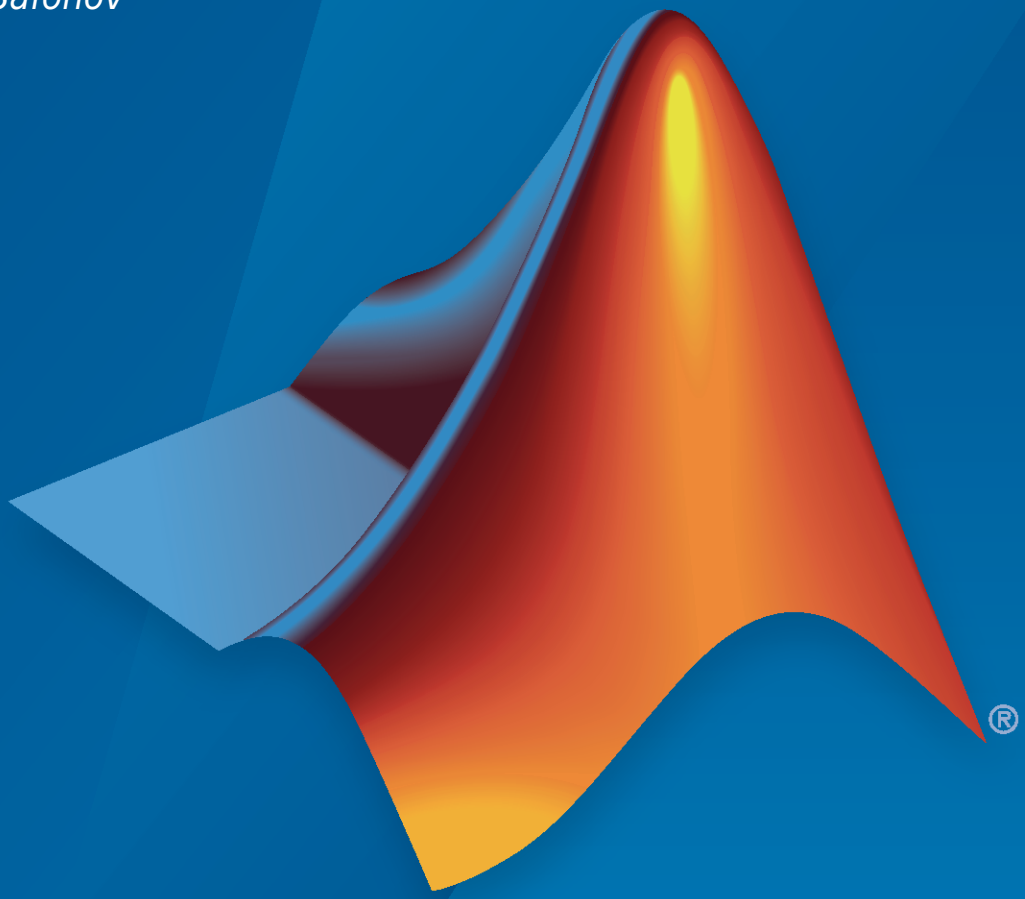
Robust Control Toolbox™

Getting Started Guide

Gary Balas
Richard Chiang
Andy Packard
Michael Safonov



MATLAB®

R2021b

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Robust Control Toolbox™ Getting Started Guide

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)
March 2015	Online only	Revised for Version 5.3 (Release 2015a)
September 2015	Online only	Revised for Version 6.0 (Release 2015b)
March 2016	Online only	Revised for Version 6.1 (Release 2016a)
September 2016	Online only	Revised for Version 6.2 (Release 2016b)
March 2017	Online only	Revised for Version 6.3 (Release 2017a)
September 2017	Online only	Revised for Version 6.4 (Release 2017b)
March 2018	Online only	Revised for Version 6.4.1 (Release 2018a)
September 2018	Online only	Revised for Version 6.5 (Release 2018b)
March 2019	Online only	Revised for Version 6.6 (Release 2019a)
September 2019	Online only	Revised for Version 6.7 (Release 2019b)
March 2020	Online only	Revised for Version 6.8 (Release 2020a)
September 2020	Online only	Revised for Version 6.9 (Release 2020b)
March 2021	Online only	Revised for Version 6.10 (Release 2021a)
September 2021	Online only	Revised for Version 6.11 (Release 2021b)

Contents

1 Introduction

2 Multivariable Loop Shaping

3 Model Reduction for Robust Control

4 Robustness Analysis

5 H-Infinity and Mu Synthesis

6 Robust Tuning

7 Tuning Fixed Control Architectures

Introduction

Robust Control Toolbox Product Description

Design robust controllers for uncertain plants

Robust Control Toolbox provides functions and blocks for analyzing and tuning control systems for performance and robustness in the presence of plant uncertainty. 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 for plant models with uncertainty. Controllers can include decentralized, fixed-structure controllers with multiple tunable blocks spanning multiple feedback loops.

Key Features

- Modeling of systems with uncertain parameters or neglected dynamics
- Worst-case stability and performance analysis
- Automatic tuning of SISO and MIMO control systems for uncertain plants
- 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

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.

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`.

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.

Summary of Robustness Analysis Tools

Function	Description
`ureal`	Create uncertain real parameter.
`ultidyn`	Create uncertain, linear, time-invariant dynamics.
`umargin`	Model uncertain gain and phase in a feedback loop.
`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.

Function	Description
`diskmargin`	Compute loop-at-a-time as well as MIMO gain and phase margins for a multiloop system, including the simultaneous gain/phase margins.
`robgain`	Robustness performance of uncertain systems.
`robstab`	Compute the robust stability margin of a nominally stable uncertain system.
`wcgain`	Compute the worst-case gain of a nominally stable uncertain system.
`wcdiskmargin`	Compute worst-case (over uncertainty) loop-at-a-time disk-based gain and phase margins.

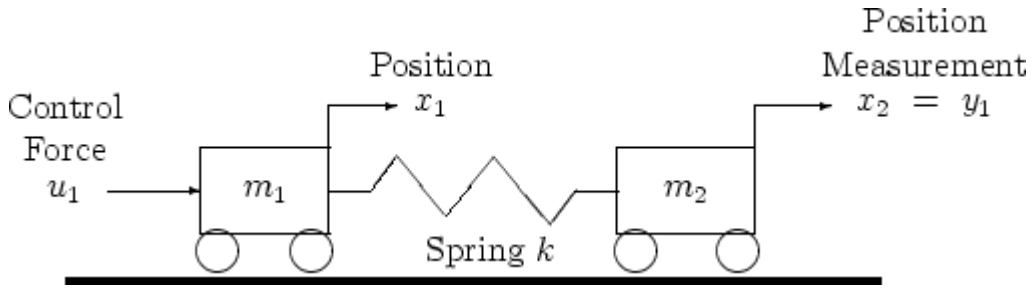
See Also

Related Examples

- “Create Models of Uncertain Systems” on page 4-2

System with Uncertain Parameters

As an example of a closed-loop system with uncertain parameters, consider the two-cart "ACC Benchmark" system [13] consisting of two frictionless carts connected by a spring shown as follows.



ACC Benchmark Problem

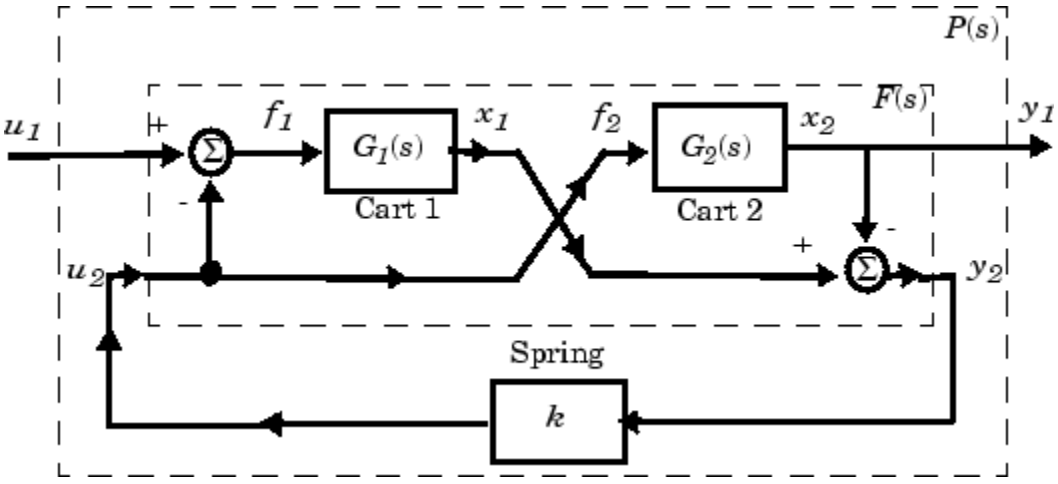
The system has the block diagram model shown below, where the individual carts have the respective transfer functions.

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

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

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);
```

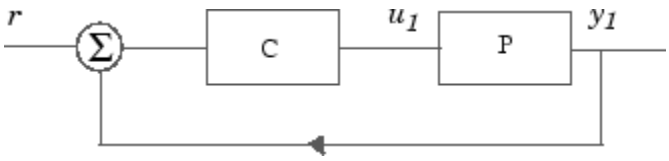
The variable P is a SISO uncertain state-space (USS) object with four states and three uncertain parameters, m1, m2, and k. You can recover the nominal plant with the command:

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

```
ans =

        1
  -------------
  s^2 (s^2 + 2)

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

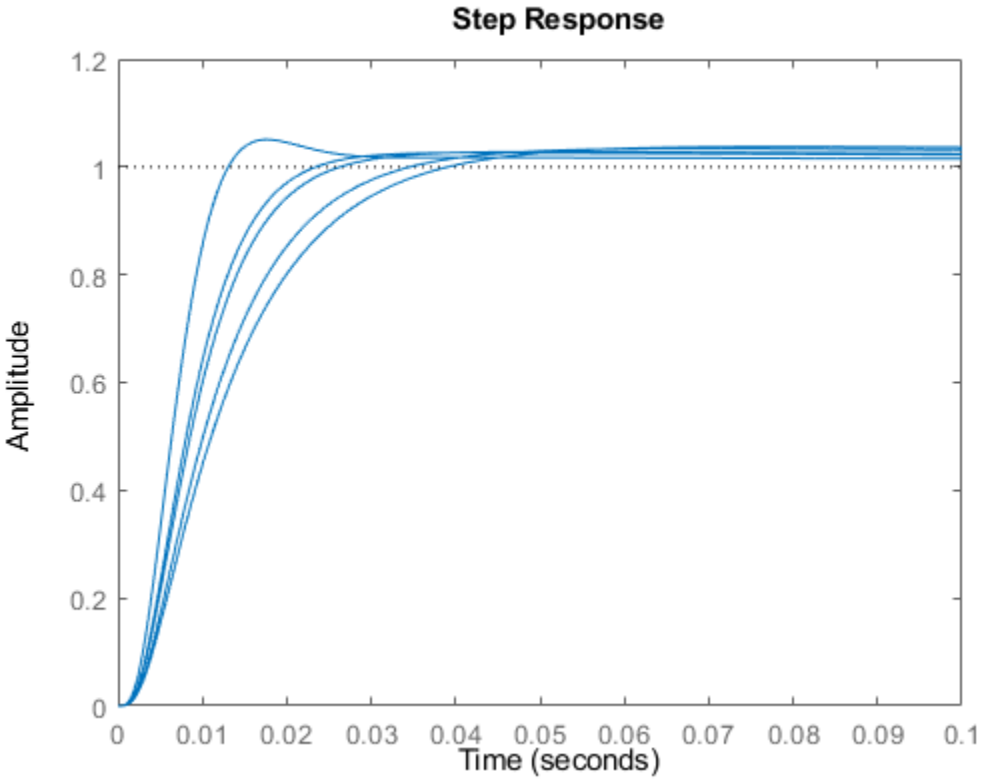
If the uncertain model P(s) has LTI negative feedback controller

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



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);
```



See Also

`ureal` | `uss`

Related Examples

- “Uncertain Real Parameters”
- “Uncertain LTI Dynamics Elements”

Building and Manipulating Uncertain Models

This example shows how to use Robust Control Toolbox™ to build uncertain state-space models and analyze the robustness of feedback control systems with uncertain elements.

We will show how to specify uncertain physical parameters and create uncertain state-space models from these parameters. You will see how to evaluate the effects of random and worst-case parameter variations using the functions `usample` and `robstab`.

Two-Cart and Spring System

In this example, we use the following system consisting of two frictionless carts connected by a spring `k`:

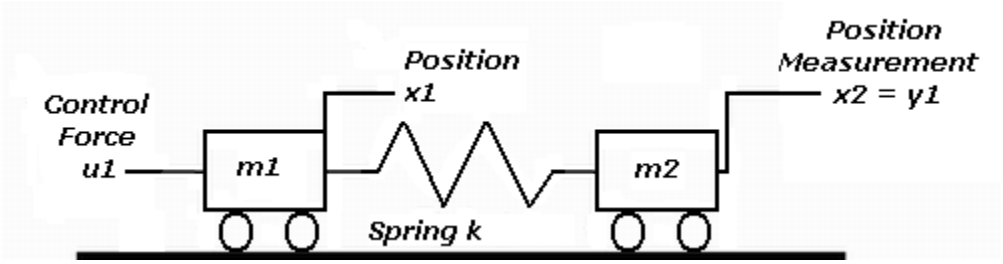


Figure 1: Two-cart and spring system.

The control input is the force `u1` applied to the left cart. The output to be controlled is the position `y1` of the right cart. The feedback control is of the following form:

$$u_1 = C(s)(r - y_1)$$

In addition, we use a triple-lead compensator:

$$C(s) = 100(s+1)^3/(0.001s+1)^3$$

We create this compensator using this code:

```
s = zpk('s'); % The Laplace 's' variable
C = 100*ss((s+1)/(.001*s+1))^3;
```

Block Diagram Model

The two-cart and spring system is modeled by the block diagram shown below.

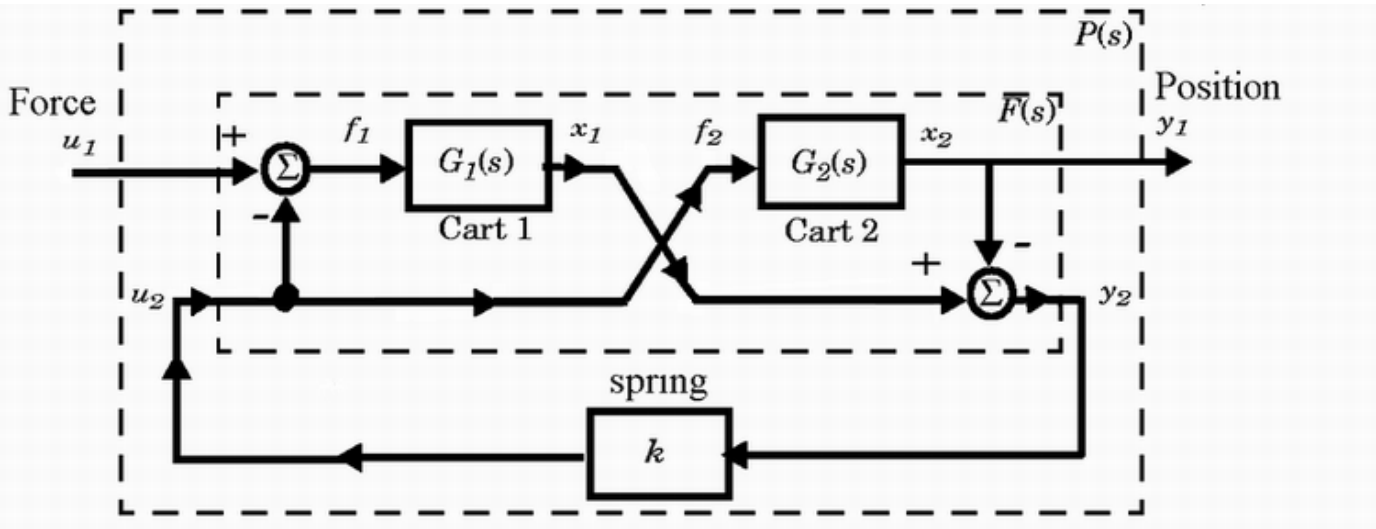


Figure 2: Block diagram of two-cart and spring model.

Uncertain Real Parameters

The problem of controlling the carts is complicated by the fact that the values of the spring constant `k` and cart masses `m1,m2` are known with only 20% accuracy: $k = 1.0 \pm 20\,\%$, $m1 = 1.0 \pm 20\,\%$, and $m2 = 1.0 \pm 20\,\%$. To capture this variability, we will create three uncertain real parameters using the `ureal` function:

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

Uncertain Cart Models

We can represent the carts models as follows:

$$G_1(s) = \frac{1}{m_1 s^2}, \quad G_2(s) = \frac{1}{m_2 s^2}$$

Given the uncertain parameters `m1` and `m2`, we will construct uncertain state-space models (USS) for G1 and G2 as follows:

```
G1 = 1/s^2/m1;
G2 = 1/s^2/m2;
```

Uncertain Model of a Closed-Loop System

First we'll construct a plant model `P` corresponding to the block diagram shown above (`P` maps u1 to y1):

```
% Spring-less inner block F(s)
F = [0;G1]*[1 -1]+[1;-1]*[0,G2]
```

```
F =

  Uncertain continuous-time state-space model with 2 outputs, 2 inputs, 4 states.
  The model uncertainty consists of the following blocks:
```

```
    m1: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m2: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences

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

Connect with the spring k

```
P = lft(F,k)
```

```
P =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 4 states.
  The model uncertainty consists of the following blocks:
    k: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m1: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m2: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences

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

The feedback control u1 = C*(r-y1) operates on the plant P as shown below:

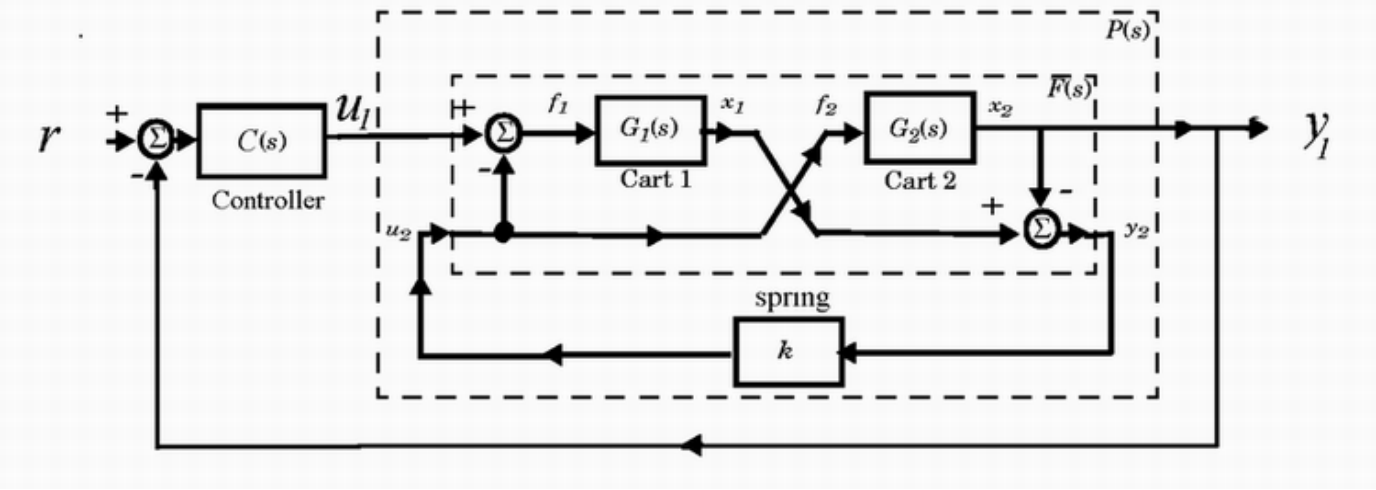


Figure 3: Uncertain model of a closed-loop system.

We'll use the `feedback` function to compute the closed-loop transfer from r to y1.

```
% Uncertain open-loop model is
L = P*C
```

```
L =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 7 states.
  The model uncertainty consists of the following blocks:
    k: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m1: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m2: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences

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

Uncertain closed-loop transfer from r to y1 is

```
T = feedback(L,1)
```

```
T =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 7 states.
  The model uncertainty consists of the following blocks:
    k: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m1: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    m2: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences

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

Note that since `G1` and `G2` are uncertain, both `P` and `T` are uncertain state-space models.

Extracting the Nominal Plant

The nominal transfer function of the plant is

```
Pnom = zpk(P.nominal)
```

```
Pnom =

         1
  -------------
  s^2 (s^2 + 2)

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

Nominal Closed-Loop Stability

Next, we evaluate the nominal closed-loop transfer function `Tnom`, and then check that all the poles of the nominal system have negative real parts:

```
Tnom = zpk(T.nominal);
maxrealpole = max(real(pole(Tnom)))
```

```
maxrealpole = -0.8232
```

Robust Stability Margin

Will the feedback loop remain stable for all possible values of `k,m1,m2` in the specified uncertainty range? We can use the `robstab` function to answer this question rigorously.

```
% Show report and compute sensitivity
opt = robOptions('Display','on','Sensitivity','on');
[StabilityMargin,wcu] = robstab(T,opt);
```

```
Computing peak...  Percent completed: 100/100
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 288% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 289% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 575 rad/seconds.
 -- Sensitivity with respect to each uncertain element is:
      12% for k. Increasing k by 25% decreases the margin by 3%.
      47% for m1. Increasing m1 by 25% decreases the margin by 11.8%.
      47% for m2. Increasing m2 by 25% decreases the margin by 11.8%.
```

The report indicates that the closed loop can tolerate up to three times as much variability in `k,m1,m2` before going unstable. It also provides useful information about the sensitivity of stability to each parameter. The variable `wcu` contains the smallest destabilizing parameter variations (relative to the nominal values).

```
wcu
```

```
wcu = struct with fields:
     k: 1.5773
    m1: 0.4227
    m2: 0.4227
```

Worst-Case Performance Analysis

Note that the peak gain across frequency of the closed-loop transfer T is indicative of the level of overshoot in the closed-loop step response. The closer this gain is to 1, the smaller the overshoot. We use `wcgain` to compute the worst-case gain `PeakGain` of T over the specified uncertainty range.

```
[PeakGain,wcu] = wcgain(T);
PeakGain
```

```
PeakGain = struct with fields:
           LowerBound: 1.0453
           UpperBound: 1.0731
    CriticalFrequency: 9.2590
```

Substitute the worst-case parameter variation `wcu` into T to compute the worst-case closed-loop transfer `Twc`.

```
Twc = usubs(T,wcu);         % Worst-case closed-loop transfer T
```

Finally, pick from random samples of the uncertain parameters and compare the corresponding closed-loop transfers with the worst-case transfer `Twc`.

```
Trand = usample(T,4);         % 4 random samples of uncertain model T
clf
subplot(211), bodemag(Trand,'b',Twc,'r',{10 1000});  % plot Bode response
subplot(212), step(Trand,'b',Twc,'r',0.2);           % plot step response
```

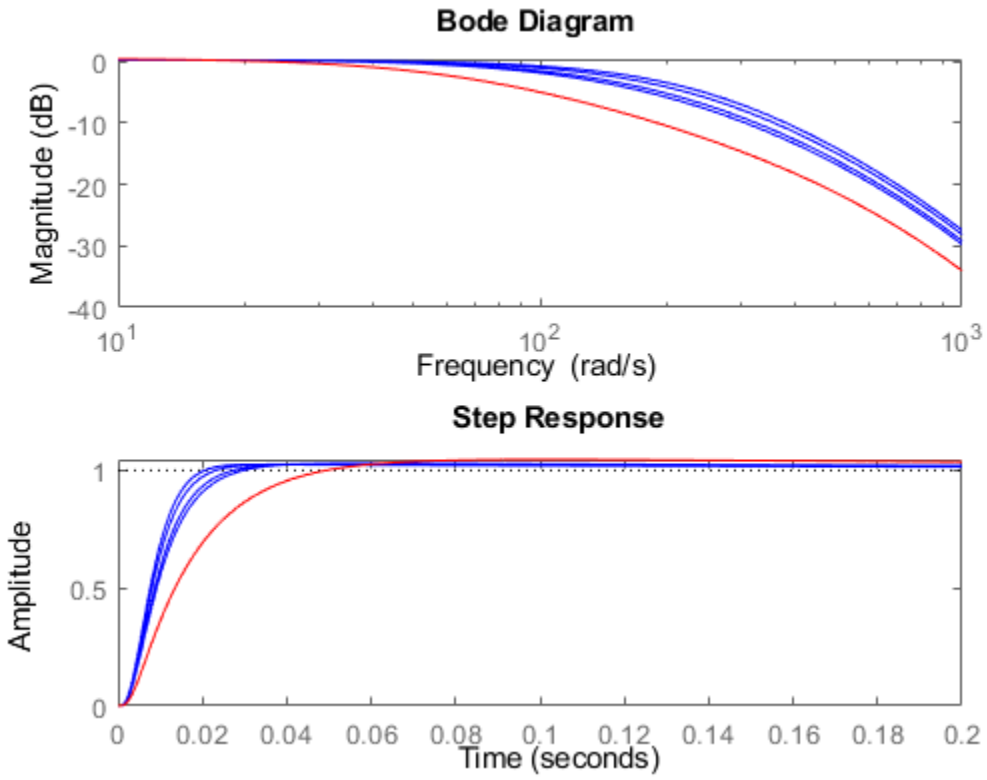


Figure 4: Bode diagram and step response.

In this analysis, we see that the compensator C performs robustly for the specified uncertainty on k,m1,m2.

See Also

`ureal` | `uss` | `robstab` | `wcgain` | `usubs`

More About

- “Robustness and Worst-Case Analysis”

Robust Stability and Worst-Case Gain of Uncertain System

This example shows how to calculate the robust stability and examine the worst-case gain of the closed-loop system described in “System with Uncertain Parameters” on page 1-6. 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. Use `robstab` to analyze whether the closed-loop system `T` is robustly stable for all combinations of possible values of these three parameters.

```
[stabmarg,wcus] = robstab(T);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 2.8803
           UpperBound: 2.8864
    CriticalFrequency: 575.0339
```

The data in the structure `stabmarg` includes bounds on the stability margin, which indicate that the control system can tolerate almost 3 times the specified uncertainty before going unstable. It is stable for all parameter variations in the specified ±20% range. The critical frequency is the frequency at which the system is closest to instability.

The structure `wcus` contains the smallest destabilization perturbation values for each uncertain element.

```
wcus
```

```
wcus = struct with fields:
     k: 1.5773
    m1: 0.4227
    m2: 0.4227
```

You can evaluate the uncertain model at these perturbation values using `usubs`. Examine the pole locations of that worst-case model.

```
Tunst = usubs(T,wcus);
damp(Tunst)
```

```
         Pole              Damping       Frequency      Time Constant
                                       (rad/seconds)      (seconds)

 -8.82e-01 + 1.55e-01i     9.85e-01       8.95e-01         1.13e+00
 -8.82e-01 - 1.55e-01i     9.85e-01       8.95e-01         1.13e+00
 -1.25e+00                 1.00e+00       1.25e+00         7.99e-01
  1.15e-06 + 5.75e+02i    -2.01e-09       5.75e+02        -8.66e+05
  1.15e-06 - 5.75e+02i    -2.01e-09       5.75e+02        -8.66e+05
 -1.50e+03 + 6.44e+02i     9.19e-01       1.63e+03         6.67e-04
 -1.50e+03 - 6.44e+02i     9.19e-01       1.63e+03         6.67e-04
```

The system contains a pair of poles very close to the imaginary axis, with a damping ratio of less than `1e-7`. This result confirms that the worst-case perturbation is just enough to destabilize the system.

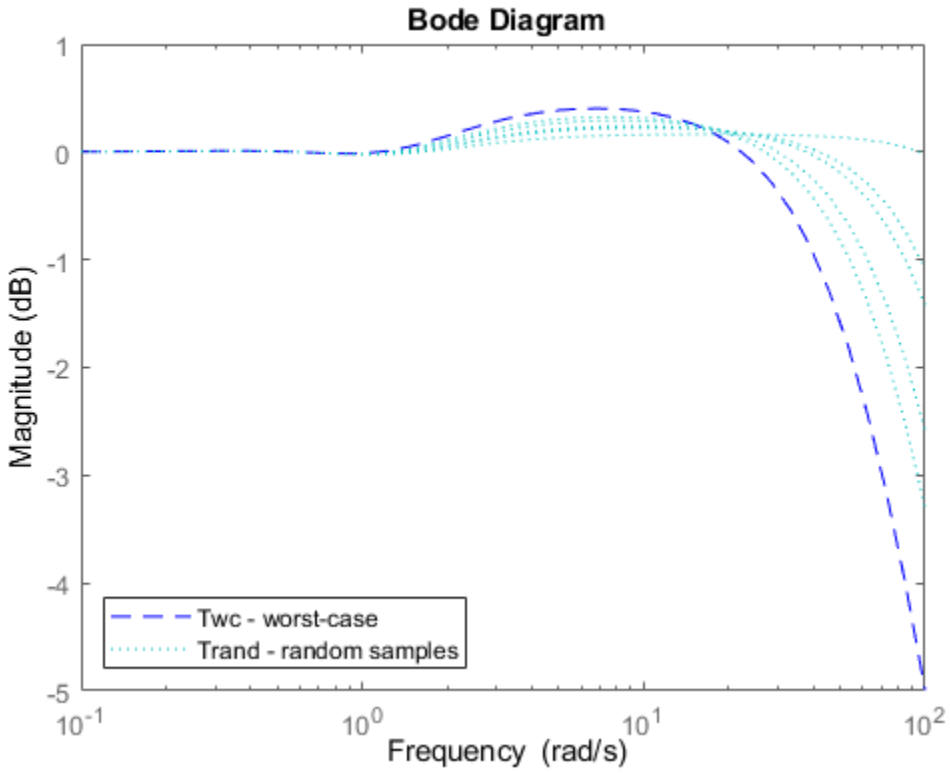
Use `wcgain` to calculate the worst-case peak gain, the highest peak gain occurring within the specified uncertainty ranges.

```
[wcg,wcug] = wcgain(T);
wcg
```

```
wcg = struct with fields:
           LowerBound: 1.0453
           UpperBound: 1.0731
    CriticalFrequency: 9.2590
```

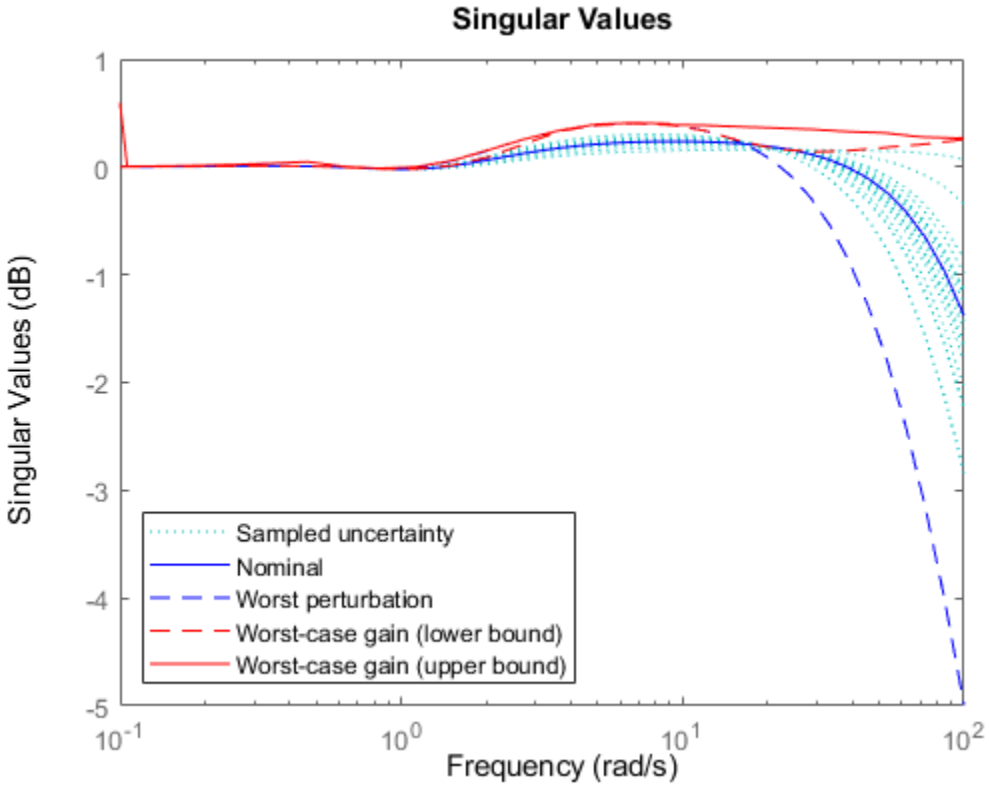
`wcug` contains the values of the uncertain elements that cause the worst-case gain. Compute a closed-loop model with these values, and plot its frequency response along with some random samples of the uncertain system.

```
Twc = usubs(T,wcug);
Trand = usample(T,5);
bodemag(Twc,'b--',Trand,'c:',{.1,100});
legend('Twc - worst-case','Trand - random samples','Location','SouthWest');
```



Alternatively use `wcsigmaplot` to visualize the highest possible gain at each frequency, the system with the highest peak gain, and random samples of the uncertain system.

```
wcsigmaplot(T,{.1,100})
```



See Also

`robstab` | `wcgain` | `wcsigmaplot`

Related Examples

- “Robustness and Worst-Case Analysis”

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

Loop Shaping for Performance and Robustness

Performance and robustness requirements can often be expressed in terms of the open-loop response gain. For example, high gain at low frequencies reduces steady-state offsets and improves disturbance rejection. Similarly, high-frequency roll-off improves stability where the plant model is uncertain or inaccurate. Loop shaping is an approach to control design in which you determine a suitable profile for the open-loop system response and design a controller to achieve that shape.

Tradeoff Between Performance and Robustness

The uncertainty in your plant model can be a limiting factor in determining what you can achieve with feedback. High loop gains can attenuate the effects of plant model uncertainty and reduce the overall sensitivity of the system to disturbances. 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.

For this reason, most controller designs involve a tradeoff between performance and robustness against uncertainty. Robust Control Toolbox commands for loop-shaping controller design let you determine the tradeoff that best meets the requirements of your system.

- `loopsyn` — Designs a stabilizing controller that shapes the open-loop response to approximate the target loop shape that you provide. You can adjust the balance between performance and robustness.
- `mixsyn` — Controller design optimized for performance. This function allows you more precise specification of the shapes of different loop responses.
- `ncfsyn` — Controller design optimized for robustness (stability margin). You provide weighting functions that shape the plant to a desirable profile.

The performance optimization of `mixsyn` tends to produce plant-inverting designs, which can be less robust. In particular, `mixsyn` designs can be fragile for ill-conditioned MIMO plants and for plants with structured uncertainty, such as uncertainty on the damping and natural frequency of resonant modes. In contrast, `ncfsyn` deters control strategies like plant inversion that rely on exact knowledge of the plant poles and zeroes. Thus `ncfsyn` adds some of the robustness to structured uncertainty that is missing in `mixsyn` designs. By combining elements of both `ncfsyn` and `mixsyn`, the `loopsyn` approach can provide robustness to both structured and unstructured uncertainty while also providing good performance.

Choosing a Target Loop Shape

Here are some basic design tradeoffs to consider when choosing a target loop shape.

- **Robust Stability.** Use a target loop shape with gain as low 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.** Use a target loop shape with gain as high as possible at frequencies where your model is good. Doing so ensures good reference tracking and good disturbance attenuation.
- **Crossover and Rolloff.** Use a target loop shape with its 0 dB crossover frequency ω_c between the above two frequency ranges. Ensure that the target loop shape rolls off with a slope between –20 dB/decade and –30 dB/decade near ω_c. This rolloff helps keep phase lag approximately between –130° and –90° near crossover for good phase margins.

Keep these principles in mind when choosing your target loop shape for `loopsyn` or the shaping filters for `ncfsyn`. For further details about choosing weighting functions for `mixsyn`, see “Mixed-Sensitivity Loop Shaping” on page 2-25.

Limitations on Control Bandwidth

Other considerations that might affect your choice of loop shape are the unstable poles and zeros of the plant, which impose fundamental limits on your 0 dB crossover frequency ω_c (see [1]). For instance, ω_c must be greater than the natural frequency of any unstable pole of the plant, and smaller than the natural frequency of any unstable zero of the plant.

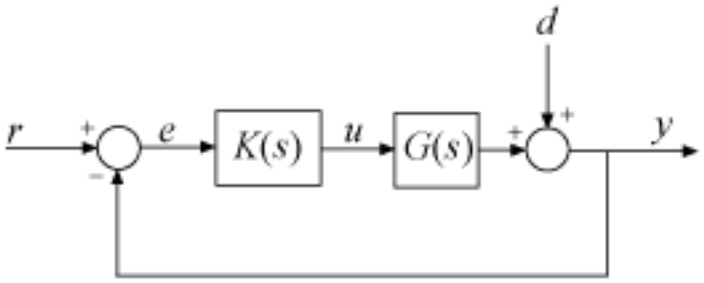
$$\max_{\mathrm{Re}(p_i) > 0} |p_i| < \omega_c < \min_{\mathrm{Re}(z_i) > 0} |z_i| .$$

If you do not take care to choose a target loop shape that conforms to these fundamental constraints, then you might not achieve good results. For instance, `loopsyn` will compute the optimal loop-shaping controller `K` for a target loop shape `Gd` that does not meet this requirement, but the resulting response `L = G*K` might have a poor fit to the target loop shape `Gd`, and consequently it might be impossible to meet your performance goals.

Additionally, because plant uncertainty typically increases with frequency, there is a limit on the bandwidth that you can reliably achieve. For instance, consider an approximate model G_0 of a SISO plant G. You can express the uncertainty in this plant as a multiplicative uncertainty Δ_M, such that $G = G_0(1 + \Delta_M)$. The uncertainty is bounded at each frequency, $|\Delta_M(j\omega)| \le \beta(\omega)$, where $\beta(\omega)$ is the percentage of model uncertainty. Typically, $\beta(\omega)$ is small at low frequencies (accurate model) and increases at high frequencies (inaccurate model). The frequency where $\beta(\omega) = 2$ 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 two 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 $|\Delta_M|<1$.

Loop Shapes, Performance, and Robustness

For a deeper understanding of the relationship between loop shapes, performance, and robustness, consider the multivariable feedback control system shown in the following figure.



To quantify the multivariable stability margins and performance of such systems, you can use the closed-loop sensitivity function S and complementary sensitivity function T, defined as:

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

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

where the $L(s)$ is the open-loop transfer function

$$L(s) = G(s)K(s)\,.$$

Specifying a target shape $G_d(s)$ for the open-loop transfer function $L(s)$ is equivalent to imposing constraints on the singular values of the sensitivity $S(s)$ and complementary sensitivity $T(s)$. For instance, for a target loop shape with high gain at low frequency, the condition $\underline{\sigma}(L(s)) > \underline{\sigma}(G_d(s)) \gg 1$ is equivalent to $\bar{\sigma}(S(s)) < 1/\underline{\sigma}(G_d(s))$, where $\bar{\sigma}$ and $\underline{\sigma}$ denote the largest and smallest singular values, respectively. Similarly, for a target loop shape with low gain at high frequency, $\bar{\sigma}(L(s)) < \bar{\sigma}(G_d(s)) \ll 1$ is equivalent to $\bar{\sigma}(T(s)) < \bar{\sigma}(G_d(s))$.

When using `loopsyn`, you specify $G_d(s)$ directly, and `loopsyn` approximately imposes these constraints on the sensitivity and complementary sensitivity. For `mixsyn`, you specify weighting functions $W_1(s)$ and $W_3(s)$ such that $W_1(s)$ matches $1/G_d(s)$ at low frequency is smaller than 1 elsewhere, and $W_3(s)$ matches $G_d(s)$ at high frequency and is smaller than 1 elsewhere. (See “Mixed-Sensitivity Loop Shaping” on page 2-25). Then `mixsyn` approximately imposes the constraints $\bar{\sigma}(S) < \left|W_1^{-1}\right|$ and $\bar{\sigma}(T) < \left|W_3^{-1}\right|$, which roughly enforce the loop shape G_d.

Additionally, robustness to multiplicative plant uncertainty is equivalent to imposing a small-gain constraint on $T(s)$ (see [1], p.342). Thus, enforcing rolloff in the loop shape G_d (or equivalently, $\bar{\sigma}(T) < \left|W_3^{-1}\right|$), provides some robustness against unmodeled plant dynamics at high frequency.

Guaranteed Gain and Phase Margins

For those who are more comfortable with classical single-loop concepts, there are the important connections between the multiplicative stability margins predicted by the gain of $T(s)$ and those predicted by classical M-circles, as found on the Nichols chart. In the SISO case, the largest singular value of $T(s)$ is just the peak gain, given by:

$$|T(s)| = \left|\frac{L(s)}{1 + L(s)}\right|.$$

This quantity is the same quantity you obtain from Nichols chart M-circles. The H_∞ norm $\|T\|_\infty$ (see `hinfnorm`) 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. You can relate $\|T\|_\infty$ and $\|S\|_\infty$ to the classical gain margin G_M and phase margin θ_M in each feedback loop of the multivariable feedback system illustrated above, via the formulas:

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

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

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

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

(See [2].) 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:

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

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

For more information about the relation between sensitivity functions and gain and phase margins, see [3].

References

[1] Skogestad, Sigurd, Ian Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. Chichester; New York: Wiley, 1996.

[2] Lehtomaki, N., N. Sandell, and M. Athans. "Robustness Results in Linear-Quadratic Gaussian Based Multivariable Control Designs." *IEEE Transactions on Automatic Control* 26, no.1 (February 1981):75–93.

[3] Seiler, Peter, Andrew Packard, and Pascal Gahinet. “An Introduction to Disk Margins [Lecture Notes].” *IEEE Control Systems Magazine* 40, no. 5 (October 2020): 78–95.

See Also

`loopsyn` | `mixsyn` | `ncfsyn`

Related Examples

- “Loop-Shaping Controller Design” on page 2-8
- “Mixed-Sensitivity Loop Shaping” on page 2-25
- “Loop Shaping Using the Glover-McFarlane Method” on page 2-32

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:

H2 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 $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p > 0$, $p \leq \min\{m, n\}$.

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

The greatest singular value σ_1 is sometimes denoted

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

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

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

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:

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

H_∞-norm:

$$\|G\|_\infty \triangleq \sup_{\omega} \bar{\sigma}(G(j\omega))$$

where sup denotes the least upper bound.

See Also

“Interpretation of H-Infinity Norm” on page 5-2

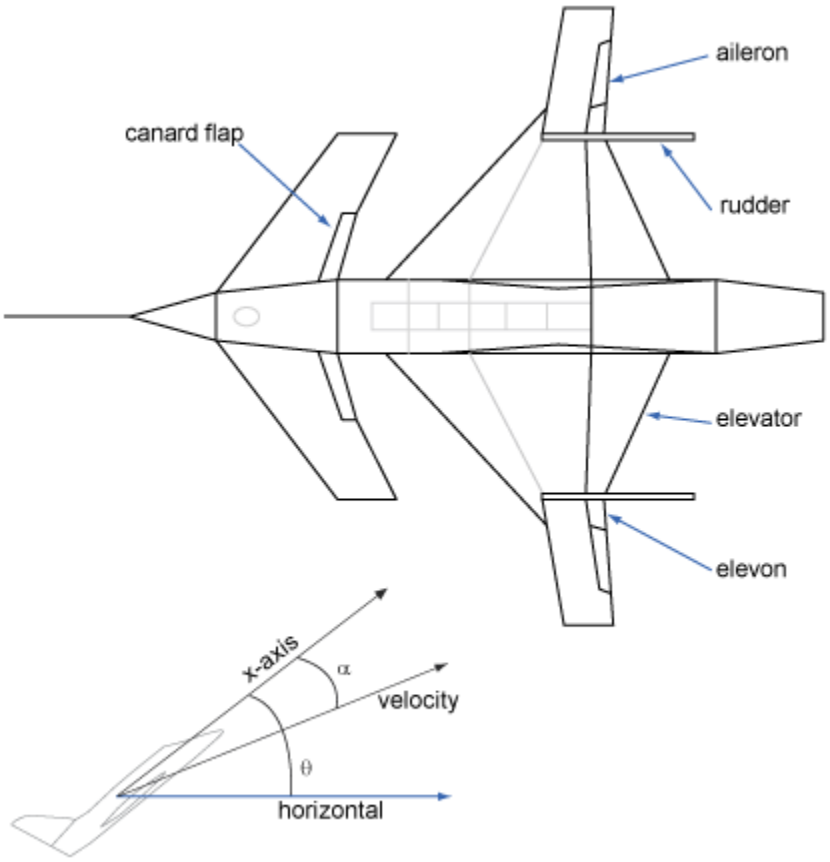
Loop-Shaping Controller Design

This example shows how to design a controller by specifying a desired shape for the open-loop response of the plant with the controller. The `loopsyn` command designs a controller that shapes the open-loop response to approximately match the target loop shape you provide. `loopsyn` lets you adjust the tradeoff between performance and robustness to obtain satisfactory time-domain responses while avoiding fragile designs with plant inversion or flexible mode cancellation.

In this example, you design a controller for an aircraft model.The example shows how varying the balance between performance and robustness affects loop shape and closed-loop response. The example then shows how to reduce the controller order while preserving desirable characteristics of the response.

Plant Model

This example uses the two-input, two-output NASA HiMAT aircraft model [1]. The aircraft is shown in the following diagram.



The control variables are the elevon and canard actuators (δ_e and δ_c). The output variables are the angle of attack (α) and attitude angle (θ). The model has six states, given by

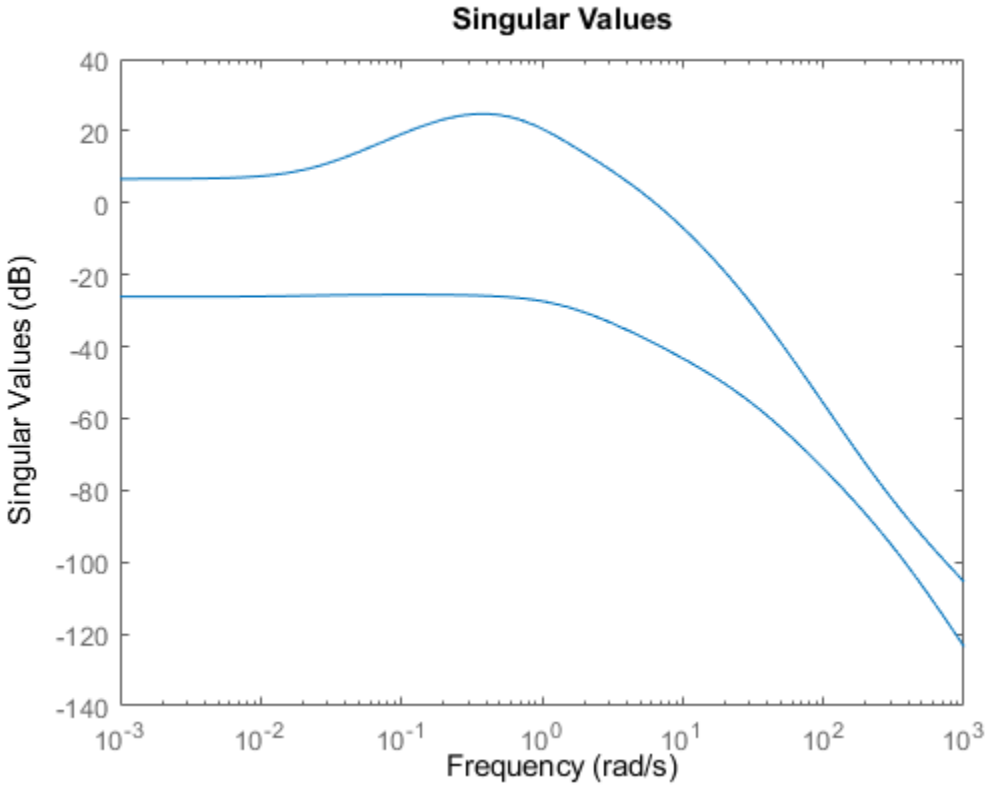
$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} \dot{\alpha} \\ \alpha \\ \dot{\theta} \\ \theta \\ x_e \\ x_c \end{bmatrix},$$

where x_e and x_c are the elevator and canard actuator states, respectively. Using the following state-space matrices, create the model of this plant.

```
A = [ -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];
B = [0     0;
     0     0;
     0     0;
     0     0;
    30     0;
     0    30];
C = [0     1     0     0     0     0;
     0     0     0     1     0     0];
D = [0     0;
     0     0];

G = ss(A,B,C,D);
G.InputName = {'elevon','canard'};
G.OutputName = {'attack','attitude'};
```

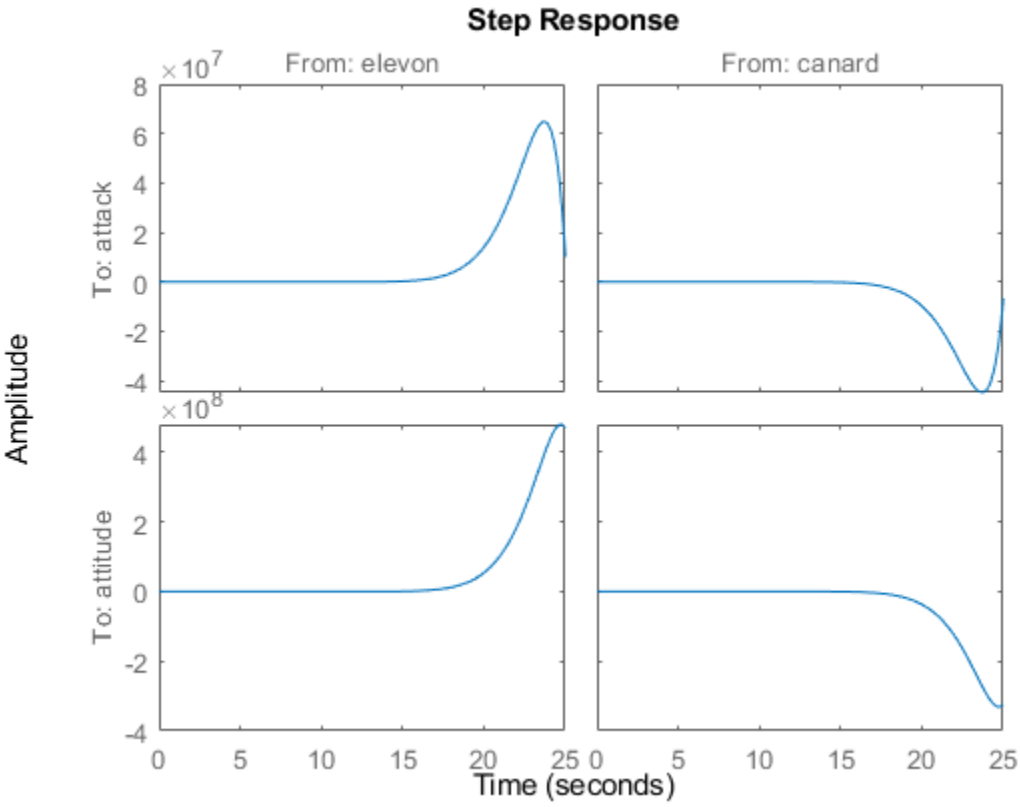
Examine the singular values of the model.

```
sigma(G)
```



This plant is ill-conditioned, in the sense that there is a gap of about 40 dB between the largest and smallest singular values in the vicinity of the desired control bandwidth of 8 rad/s. Further, as a step plot shows, the open-loop response of this plant is unstable.

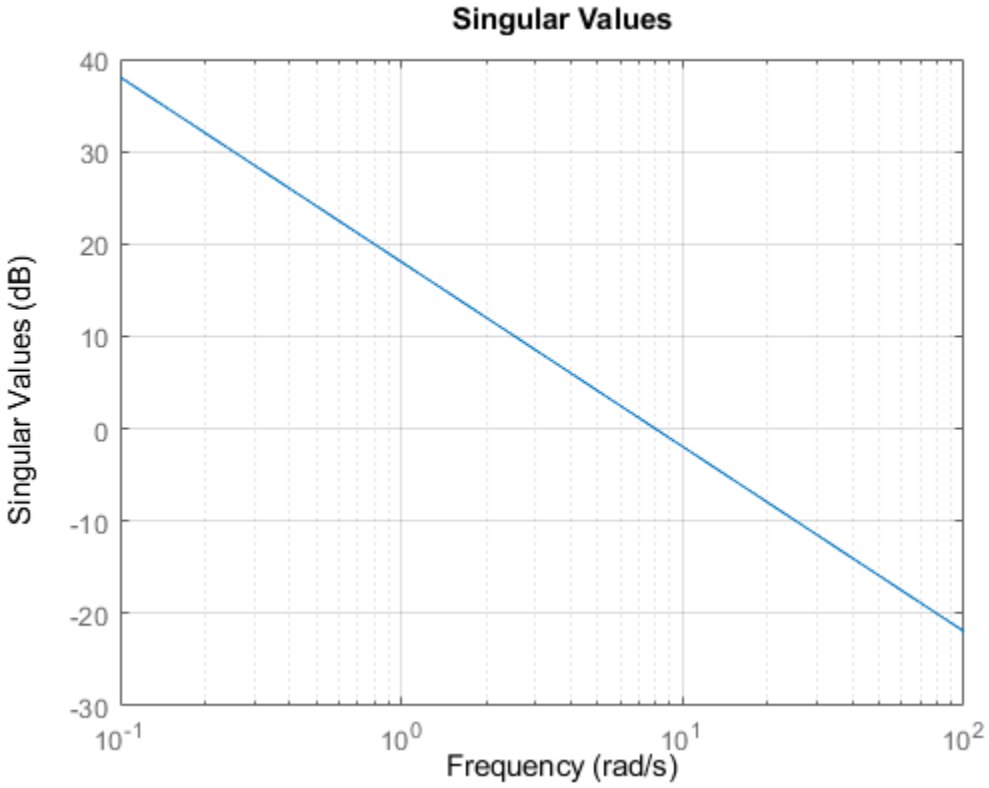
```
step(G)
```



Initial Controller Design

To design a stabilizing controller for this plant, select a target loop shape. A typical loop shape has low gain at high frequencies for robustness, and high gain at low frequencies for performance. For the desired crossover frequency of 8 rad/s, a simple target loop shape that meets these requirements is $G_d = 8/s$.

```
Gd = tf(8,[1 0]);
sigma(Gd,{0.1 100})
grid on
```



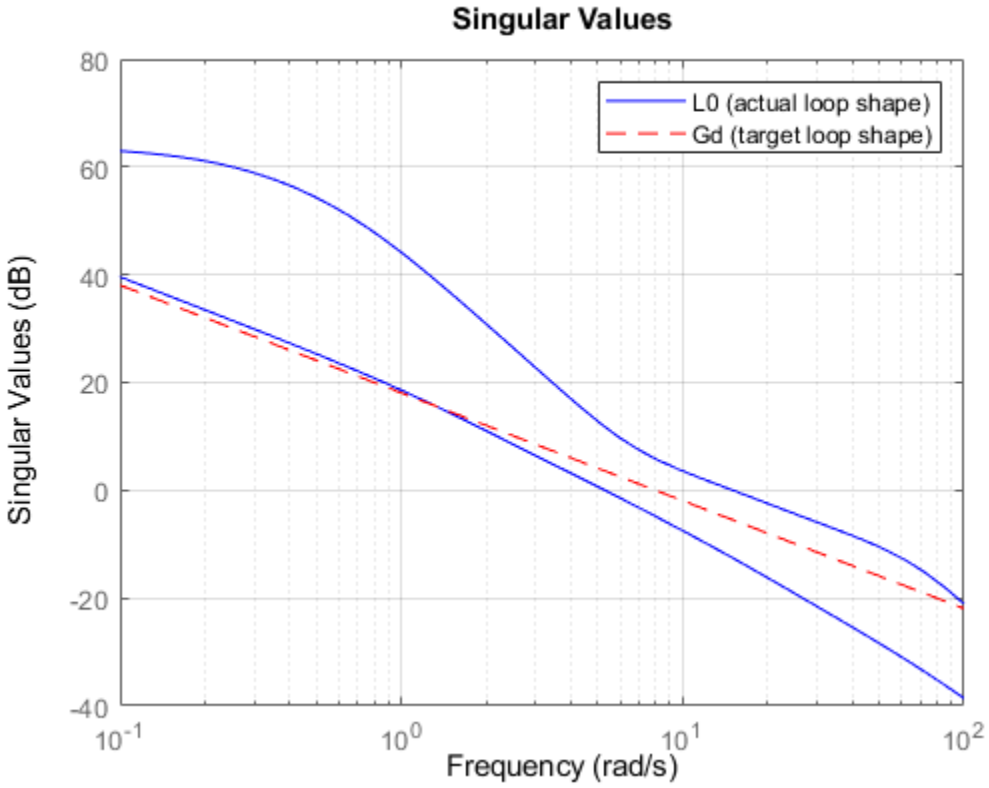
Design the initial controller with `loopsyn`.

```
[K0,CL0,gamma0,info0] = loopsyn(G,Gd);
gamma0
```

```
gamma0 = 1.2931
```

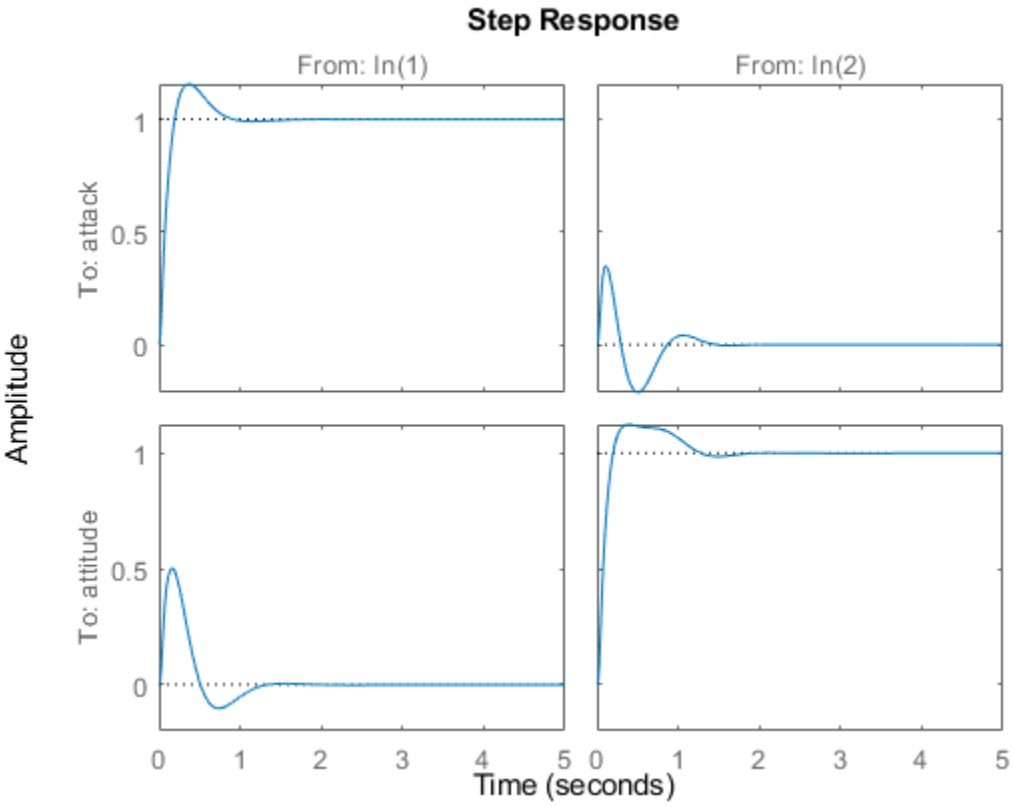
The performance `gamma` is a measure of how well the loop shape with `K0` matches the desired loop shape. Values near or below 1 indicate that `G*K0` is close to `Gd`. Compare the achieved loop shape with the target.

```
L0 = G*K0;
sigma(L0,"b",Gd,"r--",{.1,100});
grid
legend("L0 (actual loop shape)","Gd (target loop shape)");
```



The match is not very close at low frequencies, though it improves near crossover. Moreover, the two singular values are still somewhat far apart around crossover, such that there are effectively two crossover frequencies. Examine how this open-loop shape affects the closed-loop step response.

```
step(CL0,5)
```



The bump in `attitude` tracking (lower-right plot) is the result of the separation of the two singular values, leading to a response with two time constants. Also, there is significant coupling between `attack` and `attitude`. It is desirable to adjust the controller to reduce the bump in `attitude` tracking, reduce the coupling, and if possible reduce the overshoot in the `attack` response.

Design Controller for Performance

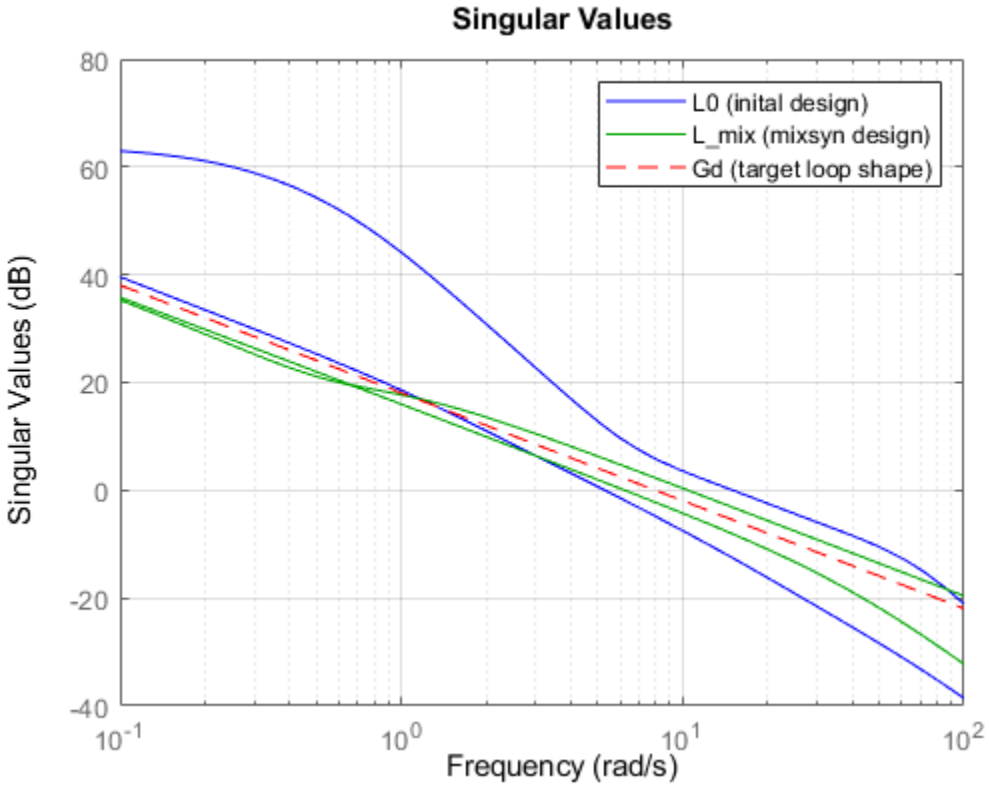
To improve the design, you can try changing the balance that `loopsyn` strikes between performance and robustness. To do so, use the `alpha` input argument to `loopsyn`. By default, `loopsyn` uses `alpha = 0.5`, which optimizes performance subject to the robustness being no worse than half the maximum achievable robustness. `alpha = 0` optimizes for performance (`mixsyn` design). Setting `alpha = 1` uses the robustness-maximizing `ncfsyn` design. First, consider the pure `mixsyn` design.

```
alpha = 0;
[K_mix,CL_mix,gamma_mix,info_mix] = loopsyn(G,Gd,alpha);
gamma_mix
```

```
gamma_mix = 0.7723
```

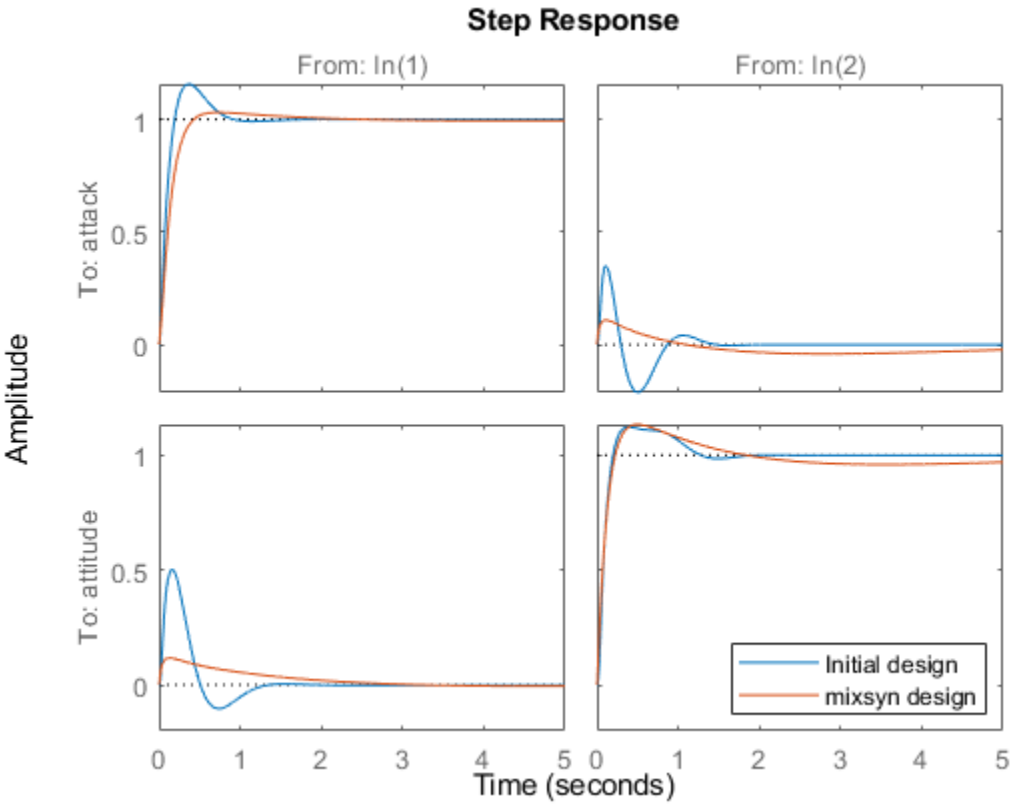
The `gamma` value indicates a much closer match to the target loop shape, which you can confirm by plotting the open-loop responses.

```
L_mix = G*K_mix;
sigma(L0,"b",L_mix,"g",Gd,"r--",{.1,100});
grid
legend("L0 (inital design)","L_mix (mixsyn design)","Gd (target loop shape)");
```



This design roughly inverts the plant. As a result, the singular values of of `L_mix` converge near the crossover frequency and are generally much closer together than in the original plant. With this plant-inverting controller, the closed-loop response shows good performance, with minimal overshoot and cross-coupling.

```
step(CL0,CL_mix,5)
legend("Initial design","mixsyn design","Location","southeast")
```



However, this performance comes at the cost of robustness. Compare the stability margins of the system with the initial design and the `mixsyn` design.

```
DM0 = diskmargin(G,K0);
DM_mix = diskmargin(G,K_mix);
DM0.DiskMargin
```

```
ans = 0.1319
```

```
DM_mix.DiskMargin
```

```
ans = 0.0517
```

The plant-inverting design has poor robustness. For instance, if the smallest singular value of the plant model is 1% of the largest singular value, inverting the plant amplifies model errors by a factor of 100 in the direction of the smallest singular value. Thus, unless you have a highly accurate model, it is preferable to use a design with better robustness.

