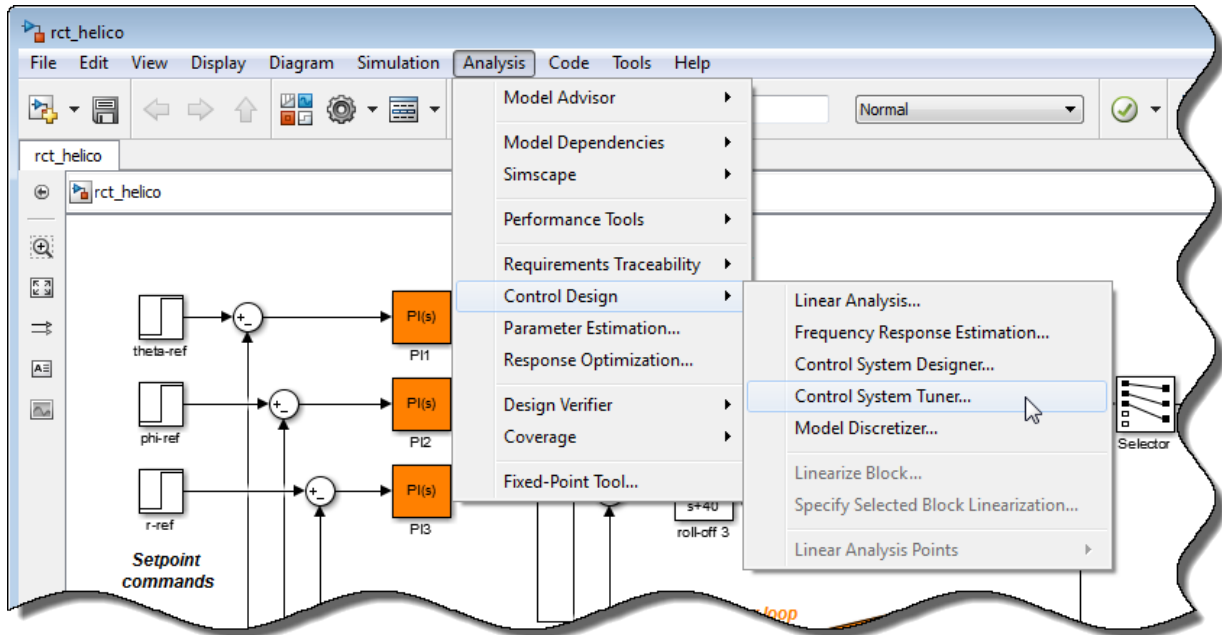
Each instance of Control System Tuner is linked to the Simulink model from which it is opened. The title bar of the Control System Tuner window reflects the name of the associated Simulink model.
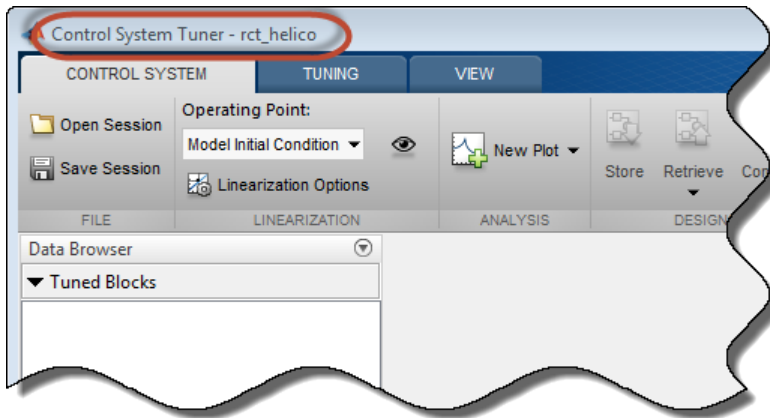
Command-Line Equivalents

At the MATLAB command line, use the `controlSystemTuner` command to open Control System Tuner for tuning a Simulink model. For example, the following command opens Control System Tuner for the model `rct_helico.slx`.

```
controlSystemTuner('rct_helico')
```

If `SLT0` is an `slTuner` interface to the Simulink model, the following command opens Control System Tuner using the information in the interface, such as blocks to tune and analysis points.

```
controlSystemTuner(SLT0)
```

Related Examples

- "Specify Operating Points for Tuning in Control System Tuner" on page 6-24
- "Specify Blocks to Tune in Control System Tuner" on page 6-31

More About

- "Automated Tuning Workflow" on page 6-8

Specify Operating Points for Tuning in Control System Tuner

In this section...
"About Operating Points in Control System Tuner" on page 6-24
"Linearize at Simulation Snapshot Times" on page 6-24
"Compute Operating Points at Simulation Snapshot Times" on page 6-26
"Compute Steady-State Operating Points" on page 6-28

About Operating Points in Control System Tuner

When you use Control System Tuner with a Simulink model, the software computes system responses and tunes controller parameters for a linearization of the model. That linearization can depend on model operating conditions.

By default, Control System Tuner linearizes at the operating point specified in the model, which comprises the initial state values in the model (the model initial conditions). You can specify one or more alternate operating points for tuning the model. Control System Tuner lets you compute two types of alternate operating points:

*   Simulation snapshot time. Control System Tuner simulates the model for the amount of time you specify, and linearizes using the state values at that time. Simulation snapshot linearization is useful, for instance, when you know your model reaches an equilibrium state after a certain simulation time.

    For more information about simulation snapshot linearization, see "Choosing Between Simulation Snapshot and Operating Point from Specifications" in the Simulink Control Design documentation.
*   Steady-state operating point. Control System Tuner finds a steady-state operating point at which some specified condition is met (trimming). For example, if your model represents an automobile motor, you can compute an operating point at which the motor operates steadily at 2000 rpm.

    For more information about steady-state operating points, see "Steady-State Operating Point (Trimming)" in the Simulink Control Design documentation.

Linearize at Simulation Snapshot Times

This example shows how to compute linearizations at one or more simulation snapshot times.

In the **Control System** tab, in the **Operating Point** menu, select `Linearize At`.
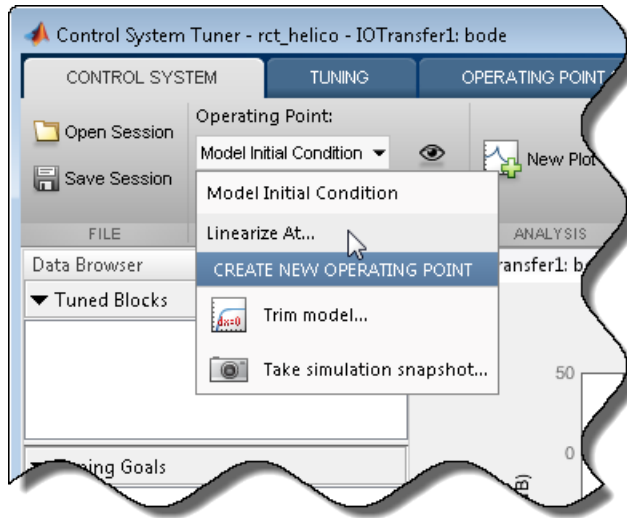
In the **Enter snapshot times to linearize** dialog box, specify one or more simulation snapshot times. Click **OK**.
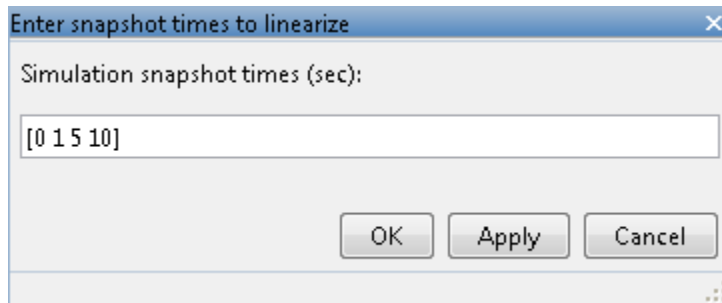
When you are ready to analyze system responses or tune your model, Control System Tuner computes linearizations at the specified snapshot times. If you enter multiple snapshot times, Control System Tuner computes an array of linearized models, and displays analysis plots that reflect the multiple linearizations in the array. In this case, Control System Tuner also takes into account all linearizations when tuning parameters.

This helps to ensure that your tuned controller meets your design requirements at a variety of operating conditions.

Compute Operating Points at Simulation Snapshot Times

This example shows how to compute operating points at one or more simulation snapshot times. Doing so stores the operating point within Control System Tuner. When you later want to analyze or tune the model at a stored operating point, you can select the stored operating point from the **Operating Point** menu.

In the **Control System** tab, in the **Operating Point** menu, select `Take simulation snapshot`.
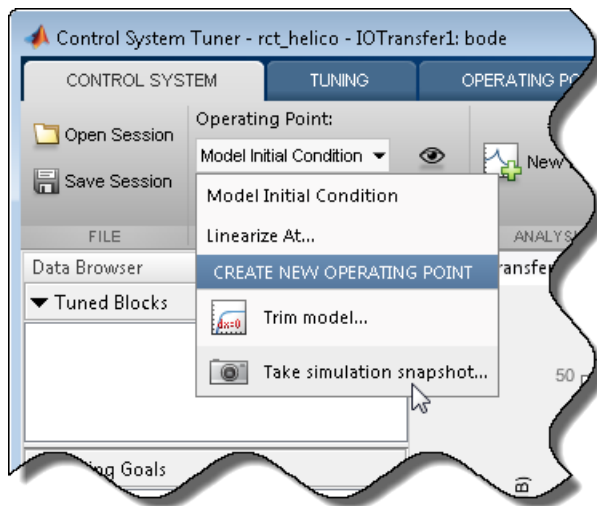
In the **Operating Point Snapshots** tab, in the **Simulation Snapshot Times** text box, enter one or more simulation snapshot times.
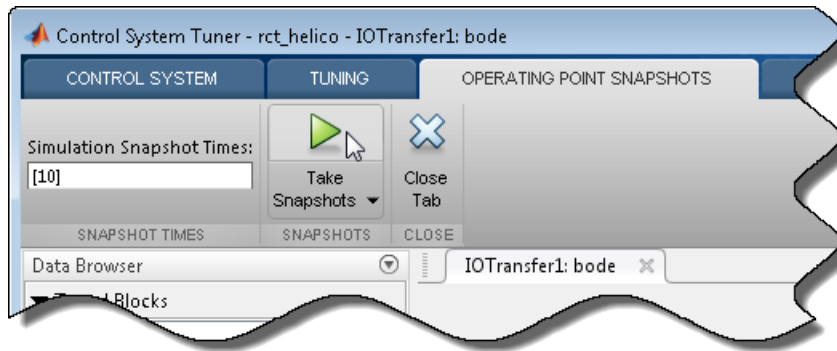
Click ▶.

Control System Tuner simulates the model and computes the snapshot operating points.

Compute additional snapshot operating points if desired. Enter additional snapshot

times and click .

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.
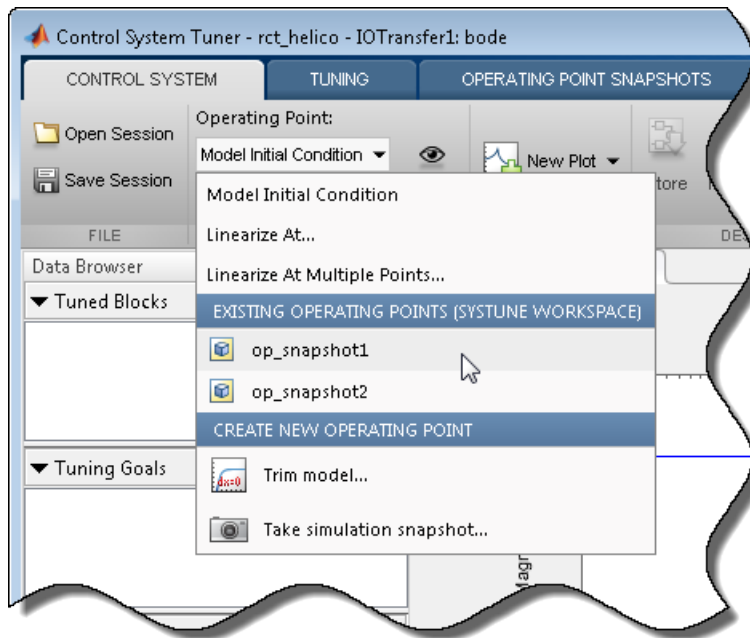
Compute Steady-State Operating Points

This example shows how to compute a steady-state operating point with specified conditions. Doing so stores the operating point within Control System Tuner. When you later want to analyze or tune the model at a stored operating point, you can select the stored operating point from the **Operating Point** menu.

In the **Control System** tab, in the **Operating Point** menu, select `Trim model`.
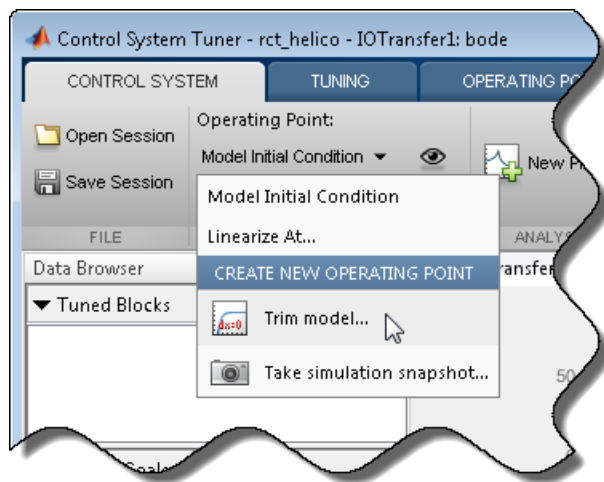
The **Trim Model** tab opens.

Enter the specifications for the steady-state state values at which you want to find an operating point. Click ▶▪ **Specifications** to open the **Specifications for trim** dialog box in which you can specify target state values.

For examples showing how to use **Specifications for trim** to specify the conditions for a steady-state operating point search, see "Steady-State Operating Points (Trimming) from Specifications" in the Simulink Control Design documentation.

When you have entered your state specifications, click ▶. Control System Tuner finds an operating point that meets the state specifications and stores it.

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.
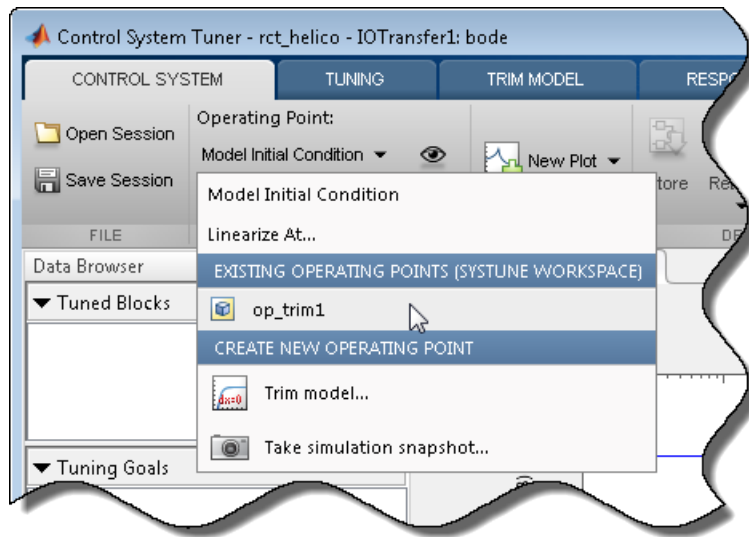
Related Examples

· "Specify Blocks to Tune in Control System Tuner" on page 6-31

Specify Blocks to Tune in Control System Tuner

To select which blocks of your Simulink model to tune:

1   In the **Tuning** tab, click ⊞ **Select Blocks**. The **Select tuned Blocks** dialog opens.

2   Click **Add Blocks**. Control System Tuner analyzes your model to find blocks that can be tuned.

3   In the **Select Blocks to Tune** dialog box, use the nodes in the left panel to navigate through your model structure to the subsystem that contains blocks you want to tune. Check **Tune?** for the blocks you want to tune. The parameters of blocks you do not check remain constant when you tune the model.
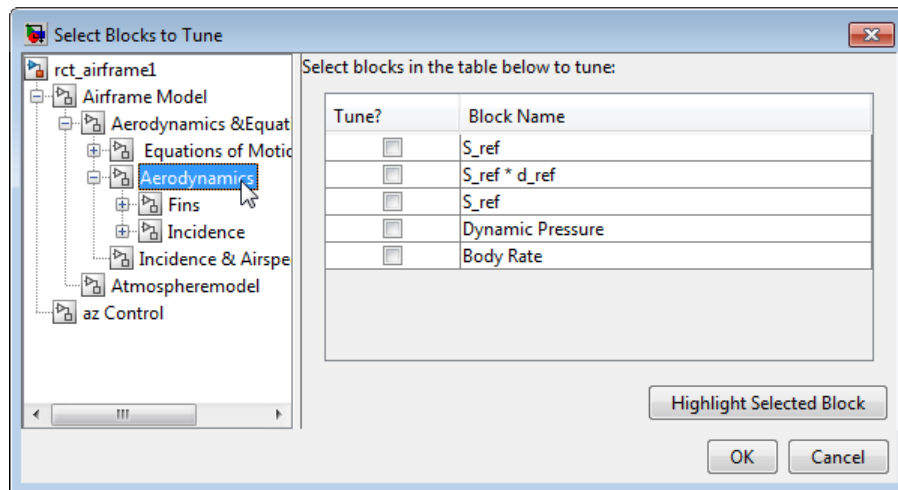
Tip To identify the location of a block in your model, select the block in the **Block Name** list and click **Highlight Selected Block**.

---

4   Click **OK**. The **Select tuned blocks** dialog box now reflects the blocks you added.
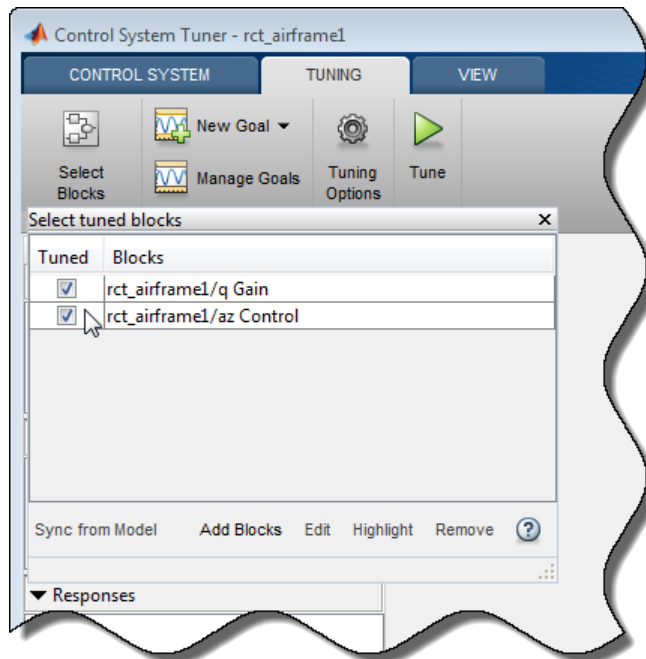
To import the current value of a block from your model into the current design in Control System Tuner, select the block in the **Blocks** list and click **Sync from Model**. Doing so is useful when you have tuned a block in Control System Tuner, but wish to restore that block to its original value. To store the current design before restoring a block value, in the **Control System** tab, click ⬛ **Store**.

Related Examples

- "View and Change Block Parameterization in Control System Tuner" on page 6-33

More About

- "How Tuned Simulink Blocks Are Parameterized" on page 6-42

View and Change Block Parameterization in Control System Tuner

Every block that you designate for tuning is parameterized in Control System Tuner.

- When you tune a Simulink model, Control System Tuner automatically assigns a default parameterization to tunable blocks in the model. The default parameterization depends on the type of block. For example, a PID Controller block configured for PI structure is parameterized by proportional gain and integral gain as follows:

$$u = K_p + K_i \frac{1}{s}.$$

K_p and K_i are the tunable parameters whose values are optimized to satisfy your specified tuning goals.

- When you tune a predefined control architecture or a MATLAB (generalized state-space) model, you define the parameterization of each tunable block when you create it at the MATLAB command line. For example, you can use `ltiblock.pid` to create a tunable PID block.

Control System Tuner lets you view and change the parameterization of any block to be tuned. Changing the parameterization can include changing the structure or current parameter values. You can also designate individual block parameters fixed (non-tunable) or limit their tuning range.

To view and change the parameterization of a block:

1   In the **Tuning** tab, click ⊞ **Select Blocks**.

2   In the **Select tuned blocks** dialog box, in the **Blocks** list, select a block.

3   Click **Edit**. The **Tuned Block Editor** dialog box opens, displaying the current block parameterization.
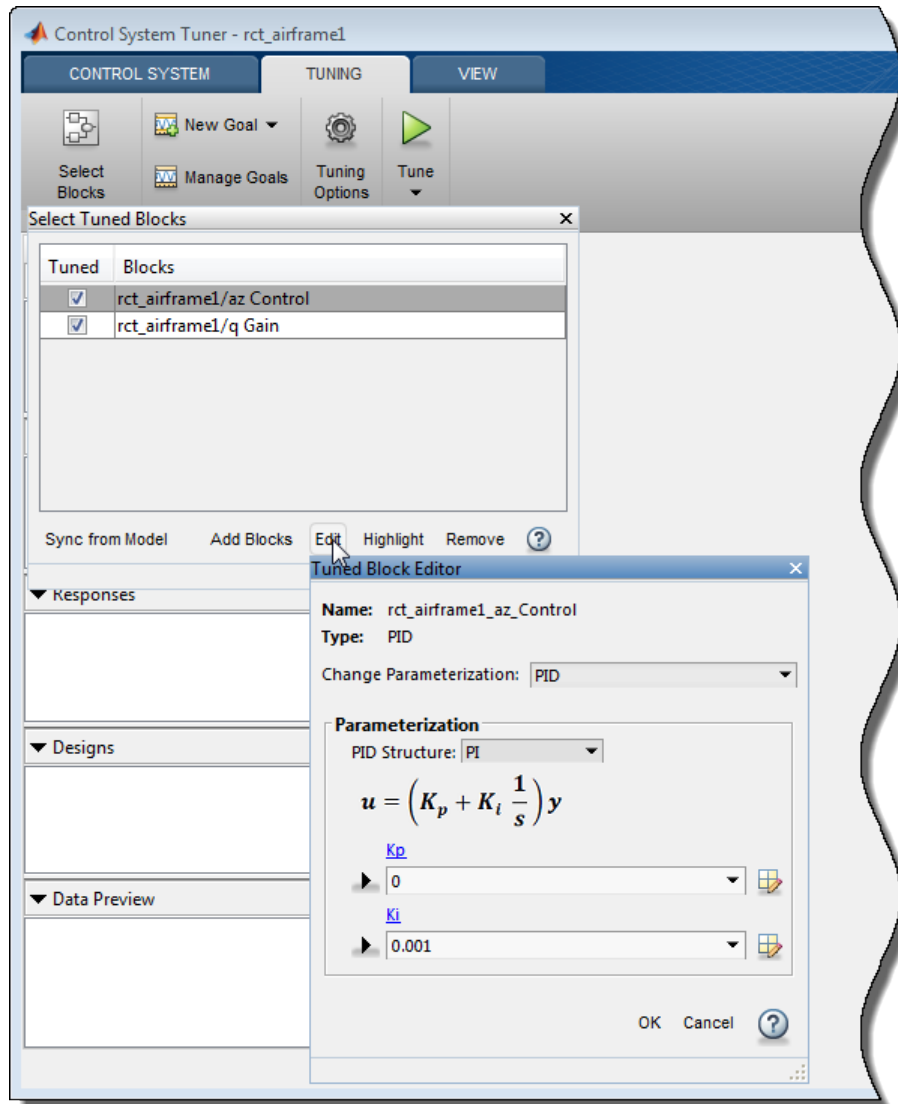
4   Use the fields of the **Tuned Block Editor** dialog box to edit the parameterization. See "Tuned Block Editor" on page 6-36 for more detailed information about these fields.

5   When you are finished editing, click **OK** to update the parameterization.

See Also

"Tuned Block Editor" on page 6-36

More About

·   "How Tuned Simulink Blocks Are Parameterized" on page 6-42

Tuned Block Editor

The **Tuned Block Editor** dialog box lets you view and change the parameterization of a tunable block. The dialog box contains the following fields:

- **Name** — Name of the tunable block.
- **Type** — Type of parameterization.
- **Change Parameterization** — Specify the type of parameterization. The available types are:

  - `PID` — SISO PID parameterization. Parameters are the PID gains and filter constant.
  - `2-DOF PID` — Two-degree-of-freedom PID parameterization. Available only when the tuned block has two inputs and one output.
  - `State Space` — State-space model. Parameters are entries in the state-space matrices *A*, *B*, *C*, and *D*.
  - `Transfer Function` — SISO transfer function. Parameters are the numerator and denominator coefficients.
  - `Gain` — Scalar static gain or static gain matrix.
  - `Custom` — Custom parameterization, specified as a tunable generalized state-space (`genss`) model. See "Custom Parameterization" on page 6-39.

    This option is not available when you specify your control architecture as a generalized feedback configuration as described in "Arbitrary Feedback Control Architecture" on page 6-19.

- **PID Structure** — For a block of **Type** PID, select a configuration from the **Structure** drop-down menu, such as `P` (proportional only), `PI` (proportional and integral), or `PID` (proportional, integral, and derivative).

- **Order of the system** — For a block of **Type** State Space, specify the order (number of states) of the parameterization. When you change this value, current values of the parameters are automatically updated to be consistent with the order. For example, if you enter 3, the current value of the parameter `a` is changed to a 3-by-3 array.

- **Number of poles**, **Number of zeros** — For a block of **Type** Transfer Function, specify the number of poles and number of zeros in the parameterization. When you change these value, current values of the parameters are automatically updated to be consistent with the number of poles or zeros. For example, if you specify 3 poles, the

current value of the parameter `den` is changed to an array of length 4, representing a third-order polynomial.

- **Parameterization** — Current values of the parameters. There is one parameter field for each tunable parameter in the block parameterization. For example, for a PID Controller block configured for PI structure, the **Tuned Block Editor** dialog box contains parameter fields for K_p and K_i. The text boxes display the current value of each parameter.
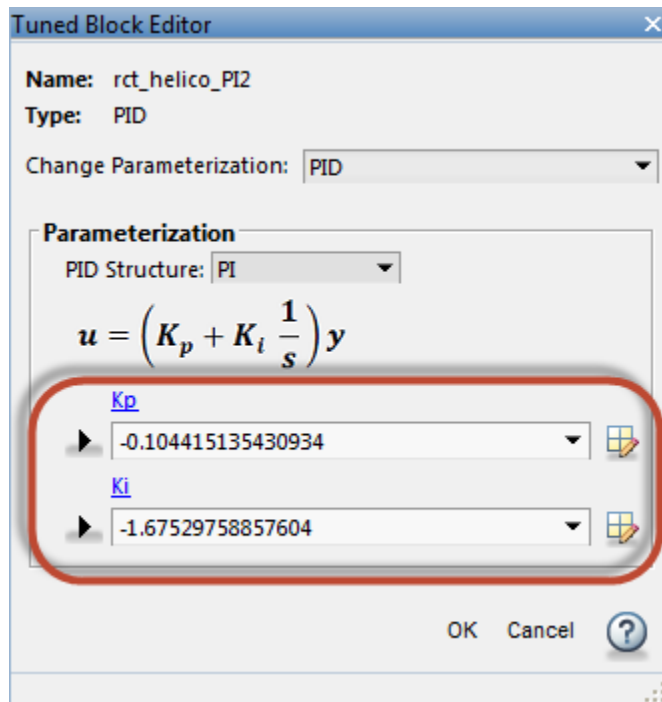
Edit Parameter Values and Properties

You can change the current value of a parameter, fix its current value (make the parameter nontunable), or limit the parameter's tuning range.

Type in a text boxes to change a current parameter value. Alternatively, click to use a variable editor to change the current value. If you attempt to enter an invalid value, the parameter returns to its previous value.

Click ▶ to access and edit additional properties of each parameter.
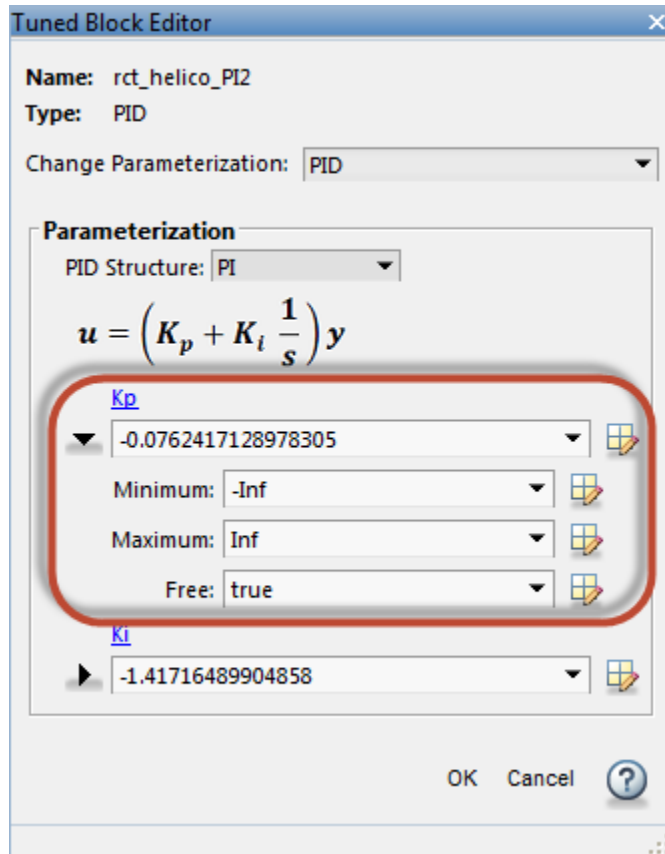
- **Minimum** — Minimum value that the parameter can take when the control system is tuned.
- **Maximum** — Maximum value that the parameter can take when the control system is tuned.
- **Free** — When the value is `true`, Control System Toolbox tunes the parameter. To fix the value of the parameter, set **Free** to `false`.

For array-valued parameters, you can set these properties independently for each entry in the array. For example, for a vector-valued gain of length 3, enter `[1 10 100]` to set

the current value of the three gains to 1, 10, and 100 respectively. Alternatively, click ![icon] to use a variable editor to specify such values.

For vector or matrix-valued parameters, you can use the **Free** parameter to constrain the structure of the parameter. For example, to restrict a matrix-valued parameter to be a diagonal matrix, set the current values of the off-diagonal elements to 0, and set the corresponding entries in **Free** to `false`.

Custom Parameterization

When tuning a control system represented by a Simulink model or by a "Predefined Feedback Architecture" on page 6-18, you can specify a custom parameterization for any tuned block using a generalized state-space (`genss`) model. To do so, create and configure a `genss` model in the MATLAB workspace that has the desired parameterization, initial parameter values, and parameter properties. In the **Change parameterization** dialog box, select `Custom`. In the **Parameterization** area, the variable name of the `genss` model.

For example, suppose you want to specify a tunable low-pass filter, $F = a/(s + a)$, where a is the tunable parameter. First, at the MATLAB command line, create a tunable `genss` model that represents the low-pass filter structure.

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


F =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs,
  1 states, and the following blocks:
    a: Scalar parameter, 2 occurrences.

Type "ss(F)" to see the current value, "get(F)" to see all properties, and
"F.Blocks" to interact with the blocks.
```

Then, in the Tuned Block Editor, enter `F` in the **Parameterization** area.
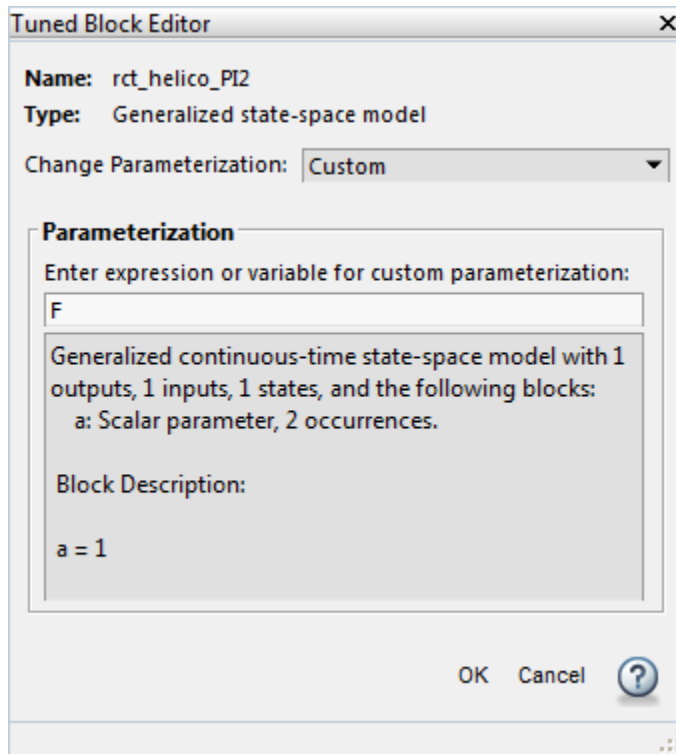
When you specify a custom parameterization for a Simulink block, you might not be able to write the tuned block value back to the Simulink model. When writing values to Simulink blocks, Control System Tuner skips blocks that cannot represent the tuned value in a straightforward and lossless manner. For example, if you reparameterize a PID Controller Simulink block as a third-order state-space model, Control System Tuner will not write the tuned value back to the block.

Related Examples

- "View and Change Block Parameterization in Control System Tuner" on page 6-33

More About

- "How Tuned Simulink Blocks Are Parameterized" on page 6-42

Setup for Tuning Control System Modeled in MATLAB

To model your control architecture in MATLAB for tuning in Control System Tuner, construct a tunable model of the control system that identifies and parameterizes its tunable elements. You do so by combining numeric LTI models of the fixed elements with parametric models of the tunable elements. The result is a tunable generalized state-space `genss` model.

Building a tunable `genss` model for Control System Tuner is the same as building such a model for tuning at the command line. For information about building such models, "Setup for Tuning MATLAB Models".

When you have a tunable `genss` model of your control system, use the `controlSystemTuner` command to open Control System Tuner. For example, if `T0` is the `genss` model, the following command opens Control System Tuner for tuning `T0`:

```
controlSystemTuner(T0)
```

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44

How Tuned Simulink Blocks Are Parameterized

Blocks With Predefined Parameterization

When you tune a Simulink model, either with Control System Tuner or at the command line through an `slTuner` interface, the software automatically assigns a predefined parameterization to certain Simulink blocks. For example, for a PID Controller block set to the PI controller type, the software automatically assigns the parameterization $K_p + K_i/s$, where K_p and K_i are the tunable parameters. For blocks that have a predefined parameterization, you can write tuned values back to the Simulink model for validating the tuned controller.

Blocks that have a predefined parameterization include the following:

Simulink Library	Blocks with Predefined Parameterization
Math Operations	Gain
Continuous	• State-Space • Transfer Fcn • Zero-Pole • PID Controller • PID Controller (2 DOF)
Discrete	• Discrete State-Space • Discrete Transfer Fcn • Discrete Zero-Pole • Discrete Filter • Discrete PID Controller • Discrete PID Controller (2 DOF)
Control System Toolbox	LTI System
Discretizing (Model Discretizer Blocks)	• Discretized State-Space • Discretized Transfer Fcn • Discretized Zero-Pole • Discretized LTI System

Simulink Library	Blocks with Predefined Parameterization
	• Discretized Transfer Fcn (with initial states)
Simulink Extras/Additional Linear	State-Space (with initial outputs)

Blocks Without Predefined Parameterization

You can specify blocks for tuning that do not have a predefined parameterization. When you do so, the software assigns a state-space parameterization to such blocks based upon the block linearization. For blocks that do not have a predefined parameterization, the software cannot write tuned values back to the block, because there is no clear mapping between the tuned parameters and the block. To validate a tuned control system that contains such blocks, you can specify a block linearization in your model using the value of the tuned parameterization. (See "Specify Linear System for Block Linearization Using MATLAB Expression" for more information about specifying block linearization.)

View and Change Block Parameterization

You can view and edit the current parameterization of every block you designate for tuning.

- In Control System Tuner, see "View and Change Block Parameterization in Control System Tuner" on page 6-33.
- At the command line, use `getBlockParam` to view the current block parameterization. Use `setBlockParam` to change the block parameterization.

Specify Goals for Interactive Tuning

This example shows how to specify your tuning goals for automated tuning in Control System Tuner.

Use the **New Goal** menu to create a tuning goal such as a tracking requirement, disturbance rejection specification, or minimum stability margins. Then, when you are ready to tune your control system, use **Manage Goals** to designate which goals to enforce.

This example creates tuning goals for tuning the sample model `rct_helico`.

Choose Tuning Goal Type

In Control System Tuner, in the **Tuning** tab, click **New Goal**. Select the type of goal you want to create. A tuning goal dialog box opens in which you can provide the detailed specifications of your goal. For example, select **Tracking of step commands** to make a particular step response of your control system match a desired response.

Step Tracking Goal

Name: StepTrackingGoal1

Purpose

Make specific closed-loop step response closely match the desired response.

Step Response Selection

Specify step-response inputs:

+ Add signal to list

Specify step-response outputs:

+ Add signal to list

Compute step response with the following loops open:

+ Add loop opening location to list

Desired Response

Specify as
- First-order characteristics
- Second-order characteristics
- Custom reference model

Time constant: 1

Options

Keep % mismatch below: 10

Adjust for step amplitude: No

Apply goal to:
- All models
- Only models: [1 2]

OK Apply Cancel ?

Choose Signal Locations for Evaluating Tuning Goal

Specify the signal locations in your control system at which the tuning goal is evaluated. For example, the step response goal specifies that a step signal applied at a particular input location yields a desired response at a particular output location. Use the **Step**

Response Selection section of the dialog box to specify these input and output locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)

Under **Specify step-response inputs**, click ➕ **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.
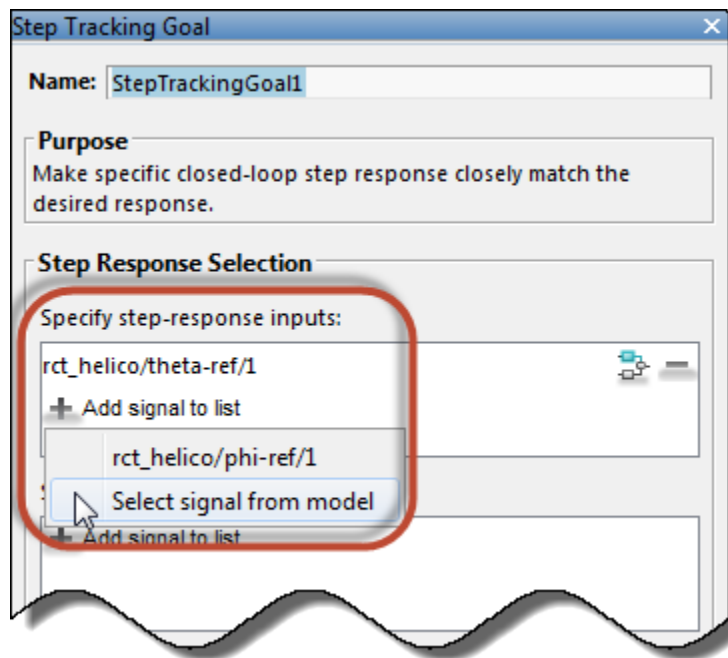
The **Add signal from the model** dialog box contains a list of signals that are currently selected in the Simulink model. If the signal you want to designate is listed, select it and click **Add signal**.

If the signal you want is not listed, select the signal in the Simulink model editor. Shift-click to select multiple signals at once. Then, return to the **Add signal from the model** dialog box and click **Add signal**.
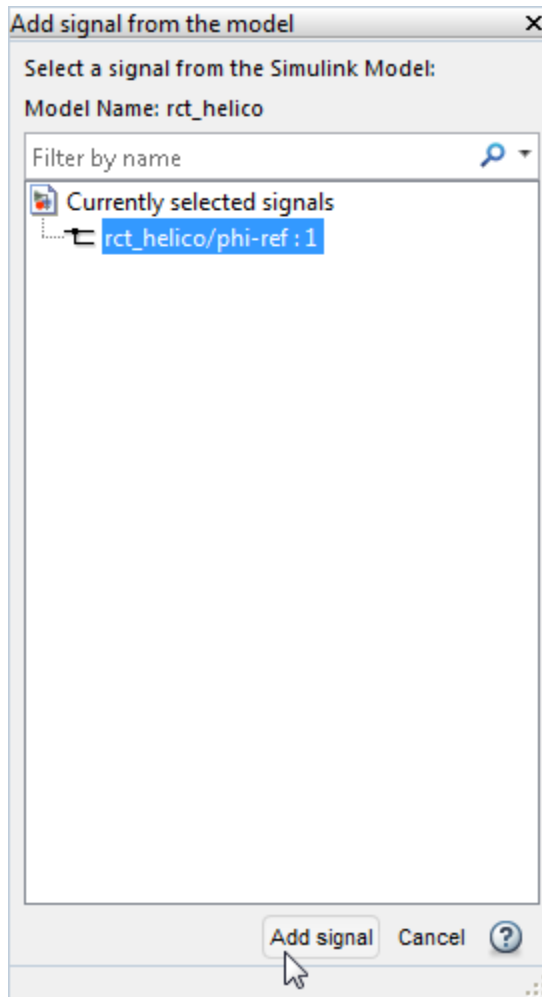
The signal you selected now appears in the list of step-response inputs.

Click ✚ **Add signal to list** again to add an additional signal to the step-response inputs list, if you want to specify a MIMO tuning goal.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration constrains the response

to a step input applied at `theta-ref` and measured at `theta` in the Simulink model `rct_helico`.
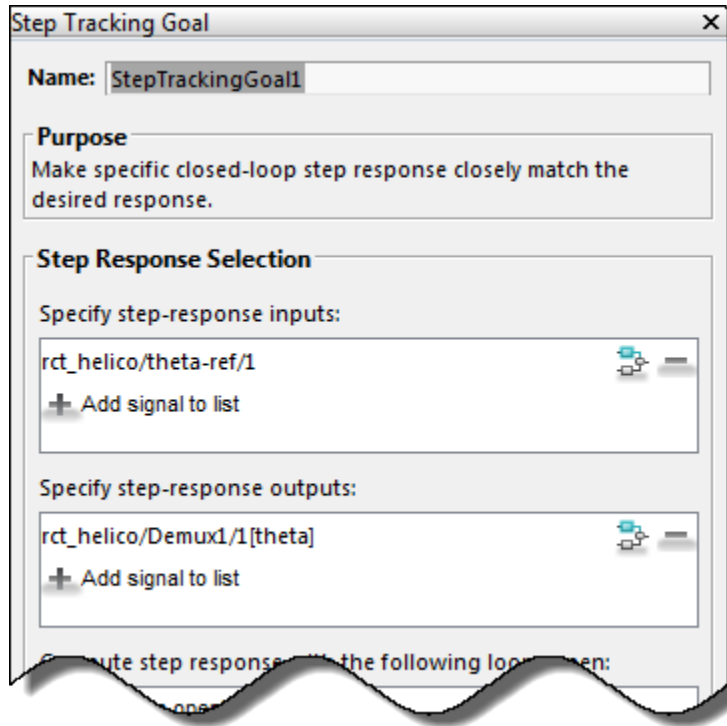
Tip To highlight any selected signal in the Simulink model, click 📑. To remove a signal from the input or output list, click ▬ .

Specify Loop Openings

Most tuning goals can be enforced with loops open at one or more locations in the control system. Click ➕**Add loop opening location to list** to specify such locations for the tuning goal.

Define Other Specifications of the Tuning Goal

The tuning goal dialog box prompts you to specify other details about the tuning goal. For example, to create a step response requirement, you provide details of the desired step response in the **Desired Response** area of the **Step Response Goal** dialog box. Some tuning goals have additional options in an **Options** section of the dialog box.

For information about the fields for specifying a particular tuning goal, click in the tuning goal dialog box.

Store the Tuning Goal for Tuning

When you have finished specifying the tuning goal, click **OK** in the tuning goal dialog box. The new tuning goal appears in the **Tuning Goals** section of the Data Browser. A new figure opens displaying a graphical representation of the tuning goal. When you tune your control system, you can refer to this figure to evaluate graphically how closely the tuned system satisfies the tuning goal.
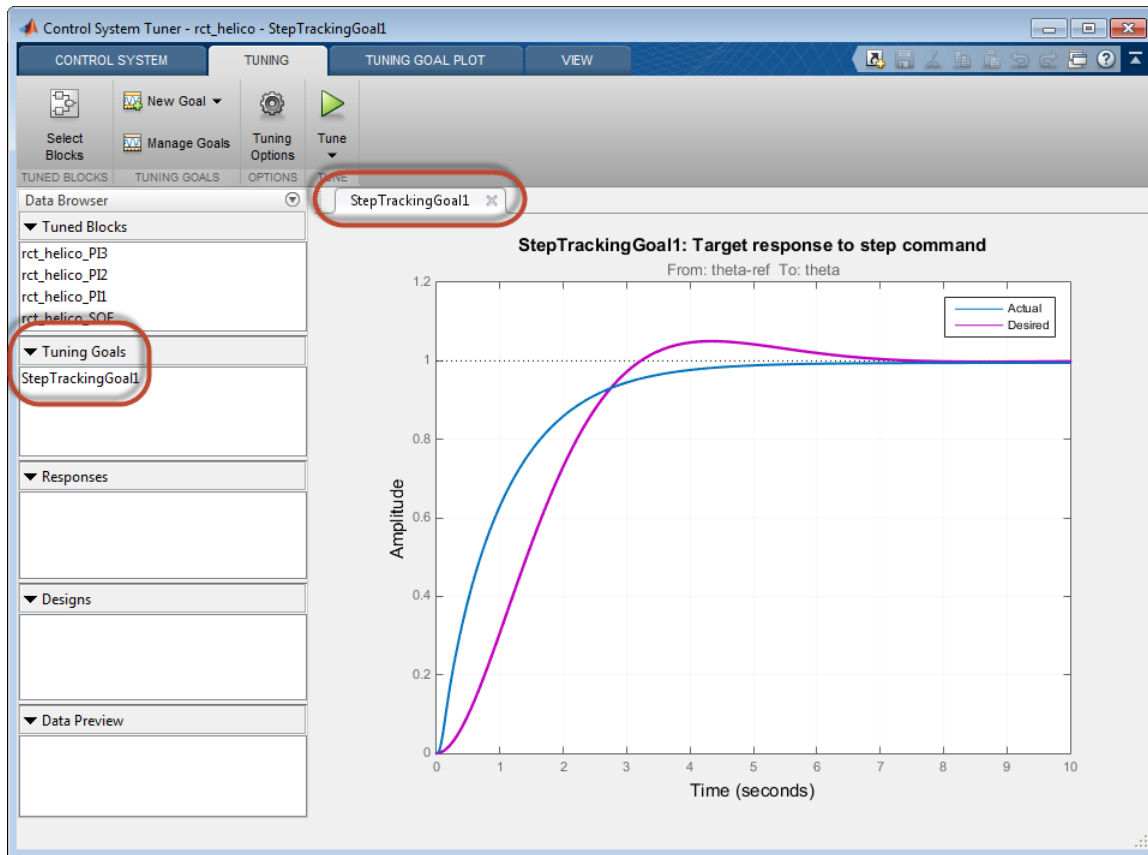
Tip To edit the specifications of the tuning goal, double-click the tuning goal in the Data Browser.

Activate the Tuning Goal for Tuning

When you have saved your tuning goal, click ![icon] **New Goal** to create additional tuning goals.

When you are ready to tune your control system, click ![icon] **Manage Goals** to select which tuning goals are active for tuning. In the **Manage Tuning Goals** dialog box,

Active is checked by default for any new goals. Uncheck **Active** for any tuning goal that you do not want enforced.

You can also designate one or more tuning goals as **Hard** goals. Control System Tuner attempts to satisfy hard requirements, and comes as close as possible to satisfying remaining (soft) requirements subject to the hard constraints. By default, new goals are designated soft goals. Check **Hard** for any goal to designate it a hard goal.

For example, if you tune with the following configuration, Control System Tuner optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.
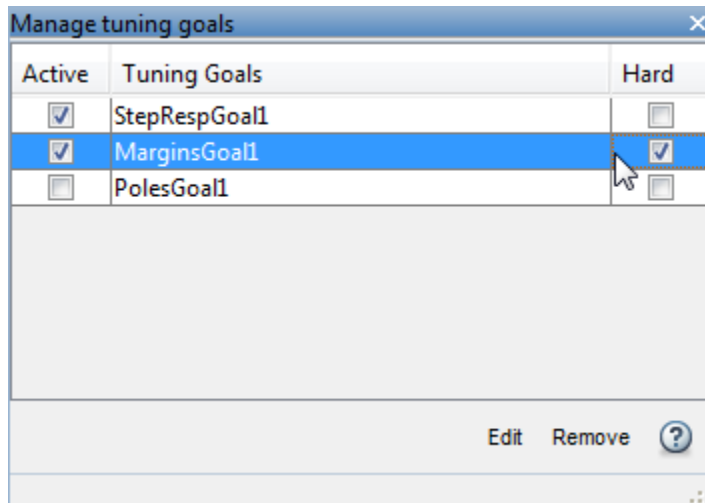
Deactivating tuning goals or designating some goals as soft requirements can be useful when investigating the tradeoffs between different tuning requirements. For example, if you do not obtain satisfactory performance with all your tuning goals active and hard, you might try another design in which less crucial goals are designated as soft or deactivated entirely.

Related Examples

- "Manage Tuning Goals" on page 6-152
- "Quick Loop Tuning of Feedback Loops in Control System Tuner" on page 6-52
- "Create Response Plots in Control System Tuner" on page 6-163

Quick Loop Tuning of Feedback Loops in Control System Tuner

This example shows how to tune a Simulink model of a control system to meet a specified bandwidth and stability margins in Control System Tuner, without explicitly creating tuning goals that capture these requirements. You can use a similar approach for quick loop tuning of control systems modeled in MATLAB.

This example demonstrates how the **Quick Loop Tuning** option of Control System Tuner generates tuning goals from your crossover frequency and gain and phase margin specifications. This option lets you quickly set up SISO or MIMO feedback loops for tuning using a loop-shaping approach. The example also shows how to add further tuning requirements to the control system after using the **Quick Loop Tuning** option.

Quick Loop Tuning is the Control System Tuner equivalent of the `looptune` command.

Set up the Model for Tuning

Open the Simulink model.

```
open_system('rct_distillation')
```
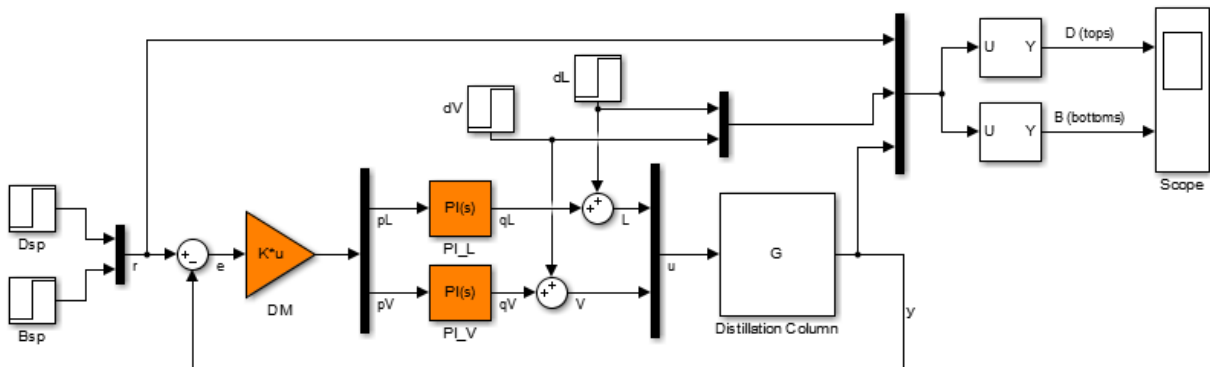

This model represents a distillation column, captured in the two-input, two-output plant `G`. The tunable elements are the decoupling gain matrix `DM`, and the two PI controllers, `PI_L` and `PI_V`. (For more information about this model, see "Decoupling Controller for a Distillation Column".)

Suppose your goal is to tune the MIMO feedback loop between r and y to a bandwidth between 0.1 and 0.5 rad/s. Suppose you also require a gain margin of 7 dB and a phase margin of 45 degrees. You can use the **Quick Loop Tuning** option to quickly configure Control System Tuner for these goals.

In the Simulink model editor, open Control System Tuner by selecting **Analysis > Control Design > Control System Tuner**.

Designate the blocks you want to tune. In the **Tuning** tab of Control System Tuner, click ⊞ **Select Blocks**. In the **Select tuned blocks** dialog box, click **Add blocks**. Then, select DM, PI_L, and PI_V for tuning. (For more information about selecting tuned blocks, see "Specify Blocks to Tune in Control System Tuner" on page 6-31.)

The model is now ready to tune to the target bandwidth and stability margins.

Specify the Goals for Quick Loop Tuning

In the **Tuning** tab, select **New Goal > Quick Loop Tuning**.
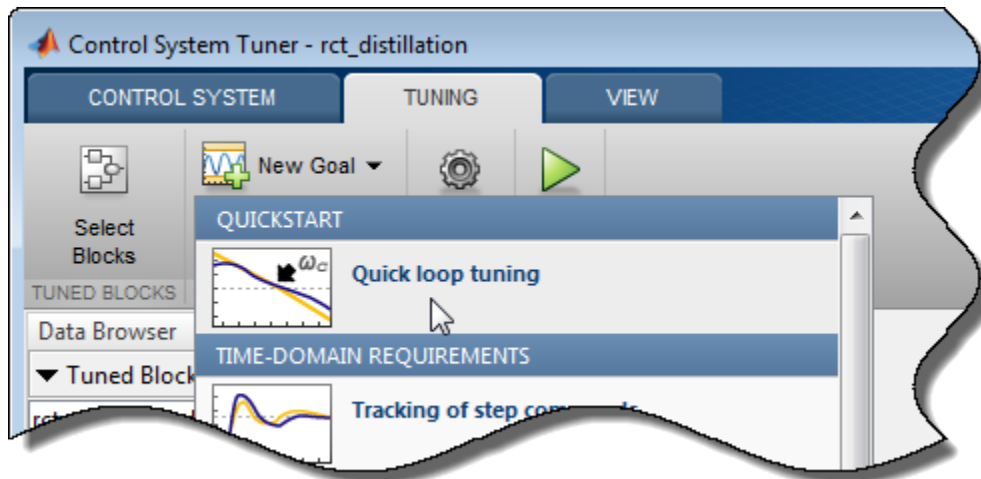
For Quick Loop Tuning, you need to identify the actuator signals and sensor signals that separate the plant portion of the control system from the controller, which for the purpose of Quick Loop Tuning is the rest of the control system. The actuator signals are the controller outputs that drive the plant, or the plant inputs. The sensor signals are the measurements of plant output that feed back into the controller. In this control system,

the actuator signals are represented by the vector signal u, and the sensor signals by the vector signal y.

In the **Quick Loop Tuning** dialog box, under **Specify actuator signals (controls)**, add the actuator signal, u. Similarly, under **Specify sensor signals (measurements)**, add the sensor signal, y (For more information about specifying signals for tuning, see "Specify Goals for Interactive Tuning" on page 6-44.)

Under **Desired Goals**, in the **Target gain crossover region** field, enter the target bandwidth range, [0.1 0.5]. Enter the desired gain margin and phase margin in the corresponding fields.

Quick Loop Tuning

Name: LoopTuning1

Purpose

Tune SISO or MIMO feedback loops using a loop shaping approach.

Feedback Loop Selection

Specify actuator signals (controls):

rct_distillation/Mux1/1[u]

+ Add signal to list

Specify sensor signals (measurements):

rct_distillation/Distillation Column/1[y]

+ Add signal to list

Compute the response with the following loops open:

+ Add loop opening location to list

Desired Goals

Target gain crossover region:	[0.1 0.5]
Gain margin (dB):	7
Phase margin (degrees):	45

Keep poles inside the following region:

| Minimum decay rate: | 0 |
| Maximum natural frequency: | Inf |

Options

Apply goal to

○ All models

○ Only models: [1 2]

OK   Cancel   ⑦

Click **OK**. Control System Tuner automatically generates tuning goals that capture the desired goals you entered in the dialog box.
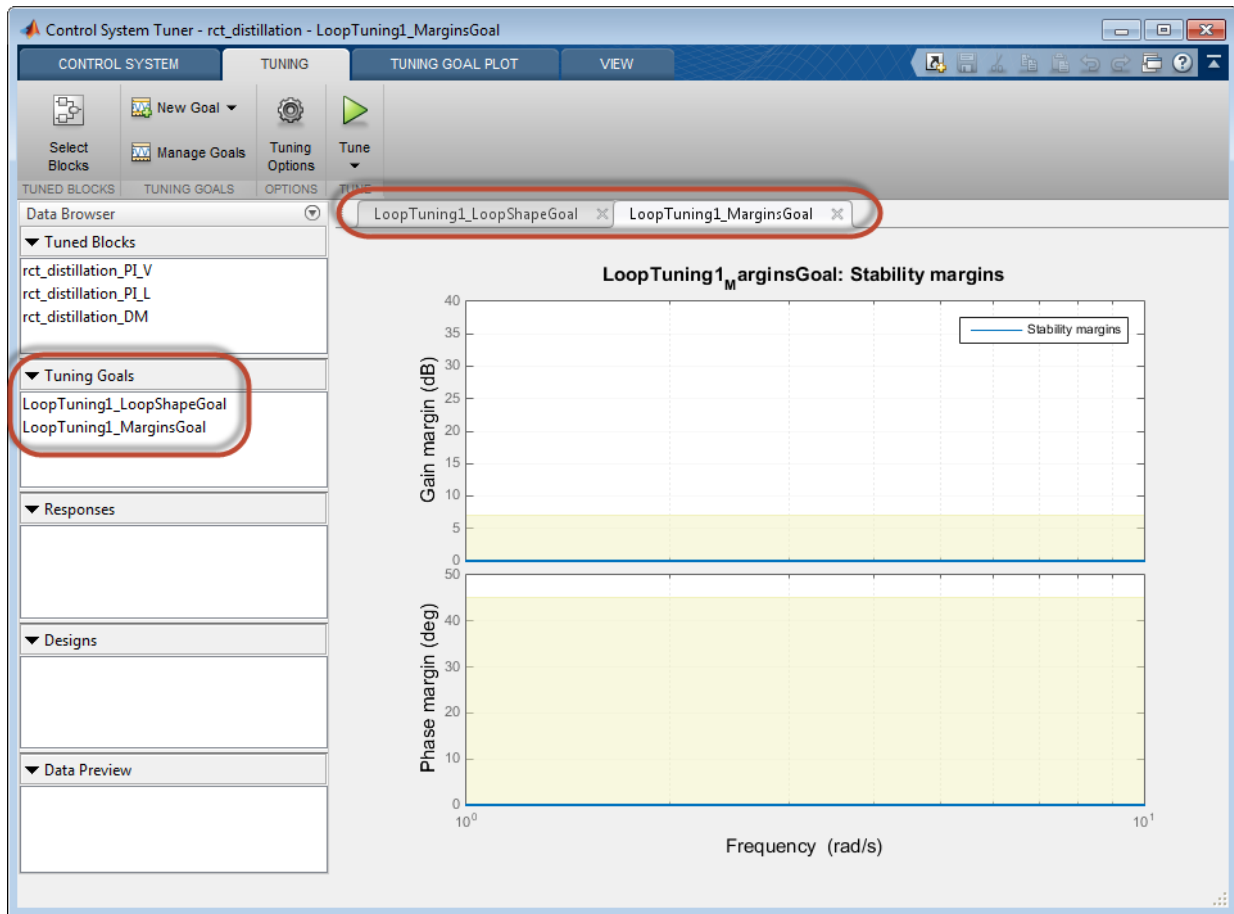
Examine the Automatically-Created Tuning Goals

In this example, Control System Tuner creates a Loop Shape Goal and a Margins Goal. If you had changed the pole-location settings in the **Quick Loop Tuning** dialog box, a Poles goal would also have been created.

Click [icon] **Manage Goals** to examine the automatically-created goals. By default, the goals are active and designated as soft tuning goals.
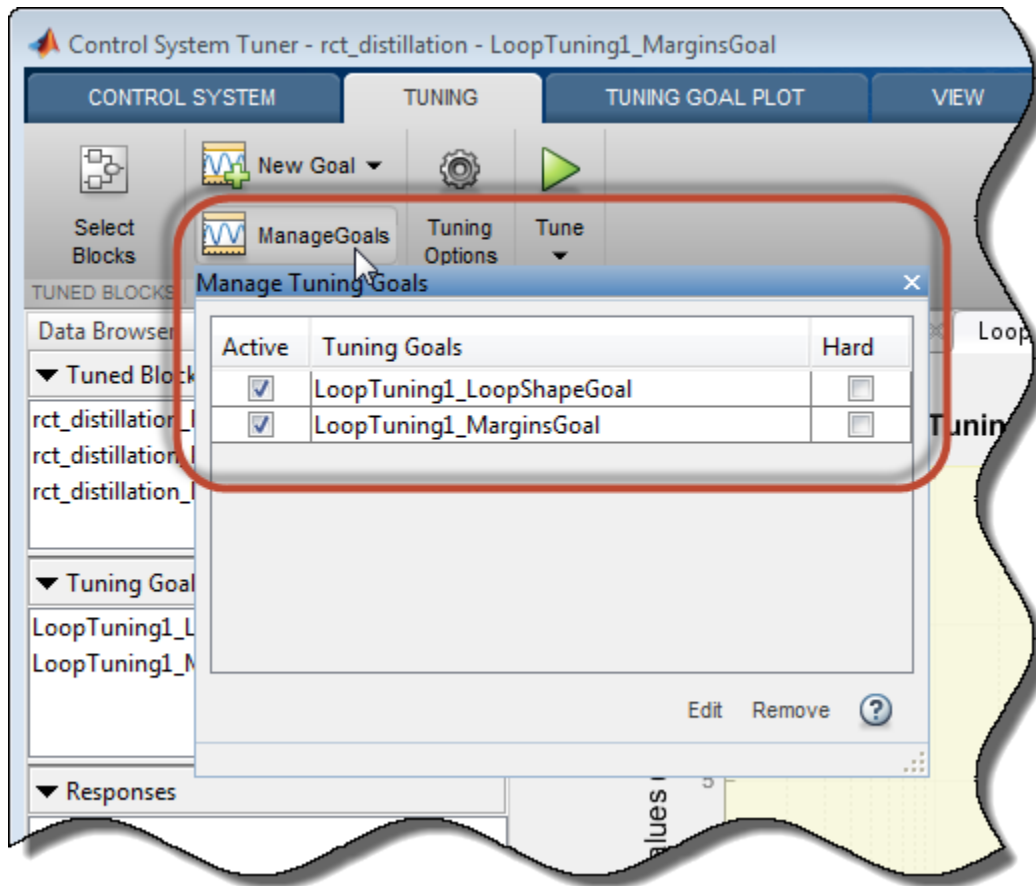
You can double-click the tuning goals to examine their parameters, which are automatically computed and populated. You can also examine the graphical representations of the tuning goals. In the **Tuning** tab, examine the **LoopTuning1_LoopShapeGoal** plot.
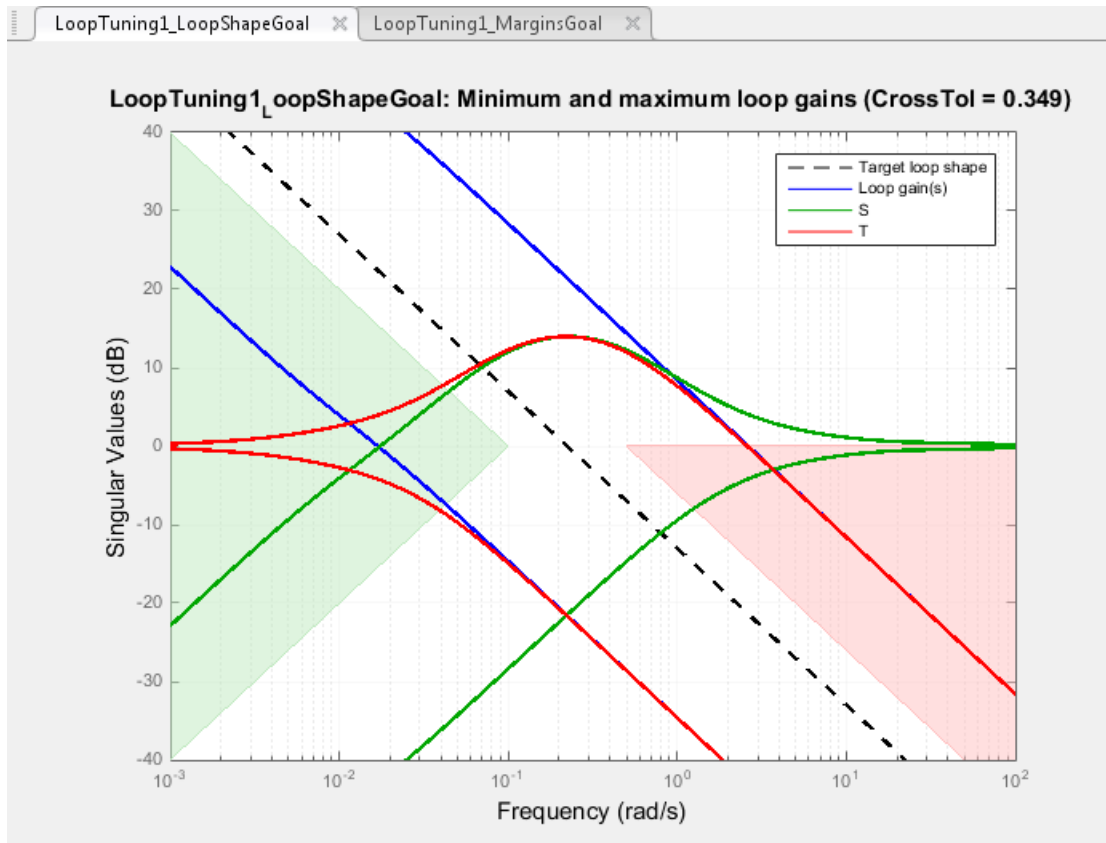
The target crossover range is expressed as a Loop Shape goal with an integrator open-loop gain profile. The shaded areas of the graph show that the permitted crossover range is [0.1 0.5] rad/s, as you specified in the **Quick Loop Tuning** dialog box.

Similarly, your margin requirements are captured in the **LoopTuning1_MarginsGoal** plot.

Tune the Model

Click ![tune icon] **Tune** to tune the model to meet the automatically-created tuning goals. In the tuning goal plots, you can see that the requirements are satisfied.

To create additional plots for examining other system responses, see "Create Response Plots in Control System Tuner" on page 6-163.

Change Design Requirements

If you want to change your design requirements after using Quick Loop Tuning, you can edit the automatically-created tuning goals and tune the model again. You can also create additional tuning goals.

For example, add a requirement that the outputs, y, track the reference inputs, r, with response time of 15 s and tracking error of no more than 0.1%. Select **New Goal** > **Reference Tracking** and enter appropriate values in the Reference Tracking Goal dialog box. (For more information about creating tuning goals, see "Specify Goals for Interactive Tuning" on page 6-44.)

Reference Tracking Goal ✕

Name: TrackingGoal1

Purpose
Follow reference commands with prescribed performance and fidelity. Limit cross-coupling in MIMO systems.

Response Selection

Specify reference inputs:

rct_distillation/Mux/1[r]

➕ Add signal to list

Specify reference-tracking outputs:

rct_distillation/Distillation Column/1[y]

➕ Add signal to list

Evaluate tracking performance with the following loops open:

➕ Add loop opening location to list

Tracking Performance

Specify as
◉ Response time, DC error, and peak error
◯ Maximum error as a function of frequency

Response Time:	15
Steady-state error (%):	0.1
Peak error across frequency (%):	100

Options

Enforce goal in frequency range	[0 Inf]
Adjust for signal amplitude	No ▼

Apply goal to
◉ All models
◯ Only models: [1 2]

OK   Apply   Cancel   ⓘ

You can now retune the model to meet all these tuning goals.

See Also
`looptune (for slTuner)`

Related Examples
- "Specify Operating Points for Tuning in Control System Tuner" on page 6-24
- "Manage Tuning Goals" on page 6-152
- "Setup for Tuning Control System Modeled in MATLAB" on page 6-41

Quick Loop Tuning

Purpose

Tune SISO or MIMO feedback loops using a loop-shaping approach.

Description

Quick Loop Tuning lets you tune your system to meet open-loop gain crossover and stability margin requirements without explicitly creating tuning goals that capture these requirements. You specify the feedback loop whose open-loop gain you want to shape by designating the actuator signals (controls) and sensor signals (measurements) that form the loop. Actuator signals are the signals that drive the plant. The sensor signals are the plant outputs that feed back into the controller.

You enter the target loop bandwidth and desired gain and phase margins. You can also specify constraints on pole locations of the tuned system, to eliminate fast dynamics. Control System Tuner automatically creates Tuning Goals that capture your specifications and ensure integral action at frequencies below the target loop bandwidth.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Quick Loop Tuning** to specify loop-shaping requirements.

Command-Line Equivalent

When tuning control systems at the command line, use `looptune (for slTuner)` or `looptune` for tuning feedback loops using a loop-shaping approach.

Feedback Loop Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify actuator signals (controls)**

  Designate one or more signals in your model as actuator signals. These are the input signals that drive the plant. To tune a SISO feedback loop, select a single-valued input signal. To tune MIMO loop, select multiple signals or a vector-valued signal.

- **Specify sensor signals (measurements)**

Designate one or more signals in your model as sensor signals. These are the plant outputs that provide feedback into the controller. To tune a SISO feedback loop, select a single-valued input signal. To tune MIMO loop, select multiple signals or a vector-valued signal.

- **Compute the response with the following loops open**

  Designate additional locations at which to open feedback loops for the purpose of tuning the loop defined by the control and measurement signals.

  Quick Loop Tuning tunes the open-loop response of the loop defined by the control and measurement signals. If you want your specifications for that loop to apply with other feedback loops in the system opened, specify loop-opening locations in this section of the dialog box. For example, if you are tuning a cascaded-loop control system with an inner loop and an outer loop, you might want to tune the inner loop with the outer loop open.

For an example showing in more detail how to specify signal locations, see "Specify Goals for Interactive Tuning" on page 6-44.

Desired Goals

Use this section of the dialog box to specify desired characteristics of the tuned system. Control System Tuner converts these into Loop Shape, Margin, and Poles goals.

- **Target gain crossover region**

  Specify a frequency range in which the open-loop gain should cross 0 dB. Specify the frequency range as a row vector of the form `[min,max]`, expressed in frequency units of your model. Alternatively, if you specify a single target frequency, `wc`, the target range is taken as `[wc/10^0.1,wc*10^0.1]`, or `wc` ± 0.1 decade.

- **Gain margin (db)**

  Specify the desired gain margin in decibels. For MIMO control system, the gain margin is the multiloop disk margin. See `loopmargin` for information about multiloop disk margins.

- **Phase margin (degrees)**

  Specify the desired phase margin in degrees. For MIMO control system, the phase margin is the multiloop disk margin. See `loopmargin` for information about multiloop disk margins.

- **Keep poles inside the following region**

  Specify minimum decay rate and maximum natural frequency for the closed-loop poles of the tuned system. While the other Quick Loop Tuning options specify characteristics of the open-loop response, these specifications apply to the closed-loop dynamics.

  The minimum decay rate you enter constrains the closed-loop pole locations to:

  - `Re(s) < -mindecay`, for continuous-time systems.
  - `log(|z|) < -mindecay*Ts`, for discrete-time systems with sample time `Ts`.

  The maximum frequency you enter constrains the closed-loop poles to satisfy `|s| < maxfreq` for continuous time, or `|log(z)| < maxfreq*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the closed-loop system.

Options

Use this section of the dialog box to specify additional characteristics.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

Control System Tuner uses `looptuneSetup (for slTuner)` or `looptuneSetup` to convert Quick Loop Tuning specifications into tuning goals.

Related Examples

- "Quick Loop Tuning of Feedback Loops in Control System Tuner" on page 6-52

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Step Tracking Goal

Purpose

Make the step response from specified inputs to specified outputs closely match a target response.

Description

Step Tracking Goal constrains the step response between the specified signal locations to match the step response of a stable reference system. The constraint is satisfied when the relative difference between the tuned and target responses falls within the tolerance you specify. You can use this goal to constrain a SISO or MIMO response of your control system.

You can specify the reference system for the target step response in terms of first-order system characteristics (time constant) or second-order system characteristics (natural frequency and percent overshoot). Alternatively, you can specify a custom reference system as a numeric LTI model.
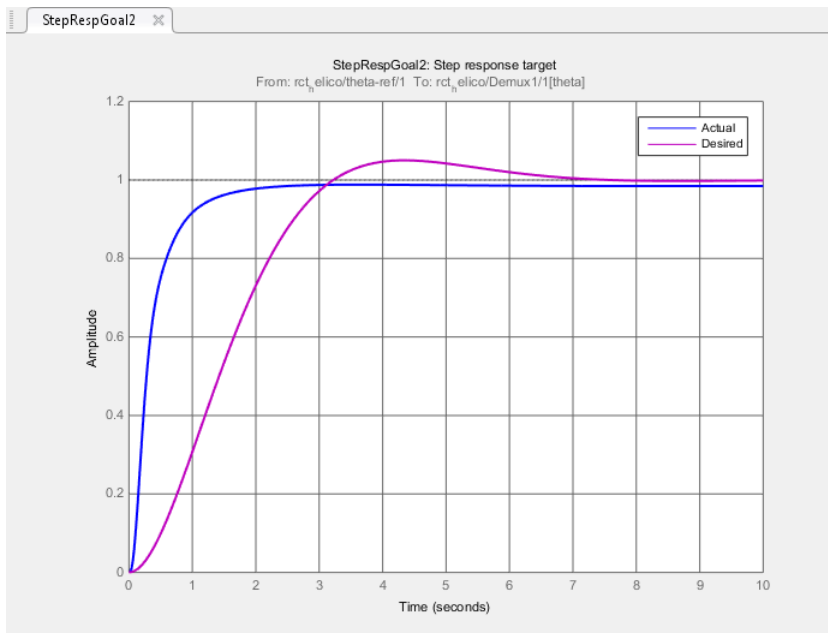
Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Tracking of step commands** to create a Step Tracking Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.StepTracking` to specify a step response goal.

Step Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step-response inputs**

  Select one or more signal locations in your model at which to apply the step input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click

  ✚ **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step-response outputs**

  Select one or more signal locations in your model at which to measure the response to the step input. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location

  named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Compute step response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location

  named `'x'`, ✚ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Desired Response

Use this section of the dialog box to specify the shape of the desired step response.

- **First-order characteristics**

  Specify the desired step response (the reference model H_{ref}) as a first-order response with time constant τ:

  $$H_{ref} = \frac{1/\tau}{s + 1/\tau}.$$

  Enter the desired value for τ in the **Time Constant** text box. Specify τ in the time units of your model.

- **Second-order characteristics**

  Specify the desired step response as a second-order response with time constant τ, and natural frequency $1/\tau$.

  Enter the desired value for τ in the **Time Constant** text box. Specify τ in the time units of your model.

  Enter the target overshoot percentage in the **Overshoot** text box.

  The second-order reference system has the form:

  $$H_{ref} = \frac{\left(1/\tau\right)^2}{s^2 + 2\left(\zeta/\tau\right)s + \left(1/\tau\right)^2}.$$

  The damping constant ζ is related to the overshoot percentage by $\zeta =$ `cos(atan2(pi,-log(overshoot/100)))`.

- **Custom reference model**

  Specify the reference system for the desired step response as a dynamic system model, such as a `tf`, `zpk`, or `ss` model.

  Enter the name of the reference model in the MATLAB workspace in the **LTI model to match** text field. Alternatively, enter a command to create a suitable reference model, such as `tf(1,[1 1.414 1])`.

The reference model must be stable and must have DC gain of 1 (zero steady-state error). The model can be continuous or discrete. If the model is discrete, it can include time delays which are treated as poles at `z` = 0.

The reference model can be MIMO, provided that it is square and that its DC singular value (`sigma`) is 1. Then number of inputs and outputs of the reference model must match the dimensions of the inputs and outputs specified for the step response goal.

For best results, the reference model should also include intrinsic system characteristics such as non-minimum-phase zeros (undershoot).

If your selected inputs and outputs define a MIMO system and you apply a SISO reference system, the software attempts to match the diagonal channels of the MIMO system. In that case, cross-couplings tend to be minimized.

Options

Use this section of the dialog box to specify additional characteristics of the step response goal.

- **Keep % mismatch below**

  Specify the relative matching error between the actual (tuned) step response and the target step response. Increase this value to loosen the matching tolerance. The relative matching error, e_{rel}, is defined as:

  $$e_{rel} = \frac{\left\| y(t) - y_{ref}(t) \right\|_2}{\left\| 1 - y_{ref}(t) \right\|_2}.$$

  $y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref}(t)$ is the step-tracking error of the target model. $\left\| \cdot \right\|_2$ denotes the signal energy (2-norm).

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to

tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

For example, suppose that tuning goal is that outputs `'y1'` and `'y2'` track reference signals `'r1'` and `'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. This tells Control System Tuner to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, `No`, means no scaling is applied.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Step Response Goal**, $f(x)$ is given by:

$$f(x) = \frac{\left\| T(s,x) - \frac{1}{s} H_{ref}(s) \right\|_2}{e_{rel} \left\| \frac{1}{s} \left(H_{ref}(s) - I \right) \right\|_2}.$$

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $H_{ref}(s)$ is the reference model. e_{rel} is the relative error (see "Options" on page 6-69). $\left\| \cdot \right\|_2$ denotes the H_2 norm (see `norm`).

Related Examples

Step Rejection Goal

Purpose

Set a minimum standard for rejecting step disturbances.

Description

Use **Step Rejection Goal** to specify how a step disturbance injected at a specified location in your control system affects the signal at a specified output location.

You can specify the desired response in time-domain terms of peak value, settling time, and damping ratio. Control System Tuner attempts to make the actual rejection at least as good as the desired response. Alternatively, you can specify the response as a stable reference model having DC-gain. In that case, the tuning goal is to reject the disturbance as well as or better than the reference model.

To specify disturbance rejection in terms of a frequency-domain attenuation profile, use **Disturbance Rejection Goal**.
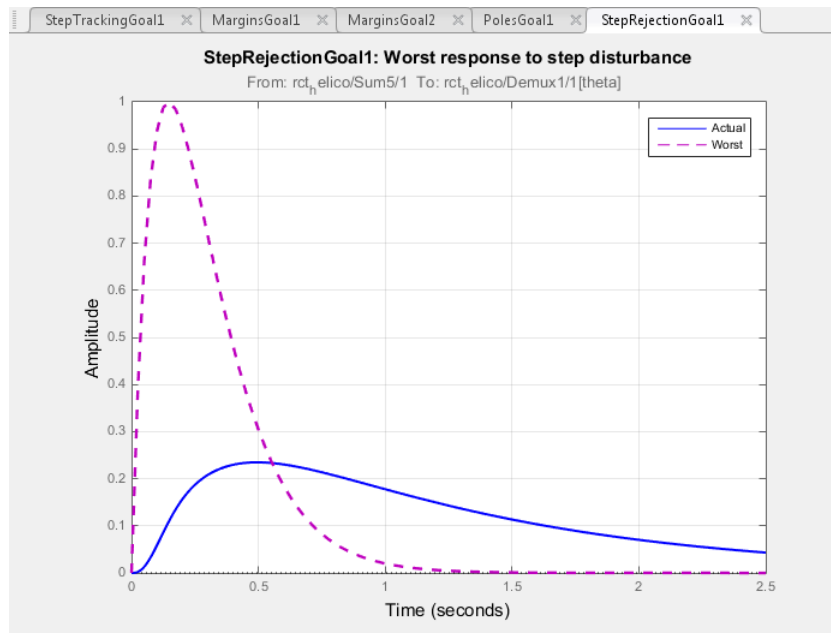
Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Rejection of step disturbance** to create a Step Rejection Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.StepRejection` to specify a step response goal.

Step Disturbance Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step disturbance inputs**

  Select one or more signal locations in your model at which to apply the input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step-disturbance response from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step response outputs**

  Select one or more signal locations in your model at which to measure the response to the step disturbance. To constrain a SISO response, select a single-valued output signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Compute the response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Desired Response to Step Disturbance

Use this section of the dialog box to specify the shape of the desired response to the step disturbance. Control System Tuner attempts to make the actual response at least as good as the desired response.

- **Response Characteristics**

  Specify the desired response in terms of time-domain characteristics. Enter the maximum amplitude, maximum settling time, and minimum damping constant in the text boxes.

- **Reference Model**

  Specify the desired response in terms of a reference model.

  Enter the name of the reference model in the MATLAB workspace in the **Reference Model** text field. Alternatively, enter a command to create a suitable reference model, such as `tf(1 0),[1 1.414 1])`.

  The reference model must be stable and must have zero DC gain. The model can be continuous or discrete. If the model is discrete, it can include time delays which are treated as poles at `z = 0`.

  For best results, the reference model and the open-loop response from the disturbance to the output should have similar gains at the frequency where the reference model gain peaks.

Options

Use this section of the dialog box to specify additional characteristics of the step rejection goal.

- **Adjust for amplitude of input signals** and **Adjust for amplitude of output signals**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued signals. This information is used to scale the off-diagonal terms in the transfer function from the tuning goal inputs to outputs. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

When these options are set to No, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to Yes to provide the relative amplitudes of the input signals and output signals of your transfer function.

For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select Yes and enter [1,100] in the **Amplitudes of input signals** text box.

Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitudes of output signals** and **Amplitudes of input signals** values on the diagonal, respectively.

The default value, No, means no scaling is applied.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning requirement is a hard constraint.

Step Rejection Goal aims to keep the gain from disturbance to output below the gain of the reference model. The scalar value of the requirement $f(x)$ is given by:

$$f\left(x\right) = \left\Vert \frac{T\left(s, x\right)}{T_{ref}\left(s\right)} \right\Vert_{\infty}.$$

$T(s,x)$ is the closed-loop transfer function from the input to the output. $T_{ref}(s)$ is the reference model. $\left\Vert \cdot \right\Vert_{\infty}$ denotes the H_{∞} norm (see `norm`).

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Transient Goal

Purpose

Shape how the closed-loop system responds to a specific input signal. Use a reference model to specify the desired transient response.

Description

Transient Goal constrains the transient response from specified input locations to specified output locations. This requirement specifies that the transient response closely match the response of a reference model. The constraint is satisfied when the relative difference between the tuned and target responses falls within the tolerance you specify.

You can constrain the response to an impulse, step, or ramp input signal. You can also constrain the response to an input signal that is given by the impulse response of an input filter you specify.
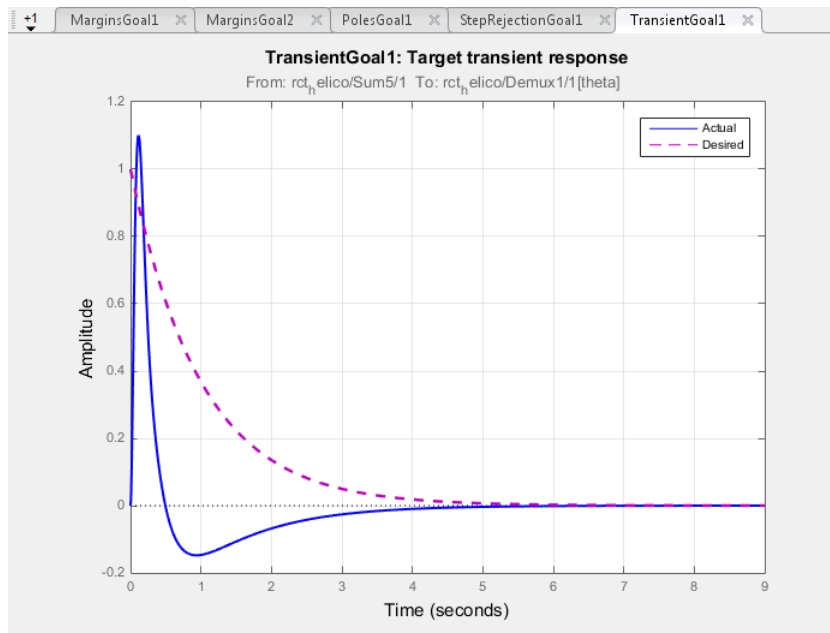
Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Transient response matching** to create a Transient Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Transient` to specify a step response goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify response inputs**

  Select one or more signal locations in your model at which to apply the input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify response outputs**

  Select one or more signal locations in your model at which to measure the transient response. To constrain a SISO response, select a single-valued output signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Compute the response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Initial Signal Selection

Select the input signal shape for the transient response you want to constrain.

- `Impulse` — Constrain the response to a unit impulse.
- `Step` — Constrain the response to a unit step. Using `Step` is equivalent to using a Step Tracking Goal.
- `Ramp` — Constrain the response to a unit ramp, `u = t`.
- `Other` — Constrain the response to a custom input signal. Specify the custom input signal by entering a transfer function (`tf` or `zpk` model) in the **Use impulse response of filter** field. The custom input signal is the response of this transfer function to a unit impulse.

  This transfer function represents the Laplace transform of the desired custom input signal. For example, to constrain the transient response to a unit-amplitude sine wave of frequency `w`, enter `tf(w,[1,0,w^2])`. This transfer function is the Laplace transform of sin(wt).

  The transfer function you enter must be continuous, and can have no poles in the open right-half plane. The series connection of this transfer function with the reference system for the desired transient response must have no feedthrough term.

Desired Transient Response

Specify the reference system for the desired transient response as a dynamic system model, such as a `tf`, `zpk`, or `ss` model. The Transient Goal constrains the system response to closely match the response of this system to the input signal you specify in **Initial Signal Selection**.

Enter the name of the reference model in the MATLAB workspace in the **Reference Model** field. Alternatively, enter a command to create a suitable reference model, such as `tf(1,[1 1.414 1])`. The reference model must be stable, and the series connection of the reference model with the input shaping filter must have no feedthrough term.

Options

Use this section of the dialog box to specify additional characteristics of the transient response goal.

- **Keep % mismatch below**

  Specify the relative matching error between the actual (tuned) transient response and the target response. Increase this value to loosen the matching tolerance. The relative matching error, e_{rel}, is defined as:

  $$\text{gap} = \frac{\left\| y(t) - y_{ref}(t) \right\|_2}{\left\| y_{ref(tr)}(t) \right\|_2}.$$

  $y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref(tr)}(t)$ is the transient portion of y_{ref} (deviation from steady-state value or trajectory). $\left\| \cdot \right\|_2$ denotes the signal energy (2-norm). The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient.

- **Adjust for amplitude of input signals** and **Adjust for amplitude of output signals**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued signals. This information is used to scale the off-diagonal terms in the transfer function from the tuning goal inputs to outputs. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  When these options are set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitudes of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1} T(s) D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitudes of output signals** and **Amplitudes of input signals** values on the diagonal, respectively.

The default value, No, means no scaling is applied.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 6-81), is infinite for continuous-time systems with nonzero feedthrough.

Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 6-33.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the

control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning requirement is a hard constraint.

For **Transient Goal**, $f(x)$ is based upon the relative gap between the tuned response and the target response:

$$\text{gap} = \frac{\left\| y(t) - y_{ref}(t) \right\|_2}{\left\| y_{ref(tr)}(t) \right\|_2}.$$

$y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref(tr)}(t)$ is the transient portion of y_{ref} (deviation from steady-state value or trajectory). $\left\| \cdot \right\|_2$ denotes the signal energy (2-norm).

The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

LQR/LQG Goal

Purpose

Minimize or limit Linear-Quadratic-Gaussian (LQG) cost in response to white-noise inputs

Description

LQR/LQG Goal specifies a tuning requirement for quantifying control performance as an LQG cost. It is applicable to any control structure, not just the classical observer structure of optimal LQG control.

The LQG cost is given by:
$J = E(z(t)' \, QZ \, z(t))$.

$z(t)$ is the system response to a white noise input vector $w(t)$. The covariance of $w(t$ is given by:
$E(w(t)w(t)') = QW$.

The vector $w(t)$ typically consists of external inputs to the system such as noise, disturbances, or command. The vector $z(t)$ includes all the system variables that characterize performance, such as control signals, system states, and outputs. $E(x)$ denotes the expected value of the stochastic variable x.

The cost function J can also be written as an average over time:

$$J = \lim_{T \to \infty} E\left(\frac{1}{T} \int_0^T z(t)' \, QZ \, z(t) dt \right).$$

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **LQR/LQG objective** to create an LQR/LQG Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.LQG` to specify an LQR/LQG goal.

Signal Selection

Use this section of the dialog box to specify noise input locations and performance output locations. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify noise inputs (w)**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the LQG cost for a noise input 'u' and performance output 'y', click ✚ **Add signal to list** and select 'u'. To constrain the LQG cost for a MIMO response, select multiple signals or a vector-valued signal.

- **Specify performance outputs (z)**

  Select one or more signal locations in your model as performance outputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the LQG cost for a noise input 'u' and performance output 'y', click ✚ **Add signal to list** and select 'y'. To constrain the LQG cost for a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate LQG objective with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named 'x', ✚ **Add signal to list** and select 'x'.

LQG Objective

Use this section of the dialog box to specify the noise covariance and performance weights for the LQG goal.

- **Performance weight Qz**

  Performance weights, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix. Use a diagonal matrix to independently scale or penalize the contribution of each variable in *z*.

The performance weights contribute to the cost function according to:
$J = E(z(t)'\, Qz\, z(t))$.

When you use the LQG goal as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select Qz values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

- **Noise Covariance Qw**

  Covariance of the white noise input vector $w(t)$, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix with as many rows as there are entries in the vector $w(t)$. A diagonal matrix means the entries of $w(t)$ are uncorrelated.

  The covariance of $w(t$ is given by:
  $E(w(t)w(t)') = QW$.

  When you are tuning a control system in discrete time, the LQG goal assumes:
  $E(w[k]w[k]') = QW/T_s$.

  T_s is the model sample time. This assumption ensures consistent results with tuning in the continuous-time domain. In this assumption, $w[k]$ is discrete-time noise obtained by sampling continuous white noise $w(t)$ with covariance QW. If in your system $w[k]$ is a truly discrete process with known covariance QWd, use the value T_s*QWd for the QW value.

Options

Use this section of the dialog box to specify additional characteristics of the LQG goal.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model

array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal, is infinite for continuous-time systems with nonzero feedthrough.

Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 6-33.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **LQR/LQG Goal**, $f(x)$ is given by the cost function J:
$J = E(z(t)'\ Qz\ z(t))$.

When you use the LQG requirement as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select Qz values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

Related Examples

Gain Goal

Purpose

Limit gain of a specified input/output transfer function.

Description

Gain Goal limits the gain from specified inputs to specified outputs. If you specify multiple inputs and outputs, Gain Goal limits the largest singular value of the transfer matrix. (See `sigma` for more information about singular values.) You can specify a constant maximum gain at all frequencies. Alternatively, you can specify a frequency-dependent gain profile.

Use Gain Goal, for example, to enforce a custom roll-off rate in a particular frequency band. To do so, specify a maximum gain profile in that band. You can also use Gain Goal to enforce disturbance rejection across a particular input/output pair by constraining the gain to be less than 1.
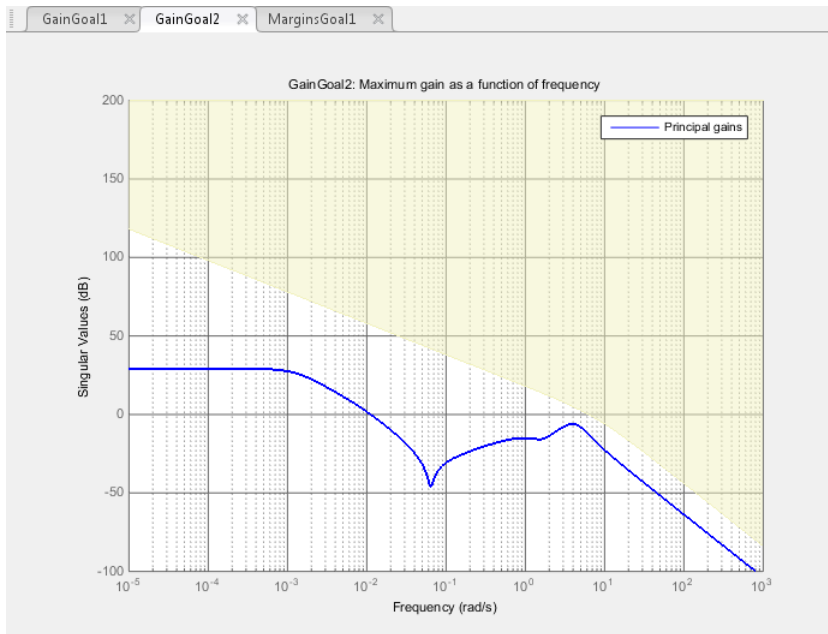
In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the maximum gain goal is not met.

By default, Gain Goal constrains a closed-loop gain. To constrain a gain computed with one or more loops open, specify loop-opening locations in the **I/O Transfer Selection** section of the dialog box.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Gain limits** to create a Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Gain` to specify a maximum gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click ➕ **Add signal to list** and select `'u'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click ➕ **Add signal to list** and select `'y'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Options

Use this section of the dialog box to specify additional characteristics of the gain goal.

*   **Limit gain to**

    Enter the maximum gain in the text box. You can specify a scalar value or a frequency-dependent gain profile. To specify a frequency-dependent gain profile, enter a SISO numeric LTI model. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise maximum gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum disturbance rejection. For example, to specify a gain profile that rolls off at –40dB/decade in the frequency band from 8 to 800 rad/s, enter `frd([0.8 8 800],[10 1 1e-4])`.

*   **Stabilize I/O transfer**

    By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

*   **Enforce goal in frequency range**

    Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

*   **Adjust for signal amplitude**

When this option is set to No, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to Yes to provide the relative amplitudes of the input signals and output signals of your transfer function.

For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select Yes and enter [1,100] in the **Amplitude of input signals** text box.

Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitude of output signals** and **Amplitude of input signals** values on the diagonal, respectively.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| \frac{1}{\mathrm{MaxGain}} D_o^{-1} T(s,x) D_i \right\|_\infty .$$

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. MaxGain is the maximum gain profile you provide for the gain goal. D_o and D_i are the scaling matrices described in "Options" on page 6-90.

$\left\| \cdot \right\|_\infty$ denotes the H_∞ norm (see norm).

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Variance Goal

Purpose

Limit white-noise impact on specified output signals.

Description

Variance Goal imposes a noise attenuation constraint that limits the impact on specified output signals of white noise applied at specified inputs. The noise attenuation is measured by the ratio of the noise variance to the output variance.

For stochastic inputs with a nonuniform spectrum (colored noise), use "Weighted Variance Goal" on page 6-120 instead.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Signal variance attenuation** to create a Variance Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Variance` to specify a constraint on noise amplification.

I/O Transfer Selection

Use this section of the dialog box to specify noise input locations and response outputs. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify stochastic inputs**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from

  a location named `'u'` to a location named `'y'`, click ➕ **Add signal to list** and select `'u'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify stochastic outputs**

  Select one or more signal locations in your model as outputs for computing response to the noise inputs. To constrain a SISO response, select a single-valued output signal.

For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute output variance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Options

Use this section of the dialog box to specify additional characteristics of the variance goal.

- **Attenuate input variance by a factor**

  Enter the desired noise attenuation from the specified inputs to outputs. This value specifies the maximum ratio of noise variance to output variance.

  When you tune a control system in discrete time, this requirement assumes that the physical plant and noise process are continuous, and interprets the desired noise attenuation as a bound on the continuous-time H_2 norm. This ensures that continuous-time and discrete-time tuning give consistent results. If the plant and noise processes are truly discrete, and you want to bound the discrete-time H_2 norm instead, multiple the desired attenuation value by $\sqrt{T_s}$. T_s is the sampling time of the model you are tuning.

- **Adjust for signal amplitude**

  When this option is set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitude of input signals** text box.

Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitude of output signals** and **Amplitude of input signals** values on the diagonal, respectively.

·   **Apply goal to**

This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 6-96), is infinite for continuous-time systems with nonzero feedthrough.

Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 6-33.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Variance Goal**, $f(x)$ is given by:

$$f(x) = \left\| \text{Attenuation} \cdot T(s,x) \right\|_2.$$

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. $\|\cdot\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \left\| \frac{\text{Attenuation}}{\sqrt{T_s}} T(z,x) \right\|_2.$$

T_s is the sampling time of the discrete-time transfer function $T(z,x)$.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Reference Tracking Goal

Purpose

Make specified outputs track reference inputs with prescribed performance and fidelity. Limit cross-coupling in MIMO systems.

Description

Tracking Goal constrains tracking between the specified signal locations. The constraint is satisfied when the maximum relative tracking error falls below the value you specify at all frequencies. The *relative error* is the gain from reference input to tracking error as a function of frequency.

You can specify the maximum error profile directly as a function of frequency. Alternatively, you can specify the tracking goal a target DC error, peak error, and response time. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\mathrm{MaxError} = \frac{(\mathrm{PeakError})s + \omega_c\,(\mathrm{DCError})}{s + \omega_c}.$$

Here, ω_c is 2/(response time). The following plot illustrates these relationships for an example set of values.
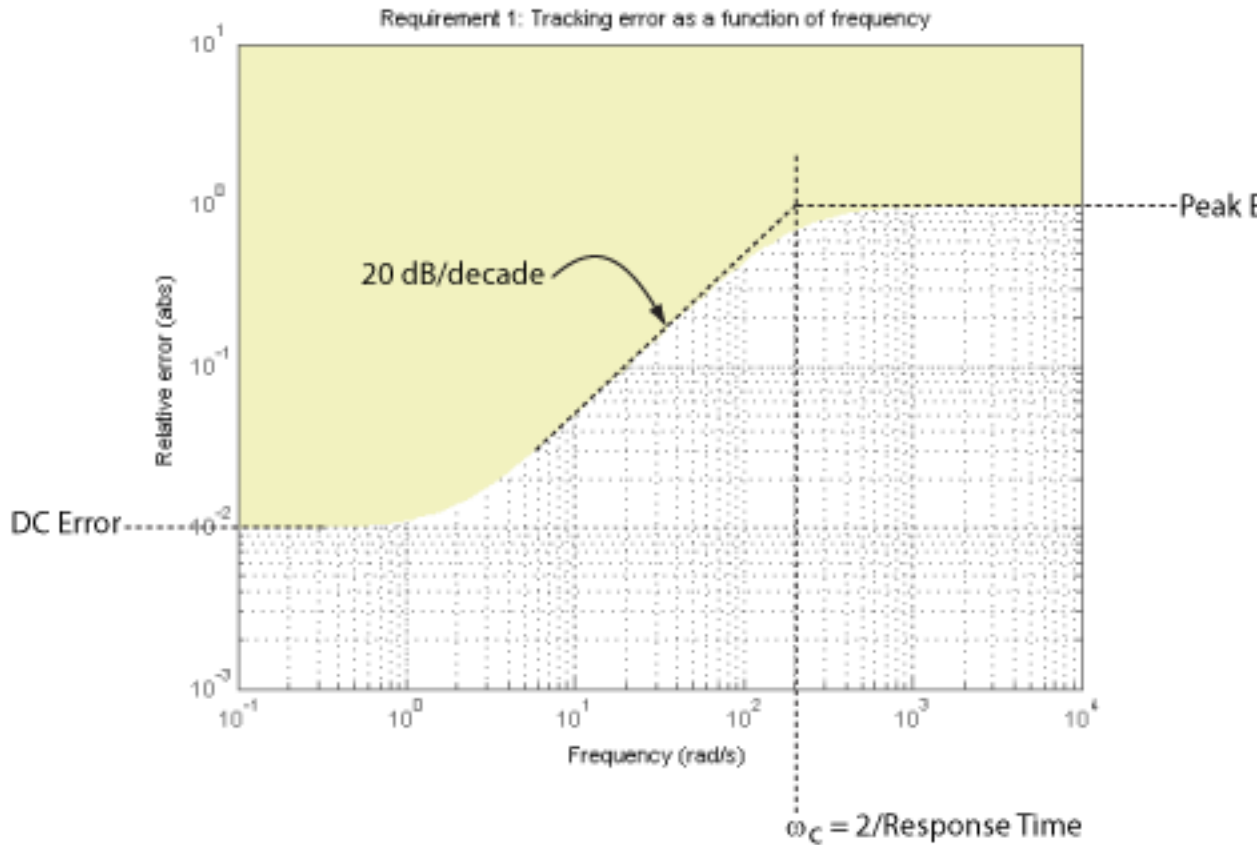
Requirement 1: Tracking error as a function of frequency

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the tracking goal is not met.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Reference Tracking** to create a Reference Tracking Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Tracking` to specify a tracking goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify reference inputs**

  Select one or more signal locations in your model as reference signals. To constrain a SISO response, select a single-valued reference signal. For example, to constrain the response from a location named `'u'` to a location named `'y'`, click ➕ **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify reference-tracking outputs**

  Select one or more signal locations in your model as reference-tracking outputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click ➕ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Evaluate tracking performance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ➕ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Tracking Performance

Use this section of the dialog box to specify frequency-domain constraints on the tracking error.

Response time, DC error, and peak error

Select this option to specify the tracking error in terms of response time, percent steady-state error, and peak error across all frequencies. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\mathrm{MaxError} = \frac{(\mathrm{PeakError})s + \omega_c (\mathrm{DCError})}{s + \omega_c}.$$

When you select this option, enter the following parameters in the text boxes:

- **Response Time** — Enter the target response time. The tracking bandwidth is given by $\omega_c = 2/\mathrm{Response\ Time}$. Express the target response time in the time units of your model.

- **Steady-state error (%)** — Enter the maximum steady-state fractional tracking error, expressed in percent. For MIMO tracking goals, this steady-state error applies to all I/O pairs. The steady-state error is the DC error expressed as a percentage, `DCError`/100.

- **Peak error across frequency (%)** — Enter the maximum fractional tracking error across all frequencies, expressed in percent.

Maximum error as a function of frequency

Select this option to specify the maximum tracking error profile as a function of frequency.

Enter a SISO numeric LTI model in the text box. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise error profile using an `frd` model. When you do so, the software automatically maps the error profile to a smooth transfer function that approximates the desired error profile. For example, to specify a maximum error of 0.01 below about 1 rad/s, gradually rising to a peak error of 1 at 100 rad/s, enter `frd([0.01 0.01 1],[0 1 100])`.

For MIMO tracking goals, this error profile applies to all I/O pairs.

Options

Use this section of the dialog box to specify additional characteristics of the tracking goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  For example, suppose that tuning goal is that outputs `'y1'` and `'y2'` track reference signals `'r1'` and `'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. This tells Control System Tuner to take into account that the first reference signal is 100 times greater than the second reference signal.

  The default value, `No`, means no scaling is applied.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter

the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Tracking Goal**, $f(x)$ is given by:

$$f(x) = \left\| \frac{1}{\text{MaxError}} \big(T(s,x) - I \big) \right\|_\infty .$$

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $\left\| \cdot \right\|_\infty$ denotes the H_∞ norm (see norm).

Related Examples
- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152
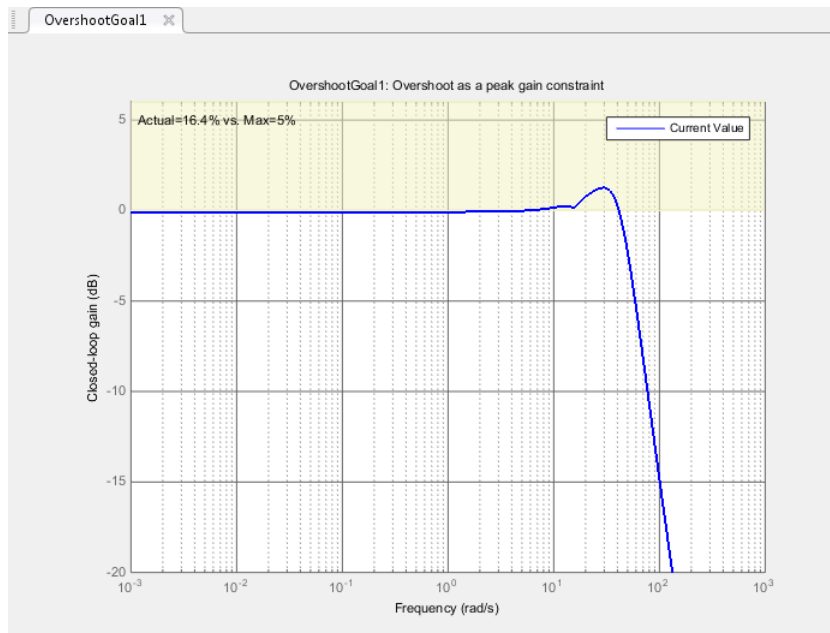
Overshoot Goal

Purpose

Limit overshoot in the step response from specified inputs to specified outputs.

Description

Overshoot Goal limits the overshoot in the step response between the specified signal locations. The constraint is satisfied when the overshoot in the tuned response is less than the target overshoot

The software maps the maximum overshoot to a peak gain constraint, assuming second-order system characteristics. Therefore, for tuning higher-order systems, the overshoot constraint is only approximate. In addition, the Overshoot Goal cannot reliably reduce the overshoot below 5%.

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the overshoot goal is not met.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Maximum overshoot** to create an Overshoot Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Overshoot` to specify a step response goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step-response inputs**

  Select one or more signal locations in your model at which to apply the step input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click

  ✚ **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step-response outputs**

  Select one or more signal locations in your model at which to measure the response to the step input. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location

  named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Evaluate overshoot with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location

  named `'x'`, ✚ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Options

Use this section of the dialog box to specify additional characteristics of the overshoot goal.

- **Limit % overshoot to**

  Enter the maximum percent overshoot. Overshoot Goal cannot reliably reduce the overshoot below 5%

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  For example, suppose that tuning goal is that outputs `'y1'` and `'y2'` track reference signals `'r1'` and `'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. This tells Control System Tuner to take into account that the first reference signal is 100 times greater than the second reference signal.

  The default value, `No` , means no scaling is applied.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Overshoot Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. The percent deviation from $f(x) = 1$ roughly corresponds to the percent deviation from the specified overshoot target. For example, $f(x) = 1.2$ means the actual overshoot exceeds the target by roughly 20%, and $f(x) = 0.8$ means the actual overshoot is about 20% less than the target.

Overshoot Goal uses $\|T\|_\infty$ as a proxy for the overshoot, based on second-order model characteristics. Here, T is the closed-loop transfer function that the requirement constrains. The overshoot is tuned in the range from 5% ($\|T\|_\infty = 1$) to 100% ($\|T\|_\infty$).

Overshoot Goal is ineffective at forcing the overshoot below 5%.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

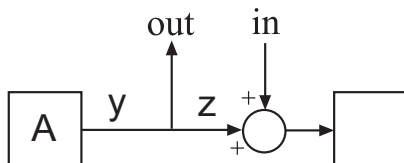# Disturbance Rejection Goal

Purpose

Attenuate disturbances at particular locations and in particular frequency bands.

Description

Disturbance Rejection Goal specifies the minimum attenuation of a disturbance injected at a specified location in a control system.

When you use this tuning goal, the software attempts to tune the system so that the attenuation of a disturbance at the specified location exceeds the minimum attenuation factor you specify. This attenuation factor is the ratio between the open- and closed-loop sensitivities to the disturbance, and is a function of frequency.

The following diagram illustrates how the attenuation factor is calculated. Suppose you specify a location in your control system, y, which is the output of a block A. In that case, the software calculates the closed-loop sensitivity at out to a signal injected at in. The software also calculates the sensitivity with the control loop opened at the location z.

To specify a Disturbance Rejection Goal, you specify one or more locations at which to attenuate disturbance. You also provide the frequency-dependent minimum attenuation factor as a numeric LTI model. You can achieve disturbance attenuation only inside the control bandwidth. The loop gain must be larger than one for the disturbance to be attenuated (attenuation factor > 1).
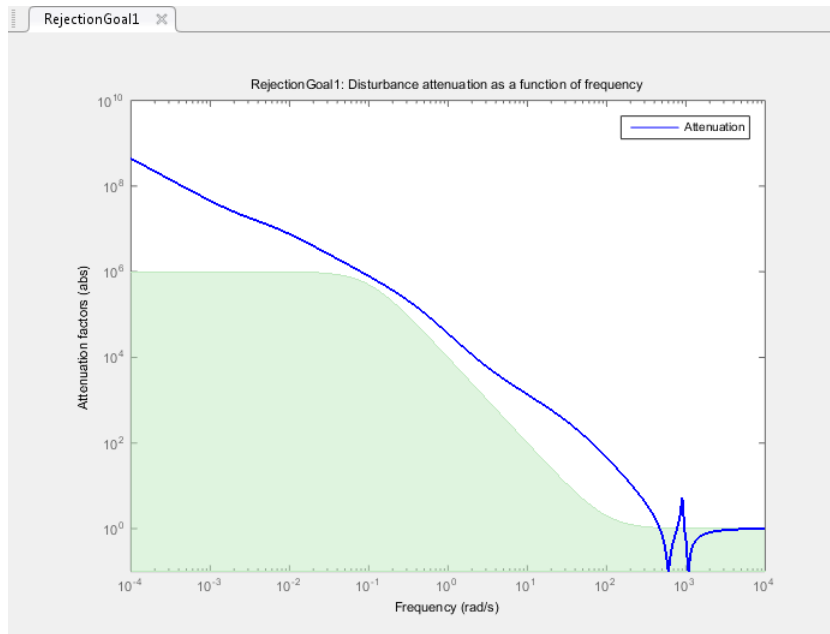
In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the disturbance rejection goal is not met.

If you prefer to specify sensitivity to disturbance at a location, rather than disturbance attenuation, you can use "Sensitivity Goal" on page 6-112.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Disturbance rejection** to create a Disturbance Rejection Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Rejection` to specify a disturbance rejection goal.

Disturbance Scenario

Use this section of the dialog box to specify the signal locations at which to inject the disturbance. You can also specify loop-opening locations for evaluating the tuning goal.

- **Inject disturbances at the following locations**

  Select one or more signal locations in your model at which to measure the disturbance attenuation. To constrain a SISO response, select a single-valued location. For

  example, to attenuate disturbance at a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location

  named `'x'`, ✚ **Add signal to list** and select `'x'`.

Rejection Performance

Specify the minimum disturbance attenuation as a function of frequency.

Enter a SISO numeric LTI model whose magnitude represents the desired attenuation profile as a function of frequency. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise minimum disturbance rejection using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum disturbance rejection. For example, to specify an attenuation factor of 100 (40 dB) below 1 rad/s, that gradually drops to 1 (0 dB) past 10 rad/s, enter `frd([100 100 1 1],[0 1 10 100])`.

Options

Use this section of the dialog box to specify additional characteristics of the disturbance rejection goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

Regardless of the limits you enter, a disturbance rejection goal can only be enforced within the control bandwidth.

- **Equalize cross-channel effects**

  For multiloop or MIMO disturbance rejection requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Disturbance Rejection Goal**, $f(x)$ is given by:

$$f(x) = \max_{\omega \in \Omega} \left\| W(j\omega) S(j\omega, x) \right\|.$$

$W(j\omega)$ is a rational transfer function whose magnitude approximates the minimum disturbance attenuation that you specify for the tuning goal. $S(j\omega, x)$ is the closed-loop sensitivity function measured at the disturbance location. Ω is the frequency interval over which the requirement is enforced.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44

-

Sensitivity Goal

Purpose

Limit sensitivity of feedback loops to disturbances.

Description

Sensitivity Goal limits the sensitivity of a feedback loop to disturbances. You specify the maximum sensitivity as a function of frequency. Constrain the sensitivity to be smaller than one at frequencies where you need good disturbance rejection.

To specify a Sensitivity Goal, you specify one or more locations at which to limit sensitivity. You also provide the frequency-dependent maximum sensitivity as a numeric LTI model whose magnitude represents the desired sensitivity as a function of frequency.
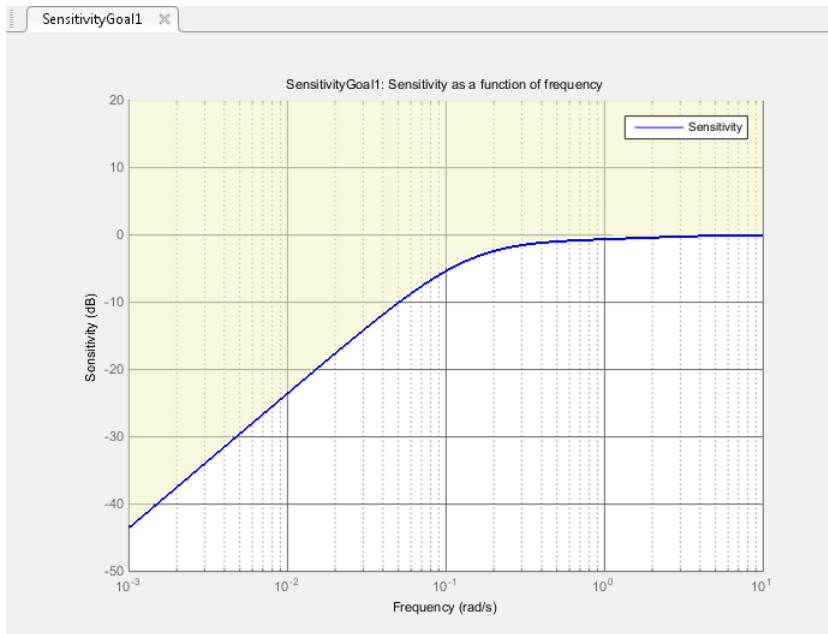
In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the sensitivity goal is not met.

If you prefer to specify disturbance attenuation at a particular location, rather than sensitivity to disturbance, you can use "Disturbance Rejection Goal" on page 6-107.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Sensitivity of feedback loops** to create a Sensitivity Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Sensitivity` to specify a disturbance rejection goal.

Sensitivity Evaluation

Use this section of the dialog box to specify the signal locations at which to compute the sensitivity to disturbance. You can also specify loop-opening locations for evaluating the tuning goal.

- **Measure sensitivity at the following locations**

  Select one or more signal locations in your model at which to measure the sensitivity to disturbance. To constrain a SISO response, select a single-valued location. For example, to limit sensitivity at a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Sensitivity Bound

Specify the maximum sensitivity as a function of frequency.

Enter a SISO numeric LTI model whose magnitude represents the desired sensitivity bound as a function of frequency. For example, you can specify a smooth transfer function

(`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise maximum sensitivity using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired sensitivity. For example, to specify a sensitivity that rolls up at 20 dB per decade and levels off at unity above 1 rad/s, enter `frd([0.01 1 1],[0.001 0.1 100])`.

Options

Use this section of the dialog box to specify additional characteristics of the sensitivity goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Equalize cross-channel effects**

  For multiloop or MIMO sensitivity requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the

control system. The software then adjusts the parameter values to minimize *f(x)* or to drive *f(x)* below 1 if the tuning goal is a hard constraint.

For **Sensitivity Goal**, *f(x)* is given by:

$$f(x) = \left\| \frac{1}{S_{max}(s)} S(s,x) \right\|_{\infty}.$$

S(s,x) is the sensitivity function of the control system at the specified location, evaluated with parameter values *x*. $S_{max}(s)$ is the frequency-dependent maximum sensitivity you specify. $\left\| \cdot \right\|_{\infty}$ denotes the H_{∞} norm (see `norm`).

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Weighted Gain Goal

Purpose

Frequency-weighted gain limit.

Description

Weighted Gain Goal limits the gain of the frequency-weighted transfer function $WL(s)H(s)WR(s)$, where $H(s)$ is the transfer function between inputs and outputs you specify. $WL(s)$ and $WR(s)$ are weighting functions that you can use to emphasize particular frequency bands. Weighted Gain Goal constrains the peak gain of $WL(s)H(s)WR(s)$ to values less than 1. If $H(s)$ is a MIMO transfer function, Weighted Gain Goal constrains the largest singular value of $H(s)$.

By default, Weighted Gain Goal constrains a closed-loop gain. To constrain a gain computed with one or more loops open, specify loop-opening locations in the **I/O Transfer Selection** section of the dialog box.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Frequency-weighted gain limit** to create a Weighted Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedGain` to specify a weighted gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

· **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a

location named `'y'`, click ➕ **Add signal to list** and select `'u'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click ➕ **Add signal to list** and select `'y'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ➕ **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 6-44.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal. The tuning goal ensures that the gain $H(s)$ from the specified input to output satisfies the inequality:
$||WL(s)H(s)WR(s)||_\infty < 1$.
WL provides the weighting for the output channels of $H(s)$, and *WR* provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting function. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify different weights for each channel by specifying MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of *WL* and *WR*. For

example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

Options

Use this section of the dialog box to specify additional characteristics of the weighted gain goal.

- **Stabilize I/O transfer**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| WL \, H(s,x) \, WR \right\|_\infty .$$

$H(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $\left\| \cdot \right\|_\infty$ denotes the H_∞ norm (see `norm`).

Related Examples
- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Weighted Variance Goal

Purpose

Frequency-weighted limit on noise impact on specified output signals.

Description

Weighted Variance Goal limits the noise impact on the outputs of the frequency-weighted transfer function $WL(s)H(s)WR(s)$, where $H(s)$ is the transfer function between inputs and outputs you specify. $WL(s)$ and $WR(s)$ are weighting functions you can use to model a noise spectrum or emphasize particular frequency bands. Thus, you can use Weighted Variance Goal to tune the system response to stochastic inputs with a nonuniform spectrum such as colored noise or wind gusts.

Weighted Variance minimizes the response to noise at the inputs by minimizing the H_2 norm of the frequency-weighted transfer function. The H_2 norm measures:

- The total energy of the impulse response, for deterministic inputs to the transfer function.
- The square root of the output variance for a unit-variance white-noise input, for stochastic inputs to the transfer function. Equivalently, the H_2 norm measures the root-mean-square of the output for such input.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Frequency-weighted variance attenuation** to create a Weighted Variance Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedVariance` to specify a weighted gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify noise input locations and response outputs. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify stochastic inputs**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'u'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify stochastic outputs**

  Select one or more signal locations in your model as outputs for computing response to the noise inputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute output variance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal.

WL provides the weighting for the output channels of *H*(*s*), and *WR* provides the weighting for the input channels.

You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting as a function of frequency. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s. To limit the response to a nonuniform noise distribution, enter as *WR* an LTI model whose magnitude represents the noise spectrum.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify

different weights for each channel by specifying MIMO weighting functions. The dimensions *H*(*s*) must be commensurate with the dimensions of *WL* and *WR*. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

Options

Use this section of the dialog box to specify additional characteristics of the weighted variance goal.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 6-123), is infinite for continuous-time systems with nonzero feedthrough.

Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 6-33.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Variance Goal**, $f(x)$ is given by:

$$f(x) = \left\| WL\, H(s,x)\, WR \right\|_2 .$$

$H(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $\left\| \cdot \right\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \frac{1}{\sqrt{T_s}} \left\| WL(z)\, H(z,x)\, WR(z) \right\|_2 .$$

T_s is the sampling time of the discrete-time transfer function $H(z,x)$.

Related Examples
- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Minimum Loop Gain Goal

Purpose

Boost gain of feedback loops at low frequency

Description

Minimum Loop Gain Goal enforces a minimum loop gain in a particular frequency band. This tuning goal is useful, for example, for improving disturbance rejection at a particular location.

Minimum Loop Gain Goal imposes a minimum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the minimum open-loop gain as a function of frequency (a minimum *gain profile*). For MIMO feedback loops, the specified gain profile is interpreted as a lower bound on the smallest singular value of L.

When you tune a control system, the minimum gain profile is converted to a minimum gain constraint on the inverse of the sensitivity function, $\mathrm{inv}(S) = (I + L)$.

The following figure shows a typical specified minimum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The green region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much larger than 1, imposing a minimum gain on $\mathrm{inv}(S)$ is a good proxy for a minimum open-loop gain.
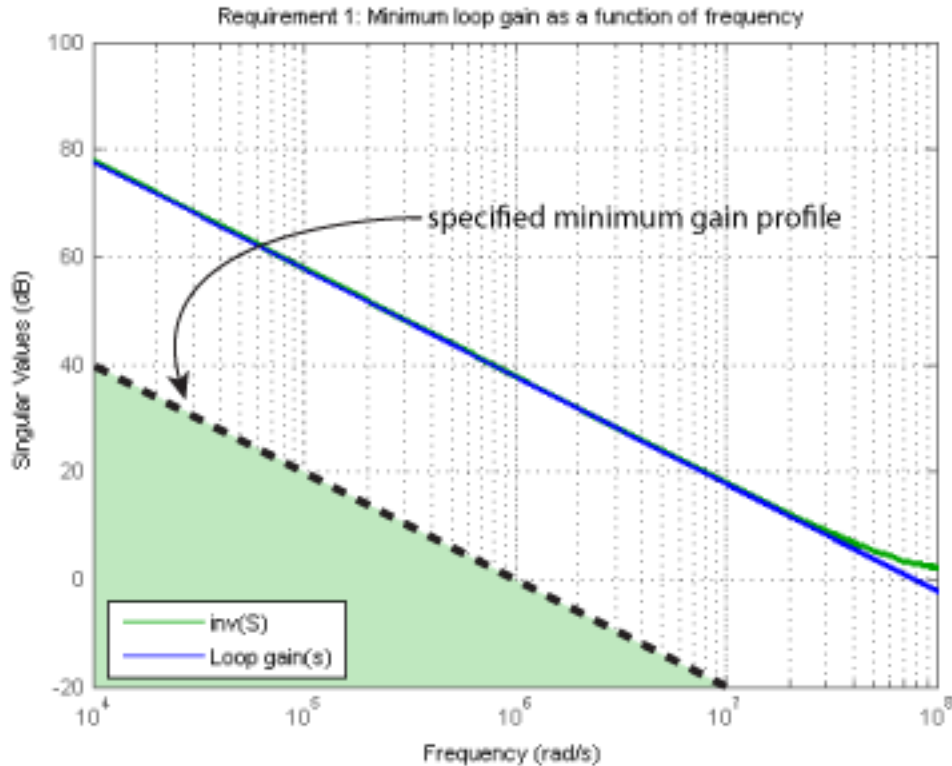
Requirement 1: Minimum loop gain as a function of frequency

Minimum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the "Open-Loop Response Selection" on page 6-126 section of the dialog box.

Minimum Loop Gain Goal and Maximum Loop Gain Goal specify only low-gain or high-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use "Loop Shape Goal" on page 6-134 to specify that target loop shape.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Minimum gain for open-loop response** to create a Minimum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.MinLoopGain` to specify a minimum loop gain goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

·   **Shape open-loop response at the following locations**

Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location.

For example, to constrain the open-loop gain at a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

·   **Compute response with the following loops open**

Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Desired Loop Gain

Use this section of the dialog box to specify the target minimum loop gain.

·   **Pure integrator K/s**

Check to specify a pure integrator shape for the target minimum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a

target minimum gain and frequency. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain above** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the minimum loop gain is 1 at 10 rad/s.

· **Other gain profile**

Check to specify the minimum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum loop gain. For example, to specify minimum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

Options

Use this section of the dialog box to specify additional characteristics of the minimum loop gain goal.

· **Enforce goal in frequency range**

Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

· **Stabilize closed loop system**

By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

· **Equalize loop interactions**

For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Minimum Loop Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_S \left(D^{-1} S D \right) \right\|_\infty .$$

W_S is the minimum loop gain profile. D is a diagonal scaling (for MIMO loops). S is the sensitivity function at the specified location.

Although S is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing a lower bound on the open-loop transfer function, L, in a frequency band where the gain of L is greater than 1. To see why, note that $S = 1/(1 + L)$. For SISO loops, when $|L| \gg 1$, $|S| \approx 1/|L|$. Therefore, enforcing the open-loop minimum gain requirement, $|L| > |W_S|$, is roughly equivalent to enforcing $|W_s S| < 1$. For MIMO loops, similar reasoning applies, with $||S|| \approx 1/\sigma_{\min}(L)$, where $\sigma_{\min}$ is the smallest singular value.

Related Examples
- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Maximum Loop Gain Goal

Purpose

Suppress gain of feedback loops at high frequency

Description

Maximum Loop Gain Goal enforces a maximum loop gain in a particular frequency band. This tuning goal is useful, for example, for increasing system robustness to unmodeled dynamics.

Maximum Loop Gain Goal imposes a maximum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the maximum open-loop gain as a function of frequency (a maximum *gain profile*). For MIMO feedback loops, the specified gain profile is interpreted as an upper bound on the largest singular value of L.

When you tune a control system, the maximum gain profile is converted to a maximum gain constraint on the complementary sensitivity function, $T) = L/(I + L)$.

The following figure shows a typical specified maximum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The pink region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much smaller than 1, imposing a maximum gain on T is a good proxy for a maximum open-loop gain.
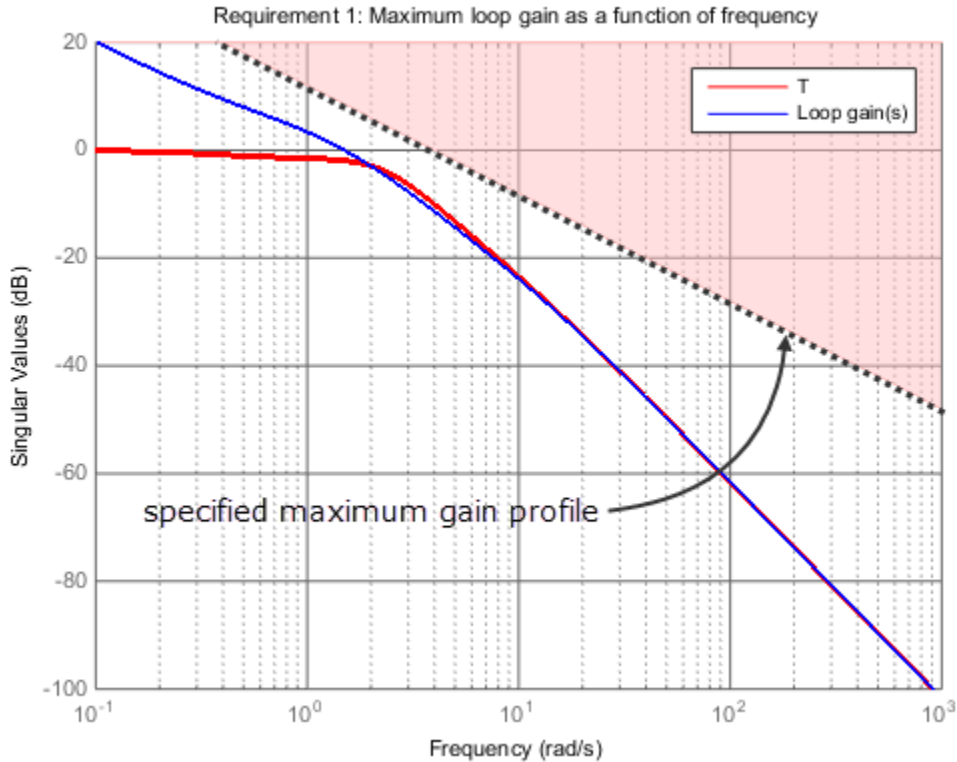
Requirement 1: Maximum loop gain as a function of frequency

Maximum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Opel-Loop Response Selection** section of the dialog box.

Maximum Loop Gain Goal and Minimum Loop Gain Goal specify only high-gain or low-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use "Loop Shape Goal" on page 6-134 to specify that target loop shape.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Maximum gain for open-loop response** to create a Maximum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.MaxLoopGain` to specify a maximum loop gain goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location.

  For example, to constrain the open-loop gain at a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Desired Loop Gain

Use this section of the dialog box to specify the target maximum loop gain.

- **Pure integrator K/s**

  Check to specify a pure integrator shape for the target maximum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a

target maximum gain and frequency. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain below** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the maximum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the maximum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired maximum loop gain. For example, to specify maximum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

Options

Use this section of the dialog box to specify additional characteristics of the maximum loop gain goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Maximum Loop Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_T \left(D^{-1} T D \right) \right\|_\infty .$$

W_T is the reciprocal of the maximum loop gain profile. D is a diagonal scaling (for MIMO loops). T is the complementary sensitivity function at the specified location.

Although T is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing an upper bound on the open-loop transfer, L, in a frequency band where the gain of L is less than one. To see why, note that $T = L/(I + L)$. For SISO loops, when $|L| \ll 1$, $|T| \approx |L|$. Therefore, enforcing the open-loop maximum gain requirement, $|L| < 1/|W_T|$, is roughly equivalent to enforcing $|W_T T| < 1$. For MIMO loops, similar reasoning applies, with $||T|| \approx \sigma_{max}(L)$, where σ_{max} is the largest singular value.

Related Examples
- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Loop Shape Goal

Purpose

Shape open-loop response of feedback loops.

Description

Loop Shape Goal specifies a target *gain profile* (gain as a function of frequency) of an open-loop response. Loop Shape Goal constrains the open-loop, point-to-point response (L) at a specified location in your control system.

When you tune a control system, the target open-loop gain profile is converted into constraints on the inverse sensitivity function $\mathrm{inv}(S) = (I + L)$ and the complementary sensitivity function $T = 1 - S$. These constraints are illustrated for a representative tuned system in the following figure.

Minimum low-frequency loop gain

Open loop CG: Minimum and maximum loop gains (CrossTol = 0.349)

Maximum high-frequency loop ga

2*CrossTol
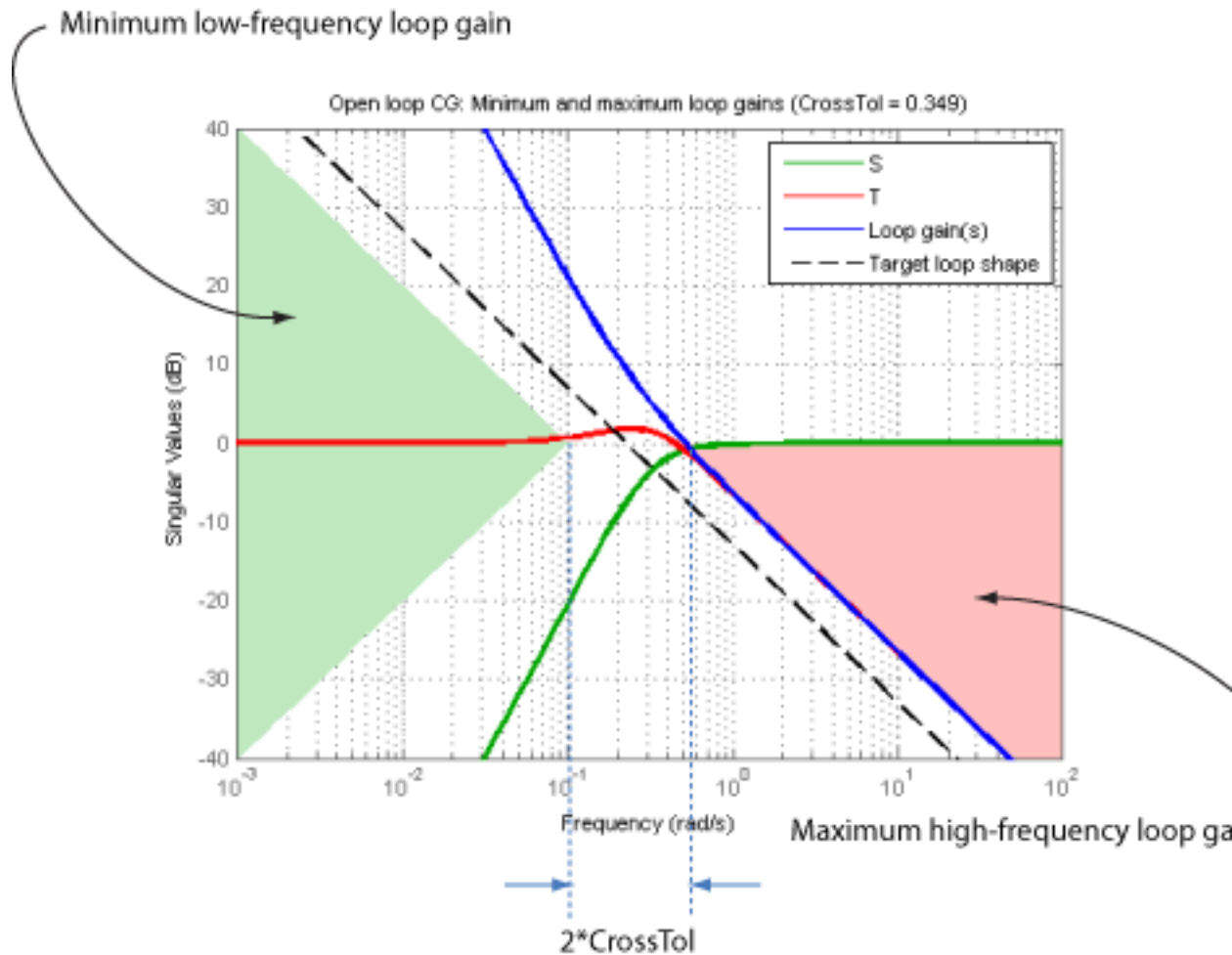
Where L is much greater than 1, a minimum gain constraint on inv(S) (green shaded region) is equivalent to a minimum gain constraint on L. Similarly, where L is much smaller than 1, a maximum gain constraint on T (red shaded region) is equivalent to a maximum gain constraint on L. The gap between these two constraints is twice the crossover tolerance, which specifies the frequency band where the loop gain can cross 0 dB.

For multi-input, multi-output (MIMO) control systems, values in the gain profile greater than 1 are interpreted as minimum performance requirements. Such values are lower

bounds on the smallest singular value of the open-loop response. Gain profile values less than one are interpreted as minimum roll-off requirements, which are upper bounds on the largest singular value of the open-loop response. For more information about singular values, see `sigma`.

Use Loop Shape Goal when the loop shape near crossover is simple or well understood (such as integral action). To specify only high gain or low gain constraints in certain frequency bands, use "Minimum Loop Gain Goal" on page 6-124 or "Maximum Loop Gain Goal" on page 6-129. When you do so, the software determines the best loop shape near crossover.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Target shape for open-loop response** to create a Loop Shape Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.LoopShape` to specify a loop-shape goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location.

  For example, to constrain the open-loop gain at a location named `'y'`, click ➕ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ➕ **Add signal to list** and select `'x'`.

Desired Loop Shape

Use this section of the dialog box to specify the target loop shape.

- **Pure integrator wc/s**

  Check to specify a pure integrator and crossover frequency for the target loop shape. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 10 in the **Crossover frequency wc** text box.

- **Other gain profile**

  Check to specify the target loop shape as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop shape using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired loop shape. For example, to specify a target loop shape of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

Options

Use this section of the dialog box to specify additional characteristics of the loop shape goal.

- **Enforce loop shape within**

  Specify the tolerance in the location of the crossover frequency, in decades. For example, to allow gain crossovers within half a decade on either side of the target crossover frequency, enter 0.5. Increase the crossover tolerance to increase the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select No to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select No.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select Off to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Loop Shape Goal**, $f(x)$ is given by:

$$f(x) = \left\| \begin{matrix} W_S S \\ W_T T \end{matrix} \right\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor. (If **Equalize loop interactions** is set to Off, then *D* = *I*.)

T = *S* − *I* is the complementary sensitivity function.

W_S and W_T are weighting functions derived from the specified loop shape.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Margins Goal

Purpose

Enforce specified gain and phase margins.

Description

Margins Goal enforces specified gain and phase margins on a SISO or MIMO feedback loop. For MIMO feedback loops, the gain and phase margins are based on the notion of disk margins, which guarantee stability for concurrent gain and phase variations in all feedback channels. See `loopmargin` for more information about disk margins.
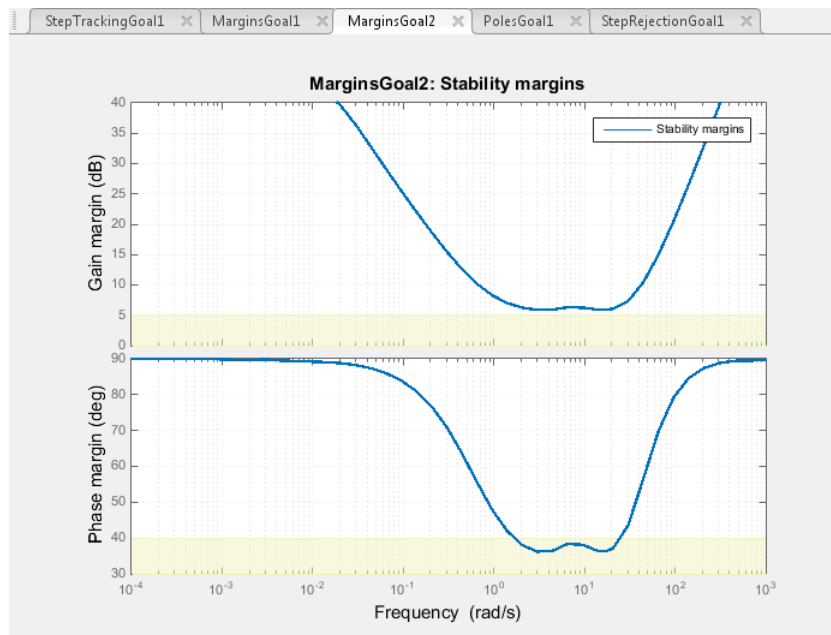
In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the margins goal is not met. For more information about interpreting this plot, see "Interpreting Stability Margins in Control System Tuning" on page 6-209.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Minimum stability margins** to create a Margins Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Margins` to specify a stability margin goal.

Feedback Loop Selection

Use this section of the dialog box to specify the signal locations at which to measure stability margins. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Measure stability margins at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the stability margins. To constrain a SISO loop, select a single-valued location. For example, to constrain the stability margins at a location named `'y'`, click ✚ **Add signal to list** and select `'y'`. To constrain a MIMO loop, select multiple signals or a vector-valued signal.

- **Measure stability margins with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Desired Margins

Use this section of the dialog box to specify the minimum gain and phase margins for the feedback loop.

- **Gain margin (dB)**

  Enter the required minimum gain margin for the feedback loop as a scalar value expressed in dB.

- **Phase margin (degrees)**

  Enter the required minimum phase margin for the feedback loop as a scalar value expressed in degrees.

For MIMO feedback loops, the gain and phase margins are based on the notion of disk margins, which guarantee stability for concurrent gain and phase variations in all feedback channels. See `loopmargin` for more information about disk margins.

Options

Use this section of the dialog box to specify additional characteristics of the stability margin goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  For best results with stability margin requirements, pick a frequency band extending about one decade on each side of the gain crossover frequencies.

- **D scaling order**

  This value controls the order (number of states) of the scalings involved in computing MIMO stability margins. Static scalings (scaling order 0) are used by default. Increasing the order may improve results at the expense of increased computations. If the stability margin plot shows a large gap between the optimized and actual margins, consider increasing the scaling order. See "Interpreting Stability Margins in Control System Tuning" on page 6-209.

- **Apply goal to**

  This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model

array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Margins Goal**, $f(x)$ is given by:

$$f(x) = \|2\alpha S - \alpha I\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor.

α is a scalar parameter computed from the specified gain and phase margin.

Related Examples
*   "Specify Goals for Interactive Tuning" on page 6-44
*   "Manage Tuning Goals" on page 6-152

More About
*   "Interpreting Stability Margins in Control System Tuning" on page 6-209

Poles Goal

Purpose

Constrain the dynamics of the closed-loop system, specified feedback loops, or specified open-loop configurations.

Description

Poles Goal constrains the dynamics of your entire control system or of specified feedback loops of your control system. Constraining the dynamics of a feedback loop means constraining the dynamics of the sensitivity function measured at a specified location in the control system.

Using Poles Goal, you can specify finite minimum decay rate or minimum damping for the poles in the control system or specified loop. You can specify a maximum natural frequency for these poles, to eliminate fast dynamics in the tuned control system.
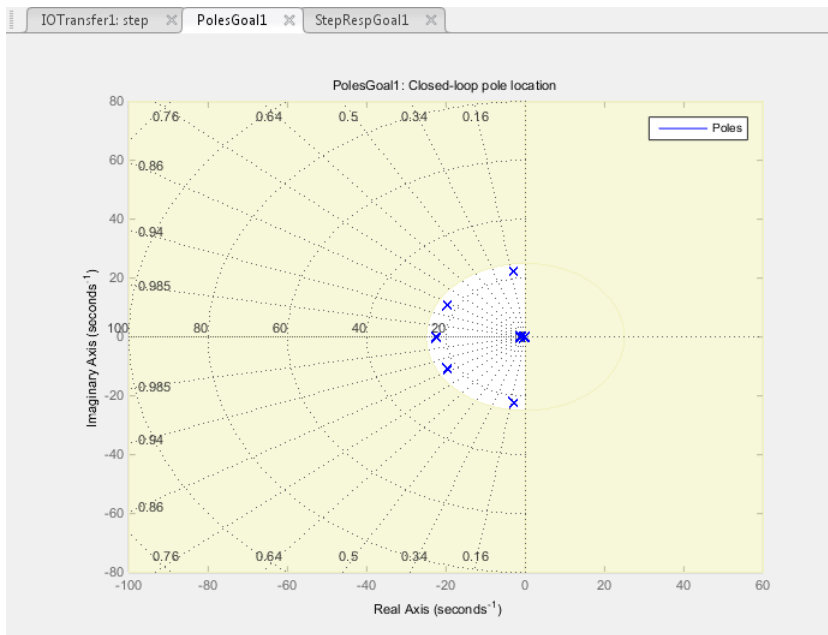
In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met.

To constrain dynamics or ensure stability of a single tunable component of the control system, use "Controller Poles Goal" on page 6-149.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Constraint on closed-loop dynamics** to create a Poles Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Poles` to specify a disturbance rejection goal.

Feedback Configuration

Use this section of the dialog box to specify the portion of the control system for which you want to constrain dynamics. You can also specify loop-opening locations for evaluating the tuning goal.

- **Entire system**

  Select this option to constrain the locations of closed-loop poles of the control system.

- **Specific feedback loop(s)**

  Select this option to specify one or more feedback loops to constrain. Specify a feedback loop by selecting a signal location in your control system. Poles Goal constrains the dynamics of the sensitivity function measured at that location. (See `getSensitivity` for information about sensitivity functions.)

  To constrain the dynamics of a SISO loop, select a single-valued location. For example, to constrain the dynamics of the sensitivity function measured at a location named `'y'`, click ✚**Add signal to list** and select `'y'`. To constrain the dynamics of a MIMO loop, select multiple signals or a vector-valued signal.

- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you

identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ➕ **Add signal to list** and select `'x'`.

Pole Location

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

  Enter the target minimum decay rate for the system poles. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDecay` for continuous-time systems, or `log(|z|) < -MinDecay*Ts` for discrete-time systems with sample time `Ts`. This constraint helps ensure stable dynamics in the tuned system.

  Enter 0 to impose no constraint on the decay rate.

- **Minimum damping**

  Enter the target minimum damping of closed-loop poles of tuned system, as a value between 0 and 1. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s=log(z)/Ts`.

  Enter 0 to impose no constraint on the damping ratio.

- **Maximum natural frequency**

  Enter the target maximum natural frequency of poles of tuned system, in the units of the control system model you are tuning. When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `|s| < MaxFrequency` for continuous-time systems, or `|log(z)| < MaxFrequency*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the control system.

  Enter `Inf` to impose no constraint on the natural frequency.

Options

Use this section of the dialog box to specify additional characteristics of the poles goal.

- **Enforce goal in frequency range**

Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form [min,max], expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter [1,100]. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

The Poles Goal applies only to poles with natural frequency within the range you specify.

- **Apply goal to**

This option applies when you are tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter 2:4 in the **Only Models** text box.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Poles Goal constrains the closed-loop poles of a feedback loop to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the smallest damping among the constrained poles is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the smallest damping $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the smallest damping $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44

- "Manage Tuning Goals" on page 6-152

Controller Poles Goal

Purpose

Constrain the dynamics of a specified tunable block in the tuned control system

Description

Controller Poles Goal constrains the dynamics of a tunable block in your control system model. Controller Poles Goal can impose a stability constraint on the specified block. You can also specify a finite minimum decay rate, a minimum damping rate, or a maximum natural frequency for the poles of the block. These constraints allow you to eliminate fast dynamics and control ringing in the response of the tunable block.
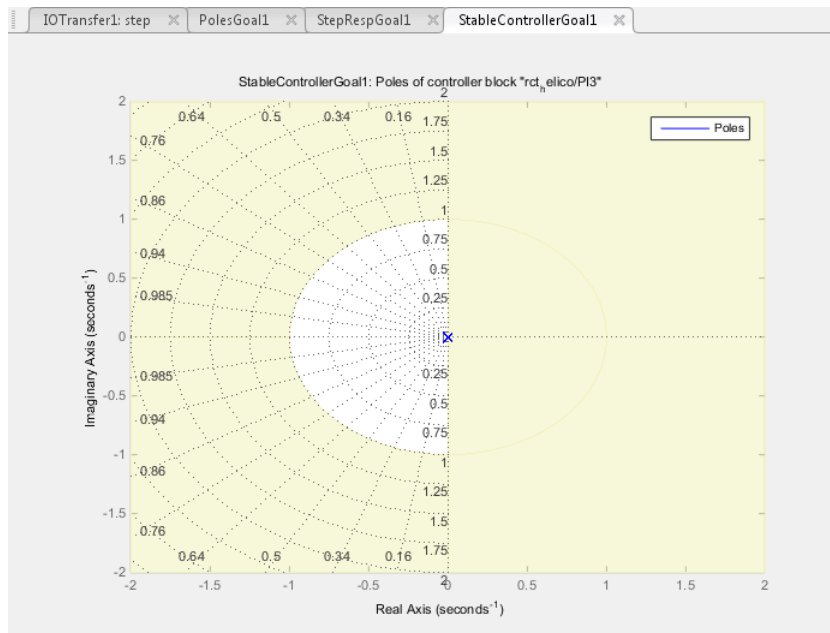
In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met. The constraint applies to all poles in the block except fixed integrators, such as the I term of a PID controller.

To constrain dynamics or ensure stability of an entire control system or a feedback loop in the control system, use "Poles Goal" on page 6-144.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Constraint on controller dynamics** to create a Controller Poles Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.ControllerPoles` to specify a disturbance rejection goal.

Constrain Dynamics of Tuned Block

From the drop-down menu, select the tuned block in your control system to which to apply the Controller Poles Goal.

If the block you want to constrain is not in the list, add it to the Tuned Blocks list. In Control System Tuner, in the **Tuning** tab, click 🔡**Select Blocks**. For more information about adding tuned blocks, see "Specify Blocks to Tune in Control System Tuner" on page 6-31.

Keep Poles Inside the Following Region

Use this section of the dialog box to specify the limits on pole locations.

• **Minimum decay rate**

   Enter the desired minimum decay rate for the poles of the tunable block. Poles of the block are constrained to satisfy `Re(s) < -MinDecay` for continuous-time blocks, or `log(|z|) < -MinDecay*Ts` for discrete-time blocks with sample time `Ts`.

   Specify a nonnegative value to ensure that the block is stable. If you specify a negative value, the tuned block can include unstable poles.

• **Minimum damping**

   Enter the desired minimum damping ratio of poles of the tunable block, as a value between 0 and 1. Poles of the block that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s=log(z)/Ts`.

- **Maximum natural frequency**

    Enter the target maximum natural frequency of poles of the tunable block, in the units of the control system model you are tuning. Poles of the block are constrained to satisfy `|s| < MaxFrequency` for continuous-time blocks, or `|log(z)| < MaxFrequency*Ts` for discrete-time blocks with sample time `Ts`. This constraint prevents fast dynamics in the tunable block.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Controller Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Controller Poles Goal constrains the pole of a tuned block to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the damping of the pole is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the damping is $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the damping is $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

Related Examples

- "Specify Goals for Interactive Tuning" on page 6-44
- "Manage Tuning Goals" on page 6-152

Manage Tuning Goals

Control System Tuner lets you designate one or more tuning goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have goals. Control System Tuner attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

By default, new goals are designated soft goals. In the **Tuning** tab, click ⣿⣿**Manage Goals** to open the **Manage tuning goals** dialog box. Check **Hard** for any goal to designate it a hard goal.

You can also designate any tuning goal as inactive for tuning. In this case the software ignores the tuning goal entirely. Use this dialog box to select which tuning goals are active when you tune the control system. **Active** is checked by default for any new goals. Uncheck **Active** for any design goal that you do not want enforced.

For example, if you tune with the following configuration, Control System Tuner optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.
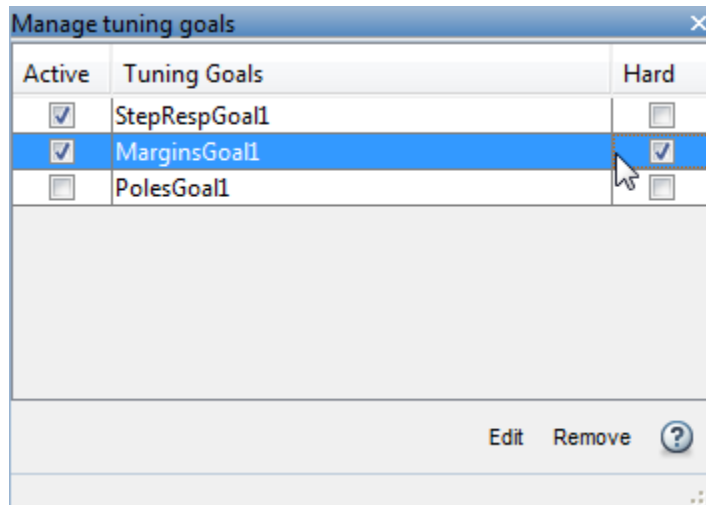
All tuning goals you have created in the Control System Tuner session are listed in the dialog box. To edit an existing tuning goal, select it in the list and click **Edit**. To delete a tuning goal from the list, select it and click **Remove**.

To add more tuning goals to the list, in Control System Tuner, in the **Tuning** tab, click **New Goal**. For more information about creating tuning goals, see "Specify Goals for Interactive Tuning" on page 6-44.

Generate MATLAB Code from Control System Tuner for Command-Line Tuning

This example shows how to generate a MATLAB script in Control System Tuner for tuning a control system at the command line. Generated scripts are useful when you want to programmatically reproduce a result you obtained interactively. A generated MATLAB script also enables you to programmatically perform multiple tuning operations with variations in tuning goals, system parameters, or model conditions such as operating point.

Tip You can also save a Control System Tuner session to reproduce within Control System Tuner. To do so, in the **Control System** tab, click ⊟ **Save Session**.

To generate a MATLAB script in Control System Tuner, in the **Tuning** tab, click **Tune**▾. Select **Script with current values**.
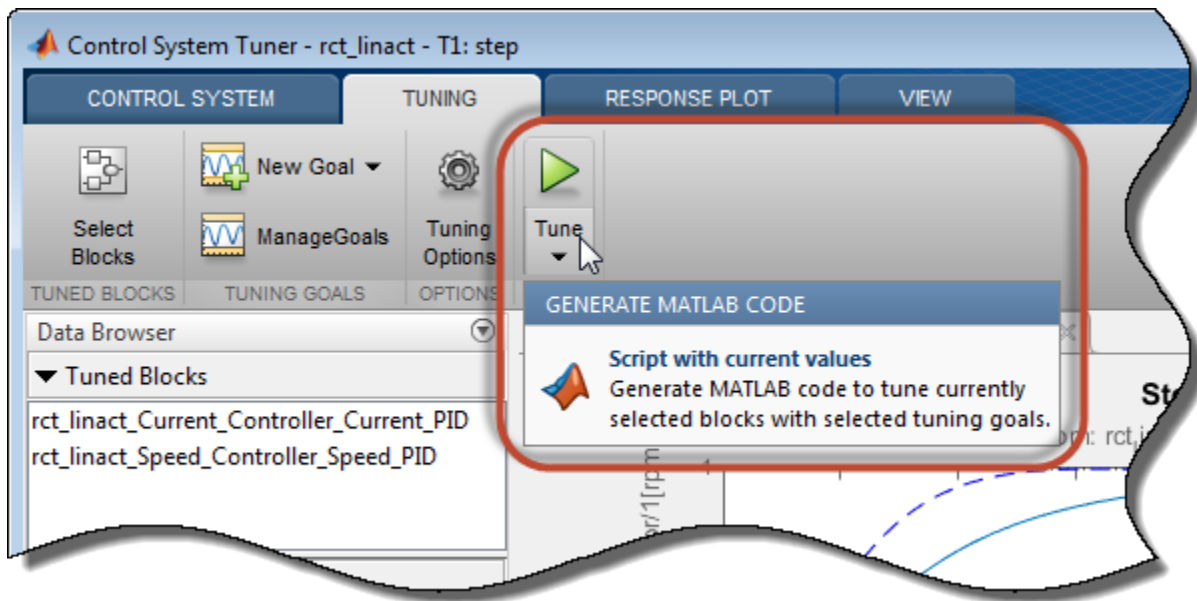
The MATLAB Editor displays the generated script, which script reproduces programmatically the current tuning configuration of Control System Tuner.

For example, suppose you generate a MATLAB script after completing all steps in the example "Control of a Linear Electric Actuator Using CONTROL SYSTEM TUNER" on page 6-289. The generated script computes the operating point used for tuning, designates the blocks to tune, creates the tuning goals, and performs other operations to reproduce the result at the command line.

The first section of the script creates the `slTuner` interface to the Simulinkmodel (`rct_linact` in this example). The `slTuner` interface stores a linearization of the model and parameterizations of the blocks to tune.