Design Controller for Robustness

At the opposite extreme is the pure `ncfsyn` design, optimized for robustness. Compute such a controller using `alpha = 1`, and examine the resulting stability, loop shape, and responses.

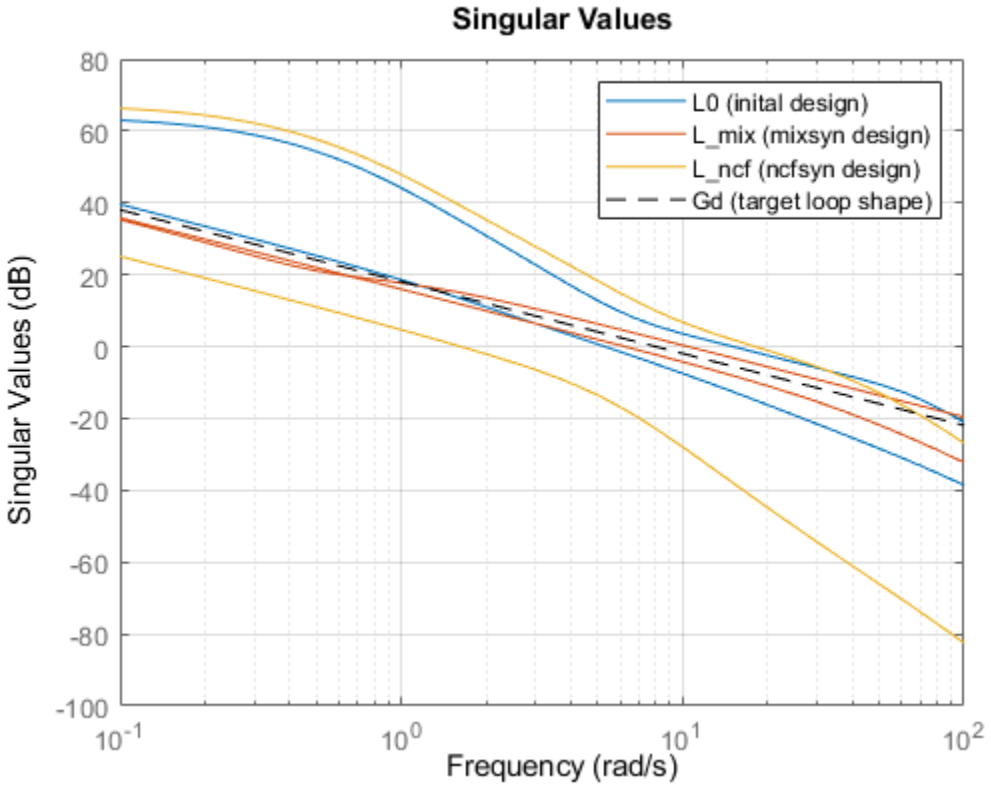
```
alpha = 1;
[K_ncf,CL_ncf,gamma_ncf,info_ncf] = loopsyn(G,Gd,alpha);
gamma_ncf
```

```
gamma_ncf = 2.8360
```

```
DM_ncf = diskmargin(G,K_ncf);
DM_ncf.DiskMargin
```

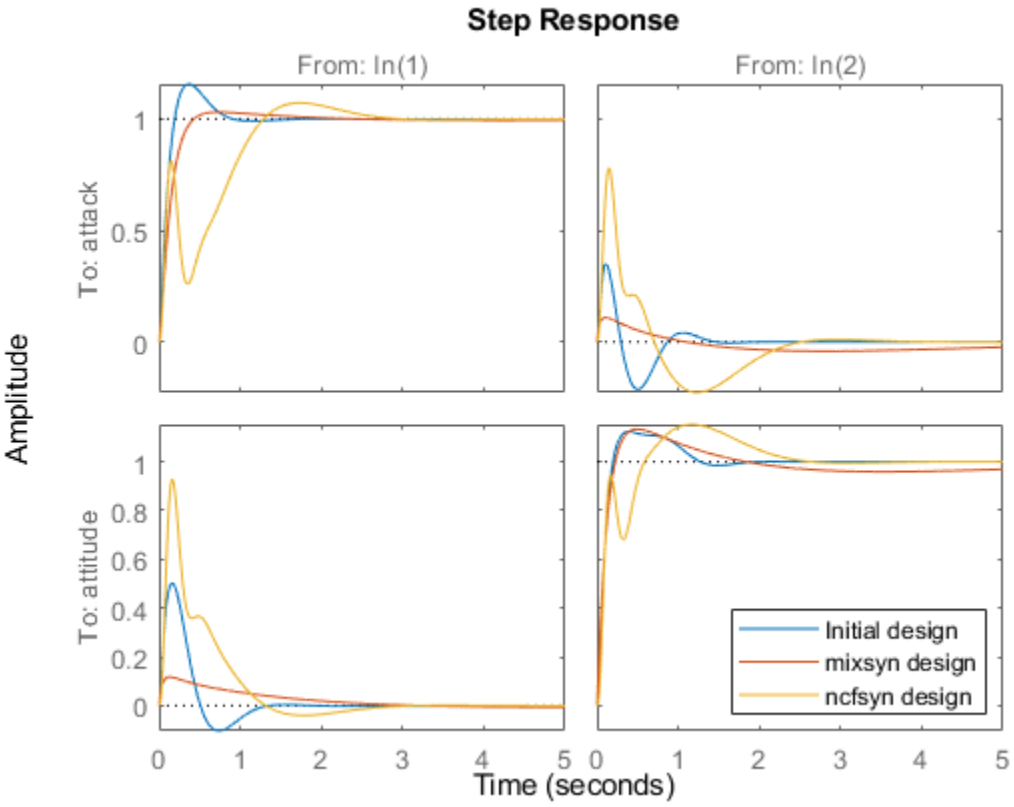
```
ans = 0.2201
```

```
L_ncf = G*K_ncf;
sigma(L0,L_mix,L_ncf,Gd,"k--",{.1,100});
grid
legend("L0 (inital design)","L_mix (mixsyn design)","L_ncf (ncfsyn design)","Gd (target loop shap
```



The increased value of `gamma` indicates poor performance, though the stability margin is improved, as expected. The singular-value plot shows that this controller inverts the plant even less than the initial, which is evident in that the separation of the singular values is roughly the same as it was for the open-loop plant. The separation of crossover frequencies results in slow and fast time constants in the step response, which is even poorer than the initial design. The kick resulting from the wide crossover region is now apparent in all four I/O channels.

```
step(CL0,CL_mix,CL_ncf,5)
legend("Initial design","mixsyn design","ncfsyn design","Location","southeast")
```



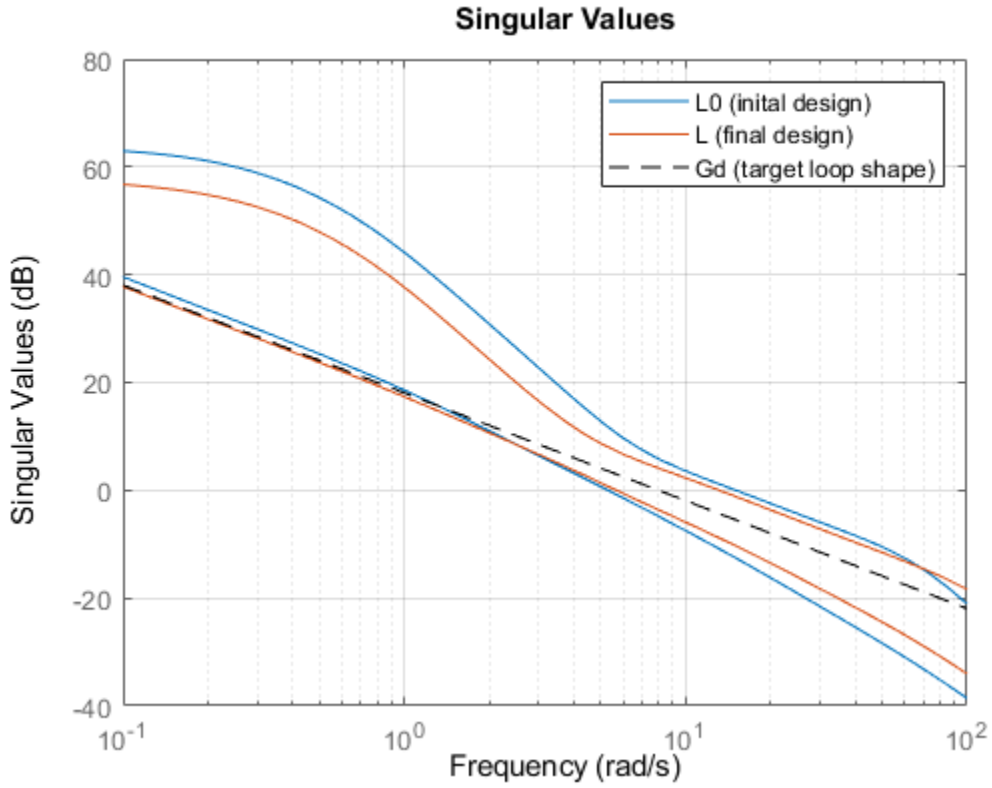
Choosing a Satisfactory Design

Thus, to improve on the default design, slightly favoring the `mixsyn` design without throwing away too much stability margin might yield a suitable design for this plant. You can control how much `loopsyn` favors performance or robustness by setting `alpha` to any value between 0 and 1. The default value used in the initial controller is `alpha = 0.5`. Try a value that slightly favors performance, and compare the results with the initial design.

```
alpha = 0.25;
[K,CL,gamma,info] = loopsyn(G,Gd,alpha);
gamma
```

```
gamma = 1.0119
```

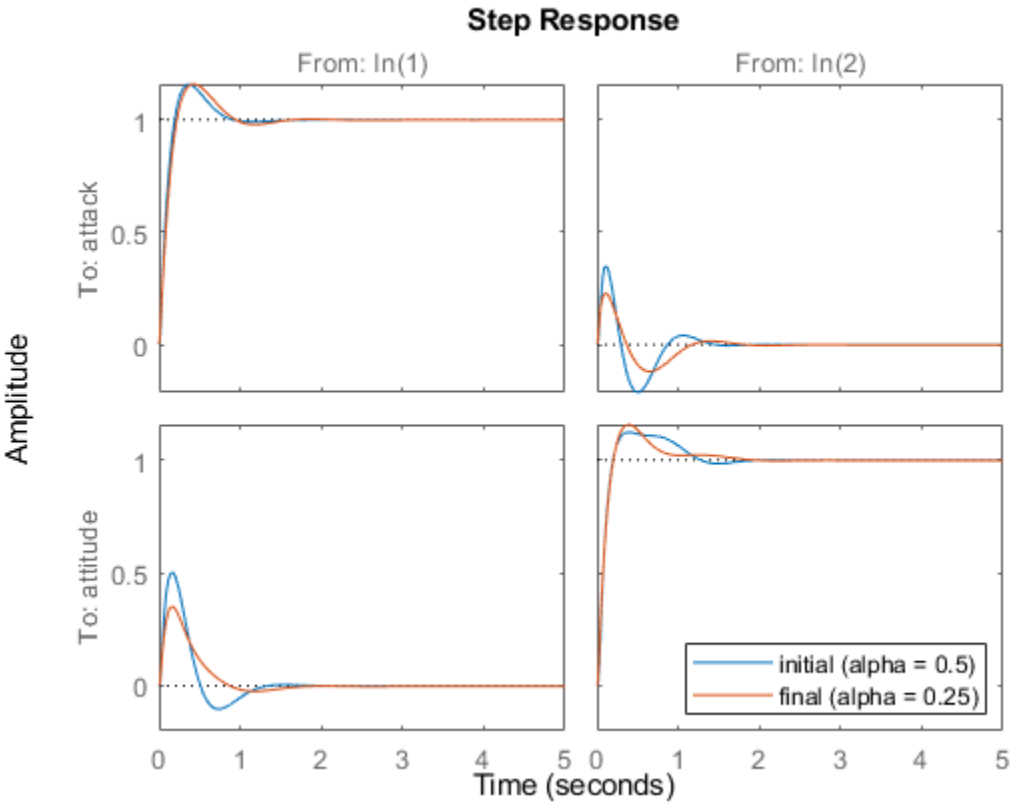
```
L = G*K;
sigma(L0,L,Gd,"k--",{.1,100});
grid
legend("L0 (inital design)","L (final design)","Gd (target loop shape)");
```




```
DM = diskmargin(G,K);
DM.DiskMargin
```

```
ans = 0.0950
```

```
step(CL0,CL,5)
legend("initial (alpha = 0.5)","final (alpha = 0.25)","Location","southeast")
```



The `alpha = 0.25` design yields reasonably good performance, reducing coupling and eliminating the bump in the `attitude` response. It has a slightly smaller stability margin (disk margin of about 0.09, compared to about 0.125 for the initial design). For your application, you can select whatever value of `alpha` between 0 and 1 achieves an acceptable balance between performance and robustness.

Reduce Controller Order

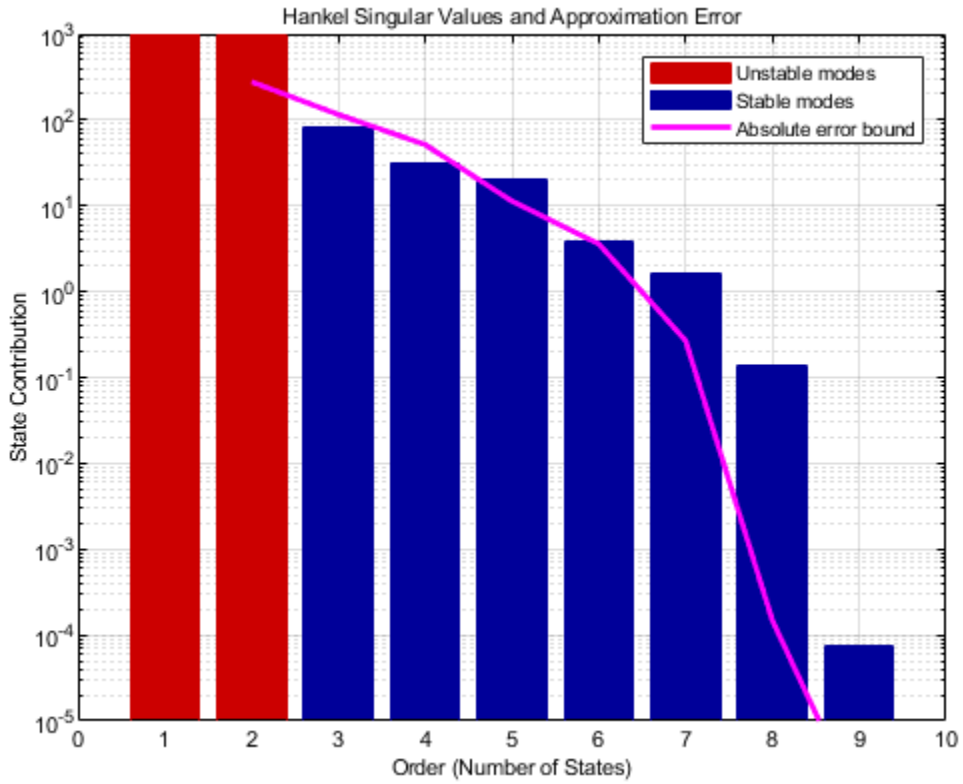
It is sometimes possible to simplify the controller returned by `loopsyn` while preserving desirable characteristics of the system response. In this example, the controller `K` is ninth order.

```
order(K)
```

```
ans = 9
```

To see whether it is possible to simplify `K`, use the `balred` command.

```
balred(K)
```



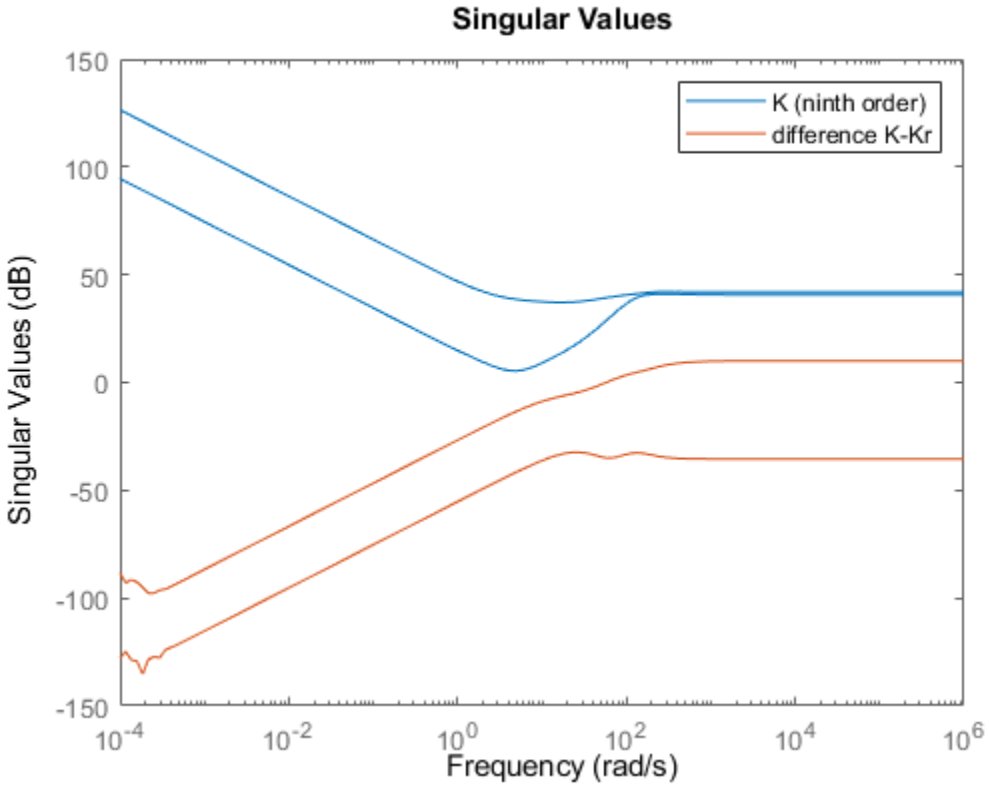
The plot shows the Hankel singular values of the controller, which indicates the relative energy contribution of each mode. The Hankel Singular value decreases sharply after sixth order, so try reducing the controller accordingly.

```
Kr = balred(K,6);
order(Kr)
```

```
ans = 6
```

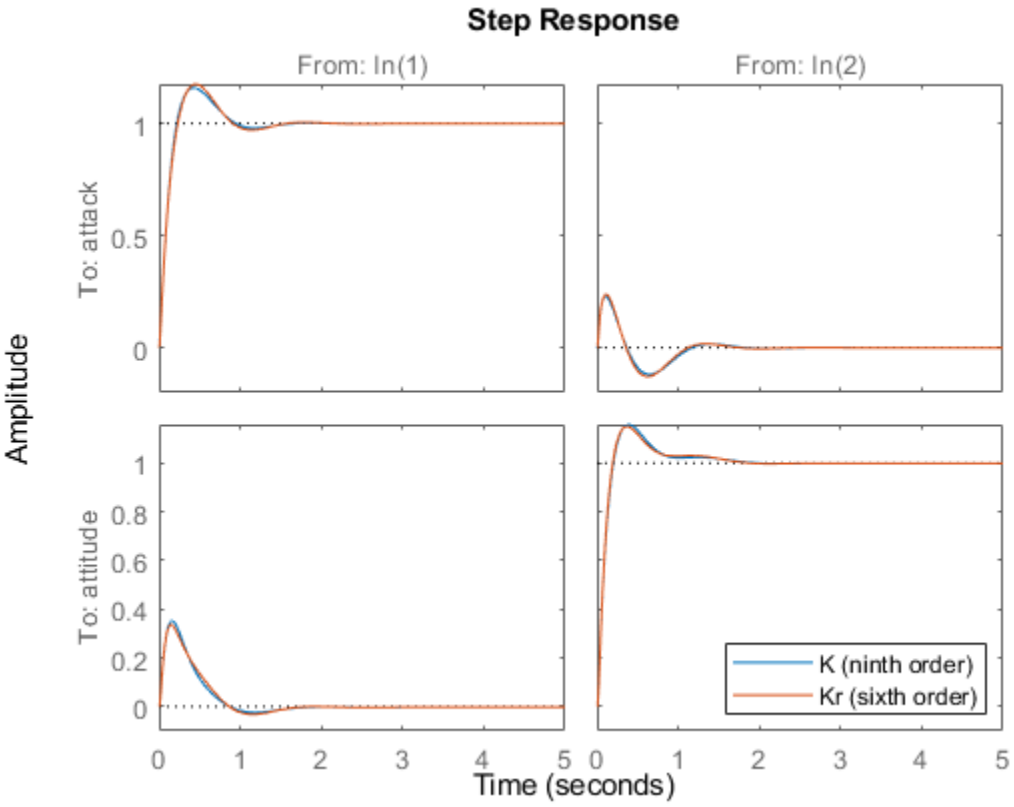
Compare the singular values of the reduced and full-order controllers to confirm that the difference between them is small.

```
sigma(K,K-Kr,{1e-4,1e6})
legend("K (ninth order)","difference K-Kr")
```



You can also confirm that the reduced-order controller produces a virtually identical closed-loop response.

```
CLr = feedback(G*Kr,eye(2));
step(CL,CLr,5)
legend("K (ninth order)","Kr (sixth order)","Location","southeast")
```

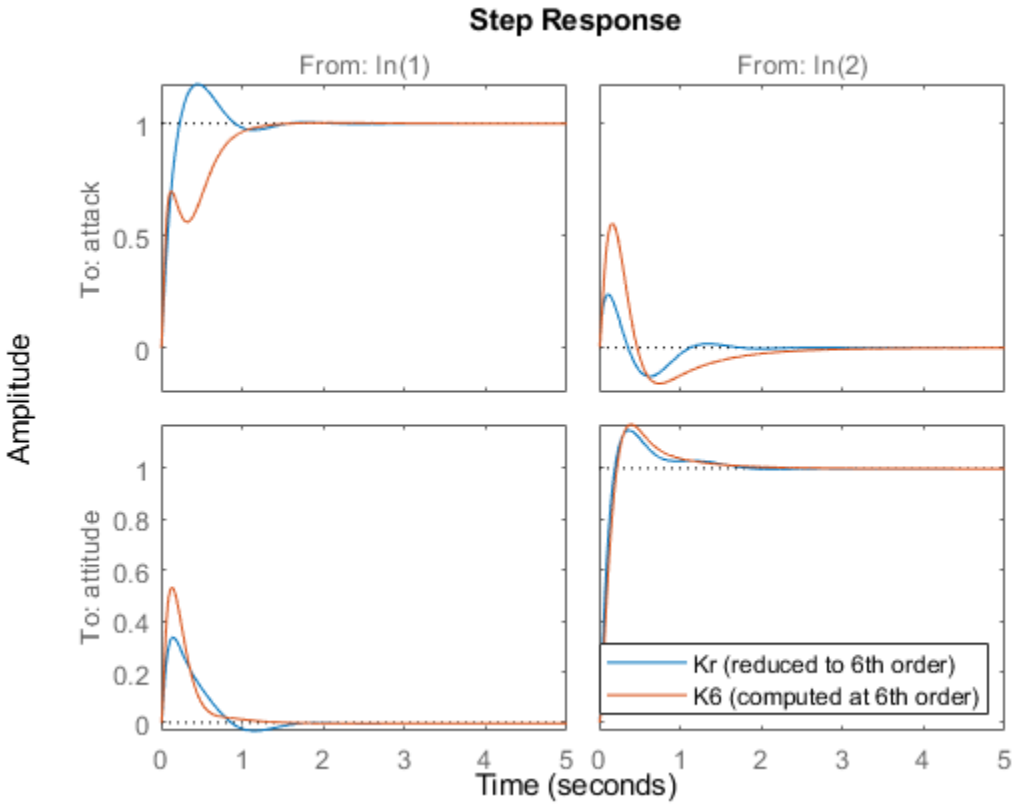


Design Controller with Reduced Order

Knowing that a sixth-order controller is sufficient to achieve the desired responses, you can use `loopsyn` to design a new controller, specifying the target order with the `ord` input argument. This approach is an alternative to the previous approach of designing and a full-order controller followed by reduction.

Design a new sixth-order controller with `alpha = 0.25` and compare the responses to the response obtained with the reduced controller.

```
alpha = 0.25;
[K6,CL6,gamma6,info6] = loopsyn(G,Gd,alpha,6);
step(CLr,CL6,5)
legend("Kr (reduced to 6th order)","K6 (computed at 6th order)","Location","southeast")
```



Designing the sixth-order controller directly yields a similar step response, although for this particular system this approach leads to some reduction in performance (`gamma6 = 1.4`, compared to `gamma = 1.0` for the full-order, `alpha = 0.25` controller `K`). However, for some systems, this approach can be better because it optimizes the lower-order controller itself, rather than removing potentially important dynamics from an optimized controller.

Conclusion

`loopsyn` lets you adjust the tradeoff between performance and robustness to strike a suitable balance for your application. You can try different values of `alpha` to find a controller that works for your requirements. You can then reduce controller order with `balred`, or use the `ord` argument of `loopsyn` to synthesize a lower-order controller directly.

References

[1] Safonov, M., A. Laub, and G. Hartmann. “Feedback Properties of Multivariable Systems: The Role and Use of the Return Difference Matrix.” *IEEE Transactions on Automatic Control* 26, no. 1 (February 1981): 47–65.

See Also

`loopsyn`

Related Examples

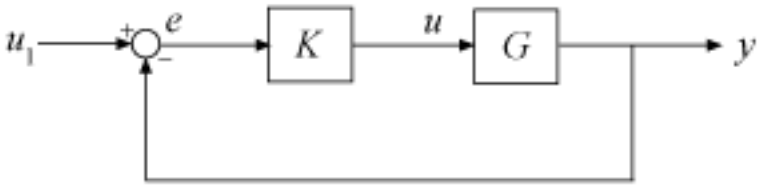
- “Loop Shaping for Performance and Robustness” on page 2-2

Mixed-Sensitivity Loop Shaping

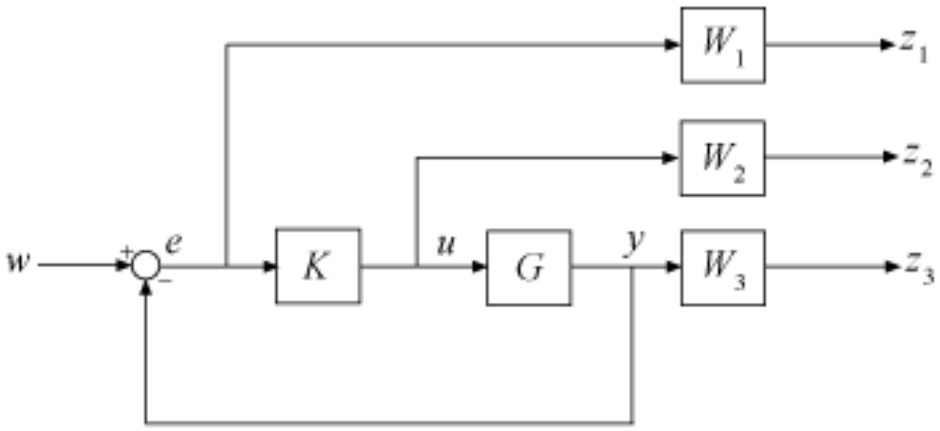
Mixed-sensitivity loop shaping lets you design an H_∞ controller by simultaneously shaping the frequency responses for tracking and disturbance rejection, noise reduction and robustness, and controller effort. This technique is a useful way to balance the necessary tradeoff between performance and robustness (see “Loop Shaping for Performance and Robustness” on page 2-2). To use this technique, you convert your desired responses into up to three weighting functions that the `mixsyn` command uses to synthesize the controller.

Problem Setup

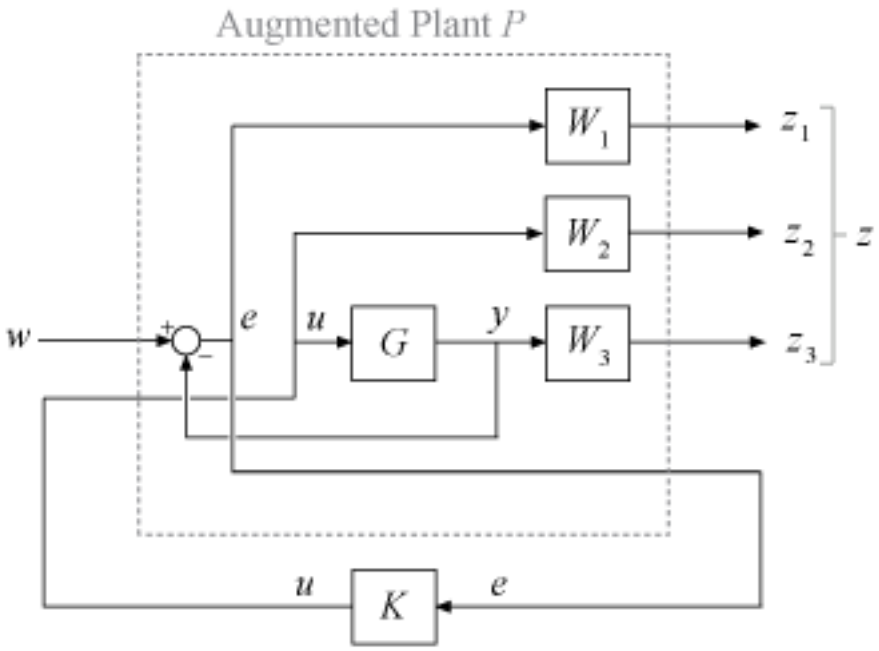
`mixsyn` designs a controller K for your plant G, assuming the standard control configuration of the following diagram.



To do so, the function appends the weighting functions you provide, $W_1(s)$, $W_2(s)$, and $W_3(s)$, to the control system, as shown in the following diagram.



`mixsyn` treats the problem as an H_∞ synthesis problem (see `hinfsyn`). It analyzes the weighted control system as $LFT(P,K)$, where P is an augmented plant P such that $\{z;e\} = P\{w;u\}$, as shown in the following diagram.



The transfer function from w to z can be expressed as

$$M(s) = \begin{bmatrix} W_1 S \\ W_2 K S \\ W_3 T \end{bmatrix},$$

where

- $S = (I + GK)^{-1}$ is the sensitivity function.
- KS is the transfer function from w to u (the control effort).
- $T = (I - S) = GK(I + GK)^{-1}$ is the complementary sensitivity function.

`mixsyn` seeks a controller K that minimizes $||M(s)||_\infty$, the H_∞ norm (peak gain) of M. To do so, it invokes `hinfsyn` on the augmented plant `P = augw(G,W1,W2,W3)`.

Choose Weighting Functions

For loop gain $L = GK$, to achieve good reference tracking and disturbance rejection, you typically want high loop gain at low frequency. To achieve robustness and attenuation of measurement noise, you typically want L to roll off at high frequency. This loop shape is equivalent to small S at low frequency and small T at high frequency.

For mixed-sensitivity loop shaping, you choose weighting functions to specify those target shapes for S and T as well as the control effort KS. The H_∞ design constraint,

$$\|M(s)\|_\infty = \left\| \begin{bmatrix} W_1 S \\ W_2 KS \\ W_3 T \end{bmatrix} \right\|_\infty \leq 1,$$

means that

$$\|S\|_\infty \leq \left|W_1^{-1}\right|$$
$$\|KS\|_\infty \leq \left|W_2^{-1}\right|$$
$$\|T\|_\infty \leq \left|W_3^{-1}\right|.$$

Therefore, you set the weights equal to the reciprocals of the desired shapes for S, KS, and T. In particular,

- For good reference-tracking and disturbance-rejection performance, choose W_1 large inside the control bandwidth to obtain small S.
- For robustness and noise attenuation, choose W_3 large outside the control bandwidth to obtain small T.
- To limit control effort in a particular frequency band, increase the magnitude of W_2 in this frequency band to obtain small KS.

`mixsyn` returns the minimum $||M(s)||_\infty$ in the output argument `gamma`. For the returned controller K, then,

$$\|S\|_\infty \leq \gamma\left|W_1^{-1}\right|$$
$$\|KS\|_\infty \leq \gamma\left|W_2^{-1}\right|$$
$$\|T\|_\infty \leq \gamma\left|W_3^{-1}\right|.$$

If you do not want to restrict control effort, you can omit W_2. In that case, `mixsyn` minimizes the H_∞ norm of

$$M(s) = \begin{bmatrix} W_1 S \\ W_3 T \end{bmatrix}.$$

You can use `makeweight` to create weighting functions with the desired gain profiles. The following example illustrates how to choose and create weighting functions for controller design with `mixsyn`.

Numeric Considerations

Do not choose weighting functions with poles very close to $s = 0$ ($z = 1$ for discrete-time systems). For instance, although it might seem sensible to choose $W_1 = 1/s$ to enforce zero steady-state error, doing so introduces an unstable pole that cannot be stabilized, causing synthesis to fail. Instead, choose $W_1 = 1/(s + \delta)$. The value δ must be small but not very small compared to system dynamics. For instance, for best numeric results, if your target crossover frequency is around 1 rad/s, choose $\delta = 0.0001$ or 0.001. Similarly, in discrete time, choose sample times such that system and weighting dynamics are not more than a decade or two below the Nyquist frequency.

Mixed-Sensitivity Loop-Shaping Controller Design

Load a plant model for a mixed-sensitivityH_∞ controller design. This two-input, two-output, six-state model is described in the example “Loop-Shaping Controller Design” on page 2-8.

```
load mixsynExampleData G
size(G)
```

```
State-space model with 2 outputs, 2 inputs, and 6 states.
```

To design a controller for performance and robustness, shape the sensitivity and complementary sensitivity functions. Choose weights that are the inverse of the desired shapes.

To achieve good reference-tracking and disturbance-rejection performance, shape S to be small inside the control bandwidth, which means choosing `W1` large at low frequency, rolling off at high frequency. For this example, specify `W1` with:

- Low-frequency gain of about 30 dB (33 in absolute units)
- High-frequency gain of about –6 dB (0.5 in absolute units)
- 0 dB crossover at about 5 rad/s.

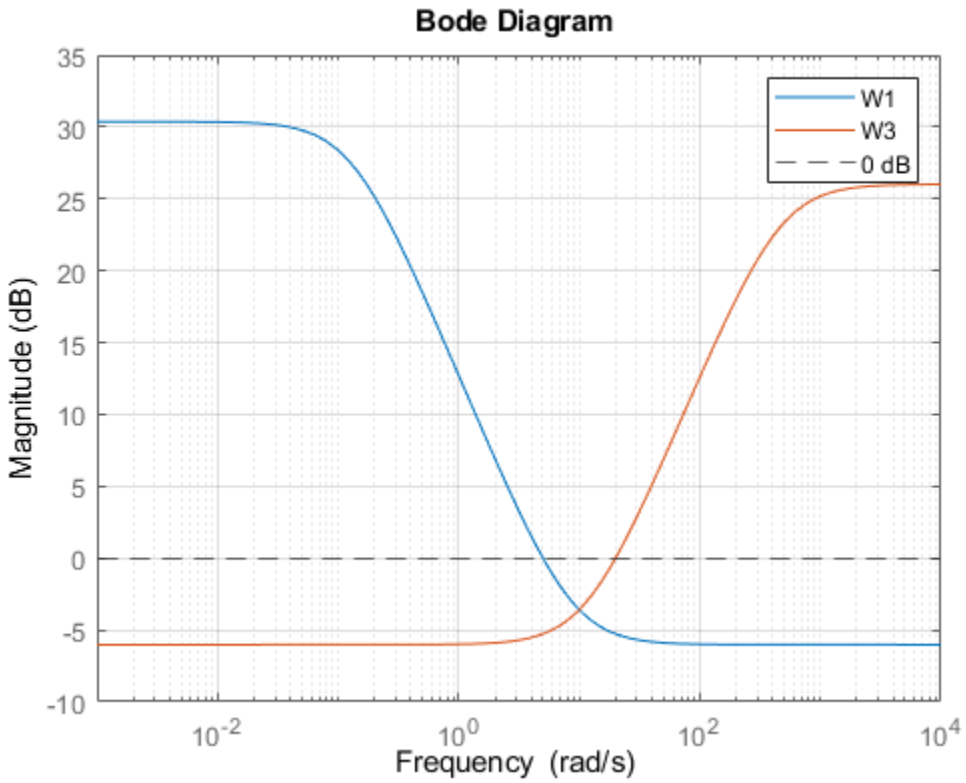
```
W1 = makeweight(33,5,0.5);
```

For robustness and noise attenuation, shape `T` to be small outside the control bandwidth, which means choosing `W3` large at high frequency.

```
W3 = makeweight(0.5,20,20);
```

Examine both weighting functions. Their inverses are the target shapes for `S` and `T`.

```
bodemag(W1,W3)
yline(0,'--');
legend('W1','W3','0 dB')
grid on
```



Because `S + T = I`, `mixsyn` cannot make both `S` and `T` small (less than 0 dB) in the same frequency range. Therefore, when you specify weights, there must be a frequency band in which both `W1` and `W3` are below 0 dB.

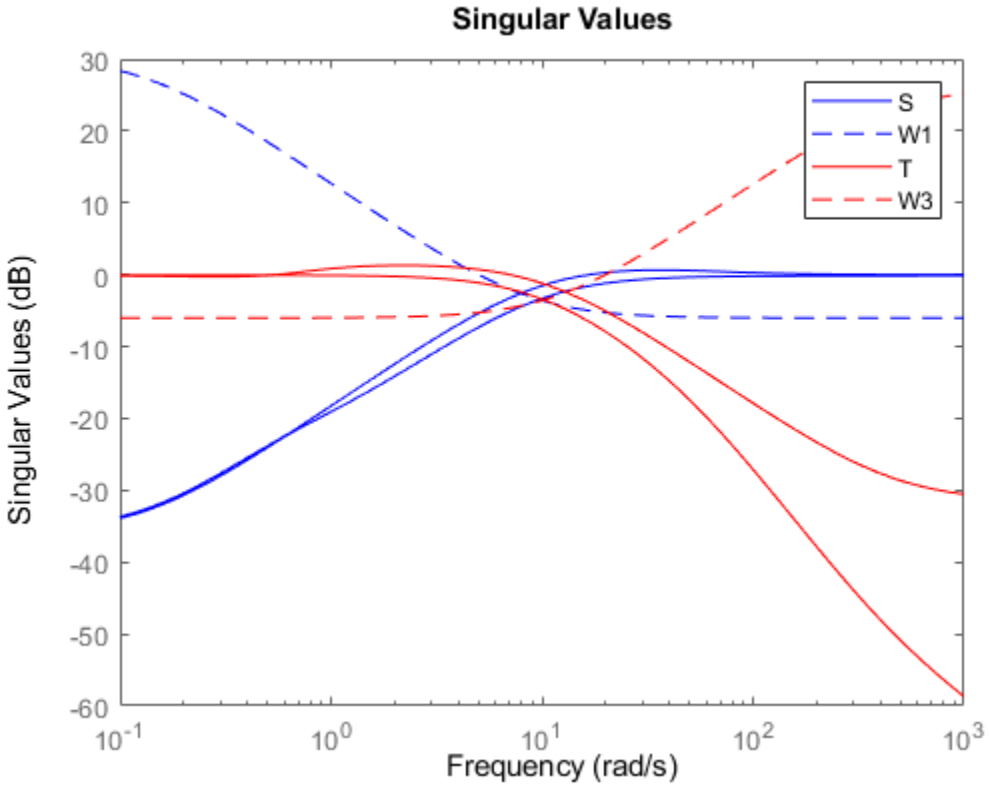
Use `mixsyn` to compute the optimal mixed-sensitivity controller with these weights. For this example, impose no penalty on controller effort by setting `W2` to `[]`.

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

```
gamma = 0.7331
```

The resulting `gamma`, which is the peak singular value across all frequencies, is well below 1, indicating that the closed-loop system meets the design requirements. Examine the resulting system responses. First, compare the resulting sensitivity `S` and complementary sensitivity `T` to the corresponding weighting functions `W1` and `W3`.

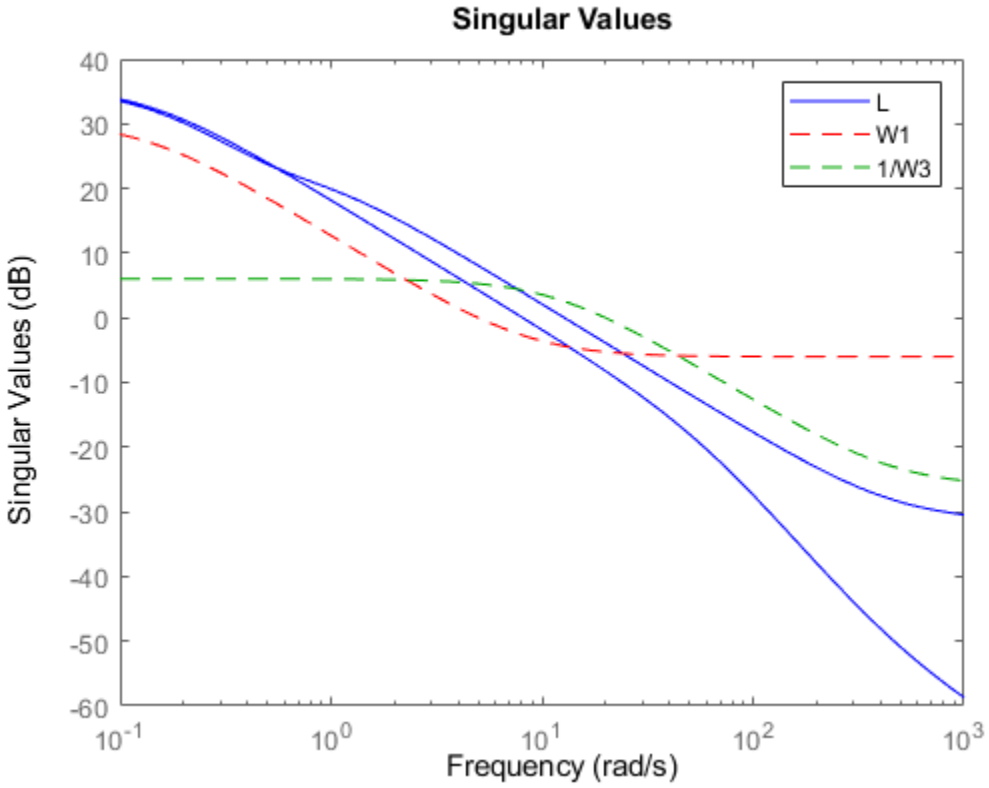
```
L = G*K;
I = eye(size(L));
S = feedback(I,L);
T= I-S;

sigma(S,'b',W1,'b--',T,'r',W3,'r--',{0.1,1000})
legend('S','W1','T','W3')
```



The plot shows that S and T achieve the desired loop shape, where S is large inside the control bandwidth and a is small outside the control bandwidth.

To see how mixed-sensitivity loop-shaping achieves the goals of classic loop shaping, compare the open-loop response L to the weighting functions. `L ~ W1` where `W1` is large, and `L ~ 1/W3` where `W3` is large.

```
sigma(L,'b',W1,'r--',1/W3,'g--',{0.1,1000})
legend('L','W1','1/W3')
```



See Also

`mixsyn` | `augw`

Related Examples

- “Loop-Shaping Controller Design” on page 2-8

Loop Shaping Using the Glover-McFarlane Method

This example shows how to use `ncfsyn` to shape the open-loop response while enforcing stability and maximizing robustness. `ncfsyn` measures robustness in terms of the normalized coprime stability margin computed by `ncfmargin`.

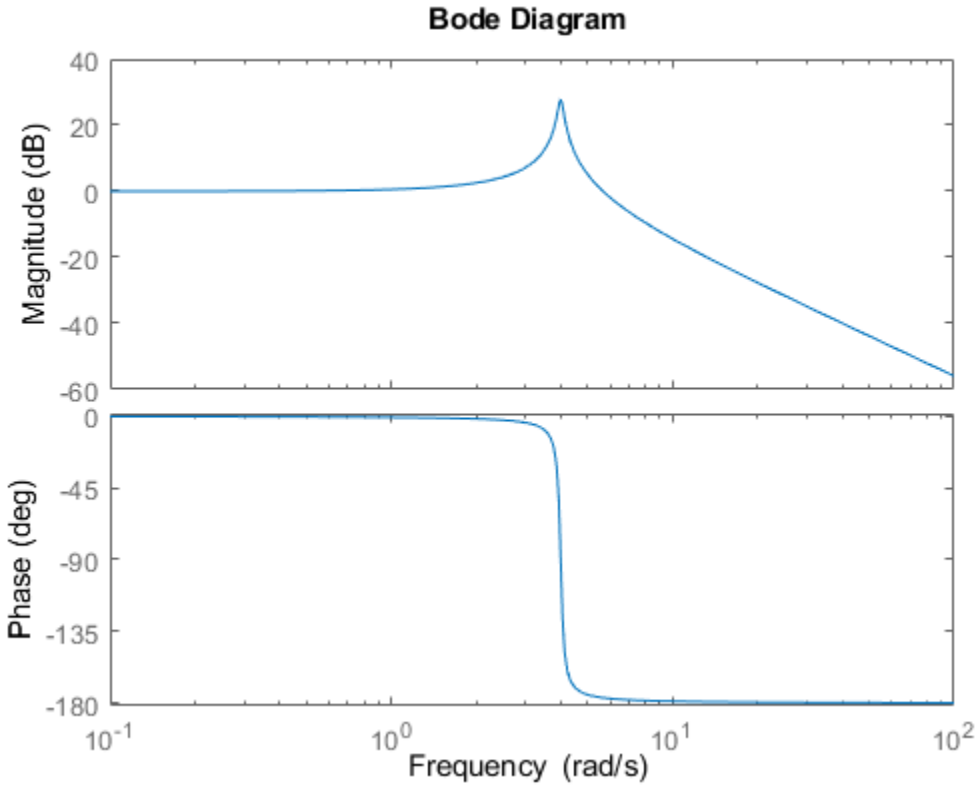
Plant Model

The plant model is a lightly damped, second-order system.

$$P(s) = \frac{16}{s^2 + 0.16s + 16}.$$

A Bode plot shows the resonant peak.

```
P = tf(16,[1 0.16 16]);
bode(P)
```



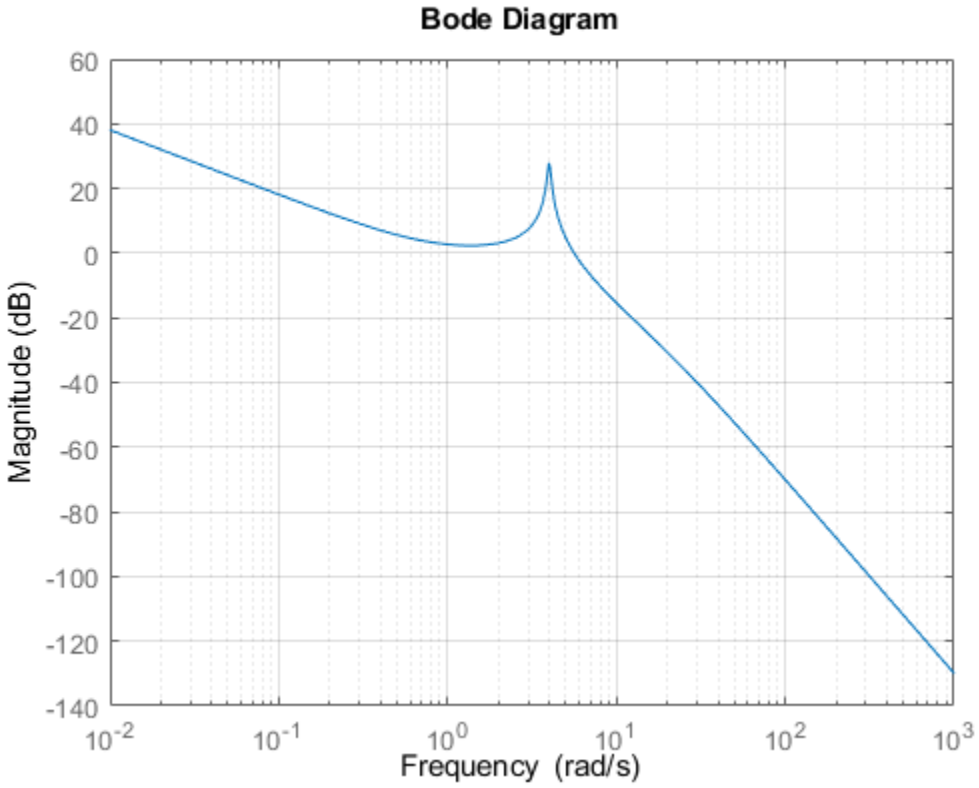
Design Objectives and Initial Compensator Design

The design objectives for the closed-loop are the following.

- Insensitivity to noise, including 60dB/decade attenuation beyond 20 rad/sec
- Integral action and a bandwidth of at least 0.5 rad/s
- Gain crossover frequencies no larger than 7 rad/s

In loop-shaping control design, you translate these requirements into a desired shape for the open-loop gain and seek a compensator that enforces this shape. For example, a compensator consisting of a PI term in series with a high-frequency lag component achieves the desired loop shape.

```
K_PI = pid(1,0.8);
K_rolloff = tf(1,[1/20 1]);
Kprop = K_PI*K_rolloff;
bodemag(P*Kprop); grid
```



Unfortunately, the compensator `Kprop` does not stabilize the closed-loop system. Examining the closed-loop dynamics shows poles in the right half-plane.

```
pole(feedback(P*Kprop,1))
```

```
ans = 4×1 complex

 -20.6975 + 0.0000i
   0.4702 + 5.5210i
   0.4702 - 5.5210i
  -0.4029 + 0.0000i
```

Enforcing Stability and Robustness with `ncfsyn`

You can use `ncfsyn` to enforce stability and adequate stability margins without significantly altering the loop shape. Use the initial design `Kprop` as loop-shaping pre-filter. `ncfsyn` assumes a positive feedback control system (see `ncfsyn`), so flip the sign of `Kprop` and of the returned controller.

```
[K,~,gamma] = ncfsyn(P,-Kprop);
K = -K;   % flip sign back
gamma
```

```
gamma = 1.9903
```

A value of the performance `gamma` less than 3 indicates success (modest gain degradation along with acceptable robustness margins). The new compensator `K` stabilizes the plant and has good stability margins.

```
allmargin(P*K)
```

```
ans = struct with fields:
     GainMargin: [6.2984 10.9082]
    GMFrequency: [1.6108 15.0285]
    PhaseMargin: [79.9812 -99.6214 63.7590]
    PMFrequency: [0.4467 3.1469 5.2304]
    DelayMargin: [3.1253 1.4441 0.2128]
    DMFrequency: [0.4467 3.1469 5.2304]
         Stable: 1
```

With `gamma` approximately 2, the expect at most `20*log10(gamma)` = 6dB gain reduction in the high-gain region and at most 6dB gain increase in the low-gain region. The Bode magnitude plot confirms this. Note that `ncfsyn` modifies the loop shape mostly around the gain crossover to achieve stability and robustness.

```
subplot(1,2,1)
bodemag(Kprop,'r',K,'g',{1e-2,1e4}); grid
legend('Initial design','NCFSYN design')
title('Controller Gains')
subplot(1,2,2)
bodemag(P*Kprop,'r',P*K,'g',{1e-3,1e2}); grid
legend('Initial design','NCFSYN design')
title('Open-Loop Gains')
```

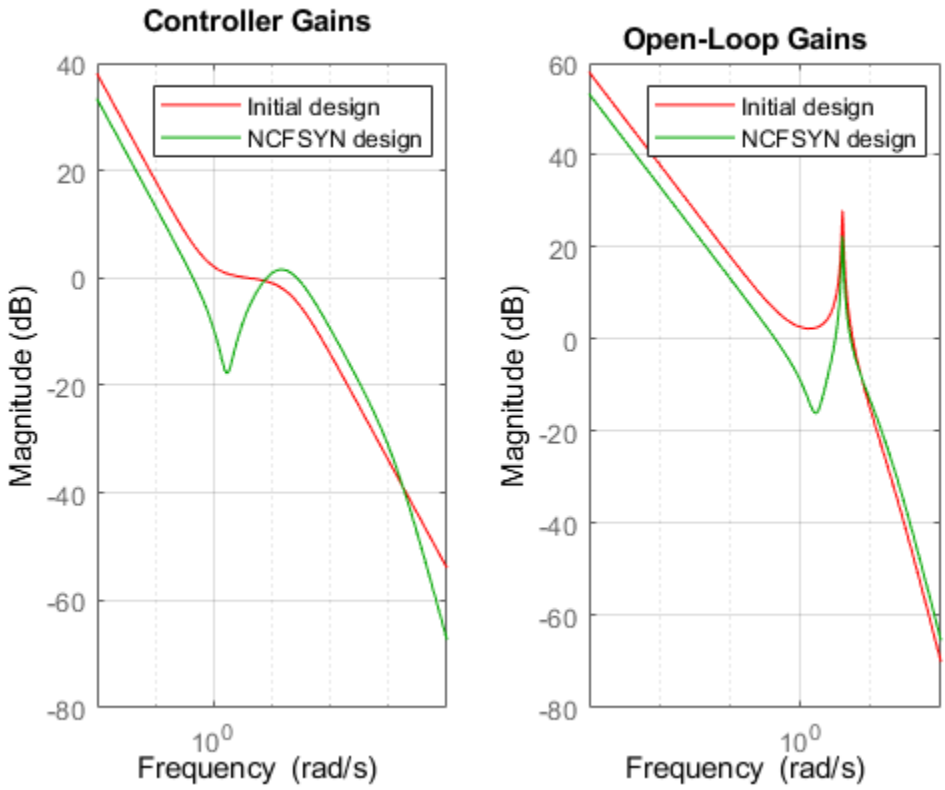
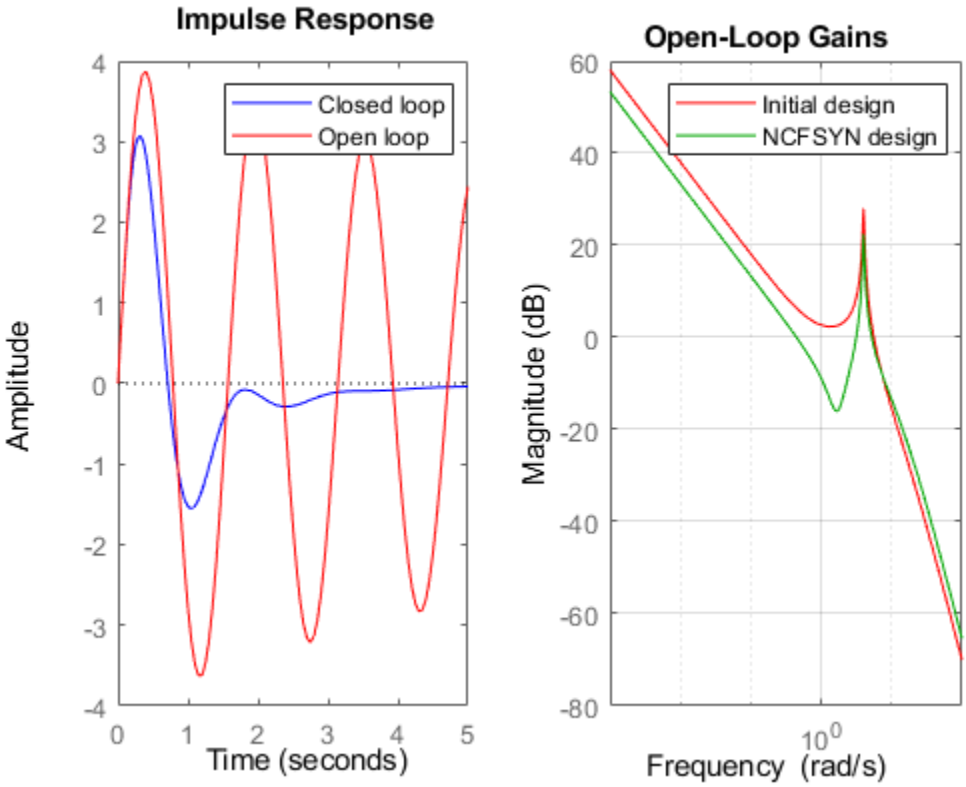


Figure 1: Compensator and open-loop gains.

Impulse Response

With the `ncfsyn` compensator, an impulse disturbance at the plant input is damped out in a few seconds. Compare this response to the uncompensated plant response.

```
subplot(1,2,1)
impulse(feedback(P,K),'b',P,'r',5);
legend('Closed loop','Open loop')
```




```
subplot(1,2,2);
impulse(-feedback(K*P,1),'b',5)
title('Control action')
```

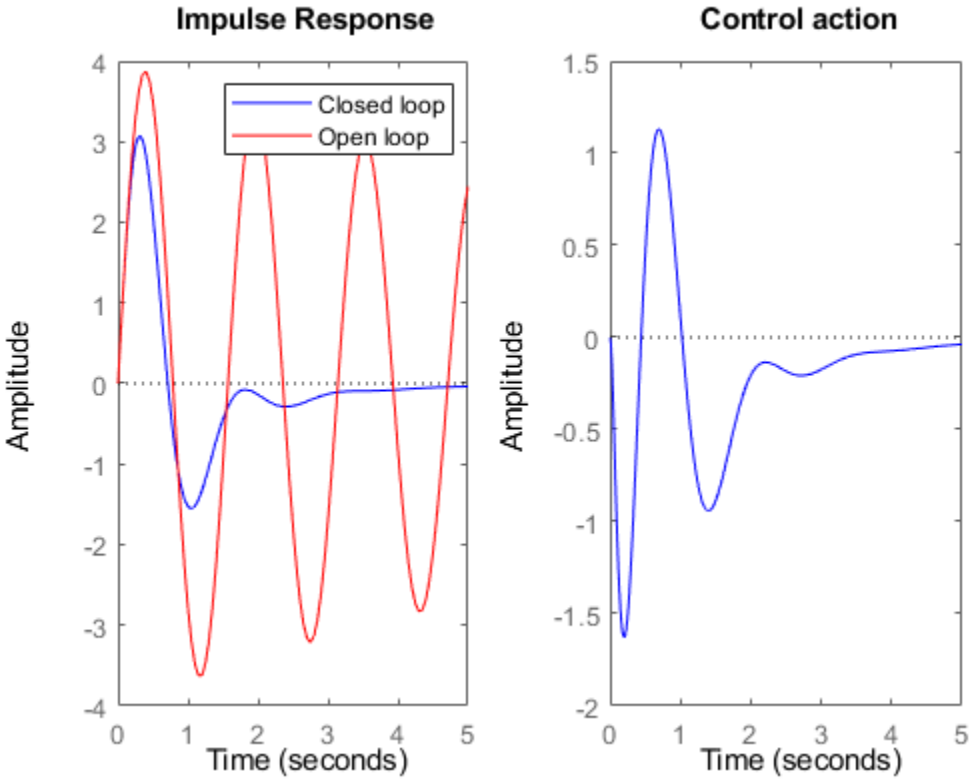
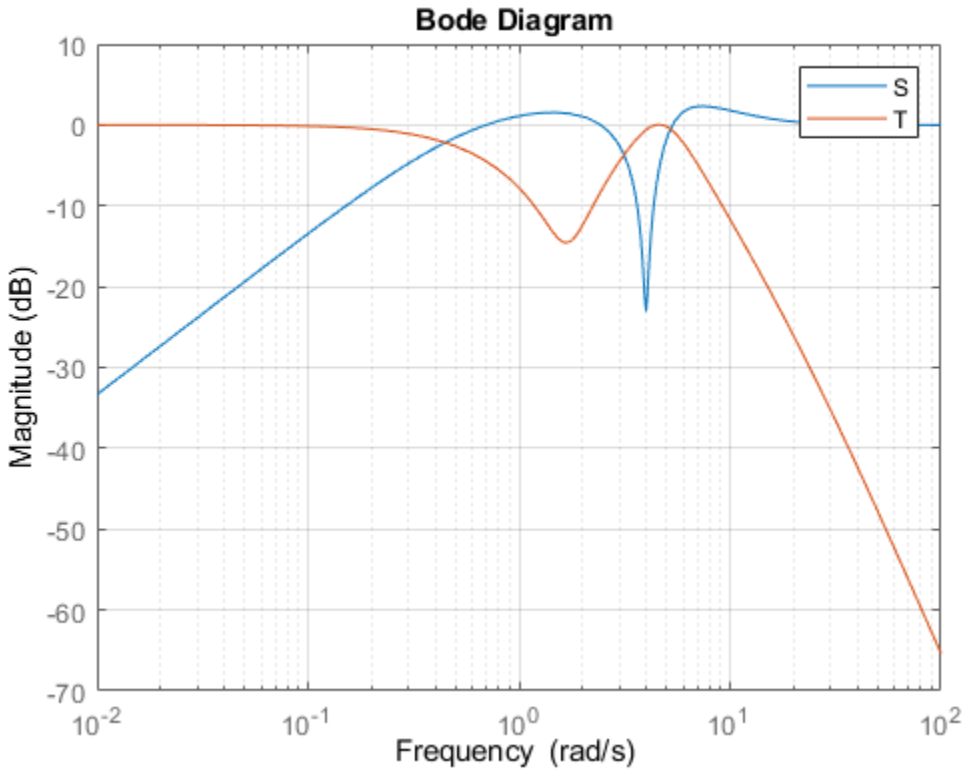


Figure 2: Response to impulse at plant input.

Sensitivity Functions

The closed-loop sensitivity and complementary sensitivity functions show the desired sensitivity reduction and high-frequency noise attenuation expressed in the closed-loop performance objectives.

```
S = feedback(1,P*K);
T = 1-S;
clf
bodemag(S,T,{1e-2,1e2}), grid
legend('S','T')
```



Conclusion

In this example, you used the function `ncfsyn` to adjust a hand-shaped compensator to achieve closed-loop stability while approximately preserving the desired loop shape.

See Also

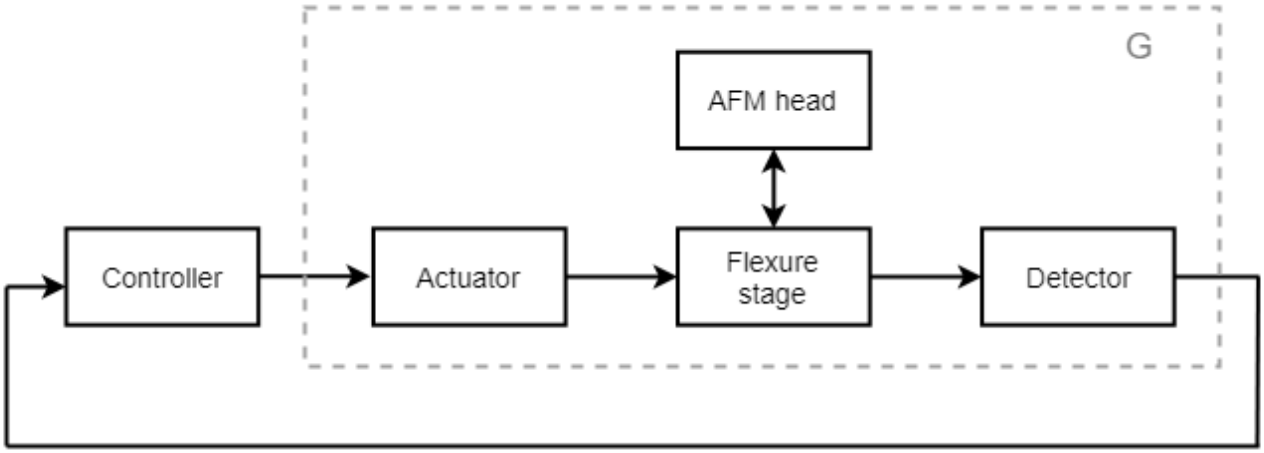
`ncfsyn` | `ncfmargin`

Robust Loop Shaping of Nanopositioning Control System

This example shows how to use the Glover-McFarlane technique to obtain loop-shaping compensators with good stability margins. The example applies the technique to a nanopositioning stage. These devices can achieve very high precision positioning which is important in applications such as atomic force microscopes (AFMs). For more details on this application, see [1].

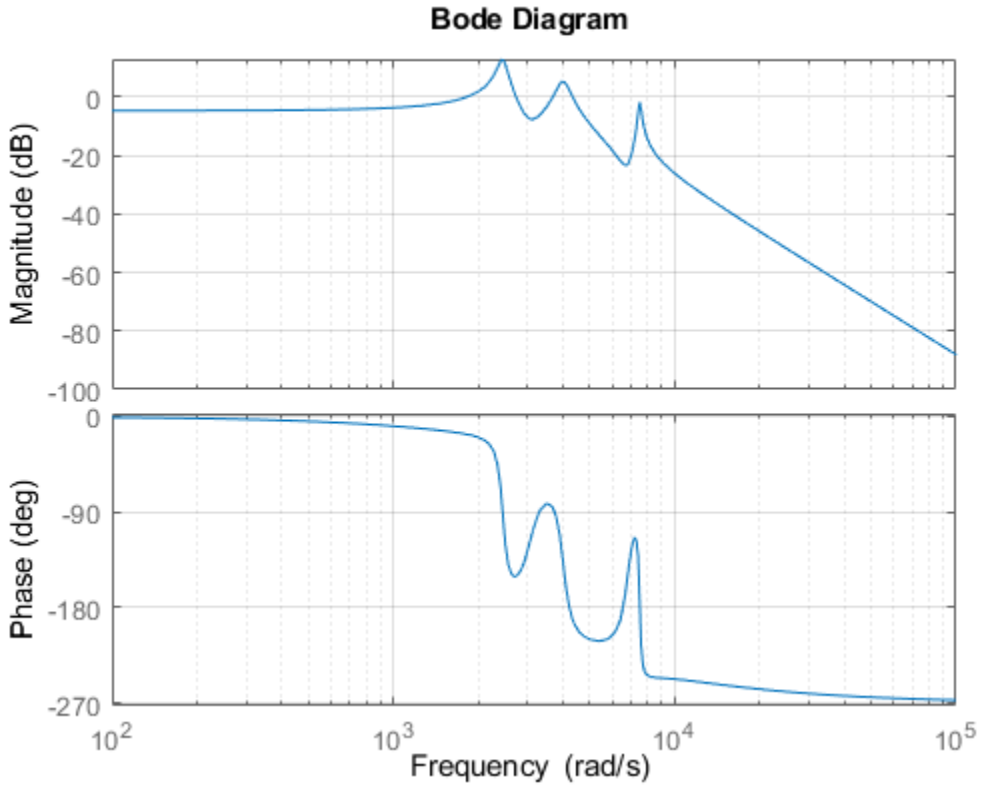
Nanopositioning System

The following illustration shows a feedback diagram of a nanopositioning device. The system consists of piezo-electric actuation, a flexure stage, and a detection system. The flexure stage interacts with the head of the AFM.