```
%% Create system data with slTuner interface
TunedBlocks = {'rct_linact/Current Controller/Current PID'; ...
               'rct_linact/Speed Controller/Speed PID'};
AnalysisPoints = {'rct_linact/Speed Demand (rpm)/1'; ...
                  'rct_linact/Current Sensor/1'; ...
                  'rct_linact/Hall Effect Sensor/1'; ...
                  'rct_linact/Speed Controller/Speed PID/1'; ...
                  'rct_linact/Current Controller/Current PID/1'};
OperatingPoints = 0.5;
% Create the slTuner object
CLO = slTuner('rct_linact',TunedBlocks,AnalysisPoints,OperatingPoints);
```

The `slTuner` interface also specifies the operating point at which the model is linearized, and marks as analysis points all the signal locations required to specify the tuning goals for the example. (See "Create and Configure slTuner Interface to Simulink Model" on page 6-178.)

If you are tuning a control system modeled in MATLAB instead of Simulink, the first section of the script constructs a `genss` model that has equivalent dynamics and parameterization to the `genss` model of the control system that you specified Control System Tuner.

Next, the script creates the three tuning goals specified in the example. The script uses `TuningGoal` objects to capture theses tuning goals. For instance, the script uses `TuningGoal.Tracking` to capture the Tracking Goal of the example.

```
%% Create tuning goal to follow reference commands with prescribed performance
% Inputs and outputs
Inputs = {'rct_linact/Speed Demand (rpm)/1'};
Outputs = {'rct_linact/Hall Effect Sensor/1[rpm]'};
```

```
% Tuning goal specifications
ResponseTime = 0.1; % Approximately reciprocal of tracking bandwidth
DCError = 0.001; % Maximum steady-state error
PeakError = 1; % Peak error across frequency
% Create tuning goal for tracking
TR = TuningGoal.Tracking(Inputs,Outputs,ResponseTime,DCError,PeakError);
TR.Name = 'TR'; % Tuning goal name
```

After creating the tuning goals, the script sets any algorithm options you had set in Control System Tuner. The script also designates tuning goals as soft or hard goals, according to the configuration of tuning goals in Control System Tuner. (See "Manage Tuning Goals" on page 6-152.)

```
%% Create option set for systune command
Options = systuneOptions();

%% Set soft and hard goals
SoftGoals = [ TR ; ...
              MG1 ; ...
              MG2 ];
HardGoals = [];
```

In this example, all the goals are designated as soft goals when the script is generated. Therefore, `HardGoals` is empty.

Finally, the script tunes the control system by calling `systune` on the `slTuner` interface using the tuning goals and options.

```
%% Tune the parameters with soft and hard goals
[CL1,fSoft,gHard,Info] = systune(CL0,SoftGoals,HardGoals,Options);
```

The script also includes an optional call to `viewSpec`, which displays graphical representations of the tuning goals to aid you in interpreting and validating the tuning results. Uncomment this line of code to generate the plots.

```
%% View tuning results
% viewSpec([SoftGoals;HardGoals],CL1,Info);
```

You can add calls to functions such `getIOTransfer` to make the script generate additional analysis plots.

Related Examples

- "Create and Configure slTuner Interface to Simulink Model" on page 6-178

- "Tune Control System at the Command Line" on page 6-215
- "Validate Tuned Control System at the Command Line" on page 6-219

Tuning Options

To specify options for the tuning algorithm, in the **Tuning** tab, click ⚙ **Tuning Options**. The Tuning Options dialog box lets you specify the following options:

Optimization

- **Use multiple starting points**

  Control System Tuner tunes by finding a local minimum of a gain minimization problem. When this option is unchecked, the software performs a single optimization run starting from the initial values of the tunable parameters. To increase the likelihood of finding parameter values that meet your design requirements, you can perform multiple optimization runs that begin from randomized parameter values.

  To do so, check this option, and enter a number of optimization runs to perform in the **Number of randomized starts** text box. Control System Tuner selects the best design that results from the multiple optimization runs.

- **Run multiple starts in parallel**

  Check this option enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

  - If `Automatically create a parallel pool` is selected in your "Parallel Computing Toolbox™ preferences", then the software starts a parallel pool using the settings in those preferences.

  - If `Automatically create a parallel pool` is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

  If `Automatically create a parallel pool` is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

  Using parallel processing requires Parallel Computing Toolbox software.

- **Stop when soft goal values less than**

Check this box to specify a custom target value for soft constraints. The optimization stops when the largest soft constraint value falls below this target value, assuming it is possible to find parameter values that achieve this result. When the box is unchecked, the software attempts to minimize the soft goals subject to the hard constraints.

- **Show report with**

  When this box is checked, Control System Tuner displays a tuning report each time you tune a model. Select the level of detail to display in the report.

  - `Final summary` — Displays tuning summary containing the best achieved value for the hard goals and soft goals. Also displays hard goal, soft goal, and number of iterations for each independent optimization run.

  - `Intermediate results` — In addition to the tuning summary, displays the results of each optimization subproblem.

    When you designate some tuning goals as hard goals, the software divides the optimization into subproblems. First, the software attempts to satisfy the hard goals. Then, it attempts to minimize the soft goals, subject to remaining in a parameter-space region in which the hard goals are satisfied. When you select `Intermediate results`, the report includes the results of each of these subproblems.

  - `Detailed progress` — Displays the result of every iteration in each optimization run.

Stabilization

- **Minimum decay rate**

  Specify the minimum decay rate for closed-loop poles in the tuned system.

  Constrains all closed-loop pole locations $|p|$ to satisfy `Re(p) < -MinDecay`. Adjust the minimum value if the optimization cannot meet the default minimum value, or if the default minimum value conflicts with other requirements. For specifying other constraints on the closed-loop pole locations, use "Poles Goal" on page 6-144.

  Default: 10^{-7}

Solver Parameters

- **Maximum iterations**

  Specify the maximum number of iterations in each optimization run, when the run does not converge to within tolerance.

  Default: 300

- **Termination tolerance for soft goals**

  Specify the relative tolerance for termination.

  The optimization terminates when the relative decrease in the soft constraint value decreases by less than this value over 10 consecutive iterations. Increasing this value speeds up termination. Decreasing the value yields tighter final values.

  Default: 0.001

- **Guess at best feasible value for soft goals**

  Specify an priori estimate of best soft constraint value.

  For problems that mix soft and hard constraints, providing a rough estimate of the optimal value of the soft constraints (subject to the hard constraints) helps to speed up the optimization.

  Default: 1

More About

Interpreting Tuning Results

The automated tuning software converts each soft and hard tuning goal into normalized values $f_i(x)$ and $g_j(x)$, respectively. Here, x is the vector of tunable parameters in the control system to tune.

The software then solves the minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system.

When you tune a control system, you obtain the following output:

· In Control System Tuner, the current design is updated to use the tuned parameters that best satisfy the minimization problem. By default, the current design is reflected in all tuning goal plots and response plots you have active in Control System Tuner.

Control System Tuner also displays a Tuning Report summarizing the best achieved values of $f_i(x)$ and $g_j(x)$.

The **Worst Value** entry reflects the largest of the optimized soft goal values. In other words, the software minimizes the $f_i(x)$ values as well as it can subject to the hard constraints. **Worst Value** highlights the largest of the minimized $f_i(x)$ values. The closer each value is to 1, the closer that requirement is to being satisfied.

- For command-line tuning, `systune` returns the control system model or `slTuner` interface with the tuned parameter values. `systune` also returns the best achieved values of each $f_i(x)$ and $g_j(x)$ as the vector-valued output arguments `fSoft` and `gHard`, respectively. See the `systune` reference page for more information.

For information about the functions $f_i(x)$ and $g_j(x)$ for each type of constraint, see the reference pages for each tuning goal.

The software uses the nonsmooth optimization algorithms described in [1].

The software computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] Apkarian, P. and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, No. 1, (2006), pp. 71–86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, Vol. 14, No, 4 (1990), pp. 287–293.

Related Examples

Create Response Plots in Control System Tuner

This example shows how to create response plots for analyzing system performance in Control System Tuner. Control System Tuner can generate many types of response plots in the time and frequency domains. You can view responses of SISO or MIMO transfer functions between inputs and outputs at any location in your model. Use response plots to validate the performance of your tuned control system.

This example creates response plots for analyzing the sample model `rct_helico`.

Choose Response Plot Type

In Control System Tuner, in the **Control System** tab, click ⊞ **New Plot**. Select the type of plot you want to create.

A new plot dialog box opens in which you specify the inputs and outputs of the portion of your control system whose response you want to plot. For example, select **New step** to create a step response plot from specified inputs to specified outputs of your system.
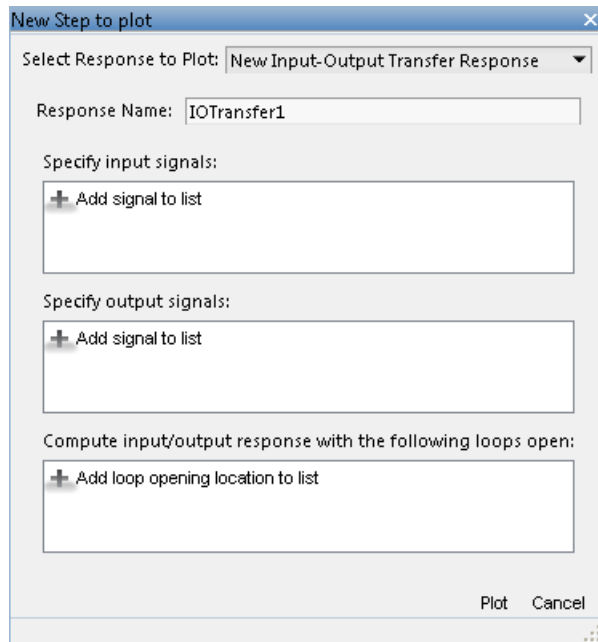
Specify Transfer Function

Choose which transfer function associated with the specified inputs and outputs you want to analyze.
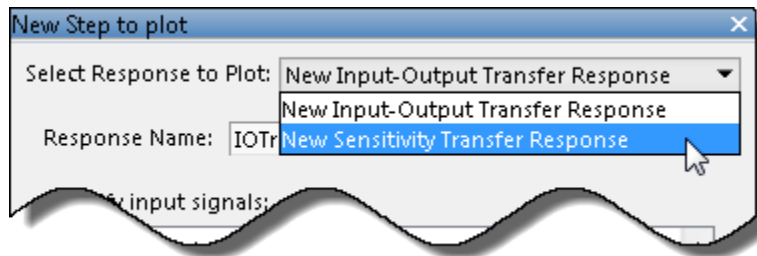
For most response plots types, the **Select Response to Plot** menu lets you choose one of the following transfer functions:

- `New Input-Output Transfer Response` — Transfer function between specified inputs and outputs, computed with loops open at any additionally specified loop-opening locations.
- `New Sensitivity Transfer Response` — Sensitivity function computed at the specified location and with loops open at any specified loop-opening locations.
- `New Open-Loop Response` — Open loop point-to-point transfer function computed at the specified location and with loops open at any additionally specified loop-opening locations.
- `Entire System Response` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations for the entire closed-loop control system.
- `Response of Tuned Block` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations of tuned blocks.

Name the Response

Type a name for the response in the **Response Name** text box. Once you have specified signal locations defining the response, Control System Tuner stores the response under this name. When you create additional new response plots, the response appears by this name in **Select Response to Plot** menu.

Choose Signal Locations for Evaluating System Response

Specify the signal locations in your control system at which to evaluate the selected response. For example, the step response plot displays the response of the system at one

or more output locations to a unit step applied at one or more input locations. Use the **Specify input signals** and **Specify output signals** sections of the dialog box to specify these locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)

Under **Specify input signals**, click ✚ **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.
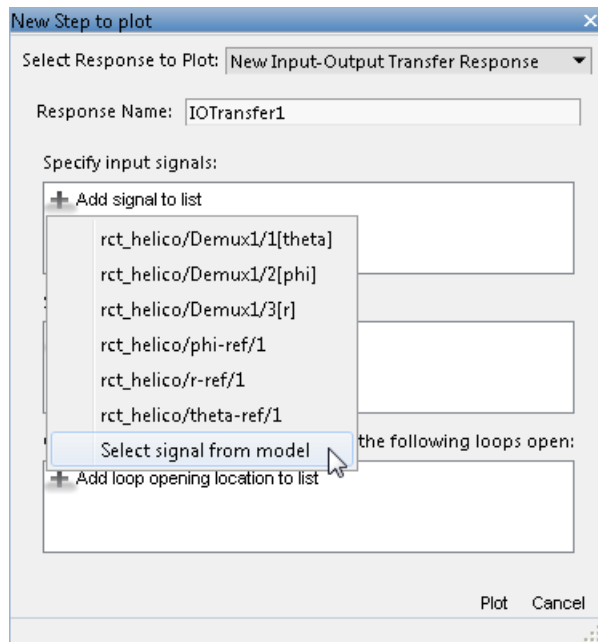
The **Add signal from the model** dialog box contains a list of signals that are currently selected in the Simulink model. If the signal you want to designate is listed, select it and click **Add signal**.

If the signal you want is not listed, select the signal in the Simulink model editor. Then, return to the **Add signal from the model** dialog box and click **Add signal**.
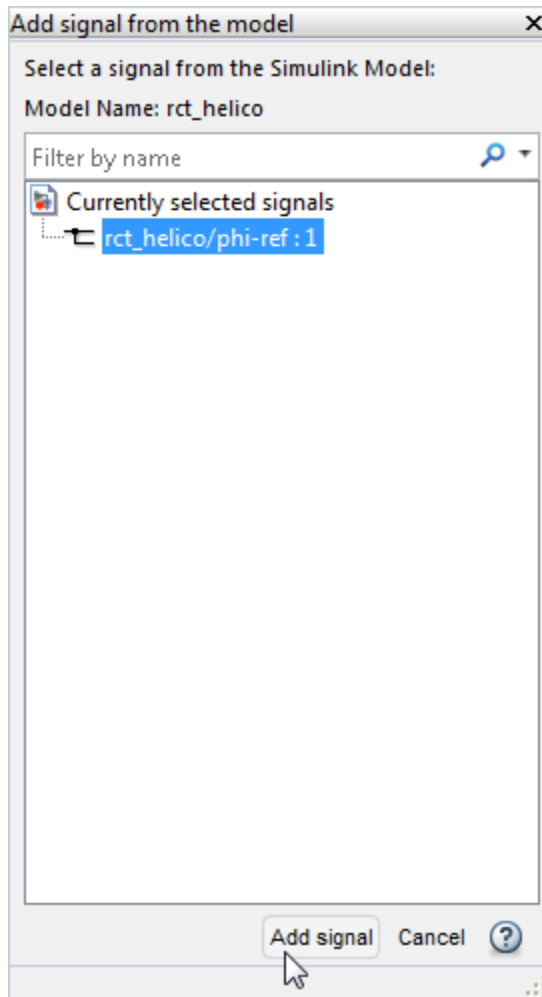
The signal you selected now appears in the list of step-response inputs.

Click ✚ **Add signal to list** again to add an additional signal to the step-response inputs list, if you want to specify a MIMO response.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration plots the MIMO response

to a step input applied at `theta-ref` and `phi-ref` and measured at `theta` and `phi` in the Simulink model `rct_helico`.
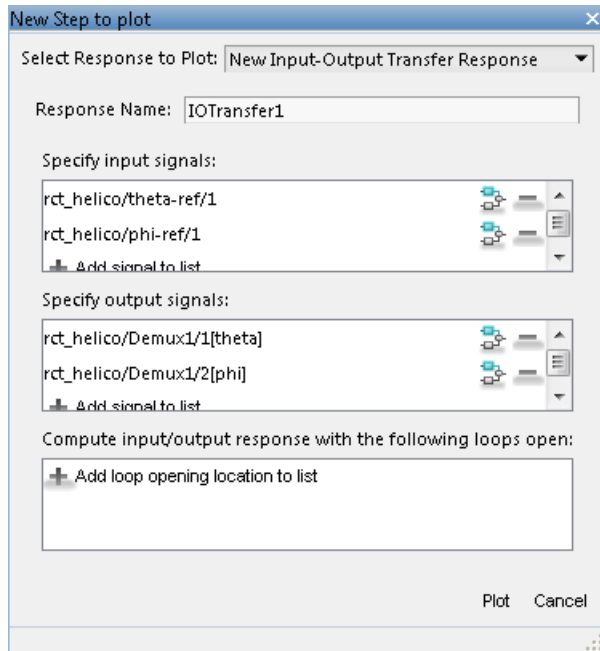
Tip To highlight any selected signal in the Simulink model, click ⬚. To remove a signal from the input or output list, click ▬.

Specify Loop Openings

You can evaluate most system responses with loops open at one or more locations in the control system. Click ╋**Add loop opening location to list** to specify such locations for the response.

Store and Plot the Response

When you have finished specifying the response, click **Plot** in the new plot dialog box. The new response appears in the **Responses** section of the Data Browser. A new figure

opens displaying the response plot. When you tune your control system, you can refer to this figure to evaluate the performance of the tuned system.
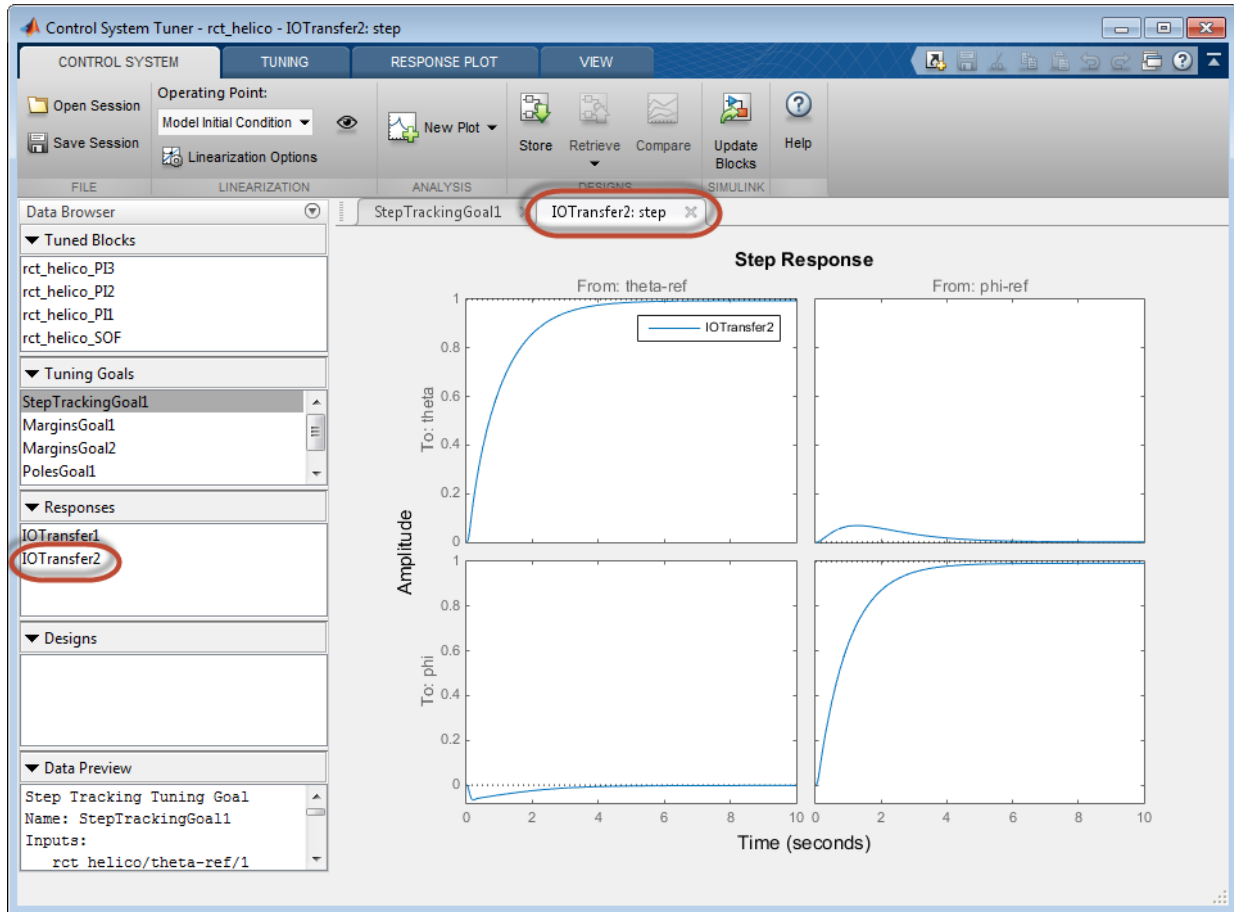
Tip To edit the specifications of the response, double-click the response in the Data Browser. Any plots using that response update to reflect the edited response.

View response characteristics such as rise-times or peak values by right-clicking on the plot. Other options for managing and organizing multiple plots are available in the **View** tab.
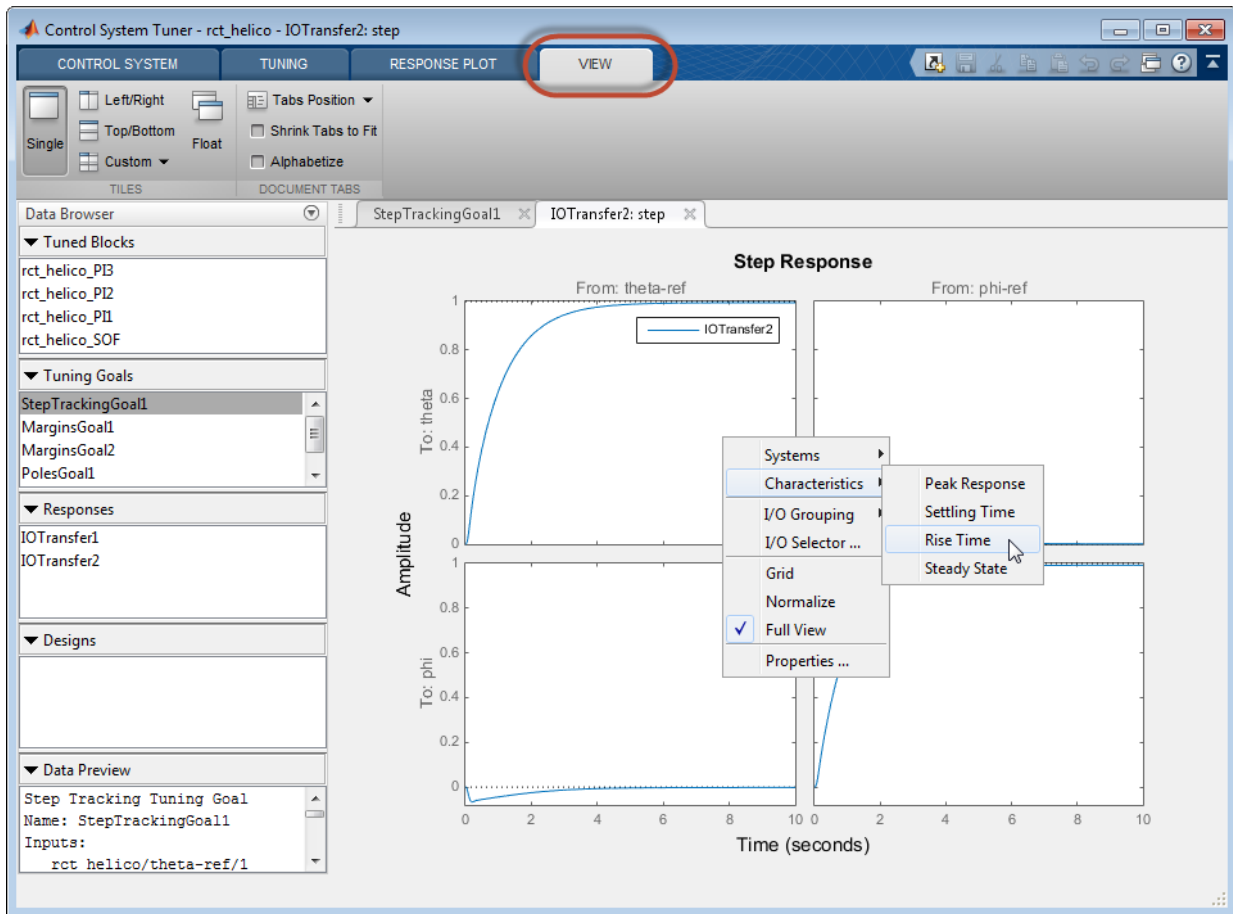
Related Examples

- "Compare Performance of Multiple Tuned Controllers" on page 6-172
- "Examine Tuned Controller Parameters in Control System Tuner" on page 6-170
- "Validate Tuned Controller in Simulink" on page 6-177

Examine Tuned Controller Parameters in Control System Tuner

After you tune your control system, Control System Tuner gives you two ways to view the current values of the tuned block parameters:

- In the Data Browser, in the **Tuned Blocks** area, select the block whose parameters you want to view. A text summary of the block and its current parameter values appears in the Data Browser in the **Data Preview** area.

- In the Data Browser, in the **Tuned Blocks** area, double-click the block whose parameters you want to view. The **Tuned Block Editor** window opens, displaying

  the current values of the parameters. For array-valued parameters, click to open a variable editor displaying values in the array.

Related Examples

- "View and Change Block Parameterization in Control System Tuner" on page 6-33

More About

- "Tuned Block Editor" on page 6-36

Compare Performance of Multiple Tuned Controllers

This example shows how to compare the performance of a control system tuned with two different sets of tuning goals. Such comparison is useful, for example, to see the effect on performance of changing a tuning goal from hard goal to soft goal. Comparing performance is also useful to see the effect of adding an additional tuning goal when an initial design fails to satisfy all your performance requirements either in the linearized system or when validated against a full nonlinear model.

This example compares tuning results for the sample model `rct_linact`.

Store First Design

After tuning a control system with a first set of design requirements, store the design in Control System Tuner.

In the **Control System** tab, click ⬛ **Store**. The stored design appears in the Data Browser in the **Designs** area.
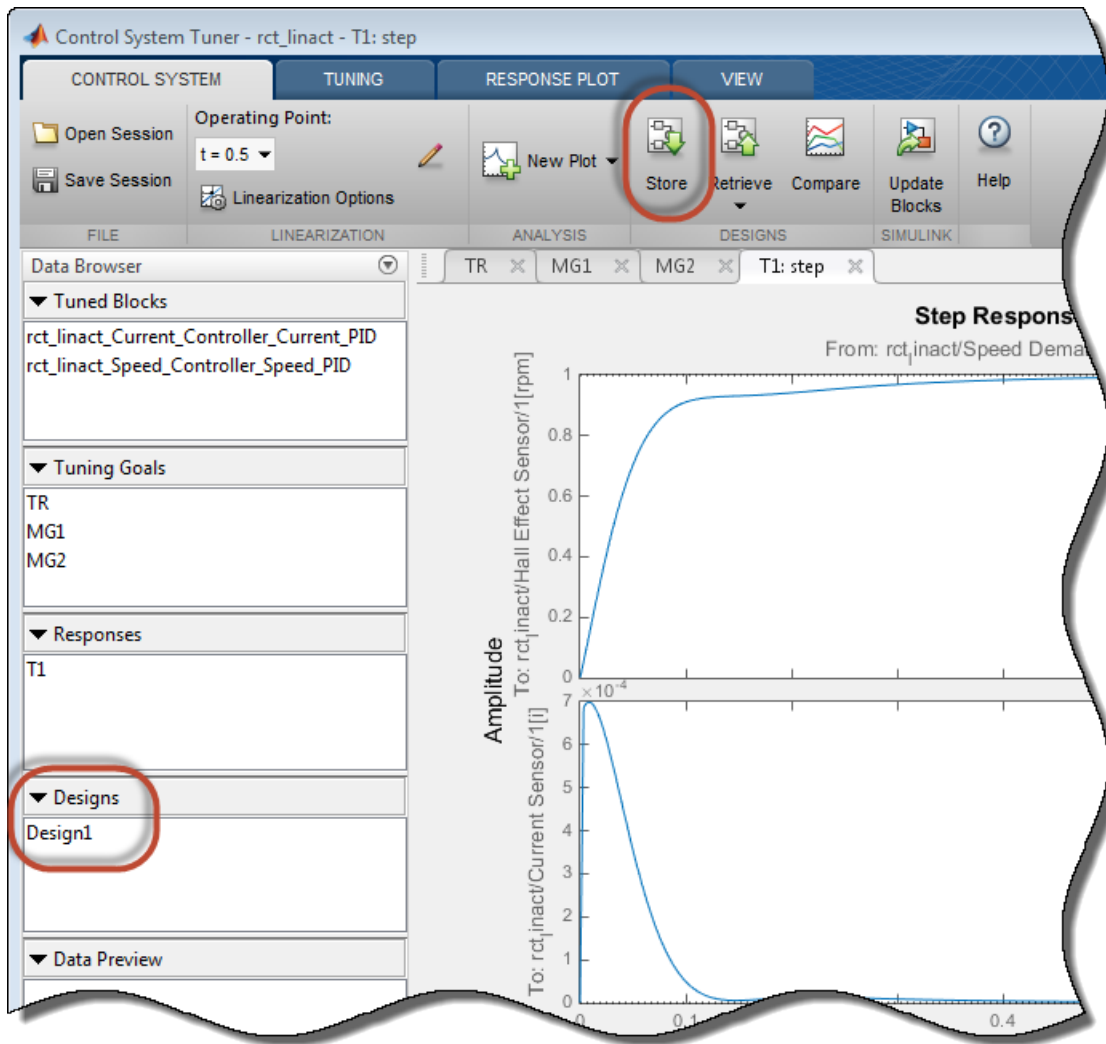
Change the name of the stored design, if desired, by right-clicking on the data browser entry.

Compute New Design

In the **Tuning** tab, make any desired changes to the tuning goals for the second design. For example, add new tuning goals or edit existing tuning goals to change specifications.

Or, in ![WW] **Manage Goals**, change which tuning goals are active and which are designated hard constraints or soft requirements.

When you are ready, retune the control system with the new set of tuning goals. Click

![Tune icon] **Tune**. Control System Tuner updates the current design (the current set of controller parameters) with the new tuned design. All existing plots, which by default show the current design, are updated to reflect the new current design.

Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design. In the **Control System** tab, click ![icon] **Compare**. The **Compare Designs** dialog box opens.
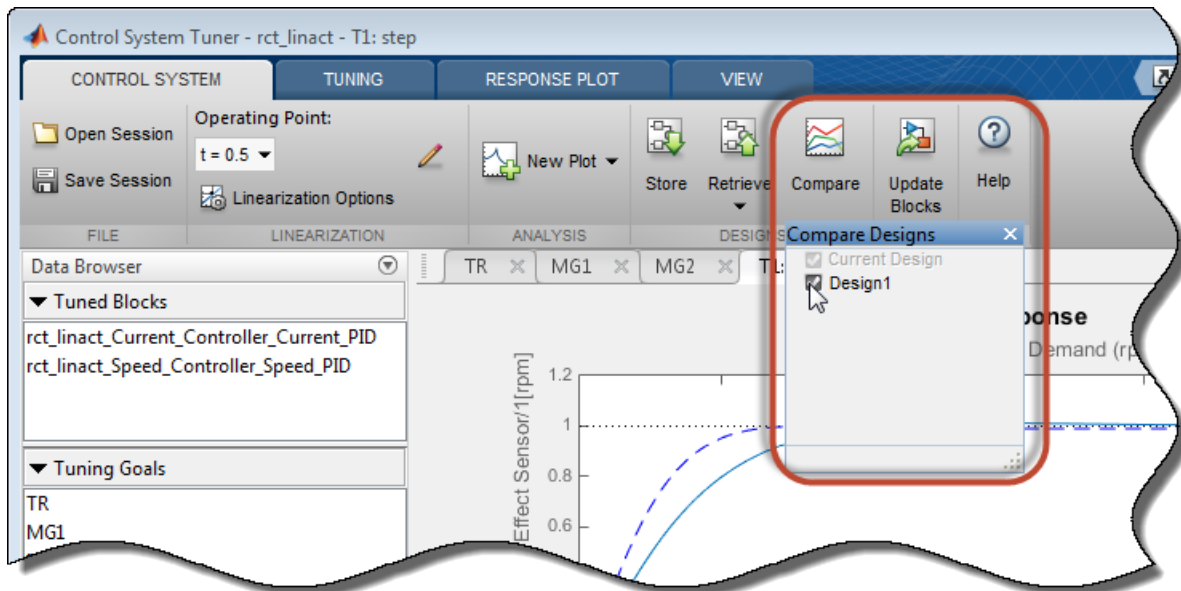
In the **Compare Designs** dialog box, the current design is checked by default. Check the box for the design you want to compare to the current design. All response plots and

tuning goal plots update to reflect the checked designs. The solid trace corresponds to the current design. Other designs are identified by name in the plot legend.
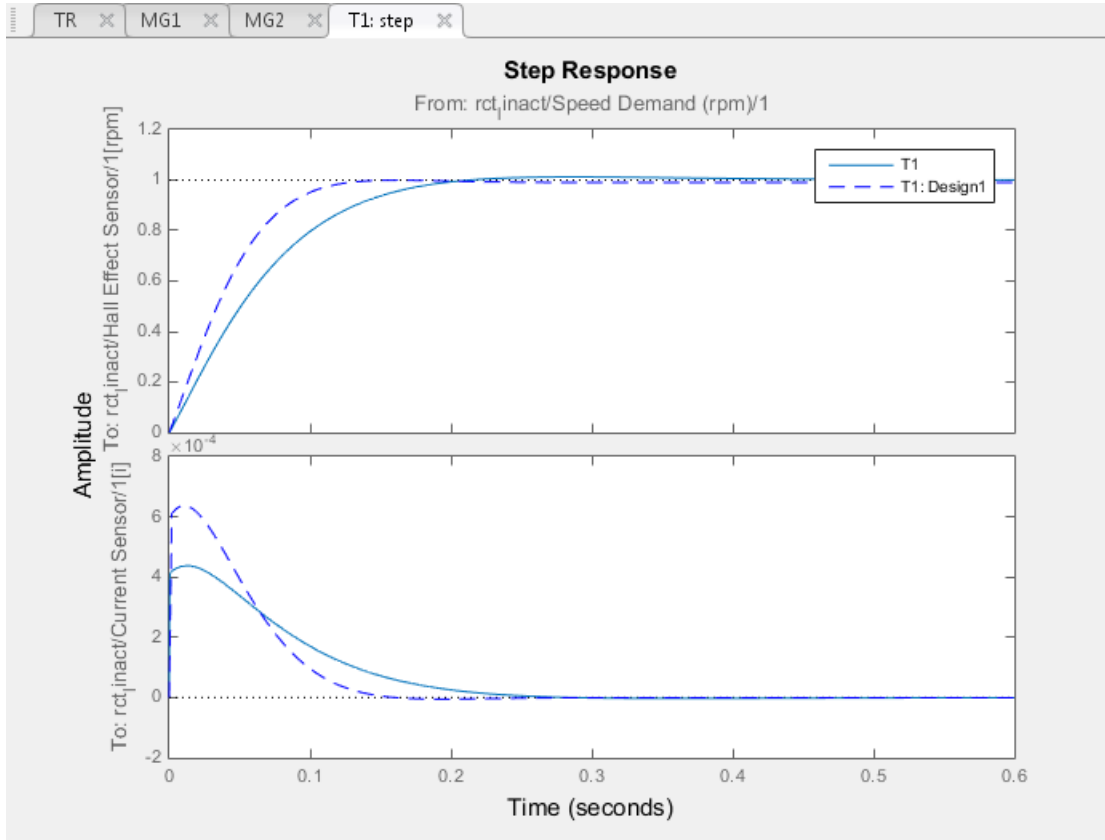
Use the same procedure save and compare as many designs as you need.

Restore Previously Saved Design

Under some conditions, it is useful to restore the tuned parameter values from a previously saved design as the current design. For example, clicking ⏹ **Update Blocks** writes the current parameter values to the Simulink model. If you decide to test a stored controller design in your full nonlinear model, you must first restore those stored values as the current design.

To do so, click ![Retrieve icon] **Retrieve**. Select the stored design that you want to make the current design.
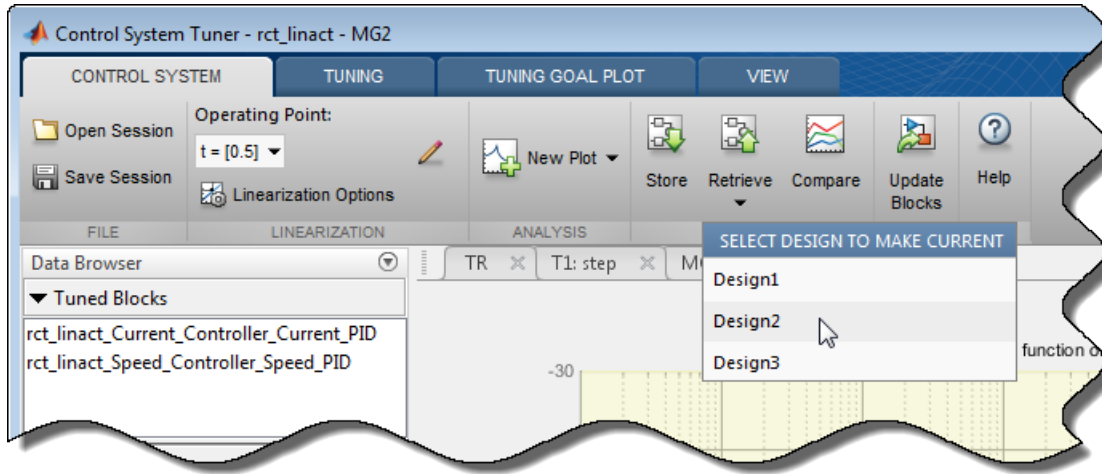
Related Examples

- "Create Response Plots in Control System Tuner" on page 6-163

Validate Tuned Controller in Simulink

Because Control System Tuner designs for a linearization of your Simulink model, tuned block parameters must be validated by simulating the full nonlinear model, even if the tuned system meets all your tuning goals in Control System Tuner.

To write tuned block parameters to a Simulink model, in the **Control System** tab, click ![icon]**Update Blocks**.
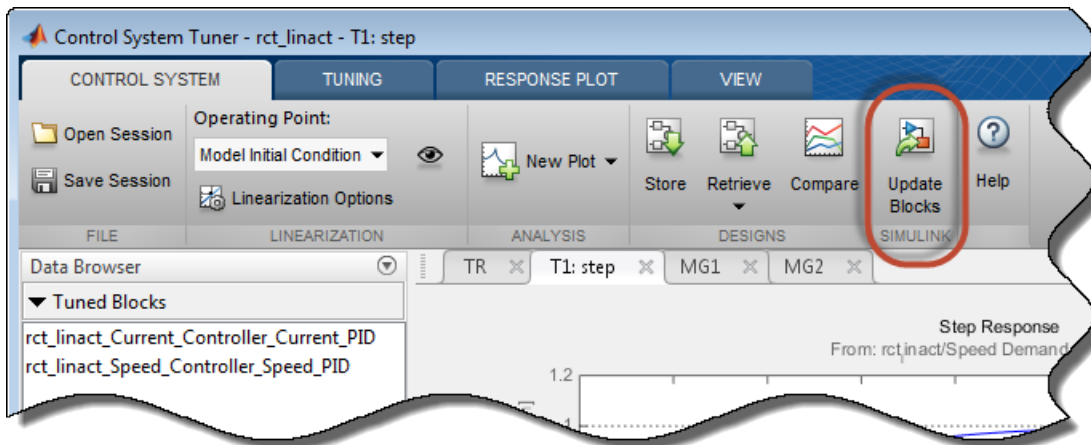
Control System Tuner transfers the current values of the tuned block parameters to the corresponding blocks in the Simulink model. Simulate the model to evaluate model performance using the tuned values.

Tip If you tune the Simulink model at an operating point other than the model initial condition, you might want to initialize the model at the same operating point before simulating. See "Simulate Simulink Model at Specific Operating Point" in the Simulink Control Design documentation.

To update Simulink model with parameter values from a previous design stored in Control System Tuner, click ![icon]**Retrieve** and select the stored design that you want to make the current design. Then click ![icon]**Update Blocks**.

Create and Configure slTuner Interface to Simulink Model

This example shows how to create and configure an `slTuner` interface for a Simulink® model. The `slTuner` interface parameterizes blocks in your model that you designate as tunable and and allows you to tune them using `systune`. The `slTuner` interface generates a linearization of your Simulink model, and also allows you to extract linearized system responses for analysis and validation of the tuned control system.

For this example, create and configure an `slTuner` interface for tuning the Simulink model `rct_helico`, a multiloop controller for a rotorcraft. Open the model.

```
open_system('rct_helico');
```
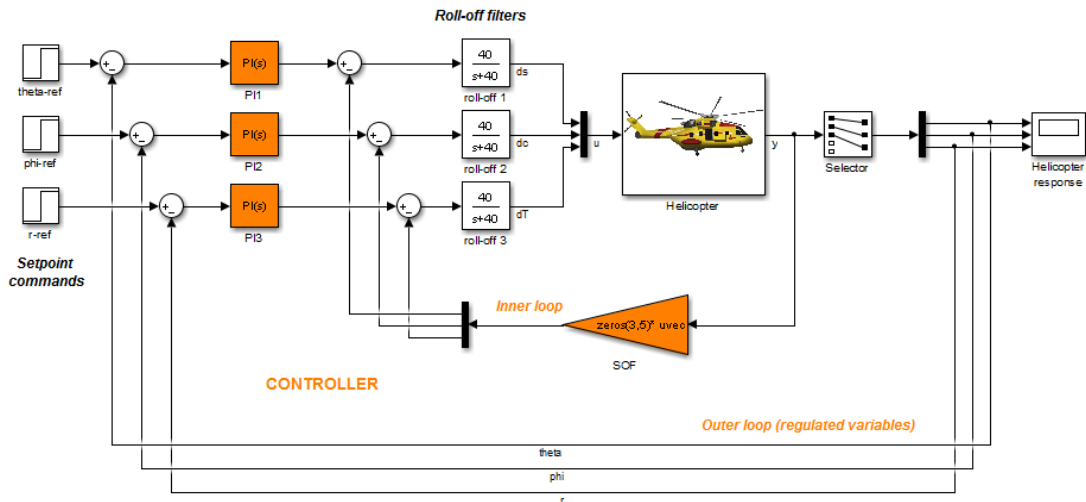

The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance.

Suppose that you want to tune this model to meet the following control objectives:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, specified rise times, 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).

The `systune` command can jointly tune the controller blocks `SOF` and the PI controllers) to meet these design requirements. The `slTuner` interface sets up this tuning task.

Create the `slTuner` interface.

```
ST0 = slTuner('rct_helico',{'PI1','PI2','PI3','SOF'});
```

This command initializes the `slTuner` interface with the three PI controllers and the `SOF` block designated as tunable. Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model.

To configure the `slTuner` interface, designate as analysis points any signal locations of relevance to your design requirements. First, add the outputs and reference inputs for the tracking requirements.

```
addPoint(ST0,{'theta-ref','theta','phi-ref','phi','r-ref','r'});
```

When you create a `TuningGoal.Tracking` object that captures the tracking requirement, this object references the same signals.

Configure the `slTuner` interface for the stability margin requirements. Designate as analysis points the plant inputs and outputs (control and measurement signals) where the stability margins are measured.

```
addPoint(ST0,{'u','y'});
```

Display a summary of the `slTuner` interface configuration in the command window.

```
ST0
```

```
slTuner tuning interface for "rct_helico":

4 Tuned blocks: (Read-only TunedBlocks property)
-------------------------
Block 1: rct_helico/PI1
Block 2: rct_helico/PI2
Block 3: rct_helico/PI3
Block 4: rct_helico/SOF
```

```
8 Analysis points:
-------------------------
Point 1: Port 1 of rct_helico/theta-ref
Point 2: Signal "theta", located at port 1 of rct_helico/Demux1
Point 3: Port 1 of rct_helico/phi-ref
Point 4: Signal "phi", located at port 2 of rct_helico/Demux1
Point 5: Port 1 of rct_helico/r-ref
Point 6: Signal "r", located at port 3 of rct_helico/Demux1
Point 7: Signal "u", located at port 1 of rct_helico/Mux3
Point 8: Signal "y", located at port 1 of rct_helico/Helicopter

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

In the command window, click on any highlighted signal to see its location in the Simulink model.

In addition to specifying design requirements, you can use analysis points for extracting system responses. For example, extract and plot the step responses between the reference signals and `'theta'`, `'phi'`, and `'r'`.

```
T0 = getIOTransfer(ST0,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
stepplot(T0,1)
```
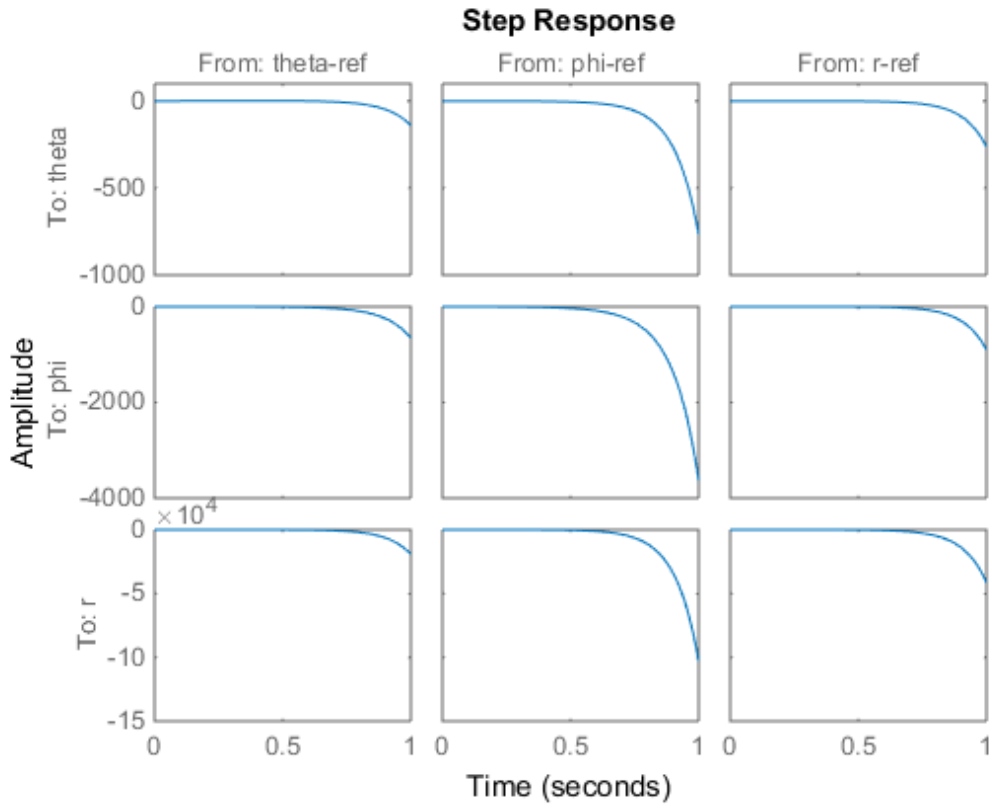
All the step responses are unstable, including the cross-couplings, because this model has not yet been tuned.

After you tune the model, you can similarly use the designated analysis points to extract system responses for validating the tuned system. If you want to examine system responses at locations that are not needed to specify design requirements, add these locations to the slTuner interface as well. For example, plot the sensitivity function measured at the output of the block roll-off 2.

```
addPoint(STO,'dc')
dcSO = getSensitivity(STO,'dc');
bodeplot(dcSO)
```

Bode Diagram

From: dc To: dc
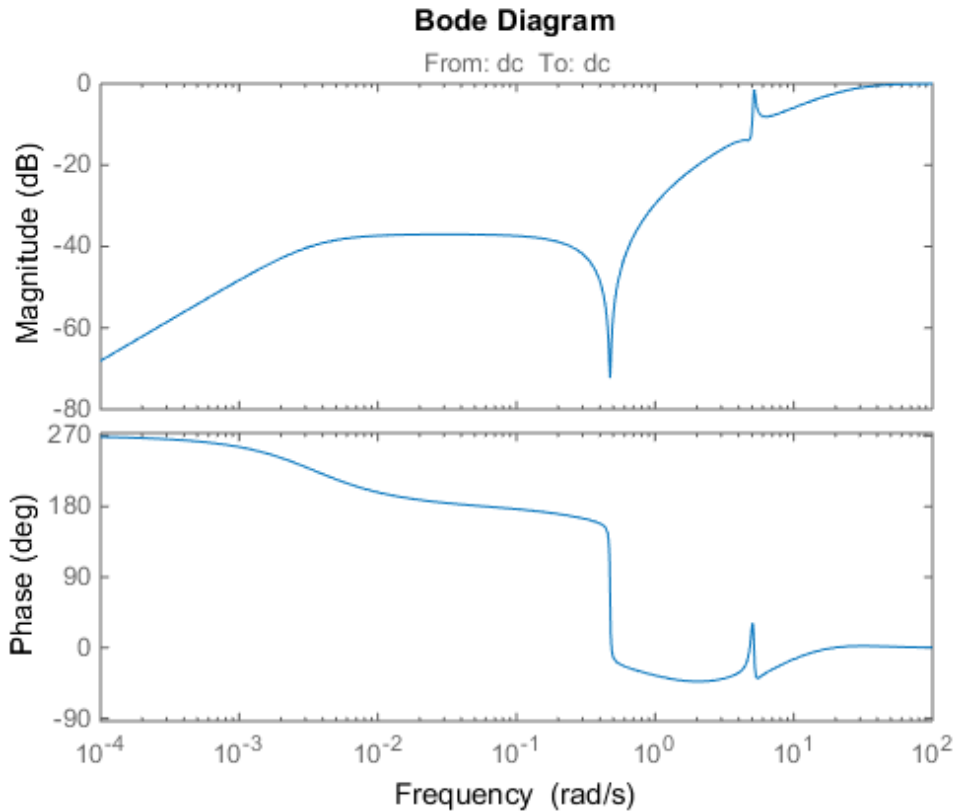
Suppose you want to change the parameterization of tunable blocks in the slTuner interface. For example, suppose that after tuning the model, you want to test whether changing from PI to PID controllers yields improved results. Change the parameterization of the three PI controllers to PID controllers.

```
PID0 = pid(0,0.001,0.001,.01);  % initial value for PID controllers
PID1 = ltiblock.pid('C1',PID0);
PID2 = ltiblock.pid('C2',PID0);
PID3 = ltiblock.pid('C3',PID0);

setBlockParam(ST0,'PI1',PID1);
setBlockParam(ST0,'PI2',PID2);
setBlockParam(ST0,'PI3',PID3);
```

After you configure the `slTuner` interface to your Simulink model, you can create tuning goals and tune the model using `systune` or `looptune`.

See Also

`addBlock` | `addPoint` | `getIOTransfer` | `getSensitivity` | `setBlockParam` | `slTuner`

Related Examples

- "Multi-Loop Control of a Helicopter"
- "Control of a Linear Electric Actuator"

More About

- "Managing Signals in Control System Analysis and Design"
- "Tuning Goals"

Time-Domain Specifications

This example gives a tour of available time-domain requirements for control system tuning with `systune` or `looptune`.

Background

The `systune` and `looptune` commands 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.

Step Command Following

The `TuningGoal.StepTracking` requirement specifies how the tuned closed-loop system should respond to a step input. You can specify the desired response either in terms of first- or second-order characteristics, or as an explicit reference model. This requirement is satisfied when the relative gap between the actual and desired responses is small enough in the least-squares sense. For example,

```
R1 = TuningGoal.StepTracking('r','y',0.5);
```

stipulates that the closed-loop response from `r` to `y` should behave like a first-order system with time constant 0.5, while

```
R2 = TuningGoal.StepTracking('r','y',zpk(2,[-1 -2],-1));
```

specifies a second-order, non-minimum-phase behavior. Use `viewSpec` to visualize the desired response.

```
viewSpec(R2)
```

Requirement 1: Target response to step command
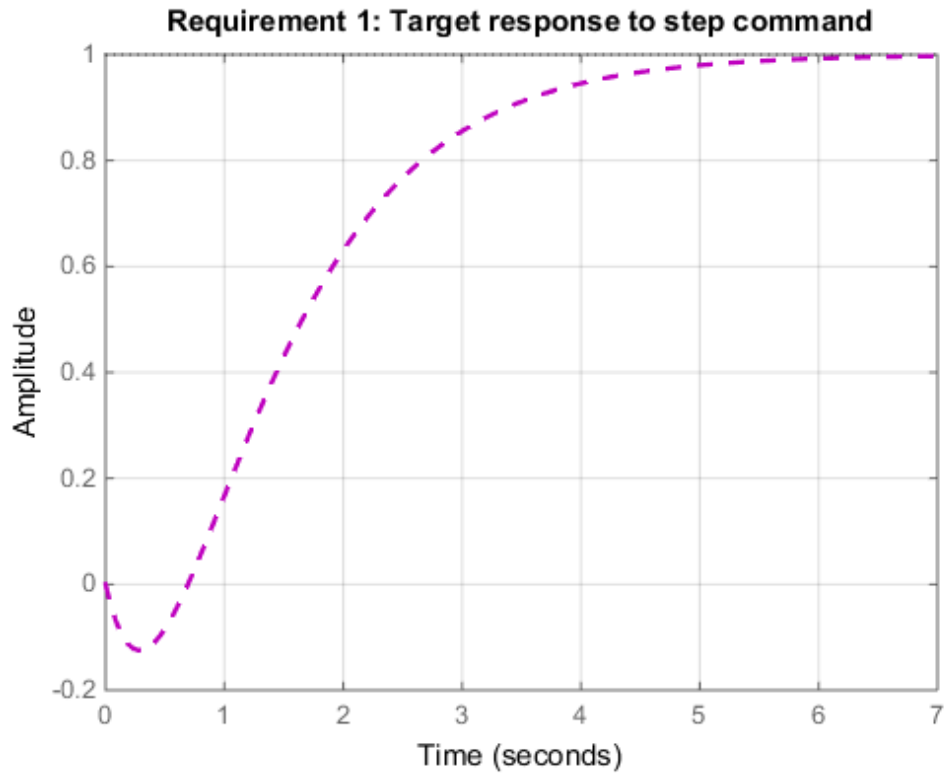
This requirement can be used to tune both SISO and MIMO step responses. In the MIMO case, the requirement ensures that each output tracks the corresponding input with minimum cross-couplings.

Step Disturbance Rejection

The `TuningGoal.StepRejection` requirement specifies how the tuned closed-loop system should respond to a step disturbance. You can specify worst-case values for the response amplitude, settling time, and damping of oscillations. For example,

```
R1 = TuningGoal.StepRejection('d','y',0.3,2,0.5);
```

limits the amplitude of $y(t)$ to 0.3, the settling time to 2 time units, and the damping ratio to a minimum of 0.5. Use `viewSpec` to see the corresponding time response.

```
viewSpec(R1)
```
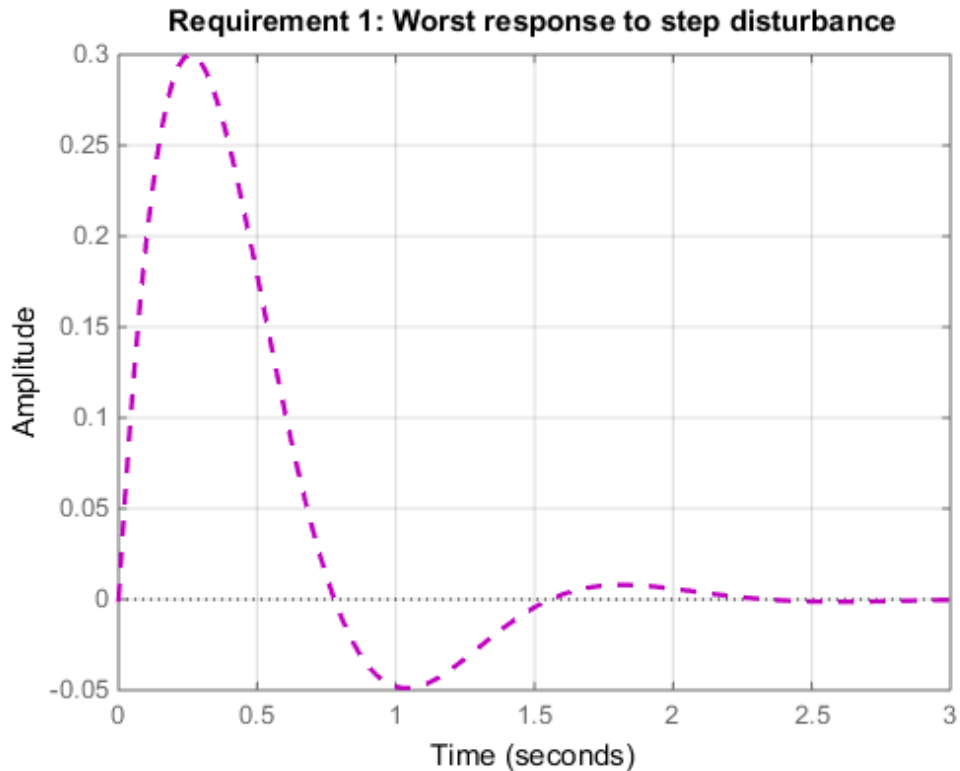

You can also use a "reference model" to specify the desired response. Note that the actual and specified responses may differ substantially when better disturbance rejection is possible. Use the `TuningGoal.Transient` requirement when a close match is desired. For best results, adjust the gain of the reference model so that the actual and specified responses have similar peak amplitudes (see `TuningGoal.StepRejection` documentation for details).

Transient Response Matching

The `TuningGoal.Transient` requirement specifies the transient response for a specific input signal. This is a generalization of the `TuningGoal.StepTracking` requirement. For example,

```
R1 = TuningGoal.Transient('r','y',tf(1,[1 1 1]),'impulse');
```

requires that the tuned response from r to y look like the impulse response of the reference model $1/(s^2 + s + 1)$.

```
viewSpec(R1)
```
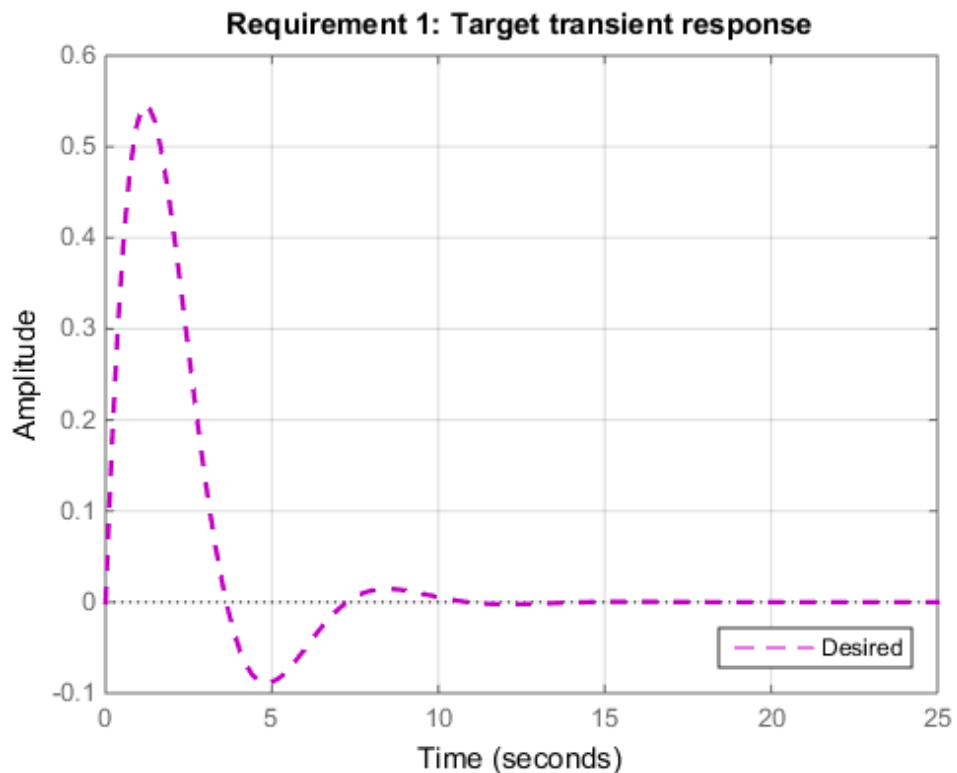

The input signal can be an impulse, a step, a ramp, or a more general signal modeled as the impulse response of some input shaping filter. For example, a sine wave with frequency ω_0 can be modeled as the impulse response of $\omega_0^2/(s^2 + \omega_0^2)$.

```
w0 = 2;
```

```
F = tf(w0^2,[1 0 w0^2]);  % input shaping filter
R2 = TuningGoal.Transient('r','y',tf(1,[1 1 1]),F);
viewSpec(R2)
```
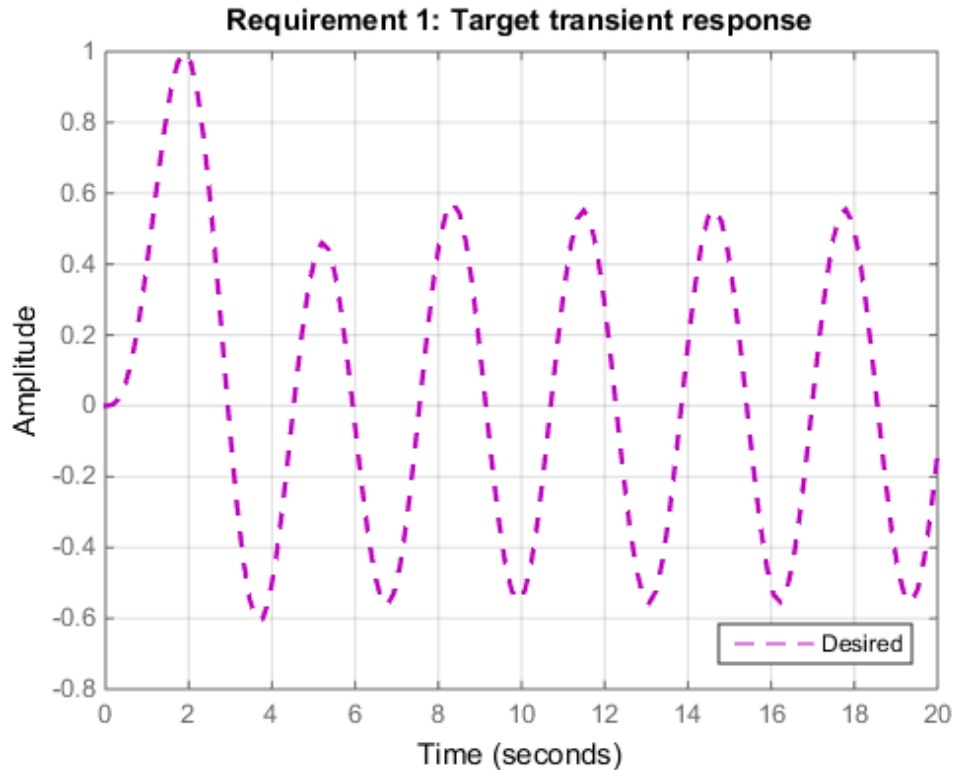

LQG Design

Use the `TuningGoal.LQG` requirement to create a linear-quadratic-Gaussian objective for tuning the control system parameters. This objective is applicable to any control structure, not just the classical observer structure of LQG control. For example, consider the simple PID loop of Figure 2 where d and n are unit-variance disturbance and noise inputs, and S_d and S_n are lowpass and highpass filters that model the disturbance and noise spectral contents.
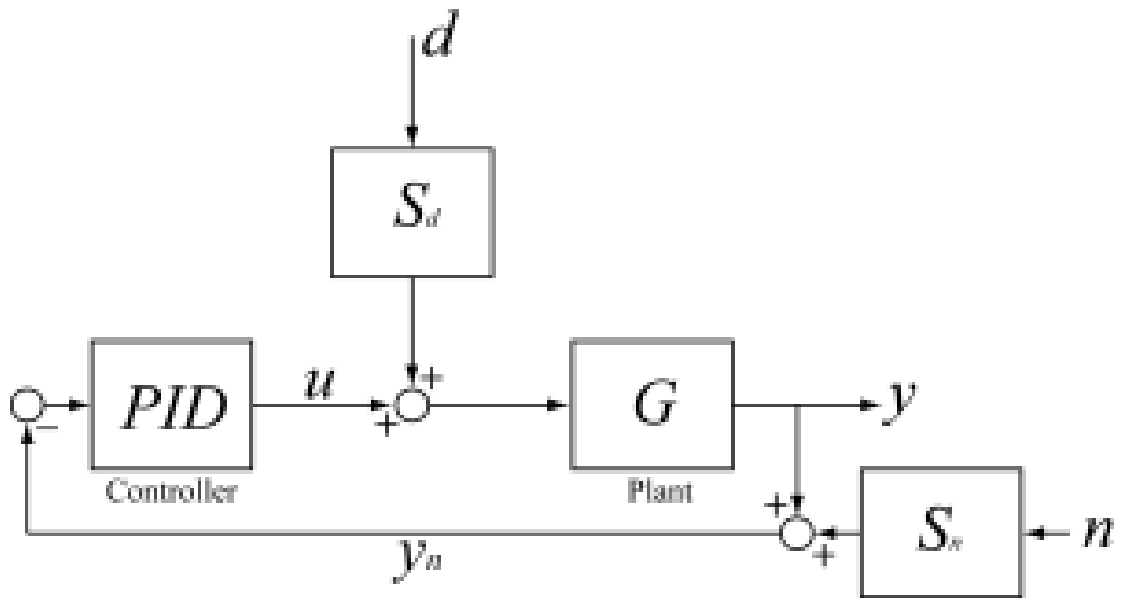
Figure 2: Regulation loop.

To regulate y around zero, you can use the following LQG criterion:

$$J = lim_{T \to \infty} E \left(\frac{1}{T} \int_0^T (y^2(t) + 0.05u^2)dt \right)$$

The first term in the integral penalizes the deviation of $y(t)$ from zero, and the second term penalizes the control effort. Using `systune`, you can tune the PID controller to minimize the cost J. To do this, use the LQG requirement

```
Qyu = diag([1 0.05]);  % weighting of y^2 and u^2
R4 = TuningGoal.LQG({'d','n'},{'y','u'},1,Qyu);
```

Frequency-Domain Specifications

This example gives a tour of available frequency-domain requirements for control system tuning with `systune` or `looptune`.

Background

The `systune` and `looptune` commands 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.

Gain Limit

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

```
s = tf('s');
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)
```

Requirement 1: Maximum gain as a function of frequency
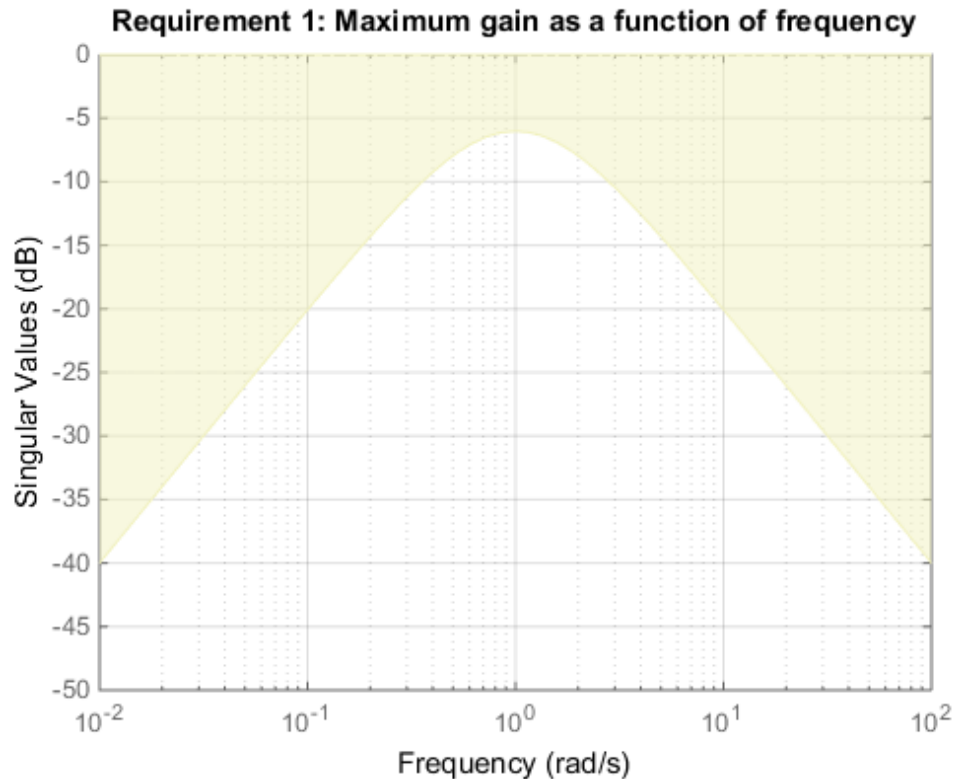
It is often convenient to just sketch the asymptotes of the desired gain profile. For example, instead of the transfer function $s/(s+1)^2$, we could just specify gain values of 0.01,1,0.01 at the frequencies 0.01,1,100, the point (1,1) being the breakpoint of the two asymptotes s and $1/s$.

```
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')
```
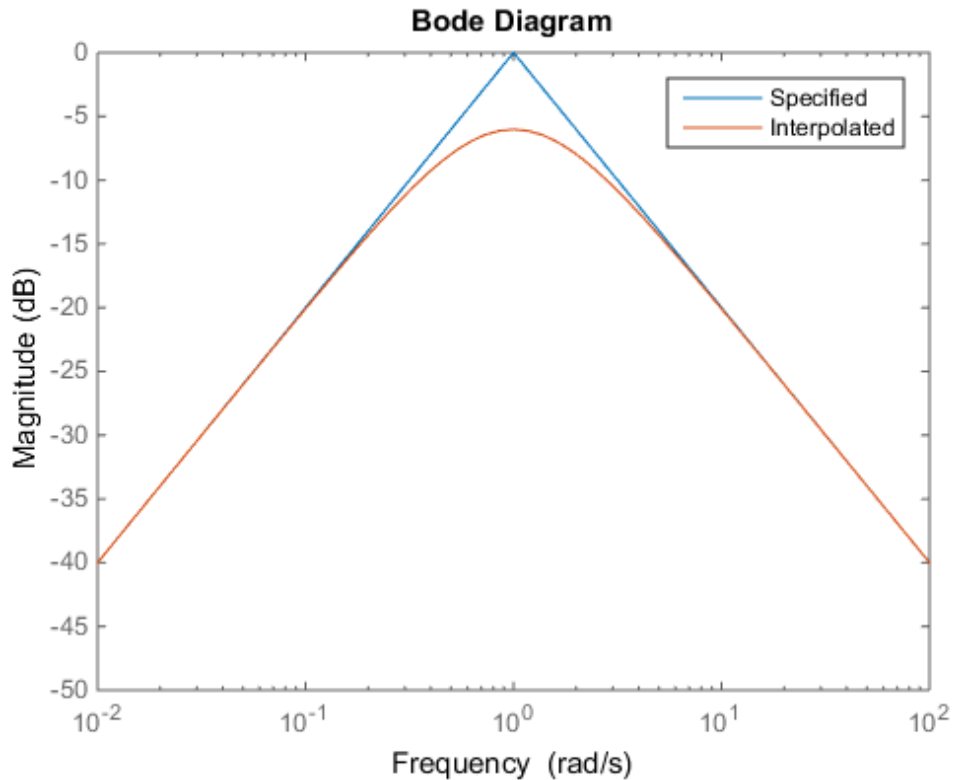
Variance Amplification

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.