Load the plant model for the nanopositioning stage. This model is a seventh-order state-space model fitted to frequency response data obtained from the device.

```
load npfit A B C D
G = ss(A,B,C,D);
bode(G), grid
```



Typical design requirements for the control law include high bandwidth, high resolution, and good robustness. For this example, use:

- Bandwidth of approximately 50 Hz
- Roll-off of -40 dB/decade past 250 Hz
- Gain margin in excess of 1.5 (3.5 dB) and phase margin in excess of 60 degrees

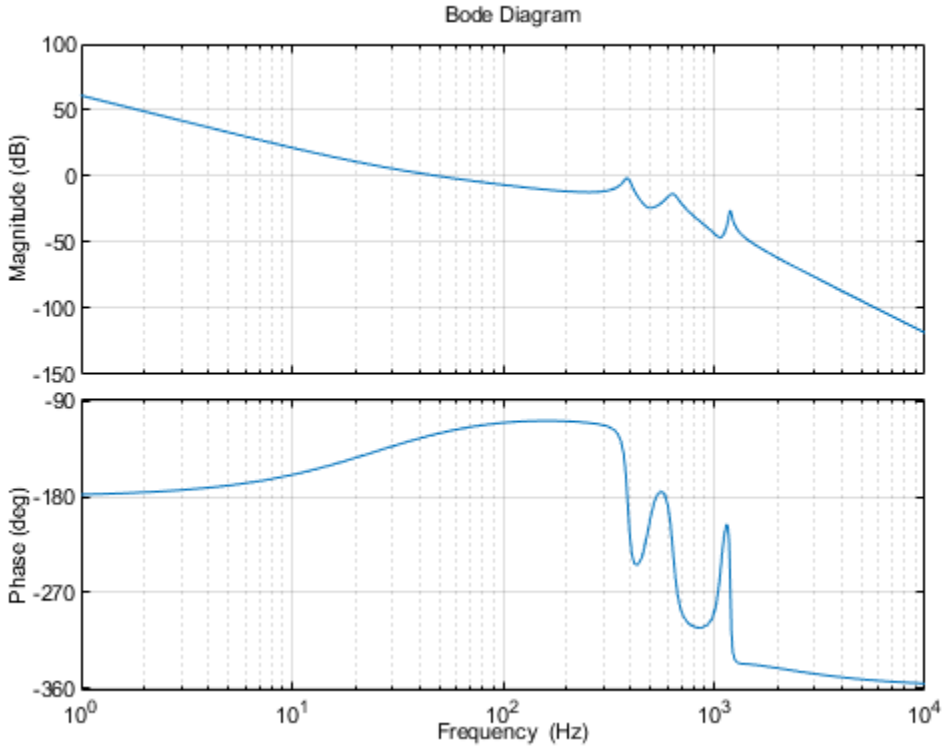
Additionally, when the nanopositioning stage is used for scanning, the reference signal is triangular, and it is important that the stage tracks this signal with minimal error in the midsection of the triangular wave. One way of enforcing this is to add the following design requirement:

- A double integrator in the control loop

PI Design

First try a PI design. To accommodate the double integrator requirement, multiply the plant by $1/s$. Set the desired bandwidth to 50 Hz. Use `pidtune` to automatically tune the PI controller.

```
Integ = tf(1,[1 0]);
bw = 50*2*pi;  % 50 Hz in rad/s
PI = pidtune(G*Integ,'pi',50*2*pi);
C = PI*Integ;

bopt = bodeoptions;
bopt.FreqUnits = 'Hz';  bopt.XLim = [1e0 1e4];
bodeplot(G*C,bopt), grid
```



This compensator meets the bandwidth requirement and almost meets the roll-off requirement. Use `allmargin` to calculate the stability margins.

```
allmargin(G*C)
```

```
ans = struct with fields:
     GainMargin: [0 1.1531 13.7832 7.4195 Inf]
    GMFrequency: [0 2.4405e+03 3.3423e+03 3.7099e+03 Inf]
    PhaseMargin: 60.0024
    PMFrequency: 314.1959
    DelayMargin: 0.0033
    DMFrequency: 314.1959
         Stable: 1
```

The phase margin is satisfactory, but the smallest gain margin is only 1.15, far below the target of 1.5. You could try adding a lowpass filter to roll off faster beyond the gain crossover frequency, but this would most likely reduce the phase margin.

Glover-McFarlane Loop Shaping

The Glover-McFarlane technique provides an easy way to tweak the candidate compensator `C` to improve its stability margins. This technique seeks to maximize robustness (as measured by `ncfmargin`) while roughly preserving the loop shape of `G*C`. Use `ncfsyn` to apply this technique to this application. Note that `ncfsyn` assumes positive feedback so you need to flip the sign of the plant `G`.

```
[K,~,gam] = ncfsyn(-G,C);
```

Check the stability margins with the refined compensator K.

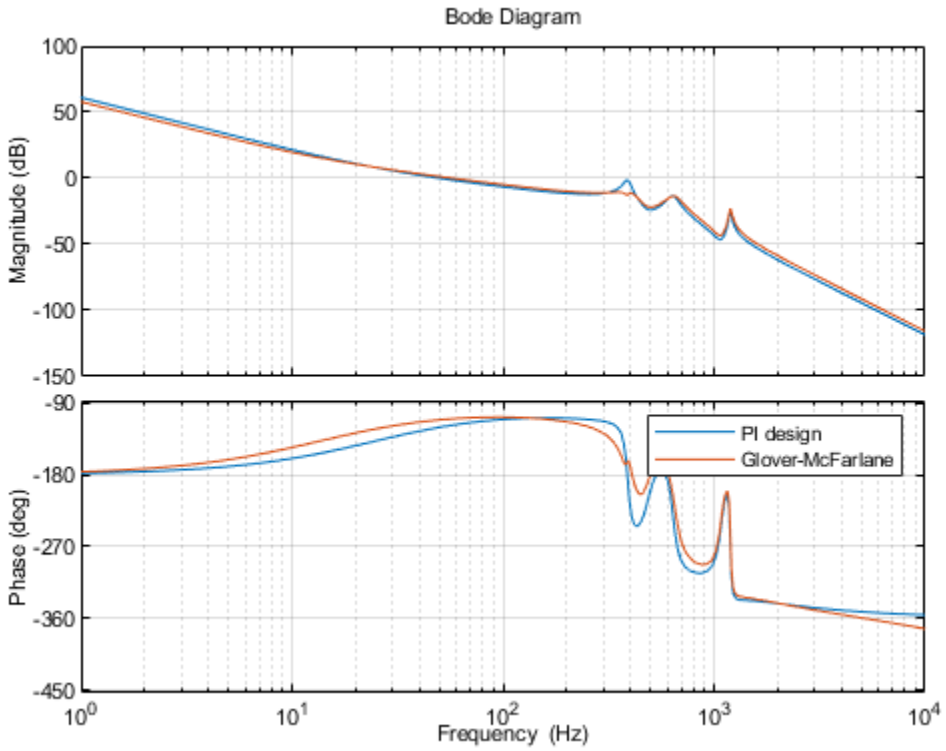
```
[Gm,Pm] = margin(G*K)
```

```
Gm = 3.7267
```

```
Pm = 70.7109
```

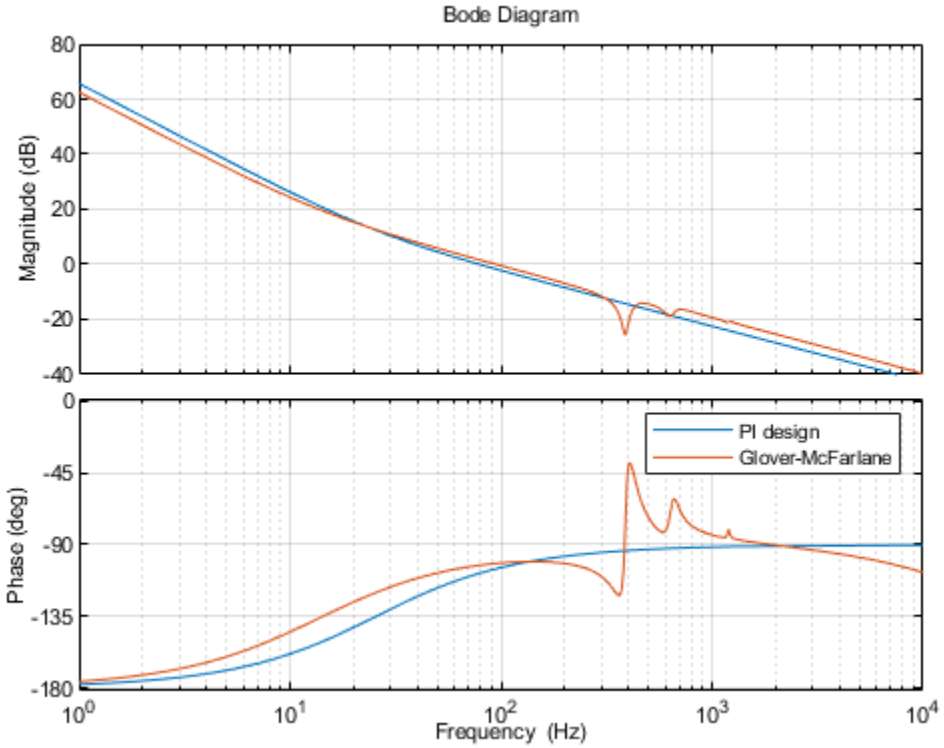
The `ncfsyn` compensator increases the gain margin to 3.7 and the phase margin to 70 degrees. Compare the loop shape for this compensator with the loop shape for the PI design.

```
bodeplot(G*C,G*K,bopt), grid
legend('PI design','Glover-McFarlane')
```



The Glover-McFarlane compensator attenuates the first resonance responsible for the weak gain margin while boosting the lead effect to preserve and even improve the phase margin. This refined design meets all requirements. Compare the two compensators.

```
bodeplot(C,K,bopt), grid
legend('PI design','Glover-McFarlane')
```



The refined compensator has roughly the same gain profile. `ncfsyn` automatically added zeros in the right places to accommodate the plant resonances.

Compensator Simplification

The `ncfsyn` algorithm produces a compensator of relatively high order compared to the original second-order design.

```
order(K)
```

```
ans = 11
```

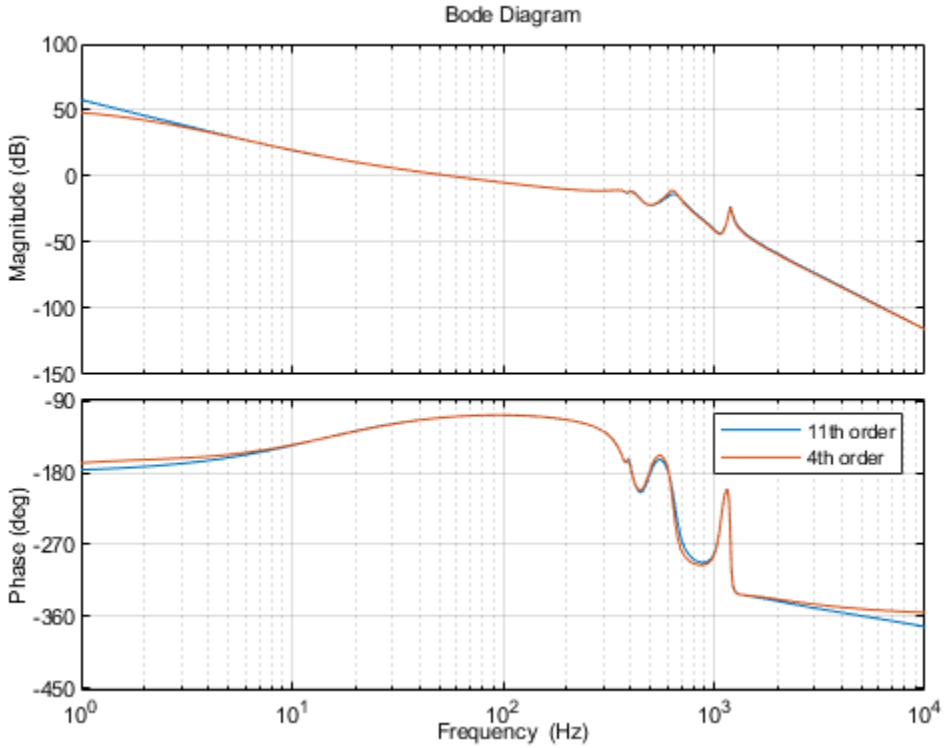
You can use `ncfmr` to reduce this down to something close to the original order. For example, try order 4.

```
ord = 4;
Kr = ncfmr(K,ord);
[Gm,Pm] = margin(G*Kr)
```

```
Gm = 3.8139
```

```
Pm = 70.6770
```

```
bodeplot(G*K,G*Kr,bopt), grid
legend('11th order','4th order')
```



The reduced-order compensator `Kr` has very similar loop shape and stability margins and is a reasonable candidate for implementation.

References

1 Salapaka, S., A. Sebastian, J. P. Cleveland, and M. V. Salapaka. “High Bandwidth Nano-Positioner: A Robust Control Approach.” *Review of Scientific Instruments* 73, no. 9 (September 2002): 3232–41.

See Also

`ncfsyn` | `ncfmargin` | `ncfmr`

More About

- “Loop Shaping Using the Glover-McFarlane Method” on page 2-32

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.

See Also

Related Examples

- “Hankel Singular Values” on page 3-3

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] on page 1-23

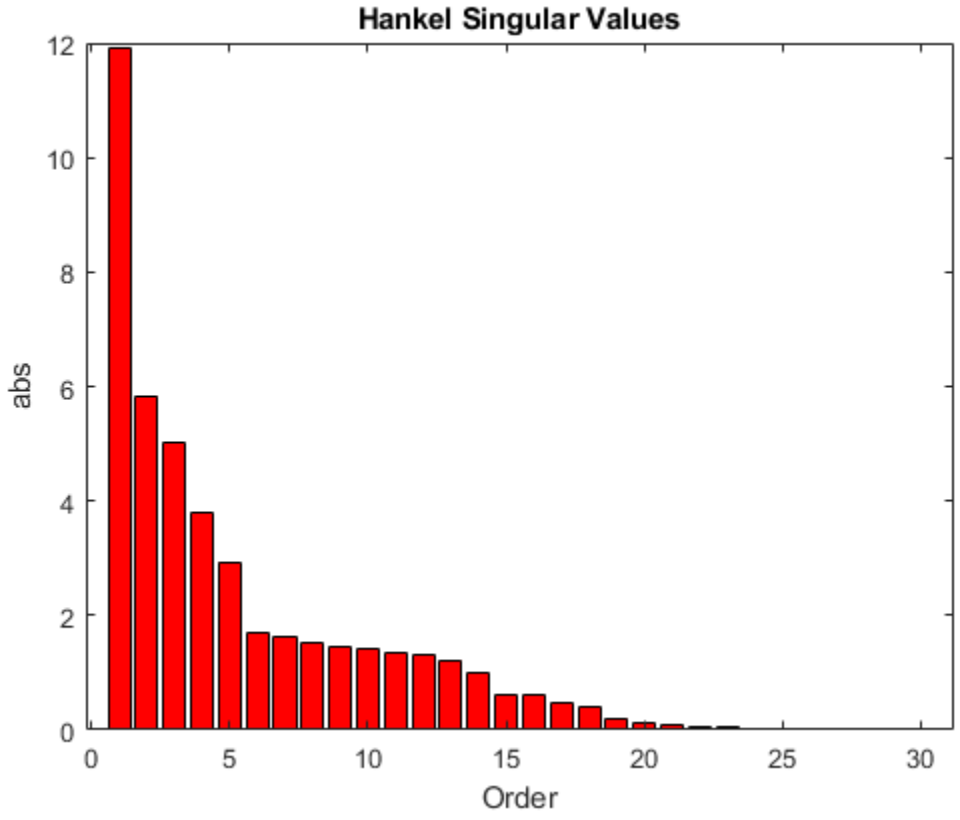
$$\sigma_H = \sqrt{\lambda_i(PQ)}$$

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 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.

See Also

Related Examples

- “Approximate Plant Model by Additive Error Methods” on page 3-7
- “Approximate Plant Model by Multiplicative Error Method” on page 3-9

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

These theoretical bounds are based on the “tails” of the Hankel singular values of the model, which are given as follows.

- Additive error bound:

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

  Here, σ_i are denoted the ith Hankel singular value of the original system `G`.
- Multiplicative (relative) error bound:

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

  Here, σ_i are denoted the ith Hankel singular value of the phase matrix of the model `G` (see the `bstmr` reference page).

Commands for Model Reduction

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

See Also

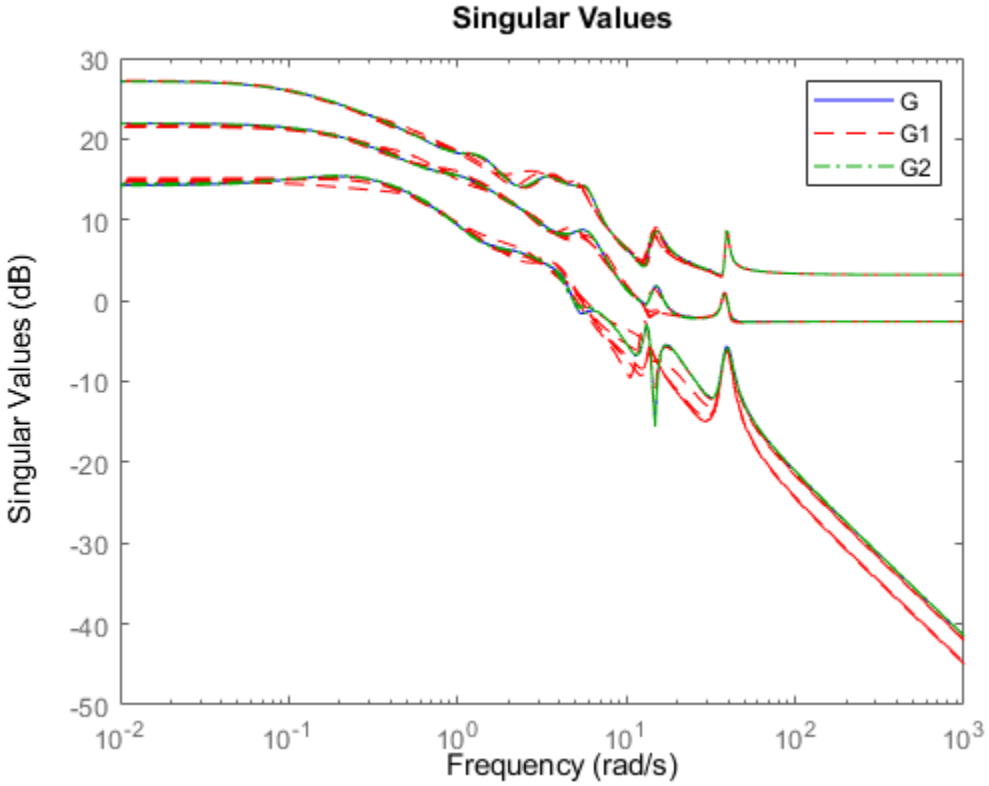
Related Examples

- “Approximate Plant Model by Additive Error Methods” on page 3-7
- “Approximate Plant Model by Multiplicative Error Method” on page 3-9

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')
```

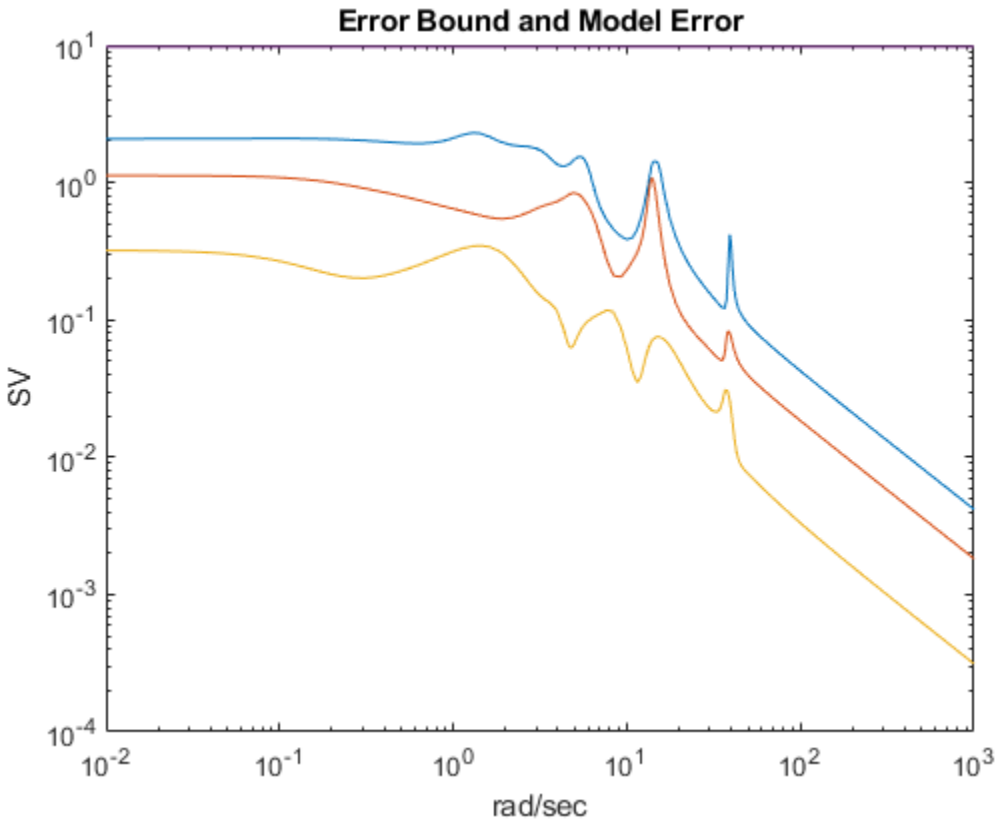
```
ans = 2.2965
```

```
info1.ErrorBound(1)
```

```
ans = 9.7120
```

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')
```



See Also

`balancmr`

Related Examples

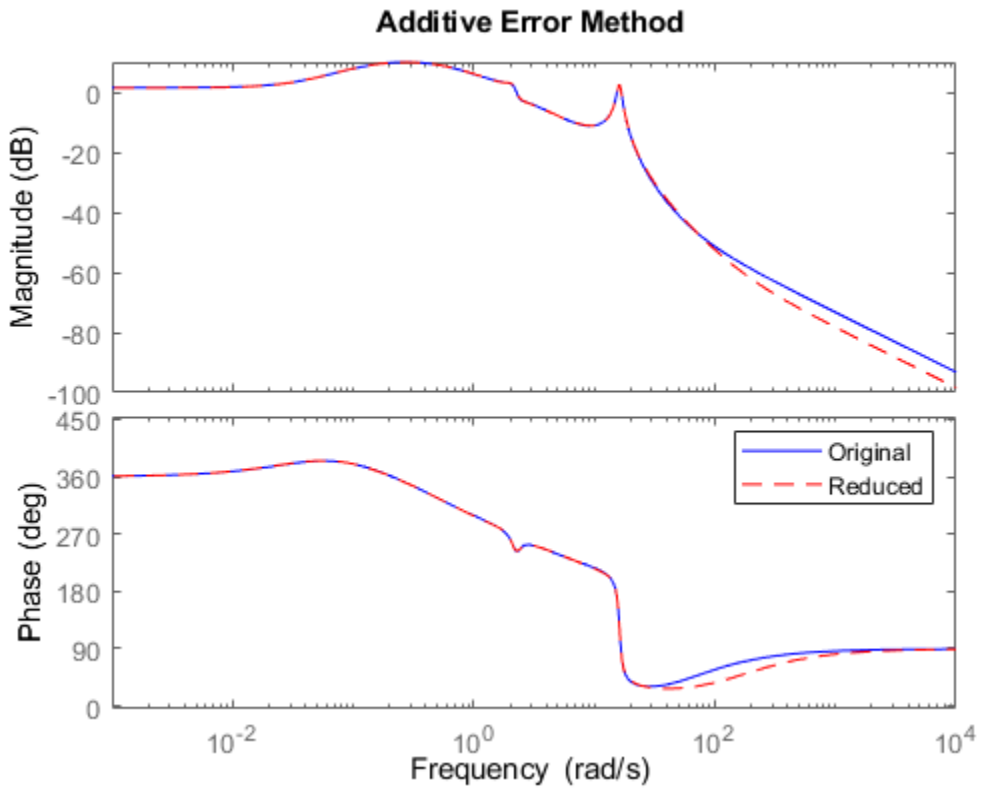
- “Model Reduction Techniques” on page 3-5
- “Approximate Plant Model by Multiplicative Error Method” on page 3-9

Approximate Plant Model by Multiplicative Error Method

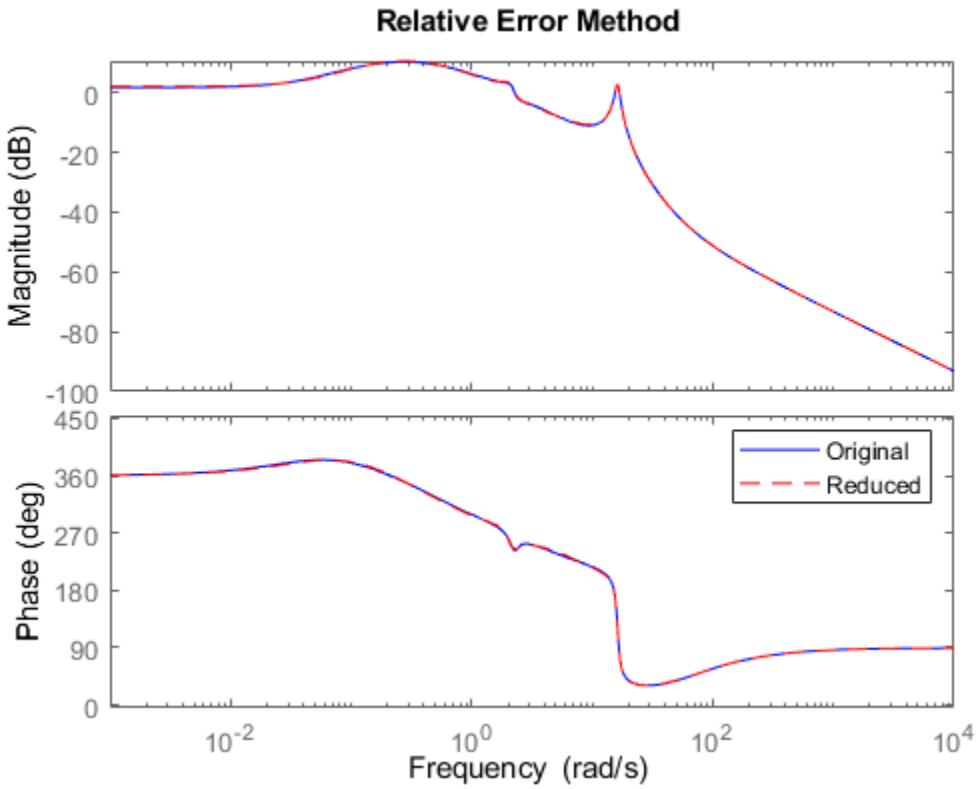
In most cases, the multiplicative error model reduction method `bstmr` tends to bound the relative error between the original and reduced-order models across the frequency range of interest, hence producing a more accurate reduced-order model than the additive error methods. This characteristic is obvious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

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

Related Examples

- “Model Reduction Techniques” on page 3-5
- “Approximate Plant Model by Additive Error Methods” on page 3-7

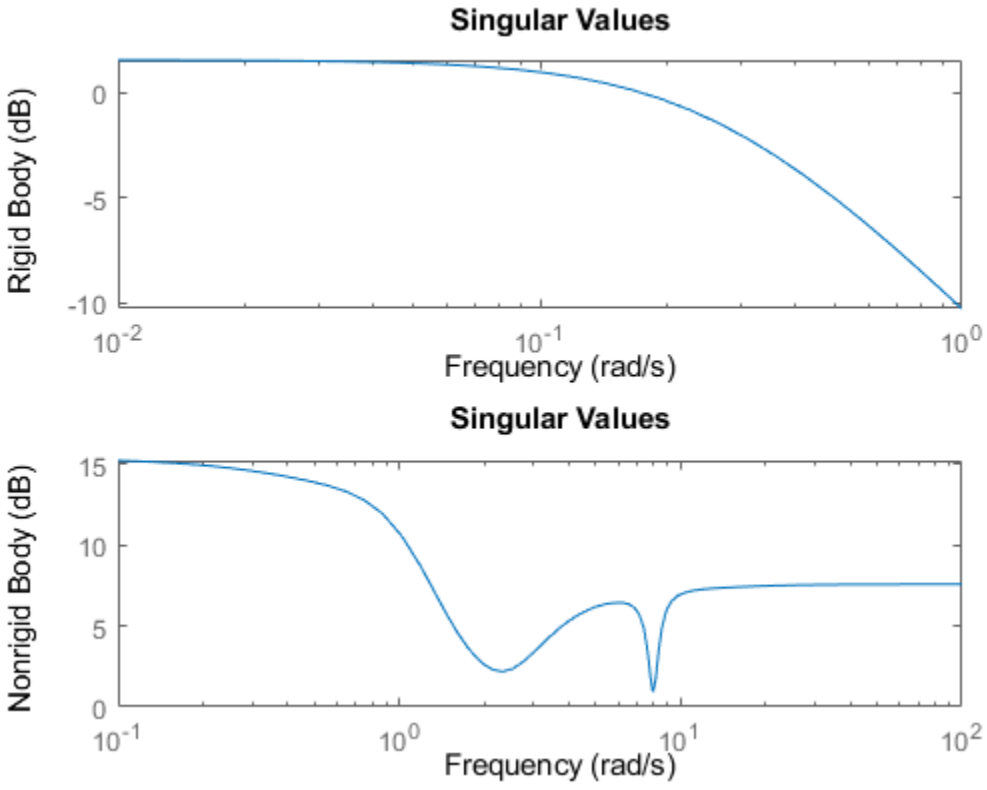
Using Modal Algorithms

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;             % put DC gain of G into G2
Gjw.D = 0;
subplot(2,1,1)
sigma(Gjw)
ylabel('Rigid Body')
subplot(2,1,2)
sigma(G2)
ylabel('Nonrigid Body')
```

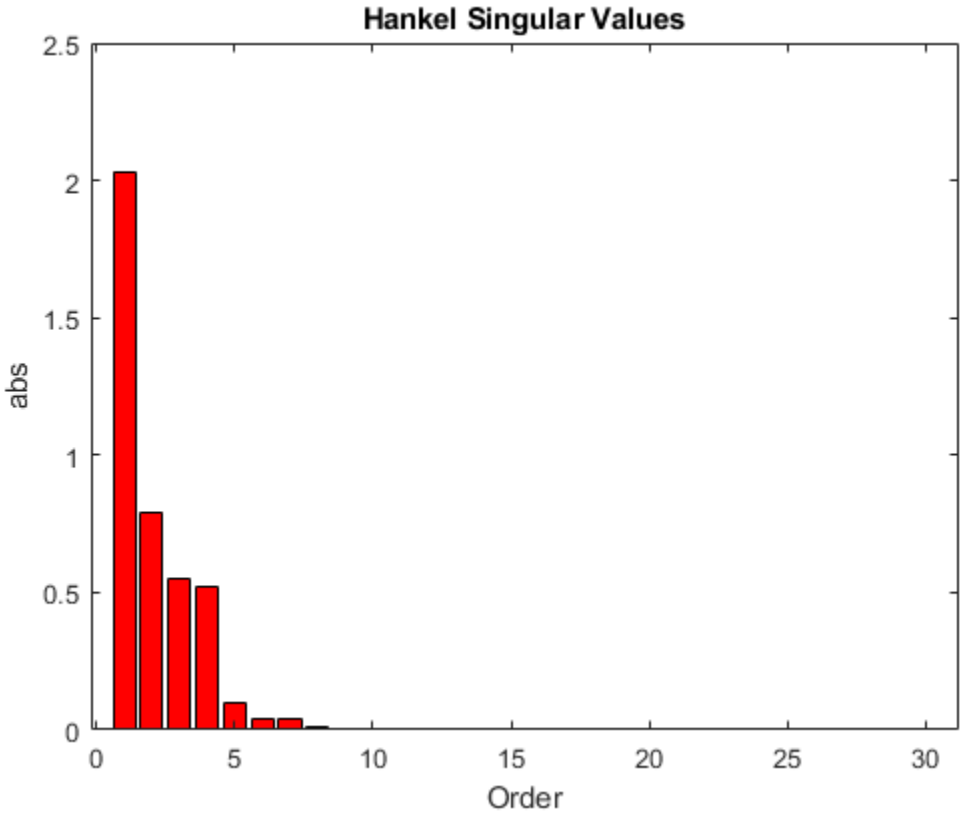


Further model reduction can be done on `G2` without any numerical difficulty. After `G2` is further reduced to `Gred`, the final approximation of the model is simply `Gjw+Gred`.

This process of splitting *jω*-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.

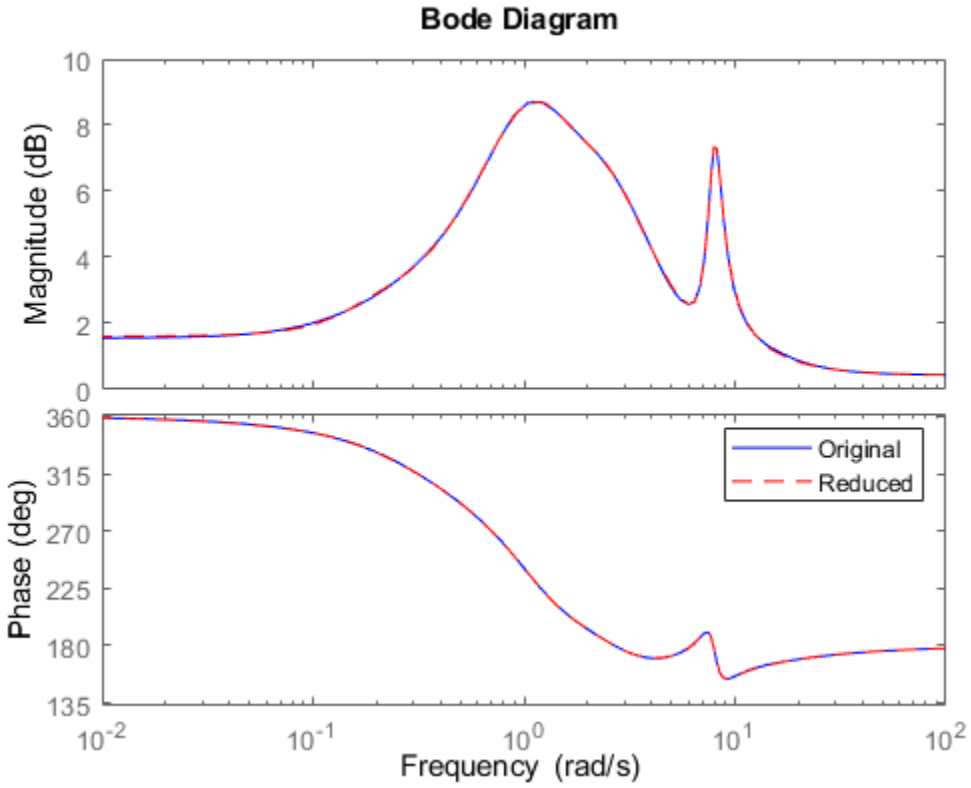
Examine the Hankel singular value plot.

```
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

`modreal` | `balancmr` | `schurmr` | `hankelmr` | `bstmr` | `hankelsv`

Related Examples

- “Model Reduction Techniques” on page 3-5
- “Reducing Large-Scale Models” on page 3-14

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.

See Also

`modreal`

Related Examples

- “Model Reduction Techniques” on page 3-5
- “Using Modal Algorithms” on page 3-11

Normalized Coprime Factor Reduction

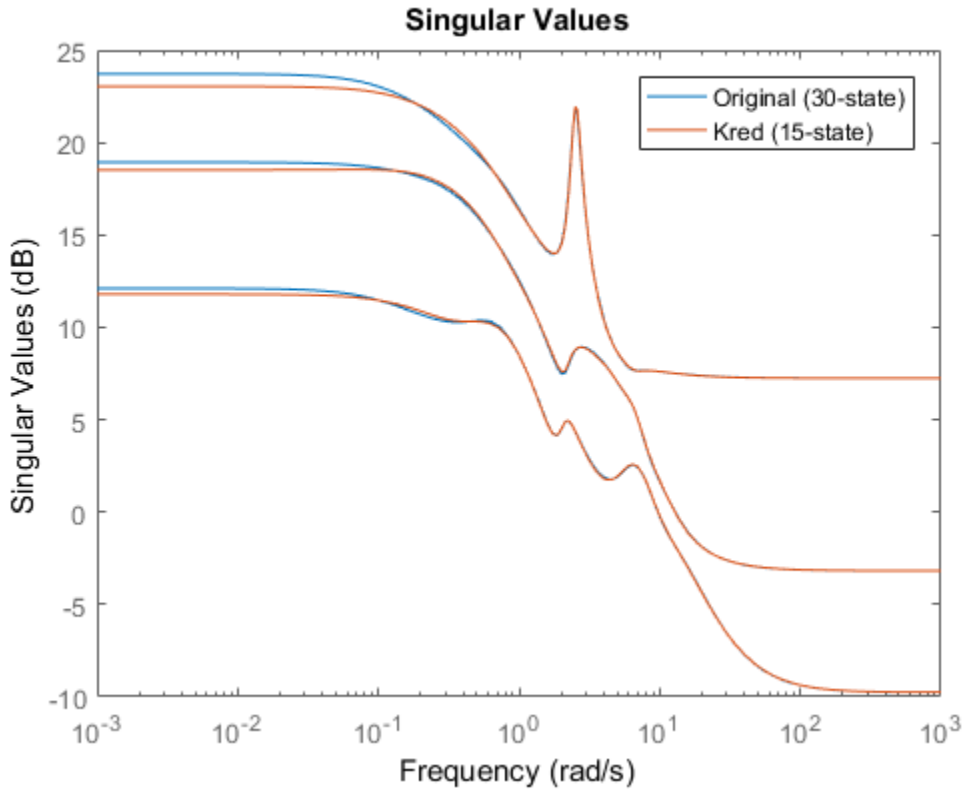
A special model reduction routine `ncfmr` produces a reduced-order model by truncating a balanced coprime set of a given model. It can directly simplify a modern controller with integrators to a smaller size by balanced truncation of the normalized coprime factors. It does not need `modreal` for pre-/postprocessing 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

`ncfmr` | `modreal` | `ncfmr`

Related Examples

- “Model Reduction Techniques” on page 3-5

Simplifying Higher-Order Plant Models

This example shows how to use Robust Control Toolbox™ to approximate high-order plant models by simpler, low-order models.

Introduction

Robust Control Toolbox offers tools to deal with large models such as:

- **High-Order Plants:** Detailed first-principles or finite-element models of your plant tend to have high order. Often we want to simplify such models for simulation or control design purposes.
- **High-Order Controllers:** Robust control techniques often yield high-order controllers. This is common, for example, when we use frequency-weighting functions for shaping the open-loop response. We will want to simplify such controllers for implementation.

For control purposes, it is generally enough to have an accurate model near the crossover frequency. For simulation, it is enough to capture the essential dynamics in the frequency range of the excitation signals. This means that it is often possible to find low-order approximations of high-order models. Robust Control Toolbox offers a variety of model-reduction algorithms to best suit your requirements and your model characteristics.

The Model Reduction Process

A model reduction task typically involves the following steps:

- Analyze the important characteristics of the model from its time or frequency-domain responses obtained from `step` or `bode`, for example.
- Determine an appropriate reduced order by plotting the Hankel singular values of the original model (`hankelsv`) to determine which modes (states) can be discarded without sacrificing the key characteristics.
- Choose a reduction algorithm. Some reduction methods available in the toolbox are: `balancmr`, `bstmr`, `schurmr`, `hankelmr`, and `ncfmr`

We can easily access these algorithms through the top-level interface `reduce`. The methods employ different measures of "closeness" between the original and reduced models. The choice is application-dependent. Let's try each of them to investigate their relative merits.

- Validation: We validate our results by comparing the dynamics of the reduced model to the original. We may need to adjust our reduction parameters (choice of model order, algorithm, error bounds etc.) if the results are not satisfactory.

Example: A Model for Rigid Body Motion of a Building

In this example, we apply the reduction methods to a model of the building of the Los Angeles University Hospital. The model is taken from SLICOT Working Note 2002-2, "A collection of benchmark examples for model reduction of linear time invariant dynamical systems," by Y. Chahlaoui and P.V. Dooren. It has eight floors, each with three degrees of freedom - two displacements and one rotation. We represent the input-output relationship for any one of these displacements using a 48-state model, where each state represents a displacement or its rate of change (velocity).

Let's load the data for the example:

```
load buildingData.mat
```

Examining the Plant Dynamics

Let's begin by analyzing the frequency response of the model:

```
bode(G)
grid on
```

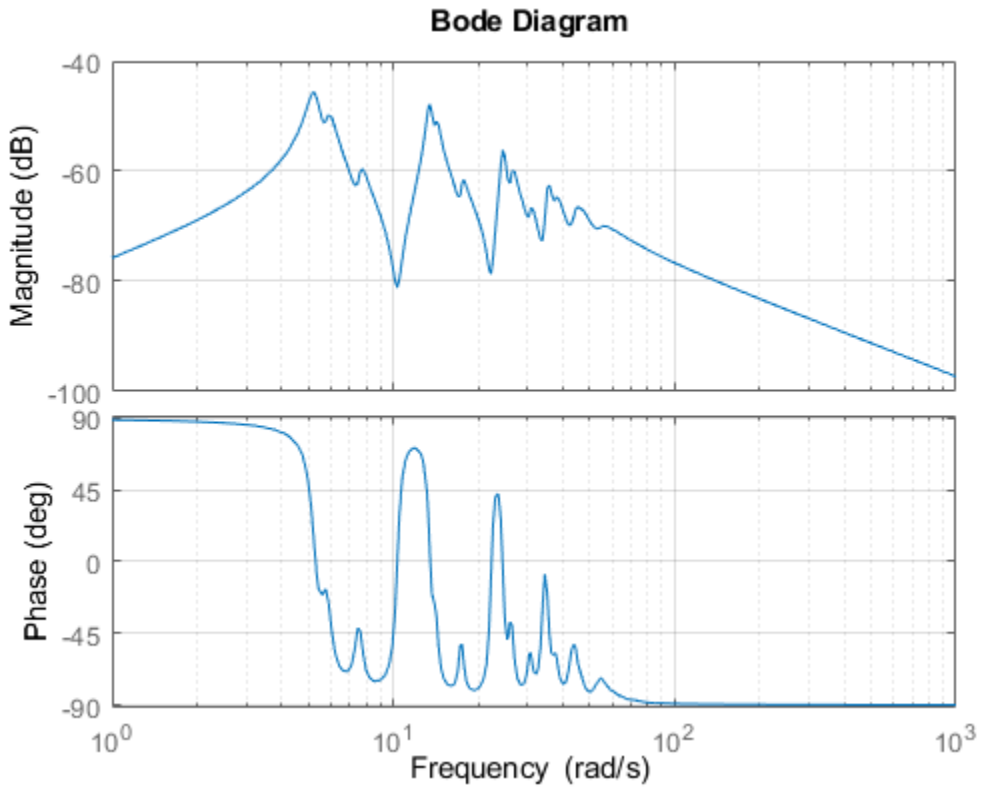


Figure 1: Bode diagram to analyze the frequency response

As observed from the frequency response of the model, the essential dynamics of the system lie in the frequency range of 3 to 50 radians/second. The magnitude drops in both the very low and the high-frequency ranges. Our objective is to find a low-order model that preserves the information content in this frequency range to an acceptable level of accuracy.

Computing Hankel Singular Values

To understand which states of the model can be safely discarded, look at the Hankel singular values of the model:

```
hsv_add = hankelsv(G);
bar(hsv_add)
title('Hankel Singular Values of the Model (G)');
xlabel('Number of States')
ylabel('Singular Values (\sigma_i)')
line([10.5 10.5],[0 1.5e-3],'Color','r','linestyle','--','linewidth',1)
text(6,1.6e-3,'10 dominant states.')
```

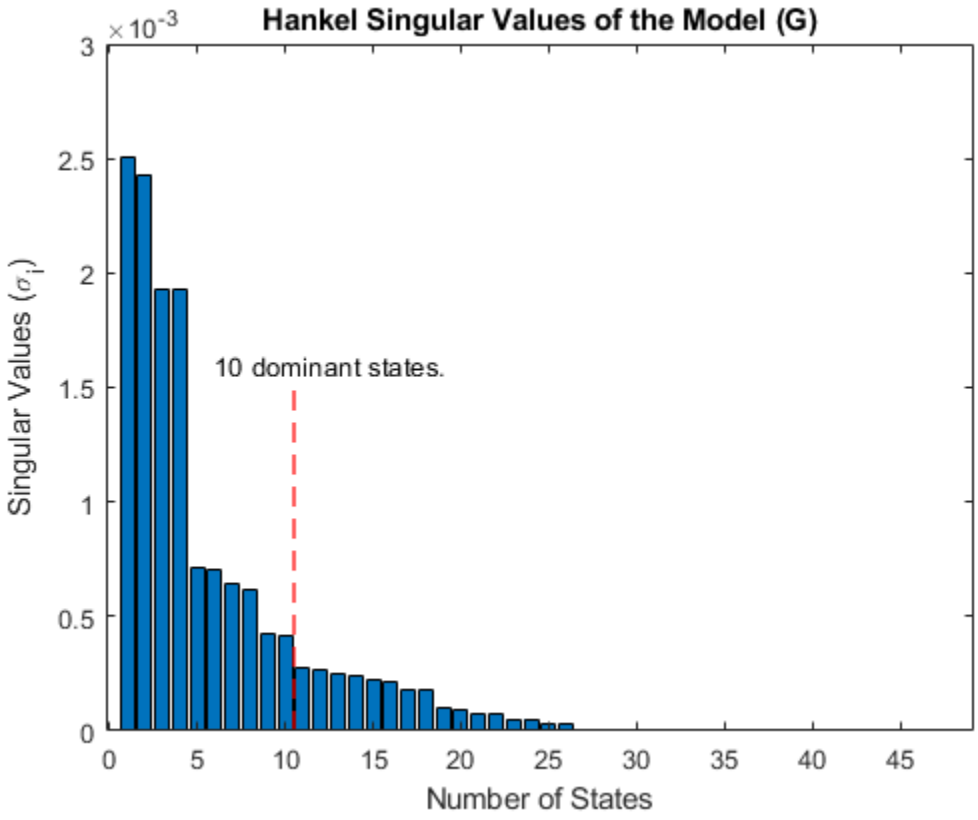


Figure 2: Hankel singular values of the model (G).

The Hankel singular value plot suggests that there are four dominant modes in this system. However, the contribution of the remaining modes is still significant. We'll draw the line at 10 states and discard the remaining ones to find a 10th-order reduced model `Gr` that best approximates the original system `G`.

Performing Model Reduction Using an Additive Error Bound

The function `reduce` is the gateway to all model reduction routines available in the toolbox. We'll use the default, square-root balance truncation ('balancmr') option of `reduce` as the first step. This method uses an "additive" error bound for reduction, meaning that it tries to keep the absolute approximation error uniformly small across frequencies.

```
% Compute 10th-order reduced model (reduce uses balancmr method by default)
[Gr_add,info_add] = reduce(G,10);

% Now compare the original model G to the reduced model Gr_add
bode(G,'b',Gr_add,'r')
grid on
title('Comparing Original (G) to the Reduced model (Gr\_add)')
legend('G - 48-state original ','Gr\_add - 10-state reduced','location','northeast')
```

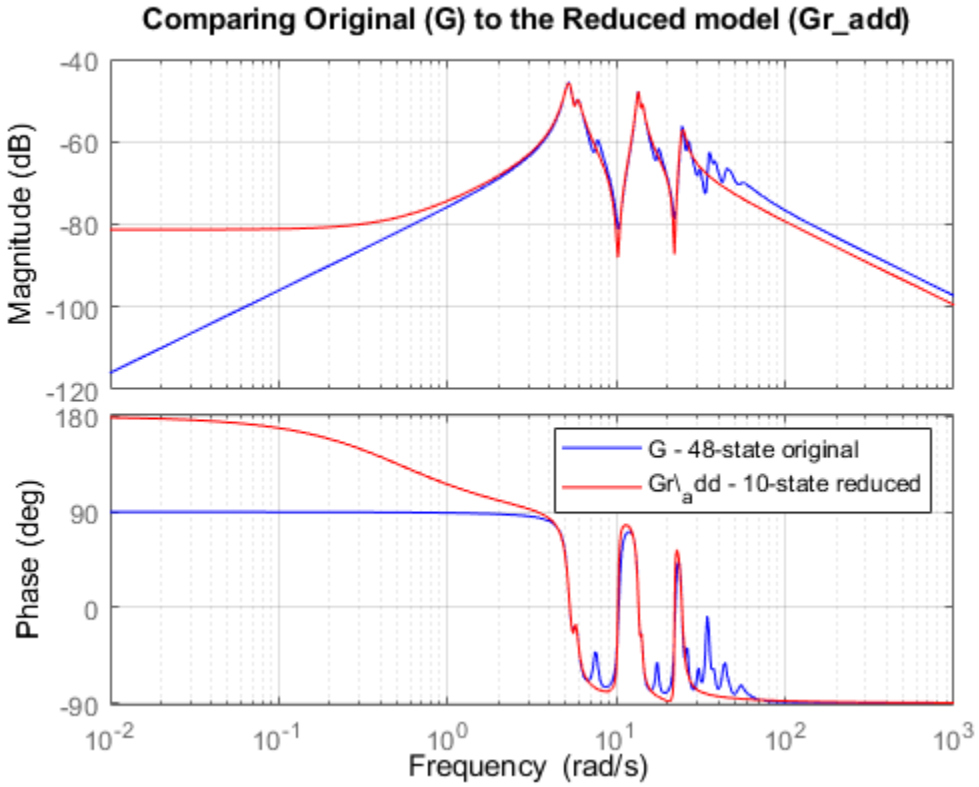


Figure 3: Comparing original (G) to the reduced model (Gr_add)

Performing Model Reduction Using a Multiplicative Error Bound

As seen from the Bode diagram in Figure 3, the reduced model captures the resonances below 30 rad/s quite well, but the match in the low frequency region (<2 rad/s) is poor. Also, the reduced model does not fully capture the dynamics in the 30-50 rad/s frequency range. A possible explanation for large errors at low frequencies is the relatively low gain of the model at these frequencies. Consequently, even large errors at these frequencies contribute little to the overall error.

To get around this problem, we can try a multiplicative-error method such as `bstmr`. This algorithm emphasizes relative errors rather than absolute ones. Because relative comparisons do not work when gains are close to zero, we need to add a minimum-gain threshold, for example by adding a feedthrough gain `D` to our original model. Assuming we are not concerned about errors at gains below -100 dB, we can set the feedthrough to 1e-5.

```
GG = G;
GG.D = 1e-5;
```

Now, let's look at the singular values for multiplicative (relative) errors (using the 'mult' option of `hankelsv`)

```
hsv_mult = hankelsv(GG,'mult');
bar(hsv_mult)
title('Multiplicative-Error Singular Values of the Model (G)');
xlabel('Number of States')
ylabel('Singular Values (\sigma_i)')
```

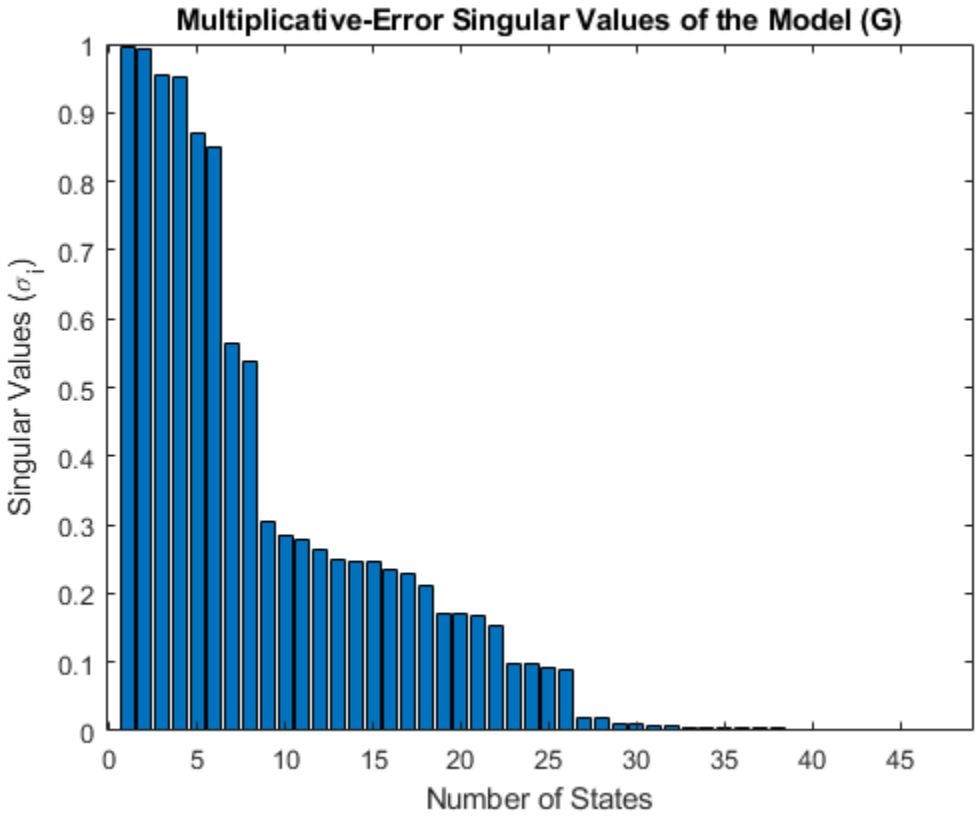


Figure 4: Multiplicative-error singular values of the model (G)

A 26th-order model looks promising, but for the sake of comparison to the previous result, let's stick to a 10th order reduction.

```
% Use bstmr algorithm option for model reduction
[Gr_mult,info_mult] = reduce(GG,10,'algorithm','bst');

%now compare the original model G to the reduced model Gr_mult
bode(G,Gr_add,Gr_mult,{1e-2,1e4}), grid on
title('Comparing Original (G) to the Reduced models (Gr\_add and Gr\_mult)')
legend('G - 48-state original ','Gr\_add (balancmr)','Gr\_mult (bstmr)','location','northeast')
```

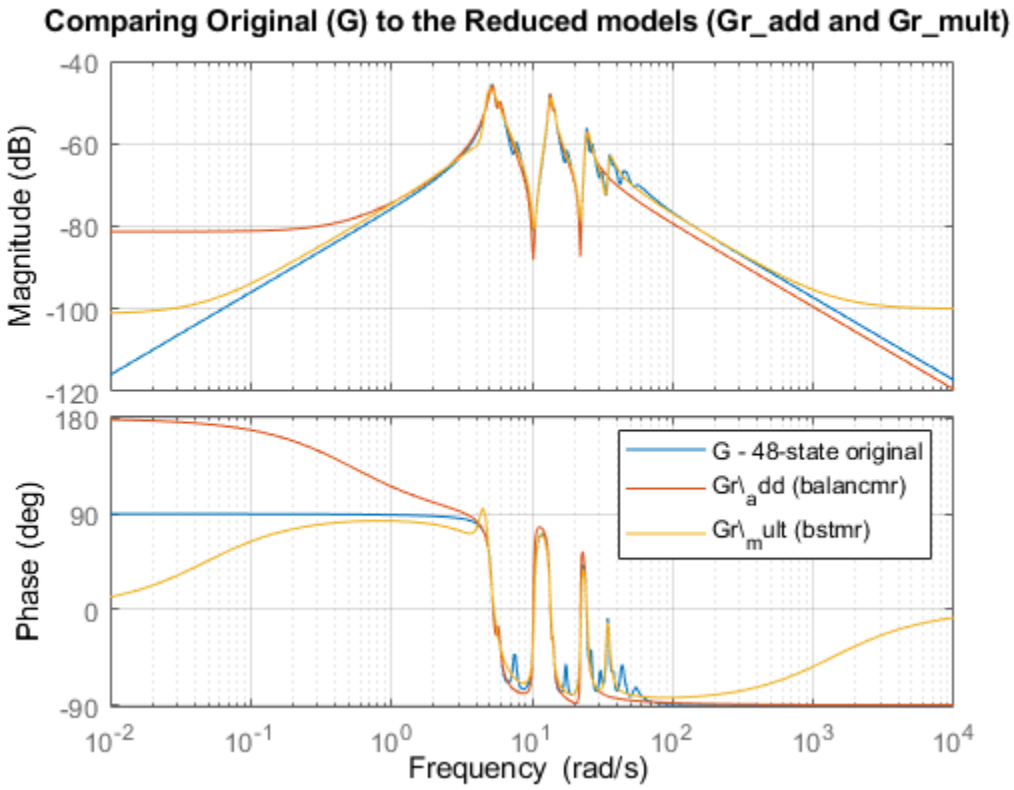


Figure 5: Comparing original (G) to the reduced models (Gr_add and Gr_mult)

The fit between the original and the reduced models is much better with the multiplicative-error approach, even at low frequencies. We can confirm this by comparing the step responses:

```
step(G,Gr_add,Gr_mult,15) %step response until 15 seconds
legend('G: 48-state original ','Gr\_add: 10-state (balancmr)','Gr\_mult: 10-state (bstmr)')
```

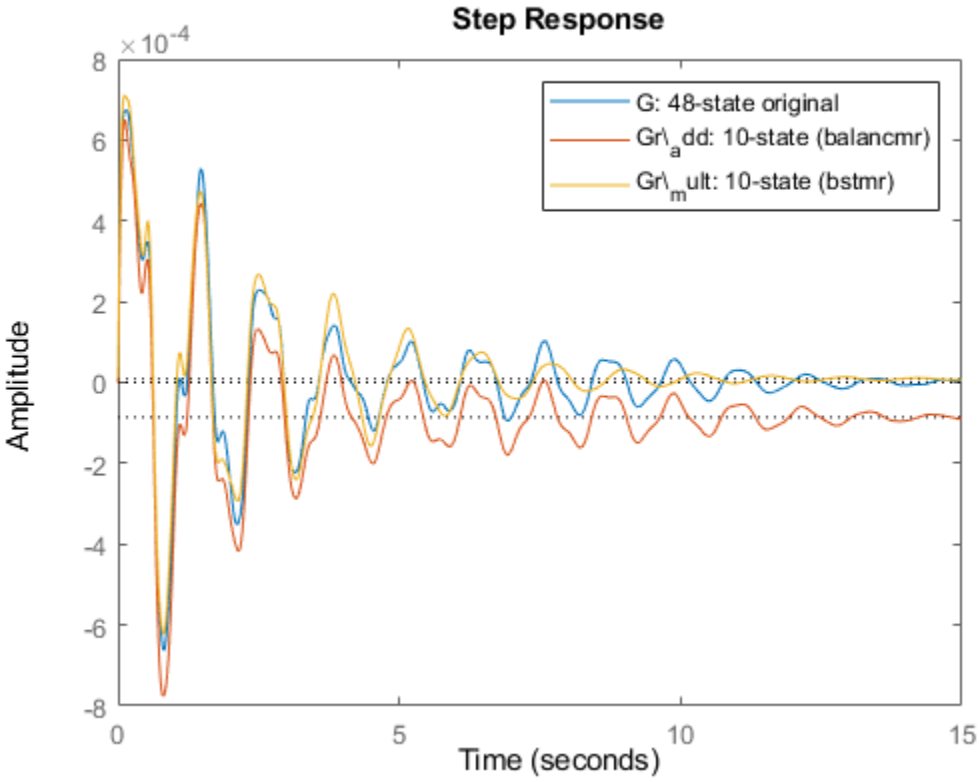


Figure 6: Step responses of the three models

Validating the Results

All algorithms provide bounds on the approximation error. For additive-error methods like `balancmr`, the approximation error is measured by the peak (maximum) gain of the error model `G-Greduced` across all frequencies. This peak gain is also known as the H-infinity norm of the error model. The error bound for additive-error algorithms looks like:

$$\|G - Gr_{add}\|_\infty \le 2 \sum_{i=11}^{48} \sigma_i := ErrorBound$$

where the sum is over all discarded Hankel singular values of `G` (entries 11 through 48 of `hsv_add`). We can verify that this bound is satisfied by comparing the two sides of this inequality:

```
norm(G-Gr_add,inf) % actual error
```

```
ans = 6.0251e-04
```

```
% theoretical bound (stored in the "ErrorBound" field of the "INFO"
% struct returned by |reduce|)
info_add.ErrorBound
```

```
ans = 0.0047
```

For multiplicative-error methods such as `bstmr`, the approximation error is measured by the peak gain across frequency of the relative error model `G\(G-Greduced)`. The error bound looks like

$$\|G^{-1}(G - Gr_{mult})\|_\infty \le \prod_{i=11}^{48} (1 + 2\sigma_i(\sigma_i + \sqrt{1+\sigma_i^2})) - 1 := ErrorBound$$

where the sum is over the discarded **multiplicative** Hankel singular values computed by `hankelsv(G,'mult')`. Again we can compare these bounds for the reduced model `Gr_mult`

```
norm(GG\(GG-Gr_mult),inf) % actual error
```

```
ans = 0.5949
```

```
% Theoretical bound
info_mult.ErrorBound
```

```
ans = 546.1730
```

Plot the relative error for confirmation

```
bodemag(GG\(GG-Gr_mult),{1e-2,1e3})
grid on
text(0.1,-50,'Peak Gain: -4.6 dB (59%) at 17.2 rad/s')
title('Relative error between original model (G) and reduced model (Gr\_mult)')
```

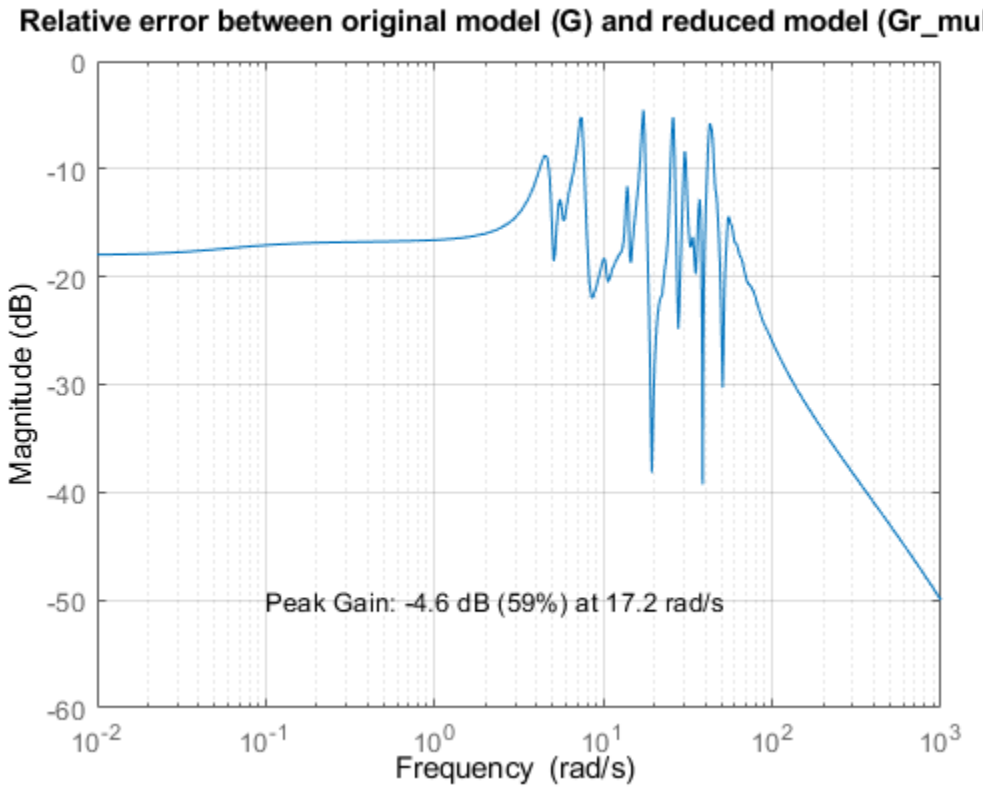


Figure 7: Relative error between original model (G) and reduced model (Gr_mult)

From the relative error plot above, there is up to 59% relative error at 17.2 rad/s, which may be more than we're willing to accept.

Picking the Lowest Order Compatible with a Desired Accuracy Level

To improve the accuracy of `Gr_mult`, we'll need to increase the order. To achieve at least 5% relative accuracy, what is the lowest order we can get? The function `reduce` can automatically select the lowest-order model compatible with our desired level of accuracy.

```
% Specify a maximum of 5% approximation error
[Gred,info] = reduce(GG,'ErrorType','mult','MaxError',0.05);
size(Gred)
```

```
State-space model with 1 outputs, 1 inputs, and 35 states.
```

The algorithm has picked a 34-state reduced model `Gred`. Compare the actual error with the theoretical bound:

```
norm(GG\(GG-Gred),inf)
```

```
ans = 0.0068
```

```
info.ErrorBound
```

```
ans = 0.0342
```

Look at the relative error magnitude as a function of frequency. Higher accuracy has been achieved at the expense of a larger model order (from 10 to 34). Note that the actual maximum error is 0.6%, much less than the 5% target. This discrepancy is a result of the function `bstmr` using the error bound rather than the actual error to select the order.

```
bodemag(GG\(GG-Gred),{1,1e3})
grid on
text(5,-75,'Peak Gain: -43.3 dB (0.6%) at 73.8 rad/s')
title('Relative error between original model (G) and reduced model (Gred)')
```

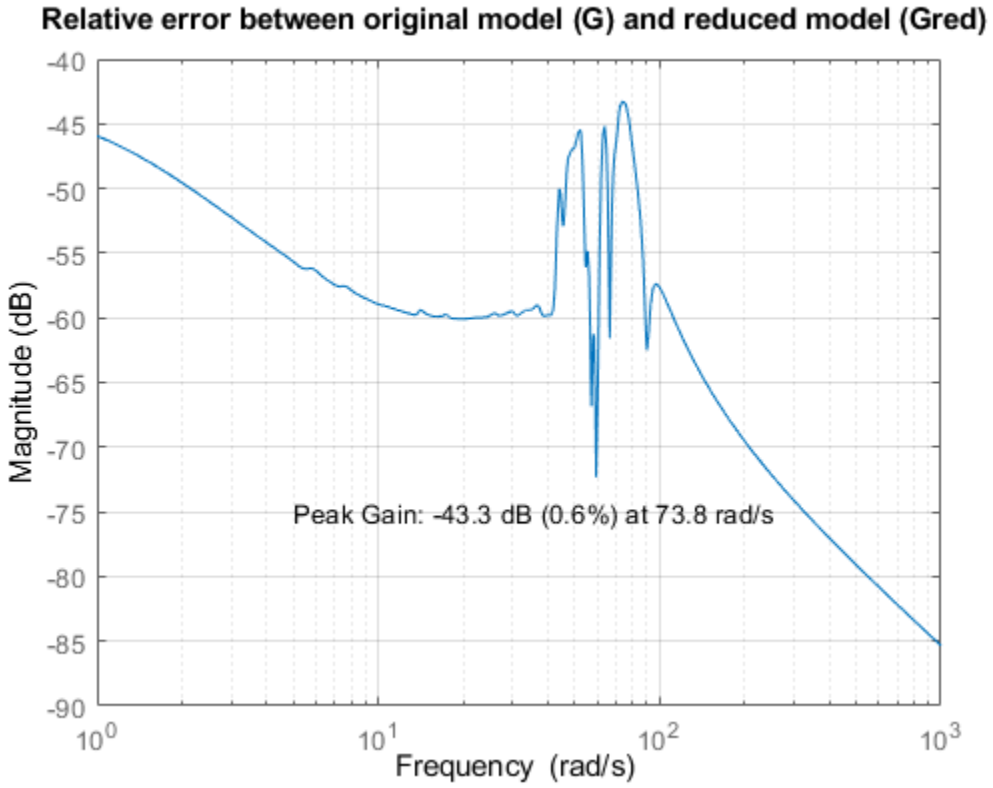


Figure 8: Relative error between original model (G) and reduced model (Gred)

Compare the Bode responses

```
bode(G,Gred,{1e-2,1e4})
grid on
legend('G - 48-state original','Gred - 34-state reduced')
```

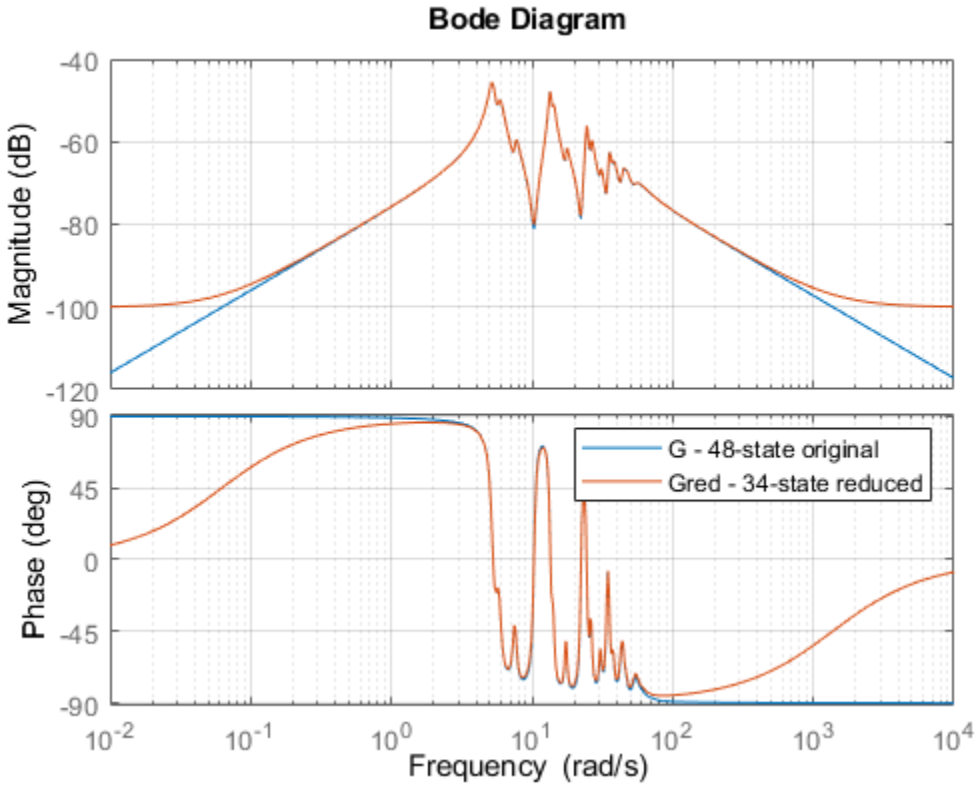


Figure 9: Bode diagram of the 48-state original model and the 34-state reduced model

Finally, compare the step responses of the original and reduced models. They are virtually indistinguishable.

```
step(G,'b',Gred,'r--',15) %step response until 15 seconds
legend('G: 48-state original ','Gred: 34-state (bstmr)')
text(5,-4e-4,'Maximum relative error <0.05')
```

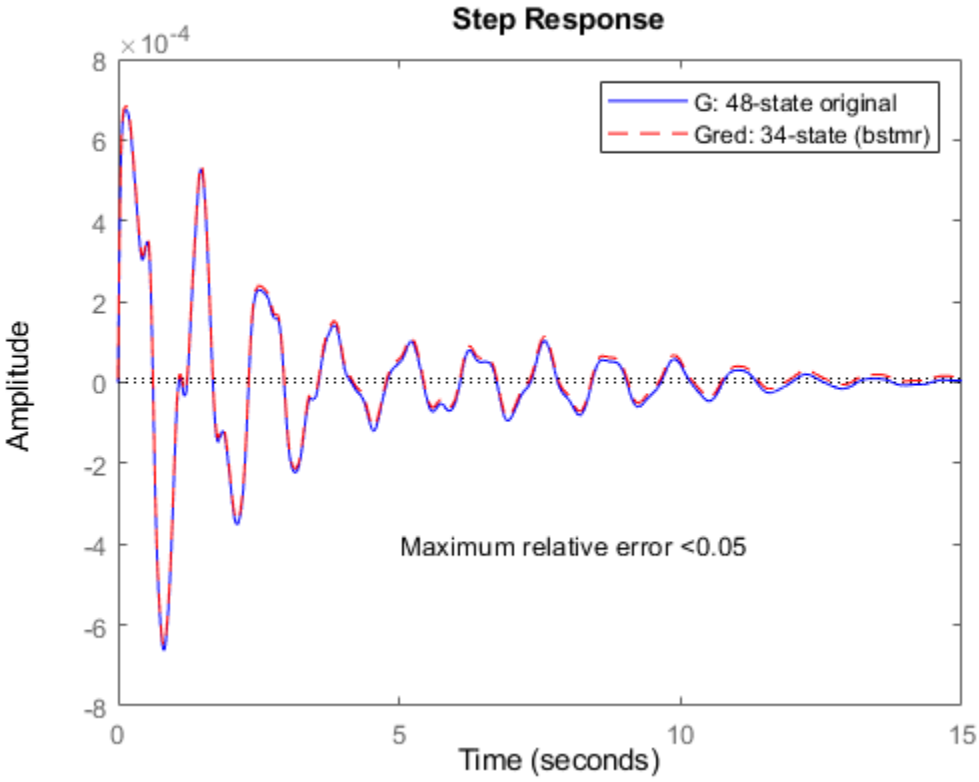


Figure 10: Step response plot of the 48-state original model and the 34-state reduced model

See Also

`hankelsv` | `reduce`

More About

- “Model Reduction Techniques” on page 3-5

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.

Robustness Analysis

Create Models of Uncertain Systems

Forms of model uncertainty include:

- 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)
```

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

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.Uncertain
```

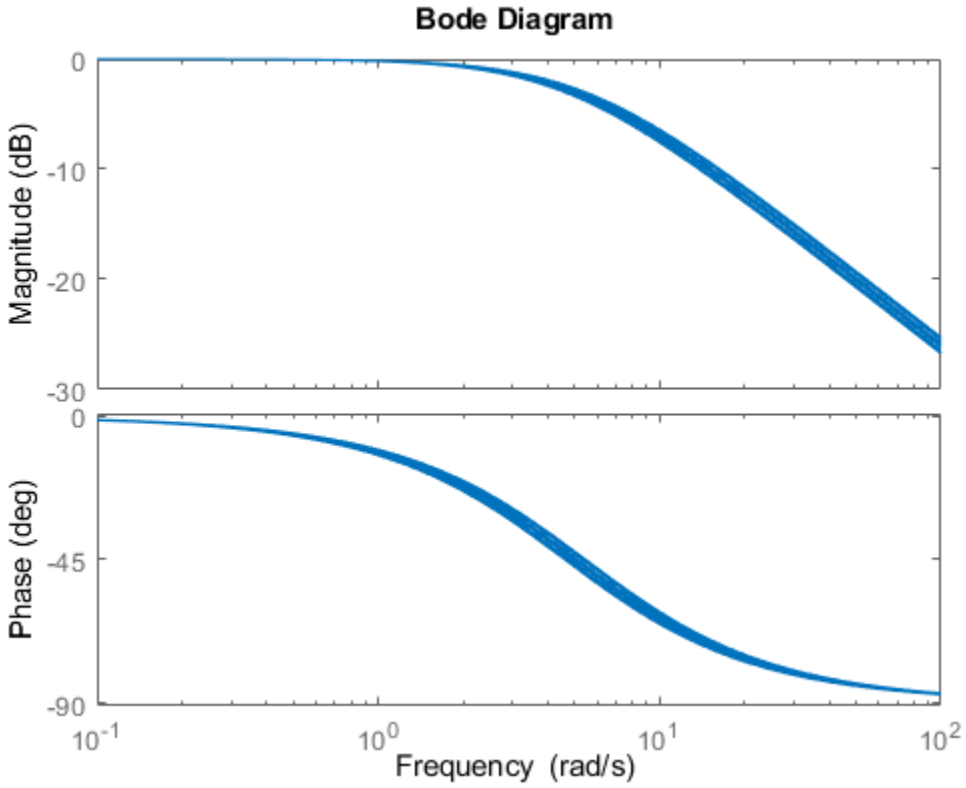
Note that the result `H` is an uncertain system, called a `uss` model. The nominal value of `H` is a state-space (`ss`) model. Verify that the pole is at -5, as expected from the uncertain parameter's nominal value of 5.

```
pole(H.NominalValue)
```

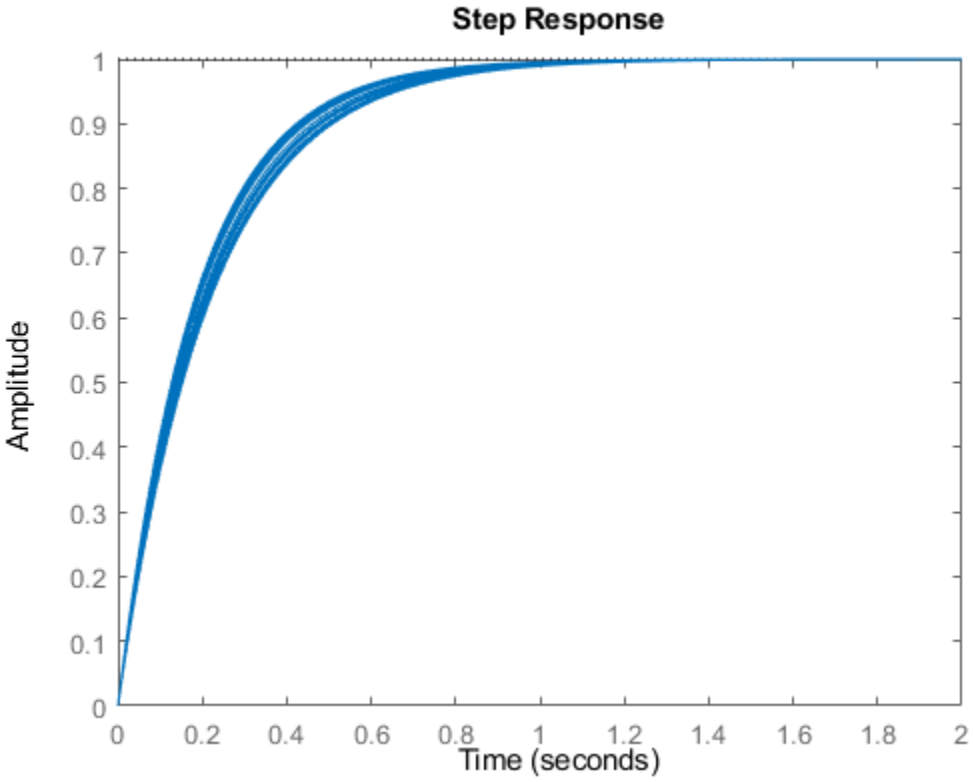
```
ans = -5
```

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.

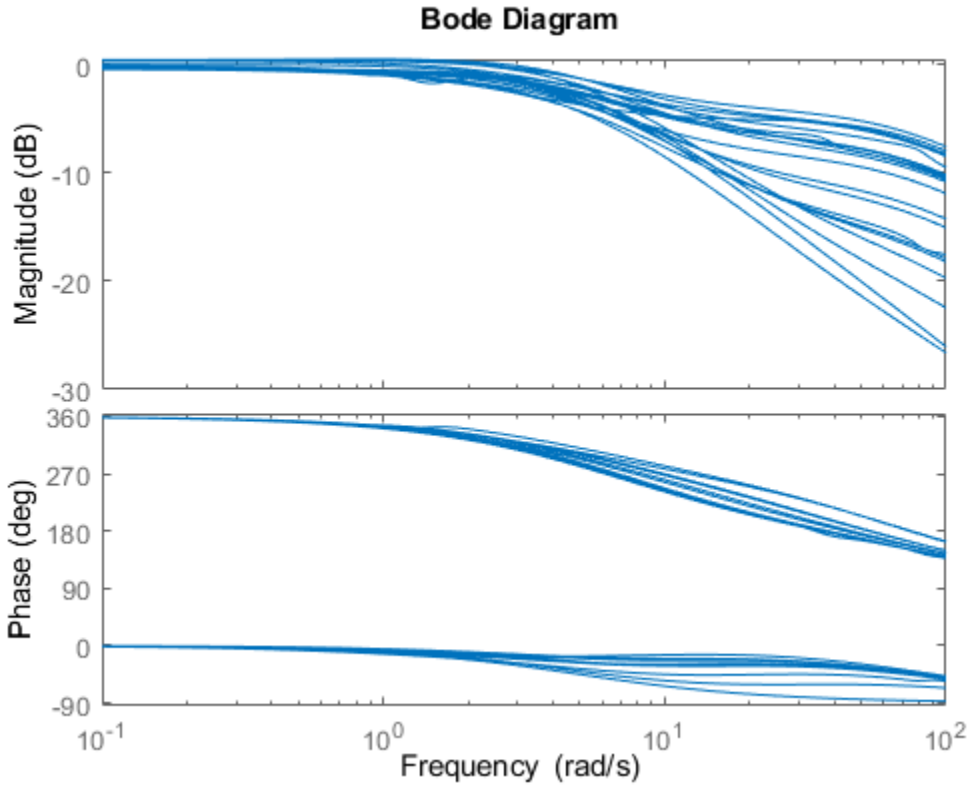
The following commands carry out these steps:

```
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.Uncertain
```

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})
```



Gain and Phase Uncertainty

A special case of dynamic uncertainty is uncertainty in the gain and phase in a feedback loop. Modeling gain and phase variations in your uncertain system model lets you verify stability margins during robustness analysis or enforce them during robust controller design. Use the `umargin` control design block to represent gain and phase variations in feedback loops. For more information, see “Uncertain Gain and Phase”.

See Also

Related Examples

- “System with Uncertain Parameters” on page 1-6
- “Systems with Unmodeled Dynamics”
- “Model Gain and Phase Uncertainty in Feedback Loops”

Robust Controller Design

This example shows how to design a feedback controller for a plant with uncertain parameters and uncertain model dynamics. The goals of the controller design are good steady-state tracking and disturbance-rejection properties.

Design a controller for the plant G described in “Robust Controller Design” on page 4-7. This plant is a first-order system with an uncertain time constant. The plant also has some uncertain dynamic deviations from first-order behavior beyond about 9 rad/s.

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

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.Uncertain
```

Design Controller

Because of the nominal first-order behavior of the plant, choose a PI control architecture. For a desired closed-loop damping ratio ξ and natural frequency ω_n, the design equations for the proportional and integral gains (based on the nominal open-loop time constant of 0.2) are:

$$K_p = \frac{2\xi\omega_n}{5} - 1,\ K_i = \frac{\omega_n^2}{5}.$$

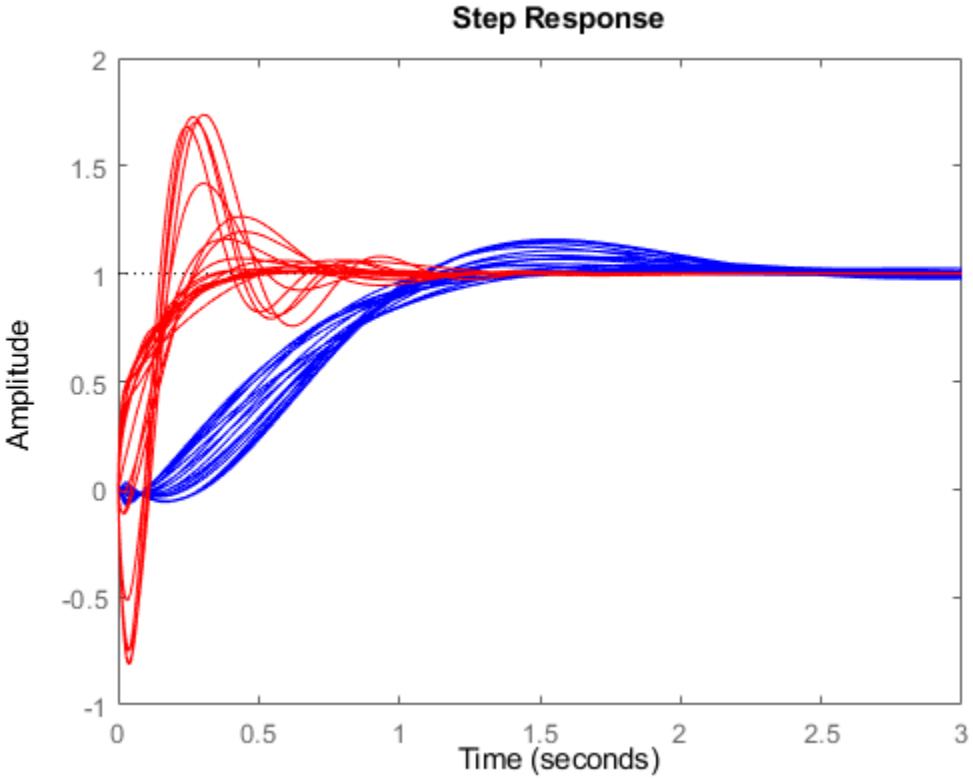
To study how the uncertainty in G affects the achievable closed-loop bandwidth, design two controllers, both achieving $\xi = 0.707$, but with different ω_n values, 3 and 7.5.

```
xi = 0.707;
wn1 = 3;
wn2 = 7.5;

Kp1 = 2*xi*wn1/5 - 1;
Ki1 = (wn1^2)/5;
C1 = tf([Kp1,Ki1],[1 0]);

Kp2 = 2*xi*wn2/5 - 1;
Ki2 = (wn2^2)/5;
C2 = tf([Kp2,Ki2],[1 0]);
```

Examine Controller Performance

The nominal closed-loop bandwidth achieved by C2 is in a region where G has significant model uncertainty. It is therefore expected that the model variations cause significant degradations in the closed-loop performance with that controller. To examine the performance, form the closed-loop systems and plot the step responses of samples of the resulting systems.

```
T1 = feedback(G*C1,1);
T2 = feedback(G*C2,1);
tfinal = 3;
step(T1,'b',T2,'r',tfinal)
```



The step responses for `T2` exhibit a faster rise time because `C2` sets a higher closed-loop bandwidth. However, as expected, the model variations have a greater impact.

You can use `robstab` to check the robustness of the stability of the closed-loop systems to model variations.

```
opt = robOptions('Display','on');
stabmarg1 = robstab(T1,opt);

Computing peak...  Percent completed: 100/100
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 401% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 401% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 3.74 rad/seconds.

stabmarg2 = robstab(T2,opt);

Computing peak...  Percent completed: 100/100
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 125% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 125% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 11.4 rad/seconds.
```

The display gives the amount of uncertainty that the system can tolerate without going unstable. In both cases, the closed-loop systems can tolerate more than 100% of the modeled uncertainty range while remaining stable. `stabmarg` contains 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.

Compare Nominal and Worst-Case Behavior

While both systems are stable for all variations, their performance is 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 function, `S = 1/(1+GC)`. This peak gain of this function is typically correlated with the amount of overshoot in a step response; peak gain greater than one indicates overshoot.

Form the closed-loop sensitivity functions and call `wcgain`.

```
S1 = feedback(1,G*C1);
S2 = feedback(1,G*C2);
[maxgain1,wcu1] = wcgain(S1);
[maxgain2,wcu2] = wcgain(S2);
```

`maxgain` 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. Examine the bounds on the worst-case gain for both systems.

```
maxgain1
```

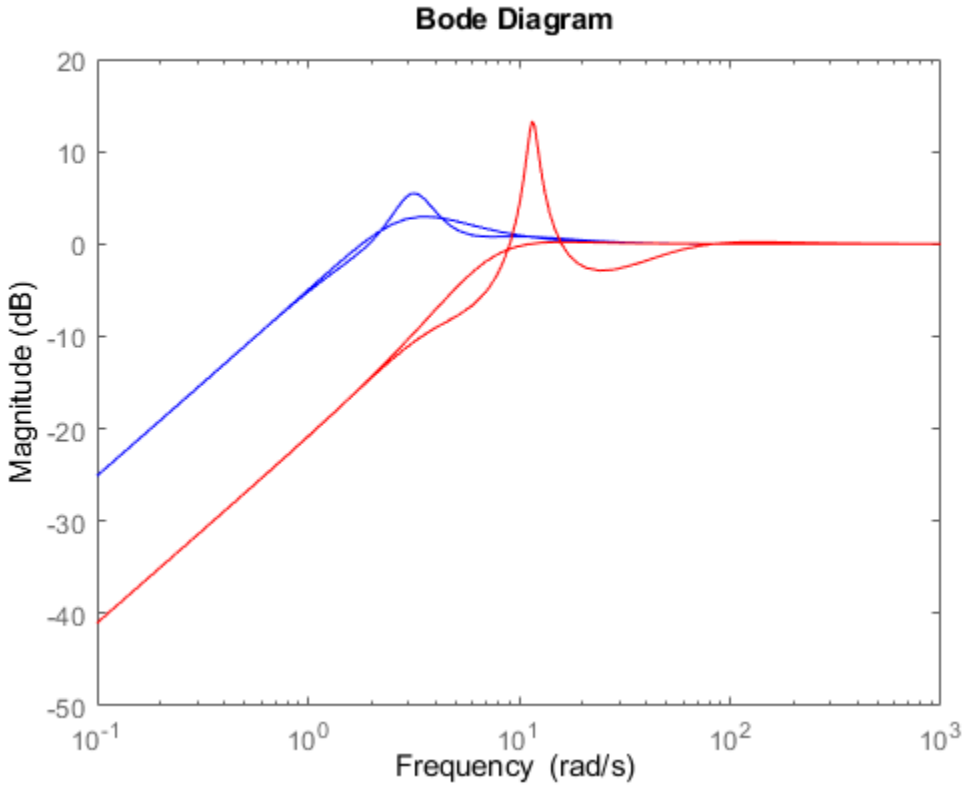
```
maxgain1 = struct with fields:
           LowerBound: 1.8831
           UpperBound: 1.8862
    CriticalFrequency: 3.1952
```

```
maxgain2
```

```
maxgain2 = struct with fields:
           LowerBound: 4.6286
           UpperBound: 4.6378
    CriticalFrequency: 11.6132
```

`wcu` contains the particular values of the uncertain elements that achieve this worst-case behavior. Use `usubs` to substitute these worst-case values for uncertain elements, and compare the nominal and worst-case behavior.

```
wcS1 = usubs(S1,wcu1);
wcS2 = usubs(S2,wcu2);
bodemag(S1.NominalValue,'b',wcS1,'b');
hold on
bodemag(S2.NominalValue,'r',wcS2,'r');
```



While `C2` achieves better nominal sensitivity than `C1`, 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 `C2` is inferior to `C1` for this particular uncertain model.

See Also

`robstab` | `wcgain` | `usubs`

Related Examples

- “MIMO Robustness Analysis” on page 4-11

MIMO Robustness Analysis

You can create and analyze uncertain state-space models made up of uncertain state-space matrices. In this example, create a MIMO system with parametric uncertainty and analyze it for robust stability and worst-case performance.

Consider 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.Uncertaint
```

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

```
get(H)
```

```
     NominalValue: [2x2 ss]
      Uncertainty: [1x1 struct]
                A: [2x2 umat]
                B: [2x2 double]
                C: [2x2 umat]
                D: [2x2 double]
                E: []
        StateName: {2x1 cell}
        StateUnit: {2x1 cell}
    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]
            Notes: [0x1 string]
         UserData: []
             Name: ''
     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.

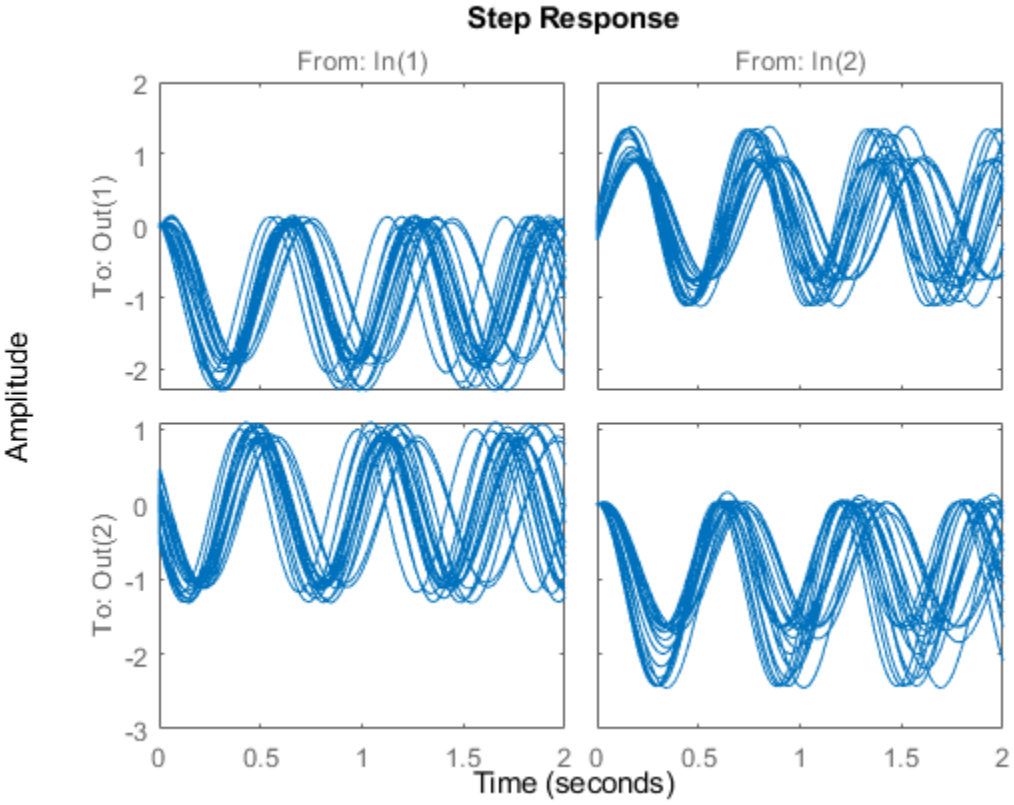
```
W1 = makeweight(.1,20,50);
W2 = makeweight(.2,45,50);
Delta1 = ultidyn('Delta1',[1 1]);
Delta2 = ultidyn('Delta2',[1 1]);
G = H*blkdiag(1+W1*Delta1,1+W2*Delta2)
```

```
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.Uncertaint
```

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.

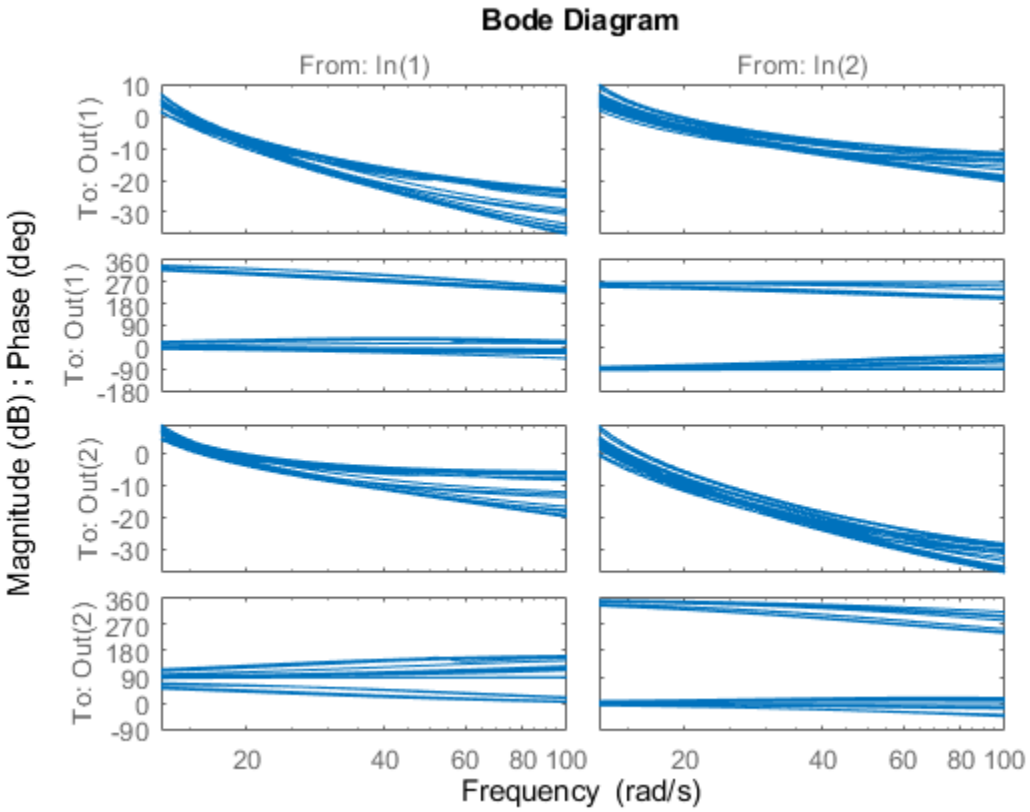
You can plot a 2-second step response of several samples of G The 10% uncertainty in the natural frequency is apparent.

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



You can create a Bode plot of samples of `G`. The high-frequency uncertainty in the model is also apparent. 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.mat')
size(K)
```

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

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

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

```
F = struct with fields:
        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:

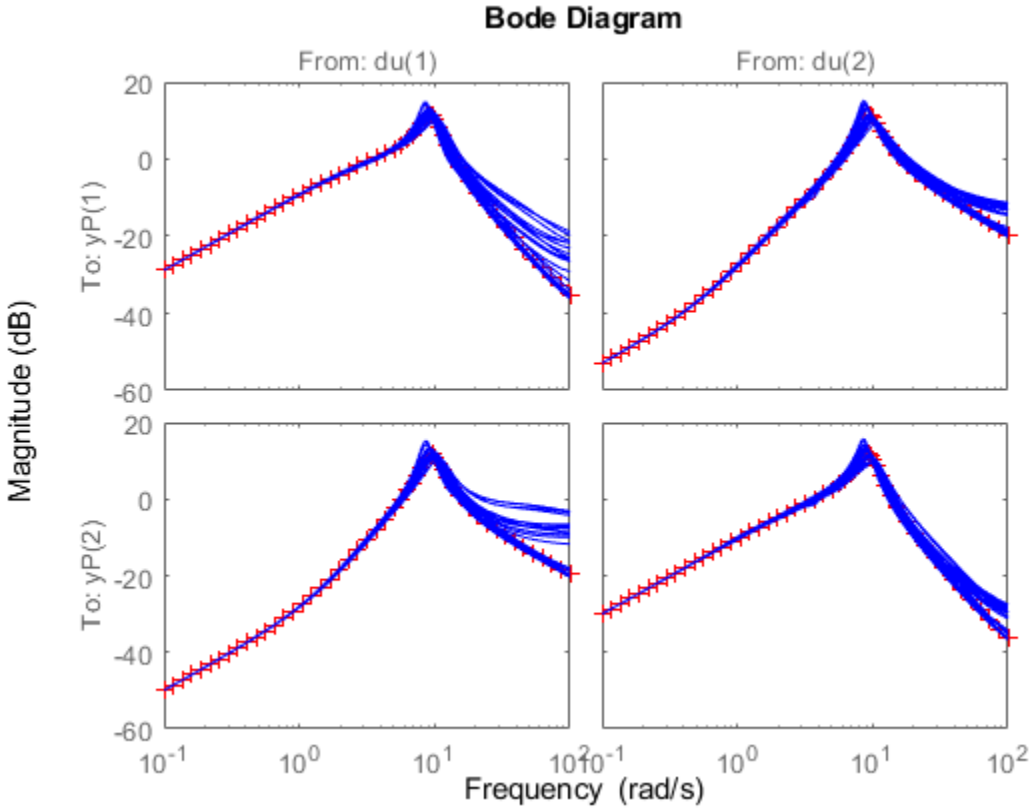
$$K(I + GK)^{-1}G$$

`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})
```



Nominal Stability Margins

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

For instance, explore the disk-based margins for gain or phase variations at the plant outputs and inputs. (For general information about disk-based margin analysis, see “Stability Analysis Using Disk Margins”.)

```
[DMo,MMo] = diskmargin(G*K);
[DMi,MMi] = diskmargin(K*G);
```

The loop-at-a-time margins are returned in the structure arrays `DMo` and `DMi`. Each of these arrays contains one entry for each of the two feedback channels. For instance, examine the margins at the plant output for the second feedback channel.

```
DMo(2)
```

```
ans = struct with fields:
           GainMargin: [0.0682 14.6726]
          PhaseMargin: [-82.2022 82.2022]
           DiskMargin: 1.7448
           LowerBound: 1.7448
           UpperBound: 1.7448
            Frequency: 4.8400
    WorstPerturbation: [2x2 ss]
```

This result tells you that the gain at the second plant output can vary by factors between about 0.07 and about 14.7, without the second loop going unstable. Similarly, the loop can tolerate phase variations at the output up to about ±82°.

The structures `MMo` and `MMi` contain the margins for concurrent and independent variations in both channels. For instance, examine the multiloop margins at the plant inputs.

```
MMi
```

```
MMi = struct with fields:
           GainMargin: [0.1186 8.4289]
          PhaseMargin: [-76.4682 76.4682]
           DiskMargin: 1.5758
           LowerBound: 1.5758
           UpperBound: 1.5790
            Frequency: 5.9828
    WorstPerturbation: [2x2 ss]
```

This result tells you that the gain at the plant input can vary in both channels independently by factors between about 1/8 and 8 without the closed-loop system going unstable. The system can tolerate independent and concurrent phase variations up about ±76°. Because the multiloop margins take loop interactions into account, they tend to be smaller than loop-at-a-time margins.

Examine the multiloop margins at the plant outputs.

```
MMo
```

```
MMo = struct with fields:
           GainMargin: [0.1201 8.3280]
          PhaseMargin: [-76.3058 76.3058]
           DiskMargin: 1.5712
           LowerBound: 1.5712
           UpperBound: 1.5744
            Frequency: 17.4276
    WorstPerturbation: [2x2 ss]
```

The margins at the plant outputs are similar to those at the inputs. This result is not always true in multiloop feedback systems.

Finally, examine the margins against simultaneous variations at the plant inputs and outputs.

```
MMio = diskmargin(G,K)
```

```
MMio = struct with fields:
           GainMargin: [0.5676 1.7619]
          PhaseMargin: [-30.8440 30.8440]
           DiskMargin: 0.5517
           LowerBound: 0.5517
           UpperBound: 0.5528
            Frequency: 9.0688
    WorstPerturbation: [1x1 struct]
```

When you consider all such variations simultaneously, the margins are somewhat smaller than those at the inputs or outputs alone. Nevertheless, these numbers indicate a generally robust closed-loop system. The system can tolerate significant simultaneous gain variations or ±30° degree simultaneous phase variations in all input and output channels of the plant.

Robust Stability Margin

With `diskmargin`, you determine various stability 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 a detailed uncertainty model, the stability margins computed by `diskmargin` may not accurately reflect how close the system is from being unstable. You can then use `robstab` to compute the robust stability margin for the specified uncertainty.

In this example, use `robstab` to compute the robust stability margin for the uncertain feedback loop comprised of `G` and `K`. You can use any of the closed-loop transfer functions in `F = loopsens(G,K)`. All of them, `F.Si, F.To`, etc., have the same internal dynamics, and hence their stability properties are the same.

```
opt = robOptions('Display','on');
stabmarg = robstab(F.So,opt)
```

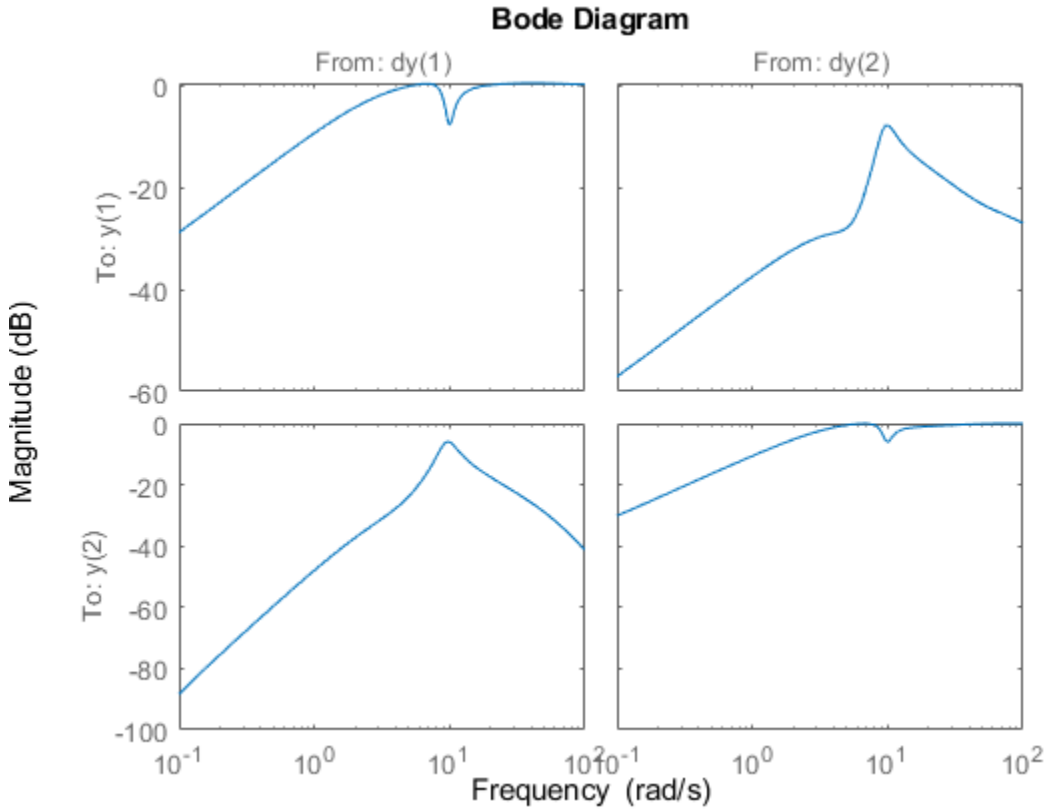
```
Computing peak...  Percent completed: 100/100
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 221% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 222% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 13.6 rad/seconds.

stabmarg = struct with fields:
           LowerBound: 2.2129
           UpperBound: 2.2173
    CriticalFrequency: 13.6333
```

This analysis confirms what the `diskmargin` 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] = getPeakGain(F.So.NominalValue)
```

```
PeakNom = 1.1317
```

```
freq = 7.1300
```

The peak is about 1.13. 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.

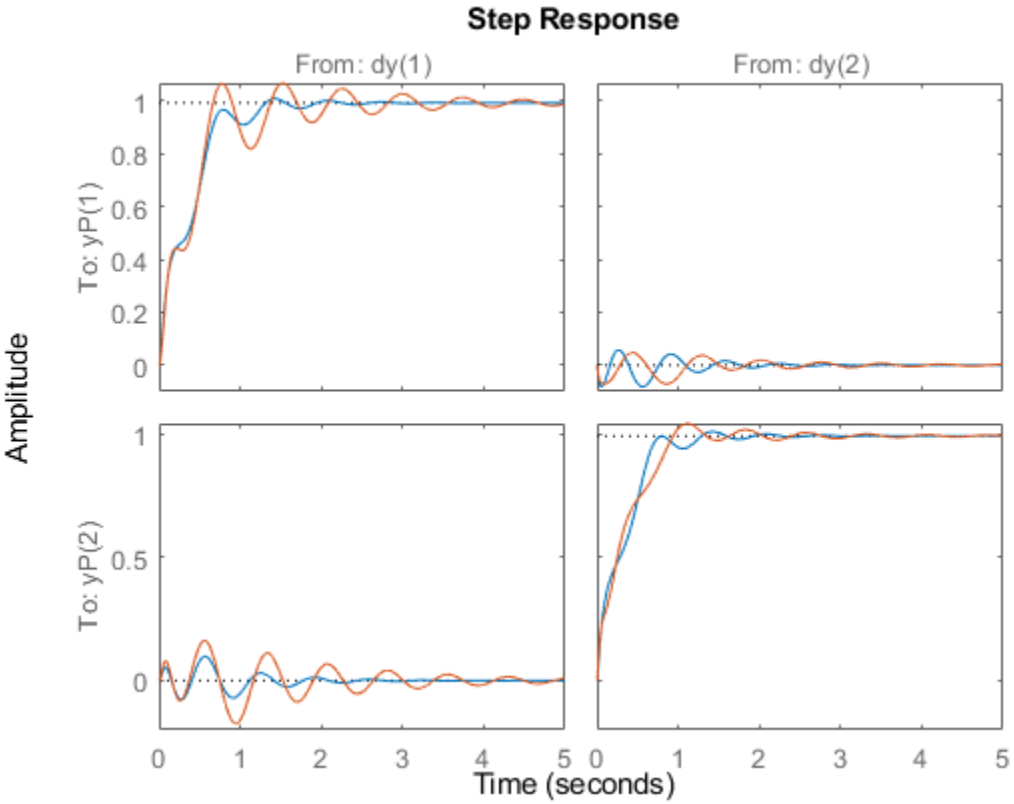
```
[maxgain,wcu] = wcgain(F.So);
maxgain
```

```
maxgain = struct with fields:
           LowerBound: 2.1599
           UpperBound: 2.1643
    CriticalFrequency: 8.3354
```

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.

Use `usubs` to replace the values for `Delta1`, `Delta2`, and `p` that achieve the gain of 2.1. Make the substitution in the output complementary sensitivity, and do a step response.

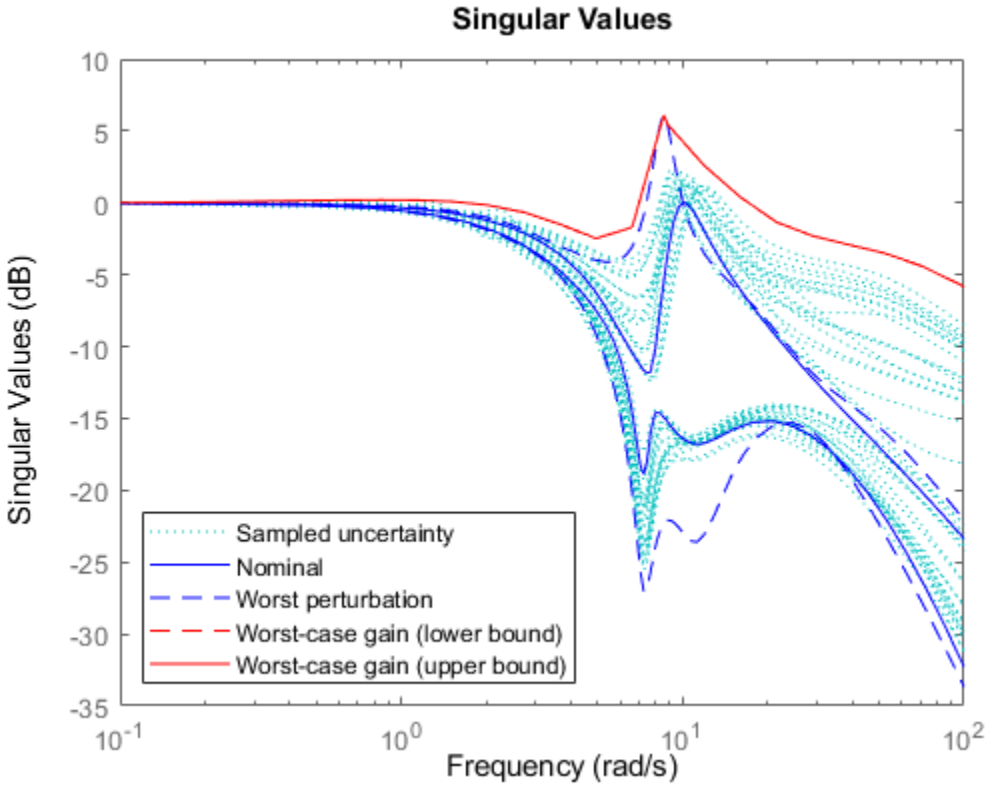
```
step(F.To.NominalValue,usubs(F.To,wcu),5)
```



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 `wcsigmaplot`.

```
wcsigmaplot(F.To,{1e-1,100})
```



See Also

`ultidyn` | `loopsens` | `diskmargin` | `robstab` | `wcgain` | `usubs` | `wcsigmaplot`

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 $e \in \mathrm{L}_2$. For vector-valued signals

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

the 2-norm is defined as

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

In 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,

$$\begin{aligned} \|T\|_2 &:= \left[\frac{1}{2\pi}\int_{-\infty}^{\infty} \|T(j\omega)\|_F^2 d\omega\right]^{\frac{1}{2}} \\ \|T\|_\infty &:= \max_{\omega \in R} \bar{\sigma}[T(j\omega)], \end{aligned}$$

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

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

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

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

then

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

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

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

Using Weighted Norms to Characterize Performance

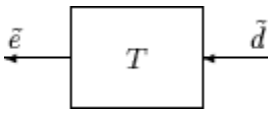
Any performance criterion must also account for

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

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

$$\|W_L T W_R\|$$

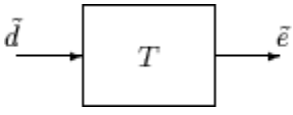
where the weighting function matrices W_L and W_R are frequency dependent, to account for bandwidth constraints and spectral content of exogenous signals. The most natural (mathematical) manner to characterize acceptable performance is in terms of the MIMO $\|\cdot\|_\infty$ (H_∞) norm. For this reason, 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” on page 5-3 with the arrows going from left to right as in “Weighted MIMO System” on page 5-5.



Unweighted MIMO System: Vectors from Left to Right

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

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

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

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

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

Moreover, there are specific disturbances d that result in the ratio $\|\tilde{e}\|_2/\|\tilde{d}\|_2$ arbitrarily close to β. 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 $\|a\|_2 \le 1$, define a time signal

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

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

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

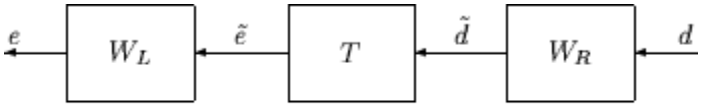
The vector $b \in R^{n_e}$ will satisfy $\|b\|_2 \le \beta$. Moreover, β, as defined in “Weighted MIMO System” on page 5-5, is the smallest number such that this is true for every $\|a\|_2 \le 1$, $\bar{\omega}$, and ϕ.

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

Consider the diagram of “Weighted MIMO System” on page 5-5, along with “Unweighted MIMO System: Vectors from Left to Right” on page 5-3.

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 $\tilde{e}\left(= T\tilde{d}\right)$ in the diagram of "Unweighted MIMO System: Vectors from Left to Right" on page 5-3. Specifically, for sinusoidal signal $\tilde{d}$, the steady-state relationship between $\tilde{e}\left(= T\tilde{d}\right)$, $\tilde{d}$ and $\|W_L T W_R\|_\infty$ is as follows. The steady-state solution $\tilde{e}_{ss}$, denoted as

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

satisfies

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

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

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

satisfying

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

if and only if $\|W_L T W_R\|_\infty \le 1$.

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

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

the steady-state error components will satisfy

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

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 $W_R(j\bar{\omega})$).

See Also

`hinfstruct` | `hinfsyn`

Related Examples

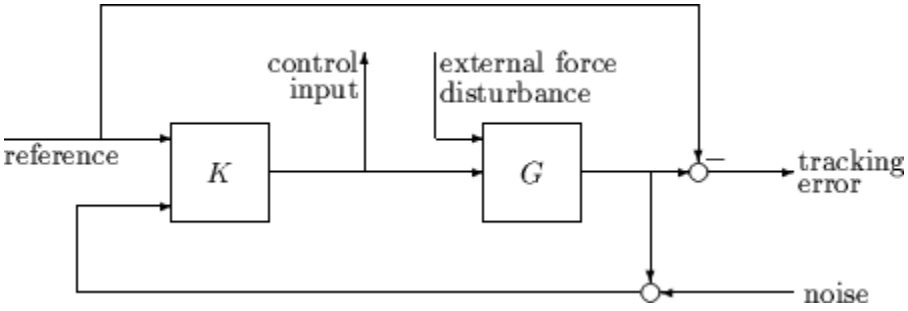
- “H-Infinity Performance” on page 5-7
- “Robust Control of an Active Suspension” on page 5-13

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” on page 5-8. 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

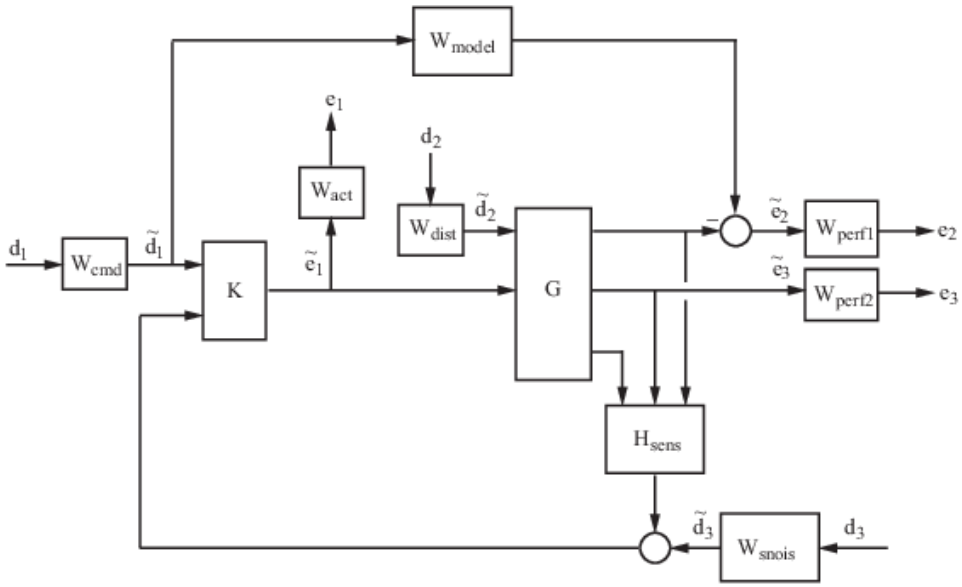
So if the performance objective is in the form of a matrix norm, it should actually be a *weighted norm*

$\|W_L T W_R\|$

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” on page 5-8. 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 $\|T_{\mathrm{ed}}\|_\infty < 1$. The weighting functions are used to scale the input/output transfer functions such that when $\|T_{\mathrm{ed}}\|_\infty < 1$, 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
$\tilde{d}_1$	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

$\mathbf{W_{cmd}}$

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:

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

$\mathbf{W_{model}}$

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 10º/second response is desired for each inch of stick motion. Then, in these units, the appropriate model is:

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

$\mathbf{W_{dist}}$

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}.

$\mathbf{W_{perf1}}$

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}.

$\mathbf{W_{perf2}}$

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.

$\mathbf{W_{act}}$

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.

$\mathbf{W_{snois}}$

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.

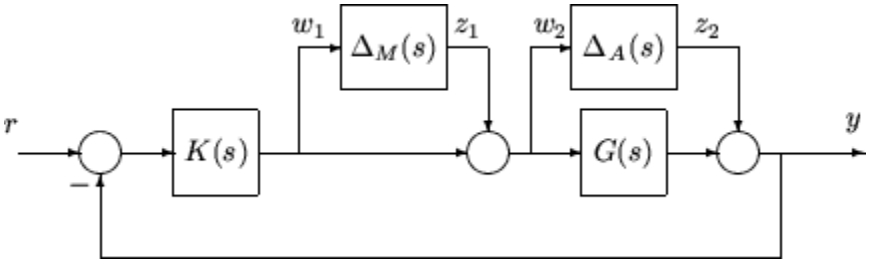
$\mathbf{H_{sens}}$

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.

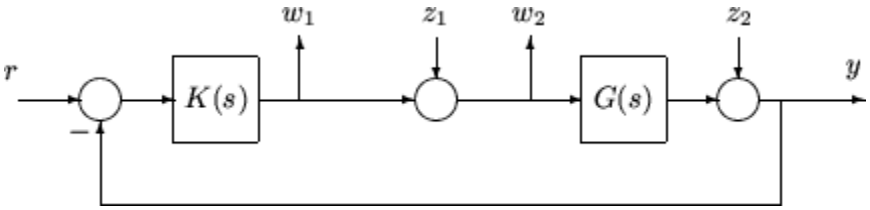


The transfer function matrices are defined as:

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

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

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, $G(s)$.

Numeric Considerations

Do not choose weighting functions with poles very close to $s = 0$ ($z = 1$ for discrete-time systems). For instance, although it might seem sensible to choose $W_{cmd} = 1/s$ to enforce zero steady-state error, doing so introduces an unstable pole that cannot be stabilized, causing synthesis to fail. Instead, choose $W_{cmd} = 1/(s + \delta)$. The value δ must be small but not very small compared to system dynamics. For instance, for best numeric results, if your target crossover frequency is around 1 rad/s, choose $\delta = 0.0001$ or 0.001. Similarly, in discrete time, choose sample times such that system and weighting dynamics are not more than a decade or two below the Nyquist frequency.

See Also

`hinfstruct` | `hinfsyn` | `mixsyn`

Related Examples

- “Norms and Singular Values” on page 2-6
- “Robust Control of an Active Suspension” on page 5-13
- “Mixed-Sensitivity Loop Shaping” on page 2-25

Robust Control of an Active Suspension

This example shows how to use Robust Control Toolbox™ to design a robust controller for an active suspension system. The example describes the quarter-car suspension model. Then, it computes an H_∞ controller for the nominal system using the `hinfsyn` command. Finally, the example shows how to use μ-synthesis to design a robust controller for the full uncertain system.

Quarter-Car Suspension Model

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

This example uses a quarter-car model of the active suspension system (see Figure 1). The mass m_b (in kilograms) represents the car chassis (body) and the mass m_w (in kilograms) 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 (all in meters) are the body travel, wheel travel, and road disturbance, respectively. The force f_s (in kiloNewtons) applied between the body and wheel assembly is controlled by feedback and represents the active component of the suspension system.

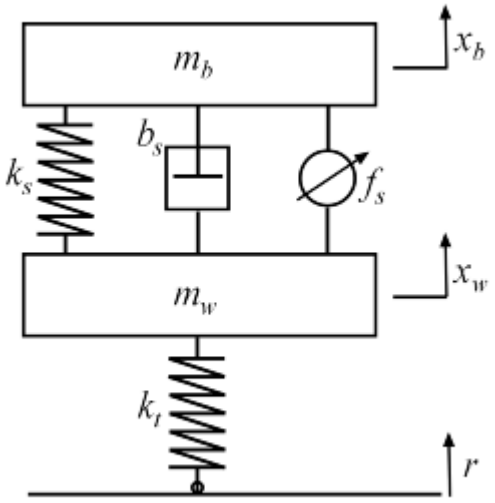


Figure 1: Quarter-car model of active suspension.

With the notation $(x_1, x_2, x_3, x_4) := (x_b, \dot{x_b}, x_w, \dot{x}_w)$, the linearized state-space equations for the quarter-car model are:

$$\dot{x_1} = x_2$$
$$\dot{x_2} = -(1/m_b)[k_s(x_1 - x_3) + b_s(x_2 - x_4) - 10^3 f_s]$$
$$\dot{x_3} = x_4$$
$$\dot{x_4} = (1/m_w)[k_s(x_1 - x_3) + b_s(x_2 - x_4) - k_t(x_3 - r) - 10^3 f_s].$$

Construct a state-space model `qcar` representing these equations.

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

% State matrices
A = [ 0 1 0 0; [-ks -bs ks bs]/mb ; ...
      0 0 0 1; [ks bs -ks-kt -bs]/mw];
B = [ 0 0; 0 1e3/mb ; 0 0 ; [kt -1e3]/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 (m)';'body vel (m/s)';...
          'wheel travel (m)';'wheel vel (m/s)'};
qcar.InputName = {'r';'fs'};
qcar.OutputName = {'xb';'sd';'ab'};
```

The transfer function from actuator to body travel and acceleration has an imaginary-axis zero with natural frequency 56.27 rad/s. This is called the *tire-hop frequency*.

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

```
ans = 2×1 complex

  -0.0000 +56.2731i
  -0.0000 -56.2731i
```

Similarly, the transfer function from actuator to suspension deflection has an imaginary-axis zero with natural frequency 22.97 rad/s. This is called the *rattlespace frequency*.

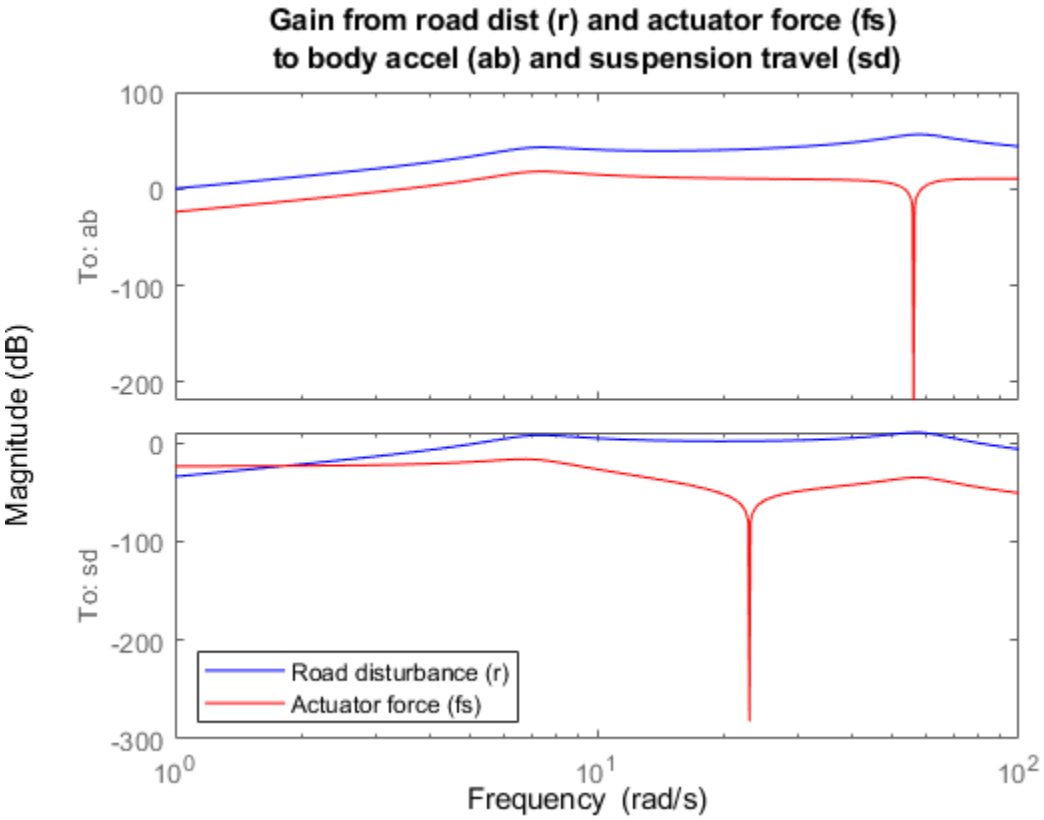
```
zero(qcar('sd','fs'))
```

```
ans = 2×1 complex

   0.0000 +22.9734i
   0.0000 -22.9734i
```

Road disturbances influence the motion of the car and suspension. Passenger comfort is associated with small body acceleration. The allowable suspension travel is constrained by limits on the actuator displacement. Plot the open-loop gain from road disturbance and actuator force to body acceleration and suspension displacement.

```
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)'})
```



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, and from r to suspension deflection s_d at the rattlespace frequency. Moreover, because of the relationship $x_w = x_b - s_d$ and the fact that the wheel position x_w roughly follows r at low frequency (less than 5 rad/s), there is an inherent trade-off between passenger comfort and suspension deflection: any reduction of body travel at low frequency will result in an increase of suspension deflection.

Uncertain 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 are represented by the first-order transfer function $1/(1 + s/60)$ with a maximum displacement of 0.05 m.

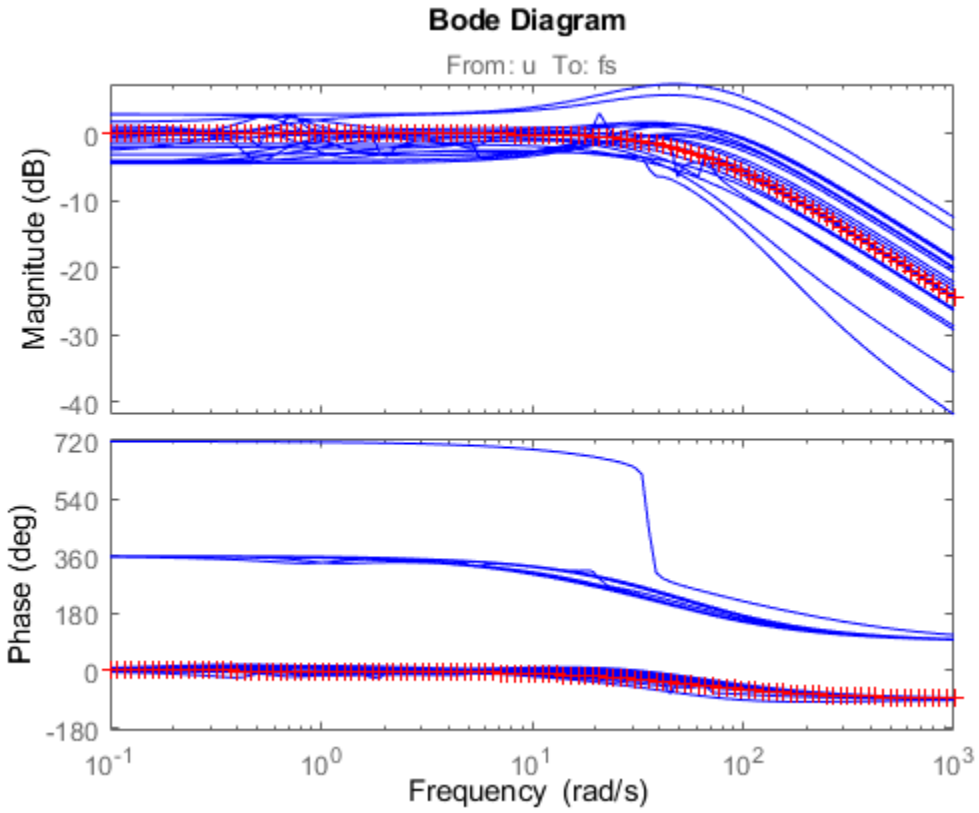
```
ActNom = tf(1,[1/60 1]);
```

This nominal model only approximates the physical actuator dynamics. We can use a family of actuator models to account for modeling errors and variability in the actuator and quarter-car models. This family consists of a nominal model with a frequency-dependent amount of uncertainty. 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/s. The weighting function W_{unc} is used to modulate the amount of uncertainty with frequency.

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

The result `Act` is an uncertain state-space model of the actuator. Plot the Bode response of 20 sample values of `Act` and compare with the nominal value.

```
rng('default')
bode(Act,'b',Act.NominalValue,'r+',logspace(-1,3,120))
```



Design Setup

The main control objectives are formulated in terms of passenger comfort and road handling, which relate to body acceleration a_b and suspension travel s_d. Other factors that influence the control design include the characteristics of the road disturbance, the quality of the sensor measurements for feedback, and the limits on the available control force. To use H_∞ synthesis algorithms, we must express these objectives as a single cost function to be minimized. This can be done as indicated Figure 2.

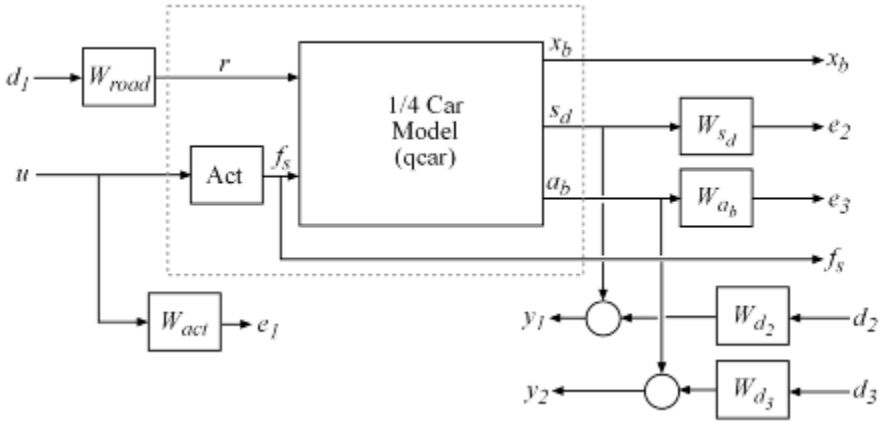


Figure 2: Disturbance rejection formulation.

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

- The road disturbance r, modeled as a normalized signal d_1 shaped by a weighting function W_{road}. To model broadband road deflections of magnitude seven centimeters, we use the constant weight $W_{road} = 0.07$
- Sensor noise on both measurements, modeled as normalized signals d_2 and d_3 shaped by weighting functions W_{d_2} and W_{d_3}. We use $W_{d_2} = 0.01$ and $W_{d_3} = 0.5$ to model broadband sensor noise of intensity 0.01 and 0.5, respectively. In a more realistic design, these weights would be frequency dependent to model the noise spectrum of the displacement and acceleration sensors.

The control objectives can be reinterpreted as a *disturbance rejection* goal: Minimize the impact of the disturbances d_1, d_2, d_3 on a weighted combination of control effort u, suspension travel s_d, and body acceleration a_b. When using the H_∞ norm (peak gain) to measure "impact", this amounts to designing a controller that minimizes the H_∞ norm from disturbance inputs d_1, d_2, d_3 to error signals e_1, e_2, e_3.

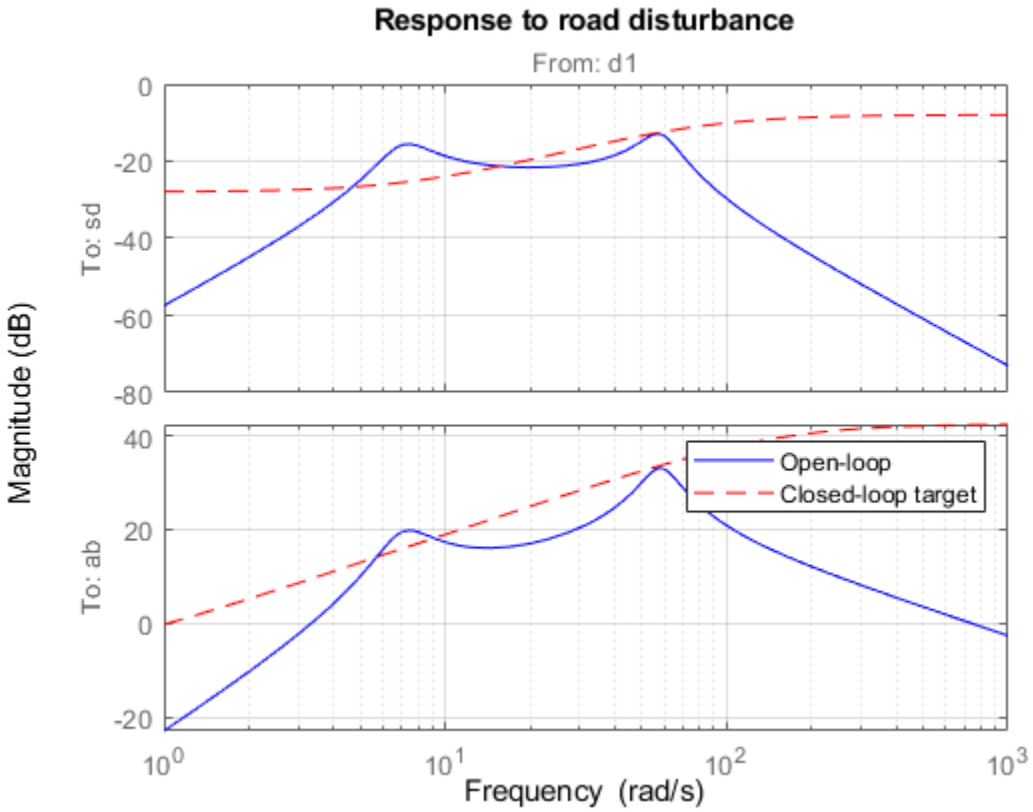
Create the weighting functions of Figure 2 and label their I/O channels to facilitate interconnection. Use a high-pass filter for W_{act} to penalize high-frequency content of the control signal and thus limit the control bandwidth.