Reference Tracking and Overshoot Reduction

The `TuningGoal.Tracking` requirement enforces reference tracking and loop decoupling objectives in the frequency domain. 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)
```

Requirement 1: Tracking error as a function of frequency
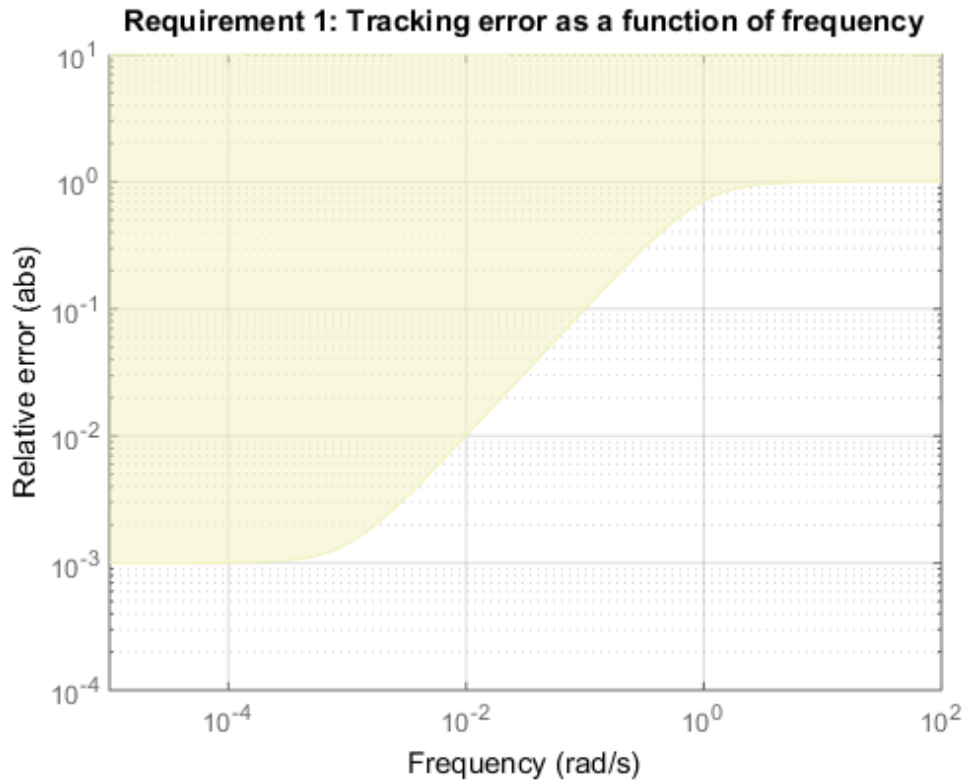
If the response has excessive overshoot, use the `TuningGoal.Overshoot` requirement in conjunction with the `TuningGoal.Tracking` requirement. For example, you can limit the overshoot from `r` to `y` to 10% using

```
R3 = TuningGoal.Overshoot('r','y',10);
```

Disturbance Rejection

In feedback loops such as the one shown in Figure 1, the open- and closed-loop responses from disturbance d to output y 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, the ratio between the open- and closed-loop sensitivities to the disturbance. Its reciprocal $S = 1/(1 + L)$ is the sensitivity at the disturbance input.
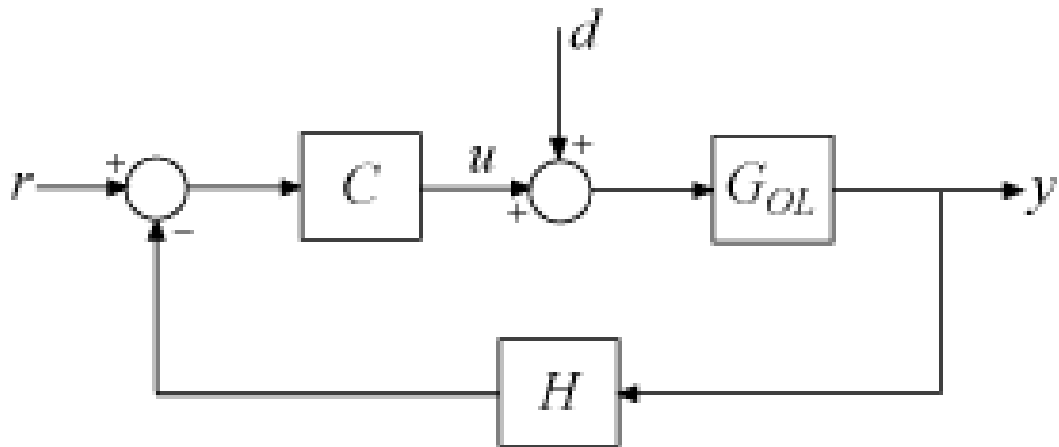
Figure 1: Sample feedback loop.

The `TuningGoal.Rejection` requirement specifies the disturbance attenuation as a function of frequency. The attenuation factor is greater than one inside the control bandwidth 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)
```

Requirement 1: Disturbance attenuation as a function of frequency
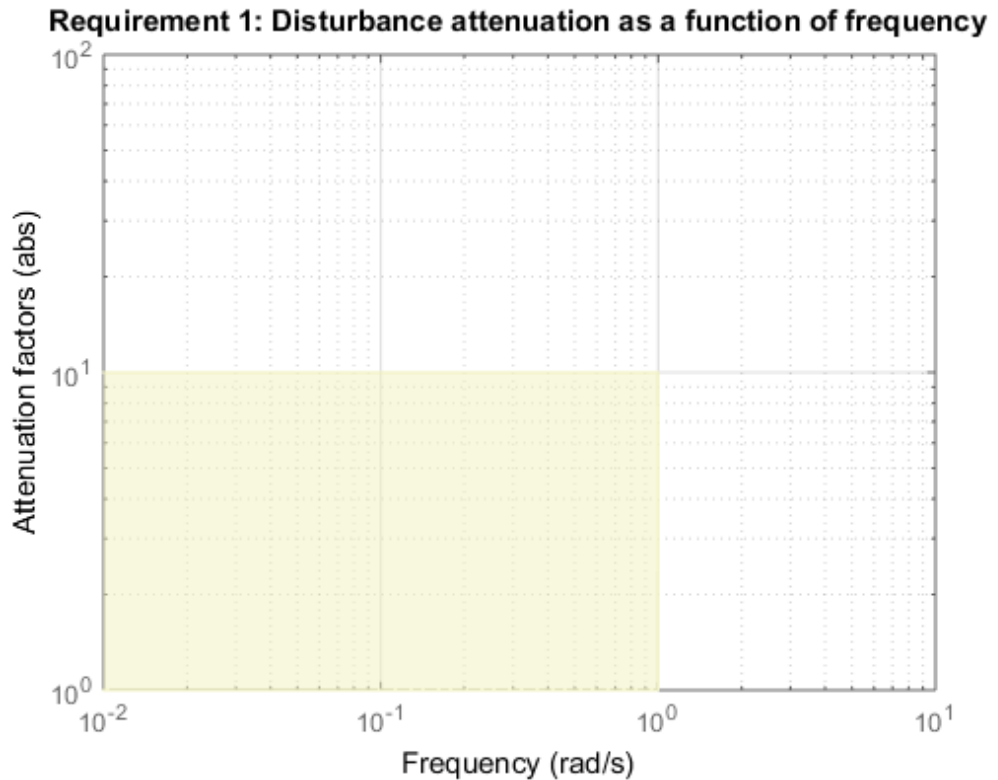
More generally, you can specify a frequency-dependent attenuation profile, for example

```
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)
```

Requirement 1: Disturbance attenuation as a function of frequency
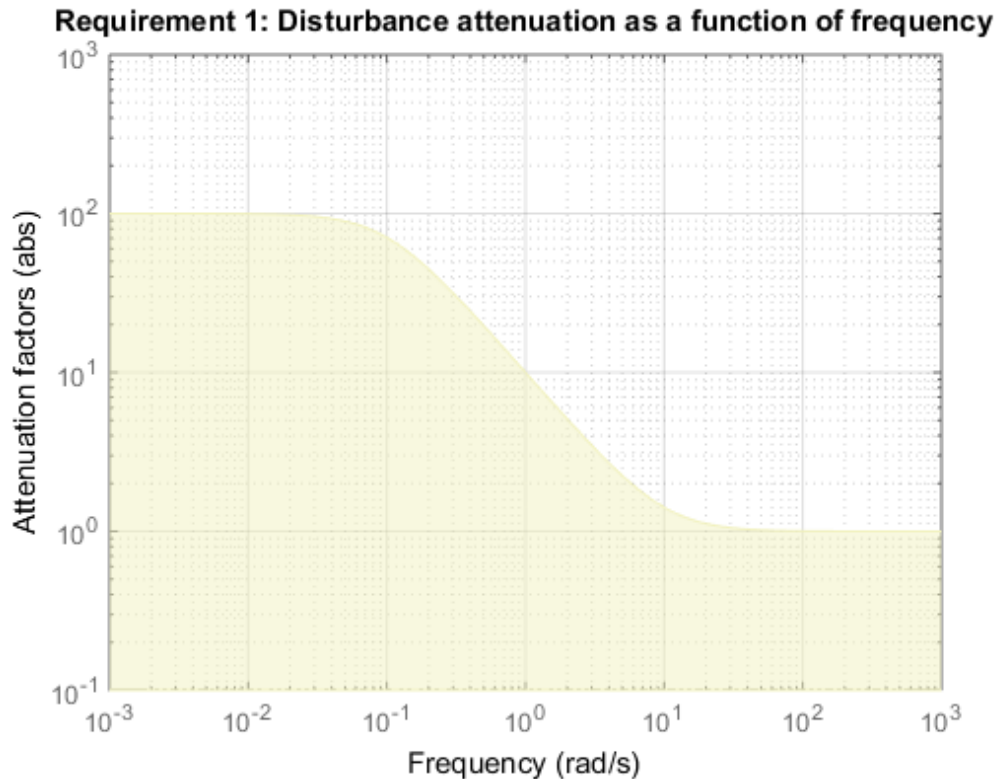
Instead of specifying the minimum attenuation, you can use the `TuningGoal.Sensitivity` requirement to specify the maximum sensitivity, that is, the maximum gain of $S = 1/(1 + L)$. For example,

```
R3 = TuningGoal.Sensitivity('u',(s+0.1)/(s+10));
```

is equivalent to the rejection requirement `R2` above. The sensitivity increases from 0.01 (1%) below 0.1 rad/s to 1 (100%) above 10 rad/s.

```
viewSpec(R3)
```
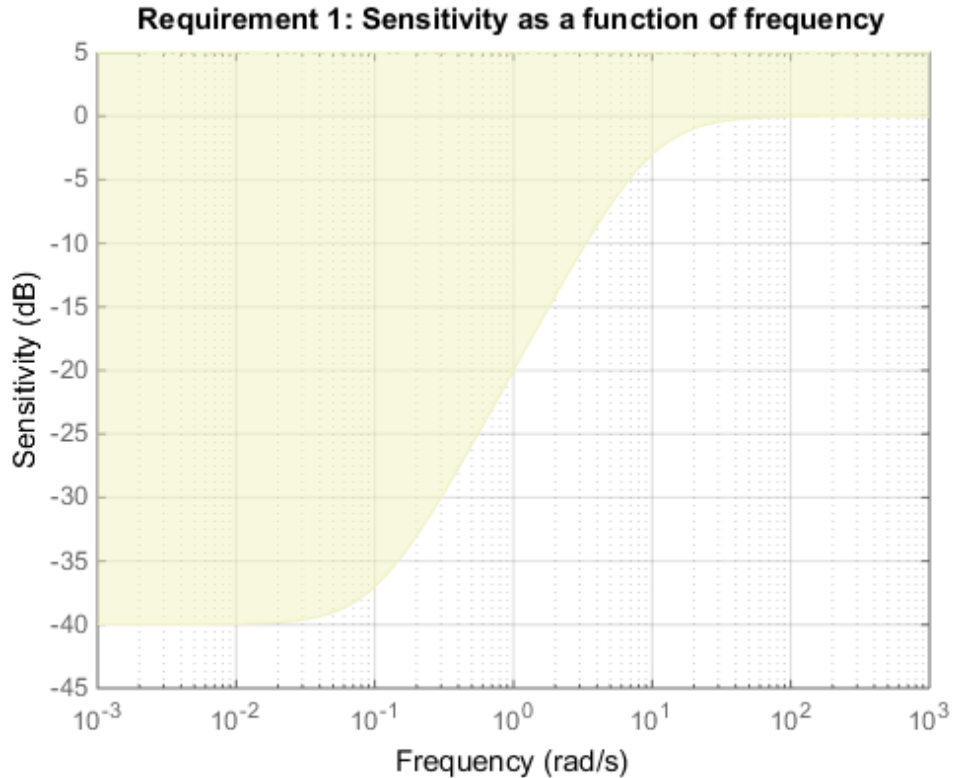
Frequency-Weighted Gain and Variance

The `TuningGoal.WeightedGain` and `TuningGoal.WeightedVariance` requirements are generalizations of the `TuningGoal.Gain` and `TuningGoal.Variance` requirements. These requirements constrain the H_∞ 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

$$\left\|\left(\begin{array}{c} \frac{1}{s+0.001}T_{re} \\ \frac{s}{0.001s+1}T_{ry} \end{array} \right)\right\|_\infty < 1$$

Note that this is a normalized gain constraint (unit bound across frequency).

```
viewSpec(R)
```

Requirement 1: Gain of W$_L$ H W$_R$ as a function of frequency
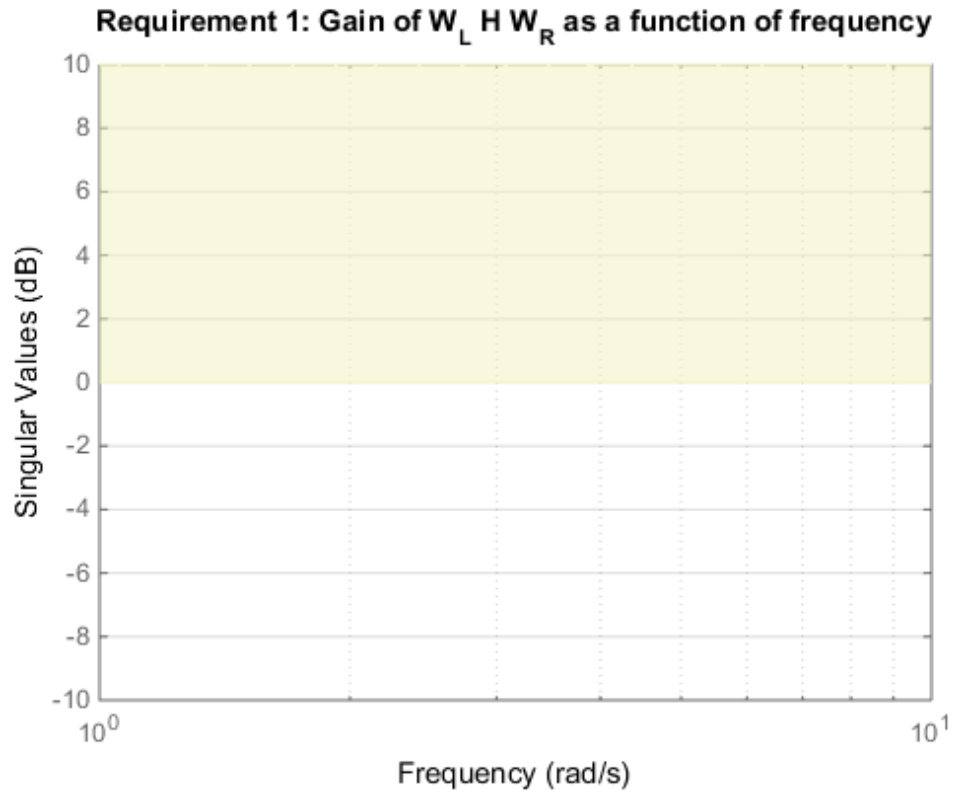
Loop Shape and Stability Margin Specifications

This example shows how to specify loop shapes and stability margins when tuning control systems with `systune` or `looptune`.

Background

The `systune` and `looptune` commands 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.

Loop Shape

The `TuningGoal.LoopShape` requirement is used to shape the open-loop response gain(s), a design approach known as *loop shaping*. For example,

```
s = tf('s');
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, use an `AnalysisPoint` block to mark the location "u", see the *"Building Tunable Models"* example for details. In Simulink, use the `addPoint` method of the `slTuner` interface to mark "u" as a point of interest.

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')
```
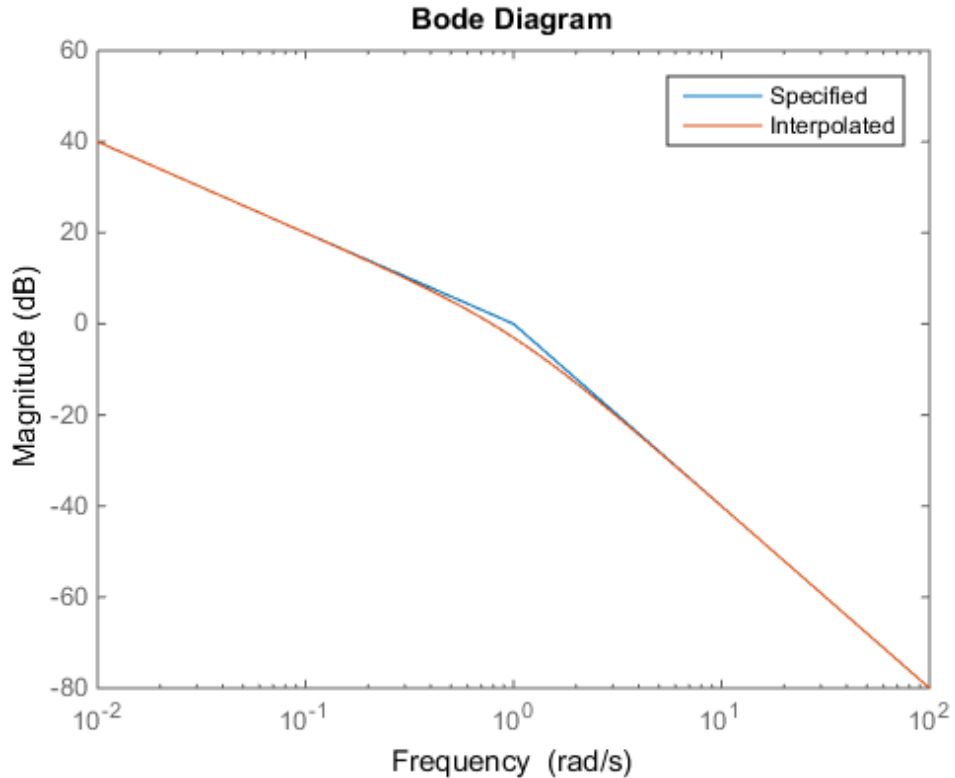
Loop shape requirements are constraints on the open-loop response L. For tuning purposes, they are converted into closed-loop gain constraints on the sensitivity function $S = 1/(1 + L)$ and complementary sensitivity function $T = L/(1 + L)$. Use `viewSpec` to visualize the target loop shape and corresponding gain bounds on S (green) and T (red).

```
viewSpec(R2)
```
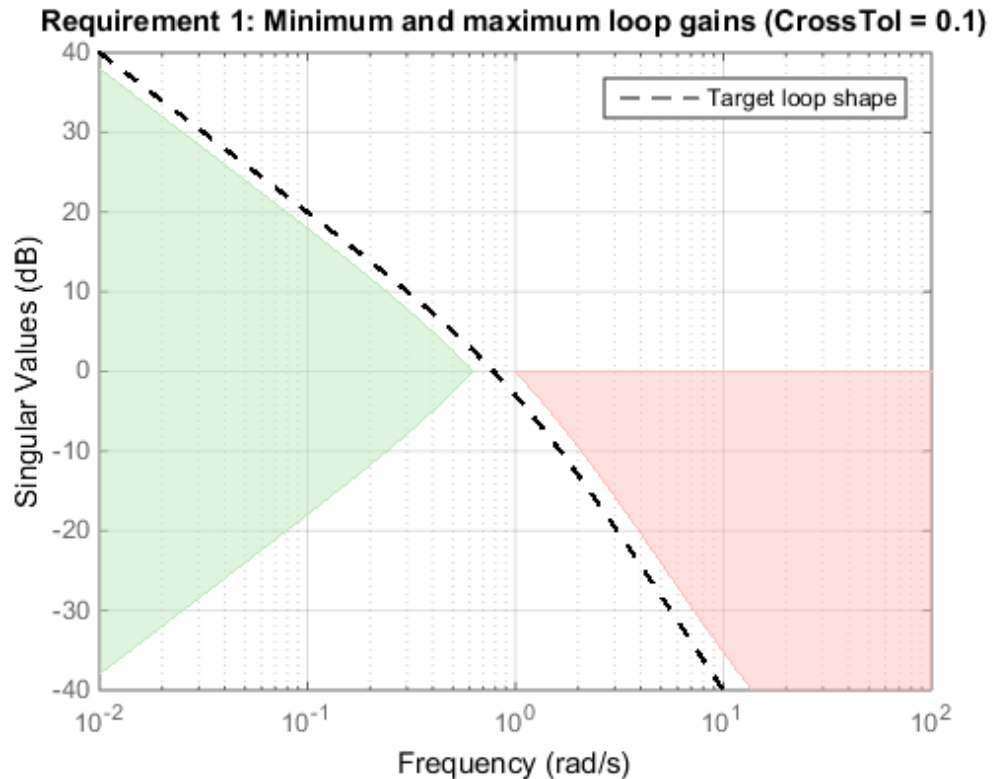
Minimum and Maximum Loop Gain

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])
```
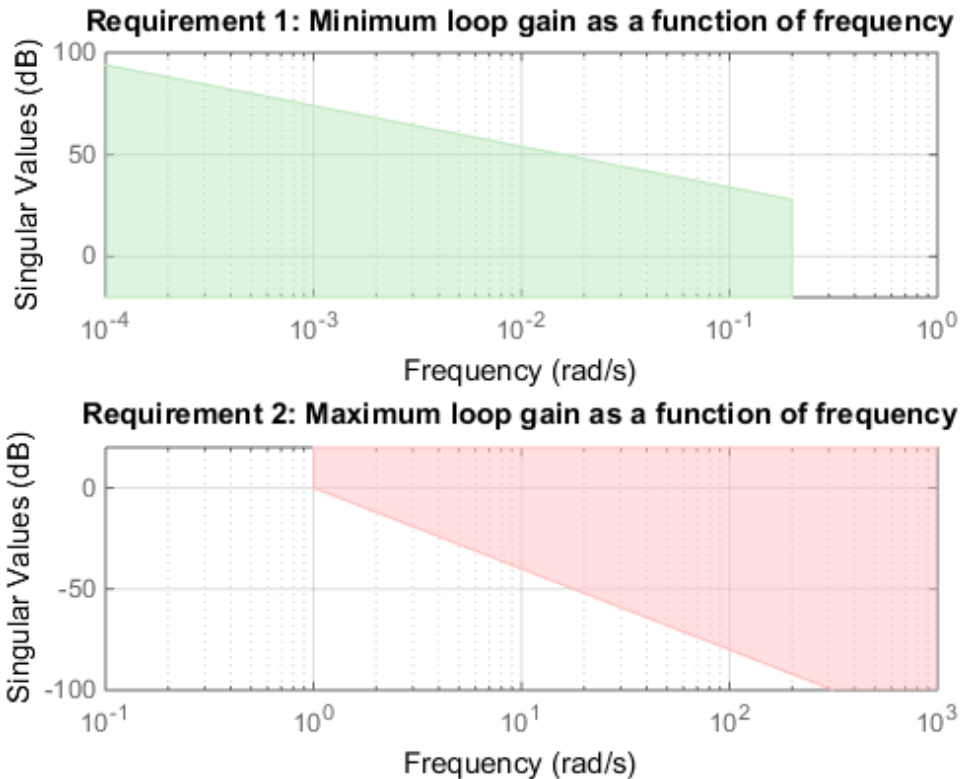

The `TuningGoal.MaxLoopGain` requirement rests on the fact that the open- and closed-loop gains are comparable when the loop gain is small ($|L| \ll 1$). As a result, it can be ineffective at keeping the loop gain below some value close to 1. For example, suppose that flexible modes cause gain spikes beyond the crossover frequency and that you need to keep these spikes below 0.5 (-6 dB). Instead of using `TuningGoal.MaxLoopGain`, you can directly constrain the gain of L using `TuningGoal.Gain` with a loop opening at "u".

```
MaxLG = TuningGoal.Gain('u','u',0.5);
```

```
MaxLG.Opening = 'u';
```

If the open-loop response is unstable, make sure to further disable the implicit stability constraint associated with this requirement.

```
MaxLG.Stabilize = false;
```

Figure 1 shows this requirement evaluated for an open-loop response with flexible modes.
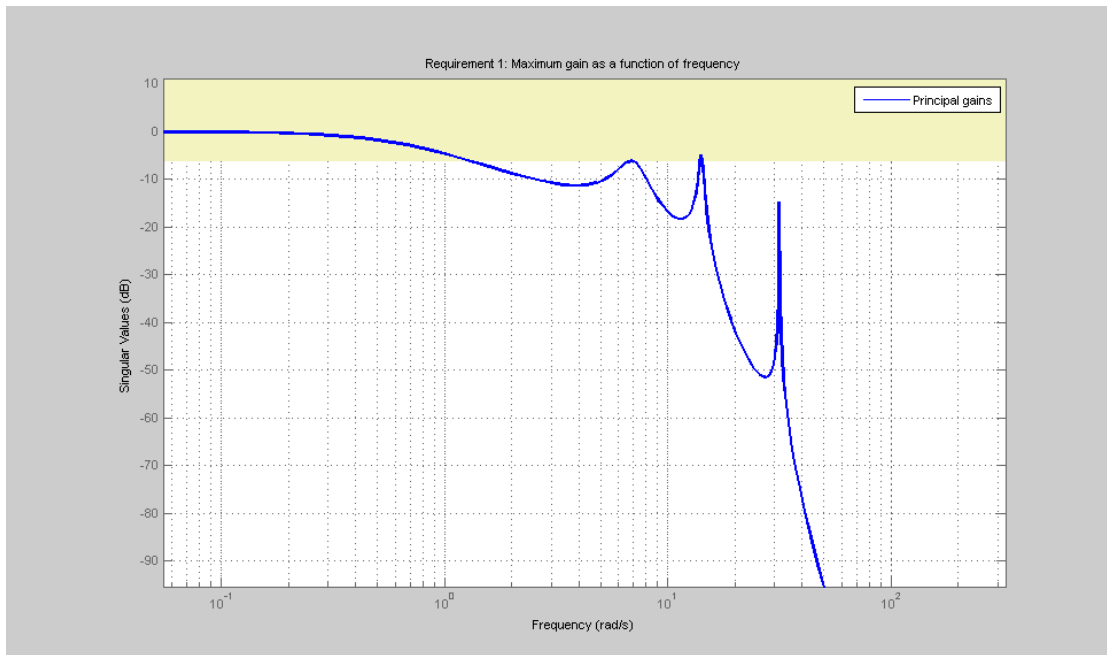
Figure 1: Gain constraint on L.

Stability Margins

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 ± 6 dB of gain margin and 45 degrees of phase margin at the location "u". In MATLAB, use a `AnalysisPoint` block to mark the location "u", see the *"Building Tunable Models"* example for details. In Simulink, use the `addPoint` method of the `slTuner` interface to mark "u" as a point of interest. 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)
```
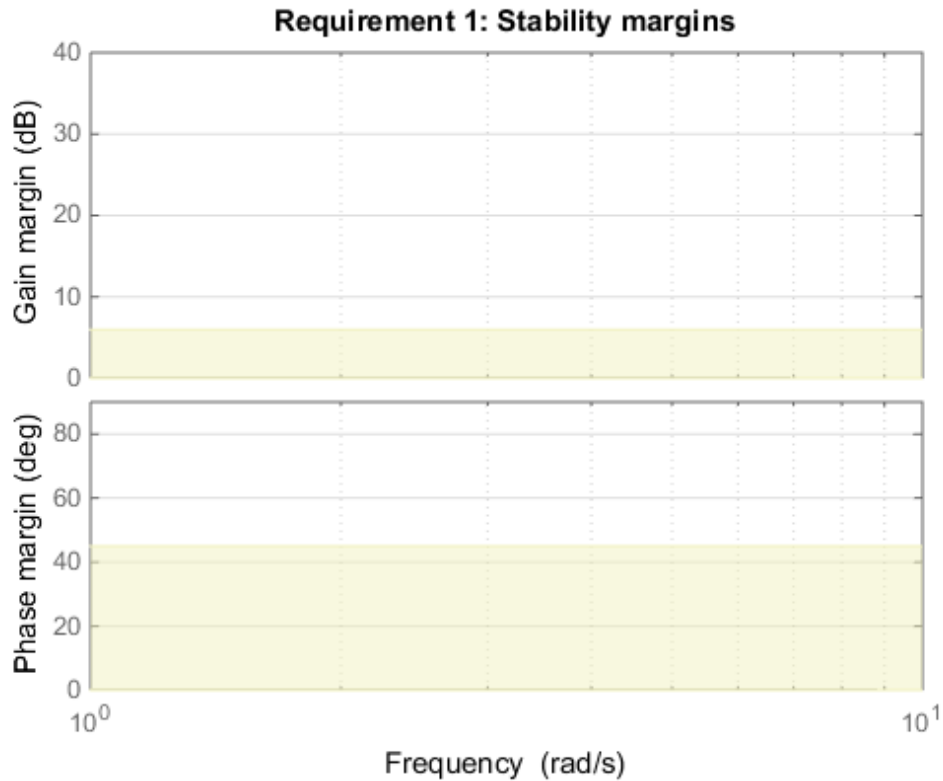
System Dynamics Specifications

This example shows how to constrain the poles of a control system tuned with `systune` or `looptune`.

Background

The `systune` and `looptune` commands 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.

Closed-Loop Poles

The `TuningGoal.Poles` goal constrains the location of the closed-loop poles. You can enforce some minimum decay rate

$$Re(s) < -\alpha,$$

impose some minimum damping ratio

$$Re(s) < -\zeta|s|,$$

or constrain the pole magnitude to

$$|s| < \omega_{\max}.$$

For example

```
MinDecay = 0.5;
MinDamping = 0.7;
MaxFrequency = 10;
R = TuningGoal.Poles(MinDecay,MinDamping,MaxFrequency);
```

constrains the closed-loop poles to lie in the white region below.

```
viewSpec(R)
```

Requirement 1: Closed-loop pole location
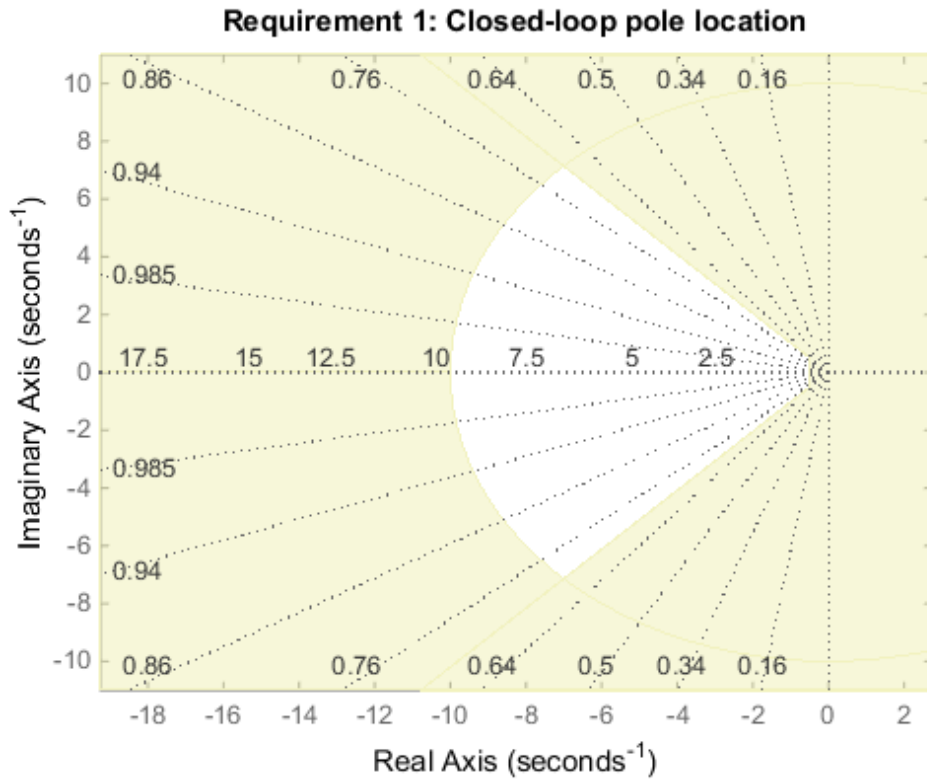
Increasing the `MinDecay` value results in faster transients. Increasing the `MinDamping` value results in better damped transients. Decreasing the `MaxFrequency` value prevents fast dynamics.

Controller Poles

The `TuningGoal.ControllerPoles` goal constrains the pole locations for tuned elements such as filters and compensators. The tuning algorithm may produce unstable compensators for unstable plants. To prevent this, use the `TuningGoal.ControllerPoles` goal to keep the compensator poles in the left-half plane. 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

```
MinDecay = 0;
R = TuningGoal.ControllerPoles('C',MinDecay);
```

Interpreting Stability Margins in Control System Tuning

In this section...

Stability Margins Plot

This topic explains how to interpret a stability-margin plot for control system tuning, such as the following plot.
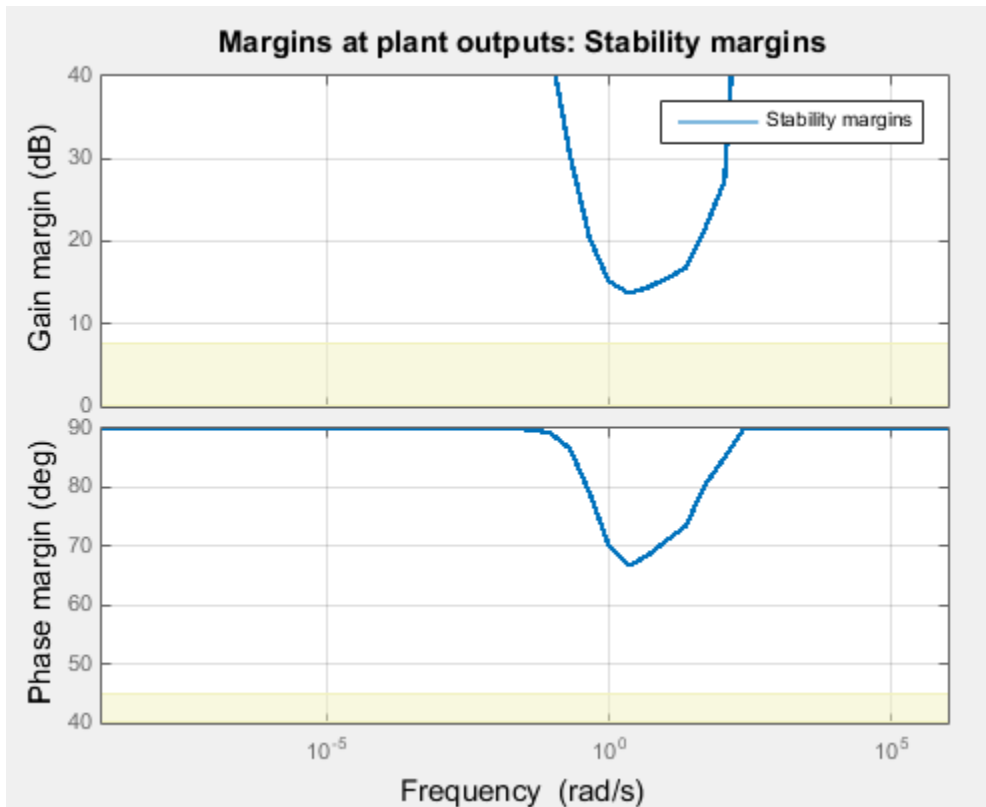
. You obtain this plot in one of the following ways:

- Tuning in Control System Tuner using a "Margins Goal" on page 6-140 or "Quick Loop Tuning" on page 6-62.
- Tuning at the command line using `systune` with `TuningGoal.Margins`. If `S` is the control system model or `slTuner` interface, and `Req` is a `TuningGoal.Margins` goal, obtain the stability-margin plot by entering:

```
viewSpec(S,Req)
```

Gain and Phase Margins

For SISO systems, the gain and phase margins at a frequency ω indicate how much the gain or phase of the open-loop response $L(j\omega)$ can change without loss of stability.

For example, a gain margin of 5dB at 2 rad/s indicates that closed-loop stability is maintained when the loop gain increases or decreases by as much as 5dB at this frequency. Gain and phase margins typically vary across frequencies.

For MIMO systems, gain and phase margins are interpreted as follows:

- Gain margin: Stability is preserved when the gain increases or decreases by up to the gain margin value in each channel of the feedback loop.
- Phase margin: Stability is preserved when the phase increases or decreases by up to the phase margin value in each channel of the feedback loop.

In MIMO systems, the gain or phase can change in all channels at once, and by a different amount in each channel. The Margins Goal and `TuningGoal.Margins` rely on the notion of disk margin for MIMO systems. (See "Algorithm" on page 6-214.) Like SISO stability margins, gain and phase margins in MIMO systems typically vary across frequency.

Combined Gain and Phase Variations

To assess robustness to changes in both gain and phase, use the following chart.
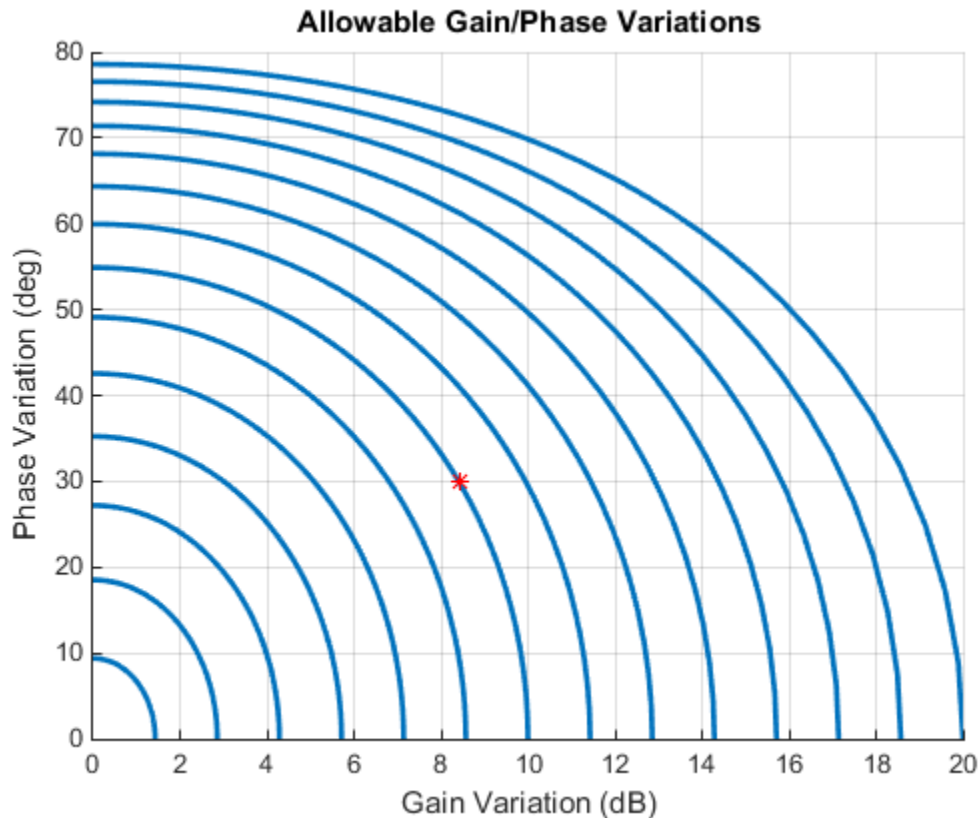
For example, if the gain margin plot in Control System Tuner indicates a 10 dB margin at a particular frequency, then trace the contour starting at (Gain,Phase) = (10,0) to see how a given amount of phase variation reduces the allowable gain variation at that frequency. For example, if the phase can vary by 30 degrees than the gain can only vary by about 8.4 dB (red mark).

Interpreting the Gain and Phase Margin Plot

The stability-margin plot for Margins Goal or TuningGoal.Margins shows in shaded yellow the region where the target margins are not met. The plot displays the current gain and phase margins (computed using the current values of the tunable parameters in the control system) as a blue trace.

These gain and phase margin curves are obtained using an exact calculation involving μ-analysis. For computational efficiency, however, the tuning algorithm uses an approximate calculation that can yield smaller margins in parts of the frequency range. To see the lower bound used by the tuner, right-click on the plot, and select **Systems > Tuned Lower Bound**.
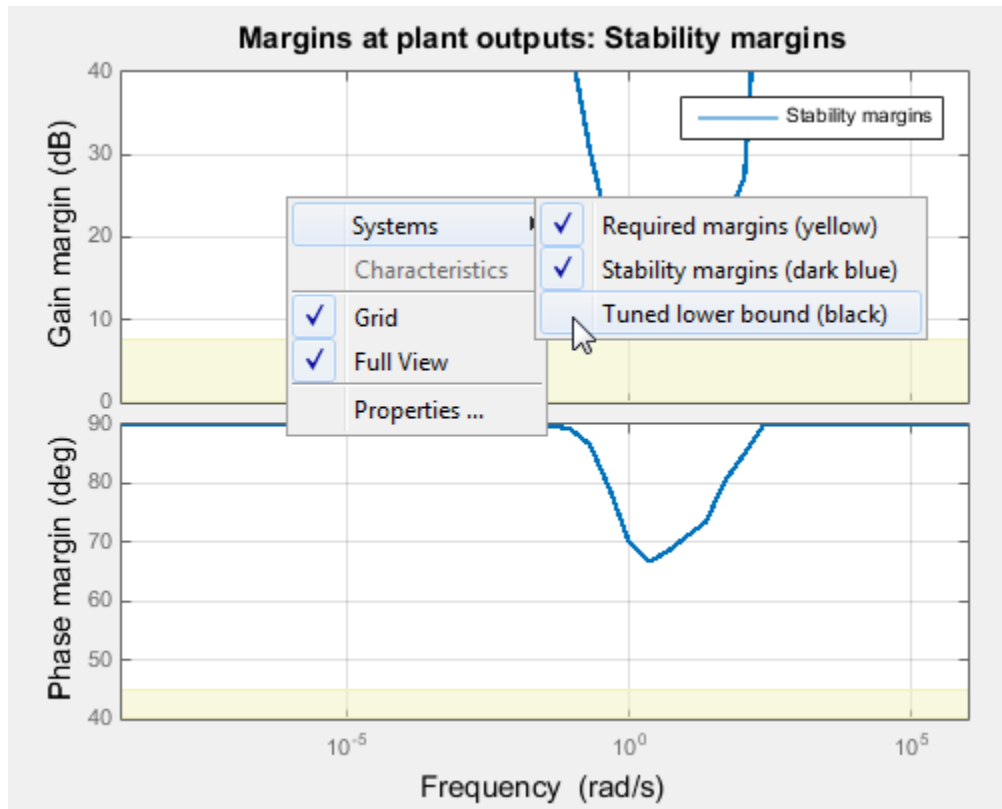
If there is a significant gap between the true margins and the tuner approximation, try increasing the D-scaling order. The default order is zero (static scaling). For tuning in Control System Tuner, set the D-scaling order in the Margins Goal dialog box. For command-line tuning, set this value using the `ScalingOrder` property of `TuningGoal.Margins`.

Algorithm

The gain and phase margin values are both derived from the disk margin (see `loopmargin`). The disk margin measures the radius of a circular exclusion region centered near the critical point. This radius is a decreasing function of the scaled norm:

$$\min_{D \text{ diagonal}} \left\| D^{-1} \left(I - L\left(j\omega \right) \right) \left(I + L\left(j\omega \right) \right)^{-1} D \right\|_2 .$$

Unlike the traditional gain and phase margins, the disk margins and associated gain and phase margins guarantee that the open-loop response $L(j\omega)$ stays at a safe distance from the critical point at all frequencies.

See Also
`TuningGoal.Margins` | `loopmargin`

More About
· "Margins Goal" on page 6-140

Tune Control System at the Command Line

After specifying your tuning goals using `TuningGoal` objects (see "Tuning Goals"), use `systune` to tune the parameters of your model.

The `systune` command lets you designate one or more design goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have tuning goals. `systune` attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

Organize your `TuningGoal` objects into a vector of soft requirements and a vector of hard requirements. For example, suppose you have created a tracking requirement, a rejection requirement, and stability margin requirements at the plant inputs and outputs. The following commands tune the control system represented by `T0`, treating the stability margins as hard goals, the tracking and rejection requirements as soft goals. (`T0` is either a `genss` model or an `slTuner` interface previously configured for tuning.)

```
SoftReqs = [Rtrack,Rreject];
HardReqs = [RmargIn,RmargOut];
[T,fSoft,gHard] = systune(T0,SoftReqs,HardReqs);
```

`systune` converts each tuning requirement into a normalized scalar value, f for the soft constraints and g for the hard constraints. The command adjusts the tunable parameters of `T0` to minimize the f values, subject to the constraint that each $g < 1$. `systune` returns the vectors `fSoft` and `gHard` that contain the final normalized values for each tuning goal in `SoftReqs` and `HardReqs`.

Use `systuneOptions` to configure additional options for the `systune` algorithm, such as the number of independent optimization runs, convergence tolerance, and output display options.

See Also

systune | systune | systuneOptions

More About

- "Interpreting Tuning Results" on page 6-161

Tune Controller Against Set of Plant Models

`systune` 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, `systune` 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 `systune`.

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

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 6-258

More About

- "Parallel Preferences"

Validate Tuned Control System at the Command Line

In this section...
"Extract and Plot System Responses" on page 6-219
"View Tuning Goals" on page 6-219
"Write Tuned Parameters to Simulink Model" on page 6-220
"Improve Tuning Results" on page 6-220

To validate your tuned control system, use the following tools and techniques.

Extract and Plot System Responses

Evaluate the performance of your tuned control system by extracting and plotting system responses. For instance, evaluate reference tracking or overshoot performance by plotting the step response of transfer function from the reference input to the controlled output. Or, evaluate stability margins by extracting an open-loop transfer function and using the `margin` command. You can extract any transfer function you need for analysis from the tuned model of your control system.

- To extract responses from a tuned generalized state-space (`genss`) model, use analysis functions such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`.

- For a Simulink tuned through an `slTuner` interface, extract responses from the interface using analysis functions such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`.

In either case, the extracted responses are represented by state-space (`ss`) models. You can analyze these models using commands such as `step`, `bode`, `sigma`, or `margin`.

View Tuning Goals

Visualize your tuning goals using the `viewSpec` command. For each type of tuning goal, `viewSpec` plots the target requirement and the achieved response of your tuned system. This visualization allows you to examine how far your control system is from ideal performance. It can also help you determine where you can achieve better tuning results by limiting the frequency range of a tuning goal, relaxing a tuning goal from hard to soft, increasing the tolerance of a tuning goal, or similar adjustments.

For example, suppose you have tuned your control system with a tracking requirement `Rtrack` and a rejection requirement `Rreject`. The following commands display plots that let you evaluate how closely the tuned system meets those requirements. (`T` is the tuned output of `systune`, either a `genss` model or an `slTuner` interface.)

```
viewSpec(Rtrack,T)
viewSpec(Rreject,T)
```

Write Tuned Parameters to Simulink Model

When you tune a Simulink model, the software evaluates tuning goals for a linearization of the model stored in the `slTuner` interface. Therefore, you must validate the tuned controllers by simulating the full nonlinear model, even if the tuned linear system meets all your tuning goals.

To write tuned block values from a tuned `slTuner` interface to the corresponding Simulink model, use the `writeBlockValue` command. For example, suppose `SLT` is a tuned `slTuner` interface returned by `systune`. The following command writes the tuned parameters from `SLT` to the associated Simulink model.

```
writeBlockValue(SLT)
```

Simulate the Simulink model to evaluate the tuned system performance.

Improve Tuning Results

If `systune` does not find a set of controller parameters that meet your design requirements, make adjustments to your set of tuning goals to improve the results. For example:

- Designate tuning goals that are must-have requirements as hard goals. Or, relax tuning goals that are not absolute requirements by designating them as soft goals.
- Limit the frequency range in which frequency-domain goals are enforced. Use the `Focus` property of the `TuningGoal` object to do this.
- Increase the tolerance of tuning goals for which a tolerance is applicable.

Related Examples

Extract Responses from Tuned MATLAB Model at the Command Line

This example shows how to analyze responses of a tuned control system by using `getIOTransfer` to compute responses between various inputs and outputs of a closed-loop model of the system.

Consider the following control system.
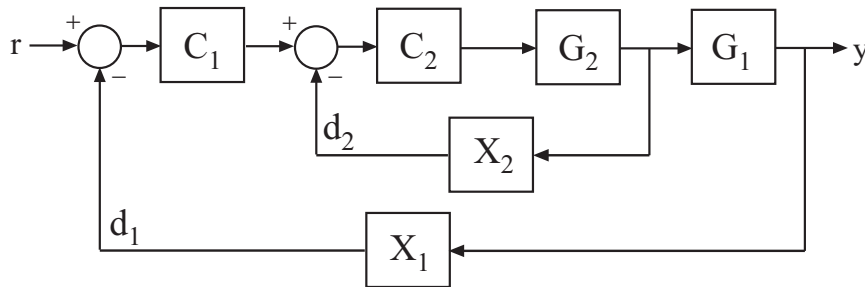
Suppose you have used `systune` to tune a `genss` model of this control system. The result is a `genss` model, T, which contains tunable blocks representing the controller elements C1 and C2. The tuned model also contains `AnalysisPoint` blocks that represent the analysis-point locations, X1 and X2.

Analyze the tuned system performance by examining various system responses extracted from T. For example, examine the response at the output, y, to a disturbance injected at the point d_1.

```
H1 = getIOTransfer(T,'X1','y');
```

H1 represents the closed-loop response of the control system to a disturbance injected at the implicit input associated with the `AnalysisPoint` block X1, which is the location of d_1:

H1 is a `genss` model that includes the tunable blocks of `T`. H1 allows you to validate the disturbance response of your tuned system. For example, you can use analysis commands such as `bodeplot` or `stepplot` to analyze H1. You can also use `getValue` to obtain the current value of H1, in which all the tunable blocks are evaluated to their current numeric values.

Similarly, examine the response at the output to a disturbance injected at the point d_2.

```
H2 = getIOTransfer(T,'X2','y');
```

You can also generate a two-input, one-output model representing the response of the control system to simultaneous disturbances at both d_1 and d_2. To do so, provide `getIOTransfer` with a cell array that specifies the multiple input locations.

```
H = getIOTransfer(T,{'X1','X2'},'y');
```

Tuning Control Systems with SYSTUNE

The systune command can jointly tune the gains of your control system regardless of its architecture and number of feedback loops. This example outlines the systune 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
```
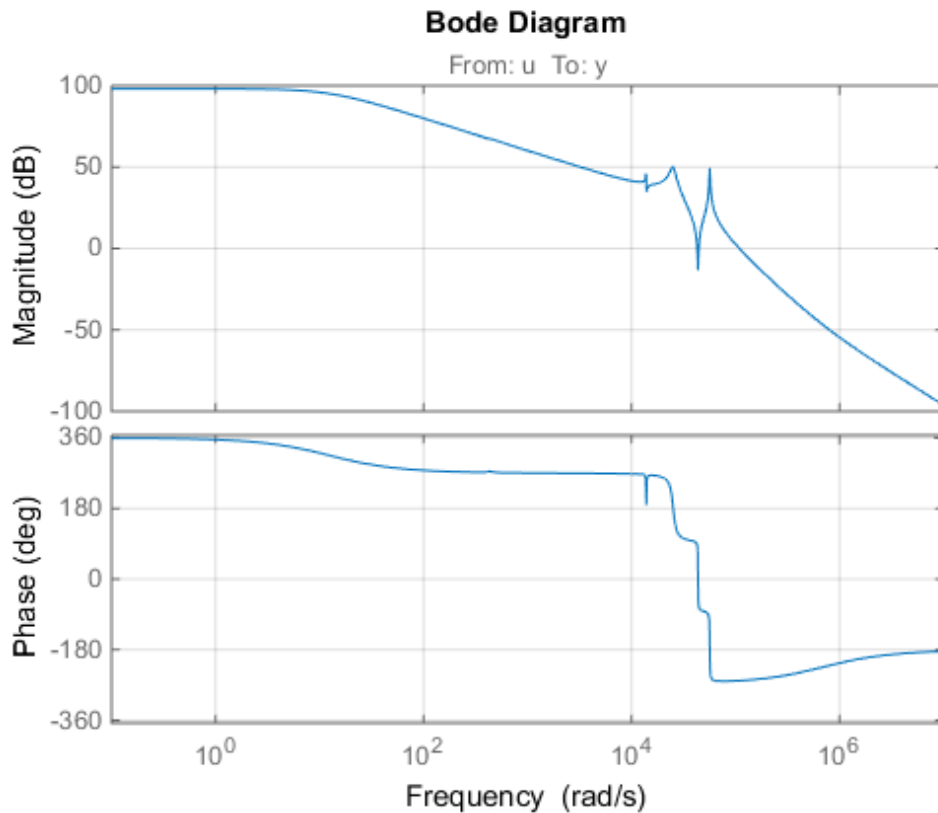

Bode Diagram

6-223

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.
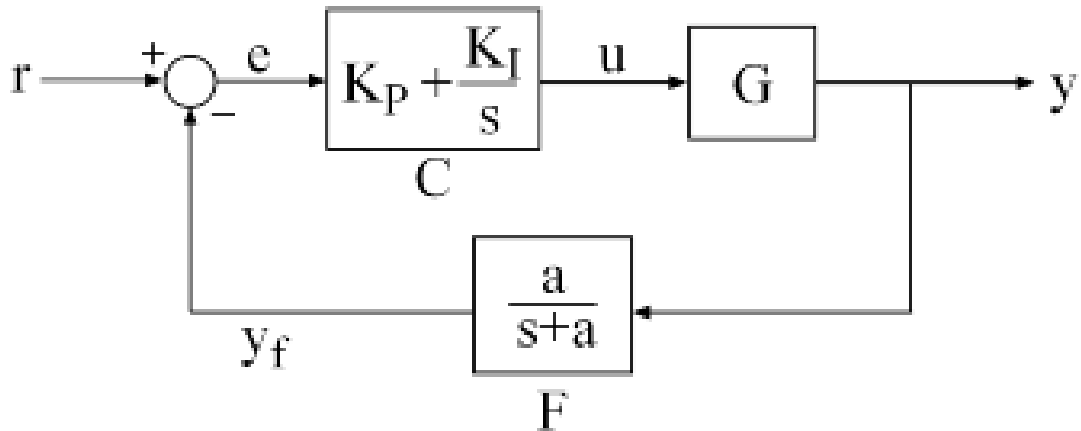
Figure 1: Control Structure

You can use `systune` 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:

```
CO = 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
F0 = 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 an analysis point at the plant input u:

```
AP = AnalysisPoint('u');
```
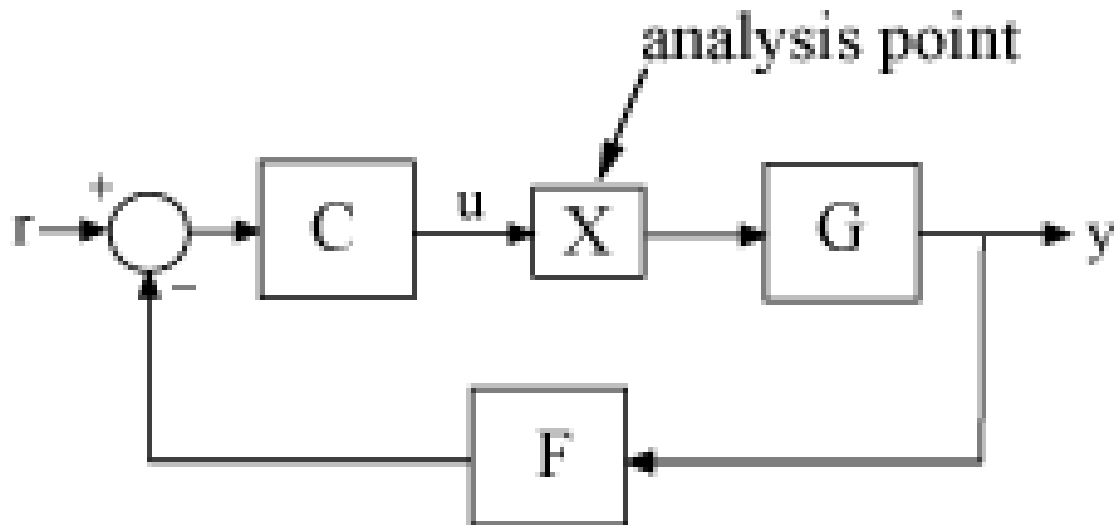

Figure 2: Analysis Point Block

Use `feedback` to build a model of the closed-loop transfer from reference r to head position y:

```
T0 = feedback(G*AP*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.

6-225

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 analysis point `AP`).

```
Req1 = TuningGoal.Tracking('r','y',0.001);
Req2 = TuningGoal.Margins('u',6,45);
```

Tuning the Controller Parameters

You can now use `systune` to tune the PI gain and filter coefficient a. 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 = systuneOptions('RandomStart',3);
[T,fSoft,~,Info] = systune(T0,[Req1,Req2],Options);

Final: Soft = 2.61, Hard = -Inf, Iterations = 120
Final: Soft = 1.35, Hard = -Inf, Iterations = 155
Final: Soft = 2.78e+03, Hard = -Inf, Iterations = 187
       Some closed-loop poles are marginally stable (decay rate near 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 =

             1
  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 =

             1
  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)
```
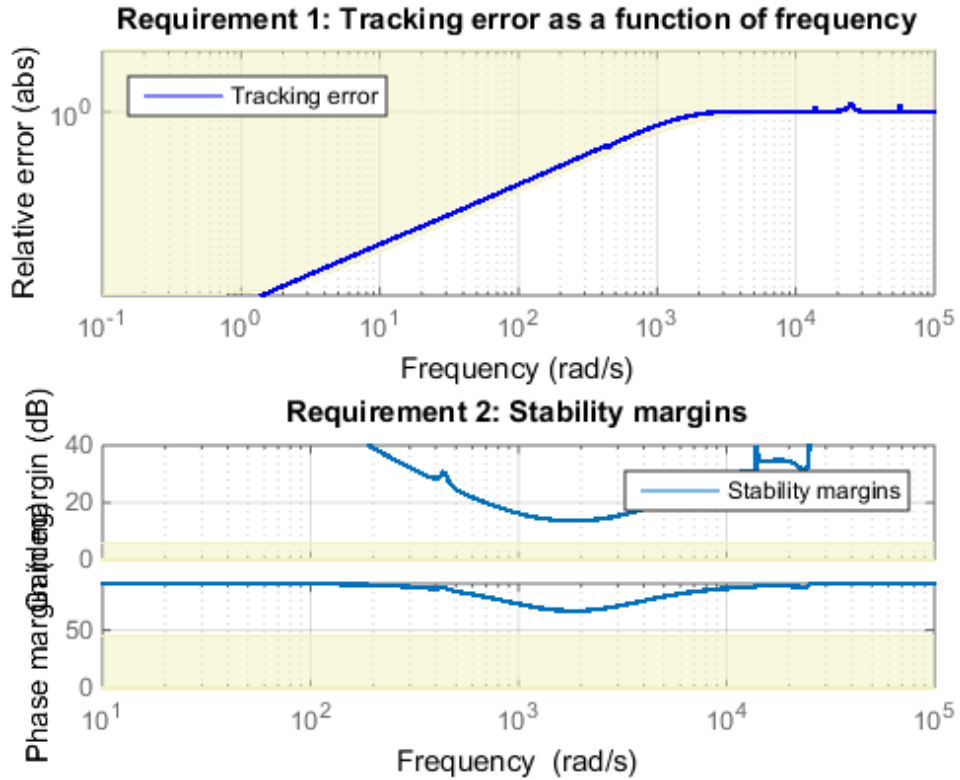

Next plot the closed-loop step response from reference r to head position y. The response has no overshoot but wobbles a little.

```
clf, step(T)
```
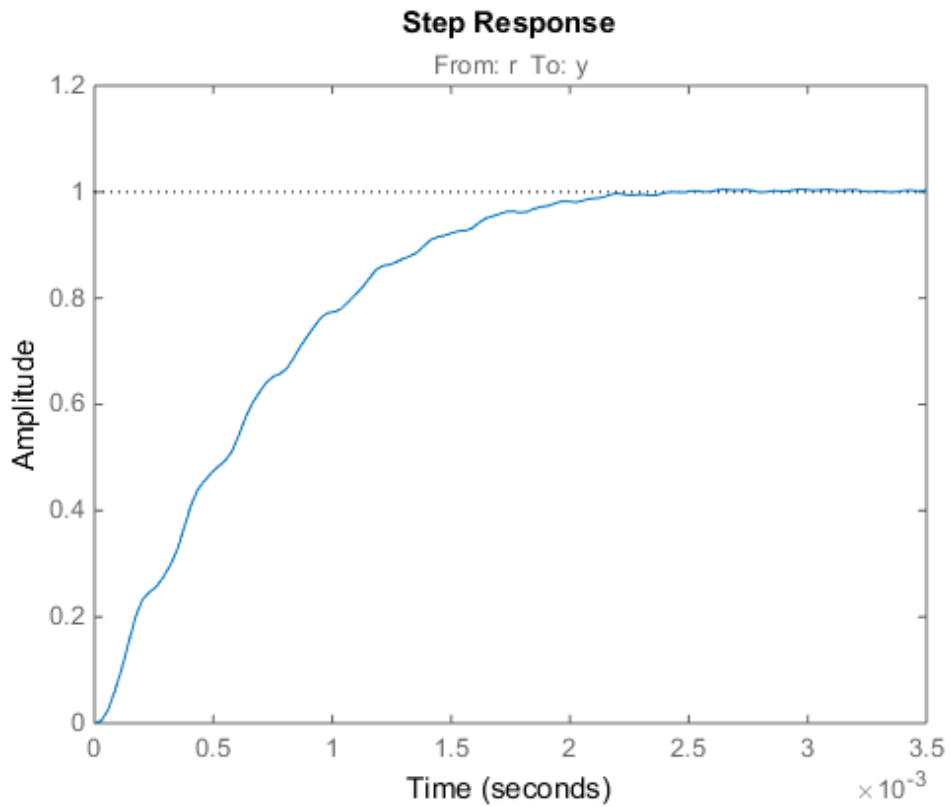
To investigate further, use `getLoopTransfer` to get the open-loop response at the plant input.

```
L = getLoopTransfer(T,'u');
bode(L,{1e3,1e6}), grid
title('Open-loop response')
```
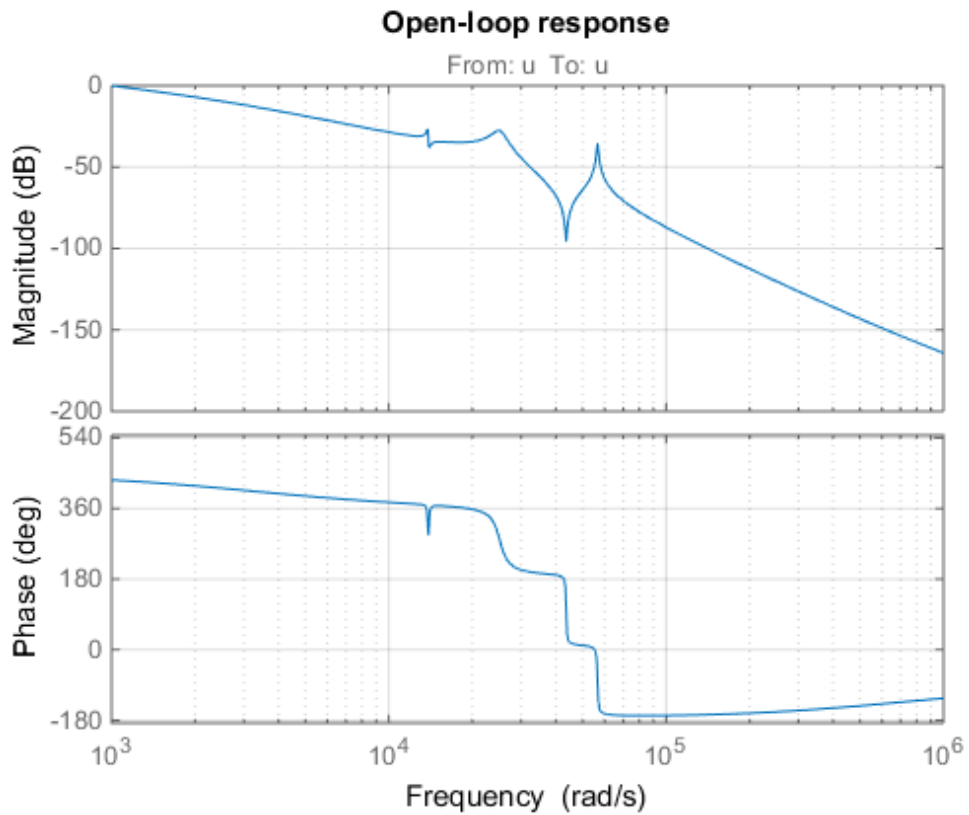
The wobble is due to the first resonance after the gain crossover. To eliminate it, you could add a notch filter to the feedback loop and tune its coefficients along with the lowpass coefficient and PI gains using `systune`.

Tuning Control Systems in Simulink

This example shows how to use `systune` 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')
```
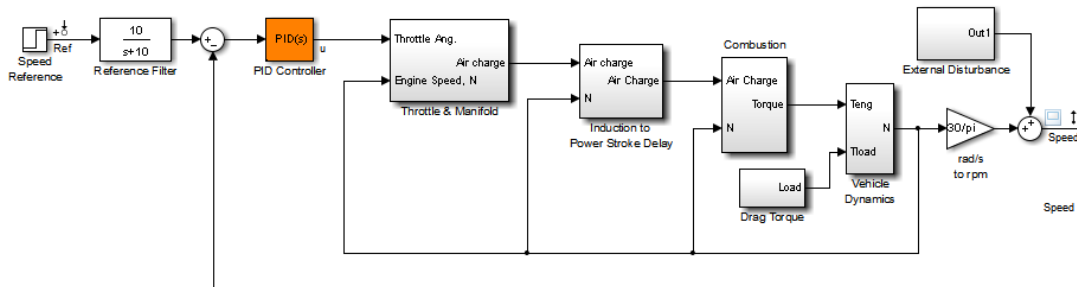

Copyright 2004-2010 The MathWorks, Inc.

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 `systune` and `looptune` workflows in Simulink.

Controller Tuning with SYSTUNE

The `slTuner` interface provides a convenient gateway to `systune` 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 `slTuner` instance for the `rct_engine_speed` model and list the "PID Controller" block (highlighted in orange) as tunable. Note that all Linear Analysis points in the model (signals "Ref" and "Speed" here) are automatically available as points of interest for tuning.

```
ST0 = slTuner('rct_engine_speed','PID Controller');
```

6-231

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.

```
getBlockValue(ST0,'PID Controller')
```

```
ans =

         1
  Ki * ---
         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 = 118
```

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:

```
getBlockValue(ST1,'PID Controller')
```

```
ans =

              1              s
  Kp + Ki * --- + Kd * --------
              s           Tf*s+1
```

```
  with Kp = 0.00188, Ki = 0.00333, Kd = 0.000469, Tf = 9.72e-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 = getIOTransfer(ST0,'Ref','Speed');
T1 = getIOTransfer(ST1,'Ref','Speed');
step(T0,T1)
legend('Initial','Tuned')
```
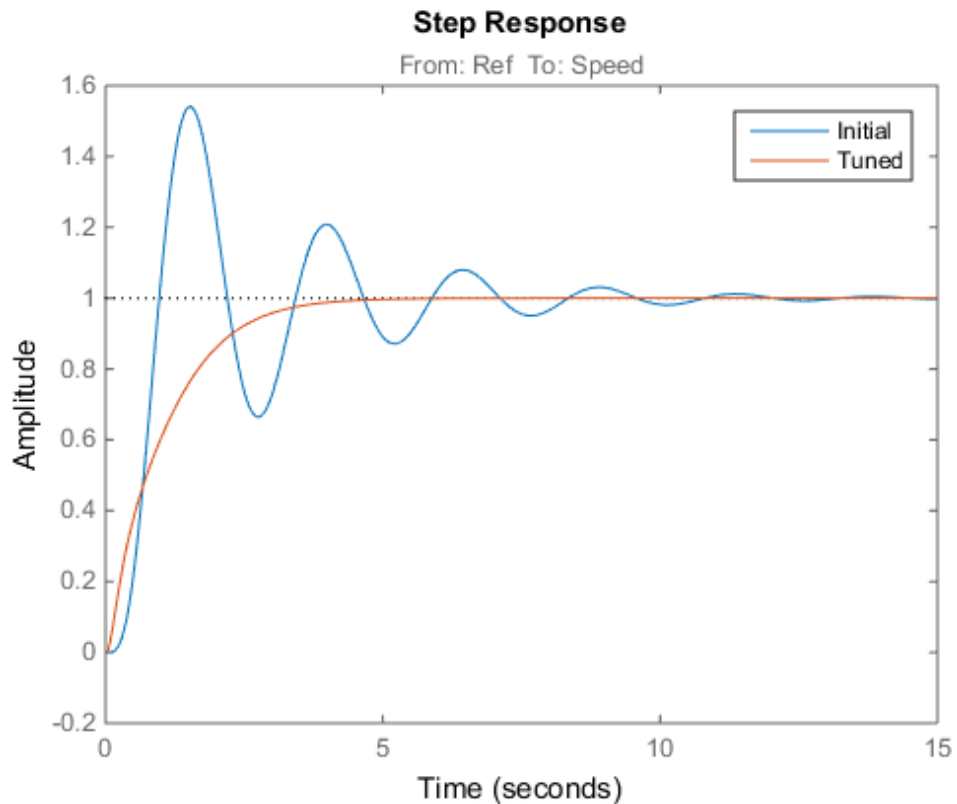
Controller Tuning with LOOPTUNE

You can also use `looptune` to tune control systems modeled in Simulink. The `looptune` workflow is very similar to the `systune` 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. 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.

```
% Mark the signal "u" as a point of interest
addPoint(ST0,'u')

% Tune the controller parameters
Control = 'u';
Measurement = 'Speed';
wc = 1;
ST1 = looptune(ST0,Control,Measurement,wc);

Final: Peak gain = 0.961, Iterations = 10
Achieved target gain value TargetGain=1.
```

Again the final value is close to 1, indicating that the target control bandwidth was achieved. As with `systune`, 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 `systune`.

```
T0 = getIOTransfer(ST0,'Ref','Speed');
T1 = getIOTransfer(ST1,'Ref','Speed');
step(T0,T1)
legend('Initial','Tuned')
```
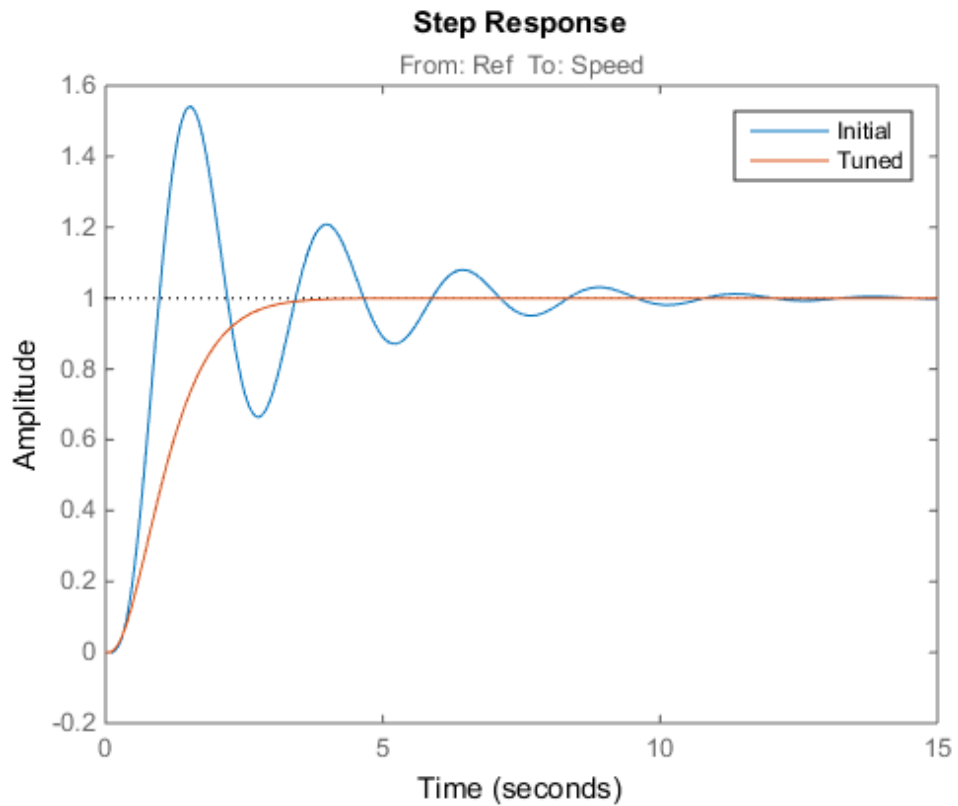
You can also perform open-loop analysis, for example, compute the gain and phase margins at the plant input u.

```
% Note: -1 because |margin| expects the negative-feedback loop transfer
L = getLoopTransfer(ST1,'u',-1);

margin(L), grid
```

Bode Diagram
Gm = 35.9 dB (at 12.8 rad/s), Pm = 73.8 deg (at 0.891 rad/s)
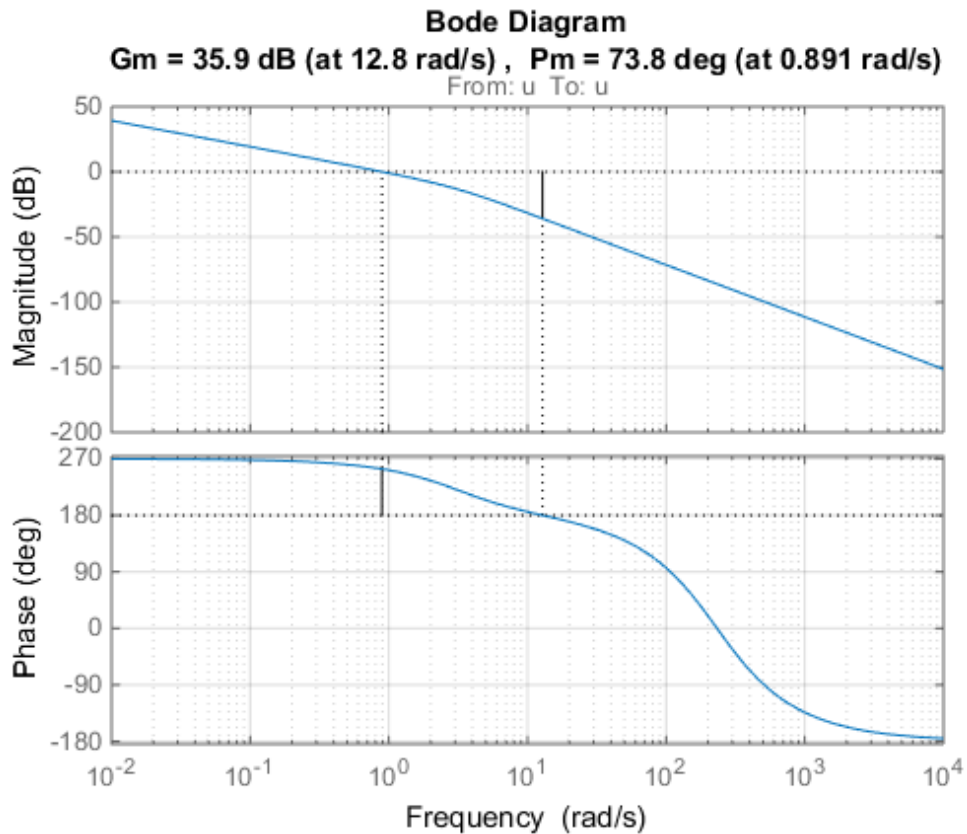
Validation in Simulink

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.

```
writeBlockValue(ST1)
```

You can now simulate the engine response with the tuned PID controller.
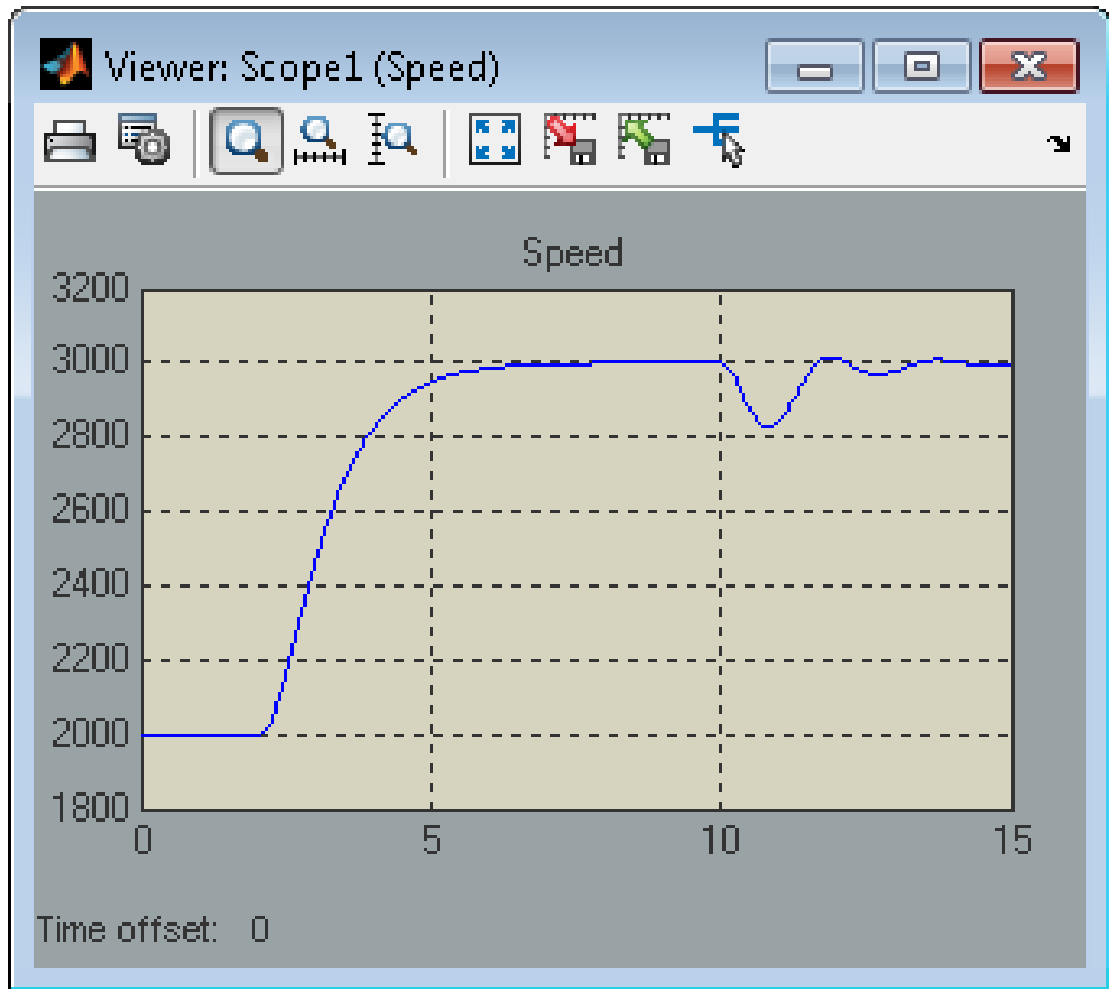
The nonlinear simulation results closely match the linear responses obtained in MATLAB.

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 Tf coefficient.

```
showTunable(ST1)
```

```
Block 1: rct_engine_speed/PID Controller =

               1              s
  Kp + Ki * --- + Kd * --------
               s           Tf*s+1

  with Kp = 0.000653, Ki = 0.00282, Kd = 0.0021, Tf = 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:

```
setBlockParam(STO,'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 = looptune(STO,Control,Measurement,wc);

Final: Peak gain = 0.915, Iterations = 5
Achieved target gain value TargetGain=1.
```

Again the final value is less than one indicating success. Compare the closed-loop response with the previous ones:

```
T2 = getIOTransfer(ST2,'Ref','Speed');
step(T0,T1,T2,'r--')
legend('Initial','PID','PI')
```
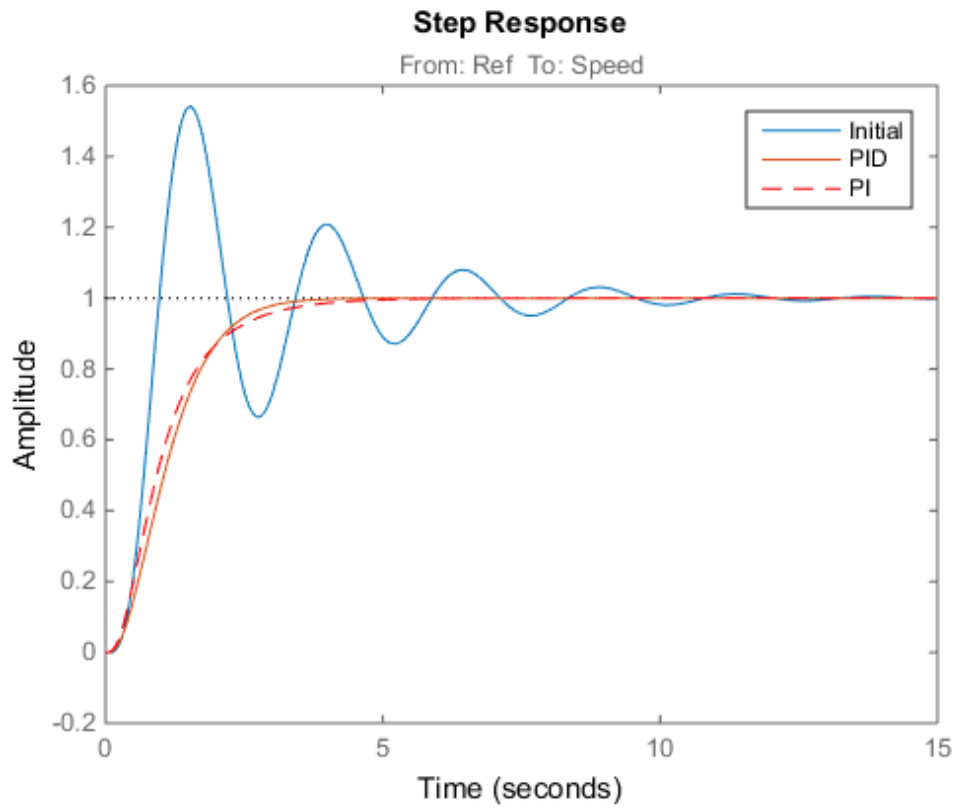
Clearly a PI controller is sufficient for this application.

Building Tunable Models

This example shows how to create tunable models of control systems for use with `systune` or `looptune`.

Background

You can tune the gains and parameters of your control system with `systune` 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.
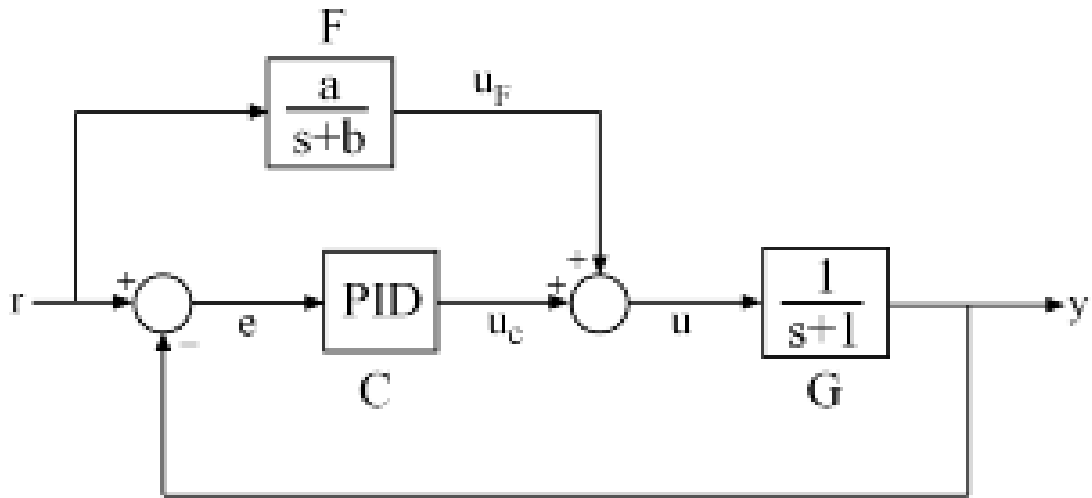
Figure 1: Control System with Feedforward and Feedback Paths

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')
```

6-241

```
T =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and
    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 "T.Blocks" t
```

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. You can use `systune` to automatically tune the PID gains and the feedforward coefficients `a,b` subject to your performance requirements. Use `showTunable` to see the current value of the tunable blocks.

```
showTunable(T)

C =

         1
  Ki * ---
         s

  with Ki = 0.001

Name: C
Continuous-time I-only controller.
---------------------------------
F =

    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 `Kp,Ki,Kd,Tf`. The tunable block `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 states, and
    a: Scalar parameter, 2 occurrences.

Type "ss(F)" to see the current value, "get(F)" to see all properties, and "F.Blocks" t
```

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 states, and
    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 "C.Blocks" t
```

Enabling Open-Loop Requirements

The `systune` 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 an `AnalysisPoint` block as shown in Figure 2.
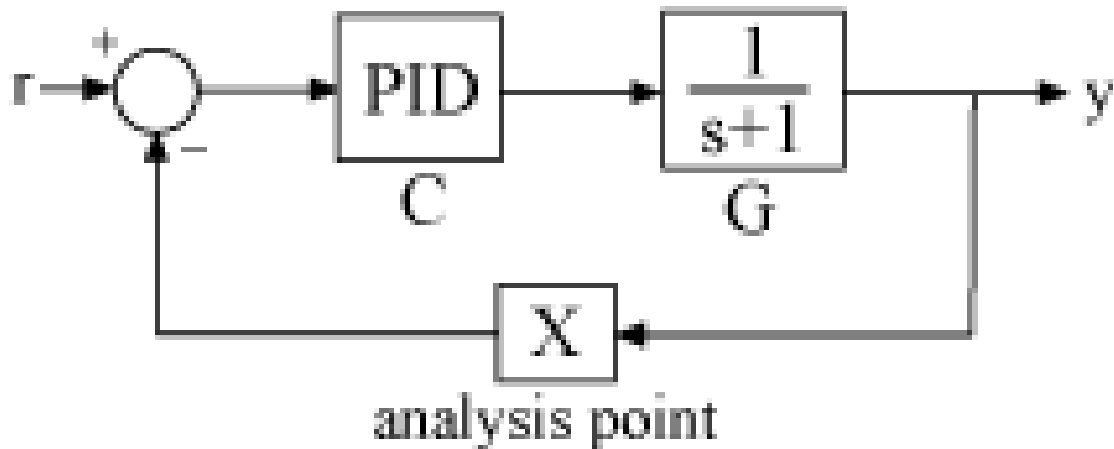
Figure 2: Analysis Point Block

The `AnalysisPoint` block can be used to mark internal signals of interest as well as locations where to open feedback loops and measure open-loop responses. This block evaluates to a unit gain and has no impact on the model 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');
AP = AnalysisPoint('X');
T = feedback(G*C,AP);
```

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--')
```

Bode Diagram

From: X To: X
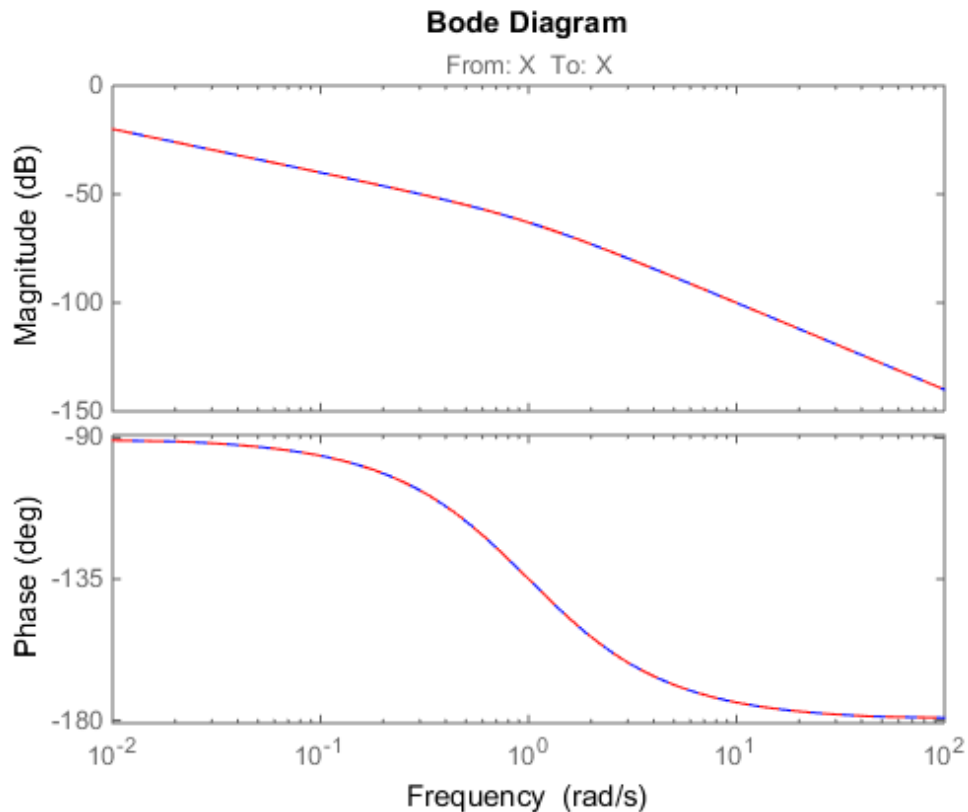
You can also refer to the location "X" when specifying target loop shapes or stability margins for `systune`. The requirement then applies to the loop transfer measured at this location.

```
% 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, loop opening locations are specified in the `Location` property of `AnalysisPoint` blocks. For single-channel analysis points, the block name is used as default location name. For multi-channel analysis points, indices are appended to the block name to form the default location names.

```
AP = AnalysisPoint('Y',2);  % two-channel analysis point
AP.Location


ans =

    'Y(1)'
    'Y(2)'
```

You can override the default location names and use more descriptive names by modifying the Location property.

```
% Rename loop opening locations to "InnerLoop" and "OuterLoop".
AP.Location = {'InnerLoop' ; 'OuterLoop'};
AP.Location


ans =

    'InnerLoop'
    'OuterLoop'
```

Validating Results

This example shows how to interpret and validate tuning results from `systune`.

Background

You can tune the parameters of your control system with `systune` or `looptune`. The design specifications are captured using `TuningGoal` requirement objects. This example shows how to interpret the results from `systune`, graphically verify the design requirements, and perform additional open- and closed-loop analysis.

Controller Tuning with SYSTUNE

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
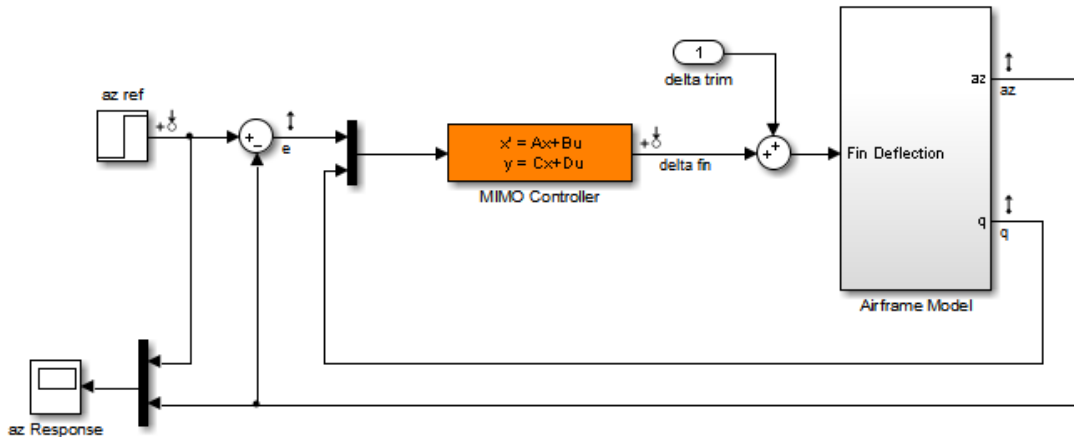
The setup and tuning steps are repeated below for completeness.

```
STO = slTuner('rct_airframe2','MIMO Controller');

% Compensator parameterization
```

```
CO = ltiblock.ss('C',2,1,2);
CO.d.Value(1) = 0;  CO.d.Free(1) = false;
setBlockParam(STO,'MIMO Controller',CO)

% 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] = systune(STO,[Req1,Req2,Req3,Req4],Opt);

Final: Soft = 1.5, Hard = -Inf, Iterations = 60
Final: Soft = 1.49, Hard = -Inf, Iterations = 95
Final: Soft = 1.15, Hard = -Inf, Iterations = 64
Final: Soft = 1.15, Hard = -Inf, Iterations = 87
```

Interpreting Results

`systune` run three optimizations from three different starting points and returned
the best overall result. The first output `ST` is an `slTuner` 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)
```
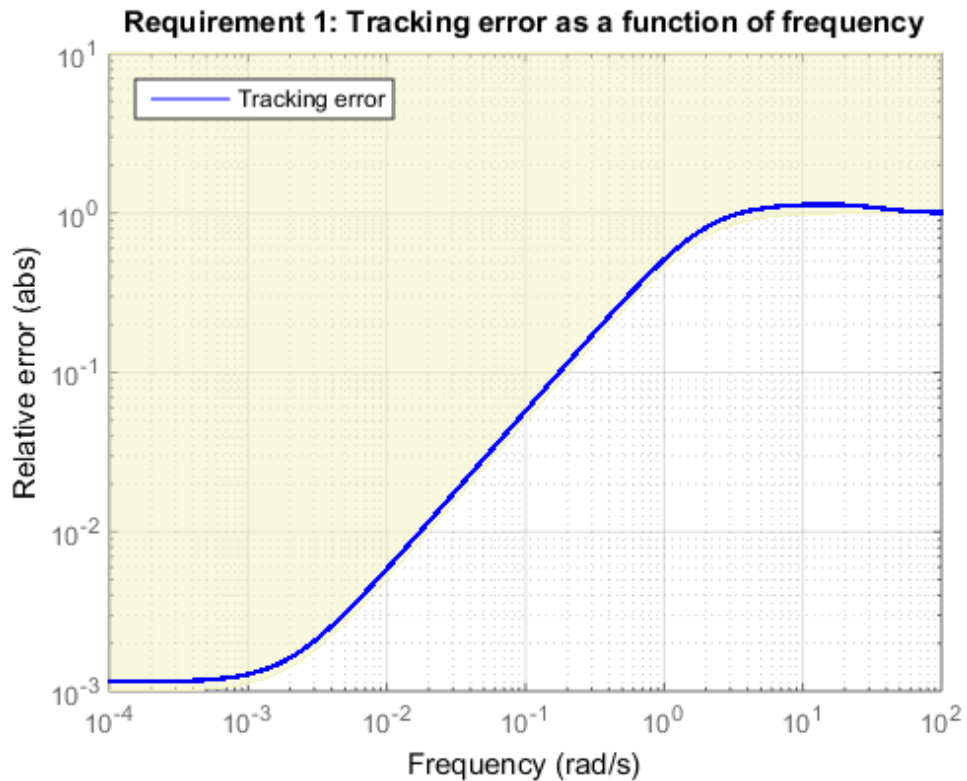

We observe a slight violation across frequency, suggesting that setpoint tracking will perform close to expectations. Similarly, verify the disturbance rejection requirement.

```
viewSpec(Req4,ST1,Info)
legend('location','NorthWest')
```

6-251
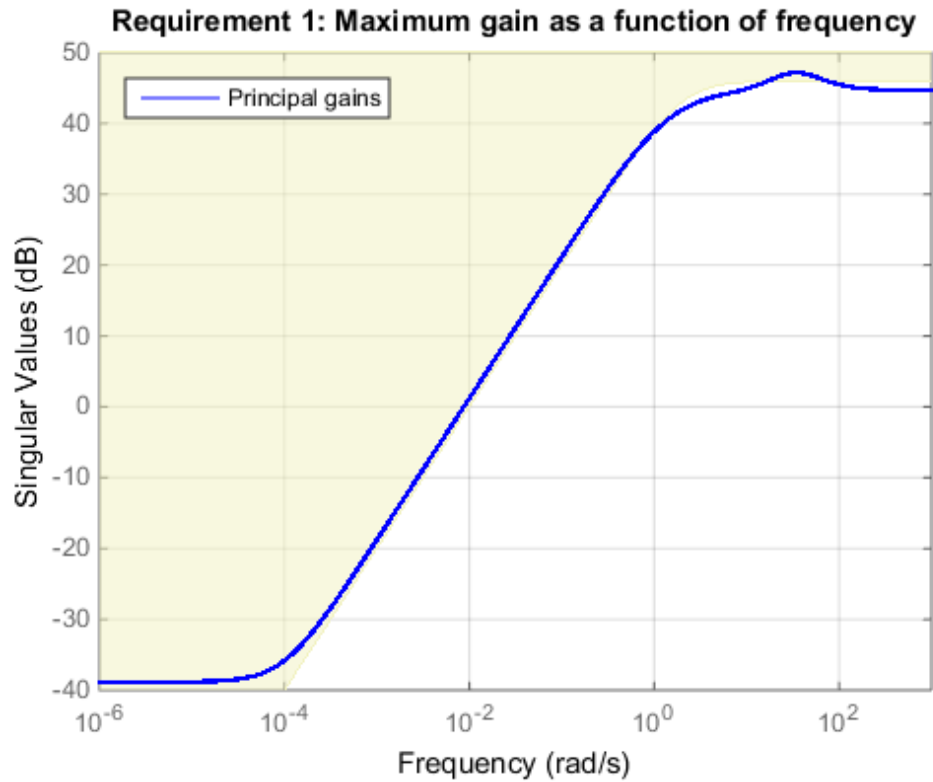
Most of the violation is at low frequency with a small bump near 35 rad/s, suggesting possible damped oscillations at this frequency. Finally, verify the stability margin requirement.

```
viewSpec(Req3,ST1,Info)
```
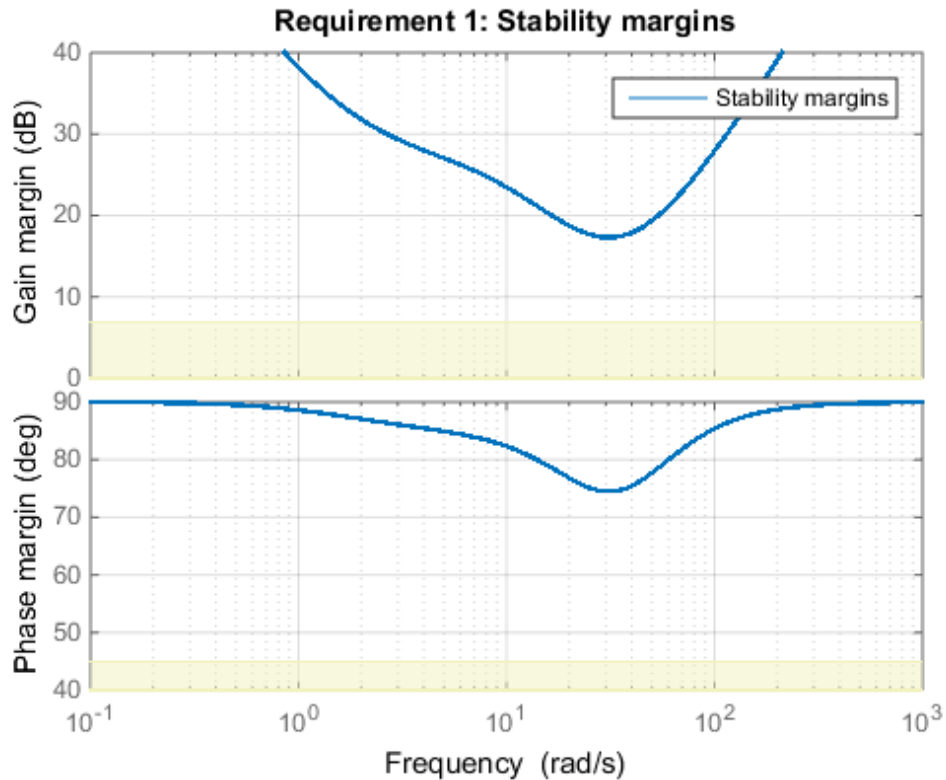
This requirement is satisfied at all frequencies, with the smallest margins achieved near the crossover frequency as expected. To see the actual margin values at a given frequency, click on the red curve and read the values from the data tip.

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 computed 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)
```

Step Response

From: az ref To: az
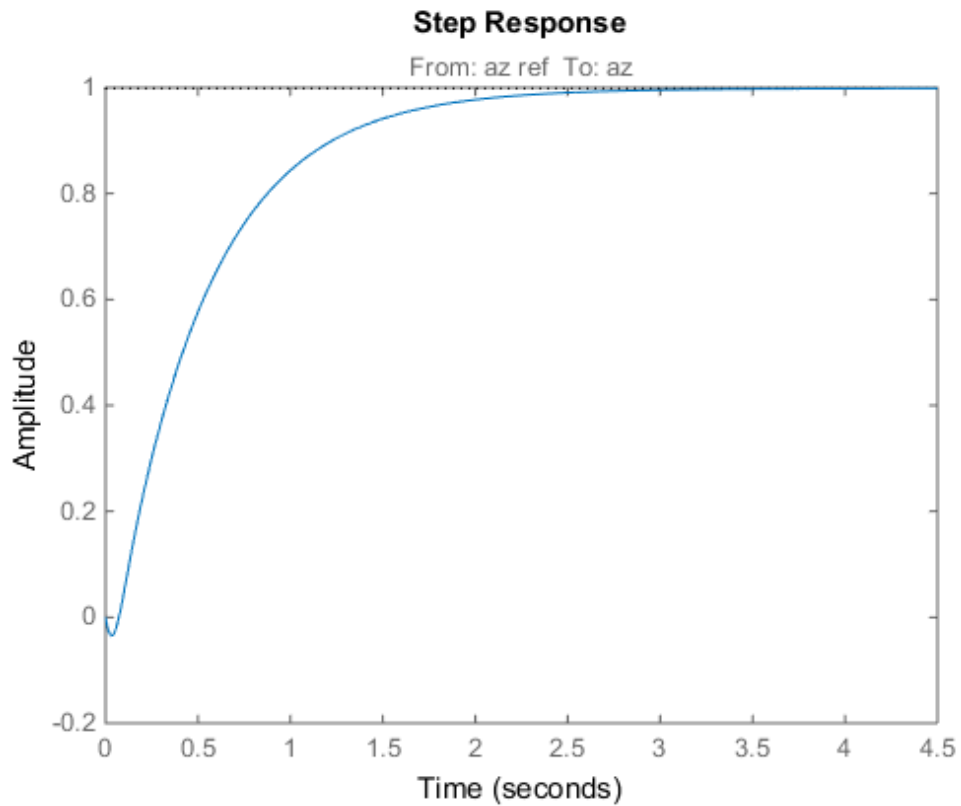
Also plot the open-loop response measured at the plant input `delta fin`. You can use this plot to assess the classical gain and phase margins at the plant input.

```
L = ST1.getLoopTransfer('delta fin',-1);  % negative-feedback loop transfer
margin(L), grid
```

Bode Diagram
Gm = -51.2 dB (at 0.386 rad/s) , Pm = 74.4 deg (at 32.5 rad/s)
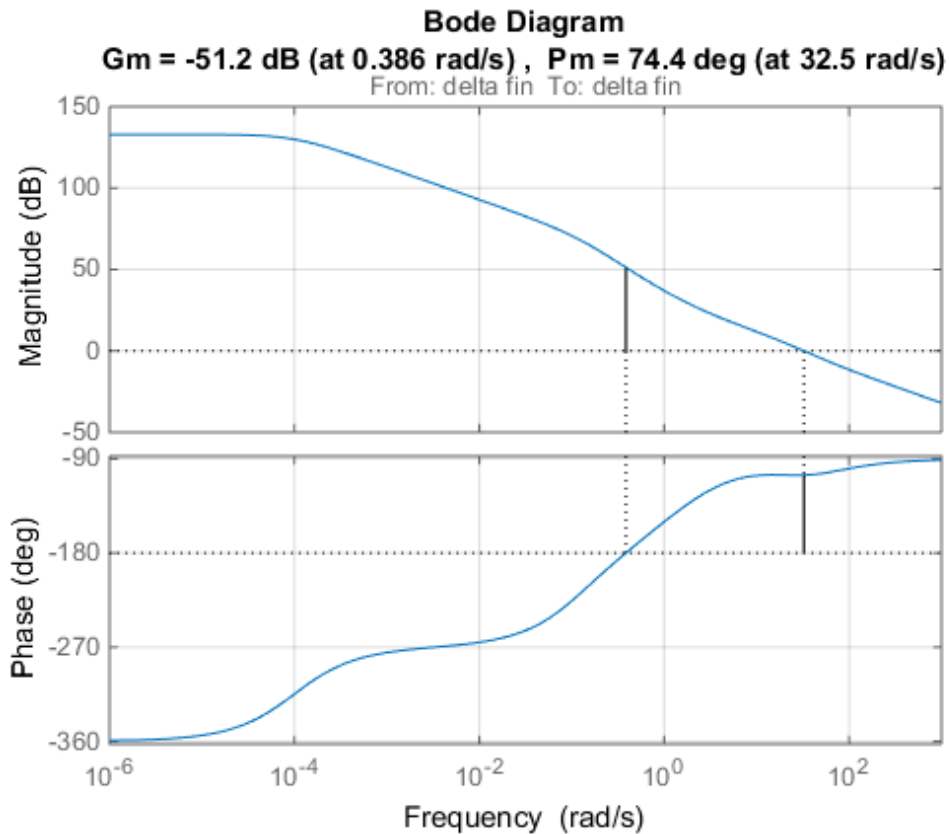
Soft vs Hard Requirements

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 = 1.29, Hard = 0.99999, Iterations = 207
```

```
fSoft
```

```
fSoft =

    1.2949    1.2949
```

gHard

```
gHard =

    0.4774    1.0000
```

Here fSoft contains the final values of the first two requirements (soft constraints) and gHard contains 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.

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
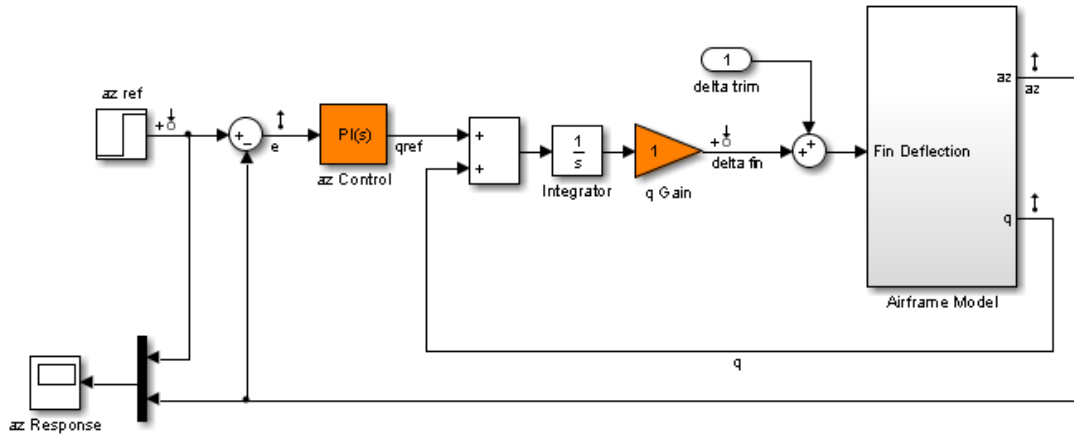
See AutopilotExample to see how you can tune the autopilot gains with the
LOOPTUNE command.

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 azref with a 1 second response
time. Use slTunable to configure this tuning task (see "Tuning of a Two-Loop Autopilot"
example for details):

```
STO = slTunable('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] = STO.looptune(wc,TrackReq,Options);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
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" =

              1
  Kp + Ki * ---
              s

  with Kp = 0.00165, Ki = 0.00166

Name: az_Control
Continuous-time PI controller in parallel form.

----------------------------------

Block "rct_airframe1/q Gain" =

  d =
          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)
```

Tuning Control Systems with CONTROL SYSTEM TUNER

This example shows how to use Control System Tuner App™ of 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,theta,v,p,phi,r] consists of

- Longitudinal velocity u (m/s)
- Lateral velocity v (m/s)
- Normal velocity w (m/s)
- Pitch angle theta (deg)
- Roll angle phi (deg)
- Roll rate p (deg/s)
- Pitch rate q (deg/s)
- Yaw rate r (deg/s).

The controller generates commands ds,dc,dT in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of theta, 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, rise 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 `Control System Tuner` App. This app 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 `Control System Tuner` to quickly set up the tuning task. First, open `Control System Tuner App` from Simulink Analysis Menu.
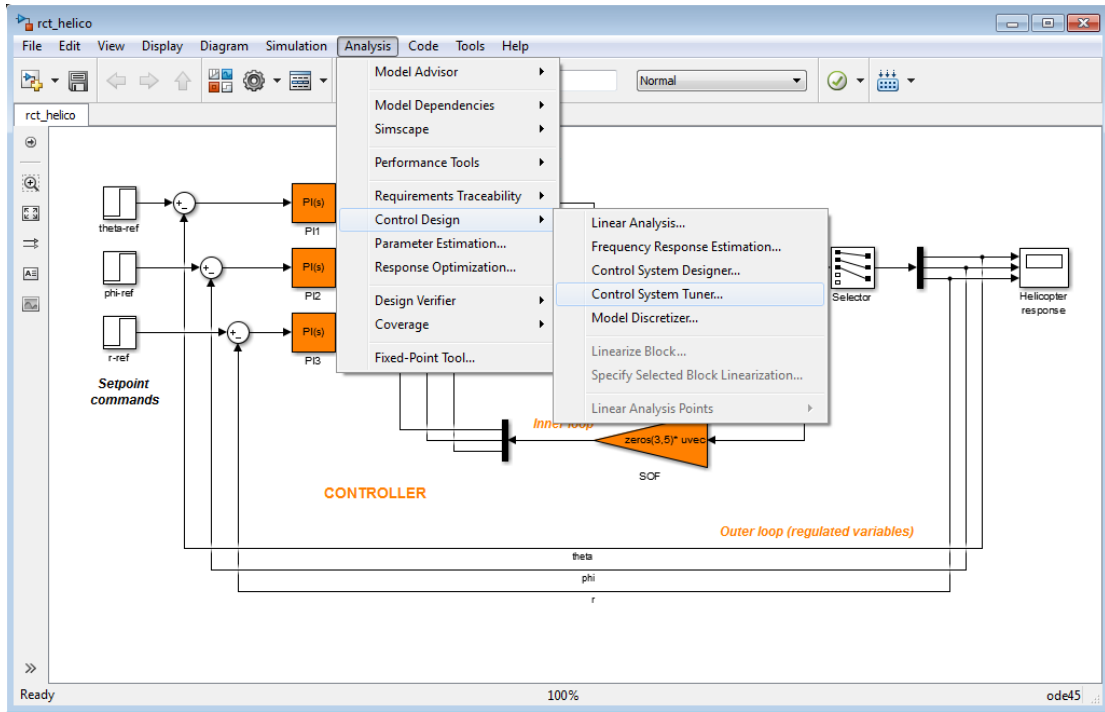
Figure 1: Control System Tuner App.