```
Wroad = ss(0.07);  Wroad.u = 'd1';   Wroad.y = 'r';
Wact = 0.8*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';
```

Specify closed-loop targets for the gain from road disturbance r to suspension deflection s_d (handling) and body acceleration a_b (comfort). Because of the actuator uncertainty and imaginary-axis zeros, only seek to attenuate disturbances below 10 rad/s.

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

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



The corresponding performance weights W_{sd}, W_{ab} are the reciprocals of these comfort and handling targets. To investigate the trade-off between passenger comfort and road handling, construct three sets of weights $(\beta W_{sd}, (1-\beta)W_{ab})$ corresponding to three different trade-offs: comfort ($\beta = 0.01$), balanced ($\beta = 0.5$), and handling ($\beta = 0.99$).

```
% Three design points
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';
```

Finally, use `connect` to construct a model `qcaric` of the block diagram of Figure 2. Note that `qcaric` is an array of three models, one for each design point β. Also, `qcaric` is an uncertain model since it contains the uncertain actuator model `Act`.

```
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 =

  3x1 array of uncertain continuous-time state-space models.
  Each model has 5 outputs, 4 inputs, 9 states, and the following uncertain blocks:
    unc: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences

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

Nominal H-infinity Design

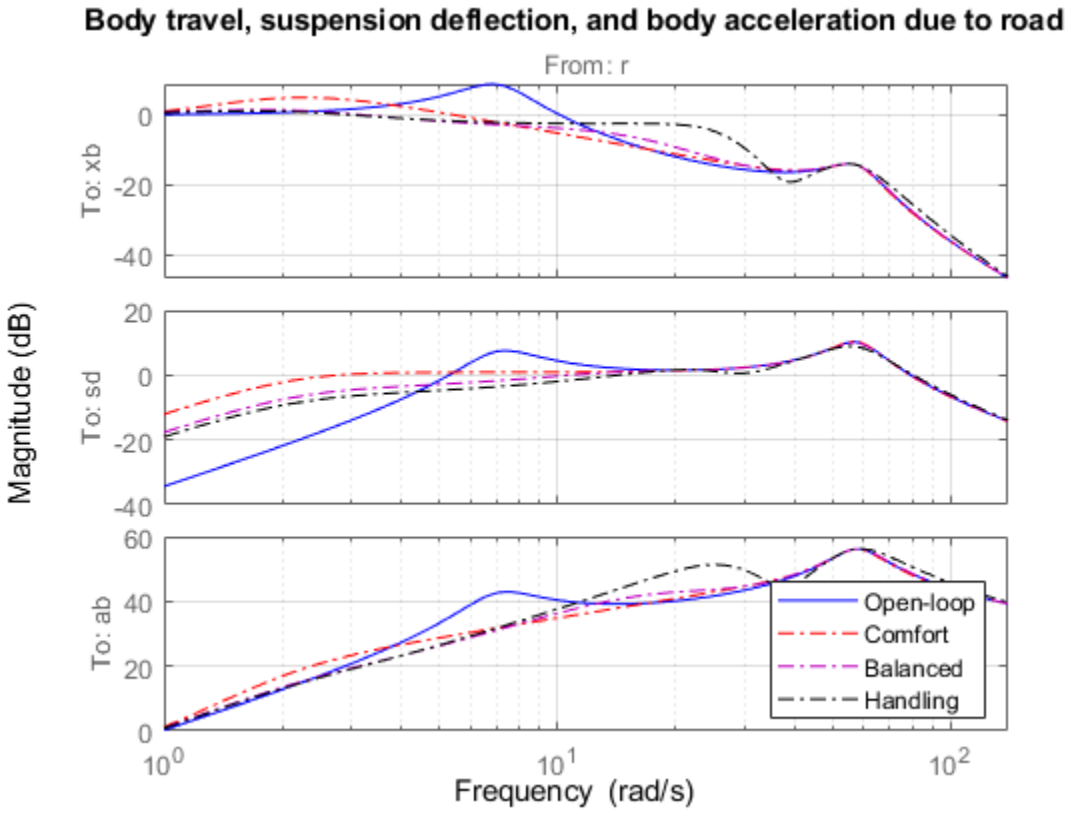
Use `hinfsyn` to compute an H_∞ controller for each value of the blending factor β.

```
ncont = 1; % one control signal, u
nmeas = 2; % two measurement signals, sd and ab
K = ss(zeros(ncont,nmeas,3));
gamma = zeros(3,1);
for i=1:3
   [K(:,:,i),~,gamma(i)] = hinfsyn(qcaric(:,:,i),nmeas,ncont);
end

gamma
```

```
gamma = 3×1

    0.9405
    0.6727
    0.8892
```

The three controllers achieve closed-loop H_∞ norms of 0.94, 0.67 and 0.89, respectively. Construct the corresponding closed-loop models and compare the gains from road disturbance to x_b, s_d, a_b for the passive and active suspensions. Observe that all three controllers reduce suspension deflection and body acceleration below the rattlespace frequency (23 rad/s).

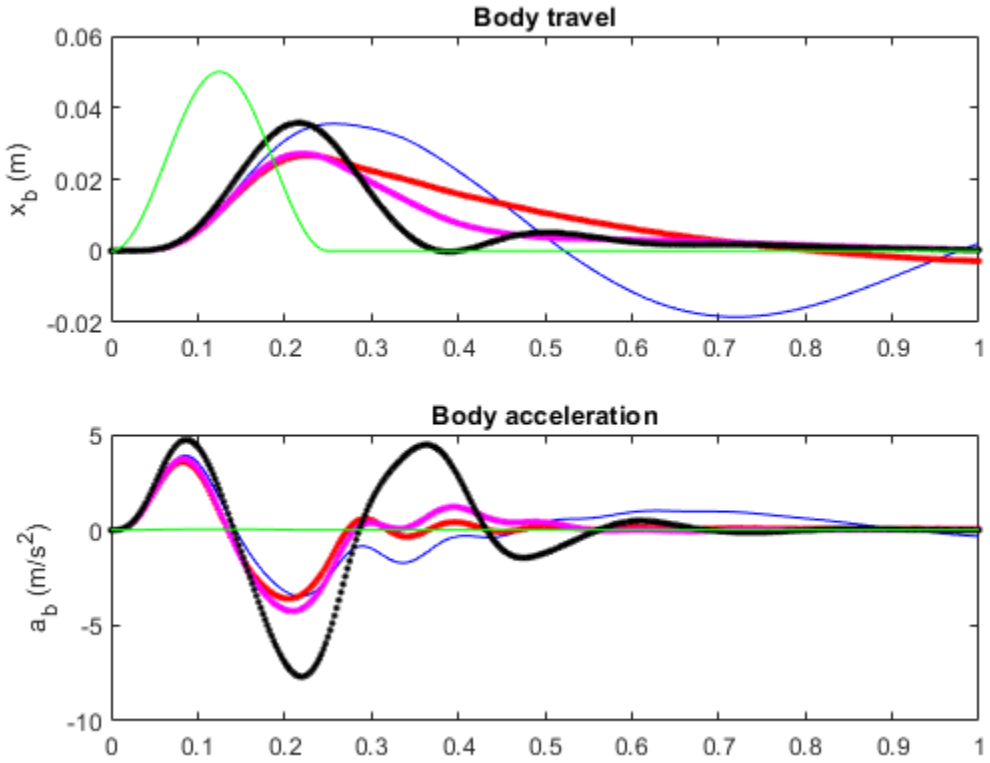
```
% Closed-loop models
K.u = {'sd','ab'};  K.y = 'u';
CL = connect(qcar,Act.Nominal,K,'r',{'xb';'sd';'ab'});

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')
```



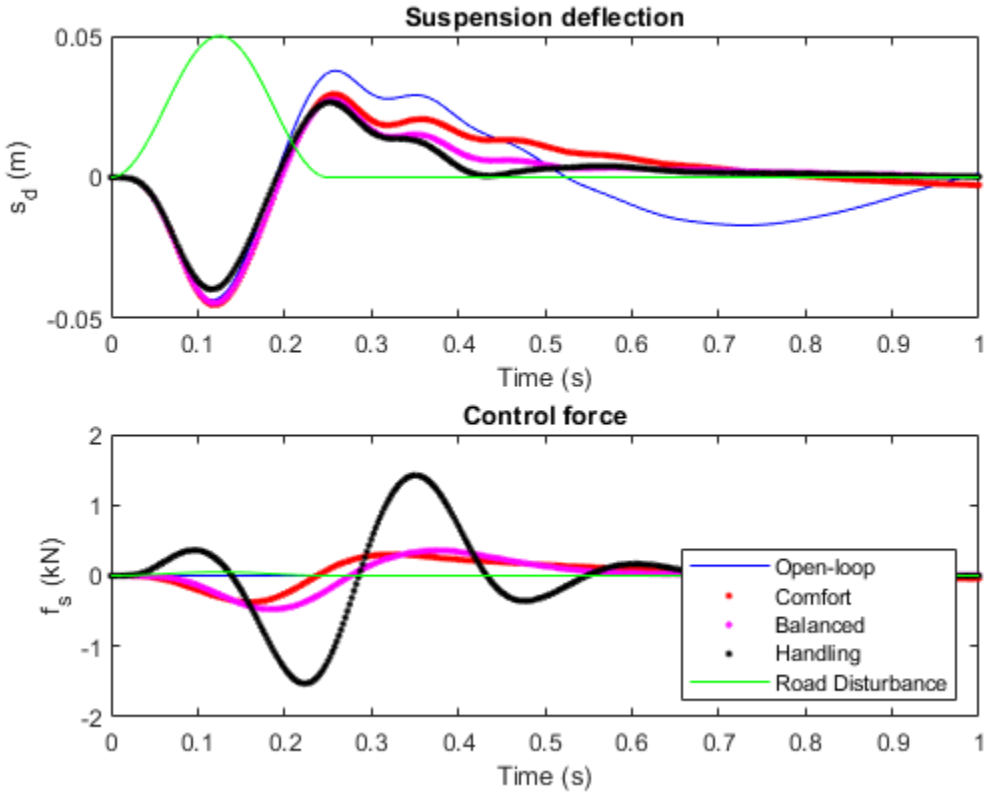
Time-Domain Evaluation

To further evaluate the three designs, perform time-domain simulations using a road disturbance signal $r(t)$ representing a road bump of height 5 cm.

```
% Road disturbance
t = 0:0.0025:1;
roaddist = zeros(size(t));
roaddist(1:101) = 0.025*(1-cos(8*pi*t(1:101)));

% Closed-loop model
SIMK = connect(qcar,Act.Nominal,K,'r',{'xb';'sd';'ab';'fs'});

% Simulate
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);

% Plot results
subplot(211)
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(212)
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(211)
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(212)
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 (kN)')
legend('Open-loop','Comfort','Balanced','Handling','Road Disturbance','location','SouthEast')
```



Observe that the body acceleration is smallest for the controller emphasizing passenger comfort and largest for the controller emphasizing suspension deflection. The "balanced" design achieves a good compromise between body acceleration and suspension deflection.

Robust Mu Design

So far you have designed H_∞ controllers that meet the performance objectives for the *nominal* actuator model. As discussed earlier, this model is only an approximation of the true actuator and you need to make sure that the controller performance is maintained in the face of model errors and uncertainty. This is called *robust performance*.

Next use μ-synthesis to design a controller that achieves robust performance for the entire family of actuator models. The robust controller is synthesized with the musyn function using the uncertain model `qcaric(:,:,2)` corresponding to "balanced" performance ($\beta = 0.5$).

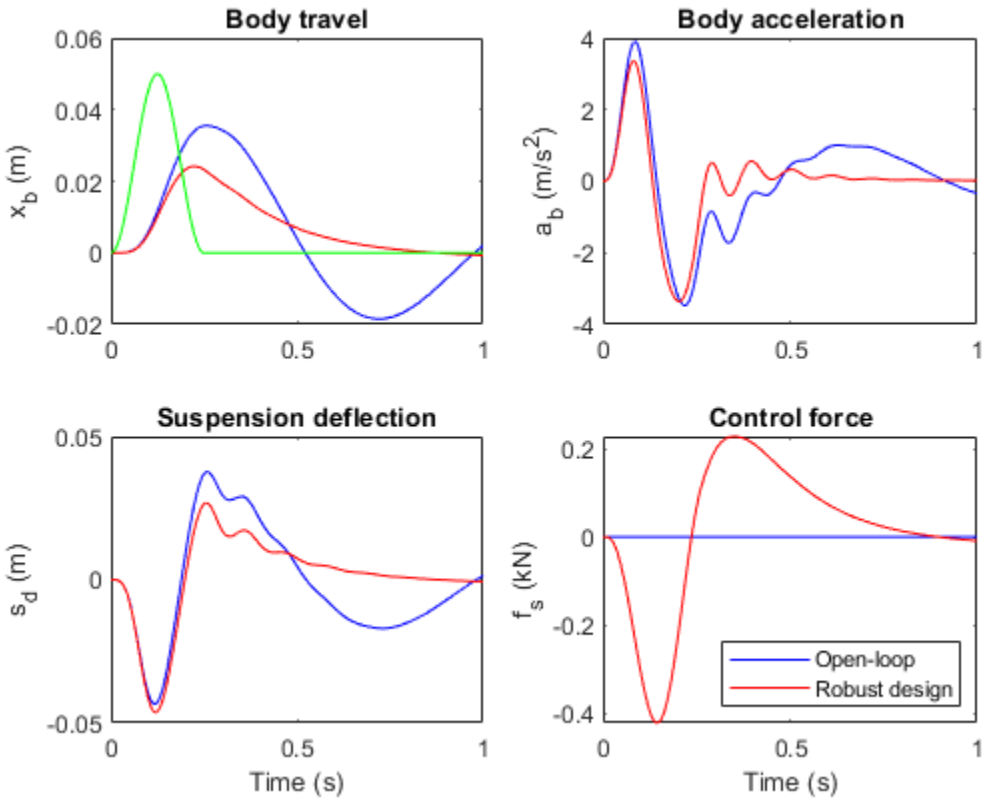
```
[Krob,rpMU] = musyn(qcaric(:,:,2),nmeas,ncont);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                      Robust performance              Fit order
-----------------------------------------------------------------
  Iter         K Step       Peak MU       D Fit            D
    1          1.193         1.125        1.139            4
    2          1.091         1.025        1.033            4
    3         0.9991         0.946       0.9559            4
    4         0.9358         0.932       0.9348            4
    5         0.9096        0.9057       0.9114            8
```

```
    6          0.9103        0.907       0.9096             8
    7          0.9091       0.9066       0.9094             6

Best achieved robust performance: 0.906
```

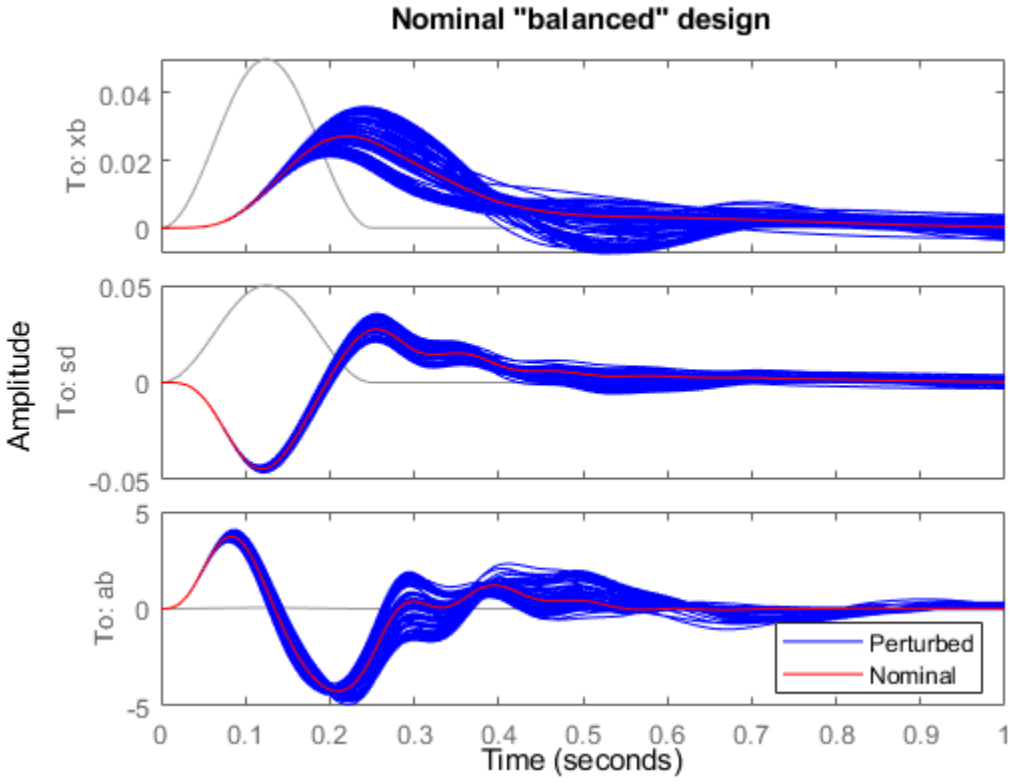
Simulate the nominal response to a road bump with the robust controller `Krob`. The responses are similar to those obtained with the "balanced" H_∞ controller.

```
% Closed-loop model (nominal)
Krob.u = {'sd','ab'};
Krob.y = 'u';
SIMKrob = connect(qcar,Act.Nominal,Krob,'r',{'xb';'sd';'ab';'fs'});

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

% Plot results
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 (kN)')
legend('Open-loop','Robust design','location','SouthEast')
```

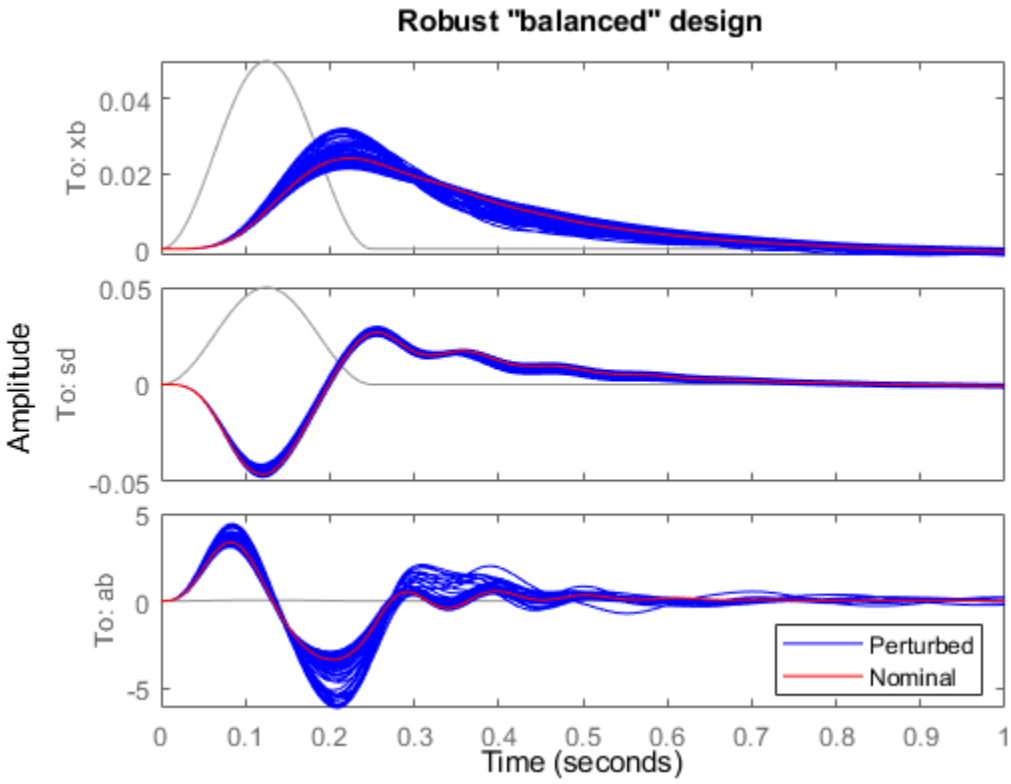


Next simulate the response to a road bump for 100 actuator models randomly selected from the uncertain model set `Act`.

```
rng('default'), nsamp = 100;  clf

% Uncertain closed-loop model with balanced H-infinity controller
CLU = connect(qcar,Act,K(:,:,2),'r',{'xb','sd','ab'});
lsim(usample(CLU,nsamp),'b',CLU.Nominal,'r',roaddist,t)
title('Nominal "balanced" design')
legend('Perturbed','Nominal','location','SouthEast')
```




```
% Uncertain closed-loop model with balanced robust controller
CLU = connect(qcar,Act,Krob,'r',{'xb','sd','ab'});
lsim(usample(CLU,nsamp),'b',CLU.Nominal,'r',roaddist,t)
title('Robust "balanced" design')
legend('Perturbed','Nominal','location','SouthEast')
```



The robust controller `Krob` reduces variability due to model uncertainty and delivers more consistent performance.

Controller Simplification: Order Reduction

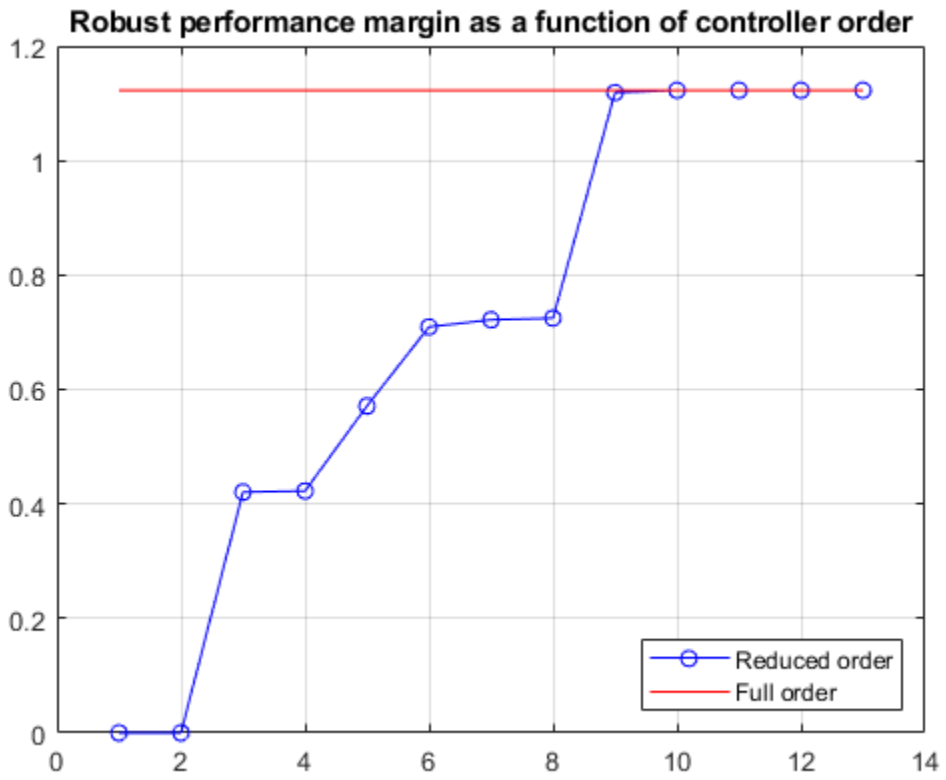
The robust controller `Krob` has relatively high order compared to the plant. You can use the model reduction functions to find a lower-order controller that achieves the same level of robust performance. Use `reduce` to generate approximations of various orders.

```
% Create array of reduced-order controllers
NS = order(Krob);
StateOrders = 1:NS;
Kred = reduce(Krob,StateOrders);
```

Next use `robgain` to compute the robust performance margin for each reduced-order approximation. The performance goals are met when the closed-loop gain is less than $\gamma = 1$. The robust performance margin measures how much uncertainty can be sustained without degrading performance (exceeding $\gamma = 1$). A margin of 1 or more indicates that we can sustain 100% of the specified uncertainty.

```
% Compute robust performance margin for each reduced controller
gamma = 1;
CLP = lft(qcaric(:,:,2),Kred);
for k=1:NS
   PM(k) = robgain(CLP(:,:,k),gamma);
end

% Compare robust performance of reduced- and full-order controllers
```

```
PMfull = PM(end).LowerBound;
plot(StateOrders,[PM.LowerBound],'b-o',...
   StateOrders,repmat(PMfull,[1 NS]),'r');
grid
title('Robust performance margin as a function of controller order')
legend('Reduced order','Full order','location','SouthEast')
```



You can use the smallest controller order for which the robust performance is above 1.

Controller Simplification: Fixed-Order Tuning

Alternatively, you can use `musyn` to directly tune low-order controllers. This is often more effective than a-posteriori reduction of the full-order controller `Krob`. For example, tune a third-order controller to optimize its robust performance.

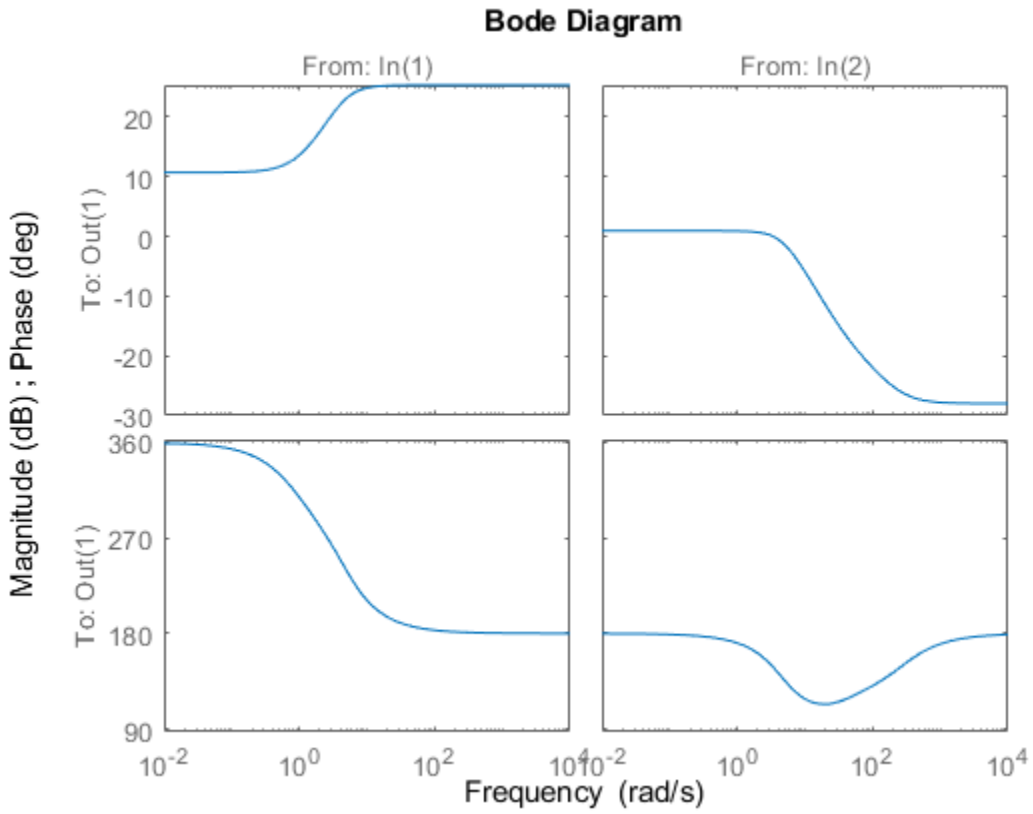
```
% Create tunable 3rd-order controller
K = tunableSS('K',3,ncont,nmeas);

% Tune robust performance of closed-loop system CL
CL0 = lft(qcaric(:,:,2),K);
[CL,RP] = musyn(CL0);
```

```
D-K ITERATION SUMMARY:
--------------------------------------------------------------------
                      Robust performance              Fit order
--------------------------------------------------------------------
  Iter          K Step       Peak MU       D Fit             D
    1           1.189        1.104         1.12             10
```

```
    2          1.076        1.062        1.073           10
    3         0.9899       0.9609       0.9699            6
    4         0.9216       0.9206       0.9326           10
    5         0.9195       0.9157        0.921           10
    6         0.9198       0.9177       0.9272           10

Best achieved robust performance: 0.916
```

The tuned controller has performance `RP=0.92`, very close to that of `Krob`. You can see its Bode response using

```
K3 = getBlockValue(CL,'K');
bode(K3)
```



See Also

`hinfsyn` | `musyn`

Related Examples

- “H-Infinity Performance” on page 5-7

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.

Robust Tuning

Robust Tuning Approaches

Robust Tuning and Multimodel Tuning

The Robust Control Toolbox tuning tools, `systune` and Control System Tuner, allow you to tune control systems for robustness against plant variation. You can tune controllers to accommodate uncertainty in physical parameters.

You can also tune control systems to ensure performance across a range of operating conditions. You can use multimodel tuning to ensure reliable control over multiple system configurations, such as different failure modes of a system. When you tune for multiple models, the software seeks values of controller parameters that best satisfy the specified tuning objectives for all plant models.

Choosing a Robust Tuning Approach

Which approach to take to robust tuning depends on the system variations in your application. The following table summarizes these approaches.

Robust Tuning Scenario	Approach
Tune control system for robustness against parameter uncertainty, such as a mass-spring-damper system in which the spring constant and damping coefficient are uncertain.	Model the uncertain parameter values with `ureal` blocks. See “Tuning for Parameter Uncertainty” on page 6-2.
Tune fixed-structure control system for robustness against real and complex parameter uncertainty and dynamic uncertainty	Model the uncertain parameters with `ureal`, `ucomplex`, and `ultidyn` blocks. Model tunable control system components with control design blocks such as `tunableGain` and `tunablePID` blocks. Use `musyn` to tune the control system to optimize robust H_∞ performance.
Tune control system for a few critical values of the plant parameters.	Simultaneously tune multiple models corresponding to the parameter values. This approach is useful when you cannot model the plant variations as `ureal` blocks. See “Tuning for Parameter Variations” on page 6-3.
• Ensure performance across different operating conditions, such as the response of aircraft flight controls at different altitudes. • Tune for reliable control over multiple system configurations, such as different failure modes of a system.	Simultaneously tune multiple models obtained at different operating points or representing different system configurations. “Tune Against Multiple Plant Models” on page 6-5.

Tuning for Parameter Uncertainty

The physical parameters of a system are often uncertain for various reasons, including imprecise measurements, manufacturing tolerances, or wear and tear. You can use Control System Tuner or the `systune` command to tune control systems for robustness against real parameter uncertainty in the plant. You represent parameter uncertainty in your control system model using uncertain real parameters `ureal`. The software automatically finds the worst combinations of parameter values and tunes the controller to maximize performance over the parameter uncertainty range.

Robust tuning against parameter uncertainty is also useful to avoid “over-tuning” the control system. When you tune against a single plant, the software might optimize performance at the expense of robustness. It is possible to obtain a design that maximizes performance but is not very robust against variations in the plant. Specifying some amount of plant variability lets the tuning software avoid such fragile designs and achieve robust performance, often with only modest degradation of nominal performance.

Control System Modeled in Simulink

To set up a Simulink model of a control system for robust tuning, use linearization with block substitution. (Requires Simulink Control Design™ software.) Use Gain blocks to model the plant parameters and use block substitution to replace them with uncertain values represented by `ureal` objects. Or, replace an entire subsystem with an uncertain state-space model (`uss`) of the subsystem. For more information, see “Model Uncertainty in Simulink for Robust Tuning” on page 6-17.

As with control systems modeled in MATLAB, the software automatically tunes the model for the worst combination of parameter values within the uncertainty range.

Control Systems Modeled in MATLAB

To represent real parameter uncertainty in the plant, build a generalized state-space (`genss`) model of the control system using `ureal` blocks. Use control design blocks such as `tunablePID` or `tunableTF` to represent tunable controller elements in the model. (See “Build Tunable Control System Model With Uncertain Parameters” on page 6-13.) Tune the model with `systune` or in Control System Tuner exactly as you would for a tunable control system model without uncertainty.

- **Command line:** Use the `genss` model as the first input argument to `systune`. For a detailed example, see “Robust Tuning of Positioning System” on page 6-40.
- **Control System Tuner:** Import the model into the app by selecting **Edit Architecture** > **Generalized feedback configuration** and entering the name of the `genss` model into the text box. Then, use the app exactly as you would for a control system model without uncertainty.

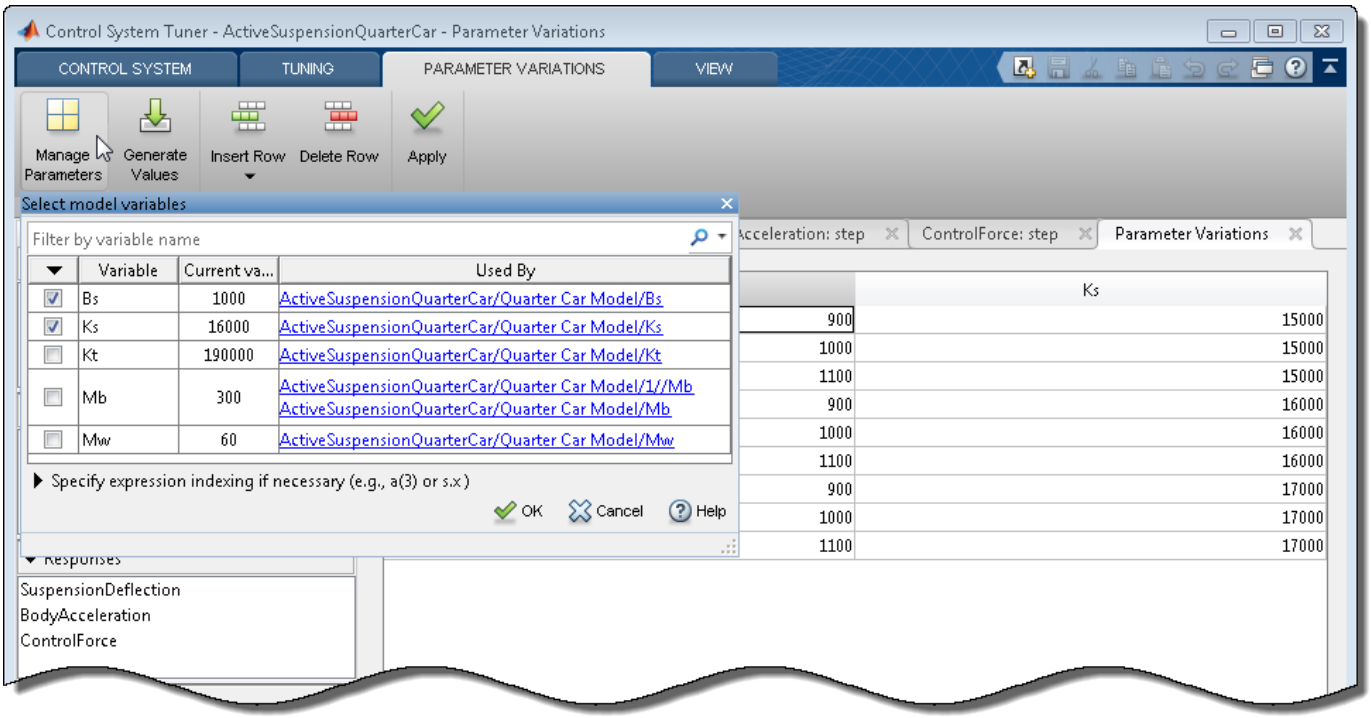
In both cases, when you tune the model, the software automatically adjusts the tunable components to optimize performance throughout the uncertainty range. Analysis plots automatically display random samples of the uncertain system to give you a visual sense of the performance variation.

Tuning for Parameter Variations

The block-substitution approach to modeling uncertainty, requires replacing an entire block of your model with a `ureal` parameter or `uss` uncertain system. In some cases, you might not be able to make such a substitution. As an alternative, you can vary system parameters over a specified range, grid, or nonuniform set of values. When you use `systune` or Control System Tuner to tune a system with parameter variation, you can obtain a controller that robustly meets performance goals over a range of model-coefficient values or over multiple plant configurations.

Specifying Parameter Variations in Control System Tuner

In Control System Tuner, specify block-parameter variations on the **Control System** tab. In the **Parameter Variations** drop-down list, select `Select parameters to vary`. This action opens the **Parameter Variations** tab, in which you can specify the block parameters to vary and the values they take. Control System Tuner linearizes your Simulink model at each combination of block-parameter values that you provide. The app then finds a set of controller gains that best meets your tuning goals for all the linearized models simultaneously.



For a detailed example that shows how to use Control System Tuner to tune a control system for multiple values of block parameters, see “Tuning for Multiple Values of Plant Parameters” on page 6-65.

For more information about using the **Parameter Variations** tab to generate linear models at multiple values of block parameters, see “Specify Parameter Samples for Batch Linearization” (Simulink Control Design). The procedure for applying parameter variation in **Model Linearizer** is similar to the procedure in Control System Tuner.

Specifying Parameter Variations With slTuner

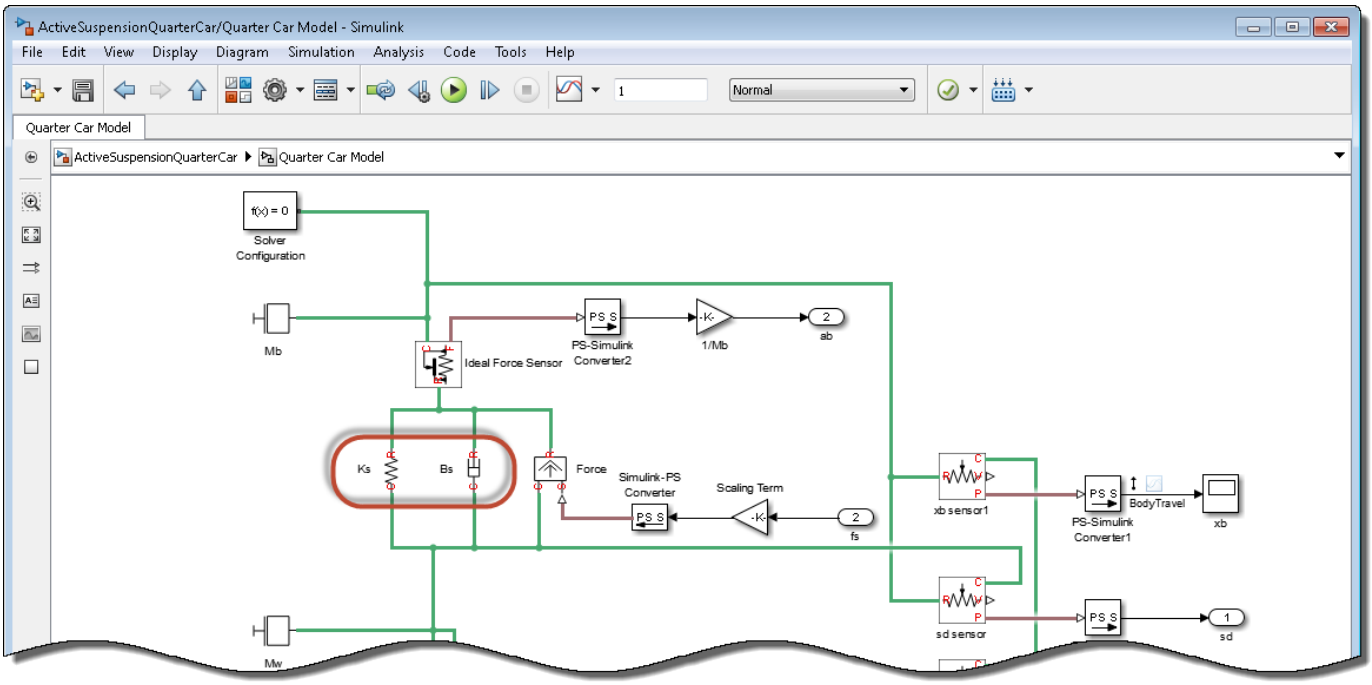
For command-line tuning of a control system modeled in Simulink, use the parameter-variation feature of `slTuner`. To so, you construct a structure that contains the parameter-value grid over which you want to tune the model. For an example illustrating parameter variation with `slLinearizer`, see “Vary Parameter Values and Obtain Multiple Transfer Functions” (Simulink Control Design). The procedure for configuring an `slTuner` interface for parameter variations is the same. After you configure the `slTuner` interface, create tuning goals and tune the interface with `systune`. The software tunes the system to meet your tuning goals for all parameter values simultaneously.

Varying Block Parameters vs. Tuning Controller Parameters

The block parameters that you vary to generate multiple plant models are different from the controller parameters that you tune to meet your tuning goals.

Block parameters are the values that specify attributes of the blocks in your Simulink model. Block parameters can specify numeric values such as the gain of a gain block, a spring constant, or other physical parameters of a system. Block parameters can also specify structural attributes of a block, such as the dimensions of a lookup table.

You can vary any block parameter whose value is stored as a variable in the model workspace or MATLAB workspace. However, do not vary the controller-block parameters that you designate for tuning (see “Specify Blocks to Tune in Control System Tuner”). Rather, vary parameters that specify attributes of the plant in your control system. For example, in the model `ActiveSuspensionQuarterCar`, block parameters specified as variables include a spring constant, `Ks`, and a damping constant, `Bs`.



The example “Tuning for Multiple Values of Plant Parameters” on page 6-65 shows how to tune the control system of the `ActiveSuspensionQuarterCar` model for a range of values of these parameters.

Controller parameters are the coefficients that the tuning software adjusts to yield control system performance that meets your tuning goals. When you select blocks to tune, the software assigns a parameterization to each block, as described in “View and Change Block Parameterization in Control System Tuner”. The coefficients of these parameterizations are the controller parameters that the software tunes. For example, if you select a PID Controller block to tune, the tuning software assigns a parameterization whose tunable coefficients are the PID gains and filter constant.

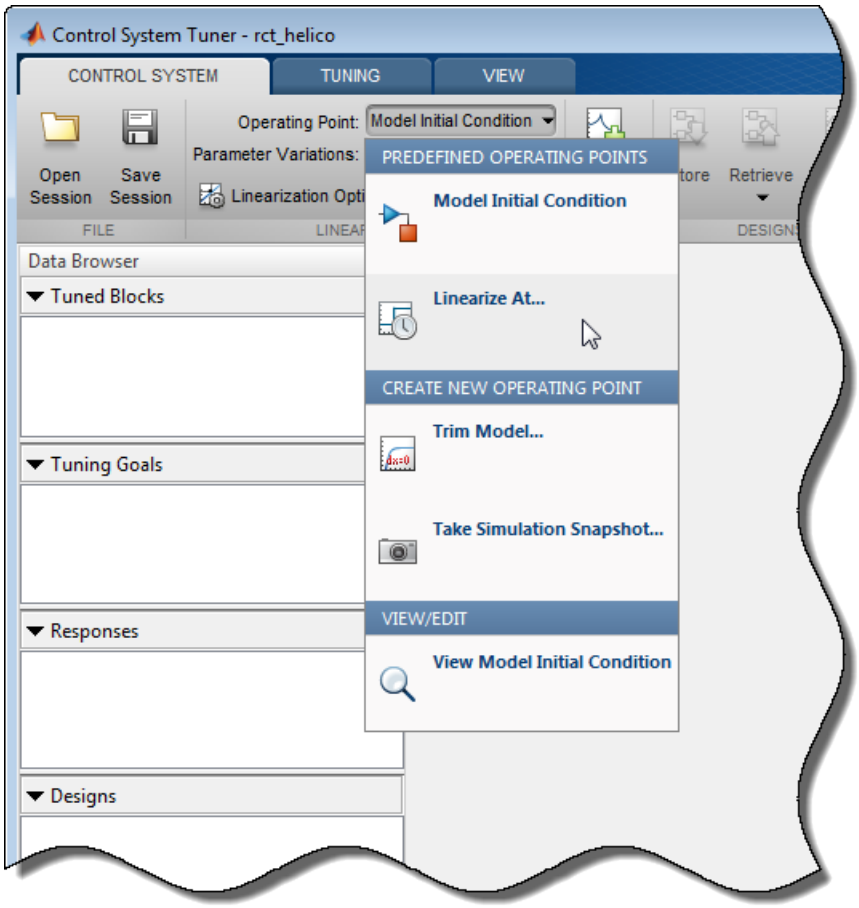
Thus, you specify controller parameters by selecting blocks to tune, and optionally customizing the parameterization of those blocks. You specify other system parameters to vary to obtain multiple plant models for tuning. In the example “Tuning for Multiple Values of Plant Parameters” on page 6-65, the block selected for tuning is a State-Space block. In that example, the controller parameters are the entries in the state-space matrices.

Tune Against Multiple Plant Models

When you tune controller gains against multiple models, the software seeks values of controller parameters that best satisfy the specified tuning objectives for all plant models. This is useful to ensure robust performance across a range of operating conditions, or for multiple system configurations.

Tuning for Multiple Operating Points

Control System Tuner can tune controller parameters for a linearization of your Simulink model obtained at any simulation snapshot time or steady-state operating point. In the **Control System** tab, use the **Operating Point** menu to compute and select operating points at which to linearize and tune.



See “Specify Operating Points for Tuning in Control System Tuner” for more information.

If you specify multiple operating points, Control System Tuner attempts to tune controller parameters to satisfy your tuning goals at all the specified operating points. You can restrict which tuning goals Control System Tuner enforces at each operating point. See “Selective Application of Tuning Goals” on page 6-7.

At the command line, you can tune for multiple operating points by passing an array of operating-point objects to `slTuner`.

Tuning for Multiple System Configurations

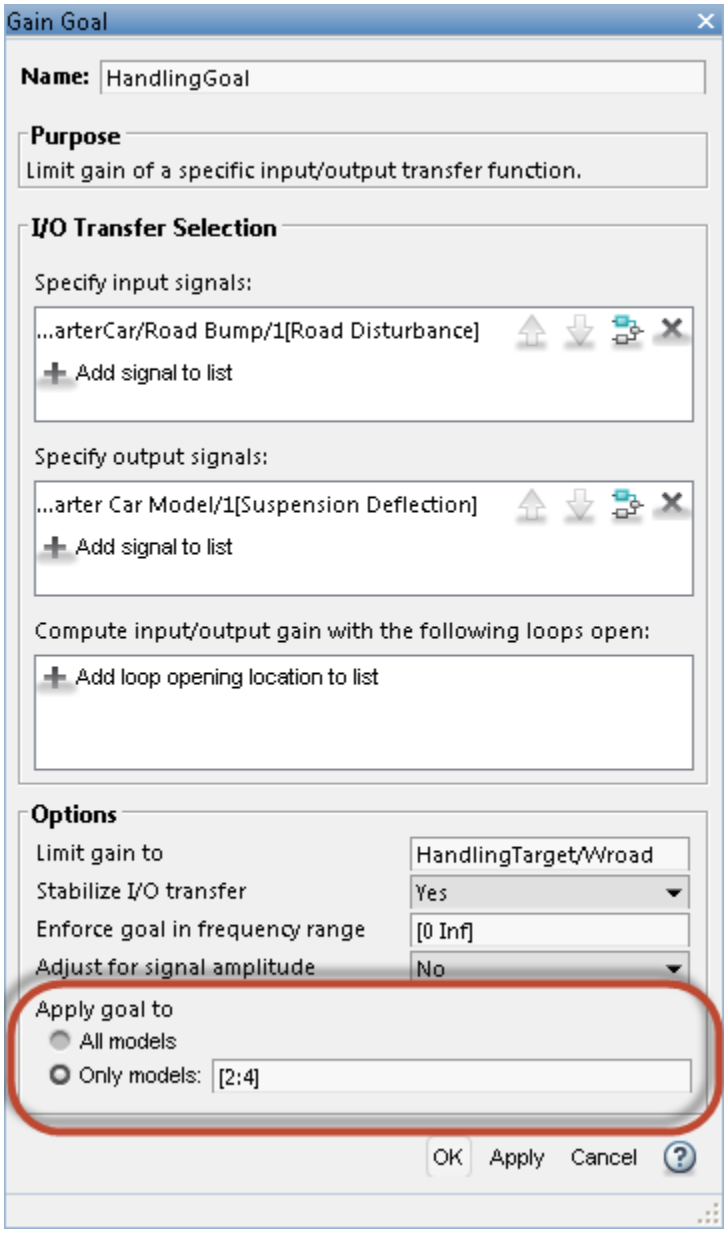
You can tune a controller that is robust against multiple system configurations by building an array of models representing those conditions. For example, you can create an array of `genss` models that represent different failure modes of the system. In Simulink, use `slTuner` to linearize your model under an array of operating conditions that represent different failure modes. For an example, see the model in “Fault-Tolerant Control of a Passenger Jet” on page 6-56. That model uses a gain block that,

when set to zero, breaks a feedback loop to simulate the loss of control of a system actuator. The example then uses `slTuner` to sample the model with different channels of this gain block set to zero. Tuning that `slTuner` with `systune` finds values of tunable controller parameters that optimize the design goals over all failure modes.

Selective Application of Tuning Goals

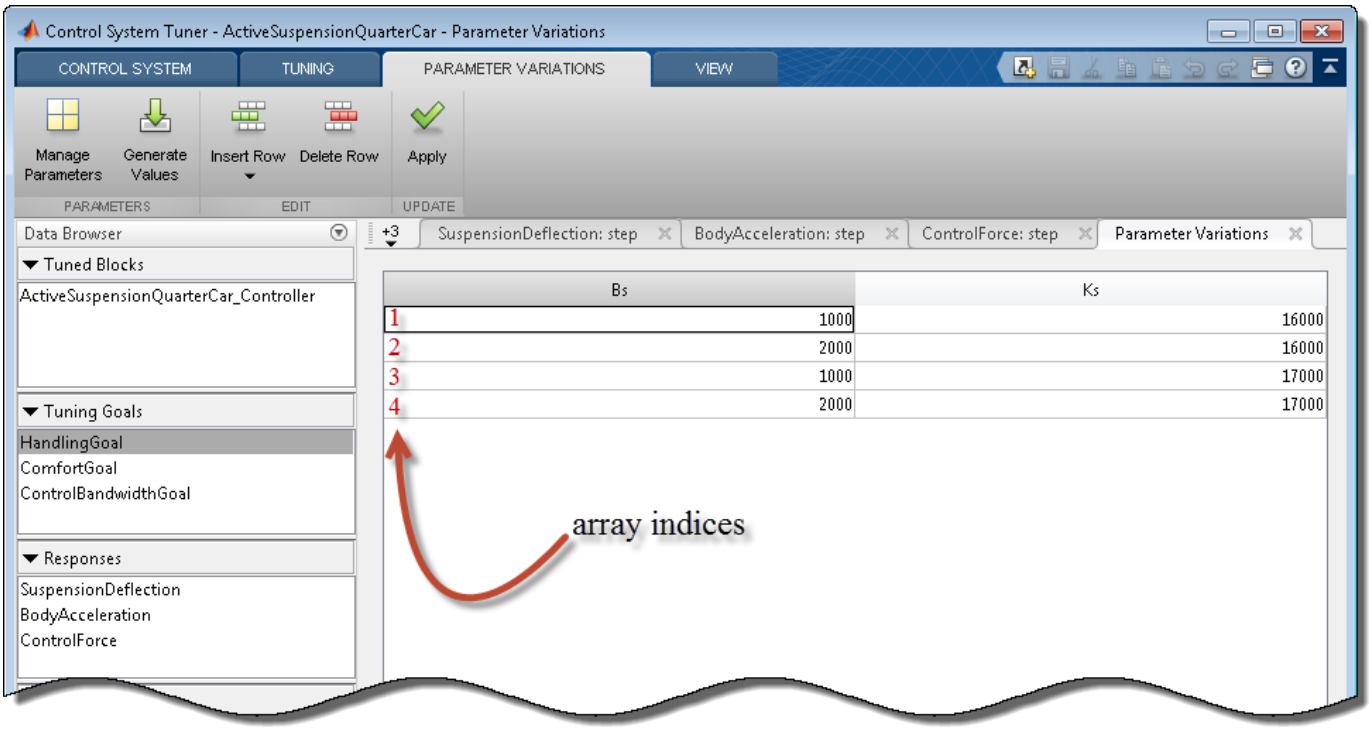
Sometimes you want to restrict application of your tuning goals to a subset of the models for which you are simultaneously tuning. For example, suppose that you linearize your model at four snapshot times, `t = [0,5,10,20]`. You want to tune the model to meet your design goals at all these times. However, suppose further that you have one tuning goal that you do not want to enforce at `t = 0` because it should only apply after the model has reached steady state operation. To limit the application of this tuning goal:

- At the command line, set the `Models` property of the tuning goal to the array indices of the models to which you want to apply the goal.
- In Control System Tuner, use the **Apply goal to** field of the tuning goal.



Select **Only models** and enter the array indices of the models for which the goal is enforced. In this example, linearizing at `t = [0,5,10,20]` yields an array of four models, and you want to exclude the first model in that array (`t = 0`) from the tuning goal. Therefore, enter array indices `2:4`.

For multiple models obtained using the **Parameter Variations** tab, array indices are assigned in the order that parameter combinations appear in the Parameter Variations table. For example, if you apply the parameter variations of the following illustration, array indices are assigned as shown.



Thus, for example, to apply a tuning goal only to those models with `Bs = 1000`, regardless of the `Ks` value, enter `[1,3]` in the **Only models** field of the tuning goal.

Application to Nominal System

When performing robust tuning of a system with parameter uncertainty, you sometimes want to apply certain tuning goals to the nominal system only. Or, you might want to treat a tuning goal as a hard constraint for the nominal system, but as a soft constraint over the rest of the uncertainty range. When tuning a control system modeled in MATLAB, you can do this by putting the nominal system in an model array with the uncertain system. For example, suppose `CL0` is a `genss` model having both uncertain and tunable blocks. Create a model array of the nominal and full uncertain systems.

```
CL = [getNominal(CL0),CL0];
```

Suppose that you have created two tuning goals for this system, `Req1` and `Req2`. You want `Req2` to apply to the nominal system only. To do so, use the `Models` property to restrict `Req2` to the first entry in the array.

```
Req2.Models = [1];
```

You can now use `Req2` as with `systune` as either a hard goal or a soft goal.

To treat `Req2` as a hard constraint for the nominal system and a soft constraint otherwise, make a copy of the tuning goal. To restrict the copy to the second entry in the array, set the `Models` property of the copy.

```
Req3 = Req2;
Req3.Models = [2];
hard = [Req1,Req2];
soft = Req3;
[CLt,fSoft,gHard] = systune(CL,soft,hard);
```

See Also

`slTuner` | `systune (for slTuner)` | `systune (for genss)` | `replaceBlock`

Related Examples

- “Model Uncertainty in Simulink for Robust Tuning” on page 6-17
- “Tuning for Multiple Values of Plant Parameters” on page 6-65
- “Robust Tuning of Positioning System” on page 6-40

Interpreting Results of Robust Tuning

When you tune a control system with `systune` or Control System Tuner, the software reports on the tuning progress and results as described in “Interpret Numeric Tuning Results”. When you tune a control system with parameter uncertainty, the results contain additional information about the progress of the tuning algorithm toward tuning for the worst-case parameter values.

Robust Tuning Algorithm

The software begins the robust tuning process by tuning for the nominal plant model. Then, the software performs the following steps iteratively:

1. Identify a parameter combination within the uncertainty ranges that violates the design requirements (analysis step).
2. Adds a model evaluated at these parameter values to the set of models over which the software is tuning.
3. Repeats tuning for the expanded model set (tuning step).

This process terminates when the analysis step is unable to find a parameter combination that yields a significantly worse performance index than the value obtained in the last iteration of the tuning step. The performance index is a weighted combination of the soft constraint value `fSoft` and the hard constraint value `gHard`. (See “Interpret Numeric Tuning Results” for more information.)

Displayed Results

The result is that on each iteration of this process, the algorithm returns a range of values for each of `fSoft` and `gHard`. The minimum is the best achieved value for that iteration, tuning the controller parameters over all the models in the expanded model set. The maximum is the worst value the software can find in the uncertainty range, using that design (set of tuned controller-parameter values). This range is reflected in the default display at the command line or in the Tuning Report in Control System Tuner. For example, the following is a typical report for robust tuning of an uncertain system using only soft constraints.

```
Soft: [0.906,18.3], Hard: [-Inf,-Inf], Iterations = 106
Soft: [1.02,3.77], Hard: [-Inf,-Inf], Iterations = 55
Soft: [1.25,1.85], Hard: [-Inf,-Inf], Iterations = 67
Soft: [1.26,1.26], Hard: [-Inf,-Inf], Iterations = 24
Final: Soft = 1.26, Hard = -Inf, Iterations = 252
```

Each of the first four lines corresponds to one iteration in the robust tuning process. In the first iteration, the soft goals are satisfied for the nominal system (`fSoft < 1`). That design is not robust against the entire uncertainty range, as shown by the worst-case `fSoft = 18.3`. Adding that worst-case model to the expanded model set, the algorithm finds a new design with `fSoft = 1.02`. Testing that design over the uncertainty range yields a worst case of `fSoft = 3.77`. With each iteration, the gap between the performance of the model set used for tuning and the worst-case performance narrows. In the final iteration, the worst-case performance matches the multi-model performance. The multi-model values typically increase as the algorithm tunes the controller against a larger set of models, so that the robust `fSoft` and `gHard` values are typically larger than the nominal values. `systune` returns the final values as output arguments.

Robust Tuning With Random Starts

When you use `systuneOptions` to set `RandomStart > 0`, the tuning software performs nominal tuning from each of the random starting points. It then performs the robust tuning process on each nominal design, starting with the best design. The “robustification” of any particular design is aborted when the minimum value of `fSoft` (the lower bound on robust performance) becomes much higher than the best robust performance achieved so far.

The default display includes the `fSoft` and `gHard` values for all the nominal designs and the results of each robust-tuning iteration. The software selects the best result of robust tuning from among the randomly started designs.

Validation

The robust-tuning algorithm finds locally optimal designs that meet your design requirements. However, identifying the worst-case parameter combinations for a given design is a difficult process. Although it rarely happens in practice, it is possible for the algorithm to miss a worst-case parameter combination. Therefore, independent confirmation of robustness, such as using μ-analysis, is recommended.

See Also

Related Examples

- “Robust Tuning of DC Motor Controller” on page 6-32
- “Robust Tuning of Mass-Spring-Damper System” on page 6-24

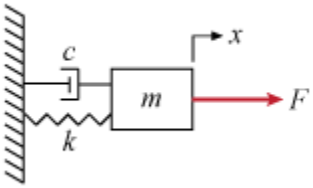
More About

- “Robust Tuning Approaches” on page 6-2

Build Tunable Control System Model With Uncertain Parameters

This example shows how to construct a generalized state-space (`genss`) model of a control system that has both tunable and uncertain parameters. You can use `systune` to tune the tunable parameters of such a model to achieve performance that is robust against the uncertainty in the system.

For this example, the plant is a mass-spring-damper system. The input is the applied force, F, and the output is x, the position of the mass.



In this system, the mass m, the damping constant c, and the spring constant k all have some uncertainty. Use uncertain `ureal` parameters to represent these quantities in terms of their nominal or most probable value and a range of uncertainty around that value.

```
um = ureal('m',3,'Percentage',40);
uc = ureal('c',1,'Percentage',20);
uk = ureal('k',2,'Percentage',30);
```

The transfer function of a mass-spring-damper system is a second-order function given by:

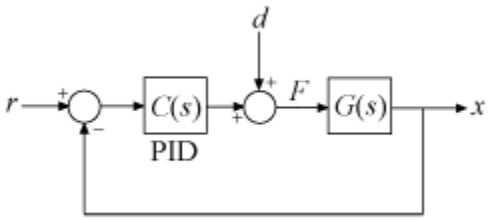
$$G(s) = \frac{1}{ms^2 + cs + k}.$$

Create this transfer function in MATLAB® using the uncertain parameters and the `tf` command. The result is an uncertain state-space (`uss`) model.

```
G = tf(1,[um uc uk])
```

```
G =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    c: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    k: Uncertain real, nominal = 2, variability = [-30,30]%, 1 occurrences
    m: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

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

Suppose you want to control this system with a PID controller, and that your design requirements include monitoring the response to noise at the plant input. Build a model of the following control system.



Use a tunable PID controller, and insert an analysis point to provide access to the disturbance input.

```
C0 = tunablePID('C','PID');
d = AnalysisPoint('d');
```

Connect all the components to create the control system model.

```
T0 = feedback(G*d*C0,1)
```

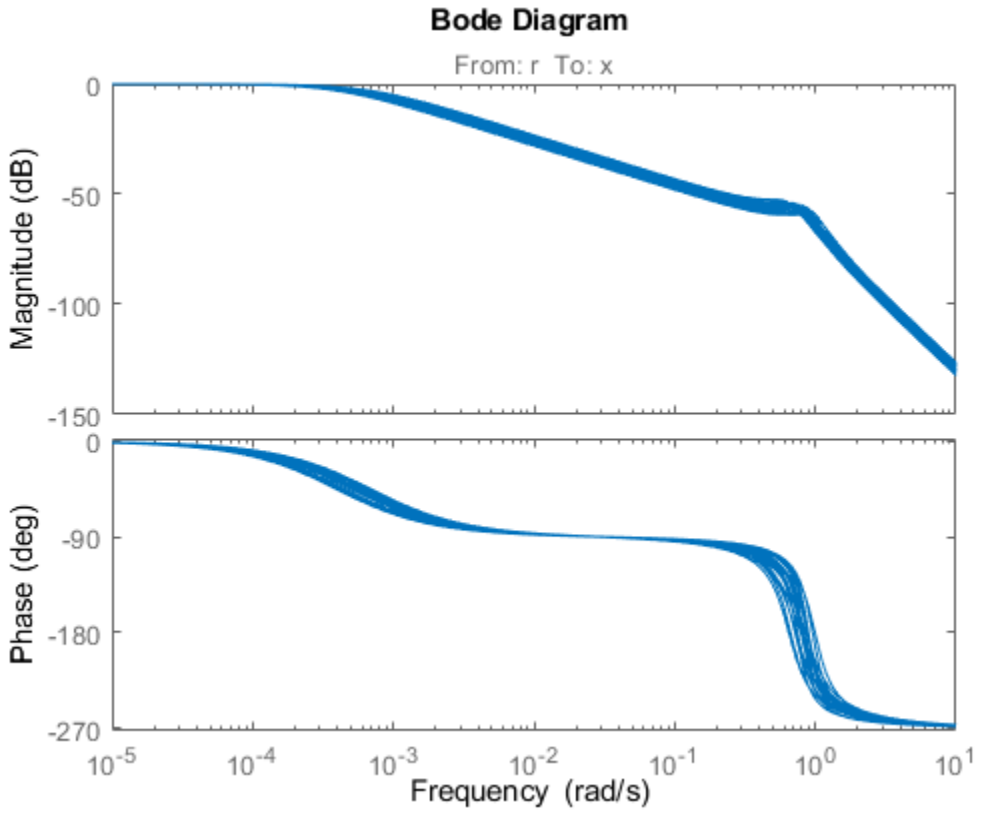
```
T0 =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and the follo
    C: Tunable PID controller, 1 occurrences.
    c: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    d: Analysis point, 1 channels, 1 occurrences.
    k: Uncertain real, nominal = 2, variability = [-30,30]%, 1 occurrences
    m: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "ss(T0)" to see the current value, "get(T0)" to see all properties, and "T0.Blocks" to inte
```

```
T0.InputName = 'r';
T0.OutputName = 'x';
```

`T0` is a generalized state-space (`genss`) model that has both tunable and uncertain blocks. In general, you can use `feedback` and other model interconnection commands, such as `connect`, to build up models of more complex tunable and uncertain control systems from fixed-value LTI components, uncertain components, and tunable components.

When you plot system responses of a `genss` model that is both tunable and uncertain, the plot displays multiple responses computed at random values of the uncertain components. This sampling provides a general sense of the range of possible responses. All plots use the current value of the tunable components.

```
bodeplot(T0)
```

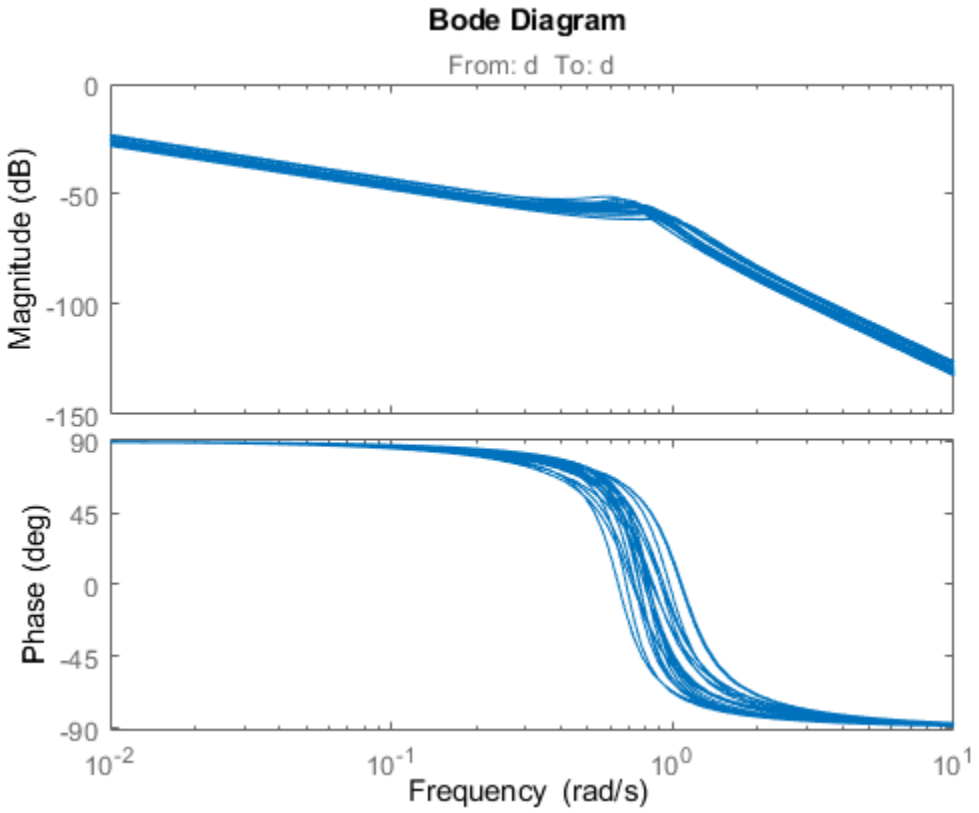


When you extract responses from a tunable and uncertain `genss` model, the responses also contain both tunable and uncertain blocks. For example, examine the loop transfer function at the disturbance input.