Then, open the editor for tuned blocks by clicking `Select Blocks` from `Tuning` tab.
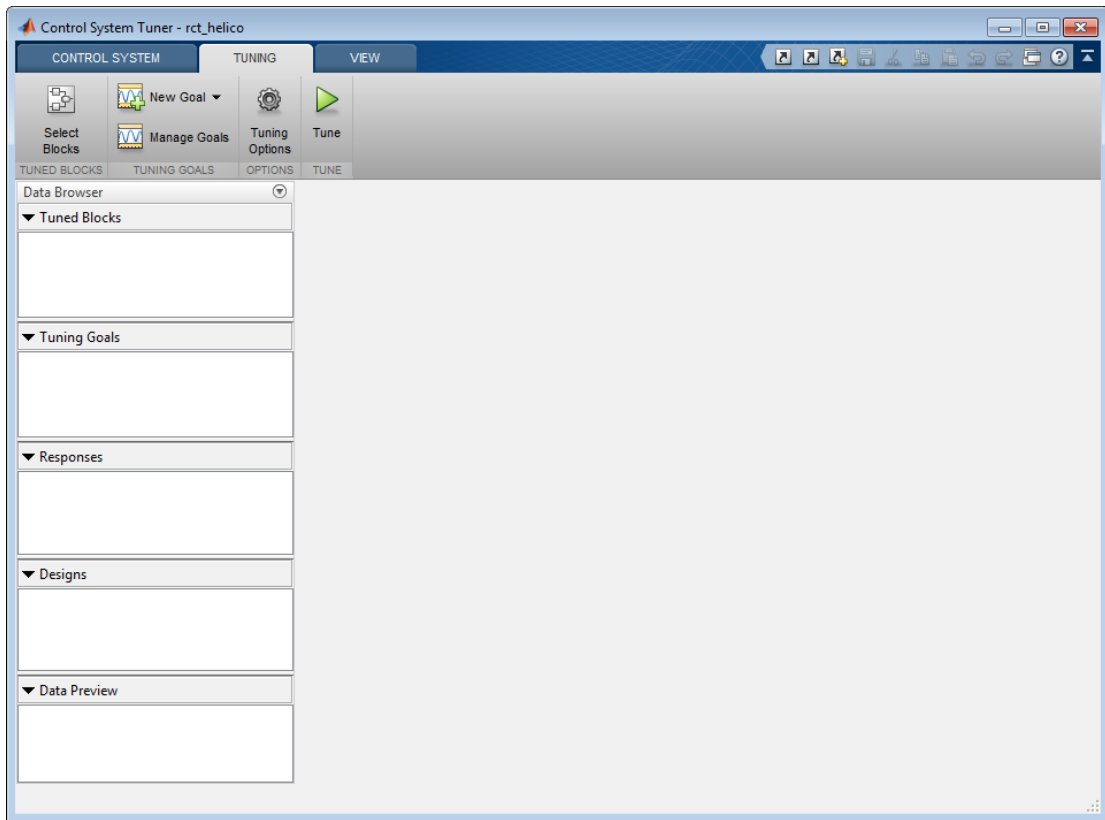
Figure 2: Tuning Tab of Control System Tuner App.
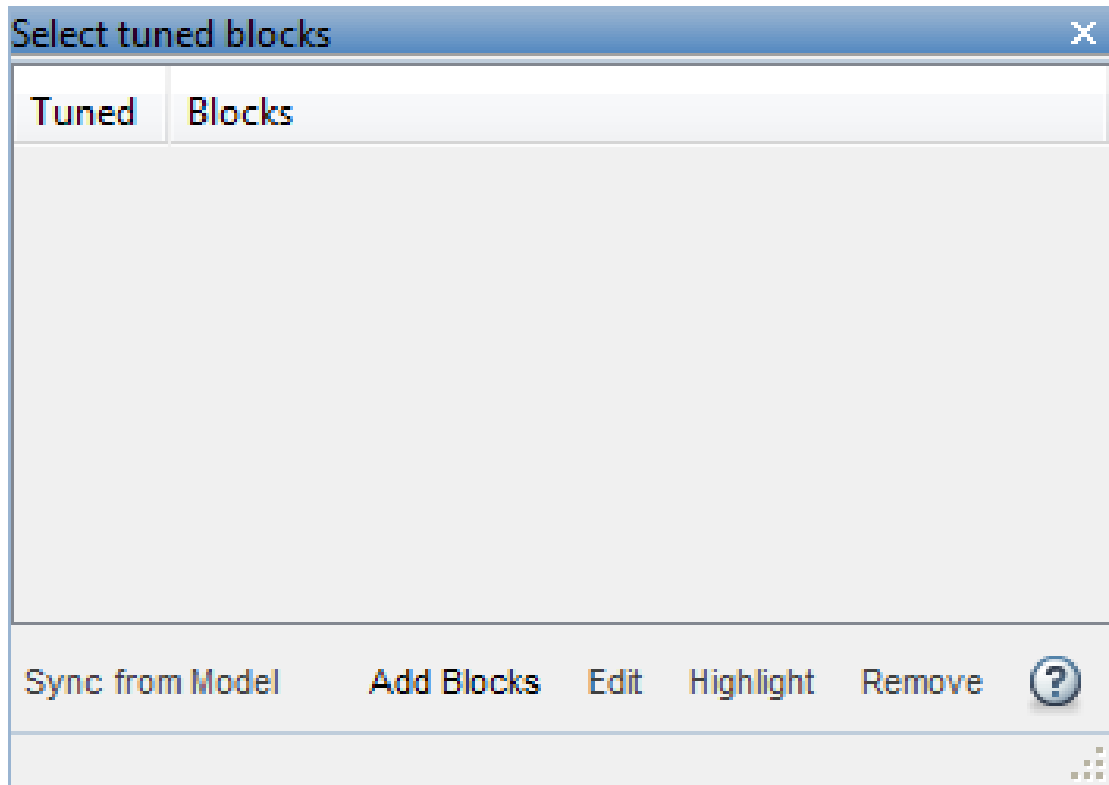
Figure 3: Editor for Tuned Blocks of Control System Tuner App.

Add list of blocks, PI1, PI2, PI3, SOF to be tuned into `Control System Tuner` from `Add Blocks`.
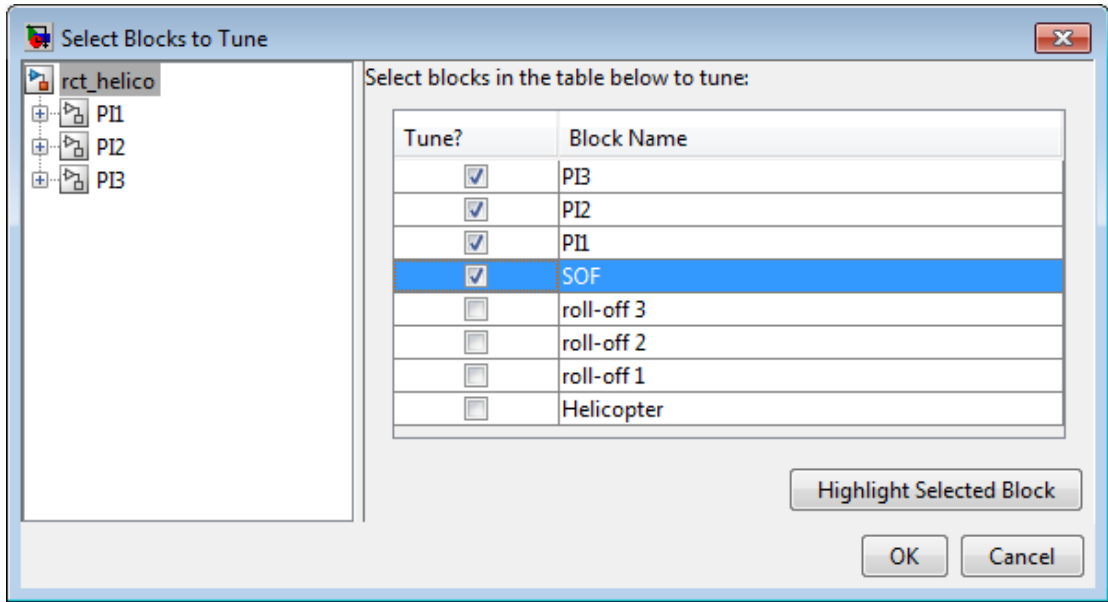
Figure 4: Selection of Tuned Blocks of Control System Tuner App.

Selected tuned blocks appears both in the editor and in the `Tuned Blocks` section of Data Browser on the left side of Control System Tuner App.
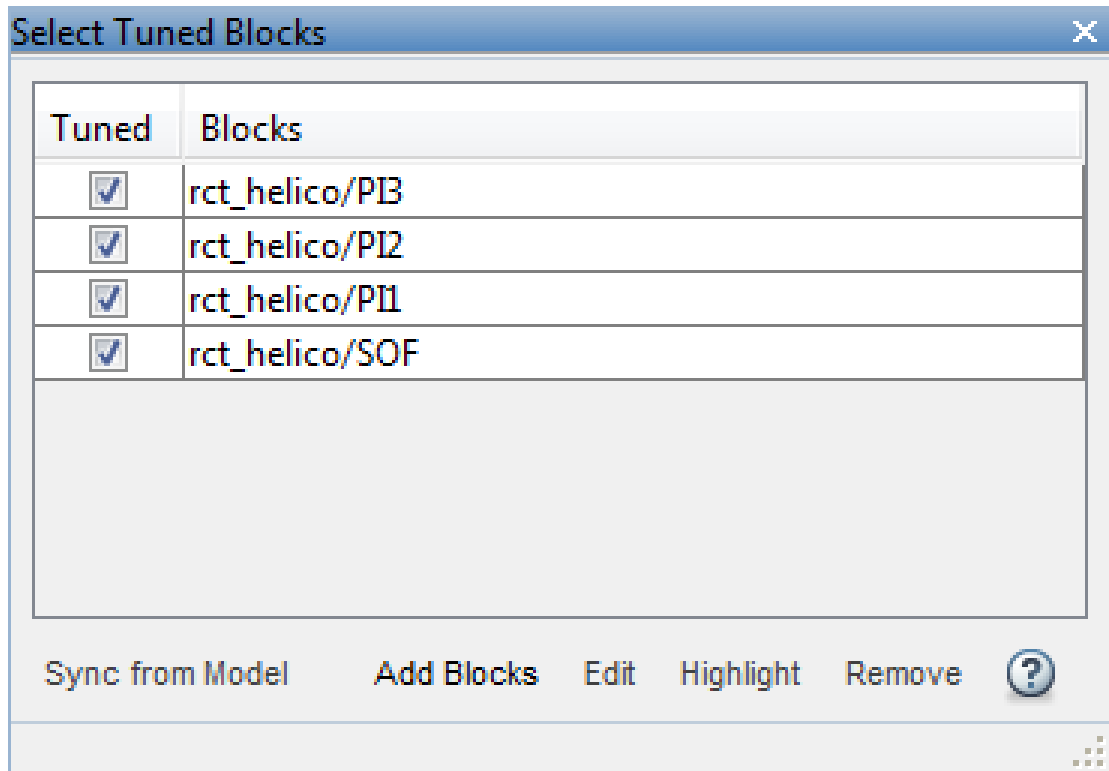
Figure 5: Editor Updated with Selected Tuned Blocks.
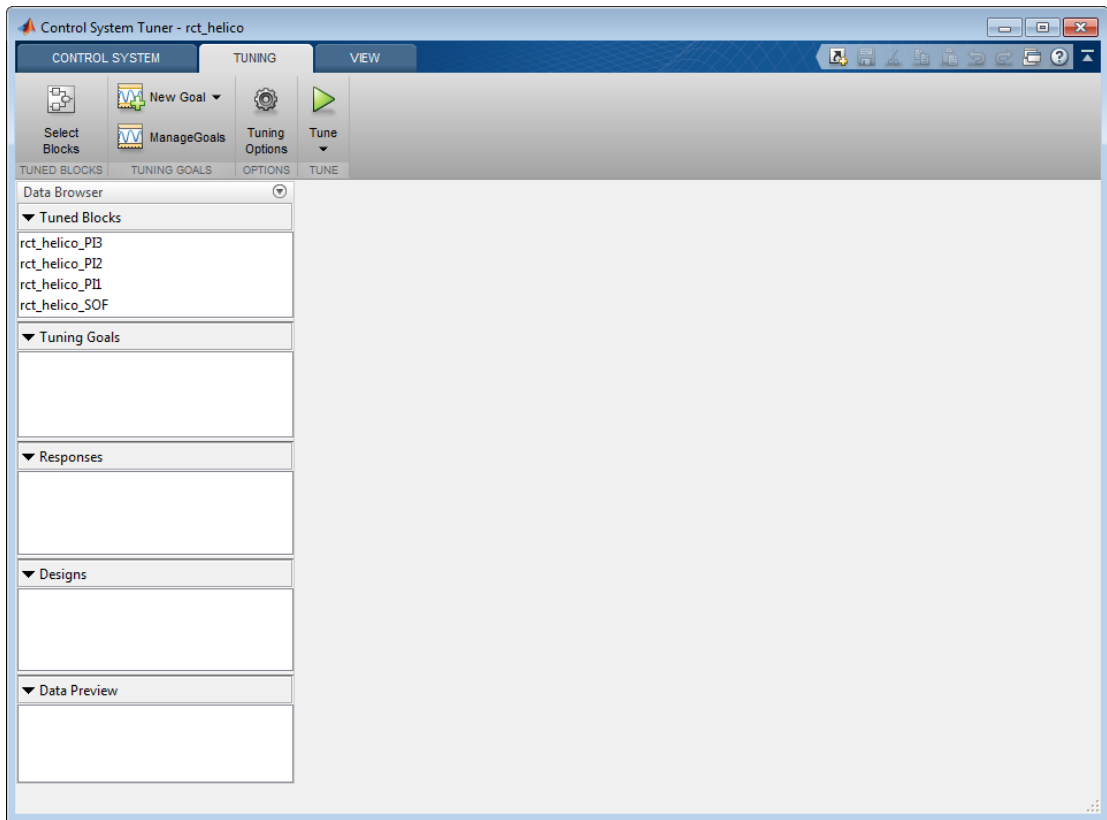
Figure 6: Updated Tuned Blocks in Control System Tuner App.

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:

Capture the design requirements using `TuningGoal` objects. We use the following requirements for this example:

- **Tracking requirement**: The response of `theta`, `phi`, `r` to step commands `theta_ref`, `phi_ref`, `r_ref` must resemble a decoupled first-order response with a one-second time constant

- **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 25 to prevent fast dynamics and jerky transients

6-271

Add all requirements from `New Goal` starting with Tracking of Step Commands requirement.
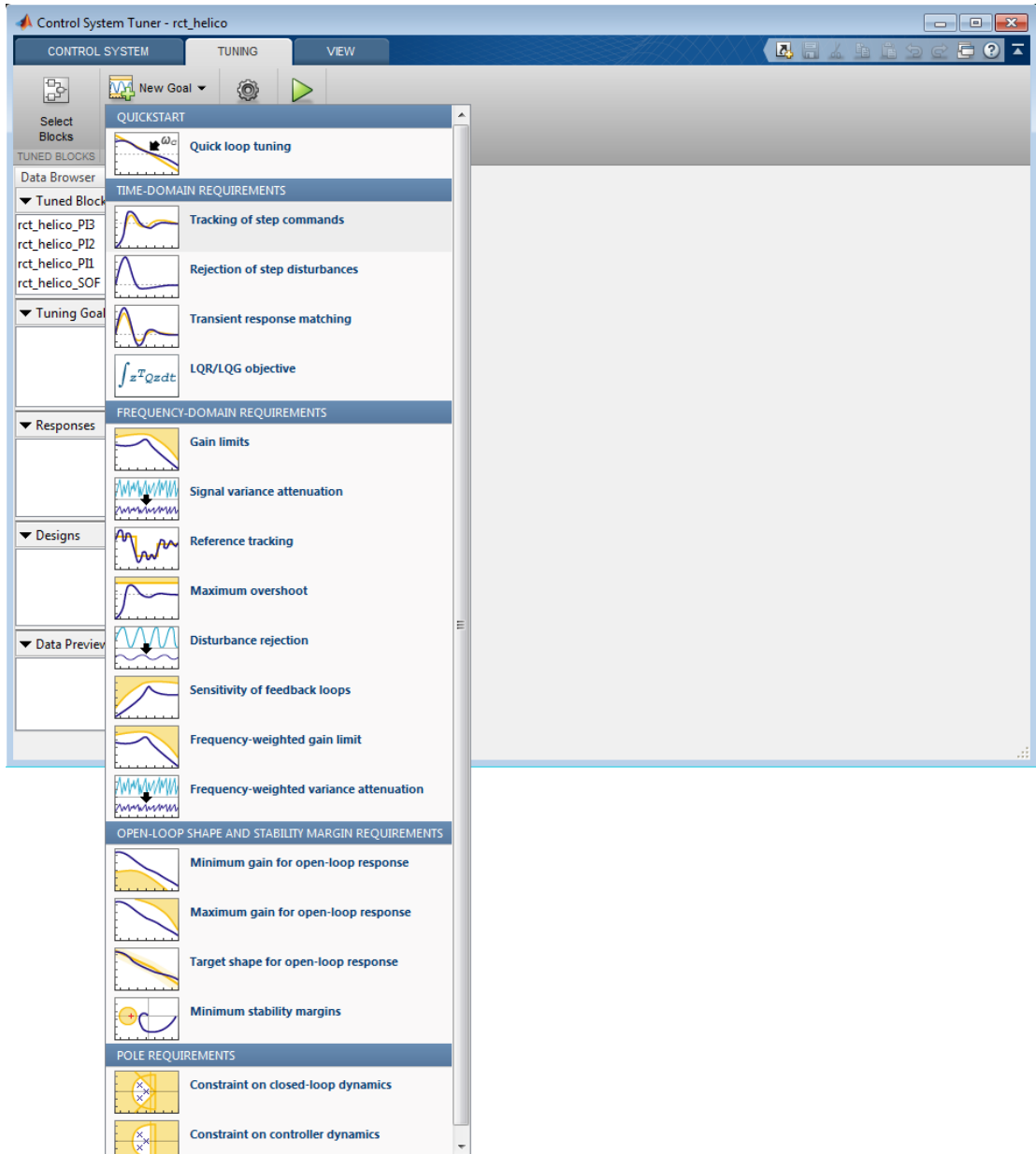
Figure 6: Available Goals for Selection in Control System Tuner App.

In the Step Tracking Goal dialog, add commands signals, `theta_ref`, `phi_ref`, `r_ref` and corresponding outputs, `theta`, `phi`, `r` from `Add signal to list` by `Select signal from Model`. Set time constant to 1 and allow mismatch less than 20% mismatch with reference model 1/(s+1).

Step Tracking Goal

Name: StepTrackingGoal1

Purpose
Make specific closed-loop step response closely match the desired response.

Step Response Selection

Specify step-response inputs:

rct_helico/theta-ref/1

rct_helico/phi-ref/1

rct_helico/r_ref/1

Specify step-response outputs:

rct_helico/Demux1/1[theta]

rct_helico/Demux1/2[phi]

rct_helico/Demux1/3[r]

Compute step response with the following loops open:

+ Add loop opening location to list

Desired Response

Specify as

◉ First-order characteristics

◯ Second-order characteristics

Figure 7: Step Tracking Goal Dialog in Control System Tuner App.

In Margins Goal dialogs, add plant inputs and outputs and set gain and phase margins requirements

Margins Goal

Name: MarginsGoal1

Purpose

Enforce specific gain and phase margins (disk margins for MIMO feedback loops).

Feedback Loop Selection

Measure stability margins at the following locations:

rct_helico/Mux3/1[u]

➕ Add signal to list

Measure stability margins with the following loops open:

➕ Add loop opening location to list

Desired Margins

Gain margin (dB):   5

Phase margin (degrees):   40

Options

Enforce goal in frequency range   [0 Inf]

D scaling order   0

Figure 8: Margins Goal at Plant Input Dialog in Control System Tuner App.

Margins Goal   ✕

Name: | MarginsGoal2

Purpose

Enforce specific gain and phase margins (disk margins for MIMO feedback loops).

Feedback Loop Selection

Measure stability margins at the following locations:

rct_helico/Helicopter/1[y]

➕ Add signal to list

Measure stability margins with the following loops open:

➕ Add loop opening location to list

Desired Margins

Gain margin (dB): | 5

Phase margin (degrees): | 40

Options

Enforce goal in frequency range | [0 Inf]

D scaling order | 0

Figure 9: Margins Goal at Plant Output Dialog in Control System Tuner App.

Limit on fast dynamics by putting constraint on closed-loop dynamics and selecting Poles Goal.

Poles Goal ✕

Name: PolesGoal1

Purpose

Constrain the dynamics of the closed-loop system, specific feedback loops, or specific open-loop configurations.

Feedback Configuration

Compute poles of:

○ Entire system

○ Specific feedback loop(s)

Compute poles with the following loops open:

＋ Add loop opening location to list

Pole Location

Keep poles inside the following region:

Minimum decay rate: 0

Minimum damping: 0

Maximum natural frequency: 25

Options

Enforce goal in frequency range: [0 Inf]

Figure 10: Poles Goal Dialog in Control System Tuner App.

The plots for specified requirements are created in Control System Tuner App and `Tuning Goals` section of Data Browser on the left side is updated.
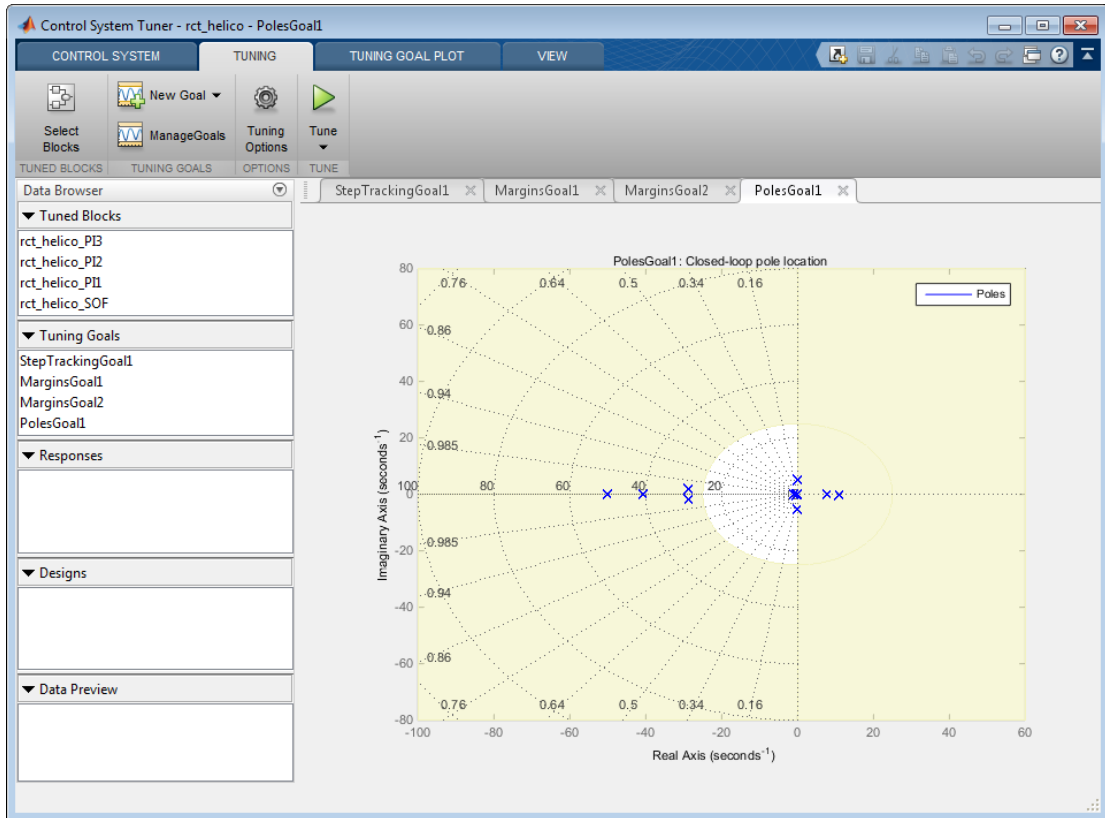
Figure 11: Updated Control System Tuner App with Tuning Goals Created.

You can now tune all controller parameters by `Tune` button in `Control System Tuner` App. The parameters of tuned blocks and the plots of tuning goals in `Control System Tuner` App are updated. The tuning report is created.
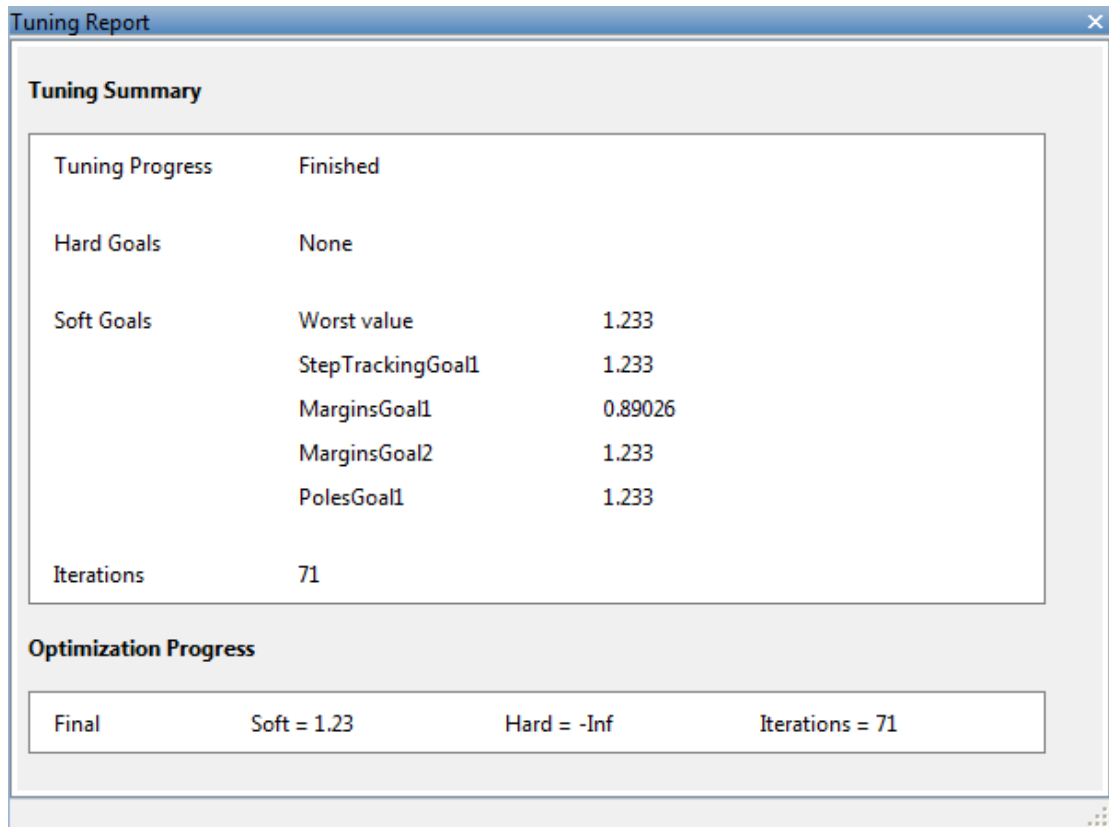
Figure 12: Tuning Report in Control System Tuner App.

The final value is close to 1 so the requirements are nearly met. Check the tuned step responses of tracking requirements.
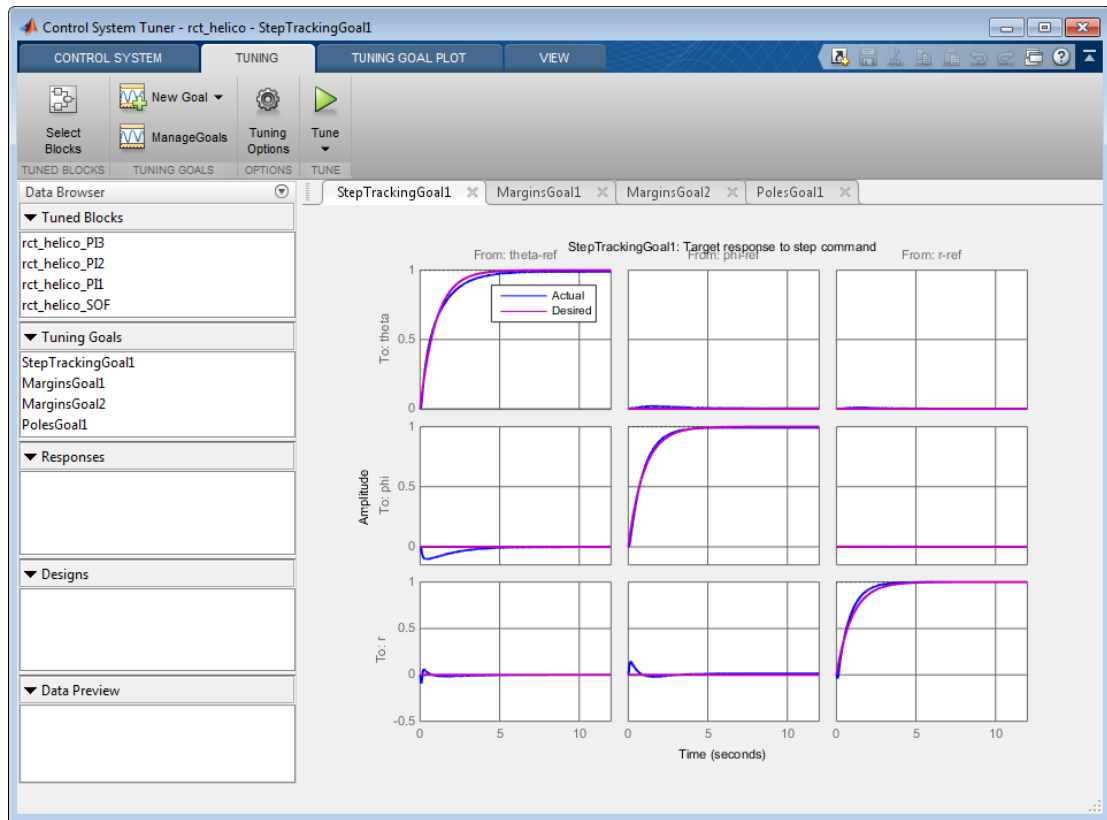
Figure 13: Control System Tuner App After Tuning.

The rise time is about two seconds with no overshoot and little cross-coupling. You can see the plots of each requirement, including a visual assessment of the multivariable stability margins (see `loopmargin` for details).
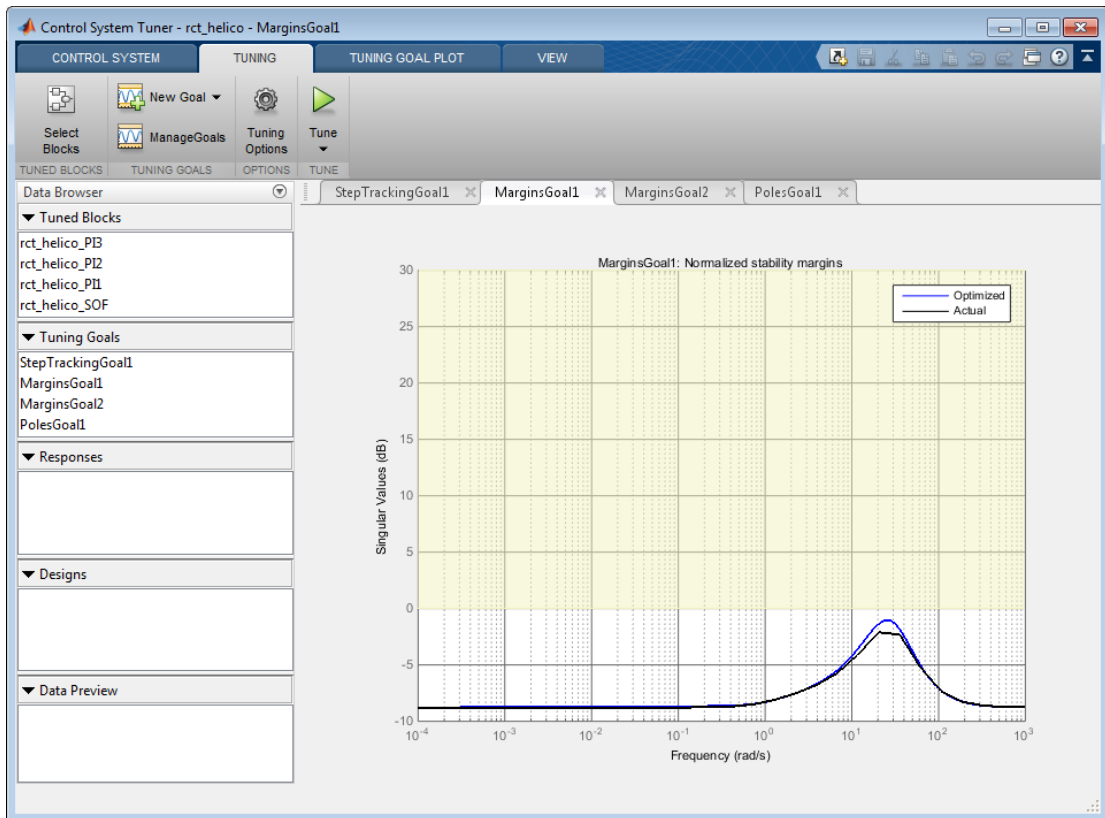
Figure 14: Control System Tuner App with Updated Margins Goal Plot.

You can store the current design by clicking `Store` from `Control System` Tab and inspect the tuned values of the PI controllers and static output-feedback gain by double clicking on `Design1`.
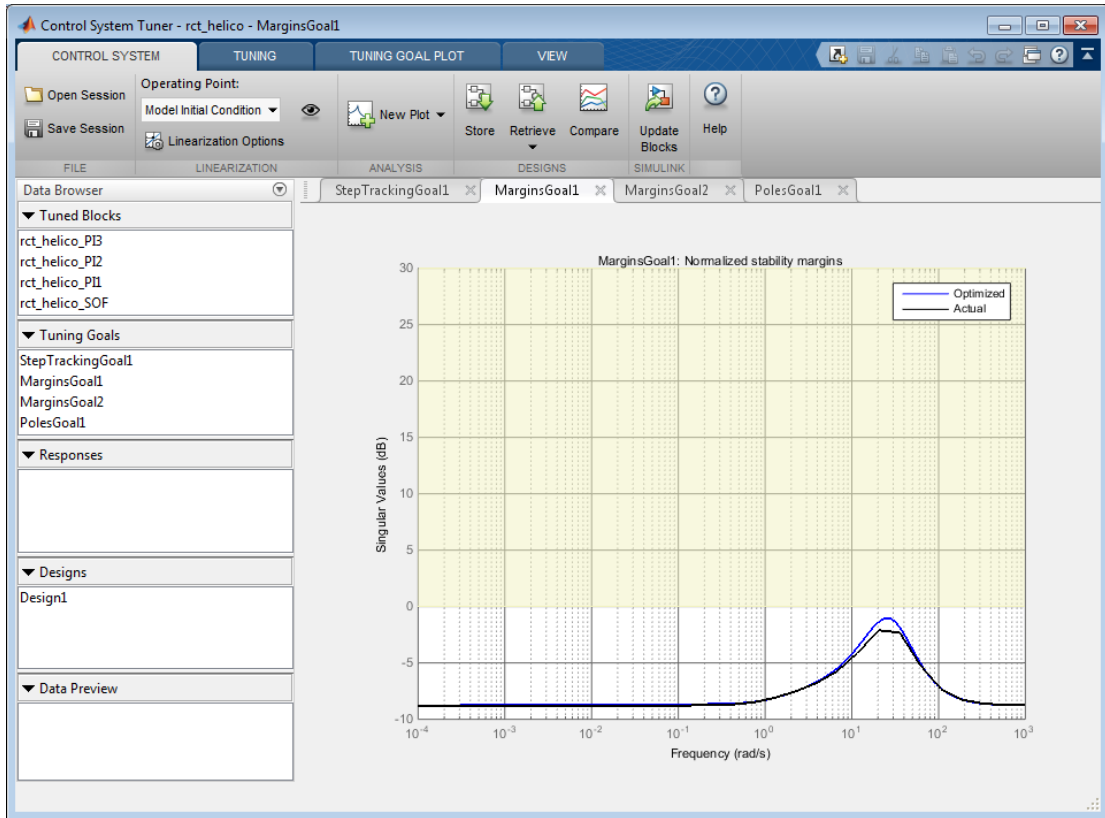
Figure 15: Control System Tuner App with Stored Design1.

Design Summary ✕

```
Design Data
Name: Design1
Sample Time: 0
Value:

rct_helico_PI3 =

              1
   Kp + Ki * ---
              s

   with Kp = 0.179, Ki = -2.51


rct_helico_PI2 =

              1
   Kp + Ki * ---
              s

   with Kp = -0.0853, Ki = -1.94


rct_helico_PI1 =

              1
   Kp + Ki * ---
              s

   with Kp = 0.862, Ki = 1.41


rct_helico_SOF =

   d =
              u1         u2          u3          u4          u5
     y1     1.534    -0.08073    0.006617      0.6412     0.02703
```
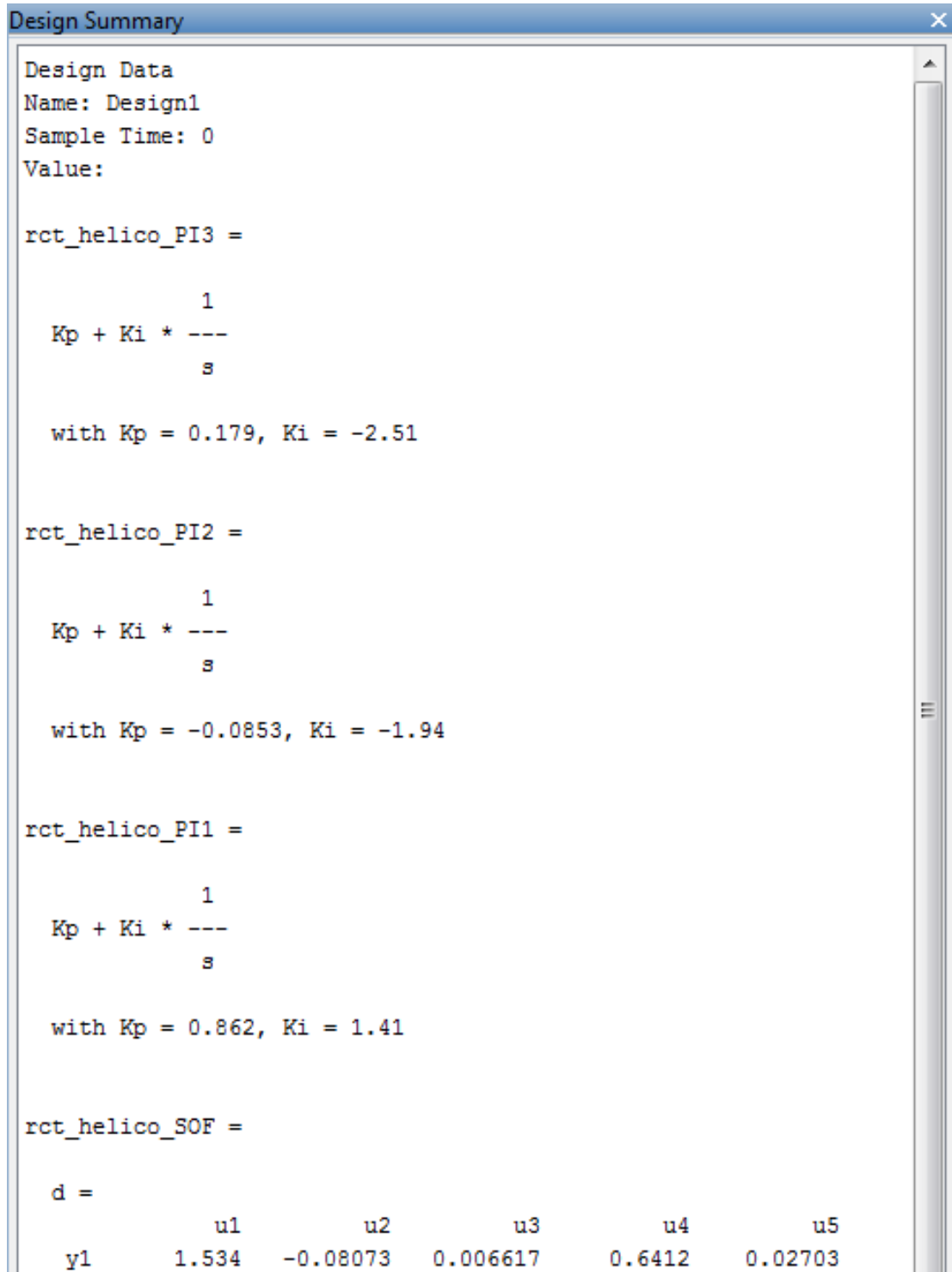
Figure 16: Tuned Values of PI Controllers and Static Output-Feedback Gain.

Control of a Linear Electric Actuator Using CONTROL SYSTEM TUNER

This example shows how to use Control System Tuner App™ of 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')

Linear Electric Actuator (System-Level Model)

This linear actuator consists of a DC motor driving a worm gear which in turn drives a lead screw to produce linear motion. This model is a system-level model suitable for motor current and speed controller tuning. The Simulation Mode parameter is set to Averaged for both the Controlled PWM Voltage and H-Bridge blocks, resulting in fast simulation and correct linearization.

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.

Current Controller

Speed Controller

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 `Control System Tuner App` to jointly tune both feedback loops. First, open `Control System Tuner App` from Simulink Analysis Menu.
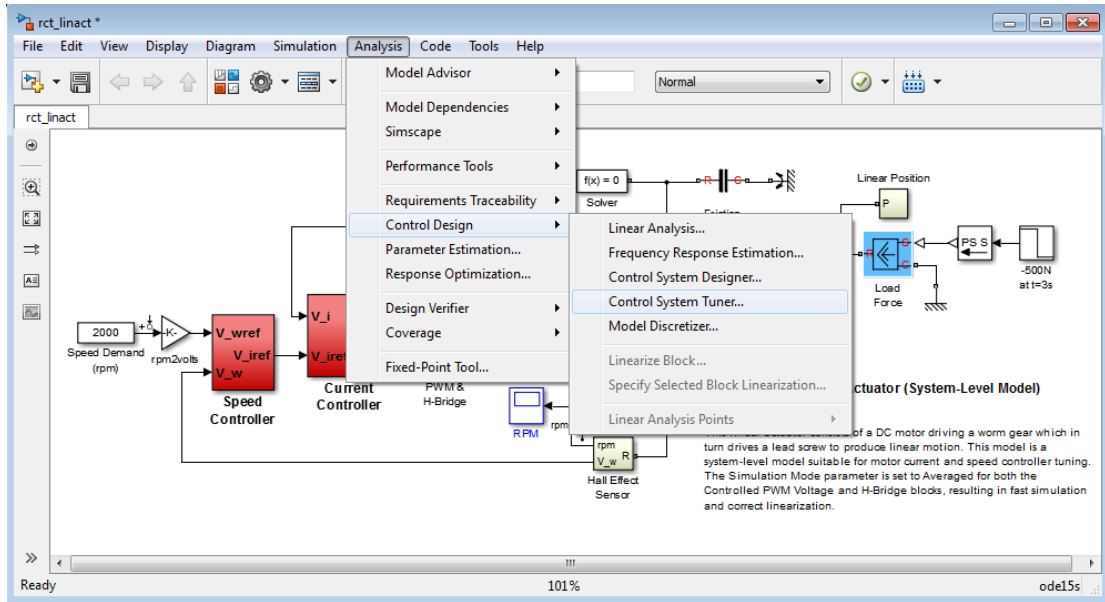
Figure 3: Opening Control System Tuner App.

This opens Control System Tuner App.
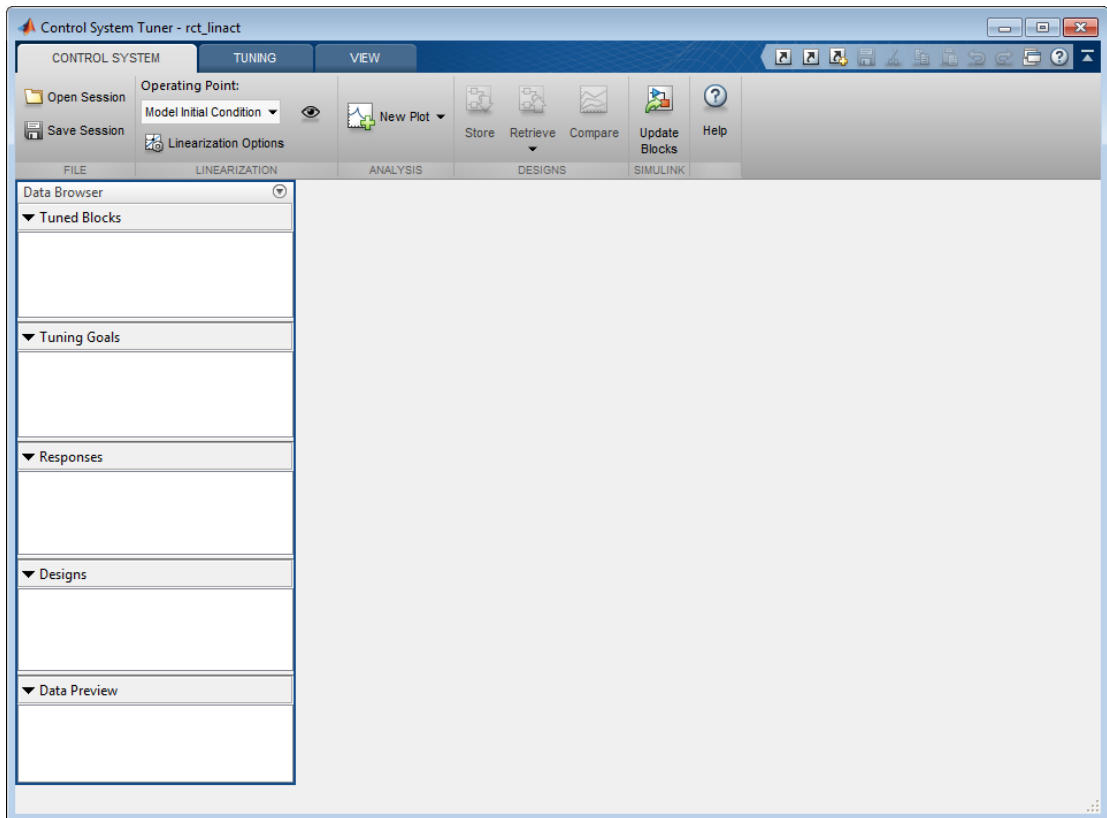
Figure 4: Control System Tuner App.

You linearize the model at t=0.5 to avoid discontinuities in some derivatives at t=0. You can set the operating point in `Linearize At...`.
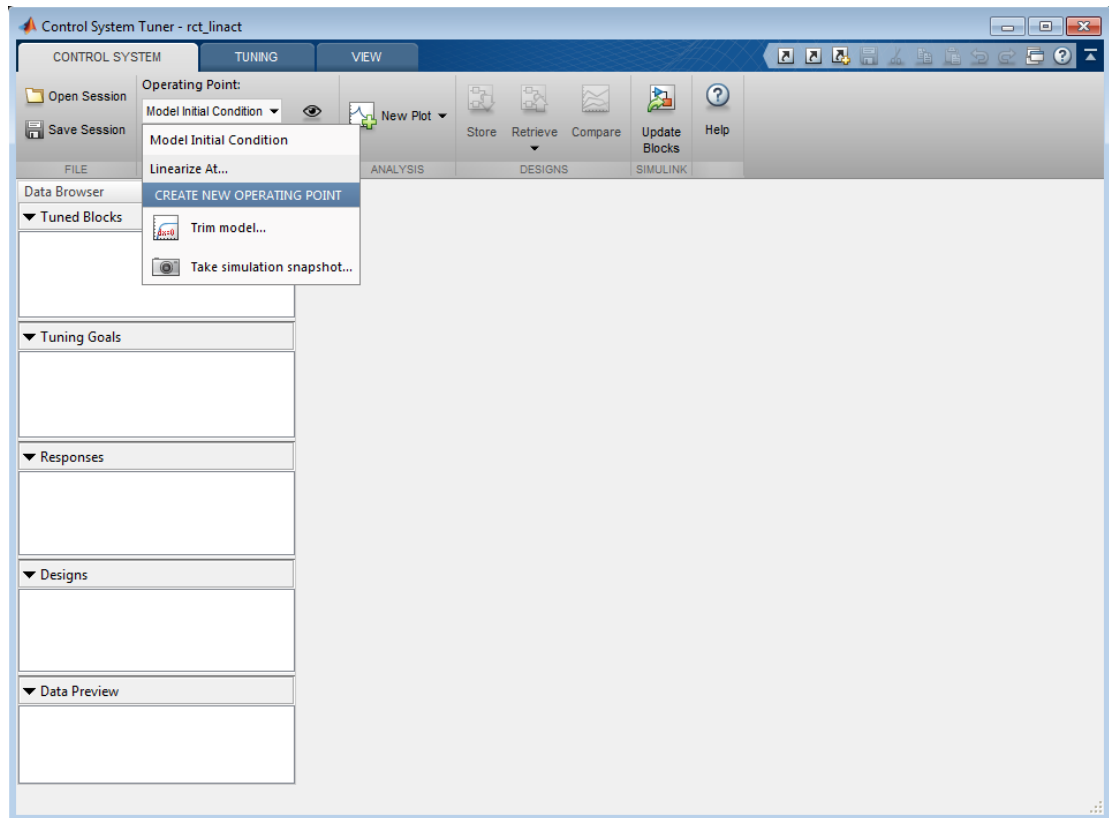
Figure 5: Setting Operating Point for Linearization.

Set the linearization snapshot time at t=0.5.
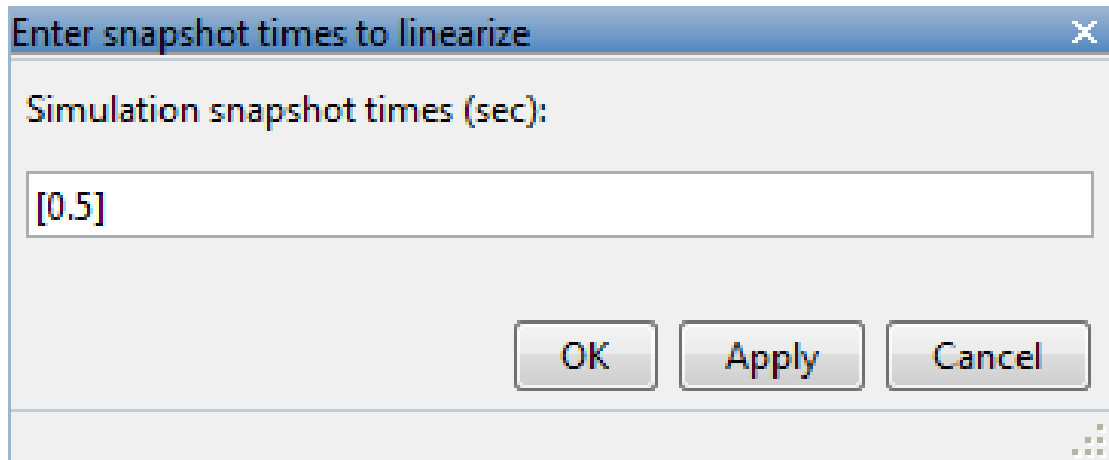
Figure 6: Setting the Linearization Snapshot Time.

In order to set the tuned blocks of the control system, open `Select Blocks` from `Tuning` tab.
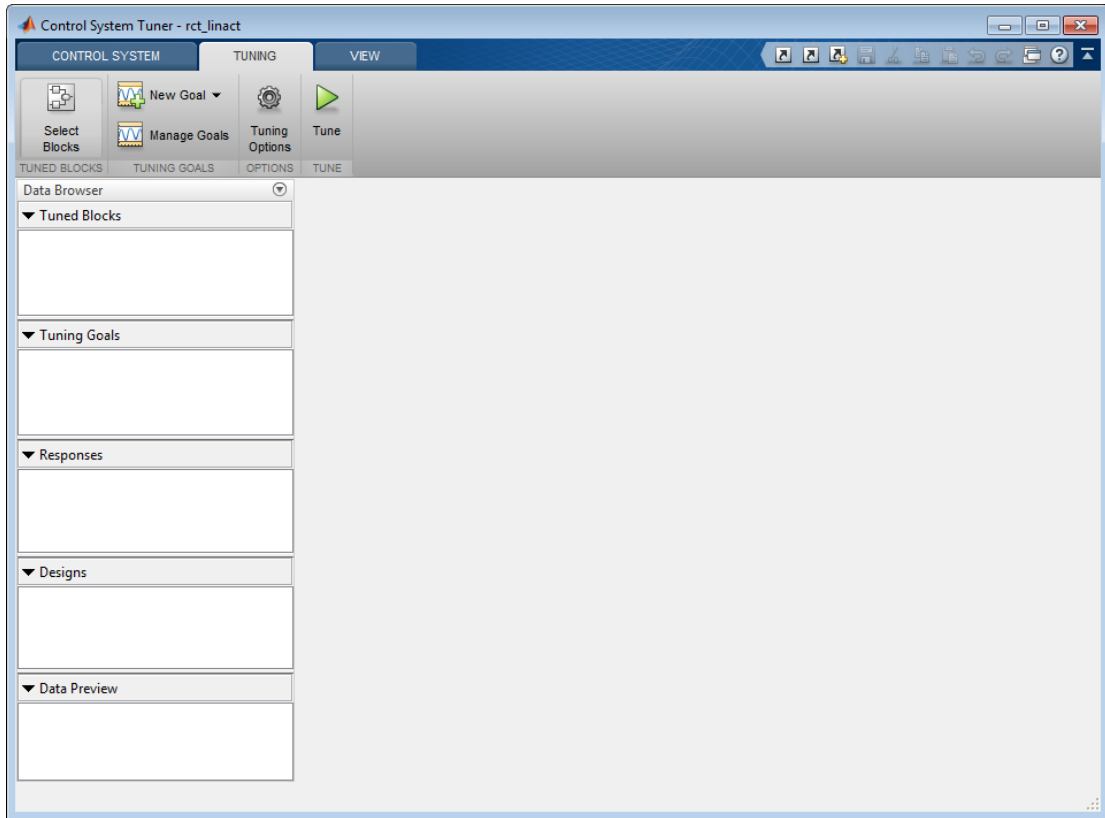
Figure 7: Tuning Tab of Control System Tuner App.

This shows the editor for tuned blocks where you can `Add Blocks`.
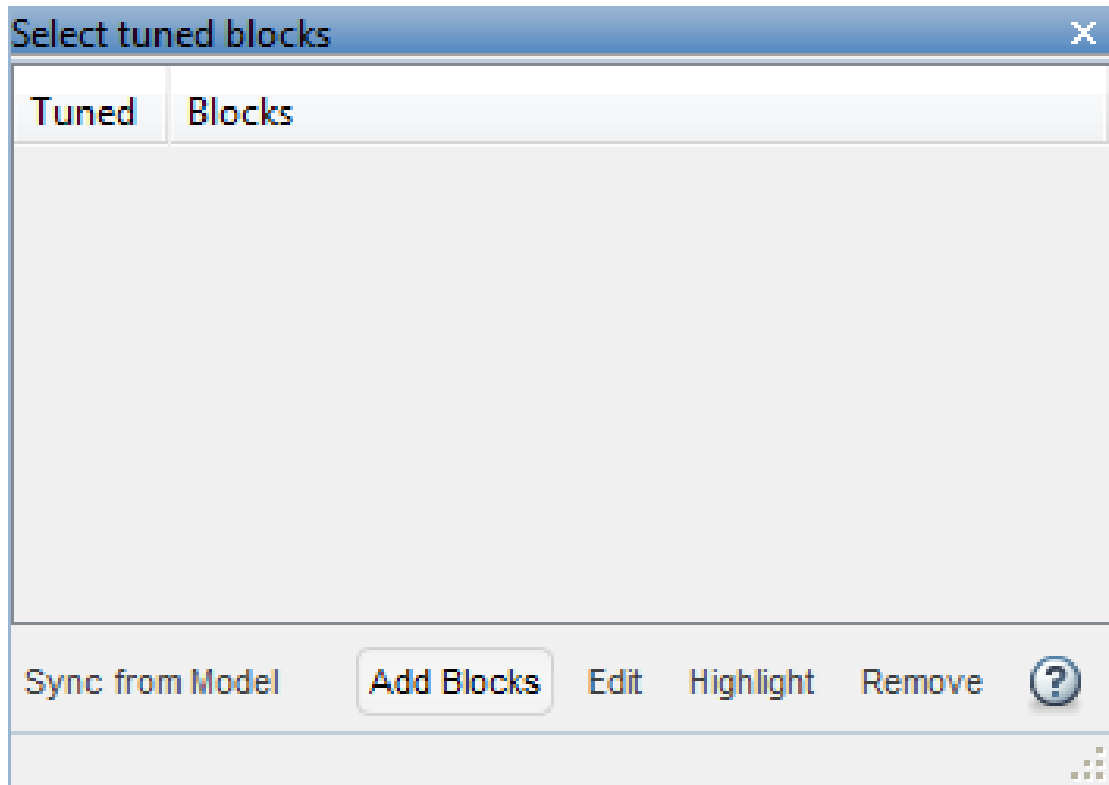
Figure 8: Editor for Tuned Blocks.

Set the tuned blocks `Current PID` and `Speed PID` by navigating through the tree on the left.
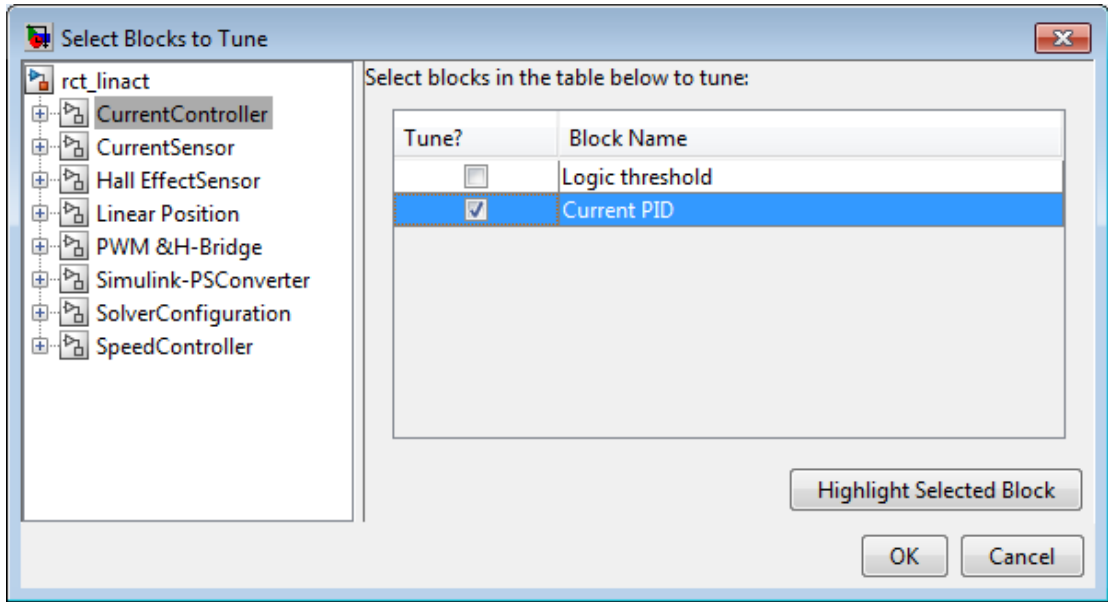
Figure 9: Selecting Tuned Block `Current PID`.
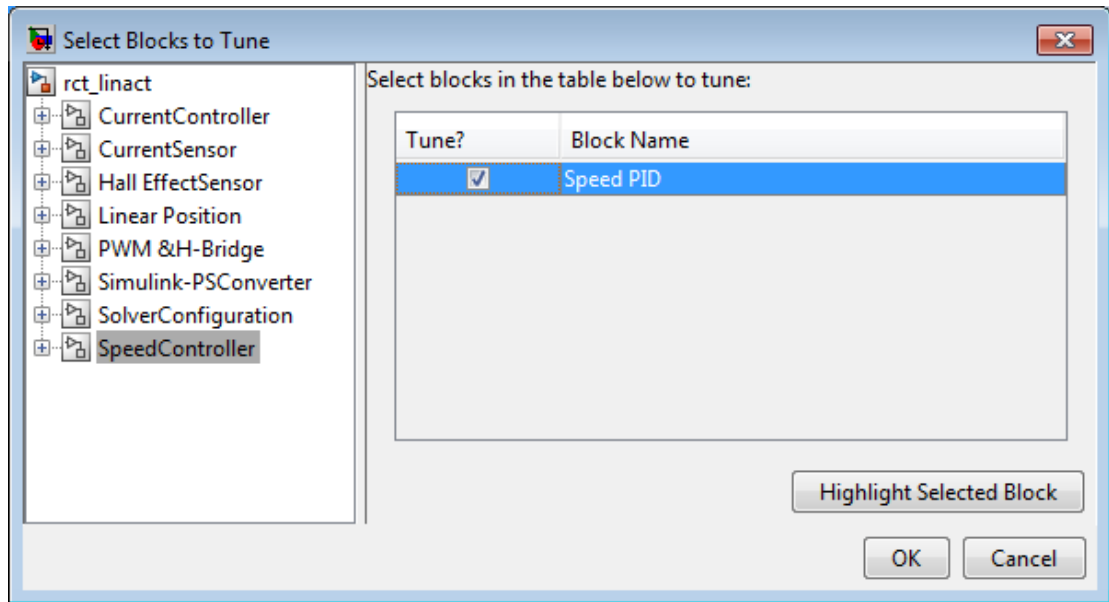
Figure 10: Selecting Tuned Block Speed PID.

Selected tuned blocks Current PID and Speed PID show in the editor for tuned blocks.
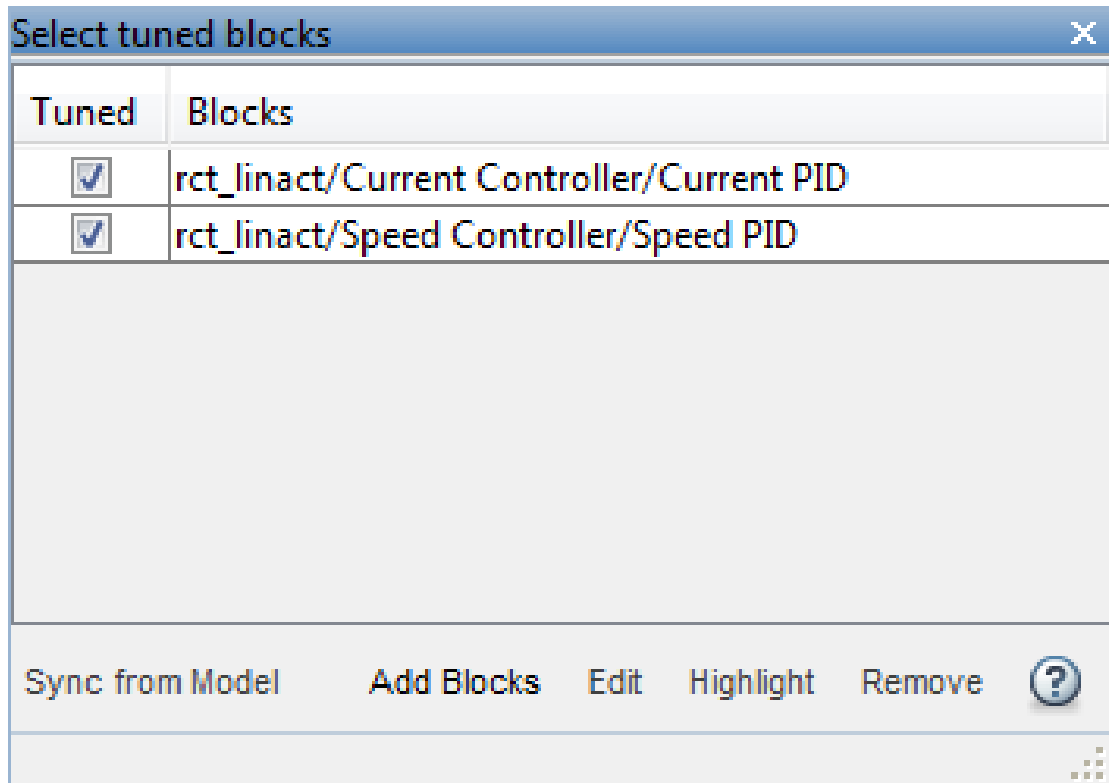
Figure 11: Editor Updated with Selected Tuned Blocks.

They also appear in the `Tuned Blocks` section of Data Browser on the left side of Control System Tuner App.
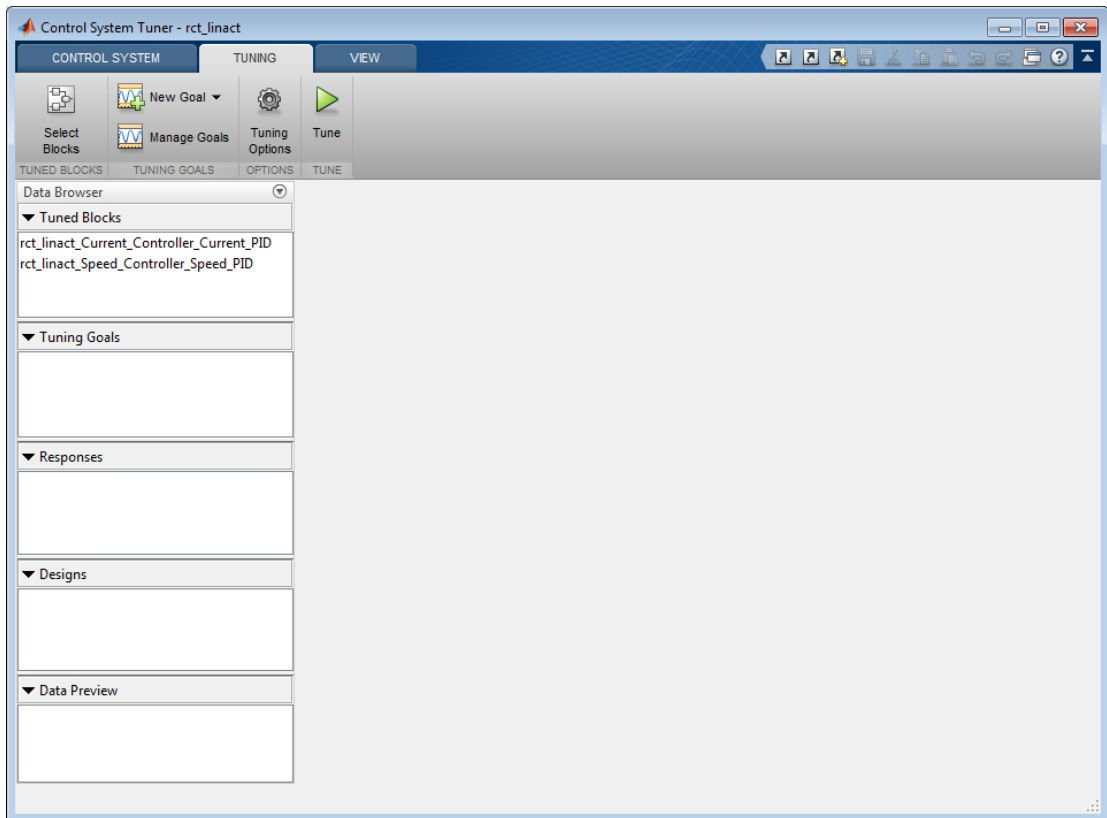
Figure 12: Updated Tuned Blocks in Control System Tuner App.

Next specify the tracking goal that the DC motor should follow a 2000 rpm speed demand in 0.1 seconds. See different types of goals under `New Goal` and select `Reference Tracking`.
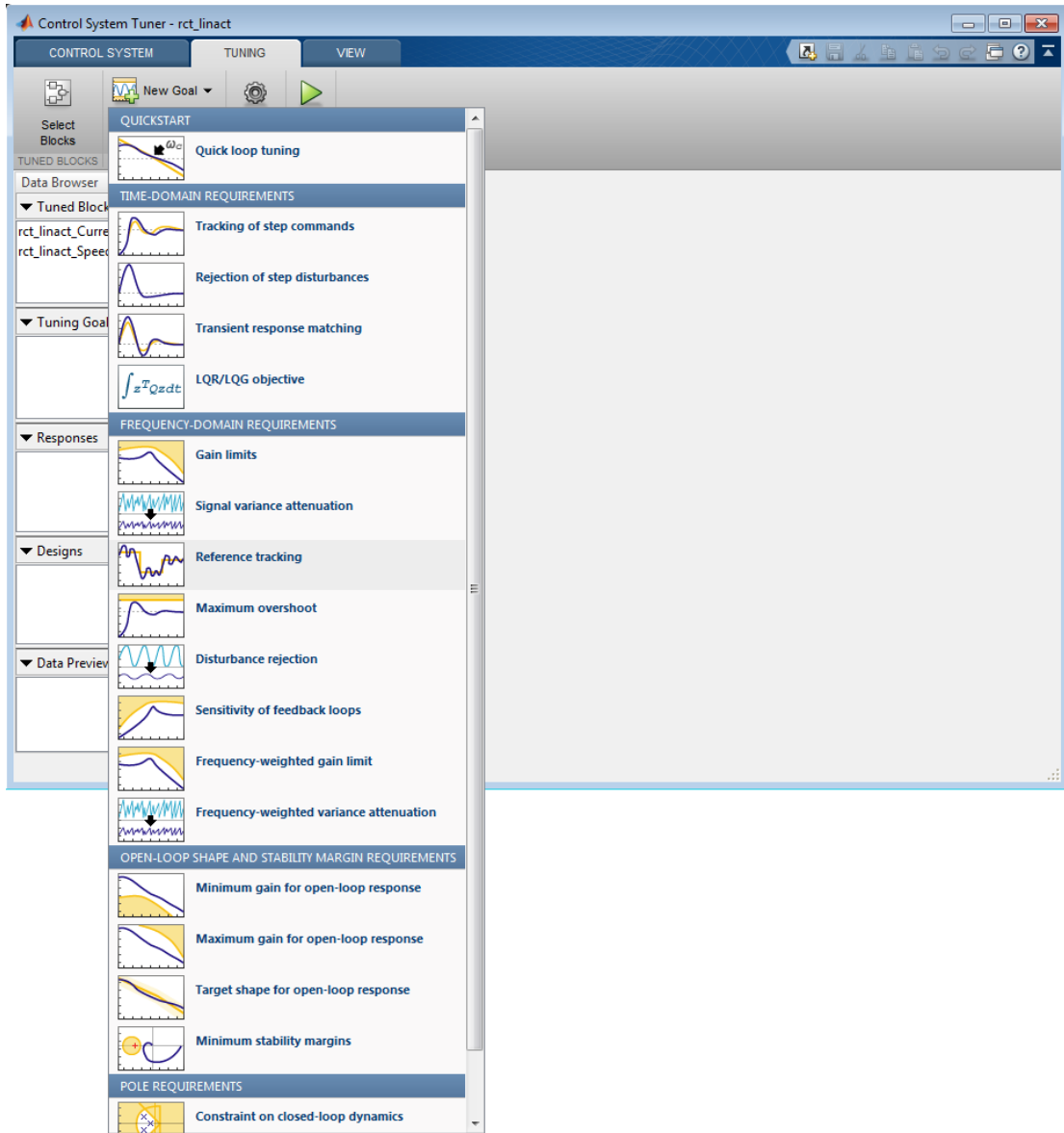
Figure 13: Available Goals for Selection in Control System Tuner App.

Name the tracking goal as `TR`, specify the tracking goal from the reference input `rct_linact/Speed Demand(rpm)/1` to the reference-tracking output `rct_linact/Hall Effect Sensor/1[rpm]` with the response time 0.1 seconds.

Reference Tracking Goal   ✕

Name: TR

Purpose

Follow reference commands with prescribed performance and
fidelity. Limit cross-coupling in MIMO systems.

Response Selection

Specify reference inputs:

rct_linact/Speed Demand (rpm)/1

➕ Add signal to list

Specify reference-tracking outputs:

➕ Add signal to list

    rct_linact/Current Sensor/1[i]

    rct_linact/Hall Effect Sensor/1[rpm]

    rct_linact/Speed Controller/Speed PID/1

    rct_linact/Speed Demand (rpm)/1

    Select signal from model

oops open:

Tracking Performance

Specify as

  ◉ Response time, DC error, and peak error

  ◯ Maximum error as a function of frequency

Figure 13: Reference Tracking Dialog in Control System Tuner App.

The plot for specified tracking goal appears in Control System Tuner App and `Tuning Goals` section of Data Browser on the left side is updated.
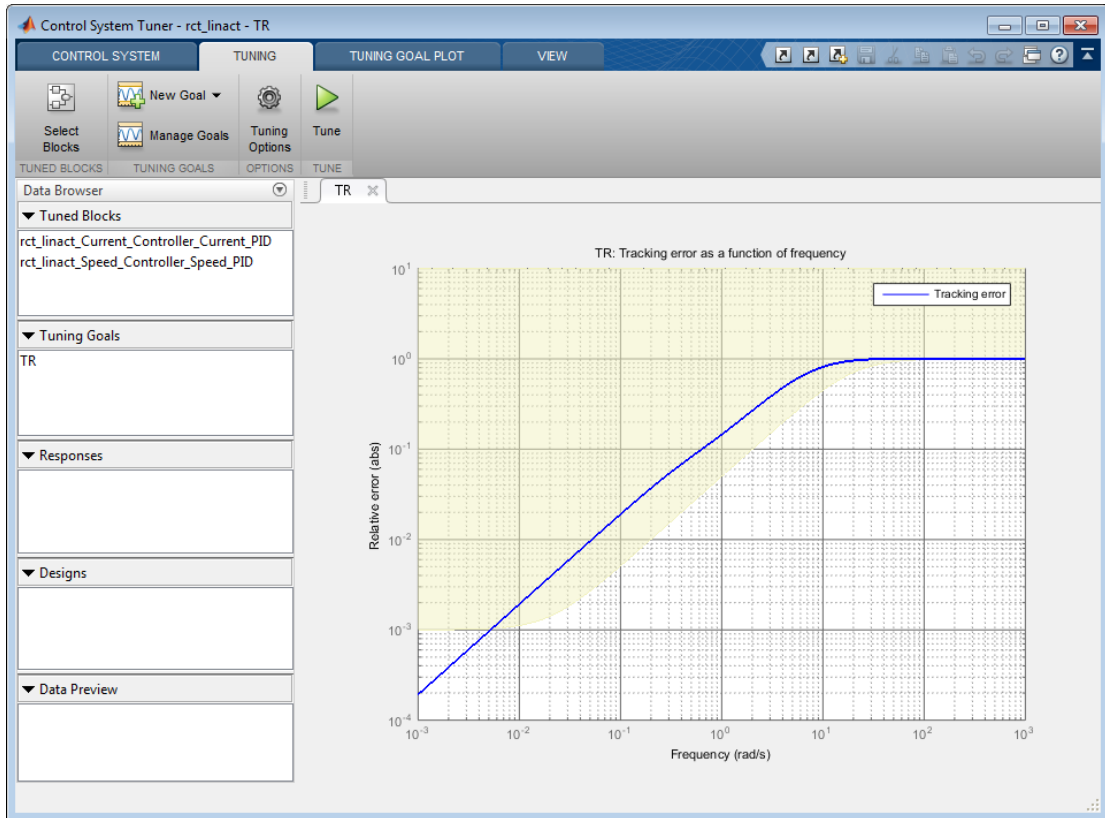
Figure 14: Tracking Tuning Goal in Control System Tuner App.

You can now tune the proportional and integral gains with `Control System Tuner App` from clicking `Tune` button. The plot for tracking goal is updated
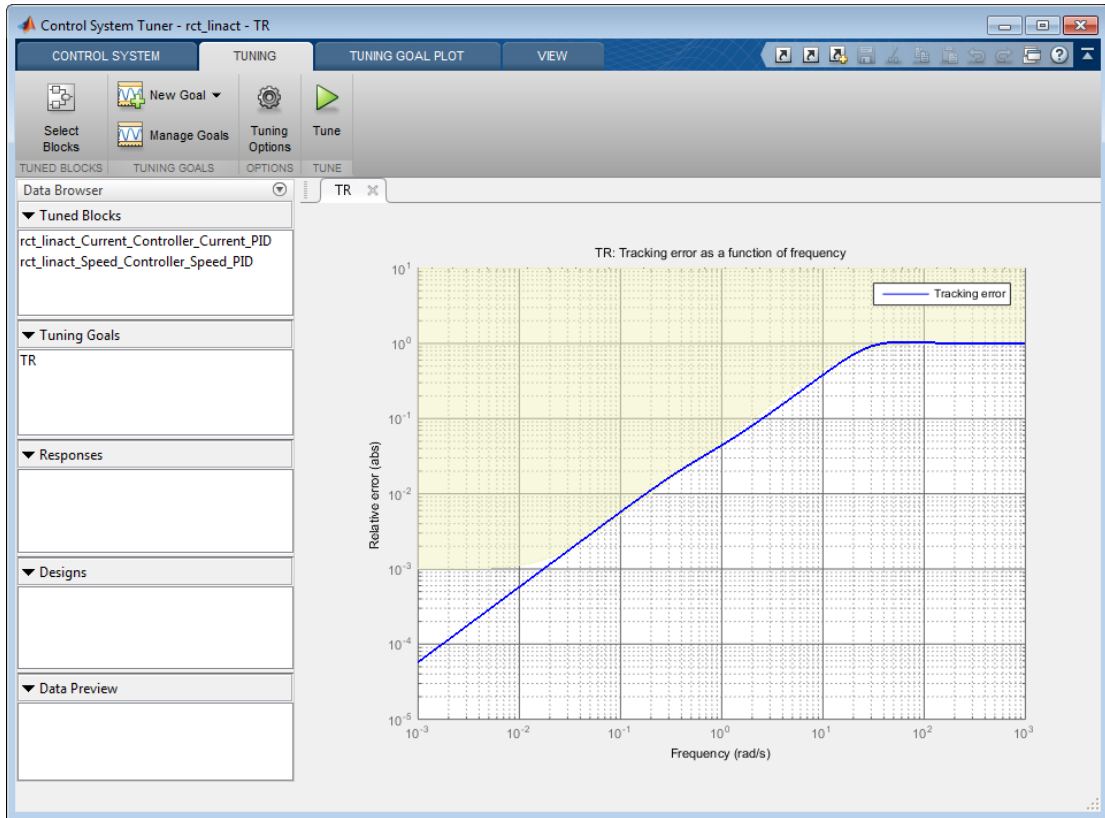
Figure 15: Updated Tracking Goal Plot with Tuned Blocks in Control System Tuner App.

and you see `Tuning Report` which gives additional information about the tuning.
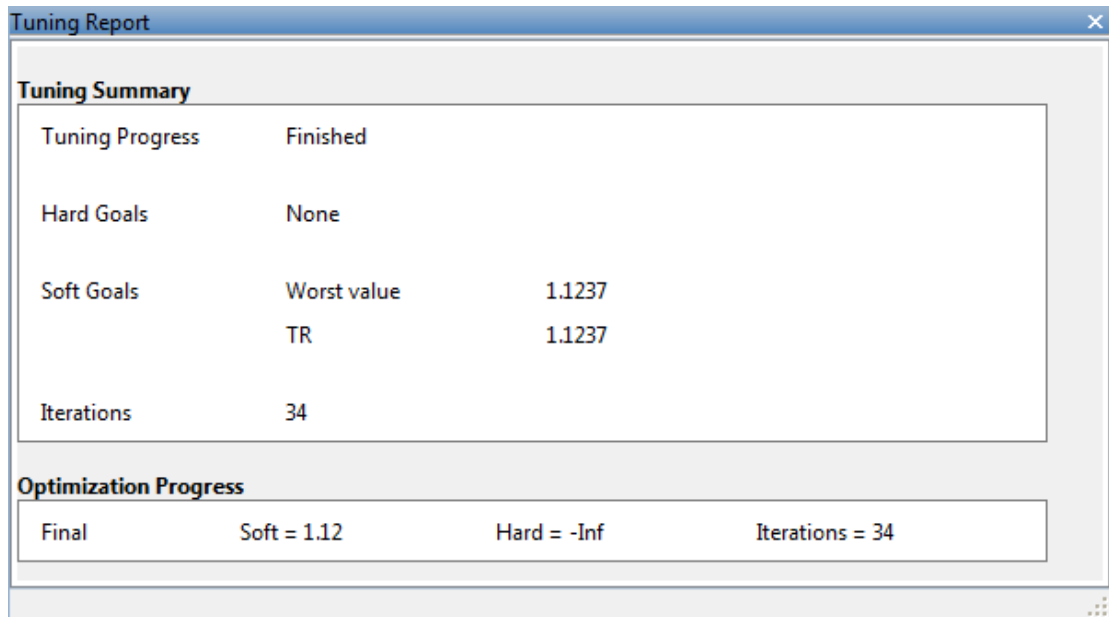
Figure 16: Tuning Report in Control System Tuner App.

Tuned blocks are updated with the tuned gain values. To validate this design, plot the closed-loop response from speed demand to speed from `New Plot` of `Control System` Tab.
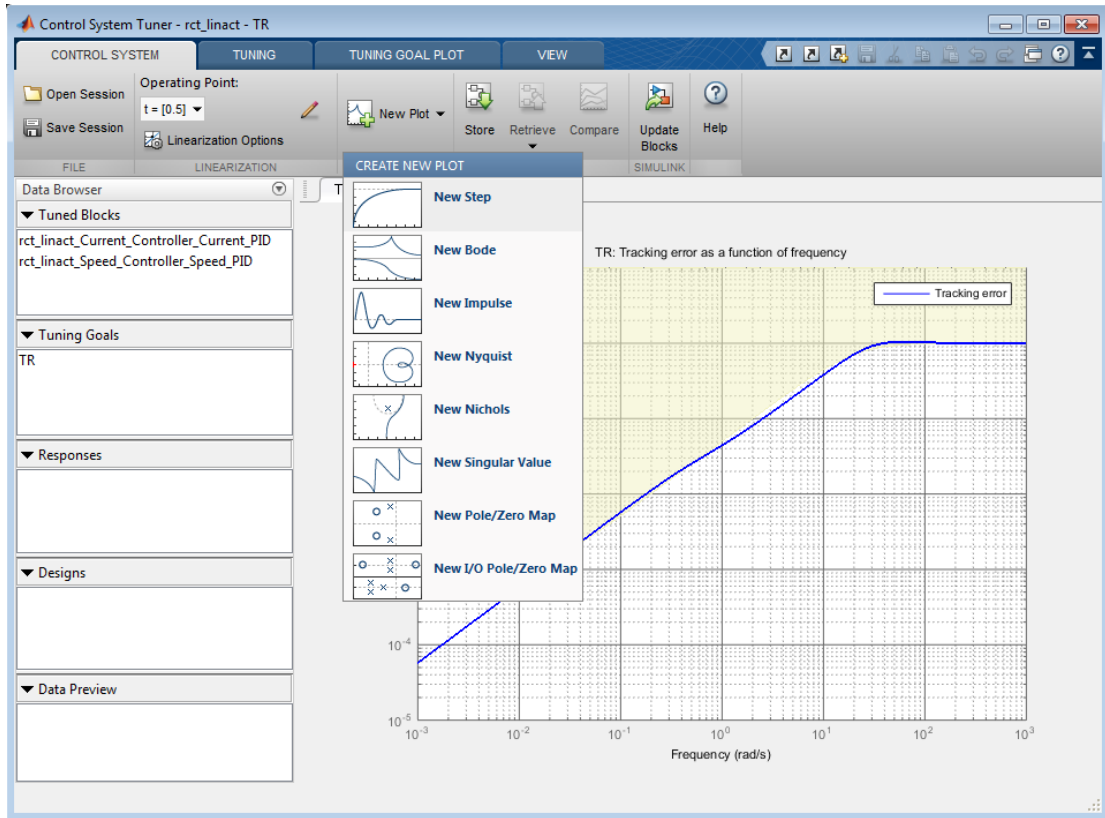
Figure 17: New Plot in Control System Tuner App.

Specify the closed-loop response from speed demand to speed by the step plot dialog.
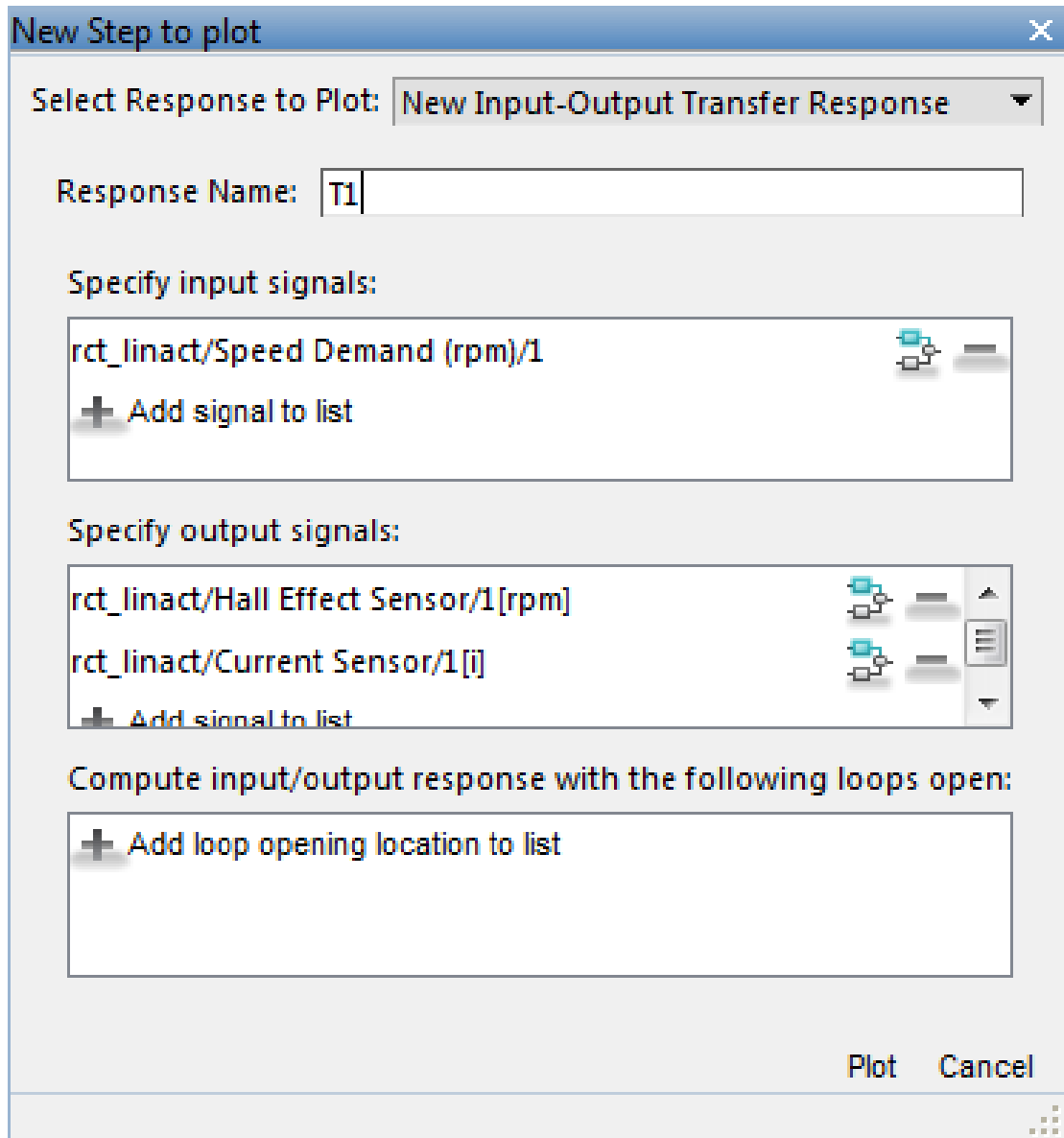
Figure 18: Step Plot Dialog in Control System Tuner App.

You see the step plot of the response in `Control System Tuner App`.
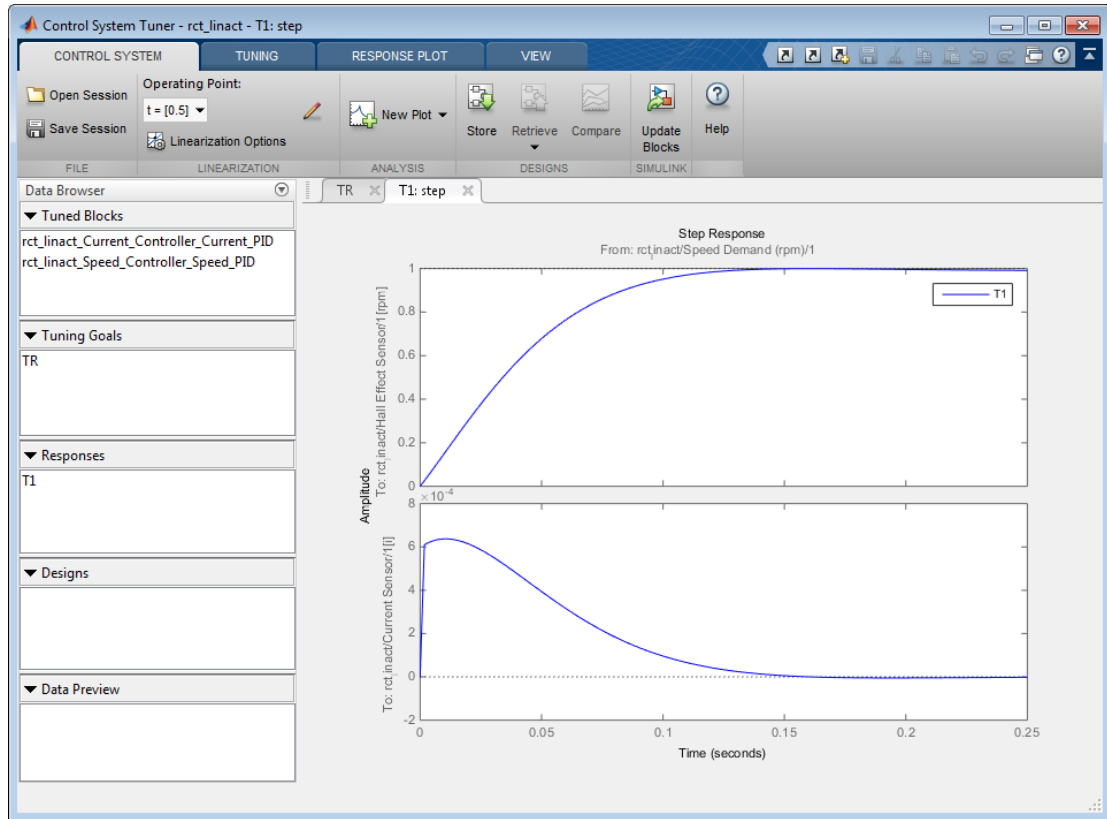
Figure 19: Step Plot in Control System Tuner App.

The response looks good in the linear domain so first store the current design by clicking `Store` and push the tuned gain values to Simulink by clicking `Update Blocks` and further validate the design in the nonlinear model.
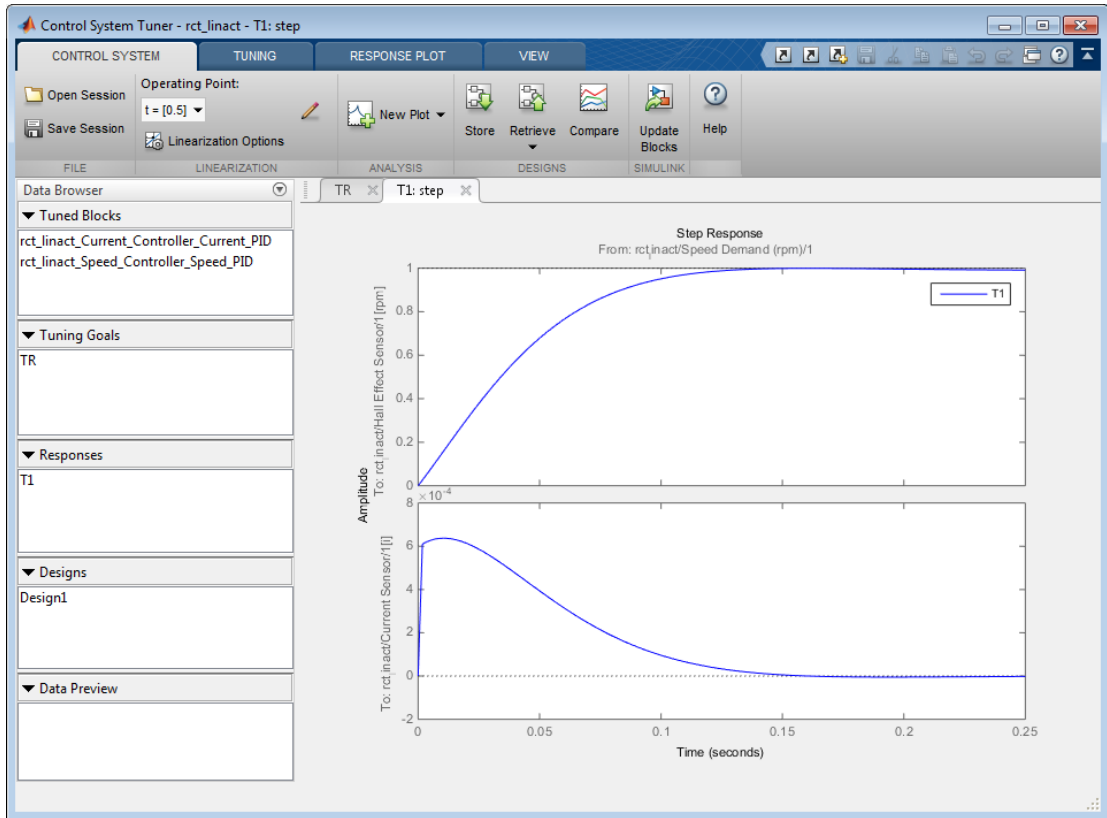
Figure 20: Stored Values of Tuned Blocks in Control System Tuner App.

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).
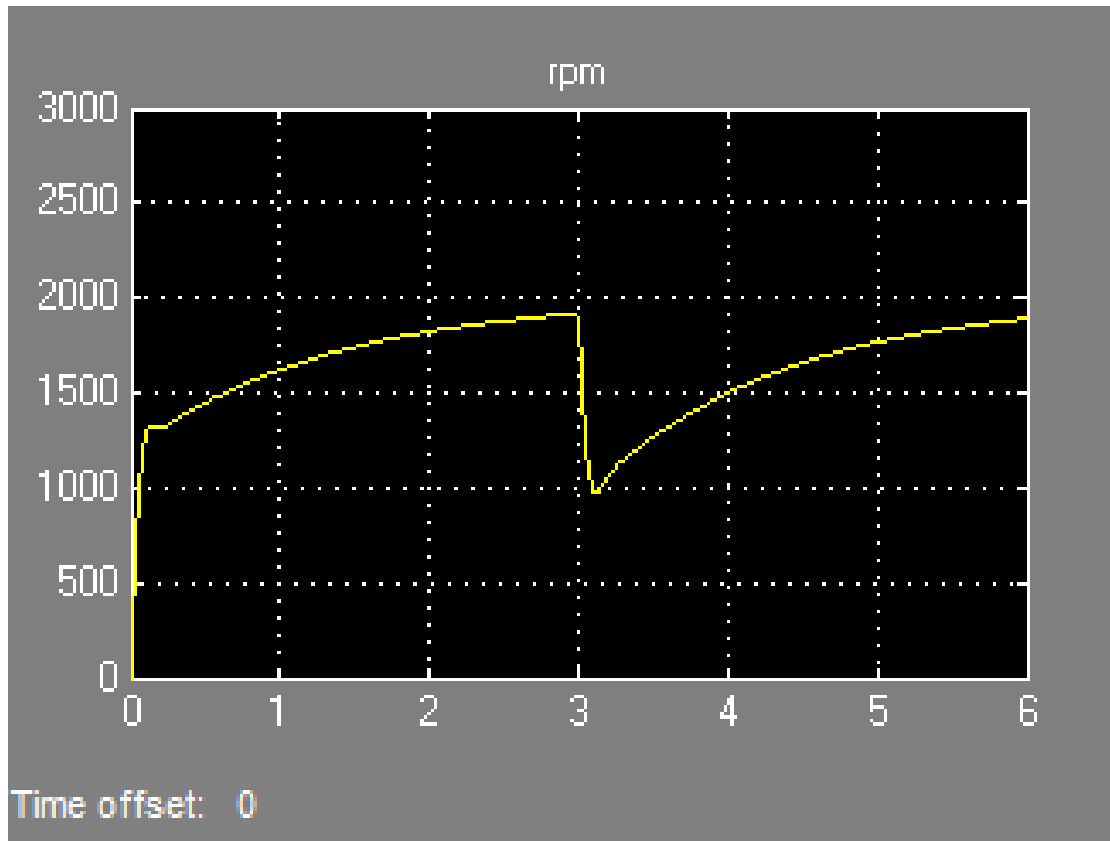
Figure 21: Nonlinear Simulation of Tuned Controller.
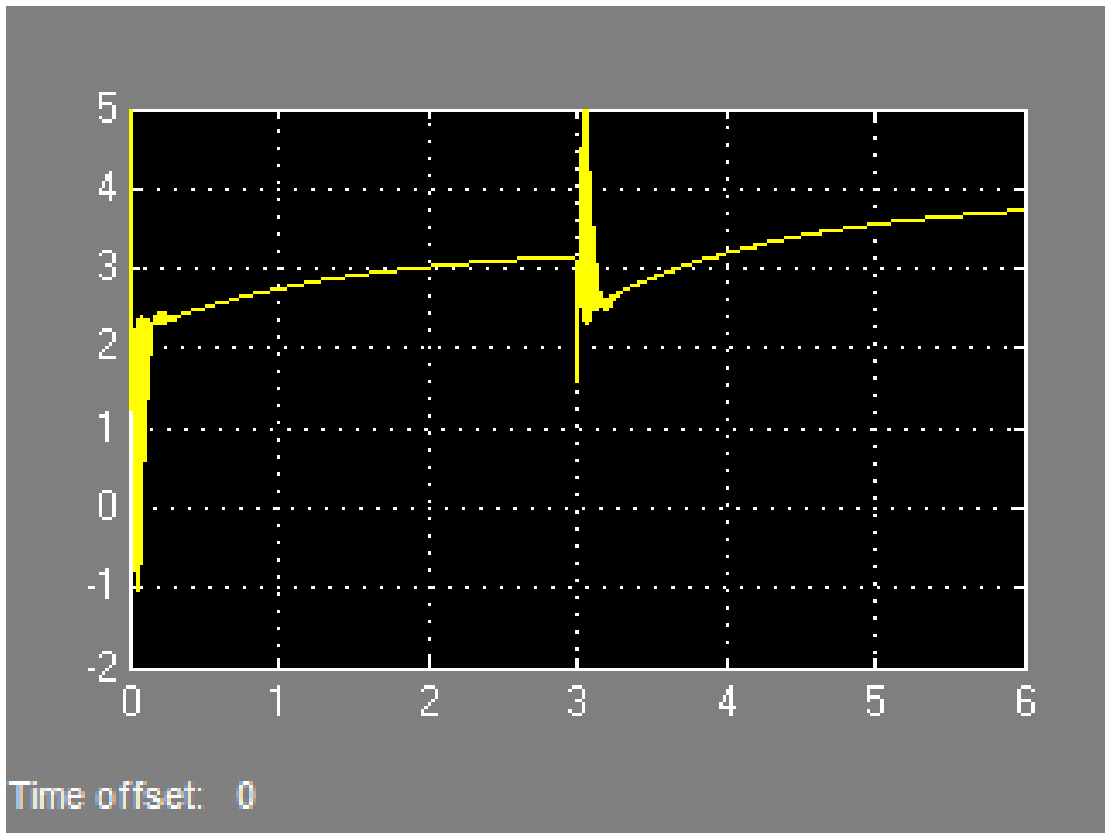
Figure 22: Current Controller Output (limited to plus/minus 5).

Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves `systune` 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.

Limit gain from speed demand to control signals to avoid saturation by specifying two new goals from Tuning tab. You need to select control signals from Simulink model since they are not defined previously.

Gain Goal

Name: MG1

Purpose

Limit gain of a specific input/output transfer function.

I/O Transfer Selection

Specify input signals:

rct_linact/Speed Demand (rpm)/1

+ Add signal to list

Specify output signals:

rct_linact/Speed Controller/Speed PID/1

+ Add signal to list

Compute input/output gain with the following loops open:

+ Add loop opening location to list

Options

Limit gain to	0.001
Stabilize I/O transfer	Yes
Enforce goal in frequency range	[0 Inf]
Adjust for signal amplitude	No

Figure 23: Gain Goal Dialog from Speed Demand to Control Signal of Speed PID.

Gain Goal

Name: MG2

Purpose

Limit gain of a specific input/output transfer function.

I/O Transfer Selection

Specify input signals:

rct_linact/Speed Demand (rpm)/1

+ Add signal to list

Specify output signals:

rct_linact/Current Controller/Current PID/1

+ Add signal to list

Compute input/output gain with the following loops open:

+ Add loop opening location to list

Options

Limit gain to	0.001
Stabilize I/O transfer	Yes
Enforce goal in frequency range	[0 Inf]
Adjust for signal amplitude	No

Figure 24: Gain Goal Dialog from Speed Demand to Control Signal of Current PID.

New gain goals appear in Tuning Goals section of Control System Tuner App.
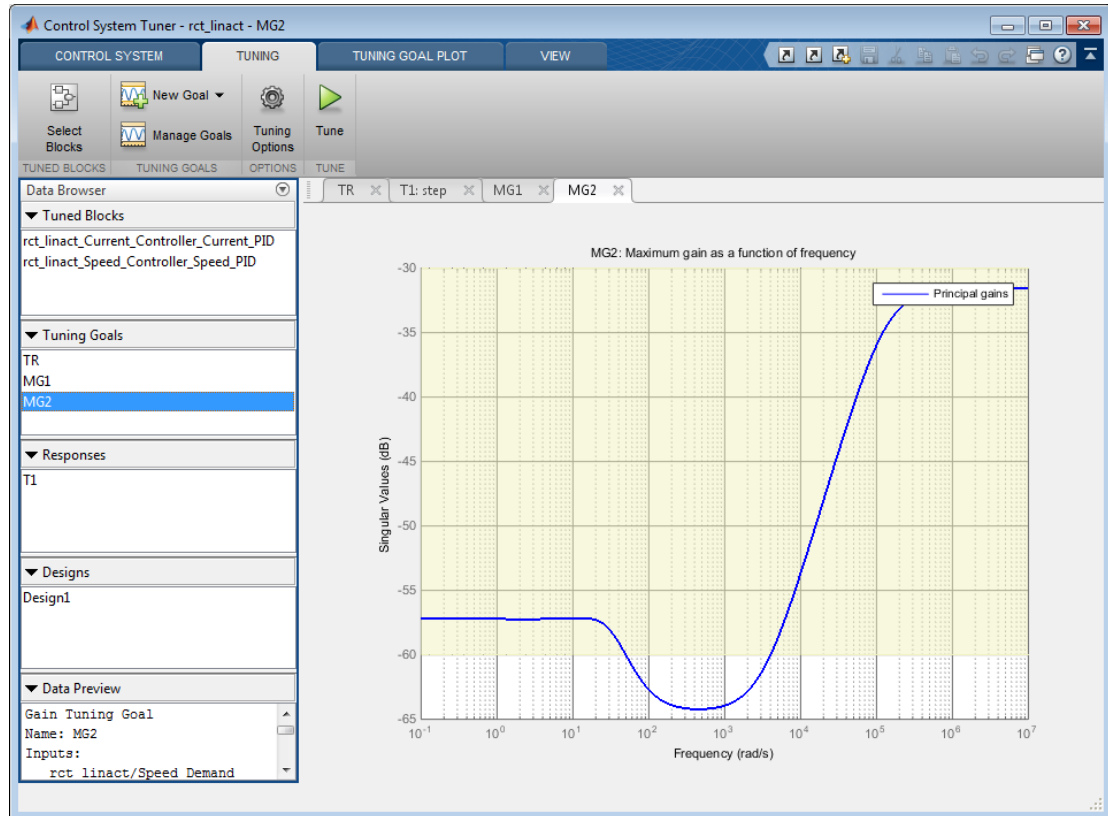
Figure 25: Two Gain Goals Added to Control System Tuner App.

Retune with these additional requirements. Tuning Report shows the worst gain 1.39 indicating that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1)
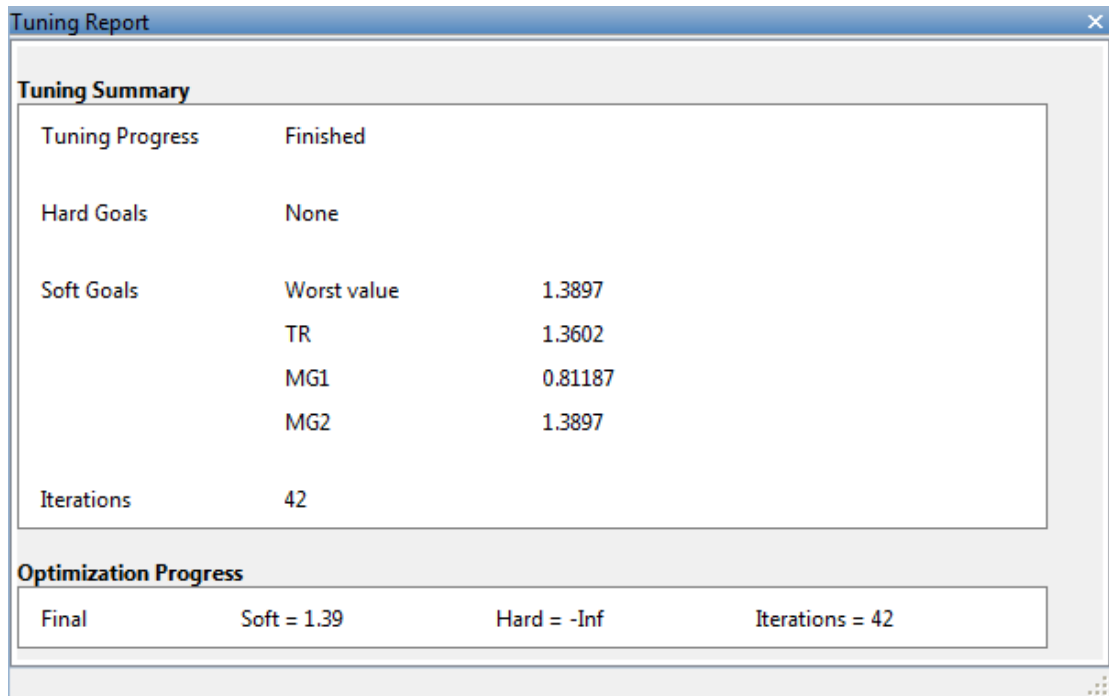
Figure 26: Tuning Report After Retuning.

Next compare the two designs in the linear domain by clicking `Compare` in `Control System` tab.
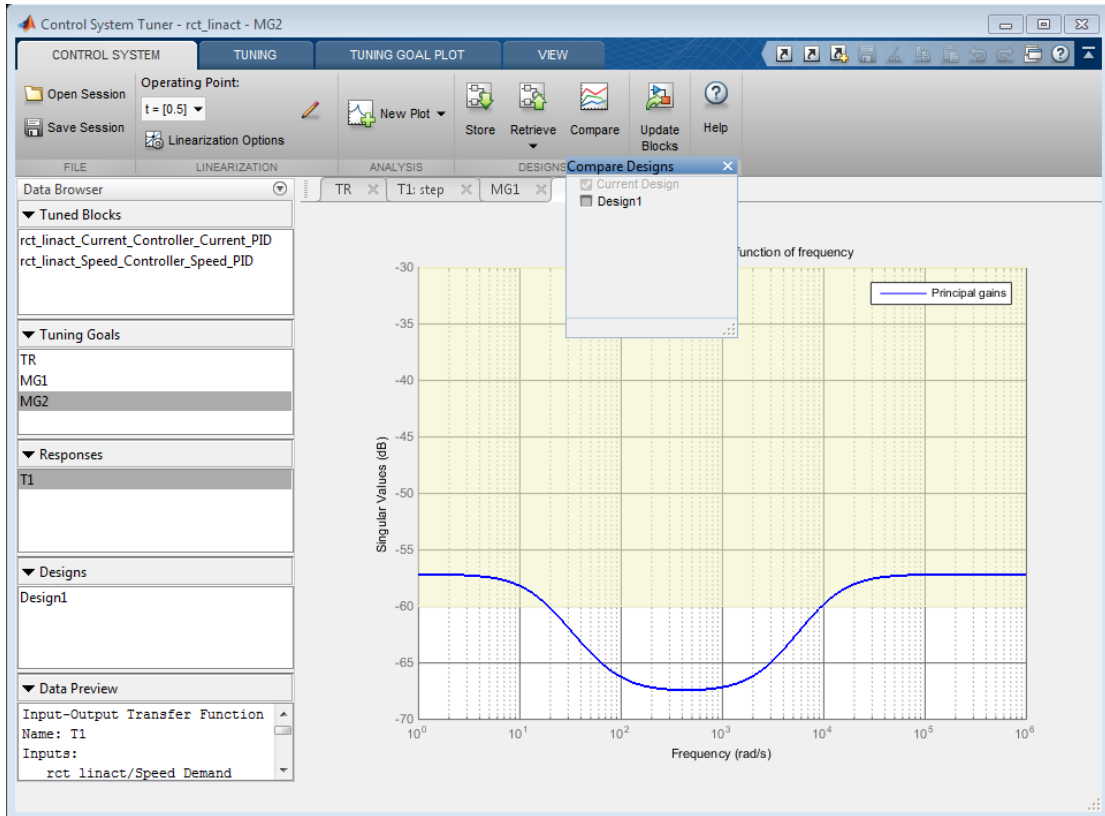
Figure 27: Comparing Two Designs.

The second design is less aggressive but still meets the response time requirement.
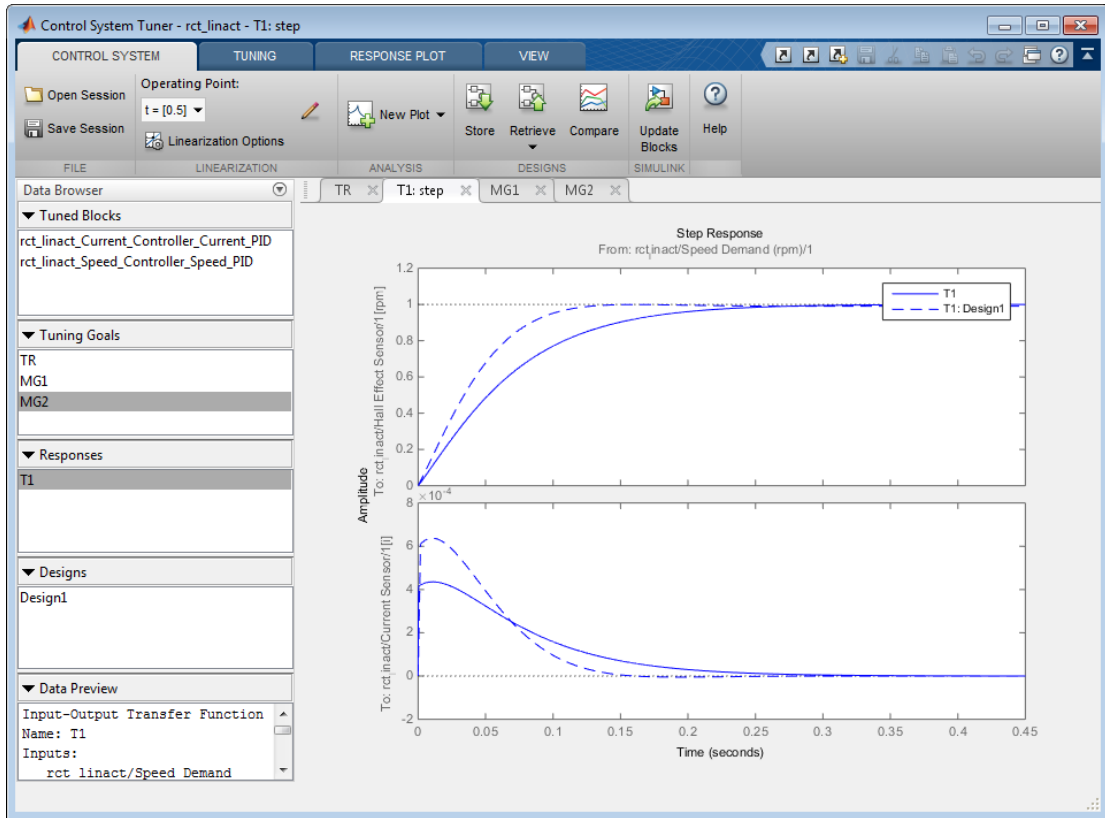
Figure 28: Step Responses of Two Designs.

Finally, push the new tuned gain values to the Simulink model by `Update Blocks` 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.
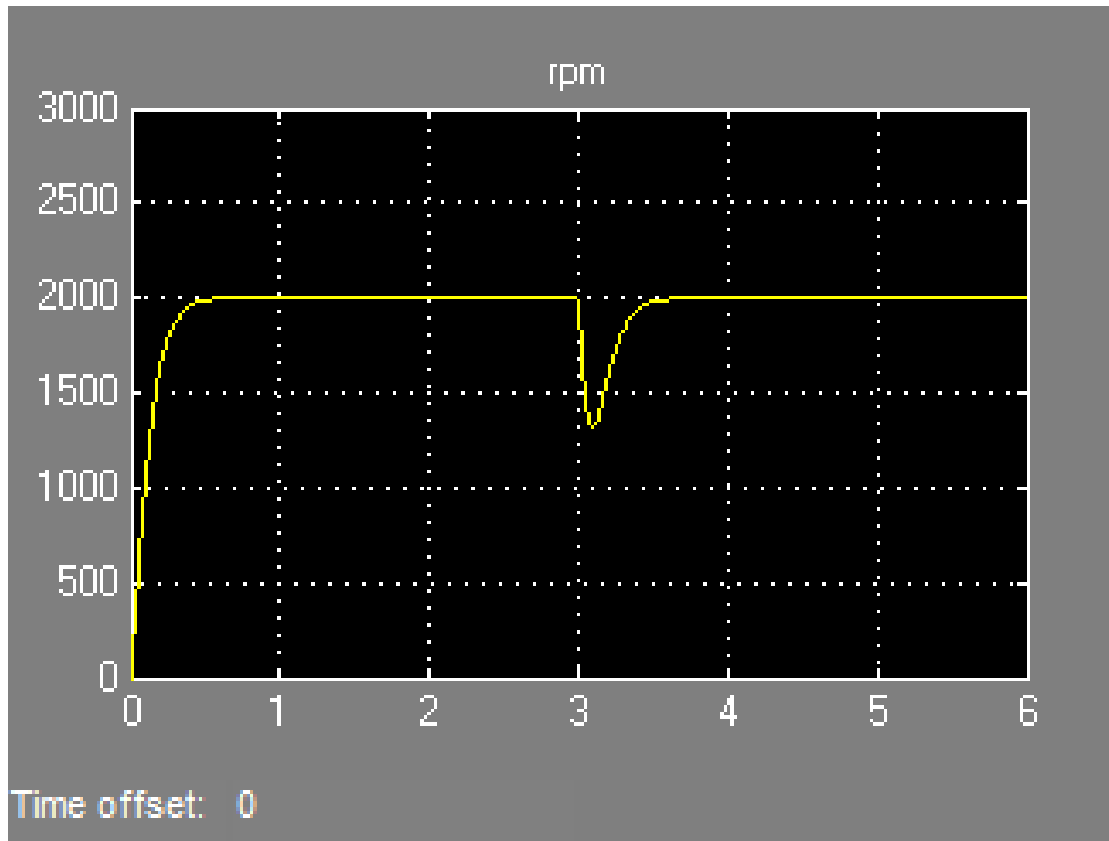
Figure 29: Nonlinear Response of Tuning with Gain Constraints.
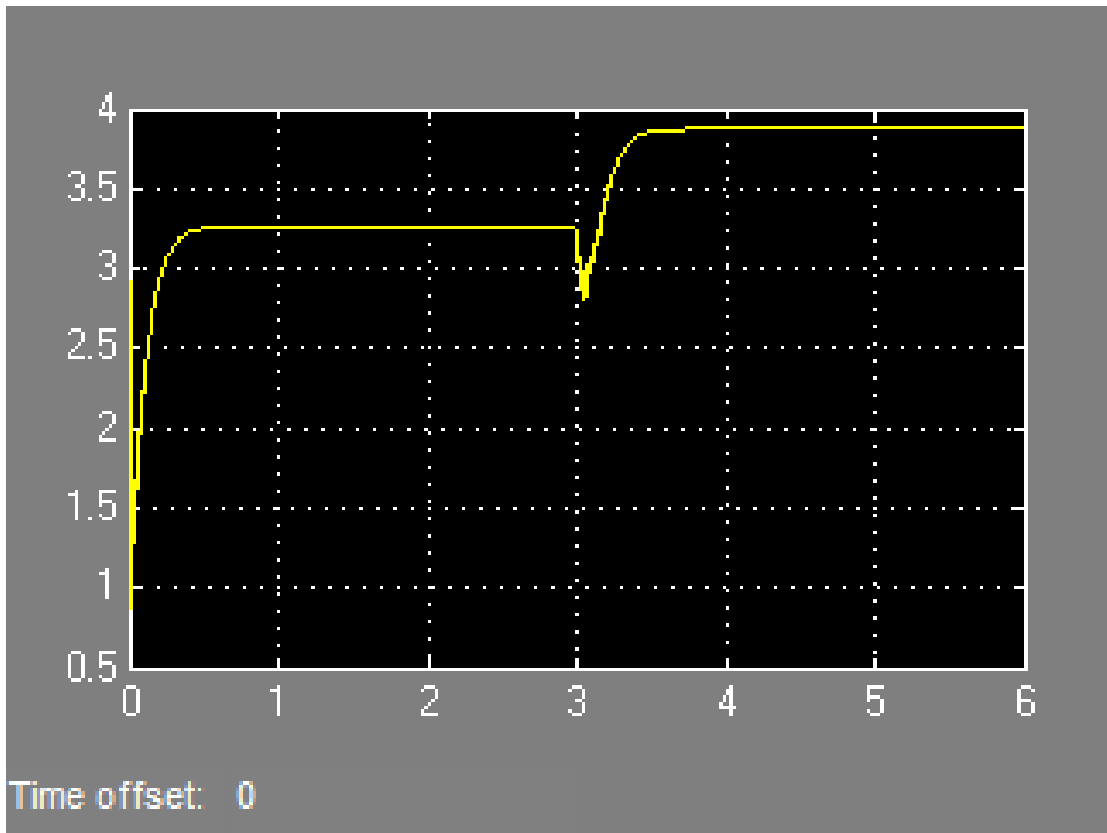
Figure 30: Current Controller Output.

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.

7

Tuning Fixed Control Architectures

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 `systune`, `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).

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

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

Traditional H_∞ synthesis (performed using the `hinfsyn` or `loopsyn` commands) designs a full-order, centralized controller. Traditional H_∞ 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_∞ synthesis requires you to express all design requirements in terms of a single weighted MIMO transfer function.

In contrast, structured H_∞ 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.
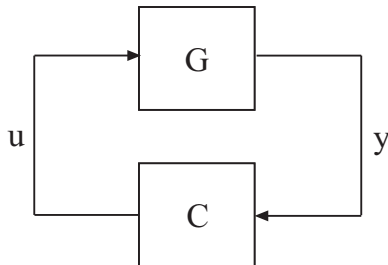
C represents the controller and G represents the plant. The sensor outputs y (*measurement signals*) and actuator outputs u (*control signals*) define the boundary between plant and controller. The controller is the portion of your control system whose inputs are measurements, and whose outputs are control 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 qL and qV as outputs.
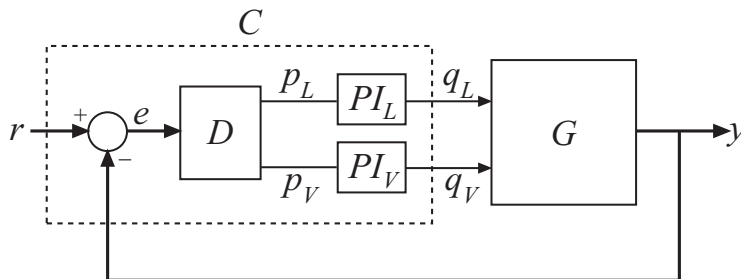
The controller C has a fixed internal structure. C includes a gain matrix D , the PI controllers PI_L and PI_V, and a summing junction. The looptune command tunes free parameters of C such as the gains in D and the proportional and integral gains of PI_L and PI_V. You can also use looptune to co-tune 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 looptunein MATLAB" on page 7-6
"Set Up Your Control System for looptune in Simulink" on page 7-6

Set Up Your Control System for looptunein 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 `systune` (requires Simulink Control Design software):

1   Use `slTuner` 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 `addPoint` to specify the control and measurement signals that define the boundaries between plant and controller. Use `addOpening` to mark optional loop-opening or signal injection sites for specifying and assessing open-loop requirements.

The `slTuner` interface automatically linearizes your Simulink model. The `slTuner` interface also automatically parametrizes the blocks that you specify as tunable blocks. For more information about this linearization, see the `slTuner` reference page and "How Tuned Simulink Blocks Are Parameterized" on page 6-42.

Related Examples

-   "Tune MIMO Control System for Specified Bandwidth" on page 7-8

•   "Tuning Feedback Loops with LOOPTUNE" on page 7-30

More About

•   "Structure of Control System for Tuning With looptune" on page 7-5

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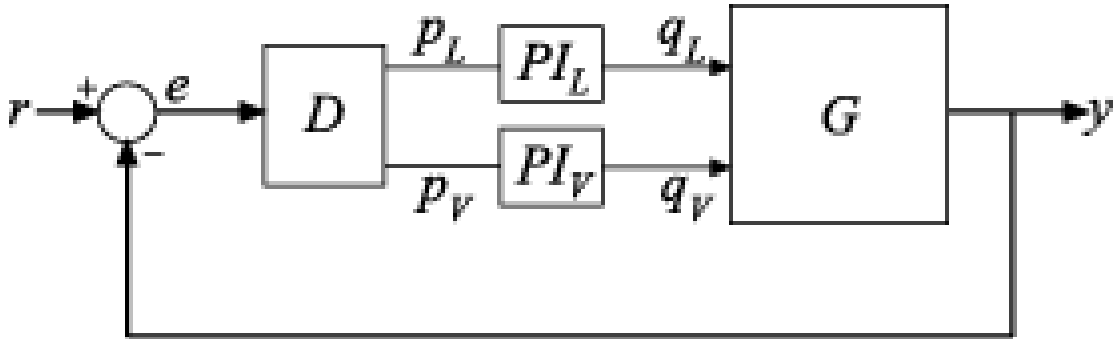
The plant, `G`, is a two-input, two-output model (*y* is a two-element vector signal). For this example, the transfer function of `G` is given by:

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

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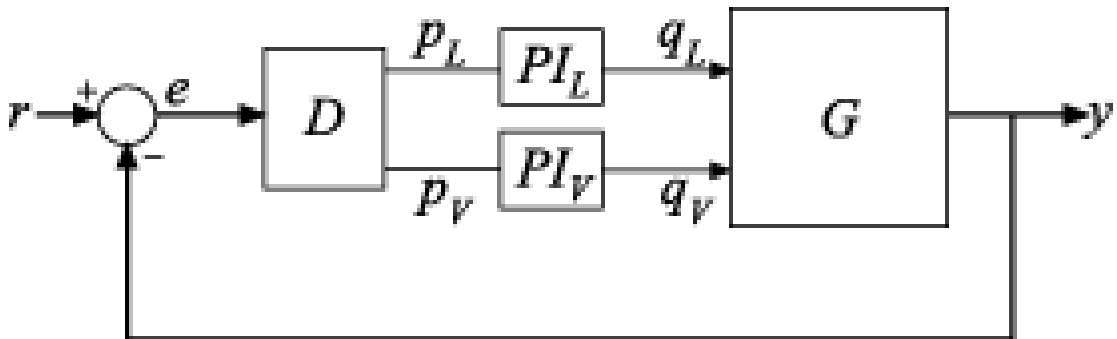
When you tune the control system, `looptune` uses these channel names to interconnect `C` and `G`. The controller `C` also includes the summing junction `sum1`. This a two-channel summing junction, because `r` and `y` are vector-valued signals of dimension 2.

Connect the controller components.

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

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

Tune the control system.

The inputs to `looptune` are `G` and `CO`, 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,CO,wc);

Final: Peak gain = 0.956, Iterations = 19
Achieved target gain value TargetGain=1.
```

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 `CO`. 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 =
           u1       u2
   y1    1.561   -0.9838
   y2   -1.128    0.9175
```

```
Name: Decoupler
Static gain.
----------------------------------
PI_L =

             1
  Kp + Ki * ---
             s

  with Kp = 2.17, Ki = 0.00986

Name: PI_L
Continuous-time PI controller in parallel form.
----------------------------------
PI_V =

             1
  Kp + Ki * ---
             s

  with Kp = -2.68, Ki = -0.0376

Name: PI_V
Continuous-time PI controller in parallel form.
```
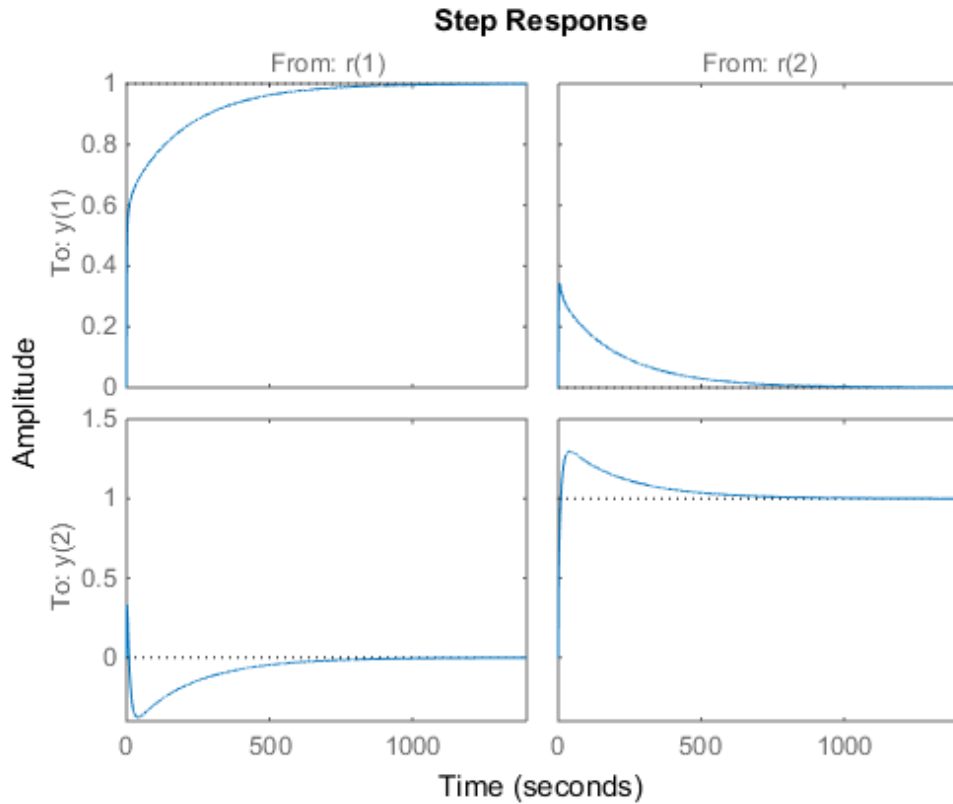
Check the time-domain response for the control system with the tuned coefficients. To produce a plot, construct a closed-loop model of the tuned control system. Plot the step response from reference to output.

```
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 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)
```
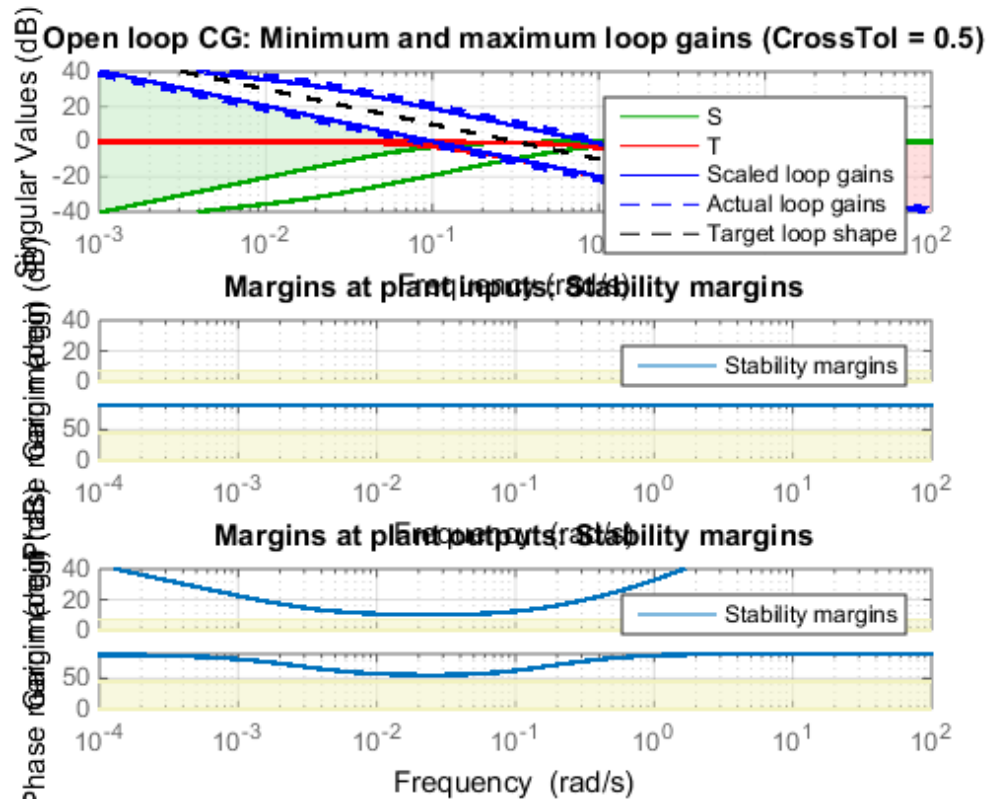
The first plot shows that the open-loop gain crossovers fall within the specified interval [0.1,1]. This plot also includes the maximum and tuned values of the sensitivity function $S = (I - GC)^{-1}$ and complementary sensitivity $T = I - S$. The second and third plots show that the MIMO stability margins of the tuned system (blue curve) do not exceed the upper limit (yellow curve).

Related Examples

- "Decoupling Controller for a Distillation Column" on page 7-61

More About

- "Structure of Control System for Tuning With looptune" on page 7-5

What Is hinfstruct?

`hinfstruct` lets you use the frequency-domain methods of H_∞ 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_∞ 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_∞ 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_∞ constraint of the form:

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

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_∞ synthesis using loop shaping, see "Fixed-Structure H-infinity Synthesis with HINFSTRUCT" on page 7-179.

For more information about constructing weighting functions from design requirements, see "H-Infinity Performance".

Structured H-Infinity Synthesis Workflow

Performing structured H_∞ synthesis requires the following steps:

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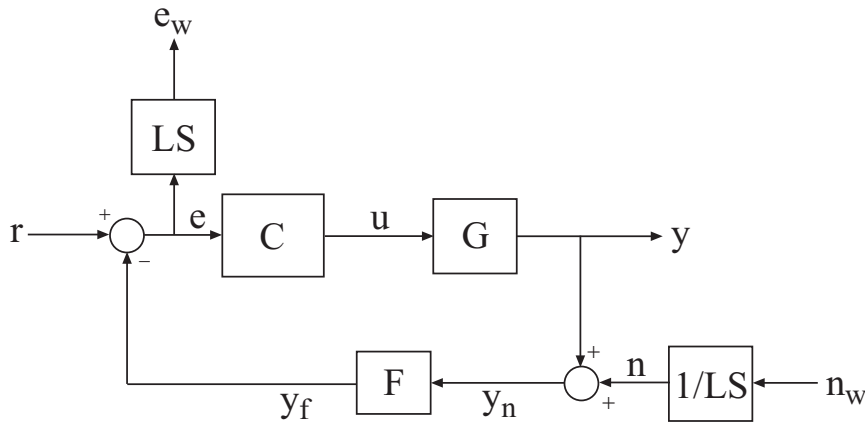
This block diagram represents a head-disk assembly (HDA) in a hard disk drive. The architecture includes the plant `G` in a feedback loop with a PI controller `C` and a low-pass filter, `F = a/(s+a)`. The tunable parameters are the PI gains of `C` and the filter parameter `a`.

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_∞ constraint:

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

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

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

This value of *LS* corresponds to the following open-loop response shape.
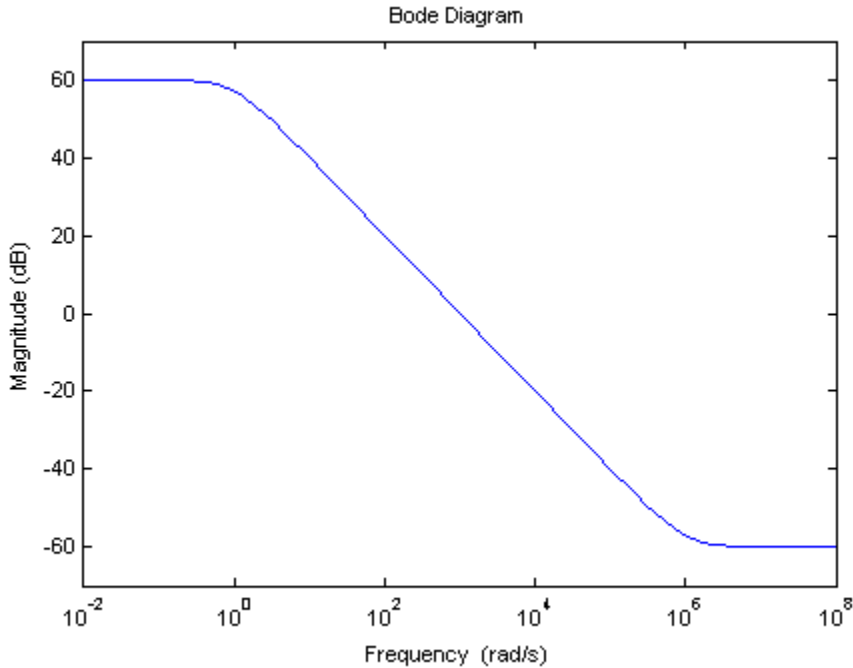
To tune the HDA control system with `hinfstruct`, construct a tunable model of the closed-loop system *T*(*s*), including the weighting functions, as follows.

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*).

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

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

```
TO.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 7-24.

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 `slTuner` 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 `slTuner` 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');
```
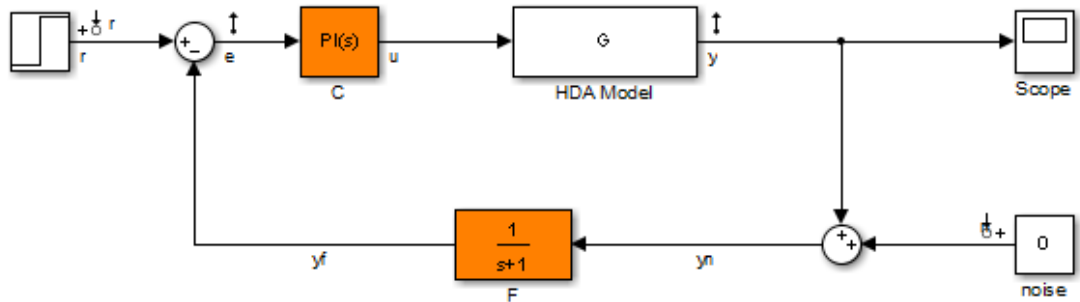
See hinfstruct_demo to see how you can tune the PI gains and the filter coefficient
with the HINFSTRUCT command.

Copyright 2004-2010 The MathWorks, Inc.

2 Create an `slTuner` 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 `slTuner` reference page.)

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

This command specifies that `C` and `F` are the tunable blocks in the model. The `slTuner` 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 7-17. *T*(*s*) is a weighted closed-loop model of the control system of `rct_diskdrive`. Tuning `T0` to enforce the H_∞ constraint

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

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 7-24.

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 7-17 using the following command:

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

Final: Peak gain = 3.88, Iterations = 108
```

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_∞ norm of this transfer function.

Interpreting gamma

gamma is the smallest H_∞ 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_∞ 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 7-25. Use these tuned models to examine the performance of the tuned system.
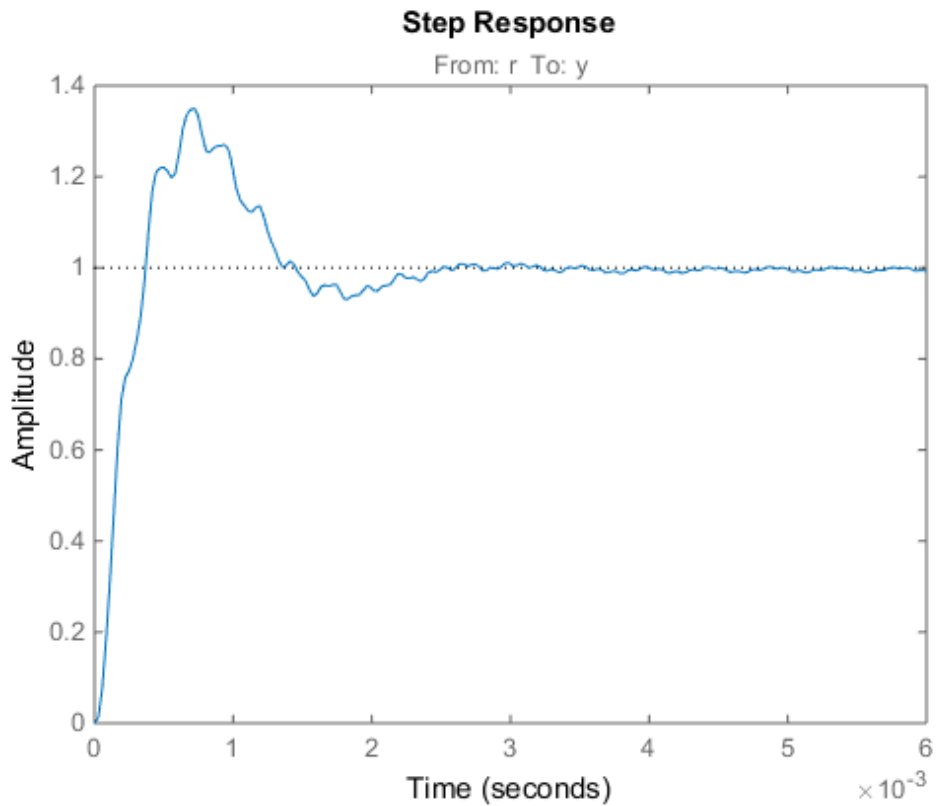
Validating the Design in MATLAB

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 7-24. To do so, use `getIOTransfer` to extract from the tuned control system the transfer function between the step input and the measured output.

```
Try = getIOTransfer(T,'r','y');
step(Try)
```

Step Response

From: r To: y

Validating the Design in Simulink

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

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

In this example, write tuned parameters to the `rct_diskdrive` system tuned in "Tune the Controller Parameters" on page 7-24.

Make a copy of the `slTuner` 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');
step(Try)
```

Step Response

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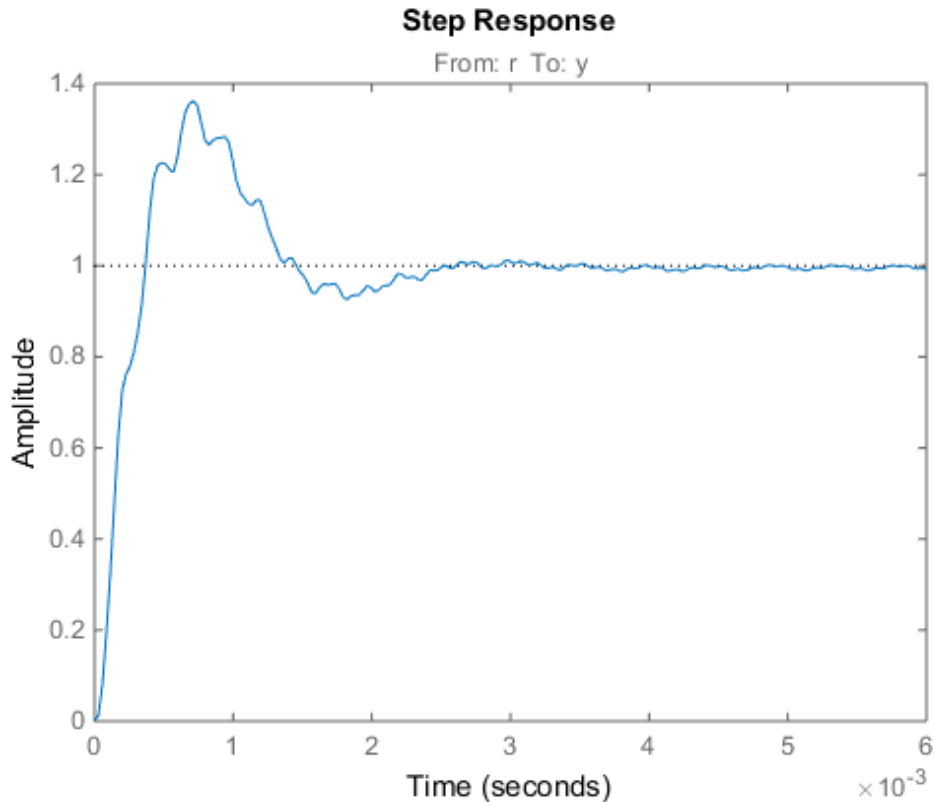
Since `hinfstruct` tunes a linearized version of your system, you should also validate the tuned controller in the full nonlinear Simulink model. To do so, write the parameter values from the `slTuner` interface to the Simulink model.

```
writeBlockValue(ST)
```

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

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.
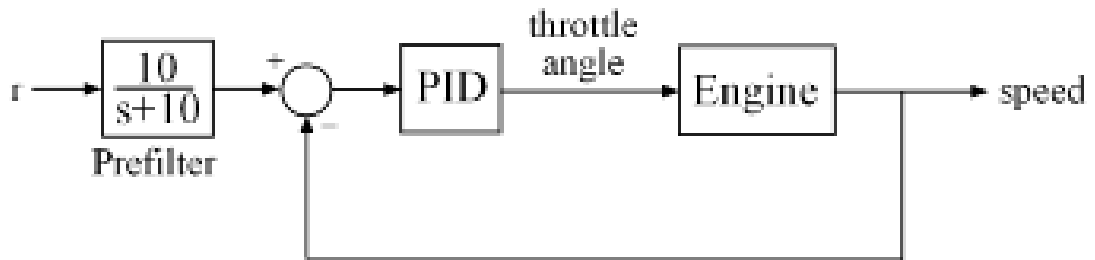
Figure 1: Engine Speed Control Loop

We use the following fourth-order model of the engine dynamics.

```
load rctExamples Engine
bode(Engine), grid
```
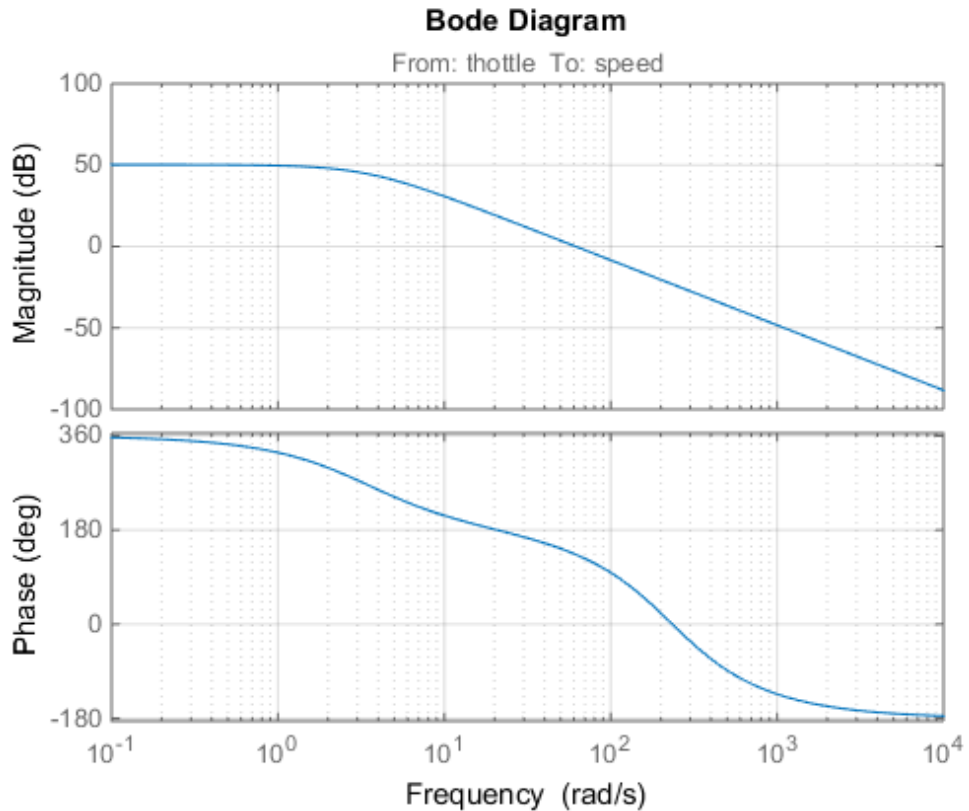
Specifying the Tunable Elements

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:

            1              s
  Kp + Ki * --- + Kd * --------
```

```
             s           Tf*s+1

  and tunable parameters Kp, Ki, Kd, Tf.

Type "pid(PIDO)" to see the current value and "get(PIDO)" 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.
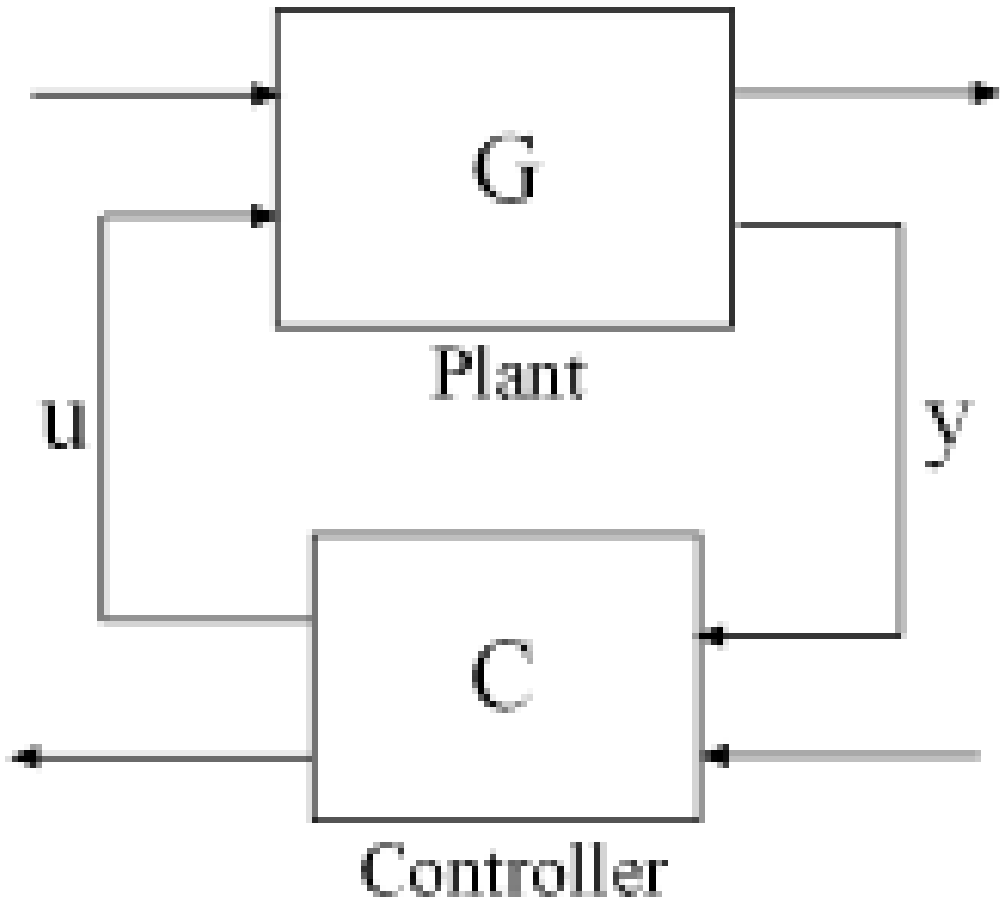
Figure 2: Generic Feedback Loop

For the speed control loop, the plant G is the engine model and the controller C consists of the PID and the prefilter F.
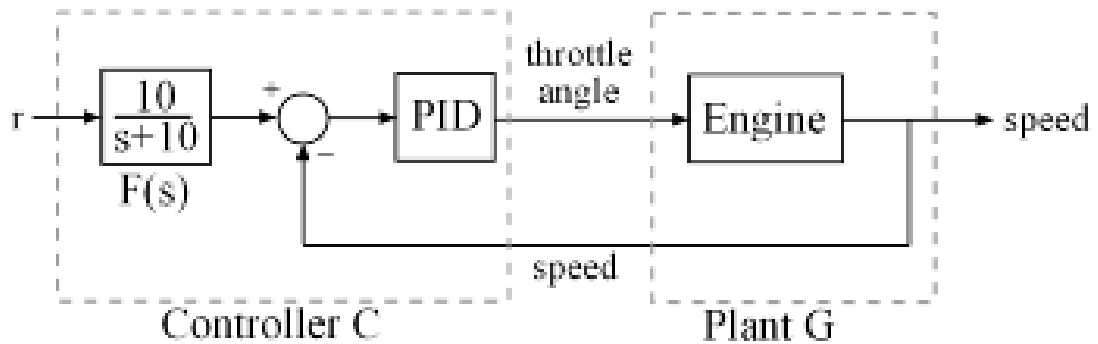
Figure 3: Feedback Loop for Engine Speed Control

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,CO,wc);

Final: Peak gain = 0.92, Iterations = 3
Achieved target gain value TargetGain=1.
```

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 =

              1              s
  Kp + Ki * --- + Kd * --------
              s            Tf*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)
```

7-35
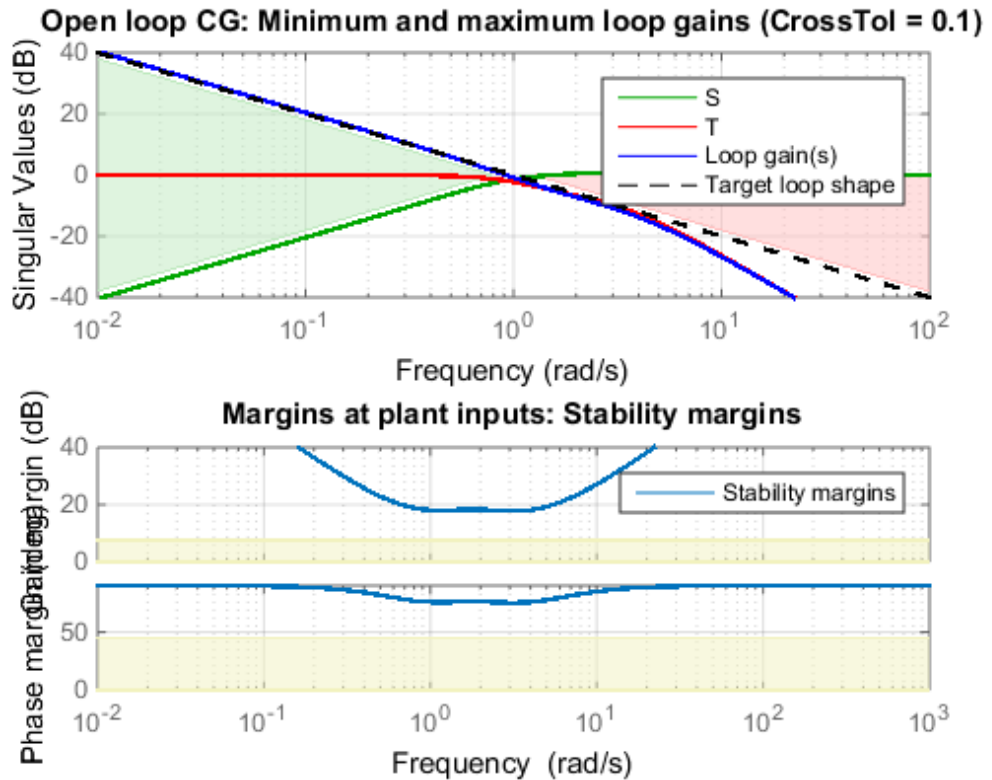
Next plot the closed-loop response to a step command in engine speed. The tuned response satisfies our requirements.

```
T = connect(G,C,'r','speed');  % closed-loop transfer from r to speed
clf, step(T)
```
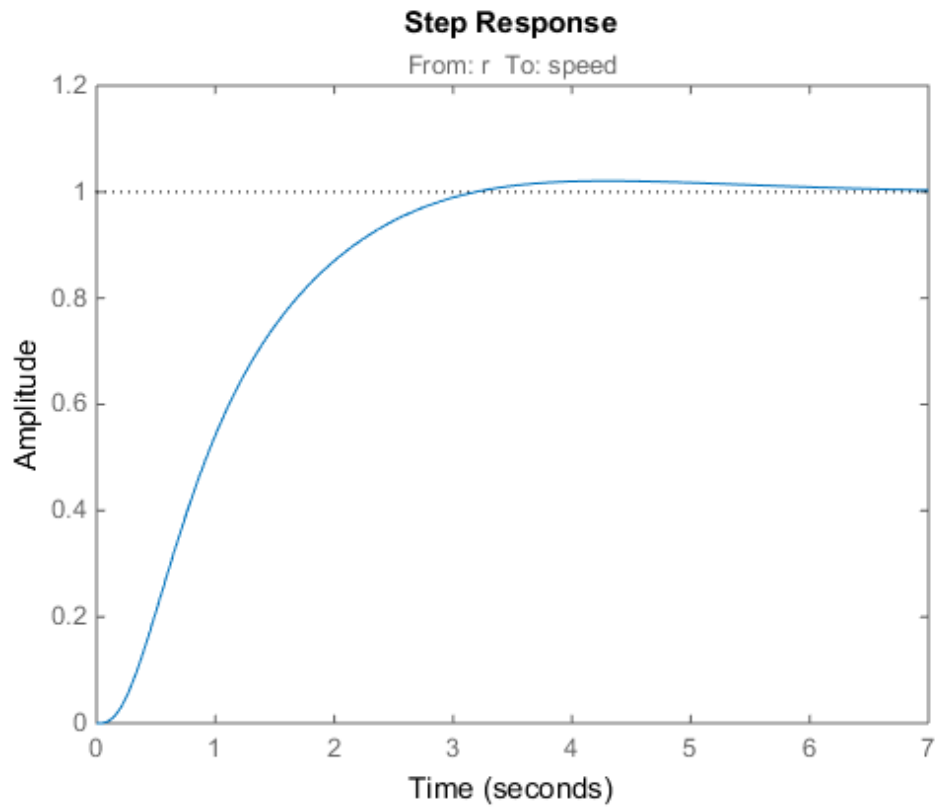
Tuning Multi-Loop Control Systems

This example shows how to jointly tune the inner and outer loops of a cascade architecture with the `systune` 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')
```
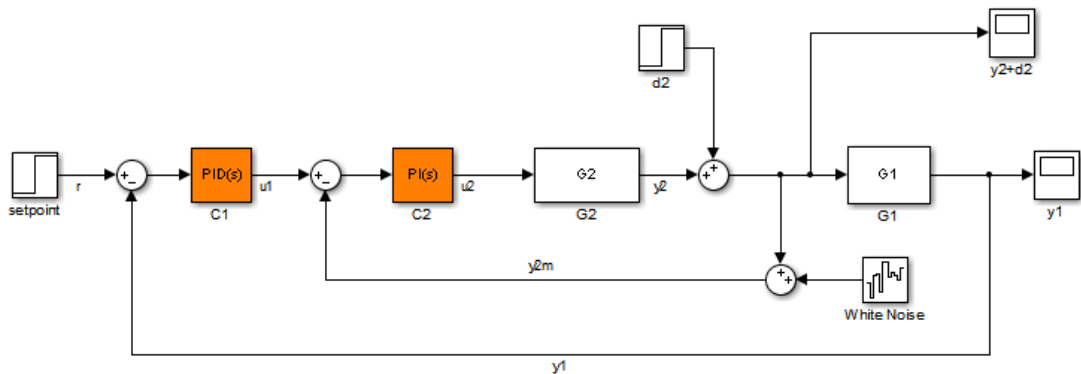

Plant Models and Bandwidth Requirements

In this example, the inner loop plant G2 is

$$G_2(s) = \frac{3}{s+2}$$

and the outer loop plant G1 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 `slTuner` interface in Simulink Control Design™ to set up the tuning task. List the tunable blocks, mark the signals `r` and `d2` as inputs of interest, and mark the signals `y1` and `y2` as locations where to measure open-loop transfers and specify loop shapes.

```
ST0 = slTuner('rct_cascade',{'C1','C2'});
addPoint(ST0,{'r','d2','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.

```
showTunable(ST0)


Block 1: rct_cascade/C1 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.1, Ki = 0.1

Name: C1
Continuous-time PI controller in parallel form.

----------------------------------

Block 2: rct_cascade/C2 =

             1
  Kp + Ki * ---
             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 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] = systune(STO,[Req1,Req2]);
```

```
Final: Soft = 0.859, Hard = -Inf, Iterations = 99
```

Use showTunable to see the tuned PID gains.

```
showTunable(ST)
```

```
Block 1: rct_cascade/C1 =

              1              s
  Kp + Ki * --- + Kd * --------
              s           Tf*s+1

  with Kp = 0.0499, Ki = 0.0186, Kd = 0.0426, Tf = 0.0133

Name: C1
```

```
Continuous-time PIDF controller in parallel form.

---------------------------------

Block 2: rct_cascade/C2 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.719, Ki = 1.58

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)
```

Outer Loop: Minimum and maximum loop gains (CrossTol = 0.1)

Inner Loop: Minimum and maximum loop gains (CrossTol = 0.1)
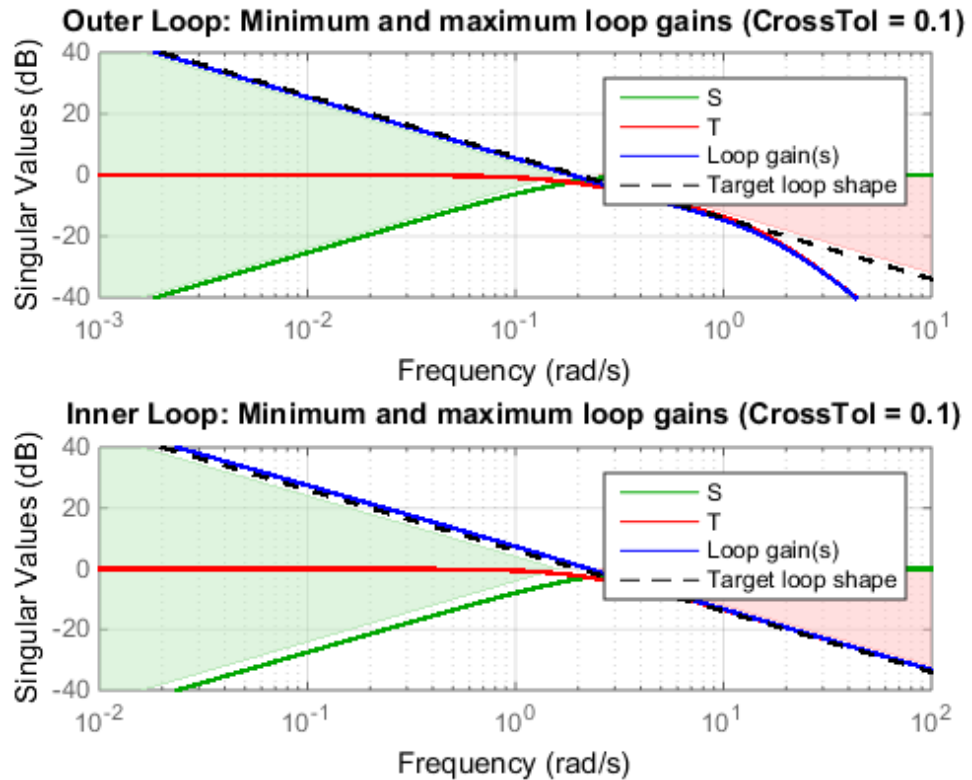
Note that the inner and outer loops have the desired gain crossover frequencies. To further validate the design, plot the tuned responses to a step command r and step disturbance d2:

```
% Response to a step command
H = getIOTransfer(ST,'r','y1');
clf, step(H,30), title('Step command')
```
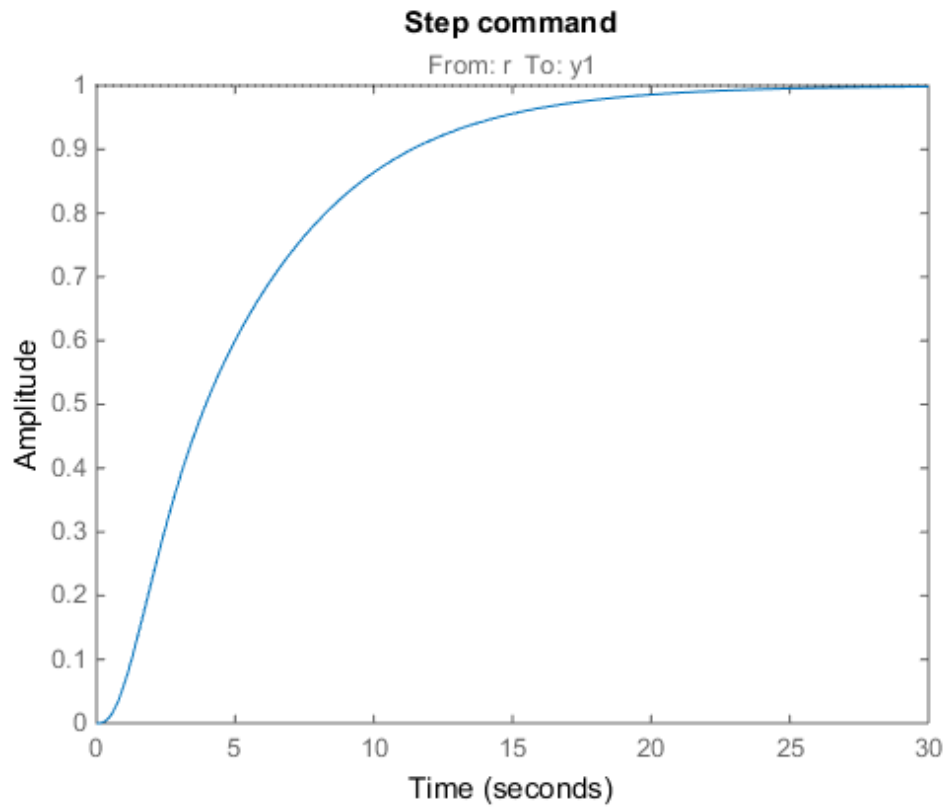
```
% Response to a step disturbance
H = getIOTransfer(ST,'d2','y1');
step(H,30), title('Step disturbance')
```
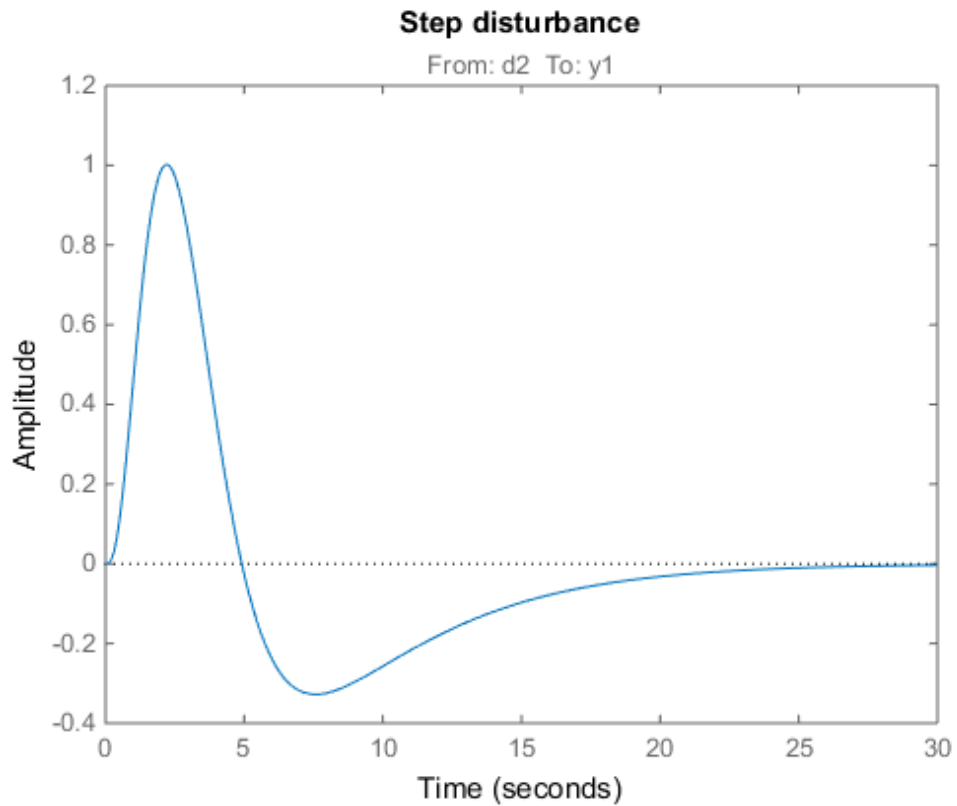
Once you are satisfied with the linear analysis results, use `writeBlockValue` to write the tuned PID gains back to the Simulink blocks. You can then conduct a more thorough validation in Simulink.

```
writeBlockValue(ST)
```

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.
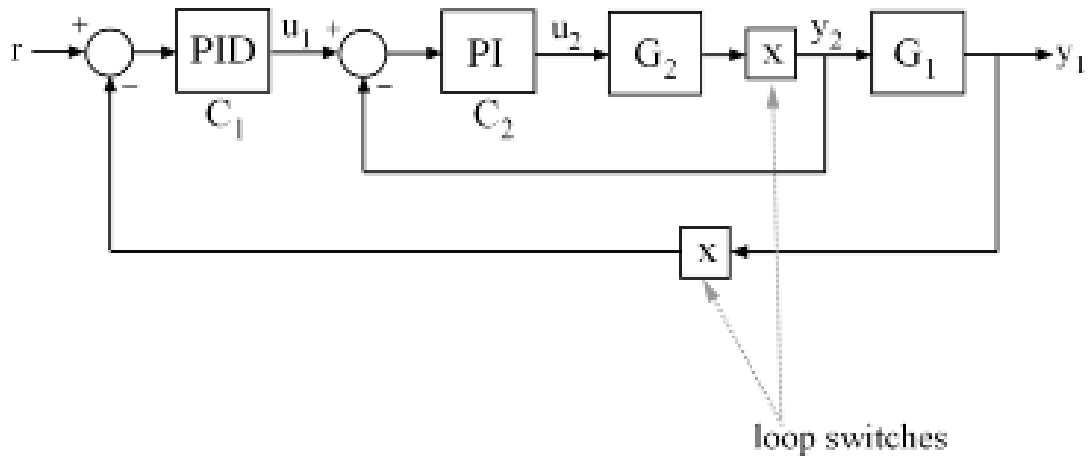
Figure 1: Cascade Architecture

First create parametric models of the tunable PI and PID controllers.

```
C1 = ltiblock.pid('C1','pid');
C2 = ltiblock.pid('C2','pi');
```

Then use "analysis point" blocks to mark the loop opening locations y1 and y2.

```
LS1 = AnalysisPoint('y1');
LS2 = AnalysisPoint('y2');
```

Finally, create a closed-loop model T0 of the overall control system by closing each feedback loop. The result is a generalized state-space model depending on the tunable elements C1 and C2.

```
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 systune.

```
[T,fSoft,~,Info] = systune(T0,[Req1,Req2]);

Final: Soft = 0.858, Hard = -Inf, Iterations = 99
```

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')
```
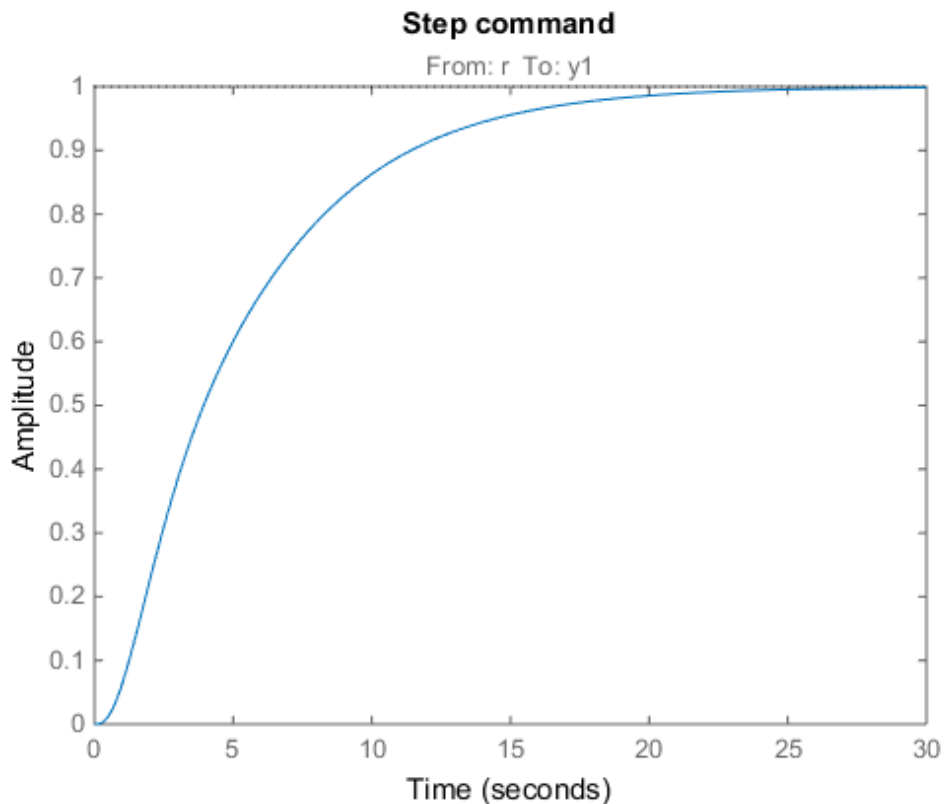

% Response to a step disturbance
```
H = getIOTransfer(T,'y2','y1');
step(H,30), title('Step disturbance')
```
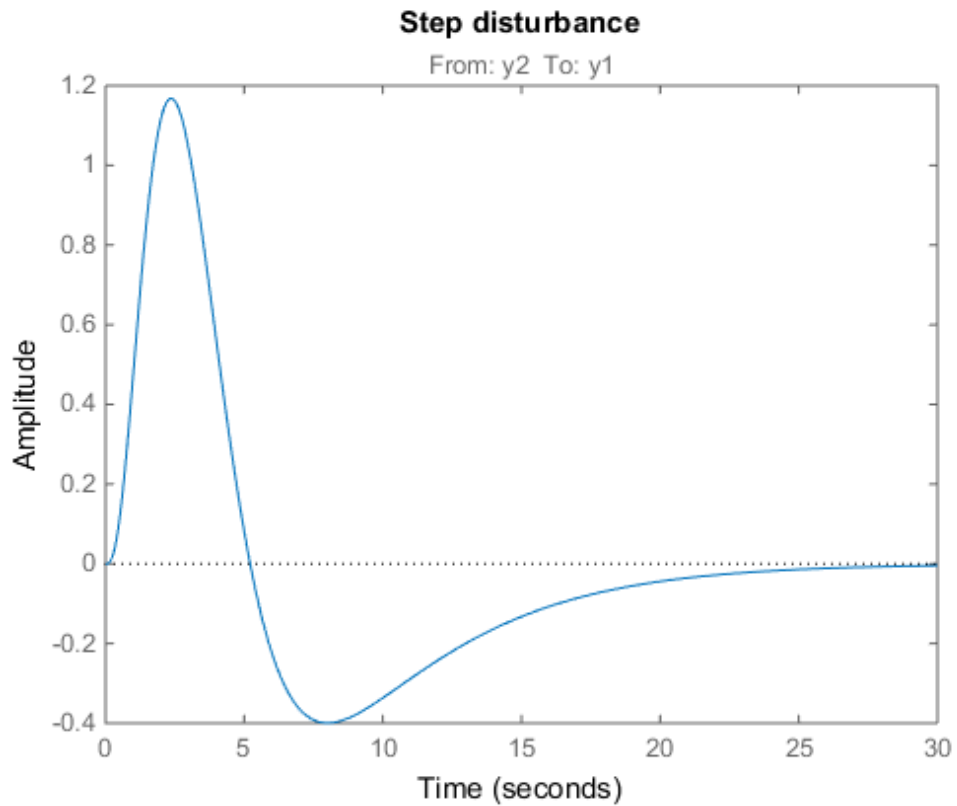
You can also plot the open-loop gains for the inner and outer loops to validate the bandwitdth requirements. Note the -1 sign to compute the negative-feedback open-loop transfer:

```
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')
```
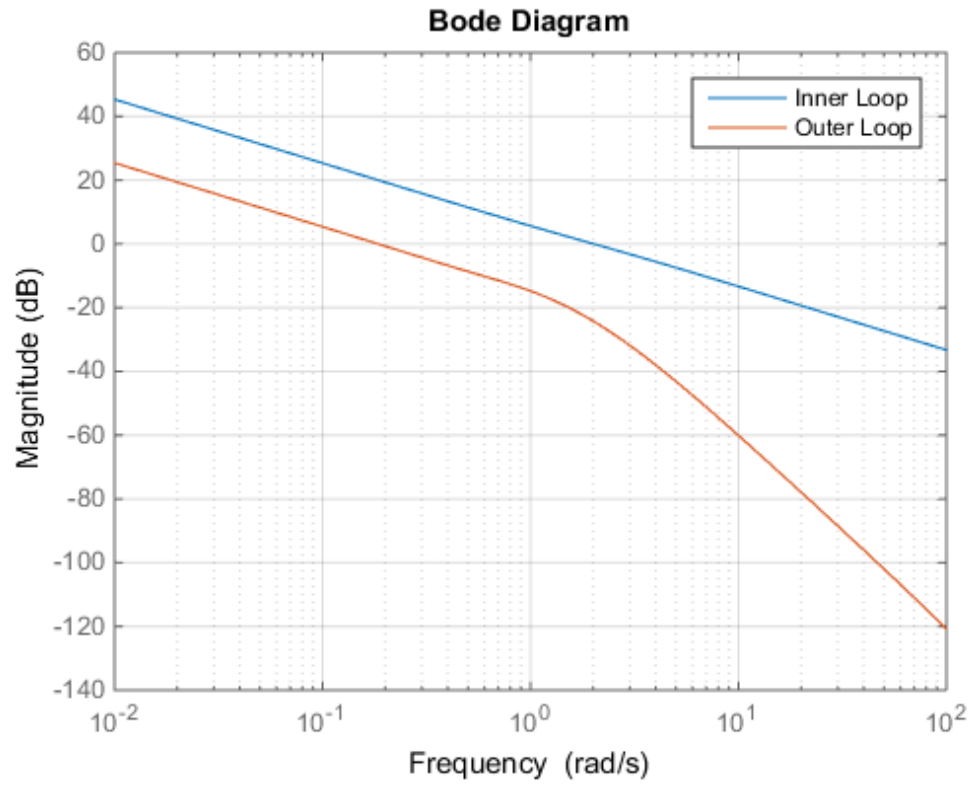
PID Tuning for Setpoint Tracking vs. Disturbance Rejection

This example uses `systune` 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.
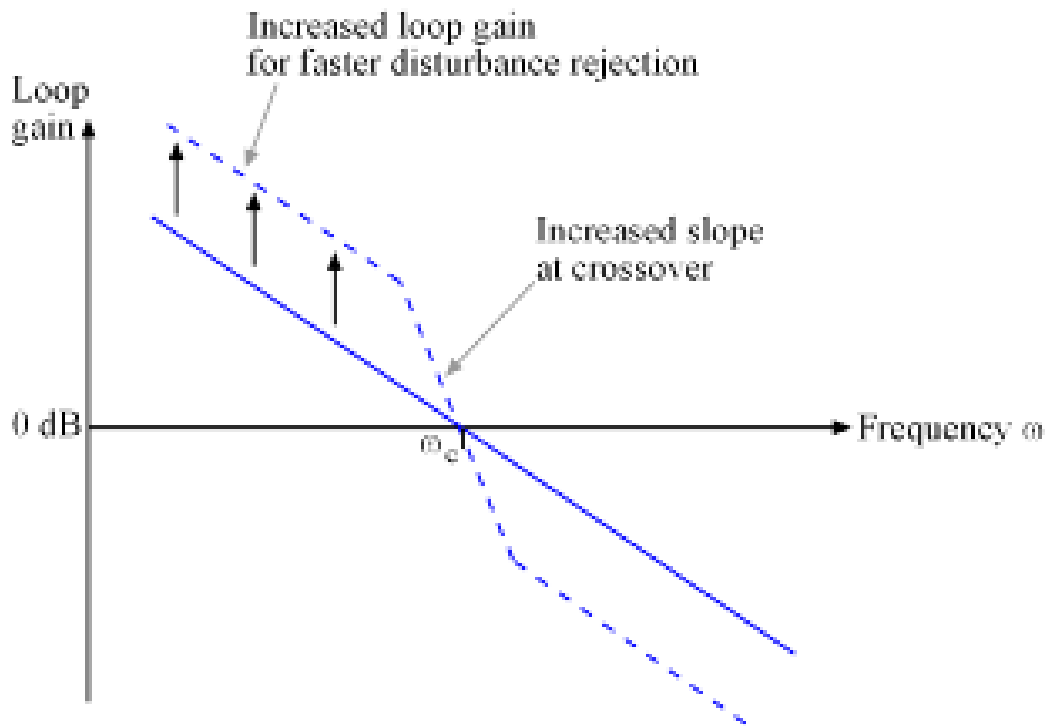
Figure 1: Trade-off in 1-DOF PID Tuning.

This example uses `systune` 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.
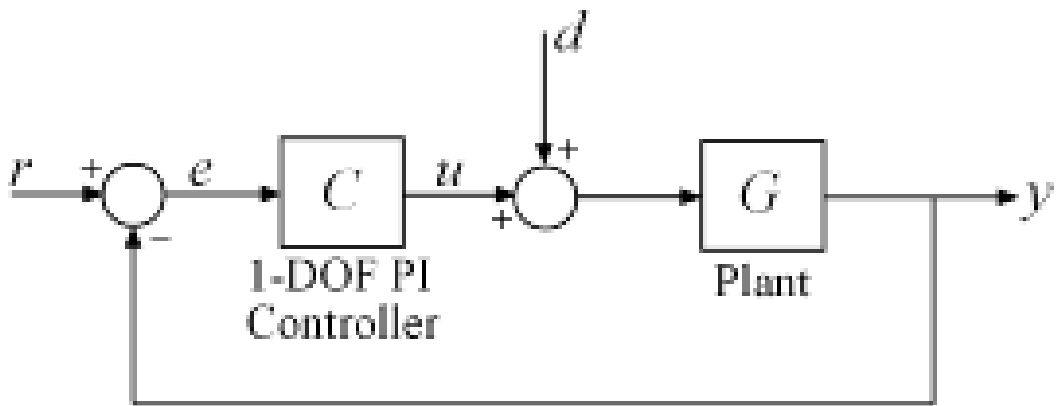
Figure 2: PI Control Loop.

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 an "analysis point" block to mark the location `u` where the disturbance enters.

```
G = zpk(-5,[-1 -2 -10],10);
C = ltiblock.pid('C','pi');
LS = AnalysisPoint('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 `systune`, 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.
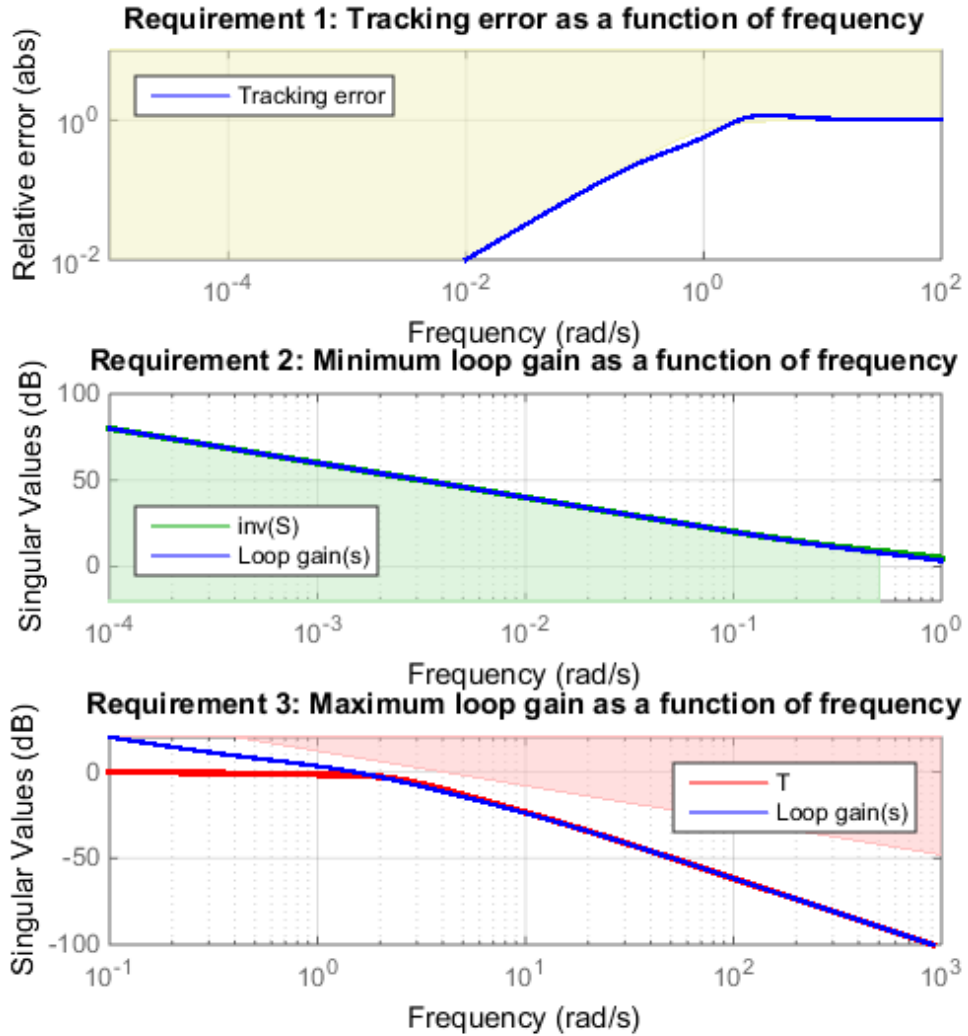
```
T1 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.24, Hard = 0.99993, Iterations = 139
```

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.

```
figure('Position',[100,100,560,580])
```

```
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 α inside the 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 = 0.99969, Iterations = 120

% Increase loop gain by factor 5
alpha = 5;
R2.MinGain = alpha*wc/s;
T3 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.52, Hard = 0.99993, Iterations = 147
```
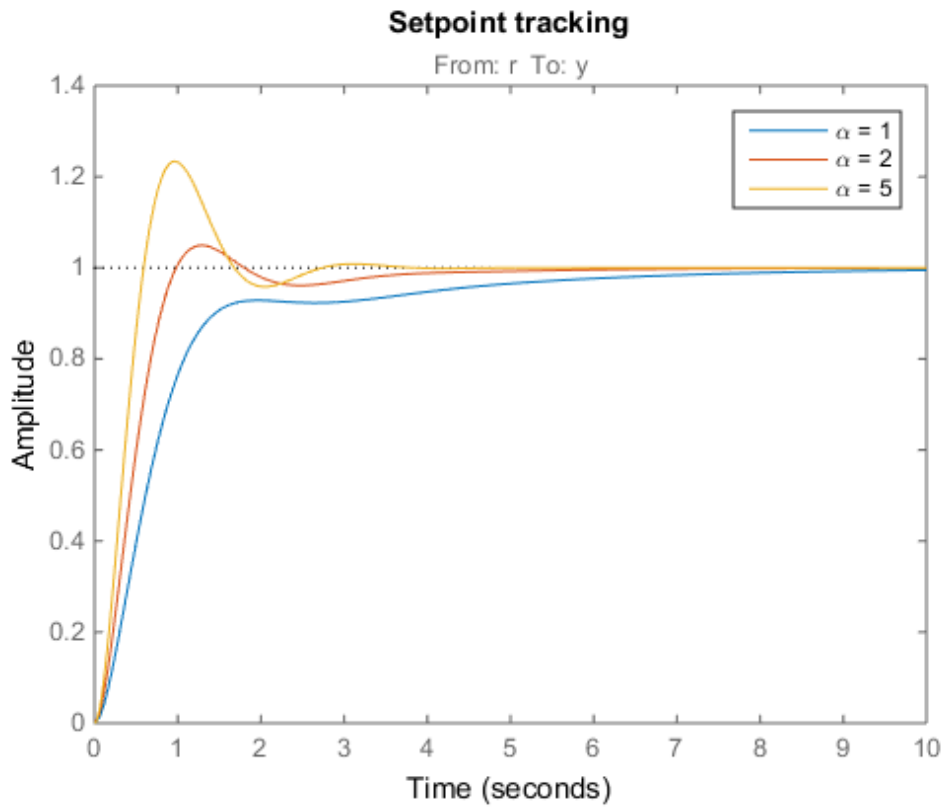
Now compare the responses to a step command r and to a step disturbance d entering at the plant input u.

```
figure, step(T1,T2,T3,10)
title('Setpoint tracking')
legend('\alpha = 1','\alpha = 2','\alpha = 5')
```

```matlab
% 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')
```
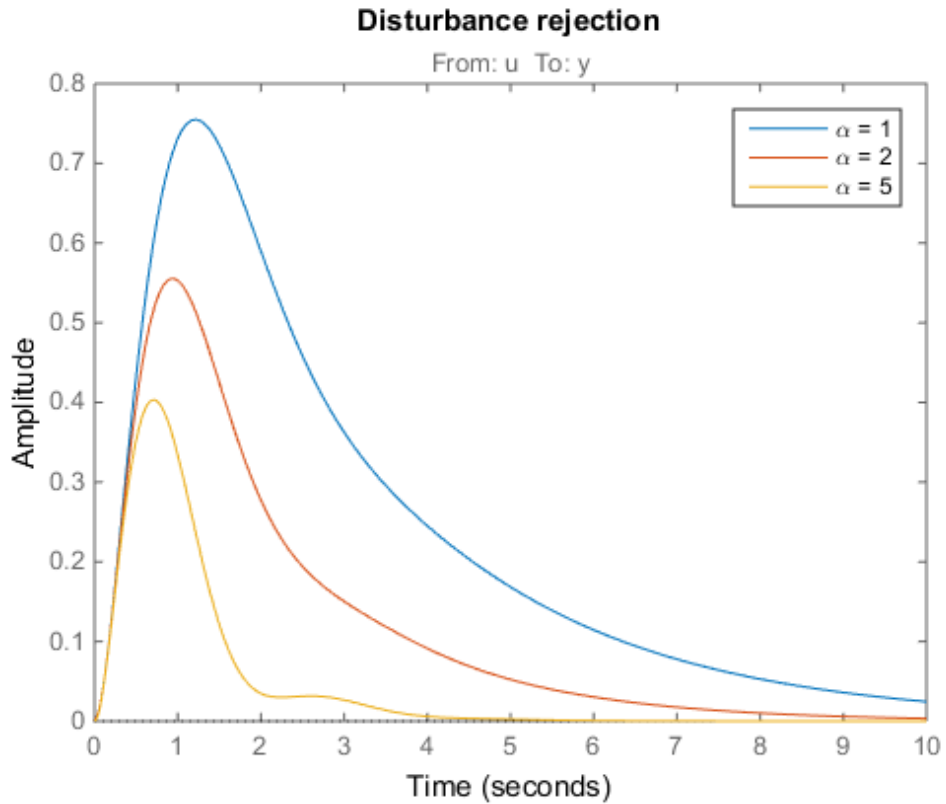
Note how disturbance rejection improves as `alpha` increases, but only at the expense of increased overshoot and oscillations in setpoint tracking. Plot the open-loop responses for the three designs, and note how the slope at crossover (0dB) increases with `alpha`.

```
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')
```
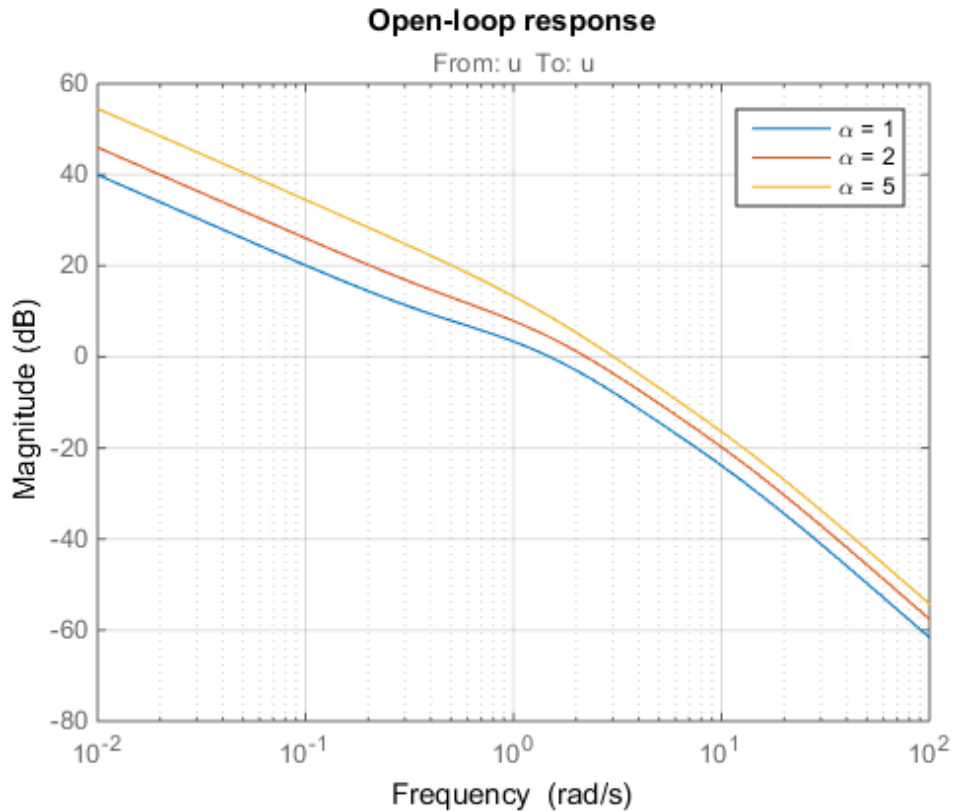
Which design is most suitable depends on the primary purpose of the feedback loop you are tuning.

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.
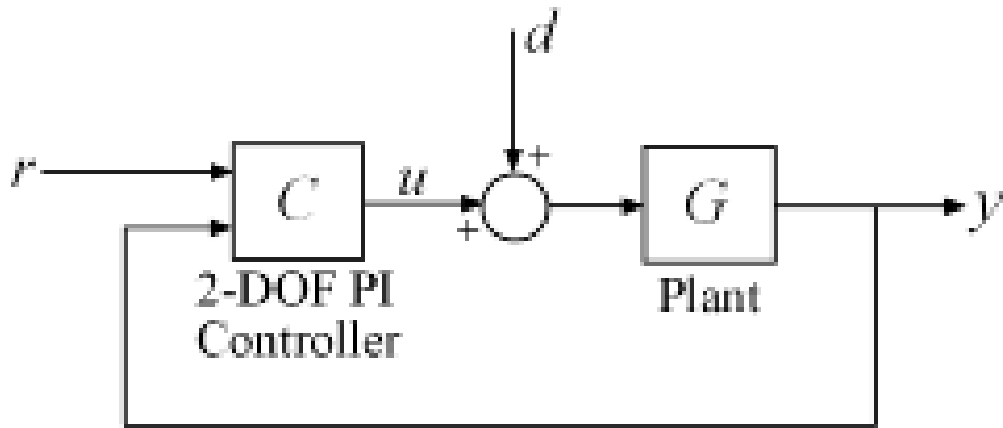
Figure 3: 2-DOF PI Control Loop.

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 = 0.99996, Iterations = 123
```
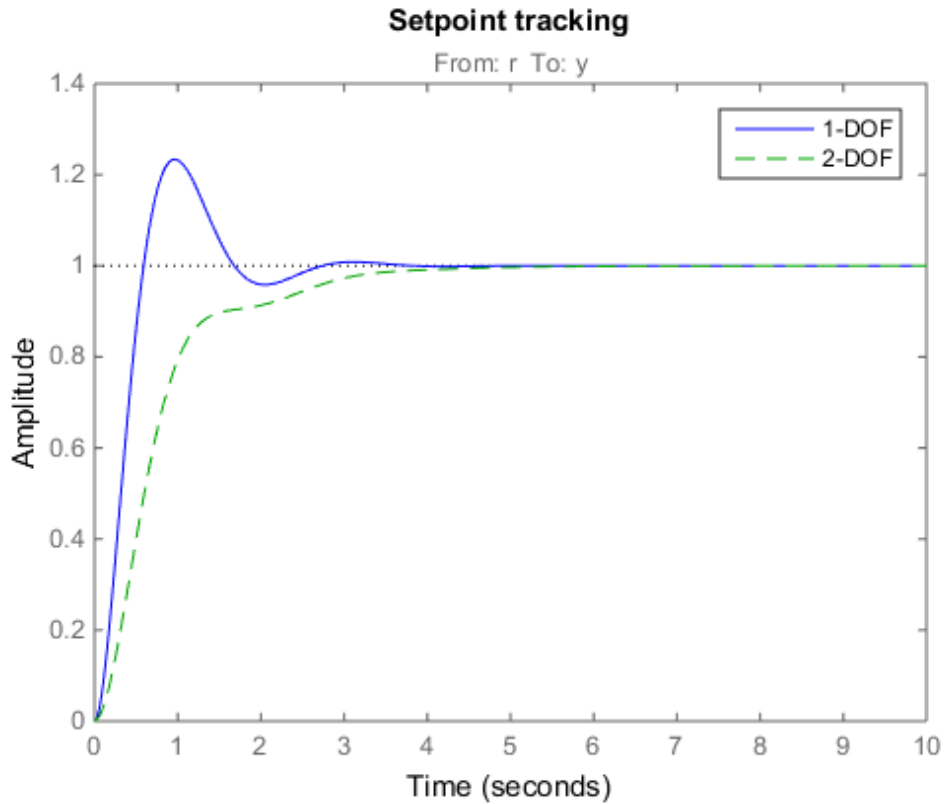
Compare the setpoint tracking and disturbance rejection properties of the 1-DOF and 2-DOF designs for $\alpha = 5$.
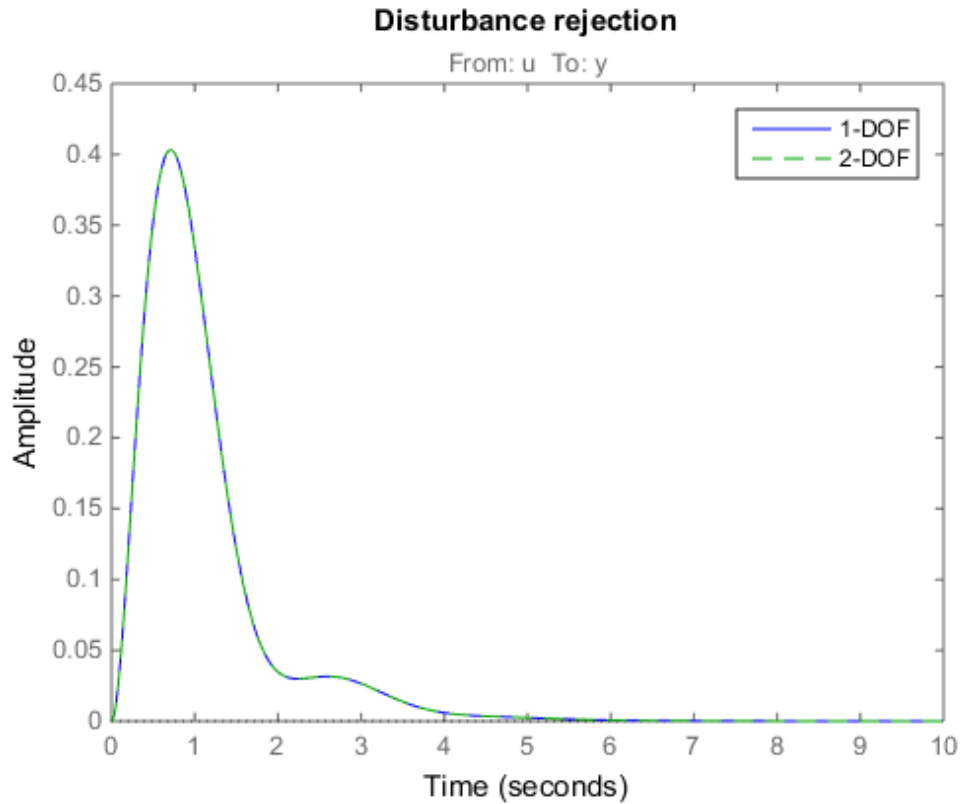
```
clf, step(T3,'b',T4,'g--',10)
```

7-57

```
title('Setpoint tracking')
legend('1-DOF','2-DOF')
```

Setpoint tracking


```
D4 = getIOTransfer(T4,'u','y');
step(D3,'b',D4,'g--',10)
title('Disturbance rejection')
legend('1-DOF','2-DOF')
```

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 =

            1
  Kp + Ki * ---
            s

  with Kp = 1.94, Ki = 2.13
```

```
Name: C
Continuous-time PI controller in parallel form.

showTunable(T4)   % 2-DOF PI

C =

                    1                 s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                    s               Tf*s+1

  with Kp = 1.943, Ki = 2.1264, Kd = 0, Tf = 1, b = 0.32476, 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.
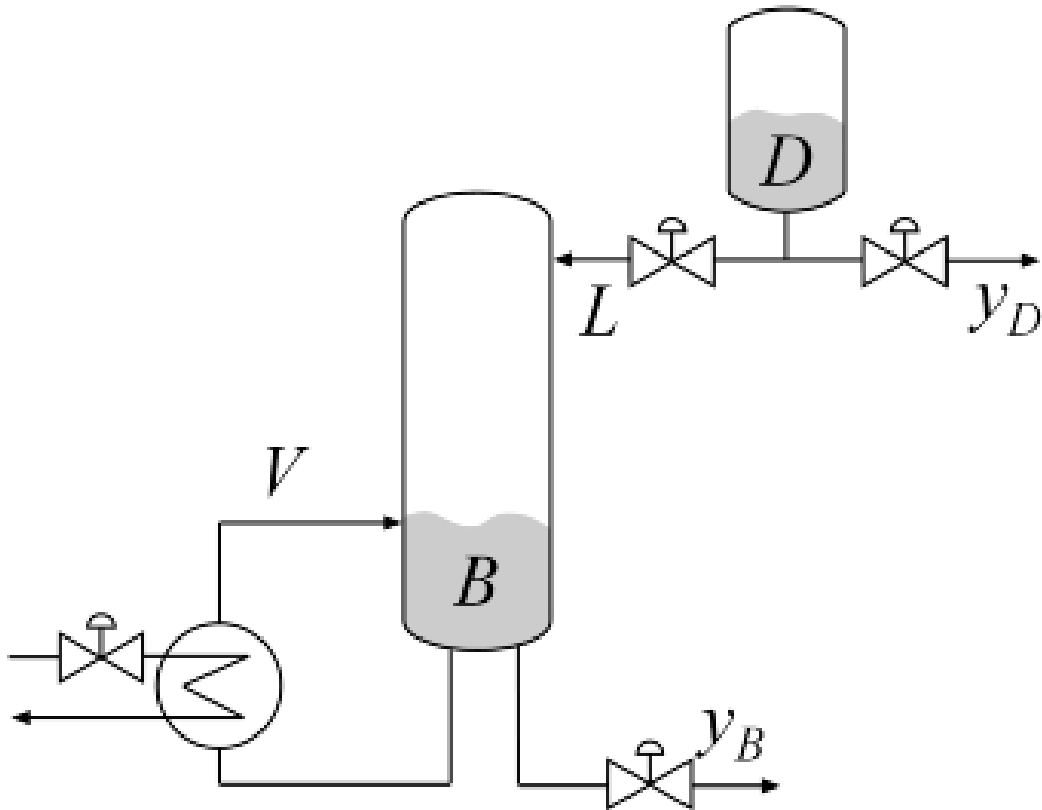

Figure 1: Distillation Column

In the so-called LV configuration, the controlled variables are the concentrations yD and yB 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')
```
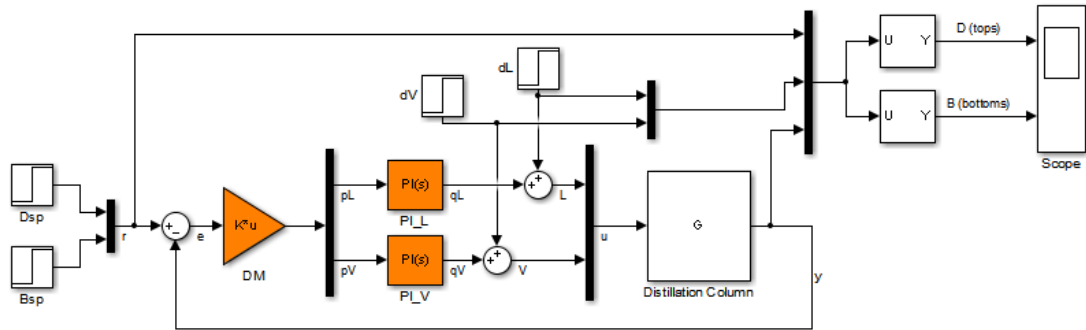
Decoupling controller for a distillation column

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 `slTuner` interface to specify the tuned blocks, the controller I/Os, and signals of interest for closed-loop validation.

```
STO = slTuner('rct_distillation',{'PI_L','PI_V','DM'});

% Signals of interest
addPoint(STO,{'r','dL','dV','L','V','y'})
```

Use `TuningGoal` objects to express the design requirements. Set the control bandwidth by using [0.1,0.5] as gain crossover band for the open-loop response.

```
wc = [0.1,0.5];
```

Specify that each loop should respond to a step command in about 15 minutes with minimum cross-coupling.

```
TR = TuningGoal.Tracking('r','y',15);
```

Also specify how the feedback loop should reject step disturbances at the plant input. The response should be well damped, settle in less than 20 minutes, and not exceed 4 in amplitude.

```
DR = TuningGoal.StepRejection({'dL','dV'},'y',4,20);
```

Next use `looptune` to tune the controller blocks `PI_L`, `PI_V`, and `DM` subject to these requirements.

```
Controls = {'L','V'};
Measurements = 'y';
[ST,gam,Info] = looptune(STO,Controls,Measurements,wc,TR,DR);
```

```
Final: Peak gain = 1, Iterations = 44
Achieved target gain value TargetGain=1.
```

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.
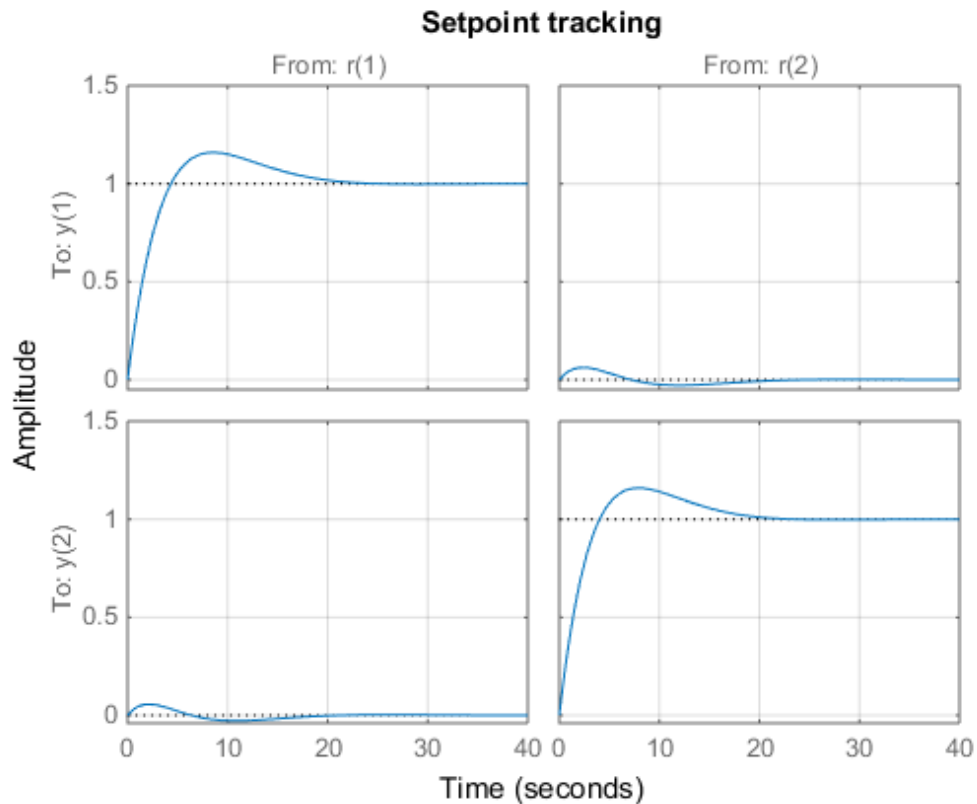
```
figure('Position',[10,10,1071,714])
loopview(ST,Info)
```
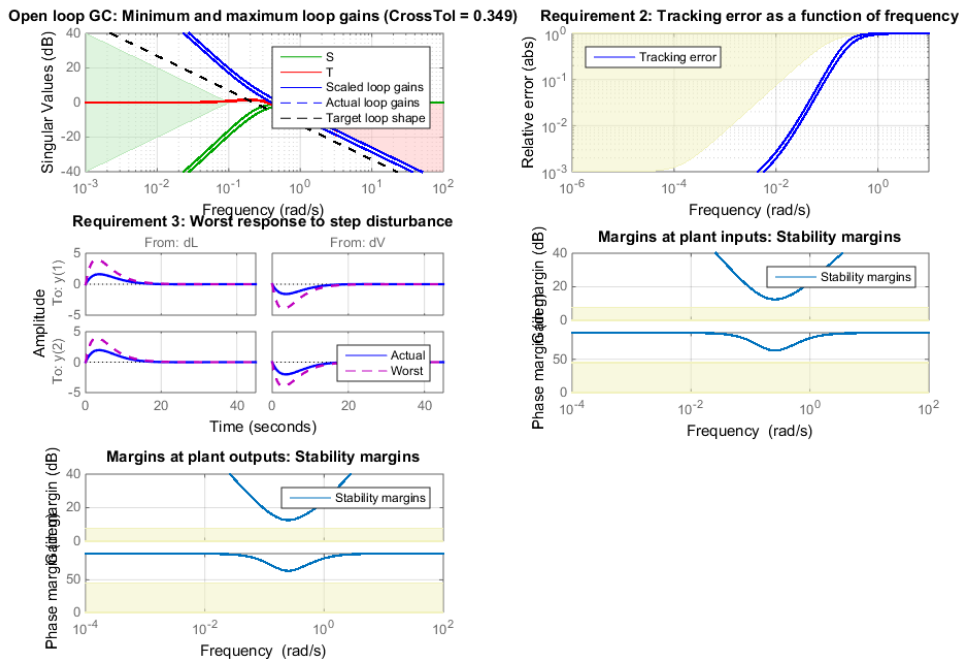
Use `getIOTransfer` to access and plot the closed-loop responses from reference and disturbance to the tops and bottoms concentrations. The tuned responses show a good compromise between tracking and disturbance rejection.

```
figure
Ttrack = getIOTransfer(ST,'r','y');
step(Ttrack,40), grid, title('Setpoint tracking')
```



```
Treject = getIOTransfer(ST,{'dV','dL'},'y');
step(Treject,40), grid, title('Disturbance rejection')
```
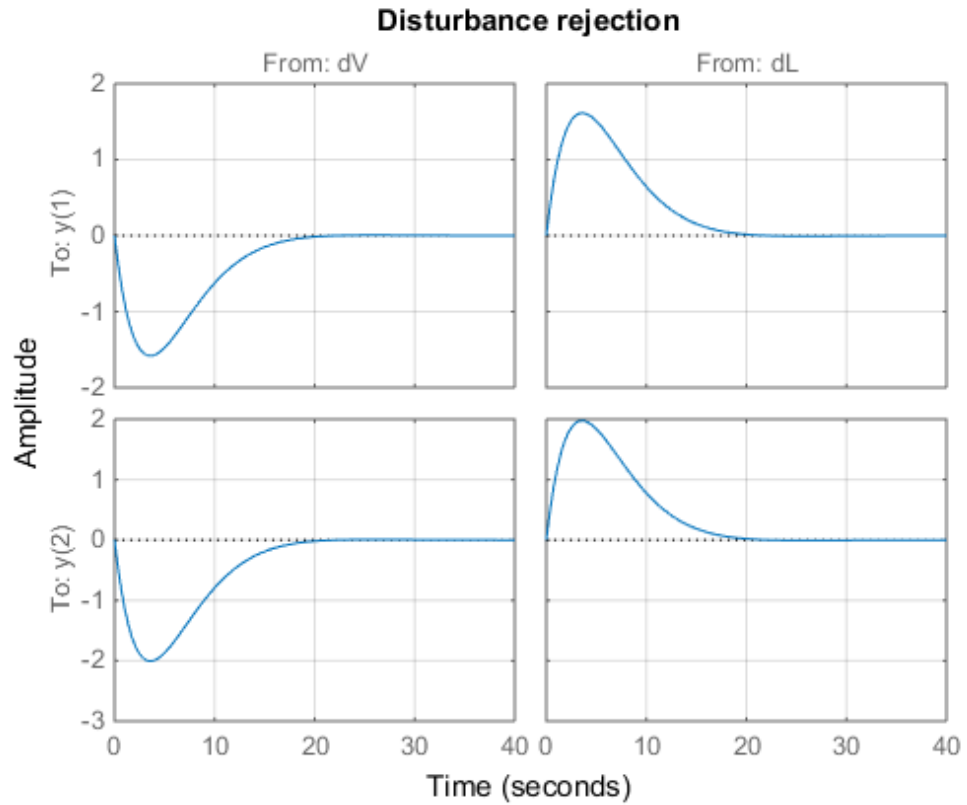
Disturbance rejection

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.
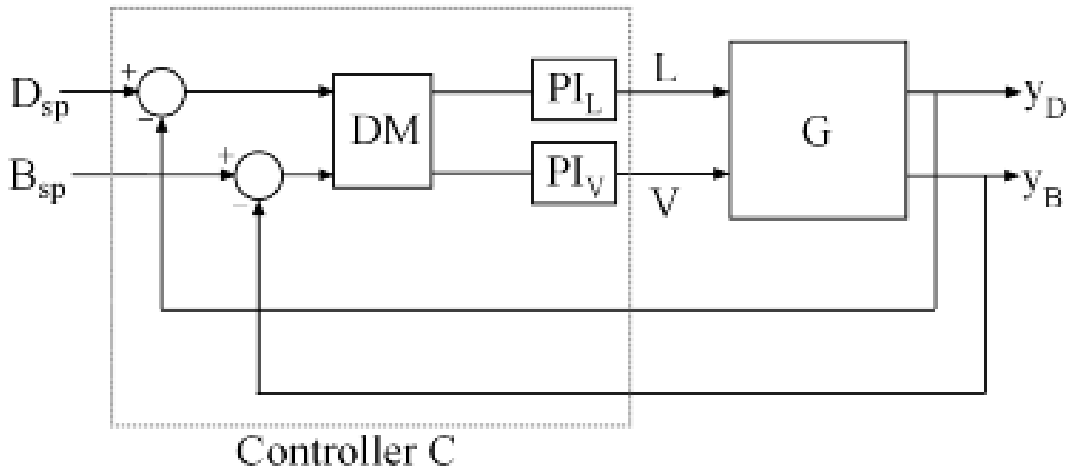
Figure 2: Block Diagram of Control System

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 `CO` of the controller C in Figure 2.

```
CO = blkdiag(PI_L,PI_V) * DM * [eye(2) -eye(2)];
```

```matlab
% 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.

```matlab
% Crossover band
wc = [0.1,0.5];

% Tracking requirement
TR = TuningGoal.Tracking({'Dsp','Bsp'},{'yD','yB'},15);

% Disturbance rejection requirement
DR = TuningGoal.StepRejection({'L','V'},{'yD','yB'},4,20);

[~,C] = looptune(G,CO,wc,TR,DR);

Final: Peak gain = 1, Iterations = 70
Achieved target gain value TargetGain=1.
```
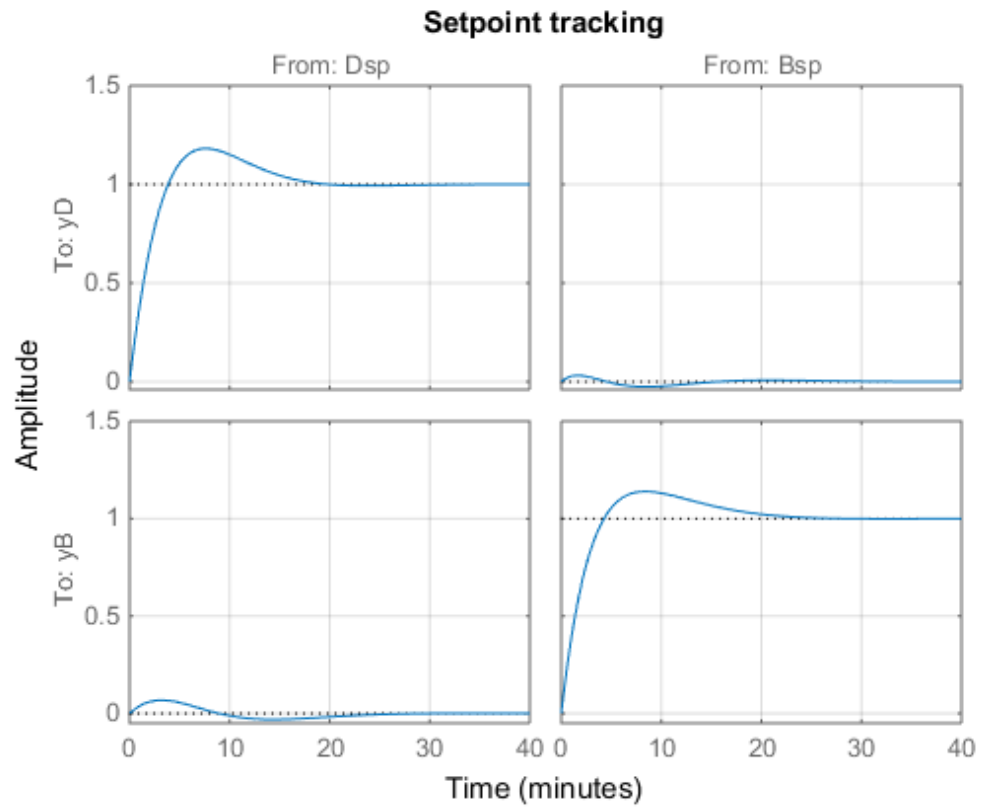
To validate the design, close the loop with the tuned compensator `C` and simulate the step responses for setpoint tracking and disturbance rejection. Also compare the open- and closed-loop disturbance rejection characteristics in the frequency domain.

```matlab
Tcl = connect(G,C,{'Dsp','Bsp','L','V'},{'yD','yB'});

Ttrack = Tcl(:,[1 2]);
step(Ttrack,40), grid, title('Setpoint tracking')
```
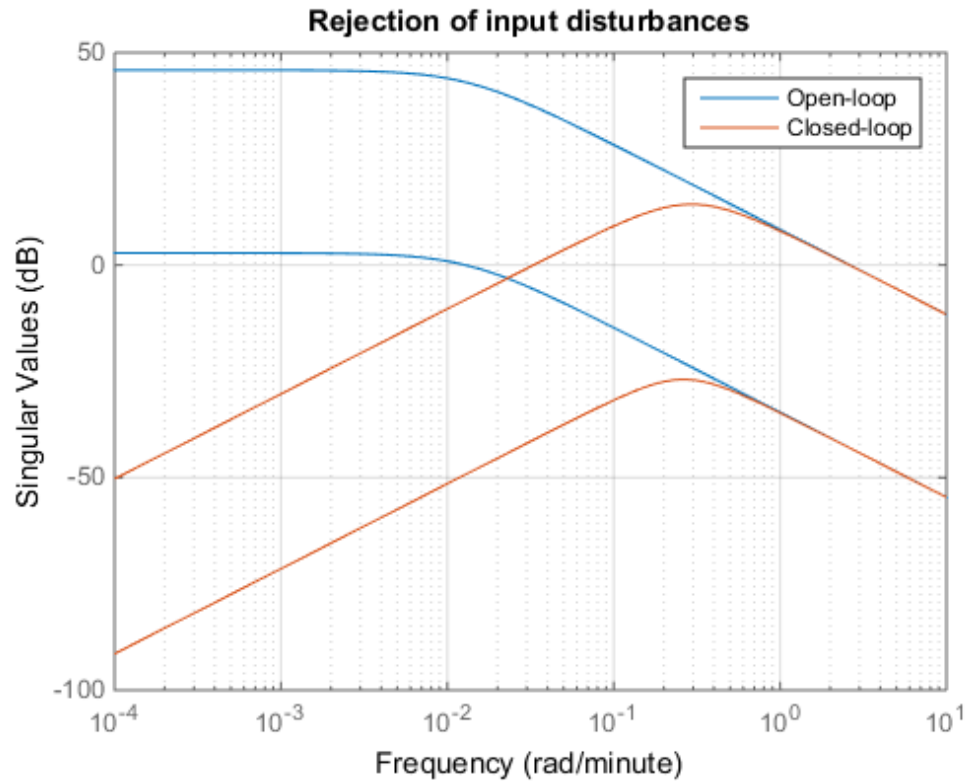
```
Treject = Tcl(:,[3 4]);
Treject.InputName = {'dL','dV'};
step(Treject,40), grid, title('Disturbance rejection')
```

Disturbance rejection
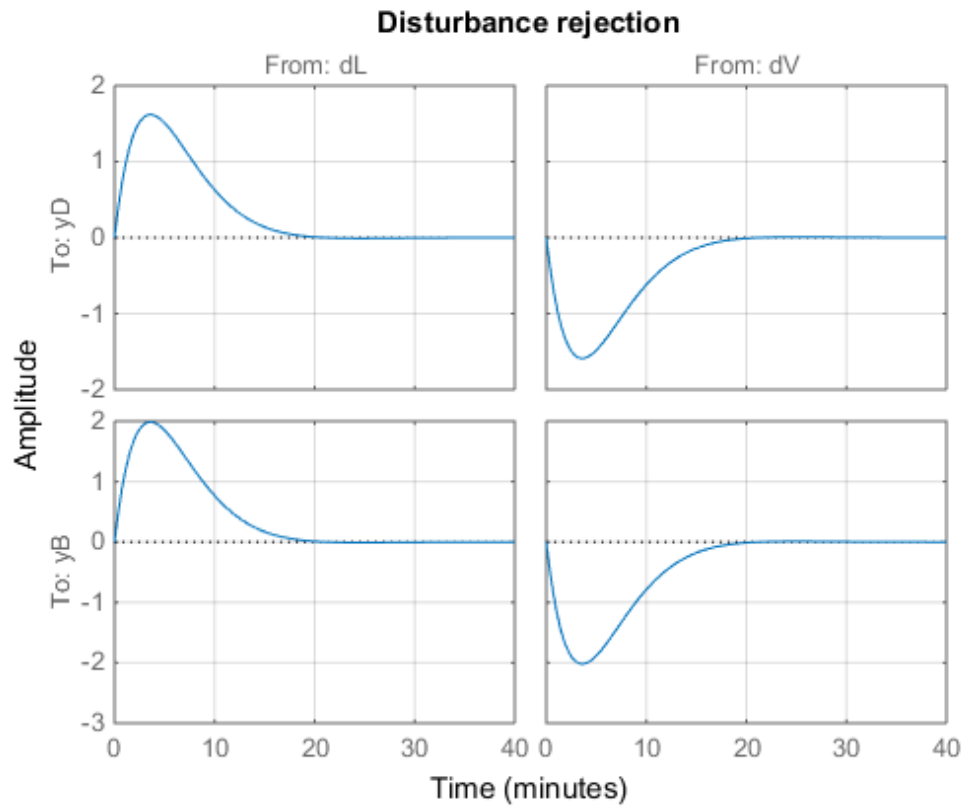
```
clf, sigma(G,Treject), grid
title('Rejection of input disturbances')
legend('Open-loop','Closed-loop')
```

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

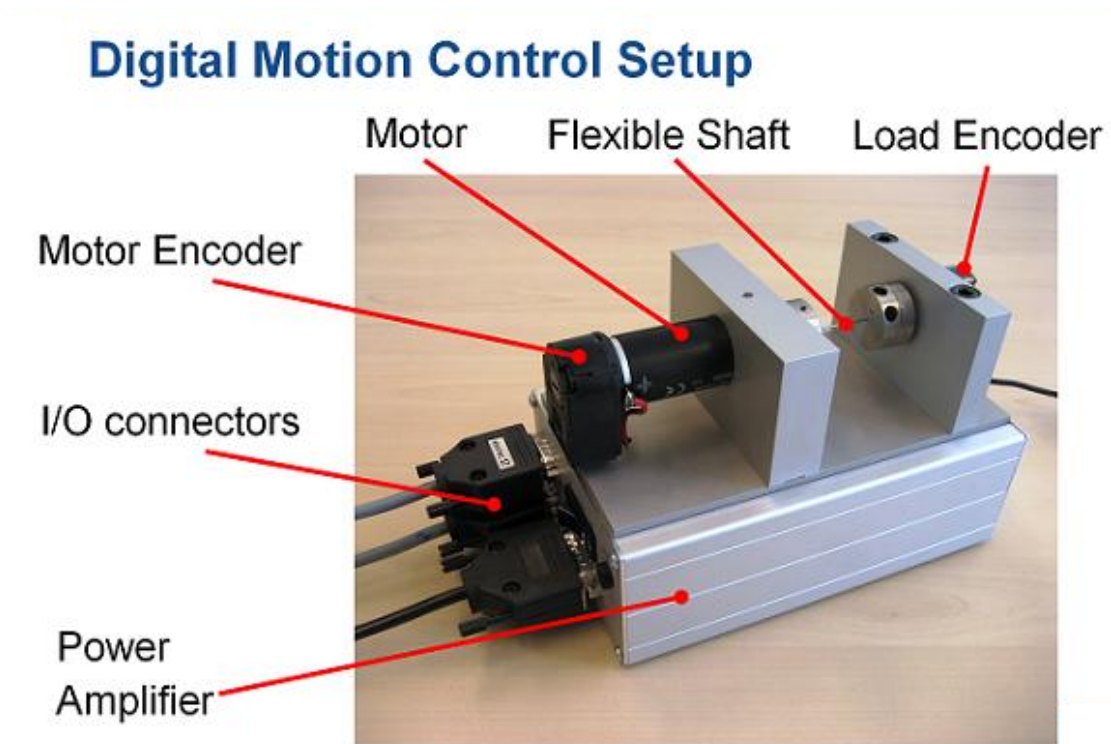The motion system under consideration is shown below.