```
S0 = getLoopTransfer(T0,'d')
```

```
S0 =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and the follo
    C: Tunable PID controller, 1 occurrences.
    c: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    d: Analysis point, 1 channels, 1 occurrences.
    k: Uncertain real, nominal = 2, variability = [-30,30]%, 1 occurrences
    m: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "ss(S0)" to see the current value, "get(S0)" to see all properties, and "S0.Blocks" to inte
```

```
bodeplot(S0)
```



You can now create tuning goals and use `systune` to tune the PID controller coefficients of T0. When you do so, `systune` automatically tunes the coefficients to maximize performance over the full range of uncertainty.

See Also

`ureal` | `genss` | `AnalysisPoint` | `connect`

Related Examples

- “Robust Tuning of DC Motor Controller” on page 6-32
- “Model Uncertainty in Simulink for Robust Tuning” on page 6-17

More About

- “Robust Tuning Approaches” on page 6-2

Model Uncertainty in Simulink for Robust Tuning

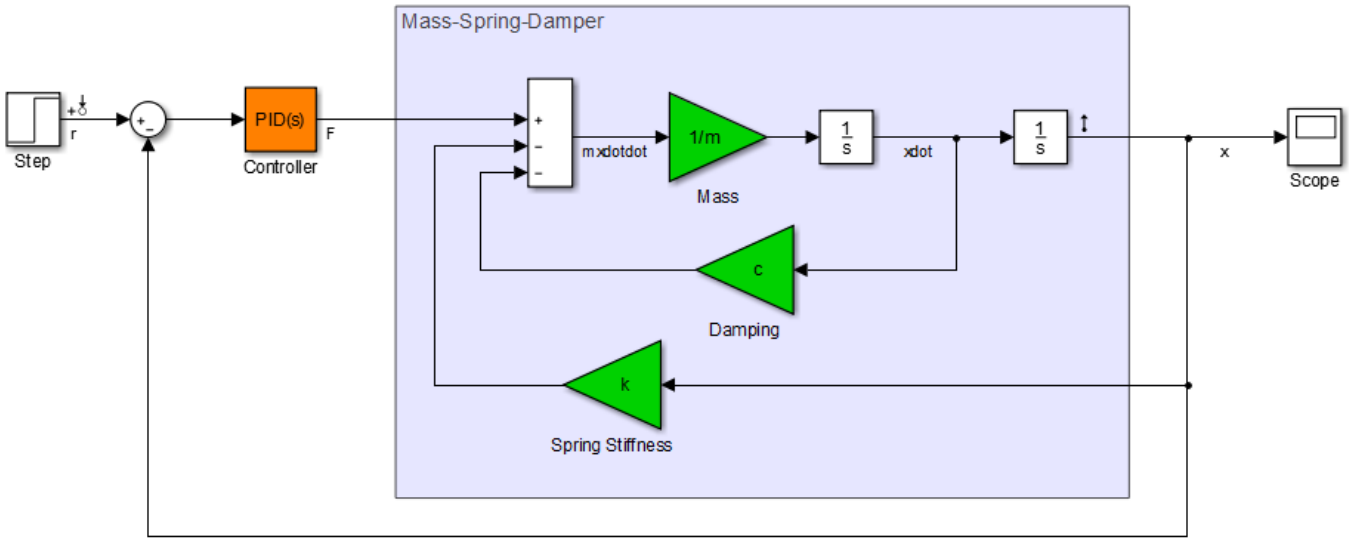
This example shows how to set up a Simulink model for robust tuning against parameter uncertainty. Robust controller tuning or robust controller synthesis for a system modeled in Simulink requires linearizing the model such that the software takes parameter uncertainty into account. Doing so requires block substitution (Simulink Control Design) for linearization, to replace the value of blocks that have parameter uncertainty with uncertain parameters or systems.

In this example, you set up a model of a mass-spring-damper system for robust tuning, where the physical parameters of the system are uncertain. The example shows how to set up the model for robust tuning using software such as Control System Tuner or `systune` for `slTuner`. It also shows how to extract an uncertain system to use for robust controller design with `musyn`.

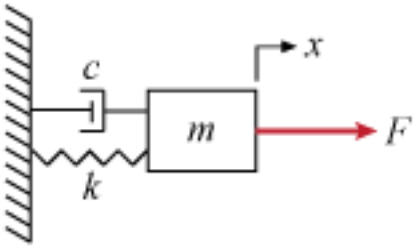
Mass-Spring-Damper System

Open the Simulink model `rct_mass_spring_damper`.

```
open_system('rct_mass_spring_damper')
```



This model represents a system for controlling the mass-spring damper system of the following illustration.



In this system, the applied force F is the plant input. The PID controller generates the force necessary to control the mass position x. When the mass m, the damping constant c, and the spring constant k are fixed and known, tuning the PID coefficients for desired performance is straightforward. In practice, however, physical system parameters can be uncertain. You can use Control System Tuner or `systune` to tune the system robustly against the uncertainty, and achieve satisfactory performance within the range of expected values for these parameters.

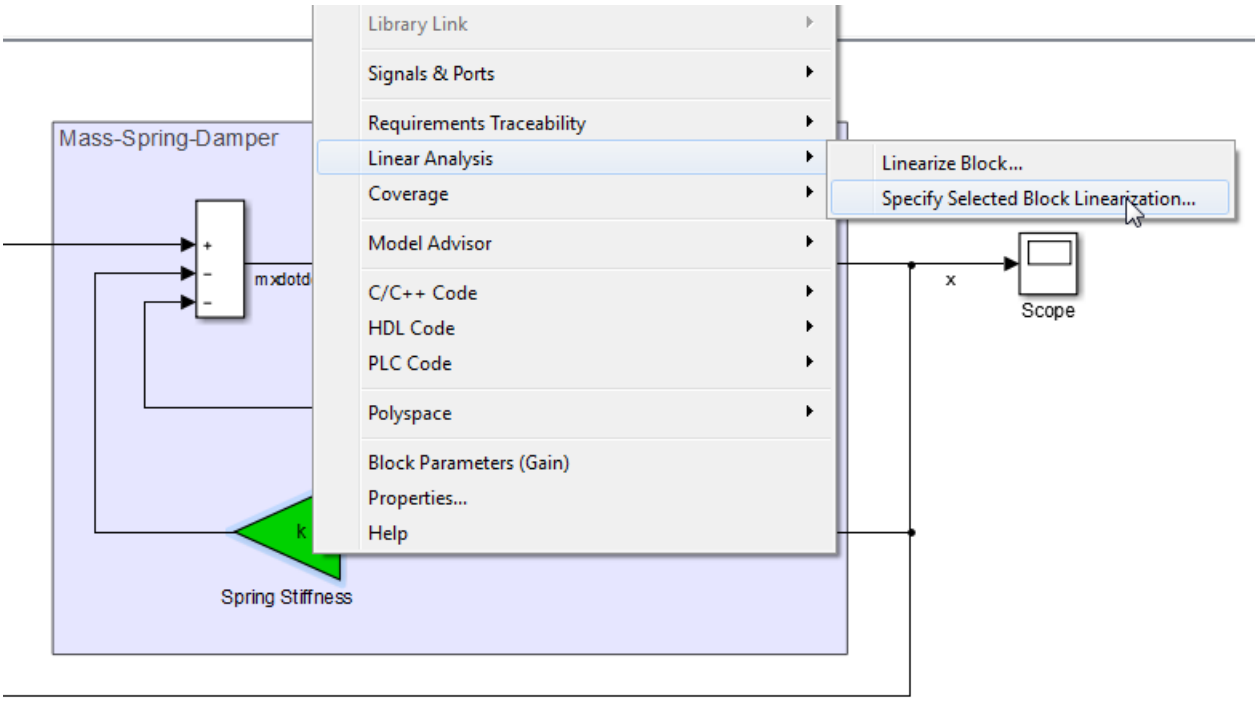
Specify Parameter Uncertainty

The model is configured to use the nominal or most probable values of the physical parameters, $m = 3$, $c = 1$, and $k = 2$. To tune the system against uncertainty in these parameters, specify the parameter uncertainty in the model.

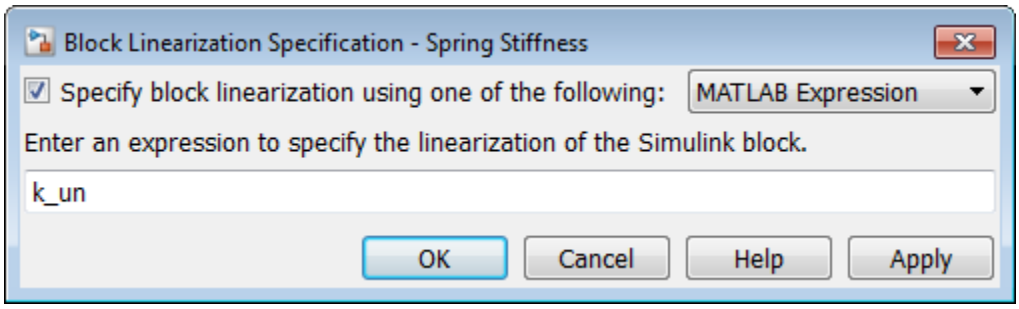
First, create uncertain real (`ureal`) parameters for each of the three uncertainties. For this example, specify the uncertainty as a percentage variation from the nominal value.

```
m_un = ureal('m',3,'Percentage',40);
c_un = ureal('c',1,'Percentage',20);
k_un = ureal('k',2,'Percentage',30);
```

To specify these uncertainties in the model, use block substitution. Block substitution lets you specify the linearization of a particular block in a Simulink model. In the model, right-click the `Spring Stiffness` block in the model and select **Linear Analysis > Specify Selected Block Linearization**.



In the Block Linearization Specification dialog box, check **Specify block linearization using one of the following** and enter `k_un` in the text field. Click **OK**.

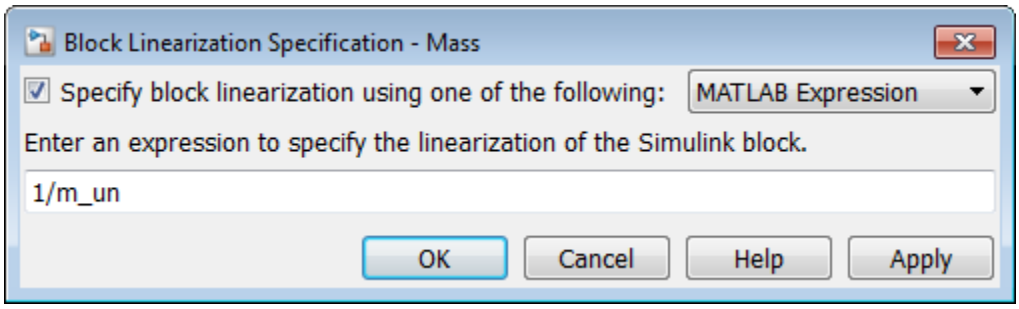


When you use Control System Tuner for this model, the software linearizes the model and tunes the tunable parameters using that linearization to compute system responses. Specifying `k_un` as the linearization of the `Spring Stiffness` block causes the software to use the uncertain parameter as the linearized value of the block instead of its nominal value, which is a constant, fixed gain of 2.

Because the uncertain parameters in this model, such as the spring stiffness, are implemented as scalar gain blocks, use a simple `ureal` parameter as the block substitution. For more complex blocks, construct a `uss` model that represents the uncertain value of the entire block.

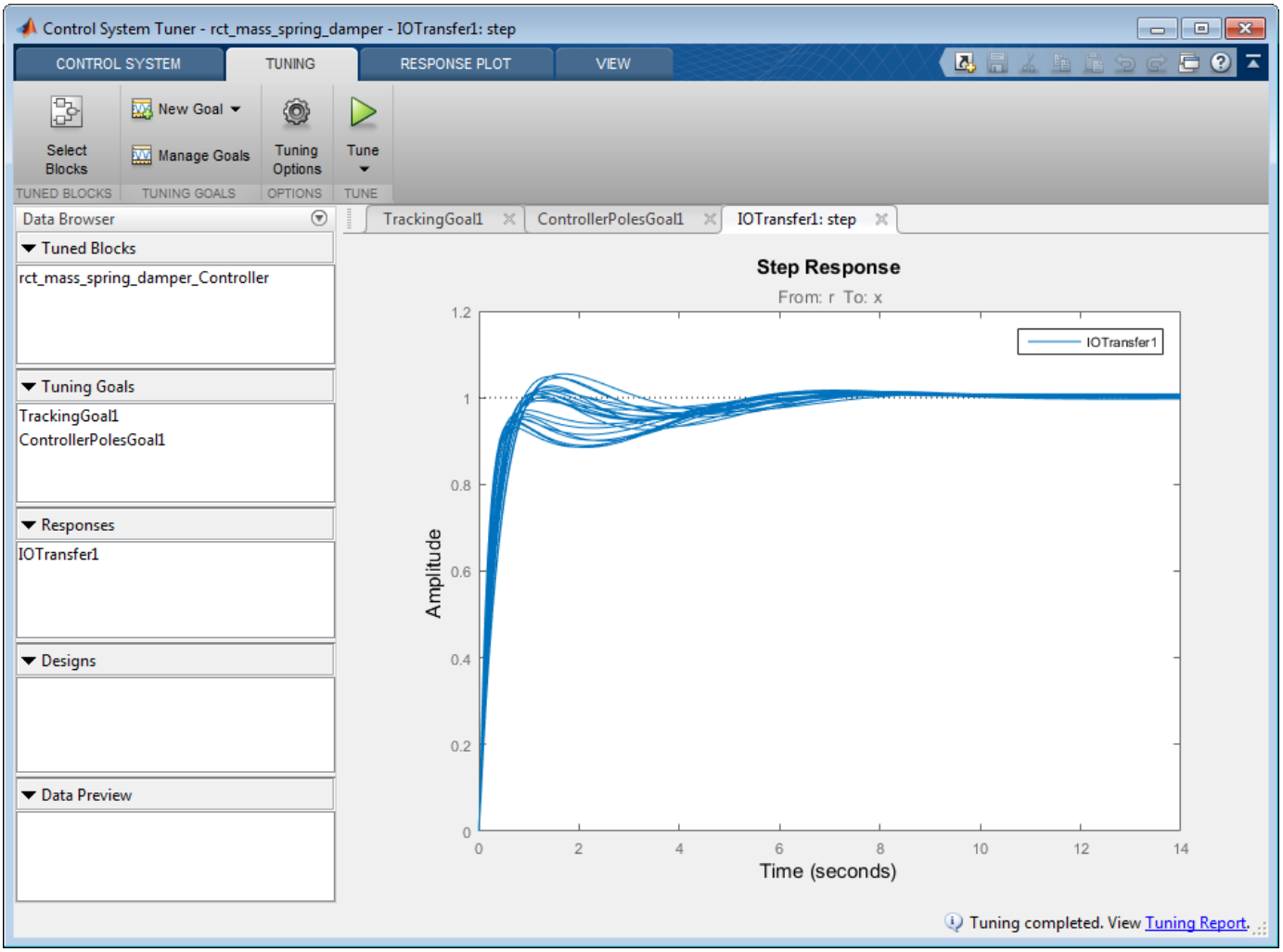
Note Use block substitution to specify the uncertainty of the block even if the block is an `Uncertain LTI System` block. Unless you explicitly specify the uncertain value as the block substitution, Control System Tuner and `slTuner` use the nominal value when linearizing `Uncertain LTI System` blocks.

In the same way, specify `c_un` as the block linearization for the `Damping` block. For the `Mass` block, in the Block Linearization Specification dialog box, enter `1/m_un` as the uncertain value, because the gain of this block is the inverse of the mass.



Tune With Control System Tuner

You can now open Control System Tuner for the model, create tuning goals, and tune the model. When you do so, Control System Tuner tunes the controller parameters to optimize performance over the entire range of uncertainty. Tuning-goal plots and response plots in Control System Tuner display multiple responses computed at random values of the uncertain parameters, as shown.



This sampling provides a general sense of the range of possible responses, but does not necessarily reflect the true worst-case response.

Configuration for `slTuner`

When you use `slTuner` for command-line tuning, you can specify uncertainties in the model using the Block Linearization Specification dialog box. Alternatively, you can specify the uncertain block substitutions without altering the model. To do so, use a block-substitution structure when you create the `slTuner` interface. For example, create a block-substitution structure for the `rct_mass_spring_damper` model.

```
blocksubs(1).Name = 'rct_mass_spring_damper/Mass';
blocksubs(1).Value = 1/m_un;
blocksubs(2).Name = 'rct_mass_spring_damper/Damping';
blocksubs(2).Value = c_un;
blocksubs(3).Name = 'rct_mass_spring_damper/Spring Stiffness';
blocksubs(3).Value = k_un;
```

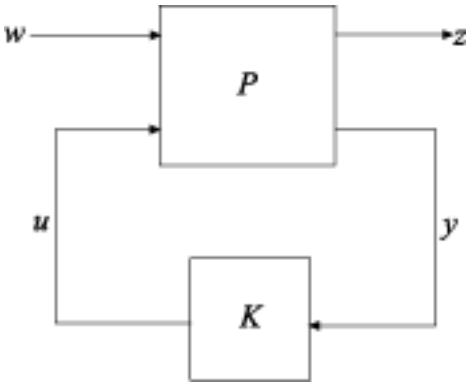
Use this structure to obtain an `slTuner` interface to the model with the uncertain values.

```
UST0 = slTuner('rct_mass_spring_damper','Controller',blocksubs);
```

You can now create tuning goals and tune the model. `systune` tunes the system to optimize performance over the entire range of uncertainty. For an example illustrating this robust-tuning workflow with `slTuner`, see “Robust Tuning of Mass-Spring-Damper System” on page 6-24.

Extract `uss` Plant Model for Robust Controller Design with `musyn`

The `musyn` command synthesizes a robust controller for a plant assuming an LFT control configuration.



Mapping this structure to the Simulink model,

- *w* is the reference input `r`, the output of the Step block.
- *u* is the control signal `F`, the output of the PID Controller block.
- *z* is the plant output `x`, the output of the Integrator block.
- *y* is the measurement signal, which is the controller input, or the output of the Sum block.

Use these signals with the `getIOTransfer` command to extract the plant *P* from the `slTuner` interface `UST0`. To do so, `UST0` must have analysis points defined at each of these locations. Examine the analysis points of `UST0`.

```
getPoints(UST0)
```

```
ans =

  2×1 cell array

    {'rct_mass_spring_damper/Step/1[r]'      }
    {'rct_mass_spring_damper/Integrator/1[x]'}
```

There are already analysis points for *w* and *z*. Add the analysis points for *u* and *y*.

```
addPoint(UST0,{'Sum1','Controller'});
getPoints(UST0)
```

```
ans =

  4×1 cell array

    {'rct_mass_spring_damper/Step/1[r]'      }
    {'rct_mass_spring_damper/Integrator/1[x]'}
    {'rct_mass_spring_damper/Sum1/1'         }
    {'rct_mass_spring_damper/Controller/1[F]'}
```

You can now extract the plant model *P* for tuning with `musyn`. Use the analysis-point signal names, shown in brackets in the output of `getPoints`, to specify the inputs and outputs of *P*. For analysis points that do not have signal names, use the block name.

```
Pg = getIOTransfer(UST0,{'r','F'},{'x','Sum'});
```

`getIOTransfer` returns a `genss` model. In this case, because `Pg` excludes the controller block, `Pg` is a `genss` model with uncertain blocks only. Convert `Pg` to `uss` for controller design with `musyn`.

```
P = uss(P)
```

```
P =

  Uncertain continuous-time state-space model with 2 outputs, 2 inputs, 3 states.
  The model uncertainty consists of the following blocks:
    c: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    k: Uncertain real, nominal = 2, variability = [-30,30]%, 1 occurrences
    m: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "P.NominalValue" to see the nominal value, "get(P)" to see all properties, and
"P.Uncertainty" to interact with the uncertain elements.
```

You can now use `musyn` to design a robust controller for *P*. For instance, to design an unstructured robust controller, note that *P* has one measurement signal and one control signal, and use the following command.

```
[K,CLperf,info] = musyn(P,1,1);
```

Alternatively, design a fixed-structure PID controller, as in the original Simulink model.

```
C0 = tunablePID('K','PID');
CL0 = lft(P,C0);
[CL,CLperf,info] = musyn(CL0);
```

For more information about robust controller design, see `musyn`.

See Also

`slTuner` | `systune` | `systune (for slTuner)` | `musyn` | `getIOTransfer`

Related Examples

- “Robust Tuning of Mass-Spring-Damper System” on page 6-24
- “Build Tunable Control System Model With Uncertain Parameters” on page 6-13
- “Robust Tuning Approaches” on page 6-2
- “Robust Controller Design Using Mu Synthesis”

Robust Tuning of Mass-Spring-Damper System

This example shows how to robustly tune a PID controller for an uncertain mass-spring-damper system modeled in Simulink.

Simulink Model of Mass-Spring-Damper System

The mass-spring-damper depicted in Figure 1 is modeled by the second-order differential equation

$$m\ddot{x} + c\dot{x} + kx = F$$

where F is the force applied to the mass and x is the horizontal position of the mass.

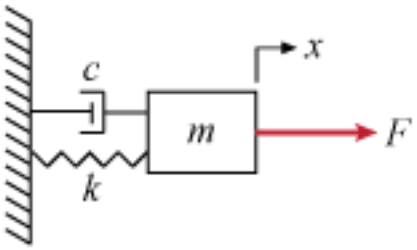
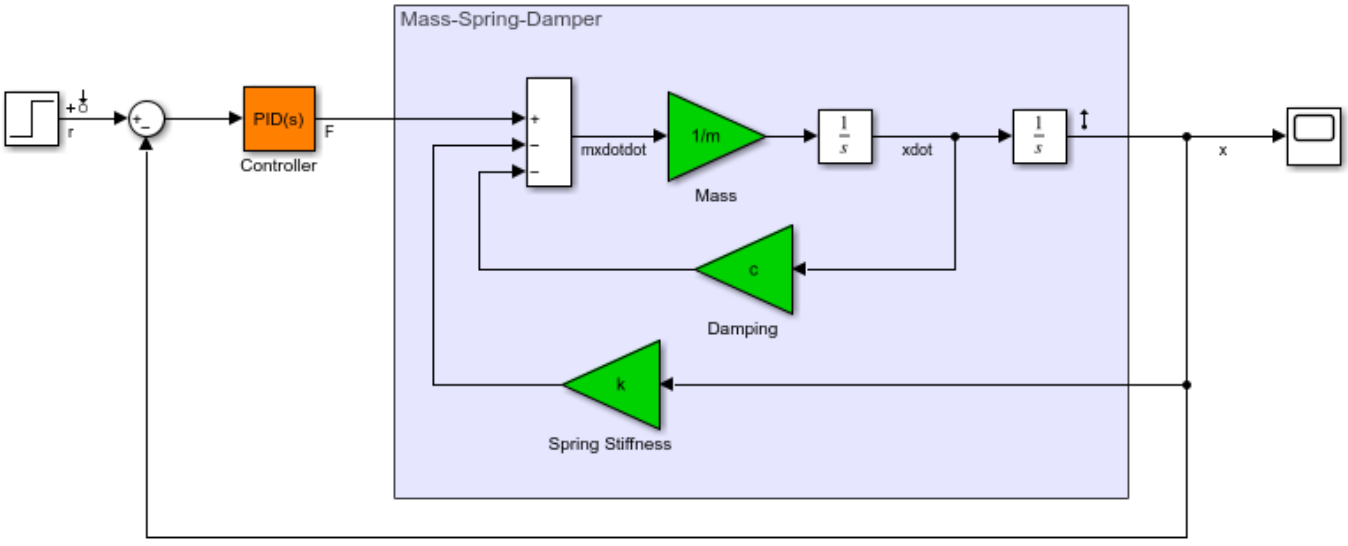


Figure 1: Mass-Spring-Damper System.

This system is modeled in Simulink as follows:

```
open_system('rct_mass_spring_damper')
```



We can use a PID controller to generate the effort F needed to change the position x. Tuning this PID controller is easy when the physical parameters m, c, k are known exactly. However this is rarely the

case in practice, due to a number of factors including imprecise measurements, manufacturing tolerances, changes in operating conditions, and wear and tear. This example shows how to take such uncertainty into account during tuning to maintain high performance within the range of expected values for m, c, k.

Uncertainty Modeling

The Simulink model uses the "most probable" or "nominal" values of m, c, k:

$$m = 3, \quad c = 1, \quad k = 2.$$

Use the "uncertain real" (`ureal`) object to model the range of values that each parameter may take. Here the uncertainty is specified as a percentage deviation from the nominal value.

```
um = ureal('m',3,'Percentage',40);
uc = ureal('c',1,'Percentage',20);
uk = ureal('k',2,'Percentage',30);
```

Nominal Tuning

First tune the PID controller for the nominal parameter values. Here we use two simple design requirements:

- Position x should track a step change with a 1 second response time
- Filter coefficient N in PID controller should not exceed 100.

These requirements are expressed as tuning goals:

```
Req1 = TuningGoal.Tracking('r','x',1);
Req2 = TuningGoal.ControllerPoles('Controller',0,0,100);
```

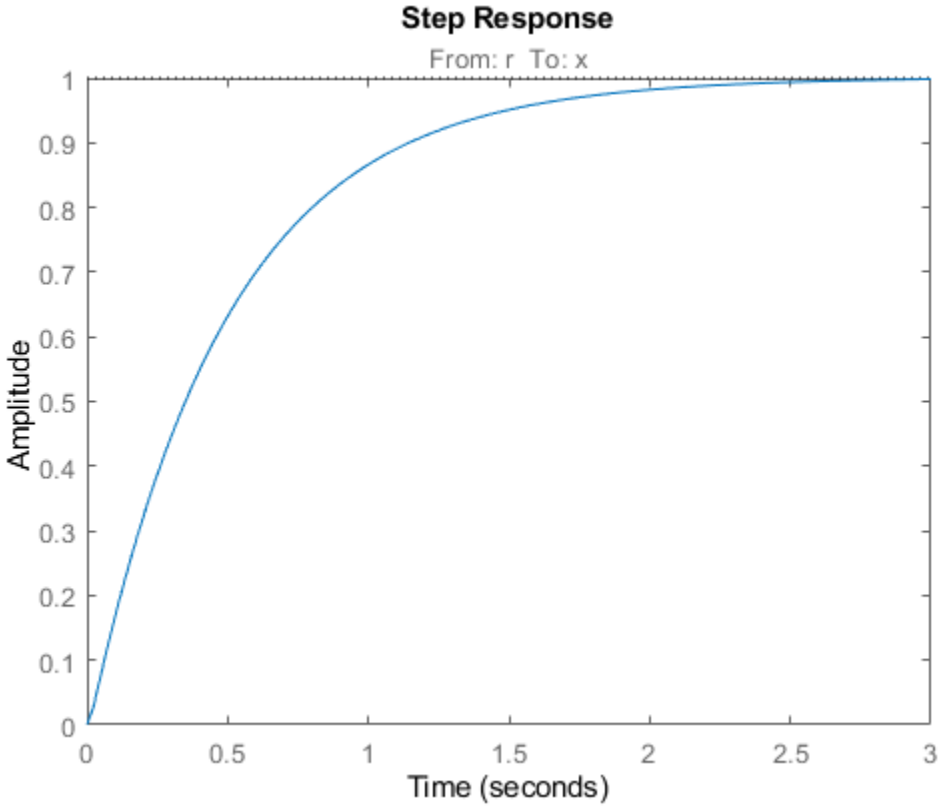
Create an `slTuner` interface for tuning the "Controller" block in the Simulink model, and use `systune` to tune the PID gains and best meet the two requirements.

```
ST0 = slTuner('rct_mass_spring_damper','Controller');

ST = systune(ST0,[Req1 Req2]);
```

```
Final: Soft = 1.02, Hard = -Inf, Iterations = 44
```

Use `getIOTransfer` to view the closed-loop step response.

```
Tnom = getIOTransfer(ST,'r','x');
step(Tnom)
```



The nominal response meets the response time requirement and looks good. But how robust is it to variations of m, c, k?

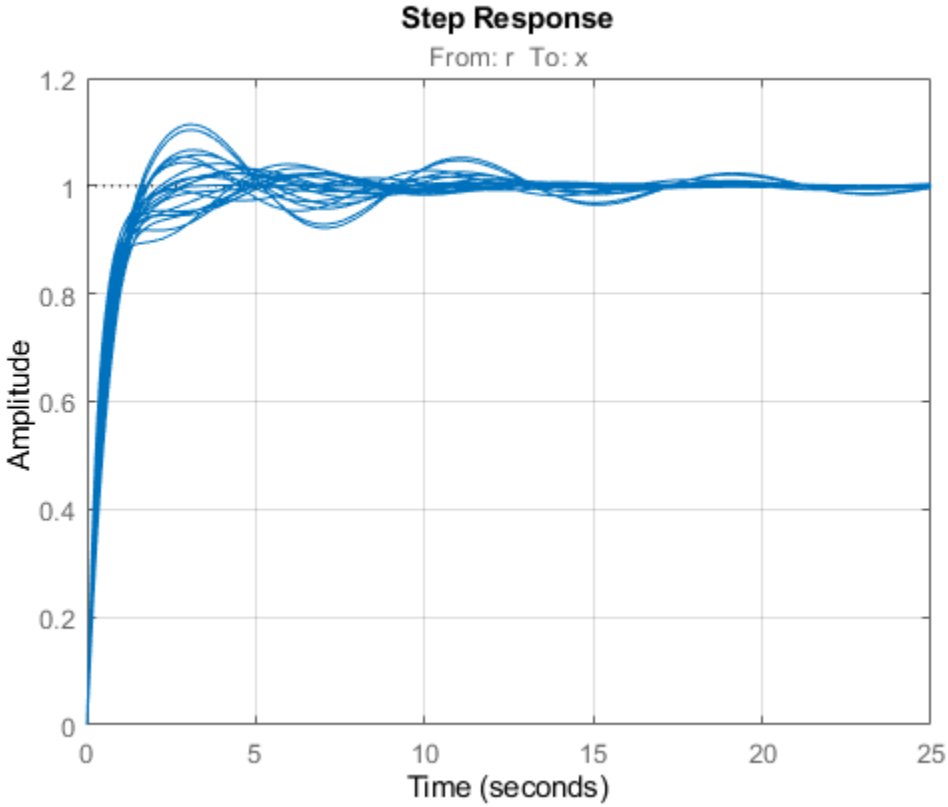
Robustness Analysis

To answer this question, use the "block substitution" feature of `slTuner` to create an uncertain closed-loop model of the mass-spring-damper system. Block substitution lets you specify the linearization of a particular block in a Simulink model. Here we use this to replace the crisp values of m, c, k by the uncertain values `um,uc,uk` defined above.

```
blocksubs(1).Name = 'rct_mass_spring_damper/Mass';
blocksubs(1).Value = 1/um;
blocksubs(2).Name = 'rct_mass_spring_damper/Damping';
blocksubs(2).Value = uc;
blocksubs(3).Name = 'rct_mass_spring_damper/Spring Stiffness';
blocksubs(3).Value = uk;
UST0 = slTuner('rct_mass_spring_damper','Controller',blocksubs);
```

To assess the robustness of the nominal tuning, apply the tuned PID gains to the (untuned) uncertain model `UST0` and simulate the "uncertain" closed-loop response.

```
% Apply result of nominal tuning (ST) to uncertain closed-loop model UST0
setBlockValue(UST0,getBlockValue(ST));
Tnom = getIOTransfer(UST0,'r','x');
rng(0), step(Tnom,25), grid
```



The `step` plot shows the closed-loop response with the nominally tuned PID for 20 randomly selected values of m, c, k in the specified uncertainty range. Observe the significant performance degradation for some parameter combinations, with poorly damped oscillations and a long settling time.

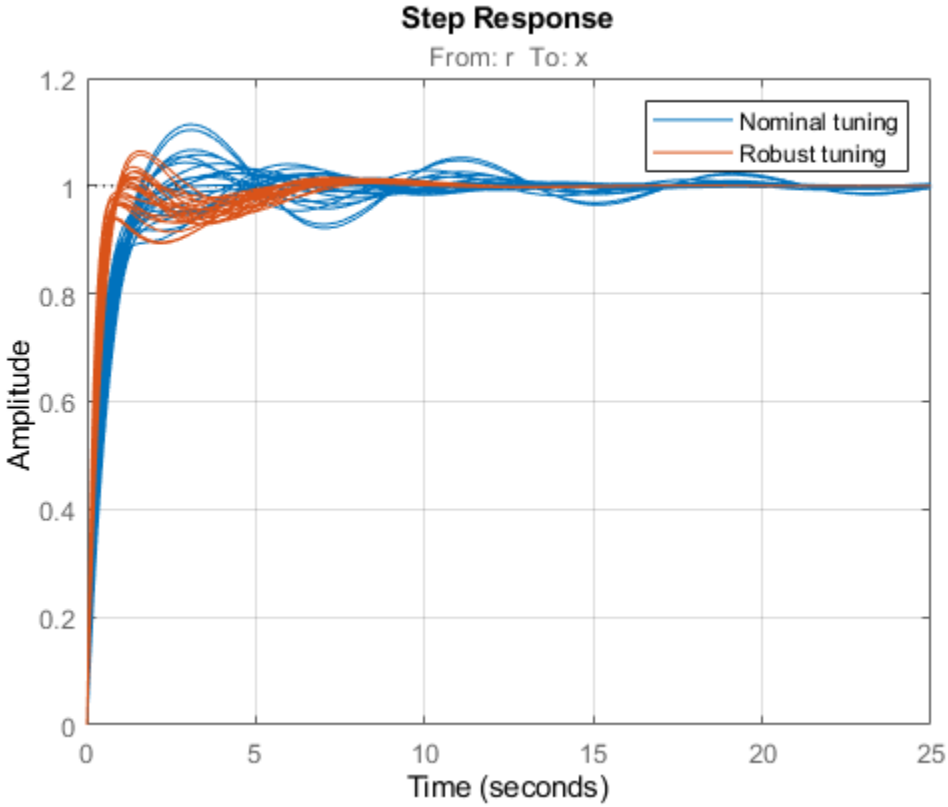
Robust Tuning

To improve the robustness of the PID controller, re-tune it using the uncertain closed-loop model `UST0` rather than the nominal closed-loop model `ST0`. Due to the presence of `ureal` components in the model, `systune` automatically tries to maximize performance over the *entire* uncertainty range. This amounts to minimizing the worst-case value of the "soft" tuning goals `Req1` and `Req2`.

```
UST0 = slTuner('rct_mass_spring_damper','Controller',blocksubs);

UST = systune(UST0,[Req1 Req2]);

Soft: [1.02,4.9], Hard: [-Inf,-Inf], Iterations = 44
Soft: [1.03,1.42], Hard: [-Inf,-Inf], Iterations = 32
Soft: [1.04,1.04], Hard: [-Inf,-Inf], Iterations = 21
Final: Soft = 1.04, Hard = -Inf, Iterations = 97
```

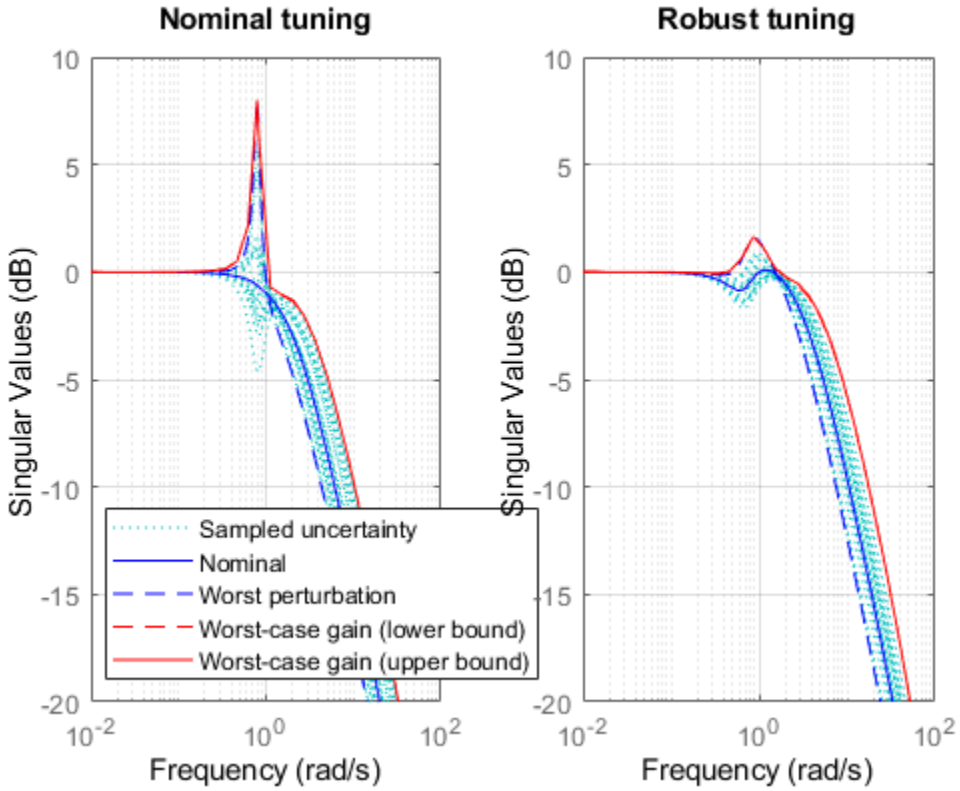
The robust performance is only slightly worse than the nominal performance, but the same uncertain closed-loop simulation shows a significant improvement over the nominal design.

```
Trob = getIOTransfer(UST,'r','x');
rng(0), step(Tnom,Trob,25), grid
legend('Nominal tuning','Robust tuning')
```



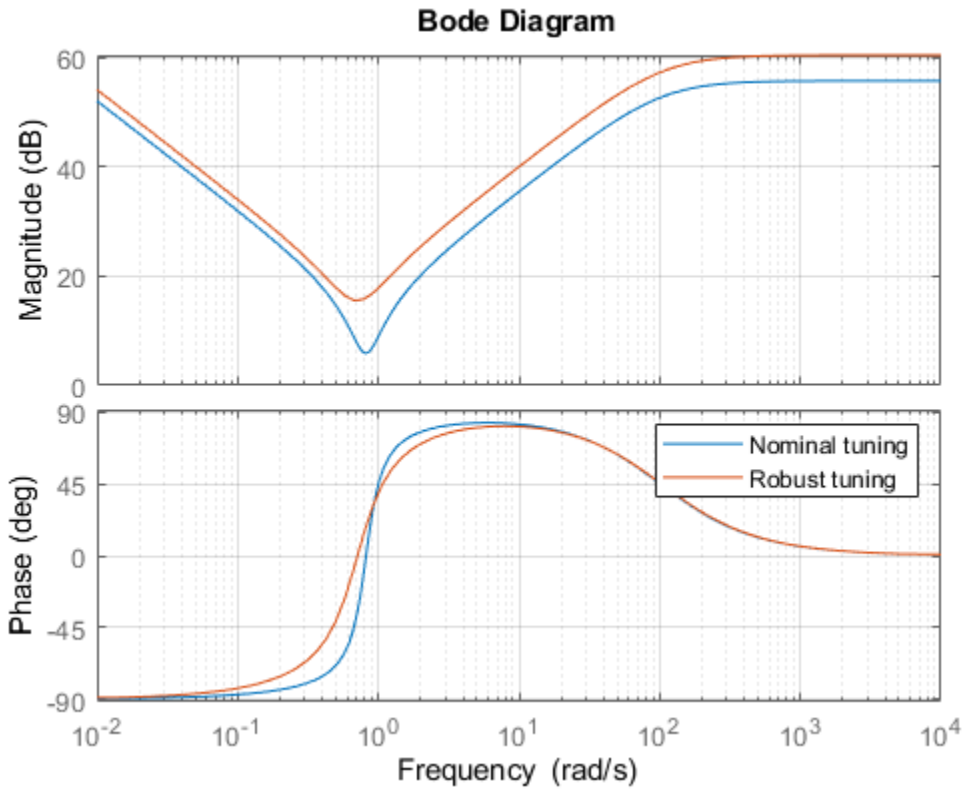
This is confirmed by plotting the worst-case gain from r to x as a function of frequency. Note the attenuated resonance near 1 rad/s.

```
clf
subplot(121), wcsigmaplot(Tnom,{1e-2,1e2}), grid
set(gca,'YLim',[-20 10]), title('Nominal tuning')
subplot(122), wcsigmaplot(Trob,{1e-2,1e2}), grid
set(gca,'YLim',[-20 10]), title('Robust tuning'), legend('off')
```



A comparison of the two PID controllers shows similar behaviors except for one key difference. The nominally tuned PID excessively relies on "cancelling" (notching out) the plant resonance, which is not a robust strategy in the presence of uncertainty on the resonance frequency.

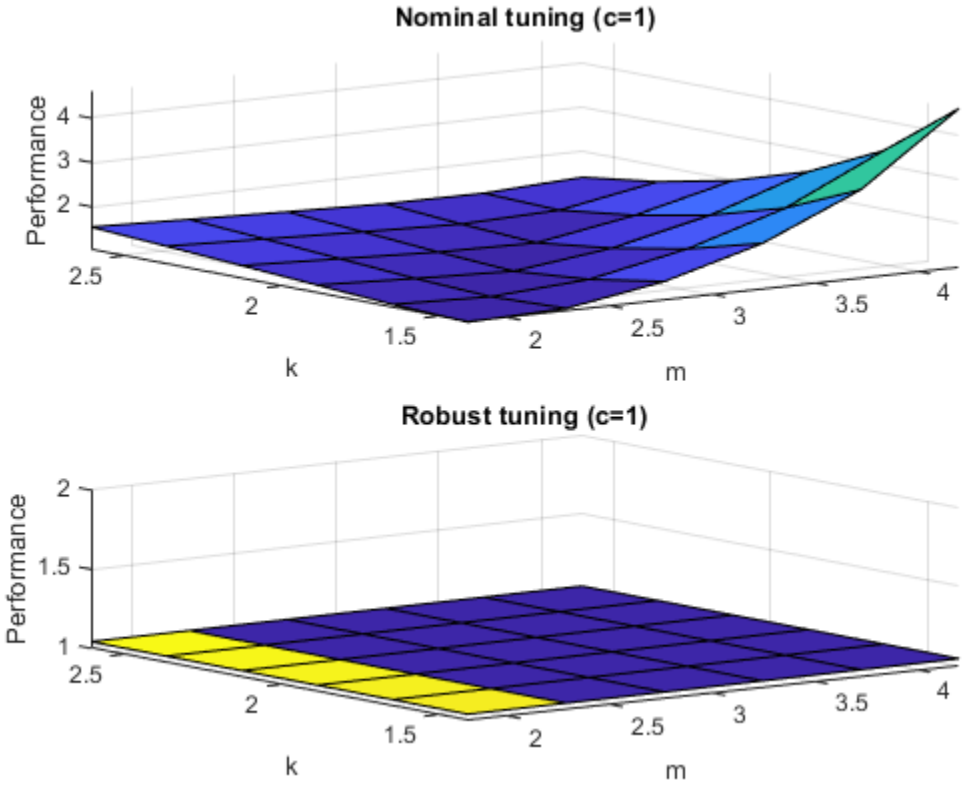
```
Cnom = getBlockValue(ST,'Controller');
Crob = getBlockValue(UST,'Controller');
clf, bode(Cnom,Crob), grid
legend('Nominal tuning','Robust tuning')
```



For further insight, plot the performance index (maximum value of the "soft" tuning goals `Req1,Req2`) as a function of the uncertain parameters m, k for the nominal damping $c = 1$. Use the "varying parameter" feature of `slTuner` to create an array of closed-loop models over a grid of m, k values covering their uncertainty ranges.

```
% Specify a 6-by-6 grid of (m,k) values for linearization
ms = linspace(um.Range(1),um.Range(2),6);
ks = linspace(uk.Range(1),uk.Range(2),6);
[ms,ks] = ndgrid(ms,ks);
params(1).Name = 'm';
params(1).Value = ms;
params(2).Name = 'k';
params(2).Value = ks;
STP = slTuner('rct_mass_spring_damper','Controller',params);

% Evaluate performance index over (m,k) grid for nominally tuned PID
setBlockValue(STP,'Controller',Cnom)
[~,F1] = evalGoal(Req1,STP);
[~,F2] = evalGoal(Req2,STP);
Fnom = max(F1,F2);

% Evaluate performance index over (m,k) grid for robust PID
setBlockValue(STP,'Controller',Crob)
[~,F1] = evalGoal(Req1,STP);
[~,F2] = evalGoal(Req2,STP);
Frob = max(F1,F2);
```

```
% Compare the two performance surfaces
clf
subplot(211), surf(ms,ks,Fnom)
xlabel('m'), ylabel('k'), zlabel('Performance'), title('Nominal tuning (c=1)')
subplot(212), surf(ms,ks,Frob), set(gca,'ZLim',[1 2])
xlabel('m'), ylabel('k'), zlabel('Performance'), title('Robust tuning (c=1)')
```



This plot shows that the nominal tuning is very sensitive to changes in mass m or spring stiffness k, while the robust tuning is essentially insensitive to these parameters. To complete the design, use `writeBlockValue` to apply the robust PID gains to the Simulink model and proceed with further validation in Simulink.

```
writeBlockValue(UST)
```

See Also

Related Examples

- “Model Uncertainty in Simulink for Robust Tuning” on page 6-17

More About

- “Interpreting Results of Robust Tuning” on page 6-11

Robust Tuning of DC Motor Controller

This example shows how to robustly tune a PID controller for a DC motor with imperfectly known parameters.

DC Motor Modeling

An uncertain model of the DC motor is derived in the "Robustness of Servo Controller for DC Motor" example. The transfer function from applied voltage to angular velocity is given by

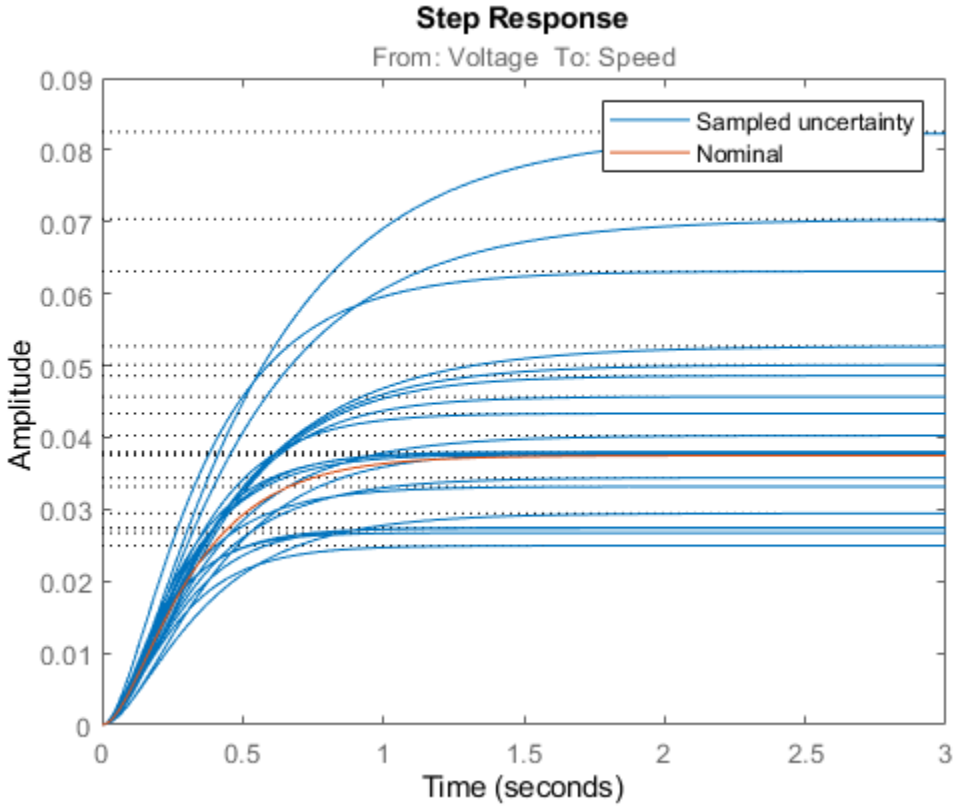
$$P(s) = \frac{K_m}{JLs^2 + (JR + LK_f)s + K_mK_b + RK_f}$$

where the resistance R, the inductance L, the EMF constant K_b, armature constant K_m, viscous friction K_f, and inertial load J are physical parameters of the motor. These parameters are not perfectly known and are subject to variation, so we model them as uncertain values with a specified range or percent uncertainty.

```
R = ureal('R',2,'Percentage',40);
L = ureal('L',0.5,'Percentage',40);
K = ureal('K',0.015,'Range',[0.012 0.019]);
Km = K; Kb = K;
Kf = ureal('Kf',0.2,'Percentage',50);
J = ureal('J',0.02,'Percentage',20);

P = tf(Km,[J*L J*R+Kf*L Km*Kb+Kf*R]);
P.InputName = 'Voltage';
P.OutputName = 'Speed';
```

Time and frequency response functions like `step` or `bode` automatically sample the uncertain parameters within their range. This is helpful to gauge the impact of uncertainty. For example, plot the step response of the uncertain plant `P` and note the large variation in plant DC gain.

```
step(P,getNominal(P),3)
legend('Sampled uncertainty','Nominal')
```



Robust PID Tuning

To robustly tune a PID controller for this DC motor, create a tunable PID block `C` and construct a closed-loop model `CL0` of the feedback loop in Figure 1. Add an analysis point `dLoad` at the plant output to measure the sensitivity to load disturbance.

```
C = tunablePID('C','pid');
AP = AnalysisPoint('dLoad');
CL0 = feedback(AP*P*C,1);
CL0.InputName = 'SpeedRef';
CL0.OutputName = 'Speed';
```

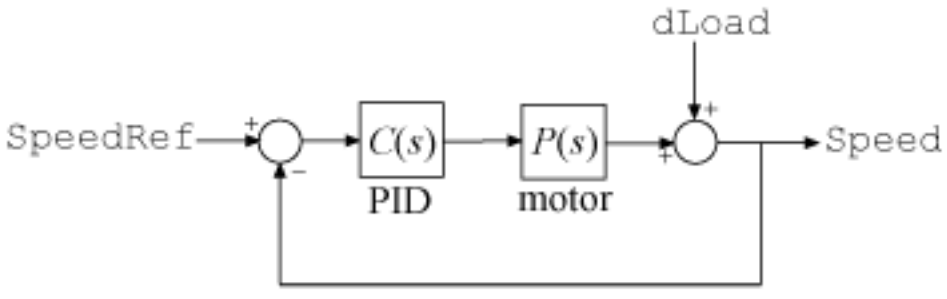


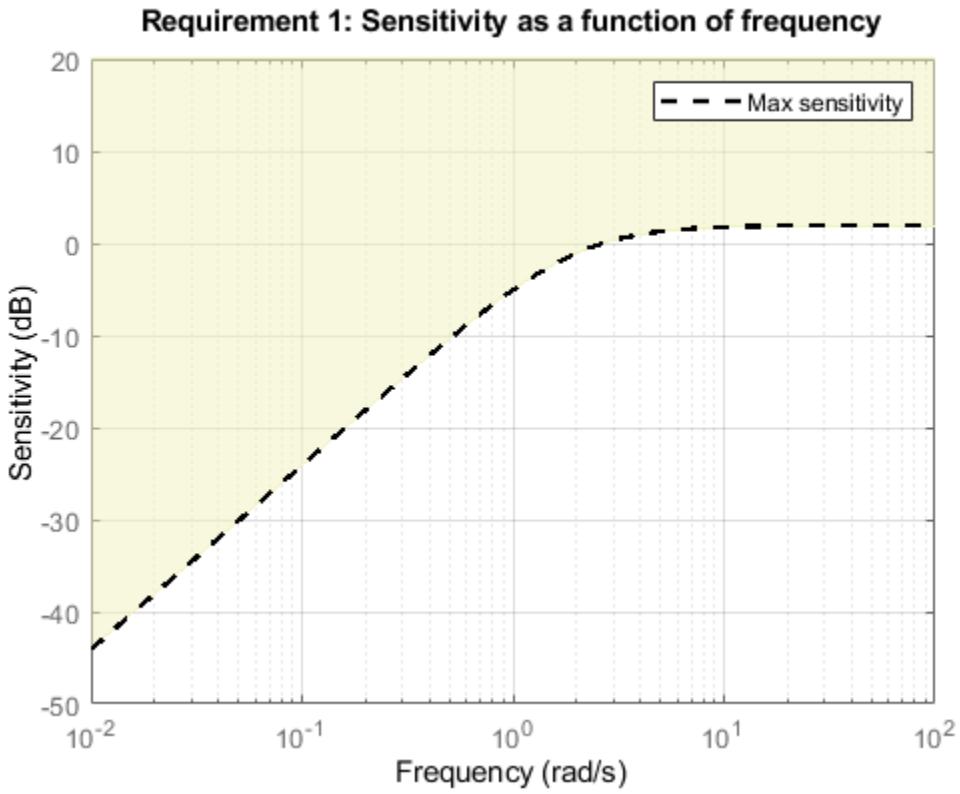
Figure 1: PID control of DC motor

There are many ways to specify the desired performance. Here we focus on sensitivity to load disturbance, roll-off, and closed-loop dynamics.

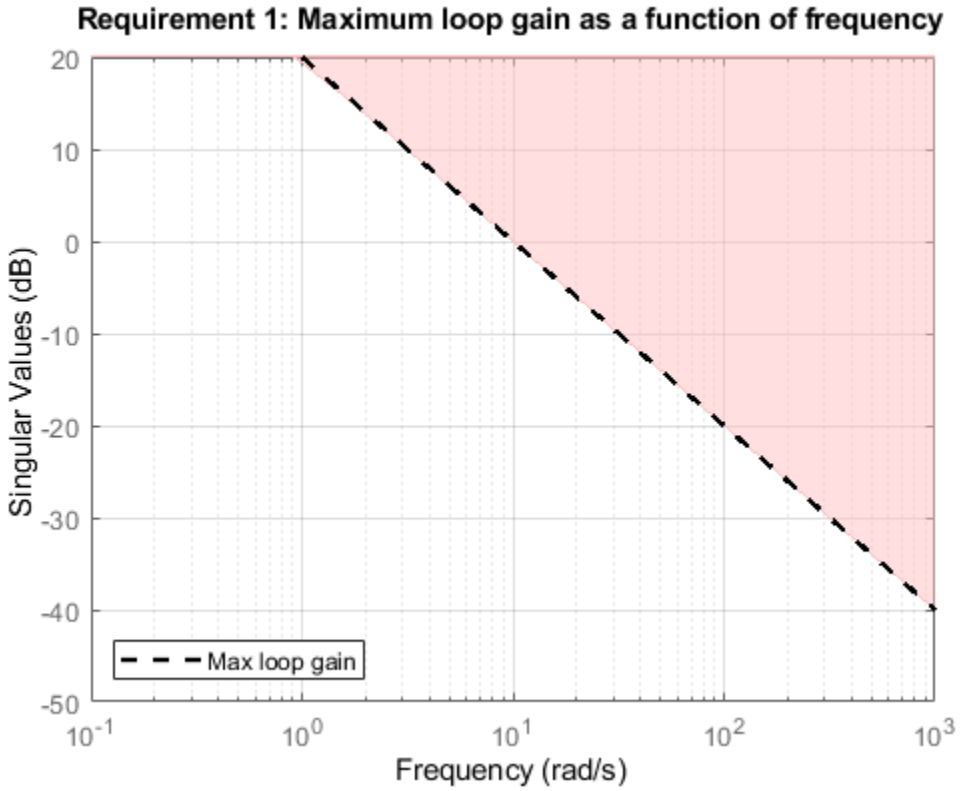
```
R1 = TuningGoal.Sensitivity('dLoad',tf([1.25 0],[1 2]));
R2 = TuningGoal.MaxLoopGain('dLoad',10,1);
R3 = TuningGoal.Poles('dLoad',0.1,0.7,25);
```

The first goal R1 specifies the desired profile for the sensitivity function. Sensitivity should be low at low frequency for good disturbance rejection. The second goal R2 imposes -20 dB/decade roll-off past 10 rad/s. The third goal R3 specifies the minimum decay, minimum damping, and maximum natural frequency for the closed-loop poles.

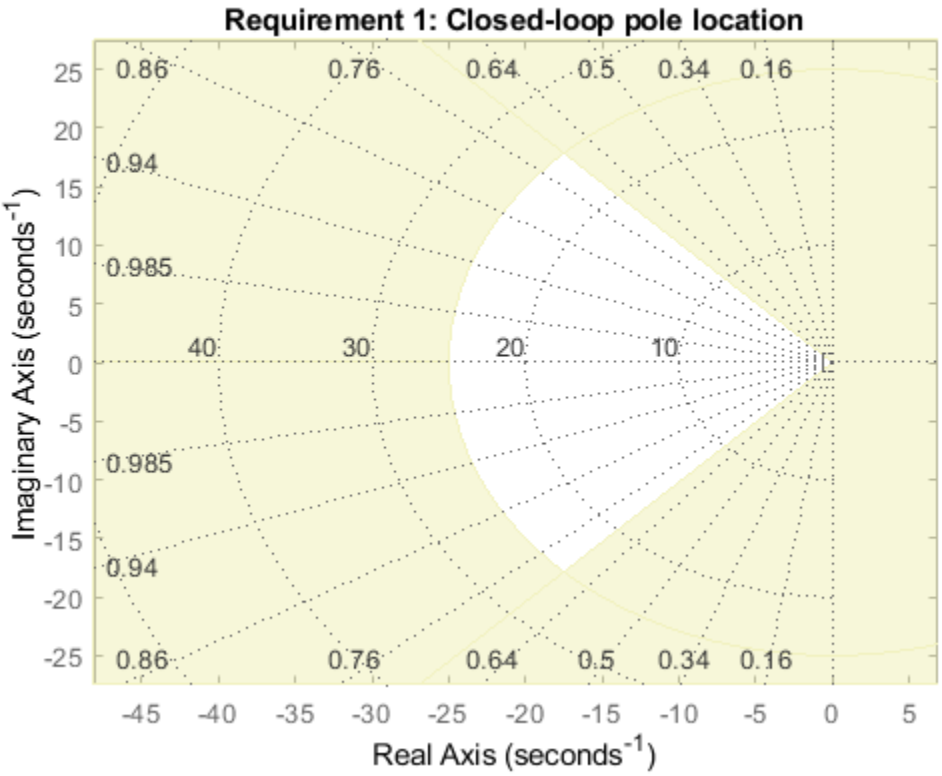
```
viewGoal(R1)
```




```
viewGoal(R2)
```


Requirement 1: Maximum loop gain as a function of frequency
Singular Values (dB)
Frequency (rad/s)
Max loop gain
20
10
0
-10
-20
-30
-40
-50
10^-1
10^0
10^1
10^2
10^3

```
viewGoal(R3)
```



You can now use `systune` to robustly tune the PID gains, that is, to try and meet the design objectives for **all** possible values of the uncertain DC motor parameters. Because local minima may exist, perform three separate tunings from three different sets of initial gain values.

```
opt = systuneOptions('RandomStart',2);
rng(0), [CL,fSoft] = systune(CL0,[R1 R2 R3],opt);
```

```
Nominal tuning:
Design 1: Soft = 0.838, Hard = -Inf
Design 2: Soft = 0.838, Hard = -Inf
Design 3: Soft = 0.914, Hard = -Inf

Robust tuning of Design 2:
Soft: [0.838,2.01], Hard: [-Inf,-Inf], Iterations = 40
Soft: [0.875,1.76], Hard: [-Inf,-Inf], Iterations = 29
Soft: [0.935,2.77], Hard: [-Inf,-Inf], Iterations = 27
Soft: [1.35,1.35], Hard: [-Inf,-Inf], Iterations = 35
Final: Soft = 1.35, Hard = -Inf, Iterations = 131

Robust tuning of Design 1:
Soft: [0.838,1.96], Hard: [-Inf,-Inf], Iterations = 65
Soft: [0.875,1.76], Hard: [-Inf,-Inf], Iterations = 28
Soft: [1.02,2.98], Hard: [-Inf,-Inf], Iterations = 34
Soft: [1.34,1.36], Hard: [-Inf,-Inf], Iterations = 32
Soft: [1.35,1.35], Hard: [-Inf,-Inf], Iterations = 20
Final: Soft = 1.35, Hard = -Inf, Iterations = 179

Robust tuning of Design 3:
```

```
Soft: [0.914,2.38], Hard: [-Inf,-Inf], Iterations = 57
Soft: [0.875,1.76], Hard: [-Inf,-Inf], Iterations = 82
Soft: [1.02,2.98], Hard: [-Inf,-Inf], Iterations = 32
Soft: [1.34,1.36], Hard: [-Inf,-Inf], Iterations = 32
Soft: [1.35,1.35], Hard: [-Inf,-Inf], Iterations = 20
Final: Soft = 1.35, Hard = -Inf, Iterations = 223
```

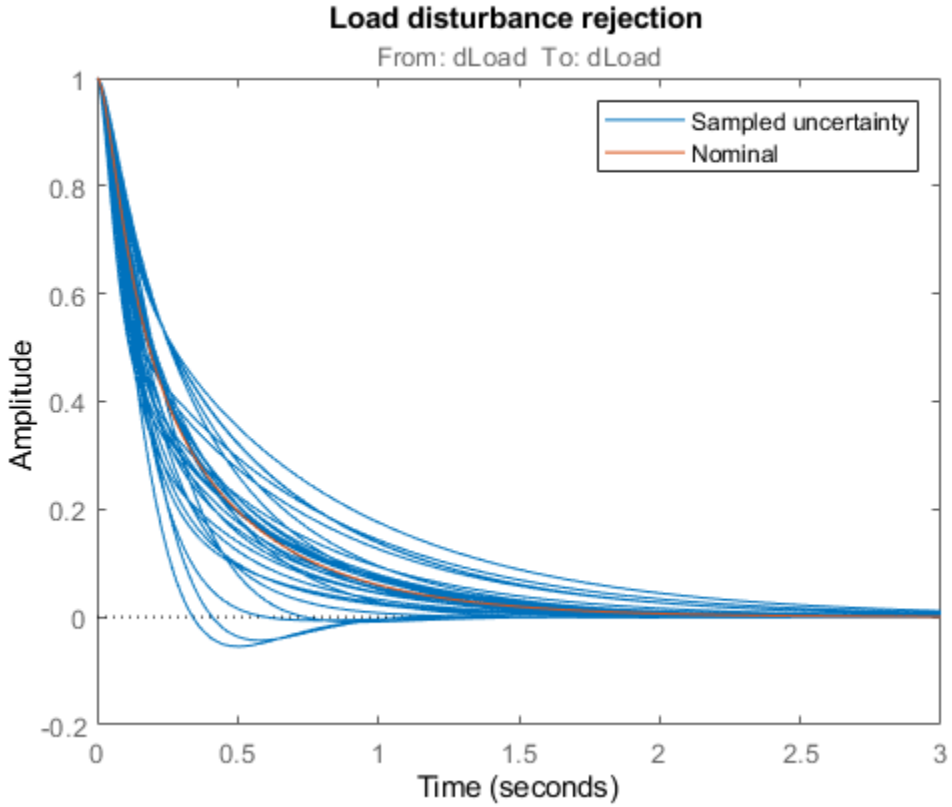
The final value is close to 1 so the tuning goals are nearly achieved throughout the uncertainty range. The tuned PID controller is

```
showTunable(CL)
```

```
C =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 33.8, Ki = 83.2, Kd = 2.34, Tf = 0.028

Name: C
Continuous-time PIDF controller in parallel form.
```

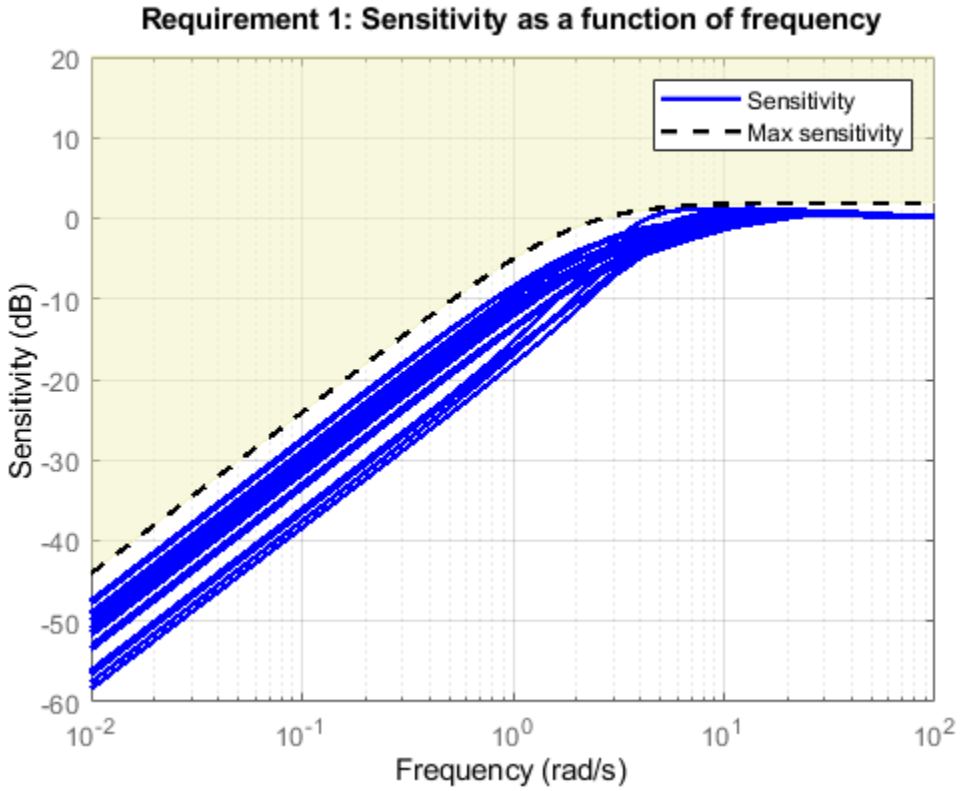
Next check how this PID rejects a step load disturbance for 30 randomly selected values of the uncertain parameters.

```
S = getSensitivity(CL,'dLoad');
clf, step(usample(S,30),getNominal(S),3)
title('Load disturbance rejection')
legend('Sampled uncertainty','Nominal')
```



The rejection performance remains uniform despite large plant variations. You can also verify that the sensitivity function robustly stays within the prescribed bound.