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
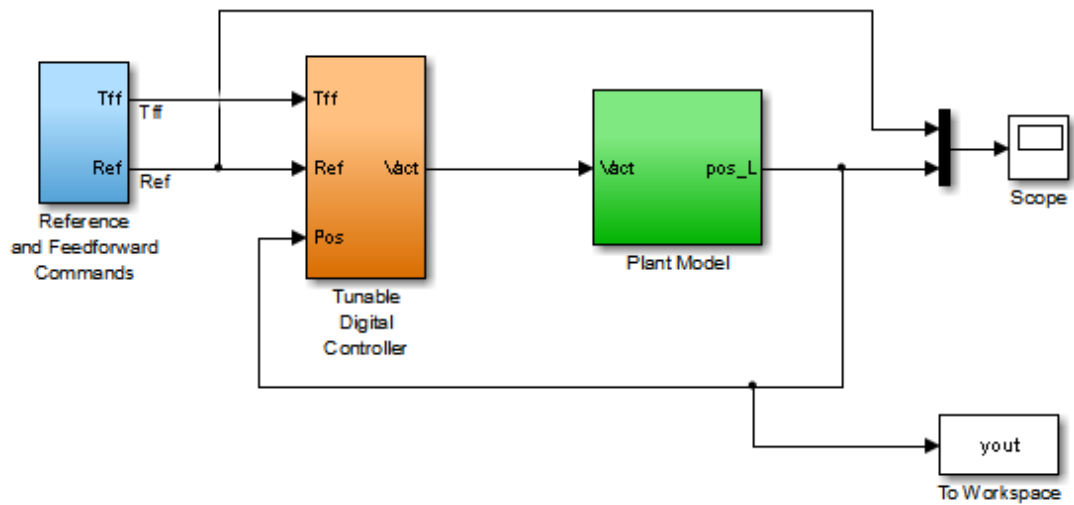
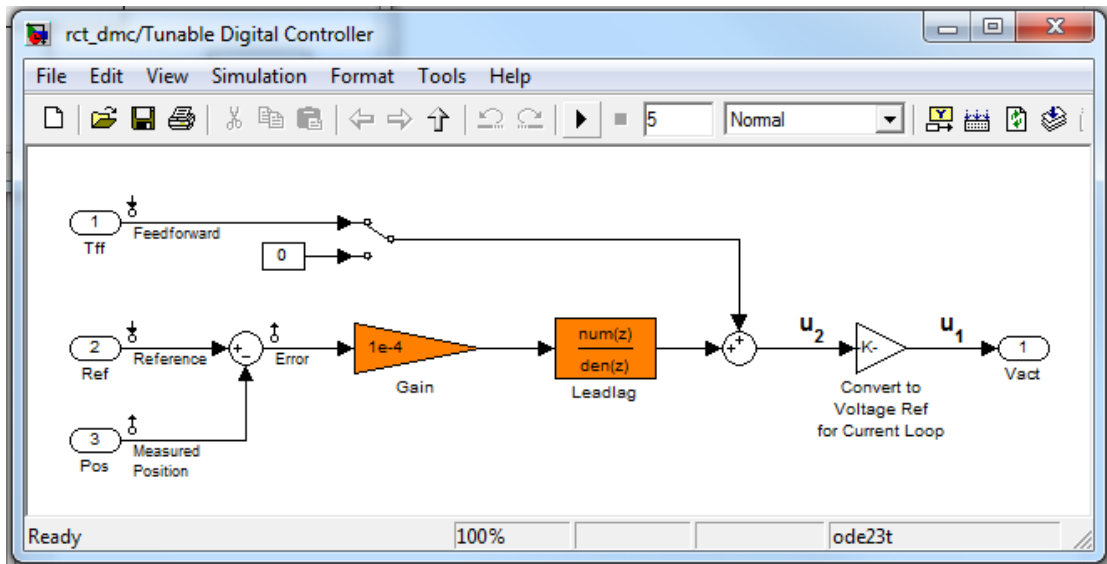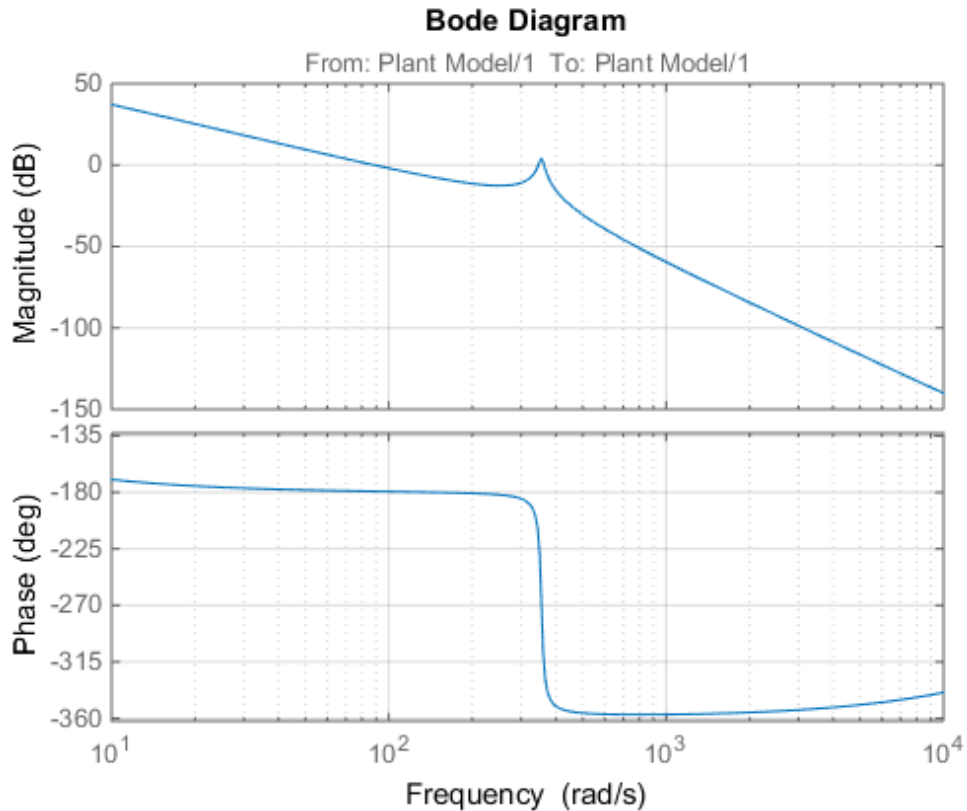The "Tunable Digital Controller" consists of a gain in series with a lead/lag controller.

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
```

Bode Diagram

From: Plant Model/1 To: Plant Model/1

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 slTuner interface to automatically acquire the control structure from Simulink. Note that the signals of interest are already marked as Linear Analysis Points in the Simulink model.

```
ST0 = slTuner('rct_dmc',{'Gain','Leadlag'});
```

Next use looptune to tune the compensator parameters for the target gain crossover frequency of 5 rad/s:

```
Measurement = 'Measured Position';  % controller input
Control = 'Leadlag';                % controller output
ST1 = looptune(STO,Control,Measurement,5);

Final: Peak gain = 0.975, Iterations = 21
Achieved target gain value TargetGain=1.
```

A final value below or near 1 indicates success. Inspect the tuned values of the gain and lead/lag filter:

```
showBlockValue(ST1)

AnalysisPoints_ =

  d =
      u1  u2  u3  u4  u5
   y1   1   0   0   0   0
   y2   0   1   0   0   0
   y3   0   0   1   0   0
   y4   0   0   0   1   0
   y5   0   0   0   0   1

Name: AnalysisPoints_
Static gain.
-----------------------------------
Gain =

  d =
           u1
   y1  2.753e-06

Name: Gain
Static gain.
-----------------------------------
Leadlag =

  30.54 s + 59.01
  ---------------
     s + 18.94

Name: Leadlag
Continuous-time transfer function.
```

Design Validation

To validate the design, use the `slTuner` interface to quickly access the closed-loop transfer functions of interest and compare the responses before and after tuning.

```
T0 = getIOTransfer(ST0,'Reference','Measured Position');
T1 = getIOTransfer(ST1,'Reference','Measured Position');
step(T0,T1), grid
legend('Original','Tuned')
```

Step Response

From: Reference To: Measured Position
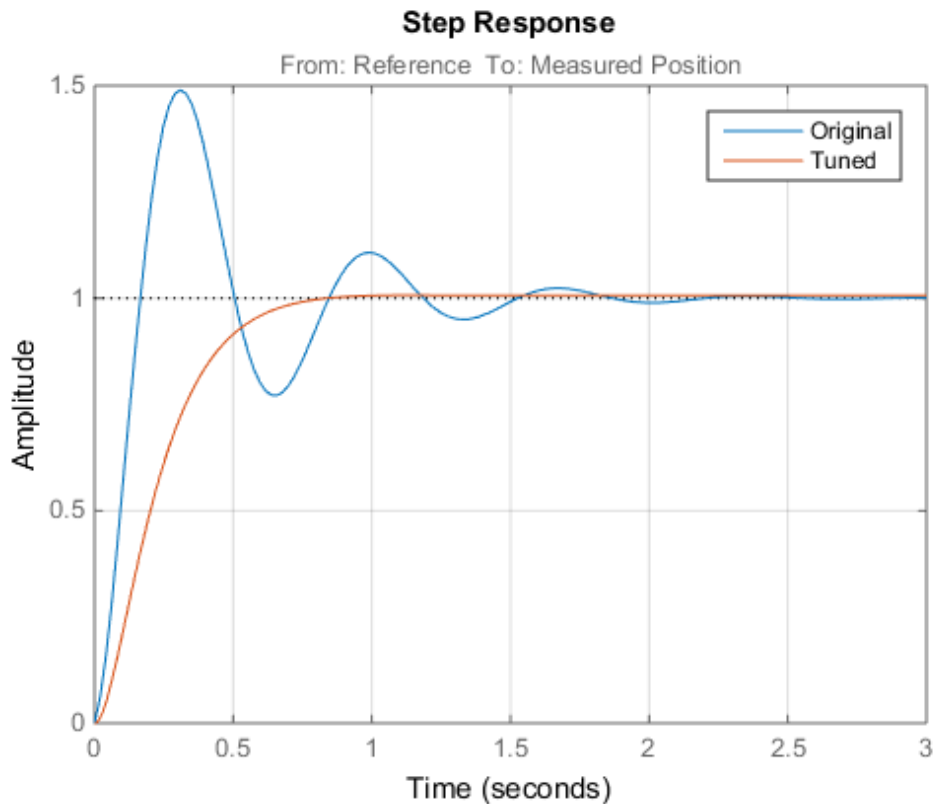
The tuned response has significantly less overshoot and satisfies the response time requirement. However these simulations are obtained using a continuous-time lead/ lag compensator (`looptune` operates in continuous time) so we need to further validate the design in Simulink using a digital implementation 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.

```
writeBlockValue(ST1)
```

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

Step Response
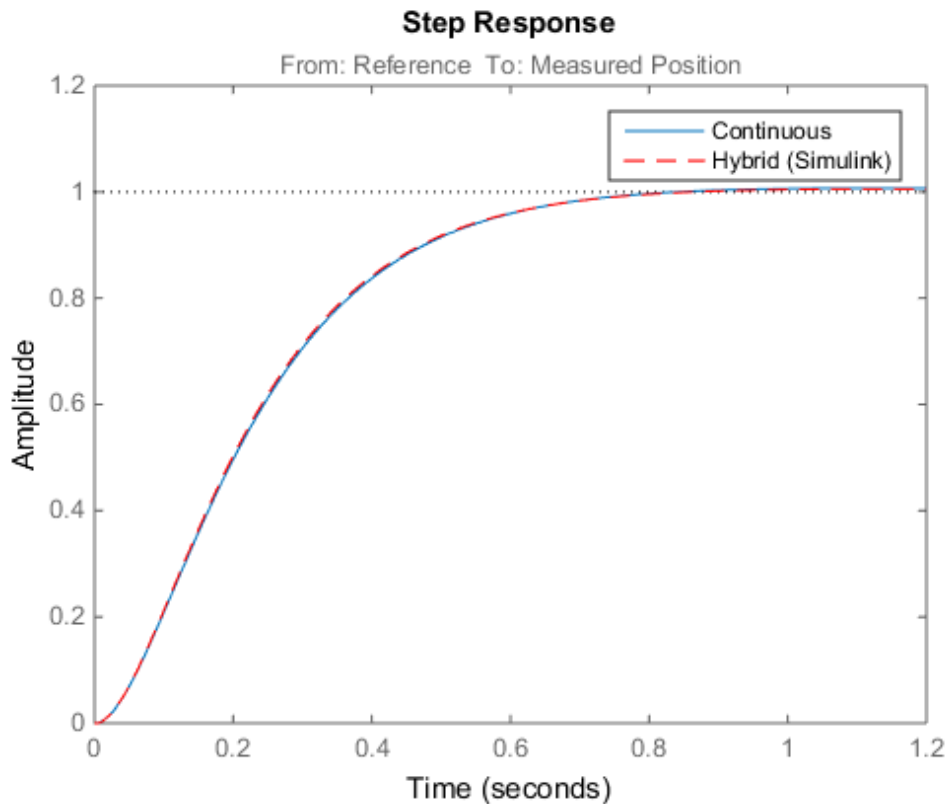
The simulations closely match and the coefficients of the digital lead/lag can be read from the "Leadlag" block in Simulink.

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.
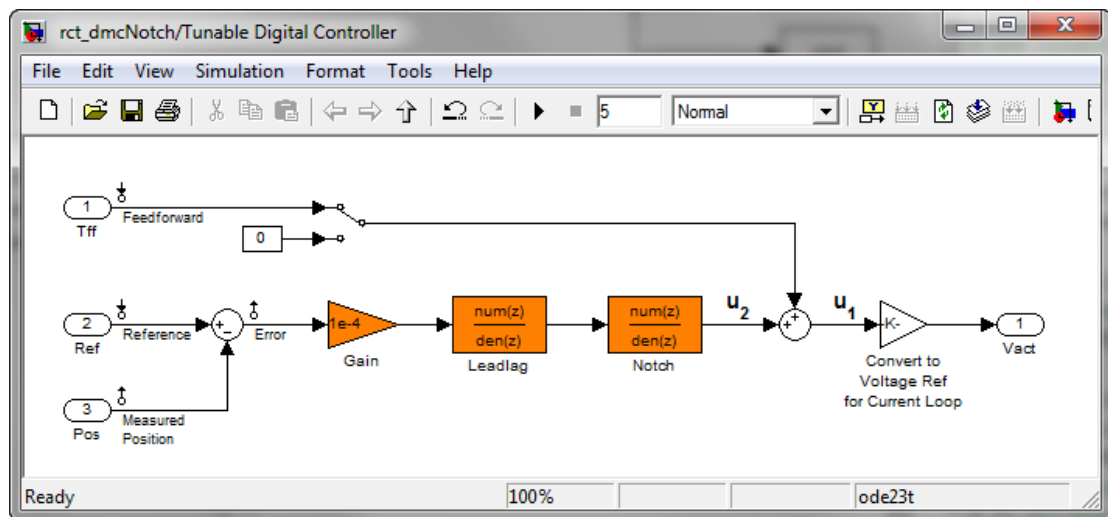

Figure 3: Digital Controller with Notch Filter

To tune this modified control architecture, create an `slTuner` instance with the three tunable blocks.

```
STO = slTuner('rct_dmcNotch',{'Gain','Leadlag','Notch'});
```

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.

```
setBlockParam(ST0,'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));

Measurement = 'Measured Position';  % controller input
Control = 'Notch';                   % controller output
ST2 = looptune(ST0,Control,Measurement,TLS);

Final: Peak gain = 1.05, Iterations = 66
```

The final gain is close to 1, indicating that all requirements are met. Compare the closed-loop step response with the previous designs.

```
T2 = getIOTransfer(ST2,'Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')
```

Step Response

From: Reference To: Measured Position
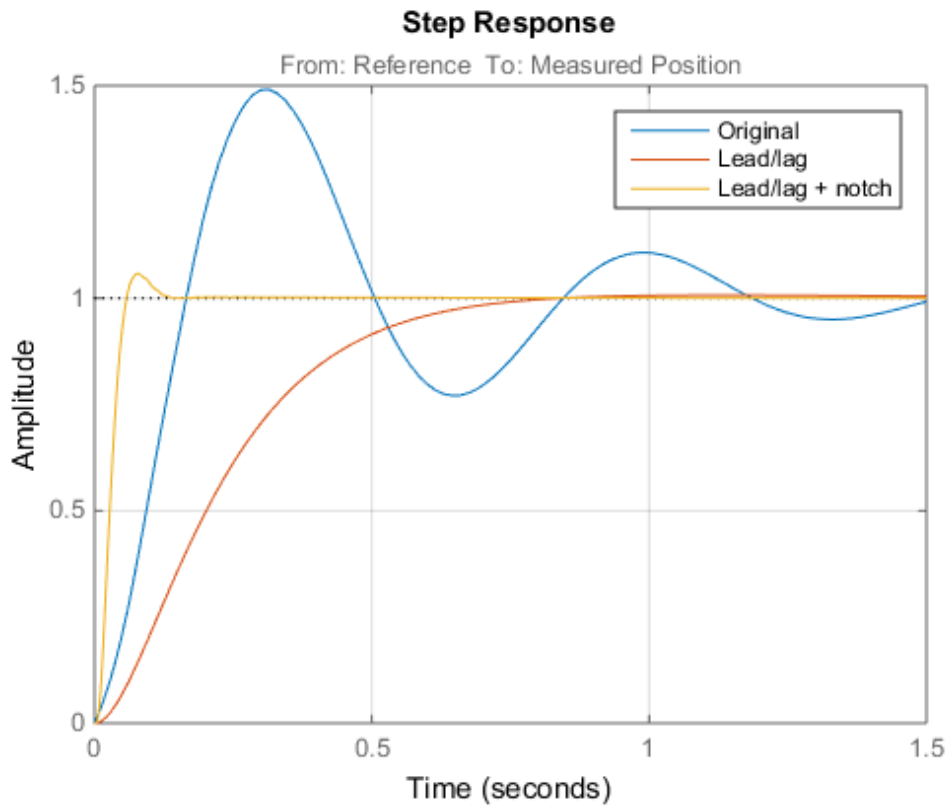
To verify that the notch filter performs as expected, evaluate the total compensator `C` and the open-loop response `L` and compare the Bode responses of `G`, `C`, `L`:

```
% Get tuned block values (in the order blocks are listed in ST2.TunedBlocks)
[g,LL,N] = getBlockValue(ST2);
C = N * LL * g;

L = getLoopTransfer(ST2,'Notch',-1);

bode(G,C,L,{1e1,1e3}), grid
legend('G','C','L')
```
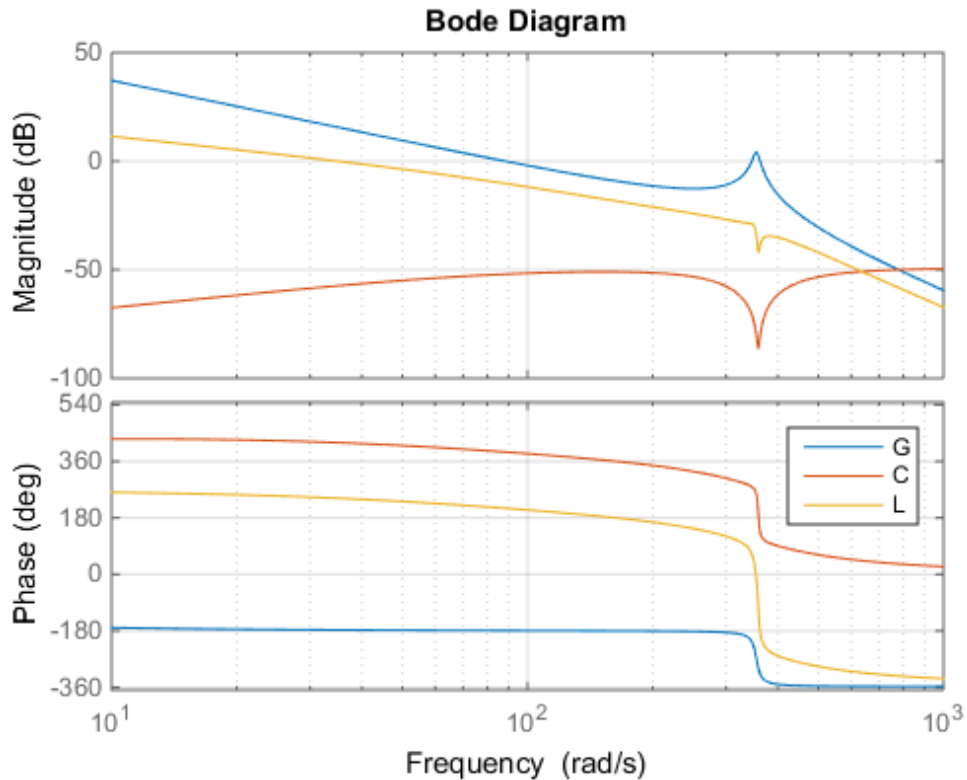
This Bode plot confirms that the plant resonance has been correctly "notched out."
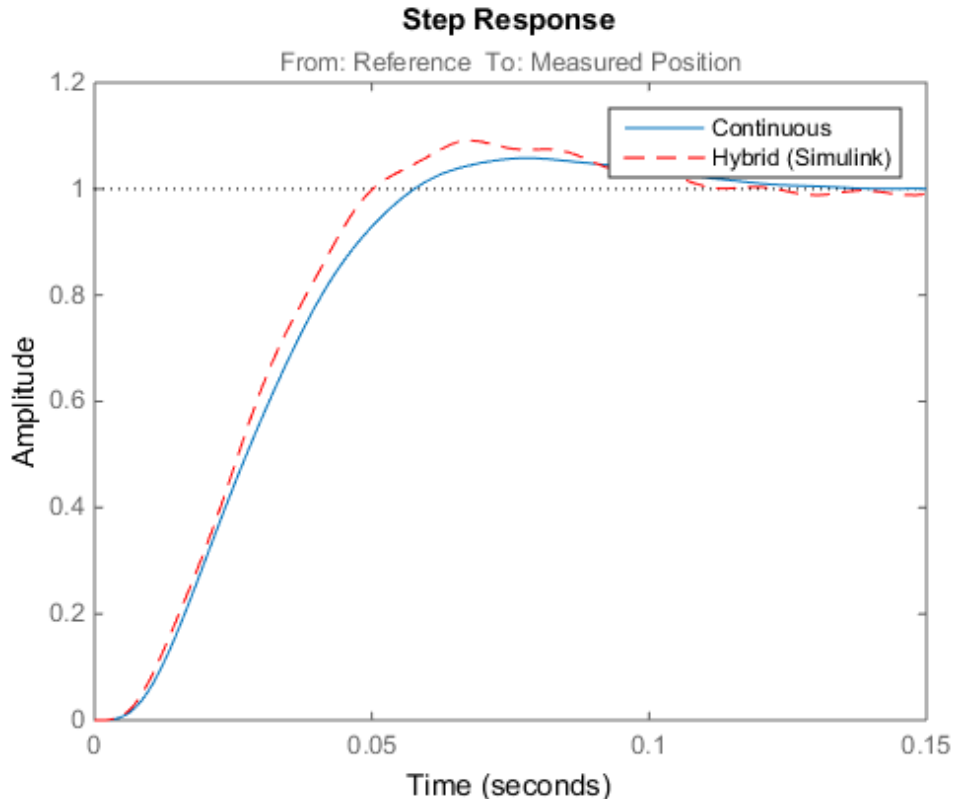
Discretizing the Notch Filter

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:

```
writeBlockValue(ST2)

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')
```
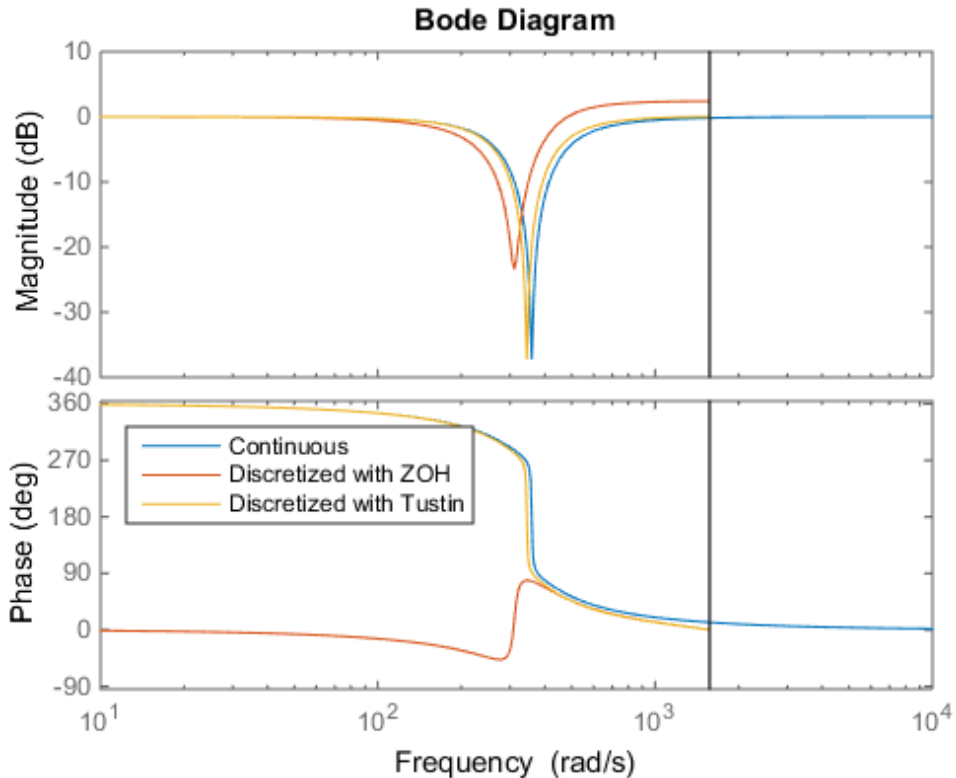

The ZOH method has significant distortion and prewarped Tustin should be used instead. To do this, specify the desired rate conversion method for the notch filter block:

```
setBlockRateConversion(ST2,'Notch','tustin',wn(1))
```

```
writeBlockValue(ST2)
```

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
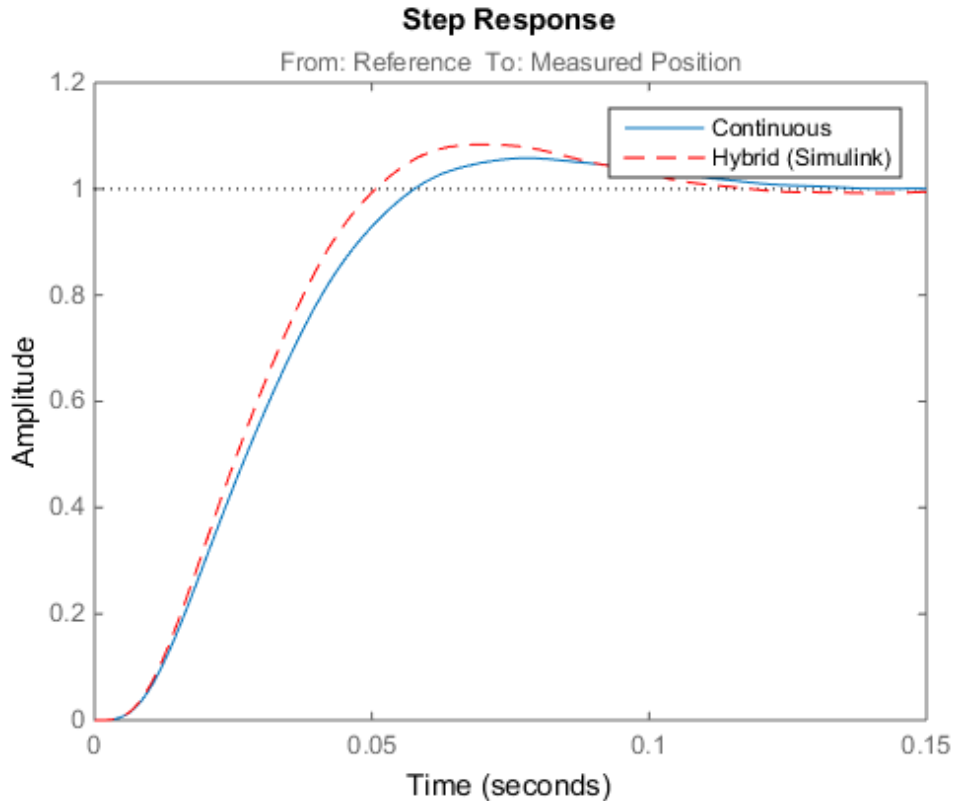
Discrete-Time Tuning

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:

```
ST0.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 = looptune(ST0,Control,Measurement,TLS,GL);
```

```
% Closed-loop responses
T2 = getIOTransfer(ST2,'Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')
```
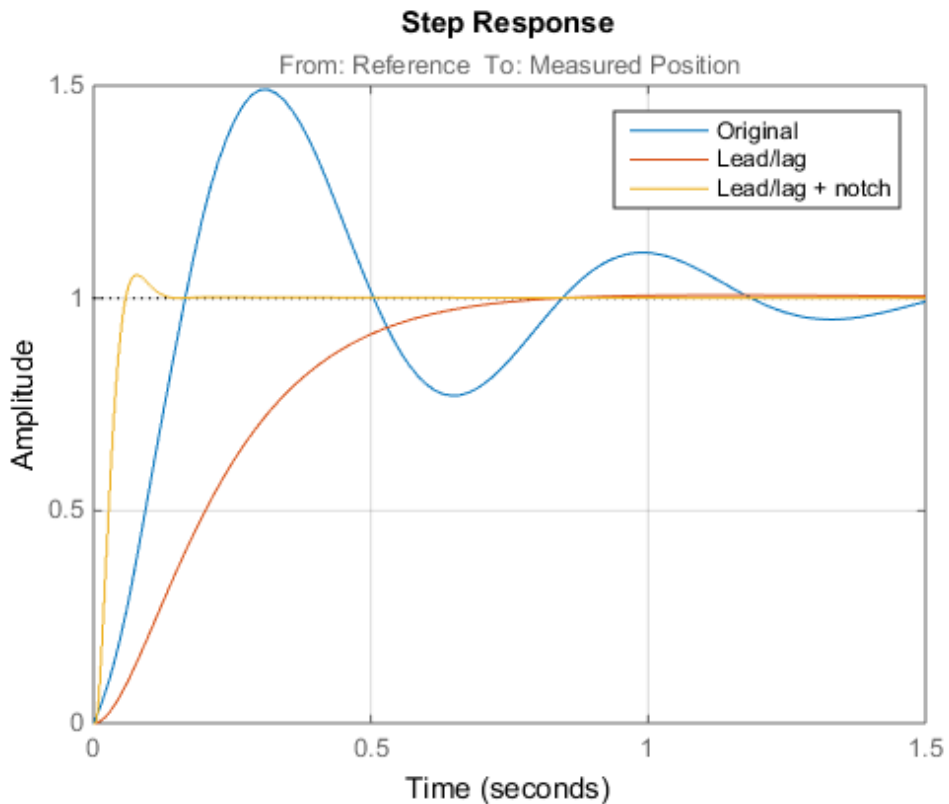
```
Final: Peak gain = 1.04, Iterations = 46
```

```
% Open-loop responses
[g,LL,N] = getBlockValue(ST2);
C = N * LL * g;
L = getLoopTransfer(ST2,'Notch',-1);
bode(G,C,L,{1e1,2e3}), grid
legend('G','C','L')
```
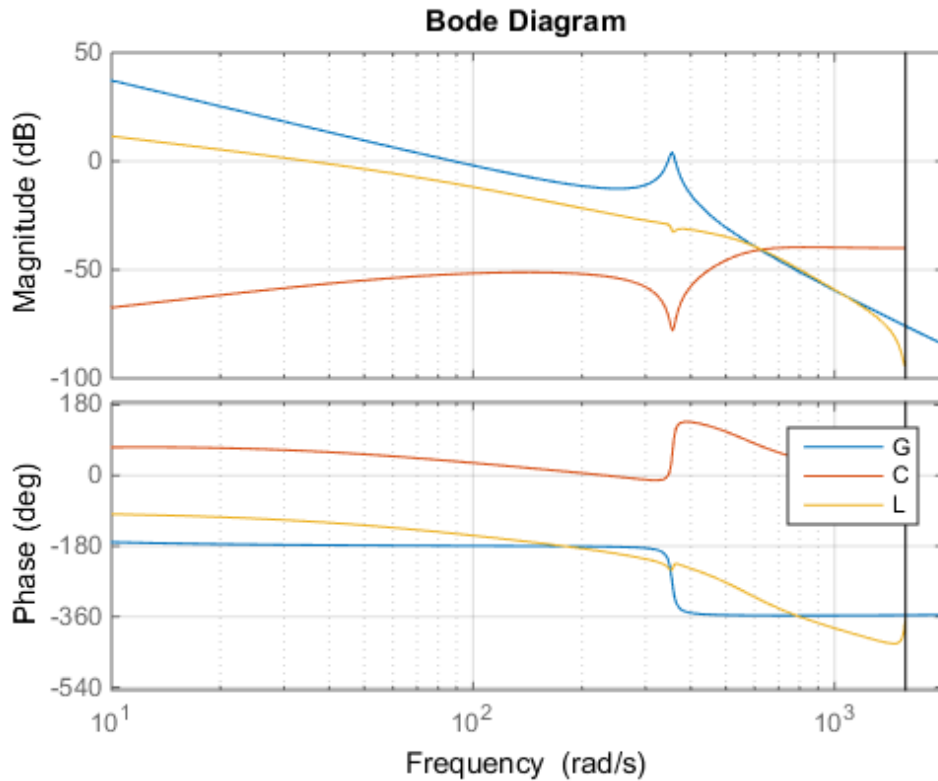

The results are similar to those obtained when tuning the controller in continuous time. Now validate the digital controller against the continuous-time plant in Simulink.
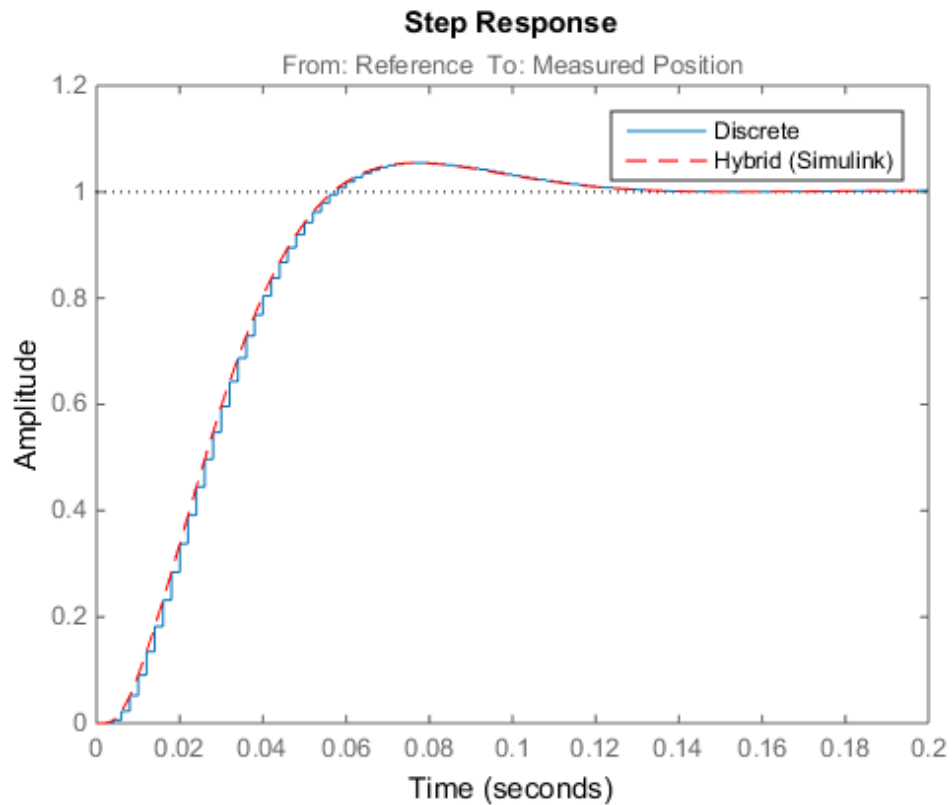
```
writeBlockValue(ST2)

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.
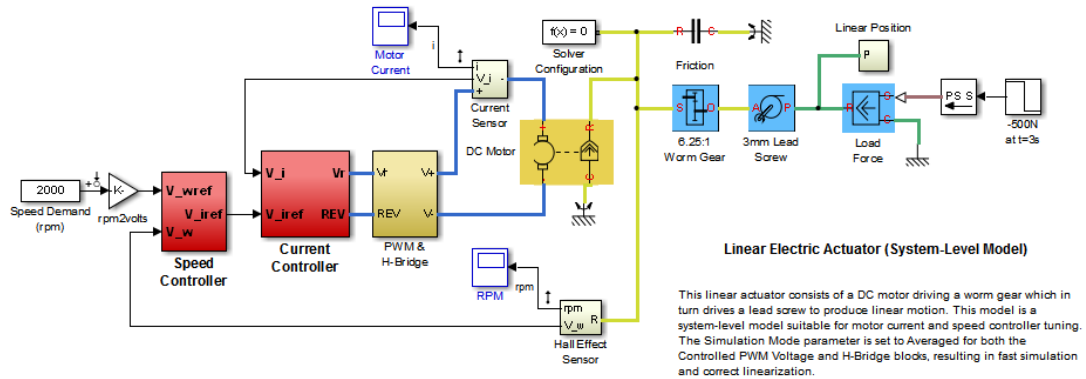
Control of a Linear Electric Actuator

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')

Linear Electric Actuator (System-Level Model)

This linear actuator consists of a DC motor driving a worm gear which in turn drives a lead screw to produce linear motion. This model is a system-level model suitable for motor current and speed controller tuning. The Simulation Mode parameter is set to Averaged for both the Controlled PWM Voltage and H-Bridge blocks, resulting in fast simulation and correct linearization.

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.

Current Controller

Speed Controller

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 `systune` to jointly tune both feedback loops. To set up the design, create an instance of the `slTuner` 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'};
tLinearize = 0.5;  % linearize at t=0.5

% Create tuning interface
ST0 = slTuner('rct_linact',TunedBlocks,tLinearize);
```

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 = getIOTransfer(ST1,'Speed Demand (rpm)',{'rpm','i'});
figure
step(T1)
```
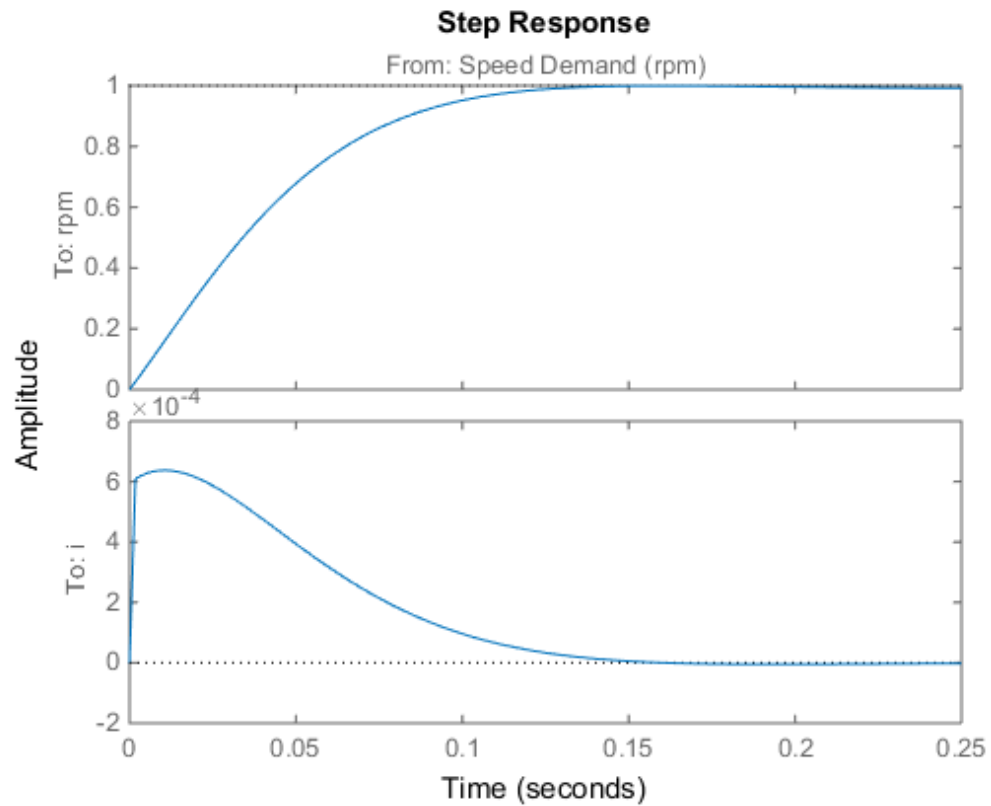
The response looks good in the linear domain so push the tuned gain values to Simulink and further validate the design in the nonlinear model.

```
writeBlockValue(ST1)
```

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).

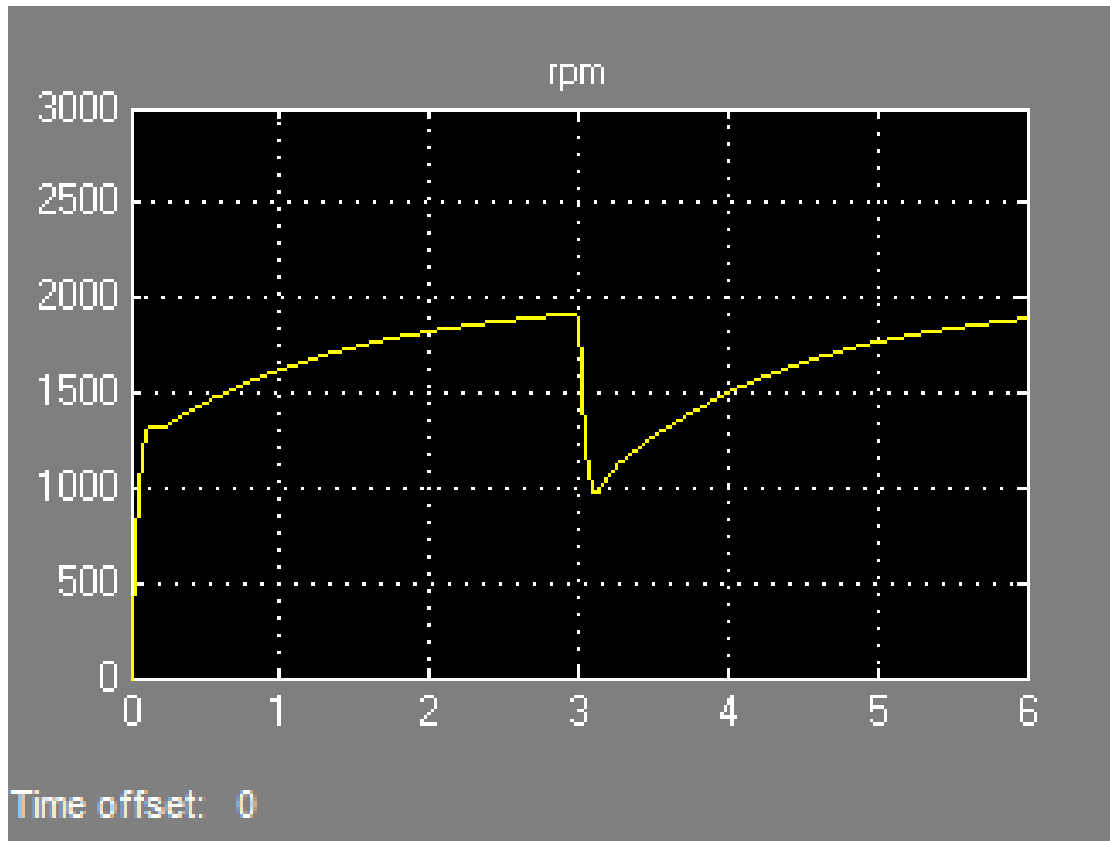**Figure 3: Nonlinear Simulation of Tuned Controller.**
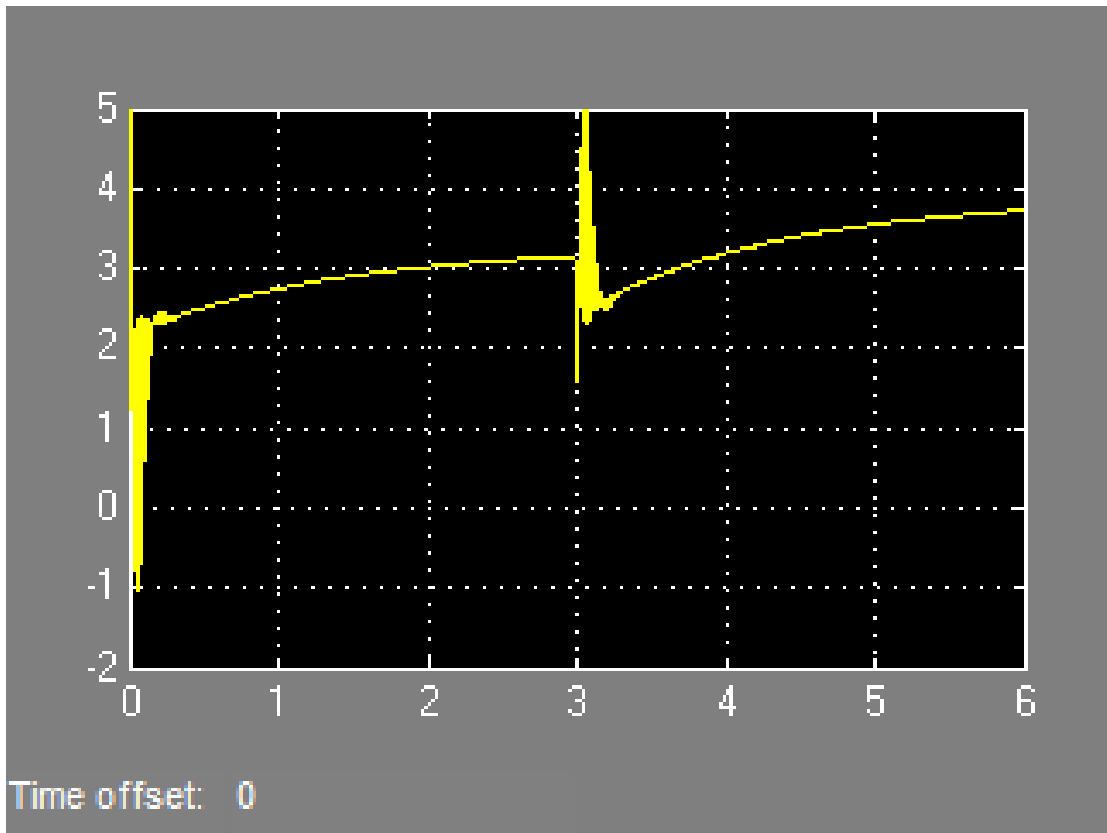
Figure 4: Current Controller Output (limited to plus/minus 5).

Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves `systune` 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.

```matlab
% Mark the control signals as points of interest so that they can be
% referenced in the gain requirements
addPoint(ST0,{'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.

```matlab
figure('Position',[100,100,560,550])
viewSpec([TR,MG1,MG2],ST2,info)
```
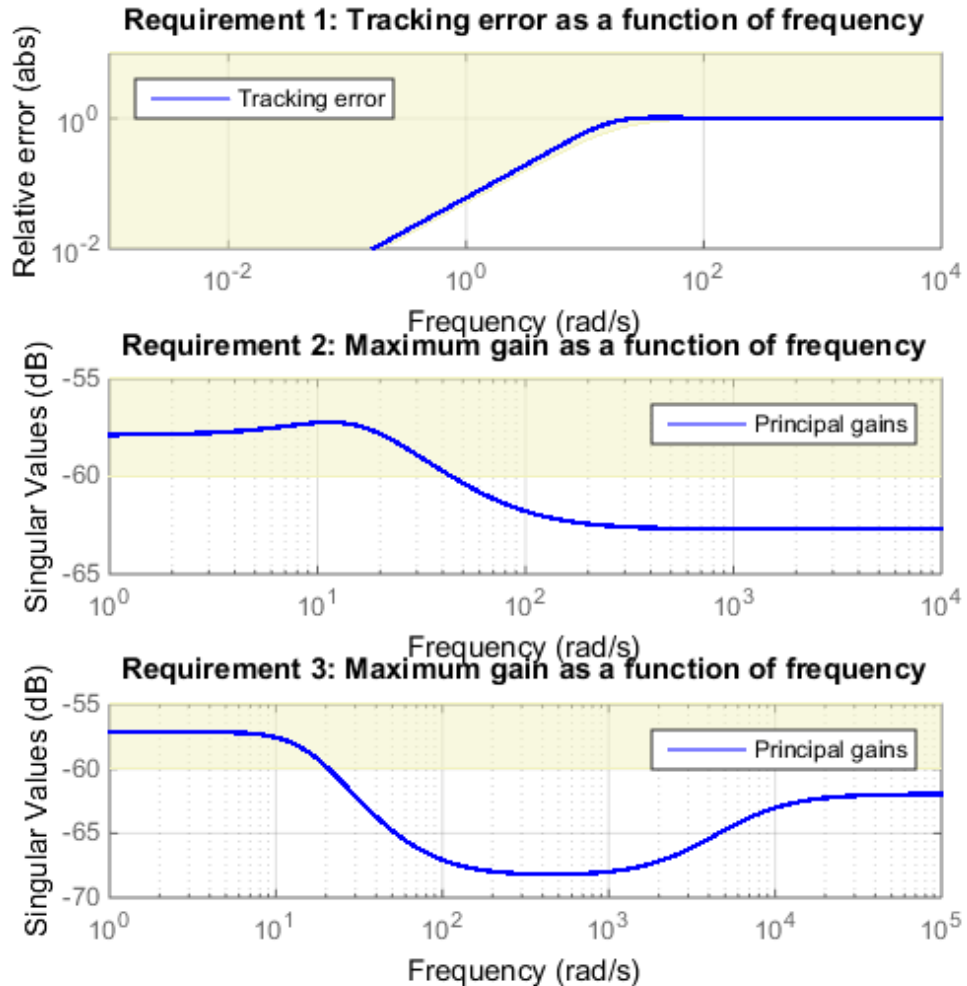
Next compare the two designs in the linear domain.

```
T2 = getIOTransfer(ST2,'Speed Demand (rpm)',{'rpm','i'});
figure
step(T1,'b',T2,'g--')
legend('Initial tuning','Tuning with Gain Constraints')
```
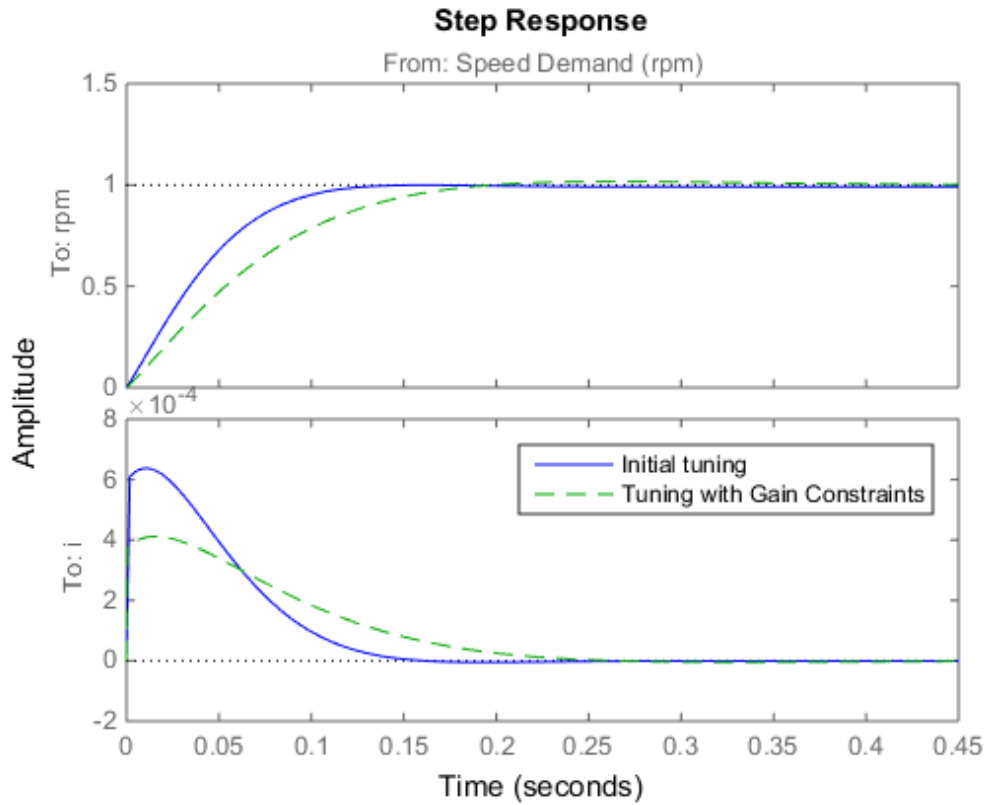
The second design is less aggressive but still meets the response time requirement. Finally, push the new tuned gain values to the Simulink model and simulate the response to a 2000 rpm speed demand and 500 N load disturbance. The simulation results appear in Figure 5 and the current controller output is shown in Figure 6.

```
writeBlockValue(ST2)
```
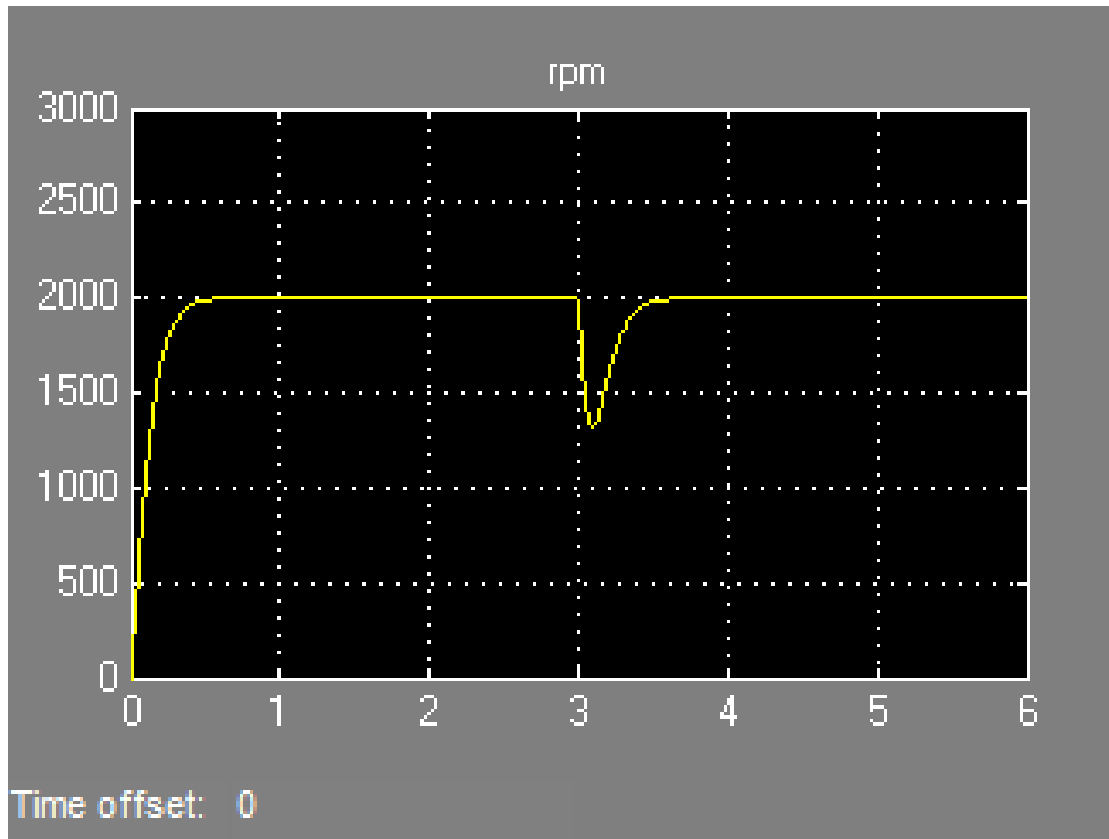
Figure 5: Nonlinear Response of Tuning with Gain Constraints.
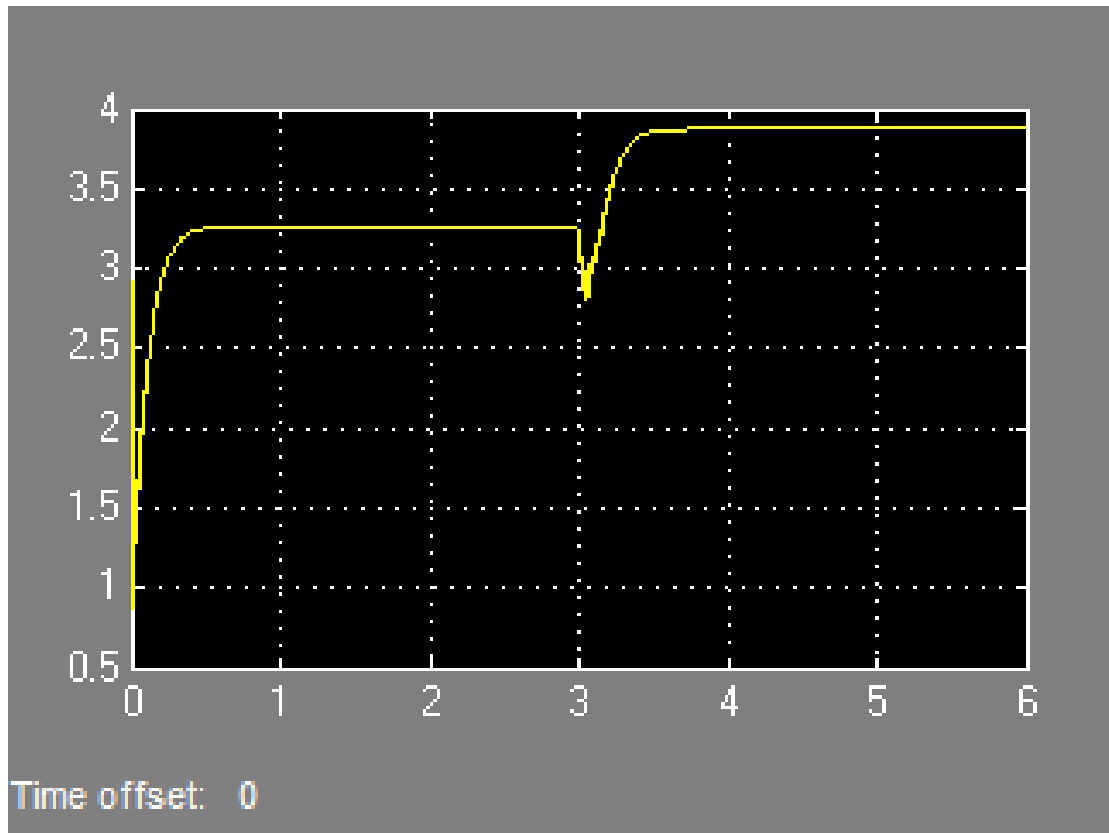
Figure 6: Current Controller Output.

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')
```
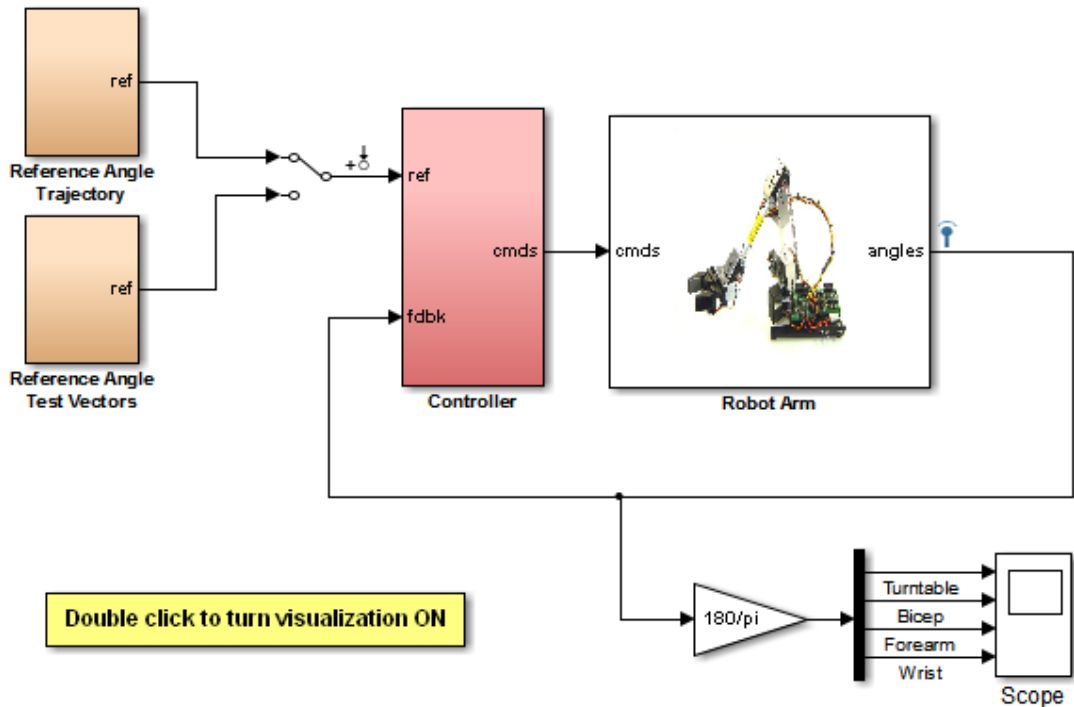

The controller consists of four PID controllers (one per joint). Each PID controller is implemented using the "2-DOF PID Controller" block from the Simulink library (see *PID Tuning for Setpoint Tracking vs. Disturbance Rejection* example for motivation).
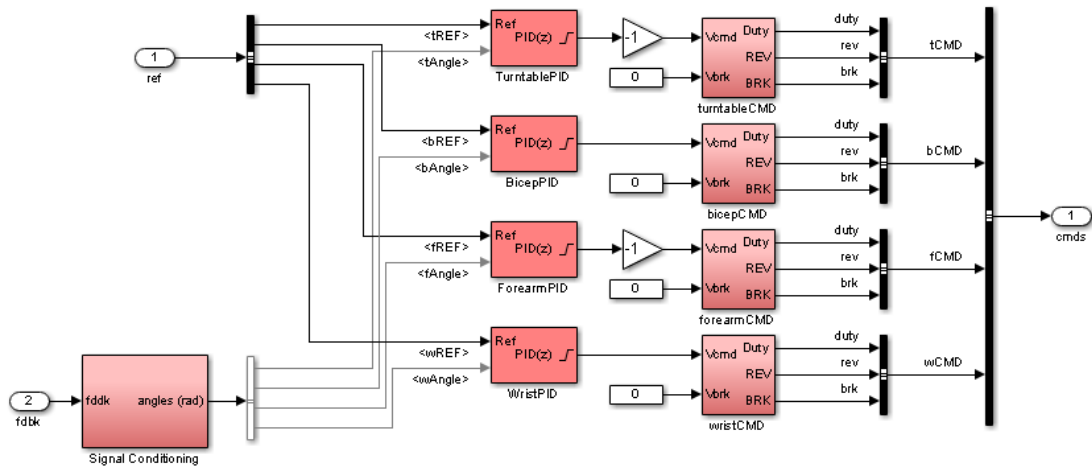
Figure 2: Controller structure.

Typically, such multi-loop controllers are tuned sequentially by tuning one PID loop at a time and cycling through the loops unt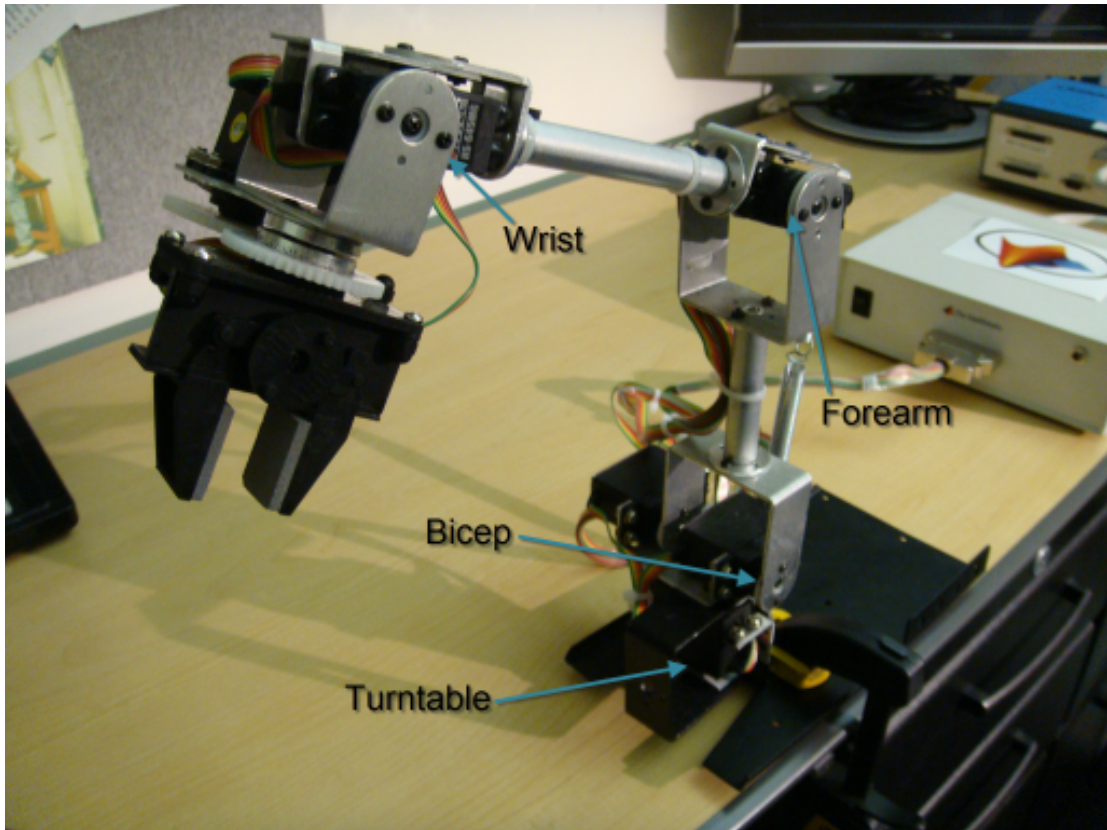il the overall behavior is satisfactory. This process can be time consuming and is not guaranteed to converge to the best overall tuning. Alternatively, you can use `systune` or `looptune` to jointly tune all four PID loops subject to system-level requirements such as response time and minimum cross-coupling.

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.

7-103
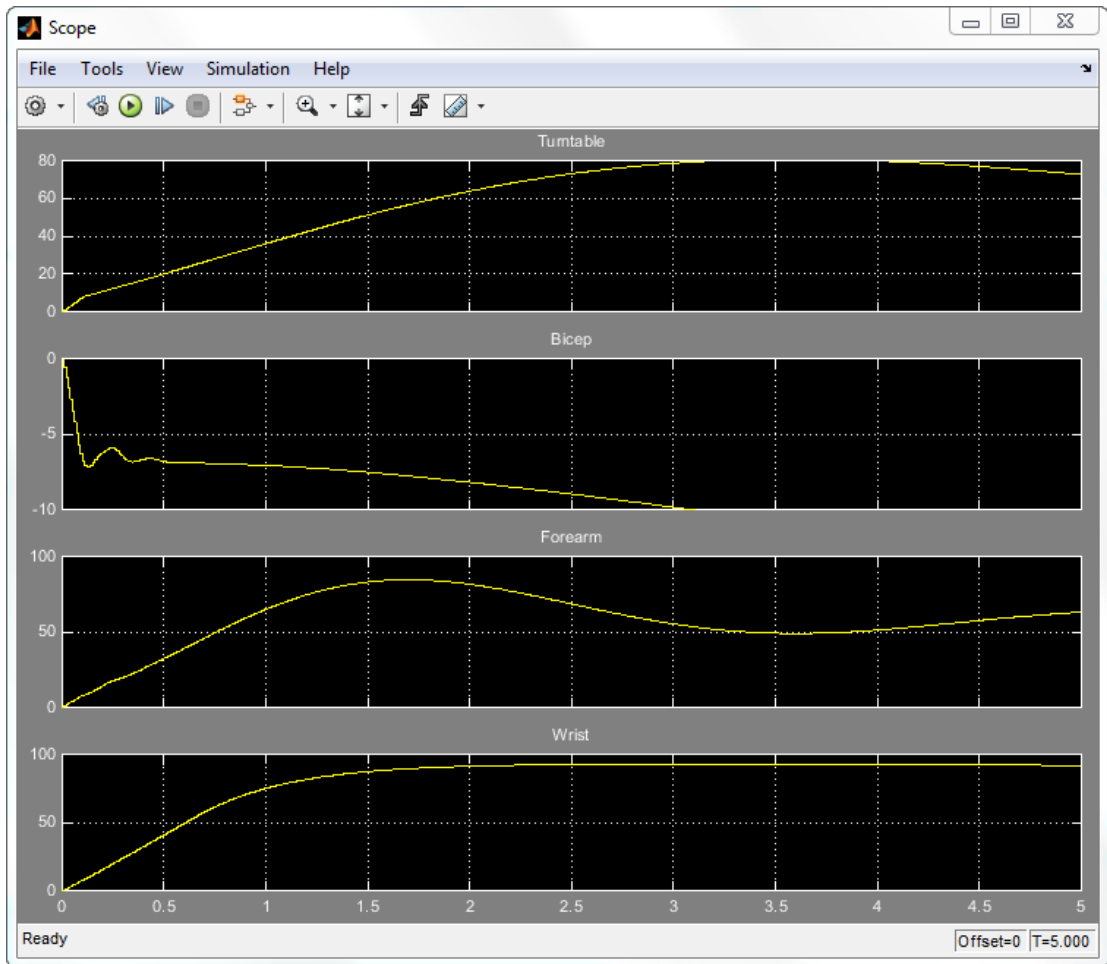
Figure 3: Untuned angular response.

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')
```
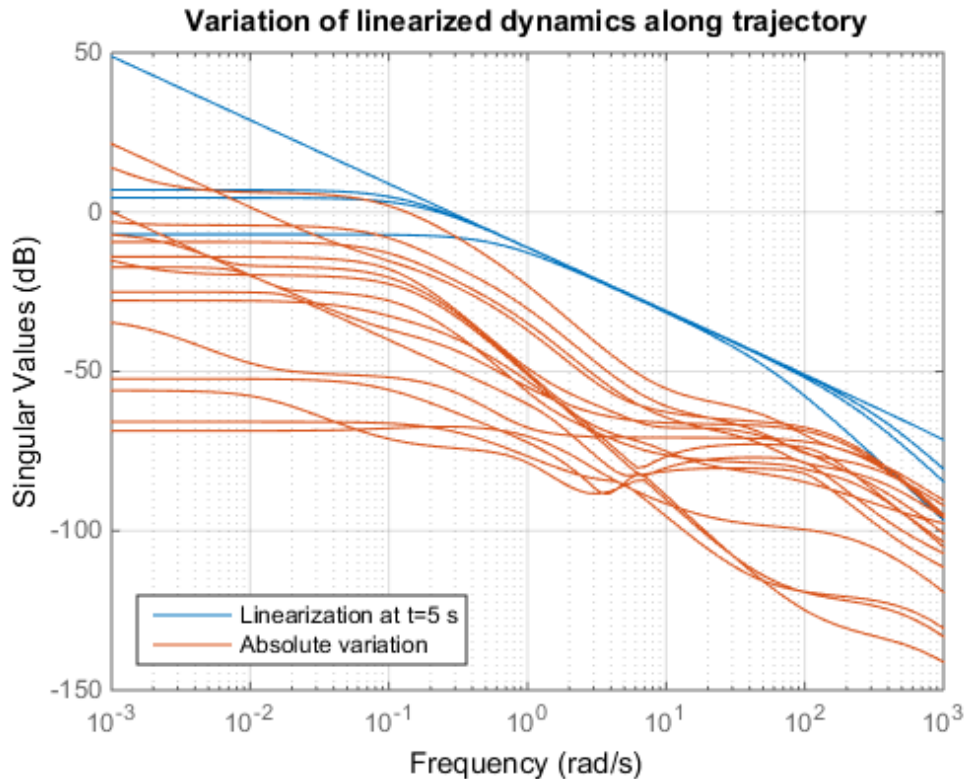
While the dynamics vary significantly at low and high frequency, the variation drops to less than 10% near 10 rad/s, which is roughly the desired control bandwidth. Small plant variations near the target gain crossover frequency suggest that we can control the arm with a single set of PID gains and need 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 `slTuner` interface to specify which blocks must be tuned and to locate the plant/controller boundary.

```
% Linearize the plant at t=3s
```

```
tLinearize = 3;

% Create slTuner interface
TunedBlocks = {'TurntablePID','BicepPID','ForearmPID','WristPID'};
ST0 = slTuner('rct_robotarm',TunedBlocks,tLinearize);

% Mark outputs of PID blocks as plant inputs
addPoint(ST0,TunedBlocks)

% Mark joint angles as plant outputs
addPoint(ST0,'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
Controls = TunedBlocks;       % actuator commands
Measurements = 'Robot Arm';  % joint angle measurements
ST1 = looptune(ST0,Controls,Measurements,wc);

Final: Peak gain = 0.999, Iterations = 59
Achieved target gain value TargetGain=1.
```

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')
```
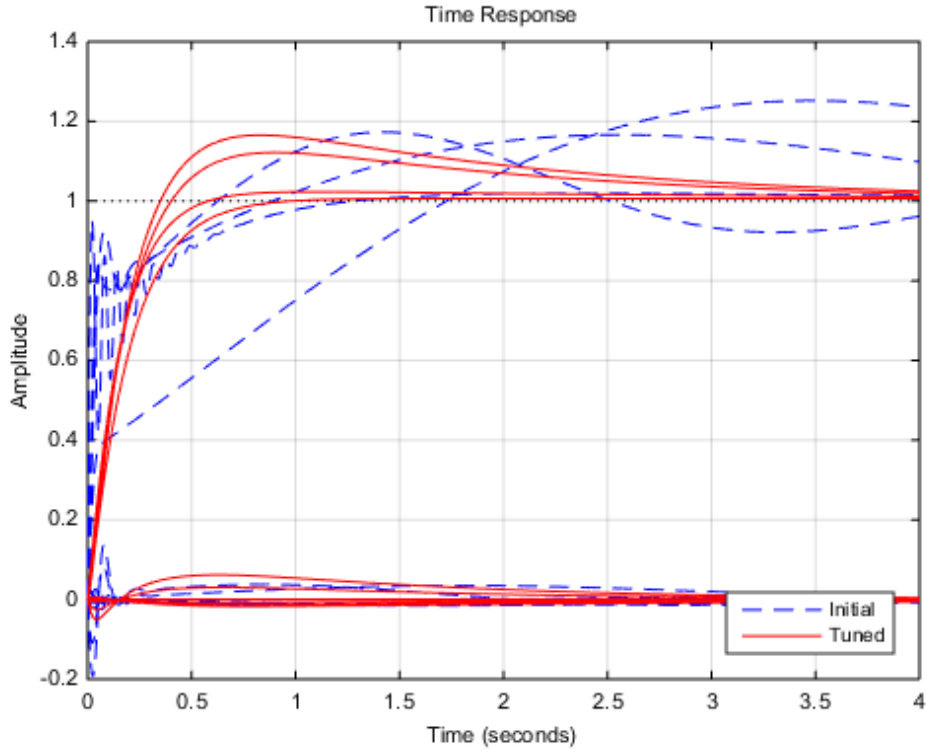
The four curves settling near y=1 represent the step responses of each joint, and the curves settling near y=0 represent the cross-coupling terms. The tuned controller is a clear improvement but should ideally settle faster with less overshoot.

Exploiting the Second Degree of Freedom

The 2-DOF PID controllers have a feedforward and a feedback component.
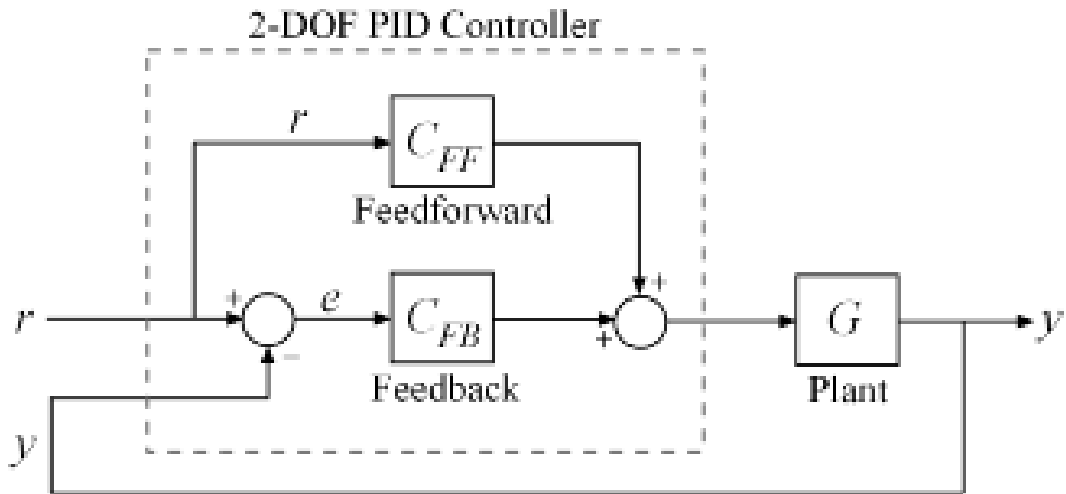
Figure 4: Two degree-of-freedom PID controllers.

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,Controls,Measurements,TR);
```

```
Final: Peak gain = 1.06, Iterations = 75
```

```
T2 = getIOTransfer(ST2,RefSignals,'Robot Arm');
stepplot(T1,'r--',T2,'g',4,opt)
legend('1-DOF tuning','2-DOF tuning','location','SouthEast')
```

7-109
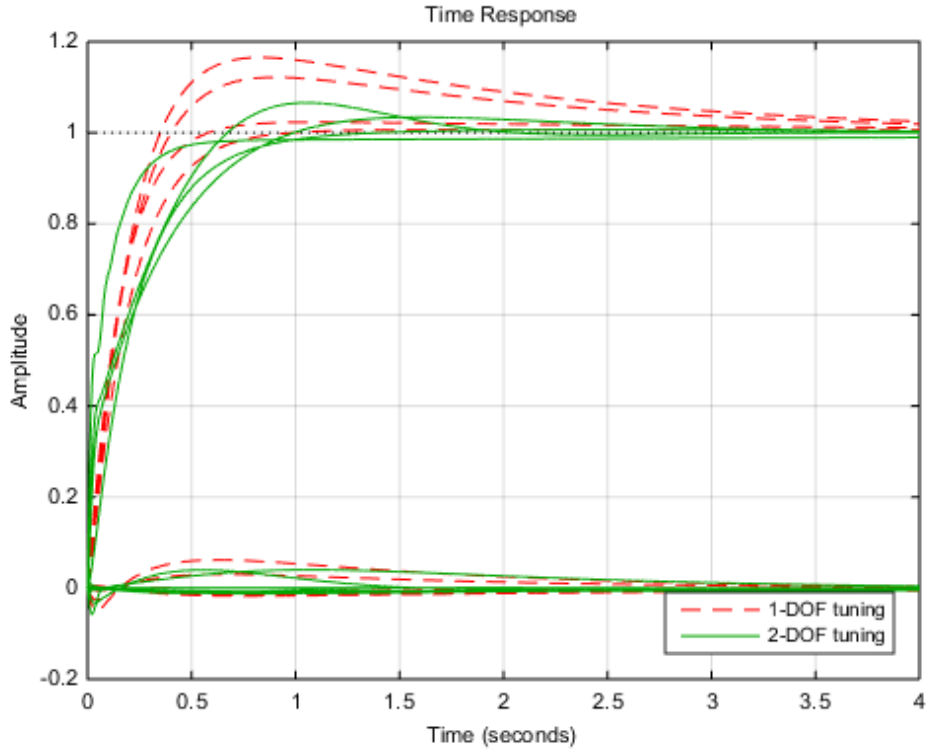
The 2-DOF tuning reduces overshoot and takes advantage of the b and c parameters as confirmed by inspecting the tuned PID gains:

```
showTunable(ST2)
```

```
Block 1: rct_robotarm/Controller/TurntablePID =

                      1                 s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                      s              Tf*s+1

  with Kp = 13.164, Ki = 11.199, Kd = 0.59535, Tf = 0.021063, b = 0.83826, c = 1.7721.

Continuous-time 2-DOF PID controller.
```

```
----------------------------------

Block 2: rct_robotarm/Controller/BicepPID =

                     1              s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                     s            Tf*s+1

  with Kp = 11.6365, Ki = 6.2564, Kd = 1.558, Tf = 0.44824, b = 0.70427, c = 1.8424.

Continuous-time 2-DOF PID controller.

----------------------------------

Block 3: rct_robotarm/Controller/ForearmPID =

                     1              s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                     s            Tf*s+1

  with Kp = 18.9585, Ki = 39.3474, Kd = 1.1049, Tf = 0.014576, b = 0.6217, c = 1.2741.

Continuous-time 2-DOF PID controller.

----------------------------------

Block 4: rct_robotarm/Controller/WristPID =

                     1              s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                     s            Tf*s+1

  with Kp = 40.9185, Ki = 5.3925, Kd = 1.4567, Tf = 0.010433, b = 0.97787, c = 0.75324.

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)
```

7-111
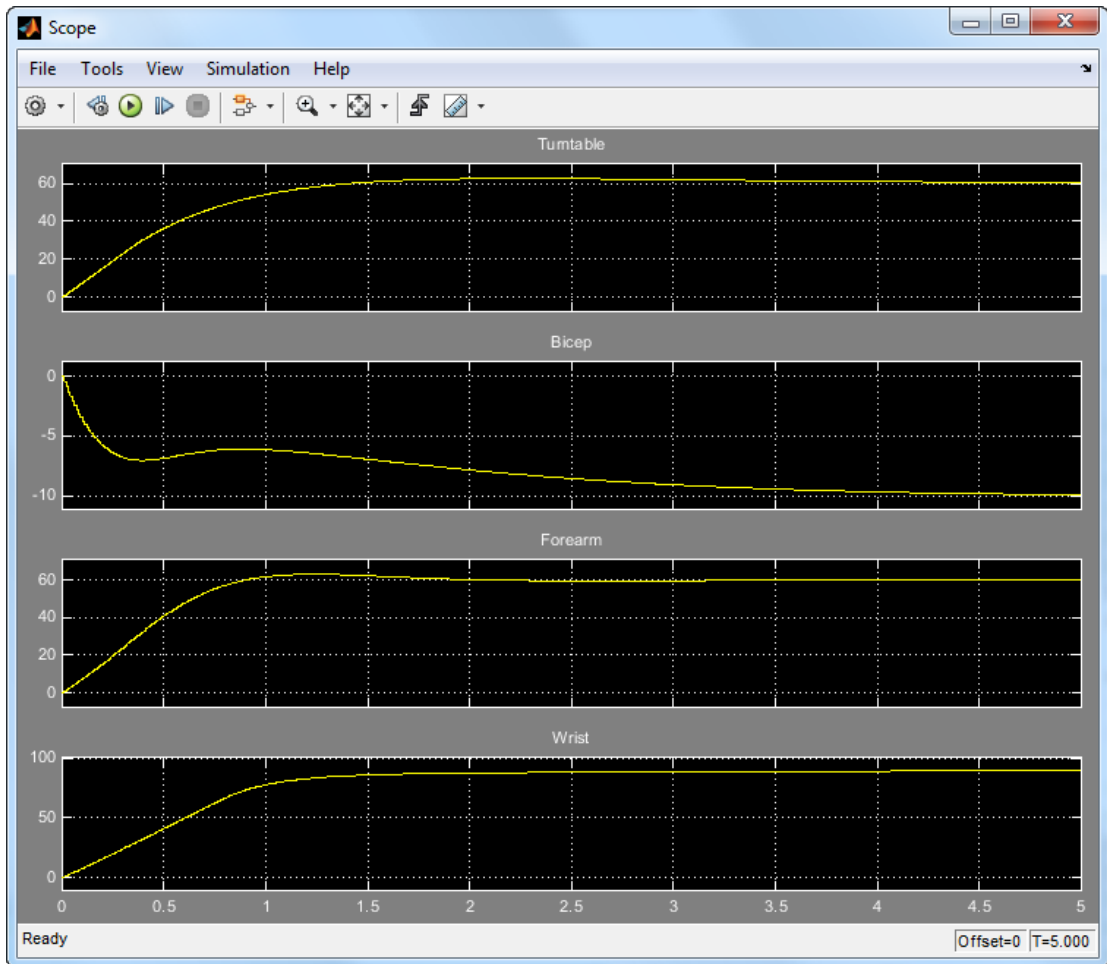
Figure 5: Tuned angular response.

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 is in fact due to 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'};
H2.y = {'Bicep','Forearm'};
step(H2,5), grid
```
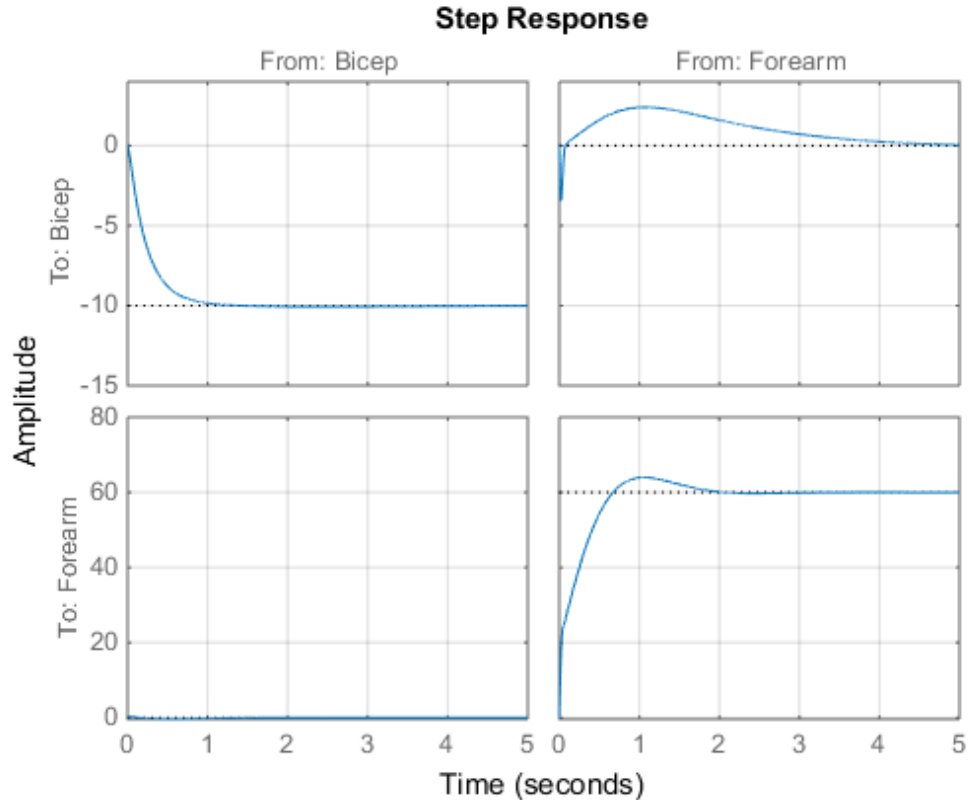

When brought to scale, the first row of plots show that a 60-degree step change in Forearm position has a sizeable and lasting impact on the Bicep position. This explains the sluggish Bicep response observed when simultaneously moving all four joints.

Refining The Design

To improve the Bicep response for this specific arm maneuver, we must keep the cross-couplings effects small *relative to* the final angular displacements in each joint. To do this, scale the cross-coupling terms in the tracking requirement by the reference angle amplitudes.

```
JointDisp = [60 10 60 90];  % commanded angular displacements, in degrees
TR.InputScaling = JointDisp;
```

To prevent jerky transients and avoid overloading the motors, limit the control bandwidth by imposing -20 dB/decade roll-off past 20 rad/s.

```
s = tf('s');
RO = TuningGoal.MaxGain(RefSignals,'Robot Arm',20/s);
```

Finally, explicitly limit the overshoot to 5% and increase the desired phase margin from its default value of 45 degrees to 60 degrees.

```
OS = TuningGoal.Overshoot(RefSignals,'Robot Arm',5);
Options = looptuneOptions('PhaseMargin',60);
```

Retune the controller with the additional requirements in force

```
ST3 = looptune(STO,Controls,Measurements,TR,RO,OS,Options);
```

```
Final: Peak gain = 1.06, Iterations = 88
```

Compare the scaled responses with the previous design. Notice the significant reduction of the coupling between Forearm/Wrist and Bicep motion, both in peak value and total energy.

```
T2s = diag(1./JointDisp) * T2 * diag(JointDisp);
T3s = diag(1./JointDisp) * getIOTransfer(ST3,RefSignals,'Robot Arm') * diag(JointDisp);
stepplot(T2s,'g--',T3s,'m',4,opt)
legend('Initial 2-DOF','Refined 2-DOF','location','SouthEast')
```
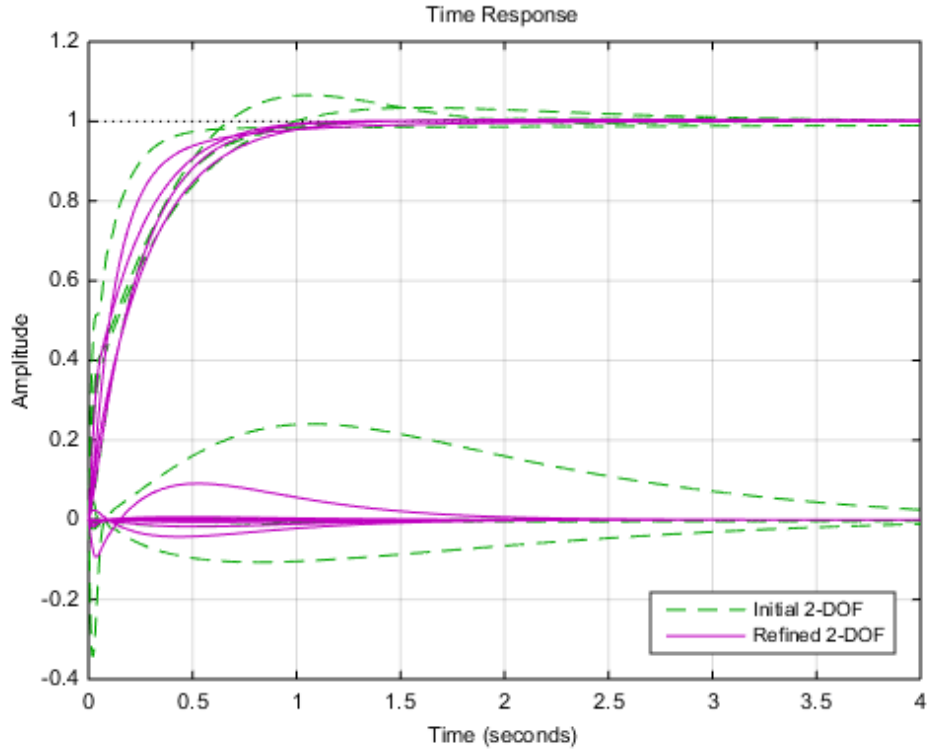
Push the retuned values to Simulink for further validation.

```
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 smooth transient.
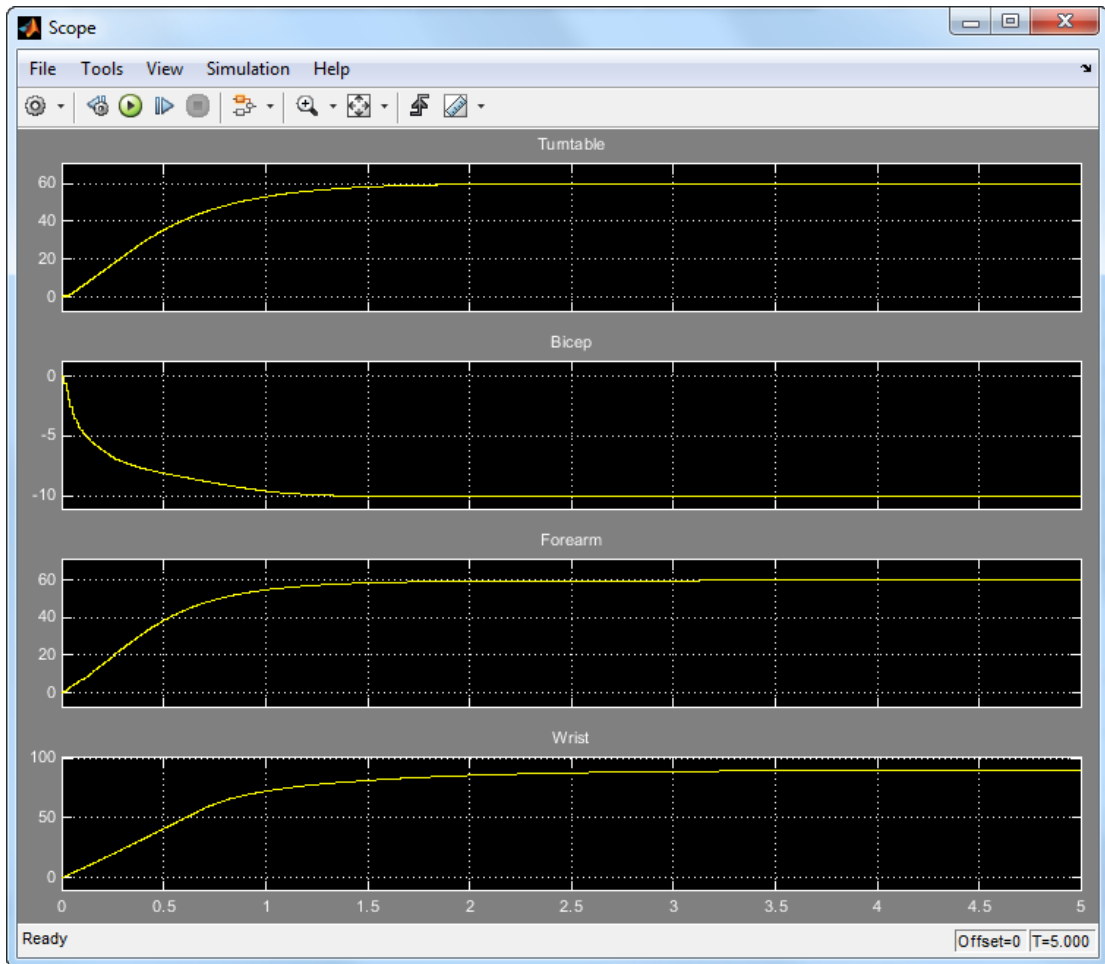
Figure 6: Angular response with refined controller.

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.

Measurements

Control
Actuator

x_m , x_{a_m}

A

x_{a_3}

A

x_{a_2}

A

x_{a_1}

x_g
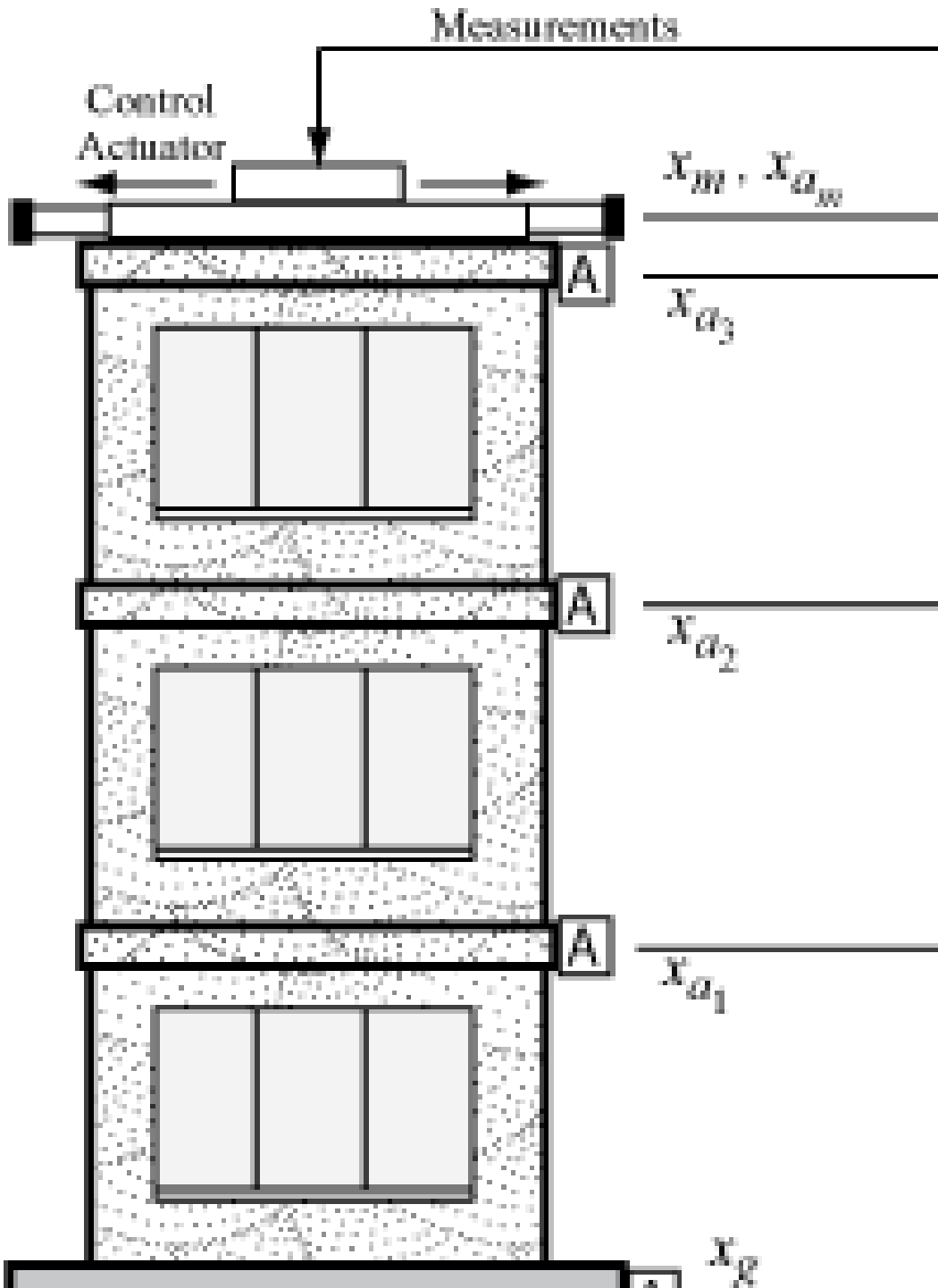
Figure 1: Active Mass Driver Control System

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)
- xm: displacement of AMD relative to 3rd floor (cm)
- xv(i): velocity of i-th floor relative to the ground (cm/s)
- xvm: velocity of AMD relative to the ground (cm/s)
- xa(i): acceleration of i-th floor relative to the ground (g)
- xam: 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 xag (in g) and the control signal u. We use 1 g = 981 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')
```

Kanai-Tajimi filter

From: n To: Out(1)
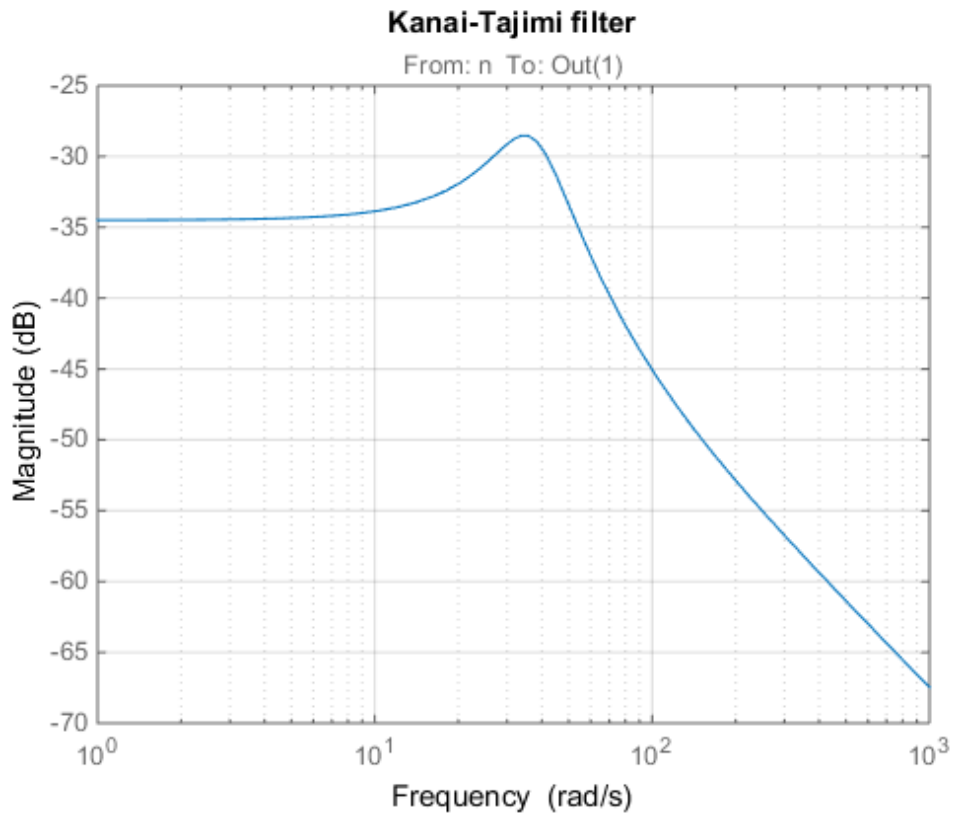
Open-Loop Characteristics

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);
dO = sqrt(diag(CV));

% Standard deviations of open-loop acceleration
```

```
CV = covar(PF('xa','n'),1);
xaO = sqrt(diag(CV));

% Plot open-loop RMS values
clf, bar([dO ; xaO])
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)'})
```

Drifts and accelerations for uncontrolled structure
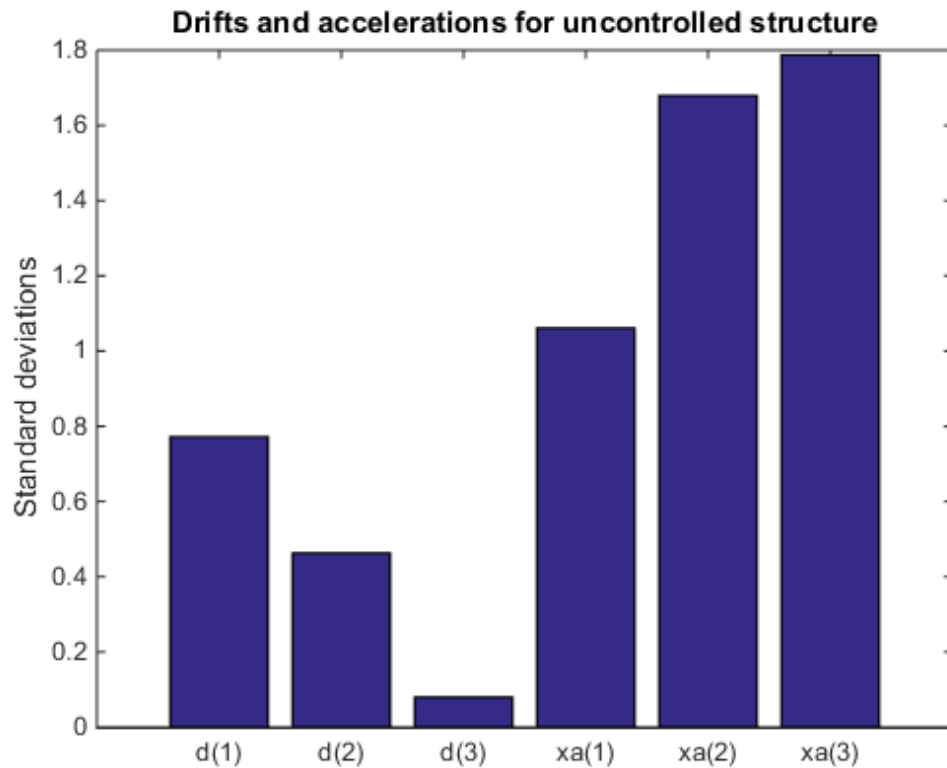
Control Structure and Design Requirements

The control structure is depicted in Figure 2.
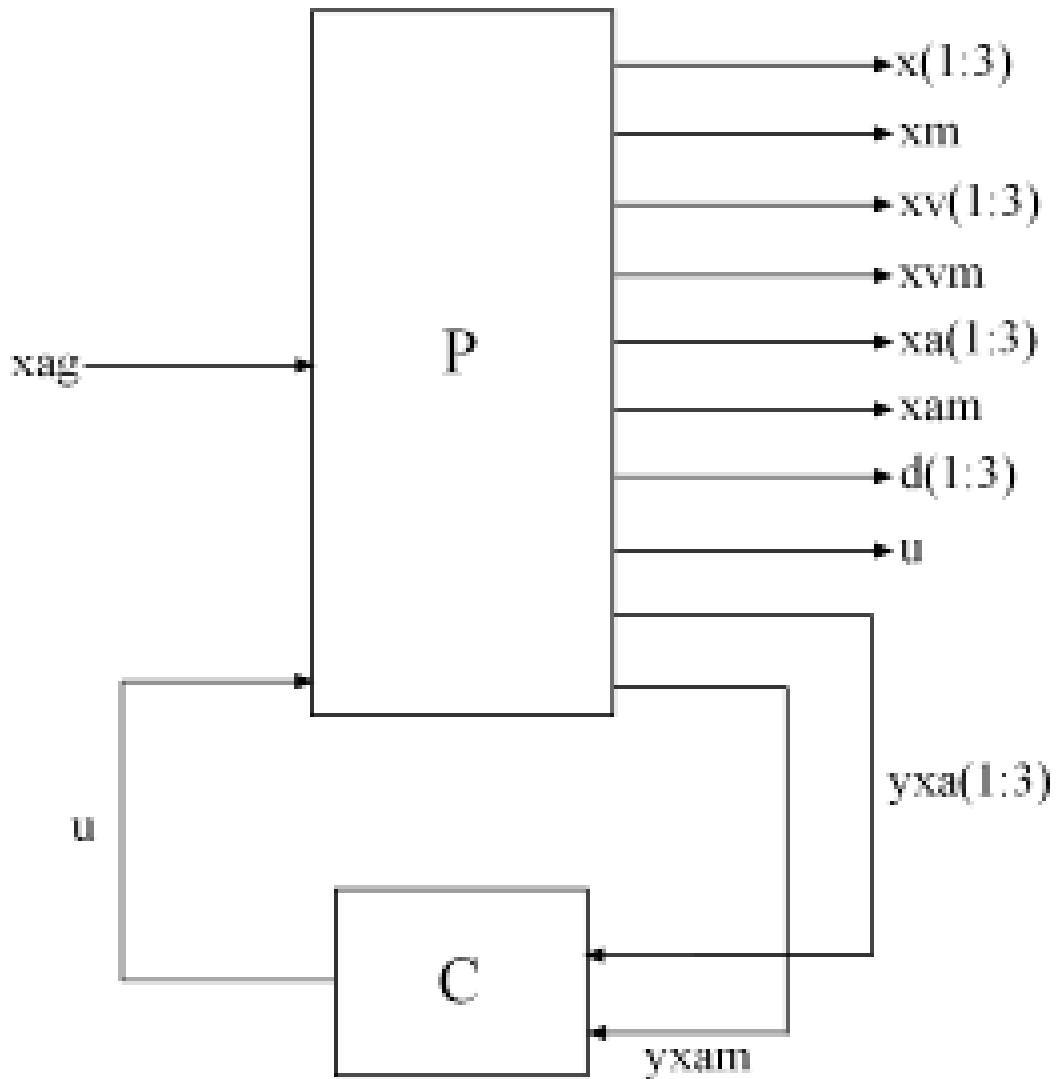
Figure 2: Control Structure

The controller C uses measurements `yxa` and `yxam` of `xa` and `xam` 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 `xa(i)`

- Hard constraints on control effort in terms of mass displacement `xm`, mass acceleration `xam`, 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

`systune` 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 `systune`.

```
% Build tunable closed-loop model
T0 = lft(PF,C);

% Tune controller parameters
[T,fSoft,gHard]  = systune(T0,Soft,Hard);

Final: Soft = 0.601, Hard = 0.99589, Iterations = 199
```

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([dO d; xaO 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')
```

Drifts and accelerations
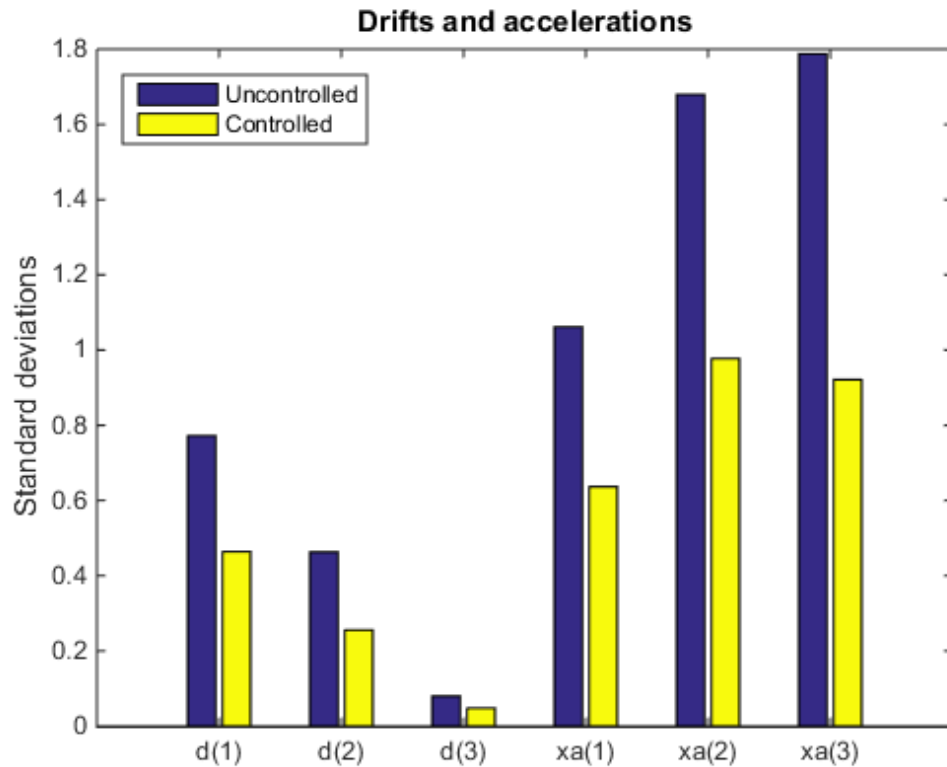
Simulate the response of the 3-story structure to an earthquake-like excitation in both open and closed loop. The earthquake acceleration is modeled as a white noise process colored by the Kanai-Tajimi filter.

```
% 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');
```
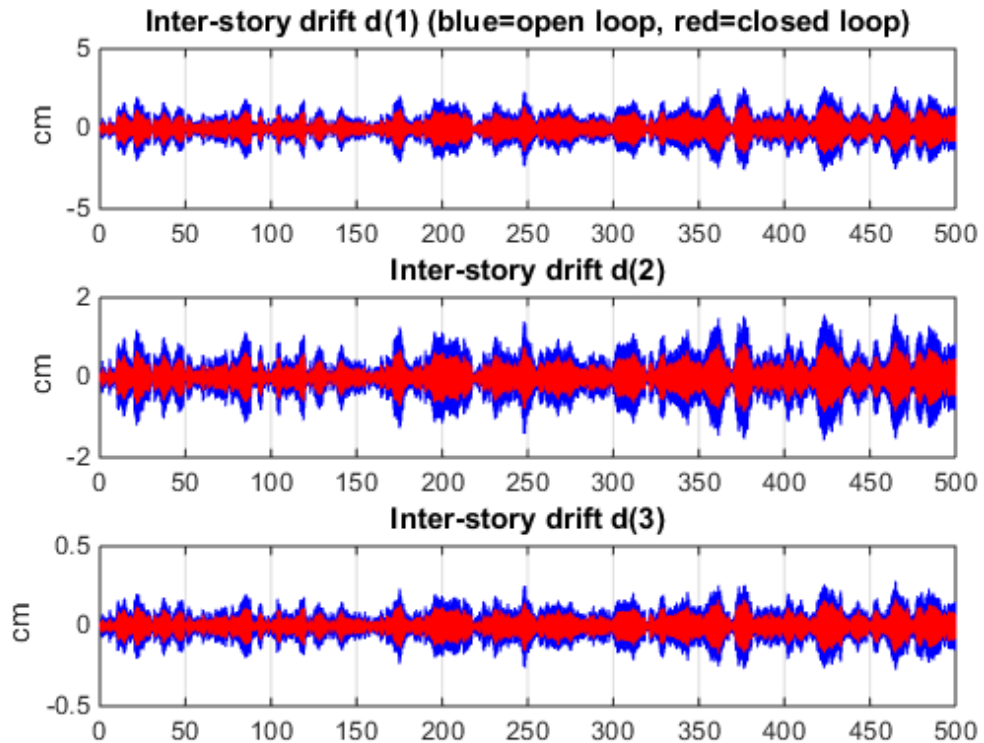


```
% Accelerations
clf, subplot(311); plot(t,ysimOL(:,9),'b',t,ysimCL(:,9),'r'); grid;
title('Acceleration of 1st floor xa(1) (blue=open loop, red=closed loop)');  ylabel('g
```

```
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');
```
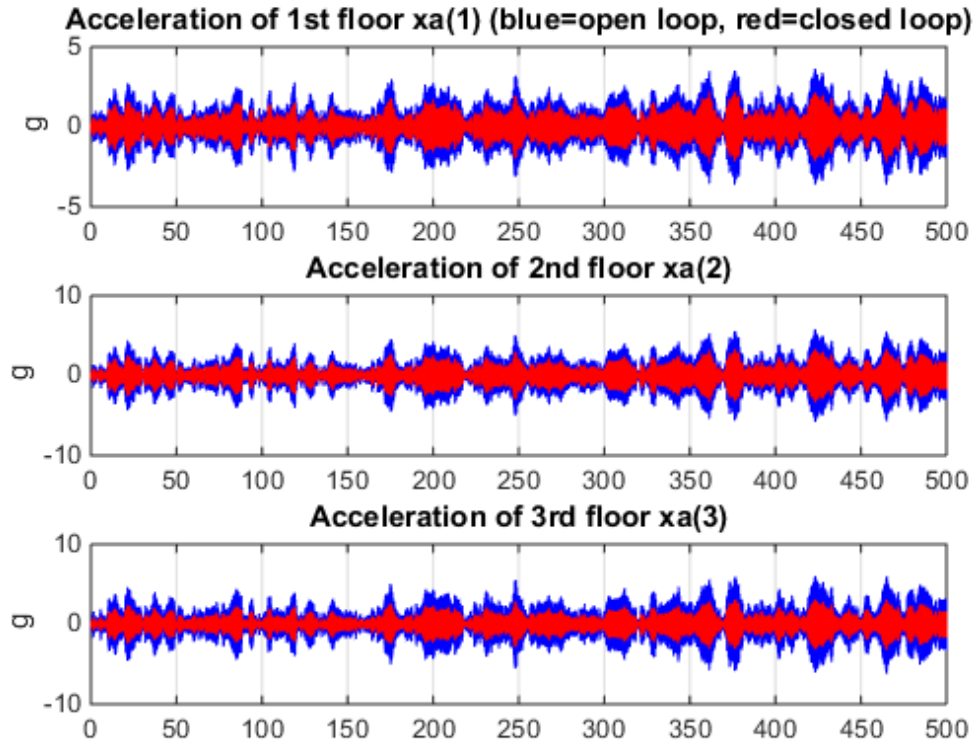


```
% Control variables
clf, subplot(311); plot(t,ysimCL(:,4),'r'); grid;
title('AMD position xm');  ylabel('cm');
subplot(312); plot(t,ysimCL(:,12),'r'); grid;
title('AMD acceleration xam');  ylabel('g');
subplot(313); plot(t,ysimCL(:,16),'r'); grid;
title('Control signal u');
```
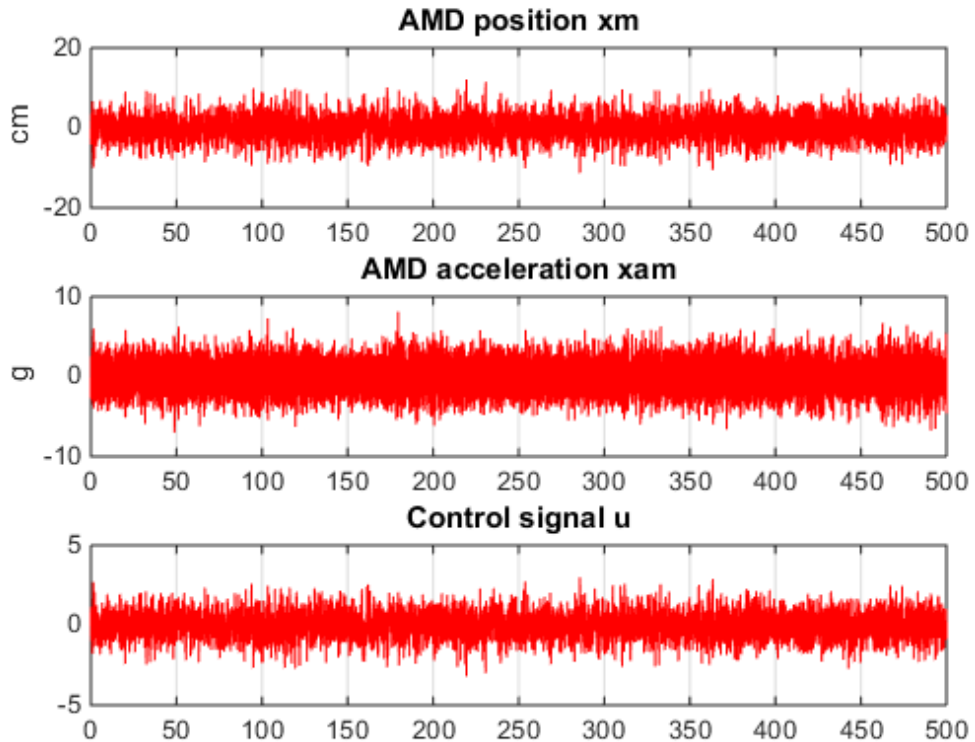
Plot the root-mean-square (RMS) of the simulated signals for both the controlled and uncontrolled scenarios. Assuming ergodicity, the RMS performance can be estimated from a single sufficiently long simulation of the process and coincides with the standard deviations computed earlier. Indeed the RMS plot closely matches the standard deviation plot obtained earlier.

```
clf, bar([std(ysimOL(:,13:15)) std(ysimOL(:,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')
```
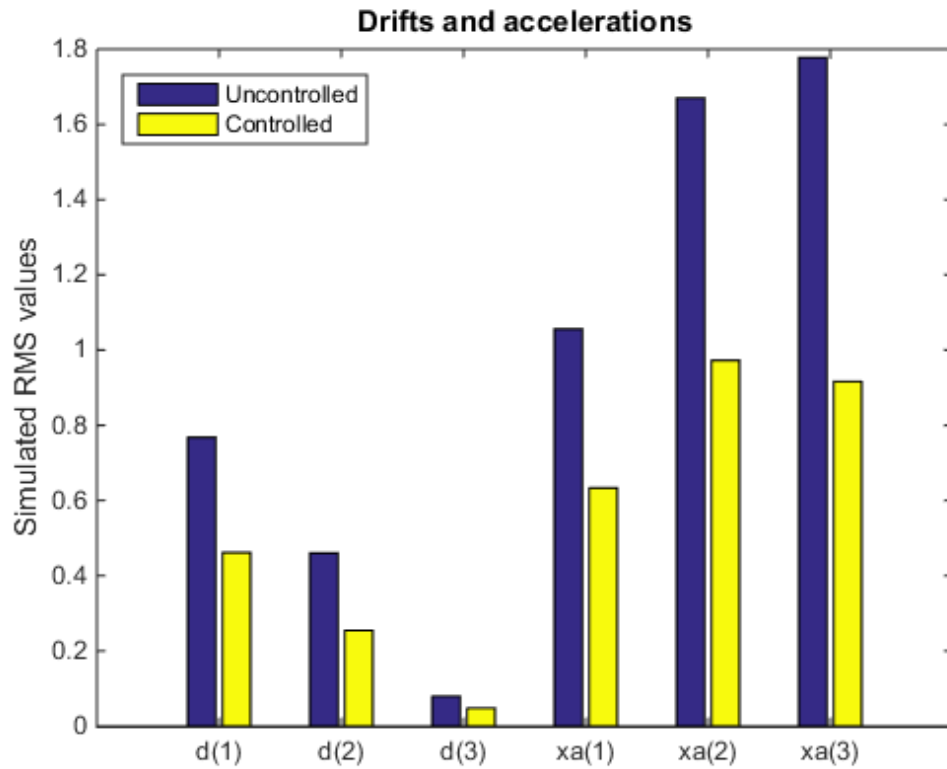
Overall, the controller achieves significant reduction of ground vibration both in terms of drift and acceleration for all stories while meeting the hard constraints on control effort and mass displacement.

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.
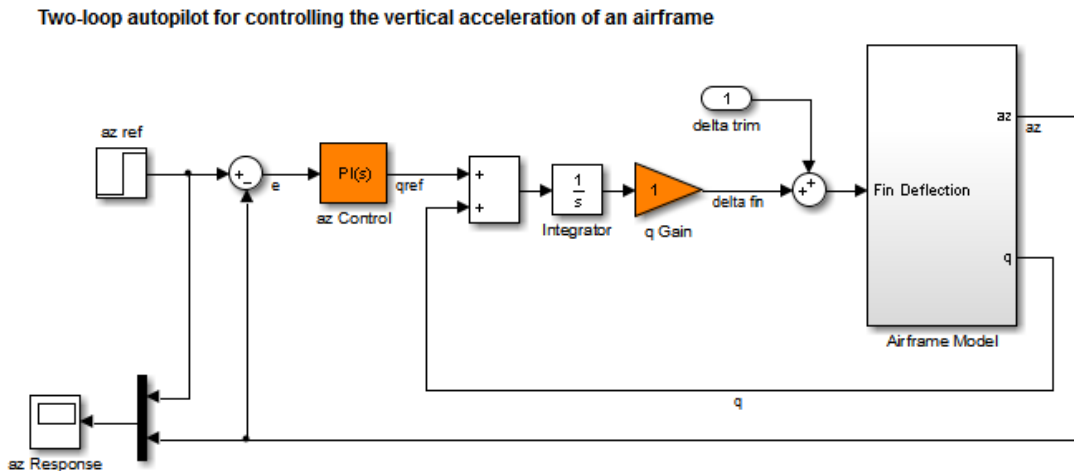
```
open_system('rct_airframe1')
```

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 `azref`. 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 `azref` 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 $V = 984 m/s$. 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/ Equations of Mot:
      x:             0      dx:           984
      x:     -3.05e+03      dx:             0
(2.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:             0      dx:      -0.00972
(3.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:           984      dx:          22.7
      x:             0      dx:     -1.44e-11 (0)
(4.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      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
```
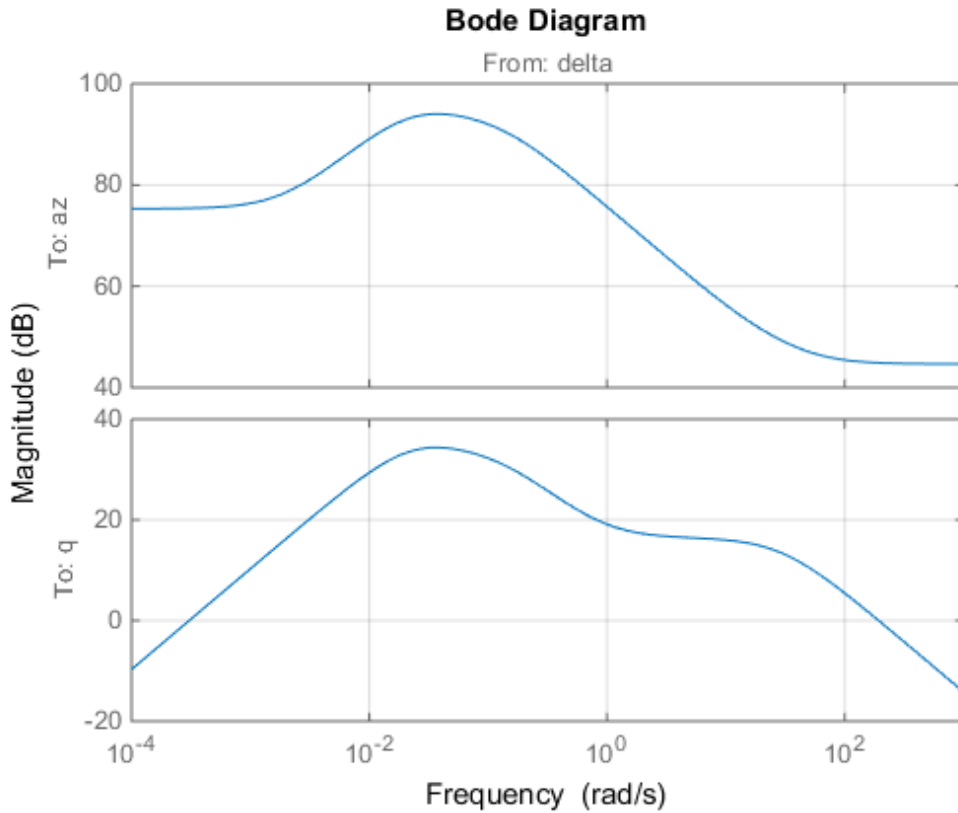
Note that the airframe model has an unstable pole:

```
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 `slTuner` 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.

```
ST0 = slTuner('rct_airframe1',{'az Control','q Gain'},op);
```

Mark the reference, control, and measurement signals as points of interest for analysis and tuning.

```
addPoint(ST0,{'az ref','delta fin','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 `wc` = 5 rad/s for the open-loop response at the plant input "delta fin".

```
wc = 5;
Controls = 'delta fin';
Measurements = {'az','q'};
[ST,gam,Info] = looptune(ST0,Controls,Measurements,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.

```
figure('Position',[100,100,560,714])
loopview(ST,Info)
```

Open loop CG: Minimum and maximum loop gains (CrossTol = 0.1)

Margins at plant inputs: Stability margins

Margins at plant outputs: Stability margins
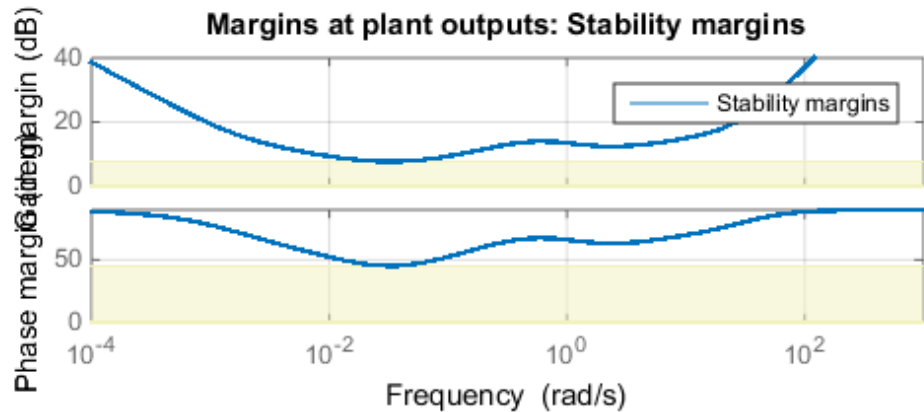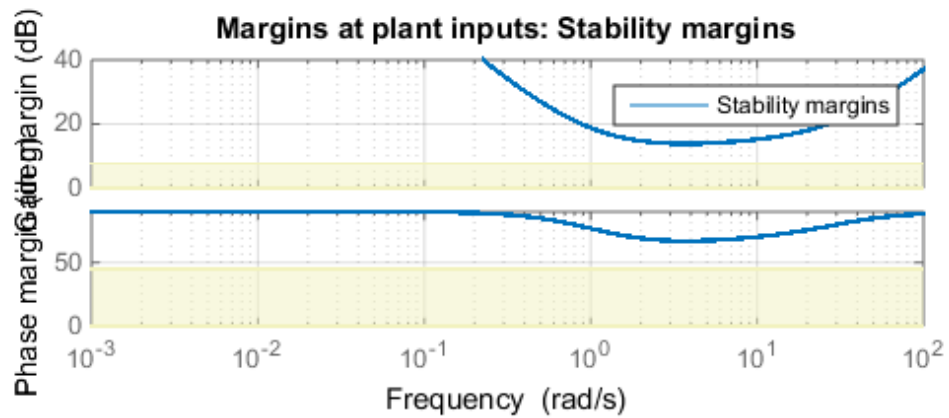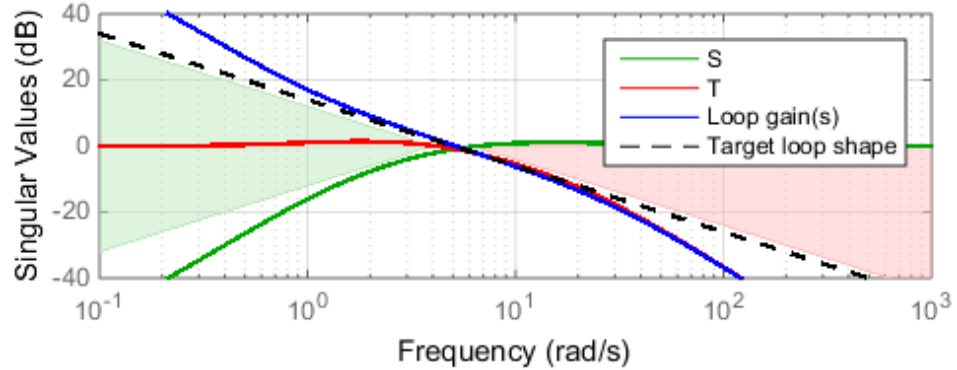
The first plot confirms that the open-loop response has integral action and the desired gain crossover frequency while the second plot shows that the MIMO stability margins are satisfactory (the blue curve should remain below the yellow bound). Next check the response from the step command `azref` to the vertical acceleration `az`:

```
T = getIOTransfer(ST,'az ref','az');
figure, step(T,5)
```

Step Response

From: az ref To: az
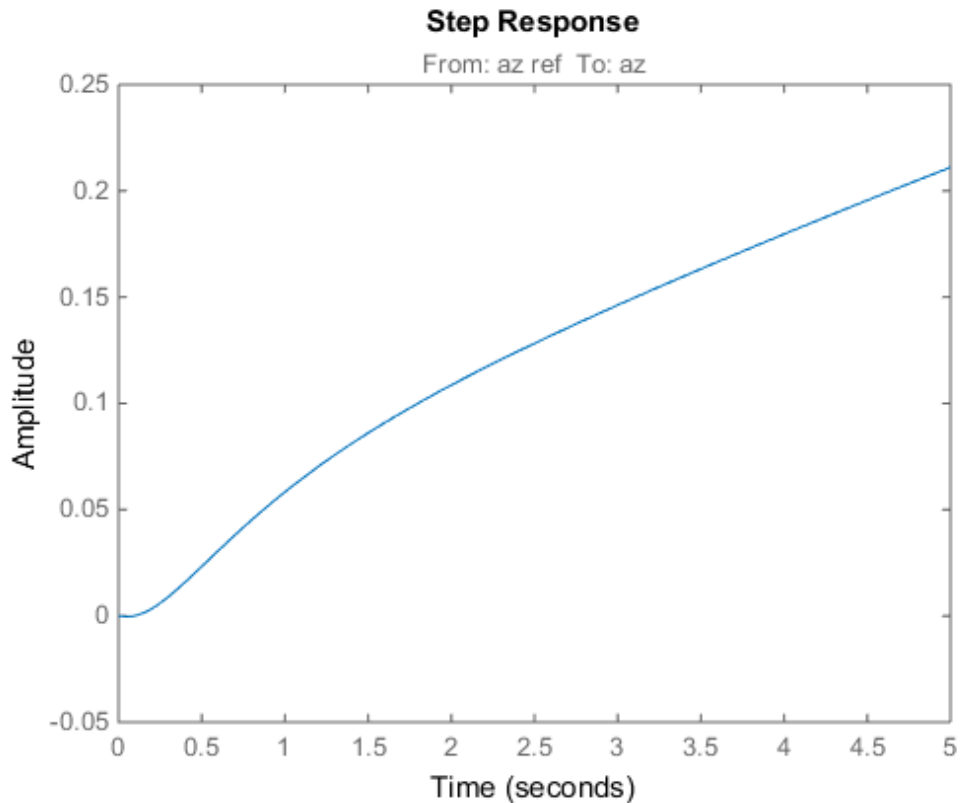
The acceleration `az` does not track `azref` despite the presence of an integrator in the loop. This is because the feedback loop acts on the two variables `az` and `q` and we have not specified which one should track `azref`.

Adding a Tracking Requirement

To remedy this issue, add an explicit requirement that `az` should follow the step command `azref` 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 = looptune(ST0,Controls,Measurements,[3,12],TrackReq);
```

```
Final: Peak gain = 1.23, Iterations = 49
```

The step response from `azref` to `az` is now satisfactory:

```
Tr1 = getIOTransfer(ST,'az ref','az');
step(Tr1,5), grid
```


Step Response

7-137
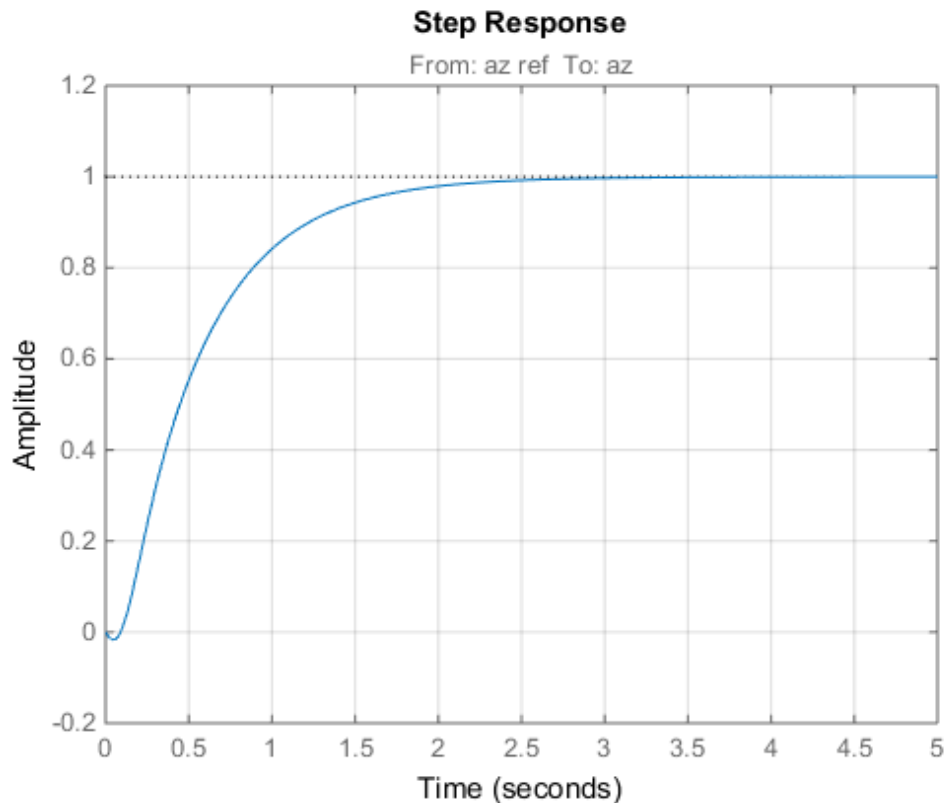
Also check the disturbance rejection characteristics by looking at the responses from a disturbance entering at the plant input

```
Td1 = getIOTransfer(ST,'delta fin','az');
bodemag(Td1), grid
```
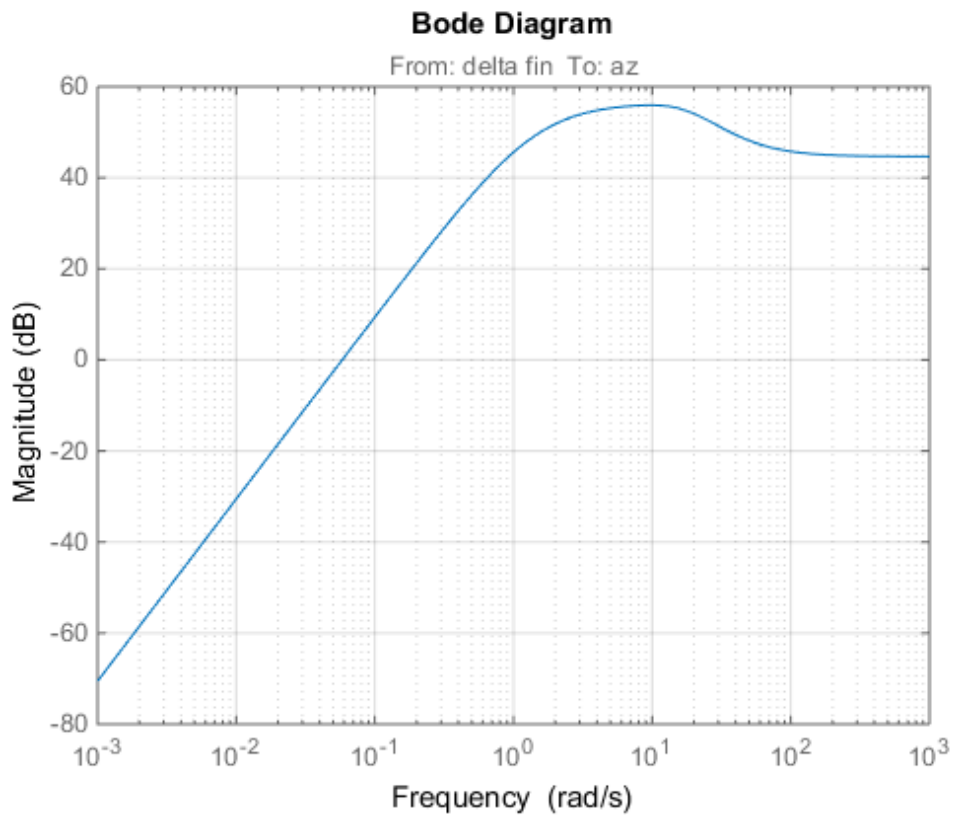


```
step(Td1,5), grid, title('Disturbance rejection')
```
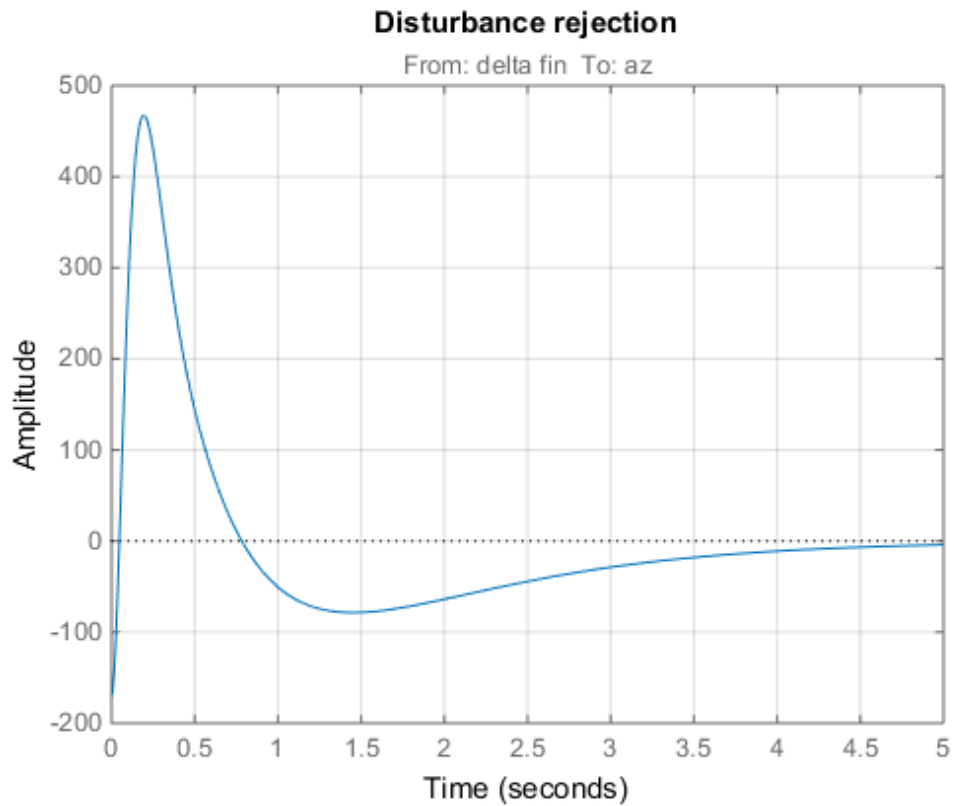
Use `showBlockValue` to see the tuned values of the PI controller and inner-loop gain

```
showBlockValue(ST)
```

```
AnalysisPoints_ =

  d =
       u1  u2  u3  u4
   y1   1   0   0   0
   y2   0   1   0   0
   y3   0   0   1   0
   y4   0   0   0   1

Name: AnalysisPoints_
Static gain.
```

```
---------------------------------
az_Control =

              1
  Kp + Ki * ---
              s

  with Kp = 0.00166, Ki = 0.0017

Name: az_Control
Continuous-time PI controller in parallel form.
---------------------------------
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.

```
writeBlockValue(ST)
```

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 `systune` or `looptune`. For variety, we now use `systune` to tune the following MIMO architecture.

```
open_system('rct_airframe2')
```

Two-loop autopilot for controlling the vertical acceleration of an airframe
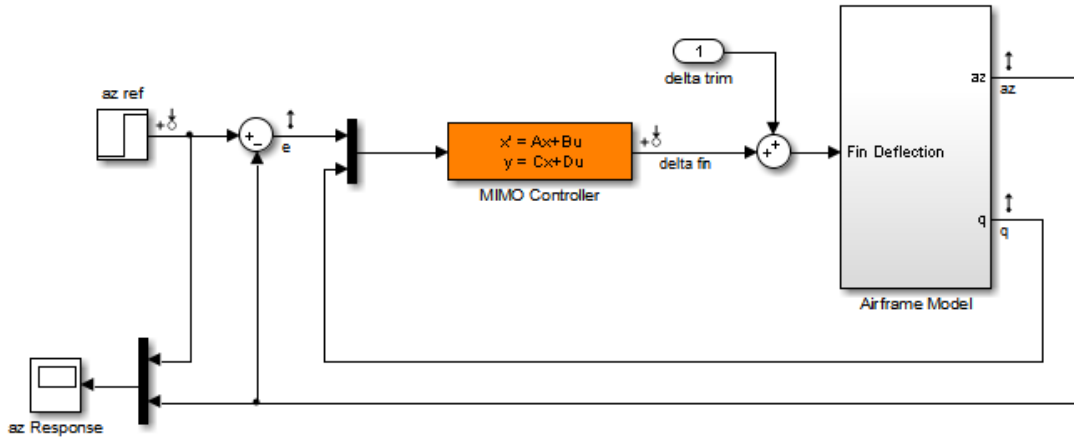
As before, compute the trim condition for $\alpha = 0$ and $V = 984 m/s$.

```
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/ Equations of Mot:
      x:              0     dx:               984
      x:      -3.05e+03     dx:                 0
(2.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:              0     dx:          -0.00972
(3.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:            984     dx:              22.7
      x:              0     dx:       2.46e-11 (0)
(4.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      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 `slTuner` interface to configure the Simulink model for tuning. Note that the signals of interest are already marked as Linear Analysis points in the Simulink model.

```
ST0 = slTuner('rct_airframe2','MIMO Controller',op);
```

Try a second-order MIMO controller with zero feedthrough from e to delta fin. To do this, create the desired controller parameterization and associate it with the "MIMO Controller" block using `setBlockParam`:

```
CO = ltiblock.ss('C',2,1,2);                  % Second-order controller
CO.d.Value(1) = 0;   CO.d.Free(1) = false;  % Fix D(1) to zero
setBlockParam(ST0,'MIMO Controller',CO)
```

Next create the tuning requirements. Here we use the following four requirements:

1 **Tracking**: `az` should respond in about 1 second to the `azref` command

2 **Bandwidth and roll-off**: The loop gain at `delta fin` should roll off after 25 rad/s with a -20 dB/decade slope

3 **Stability margins**: The margins at `delta fin` 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 points
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 `systune` to tune the controller parameters subject to these requirements.

```
AllReqs = [Req1,Req2,Req3 Req4];
Opt = systuneOptions('RandomStart',3);

rng(0)
[ST,fSoft,~,Info] = systune(ST0,AllReqs,Opt);

Final: Soft = 1.14, Hard = -Inf, Iterations = 121
Final: Soft = 1.14, Hard = -Inf, Iterations = 75
Final: Soft = 1.14, Hard = -Inf, Iterations = 83
Final: Soft = 1.14, Hard = -Inf, Iterations = 113
```

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.

```
figure('Position',[100,100,987,474])
viewSpec(AllReqs,ST,Info)
```
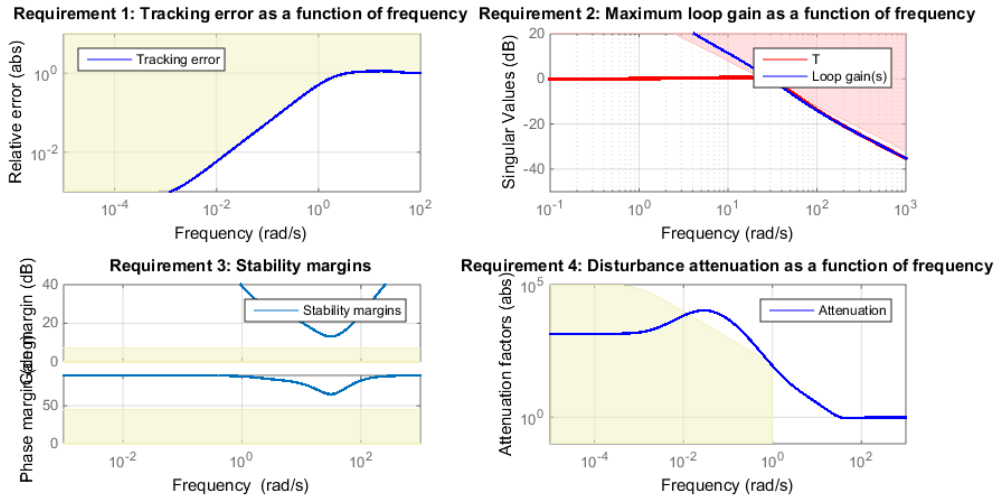
Compute the closed-loop responses and compare with the two-loop design.

```
T = getIOTransfer(ST,{'az ref','delta fin'},'az');

figure
step(Tr1,'b',T(1),'r',5)
title('Tracking'), legend('Cascade','2 dof')
```

Tracking

From: az ref To: az
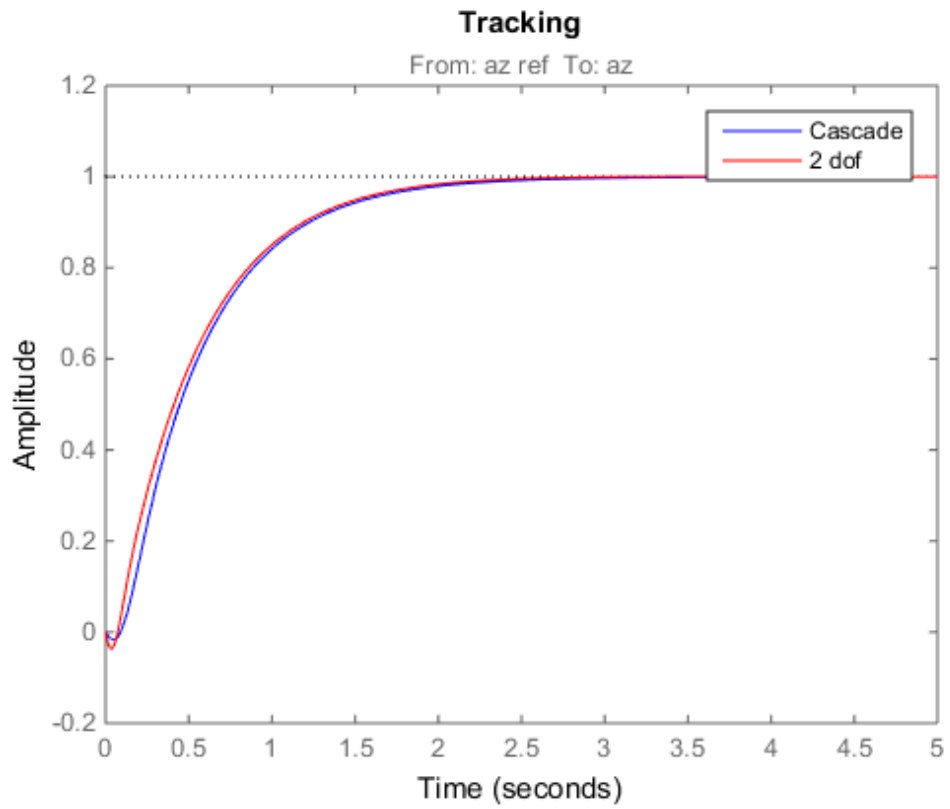

```
step(Td1,'b',T(2),'r',5)
title('Disturbance rejection'), legend('Cascade','2 dof')
```
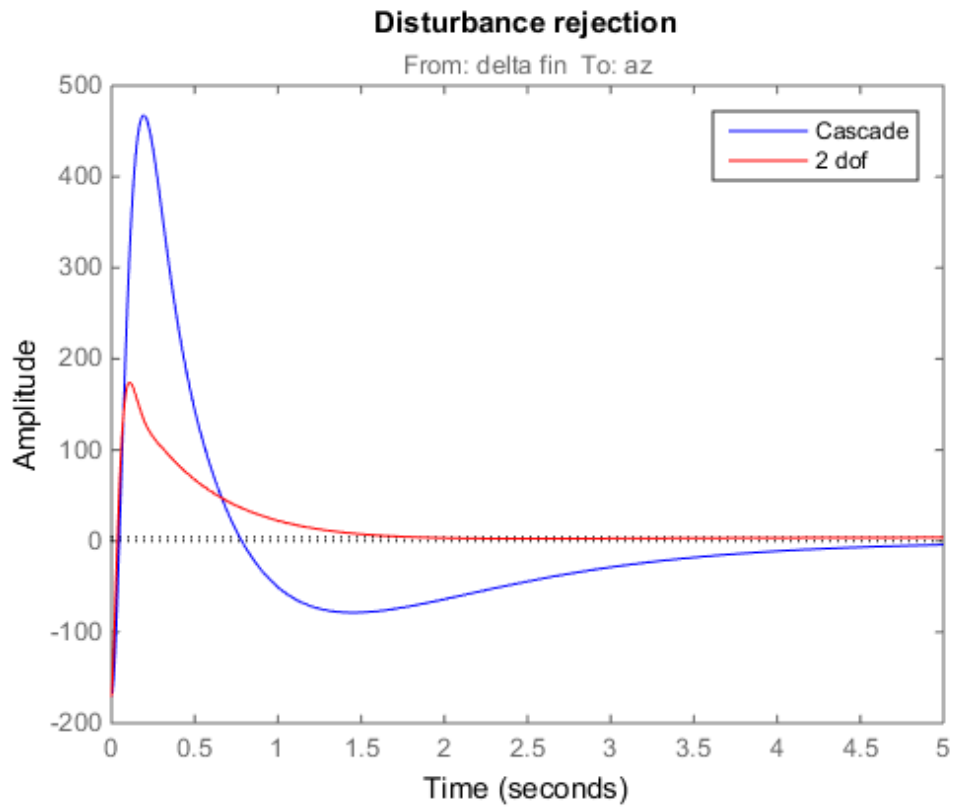
The tracking performance is similar but the second design has better disturbance rejection properties.

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,theta,v,p,phi,r] consists of

- Longitudinal velocity u (m/s)
- Lateral velocity v (m/s)
- Normal velocity w (m/s)
- Pitch angle theta (deg)
- Roll angle phi (deg)
- Roll rate p (deg/s)
- Pitch rate q (deg/s)
- Yaw rate r (deg/s).

The controller generates commands ds,dc,dT in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of theta, 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, rise 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 `slTuner` interface to quickly set up the tuning task. Create an instance of this interface with the list of blocks to be tuned.

```
ST0 = slTuner('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:
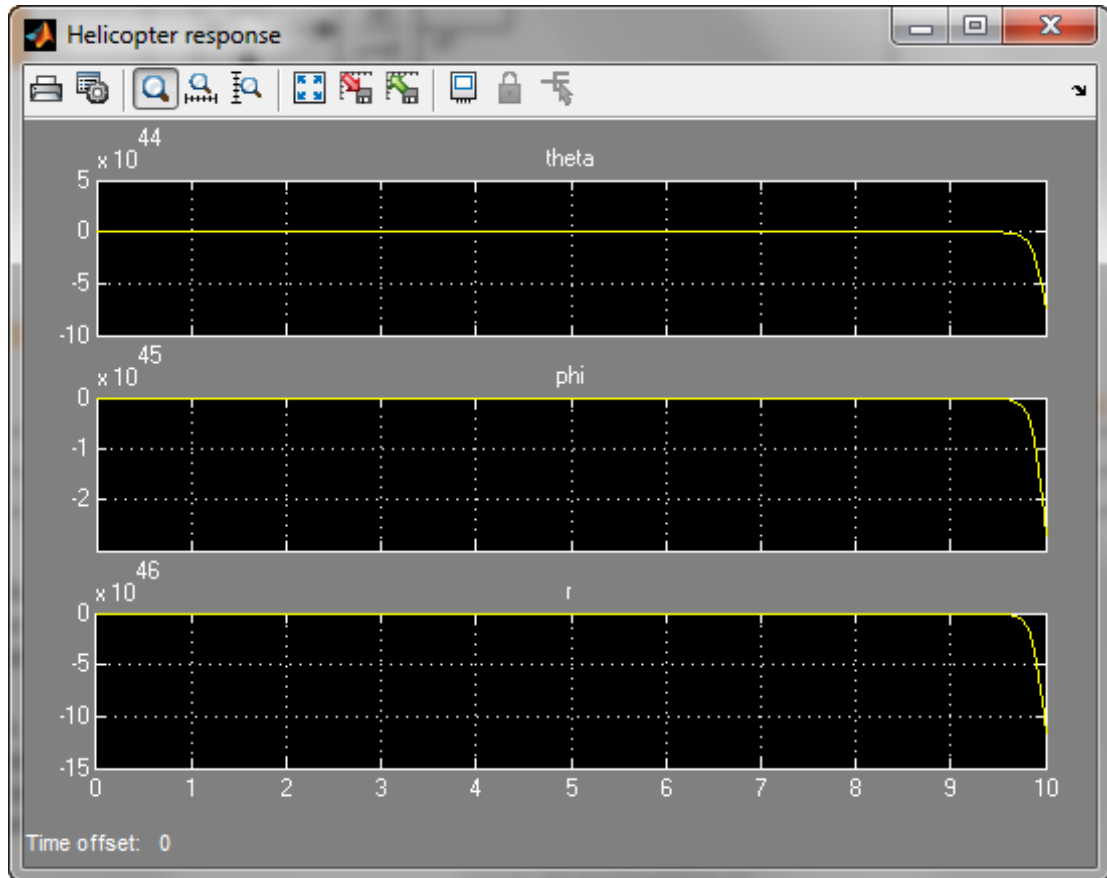
Mark the I/O signals of interest for setpoint tracking, and identify the plant inputs and outputs (control and measurement signals) where the stability margin are measured.

```
addPoint(STO,{'theta-ref','phi-ref','r-ref'})    % setpoint commands
addPoint(STO,{'theta','phi','r'})                 % corresponding outputs
addPoint(STO,{'u','y'});
```

7-149

Finally, capture the design requirements using `TuningGoal` objects. We use the following requirements for this example:

- **Tracking requirement**: The response of `theta`, `phi`, `r` to step commands `theta_ref`, `phi_ref`, `r_ref` must resemble a decoupled first-order response with a one-second time constant

- **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 25 to prevent fast dynamics and jerky transients

```
% Less than 20% mismatch with reference model 1/(s+1)
TrackReq = TuningGoal.StepTracking({'theta-ref','phi-ref','r-ref'},{'theta','phi','r'},
TrackReq.RelGap = 0.2;

% 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
MaxFrequency = 25;
PoleReq = TuningGoal.Poles(0,0,MaxFrequency);
```

You can now use `systune` 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] = systune(ST0,AllReqs);

Final: Soft = 1.13, Hard = -Inf, Iterations = 68
```

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 = getIOTransfer(ST1,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
step(T1,5)
```
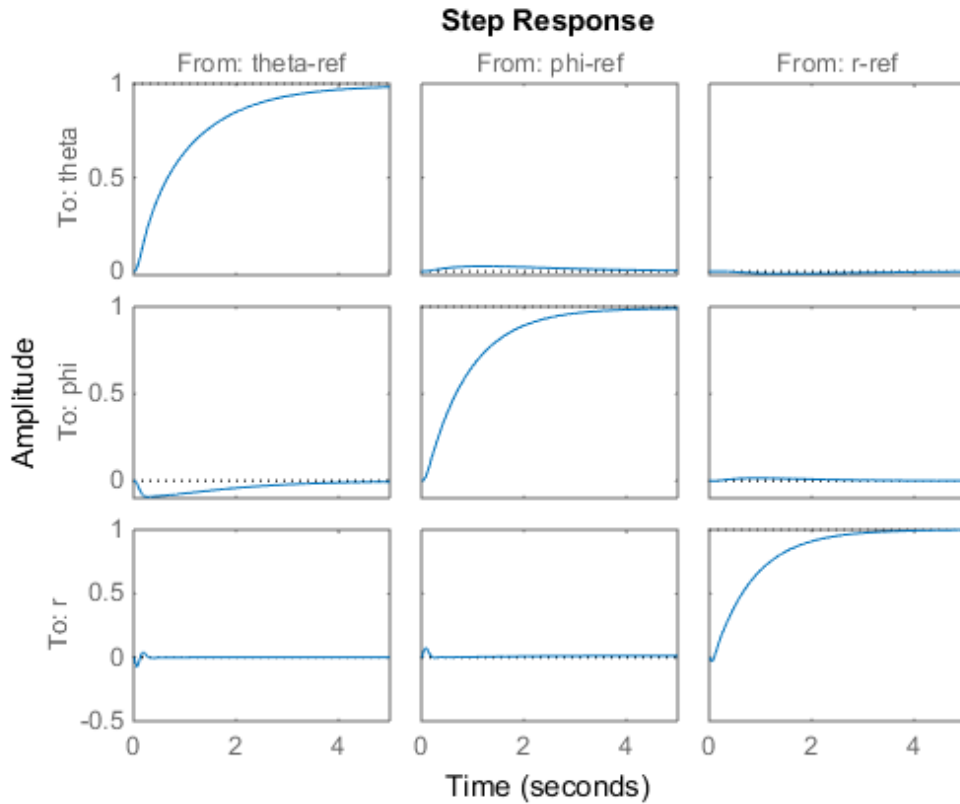
The rise time is about two seconds with no overshoot and little cross-coupling. You can use `viewSpec` for a more thorough validation of each requirement, including a visual assessment of the multivariable stability margins (see `loopmargin` for details):

```
figure('Position',[100,100,900,474])
viewSpec(AllReqs,ST1,Info)
```

7-151
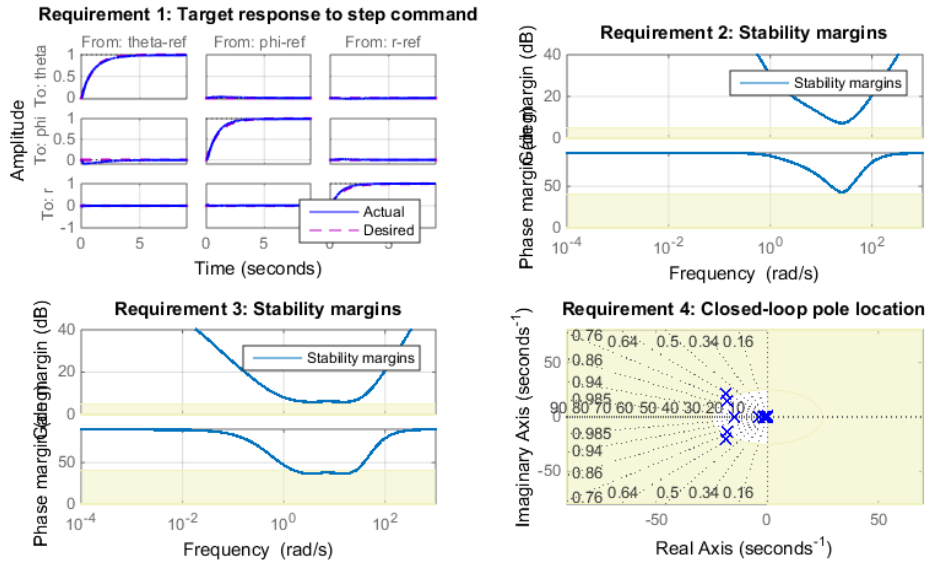
Inspect the tuned values of the PI controllers and static output-feedback gain.

```
showTunable(ST1)
```

```
Block 1: rct_helico/PI1 =

            1
  Kp + Ki * ---
            s

  with Kp = 0.735, Ki = 1.61

Name: PI1
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: rct_helico/PI2 =

            1
  Kp + Ki * ---
            s
```

```
  with Kp = -0.0719, Ki = -1.54

Name: PI2
Continuous-time PI controller in parallel form.

----------------------------------

Block 3: rct_helico/PI3 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.143, Ki = -2.59

Name: PI3
Continuous-time PI controller in parallel form.

----------------------------------

Block 4: rct_helico/SOF =

  d =
                u1          u2          u3          u4          u5
    y1       1.663     -0.1227     0.09323      0.6106  -0.0001622
    y2     -0.2907      -1.427     0.02917     -0.07969     -0.1099
    y3   -0.002587      0.0138      -2.263     -0.01216     0.03313

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:

```
addOpening(ST0,'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

removeBlock(ST0,'SOF');
setBlockParam(ST0,'PI1',ltiblock.pid('C1',PID));
setBlockParam(ST0,'PI2',ltiblock.pid('C2',-PID));
setBlockParam(ST0,'PI3',ltiblock.pid('C3',-PID));
```

Re-tune the three PID controllers and plot the closed-loop step responses.

```
[ST2,fSoft,~,Info] = systune(ST0,AllReqs);
```

Final: Soft = 4.96, Hard = -Inf, Iterations = 64

```
T2 = getIOTransfer(ST2,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
figure, step(T2,5)
```
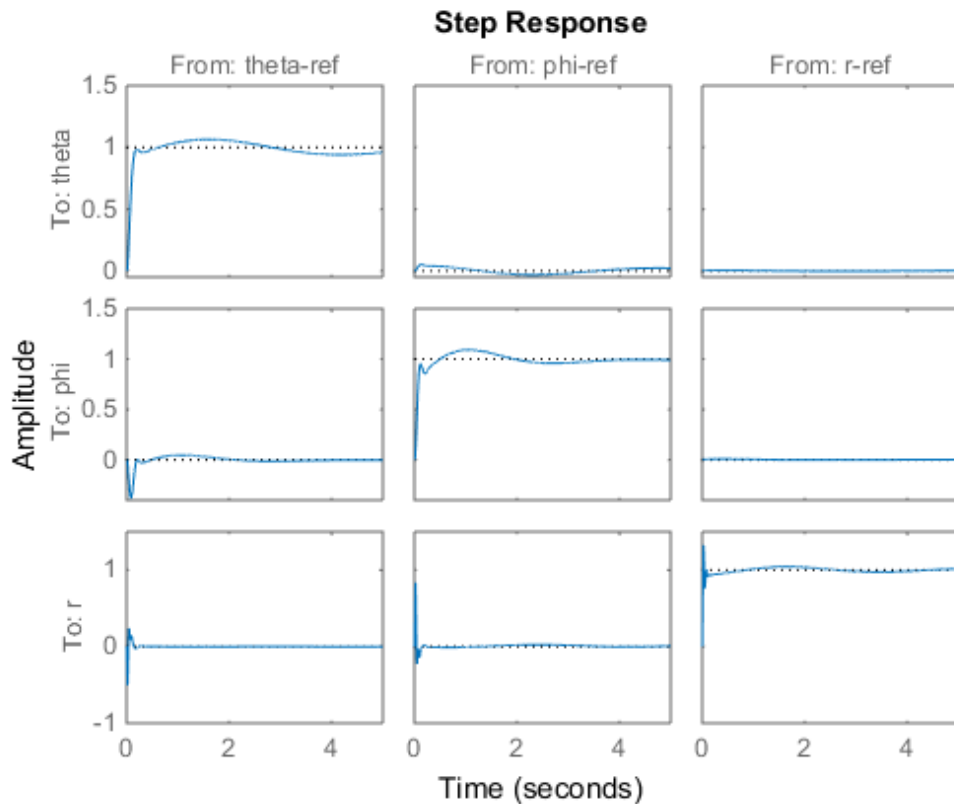
The final value is no longer close to 1 and the step responses confirm the poorer performance with regard to rise time, overshoot, and decoupling. This suggests that the inner loop has an important stabilizing effect that should be preserved.

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 Superieur de l'Aeronautique 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_{zc} 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_{zc}. 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.
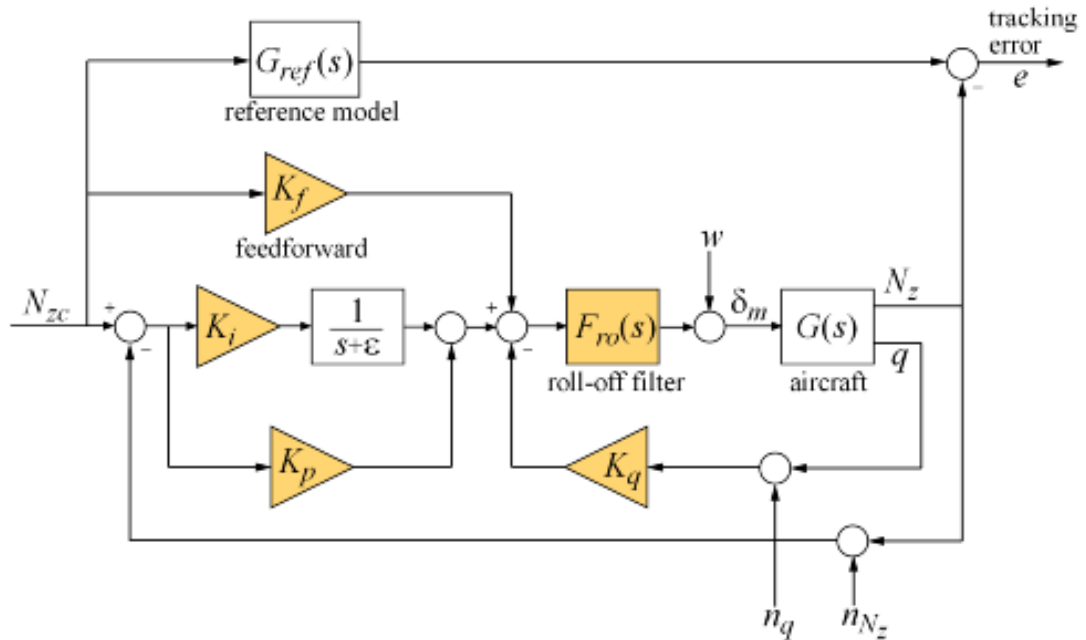
Figure 1: Longitudinal Autopilot Configuration.

The aircraft model $G(s)$ is a 5-state model, the state variables being the aerodynamic speed V (m/s), the climb angle γ (rad), the angle of attack α (rad), the pitch rate q (rad/s), and the altitude H (m). The elevator deflection δ_m (rad) is used to control the vertical load factor N_z. The open-loop dynamics include the α oscillation with frequency and damping ratio ω_n = 1.7 (rad/s) and ζ = 0.33, the phugoid mode ω_n = 0.64 (rad/s) and ζ = 0.06, and the slow altitude mode λ = -0.0026.

```
load ConcordeData G
bode(G,{1e-3,1e2}), grid
title('Aircraft Model')
```
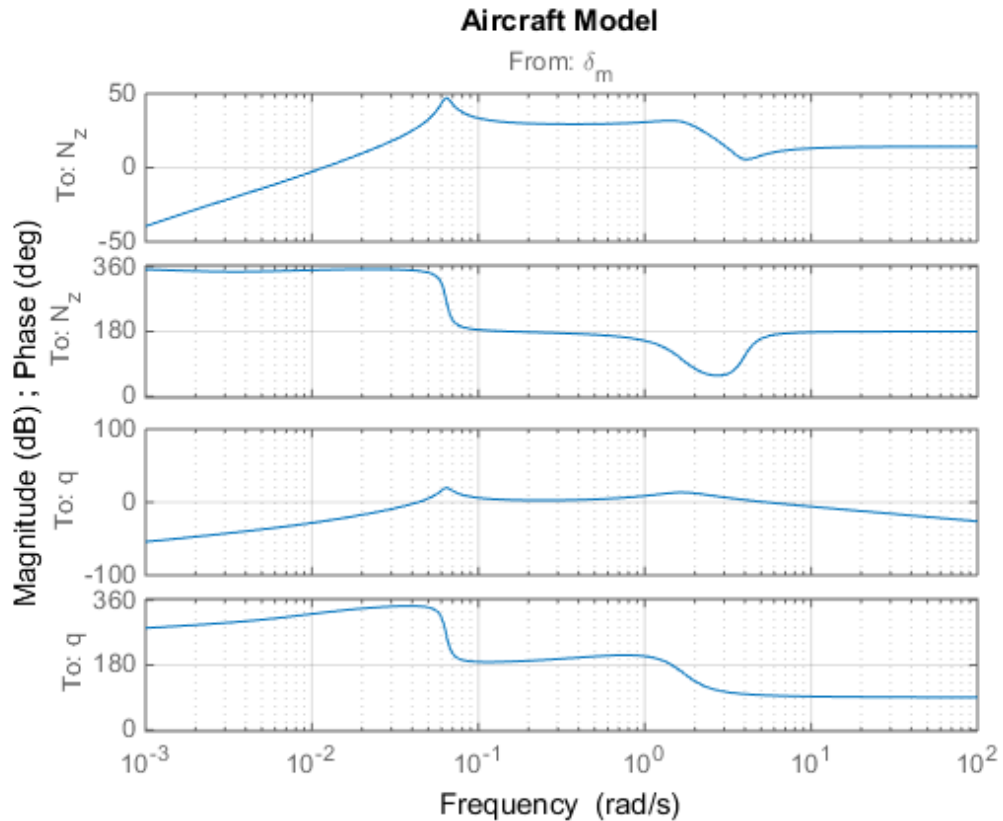
Note the zero at the origin in $G(s)$. Because of this zero, we cannot achieve zero steady-state error and must instead focus on the transient response to acceleration commands. Note that acceleration commands are transient in nature so steady-state behavior is not a concern. This zero at the origin 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 `slTuner` interface to quickly set up the tuning task. Open the Simulink model of the autopilot.

```
open_system('rct_concorde')
```
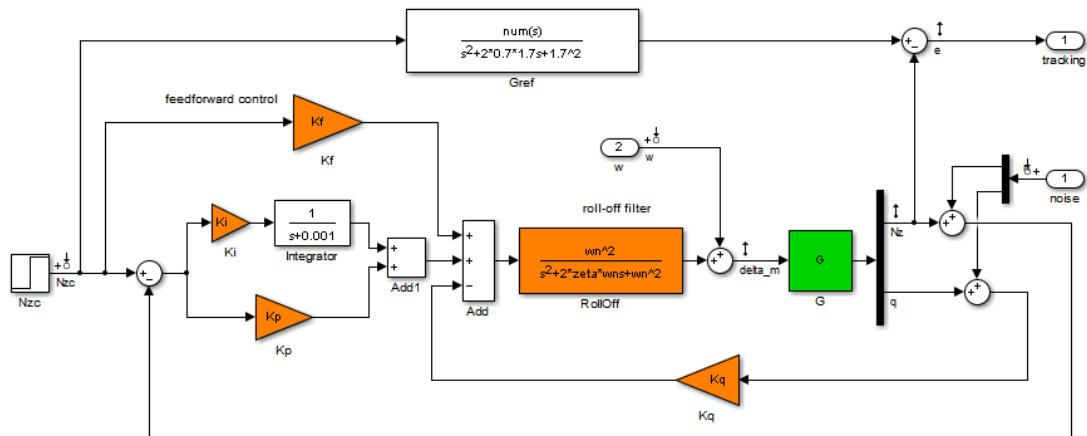
Longitudinal autopilot for a passenger jet.

You can tune this autopilot with the SYSTUNE command, see concorde_demo for details

Copyright 2004-2012 The MathWorks, Inc.

Configure the `slTuner` interface by listing the tuned blocks in the Simulink model (highlighted in orange). This automatically picks all Linear Analysis points in the model as points of interest for analysis and tuning.

```
STO = slTuner('rct_concorde',{'Ki','Kp','Kq','Kf','RollOff'});
```

This also parameterizes each tuned block and initializes the block parameters based on their values in the Simulink model. Note that the four gains `Ki,Kp,Kq,Kf` 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 ζ, ω_n, build the transfer function shown above, and associate it with the `RollOff` 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
```

```
setBlockParam(STO,'RollOff',Fro)    % use Fro to parameterize "RollOff" 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_{zc} 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 δ_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 δ_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)
```

Maximum tracking error: Maximum gain as a function of frequency
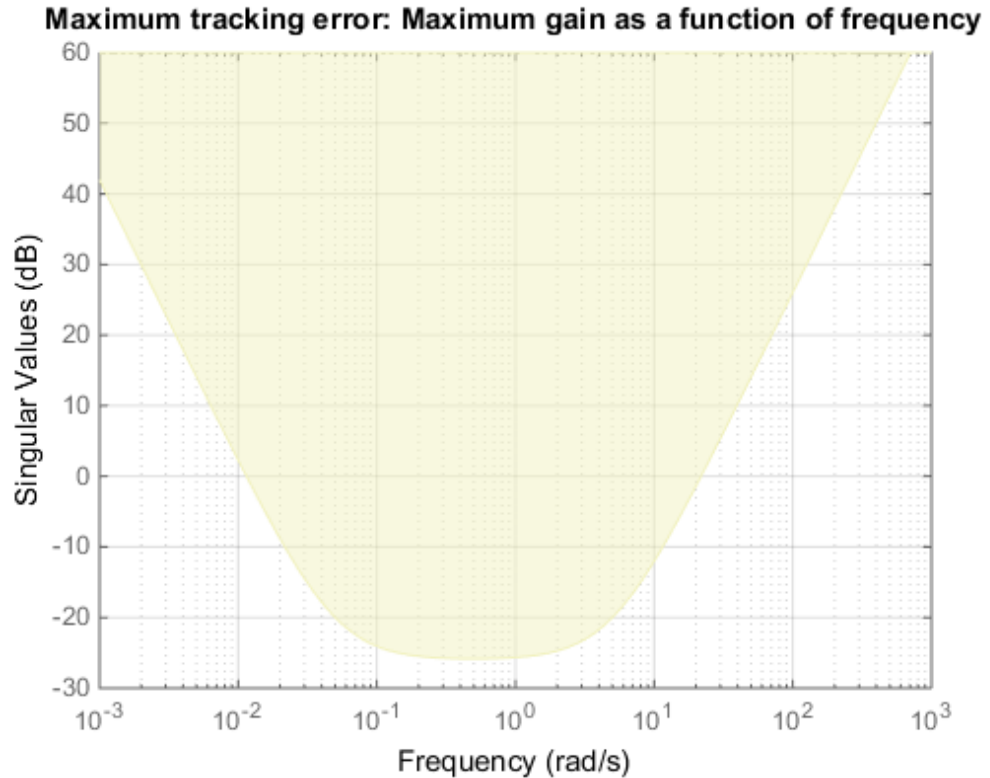
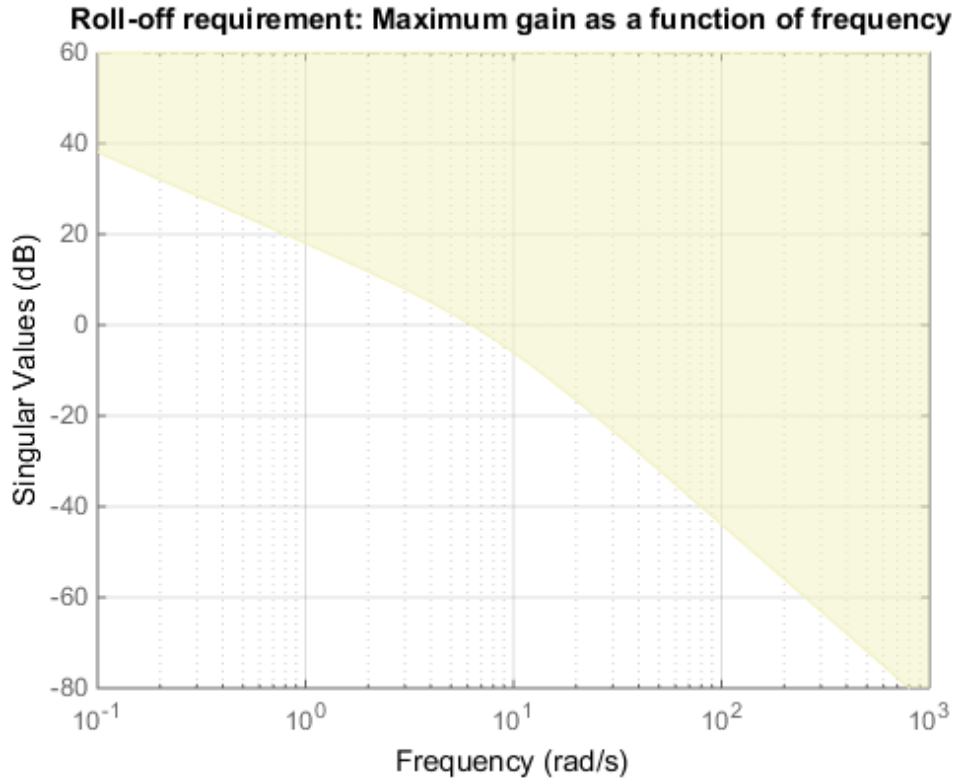Repeat the same process to limit the high-frequency gain from the noise inputs to δ_m and enforce a -40 dB/decade slope in the frequency band from 8 to 800 rad/s

```
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)
```

Roll-off requirement: Maximum gain as a function of frequency

Finally, register the plant input δ_m as a site for open-loop analysis and use `TuningGoal.Margins` to capture the stability margin requirement.

```
addPoint(STO,'delta_m')

Req3 = TuningGoal.Margins('delta_m',7,45);
```

Autopilot Tuning

We are now ready to tune the autopilot parameters with `systune`. This command takes the untuned configuration `STO` and the three design requirements and returns the tuned version `ST` of `STO`. All requirements are satisfied when the final value is less than one.

```
[ST,fSoft,~,Info] = systune(STO,[Req1 Req2 Req3]);
```

```
Final: Soft = 0.965, Hard = -Inf, Iterations = 58
```

Use `showTunable` to see the tuned block values.

```
showTunable(ST)
```

```
Block 1: rct_concorde/Ki =

  d =
            u1
   y1  -0.03004

Name: Ki
Static gain.

---------------------------------

Block 2: rct_concorde/Kp =

  d =
            u1
   y1  -0.009627

Name: Kp
Static gain.

---------------------------------

Block 3: rct_concorde/Kq =

  d =
            u1
   y1  -0.2871

Name: Kq
Static gain.

---------------------------------

Block 4: 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 = getBlockValue(ST,'RollOff');
tf(Fro)
```

```
ans =

          23.22
  --------------------
  s^2 + 4.955 s + 23.22

Continuous-time transfer function.
```
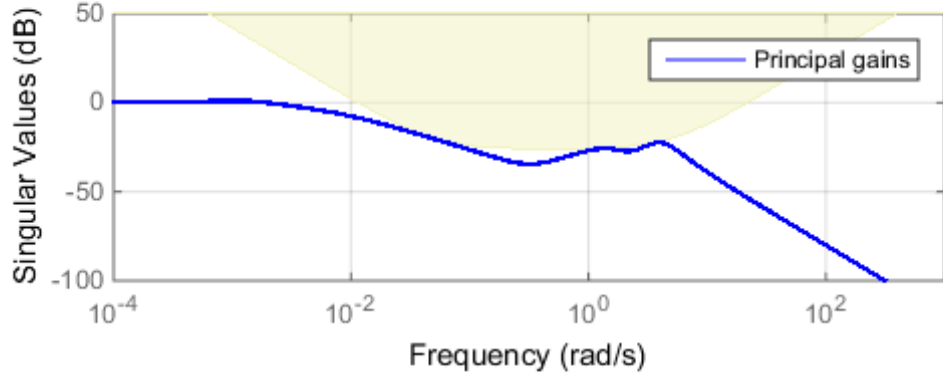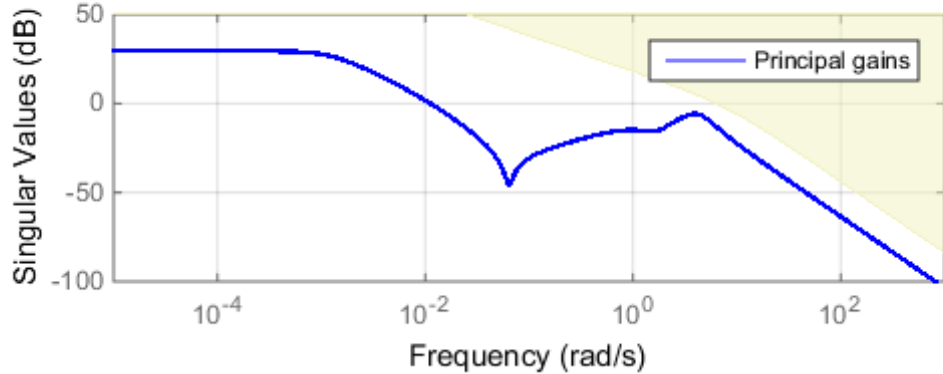
Finally, use `viewSpec` to graphically verify that all requirements are satisfied.

```
figure('Position',[100,100,550,710])
viewSpec([Req1 Req2 Req3],ST)
```

Maximum tracking error: Maximum gain as a function of frequency

Roll-off requirement: Maximum gain as a function of frequency

Requirement 3: Stability margins
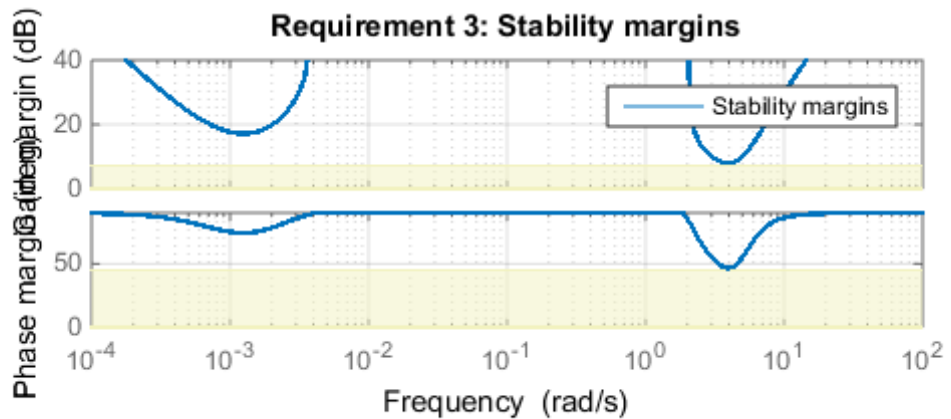
Closed-Loop Simulations

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 `Nzc` to `Nz`:

```
Gref = tf(1.7^2,[1 2*0.7*1.7 1.7^2]);     % reference model

T = getIOTransfer(ST,'Nzc','Nz');  % transfer Nzc -> Nz

figure, step(T,'b',Gref,'b--',6), grid,
ylabel('N_z'), legend('Actual response','Reference model')
```
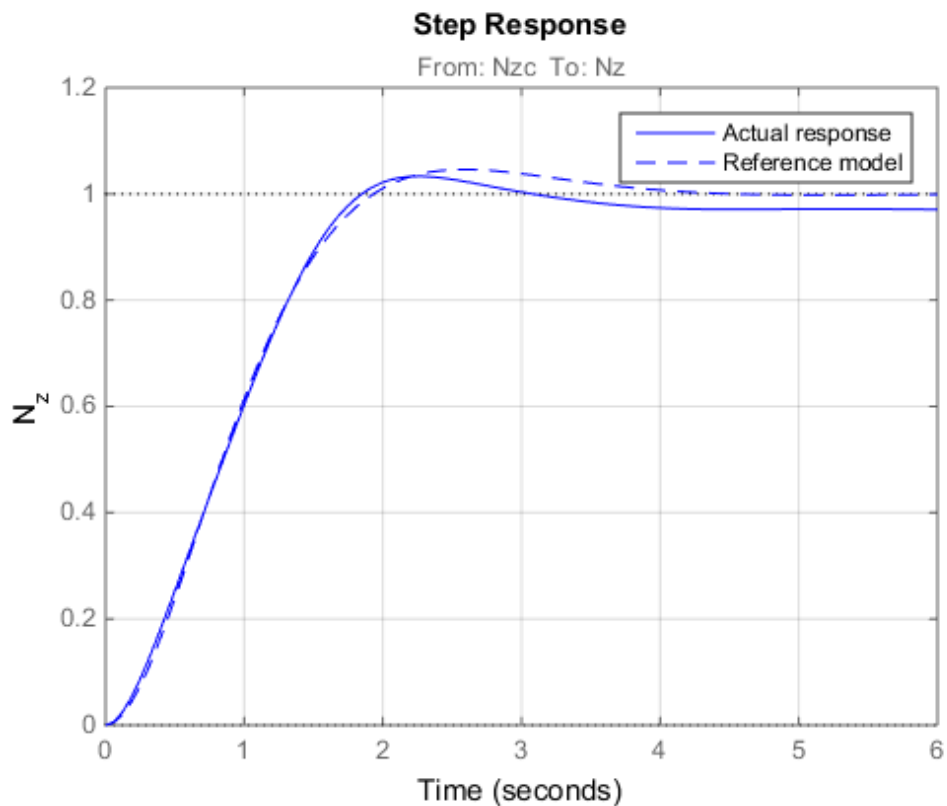
Also plot the deflection δ_m and the respective contributions of the feedforward and feedback paths:

```
T = getIOTransfer(ST,'Nzc','delta_m');  % transfer Nzc -> delta_m
Kf = getBlockValue(ST,'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')
```
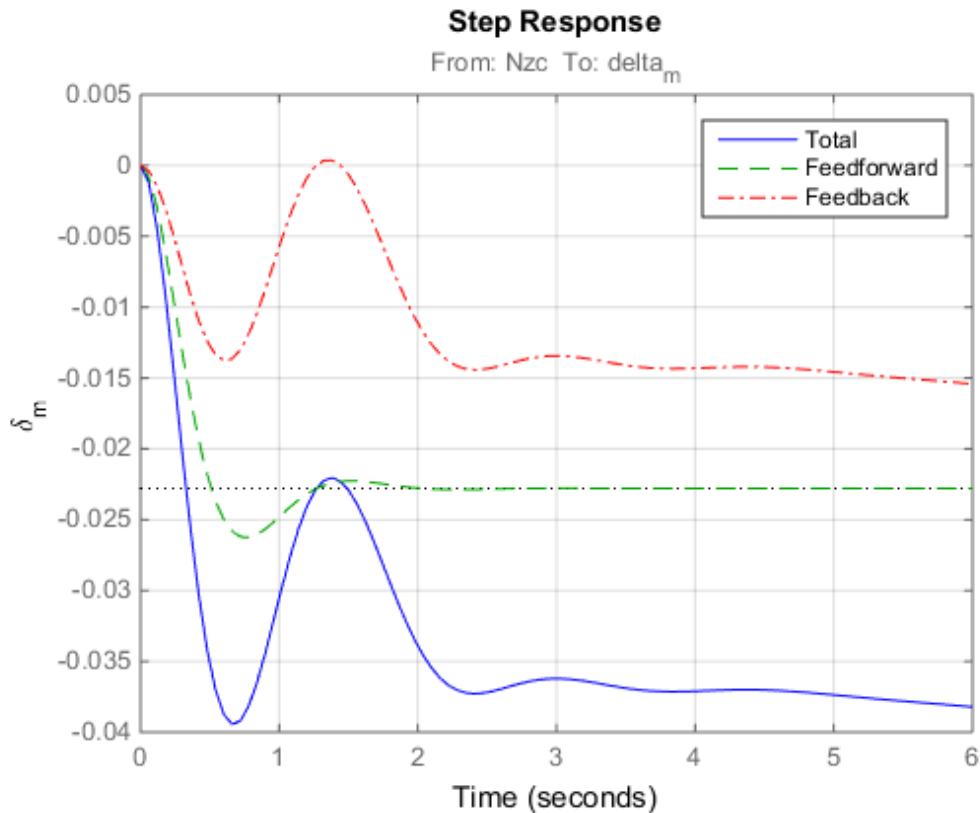

Finally, check the roll-off and stability margin requirements by computing the open-loop response at δ_m.

```
OL = getLoopTransfer(ST,'delta_m',-1); % negative-feedback loop transfer
```

```
margin(OL);
grid;
xlim([1e-3,1e2]);
```
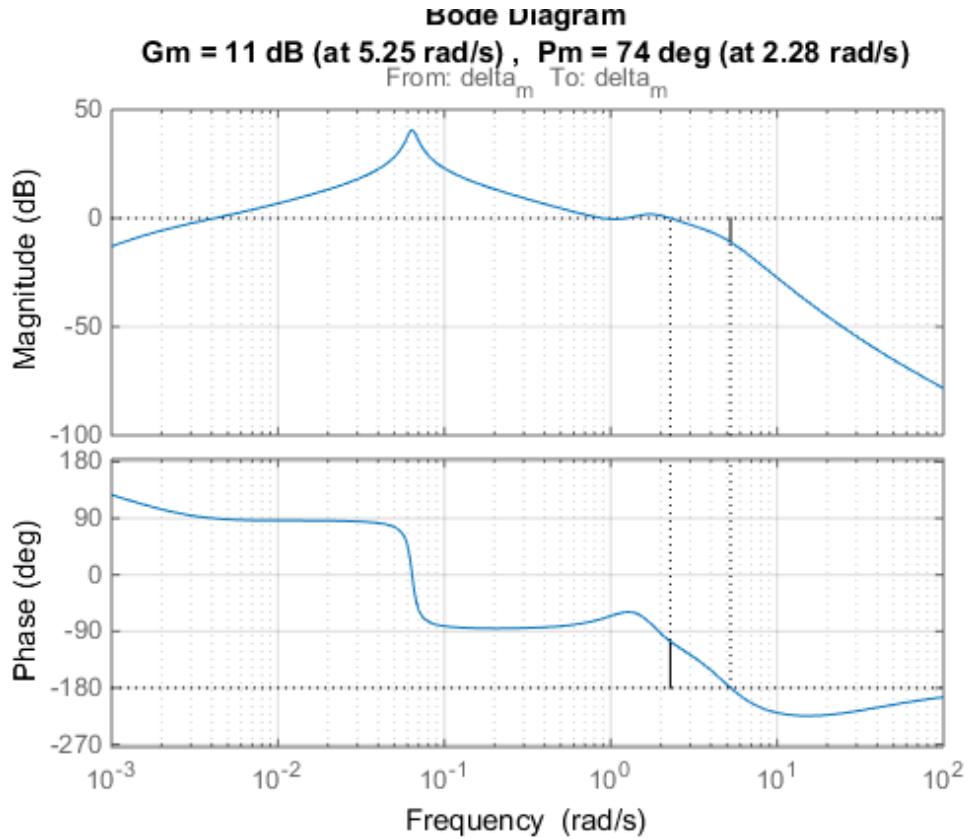

The Bode plot confirms a roll-off of -40 dB/decade past 8 rad/s and indicates gain and phase margins in excess of 10 dB and 70 degrees.

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')
```
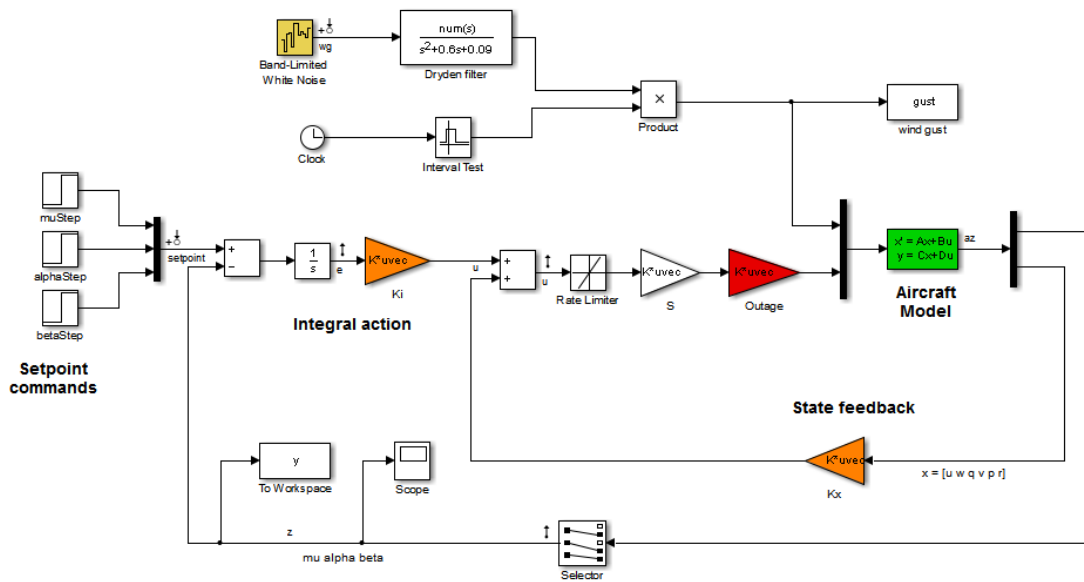
The aircraft is modeled as a rigid 6th-order state-space system with the following state variables (units are mph for velocities and deg/s for angular rates):

- 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 `Ki` and `Kx` 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`:

$$J = lim_{T \to \infty} E \left(\frac{1}{T} \int_0^T \|W_e e\|^2 + \|W_u u\|^2 dt \right).$$

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 `slTuner` interface to set up the tuning task. List the blocks to be tuned and specify the nine flight conditions by varying the `outage` variable in the Simulink model. Because you can only vary scalar parameters in `slTuner`, independently specify the values taken by each entry of the `outage` vector.

```
OutageData = struct(...
    'Name',{'outage(1)','outage(2)','outage(3)','outage(4)','outage(5)'},...
    'Value',mat2cell(OutageCases,9,[1 1 1 1 1]));
STO = slTuner('faultTolerantAircraft',{'Ki','Kx'},OutageData);
```

Use `systune` to tune the controller gains subject to the nominal requirements. Treat the wind gust alleviation as a hard constraint.

```
[ST,fSoft,gHard]  = systune(STO,SoftNom,HardNom);

Final: Soft = 22.6, Hard = 0.99952, Iterations = 277
```

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(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% Bank-angle setpoint simulation
plotResponses(OutageCases,1,0,0);
```

```
% Angle-of-attack setpoint simulation
plotResponses(OutageCases,0,1,0);
```

Flight-path bank angle rate (deg/s)

Angle of attack (deg)
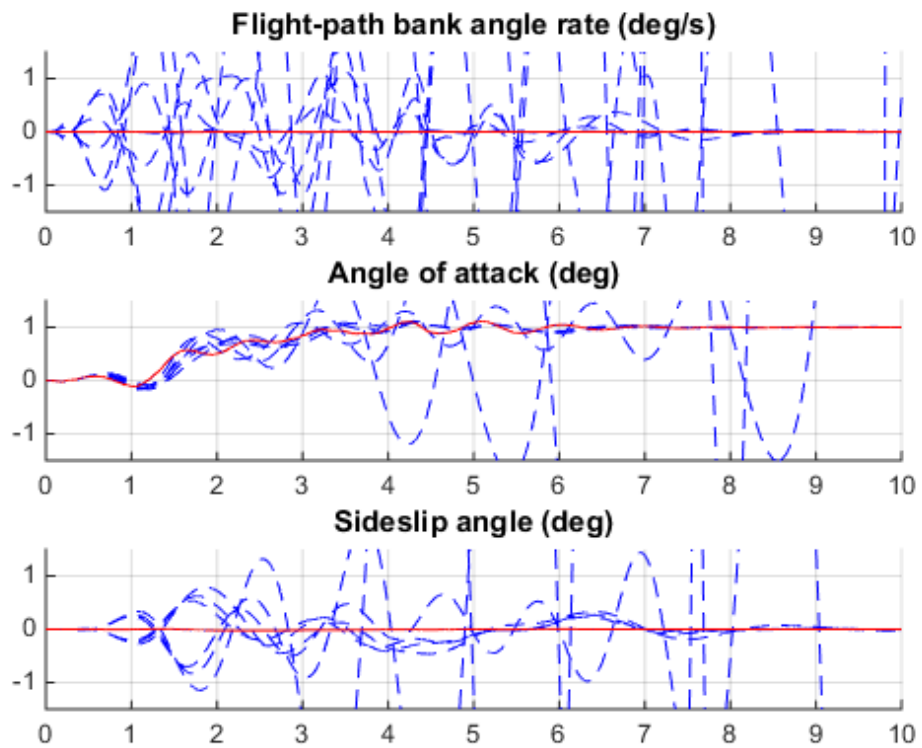
Sideslip angle (deg)

```
% 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 as well as the relaxed requirements for all eight outage scenarios.

```
[ST,fSoft,gHard]  = systune(STO,[SoftNom;SoftOut],[HardNom;HardOut]);

Final: Soft = 25.8, Hard = 0.99796, Iterations = 519
```

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(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% Bank-angle setpoint simulation
plotResponses(OutageCases,1,0,0);
```
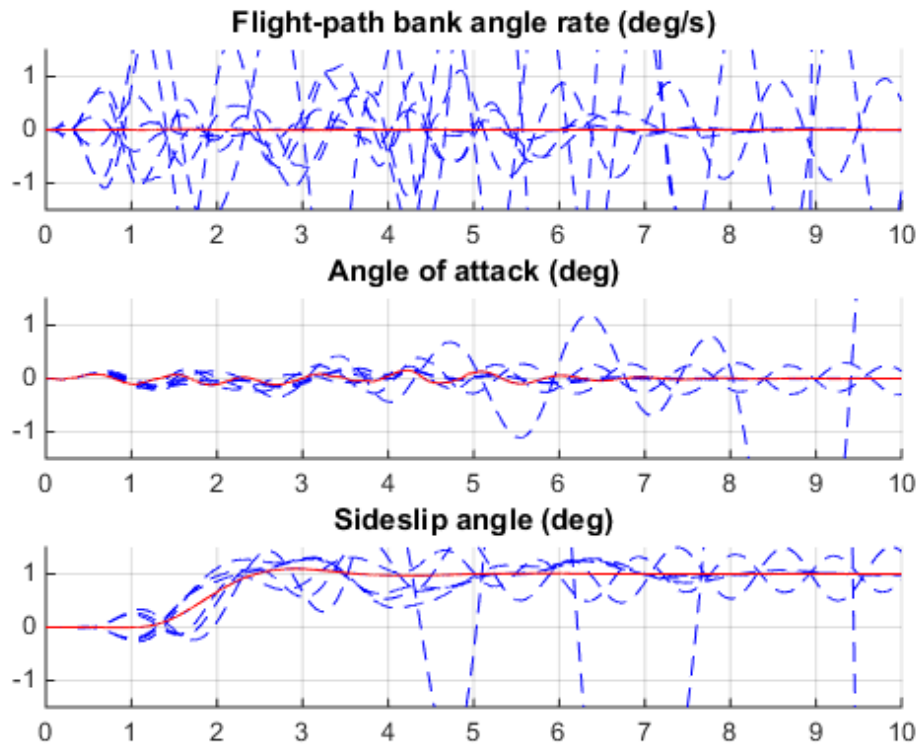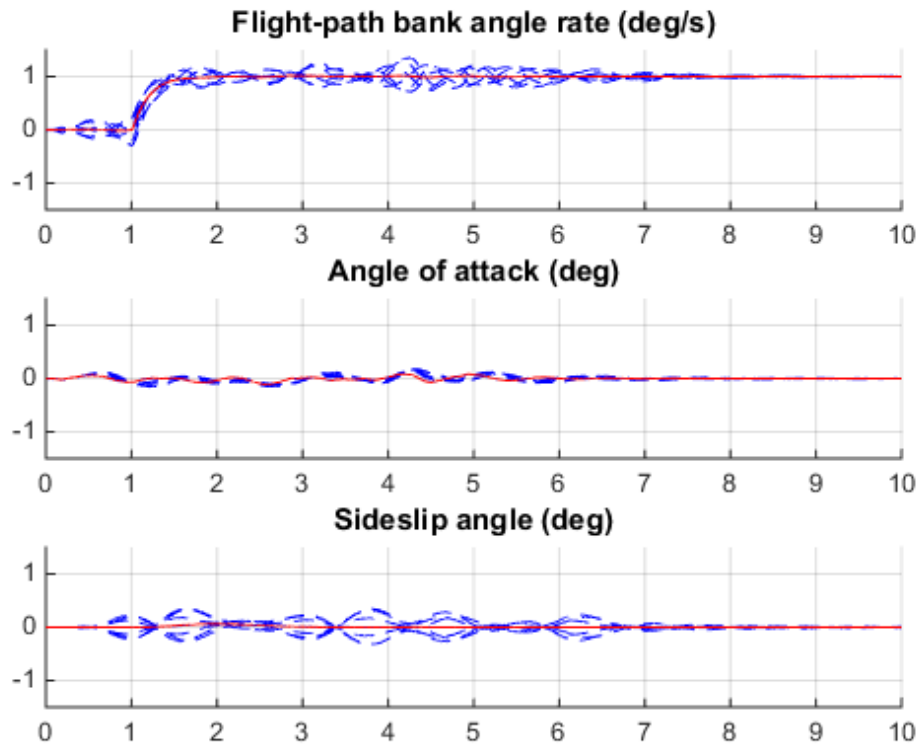


```
% Angle-of-attack setpoint simulation
plotResponses(OutageCases,0,1,0);
```
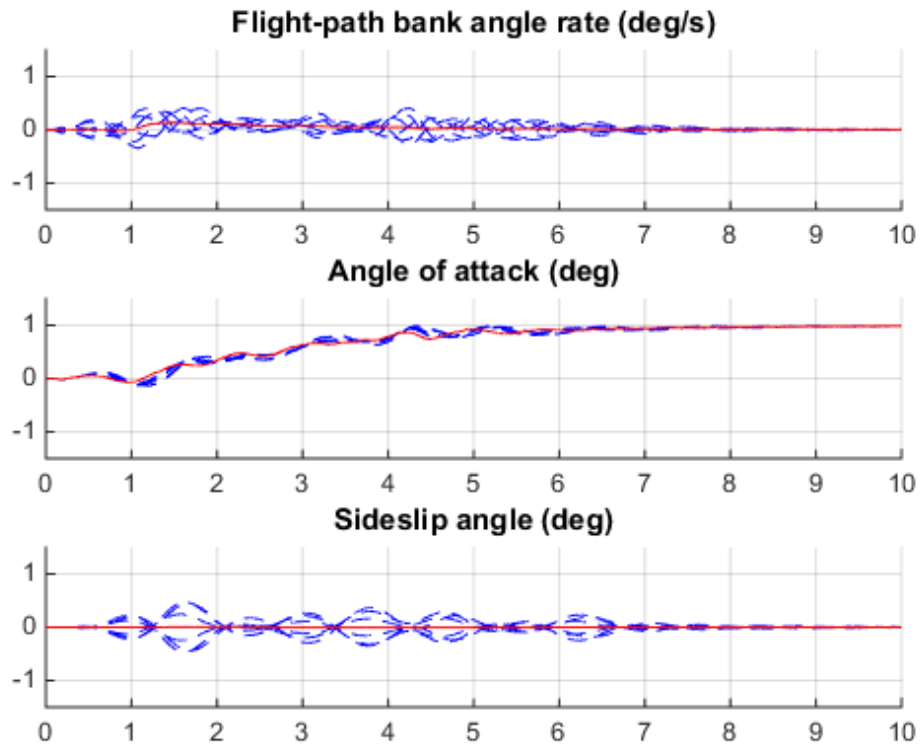
```
% Sideslip-angle setpoint simulation
plotResponses(OutageCases,0,0,1);
```
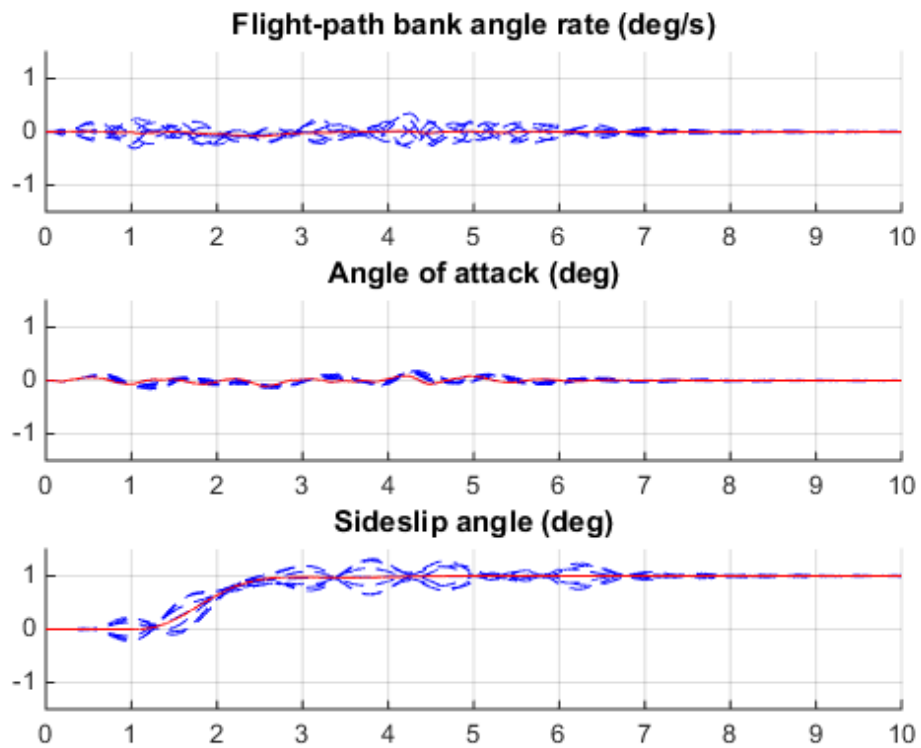
The controller now provides acceptable performance for all outage scenarios considered in this example. The design could be further refined by adding specifications such as minimum stability margins and gain limits to avoid actuator rate saturation.

Fixed-Structure H-infinity Synthesis with HINFSTRUCT

This example uses the `hinfstruct` command to tune a fixed-structure controller subject to H_∞ constraints.

Introduction

The `hinfstruct` command extends classical H_∞ synthesis (see `hinfsyn`) to fixed-structure control systems. This command is meant for users already comfortable with the `hinfsyn` workflow. If you are unfamiliar with H_∞ synthesis or find augmented plants and weighting functions intimidating, use `systune` and `looptune` instead. See *"Tuning Control Systems with SYSTUNE"* for the `systune` 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
```
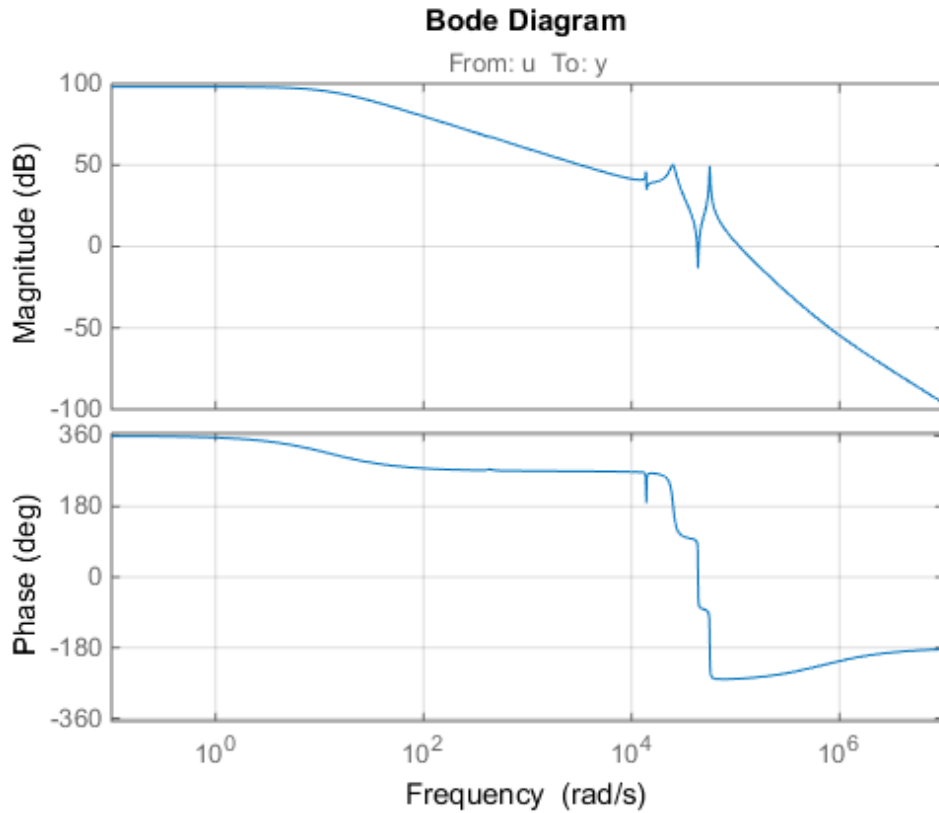
Bode Diagram

From: u To: y

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.
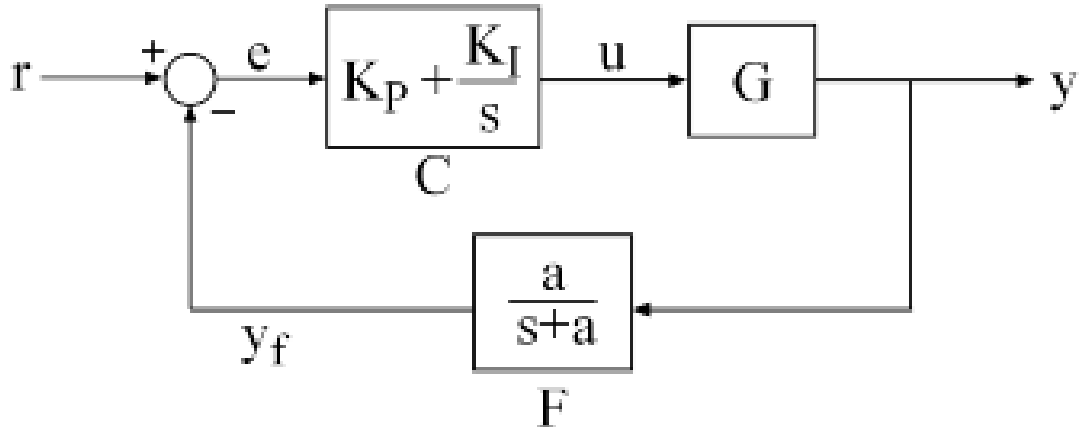
Figure 1: Control Structure

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

7-181

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')
```
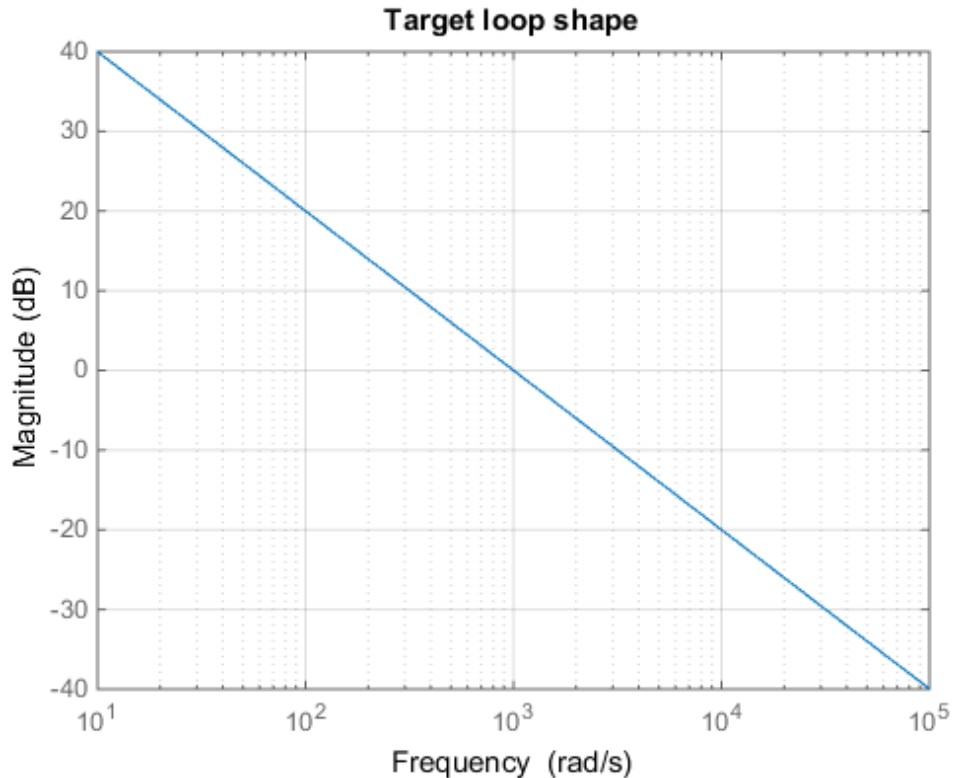

Note that we chose a bi-proper, bi-stable realization to avoid technical difficulties with marginally stable poles and improper inverses. In order to tune $C(s)$ and $F(s)$ with `hinfstruct`, we must turn this target loop shape into constraints on the closed-loop gains. A systematic way to go about this 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 nw.
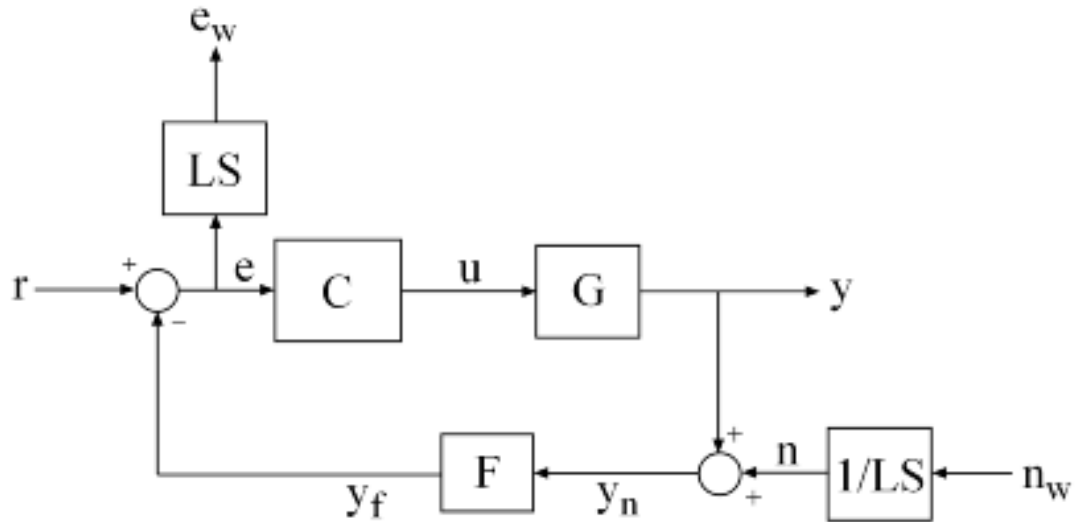
Figure 2: Closed-Loop Formulation

If $T(s)$ denotes the closed-loop transfer function from (r,nw) to (y,ew), the gain constraint

$$\|T\|_\infty < 1$$

secures the following desirable properties:

- At low frequency (w<wc), the open-loop gain stays above the gain specified by the target loop shape LS
- At high frequency (w>wc), 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
TO = connect(G,Wn,We,CO,FO,Sum1,Sum2,{'r','nw'},{'y','ew'});
```

These commands construct a generalized state-space model `TO` of $T(s)$. This model depends on the tunable blocks `C` and `a`:

```
TO.Blocks
```

```
ans =

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

Note that `TO` captures the following "Standard Form" of the block diagram of Figure 2 where the tunable components C, F are separated from the fixed dynamics.
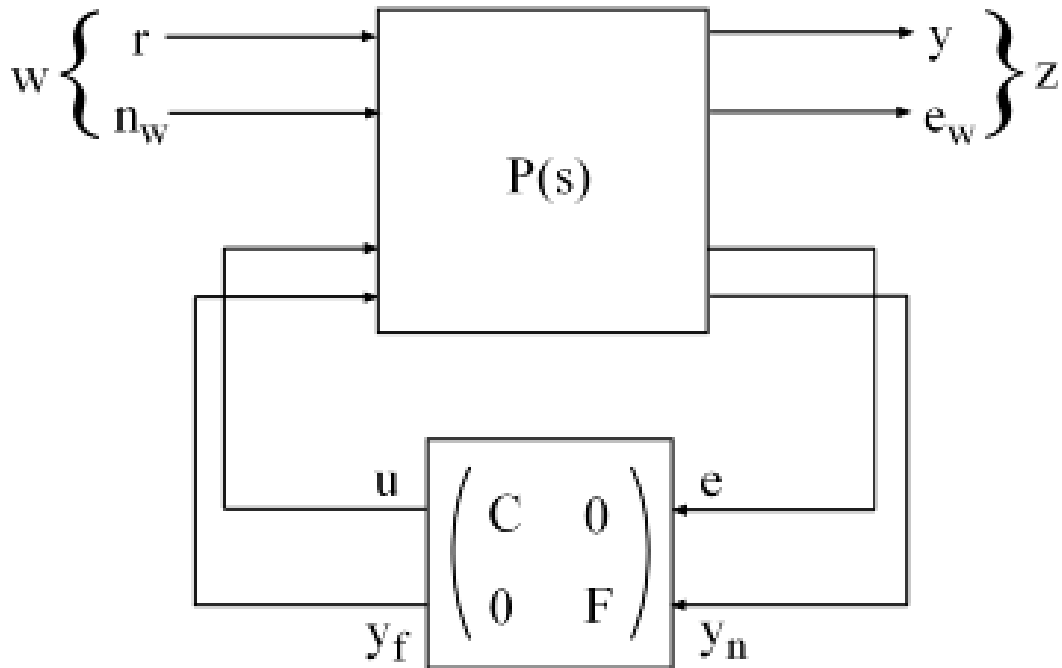
Figure 3: Standard Form for Disk-Drive Loop Shaping

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 `C0` and `F0`:

```
rng('default')
opt = hinfstructOptions('Display','final','RandomStart',5);
T = hinfstruct(T0,opt);

Final: Peak gain = 3.88, Iterations = 108
Final: Peak gain = 1.56, Iterations = 90
Final: Peak gain = 597, Iterations = 185
      Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 3.94, Iterations = 108
```

```
Final: Peak gain = 1.56, Iterations = 97
Final: Peak gain = 2.17, 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 =

            1
  Kp + Ki * ---
            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')
```
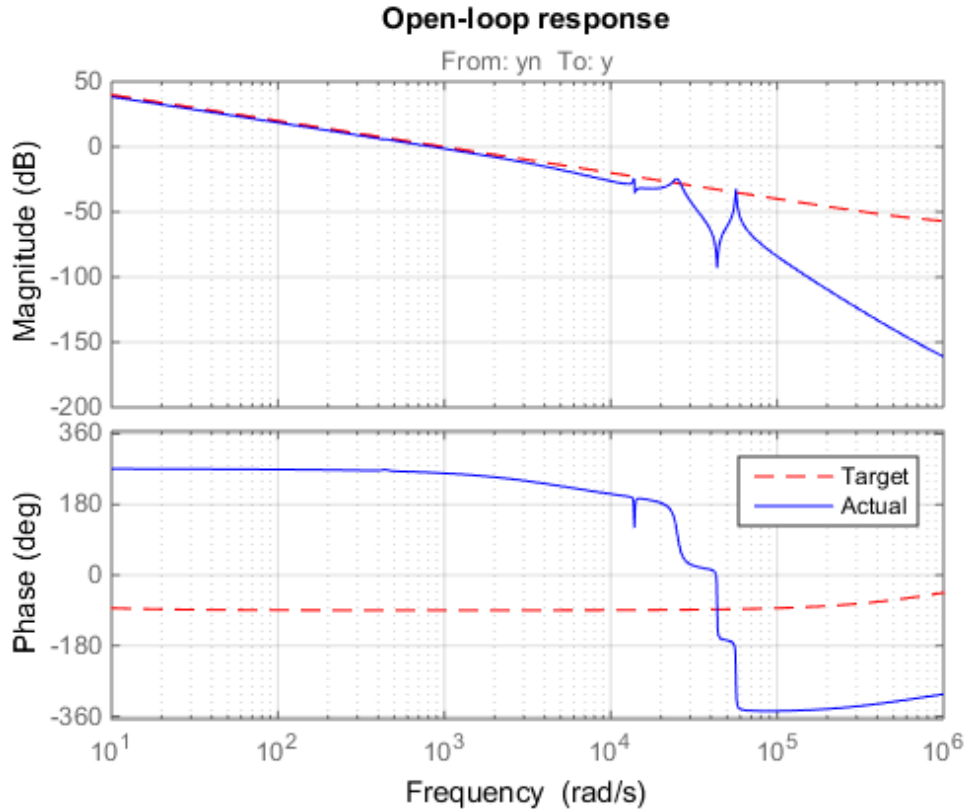

The 0dB crossover frequency and overall loop shape are as expected. The stability margins can be read off the plot by right-clicking and selecting the **Characteristics** menu. This design has 24dB gain margin and 81 degrees phase margin. Plot the closed-loop step response from reference r to position y:

```
step(feedback(G*C,F)), grid, title('Closed-loop response')
```
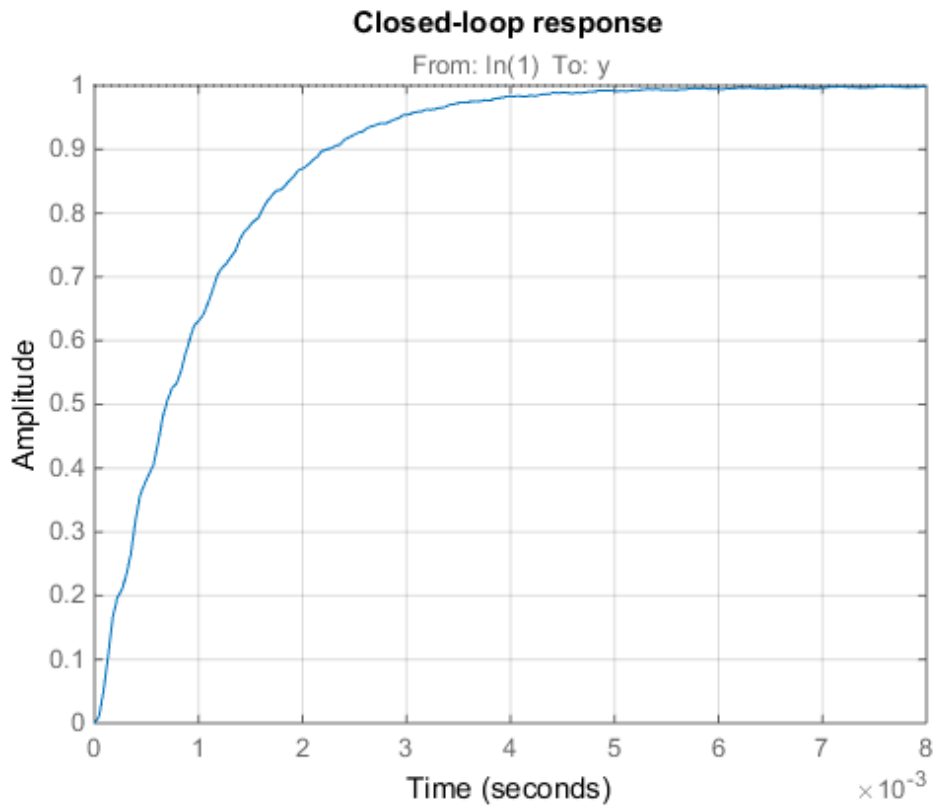
While the response has no overshoot, there is some residual wobble due to the first resonant peaks in G. You might consider adding a notch filter in the forward path to remove the influence of these modes.