```
viewGoal(R1,CL)
```



Robust tuning with `systune` is easy. Just include plant uncertainty in the tunable closed-loop model using `ureal` objects, and the software automatically tries to achieve the tuning goals for the entire uncertainty range.

See Also

Related Examples

- “Build Tunable Control System Model With Uncertain Parameters” on page 6-13
- “Robust Tuning of Positioning System” on page 6-40
- “Robust Tuning of Mass-Spring-Damper System” on page 6-24

More About

- “Interpreting Results of Robust Tuning” on page 6-11

Robust Tuning of Positioning System

This example shows how to take into account model uncertainty when tuning a motion control system.

Background

This example refines the design discussed in the "Tuning of a Digital Motion Control System" example. The positioning system under consideration is shown below.

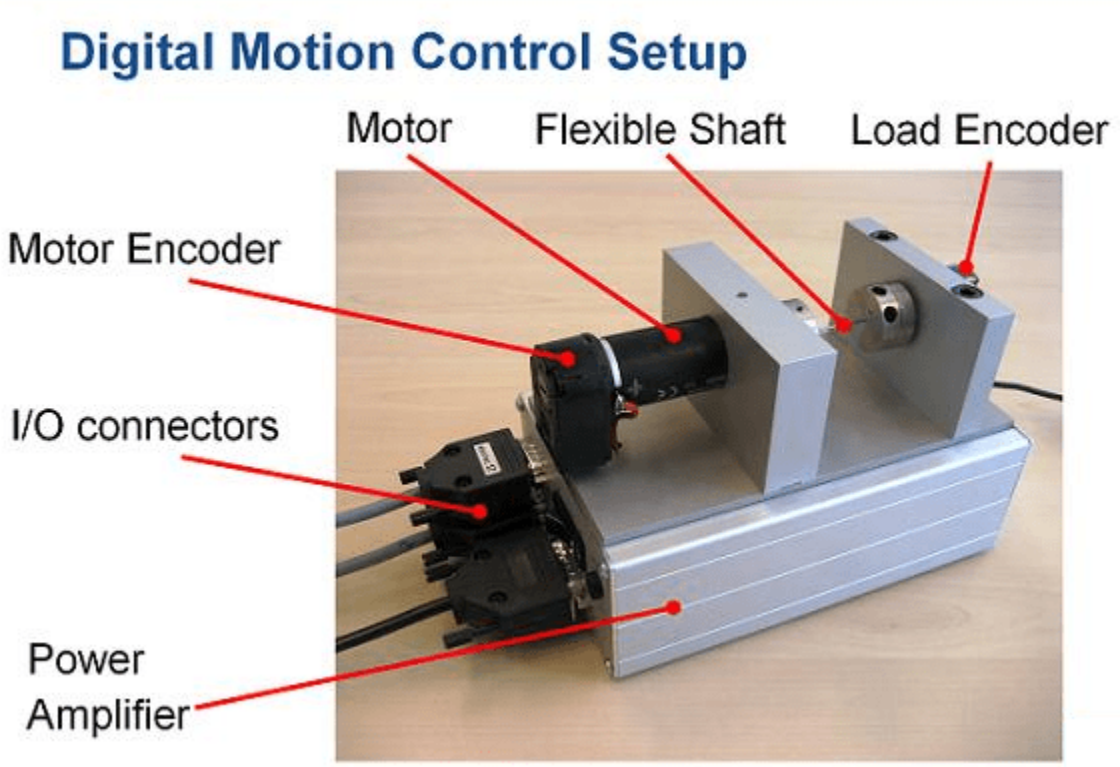
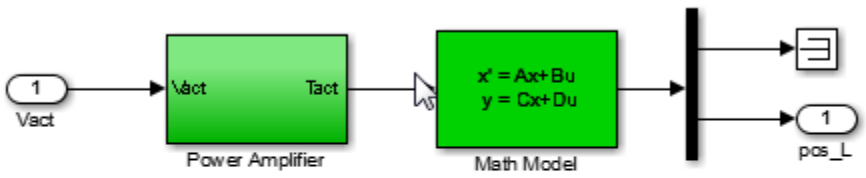


Figure 1: Digital motion control hardware

A physical model of the plant is shown in the "Plant Model" block of the Simulink model `rct_dmcNotch`:



$$J_1 x_1'' = - b_1 x_1' - k (x_1-x_2) - b_{12} (x_1'-x_2') + T$$
$$J_2 x_2'' = - b_2 x_2' + k (x_1-x_2) + b_{12} (x_1'-x_2')$$

Figure 2: Equations of motion

In the earlier example, we tuned the controller using "crisp" values for the physical parameters $J_1, J_2, b_1, b_2, b_{12}, k$. In reality, these parameter values are only known approximately and may vary over time. Because the resulting model discrepancies can adversely affect controller performance, we need to account for parameter uncertainty during tuning to ensure robust performance over the range of possible parameter values. This process is called *robust tuning*.

Modeling Uncertainty

Assume 25% uncertainty on the value of the stiffness k, and 50% uncertainty on the values of the damping coefficients b_1, b_2, b_{12}. Use the `ureal` object to model these uncertainty ranges.

```
b1 = ureal('b1',1e-6,'Percent',50);
b2 = ureal('b2',1e-6,'Percent',50);
b12 = ureal('b12',5e-7,'Percent',50);
k = ureal('k',0.013,'Percent',25);
```

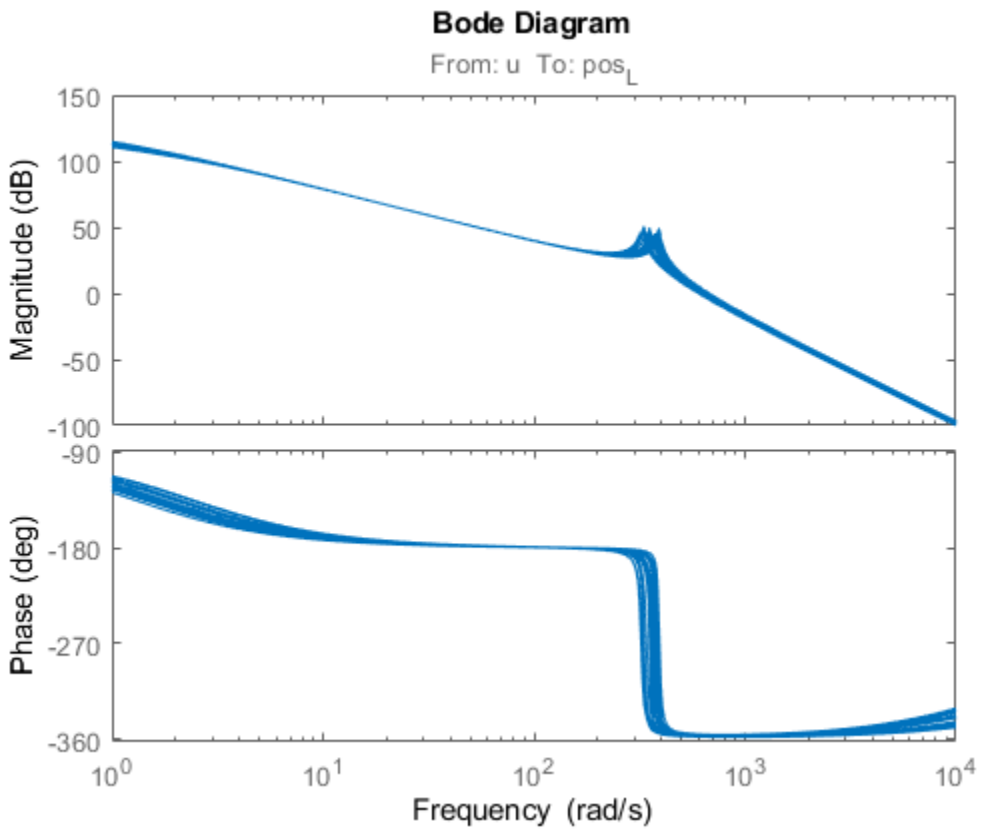
Using the equations of motion in Figure 2, we can derive a state-space model `G` of the plant expressed in terms of $J_1, J_2, b_1, b_2, b_{12}, k$:

```
J1 = 1e-6; J2 = 1.15e-7;
A = [0 1 0 0; -k/J1 -(b1+b12)/J1 k/J1 b12/J1; 0 0 0 1; k/J2 b12/J2 -k/J2 -(b2+b12)/J2 ];
B = [ 0; 1/J1 ; 0    ; 0 ];
C = [ 0  0  1  0 ];
D  = 0;
G = ss(A,B,C,D,'InputName','u','OutputName','pos_L')
```

```
G =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 4 states.
  The model uncertainty consists of the following blocks:
    b1: Uncertain real, nominal = 1e-06, variability = [-50,50]%, 1 occurrences
    b12: Uncertain real, nominal = 5e-07, variability = [-50,50]%, 1 occurrences
    b2: Uncertain real, nominal = 1e-06, variability = [-50,50]%, 1 occurrences
    k: Uncertain real, nominal = 0.013, variability = [-25,25]%, 1 occurrences

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

Note that the resulting model `G` depends on the uncertain parameters k, b_1, b_2, b_{12}. To assess how uncertainty impacts the plant, plot its Bode response for different values of (b_1, b_2, b_{12}, k). By default, the `bode` function uses 20 randomly selected values in the uncertainty range. Note that both the damping and natural frequency of the main resonance are affected.

```
rng(0), bode(G,{1e0,1e4})
```



Nominal Tuning

To compare nominal and robust tuning, we first repeat the nominal design done in the "Tuning of a Digital Motion Control System" example. The controller consists of a lead-lag compensator and a notch filter:

```
% Tunable lead-lag
LL = tunableTF('LL',1,1);

% Tunable notch (s^2+2*zeta1*wn*s+wn^2)/(s^2+2*zeta2*wn*s+wn^2)
wn = realp('wn',300);   wn.Minimum = 300;
zeta1 = realp('zeta1',1);   zeta1.Minimum = 0;   zeta1.Maximum = 1;
zeta2 = realp('zeta2',1);   zeta2.Minimum = 0;   zeta2.Maximum = 1;
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]);

% Overall controller
C = N * LL;
```

Use `feedback` to build a closed-loop model `T0` that includes both the tunable and uncertain elements.

```
AP = AnalysisPoint('u',1);  % to access control signal u
T0 = feedback(G*AP*C,1);
T0.InputName = 'ref'
```

```
T0 =
```

```
  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 7 states, and the follo
    LL: Tunable SISO transfer function, 1 zeros, 1 poles, 1 occurrences.
    b1: Uncertain real, nominal = 1e-06, variability = [-50,50]%, 1 occurrences
    b12: Uncertain real, nominal = 5e-07, variability = [-50,50]%, 1 occurrences
    b2: Uncertain real, nominal = 1e-06, variability = [-50,50]%, 1 occurrences
    k: Uncertain real, nominal = 0.013, variability = [-25,25]%, 1 occurrences
    u: Analysis point, 1 channels, 1 occurrences.
    wn: Scalar parameter, 6 occurrences.
    zeta1: Scalar parameter, 1 occurrences.
    zeta2: Scalar parameter, 1 occurrences.

Type "ss(T0)" to see the current value, "get(T0)" to see all properties, and "T0.Blocks" to inte
```

The main tuning goals are:

- Open-loop bandwidth of 50 rad/s
- Gain and phase stability margins of at least 7.6 dB and 45 degrees

To prevent fast dynamics, we further limit the natural frequency of closed-loop poles.

```
s = tf('s');
R1 = TuningGoal.LoopShape('u',50/s);
R2 = TuningGoal.Margins('u',7.6,45);
R3 = TuningGoal.Poles('u',0,0,1e3);   % natural frequency < 1000
```

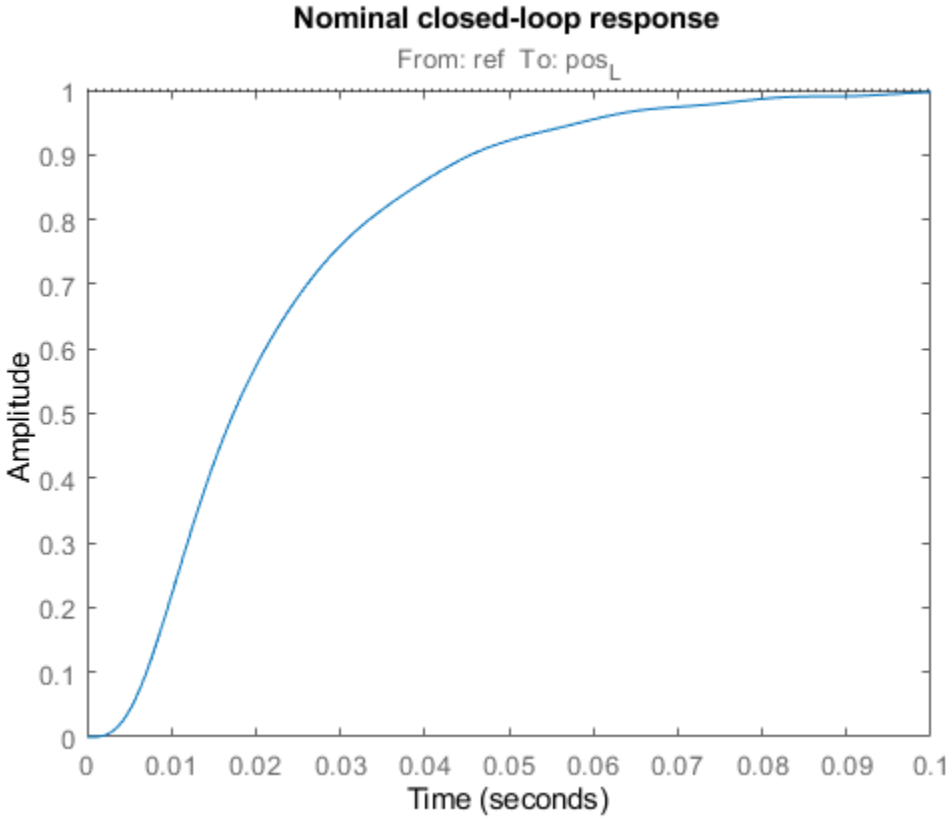
Now tune the controller parameters for the nominal plant subject to the three tuning goals.

```
T = systune(getNominal(T0),[R1 R2 R3]);
```

```
Final: Soft = 0.906, Hard = -Inf, Iterations = 118
```

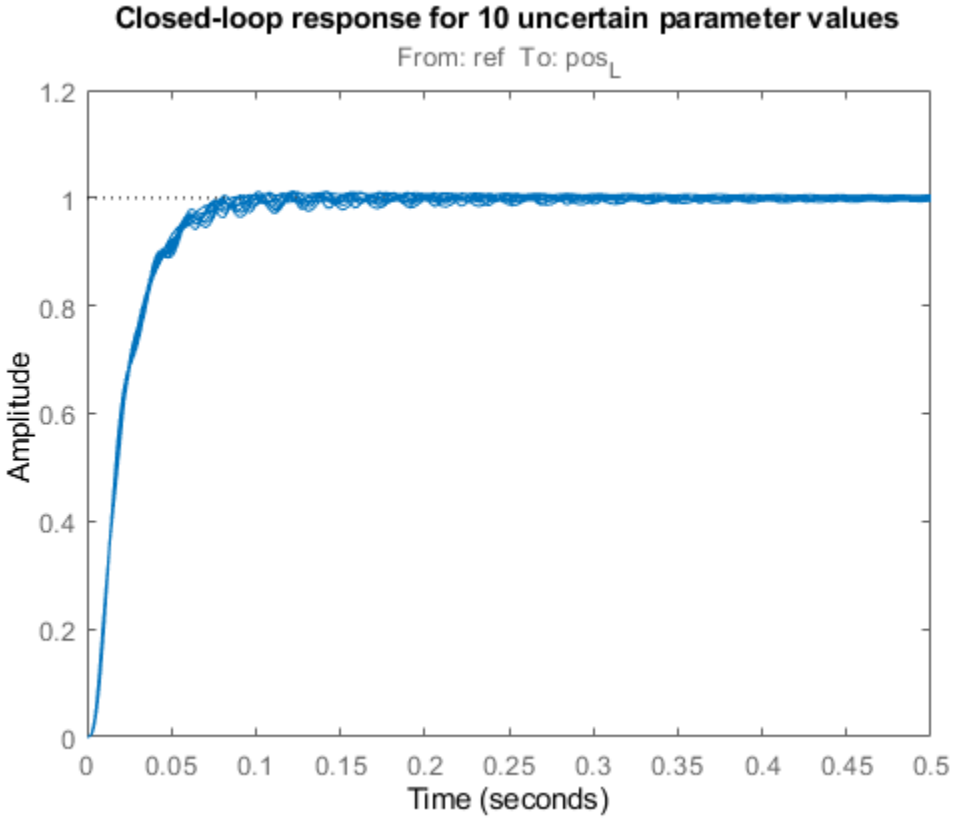
The final value indicates that all design objectives were nominally met and the closed-loop response looks good.

```
step(T), title('Nominal closed-loop response')
```



How robust is this design? To find out, update the uncertain closed-loop model `T0` with the nominally tuned controller parameters and plot the closed-loop step response for 10 random samples of the uncertain parameters.

```
Tnom = setBlockValue(T0,T);        % update T0 with tuned valued from systune
[Tnom10,S10] = usample(Tnom,10);  % sample the uncertainty
step(Tnom10,0.5)
title('Closed-loop response for 10 uncertain parameter values')
```



This plot reveals significant oscillations when moving away from the nominal values of b_1, b_2, b_{12}, k.

Robust Tuning

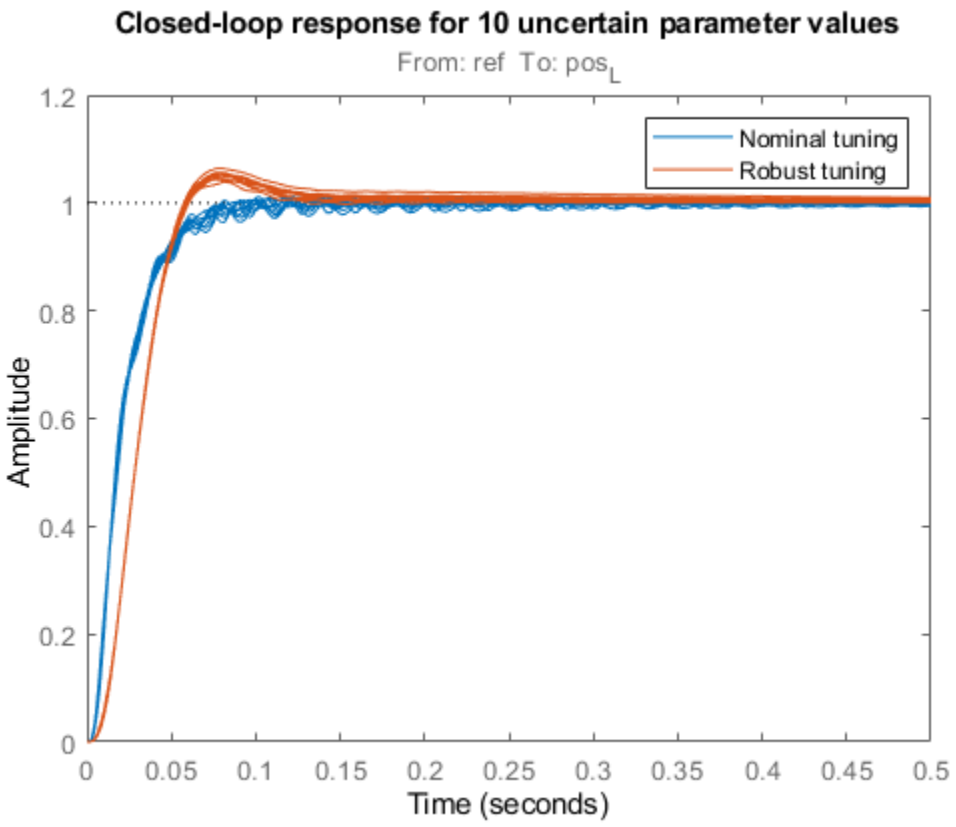
Next re-tune the controller using the uncertain closed-loop model `T0` instead of its nominal value. This instructs `systune` to enforce the tuning goals over the entire uncertainty range.

```
[Trob,fSoft,~,Info] = systune(T0,[R1 R2 R3]);
```

```
Soft: [0.906,102], Hard: [-Inf,-Inf], Iterations = 118
Soft: [1.02,3.72], Hard: [-Inf,-Inf], Iterations = 49
Soft: [1.25,1.85], Hard: [-Inf,-Inf], Iterations = 40
Soft: [1.26,1.26], Hard: [-Inf,-Inf], Iterations = 30
Final: Soft = 1.26, Hard = -Inf, Iterations = 237
```

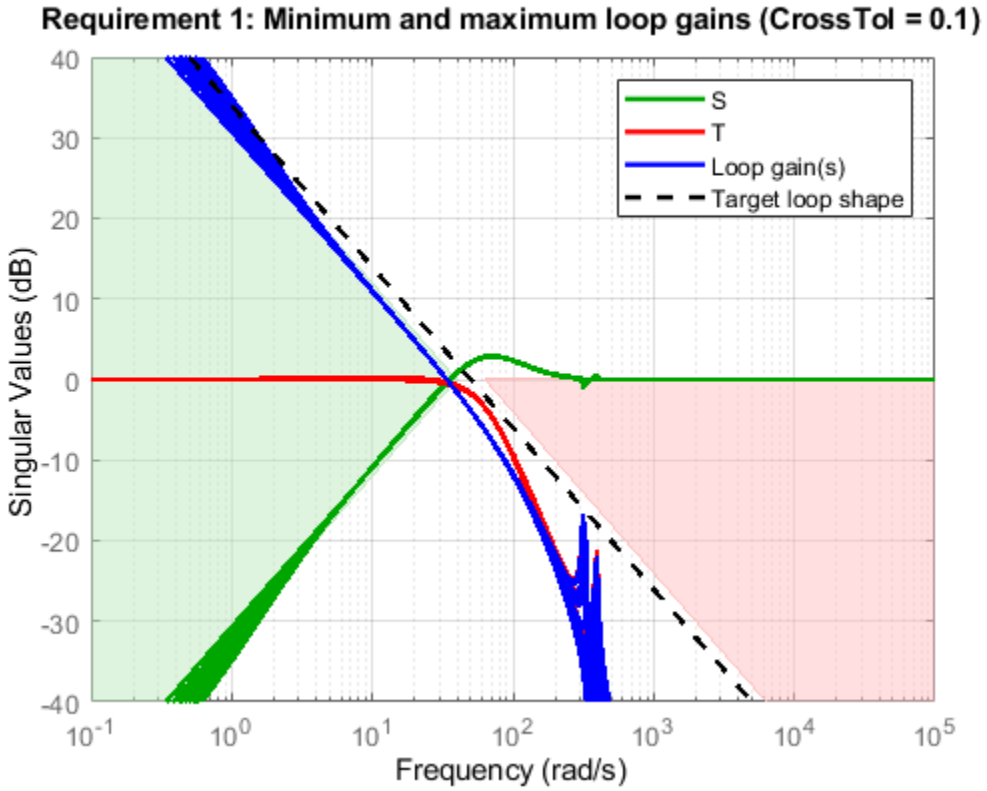
The achieved performance is a bit worse than for nominal tuning, which is expected given the additional robustness constraint. Compare performance with the nominal design.

```
Trob10 = usubs(Trob,S10); % use the same 10 uncertainty samples
step(Tnom10,Trob10,0.5)
title('Closed-loop response for 10 uncertain parameter values')
legend('Nominal tuning','Robust tuning')
```



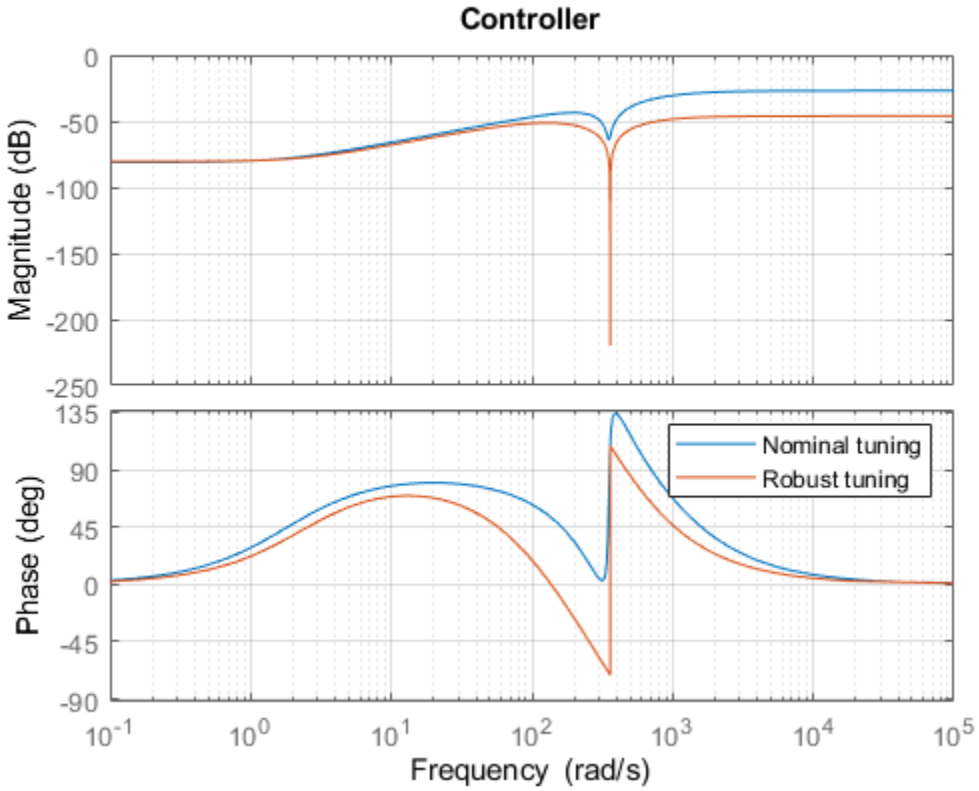
The robust design has more overshoot but is largely free of oscillations. Verify that the plant resonance is robustly attenuated.

```
viewGoal(R1,Trob)
```



Finally, compare the nominal and robust controllers.

```
Cnom = setBlockValue(C,Tnom);
Crob = setBlockValue(C,Trob);
bode(Cnom,Crob), grid, title('Controller')
legend('Nominal tuning','Robust tuning')
```



Not surprisingly, the robust controller uses a wider and deeper notch to accommodate the damping and natural frequency variations in the plant resonance. Using `systune`'s robust tuning capability, you can automatically position and calibrate the notch to best compensate for such variability.

Worst-Case Analysis

The fourth output argument of `systune` contains information about worst-case combinations of uncertain parameters. These combinations are listed in decreasing order of severity.

```
WCU = Info.wcPert
```

```
WCU =

  5x1 struct array with fields:

    b1
    b12
    b2
    k
```

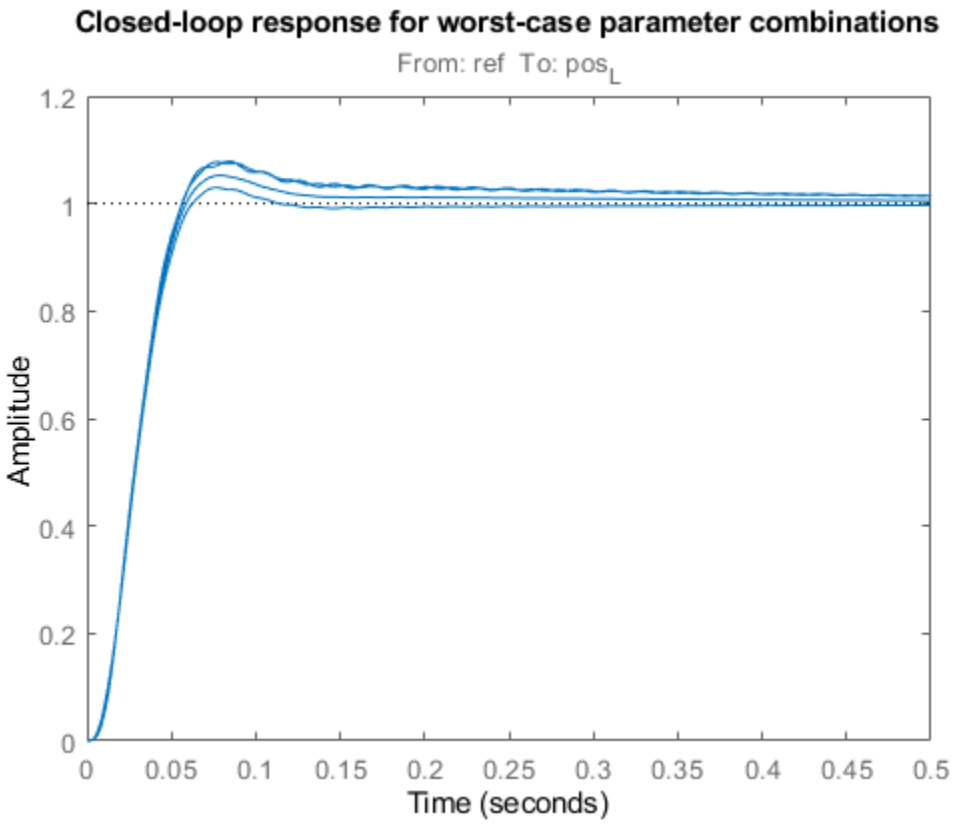
```
WCU(1)  % worst-overall combination
```

```
ans =

  struct with fields:
```

```
     b1: 5.0000e-07
    b12: 7.5000e-07
     b2: 5.0000e-07
      k: 0.0163
```

To analyze the worst-case responses, substitute these parameter values in the closed-loop model `Trob`.

```
Twc = usubs(Trob,WCU);
step(Twc,0.5)
title('Closed-loop response for worst-case parameter combinations')
```



See Also

Related Examples

- “Build Tunable Control System Model With Uncertain Parameters” on page 6-13
- “Robust Vibration Control in Flexible Beam” on page 6-50
- “Robust Tuning of Mass-Spring-Damper System” on page 6-24

More About

- “Interpreting Results of Robust Tuning” on page 6-11

Robust Vibration Control in Flexible Beam

This example shows how to robustly tune a controller for reducing vibrations in a flexible beam. This example is adapted from "Control System Design" by G. Goodwin, S. Graebe, and M. Salgado.

Uncertain Model of Flexible Beam

Figure 1 depicts an active vibration control system for a flexible beam.

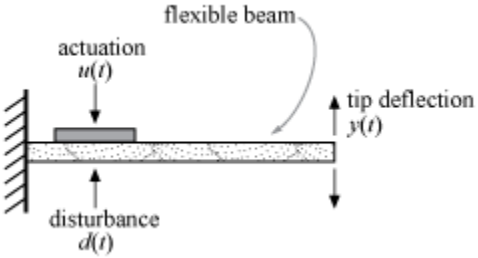


Figure 1: Active control of flexible beam

In this setup, a sensor measures the tip position $y(t)$ and the actuator is a piezoelectric patch delivering a force $u(t)$. We can model the transfer function from control input u to tip position y using finite-element analysis. Keeping only the first six modes, we obtain a plant model of the form

$$G(s) = \sum_{i=1}^{6} \frac{\alpha_i}{s^2 + 2\zeta_i\omega_i s + \omega_i^2}$$

with the following nominal values for the amplitudes α_i and natural frequencies ω_i:

$$\alpha = 9.72 \times 10^{-4}, 0.0122, 0.0012, -0.0583, -0.0013, 0.1199$$

$$\omega = 18.95, 118.76, 332.54, 651.66, 1077.2, 1609.2.$$

The damping factors ζ_i are often poorly known and are assumed to range between 0.0002 and 0.02. Similarly, the natural frequencies are only approximately known and we assume 20% uncertainty on their location. To construct an uncertain model of the flexible beam, use the `ureal` object to specify the uncertainty range for the damping and natural frequencies. To simplify, we assume that all modes have the same damping factor ζ.

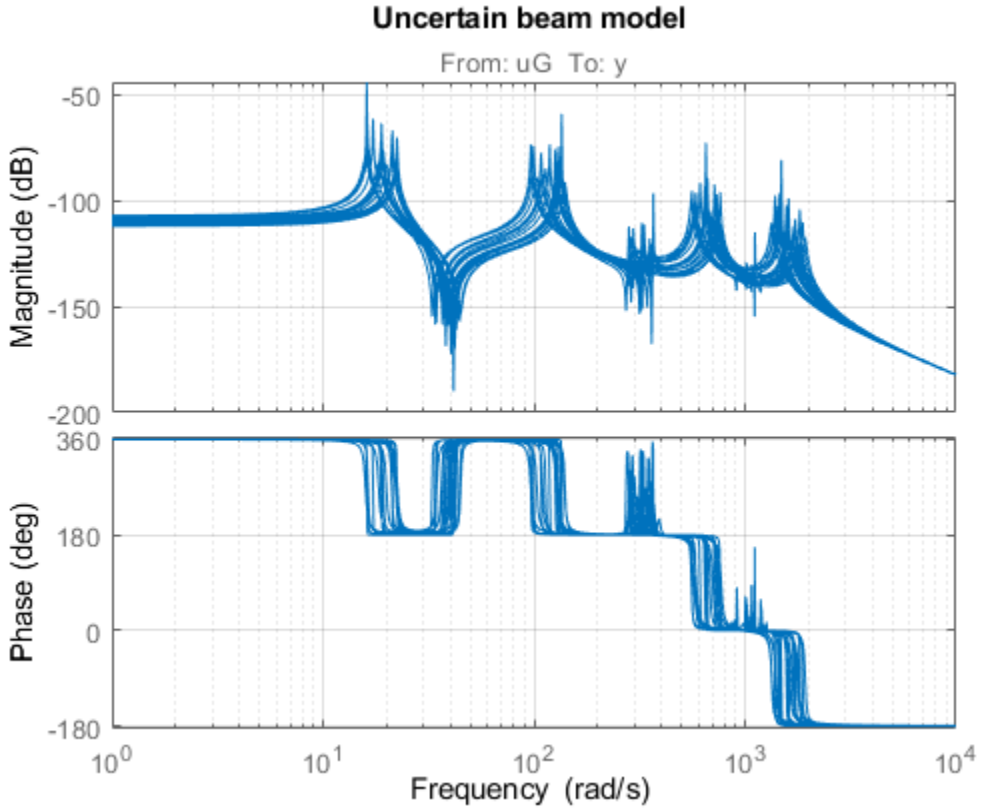
```
% Damping factor
zeta = ureal('zeta',0.002,'Range',[0.0002,0.02]);

% Natural frequencies
w1 = ureal('w1',18.95,'Percent',20);
w2 = ureal('w2',118.76,'Percent',20);
w3 = ureal('w3',332.54,'Percent',20);
w4 = ureal('w4',651.66,'Percent',20);
w5 = ureal('w5',1077.2,'Percent',20);
w6 = ureal('w6',1609.2,'Percent',20);
```

Next combine these uncertain coefficients into the expression for $G(s)$.

```
alpha = [9.72e-4 0.0122 0.0012 -0.0583 -0.0013 0.1199];
G = tf(alpha(1),[1 2*zeta*w1 w1^2]) + tf(alpha(2),[1 2*zeta*w2 w2^2]) + ...
    tf(alpha(3),[1 2*zeta*w3 w3^2]) + tf(alpha(4),[1 2*zeta*w4 w4^2]) + ...
    tf(alpha(5),[1 2*zeta*w5 w5^2]) + tf(alpha(6),[1 2*zeta*w6 w6^2]);
G.InputName = 'uG';  G.OutputName = 'y';
```

Visualize the impact of uncertainty on the transfer function from *u* to *y*. The `bode` function automatically shows the responses for 20 randomly selected values of the uncertain parameters.

```
rng(0), bode(G,{1e0,1e4}), grid
title('Uncertain beam model')
```



Robust LQG Control

LQG control is a natural formulation for active vibration control. With `systune`, you are not limited to a full-order optimal LQG controller and can tune controllers of any order. Here for example, let's tune a 6th-order state-space controller (half the plant order).

```
C = tunableSS('C',6,1,1);
```

The LQG control setup is depicted in Figure 2. The signals *d* and *n* are the process and measurement noise, respectively.

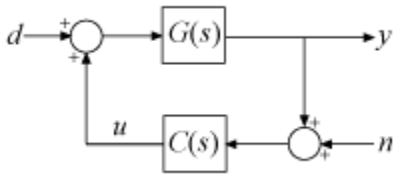


Figure 2: LQG control structure

Build a closed-loop model of the block diagram in Figure 2.

```
C.InputName = 'yn';  C.OutputName = 'u';
S1 = sumblk('yn = y + n');
S2 = sumblk('uG = u + d');
CL0 = connect(G,C,S1,S2,{'d','n'},{'y','u'});
```

Note that `CL0` depends on both the tunable controller `C` and the uncertain damping and natural frequencies.

```
CL0
```

```
CL0 =

  Generalized continuous-time state-space model with 2 outputs, 2 inputs, 18 states, and the fol
    C: Tunable 1x1 state-space model, 6 states, 1 occurrences.
    w1: Uncertain real, nominal = 18.9, variability = [-20,20]%, 3 occurrences
    w2: Uncertain real, nominal = 119, variability = [-20,20]%, 3 occurrences
    w3: Uncertain real, nominal = 333, variability = [-20,20]%, 3 occurrences
    w4: Uncertain real, nominal = 652, variability = [-20,20]%, 3 occurrences
    w5: Uncertain real, nominal = 1.08e+03, variability = [-20,20]%, 3 occurrences
    w6: Uncertain real, nominal = 1.61e+03, variability = [-20,20]%, 3 occurrences
    zeta: Uncertain real, nominal = 0.002, range = [0.0002,0.02], 6 occurrences

Type "ss(CL0)" to see the current value, "get(CL0)" to see all properties, and "CL0.Blocks" to i
```

Use an LQG criterion as control objective. This tuning goal lets you specify the noise covariances and the weights on the performance variables.

```
R = TuningGoal.LQG({'d','n'},{'y','u'},diag([1,1e-10]),diag([1 1e-12]));
```

Now tune the controller `C` to minimize the LQG cost over the entire uncertainty range.

```
[CL,fSoft,~,Info] = systune(CL0,R);
```

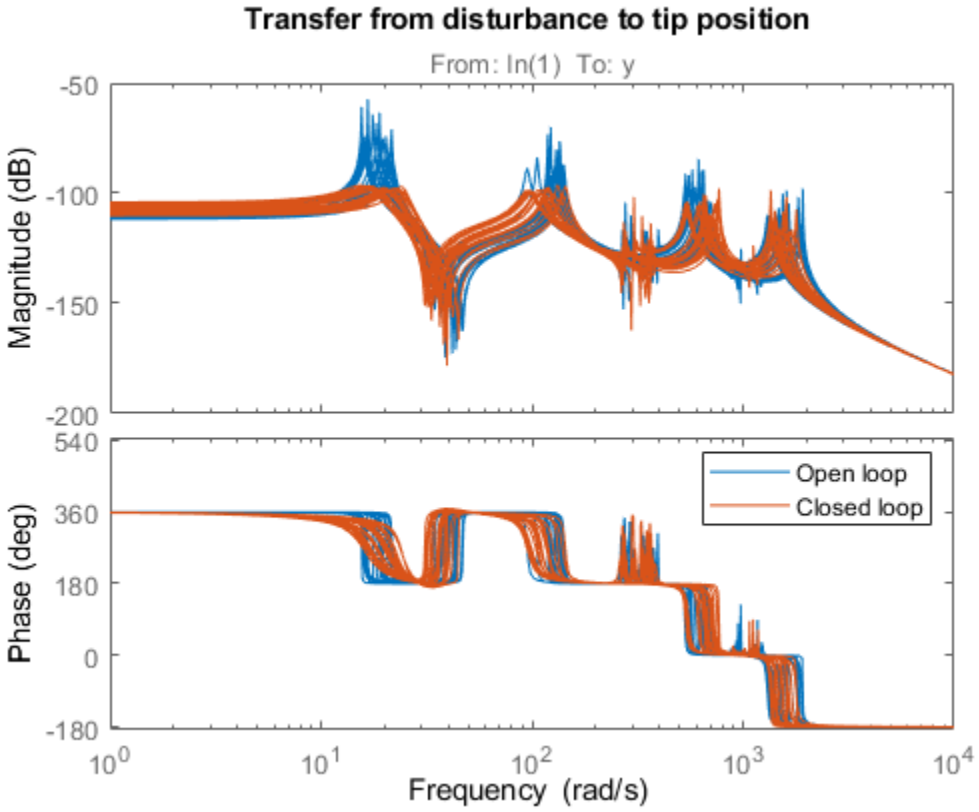
```
Soft: [5.63e-05,Inf], Hard: [-Inf,Inf], Iterations = 68
Soft: [6.27e-05,0.000102], Hard: [-Inf,-Inf], Iterations = 78
Soft: [6.96e-05,7.38e-05], Hard: [-Inf,-Inf], Iterations = 82
Soft: [7.2e-05,7.2e-05], Hard: [-Inf,-Inf], Iterations = 40
Final: Soft = 7.2e-05, Hard = -Inf, Iterations = 268
```

Validation

Compare the open- and closed-loop Bode responses from *d* to *y* for 20 randomly chosen values of the uncertain parameters. Note how the controller clips the first three peaks in the Bode response.

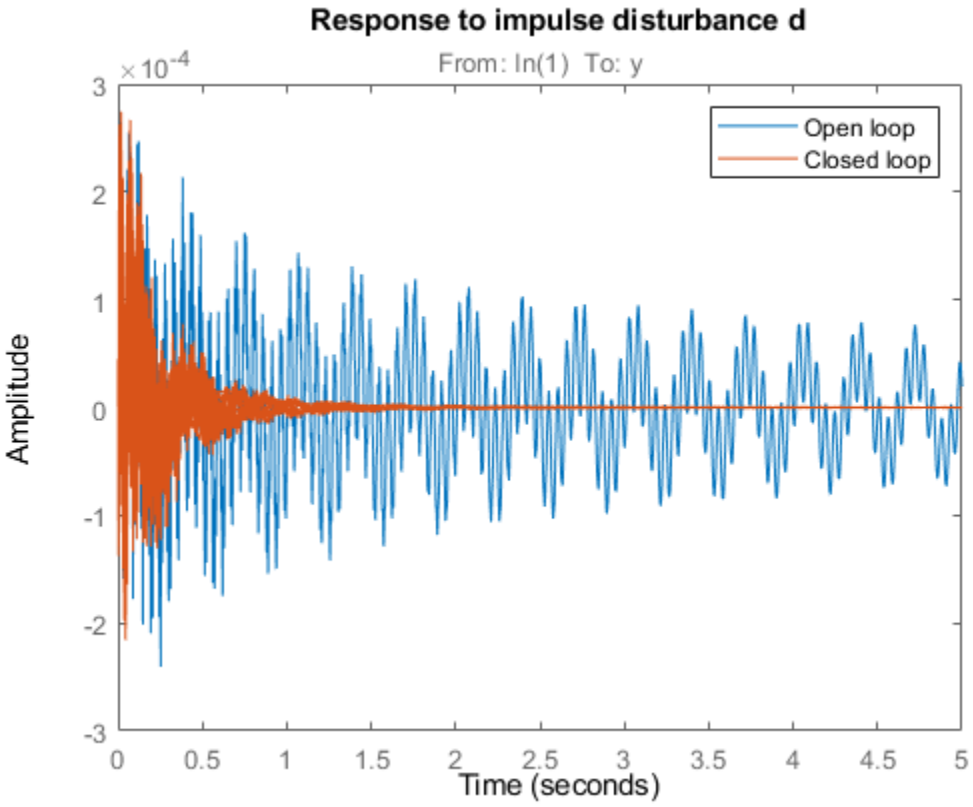
```
Tdy = getIOTransfer(CL,'d','y');
bode(G,Tdy,{1e0,1e4})
```

```
title('Transfer from disturbance to tip position')
legend('Open loop','Closed loop')
```



Next plot the open- and closed-loop responses to an impulse disturbance *d*. For readability, the open-loop response is plotted only for the nominal plant.

```
impulse(getNominal(G),Tdy,5)
title('Response to impulse disturbance d')
legend('Open loop','Closed loop')
```



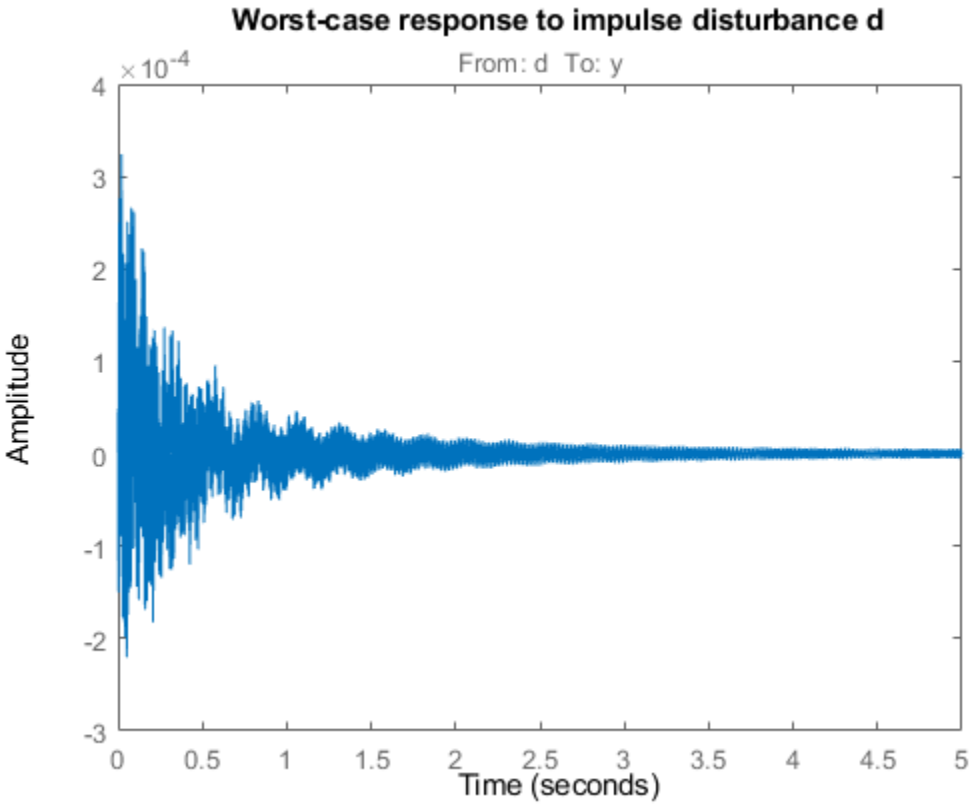
Finally, `systune` also provides insight into the worst-case combinations of damping and natural frequency values. This information is available in the output argument `Info`.

```
WCU = Info.wcPert
```

```
WCU=3×1 struct array with fields:
    w1
    w2
    w3
    w4
    w5
    w6
    zeta
```

Use this data to plot the impulse response for the two worst-case scenarios.

```
impulse(usubs(Tdy,WCU),5)
title('Worst-case response to impulse disturbance d')
```



See Also

Related Examples

- “Build Tunable Control System Model With Uncertain Parameters” on page 6-13
- “Robust Tuning of DC Motor Controller” on page 6-32
- “Robust Tuning of Mass-Spring-Damper System” on page 6-24

More About

- “Interpreting Results of Robust Tuning” on page 6-11

Fault-Tolerant Control of a Passenger Jet

This example shows how to tune a fixed-structure controller for multiple operating modes of the plant.

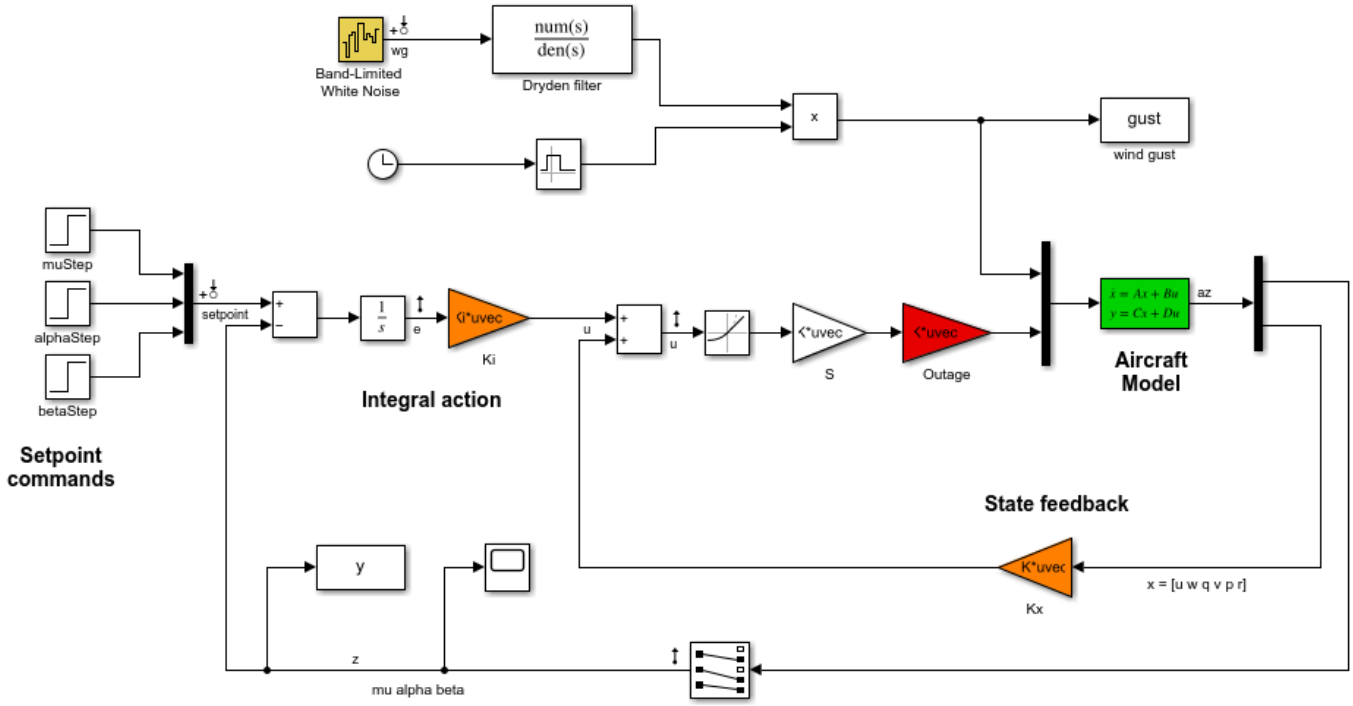
Background

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.

```
addpath(fullfile(matlabroot,'examples','control','main')) % add example data
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]));
ST0 = slTuner('faultTolerantAircraft',{'Ki','Kx'},OutageData);
```

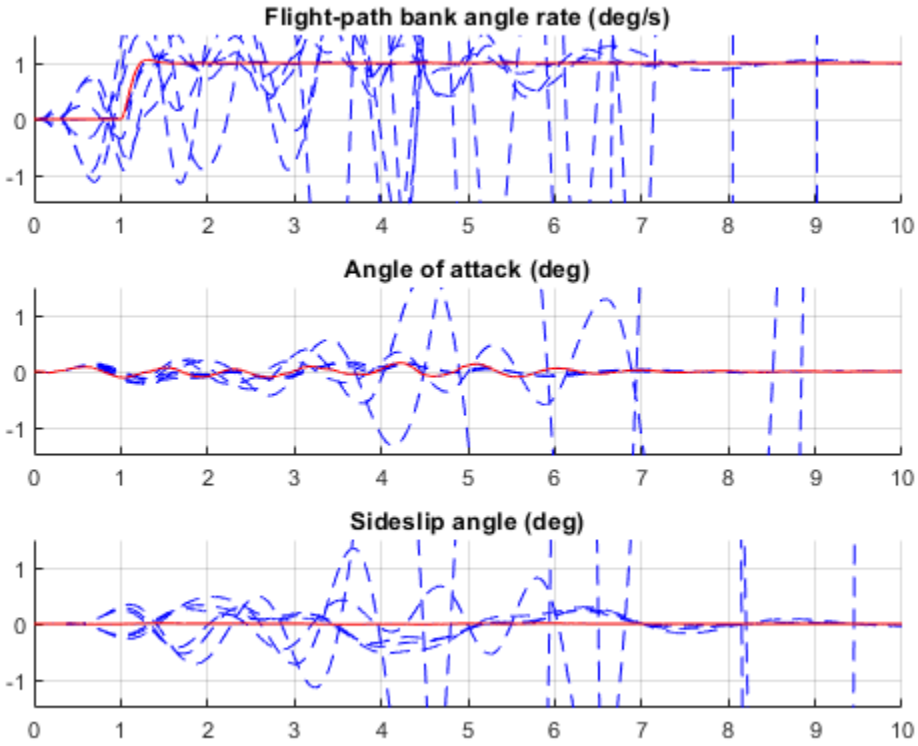
Use `systune` to tune the controller gains subject to the nominal requirements. Treat the wind gust alleviation as a hard constraint.

```
[ST,fSoft,gHard]  = systune(ST0,SoftNom,HardNom);
```

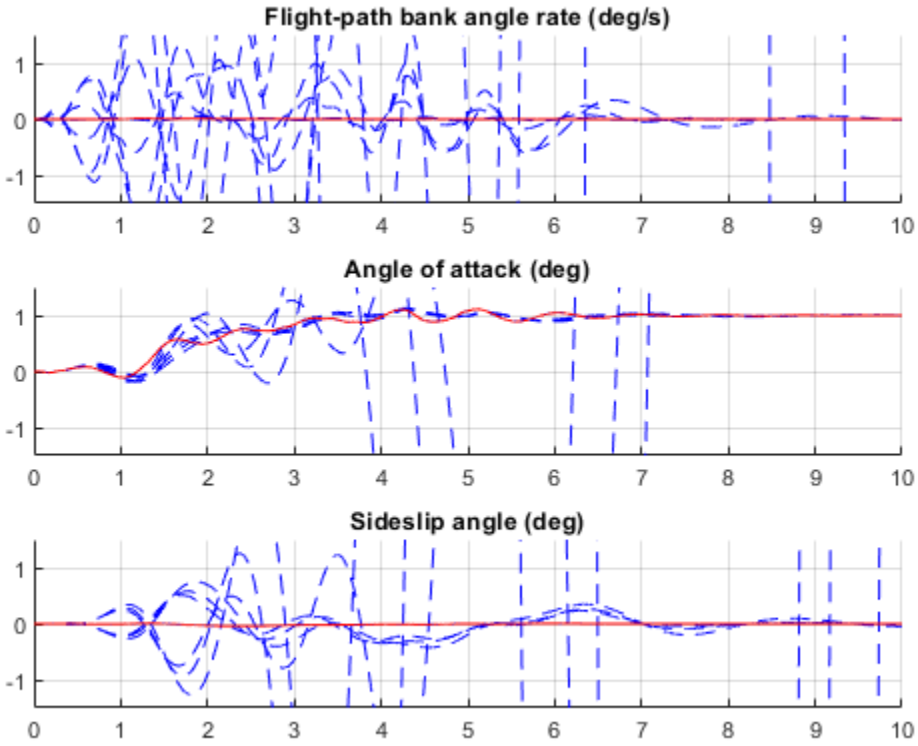
```
Final: Soft = 22.6, Hard = 0.99989, Iterations = 296
```

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.

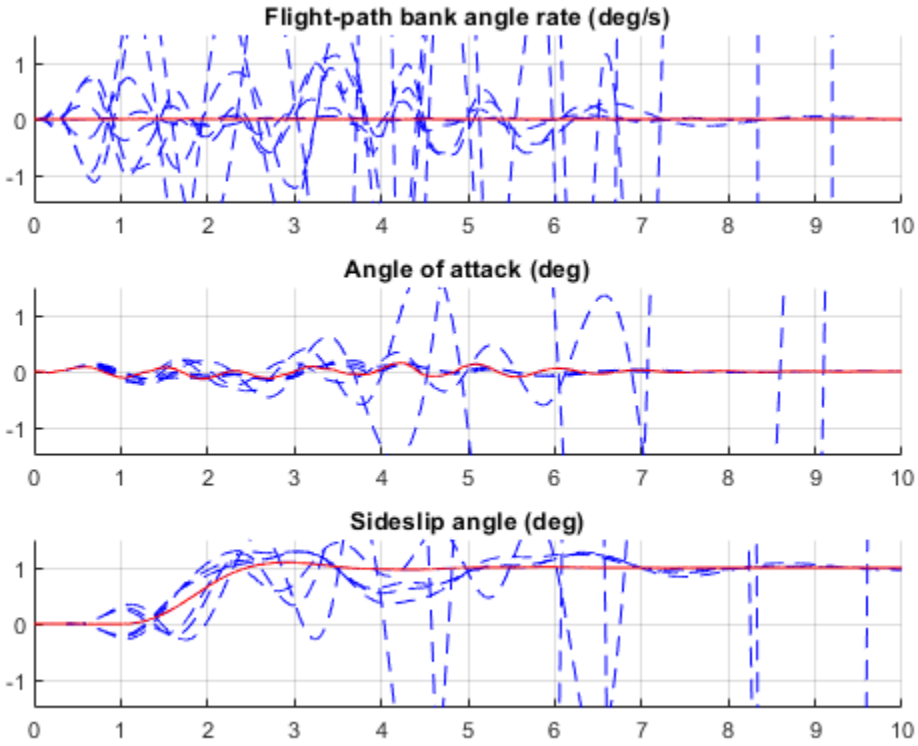
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% Bank-angle setpoint simulation
plotResponses(OutageCases,1,0,0);
```




```
% Angle-of-attack setpoint simulation
plotResponses(OutageCases,0,1,0);
```




```
% 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

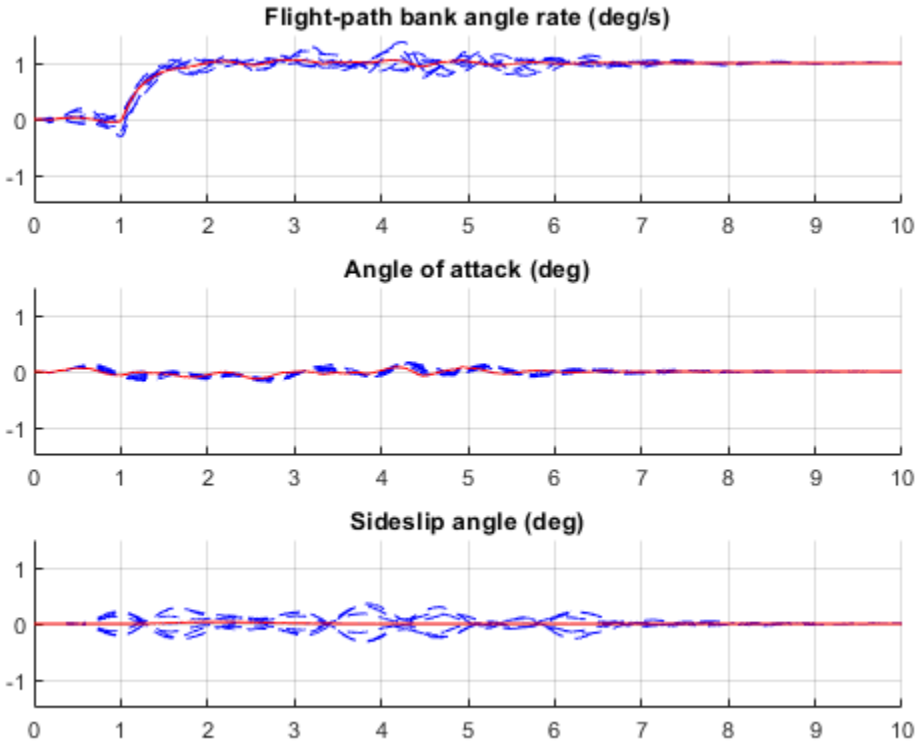
To improve reliability, retune the controller gains to meet the nominal requirement for the nominal plant as well as the relaxed requirements for all eight outage scenarios.

```
[ST,fSoft,gHard]  = systune(ST0,[SoftNom;SoftOut],[HardNom;HardOut]);
```

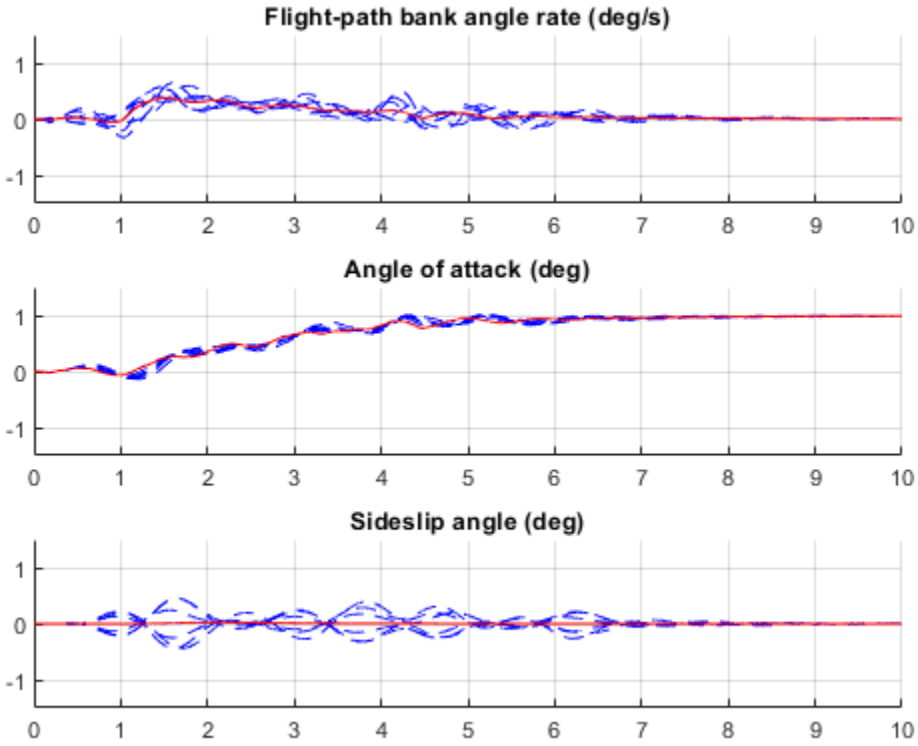
```
Final: Soft = 25.7, Hard = 0.99966, Iterations = 489
```

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).

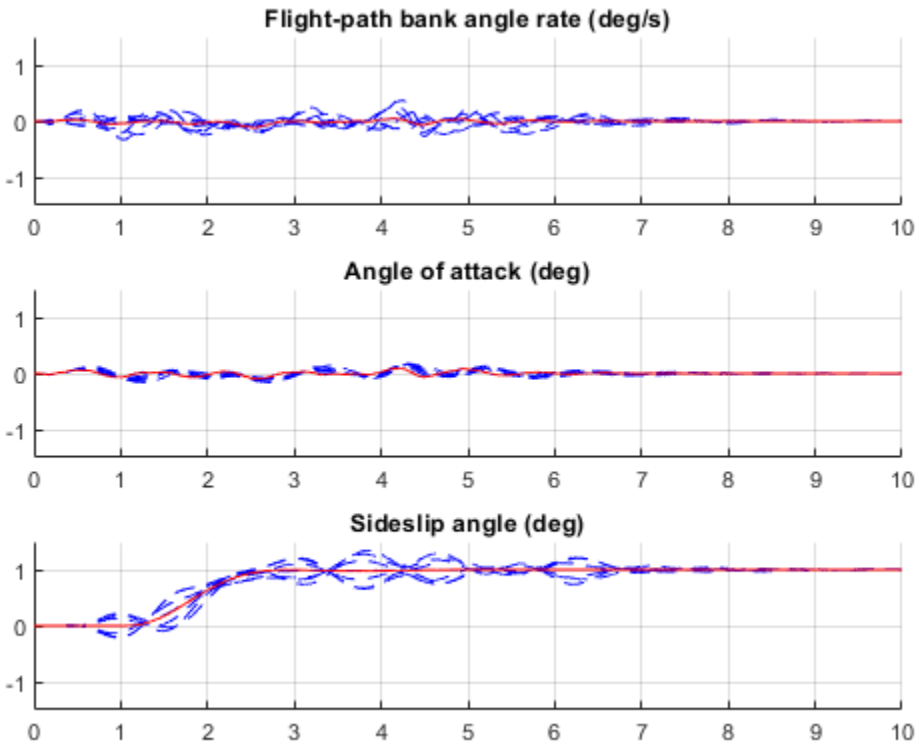
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% 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.

```
rmpath(fullfile(matlabroot,'examples','control','main')) % remove example data
```

See Also

Related Examples

- “Robust Tuning of Mass-Spring-Damper System” on page 6-24

More About

- “Robust Tuning Approaches” on page 6-2

Tuning for Multiple Values of Plant Parameters

This example shows how to use **Control System Tuner** to tune a control system when there are parameter variations in the plant. The control system used in this example is an active suspension of a quarter-car model. The example uses **Control System Tuner** to tune the system to meet performance objectives when parameters in the plant vary from their nominal values.

Quarter-Car Model and Active Suspension Control

A simple quarter-car model of an active suspension system is shown in Figure 1. The quarter-car model consists of two masses, a car chassis with mass m_b and a wheel assembly of mass m_w. There is a spring k_s and damper b_s between the masses, which models the passive spring and shock absorber. The tire between the wheel assembly and the road is modeled by the spring k_t.

Active suspension introduces a force f_s between the chassis and wheel assembly and allows the designer to balance driving objectives such as passenger comfort and road handling with the use of a feedback controller.

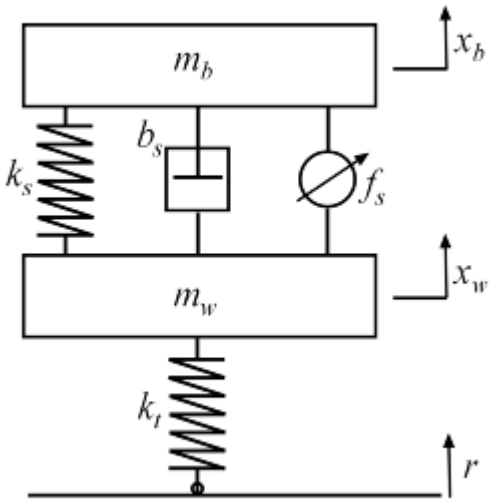
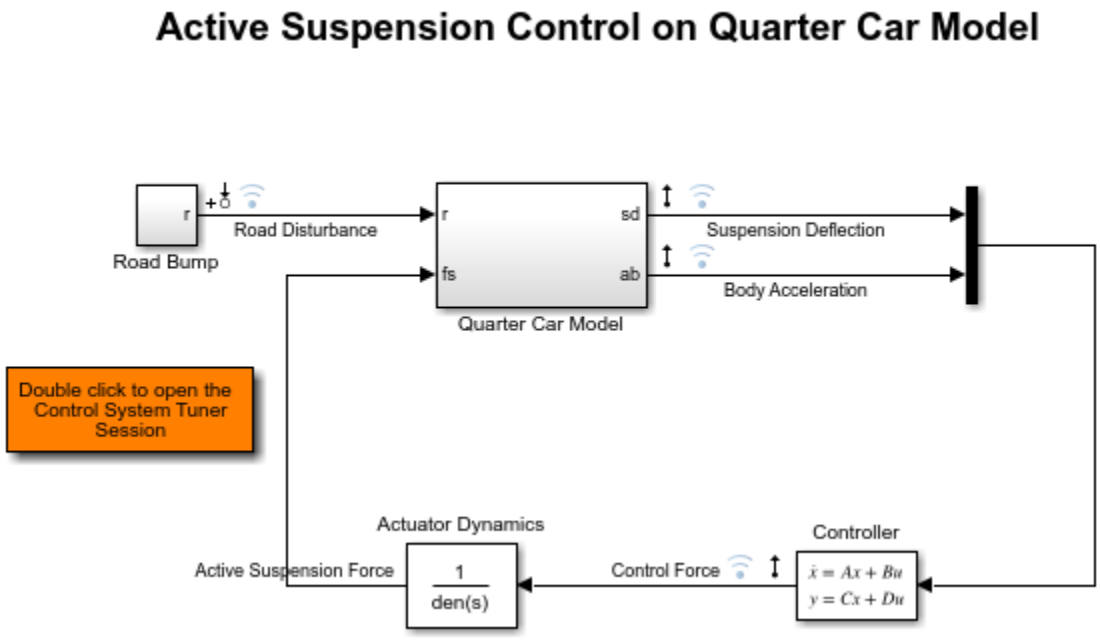


Figure 1: Quarter-car model of active suspension.