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 Linear Analysis points in the Simulink model.
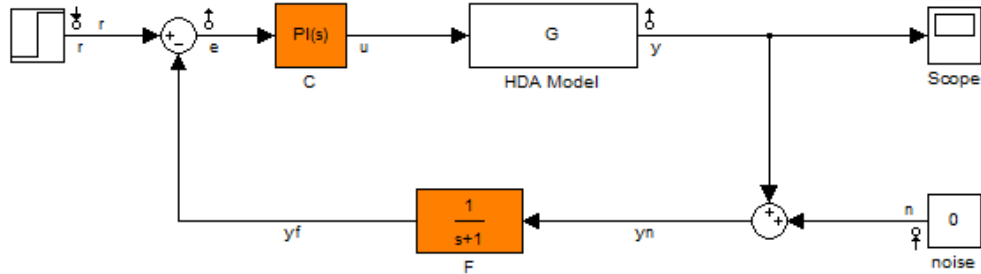
See hinfstruct_demo to see how you can tune the PI gains and the filter coefficient
with the HINFSTRUCT command.

Copyright 2004-2010 The MathWorks, Inc.

Then create an instance of the `slTuner` interface and mark the Simulink blocks `C` and `F`
as tunable:

```
STO = slTuner('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
setBlockParam(STO,'F',tf(a,[1 a]));
```

Finally, use `getIOTransfer` to derive a tunable model of the closed-loop transfer
function $T(s)$ (see Figure 2)

```
% Compute tunable model of closed-loop transfer (r,n) -> (y,e)
TO = getIOTransfer(STO,{'r','n'},{'y','e'});

% Add weighting functions in n and e channels
TO = blkdiag(1,LS) * TO * blkdiag(1,1/LS);
```

You are now ready to tune the controller gains with `hinfstruct`:

```
rng(0)
opt = hinfstructOptions('Display','final','RandomStart',5);
T = hinfstruct(TO,opt);
```

7-189

```
Final: Peak gain = 3.88, Iterations = 119
Final: Peak gain = 1.56, Iterations = 87
Final: Peak gain = 598, Iterations = 197
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 3.88, Iterations = 104
Final: Peak gain = 1.56, Iterations = 93
Final: Peak gain = 1.56, Iterations = 88
```

Verify that you obtain the same tuned values as with the MATLAB approach:

```
showTunable(T)

C =

            1
  Kp + Ki * ---
            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 `systune` 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')
```
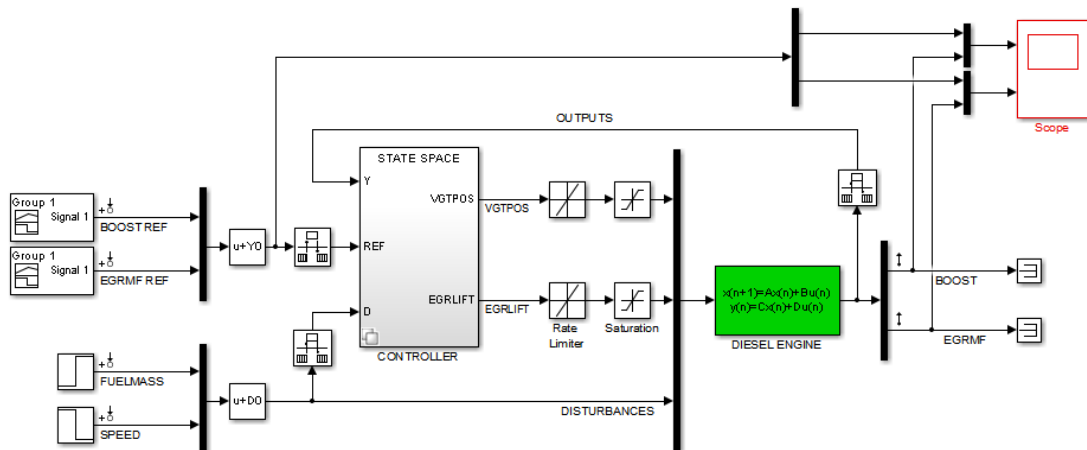

The VGT/EGR control system is modeled in Simulink. The controller adjusts the positions `EGRLIFT` and `VGTPOS` of the EGR and VGT valves. It has access to the boost pressure and EGR massflow targets and measured values, as well as fuel mass and engine speed measurements. Both valves have rate and saturation limits. The plant model is sampled every 0.1 seconds and the control signals `EGRLIFT` and `VGTPOS` are refreshed every 0.2 seconds. This example considers step changes of +10 KPa in boost pressure and +3 g/s in EGR massflow, and disturbances of +5 mg in fuel mass and -200 rpm in speed.

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 `VGTPOS` 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')
```
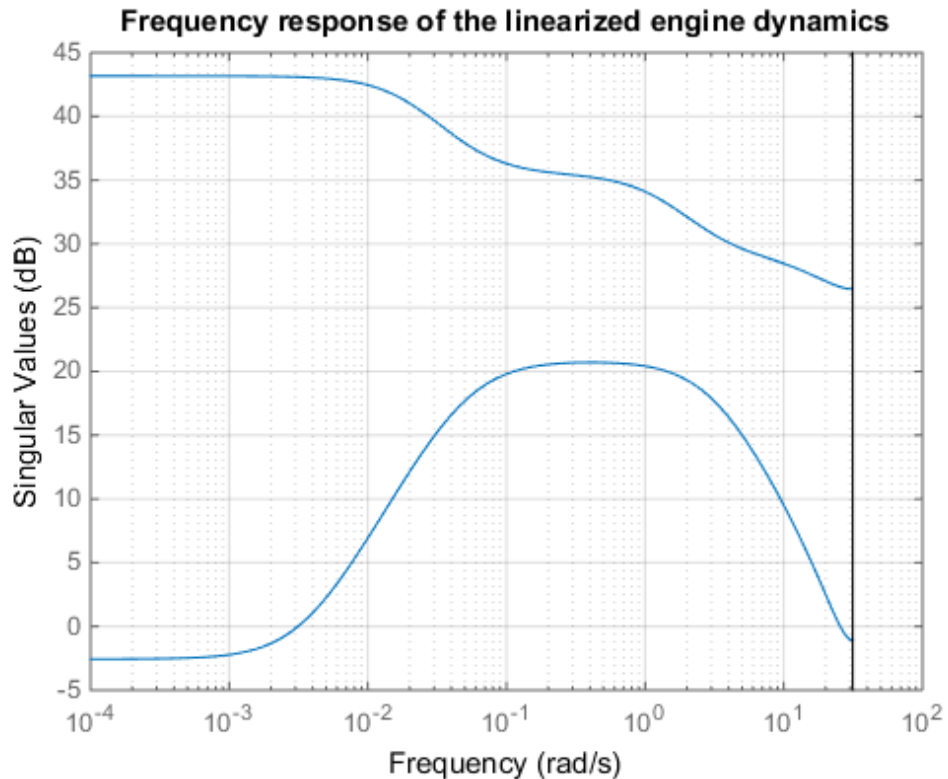

Control Objectives

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. Specify the amplitudes of the step changes to ensure that cross-couplings are small *relative* to these changes.

```
% 5-second response time, steady-state error less than 5%
TR = TuningGoal.Tracking({'BOOST REF';'EGRMF REF'},{'BOOST';'EGRMF'},5,0.05);
TR.Name = 'Setpoint tracking';
TR.InputScaling = [10 3];
```

For the second objective, treat the speed and fuel mass variations as step disturbances and specify the peak amplitude and settling time of the resulting variations in boost pressure and EGR massflow. Also specify the signal amplitudes to properly reflect the relative contribution of each disturbance.

```
% Peak<0.5, settling time<5
DR = TuningGoal.StepRejection({'FUELMASS';'SPEED'},{'BOOST';'EGRMF'},0.5,5);
DR.Name = 'Disturbance rejection';
DR.InputScaling = [5 200];
DR.OutputScaling = [10 3];
```

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 = 'Roll-off';

% 7 dB of gain margin and 45 degrees of phase margin
M1 = TuningGoal.Margins({'EGRLIFT','VGTPOS'},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.
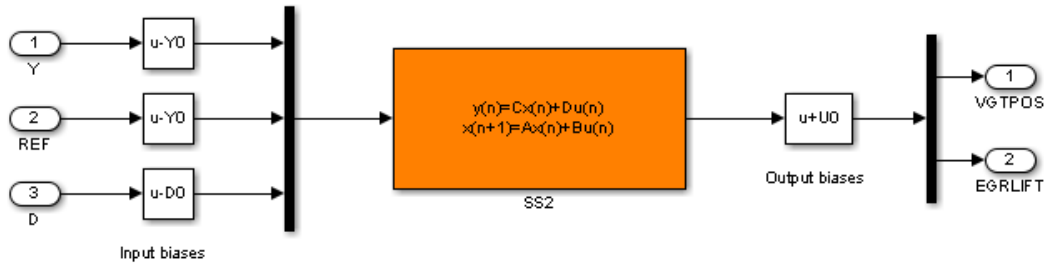
2-DOF MIMO PID with disturbance feedforward

Figure 1: Second-order blackbox controller.

Use the `slTuner` 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 = slTuner('rct_diesel','SS2');
ST0.Ts = 0.2;
addPoint(ST0,{'EGRLIFT','VGTPOS','DIESEL ENGINE'})
```

Now use `systune` 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 goals). Randomize the starting point to reduce exposure to undesirable local minima.

```
Opt = systuneOptions('RandomStart',2);
rng(0), [ST1,~,~,Info1] = systune(ST0,[TR DR],[M1 M2 RO],Opt);

Final: Soft = 1.05, Hard = 0.99841, Iterations = 723
Final: Soft = 1.05, Hard = 0.98913, Iterations = 463
Final: Soft = 1.05, Hard = 0.87833, Iterations = 447
```

All requirements are nearly met (a requirement is satisfied when its normalized value is less than 1). Verify this graphically.

```
figure('Position',[10,10,1071,714])
viewSpec([TR DR RO M1 M2],ST1,Info1)
```
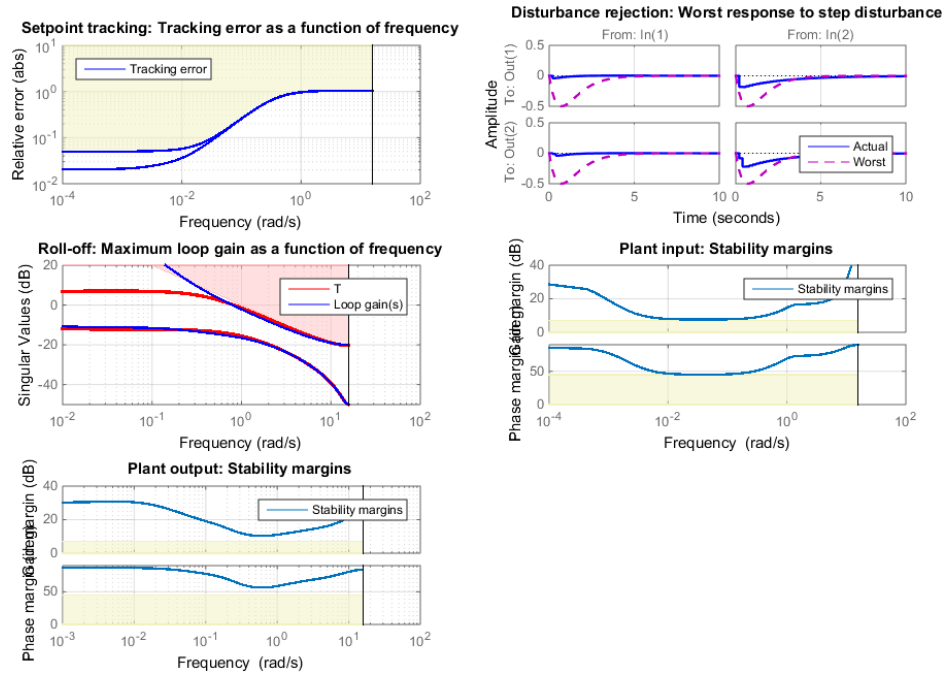
Plot the setpoint tracking and disturbance rejection responses. Scale by the signal amplitudes to show normalized effects (boost pressure changes by +10 KPa, EGR massflow by +3 g/s, fuel mass by +5 mg, and speed by -200 rpm).

```
figure('Position',[100,100,560,500])
T1 = getIOTransfer(ST1,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
T1 = diag([1/10 1/3 1 1]) * T1 * diag([10 3]);
subplot(211), step(T1(1:2,:),15), title('Setpoint tracking')
subplot(212), step(T1(3:4,:),15), title('Control effort')
```
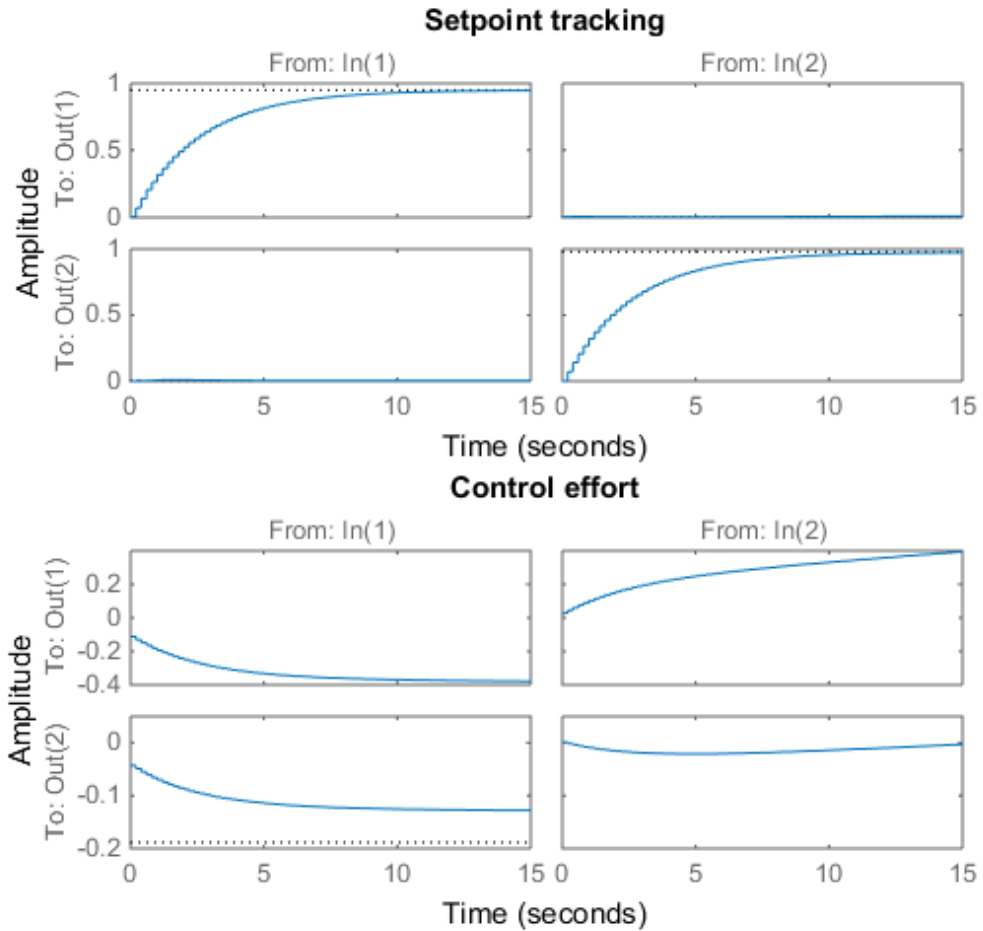
```
D1 = getIOTransfer(ST1,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
D1 = diag([1/10 1/3 1 1]) * D1 * diag([5 -200]);
subplot(211), step(D1(1:2,:),15), title('Disturbance rejection')
subplot(212), step(D1(3:4,:),15), title('Control effort')
```
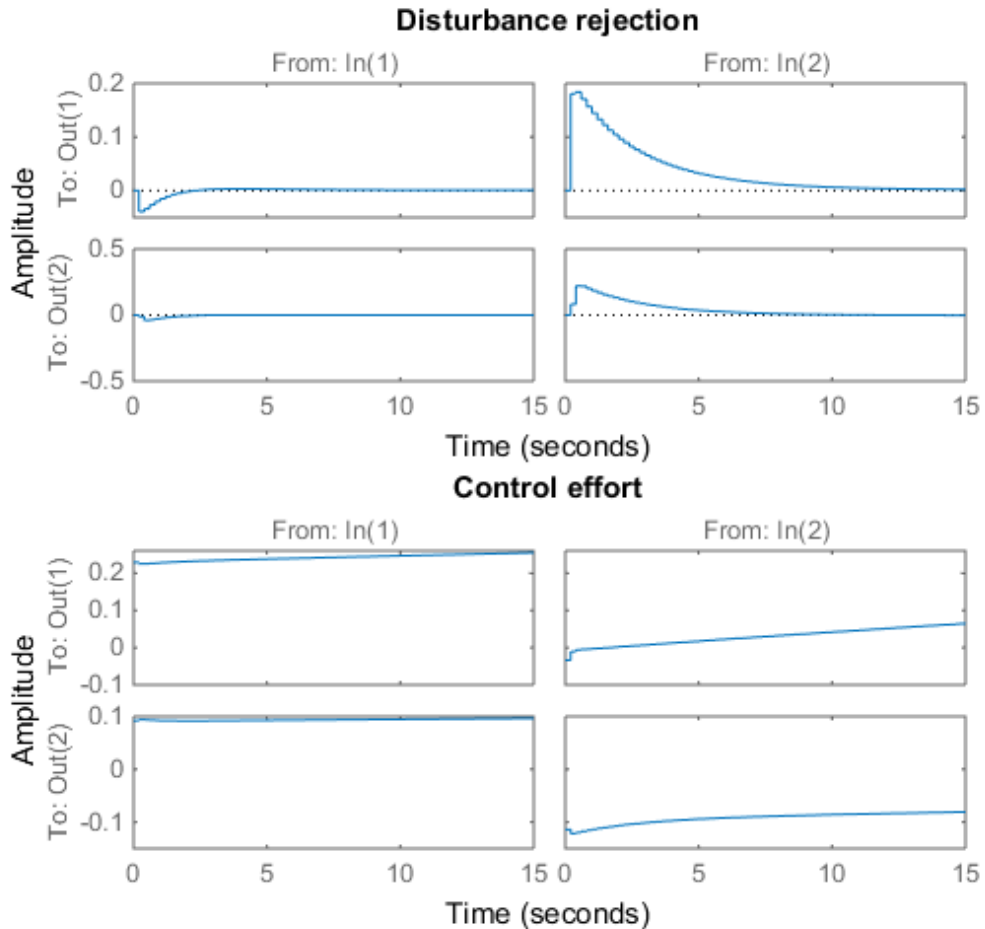
The controller responds in less than 5 seconds with minimum cross-coupling between the BOOST and EGRMF variables.

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')
```
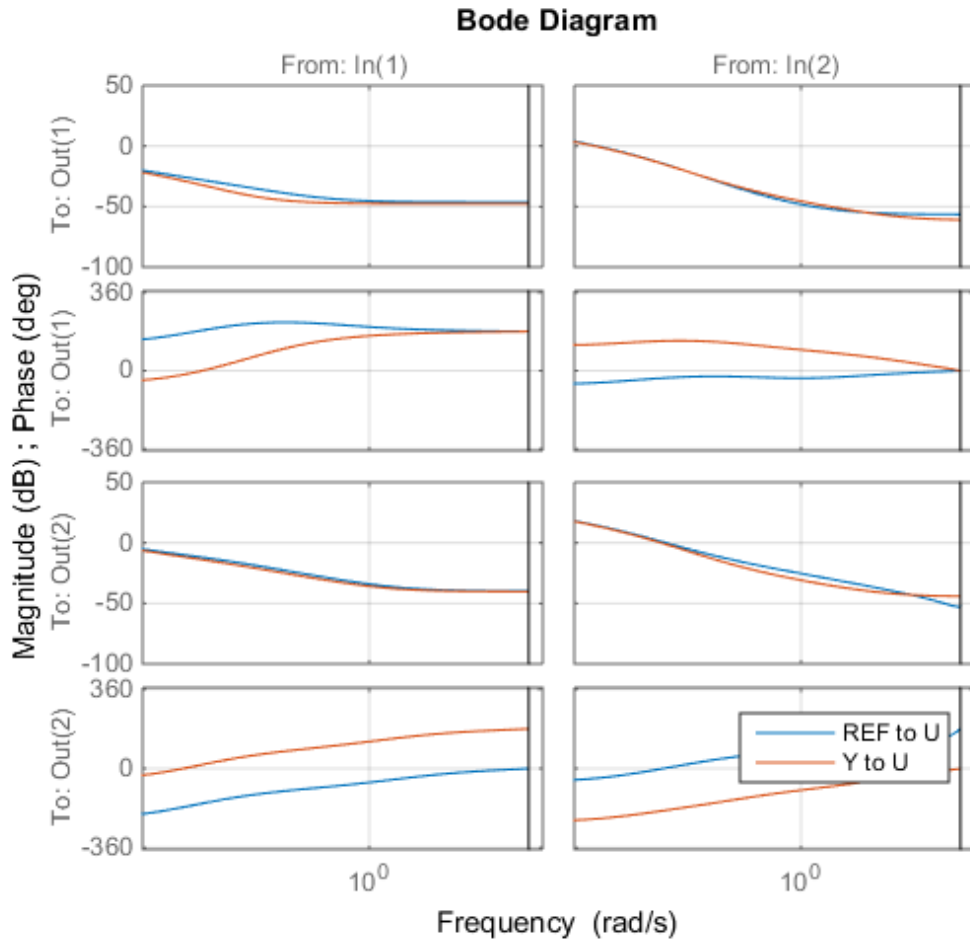


```
bodemag(C(:,5:6)), grid
title('Bode response from FUELMASS/SPEED to EGRLIFT/VGTPOS')
```

Bode response from FUELMASS/SPEED to EGRLIFT/VGTPOS
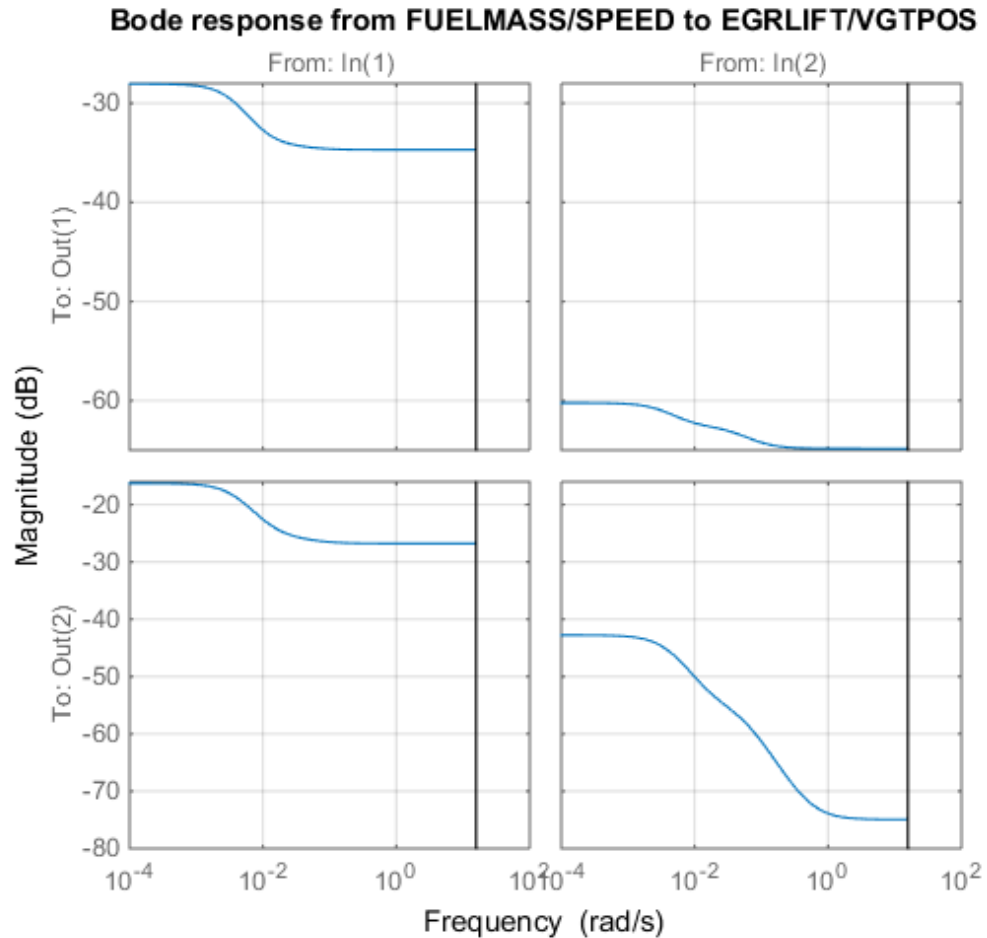
The first plot suggests that the controller essentially behaves like a PI controller acting on REF-Y (the difference between the target and actual values of the controlled variables). The second plot suggests that the transfer from measured disturbance to manipulated variables could be replaced by a gain in series with a lag network. Altogether this suggests the following simplified control structure consisting of a MIMO PI controller with a first-order disturbance feedforward.
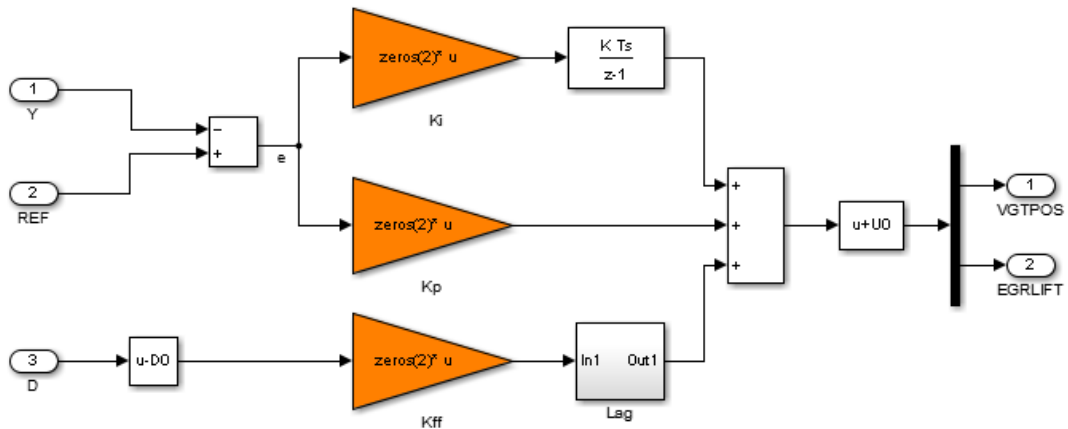
MIMO PID with disturbance feedforward

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. As before, 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 = slTuner('rct_diesel',{'Kp','Ki','Kff'});
ST0.Ts = 0.2;
addPoint(ST0,{'EGRLIFT','VGTPOS','DIESEL ENGINE'})

% Tune MIMO PI controller.
[ST2,~,~,Info2] = systune(ST0,[TR DR],[M1 M2 RO]);

Final: Soft = 1.08, Hard = 0.99985, Iterations = 221
```

Again all requirements are nearly met. Plot the closed-loop responses and compare with the state-space design.

```
clf
```

```
T2 = getIOTransfer(ST2,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
T2 = diag([1/10 1/3 1 1]) * T2 * diag([10 3]);
subplot(211), step(T1(1:2,:),T2(1:2,:),15), title('Setpoint tracking')
legend('SS2','PI+FF')
subplot(212), step(T1(3:4,:),T2(3:4,:),15), title('Control effort')
```
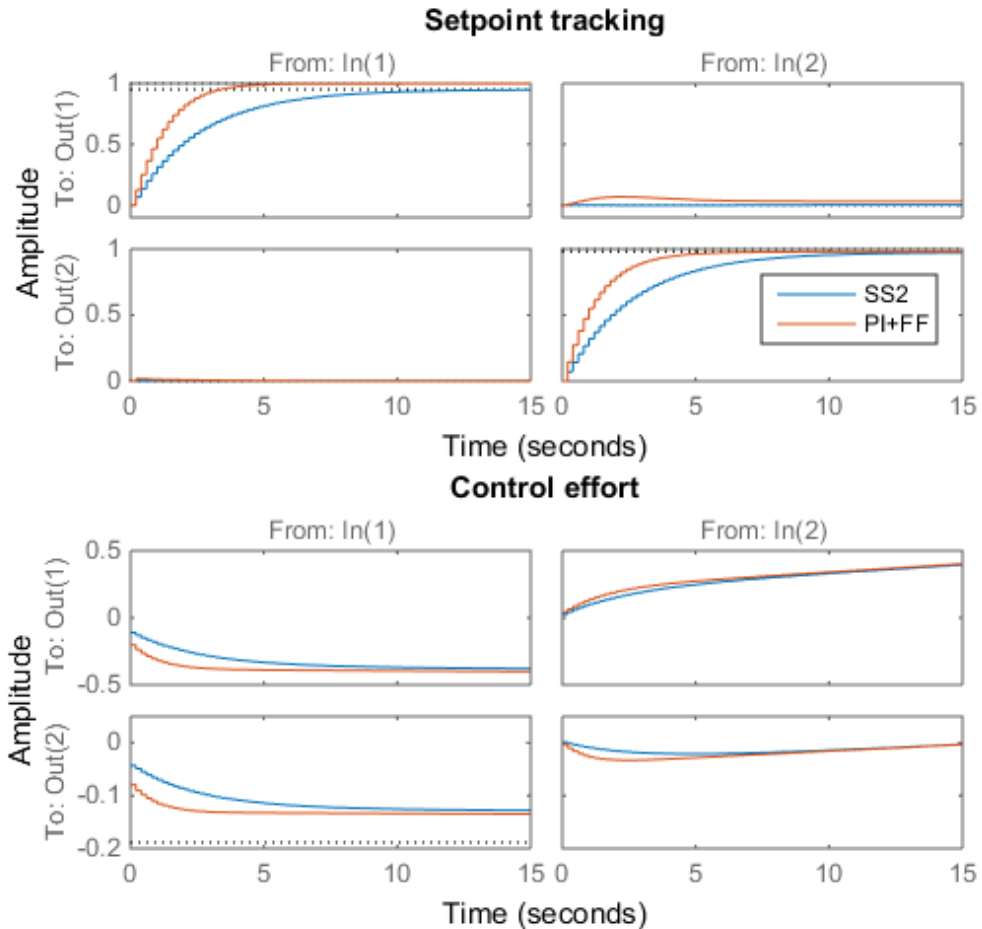


```
D2 = getIOTransfer(ST2,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
D2 = diag([1/10 1/3 1 1]) * D2 * diag([5 -200]);
subplot(211), step(D1(1:2,:),D2(1:2,:),15), title('Disturbance rejection')
```

```
legend('SS2','PI+FF')
subplot(212), step(D1(3:4,:),D2(3:4,:),15), title('Control effort')
```
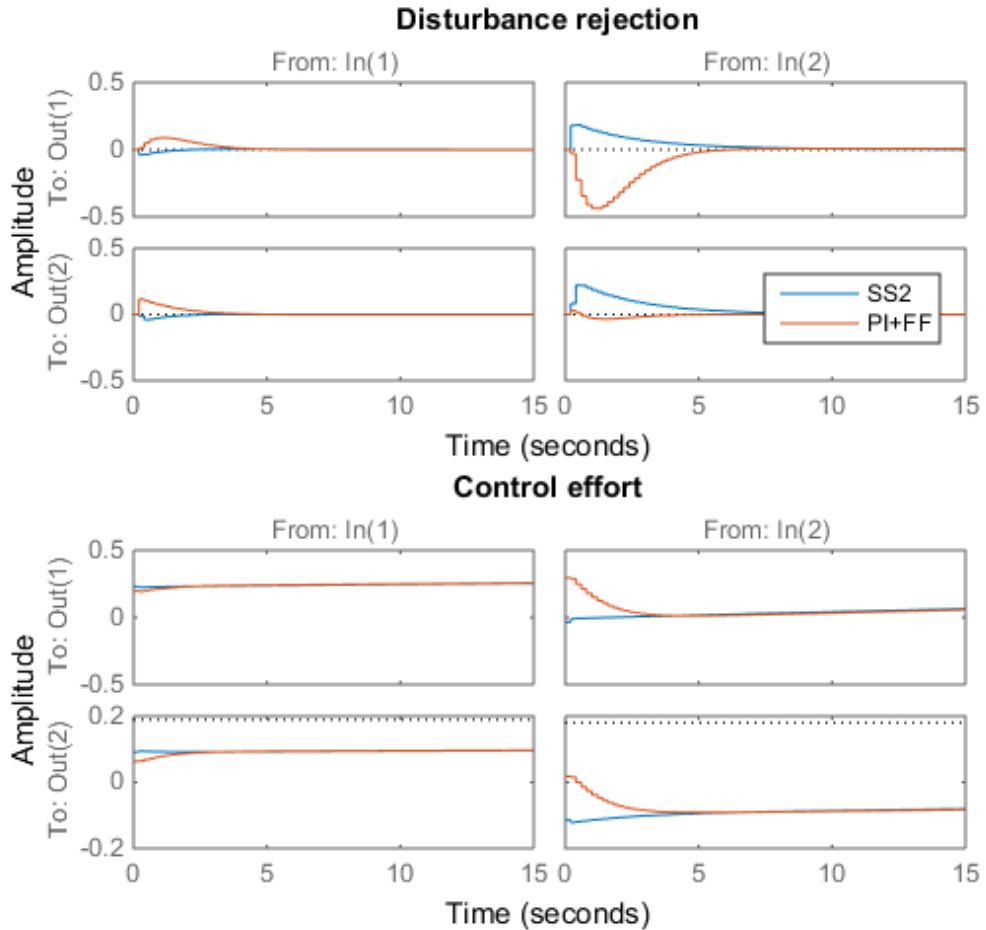

The blackbox and simplified control structures deliver similar performance. Inspect the tuned values of the PI and feedforward gains.

```
showTunable(ST2)
```

```
Block 1: rct_diesel/CONTROLLER/MIMO PID/Kp =

  d =
             u1          u2
   y1  -0.007916  -0.000869
   y2    -0.0202    0.01424

Name: Kp
Static gain.

----------------------------------

Block 2: rct_diesel/CONTROLLER/MIMO PID/Ki =

  d =
             u1          u2
   y1  -0.01052   -0.01425
   y2  -0.02998    0.04576

Name: Ki
Static gain.

----------------------------------

Block 3: rct_diesel/CONTROLLER/MIMO PID/Kff =

  d =
             u1          u2
   y1     0.01264  -8.985e-05
   y2     0.03922   -0.001477

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).
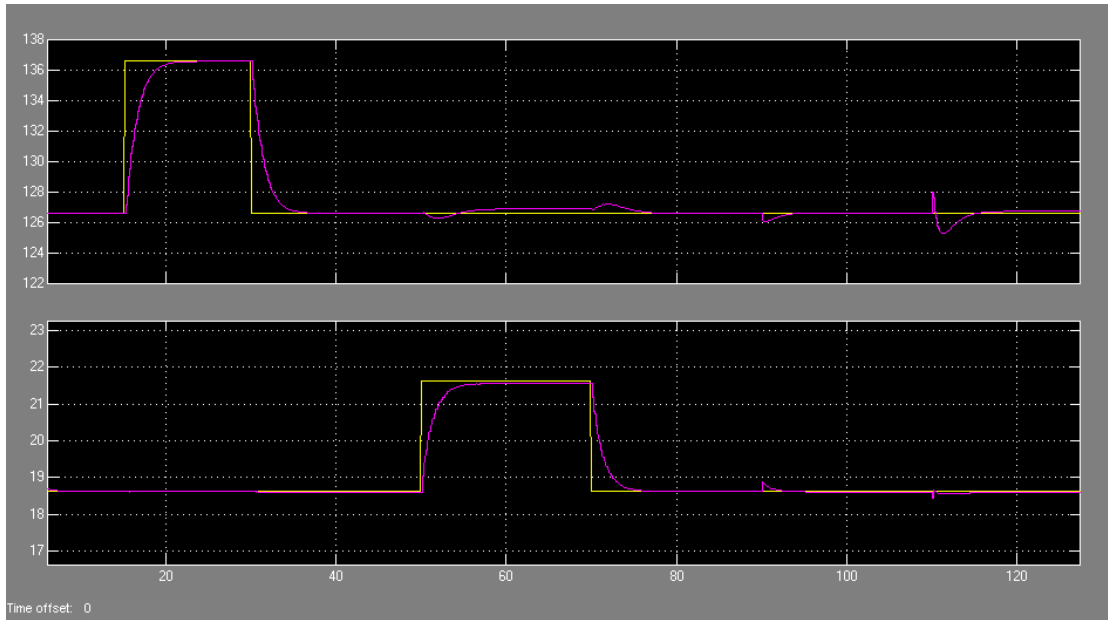
Figure 3: Simulation results with simplified controller.

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
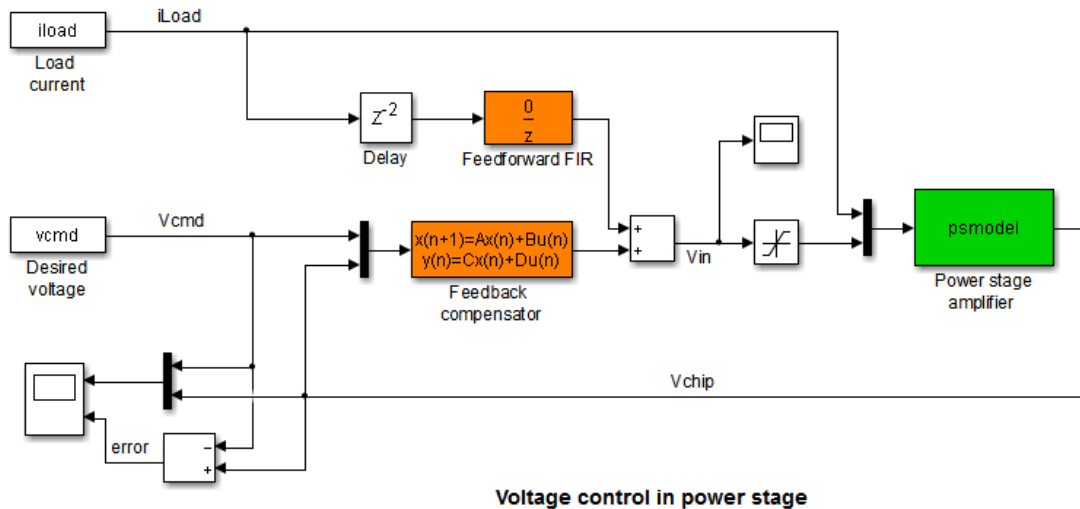
The power stage amplifier is modeled as a second-order linear system with the following frequency response:

```
bode(psmodel), grid
```
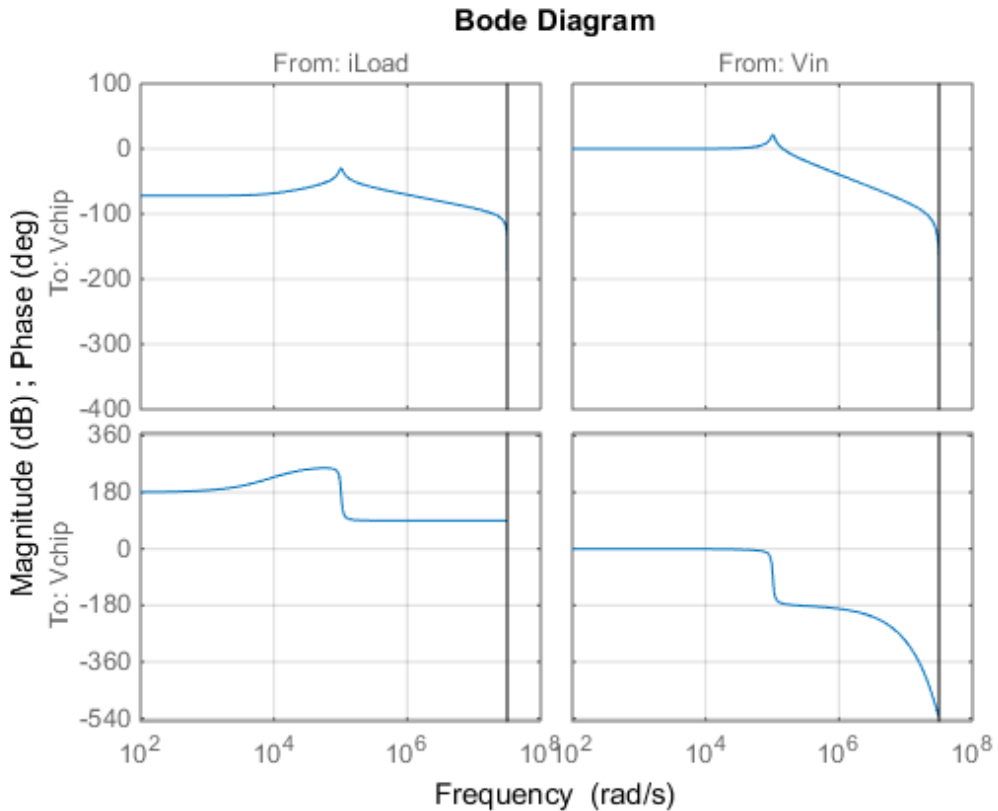
The controller must regulate the voltage Vchip delivered to the device to track the setpoint Vcmd and be insensitive to variations in load current iLoad. The control structure consists of a feedback compensator and a disturbance feedforward compensator. The voltage Vin going into the amplifier is limited to $V_{\max} = 12V$. The controller sampling rate is 10 MHz (sample time Tm is 1e-7 seconds).

Performance Requirements

This application is challenging because the controller bandwidth must approach the Nyquist frequency pi/Tm = 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 `Vcmd` in about 5 sampling periods with a peak error (across frequency) of 50%. Use a tracking requirement to capture this objective.

```
Req1 = TuningGoal.Tracking('Vcmd','Vchip',5*Tm,0,1.5);
Req1.Name = 'Setpoint change';
```

```
viewSpec(Req1)
```
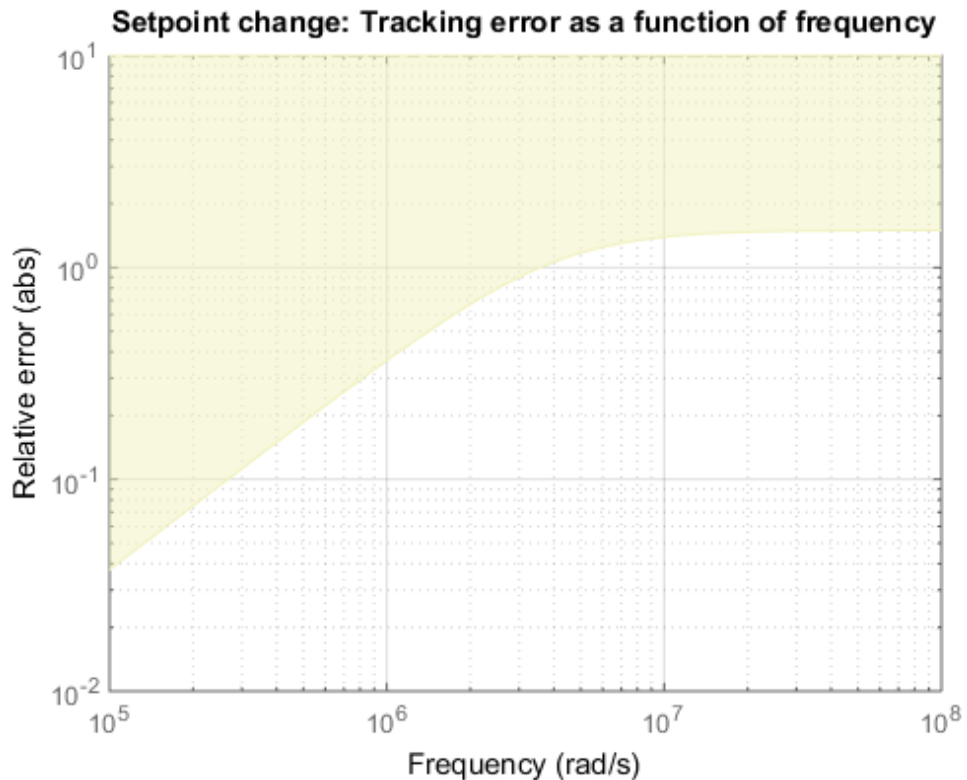

The power stage should also quickly reject load disturbances `iLoad`. Express this requirement in terms of gain from `iLoad` to `Vchip`. This gain should be low at low frequency for good disturbance rejection.

```
s = tf('s');
```

```
nf = pi/Tm;    % Nyquist frequency

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 vcmd 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(0,0.5,3*nf);
Req5.Name = 'Damping';
```

Tuning

Next use systune to tune the controller parameters subject to the requirements defined above. First use the slTuner 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 Tm.

```
TunedBlocks = {'compensator','FIR'};
STO = slTuner('rct_powerstage',TunedBlocks);
```

```
STO.Ts = Tm;

% Register points of interest for open- and closed-loop analysis
addPoint(STO,{'Vcmd','iLoad','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;
setBlockParam(STO,'FIR',FIR);
```

Note that slTuner 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 remaining requirements.

```
rng(0)
topt = systuneOptions('RandomStart',6);
ST = systune(STO,[Req1 Req2 Req3],[Req4 Req5],topt);

Final: Soft = 1.3, Hard = 0.99973, Iterations = 374
Final: Soft = 1.86, Hard = 0.9942, Iterations = 238
Final: Soft = 1.29, Hard = 0.99976, Iterations = 424
Final: Soft = 1.29, Hard = 0.99962, Iterations = 372
Final: Soft = 1.28, Hard = 0.98427, Iterations = 489
Final: Soft = 1.28, Hard = 0.97883, Iterations = 403
Final: Soft = 1.29, Hard = 0.97732, Iterations = 302
```

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.

```
figure('Position',[10,10,1071,714])
viewSpec([Req1 Req2 Req3 Req4 Req5],ST)
```

7-209
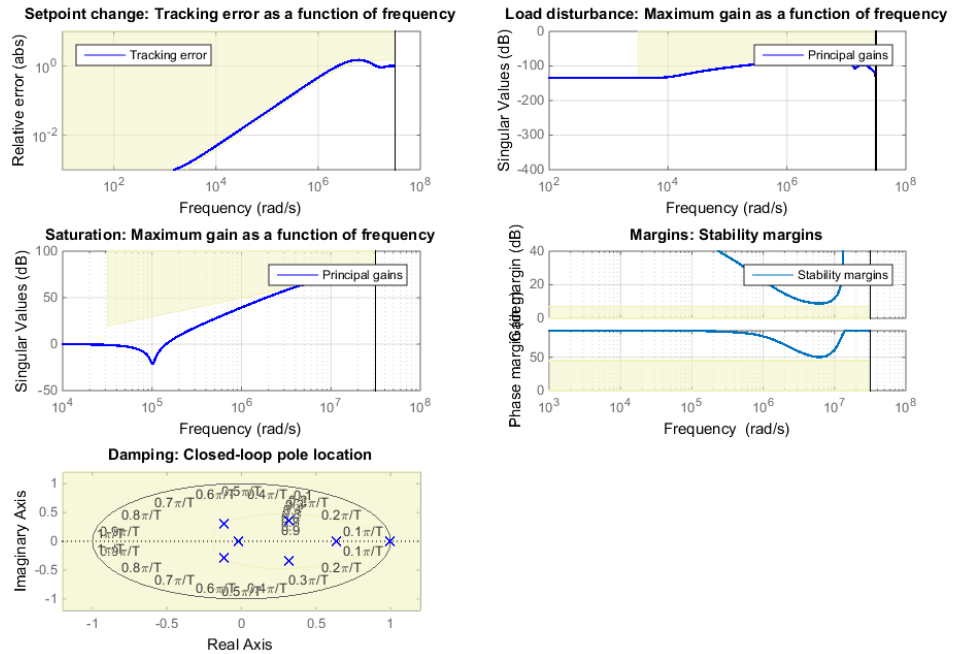
Validation

First validate the design in the linear domain using the `slTuner` interface. Plot the closed-loop response to a step command `Vcmd` and a step disturbance `iLoad`.

```
figure('Position',[100,100,560,500])
subplot(211), step(getIOTransfer(ST,'Vcmd','Vchip'),20*Tm)
title('Response to step command in voltage')
subplot(212), step(getIOTransfer(ST,'iLoad','Vchip'),20*Tm)
title('Rejection of step disturbance in load current')
```

Response to step command in voltage

Rejection of step disturbance in load current
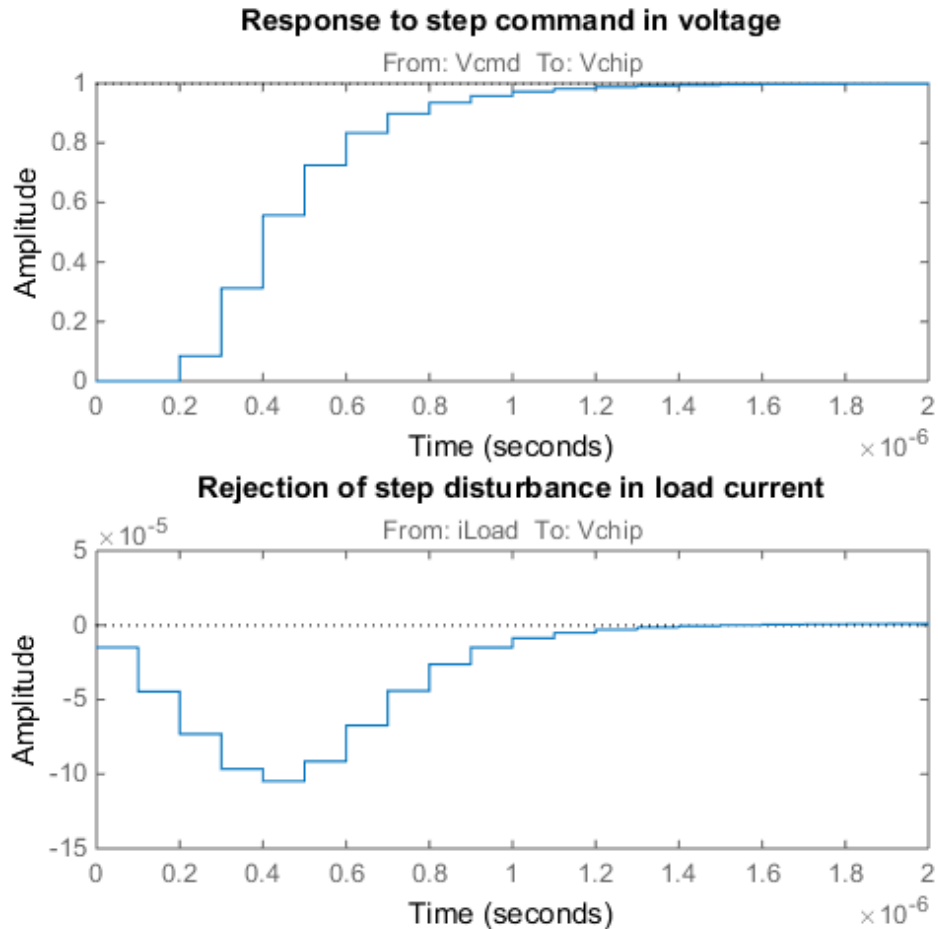
Use `getLoopTransfer` to compute the open-loop response at the plant input and superimpose the plant and feedback compensator responses.

```
clf
L = getLoopTransfer(ST,'Vin',-1);
C = getBlockValue(ST,'compensator');
bodeplot(L,psmodel(2),C(2),{1e-3/Tm pi/Tm}), grid
legend('Open-loop response','Plant','Compensator')
```
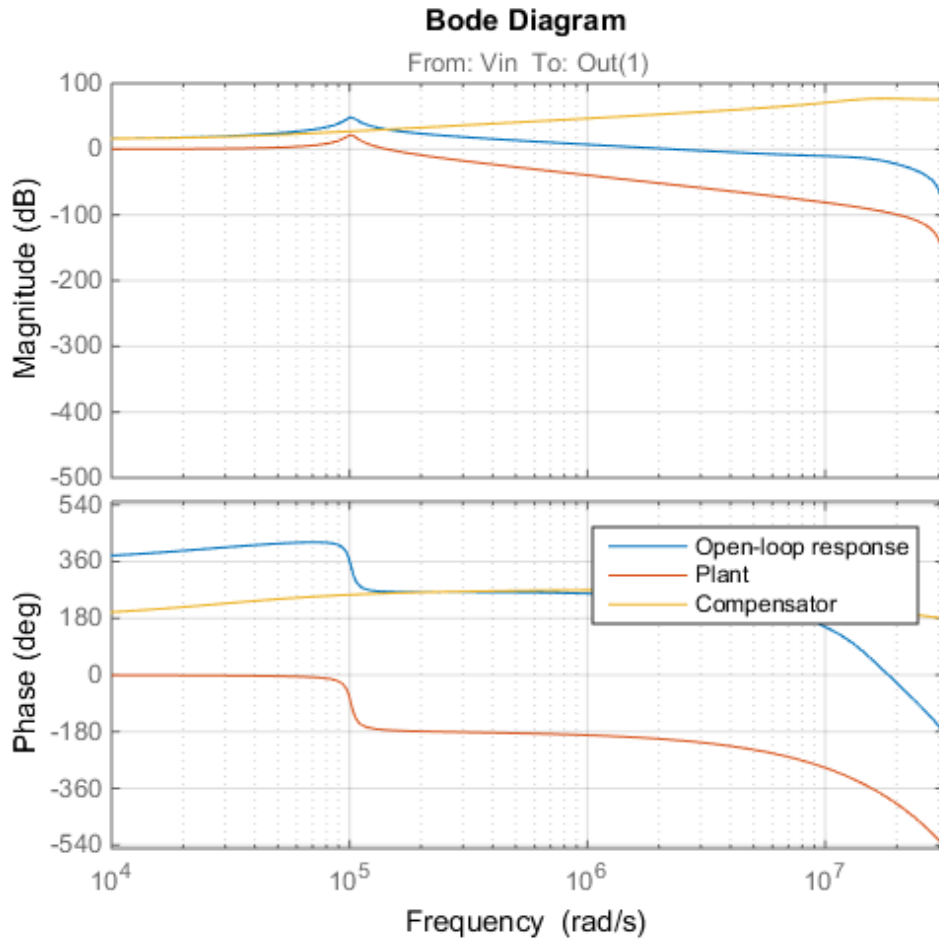
The controller achieves the desired bandwidth and the responses are fast enough. Apply the tuned parameter values to the Simulink model and simulate the tuned responses.

```
writeBlockValue(ST)
```

The results from the nonlinear simulation appear below. Note that the control signal Vin remains approximately within $\pm 12V$ saturation bounds for the setpoint tracking portion of the simulation.
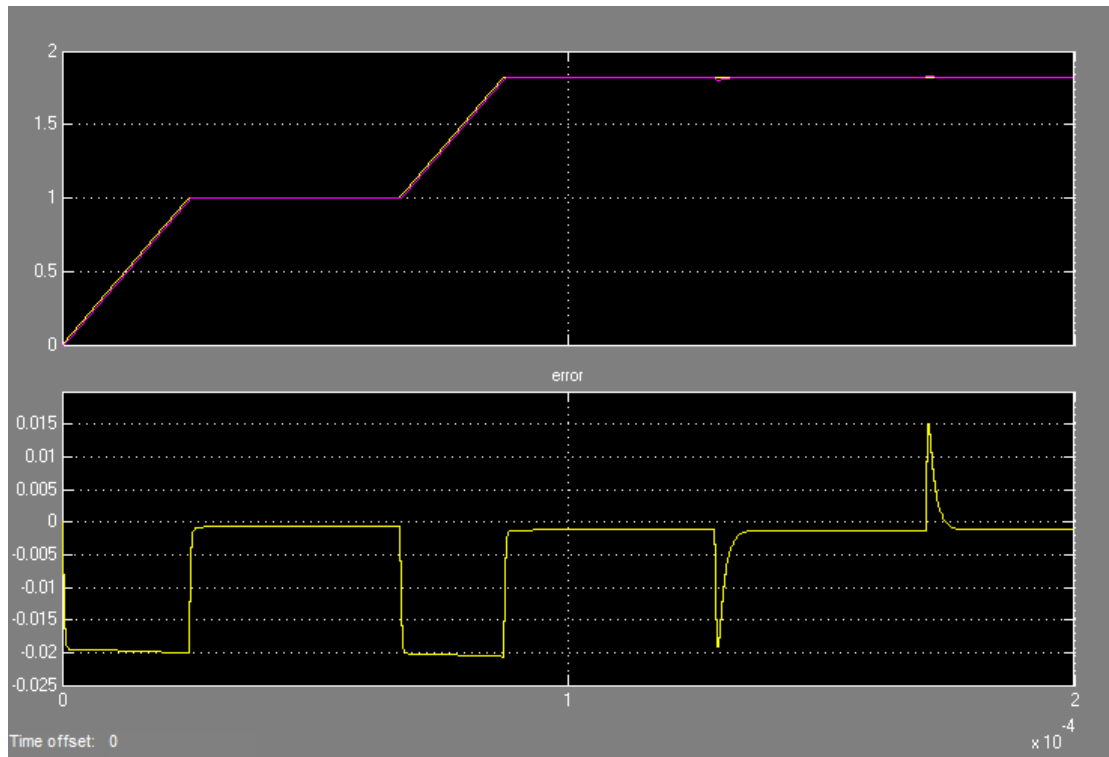
Figure 1: Response to ramp command and step load disturbances.
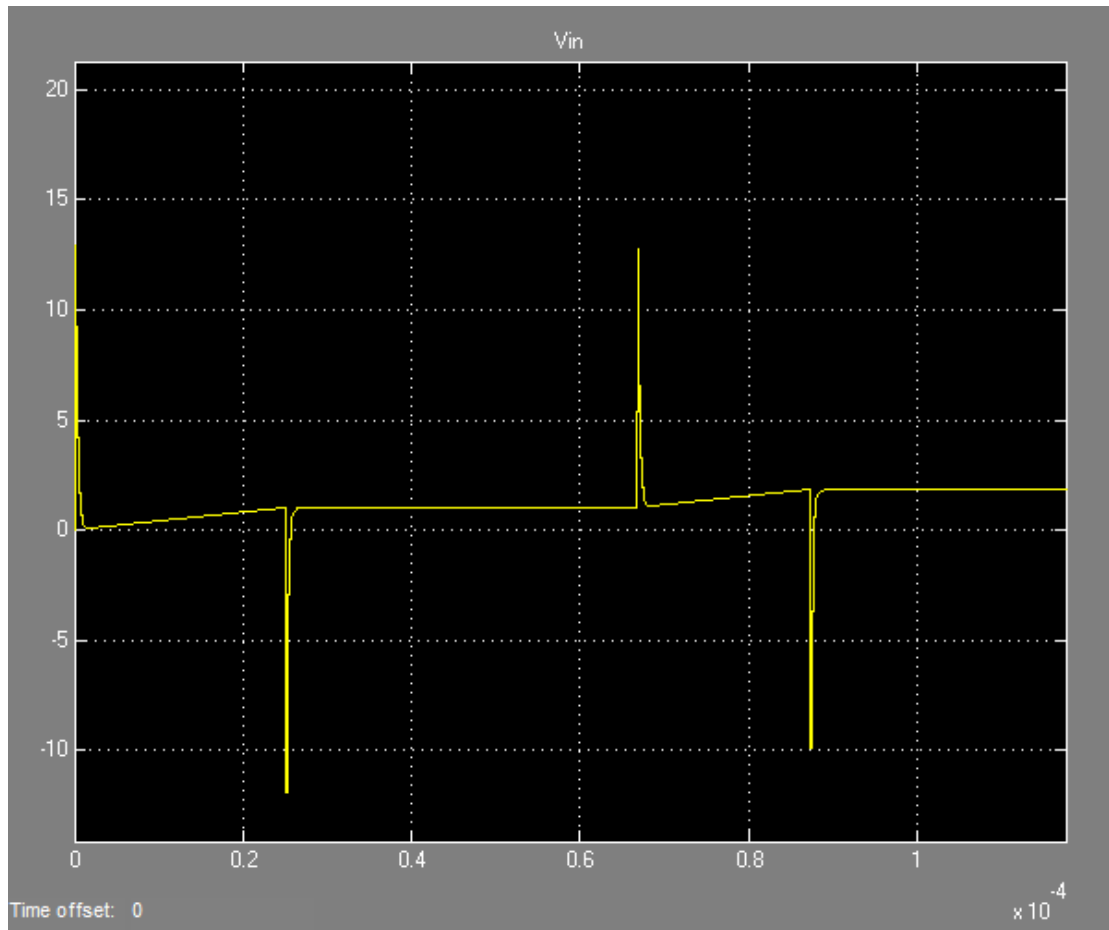
Figure 2: Amplitude of input voltage `Vin` during setpoint tracking phase.

8

Gain-Scheduled Controllers

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:

$$k(t) = k_0 + k_1 t + k_2 t^2 + k_3 t^3.$$

Here, $k_0,...,k_3$ 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 `systune` 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.

More About

Plant Models for Gain-Scheduled Control

In this section...

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, σ:

$$
\frac{dx}{dt} = A(\sigma)x + B(\sigma)u
$$
$$
y = C(\sigma)\dot{x} + D(\sigma)u.
$$

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, a, 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 σ values. This amounts to sampling the LPV dynamics over the operating range and selecting a set of representative design points.
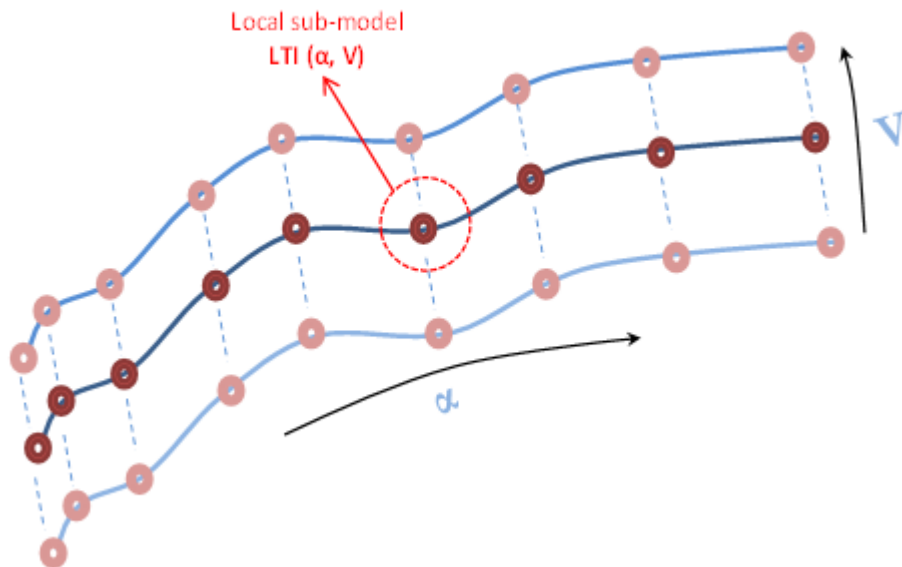
For best results with a gain-scheduled controller, the plant dynamics should vary smoothly between design points.

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, σ. For example, see "Study Parameter Variation by Sampling Tunable Model".

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:

$$\dot{x} = f(x, u)$$
$$y = g(x, u).$$

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 $(x(o),u(o))$ parameterized by the scheduling variables, o. 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:

$$\dot{\delta x} = A(\sigma)\delta x + B(\sigma)\delta u, \qquad \delta y = C(\sigma)\delta x + D(\sigma)\delta u,$$

$$A(\sigma) = \frac{\partial f}{\partial x}(x(\sigma),u(\sigma)) \qquad B(\sigma) = \frac{\partial f}{\partial u}(x(\sigma),u(\sigma))$$

$$C(\sigma) = \frac{\partial g}{\partial x}(x(\sigma),u(\sigma)) \qquad D(\sigma) = \frac{\partial g}{\partial u}(x(\sigma),u(\sigma)).$$

When repeated for a finite set of design points, o, this local linearization produces the type of sampled LPV model described in "Gain Scheduling for Linear Parameter-Varying Plants" on page 8-3.

When your nonlinear plant is modeled in Simulink, you can use Simulink Control Design linearization tools to compute $A(o)$, $B(o)$,$C(o)$, and $D(o)$ for specific values of o. If the operating points are equilibrium points, you might need to first use `findop` to compute $(x(o),u(o))$ as a function of o. If you are controlling the system around a reference trajectory $(x(o),u(o))$, you can use snapshot linearization to acquire $A(o)$, $B(o)$,$C(o)$, and $D(o)$ at various points along the o 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 o.

Typically, the gain-scheduled controller's main task is to keep the closed-loop system close to equilibrium or close to the nominal trajectory, $(x(o),u(o))$. The controller provides a corrective action δu which must be added to the nominal command $u(o)$ to determine the total actuator command.

More About

- "Gain-Scheduled Control Systems" on page 8-2

Parametric Gain Surfaces

In a gain-scheduled controller, the gains, which are the tunable parameters, are functions of the scheduling variables, σ. 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, σ, 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, σ. 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 σ, but restrict the generality of such functions by using specific basis function expansions. For example, suppose σ is a scalar variable. You can model $K_p(\sigma)$ as a quadratic function of σ:

$$K_p(\sigma) = k_0 + k_1\sigma + k_2\sigma^2.$$

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 σ):
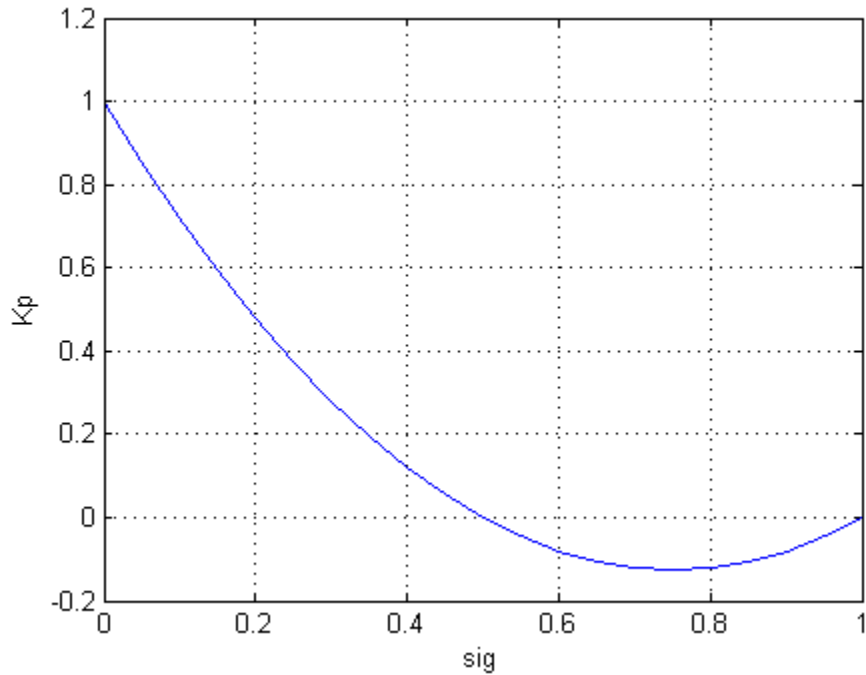
Or, suppose that σ consists of two scheduling variables, α, and V. Then, you can model $K_p(\sigma)$ as a bilinear function of α and V:

$$K_p\left(\alpha, V\right) = k_0 + k_1\alpha + k_2V + k_3\alpha V.$$

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 σ values):
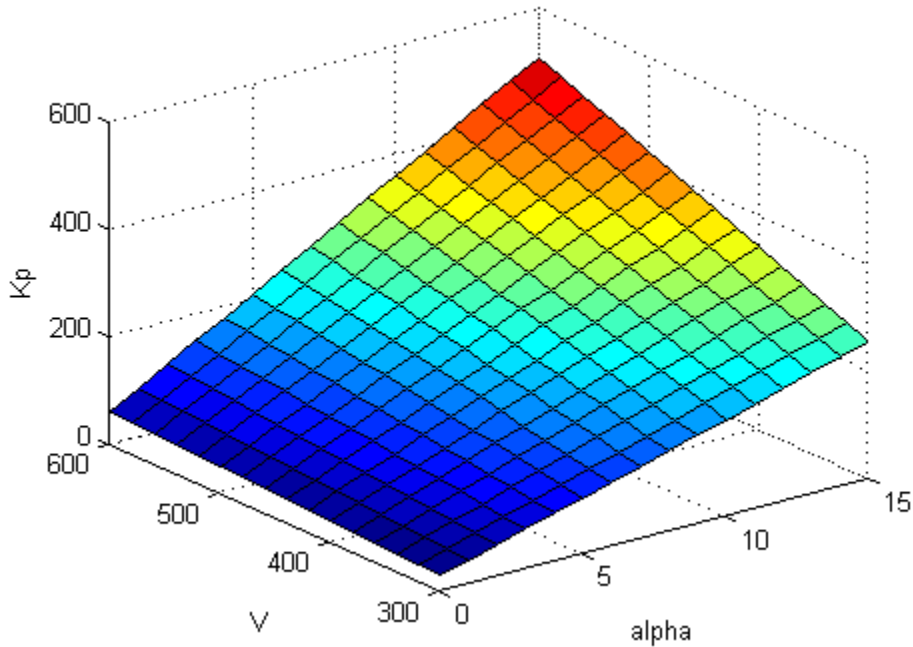
In general, a *parametric gain surface* is a particular expansion of the gain in basis functions of σ:

$$K(\sigma) = K_0 + K_1 f_1(\sigma) + \ldots + K_M f_M(\sigma).$$

The basis functions $f_1,\ldots,f_M$ are user-selected and fixed. The coefficients of the expansion, $K_0,\ldots,K_M$, are the tunable parameters of the gain surface. $K_0,\ldots,K_M$ 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 `systune`, use the `gainsurf` command to construct a tunable model of a gain surface sampled over a grid of design points (σ values). For example, consider the gain with bilinear dependence on two scheduling variables, α, and V:

$$K_p(\alpha, V) = k_0 + k_1 \alpha + k_2 V + k_3 \alpha V.$$

Suppose that a 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 $f_1 = a$, $f_2 = V$, $f_3 = aV$ over the grid of design points.

```
F1 = alpha;
F2 = V;
F3 = alpha.*V;

Kp = gainsurf('Kp',1,F1,F2,F3);
```

Kp is a tunable model of the gain surface, $K_p(a,V)$ 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 Kp and Ki sampled over the same σ 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 σ. 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 With Two Independent Scheduling Variables" on page 8-17
- "Gain-Scheduled PID Controller" on page 8-21

More About

Tuning Gain-Scheduled Controllers

You can use `systune` to automatically tune gain-scheduled controllers modeled with parametric gain surfaces. The tuning workflow is as follows:

1 Select a set of design points, *σ*, that adequately covers the operating range. The set can be a regular grid of *σ* 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, *σ* (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, *σ*.

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 `systune` to tune the gain surface coefficients subject to suitable requirements at each design point. From the `systune` 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 8-23
- Gain Scheduled Control Of a Chemical Reactor

More About
- "Plant Models for Gain-Scheduled Control" on page 8-3
- "Parametric Gain Surfaces" on page 8-7
- "Validating Gain-Scheduled Controllers" on page 8-13

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 σ 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 8-23
- Gain Scheduled Control Of a Chemical Reactor

More About

- "Improving Gain-Scheduled Tuning Results" on page 8-14

Improving Gain-Scheduled Tuning Results

In this section...
"Normalize the Scheduling Variables" on page 8-14
"Changing Requirements With Operating Condition" on page 8-15

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 $k(h)$:

$$k(h) = k_0 + k_1 h + k_2 h^2 + k_3 h^3.$$

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 With Two Independent Scheduling Variables" on page 8-17.

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, σ. Then you can construct one TuningGoal.LoopShape requirement for each design point. Associate each TuningGoal.LoopShape requirement with the corresponding design point using the Models 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, γ, 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 γ values for each design point, σ. Then you can add a new output ys = y/γ 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 TO
TO = connect(TO,yScaling,TO.InputName,[TO.OutputName ; {'ys'}]);
```

The value of γ 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 `slLinearizer` to change the variable's value with each linearization condition.

Tunable Gain With Two Independent Scheduling Variables

This example shows how to model a scalar gain *K* with a bilinear dependence on two scheduling variables, α and *V*, as follows:

$$K\left(\alpha, V\right) = K_0 + K_1\alpha + K_2 V + K_3 \alpha V.$$

For this example, α 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 $K_0, ..., K_3$ are the tunable parameters of this variable gain.

Create a grid of design points, (α, V), that are linearly spaced in α 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 `systune`, you might obtain better solver performance by normalizing the scheduling variables to fall within the interval [-1,1]. Scale the α 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 (α_N, V_N) values:

$$K\left(\alpha_N, V_N\right) = K_0 + K_1\alpha_N + K_2 V_N + K_3\alpha_N V_N.$$

In this expansion, the basis functions are:

$$\begin{aligned}
F_1\left(\alpha_N, V_N\right) &= \alpha_N \\
F_2\left(\alpha_N, V_N\right) &= V_N \\
F_3\left(\alpha_N, V_N\right) &= \alpha_N V_N.
\end{aligned}$$

Specify the values of the basis functions over the (α_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 blocks:
    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, and "K.Block
```

K is an array of generalized matrices. Each element in K corresponds to $K(\alpha_N, V_N)$ for a particular (α_N, V_N) pair, and depends on the tunable coefficients K_0,..., |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)
```
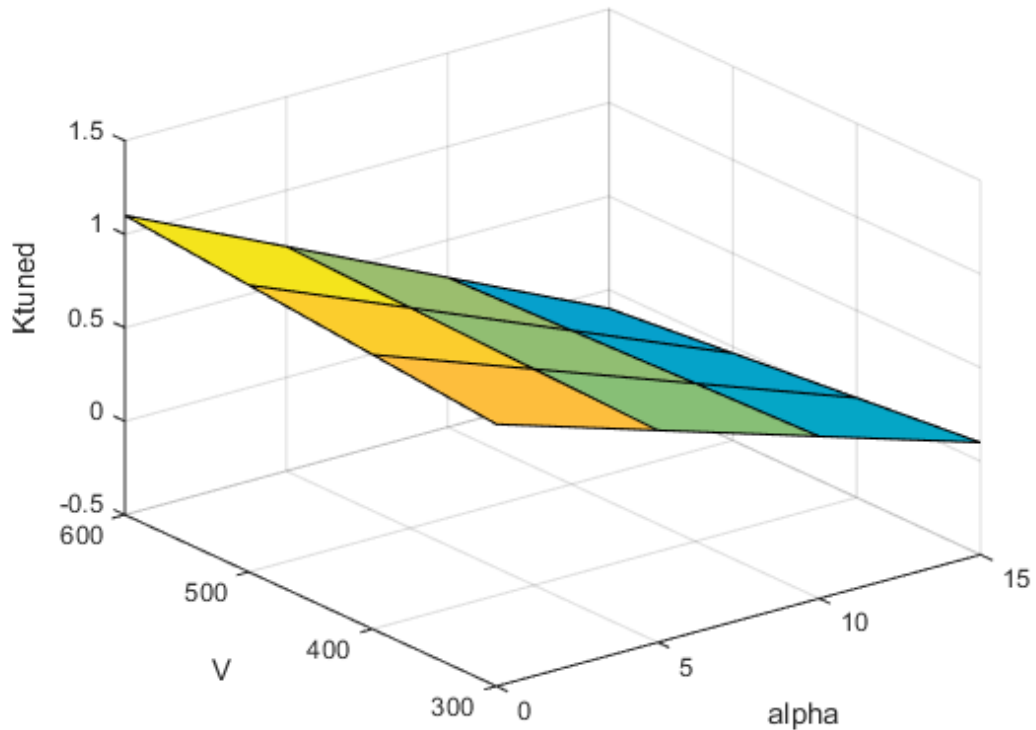
The variable names and values that you specified in the `SamplingGrid` property are used to scale and label the axes.

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 α and *V*. You can then

use `systune` 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 8-21
- "Tuning of Gain-Scheduled Three-Loop Autopilot" on page 8-23

More About

- "Parametric Gain Surfaces" on page 8-7
- "Plant Models for Gain-Scheduled Control" on page 8-3

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, α, as follows:

$$k_p\left(\alpha\right) = k_{p0} + k_{p1}\alpha$$
$$k_i\left(\alpha\right) = k_{i0} + k_{i1}\alpha + k_{i2}\alpha^2$$
$$k_p\left(\alpha\right) = k_{d0}.$$

For tuning purposes, sample these gain formulas for a number of values of α that span the operating range of the scheduling variable. For example, sample the gain formulas for five values of α (five design points) in [-1,1]. To do so, first create a vector of α values.

```
alpha = -1:0.5:1;
```

The PID gains depend on α and α^2. Evaluate these functions for the sampled values.

```
F1 = alpha;
F2 = alpha.^2;
```

Create the tunable gain surfaces that represent the proportional, integral, and derivative gains over the grid of α 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 Kp, Ki, and Kd is an array of generalized matrices. The array element Kp(:,:,1) (for example) corresponds to $k_p(\alpha)$ for α = -1. The element Kp(:,:,2) corresponds to $k_p(\alpha)$ for α = -0.5, and so on. These values depend on tunable coefficients Kp_0, and Kp_1. Thus, each of Kp, Ki, and Kd represents a gain curve in α 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 "C.Blocks" t
```

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
α.

Associate the α 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 α. You can then use
`systune` to tune the gain-scheduled controller to meet your design requirements.

Related Examples

- "Tunable Gain With Two Independent Scheduling Variables" on page 8-17
- "Tuning of Gain-Scheduled Three-Loop Autopilot" on page 8-23

More About

- "Parametric Gain Surfaces" on page 8-7
- "Plant Models for Gain-Scheduled Control" on page 8-3

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 θ, and the pitch rate $q = \dot{\theta}$. Figure 1 summarizes the relationship between the inertial and body frames, the flight path angle γ, the incidence angle α, and the pitch angle θ.
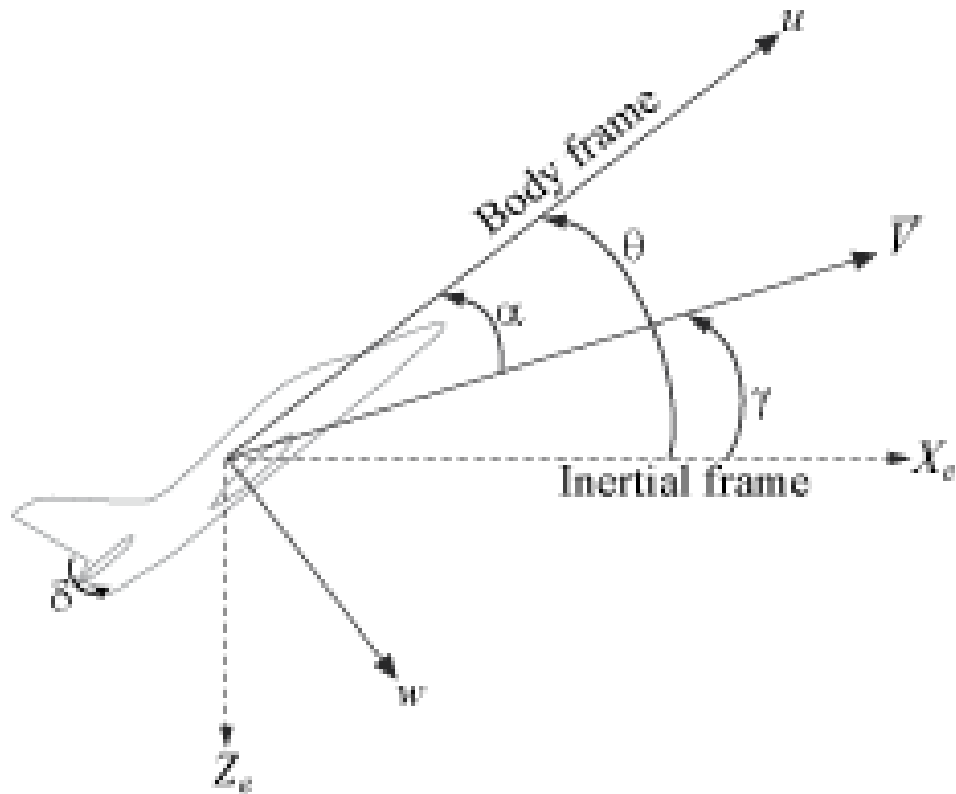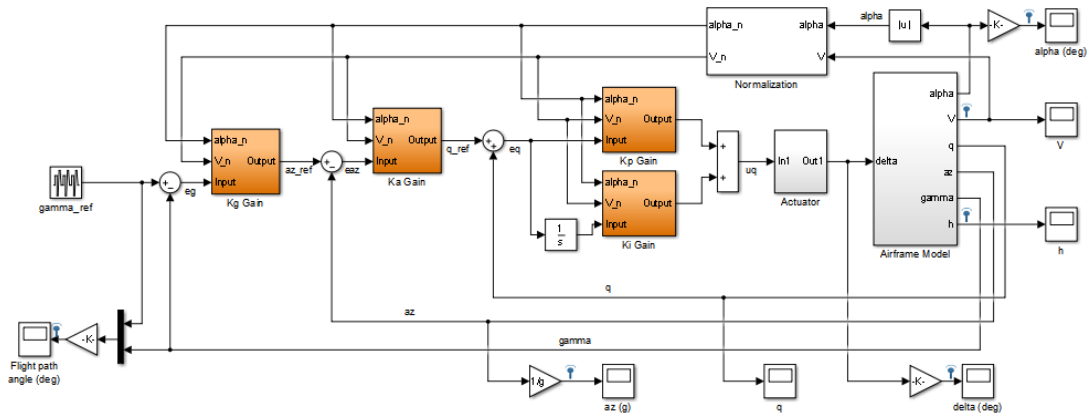
Figure 1: Airframe dynamics.

We use a classic three-loop autopilot structure to control the flight path angle γ. 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 δ 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 γ 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 α. To obtain suitable performance throughout the (α, V) flight envelope, the autopilot gains must be adjusted as a function of α and V to compensate for changes in plant dynamics. This adjustment process is called "gain scheduling" and α, 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 α, 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 α and V. Moreover, the gain schedule coefficients can be automatically tuned with systune.
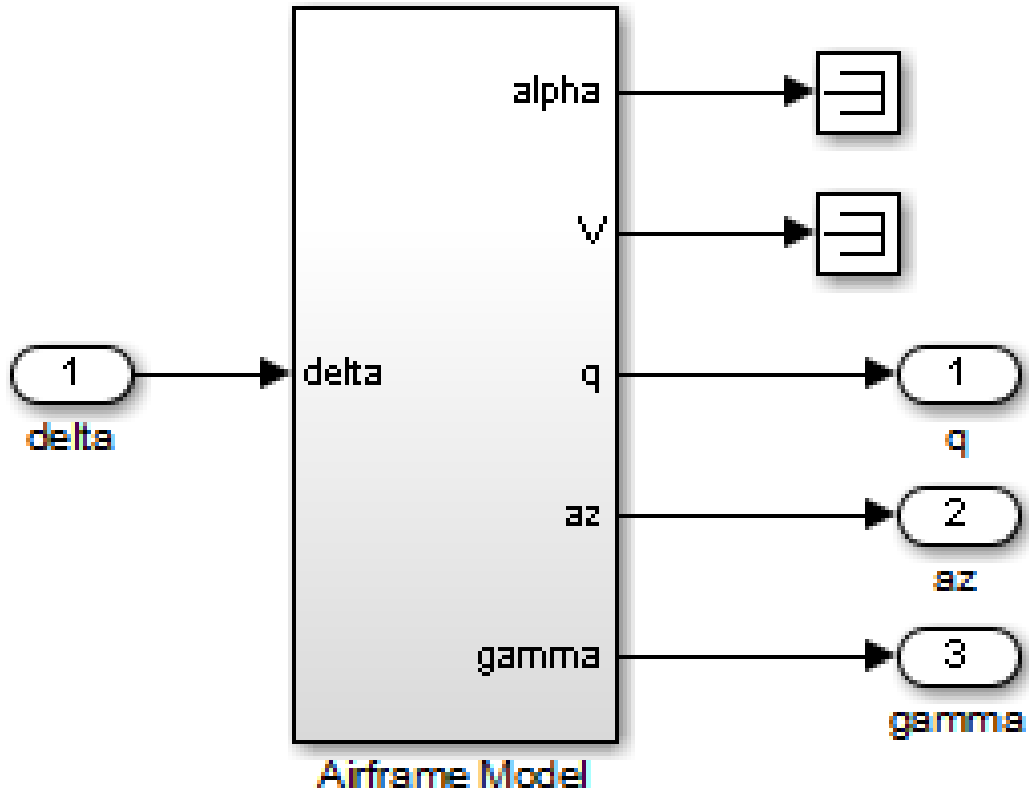
Trimming and Linearization

Assume that the incidence α 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 α so consider only positive values of α. Use a 5-by-9 grid of linearly spaced (α, 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 (α, V), linearize the airframe dynamics at trim (zero normal acceleration and pitching moment). This requires computing the elevator deflection δ 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 δ 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 (α, 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'};
```
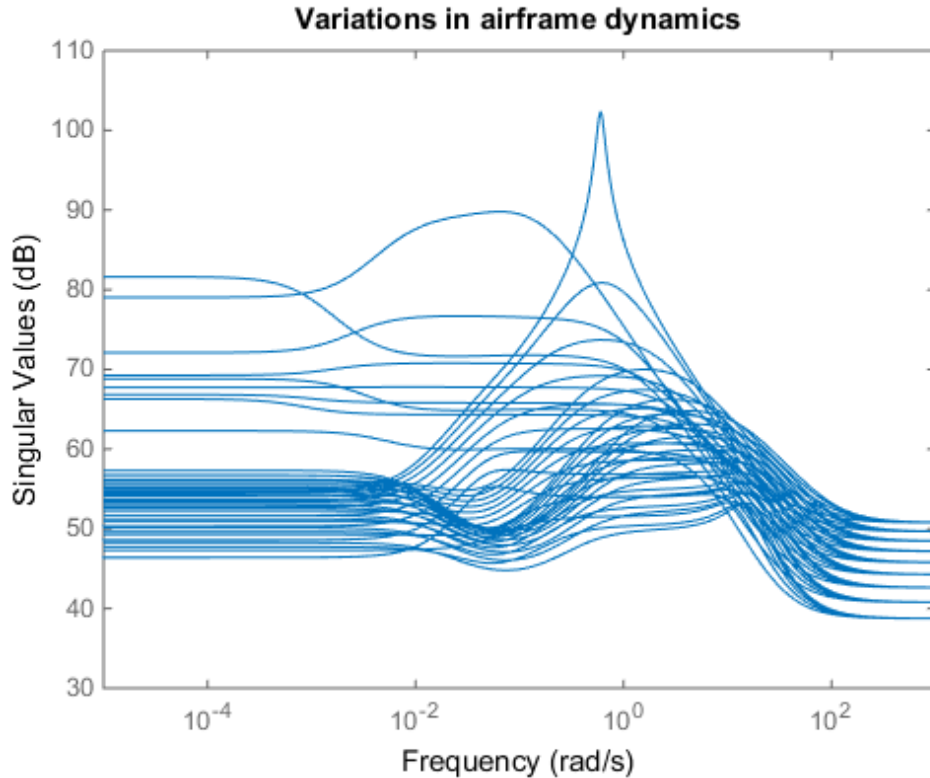
This produces a 5-by-9 array of linearized plant models at the 45 flight conditions (α, V). The plant dynamics vary substantially across the flight envelope.

```
sigma(G), title('Variations in airframe dynamics')
```

Autopilot Modeling

The autopilot consists of four gains that must be "scheduled" (adjusted) as a function of α 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 α and V:

$$K(\alpha, V) = K_0 + K_1\alpha + K_2V + K_3\alpha V.$$

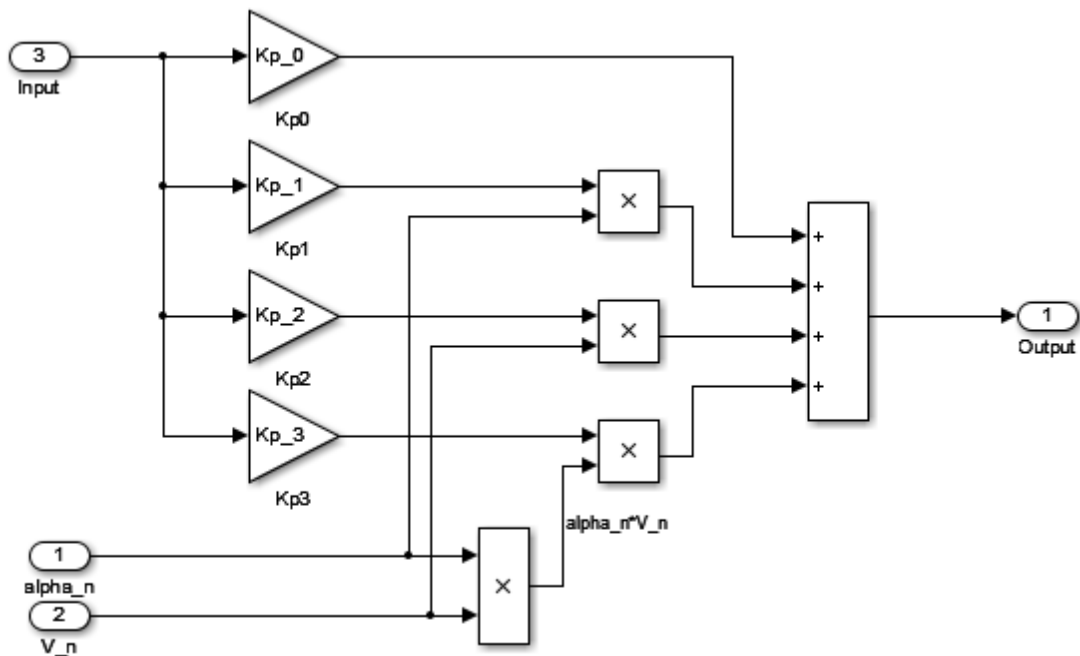These gain surfaces are modeled in Simulink as follows:

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 α, 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);
```

```matlab
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 α, V:

```matlab
% 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 TO 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).

```matlab
% 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
TO = connect(G,Cq,Ca,Cg,S1,S2,S3,S4,Act,...
   {'gamma_ref','az_ref','d'},{'gamma','az'})
```

```
TO =

  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(TO)" to see the current value, "get(TO)" to see all properties, and "TO.Blocks
```

This creates a 5-by-9 array of closed-loop models parameterized by the tunable coefficients of the four gain surfaces.

Autopilot Tuning

systune can automatically tune the gain surface coefficients for the entire flight envelope. Use TuningGoal objects to specify the performance objectives:

- γ loop: Track the setpoint with a 1 second response time, less than 2% steady-state error, and less than 30% peak error.

```
Req1 = TuningGoal.Tracking('gamma_ref','gamma',1,0.02,1.3);
viewSpec(Req1)
```

Requirement 1: Tracking error as a function of frequency
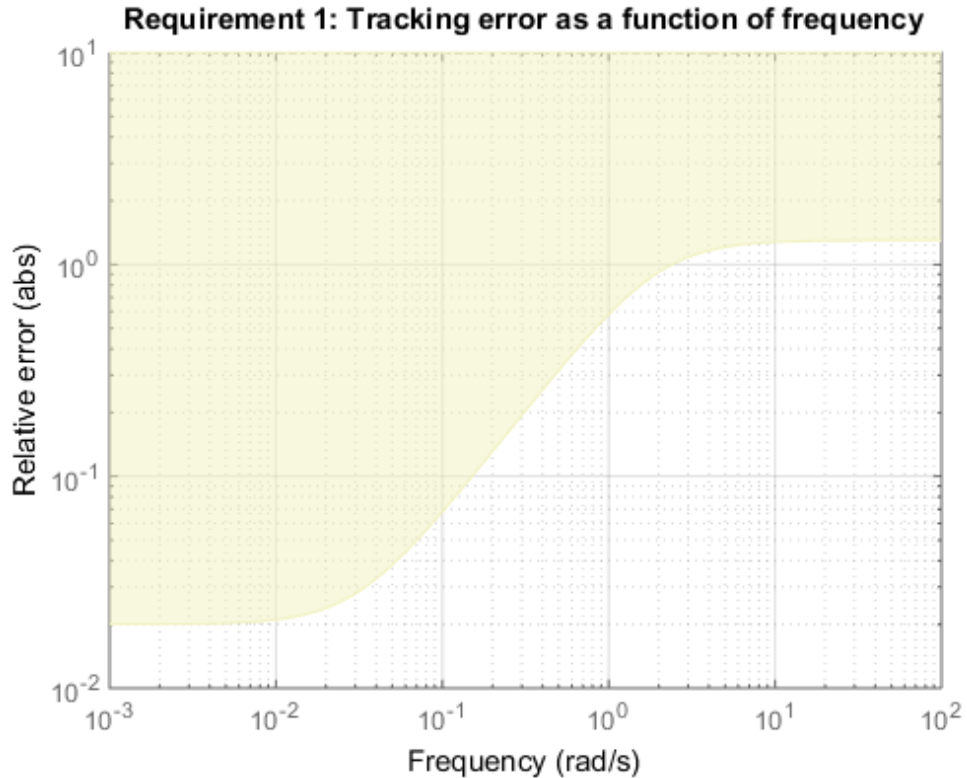
- a_z loop: Ensure good disturbance rejection at low frequency (to track acceleration demands) and past 10 rad/s (to be insensitive to measurement noise).

```
% 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)
```

Requirement 1: Maximum gain as a function of frequency
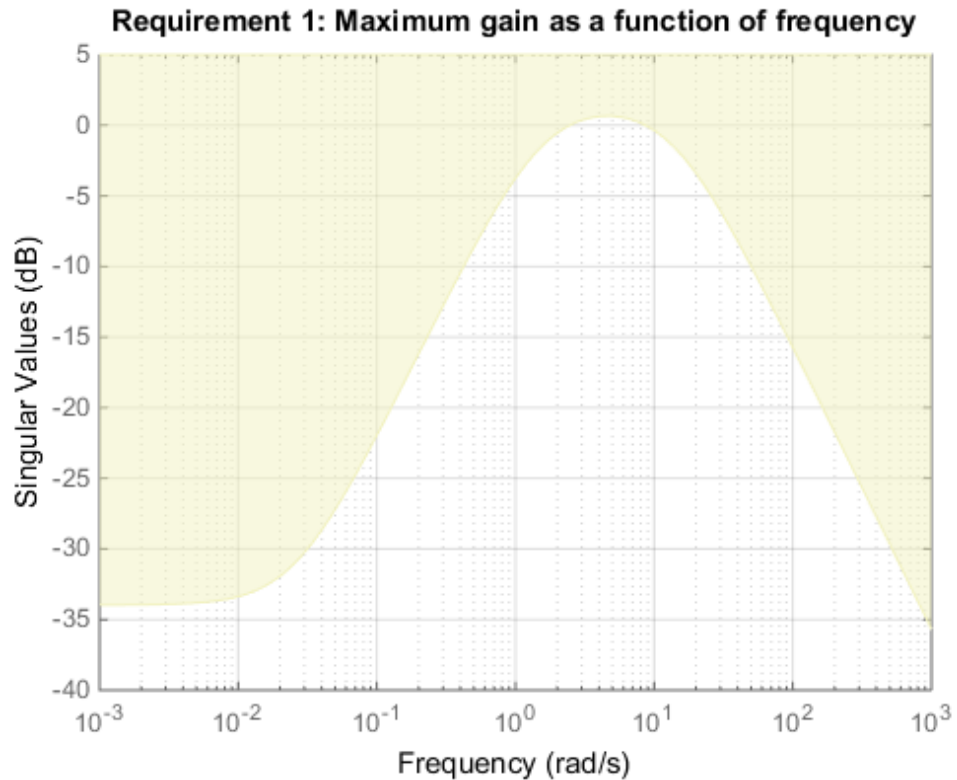
- q loop: Ensure good disturbance rejection up to 10 rad/s

```
% Note: The disturbance d enters at the plant input
Req3 = TuningGoal.Gain('d','az',600*tf([0.25 0],[0.25 1]));
viewSpec(Req3)
```

Requirement 1: Maximum gain as a function of frequency
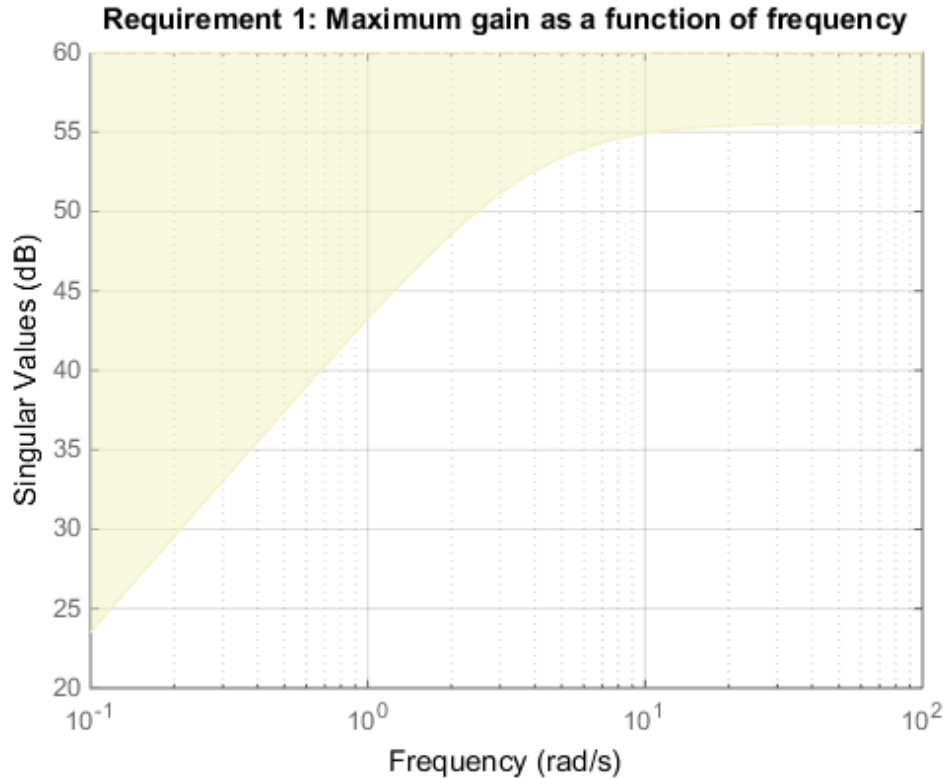
- Transients: Ensure a minimum damping ratio of 0.35 for oscillation-free transients

```
MinDamping = 0.35;
Req4 = TuningGoal.Poles(0,MinDamping);
```

Using `systune`, tune the 16 gain surface coefficients to best meet these performance requirements at all 45 flight conditions.

```
T = systune(T0,[Req1 Req2 Req3 Req4]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 78
```

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')
```
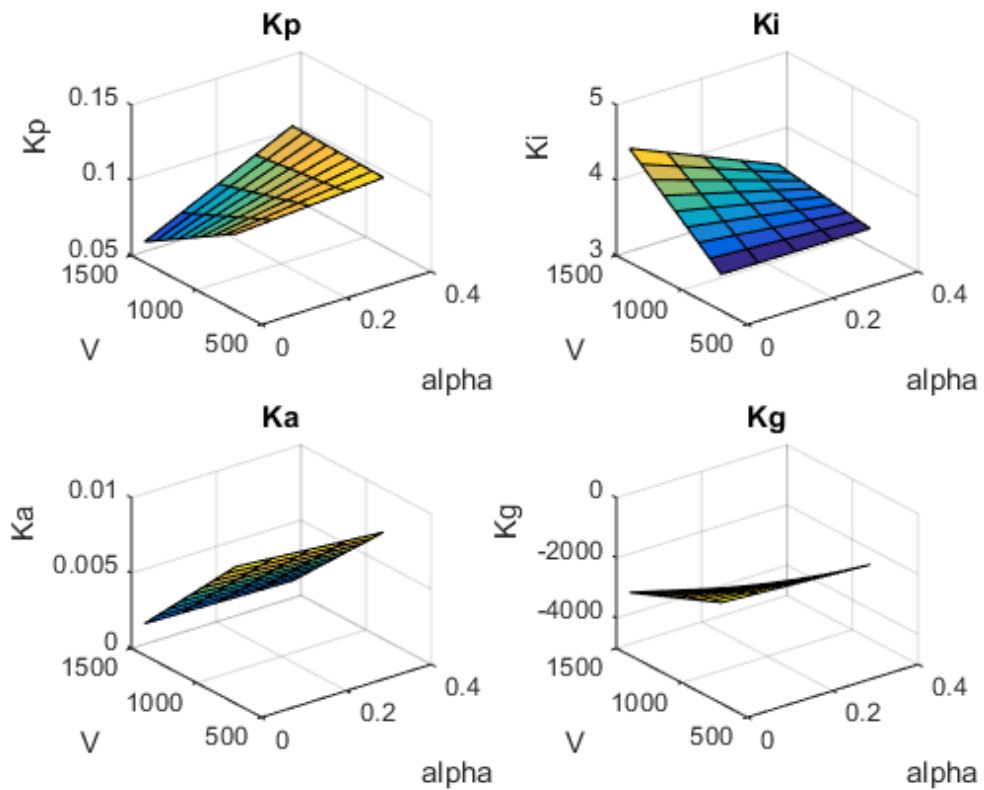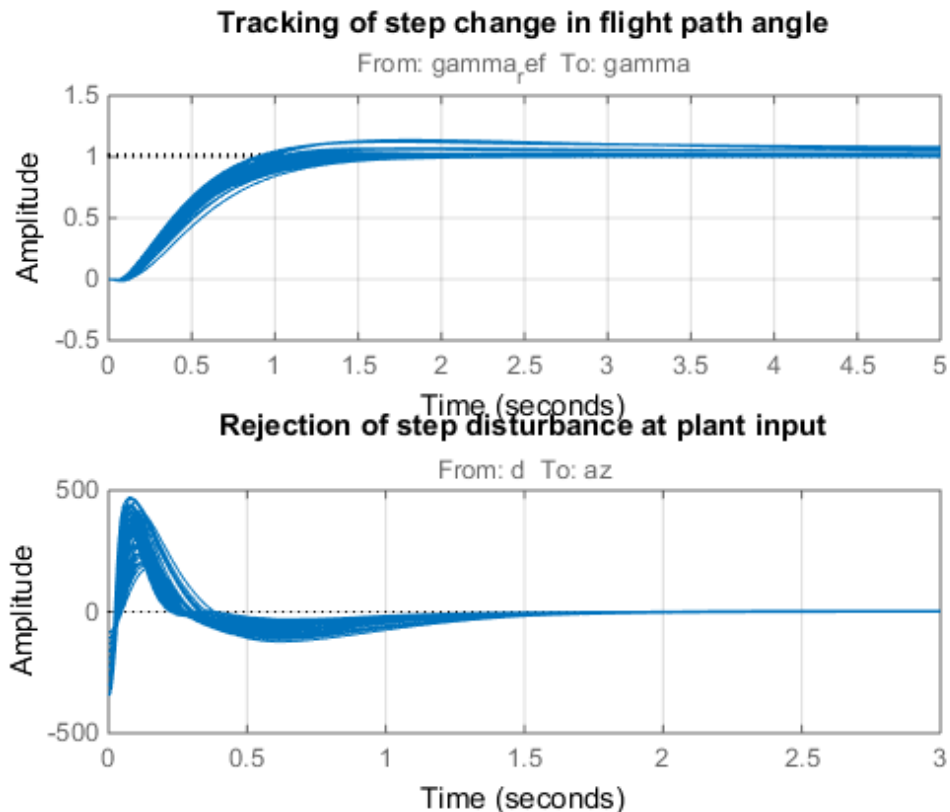

Validation

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] = gainsurfdata(Kp);
[Ki_0,Ki_1,Ki_2,Ki_3] = gainsurfdata(Ki);
[Ka_0,Ka_1,Ka_2,Ka_3] = gainsurfdata(Ka);
[Kg_0,Kg_1,Kg_2,Kg_3] = gainsurfdata(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'), grid
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')
```
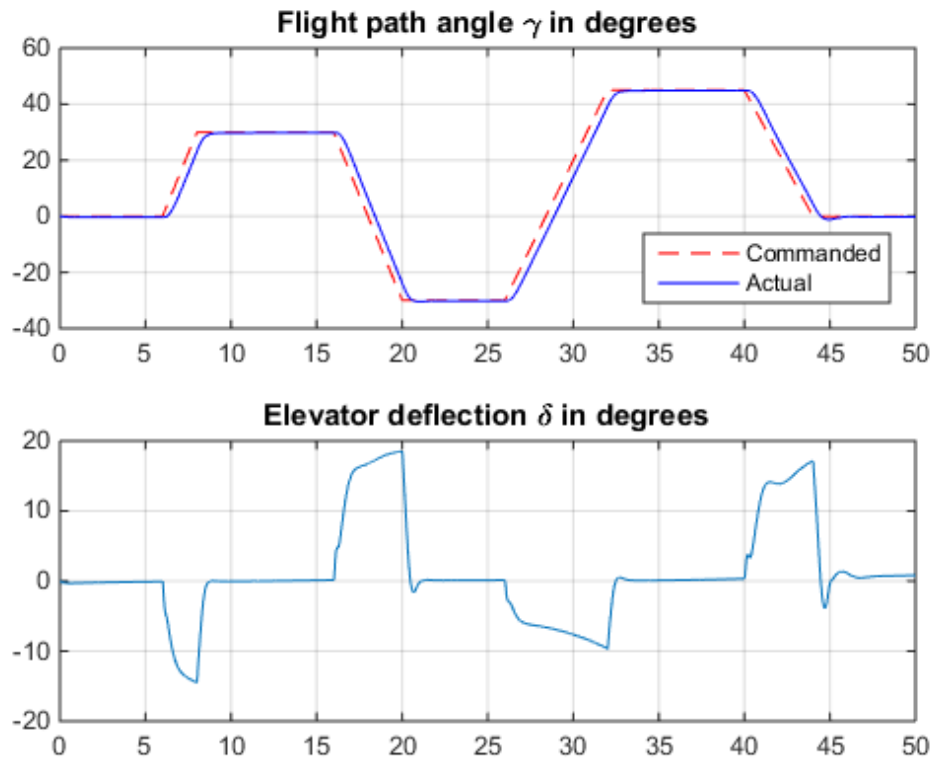
```
subplot(211)
plot(t,Sim_alpha.Values.Data), grid
title('Incidence \alpha in degrees')
subplot(212)
plot(t,Sim_V.Values.Data), grid
title('Speed V in m/s')
```
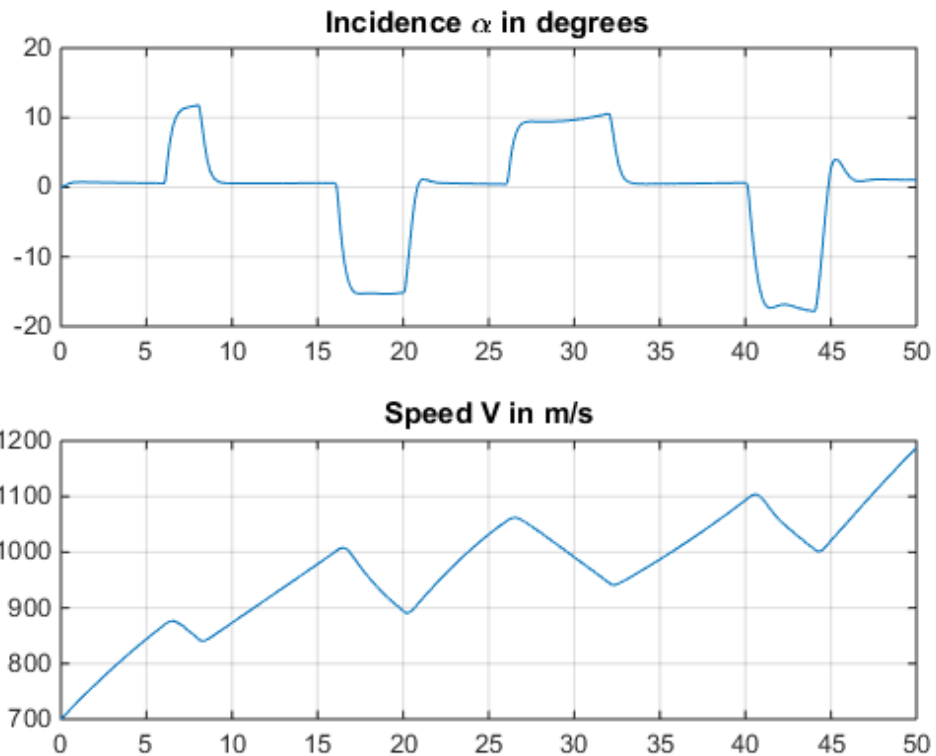
```
subplot(211)
plot(t,Sim_h.Values.Data), grid
title('Altitude h in meters')
subplot(212)
plot(t,Sim_az.Values.Data), grid
title('Normal acceleration a_z in g''s')
```

Altitude h in meters

Normal acceleration a_z in g's
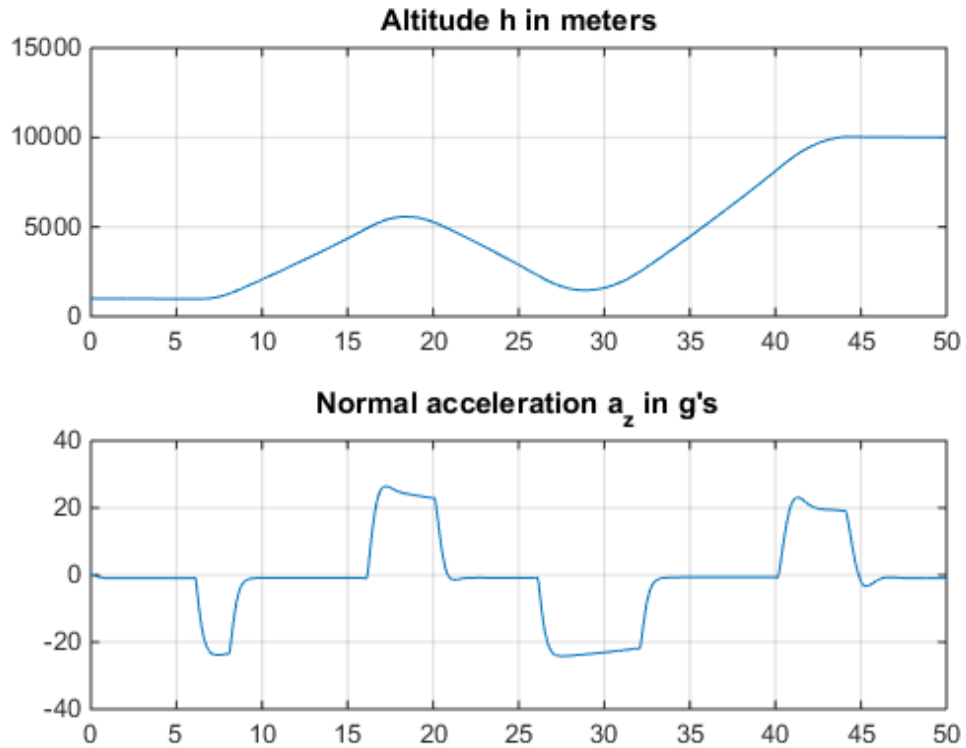
Tracking of the flight path angle profile remains good throughout the maneuver. Note that the variations in incidence α and speed V cover most of the flight envelope considered here ([-20,20] degrees for α and [700,1400] for V). And while the autopilot was tuned for a nominal altitude of 3000 m, it fares 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

More About

· "Gain-Scheduled Control Systems" on page 8-2
· "Parametric Gain Surfaces" on page 8-7