Control Architecture

The quarter-car model is implemented using Simscape. The following Simulink model contains the quarter-car model with active suspension, controller and actuator dynamics. Its inputs are road disturbance and the force for the active suspension. Its outputs are the suspension deflection and body acceleration. The controller uses these measurements to send a control signal to the actuator that creates the force for active suspension.

```
mdl = 'rct_suspension.slx';
open_system(mdl)
```



Control Objectives

The example has the following three control objectives:

- Good handling defined from road disturbance to suspension deflection.
- User comfort defined from road disturbance to body acceleration.
- Reasonable control bandwidth.

The nominal values of the spring constant k_s and damper b_s between the body and the wheel assembly are not exact and due to the imperfections in the materials, these values can be constant but different. Assess the impact on the system control using a variety of parameter values.

Model the road disturbance of magnitude seven centimeters and use a constant weight.

```
Wroad = ss(0.07);
```

Define the closed-loop target for handling from road disturbance to suspension deflection as

```
HandlingTarget = 0.044444 * tf([1/8 1],[1/80 1]);
```

Define the target for comfort from road disturbance to body acceleration.

```
ComfortTarget = 0.6667 * tf([1/0.45 1],[1/150 1]);
```

Limit the control bandwidth by the weight function from road disturbance to the control signal.

```
Wact = tf(0.1684*[1 500],[1 50]);
```

For more information on selecting the closed-loop targets and the weight function, see “Robust Control of an Active Suspension” on page 5-13.

Controller Tuning

To open a **Control System Tuner** session for active suspension control, in the Simulink model, Double click to the orange block. Tuned block is set to the second order Controller and three tuning goals are defined to achieve the handling, comfort and control bandwidth as described above. In order to see the performance of the tuning, the step responses from road disturbance to suspension deflection, the body acceleration and the control force are plotted.

Handling, comfort, and control bandwidth goals are defined as gain limits, `HandlingTarget/Wroad`, `ComfortTarget/Wroad` and `Wact/Wroad`. All gain functions are divided by `Wroad` to incorporate the road disturbance.

The open-loop system with zero controller violates the handling goal and results in highly oscillatory behavior for both suspension deflection and body acceleration with long settling time.

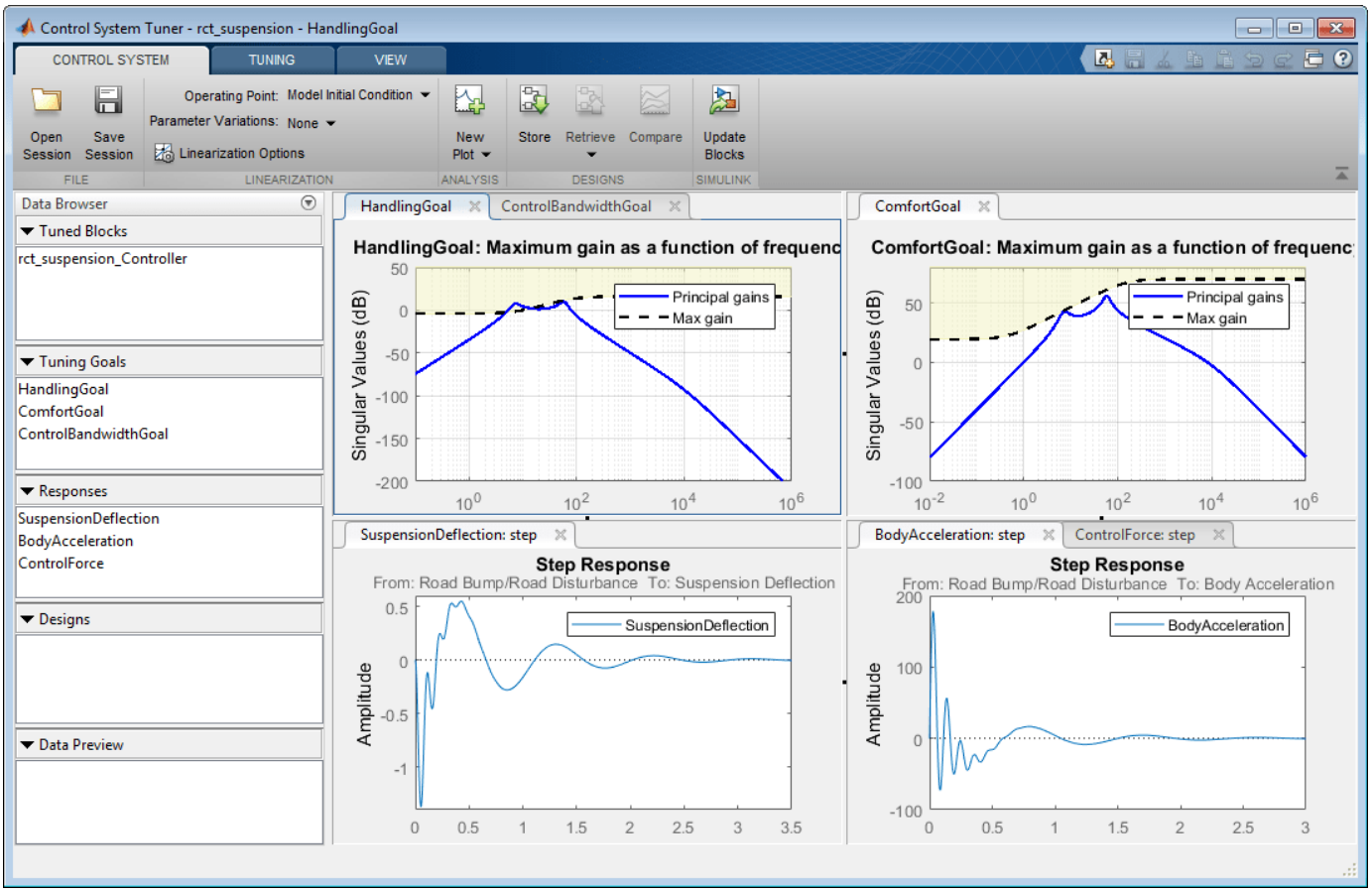


Figure 2: Control System Tuner with Session File.

To tune the controller using **Control System Tuner**, on the **Tuning** tab, click **Tune**. As shown in Figure 3, this design satisfies the tuning goals and the responses are less oscillatory and converges quickly to zero.

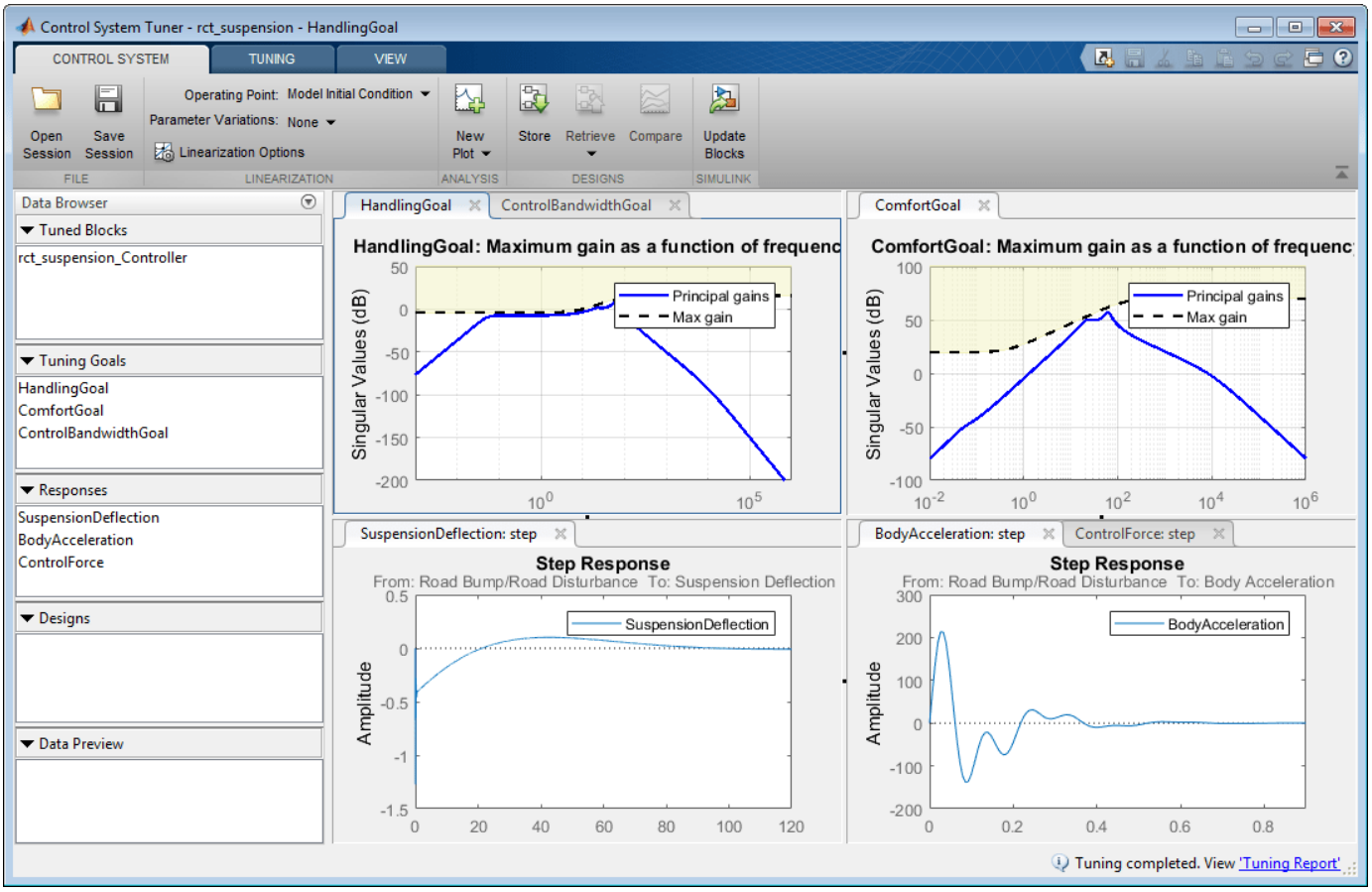


Figure 3: Control System Tuner after tuning.

Controller Tuning for Multiple Parameter Values

Now, try to tune the controller for multiple parameter values. The default value for car chassis of mass m_b is 300 kg. Vary the mass to 100 kg, 200 kg and 300 kg for different operation conditions.

In order to vary these parameters in **Control System Tuner**, on the **Control System** tab, under **Parameter Variations**, select **Select parameters to Vary**. Define the parameters in the dialog that opens.

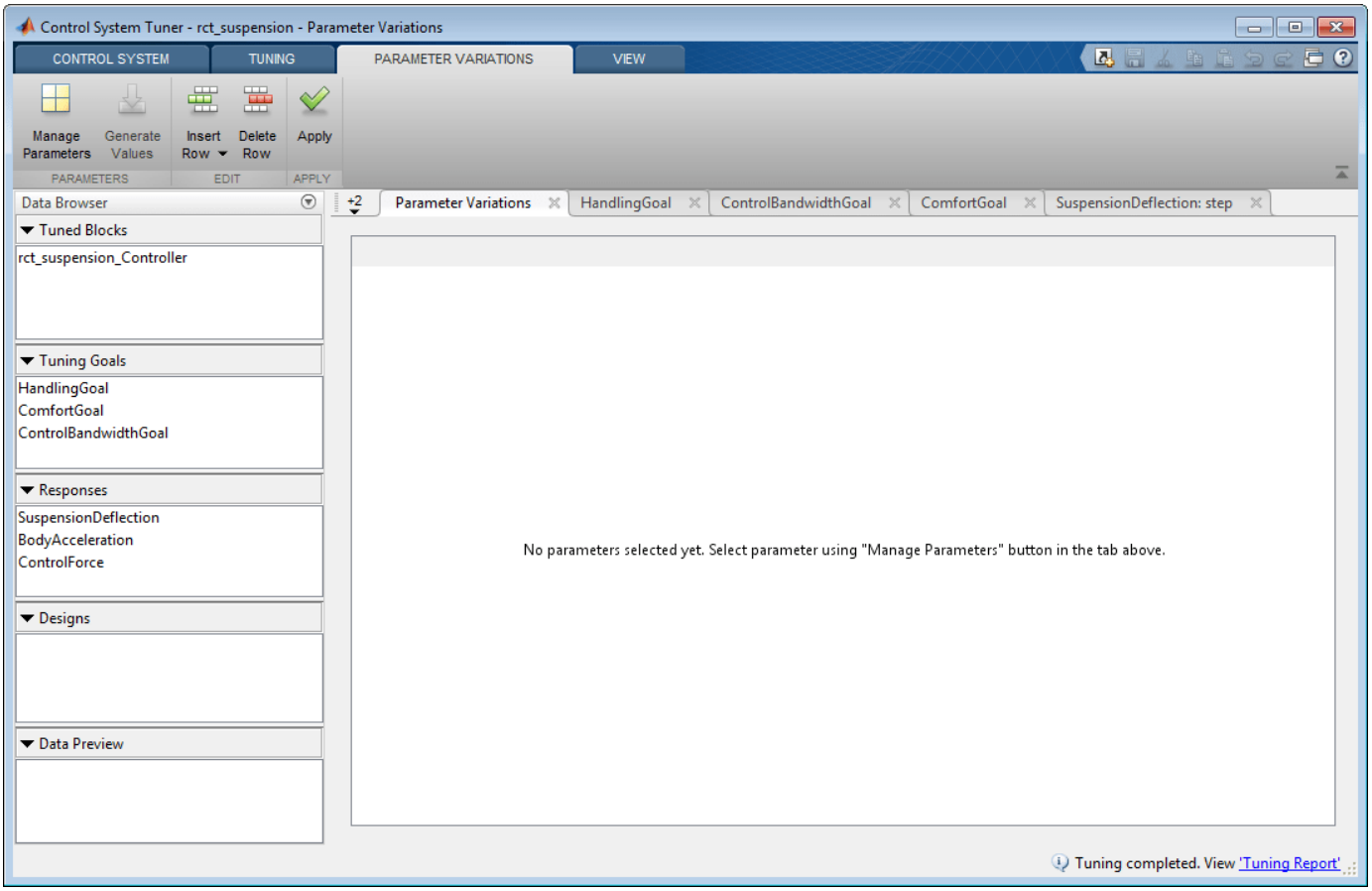


Figure 4: Defining parameter variations.

On the **Parameter Variations** tab, click **Manage Parameters**. In the Select model variables dialog box, select `Mb`.

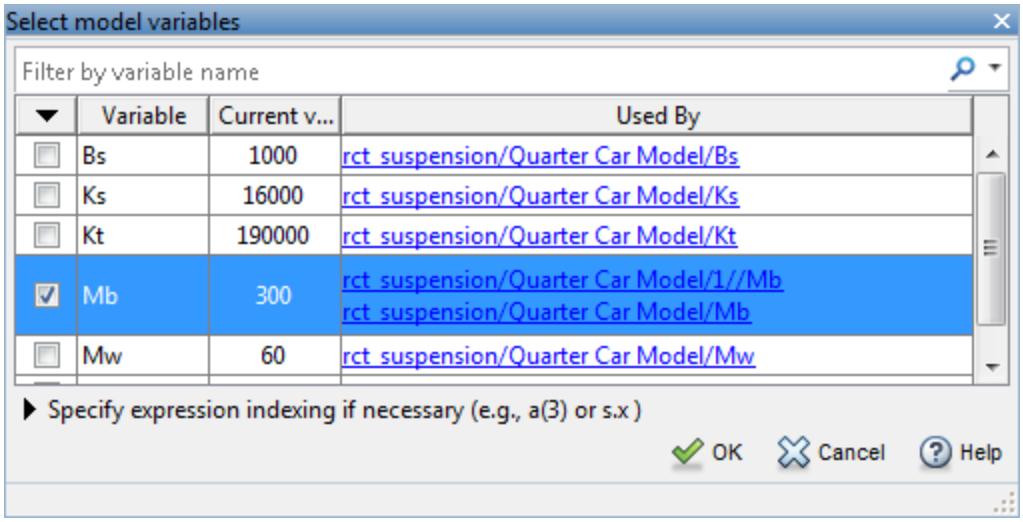


Figure 5: Select a parameter to vary from the model.

Now, the parameter `Mb` is added with default values in the parameter variations table.

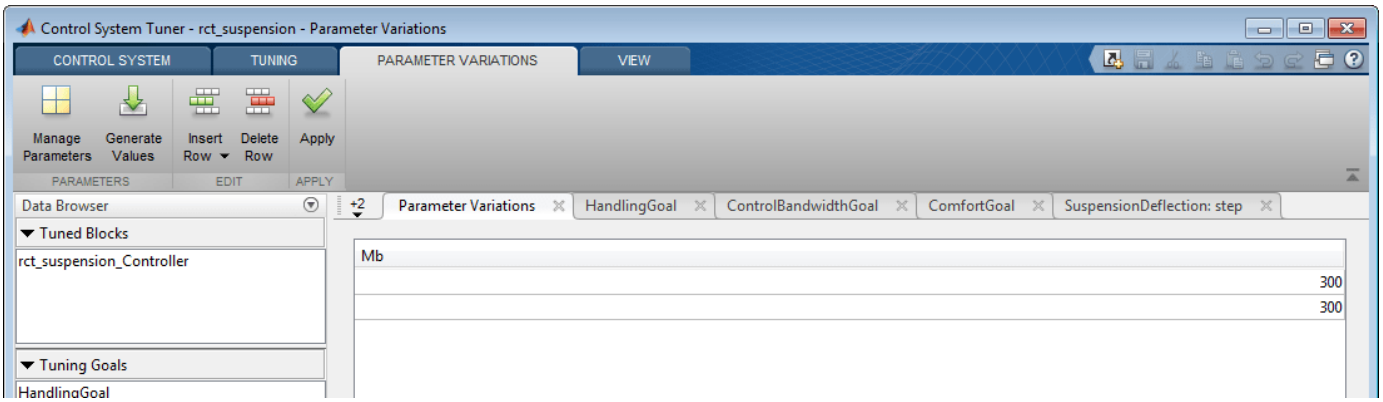


Figure 6: Parameter variations table with default values.

To generate variations quickly, click **Generate Values**. In the Generate Parameter Values dialog box, define values 100, 200, 300 for `Mb`, and click `Overwrite`.

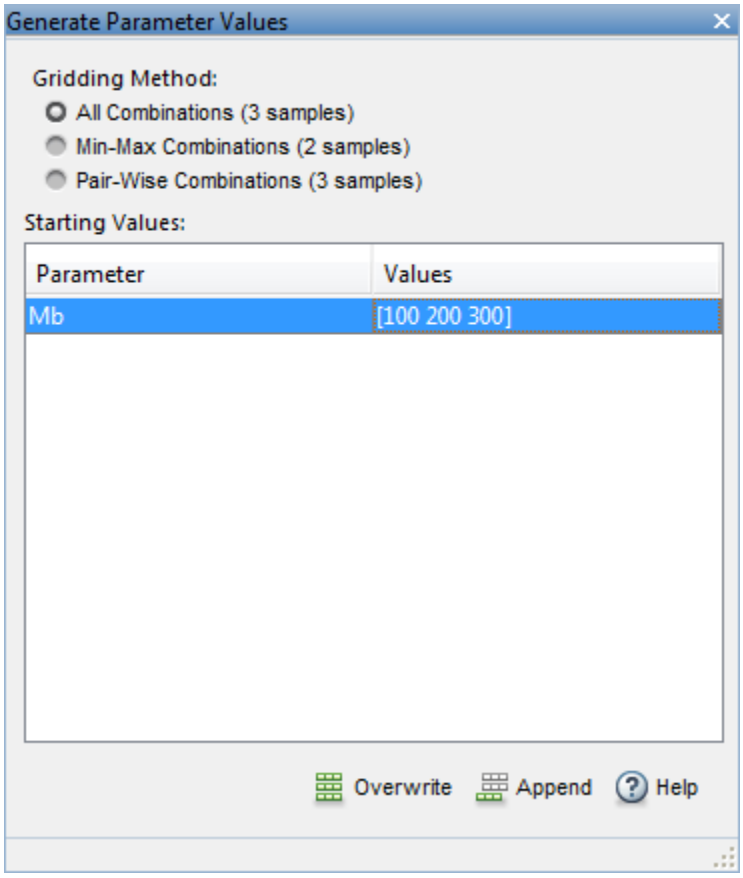


Figure 7: Generate values window.

All values are populated in the parameter variations table. To set the parameter variations to **Control System Tuner**, click **Apply**.

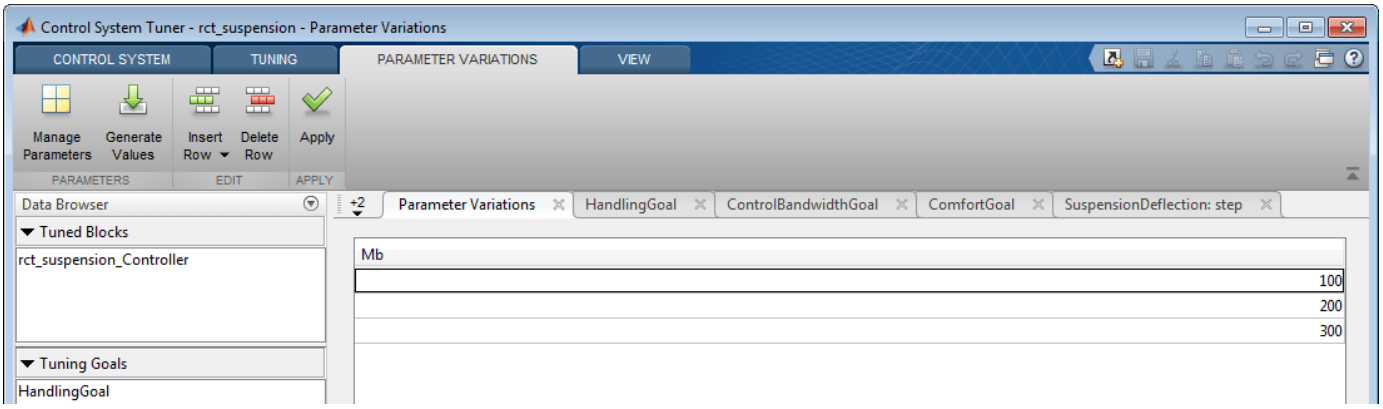


Figure 8: Parameter variations table with updated values.

Multiple lines appear in the tuning goal and response plots due to the varying parameters. The controller obtained for these nominal parameter values results in an unstable closed-loop system.

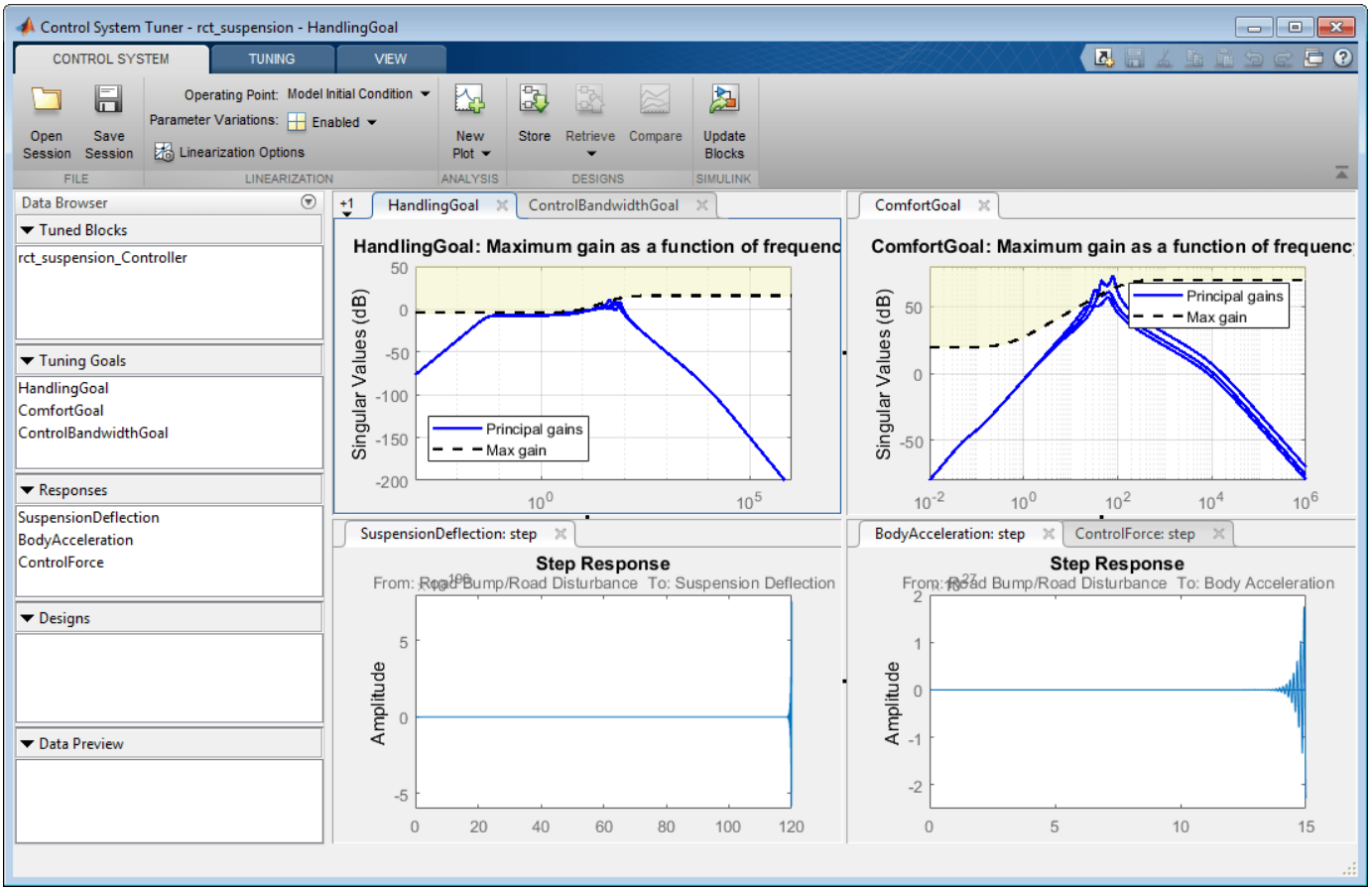


Figure 9: Control System Tuner with multiple parameter variations.

Tune the controller to satisfy the handling, comfort, and control bandwidth objectives by clicking **Tune** in **Tuning** tab. The tuning algorithm tries to satisfy these objectives for the nominal parameters

and for all parameter variations. This is a challenging task in contrast to nominal design as shown in Figure 10.

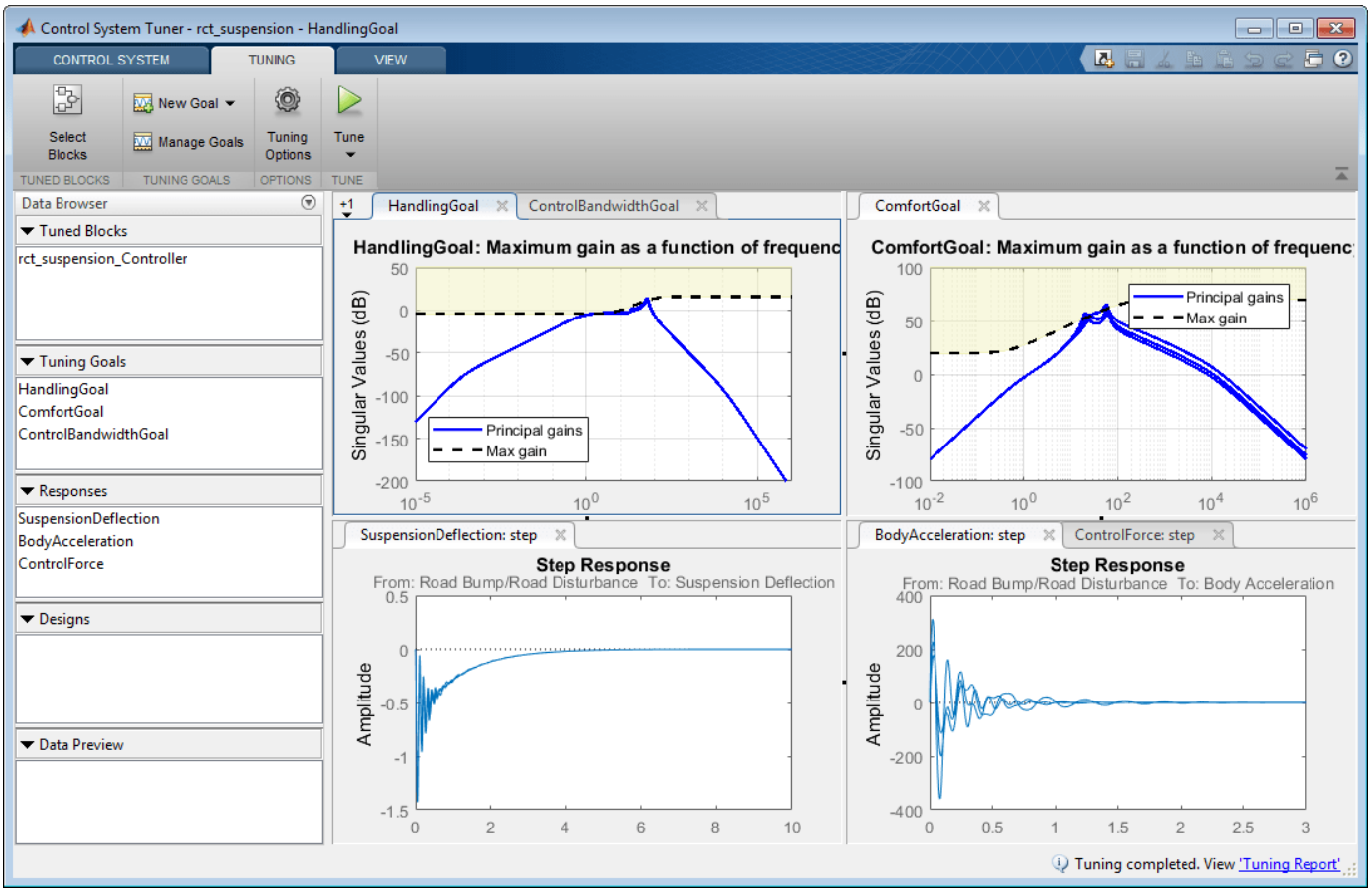


Figure 10: Control System Tuner with multiple parameter variations (Tuned).

Control System Tuner tunes the controller parameters for the linearized control system. To examine the performance of the tuned parameters on the Simulink model, update the controller in the Simulink model by clicking **Update Blocks** on the **Control System** tab.

Simulate the model for each of the parameter variations. Then, using the Simulation Data Inspector, examine the results for all simulations. The results are shown in Figure 11. For all three parameter variations, the controller tries to minimize the suspension deflection and body acceleration with minimal control effort.

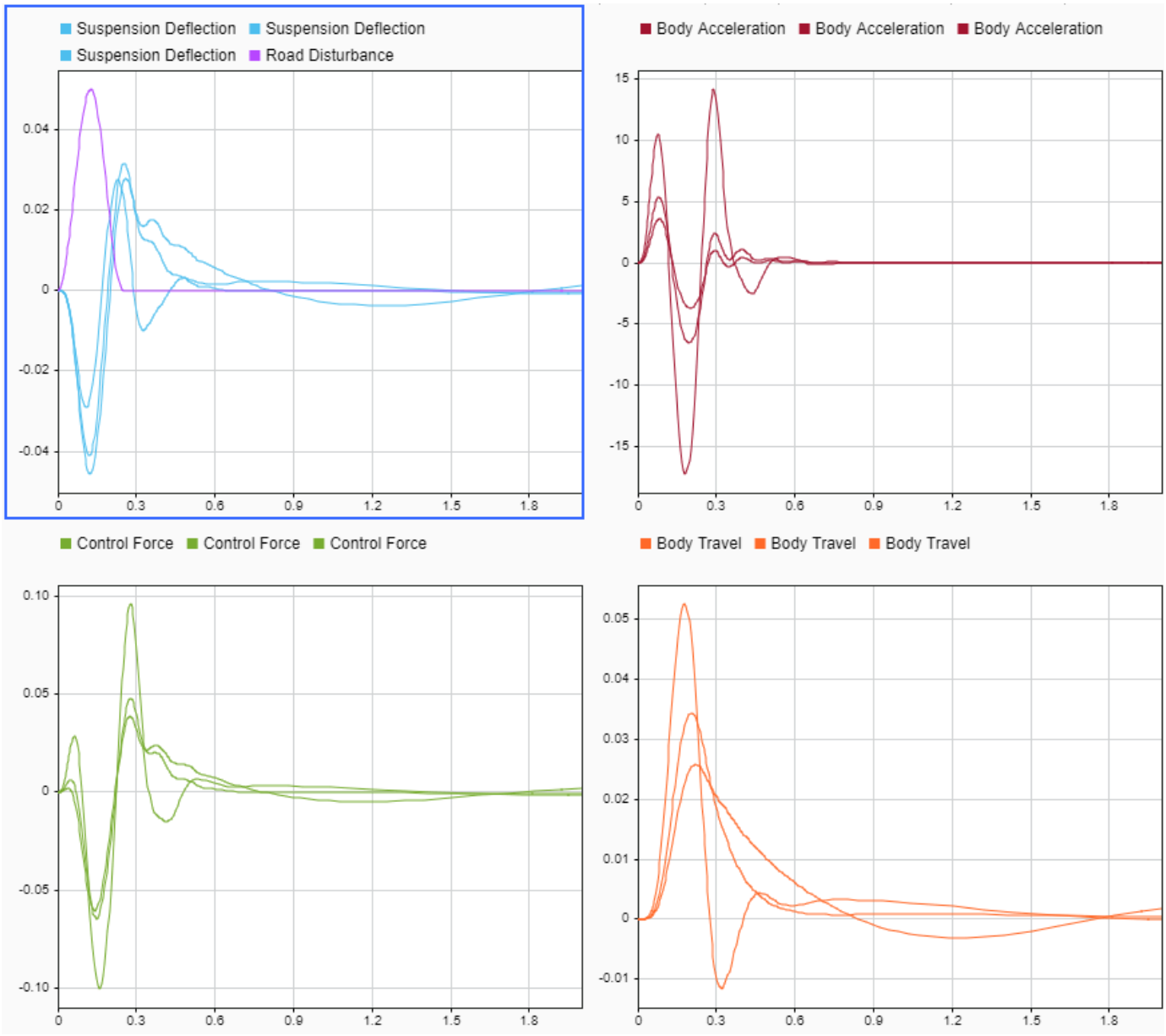


Figure 11: Controller performance on the Simulink model.

See Also

Related Examples

- “Robust Tuning of Mass-Spring-Damper System” on page 6-24

More About

- “Robust Tuning Approaches” on page 6-2

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).

See Also

Related Examples

- “Difference Between Fixed-Structure Tuning and Traditional H-Infinity Synthesis” on page 7-3

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 parameterization 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.

See Also

Related Examples

- “What Is hinfstruct?” on page 7-4

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 on page 5-2).

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

See Also

Related Examples

- “Structured H-Infinity Synthesis Workflow” on page 7-6

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:

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

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

For an example of how to formulate design requirements for H_∞ synthesis using loop shaping, see “Fixed-Structure H-infinity Synthesis with hinfstruct” on page 7-17.

For more information about constructing weighting functions from design requirements, see “H-Infinity Performance” on page 5-7.

Structured H-Infinity Synthesis Workflow

Performing structured H_∞ synthesis requires the following steps:

1. Formulate your design requirements as H_∞ constraints on page 7-5, which are constraints on the closed-loop gains from specific system inputs to specific system outputs.
2. Build tunable models on page 7-7 of the closed-loop transfer functions of Step 1.
3. Tune the control system on page 7-12 using `hinfstruct`.
4. Validate the tuned control system on page 7-14.

Build Tunable Closed-Loop Model for Tuning with hinfstruct

In “Formulating Design Requirements as H-Infinity Constraints” on page 7-5 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. on page 7-7
- Obtain the model from a Simulink model using Simulink Control Design commands. on page 7-10

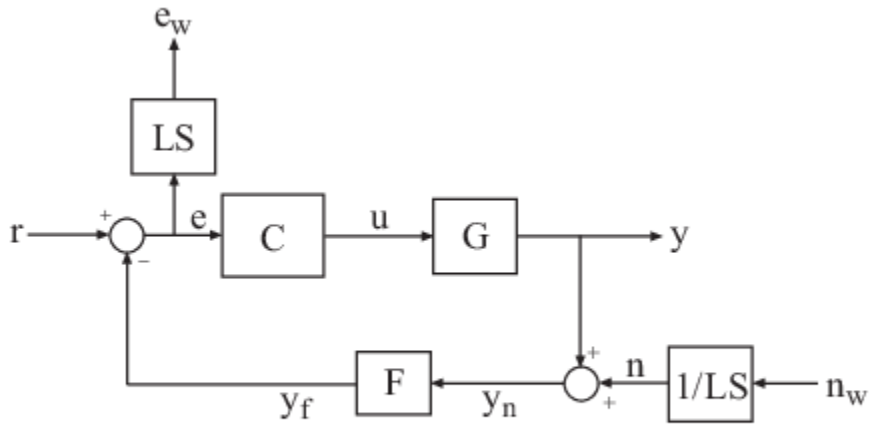
Constructing the Closed-Loop System Using Control System Toolbox Commands

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”.
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`.

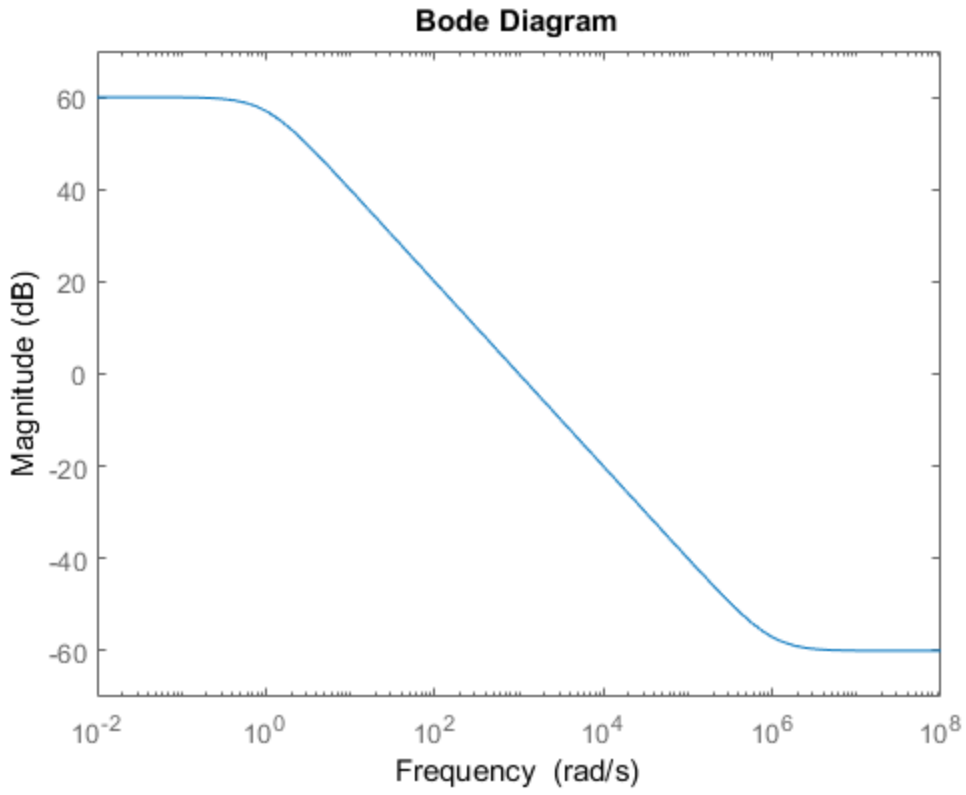
The block diagram also includes the weighting functions LS and $1/LS$, which express the loop-shaping requirements. Let $T(s)$ denote the closed-loop transfer function from inputs (r,n_w) to outputs (y,e_w). Then, the H_∞ constraint:

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

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

$$LS = \frac{1 + 0.001\frac{s}{\omega_C}}{0.001 + \frac{s}{\omega_C}}.$$

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



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

1 Load the plant `G` from a saved file.

```
load hinfstruct_demo G
```

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

2. Create a tunable model of the PI controller.

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

   ```
   C = tunablePID('C','pi');
   ```

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

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

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

4. Specify the target loop shape `LC`.

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

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

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

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

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

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

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

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

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

```
T0.Blocks
```

```
ans = struct with fields:
    C: [1x1 tunablePID]
    a: [1x1 realp]
```

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

You can now use `hinfstruct` to tune the parameters of this control system. See “Tune the Controller Parameters” on page 7-12.

In this example, the control system model `T0` is a continuous-time model (`T0.Ts = 0`). You can also use `hinfstruct` with a discrete-time model, provided that you specify a definite sample time (`T0.Ts ≠ –1`).

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`. (Simulink-based functionality is not available in MATLAB Online™.)

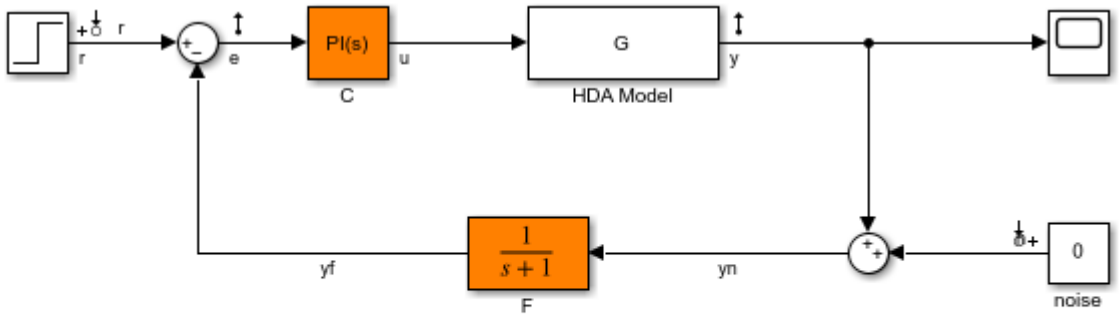
Example: Creating a Weighted Tunable Model of Control System Starting From a Simulink Model

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

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

1 Open the model.

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



2 Create an `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 7-7. $T(s)$ is a weighted closed-loop model of the control system of `rct_diskdrive`. Tuning `T0` to enforce the H_∞ constraint

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

approximately enforces the target loop shape `LS`.

You can now use `hinfstruct` to tune the parameters of this control system. See “Tune the Controller Parameters” on page 7-12.

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 7-7 using the following command:

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

```
Final: Peak gain = 3.88, Iterations = 67
```

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.

See Also

Related Examples

- “Interpret the Outputs of hinfstruct” on page 7-13

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.

See Also

Related Examples

- “Validate the Controller Design” on page 7-14

Validate the Controller Design

To validate the `hinfstruct` control design, analyze the tuned output models described in “Interpret the Outputs of hinfstruct” on page 7-13. Use these tuned models to examine the performance of the tuned system.

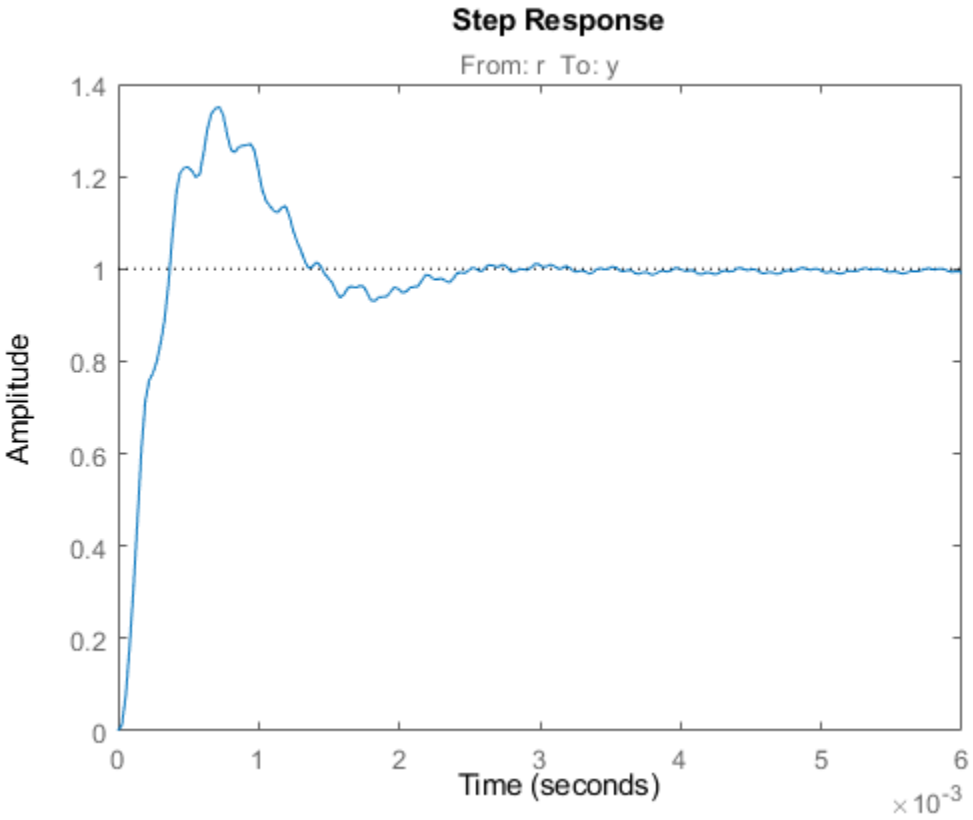
Validating the Design in MATLAB

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 7-12. 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)
```



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 7-12.

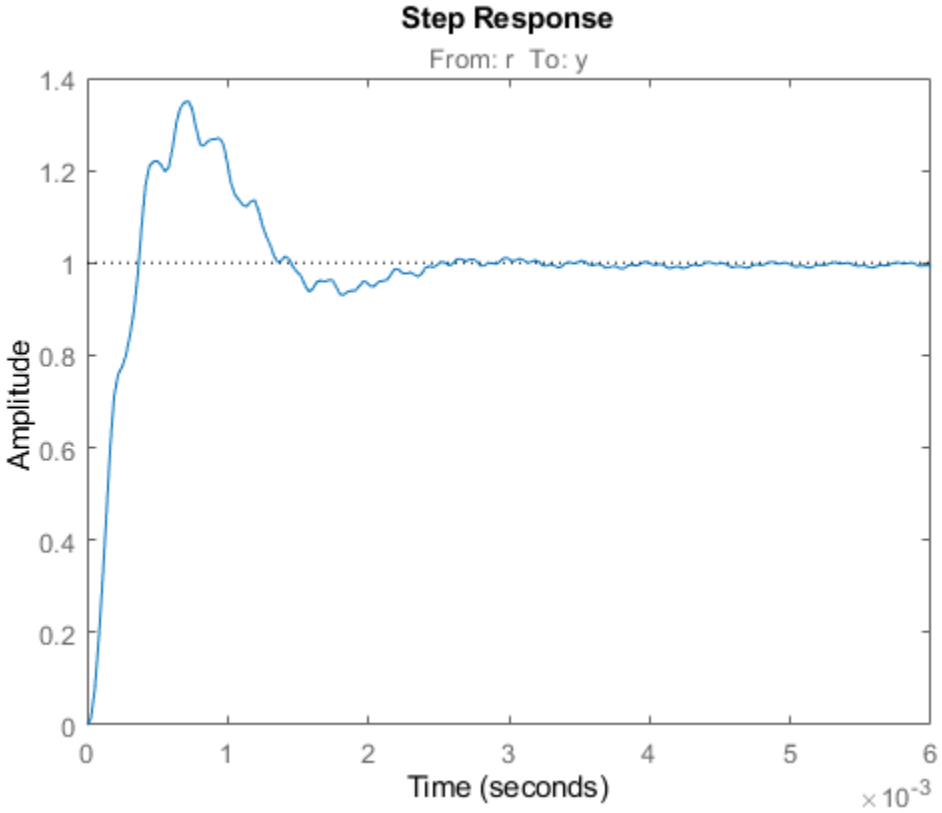
Make a copy of the `slTuner` description of the control system, to preserve the original parameter values. Then propagate the tuned parameter values to the copy.

```
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)
```



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.

See Also

Related Examples

- “Fixed-Structure H-infinity Synthesis with hinfstruct” on page 7-17

Fixed-Structure H-infinity Synthesis with hinfstruct

This example uses the `hinfstruct` command to tune a fixed-structure controller subject to H_∞ constraints.

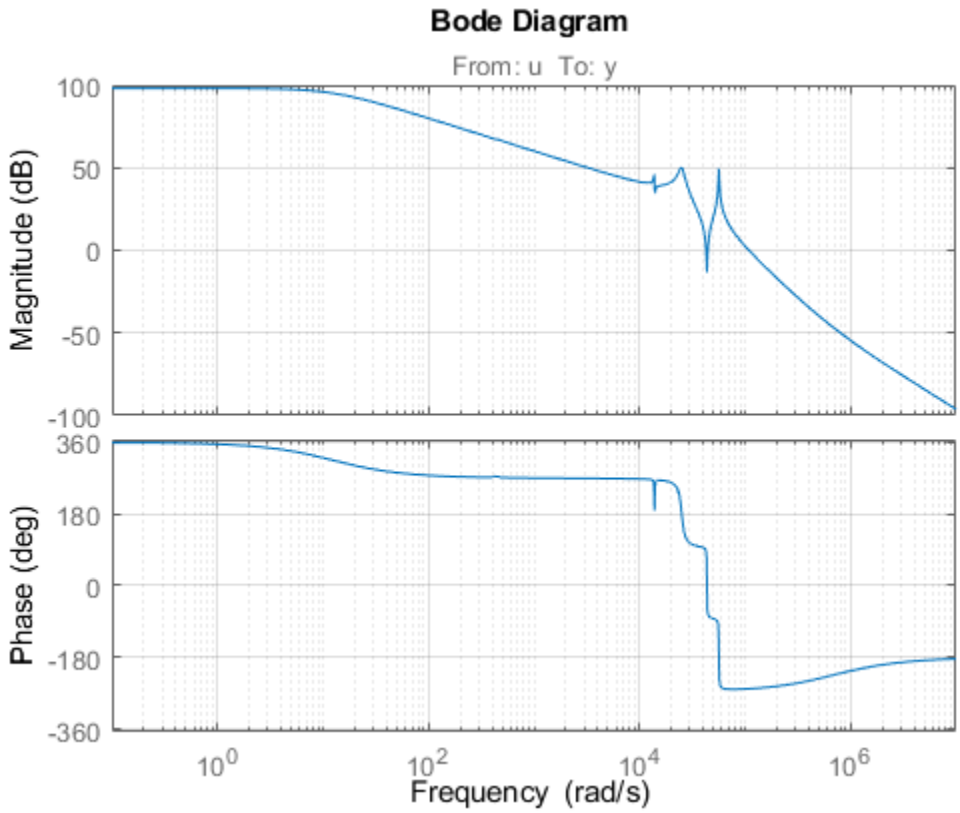
Introduction

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
```



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.

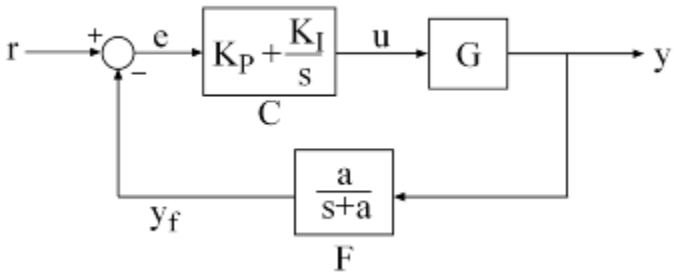


Figure 1: Control Structure

Tunable Elements

There are two tunable elements in the control structure of Figure 1: the PI controller $C(s)$ and the low-pass filter

$$F(s) = \frac{a}{s+a}.$$

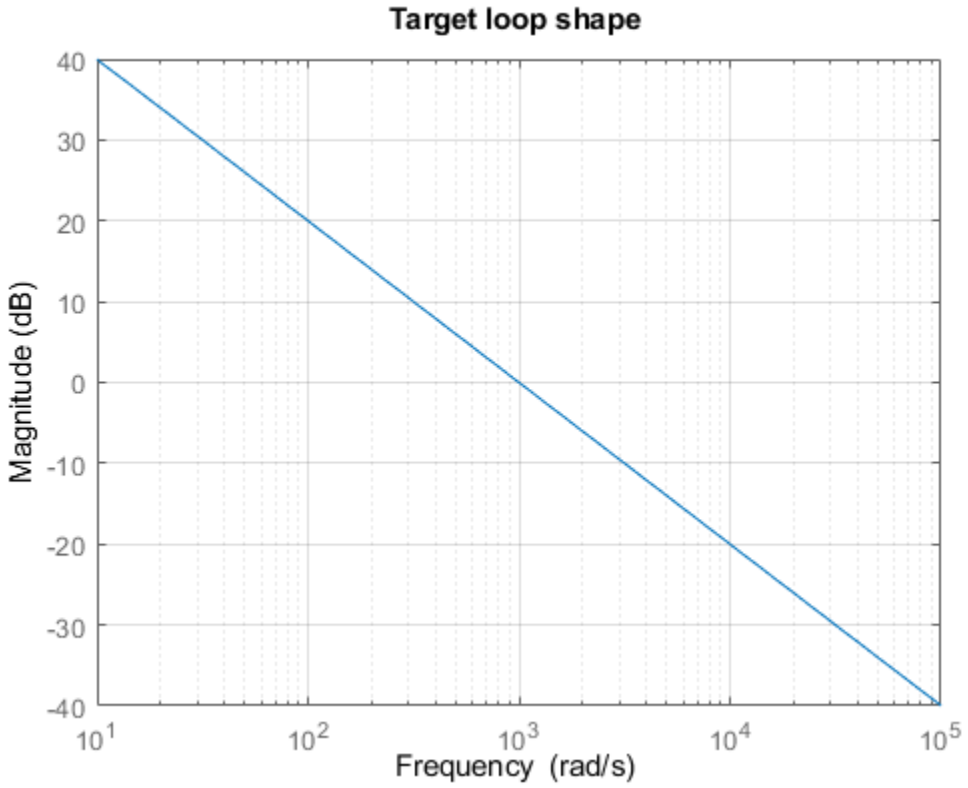
Use the `tunablePID` class to parameterize the PI block and specify the filter $F(s)$ as a transfer function depending on a tunable real parameter a.

```
C0 = tunablePID('C','pi');  % tunable PI

a = realp('a',1);    % filter coefficient
F0 = tf(a,[1 a]);    % filter parameterized by a
```

Loop Shaping Design

Loop shaping is a frequency-domain technique for enforcing requirements on response speed, control bandwidth, roll-off, and steady state error. The idea is to specify a target gain profile or "loop shape" for the open-loop response $L(s) = F(s)G(s)C(s)$. A reasonable loop shape for this application should have integral action and a crossover frequency of about 1000 rad/s (the reciprocal of the desired response time of 0.001 seconds). This suggests the following loop shape:

```
wc = 1000;  % target crossover
s = tf('s');
LS = (1+0.001*s/wc)/(0.001+s/wc);
bodemag(LS,{1e1,1e5}), grid, title('Target loop shape')
```



Note that we chose a bi-proper, bi-stable realization to avoid technical difficulties with marginally stable poles and improper inverses. In order to tune $C(s)$ and $F(s)$ with `hinfstruct`, we must turn this target loop shape into constraints on the closed-loop gains. A systematic way to go about this is to instrument the feedback loop as follows:

- Add a measurement noise signal `n`
- Use the target loop shape `LS` and its reciprocal to filter the error signal `e` and the white noise source `nw`.

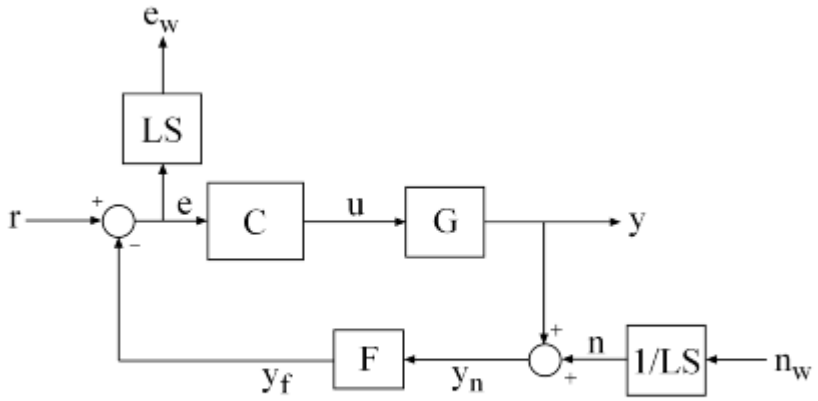


Figure 2: Closed-Loop Formulation

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';
C0.u = 'e';   C0.y = 'u';
F0.u = 'yn';  F0.y = 'yf';

% Specify summing junctions
Sum1 = sumblk('e = r - yf');
Sum2 = sumblk('yn = y + n');

% Connect the blocks together
T0 = connect(G,Wn,We,C0,F0,Sum1,Sum2,{'r','nw'},{'y','ew'});
```

These commands construct a generalized state-space model `T0` of $T(s)$. This model depends on the tunable blocks `C` and `a`:

```
T0.Blocks
```

```
ans = struct with fields:
    C: [1x1 tunablePID]
    a: [1x1 realp]
```

Note that `T0` captures the following "Standard Form" of the block diagram of Figure 2 where the tunable components C, F are separated from the fixed dynamics.

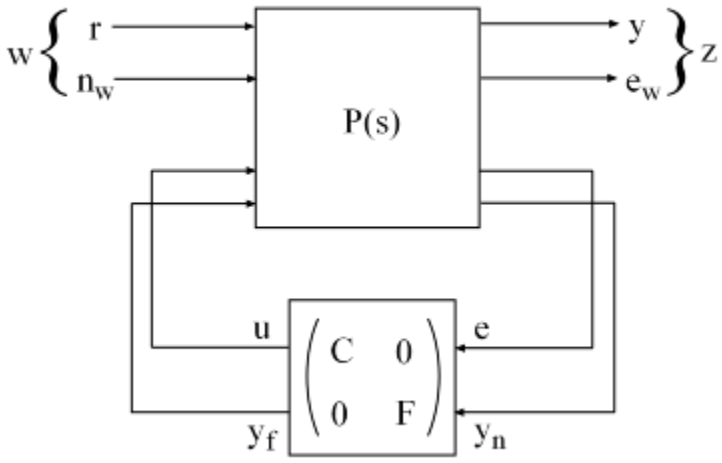


Figure 3: Standard Form for Disk-Drive Loop Shaping

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 = 67
Final: Peak gain = 597, Iterations = 181
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 597, Iterations = 176
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 3.88, Iterations = 68
Final: Peak gain = 1.56, Iterations = 102
Final: Peak gain = 1.56, Iterations = 96
```

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 =
```

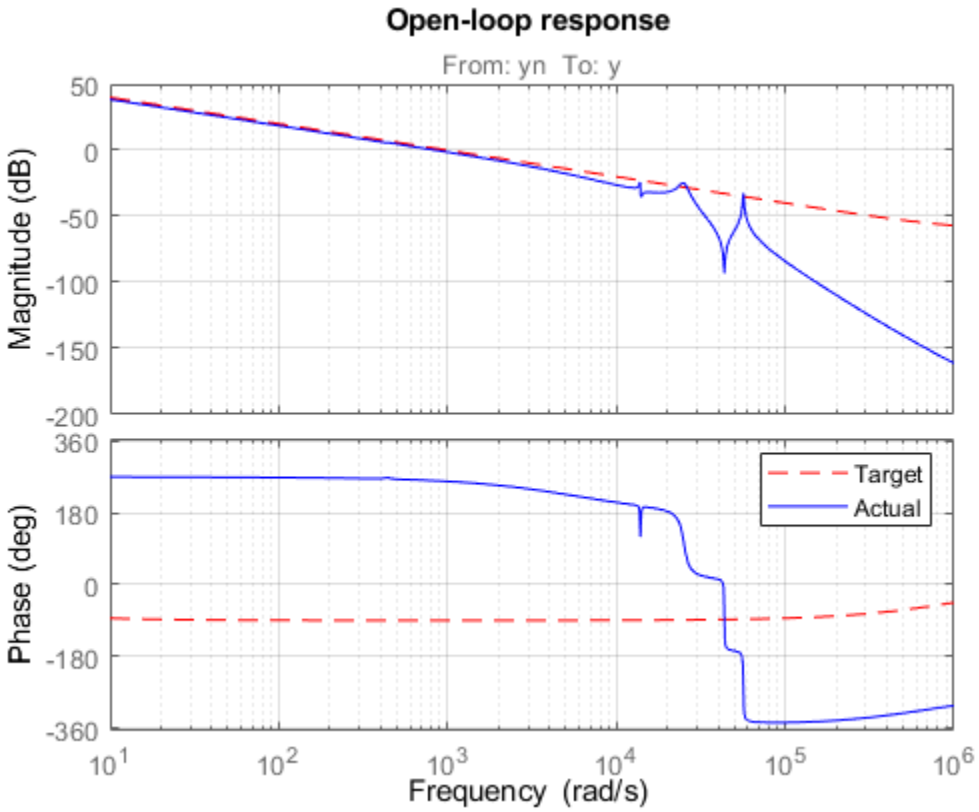
```
             1
  Kp + Ki * ---
             s

  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.
```

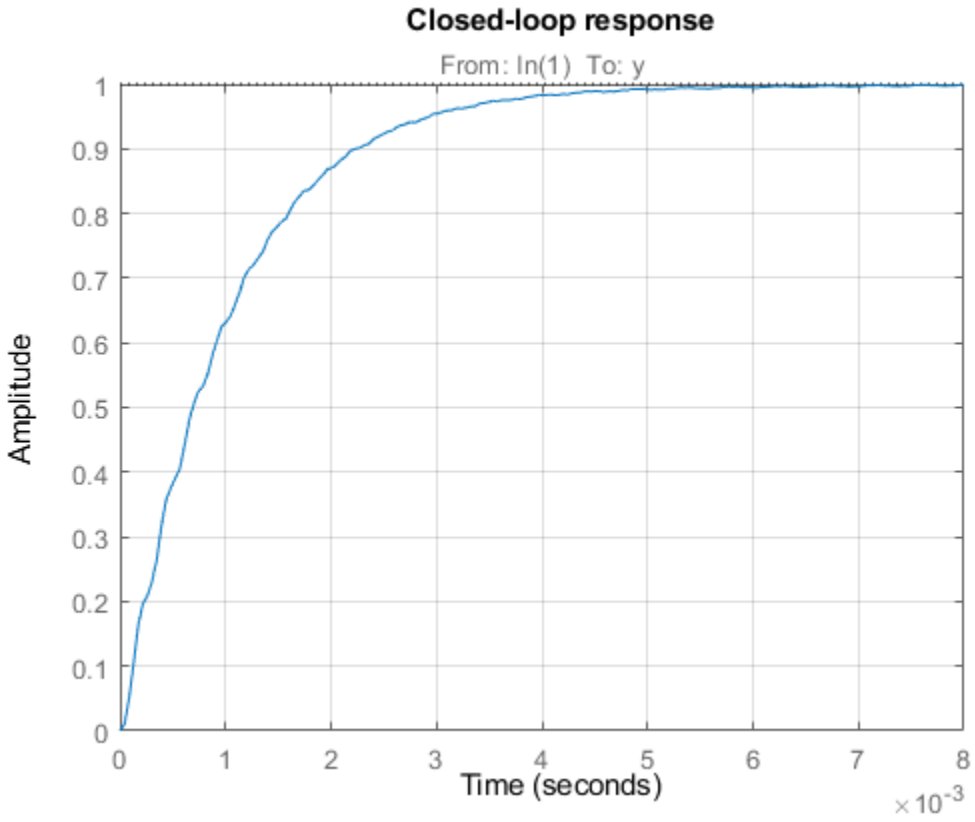
To validate the design, plot the open-loop response `L=F*G*C` and compare with the target loop shape `LS`:

```
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`:

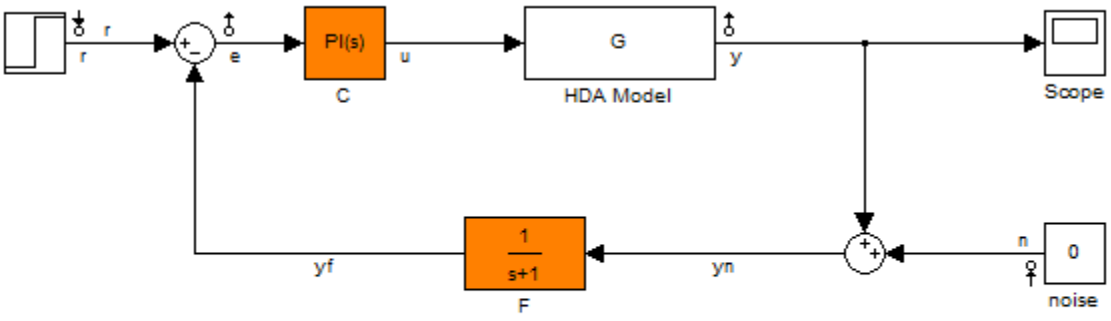
```
step(feedback(G*C,F)), grid, title('Closed-loop response')
```



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.

Then create an instance of the `slTuner` interface and mark the Simulink blocks `C` and `F` as tunable:

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

% Add weighting functions in n and e channels
T0 = blkdiag(1,LS) * T0 * 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(T0,opt);
```

```
Final: Peak gain = 3.88, Iterations = 67
Final: Peak gain = 597, Iterations = 182
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 597, Iterations = 186
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 3.88, Iterations = 68
Final: Peak gain = 1.56, Iterations = 101
Final: Peak gain = 1.56, Iterations = 108
```

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
```

See Also

`hinfstruct`

Related Examples

- “What Is a Fixed-Structure Control System?” on page 7-2